



Department of Computer Engineering  
ENCS3390 - Operating Systems

## **Course First Task**

**Prepared by:**

Ameer Rabie 1211226

**Instructor:**

Dr. Abdel Salam Sayyad

## **Introduction:**

The goal of this task is to explore process and thread management using POSIX and Pthreads in a C program. We aim to implement solutions for both processes and threads, compare their performance, and analyze trade-offs. The task is divided into four parts: Process Management, Multithreaded Processing, Performance Measurement, and Thread Management . in each part a code to process matrix multiplication is implemented with making sure to take advantage of the features that each approach have in order to get to best performance possible , then in each part an abstract of the idea is attached in the report , then the result are shown and justified .

# Contents

<b>1 Naive approach(1 main process , 1 main thread)</b>	<b>1</b>
1.1 Abstract: . . . . .	1
1.2 Time and throughput measurements: . . . . .	1
1.3 Results : . . . . .	1
<b>2 Multiple processes approach :</b>	<b>3</b>
2.1 Abstract: . . . . .	3
2.2 Time and throughput measurements: . . . . .	3
2.3 Results: . . . . .	4
<b>3 Multiple threads approach (join able and detached)</b>	<b>6</b>
3.1 Abstract . . . . .	6
3.2 Time and throughput measurements: . . . . .	7
3.3 Results: . . . . .	8
<b>4 Results comparison and conclusion:</b>	<b>11</b>
4.1 Results comparison: . . . . .	11
4.2 Suggestion and conclusion: . . . . .	12

## List of Figures

1.1	Naivee approach test 1 result . . . . .	1
1.2	Naivee approach test 2 result . . . . .	1
1.3	Naivee approach test 3 result . . . . .	1
2.1	Multiple processes approach test results . . . . .	4
3.1	Multiple joinable threads approach test results . . . . .	8
3.2	Multiple detached threads approach test 1 results . . . . .	9
3.3	Multiple detached threads approach test 2 results . . . . .	10

# 1 Naive approach(1 main process , 1 main thread)

## 1.1 Abstract:

Normal matrix multiplication implemented using only one process where each row of the first matrix gets multiplied by all the columns in the second matrix the main process start multiplying them row by row until it finishes multiplying the last row . results are held in globally defined matrix .

## 1.2 Time and throughput measurements:

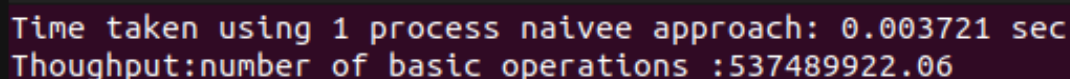
### 1. Time measurements:

Time measured includes all the process states including (ready,waiting,running,terminated) states the start time will be took right before the start of the matrix processing , and the end time will be took after the multiplication process finishes .

### 2. Throughput:

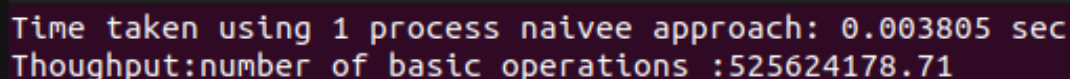
Throughput will be defined as ( number of basic operations (addition , multiplication) of cells per second ) , the code provides the time taken for the all the multiplication process to finish so the throughput = (number of operations / time taken by one operation) .

## 1.3 Results :



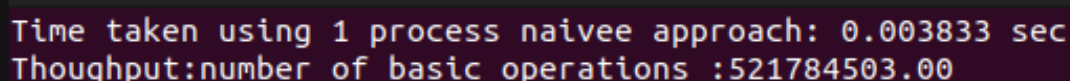
```
Time taken using 1 process naive approach: 0.003721 sec
Throughput:number of basic operations :537489922.06
```

Figure 1.1: Naive approach test 1 result



```
Time taken using 1 process naive approach: 0.003805 sec
Throughput:number of basic operations :525624178.71
```

Figure 1.2: Naive approach test 2 result



```
Time taken using 1 process naive approach: 0.003833 sec
Throughput:number of basic operations :521784503.00
```

Figure 1.3: Naive approach test 3 result

### 1. Result explanation:

Without any optimizations, the throughput of the naive approach to matrix multiplication is anticipated to be the least efficient among various methods. This is primarily due to the fact that a single process is tasked with executing all operations, resulting in prolonged execution times. The absence of optimizations exacerbates the situation by necessitating an extended duration for the CPU to complete the computations. This elongated processing time, in turn, amplifies the impact of context switching – the mechanism by which the CPU alternates between different tasks.

In the absence of optimizations, the average throughput using the naive approach is measured at .53 Giga operations per second. However, it's important to note that the observed variation in throughput values is influenced by CPU scheduling and the overall workload on the CPU. The scheduling algorithm dictates the order in which tasks are executed, impacting the efficiency of the process. Although efforts were made to minimize external interference (all user programs were closed during measurement), complete elimination of all processes in the computing environment was deemed impractical. As such, the variability in throughput values can be attributed to the intricacies of CPU scheduling and the overall busyness of the CPU.

## **2 Multiple processes approach :**

### **2.1 Abstract:**

This approach adopts a strategy of parallelization for matrix multiplication by distributing the computational workload across multiple processes. In this scheme, each process is assigned the task of multiplying a specific range of the first matrix's row by all the columns of the second matrix, generating a corresponding range of rows in the result matrix. The main process orchestrates this parallelization by creating a designated number of child processes and allocating to each a distinct processing range. The distribution is uniform across processes, except for the last one, which accommodates any remaining rows if the total number of matrix rows is not evenly divisible by the number of processes , also for further explanation of how the approach works read the code comments .

During computation, each process independently executes its designated computations and communicates the computed range along with the corresponding result rows back to the main process via interprocess communication using pipes. This enables the main process to efficiently assemble and integrate the results into the predefined result matrix structure. The utilization of this parallelized approach enhances the overall efficiency of matrix multiplication by leveraging concurrent processing capabilities, particularly beneficial when dealing with large matrices and computationally intensive tasks.

### **2.2 Time and throughput measurements:**

#### **1. Time measurements :**

In this methodology, the commencement time is recorded prior to the creation of child processes, and the conclusion time is noted after the complete computation is accomplished and the results are securely stored within the main process. Consequently, the recorded time encapsulates various key components, including the time taken for the instantiation of child processes, the computation duration for each process responsible for multiplying its assigned range of rows, the time invested in reading from and writing to interprocess communication pipes, the overhead of closing these pipes, the duration required for each child process to finalize its computation, and the time spent in the waiting state for processes to synchronize. This comprehensive time measurement offers a holistic perspective, encompassing the entire lifecycle of the parallelized matrix multiplication process, from process creation to result consolidation.

## 2. Throughput

Given that we can accurately measure the total time taken for the entire matrix multiplication process to conclude, coupled with our knowledge of the total number of operations executed during this computation, we can quantify the throughput in a manner analogous to the procedure employed in the naive approach. By dividing the total number of floating-point operations by the measured execution time, we ascertain the throughput, providing a meaningful metric to evaluate the efficiency of the matrix multiplication process. This approach enables a direct comparison of the computational efficiency, allowing for insights into the performance gains achieved through parallelization and process orchestration.

### 2.3 Results:

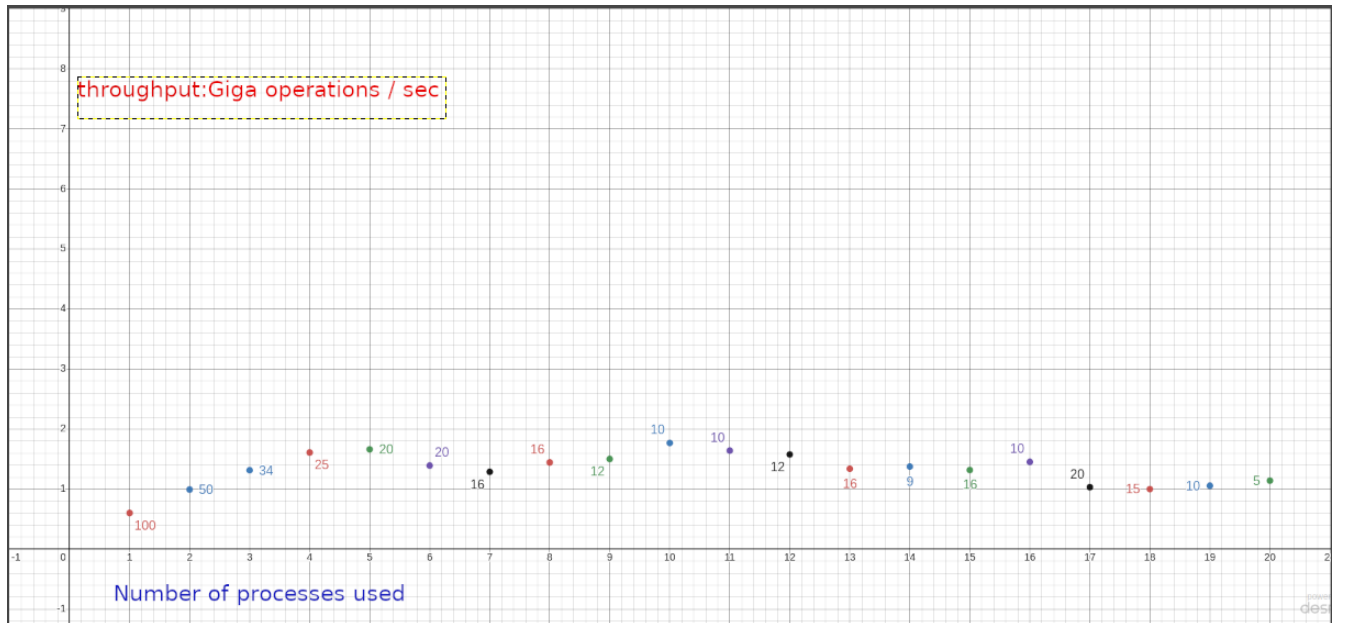


Figure 2.1: Multiple processes approach test results

#### 1. Figure description:

In the figure, the x-axis represents the number of processes employed in the matrix multiplication operations, while the y-axis denotes the number of basic operations executed per second. Additionally, it's essential to highlight that the labels assigned to each data point on the graph signify the maximum number of rows allocated to each process for processing. For instance, in a scenario where 7 processes are utilized to multiply a 100 x 100 size matrix, each process would be tasked with processing 14 rows, with the exception of the last process, which would handle 16 rows. This labeling convention provides a clear indication of the distribution of computational workload among the processes, offering insights into the granularity of their responsibilities.



## 2. Results explanation:

In the multi-processes approach, the expected outcome of increasing the number of processes is a decrease in the number of rows that each process is assigned to process, potentially leading to an enhancement in program throughput. However, this anticipated improvement is counteracted by several factors. Firstly, the increase in processes corresponds to a rise in the number of child processes to be created, as each process is forked, incurring additional time overhead during creation.

Furthermore, the augmented number of processes results in a proportional increase in the creation and closure of pipes, contributing to overall time consumption. Additionally, the escalation in the number of child processes, especially beyond the capacity of logical processors (e.g., 16 on my device), introduces the likelihood of increased context switches, further influencing throughput.

Another factor affecting throughput arises from the method of distributing rows among child processes. In the current implementation, the last child process receives any remaining rows, effectively setting a lower bound on the time required. This can lead to suboptimal efficiency, particularly when other child processes have fewer rows to process. While alternative strategies, such as distributing remaining rows incrementally among child processes, may address this issue, such approaches may introduce complexity to the code and potentially incur additional time costs. As of now, the current logic is retained due to its simplicity, and the potential alternatives are deemed less favorable in terms of code complexity and potential performance impact.

### 3 Multiple threads approach (join able and detached)

#### 3.1 Abstract

In the multithreaded approach, threads are spawned from the main thread within the main process. Each thread assumes responsibility for multiplying a specific range of rows from the first matrix by all the columns of the second matrix, thereby generating a corresponding range of result matrix rows. During thread creation, the thread's identifier is passed, enabling it to ascertain the specific range it is tasked with processing. Subsequent to computing its designated range, each thread independently executes the corresponding computation and gracefully exits.

The orchestration of these threads involves the main thread waiting for each thread to complete its computation in a sequential manner, from the first thread to the last. This synchronized joining of threads ensures that the matrix multiplication is completed systematically. In the case of detached threads, the distinction lies in the usage of `(pthread_exit())` at the conclusion of the main process. This mechanism patiently awaits the completion of all threads before permitting the main process to exit, obviating the need for explicit joins in the main process. Consequently, variations exist in the timing computation methodology, accounting for the differences in synchronization and termination between joined and detached threads.

## 3.2 Time and throughput measurements:

### 1. Time measurements:

In the context of time measurements, the approach employed for joinable and detached threads diverges. For joinable threads, the commencement time is recorded before the threads are created, and the conclusion time is marked after the main thread in the main process completes the process of joining all the threads. This comprehensive time measurement encapsulates every aspect, starting from thread creation to processing, scheduling, and ultimately the cleanup of each thread.

On the other hand, for detached threads, precise determination of when all threads will finish and be cleaned is challenging. To address this uncertainty, the time measurement strategy involves capturing the start time just before creating the detached threads. Given the unpredictability of when each detached thread will complete and be cleaned, the end time is recorded inside each detached thread, right after it concludes processing its designated segment and before it undergoes cleanup. Consequently, this time measurement excludes the thread cleaning duration. As a result, the throughput of the entire program is determined by dividing the total number of operations by the highest time among the detached threads. This approach provides an effective metric for evaluating the overall efficiency of the program in scenarios where detached threads are utilized.

### 2. Throughput:

In the joinable thread approach, the program itself calculates the throughput as it can precisely determine when the entire process concludes. However, in the detached thread scenario, manual computation is necessary since the program cannot ascertain when the threads will finish. The throughput is computed differently for each case:

For joinable threads, it is determined by dividing the total number of basic operations needed for matrix multiplication by the time calculated by the program, given that it accurately identifies when the entire process finishes.

In the case of detached threads, where the program lacks visibility into thread completion times, throughput is manually computed by dividing the total number of operations by the longest time taken by any single thread to finish. This distinction in throughput calculation reflects the inherent differences in the approach to tracking and managing thread completion in joinable versus detached threads.

### 3.3 Results:

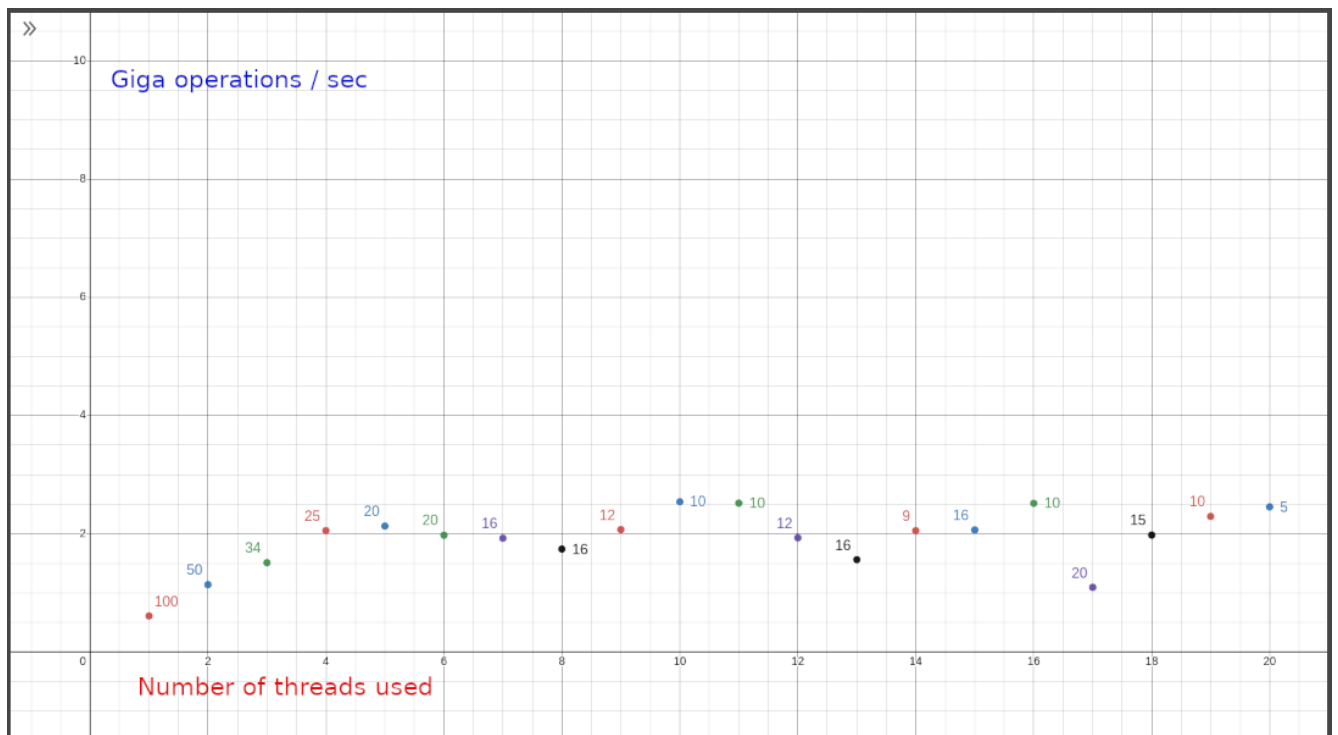


Figure 3.1: Multiple joinable threads approach test results

1. Figure description:

In the provided figure, the x-axis depicts the number of threads involved in matrix multiplication operations, while the y-axis portrays the rate of basic operations executed per second (throughput). Notably, each data point on the graph is accompanied by labels that offer pertinent insights into the maximum number of rows allocated to each thread for computation. To illustrate, envision a scenario where 7 threads are tasked with multiplying a 100 x 100 matrix. In this configuration, every thread undertakes the processing of 14 rows, with the exception of the last thread, which manages 16 rows. This labeling system serves as a critical guide to the distribution of computational workload among the threads, fostering a nuanced understanding of their individual responsibilities and enriching the interpretation of the presented data.

## 2. Results explanation:

In the multi-threaded approach, the reduction in the number of rows allocated to each thread as we increase the number of threads contributes to a decrease in processing time, resulting in an overall increase in throughput. While there are factors that can impact throughput, their effects are relatively negligible until the number of threads approaches approximately 1/5 of the total number of rows. Beyond this point, the influence of these factors becomes more pronounced.

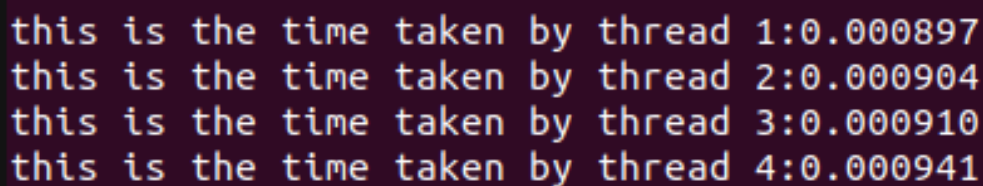
Firstly, the time required for thread creation increases with the growing number of threads since each additional thread necessitates its own creation time. Similarly, the time spent on cleaning threads also rises with an increasing number of threads.

Another factor affecting throughput is the sequential joining of threads in the main thread. If the first thread finishes last, subsequent threads must wait their turn to be joined, leading to additional time overhead. This effect becomes more significant as the number of threads increases.

Additionally, when the number of rows is not divisible by the number of threads, the remaining rows are assigned to the last thread. This can cause delays in the overall process as it awaits the completion of the last thread, akin to the scenario where all threads have an equal number of rows.

The last significant factor influencing throughput is context switching. The impact of context switching becomes more pronounced as the number of threads increases significantly. While the context switch of threads typically incurs a lower cost due to the smaller amount of data being moved to and from the CPU during the switch, its cumulative effect becomes more notable with a higher thread count.

## 3. Detached threads results and notes:



```
this is the time taken by thread 1:0.000897
this is the time taken by thread 2:0.000904
this is the time taken by thread 3:0.000910
this is the time taken by thread 4:0.000941
```

Figure 3.2: Multiple detached threads approach test 1 results

```
this is the time taken by thread 1:0.001879
this is the time taken by thread 2:0.001916

Process returned 0 (0x0)    execution time : 0.012 s
Press ENTER to continue.
□
```

Figure 3.3: Multiple detached threads approach test 2 results

We observe that the resulting execution time in the detached threads is practically indistinguishable from that of joinable threads, with the notable exception that the cleaning time for detached threads is not explicitly measured in our code. However, even if measured, the disparity would likely be negligible. The primary reason for this similarity lies in the relatively small overhead introduced by the sequential joining of threads in joinable threads.

In the context of measuring the time of detached threads, there is no precise method available. One approach involves defining a counter to tally the number of detached threads that have completed. However, this method carries the risk of a race condition, where multiple threads concurrently increment the counter, albeit with a low probability. Theoretically, it is challenging to prevent the occurrence of a race condition in this scenario. Another method entails creating a boolean array with a size equal to the number of threads, with each detached thread marking its own index upon completion. While this approach mitigates the risk of a race condition, the time taken to check the array in the main thread introduces imprecision.

In summary, detached and joinable threads exhibit nearly identical performance, with only minor differences. Their usage is determined by specific use cases. Detached threads are preferred when the results obtained by the threads are inconsequential for the program's continuation. In contrast, joinable threads are employed when the results are crucial for the program that initiated the threads, ensuring synchronization and the availability of necessary data for further computation.

## **4 Results comparison and conclusion:**

### **4.1 Results comparison:**

The comparison of the results reveals distinct performance characteristics among the employed approaches. Notably, the naive approach, utilizing a single process and main thread, exhibits the slowest performance with an average throughput of 0.53 Giga operations per second. This outcome is expected, given the absence of optimizations in the naive approach.

A noteworthy observation is the superiority of the multiple threads approach over the multiple processes approach. In the optimal scenario, the multiple threads approach achieves a throughput of nearly 2.54 Giga operations per second with 10 threads, outperforming the multiple processes approach, which attains a maximum throughput of 1.77 Giga operations per second using 10 processes. Several factors contribute to this performance difference.

Firstly, the creation of processes incurs a heavier cost, as each process necessitates the duplication of code, memory, and other resources. In contrast, thread creation is more efficient, with a lower cost, as threads share the same memory space and certain resources.

Secondly, in the multiple processes approach, the results obtained by each process must be communicated to the main thread using pipes. This inter-process communication introduces additional overhead. In contrast, multiple threads share the same memory, allowing results obtained by individual threads to directly impact the main thread's results.

Furthermore, context switching in the multiple threads approach is more efficient, as it incurs a lower time cost compared to context switching between processes. The latter requires moving more data during the switch, contributing to increased overhead.

Considering these factors collectively, the multiple threads approach emerges as the most efficient among the tested approaches, showcasing superior throughput and performance.

## 4.2 Suggestion and conclusion:

After conducting a series of comprehensive tests to identify the optimal approach for matrix multiplication tasks, it is evident that two strategies stand out as the most effective, especially when dealing with larger matrix sizes. The first recommended approach involves utilizing the multiple threads approach, where the number of threads employed should ideally be equal to the number of rows divided by 10 or the total number of rows. The second viable approach is to employ a small number of child processes, typically 10 or 4, with an equal distribution of threads across these processes, resulting in an overall number of threads equal to the number of matrix rows divided by 10 or the total number of matrix rows.

Upon experimentation with a matrix of size 10,000 x 10,000, the multiple processes approach with a small number of child processes was found to be impractical. When using a small number of child processes, the execution time for each process became excessively long, while employing a large number of child processes led to prolonged creation times. In contrast, the first approach, which involves a larger number of threads, demonstrated more feasible results within a reasonable time frame. The creation of threads in this approach proved to be less resource-intensive compared to the creation of a substantial number of processes.

Attempting to analyze the time complexity of the program for each child thread in the first approach, assuming the number of threads is equal to the number of matrix rows divided by 10, the time complexity becomes  $O(\text{size} * \text{size} * 10)$ . Factoring in the time complexity of thread creation ( $O(\text{size}/10)$ ), the overall time complexity for the first approach is approximated as  $O(\text{size} + (\text{size} * \text{size}))$ . This time complexity represents a favorable outcome, given the constraints of the basic matrix multiplication algorithm.

In conclusion, the optimal strategies for matrix multiplication, particularly with larger matrices, involve either the multiple threads approach or the utilization of a small number of child processes with an equal distribution of threads. The first approach, characterized by a larger number of threads, yields efficient results within reasonable time frames and demonstrates favorable time complexity for the given task.