Faculty of Engineering and Technology

Electrical and Computer Engineering Department

OPERATING SYSTEMS

ENCS3390

Task One Report

Process and Thread Management

Student Name: Raghad Murad Buzia

Student #: 1212214

Instructor: Abdel Salam Sayyad

Section: 1

Date: 3/11/2023

Abstract

In this task, our aim is to implement a matrices multiplication process using different techniques: a naive approach, multiple child processes with pipes as an IPC between the parent process and the child process, multiple joinable threads, and multiple detached threads. Then the performance of each method will be measured by recording the time required to perform the multiplication operation in each method, as this involves trying different numbers of a children processes and threads and then making a comparison between the methods used based on the observed results.

Table of Contents

Theory

This task is about the process of multiplying matrices using different methods: a naive approach, multiple child processes with pipes as an IPC between the parent process and the child process, multiple joinable threads, and multiple detached threads. Here is a simple explanation of the concepts mentioned in the task.

Matrix Multiplication

Matrix Multiplication is a binary operation performed on two matrices to get a new matrix called the product matrix. Suppose we take two matrices A and B such that the number of columns in the first matrix is equal to the number of rows in the second matrix then we can multiply these two matrices to get a new matrix of the same order that is called the multiplication of the two matrices A and B. Thus, it is clear that not any two matrices can be multiplied and we can multiply only those matrices that follow a specific condition. [1]

Algorithm of Matrix Multiplication:

Step 1: Check the compatibility of the matrix by checking that the number of columns in the 1st matrix equals the number of rows in the 2nd matrix.

Step 2: Multiply the elements in the first row of the first matrix with the elements in the first column of the matrix and find the sum of all the products. Then multiply the element in the first row of the first matrix with the elements of the second column in the second matrix. Repeat this process till elements of all the positions are not obtained.

Step 3: Substitute all the elements obtained in Step 2 in their respective position to find the required product matrix. [1]

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} (a_{11}*b_{11})+(a_{12}+b_{21})+(a_{13}*b_{31}) & (a_{11}*b_{12})+(a_{12}+b_{22})+(a_{13}*b_{32}) \\ (a_{21}*b_{11})+(a_{22}+b_{21})+(a_{23}*b_{31}) & (a_{21}*b_{12})+(a_{22}+b_{22})+(a_{23}*b_{32}) \end{bmatrix}$$

Matrix A_{2x3} Matrix B_{3x2} Product/Result Matrix C_{2x2}

Process and Thread Management

Process

A process is a program in execution. For example, when we write a program in any high-level programing language such as C or C++ and compile it, the compiler creates binary code. The original code and binary code are both programs. When we actually run the binary code, it becomes a process. Then a process is an 'active' entity instead of a program, which is considered a 'passive' entity. A single program can create many processes when run multiple times; for example, when we open a .exe or binary file multiple times, multiple instances begin (multiple processes are created). [2]

So, processes are basically the programs that are dispatched from the ready state and are scheduled in the CPU for execution. PCB (Process Control Block) holds the concept of process. A process can create other processes which are known as Child Processes. The process takes more time to terminate and it is isolated means it does not share the memory with any other process. The process can have the following states new, ready, running, waiting, terminated, and suspended. [3]

Process Management

Process management refers to the techniques and strategies used by organizations to design, monitor, and control their business processes to achieve their goals efficiently and effectively. It involves identifying the steps involved in completing a task, assessing the resources required for each step, and determining the best way to execute the task. [2]

Process management can help organizations improve their operational efficiency, reduce costs, increase customer satisfaction, and maintain compliance with regulatory requirements. It involves analyzing the performance of existing processes, identifying bottlenecks, and making changes to optimize the process flow. [2]

Process management includes various tools and techniques such as process mapping, process analysis, process improvement, process automation, and process control. By applying these tools and techniques, organizations can streamline their processes, eliminate waste, and improve productivity. [2]

Inter Process Communication (IPC)

A process can be of two types:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently, in reality, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience, and modularity. Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other through both: [4]

- Shared Memory
- Message passing

Pipe

A Pipe is a technique used for inter process communication. A pipe is a mechanism by which the output of one process is directed into the input of another process. Thus it provides one way flow of data between two related processes. Although pipe can be accessed like an ordinary file, the system actually manages it as FIFO queue. A pipe file is created using the pipe system call. A pipe has an input end and an output end. One can write into a pipe from input end and read from the output end. A pipe descriptor, has an array that stores two pointers, one pointer is for its input end and the other pointer is for its output end. Suppose two processes, Process A and Process B, need to communicate. In such a case, it is important that the process which writes, closes its read end of the pipe and the process which reads, closes its write end of a pipe. Essentially, for a communication from Process A to Process B the following should happen. [5]

- Process A should keep its write end open and close the read end of the pipe.
- Process B should keep its read end open and close its write end. When a pipe is created, it is given a fixed size in bytes.

When a process attempts to write into the pipe, the write request is immediately executed if the pipe is not full. However, if pipe is full the process is blocked until the state of pipe changes. Similarly, a reading process is blocked, if it attempts to read more bytes that are currently in pipe, otherwise the reading process is executed. Only one process can access a pipe at a time. [5]

Threads

Thread is the segment of a process which means a process can have multiple threads and these multiple threads are contained within a process. A thread has three states: Running, Ready, and Blocked.

The thread takes less time to terminate as compared to the process but unlike the process, threads do not isolate. [3]

A thread is also known as a lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc. More advantages of multithreading are discussed below. [6]

Multithreading is a technique used in operating systems to improve the performance and responsiveness of computer systems. Multithreading allows multiple threads (i.e., lightweight processes) to share the same resources of a single process, such as the CPU, memory, and I/O devices. [6]

In multithreading, joinable threads and detached threads represent two distinct models of thread management. Joinable threads, also known as "joinable" or "non-detached" threads, require explicit termination by the main thread using the “pthread_join” function to reclaim resources after their completion. This approach allows the main thread to synchronize and wait for individual threads to finish their tasks before proceeding. On the other hand, detached threads, often referred to as "non-joinable" threads, operate independently, automatically releasing resources upon termination without requiring an explicit join operation. While detached threads offer a more hands-off approach to thread management, joinable threads provide finer control over the synchronization of thread execution, making them suitable for scenarios where the main thread needs to coordinate and collect results from concurrent tasks. The choice between joinable and detached threads depends on the specific requirements of the application and the desired level of control over the thread lifecycle. [7]

Procedure Discussion

Generate The Matrices

Since the task is about multiplying matrices, the first thing to do is to create the matrices. You must create a matrix consisting of 100 columns and 100 rows, and this is represented by the definition of a Two Dimensional Array. We define the matrices A and B that we will multiply so that they are filled as follows:

The first matrix A should fill with the repeated of the student number.

The second matrix B should fill with the repeated of the student number * student birth year.

And we need to define the result matrix, I define four result matrices (an a result matrix for each part).

```
#define MATRIX_SIZE 100

//Define three matrices (A, B, and Result)
int matrixA[MATRIX_SIZE][MATRIX_SIZE]={0};
int matrixB[MATRIX_SIZE][MATRIX_SIZE]={0};
int matrixResult1[MATRIX_SIZE][MATRIX_SIZE]={0};
int matrixResult2[MATRIX_SIZE][MATRIX_SIZE]={0};
int matrixResult3[MATRIX_SIZE][MATRIX_SIZE]={0};
int matrixResult4[MATRIX_SIZE][MATRIX_SIZE]={0};
```

Then we need to perform the matrices multiplication using different methods:

A Naive Approach

The naive approach involved implementing a simple single-threaded matrix multiplication algorithm. Matrix multiplication is straightforward, as it iterates across rows and columns to calculate the resulting matrix.

Multiple Child Processes with Pipes

A method that uses child processes, parent processes, and inter-process communication (IPC) through pipes. Child processes were created using the fork system call, and pipes were used to communicate between the parent process and child processes. The matrix multiplication logic was divided among child processes, with each process responsible for part of the computation. Results were collected through pipes, and performance was measured.

Multiple Joinable Threads

For a multi-threaded approach, joinable threads using POSIX threads (Pthreads) were employed. Threads were created to parallelize the matrix multiplication task, with each thread handling a portion of the computation. The main thread waited for each thread to complete using pthread_join before collecting the results and measuring performance.

Multiple Detached Threads

The detached threads approach involved creating threads that operated independently without explicit joining by the main thread. we used Pthreads to create detached threads, and the matrix multiplication logic was divided among these threads. Detached threads were allowed to run concurrently, and the main thread continued its execution.

Result Discussion

After running the program, we record the execution times and the throughput for each method and try different numbers of processes and threads. The following table shows the readings that were recorded:

		Part One A Naive Approach	Part Two Multiple Child Processes with Pipes	Part Three Multiple Joinable Threads	Part Four Multiple Detached Threads
Process Number = 2 Theard Number = 2	Execution Time	0.004433	0.00174	0.000108	0.000076
	Throughput	225.580871	574.712644	9259.259259	13157.894737
Process Number = 4 Theard Number = 4	Execution Time	0.004751	0.002528	0.000299	0.000129
	Throughput	210.482004	395.56962	3344.481605	7751.937984
Process Number = 6 Theard Number = 6	Execution Time	0.005382	0.00392	0.000411	0.000257
	Throughput	185.804534	255.102041	2433.090024	3891.050584
Process Number = 8 Theard Number = 8	Execution Time	0.005736	0.004776	0.000491	0.0003
	Throughput	174.337517	209.380235	2036.659878	3333.333333
Process Number = 8 Theard Number = 8	Execution Time	0.006539	0.00491	0.001096	0.000661
	Throughput	152.928582	203.665988	912.408759	1512.859304

Based on the results obtained and presented in the table, we conclude the following:

- Execution time increases as the number of processes or threads increases, and this means that the parallelism overhead increases.

- As we increase the number of processes or threads beyond a certain point, the additional gain in performance becomes progressively smaller or less significant. The initial increase in performance is noticeable and useful, but as we continue to add more processes or threads, the overall improvement begins to decline.
- Throughput is highest in Part 4 (Multiple Detached Threads), which means that it benefits the most from parallelization.
- What is the proper/optimal number of child processes or threads?

  Choosing the optimal number of processes and threads depends on the characteristics of your system and factors such as the nature of the workload, available hardware resources, and overall increased efficiency, which means it will vary from one device to another. Based on the data that collected and recorded in the table, the most likely number of threads and processes to be optimal is either 4 or 6. Where at a certain point when the number of processes or threads increases, the additional gain in performance becomes gradually smaller or less significant. Therefore, it is necessary to find a balance in the number of processes or threads to improve performance without introducing unnecessary burdens or reducing the return on the effort invested in parallelization.

Conclusion

The aim of this task was to perform matrix multiplication using various techniques, including a naive approach, multiple child processes with inter-process communication via pipes, and multiple joinable threads, and multiple detached threads. The performance of each method was assessed by measuring the time required for matrix multiplication across different numbers of child processes and threads. The comparison of these methods revealed insights into their respective efficiencies. The results underscore the importance of selecting an appropriate parallelization strategy, considering factors such as the number of child processes or threads, to optimize the performance of matrix multiplication.

References

[1]: https://www.geeksforgeeks.org/matrix-multiplication/

[2]: https://www.geeksforgeeks.org/introduction-of-process-management/

[3]: https://www.geeksforgeeks.org/difference-between-process-and-thread/

[4]: https://www.geeksforgeeks.org/inter-process-communication-ipc/

[5]: https://www.geeksforgeeks.org/ipc-technique-pipes/

[6]: https://www.geeksforgeeks.org/thread-in-operating-system/

[7]: https://hpc-tutorials.llnl.gov/posix/joining_and_detaching/