Faculty of Engineering & Technology – Electrical & Computer Engineering Department

Second Semester 2024

Operating System ENCS3390

Project 1

Name: Rasha Daoud – 1210382

Section: 6

Instructor: Abdel Salam Sayyad

Date: 3rd May 2024

Abstract:

This project aims to explore process and thread management techniques through the implementation of matrix multiplication using three distinct approaches: Naive, Child Processes, and Joinable Threads. The primary objectives include analyzing the performance of each approach and thoroughly evaluating their execution times to gain a deeper understanding of their effectiveness in real-world scenarios.

Table Of Contents

Table Of figures:

Table Of Tables:

Theory

Parent Process

A parent process generates one or more child processes, forming a hierarchical structure common in operating systems like Linux and macOS. These parent processes initiate child creation to execute specific tasks or parallelize operations. Parent processes maintain control over their children, utilizing mechanisms such as the wait() system call to synchronize tasks. Communication between parents and children is facilitated through inter-process communication methods like pipes, sockets, or shared memory.

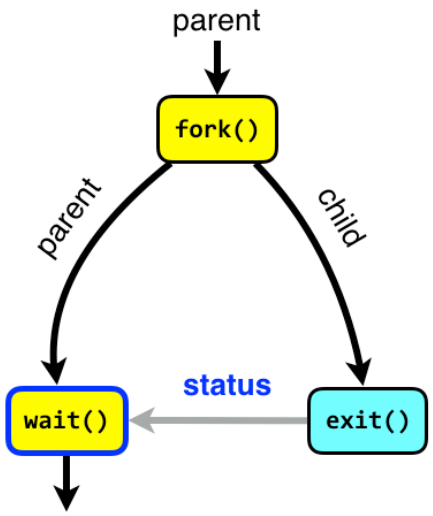

Figure 1: Parent Process, and Child Process [1].

Child Process

A child process, created by a parent process, inherits attributes such as memory space and file descriptors. In Unix-like environments, such as Linux, processes can be created to perform specific tasks. Using the fork() system call in C, the parent process creates an exact copy known as a child process. After creation, each process can execute distinct code paths or tasks independently.

Threads

A thread is the smallest unit of execution within a process. Threads enable concurrent execution of multiple tasks within the same process, allowing for parallelism and efficient utilization of system resources. Threads share the same memory space and resources within a process, making communication and data sharing between threads more straightforward compared to separate processes.

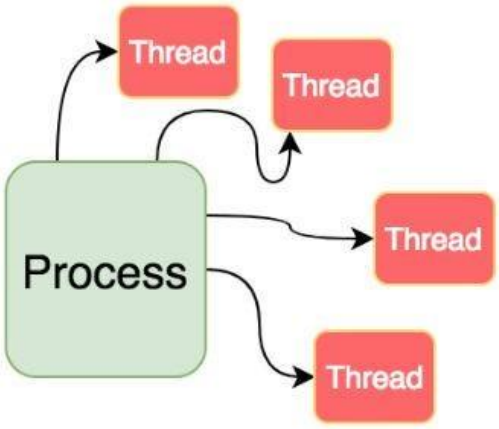

Figure 2: Threads [2].

Processes Vs. Threads

Processes and threads are units of execution in a computer. Processes are independent programs with their own memory, while threads are smaller units within a process, sharing memory. Processes isolate programs, while threads enable multiple tasks within one program. Threads can communicate directly, while processes use communication methods. Processes are heavier on resources, while threads are lightweight and faster to create and end.

Multiprocessing

Multiprocessing means running multiple processes at the same time on a computer. Each process works on its own, doing tasks separately from the others. It helps computers do more things at once, making them faster and more efficient.

Multithreading

Multithreading is a programming technique where a single process can have multiple threads running concurrently. Threads are like separate paths of execution within a process, allowing it to perform multiple tasks simultaneously. This approach can improve performance and responsiveness in applications by utilizing the available CPU resources more efficiently.

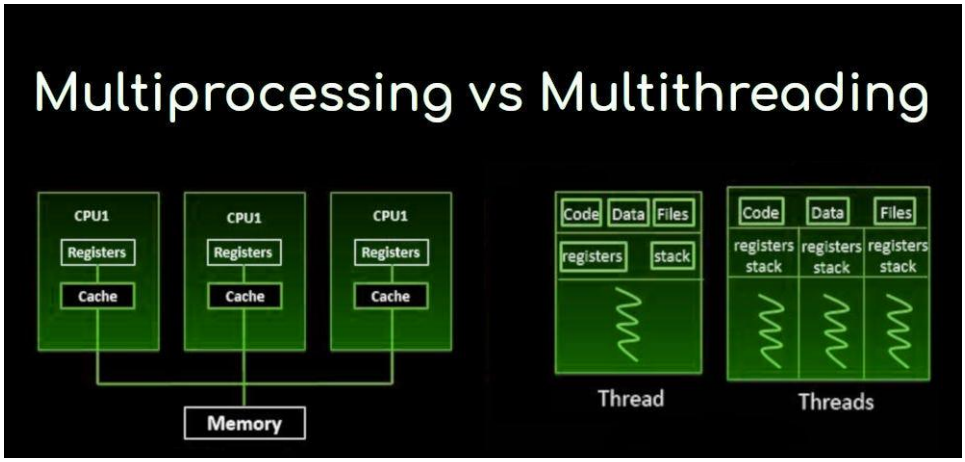

Figure 3: Multiprocessing vs. Multithreading [3].

Naïve approach

Time of naïve approach in seconds:

Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Avg.
0.000247	0.000283	0.000399	0.000253	0.000264	0.000276	0.000287

Table 1: Naive Approach Results.

Average Performance = 1 / Average Execution Time

= 1 / 0.000287=3484.32 tasks per seconds.

Multiprocessing approach

Time of multiprocessing approach in seconds:

#of Process	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Avg
2 Processes	0.000813	0.000739	0.000680	0.000737	0.000621	0.000537	0.000612
4 Processes	0.000359	0.000369	0.000888	0.000640	0.001014	0.000920	0.000698
6 Processes	0.001007	0.001189	0.000999	0.000630	0.000674	0.001233	0.000955

Table 2: Multiprocessing Approach Results.

Avg Performance (2 processes) = 1/0.000612= 1633.98
Avg Performance (4 processes) = 1/0.000698= 1432.66
Avg Performance (6 processes) = 1/0.000955= 1047.12

Multithreading approach

Time of multithreading approach in seconds:

#of Thread	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Avg
2 Threads	0.000638	0.000830	0.000533	0.000685	0.000477	0.000575	0.000623
4 Threads	0.001229	0.000924	0.000812	0.001153	0.000655	0.001028	0.000966
6 Threads	0.001383	0.000818	0.001027	0.000974	0.000959	0.000902	0.001010

Table 3: Multithreading Approach Results.

Avg Performance (2 threads) = 1/0.000623 =1605.13
Avg Performance (4 threads) = 1/0.000966 = 1035.19
Avg Performance (6 threads) = 1/0.001010 = 990.09

Summarization

Approach	Avg. Execution Time (sec.)	Performance
Naïve	0.000287	3484.32
Multiprocessing: 2	0.000612	1633.98
Multiprocessing: 4	0.000698	1432.66
Multiprocessing: 6	0.000955	1047.12
Multithreading: 2	0.000623	1605.13
Multithreading: 4	0.000966	1035.19
Multithreading: 6	0.001010	990.09

Commenting on the differences in performance

1. Naïve Approach: This approach has the fastest average execution time and highest performance. However, it doesn't utilize parallelism and relies on a single thread, limiting its scalability.
2. Multiprocessing: As the number of processes increases, the average execution time also increases, leading to a decrease in performance. This suggests that the overhead of managing multiple processes outweighs the benefits of parallel execution, especially beyond a certain point.
3. Multithreading: Like multiprocessing, increasing the number of threads beyond a certain point leads to diminishing returns and increased overhead. However, multithreading generally performs better than multiprocessing, indicating that the overhead of managing threads is lower compared to processes.

→ The API and functions that I used.

To achieve multiprocessing and multithreading requirements in C programming, I used different APIs and functions tailored for each approach.
For multiprocessing, I utilized the **fork()** system call, which creates a new process by duplicating the calling process. The child process executes independently of the parent process, allowing parallel execution of tasks. I also used functions like **wait()** to ensure proper synchronization between parent and child processes, enabling the parent process to wait for the completion of child processes.
In the case of multithreading, I employed POSIX threads (pthreads), which provide a standardized thread interface for Unix-like operating systems. This involved including the **<pthread.h>** header file and using functions like **pthread_create()** to create new threads, **pthread_join()** to wait for threads to complete, and **pthread_exit()** to terminate threads when their tasks are finished, and **pthread_self()** to obtain the unique identifier of the calling thread.
By leveraging these APIs and functions, I was able to implement multiprocessing and multithreading functionalities in the C programs, enabling parallel execution of tasks and efficient utilization of system resources.

→Amdahl's law

Speedup=1/ (S+(1- S)/N)

- ***Speedup*** is the theoretical speedup of the execution of the whole task.
- ***S*** is the percentage of the task that can be serial.
- ***N*** is the number of processing units available.

~To find the serial portion for each part:

Serial= Execution time Naïve/ Execution time for the specific approach.

1- Serial Naïve= 0.000287/0.000287=1
2- Serial Multiprocessing 2= 0.000287/0.000612 =0.4689
3- Serial Multiprocessing 4= 0.000287/0.000698 =0.4111
4- Serial Multiprocessing 6= 0.000287/0.000955 =0.3005
5- Serial Multithreading 2= 0.000287/0.000623 =0.4606
6- Serial Multithreading 4= 0.000287/0.000966 =0.2971
7- Serial Multithreading 6= 0.000287/0.001010 =0.2841

~To find the speedup:

1- Naïve= 1/(1+(0/4))=1
2- Multiprocessing 2=1/(0.4689+(1-0.4689)/4)=1.662
3- Multiprocessing 4=1/(0. 4111+(1-0. 4111)/4)=1.791
4- Multiprocessing 6=1/(0. 3005+(1-0. 3005)/4)=2.103
5- Multithreading 2=1/(0. 4606+(1-0. 4606)/4)=1.679
6- Multithreading 4=1/(0. 2971+(1-0. 2971)/4)=2.114
7- Multithreading 6=1/(0. 2841+(1-0. 2841)/4)=2.159

~the optimal number of child processes or threads is the one that strikes the right balance between performance improvement and resource utilization.

So, we can say that the optimal number of child process is: 4, and the optimal number of threads is: 4.

Conclusion:

- The naïve approach provides the best performance in terms of average execution time due to its simplicity and lack of overhead associated with parallelism.
- Multiprocessing and multithreading approaches offer scalability but are limited by overhead, particularly as the number of parallel units (processes or threads) increases.
- Choosing between multiprocessing and multithreading depends on factors such as the nature of the workload, system resources, and scalability requirements.

References

[1]: https://static.javatpoint.com/difference/images/difference-between-process-parent-process-and-child-process2.png

[2]: https://miro.medium.com/v2/resize:fit:413/0*jhbcbly_bEEYBlWQ.jpg

[3]:https://miro.medium.com/v2/resize:fit:996/1*FSEGozbKSrPGlBDf4Dl0Zg.jpeg