Faculty of Engineering & Technology Electrical & Computer Engineering Department

ENCS3390

Operating Systems
Process and Thread Management
Task Report

Prebread by:
Diaa Badaha 1210478

Instructor:
Dr. Bashar Tahayna

Section: 4
Date: 5-12-2023

Abstract

The aim of this task is to learn about processes and threads management, using matrix multiplication with different approaches: Naive, Child Processes, Joinable Threads and Detached Threads. And to learn deeply about their performances and analysing each one execution time.

Table of Contents

Table of figures

List of Tables

Theory

Child Process

A child process is the creation of a parent process, which can be defined as the main process that creates child or subprocesses to perform certain operations. Each process can have many child processes but only one parent. A child process inherits most of its parent's attributes.

A child process in C language is usually made by calling fork() function. And the parent process should wait for it's childs to finish their work by calling wait() function.

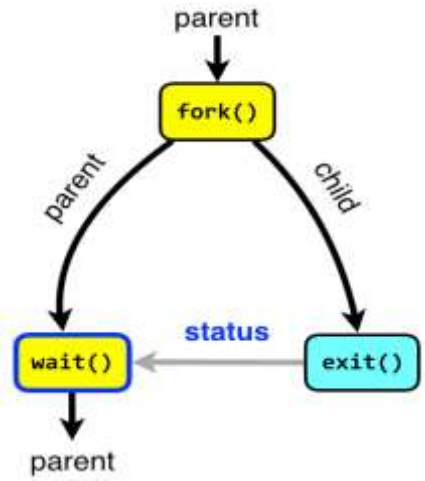

Figure 1: Child Processes

Threads

Within a program, a Thread is a separate execution path. It is a lightweight process that the operating system can schedule and run concurrently with other threads. The operating system creates and manages threads, and they share the same memory and resources as the program that created them. This enables multiple threads to collaborate and work efficiently within a single program.

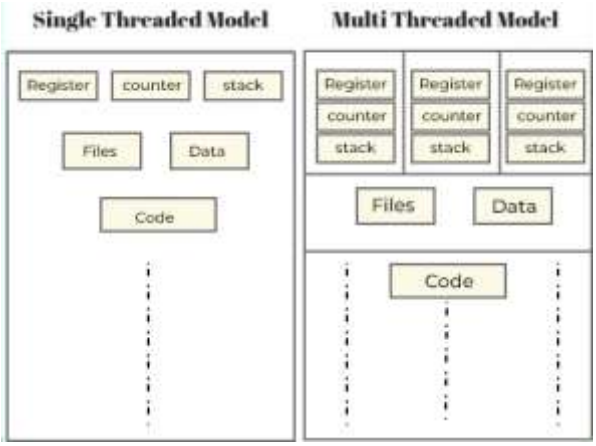

Figure 2: Threads

A thread is a single sequence stream within a process. Threads are also called lightweight processes as they possess some of the properties of processes. Each thread belongs to exactly one process. In an operating system that supports multithreading, the process can consist of many threads. But threads can be effective only if CPU is more than 1 otherwise two threads have to context switch for that single CPU.

Joinable threads VS. Detached threads

Joinable and detached threads represent two states of thread lifecycle management in multithreading environments. Joinable threads, which are the default state in POSIX threads, require explicit synchronization through pthread_join, allowing one thread to wait for another's completion and resource release. This is particularly useful for tasks

where the results or completion status of the thread are important. On the other hand, detached threads automatically free their resources upon completion, ideal for independent, "fire-and-forget" tasks where no return value or synchronization is needed. Choosing between them depends on the need for synchronization and resource management in your application, with joinable threads offering more control at the cost of requiring explicit management.

Processes VS. Threads

The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces. Threads are not independent of one another like processes are, and as a result, threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like a process, a thread has its own program counter (PC), register set, and stack space.

Results and Analysis

Naive approach

The naive approach in the code handles matrix multiplication in a sequential and straightforward manner. It does not use parallel computing techniques, making it simpler but potentially less efficient, especially for large matrices. This method is best suited for scenarios where the complexity of parallelization does not justify the performance gain.

Results of naive approach (time in seconds):

Run1	Run2	Run3	Run4	Run5	Avg
0.004777	0.004668	0.004568	0.004889	0.004215	0.0045616

Table 1: Naive approach results

Avg Throughput = 1/0.0045616 = 219.22

Child processes approach

In the child processes approach, the task of matrix multiplication is distributed among multiple child processes created using "fork()" function. Each process calculates a portion of the matrix, potentially enhancing performance on multi-core systems. However, this method involves the overhead of inter-process communication and process management.

Results of child processes approach (time in seconds):

Processes	Run1	Run2	Run3	Run4	Run5	Avg
2	0.003266	0.002393	0.002415	0.003393	0.002141	0.003122
4	0.001853	0.002132	0.001394	0.002741	0.003174	0.002259
6	0.001758	0.001816	0.002900	0.001832	0.001672	0.001995

Table 2: Child processes approach results

Avg Throughput (2 processes) = 1/0.003122 = 320.31

Avg Throughput (4 processes) = 1/0.002259 = 442.67

Avg Throughput (6 processes) = 1/0.001995 = 833.98

Joinable Threads approach

The joinable threads approach utilizes multithreading for parallel computation, dividing the matrix multiplication task among several threads. These threads are joinable, which means the main thread waits for all threads to complete, allowing for easier synchronization and result collection. This method is generally more efficient than using multiple processes.

Results of joinable threads approach (time in seconds):

Threads	Run1	Run2	Run3	Run4	Run5	Avg
2	0.002574	0.002332	0.002617	0.002315	0.002500	0.002467
4	0.001485	0.001548	0.001553	0.001591	0.001548	0.001545
6	0.001003	0.001237	0.001306	0.001314	0.001033	0.001178

Table 3: Joinable threads approach results

Avg Throughput (2 threads) = 1/0.002467 = 405.35

Avg Throughput (4 threads) = 1/0.001545 = 647.25

Avg Throughput (6 threads) = 1/0.001178 = 848.90

Detached Threads approach

Detached threads are used in the final approach, where threads are created and then detached, allowing them to run independently of the main thread. This method is suitable for tasks where immediate synchronization with the main thread is not necessary. It can lead to performance gains but makes it harder to synchronize and retrieve results from the threads.

Measuring time in detached threads

In this approach, I faced a challenge in accurately determining the total execution time for all operations without using a waiting mechanism like sleep(). To address this, I added a statement to allow each thread to print its own execution time. By running the code 10 times, I obtained a maximum value for a single thread's execution time and combined it with the average thread creation time to estimate the total operation time. However, I observed that without a waiting mechanism like sleep(), the result matrix was sometimes incorrect, as the parent thread did not wait for the detached threads to complete their tasks.

Results of detached threads approach (time in seconds):

Threads	Run1	Run2	Run3	Run4	Run5	Avg	Max Thread time
2	0.00038	0.00026	0.00024	0.00027	0.00041	0.000032	0.3970
4	0.00079	0.00072	0.00089	0.00086	0.00068	0.000078	
6	0.000120	0.000134	0.000109	0.000122	0.000117	0.000120	

Table 4: Detached threads approach results

Avg thread creation time + Max Thread time (2 threads) = 0.397032sec

Avg Throughput (2 threads) = 2.5191

Avg thread creation time + Max Thread time (4 threads) = 0.397078

Avg Throughput (4 threads) = 2.5183

Avg thread creation time + Max Thread time (6 threads) = 0.397120

Avg Throughput (6 threads) = 2.5181

Summarization

Approach	Avg Execution Time (sec)	Throughput
Naive	0.0045616	219.22
Child Processes (2)	0.003122	320.31
Child Processes (4)	0.002259	442.67
Child Processes (6)	0.001995	833.98
Joinable Threads (2)	0.002467	405.35
Joinable Threads (4)	0.001545	647.25
Joinable Threads (6)	0.001178	848.90
Detached Threads (2)	0.397032	2.5191
Detached Threads (4)	0.397078	2.5183
Detached Threads (6)	0.397120	2.5181

Table 5: Summarization of Results

Conclusion & Learnings

To summarize, this study compares various strategies for handling large computational tasks. The naive approach, while straightforward, is less efficient for larger tasks. The child processes approach improves performance by distributing the task across multiple child processes, with throughput increasing as the number of processes grows. And joinable threads, which allow for effective synchronization and result collection, generally exceeds child processes in efficiency, with throughput also rising with more threads. Conversely, detached threads, despite their operational independence, struggle with synchronization issues and less predictable throughput, which decreases a little bit with more threads. This analysis highlights the importance of carefully selecting a method based on task size, required synchronization, and efficiency.

References

https://www.techopedia.com/definition/26327/child-process

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