Assignment 1: Process vs. Thread Performance Analysis

AGITHE YESHWANTH
Roll No: MT25059
Course: Graduate Systems (CSE638)

January 23, 2026

Contents

1 Introduction

The objective of this assignment is to analyze the performance differences between **Multi-Processing** (using `fork`) and **Multi-Threading** (using `pthread`).We considered three distinct workloads CPU-intensive, Memory-intensive and IO-intensive execution to a single CPU core.This gives us an idea/observation how the Operating System schedules tasks and manages resources for processes vs threads.

2 Implementation Details(Part A & part B)

2.1 Program A: Multi Process

Program A is implemented using the `fork()` system call.This creates child processes that run in separate isolated memory spaces.This ensures that if one process crashes won't affect other processes but increases higher overhead because of having duplicate page tables and file descriptors.

2.2 Program B: Multi Threaded

Program B is implemented using the `pthread_create()` library function. Threads are not like processes, they will share the same virtual address space(heap,data segments, text) of the parent process.It has faster creation and communication but requires synchronization.

2.3 Worker Functions

Each worker executes a loop based on the last digit of the roll number logic ($9 \times 1000 = 9000$ iterations base).

CPU Worker: Performs heavy floating-point arithmetic (`sqrt`, `tan`) inside a unoptimized loop to maximize the utilization of ALU.

Memory Worker: Allocates 50MB of integer arrays,writes data to them to force RSS usage, and then frees them.

IO Worker: Repeatedly writes data blocks to the disk using system calls (`write`), this forcing the process into the `io-wait` state.

3 Profiling and Measurement (Part C)

3.1 Methodology

To ensure accurate benchmarking, the following methodology was used:

- **Core Pinning:** We used `taskset -c 0` to force all execution onto a single CPU core. This simulates a contended environment where parallelism is limited by hardware.
- **Tools:** We utilized `top` for real-time CPU and Memory monitoring, and `du` to verify disk writes.
- **Sampling:** A Bash script automated the data collection, sampling system metrics every 0.1 seconds.

3.2 Results Table

The following data was collected for 2 concurrent workers.

Table 1: Performance Comparison (Processes vs. Threads)

Program	Worker	CPU %	Mem (KB)	Disk (KB)	Time (s)
A (Fork)	CPU	98.16%	2432	0	2.885927535
A (Fork)	MEM	3.85%	2688	0	4.395452102
A (Fork)	IO	4.88%	2944	572	4.824248590
B (Thread)	CPU	97.64%	2048	0	2.758253058
B (Thread)	MEM	2.98%	2048	0	4.394685237
B (Thread)	IO	4.81%	1792	572	4.850611389

3.3 Observations & Analysis

1. **CPU intensive:** Both programs reach $\sim 98\%$ CPU utilization. Program B was slightly faster due to less cost for context-switching (i.e., program_b takes (2.76s), while program_a takes (2.89s)). Memory required for program_a is 2432 KB, while program_b takes only 2048 KB, this shows that memory over head of maintaining separate address spaces for processes.

2. **Memory intensive:** CPU utilization dropped significantly to $\sim 3-4\%$ because of only using memory.

3. **I/O-Intensive:** CPU usage dropped to $\sim 3.8\%$, confirming the CPU was idle waiting for disk writes (Uninterruptible Sleep).

4 Scaling Analysis (Part D)

4.1 Execution Time Scaling

As we increased the number of workers on a single core, execution time for CPU tasks increased **linearly**.

- 2 processes took $\sim 5.5s$, while 4 processes took $\sim 11.0s$.
- **Reasoning:** The OS must time slice the single ALU among workers. Adding more workers divides the processing power, increasing completion time.

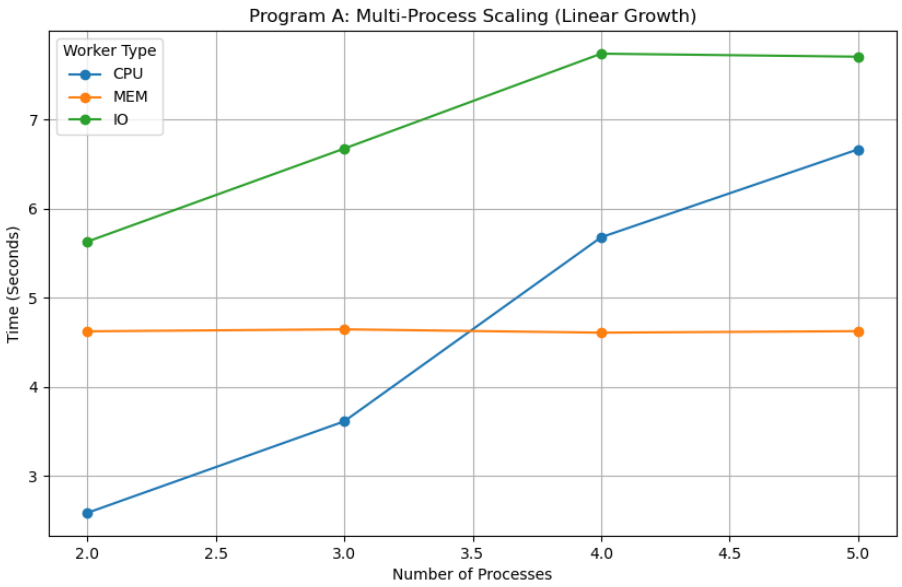

Figure 1: Process Scaling: Linear increase in execution time.

4.2 CPU Usage Scaling

While execution time increased, the total CPU usage of the system remained saturated at 100% (split among workers). More importantly, we observed distinct memory behavior:

- **Processes:** Memory usage increased linearly with count.
- **Threads:** Memory usage remained constant.

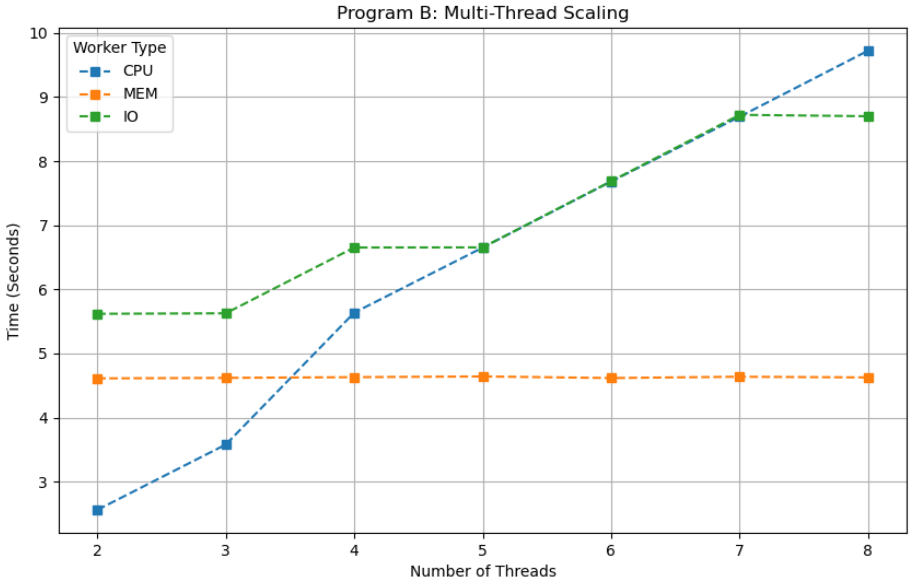

Figure 2: Thread Scaling: Constant memory usage.

5 Deliverables & Declarations

5.1 GitHub Repository

The complete source code, automation scripts, and raw CSV data can be found at:
https://github.com/agithe25059/GRS_Assignments.git

5.2 AI Usage Declaration

- **Generative AI Tool Used:** Google Gemini
- **Purpose:** Debugging C syntax errors, optimizing Bash scripts for data sanitization, and refining worker loops, also for correct sentence formation in Report.
- **Verification:** Logic verified against `man` pages; data generated via local execution.