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: In this worker function will allocating 50MB of integer arrays and forcing RSS usage by writing data to them to make high-capacity RAM bottleneck.

IO Worker: Repeatedly writing data blocks to the disk by (`write`) system calls , this forcing the process to `io-wait` state.this forcing the os to pause the process execution until the IO operation s complete.This repeated System calls ensure the task spends more time of execution on the hardware responses rather than performing computation(active).

3 Profiling and Measurement (Part C)

3.1 Methodology

To ensure accurate benchmarking, the following methodology was used:

Core Pinning: All execution are done on single CPU core for that i used `taskset -c 0` to force all execution to be on a single CPU core.This ensures the elimination of multi-core parallel processing and forces the OS manage resources by time-slicing.

Tools: We utilized `top` for real-time CPU and Memory monitoring and `disk usage(du)` to verify disk writes.This allows to measure the CPU utilization and RSS(Resident set size) metric of memory at regular intervals.

Sampling: Execution time is measured by high-precision timestamps `data +%s.%N`,While the throughput of the disk is verified by calculating size of generated actifact of `inOut_*.dat`.

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 using memory more than cpu(i.e mostly using memory only).The memory used by the program_a is significantly more than program_b, this is because the Address space for the threads are shared whereas for processes there are created separately.

3. **I/O-Intensive:** CPU usage dropped to $\sim 3.8\%$, this explains how much the CPU is waiting while the inOut_*.dat files wre synchronously written to disk.

4 Scaling Analysis (Part D)

4.1 Execution Time Scaling

Based on the experimental data, it demonstrates how the operating system(OS) handles increasing concurrent load of various worker types when restricted to a single CPU core.

- **CPU-Intensive Scaling:** Here the observation is in both Program A and Program B, the CPU worker shows clear linear increment of time as the no.of instances increases. In precess graph the time increases from approximately 4.9s for two processes to 11s for 5 processes. similarly the thread graph also incrreases from 4.9s to 16.3s at threads.

- **Reasoning:** The OS must time slice the single ALU among workers. This is because as the instances increases the time-sliced among all active workers increases by receiving fewer cycles per sec,so it results in total completion time proportionally.

- **Memory-Intensive Scaling:** The MEM worker remains flat in both the graphs, holding approximately 4.8sec to 5.0 sec. This indicates that as the instances increasing the memory allocation and deallocation cycles for 50MB blocks approximately remains constance.
- **I/O-Intensive Scaling:** It shows the stepped increment as the no.of instances increases from 3sec to 5.9 at 5 processes in program a. In program b it is scaled from 3.1sec to 8.1sec at 8 threads.

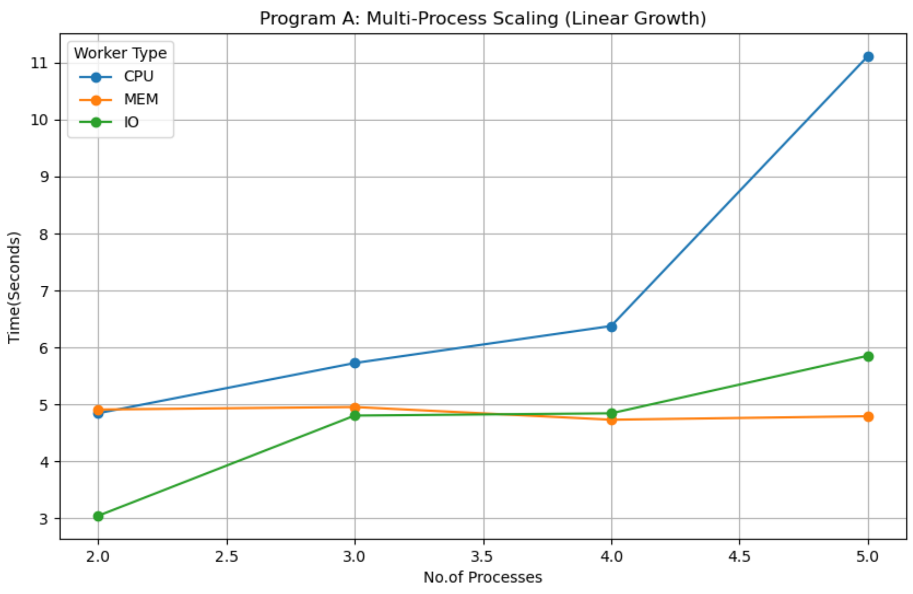

Figure 1: Process Scaling: Linear increase in execution time.

4.2 CPU Usage Scaling

Whilw the execution time increases as the no.of workers, the overall CPU utilization behavior differs significantly between model to model as we implement.

- **Processes:** Memory usage increased linearly with count.context switching impacting in the process switching.
- **Threads:** Memory usage remained constant.While threads are generally lightweight so fast context switching.Unlike processes, adding new thread did not increases memory footprint significantly, as well as they will share the same heap and global data segments.

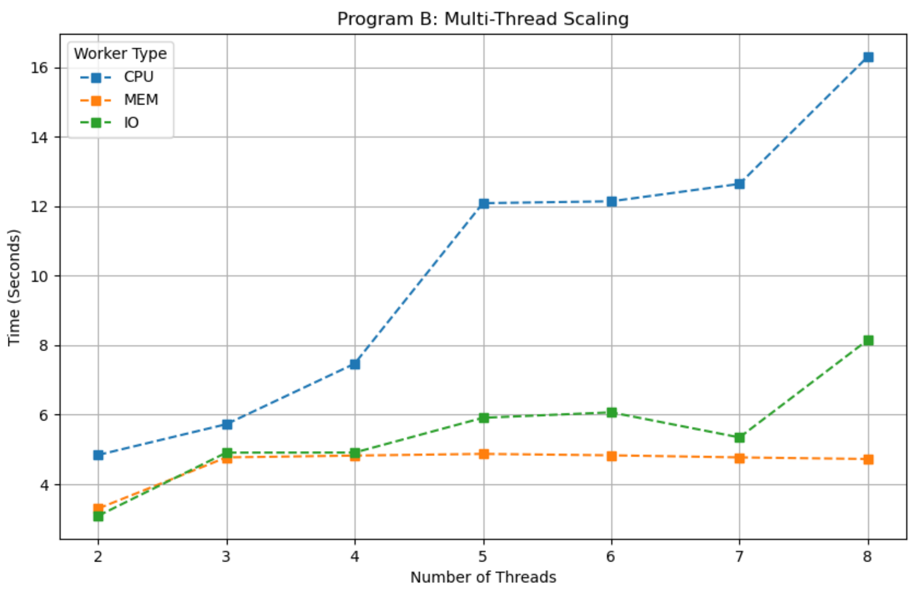

Figure 2: Thread Scaling: Constant memory usage.

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/tree/main/GRS_PA01`

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 some correct sentence formation in Report.
- **Verification:** Logic verified against `man` pages; data generated via local execution.