

Graduate Systems CSE638 - Assignment PA01

Arya Singh | MT25019

January 23, 2026

Github repository link: <https://github.com/aryas1ngh/GRS>

1 Approach to solution

The objective of this assignment was to analyze the performance characteristics and resource consumption differences between processes, created via `fork()` and POSIX Threads, created via `pthread_create()`.

To achieve this, two C programs were developed:

- **Program A (Processes):** Creates multiple child processes to perform a specific task.
- **Program B (Threads):** Creates multiple threads within a single process to perform the same task.

Three distinct worker functions were implemented to stress specific system resources:

- **CPU Intensive:** Performs complex mathematical calculations (sin, cos, square roots) in nested loops to saturate the CPU.
- **Memory Intensive:** Allocates a large buffer (100MB per worker) on the heap and writes to it repeatedly to stress the memory management unit.
- **I/O Intensive:** Performs repetitive text output operations using `fprintf()` inside a nested loop structure. To enforce true I/O latency and prevent the operating system from simply buffering the data in RAM, `fsync()` is called after every batch of writes, forcing the disk controller to physically commit the data.

Automation was achieved using Bash scripts to iterate through varying counts of workers (from 2 to 5 for processes, and 2 to 8 for threads), utilizing tools like `taskset` for CPU pinning, `top` for CPU/Memory monitoring, and `iostat` for Disk throughput measurement.

2 System Setup

The experiments were conducted in a controlled Linux environment (Docker container) to ensure `taskset` works on macOS system. Key configurations included:

- **CPU Pinning:** All programs were strictly pinned to **CPU Core 0** using the command `taskset -c 0`. This forced all processes and threads to share a single execution pipeline, highlighting the effects of context switching and time-slicing.
- **Monitoring Tools:**
 - `top -b -n 1`: Used to capture CPU utilization (%) and memory usage.
 - `ps -o rss`: Used to capture resident set size in KB.
 - `iostat -d -k 1 2`: Used to capture real-time disk write throughput (kB/s).
 - `time`: Used to measure the total wall-clock execution time.
- **Compiler:** `gcc` with `-pthread` flag and `-lm` for maths libraries.

The output of the monitoring commands is pipelined to `grep` and `awk` commands to get required values and store them in CSVs with following fields: program type (A/B), variant (CPU, memory, I/O), CPU%, memory usage in KB, I/O usage in KB/s.

3 Resulting Plots & Data

The following figures visualize the execution time, CPU Usage, and memory usage for each program variant.

3.1 CPU Intensive Workload

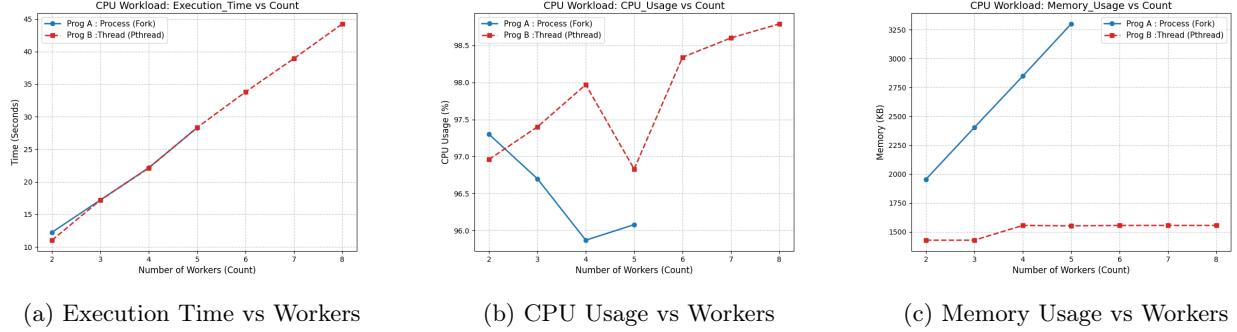


Figure 1: **CPU Intensive Task:** Execution time increases linearly as workers share the single core. CPU usage is saturated, while memory usage remains negligible.

3.2 Memory Intensive Workload

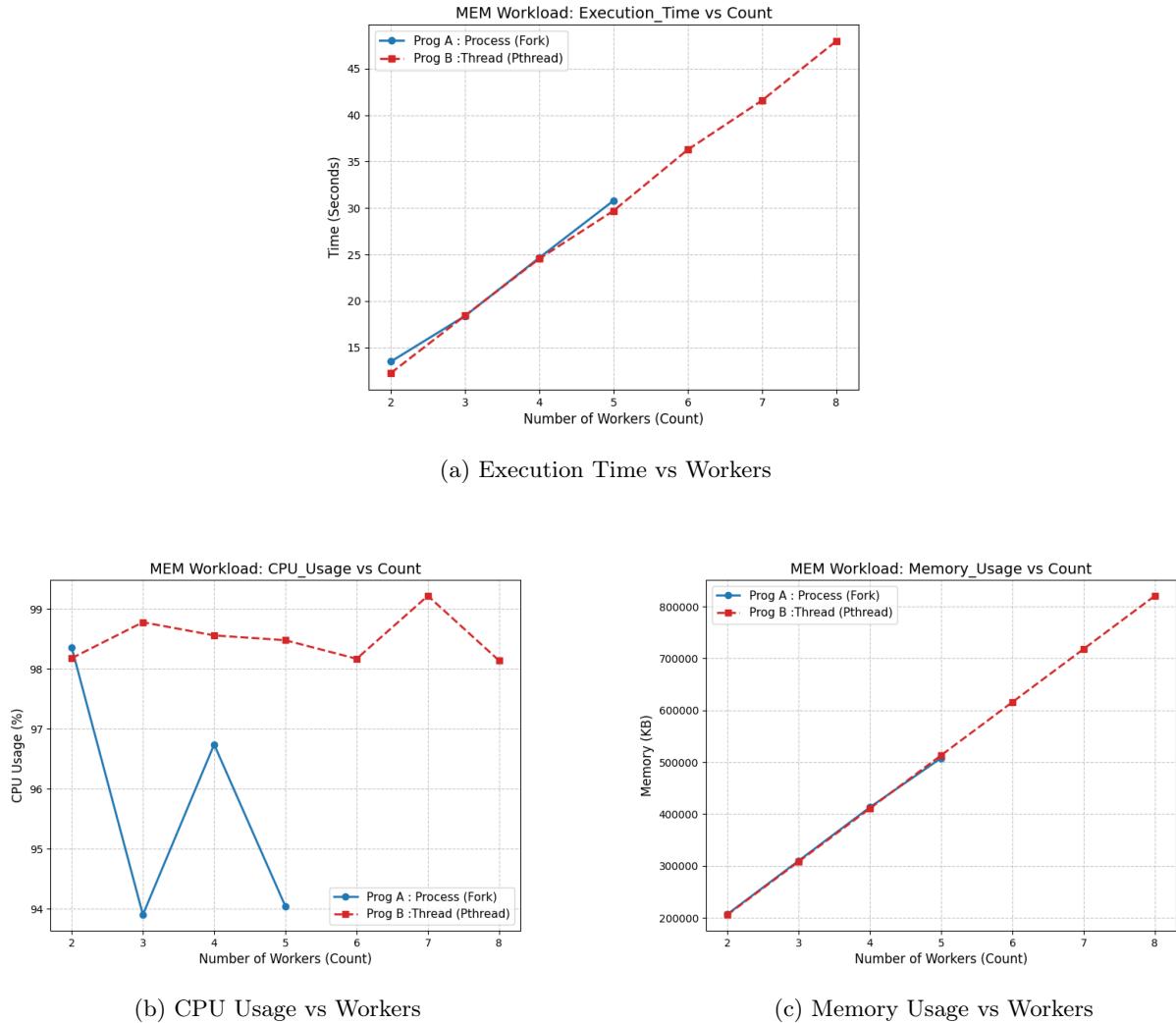


Figure 2: **Memory Intensive Task:** Both Time and Memory usage scale linearly. The CPU remains busy managing memory allocations and page faults.

3.3 I/O Intensive Workload

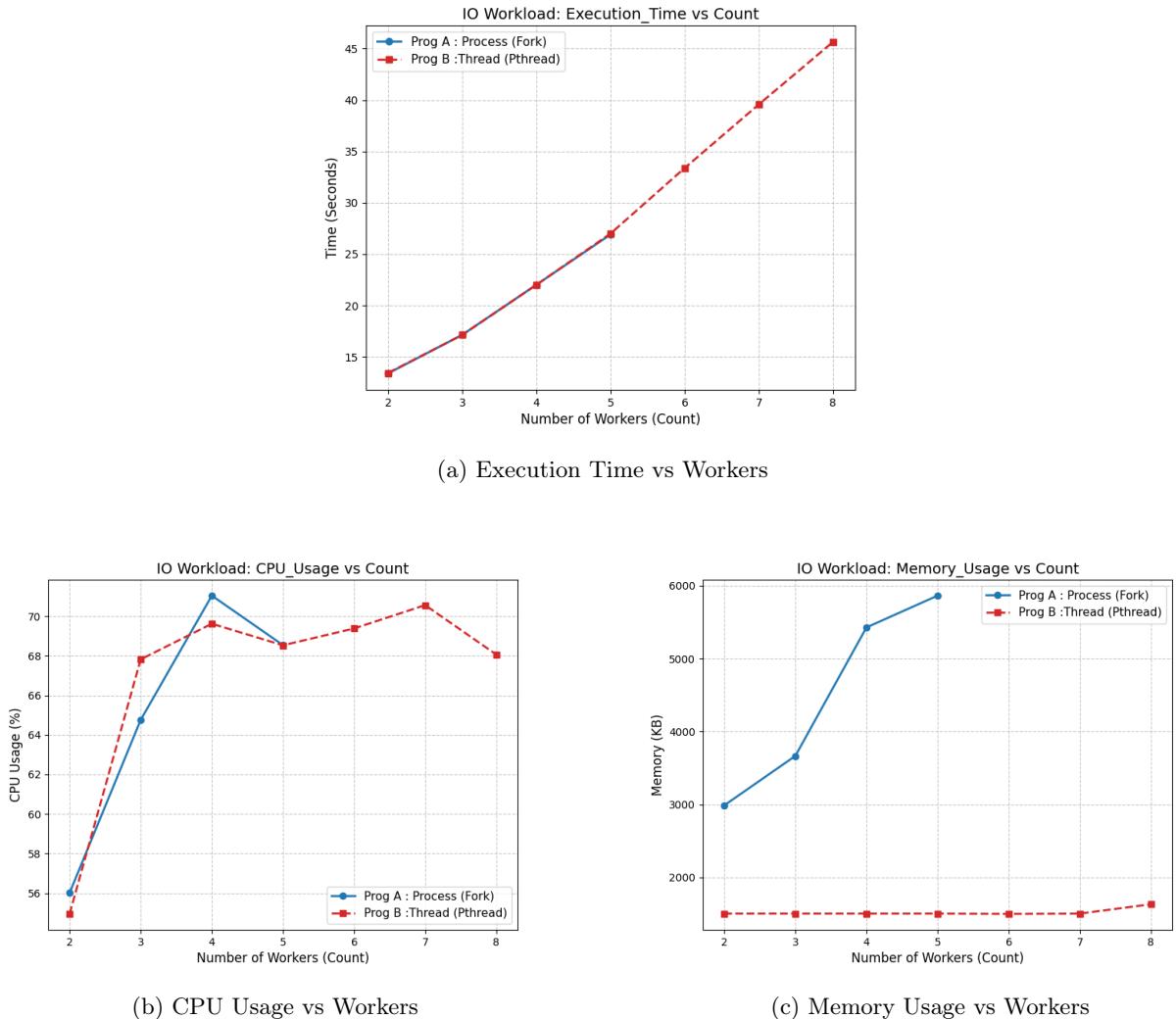


Figure 3: **I/O Intensive Task:** Execution time increases significantly due to disk contention. CPU usage drops (< 70%) as threads block waiting for I/O completion.

4 Explanation and Discussion

4.1 CPU Intensive Analysis

- Figure 1a graph shows a perfect linear relationship between worker count and execution time. Since the experiment was pinned to a single core, 2 workers took approx. 11 seconds, and 4 workers took approx. 22 seconds. This validates the **time slicing** mechanism of the Linux scheduler, since the CPU simply divides its cycles between threads, resulting in no throughput gain but proportional latency increase.
- As seen in Figure 1b, the CPU remained saturated ($\approx 99\%$), confirming the workload was purely computational and context-switch overhead was minimal.

4.2 Memory Intensive Analysis

- As shown in Figure 2a similar to the CPU task, time scales linearly. This is expected because memory operations (memset loops) are CPU instructions. The memory bandwidth of the system was likely not saturated by just 8 workers, which is why the bottleneck remained the single CPU core driving the allocations.
- As seen in Figure 2c, the linear growth in memory usage confirms that both processes and threads correctly allocated independent 100MB buffers, stressing the system's virtual memory subsystem.

4.3 I/O Intensive Analysis

- The time graph in Figure 3a shows a steep increase. Unlike CPU tasks where time doubles with count, I/O tasks often suffer from **contention overhead**. As more threads compete for the file system lock to write to the single disk, the wait times compound, making the total execution time grow faster than the ideal linear case.
- The graph in Figure 3b provides the critical insight, since we see that CPU usage drops to 60-70%. This missing 30% of CPU time represents the threads being in a **blocked state**, waiting for the slow physical disk to acknowledge the `fsync()` command. This proves the system was I/O bound.

4.4 Experimental validity and limitations

Even though the code was run inside a Docker container to provide a controlled Linux environment on macOS, the results are still influenced by host OS scheduling, hypervisor overhead, and shared hardware resources. CPU pinning via `taskset` ensures logical core isolation within the container, but the underlying macOS kernel ultimately schedules the container runtime.

Also, system monitoring tools such as `top` and `iostat` rely on sampling-based measurements, which may introduce minor temporal inaccuracies. However, by averaging over multiple samples, the impact of short-term fluctuations was minimized.

The experiments were also executed on a single physical disk, which represents a *worst-case contention* scenario for I/O workloads. In real-world systems, parallel disks or SSDs would reduce this contention significantly.

5 Conclusion

This experiment successfully highlighted the fundamental behaviors of OS schedulers and resource management. Adding more workers to a CPU-bound task on a single core does not necessarily improve performance, it simply divides the available time. For I/O-bound tasks, excessive concurrency can degrade performance due to contention and locking overhead. In modern Linux, the performance gap between processes and threads for simple computational tasks is smaller than expected, though threads remain lighter in terms of creation/teardown overhead (not measured here).

6 AI Declaration

I utilized generative AI (Gemini) during this assignment to assist with generating boilerplate code, debugging C syntax errors regarding `pthread` arguments, writing bash scripts to automate the data collection using `top` and `iostat`. All analysis, and final code implementation were verified and executed by myself.