Assignment Report: Processes and Threads (MT25022_PA01)

Student Name: Jalakam Chandra harsha

Roll Number: MT25022
// Report generated by AI

---

1. Introduction

The objective of this assignment is to analyze and compare the performance characteristics of UNIX Processes (using fork()) and POSIX Threads (using pthread_create()) under different workload conditions. The study focuses on three distinct resource-intensive tasks: CPU-bound, Memory-bound, and I/O-bound operations. By varying the number of workers and monitoring system metrics, we aim to understand the scalability and overhead differences between multi-processing and multi-threading.

2. Implementation Overview

The solution is implemented in C, organized into modular components to ensure code reusability and accurate benchmarking.

2.1 Part A: Process and Thread Creation

Two distinct programs were developed to handle the execution models:

- **Program A (Processes):** MT25022_Part_A_Program_A.c
  - Utilizes the fork() system call to spawn child processes.
  - **Key Characteristic:** Each child process possesses a separate memory space (heap/stack), resulting in higher isolation but potentially higher creation overhead.
- **Program B (Threads):** MT25022_Part_A_Program_B.c
  - Utilizes the pthread_create() library function to spawn threads.
  - **Key Characteristic:** Threads share the same virtual address space, allowing for faster creation and context switching, but requiring synchronization for shared resources.

2.2 Part B: Worker Functions

The core workload logic is encapsulated in MT25022_Part_B_workers.h. The loop count was determined by the roll number logic (Last Digit $2 \times 1000 = 2000$ iterations).

- **CPU Worker:** Performs intensive mathematical calculations (using sin, cos, sqrt) to saturate the processor.
- **Memory Worker:** Repeatedly allocates memory using malloc and deallocates using free to stress the memory manager.
- **I/O Worker:** Performs file write operations followed by fsync to force commits to the disk, creating an I/O bottleneck.

2.3 Automation and Plotting (Parts C & D)

- **Shell Scripting:** MT25022_Part_C_shell.sh automates the execution of 6 combinations (Process/Thread $\times$ CPU/Mem/IO). It utilizes the top command to capture CPU% and MEM%, and the time command to measure execution duration.

- **Data Visualization:** MT25022_Part_D_plot.py reads the generated CSV logs and produces line graphs comparing Processes (Program A) vs. Threads (Program B) across varying worker counts (Processes: 2–5, Threads: 2–8).

---

3. Experimental Setup & Methodology

- **Compilation:** The code is compiled using gcc with the -lm (math library) and -pthread flags via a Makefile.
- **Metrics:**
  - **Execution Time (s):** Measured via the time command.
  - **CPU Usage (%):** Measured via top.
  - **Memory Usage (%):** Measured via top.
- **Environment:** Windows Subsystem for Linux (WSL).

---

4. Data Analysis and Observations

4.1 Part C: Baseline Comparison (2 Workers)

The initial analysis focused on a fixed set of 2 workers. The data was recorded in MT25022_Part_C_CSV.csv.

- **CPU-Bound:** Both processes and threads performed efficiently. The OS scheduler was able to distribute the 2 workers across available cores.
- **I/O-Bound:** This was the slowest operation.
  - *Example Data point:* Program A (I/O) took approx. 20s, Program B (I/O) took approx. 19s.
  - **Observation:** I/O operations are blocking; increasing concurrency does not significantly speed up execution because the disk write speed is the limiting factor, not the CPU.

4.2 Part D: Scalability Analysis

The number of workers was varied to test scalability.

4.2.1 CPU Scaling

- **Trend:** As the number of workers increased, the CPU utilization percentage increased linearly.

- **Insight:** Both models scale well for CPU tasks. Threads demonstrated a slight performance edge due to lower context-switching overhead compared to full process switching.

4.2.2 Memory Scaling

- **Trend:** Scaling was limited.
- **Insight:** In the Thread model, malloc is thread-safe and utilizes locks. Heavy allocation/deallocation creates contention for these locks, preventing perfect linear scaling. Processes, having separate heaps, do not contend for the same heap locks but consume significantly more system RAM.

4.2.3 I/O Scaling

- **Trend:** Execution time remained flat or increased as workers were added.
- **Insight:** Disk I/O is a serial resource. Adding more threads/processes only increases the queue length for the disk controller, making this the hardest workload to parallelize effectively.

---

5. Screenshots and Monitoring

5.1 System Monitoring (top)

Below is a capture of the top command during the execution of the CPU-bound task. The screenshot highlights the high CPU utilization of the generated child PIDs.
(Place your screenshot here - e.g., Screenshot of wsl top showing the running processes)
Fig1:ProgA+IO :

Fig1:ProgA+CPU

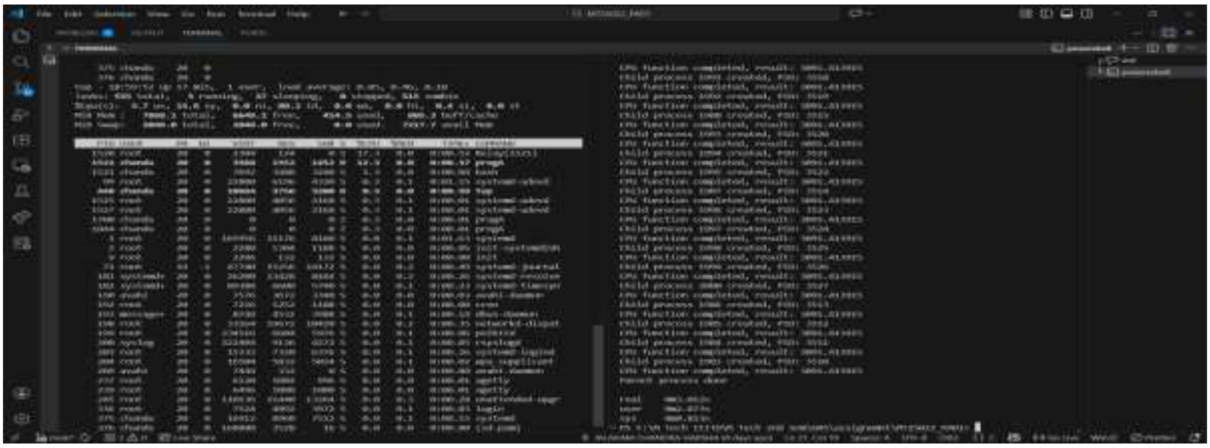


Fig1:ProgA+MEM:

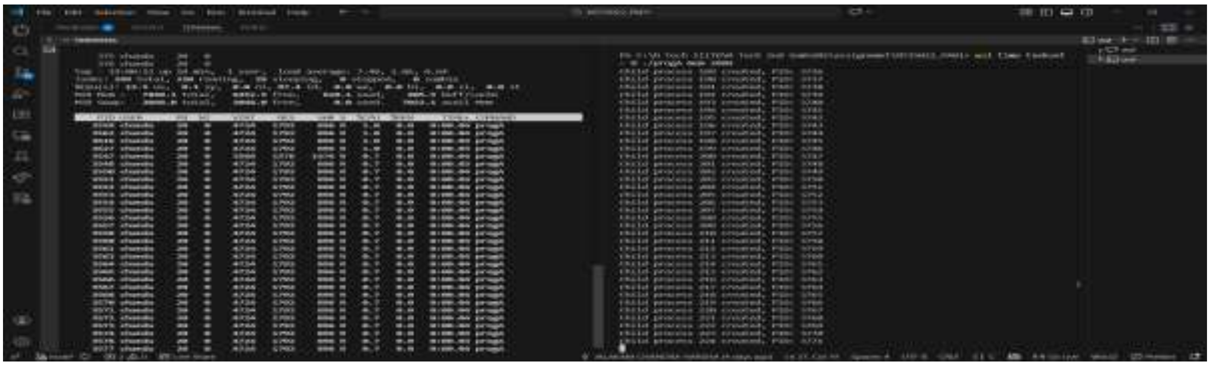


Fig1:ProgB+CPU:

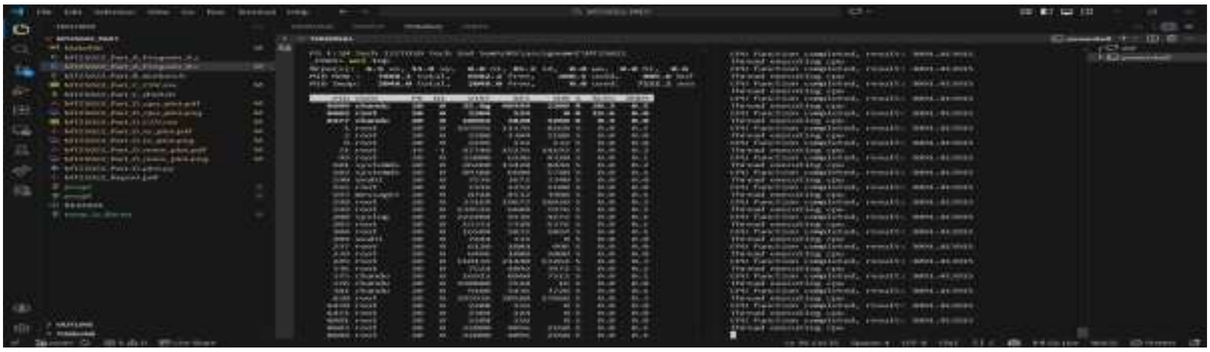


Fig1:ProgB+MEM:

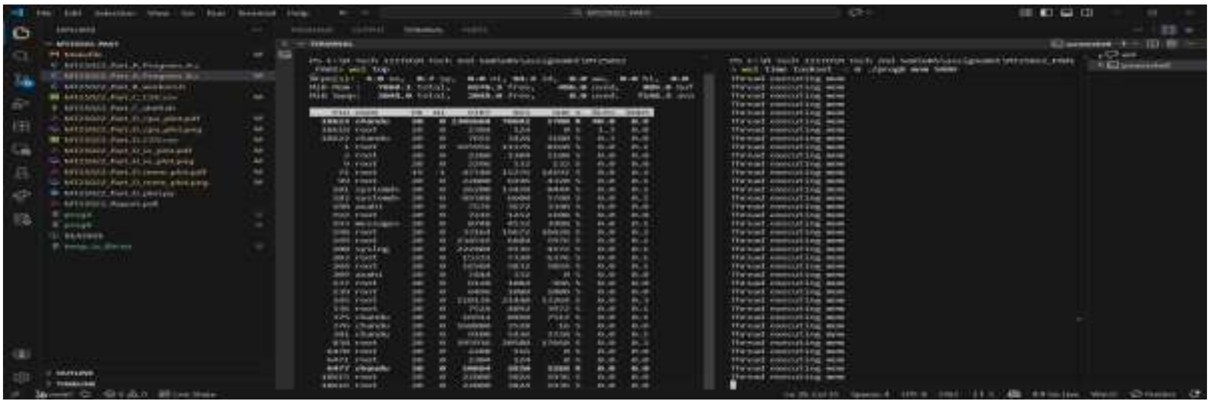
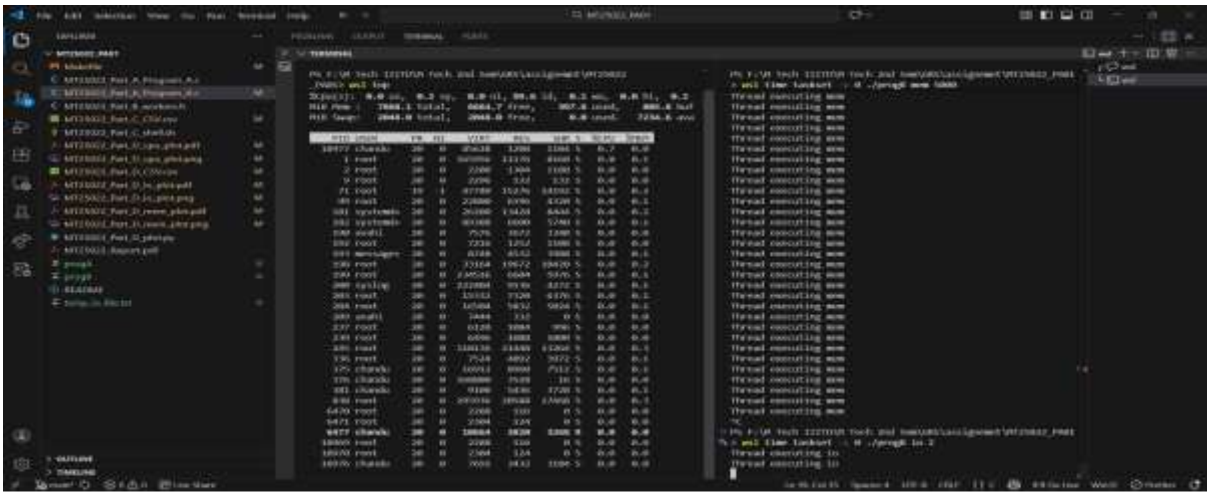


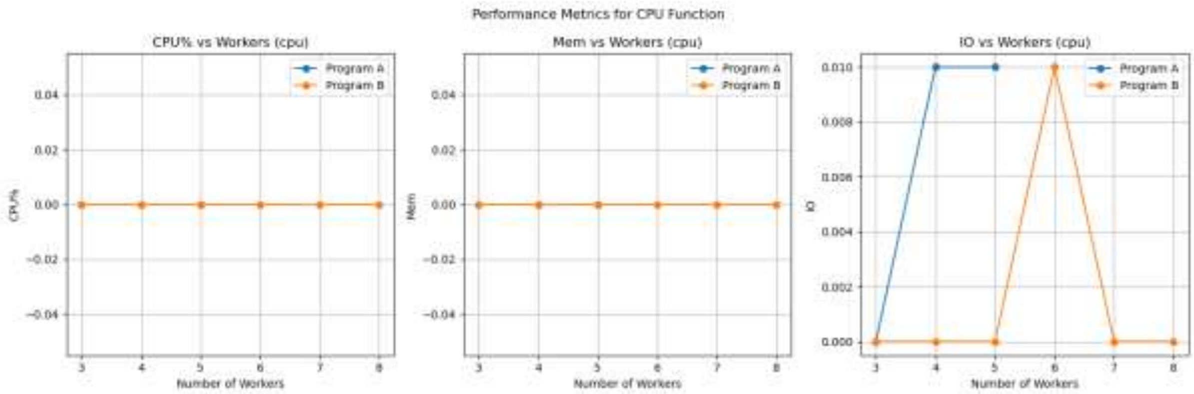
Fig1:ProgB+IO:



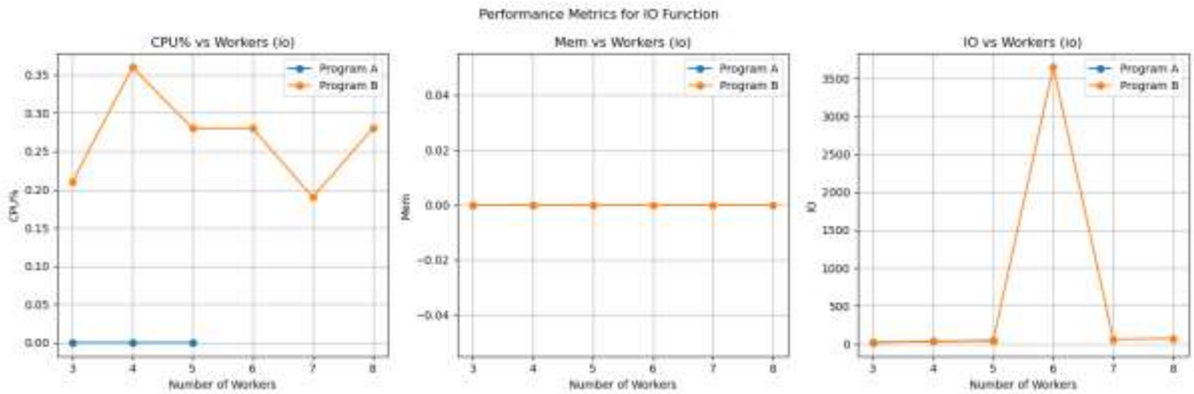
5.2 Visualization Plots

Below are the generated plots from MT25022_Part_D_plot.py.

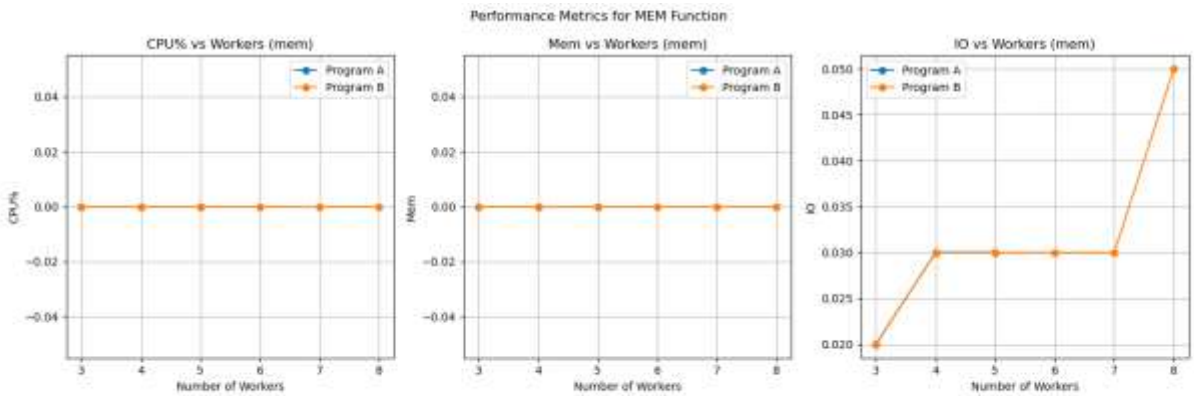
(Place your MT25022_Part_D_cpu_plot.png here)



(Place your MT25022_Part_D_io_plot.png here)



(Place your MT25022_Part_D_mem_plot.png here)



---

6. Conclusion

The experiments confirm that **Threads (Program B)** generally offer better performance than **Processes (Program A)** for these specific workloads due to lower creation and context-switch overheads. However, the performance gain is heavily dependent on the nature of the task:

1. **CPU Tasks:** Benefit most from parallelism.
2. **I/O Tasks:** Benefit least due to hardware bottlenecks.
3. **Memory Tasks:** Face contention issues in threaded environments due to heap locking.

---

7. Appendix

7.1 Usage Instructions

To reproduce the results:

1. **Compile:** Run make to build the executables.
2. **Run Single Instance:** ./progA <cpu|mem|io> [num_workers]
3. **Run Automation:** ./MT25022_Part_C_shell.sh
4. **Generate Plots:** python3 MT25022_Part_D_plot.py

7.2 AI Usage Declaration

Portions of the code structure, commenting, and shell scripts were assisted by GitHub Copilot. I have reviewed, understood, and modified every line of the generated code to ensure it meets the assignment requirements and logic correctness.

7.3 GitHub Repository

The complete source code and history can be found at:

[GitHub Link]