PA01: Processes and Threads

Graduate Systems (CSE638)

Aditya Saiteja B
Roll No: MT25058

1 Introduction

Modern operating systems support concurrency mainly through *processes* and *threads*. While both allow multiple tasks to run at the same time, they differ significantly in terms of memory isolation, creation overhead, and performance. Processes are heavier and isolated, whereas threads are lightweight and share memory.

The objective of this assignment is to study these differences practically by implementing process-based and thread-based programs and executing different types of workloads such as CPU-intensive, memory-intensive, and I/O-intensive tasks. Additionally, the assignment explores how performance changes when the number of processes or threads is increased.

This experiment helped bridge theoretical operating system concepts with real system behavior observed on a Linux machine.

2 System Configuration

All experiments were performed on the following system:

- Operating System: Ubuntu Linux
- Number of Cores: 8
- RAM: 16 GB
- Storage: SSD

Tools Used

- `fork()` for process creation
- `pthread_create()` for thread creation
- `taskset` for CPU pinning
- `top` for CPU usage monitoring
- `iostat` for disk I/O statistics

- `time` for execution time measurement

CPU pinning was used to ensure consistent and repeatable measurements.

3 Part A: Program Implementation

3.1 Program A: Process-Based Execution

Program A creates multiple child processes using the `fork()` system call. As specified in the assignment, the parent process does not perform any work and is not counted as part of the workload.

Each child process executes one of the worker functions (`cpu`, `mem`, or `io`) based on command-line arguments.

Observations:

- Each process has its own address space
- Higher memory usage compared to threads
- Strong isolation between workers

3.2 Program B: Thread-Based Execution

Program B creates worker threads using POSIX threads. The main thread is excluded from the workload, and only worker threads execute the assigned functions.

All threads share the same address space, which allows faster communication but may cause contention in memory-intensive workloads.

Observations:

- Threads are lightweight
- Faster creation and context switching
- Shared memory improves performance in some cases

4 Part B: Worker Functions

Three worker functions were implemented and used by both programs.

4.1 CPU-Intensive Worker

This worker performs heavy mathematical computations inside a loop. The goal is to keep the CPU busy for most of the execution time, testing CPU utilization and parallel execution efficiency.

4.2 Memory-Intensive Worker

This worker allocates large arrays and repeatedly accesses them. It stresses the memory subsystem and highlights cache behavior and memory contention.

4.3 I/O-Intensive Worker

This worker repeatedly reads from and writes to a file. Most of the execution time is spent waiting for disk I/O, allowing observation of blocking behavior and CPU idling.

5 Part C: Performance Comparison

Six experiments were conducted using combinations of processes and threads with the three worker functions.

5.1 CPU-Intensive Results

Processes achieved approximately 167% CPU usage, while threads achieved about 155%. Both models performed similarly, indicating that CPU-bound workloads scale well with either approach.

5.2 Memory-Intensive Results

Processes achieved around 108% CPU usage, while threads achieved about 81%. Processes performed better due to separate address spaces, which reduced cache contention present in shared-memory threads.

5.3 I/O-Intensive Results

Processes used around 17% CPU with an execution time of 1.77 seconds, while threads used around 19% CPU with an execution time of 1.52 seconds. Threads were slightly faster due to lower context-switch overhead.

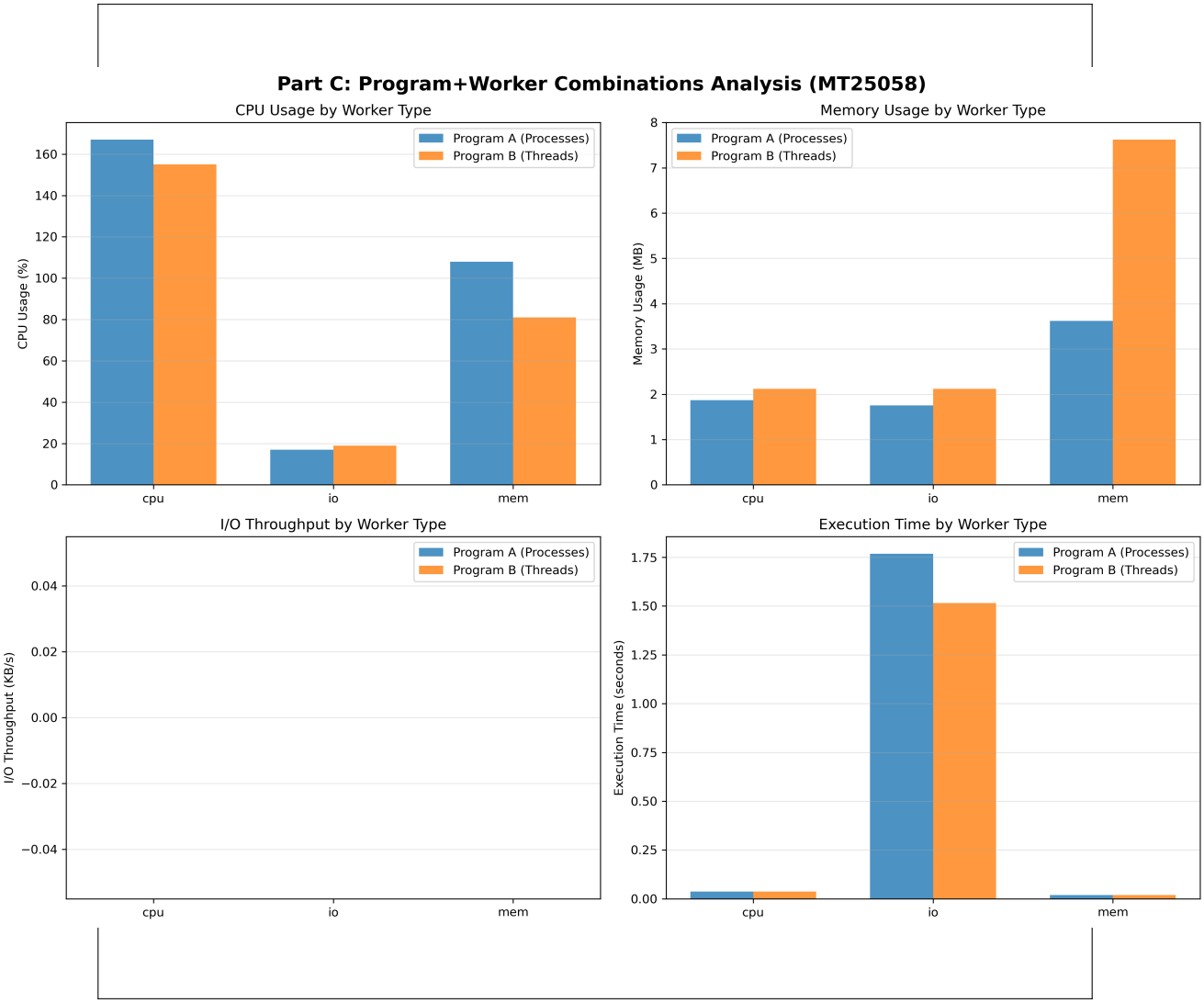

Figure 1: Part C: Processes vs Threads Performance Comparison

6 Part D: Scaling Analysis

6.1 Process Scaling

The number of processes was increased from 2 to 5. CPU usage increased from 162% to 396%. Performance improved until four processes, after which diminishing returns were observed due to scheduling and memory overhead.

6.2 Thread Scaling

The number of threads was increased from 2 to 8. CPU usage increased from 153% to 502%. Threads scaled better than processes and continued to utilize CPU efficiently beyond the number of physical cores.

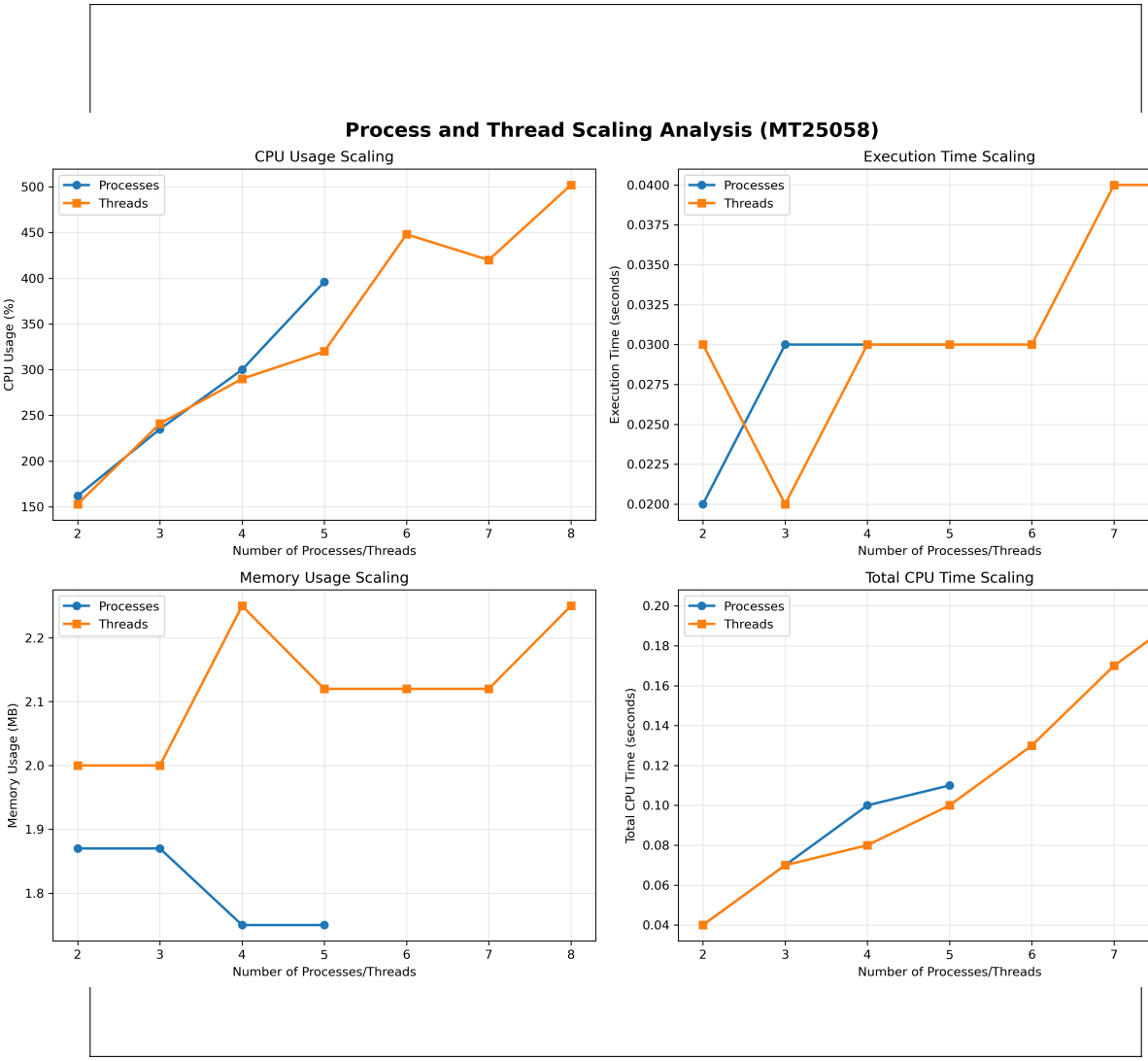

Figure 2: Part D: Scaling Analysis

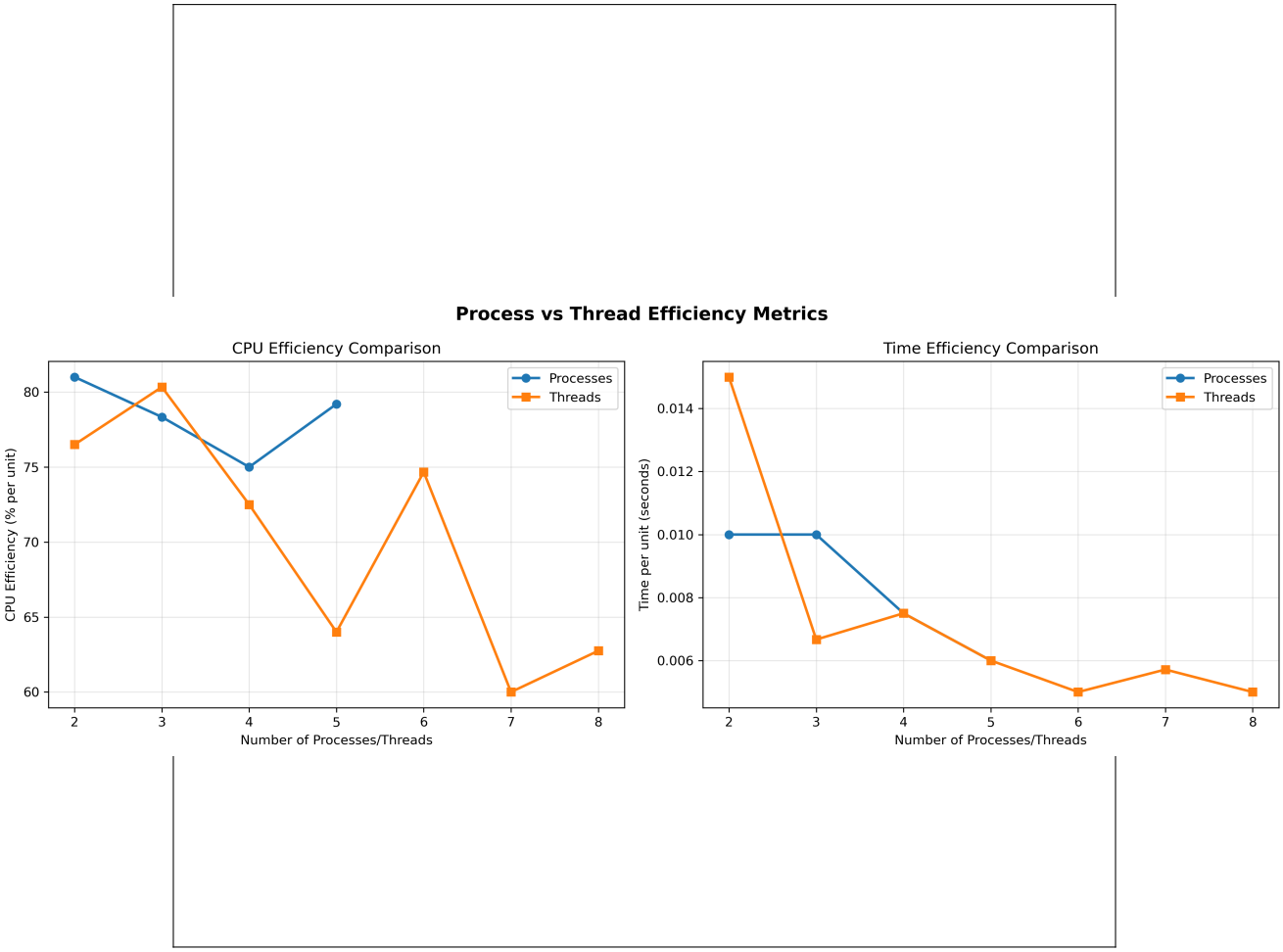

Figure 3: Efficiency Metrics for Processes and Threads

7 Key Observations

- CPU-bound workloads perform similarly with processes and threads
- Memory-intensive workloads favor processes due to isolation
- I/O-intensive workloads benefit from threads
- Threads scale better on multi-core systems
- Creating more processes than cores leads to inefficiency

8 AI Usage Declaration

AI tools (ChatGPT) were used for understanding the assignment requirements, structuring the report, and improving clarity. All code and analysis were reviewed and fully understood by the author, who is capable of explaining the work during viva.

9 Conclusion

This assignment provided hands-on experience with process and thread management in Linux. The results clearly show that threads offer better scalability and performance for most workloads, while processes provide better isolation and stability. The choice between processes and threads should depend on workload characteristics and system requirements.

GitHub Repository

https://github.com/aditya25058/MT25058_PA01