Indraprastha Institute of Information Technology Delhi

Department of Computer Science and Engineering

Graduate Systems (CSE638)

Assignment - 01
Processes and Threads

Student Name: Jayaditya Mishra
Roll Number: MT25070
Program: M.Tech (Computer Science and Engineering)
Semester: Second Semester
Instructor: Dr. Rinku Shah

Submitted in partial fulfillment of the requirements for the course

Graduate Systems (CSE638)

Submission Date: January 23, 2026

Contents

1 Introduction

Modern operating systems support concurrency through two primary abstractions: *processes* and *threads*. While processes provide strong isolation via separate address spaces, threads offer lightweight parallelism within a shared address space.

This assignment investigates the performance impact of these abstractions by evaluating CPU utilization, memory consumption, and execution time across different workload types.

2 System Setup and Experimental Environment

This section describes the hardware, software, and tools used for conducting the experiments in this assignment. Clearly specifying the experimental environment ensures reproducibility and provides context for the observed performance results.

2.1 Hardware and Operating System

Platform: Windows 11 with WSL2 (Windows Subsystem for Linux)
Linux Distribution: Ubuntu 24.04 (WSL2)
Architecture: x86-64
Processor: Multi-core CPU with 12 logical cores
Main Memory: 16 GB RAM

3 Process and Thread Creation (Part A)

3.1 Program A: Process Creation using fork()

In Program A, multiple processes are created using the `fork()` system call. As per the assignment specification, the parent process creates the required number of child processes, while the parent itself is not considered a worker. Each child process independently executes one of the worker functions (`cpu`, `mem`, or `io`). This program demonstrates process-level parallelism, where each process has its own address space and execution context

3.2 Program B: Thread Creation using pthread

In Program B, multiple threads are created using the POSIX threads (`pthread`) library. As specified in the assignment, the main thread is not counted as a worker. Each created thread executes one of the worker functions. Unlike processes, threads share the same address space, enabling lightweight parallel execution and efficient data sharing.

4 Worker Function Design (Part B)

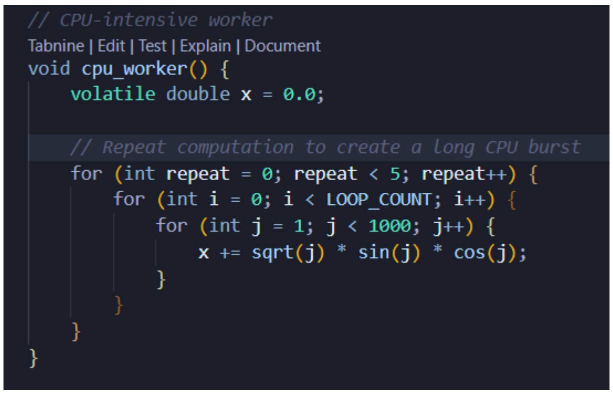
As specified in the assignment, three distinct worker functions were implemented to model CPU-intensive, memory-intensive, and I/O-intensive workloads. These worker functions are executed by both processes (Program A) and threads (Program B).

4.1 Loop Count Configuration

Each worker function runs a loop whose iteration count is derived from the last digit of the roll number, multiplied by 10^3, as required by the assignment. Since the roll number is **MT25070** and the last digit is **0**, the value **9** is used instead, resulting in the following loop count:

$$\text{Loop Count} = 9 \times 10^3 = 9000$$

4.2 CPU-Intensive Worker (`cpu`)



```c
// CPU-intensive worker
Tabnine | Edit | Test | Explain | Document
void cpu_worker() {
    volatile double x = 0.0;

    // Repeat computation to create a long CPU burst
    for (int repeat = 0; repeat < 5; repeat++) {
        for (int i = 0; i < LOOP_COUNT; i++) {
            for (int j = 1; j < 1000; j++) {
                x += sqrt(j) * sin(j) * cos(j);
            }
        }
    }
}
```

Figure 1: CPU-Intensive Worker Function (`cpu`)

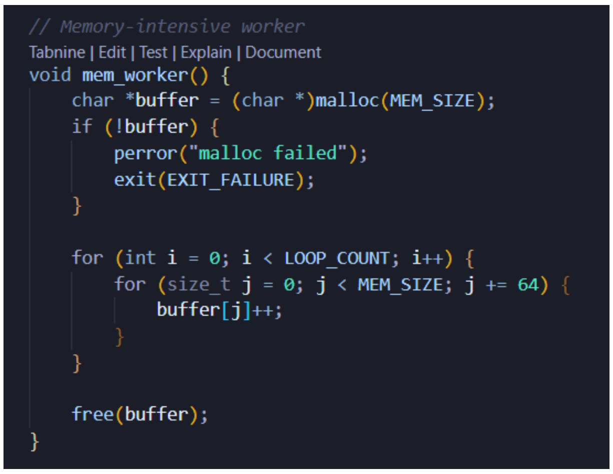
The CPU-intensive worker is designed to spend the majority of its execution time performing computational operations. It repeatedly executes floating-point mathematical functions such as square root, sine, and cosine inside nested loops.

This workload primarily stresses the processor's arithmetic and execution units while generating minimal memory and disk activity. As a result, high CPU utilization is expected, with negligible disk I/O and limited memory footprint.

4.3 Memory-Intensive Worker (`mem`)

The memory-intensive worker focuses on stressing the system's main memory subsystem. It dynamically allocates a large contiguous block of memory and repeatedly writes across the entire region for each iteration of the loop.

This access pattern causes frequent cache misses and forces data movement between the CPU and main memory. The workload is therefore bottlenecked by memory bandwidth rather than

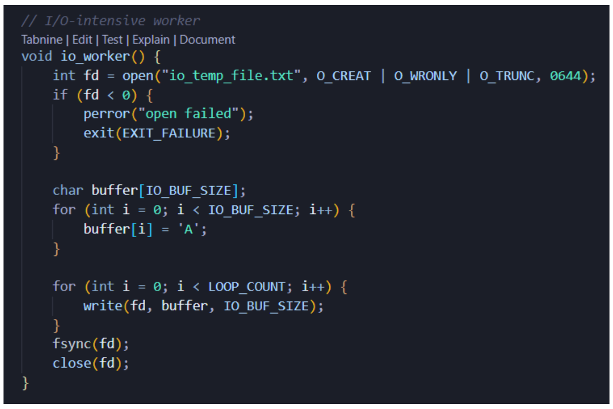


```c
// Memory-intensive worker
Tabnine | Edit | Test | Explain | Document
void mem_worker() {
    char *buffer = (char *)malloc(MEM_SIZE);
    if (!buffer) {
        perror("malloc failed");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < LOOP_COUNT; i++) {
        for (size_t j = 0; j < MEM_SIZE; j += 64) {
            buffer[j]++;
        }
    }

    free(buffer);
}
```

Figure 2: Memory-Intensive Worker Function (`mem`)

computation, leading to high memory usage and moderate CPU utilization.

4.4 I/O-Intensive Worker (`io`)



```c
// I/O-intensive worker
Tabnine | Edit | Test | Explain | Document
void io_worker() {
    int fd = open("io_temp_file.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open failed");
        exit(EXIT_FAILURE);
    }

    char buffer[IO_BUF_SIZE];
    for (int i = 0; i < IO_BUF_SIZE; i++) {
        buffer[i] = 'A';
    }

    for (int i = 0; i < LOOP_COUNT; i++) {
        write(fd, buffer, IO_BUF_SIZE);
    }
    fsync(fd);
    close(fd);
}
```

Figure 3: I/O-Intensive Worker Function (`io`)

The I/O-intensive worker is designed to spend most of its execution time waiting for disk operations to complete. It repeatedly writes data to a file on disk inside a loop, thereby generating sustained disk write activity.

Since the CPU frequently waits for I/O operations to finish, overall CPU utilization is expected to be low, while disk activity is significantly higher compared to the CPU and memory workers. This worker highlights the impact of I/O latency on program performance.

5 Experimental Methodology and Measurements (Part C)

This section describes the methodology used to measure CPU utilization, memory usage, disk I/O activity, and execution time for all six program–worker combinations: A+cpu, A+mem, A+io, B+cpu, B+mem, and B+io. All measurements were automated using a Bash script, as required by the assignment.

To reduce scheduling variability and ensure reproducible measurements, all experiments were executed with CPU affinity enforced using the `taskset` utility. Both Program A (process-based) and Program B (thread-based) were pinned to a fixed set of CPU cores during execution.

An example command used in the automation script is shown below:

```
taskset -c 0-3 ./programA 2 cpu
```

This ensures that all child processes or threads execute only on the specified CPU cores.

The number of processes or threads was fixed to two for Part C, as required, and the loop count inside each worker function was kept constant across all experiments.

5.1 Measured Metrics

For each program-worker combination, the following performance metrics were collected:

- **Average CPU Utilization (%):** Indicates the computational intensity of the workload.
- **Maximum Memory Usage (RSS in KB):** Captures peak memory consumption during execution.
- **Execution Time (Wall-clock):** Represents total program runtime, including computation, memory access, and I/O waits.

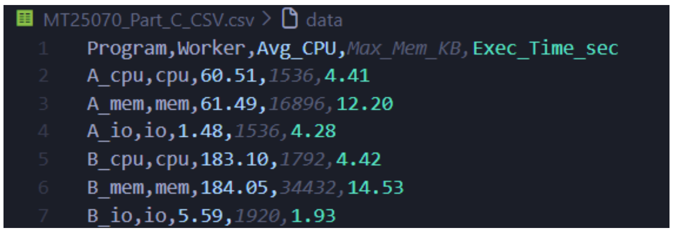
These metrics collectively provide a comprehensive view of how CPU-bound, memory-bound, and I/O-bound workloads interact with system resources.

5.2 Automation and Data Collection

A Bash script was developed to automate the execution of all six program–worker combinations and to ensure consistent data collection. The script performs the following steps:

- Executes Program A and Program B with each worker function.
- Applies CPU pinning using `taskset`.
- Runs `iostat` concurrently to capture disk statistics.
- Extracts CPU usage, memory usage and execution time.
- Stores the collected measurements in a CSV file.

All measurements for Part C are stored in the file `MT25070_Part_C_CSV.csv`. This CSV file serves as the raw data source for analysis and visualization in subsequent parts of the assignment.



```
Program,Worker,Avg_CPU,Max_Mem_KB,Exec_Time_sec
A_cpu,cpu,60.51,1536,4.41
A_mem,mem,61.49,16896,12.20
A_io,io,1.48,1536,4.28
B_cpu,cpu,183.10,1792,4.42
B_mem,mem,184.05,34432,14.53
B_io,io,5.59,1920,1.93
```

Figure 4: Sample CSV Output Generated for Part C

6 Scaling Experiments and Analysis (Part D)

In Part D, the scalability of Program A (process-based) and Program B (thread-based) was evaluated by increasing the number of processes and threads, respectively. Program A was executed with 2–5 processes, while Program B was executed with 2–8 threads, as specified in the assignment.

All measurements were automated using a Bash script, and the collected data was stored in the file `MT25070_Part_D_CSV.csv`.

6.1 Plot Generation

Plots were generated using Python and the `matplotlib` library. The CSV file generated in Part D was parsed, and line plots were created to visualize the effect of increasing parallelism. For all plots, the x-axis represents the number of processes or threads, and the y-axis represents the measured metric.

6.2 CPU Utilization Scaling

The following plots show CPU utilization trends for CPU-, memory-, and I/O-intensive workloads as the number of processes or threads increases.

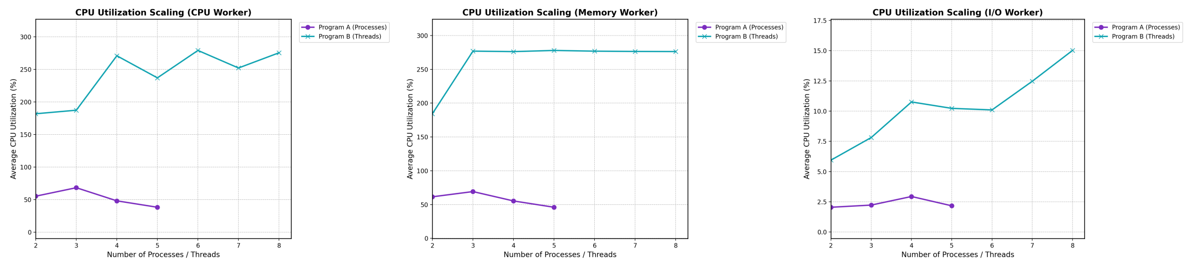


Figure 5: CPU Utilization Scaling for CPU, Memory, and I/O Workers

CPU-intensive workloads exhibit higher CPU utilization with increased parallelism, while memory- and I/O-intensive workloads show limited CPU scaling.

6.3 Memory Usage Scaling

The following plots illustrate memory usage as the number of processes and threads increases.

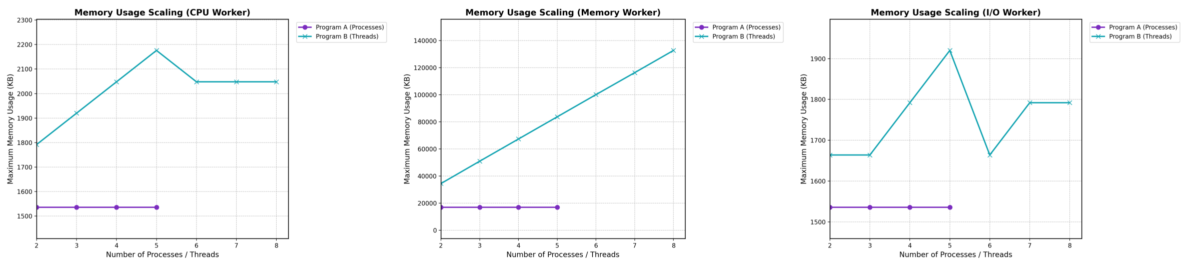


Figure 6: Memory Usage Scaling for CPU, Memory, and I/O Workers

Process-based execution shows higher memory usage due to separate address spaces, whereas thread-based execution benefits from shared memory.

6.4 Execution Time Scaling

The following plots show execution time trends for all three worker functions.

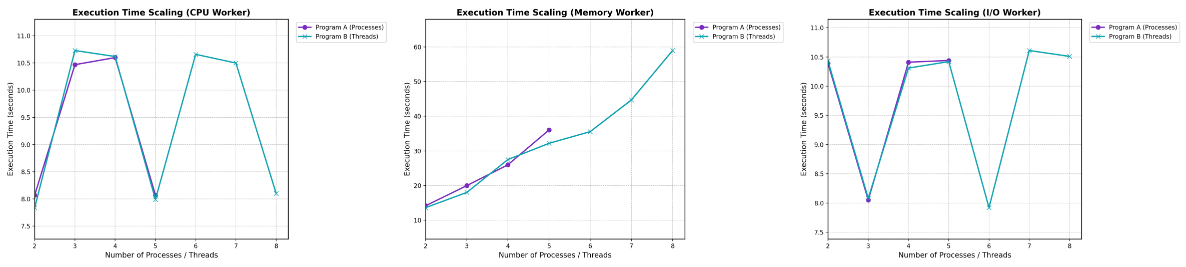


Figure 7: Execution Time Scaling for CPU, Memory, and I/O Workers

CPU-intensive workloads benefit most from increased parallelism, while memory- and I/O-intensive workloads show diminishing returns due to memory bandwidth and disk bottlenecks.

6.5 Overall Observation

Thread-based execution generally scales better than process-based execution due to lower overhead and shared address space. I/O-intensive workloads show limited scalability as performance is constrained by the storage subsystem.

7 Conclusion

This assignment explored process-based and thread-based parallelism using CPU-, memory-, and I/O-intensive workloads. Programs using `fork()` and `pthread` were implemented to study the behavioral differences between processes and threads under varying system loads. Experimental results show that thread-based execution generally scales better for CPU-bound workloads due to lower overhead and shared address space. Process-based execution incurs higher memory usage because of isolated address spaces. Memory- and I/O-intensive workloads exhibit limited scalability, as performance is constrained by memory bandwidth and disk I/O. Overall, the experiments highlight how workload characteristics and execution models significantly influence system performance.

8 AI Usage Declaration

AI-based tools were used in a limited and supportive manner during the completion of this assignment. Specifically:

- Assistance was taken to generate Python scripts for plotting graphs using `matplotlib`.
- AI tools were used to help draft and refine parts of the Bash scripts for automating experiment execution and data collection.
- AI assistance was used for structuring the `README` file and improving documentation clarity.

All core program logic, experimental design, and result analysis were understood and verified by the author.

9 GitHub Repository

https://github.com/JayM2510/GRS_PA01-Assignment-