Indraprastha Institute of Information Technology, Delhi (IIIT-Delhi)

Graduate Systems (CSE638)

Programming Assignment – PA01

Processes and Threads

Semester: Winter 2026

Title:

Performance Analysis of Processes and Threads under CPU, Memory, and I/O Intensive Workloads

Submitted by:
Name: *Nidhi Jha*
Roll Number: MT25031
Program: M.Tech (CSE)

Instructor: Dr. Rinku shah

Date of Submission:
23 January 2026

1. Part A: Process and Thread Creation

1.1 Program A: Process-based Implementation

Program A demonstrates concurrent execution using multiple processes.

The parent process creates child processes inside a loop using the `fork()` system call. Each successful call to `fork()` creates a new child process. The parent process does not perform any computation and only waits for all child processes to terminate using `wait()`. Each child process executes a selected worker function (`cpu`, `mem`, or `io`) independently.

Key properties of Program A:

- Each child process has a **unique Process ID (PID)**.
- Each process has its **own address space**.
- Memory is not shared across processes.
- Process creation incurs higher overhead compared to threads.
-

1.2 Program B: Thread-based Implementation

Program B demonstrates concurrent execution using threads.

The main thread creates worker threads using the POSIX threads library (`pthread`). Each thread executes the same worker function. All threads run within the same process and therefore share the same address space.

Key properties of Program B:

- All threads share the **same PID**.
- Threads share heap memory and global variables.
- Threads are lightweight compared to processes.
- Communication between threads is easier due to shared memory.

```
root@LAPTOP-AH8AQ2T2:~/GRS_PA01# gcc MT25031_Part_A_Program_A.c MT25031_Part_B_workers.c -o partA_process -lm
root@LAPTOP-AH8AQ2T2:~/GRS_PA01# gcc MT25031_Part_A_Program_B.c MT25031_Part_B_workers.c -o partA_thread -pthread -lm
root@LAPTOP-AH8AQ2T2:~/GRS_PA01# ./partA_process
Parent PID = 151651
Child process 1 created, PID = 151652
Child process 2 created, PID = 151653
root@LAPTOP-AH8AQ2T2:~/GRS_PA01# ./partA_thread
Main thread PID = 151690
Thread 1 created, PID = 151690
Thread 2 created, PID = 151690
root@LAPTOP-AH8AQ2T2:~/GRS_PA01#
```

Fig 1.1 Creation of two child Processes in Program A and two threads in Program B

1.3 Analysis and Observations

- In Program A, each child process has a distinct PID, confirming separate process creation.
- In Program B, all threads share the same PID, confirming execution within a single process.
- Process-based execution provides isolation, while thread-based execution allows shared memory.

2. Part B: Worker Function Implementation

2.1 Design Overview

All worker functions are implemented in a modular manner using a shared header file (`MT25031_Part_B_workers.h`) and a source file (`MT25031_Part_B_workers.c`). The header file contains the function declarations and the loop count definition, while the source file contains the actual implementations of the worker functions. A helper function `get_worker()` returns a function pointer corresponding to the worker name (`cpu`, `mem`, or `io`) passed as a command-line argument. For roll number **MT25031**, the loop count is defined as **1 × 10³ = 1000 iterations**. This design enables both Program A (process-based) and Program B (thread-based) to execute the same worker functions without code duplication.

2.2 CPU-Intensive Worker (`cpu`)

The CPU-intensive worker is designed to spend most of its execution time performing computations on the CPU without involving input/output operations or memory allocation. It repeatedly performs floating-point arithmetic operations inside a loop, keeping the CPU busy for the duration of execution. As a result, this worker exhibits high CPU utilization, minimal memory usage, and negligible disk activity. The function is declared in `MT25031_Part_B_workers.h` and implemented as `cpu()` in `MT25031_Part_B_workers.c`.

2.3 Memory-Intensive Worker (`mem`)

The memory-intensive worker stresses the system's memory subsystem by allocating a large block of memory on the heap and repeatedly accessing it within a loop. This causes frequent memory accesses and cache misses, resulting in sustained CPU usage and increased memory consumption. Disk I/O activity remains low, as the workload primarily focuses on memory operations. The function is declared in `MT25031_Part_B_workers.h` and implemented as `mem()` in `MT25031_Part_B_workers.c`.

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

The I/O-intensive worker spends most of its execution time waiting for input/output operations to complete. It repeatedly writes data to a file, forces disk writes using `fflush()`, and introduces delays using `usleep()` to simulate I/O wait. Consequently, CPU utilization remains low, while disk activity dominates execution time. Memory usage is minimal, as no large data structures are required. The function is declared in `MT25031_Part_B_workers.h` and implemented as `io()` in `MT25031_Part_B_workers.c`.

2.5 Worker Selection and Integration

The `get_worker()` function maps the worker name provided via command-line arguments to the corresponding worker function using function pointers. This mechanism ensures that the same worker implementation is reused by both processes and threads, maintaining modularity and avoiding duplicate logic. In Program A, each child process executes the selected worker function, while in Program B, each thread executes the same worker function. This satisfies the requirement that all three worker functions can be executed by both process-based and thread-based programs.

3. Part C: Observations and Analysis

CSV results for part C is shown below:

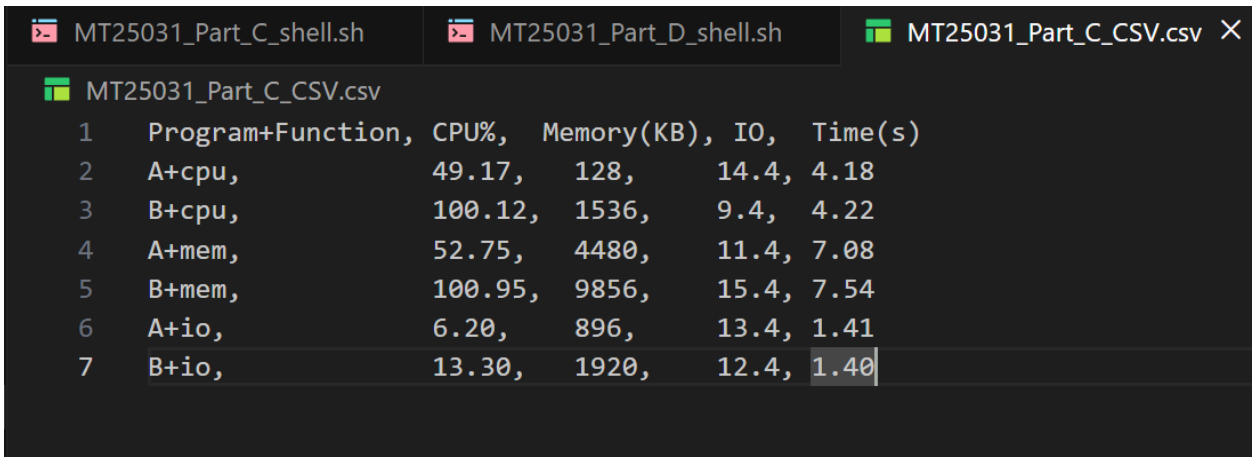

Fig3.1: CSV for part C

3.1 CPU-Intensive Workload (cpu)

For the CPU-intensive workload, the thread-based implementation (Program B) achieves nearly full CPU utilization (≈100%), while the process-based implementation (Program A) utilizes only about half of the available CPU capacity. This trend is expected because threads incur lower context-switching overhead and share the same address space, allowing the scheduler to keep the pinned CPU core fully occupied. In contrast, processes require heavier context switches and maintain separate address spaces, reducing effective CPU utilization. Memory usage remains low in both cases since the workload is purely computational, and disk I/O activity is minimal. Execution time for both programs is comparable, indicating that CPU availability, rather than total work done, dominates performance.

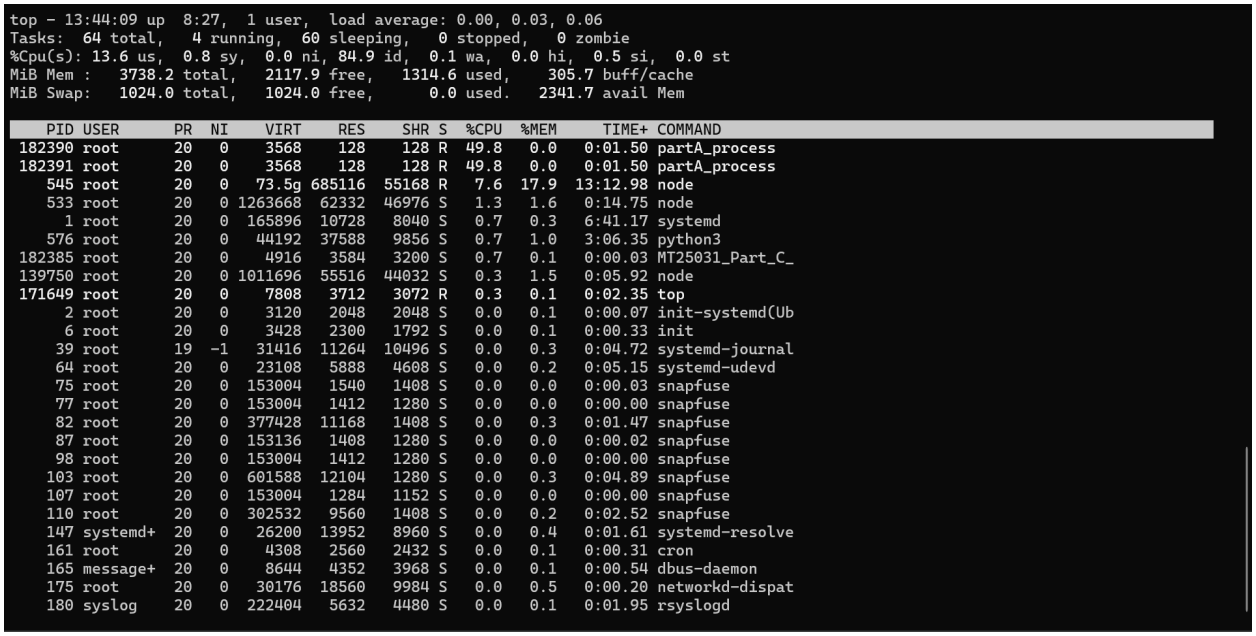

Fig3.2 Result for top command

The `top` snapshot confirms correct execution of the process-based CPU-intensive workload. Two `partA_process` instances are visible, each consuming approximately 50% CPU, indicating fair scheduling on a single core due to CPU pinning using `taskset`. The combined CPU usage approaches 100%, validating proper core isolation. Memory usage remains minimal (~128 KB per process), consistent with a computation-heavy workload without significant memory allocation or I/O activity.

3.2 Memory-Intensive Workload (`mem`)

In the memory-intensive workload, both Program A and Program B exhibit sustained CPU usage due to frequent memory accesses. However, the thread-based implementation again reaches close to 100% CPU utilization, whereas the process-based implementation remains around 50–55%. Memory consumption is significantly higher for threads, as all threads allocate memory within a shared address space, leading to cumulative memory usage. Processes, on the other hand, use less memory overall due to isolated address spaces and independent allocation. Disk I/O remains relatively low, and execution time is higher than the CPU-intensive workload, reflecting the overhead associated with repeated memory access.

3.3 I/O-Intensive Workload (`io`)

CPU utilization is low for both processes and threads, confirming that execution is dominated by disk operations rather than computation. Disk I/O activity is noticeably higher compared to CPU and memory workloads, while memory usage remains moderate and stable. Threads show slightly higher CPU usage than processes, but the difference is marginal, as both implementations spend most of their time waiting for I/O completion. Execution time is the lowest among all workloads, consistent with the I/O-bound nature of the task.

3.4 Overall Trends and Comparison

Across all workloads, thread-based execution consistently demonstrates higher CPU utilization for CPU-bound and memory-bound tasks due to reduced overhead and shared memory. Process-based execution shows lower CPU utilization but offers better memory isolation. For I/O-bound workloads, both processes and threads exhibit similar behavior, as performance is limited primarily by disk I/O rather than CPU or memory efficiency. These results align with the expected theoretical differences between processes and threads in Linux systems.

```
root@LAPTOP-AH8AQ2T2:~/GRS_PA01# time taskset -c 0 ./partA_process cpu

real    0m4.561s
user    0m4.540s
sys     0m0.009s
root@LAPTOP-AH8AQ2T2:~/GRS_PA01# time taskset -c 0 ./partA_thread mem

real    0m8.154s
user    0m8.000s
sys     0m0.150s
```

Fig 3.3 using time command for Program A and Program B.

Conclusion: The CPU-intensive worker in Program A spends nearly all execution time performing computations in user space, with negligible system call overhead, confirming its CPU-bound nature.

Memory-intensive workloads show higher system time because frequent dynamic memory allocation, page faults, and virtual memory management operations require kernel intervention, increasing time spent in system mode.

4. Part D:Scaling Study of Processes and Threads

4.1 Experimental Methodology and Metrics Collection

For Part D, scalability is evaluated by varying the number of worker processes in Program A from 2 to 5 and the number of worker threads in Program B from 2 to 8, excluding the main process and main thread from the worker count. Each configuration is executed separately for the three worker functions: cpu, mem, and io, resulting in nine distinct experiment combinations used to generate nine corresponding graphs.

All executions are pinned to a single CPU core using taskset to ensure fair and consistent comparison between processes and threads. During execution, CPU and memory usage are sampled periodically using top, while disk activity is monitored concurrently using iostat until program termination.

The following metrics are collected for every run:
CPU utilization is computed as the average %CPU observed across all samples.
Memory usage represents the peak resident set size; for processes, this is calculated as the sum of resident memory across all worker processes, while for threads it corresponds to the single shared process memory.
Disk I/O is measured as the aggregate disk activity reported by iostat over the execution period.

All results are logged automatically in CSV format as Program+Worker, Count, CPU, Memory(KB), IO, and are used to analyze and visualize process-versus-thread scaling behavior across CPU-intensive, memory-intensive, and I/O-intensive workloads.

4.2: Results and Graphical Analysis

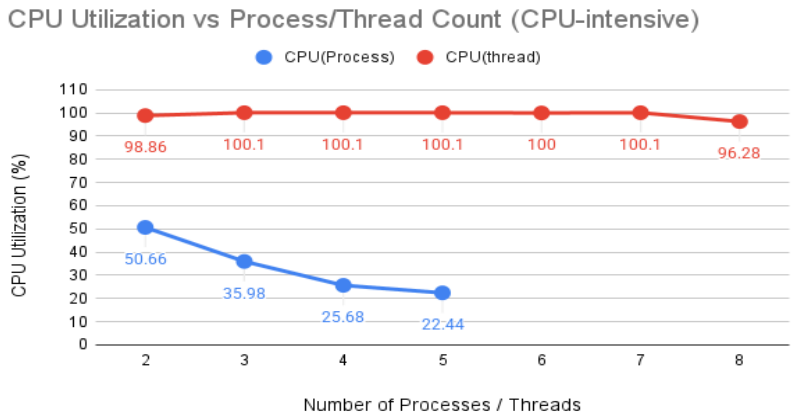

Fig 4.2.1

As the number of processes increases, average CPU utilization per process decreases because multiple processes compete for a single pinned core, increasing context-switch overhead. In contrast, thread-based execution maintains close to 100% CPU utilization across all thread counts. This shows that threads share the same address space and are scheduled more efficiently. Threads therefore utilize the CPU more effectively for CPU-bound workload.

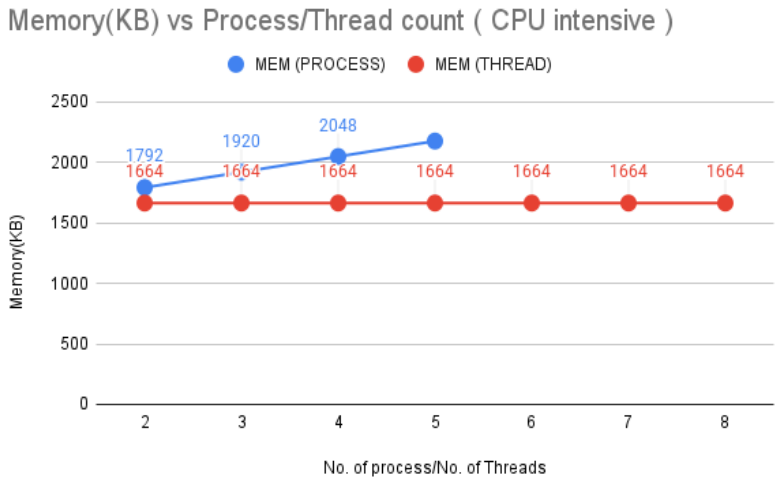

Fig 4.2.2

Memory usage increases gradually with the number of processes because each process maintains its own memory space. Thread-based memory usage remains nearly constant since all threads share the same address space. This demonstrates the lower memory overhead of threads compared to processes. Threads are therefore more memory-efficient for CPU-intensive workloads.

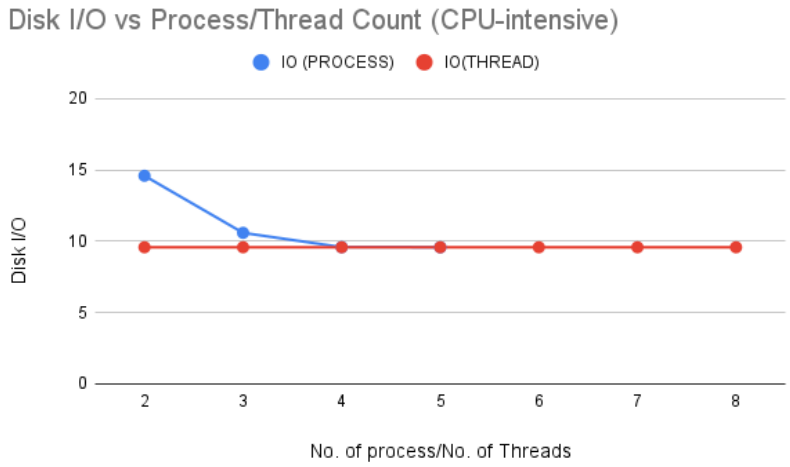

Fig 4.2.3

Disk I/O remains low and nearly constant for both processes and threads, confirming that the workload is CPU-bound. Slight variation in process-based I/O is due to shorter execution slices as process count increases. Thread-based I/O remains stable across counts. Disk activity is not a dominant factor in CPU-intensive execution.

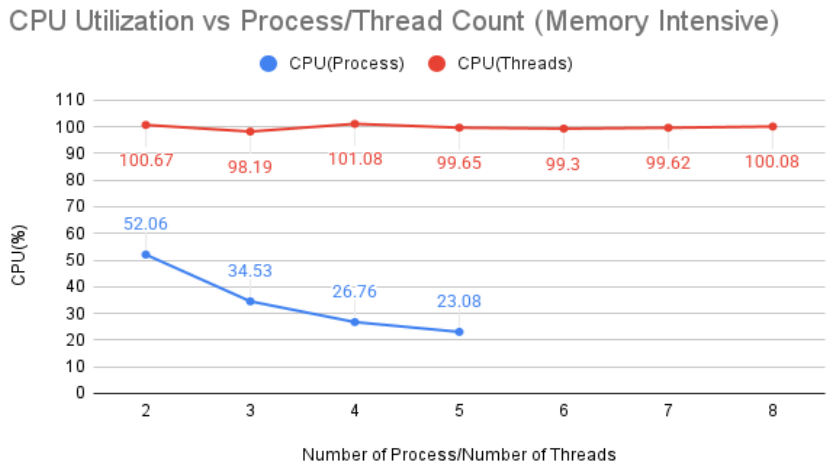

Fig 4.2.4

Thread-based execution sustains close to 100% CPU utilization even under memory-intensive workloads because all threads run within a single process and continuously issue memory accesses without blocking the CPU. While some threads wait for memory operations, others are scheduled immediately, keeping the CPU busy. In contrast, process-based execution incurs higher context-switch and address-space management overhead, which reduces effective CPU utilization as the number of processes increases. This demonstrates that threads hide memory latency more effectively and scale better for memory-intensive workloads.

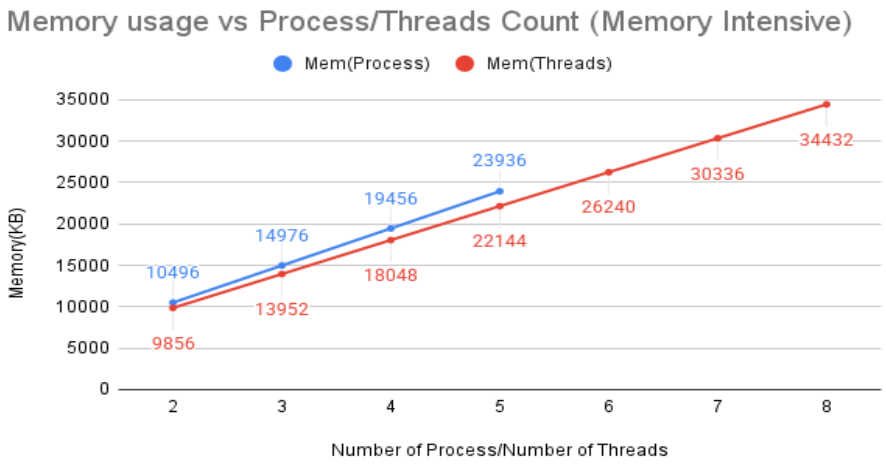

Fig 4.2.5

Memory usage increases almost linearly with both process and thread count, indicating that each worker allocates additional memory. Process-based memory usage is slightly higher due to separate address spaces. Thread-based memory growth is smoother because memory is managed within a single process. This reflects efficient memory sharing in multithreaded execution.

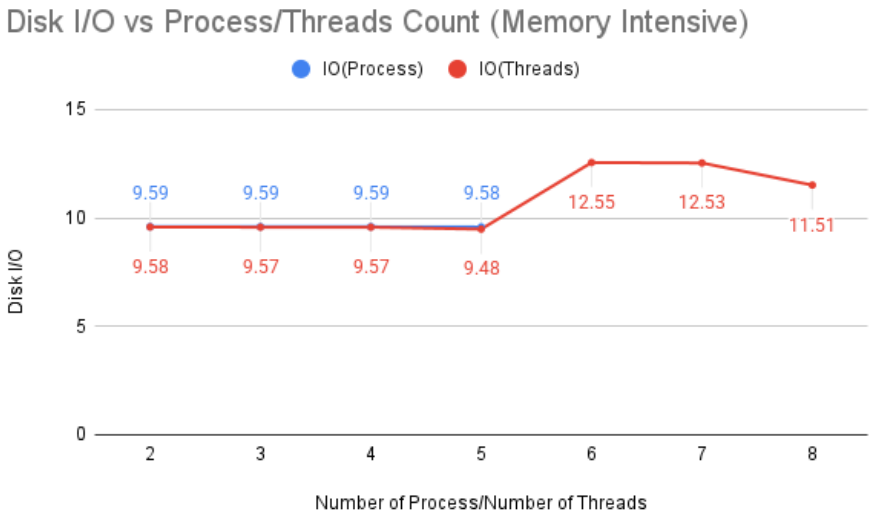

Fig 4.2.6

Disk I/O remains low and mostly constant, confirming that the memory worker stresses RAM rather than disk. Slight increases for threads at higher counts may result from paging or background kernel activity. Overall, the workload remains memory-bound and not I/O-driven. The similarity across counts validates the workload design.

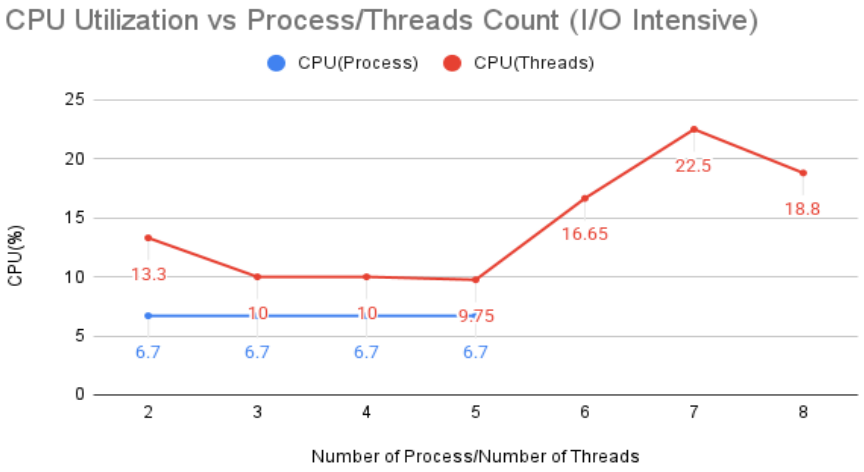

Fig 4.3.7

Process-based execution shows consistently low CPU utilization because each process frequently blocks while waiting for I/O, leaving the CPU idle. Threads show slightly higher CPU usage as the thread count increases because while one thread waits for I/O, another thread can execute, allowing better overlap of I/O wait time. Despite this improvement, CPU utilization remains low overall, confirming that the CPU is not the performance bottleneck for this workload.

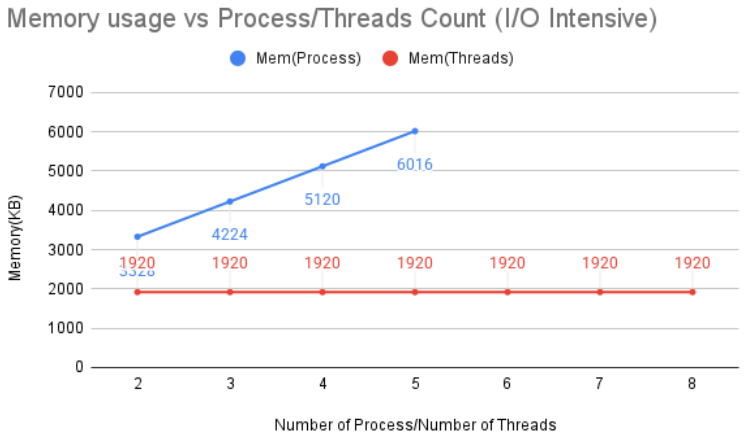

Fig 4.4.8

Memory usage increases with the number of processes because each process maintains its own stack, buffers, and address space, leading to higher cumulative memory consumption. In contrast, thread-based execution shows almost constant memory usage since all threads share the same address space and only require small per-thread stacks. This demonstrates the lower memory overhead of threads, which is particularly beneficial in I/O-heavy applications.

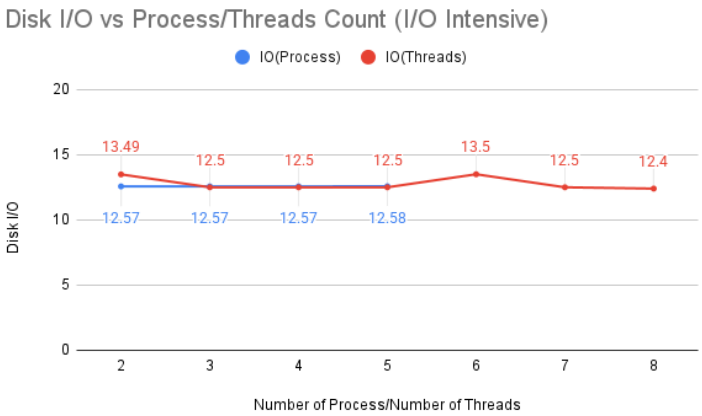

Fig 4.2.9

Disk I/O activity remains high and relatively constant across different process and thread counts, indicating that disk access dominates execution time. Increasing the number of processes or threads does not significantly increase total I/O throughput because the disk itself becomes the limiting resource. Threads show slightly more variation due to concurrent I/O requests being issued, but overall scaling is limited. This confirms that the workload is truly I/O-bound and constrained by storage performance rather than CPU or memory.

b) AI USAGE DECLARATION

An AI-based assistant (ChatGPT) was used as a support tool in the following components of this assignment:

- Assistance in drafting and refining bash automation scripts for Parts C and D.
- Assistance in integrating Program A and Program B with the shared worker functions.
- Refinement and rephrasing of report text, including methodology and analysis sections.

All C source code, shell scripts, and report content were reviewed, modified, tested, and finalized by the author.
All experiments, executions, metric collection, CSV generation, and graph plotting were performed independently by the author.

No AI tool was used to fabricate results or bypass implementation requirements.

c) Repository URL:
https://github.com/Nidhi-Jha03/GRS_Assignment/tree/main/GRS_PA01