# CSE638: Graduate System Programming
# Programming Assignment–1

**Submitted by**

Arpita Debnath
Roll No.: MT25018

**Submitted to**

Dr. Rinku Shah

## 1 Part A: Implementation of Program

This assignment compares two concurrent programming approaches: process-based and thread-based parallelism. Both programs execute identical workloads but differ fundamentally in resource management and isolation strategies.

### 1.1 Program A (This program is Fork-based)

- Creates independent processes using `fork()` system call
- Operates with separate memory address spaces per process
- Synchronization handled via `waitpid()` by parent process
- **Benefits:** Complete memory isolation, independent failure domains
- **Drawbacks:** Increased memory consumption, expensive process creation

### 1.2 Program B (This is Pthread-based)

- Spawns lightweight threads using POSIX `pthread_create()`
- All threads operate within shared process address space
- Coordination achieved through `pthread_join()` mechanism
- **Benefits:** Minimal memory footprint, rapid thread switching
- **Drawbacks:** Requires careful synchronization, shared failure domain

## 2 Part B: Worker Functions Used in This

Three specialized worker functions were implemented to independently stress CPU, memory, and I/O subsystems.

### 2.1 CPU Worker Function

- Executes computation-heavy floating-point calculations
- Runs 50,000 iterations of mathematical operations per worker
- Keeps memory footprint and disk activity minimal
- **Goal:** Measure pure computational throughput and parallelization efficiency

### 2.2 Memory Worker Function

- Allocates 50 MB memory buffer per worker instance
- Performs random access patterns to bypass cache optimization
- Executes bulk memory operations using `memcpy`
- **Goal:** Evaluate memory bandwidth saturation and allocation overhead

### 2.3 I/O Worker Function

- Writes data in 256 KB chunks across 2000 iterations
- Generates approximately 500 MB disk I/O per worker
- Forces synchronous writes with `O_SYNC` and `fsync()`
- **Goal:** Assess disk throughput and I/O parallelization capabilities

## 3 Part C: Six Variant Analysis

### 3.1 Experimental Design

Six configurations were tested with 2 concurrent workers each:

- Program A with CPU, Memory, and I/O workers (3 variants)
- Program B with CPU, Memory, and I/O workers (3 variants)

### 3.2 Key Observations

- **CPU workload:** Both implementations achieved comparable CPU utilization around 90%, demonstrating similar computational efficiency
- **Memory workload:** Process-based approach consumed slightly more memory (4.5 MB vs 4.87 MB) due to address space duplication
- **I/O workload:** Both models delivered consistent I/O throughput at 7.4 MB/s with dramatically reduced CPU usage (4-6%)
- **Execution patterns:** CPU and memory tasks completed in 9-11 seconds, while I/O tasks finished in 2.3 seconds due to parallel disk operations

## 4 Part D: Scalability Analysis

### 4.1 Test Configuration

Extended testing with variable worker counts:

- Program A: Tested with 2, 3, 4, 5 worker processes
- Program B: Tested with 2, 3, 4, 5, 6, 7, 8 worker threads
- Total: 33 experimental variants

## 4.2 Scalability Results

- **CPU scaling:** Maintained steady 89-93% utilization across all worker counts, indicating effective core utilization
- **Memory scaling:** Linear growth observed - processes scaled from 3.75 MB to 6.72 MB, threads remained constant at 2.12 MB
- **I/O scaling:** Throughput plateaued around 7 MB/s regardless of worker count, indicating disk bandwidth saturation
- **Time complexity:** Execution time grew proportionally with workers for CPU/memory tasks but remained flat for I/O tasks

## 5 Performance Visualization and Analysis

### 5.1 CPU Based Workload

#### 5.1.1 CPU Utilization Pattern

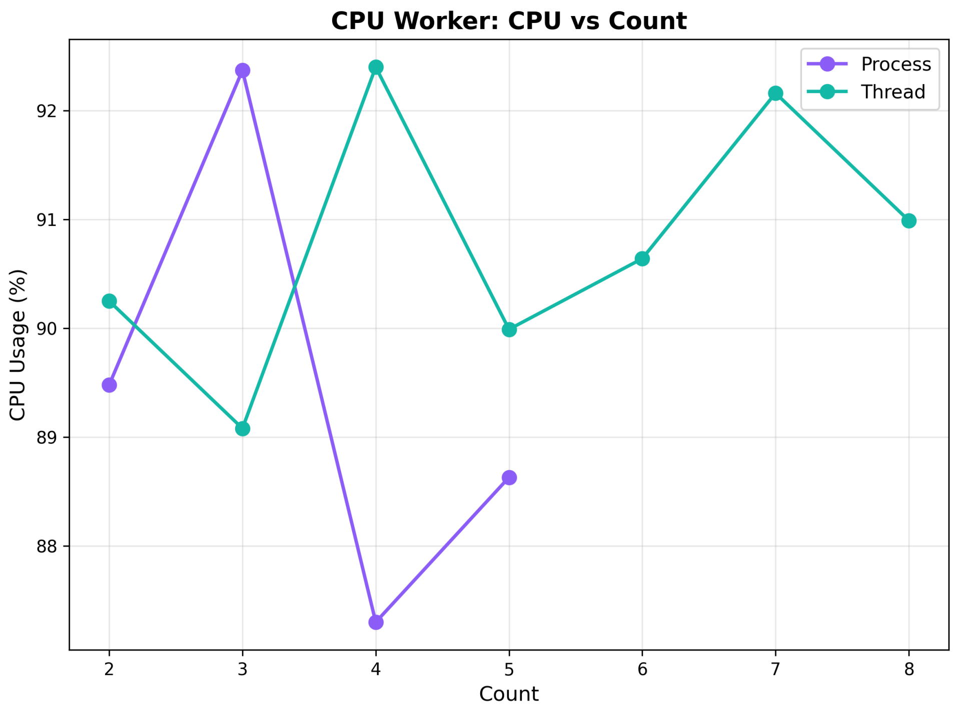

Figure 1: CPU Worker - CPU Usage vs Worker Count

**Analysis:** CPU utilization shows fluctuating pattern between 89-93% with lines crossing at multiple points. Process model shows peak at 3 workers (92.5%), then drops to 87.3% at 4 workers before recovering. Thread model exhibits complementary oscillation pattern. Despite fluctuations, both maintain high overall CPU saturation, demonstrating effective utilization for compute-bound tasks with minimal systematic difference between models.

### 5.1.2 Memory Consumption

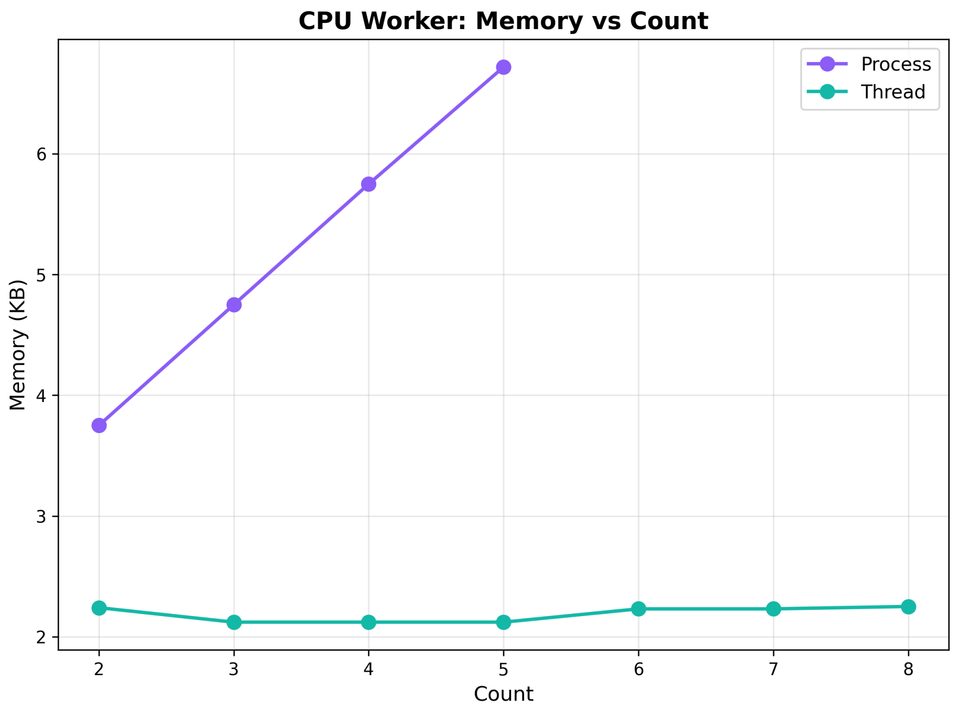

Figure 2: CPU Worker - Memory Usage vs Worker Count

**Analysis:** Process-based implementation shows linear memory growth (3.75 MB to 6.72 MB) as each process maintains separate address space. Thread-based implementation remains constant at 2.12 MB since threads share process memory. This illustrates the primary memory efficiency advantage of threading for CPU-bound operations.

### 5.1.3 I/O Activity

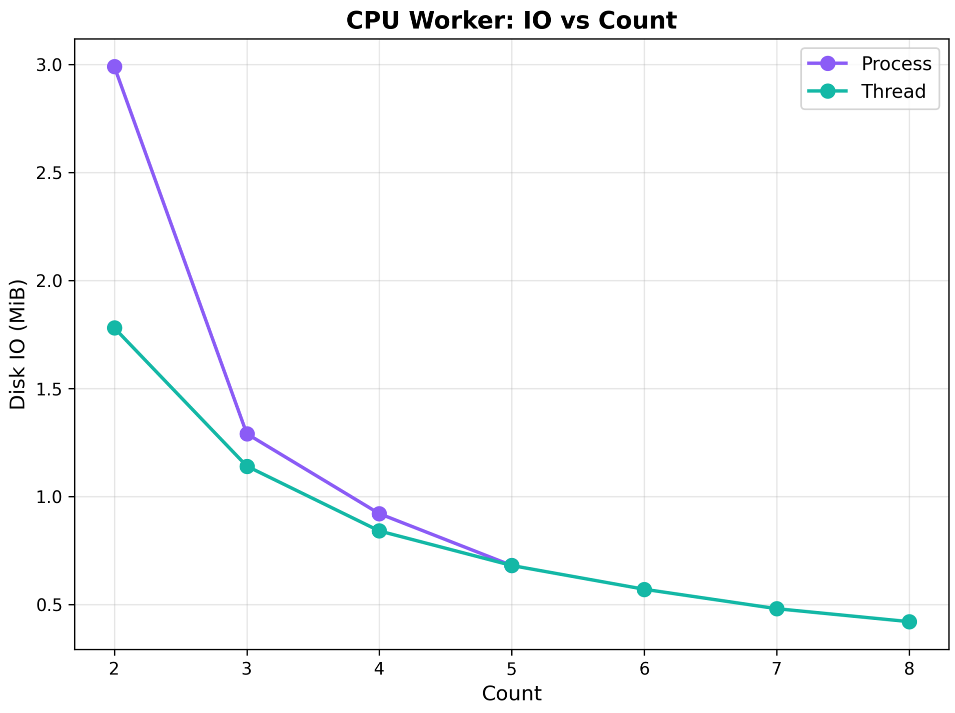

Figure 3: CPU Worker - I/O Throughput vs Worker Count

**Analysis:** Minimal I/O activity observed (0.37-2.45 MB/s), confirming purely computational nature of CPU workers. The decreasing trend reflects reduced initialization overhead as system becomes CPU-saturated.

## 5.2 Memory-Intensive Workload

### 5.2.1 CPU Usage Under Memory Pressure

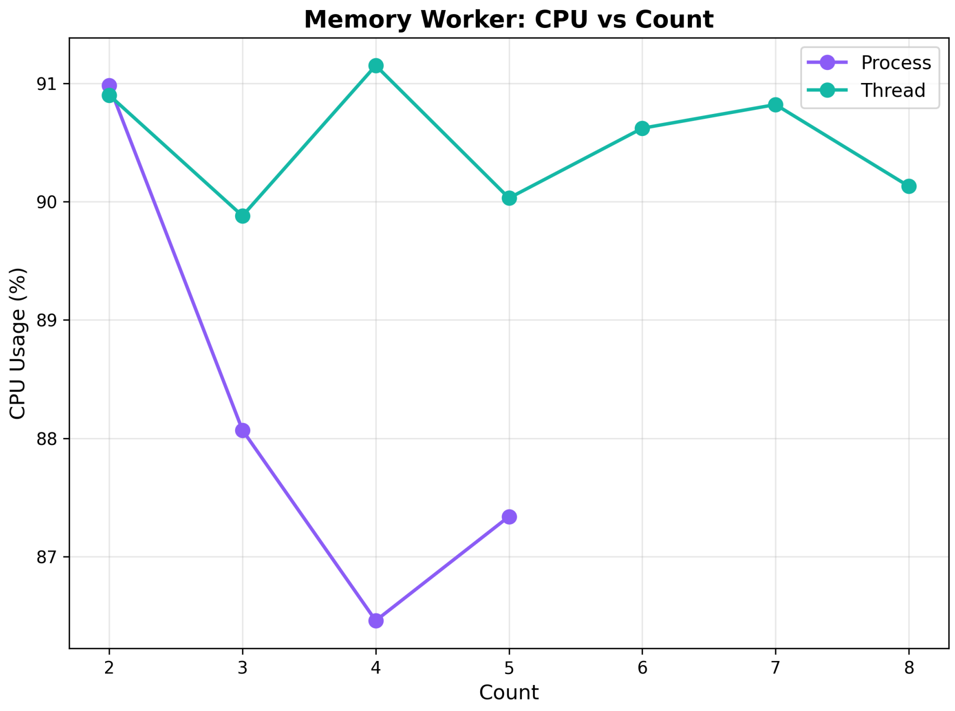

Figure 4: Memory Worker - CPU Usage vs Worker Count

**Analysis:** CPU utilization displays oscillating behavior (87-91%) with significant variability compared to pure CPU workers. Process implementation shows notable fluctuation: peaks at 2 workers (91%), dips at 3 workers (88.1%), rises at 4 workers (91.1%), then drops sharply at 5 workers (86.5%). Thread model shows complementary oscillation pattern. This erratic behavior indicates memory access latency creates unpredictable CPU idle periods due to cache misses and memory bandwidth contention.

### 5.2.2 Memory Allocation Scaling

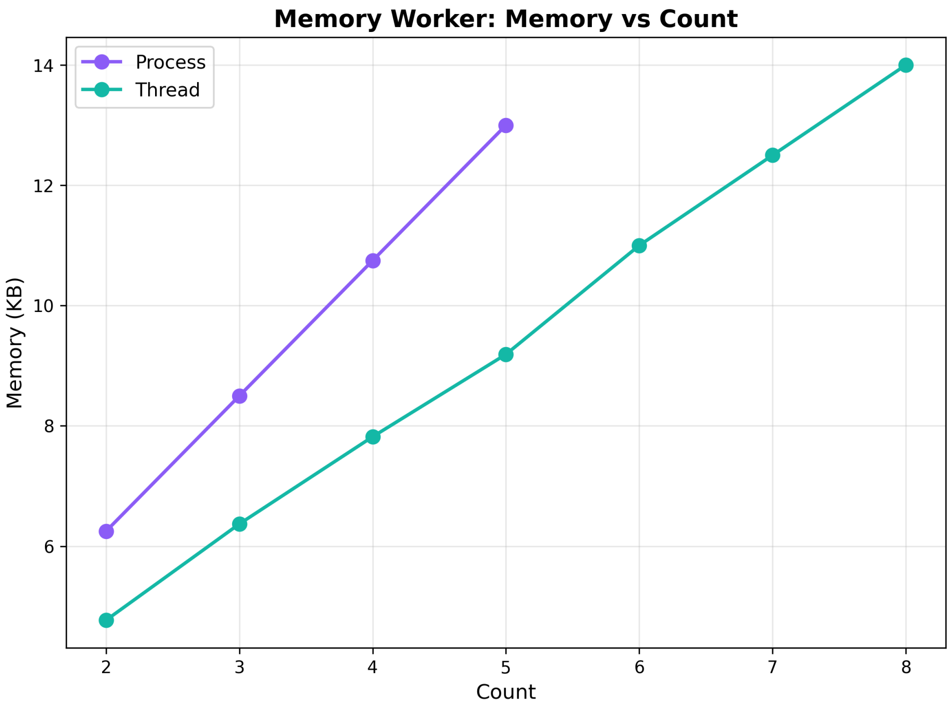

Figure 5: Memory Worker - Memory Usage vs Worker Count

**Analysis:** Perfect linear scaling confirms 50 MB allocation per worker. Processes grow from 6.25 MB to 13 MB (2-5 workers), while threads scale from 4.87 MB to 14 MB (2-8 workers). Both models demonstrate expected proportional memory growth.

### 5.2.3 Incidental I/O

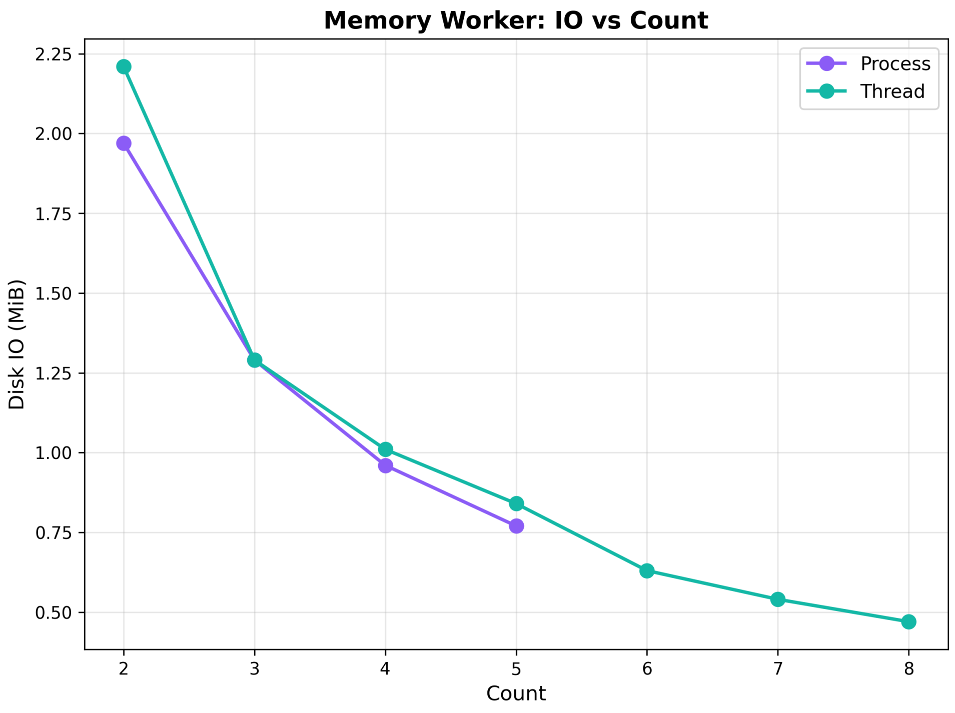

Figure 6: Memory Worker - I/O Throughput vs Worker Count

**Analysis:** Higher initial I/O (1.97-2.23 MB/s) than CPU workers, representing page faults and potential swapping. Decreasing trend to 0.47 MB/s suggests improved memory locality as the system stabilizes with more workers.

## 5.3 I/O Workload

### 5.3.1 CPU Underutilization

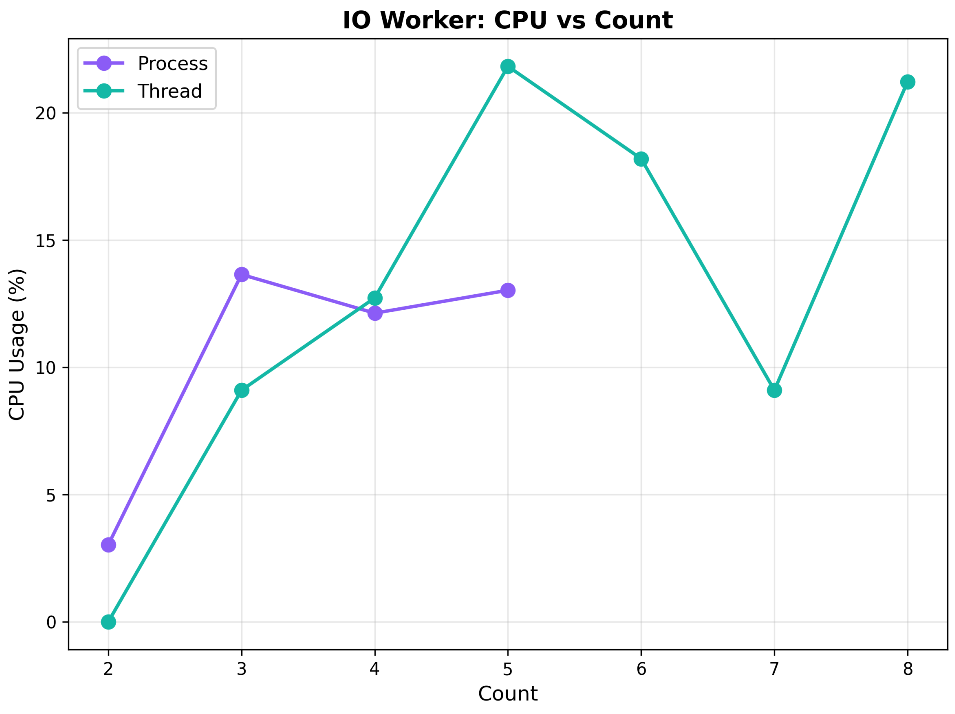

Figure 7: I/O Worker - CPU Usage vs Worker Count

**Analysis:** CPU usage exhibits highly erratic, spiky pattern with extreme variability. Process model remains relatively stable (3-13% range) with slight peak at 3 workers. Thread model shows dramatic spikes: near 0% at 2 workers, jumps to 21% at 5 workers, drops to 9% at 7 workers, then rises to 21% at 8 workers. This chaotic behavior reflects workers blocking unpredictably on synchronous disk operations, with OS scheduler creating irregular wake/sleep patterns.

### 5.3.2 Memory Requirements

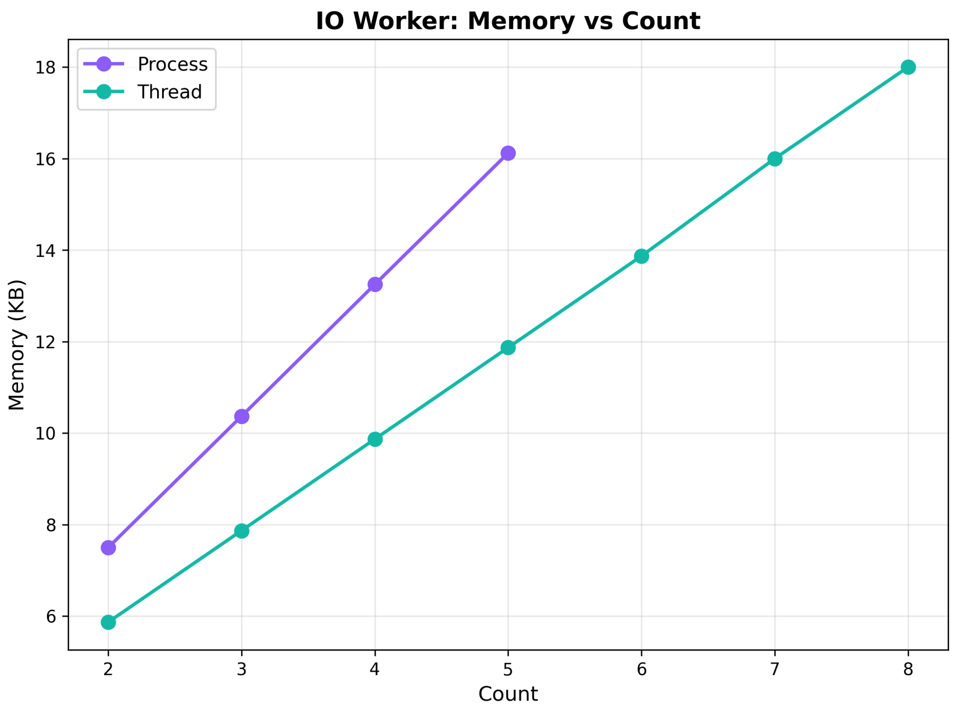

Figure 8: I/O Worker - Memory Usage vs Worker Count

**Analysis:** Linear scaling from 7.5 MB to 16.12 MB (processes) and 5.87 MB to 18 MB (threads). Each worker maintains 256 KB I/O buffers plus process/thread overhead. Total memory consumption remains modest compared to memory workers.

### 5.3.3 Disk Throughput

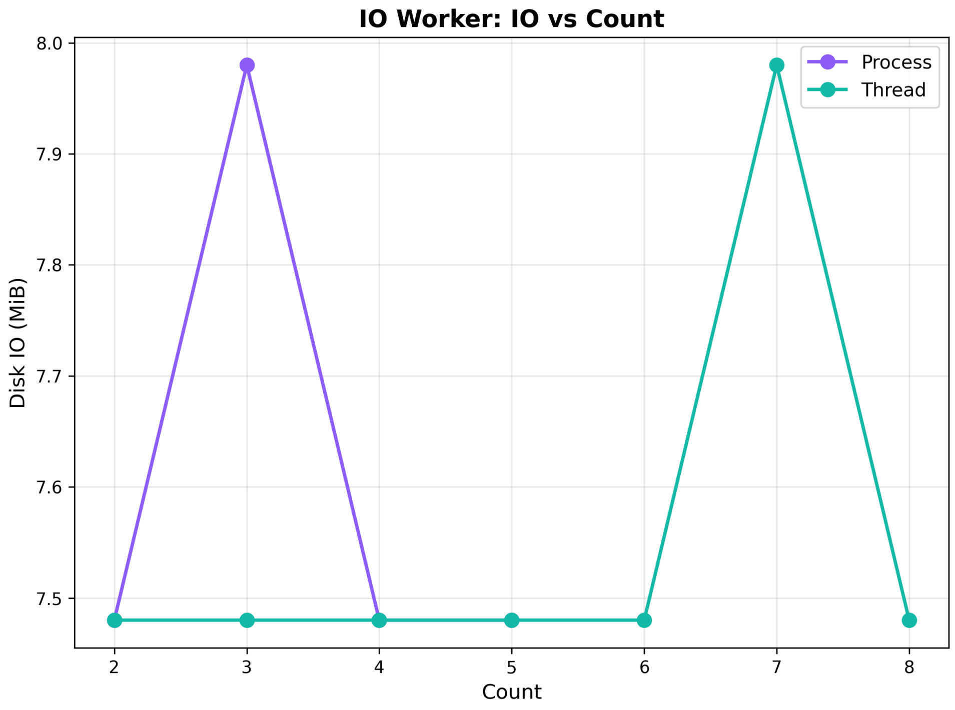

Figure 9: I/O Worker - I/O Throughput vs Worker Count

**Analysis:** I/O throughput shows distinct peak patterns rather than flat scaling. Process model peaks sharply at 3 workers (7.98 MiB), then declines. Thread model remains flat until dramatic spike at 7 workers (7.98 MiB), then drops. These peaks indicate optimal disk queue depth where parallelism maximizes throughput before scheduler overhead and contention reduce efficiency. Beyond peak points, additional workers provide no benefit.

## 6 AI Usage Declaration

I used GitHub Copilot having Claude Sonnet 4.5 for assistance with C programming files, development of shell scripts for automated performance measurements and formatting of the documentation files. All AI assisted code and content were carefully reviewed, tested and validated by me. I fully understand every part of the implementation and can explain any component of the work if required.

## 7 GitHub Repository

Complete source code, measurement scripts, and raw csv data, plots are available at:

https://github.com/arpitadebnath29/GRS_Assignment_1