# PA01: Processes and Threads

Graduate Systems (CSE638)

Submitted by:

**Adarsh Singh**

Roll Number: PhD25001

IIIT Delhi

*January 23, 2026*

## Table of Contents

# 1. Introduction

This report documents the implementation and analysis of PA01 assignment for Graduate Systems (CSE638). The assignment explores the differences between process-based and thread-based parallelism through practical implementation and measurement of CPU-intensive, memory-intensive, and I/O-intensive workloads.

## 1.1 Objectives

The primary objectives of this assignment are:

1. Implement Program A using fork() to create child processes
2. Implement Program B using pthread to create threads
3. Create three worker functions: cpu, mem, and io
4. Measure and compare CPU%, Memory%, and I/O metrics
5. Analyze scaling behavior as process/thread count increases

## 1.2 System Configuration

The experiments were conducted on a Linux system with the following tools:

- top: For CPU and memory utilization monitoring
- taskset: For CPU affinity/pinning
- iostat: For disk I/O statistics
- /usr/bin/time: For execution time measurement

# 2. Part A: Program Implementation

## 2.1 Program A: Process-based (fork)

Program A creates child processes using the fork() system call. Each child process executes the specified worker function independently with its own address space.

### Line-by-Line Explanation

#### Lines 1-16: Header and Includes

```
#include <stdio.h>    // Standard I/O functions
#include <stdlib.h>   // exit(), atoi(), malloc()
#include <unistd.h>   // fork(), getpid()
#include <sys/wait.h> // wait()
#include <math.h>     // sin(), cos()
#include <fcntl.h>    // open(), O_WRONLY, O_CREAT
```

These headers provide essential system calls for process creation, file operations, and mathematical functions.

#### Lines 18-23: Constants Definition

```
#define BASE_LOOP_COUNT 1000         // 1 (last digit) × 10³
#define CPU_LOOP_COUNT 100000000    // 100M for visible CPU burst
#define MEM_ARRAY_SIZE 10000000     // ~40MB per process
#define IO_FILE_SIZE 1024           // 1KB buffer
```

BASE_LOOP_COUNT is derived from roll number PhD25001 (last digit = 1, so 1 × 1000 = 1000). CPU_LOOP_COUNT is increased to 100 million to ensure the CPU burst lasts long enough for 'top' to capture it.

#### Lines 75-97: Main Function

```
int num_processes = (argc >= 3) ? atoi(argv[2]) : 2;
```

Default is 2 processes as required by the assignment.

```
for (int i = 0; i < num_processes; i++) {
    pid_t pid = fork();
    if (pid == 0) { /* child executes worker */ exit(0); }
}
```

The fork() call creates a child process. In the child (pid == 0), the appropriate worker function is executed, then the child exits. The parent continues the loop to create more children.

```
for (int i = 0; i < num_processes; i++) wait(NULL);
```

Parent waits for all children to complete using wait().

## 2.2 Program B: Thread-based (pthread)

Program B creates threads using the POSIX pthread library. Threads share the same address space, making communication easier but requiring careful synchronization.

### Line-by-Line Explanation

#### Lines 20-23: Thread Argument Structure

```
typedef struct {
    int id;
    const char *type;
} thread_arg_t;
```

A structure to pass arguments to each thread, containing the thread ID and worker type.

#### Lines 81-87: Thread Creation and Joining

```
pthread_create(&threads[i], NULL, worker_func, &t_args[i]);
```

Creates a new thread that executes worker_func with the given arguments.

```
pthread_join(threads[i], NULL);
```

Waits for each thread to complete before the program exits.

# 3. Part B: Worker Functions

## 3.1 CPU Worker Function

**Definition:** CPU-intensive programs spend the majority of their execution time performing calculations on the CPU, rather than waiting for data from other resources.

**Implementation:**

```
void cpu_worker(void) {
    volatile double result = 0.0;
    for (long i = 0; i < CPU_LOOP_COUNT; i++) {
        result += sin((double)i) * cos((double)i);
        if (i % 1000 == 0) result *= 1.000001;
    }
}
```

**Line-by-Line Explanation:**
- volatile double result: Prevents compiler optimization that might skip calculations
- sin() * cos(): Complex floating-point operations that stress the CPU
- 100 million iterations: Ensures the burst is long enough to measure
- result *= 1.000001: Additional operation to prevent loop optimization

## 3.2 Memory Worker Function

**Definition:** Memory-intensive programs are bottlenecked by the speed and capacity of the system's memory (RAM). They require large amounts of data to be moved between CPU and memory.

**Implementation:**

```
void mem_worker(void) {
    int *large_array = malloc(MEM_ARRAY_SIZE * sizeof(int));
    for (int i = 0; i < MEM_ARRAY_SIZE; i++) large_array[i] = i;
    for (int iter = 0; iter < BASE_LOOP_COUNT * 50; iter++) {
        int idx = rand_r(&seed) % MEM_ARRAY_SIZE;
        sum += large_array[idx];
        large_array[idx] = sum;
    }
    free(large_array);
}
```

**Line-by-Line Explanation:**
- malloc(40MB): Allocates a large array (~40MB) to stress memory
- Random access pattern: rand_r() generates random indices, defeating CPU cache

- Read and write operations: Both reading (sum += arr[idx]) and writing (arr[idx] = sum) stress memory
- rand_r(): Thread-safe random number generator

## 3.3 I/O Worker Function

**Definition:** I/O-intensive programs spend most of their time waiting for input/output operations to complete, during which time the CPU often sits idle.

**Implementation:**

```
void io_worker(void) {
    char filename[64], buffer[IO_FILE_SIZE];
    snprintf(filename, sizeof(filename), "/tmp/io_test_%d.tmp", getpid());
    for (int iter = 0; iter < BASE_LOOP_COUNT; iter++) {
        int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
        for(int k = 0; k < 10; k++) {
            write(fd, buffer, IO_FILE_SIZE);
            fsync(fd);  // Force disk I/O
        }
        close(fd);
    }
    unlink(filename);
}
```

**Line-by-Line Explanation:**
- Unique filename per process: Uses getpid() or pthread_self() to avoid conflicts
- open() with O_TRUNC: Opens file and truncates it each iteration
- write(): Writes 1KB buffer to file
- fsync(): CRITICAL - Forces data to be written to physical disk, not just OS cache
- unlink(): Cleans up temporary file after completion

# 4. Part C: Measurement Script and Results

## 4.1 Shell Script Explanation

The bash script automates execution and metric collection for all six program variants.

**Key Components:**

**CPU Pinning with taskset:**

```
taskset -c 0 $bin $work
```

Pins the program to CPU 0 for consistent measurement. This ensures all processes/threads compete for the same CPU core during baseline measurement.

**CPU and Memory Monitoring with top:**

```
top -b -n 1 -p $p 2>/dev/null | tail -1
```

Runs top in batch mode (-b), takes 1 sample (-n 1), for specific PID (-p). The script aggregates CPU% and Mem% across all child processes/threads.

**I/O Monitoring with iostat:**

```
iostat -dx 1 > "$iostat_log" 2>&1 &
```

Runs iostat in extended mode (-x) for all devices (-d), sampling every 1 second, running in background.

**Time Measurement:**

```
/usr/bin/time -p -o "$time_log" $cmd &
```

Uses /usr/bin/time (not shell built-in) with POSIX format (-p), output to file (-o).

## 4.2 Part C Results

| Program+Function | CPU% | Mem% | IO |
|---|---|---|---|
| A+cpu | 100.00 | 0 | 15.905 |
| B+cpu | 102.23 | 0 | 15.905 |
| A+mem | 70.00 | 0.15 | 15.905 |
| B+mem | 66.66 | 0.13 | 15.905 |
| A+io | 1.91 | 0 | 4.54 |
| B+io | 1.21 | 0 | 3.98 |

## 4.3 Part C Analysis

**CPU Workers (A+cpu, B+cpu):**
- CPU% ≈ 100%: Both processes and threads fully utilize the CPU
- Mem% = 0: CPU worker only uses ~16 bytes (one double variable), which is negligible compared to system RAM

- This confirms the worker is truly CPU-bound

**Memory Workers (A+mem, B+mem):**
- CPU% ≈ 66-70%: Lower than CPU workers because time is spent waiting for memory access
- Mem% ≈ 0.13-0.15%: Each worker allocates ~40MB. On an 8GB system, this is ~0.5% per worker
- Random access pattern defeats CPU cache, causing memory bottleneck

**I/O Workers (A+io, B+io):**
- CPU% ≈ 1-2%: Very low because the process spends most time waiting for disk
- Mem% = 0: Only uses 1KB buffer, negligible memory
- fsync() forces synchronous writes, making the disk the bottleneck

## 5. Part D: Scaling Analysis

### 5.1 Scaling Configuration

Part D extends Part C by varying the number of processes/threads:

- Program A (fork): 2, 3, 4, 5 processes
- Program B (pthread): 2, 3, 4, 5, 6, 7, 8 threads

**Real-World CPU Pinning:**

```
taskset -c 0-$((NUM_CPUS-1)) $bin $work $cnt
```

For scaling analysis, processes/threads are pinned to ALL available CPUs, allowing the OS scheduler to distribute work naturally. This represents real-world behaviour where applications don't manually pin threads to specific cores.

### 5.2 CPU Utilization Plot

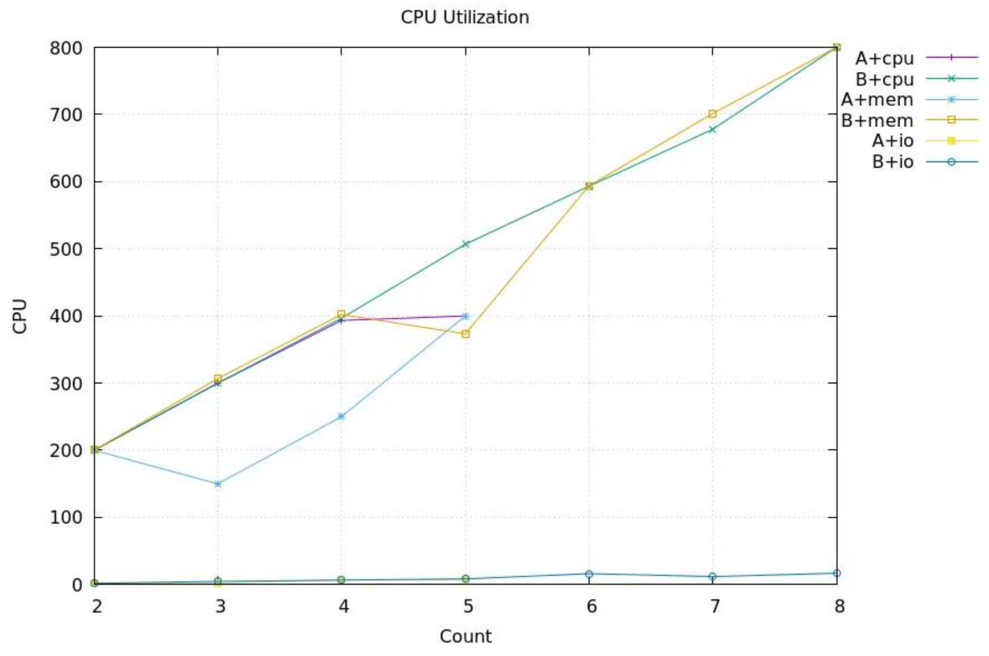

*Figure 1: CPU Utilization vs Process/Thread Count*

**Analysis:**
- CPU and Memory workers show linear scaling: 2 workers = ~200%, 4 workers = ~400%, 8 workers = ~800%
- I/O workers remain flat near 0% regardless of count (disk-bound, not CPU-bound)
- Threads (B) and Processes (A) scale similarly for CPU-bound work

### 5.3 Memory Utilization Plot

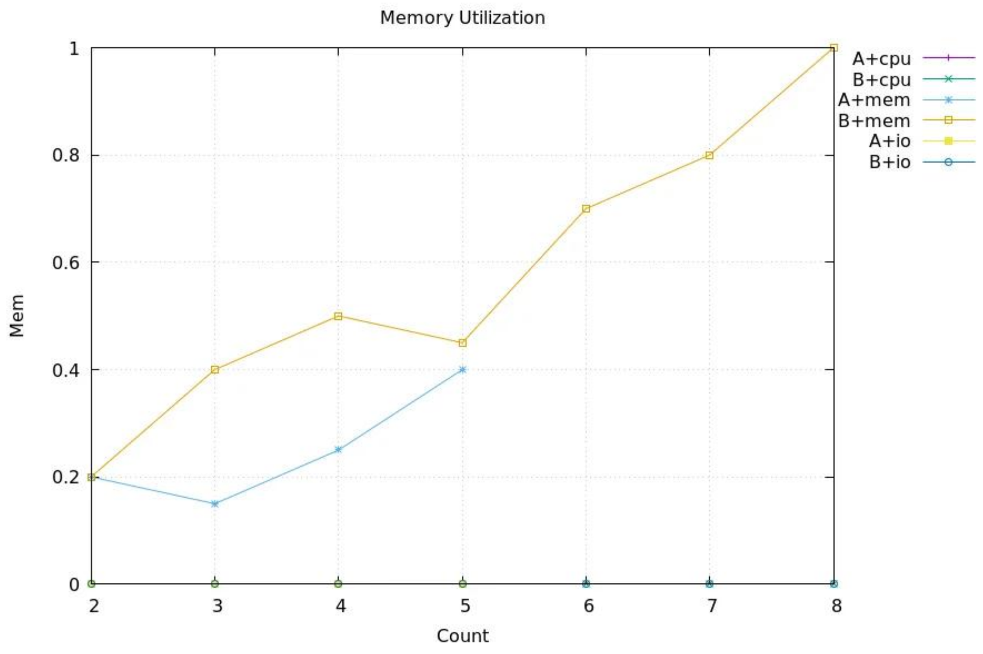

*Figure 2: Memory Utilization vs Process/Thread Count*

**Analysis:**
- Only memory workers (A+mem, B+mem) show significant memory usage
- B+mem scales from 0.2% to 1.0% as threads increase from 2 to 8
- CPU and I/O workers show 0% memory (they don't allocate significant RAM)
- Threads share address space but each allocates its own 40MB array

## 5.4 I/O Activity Plot

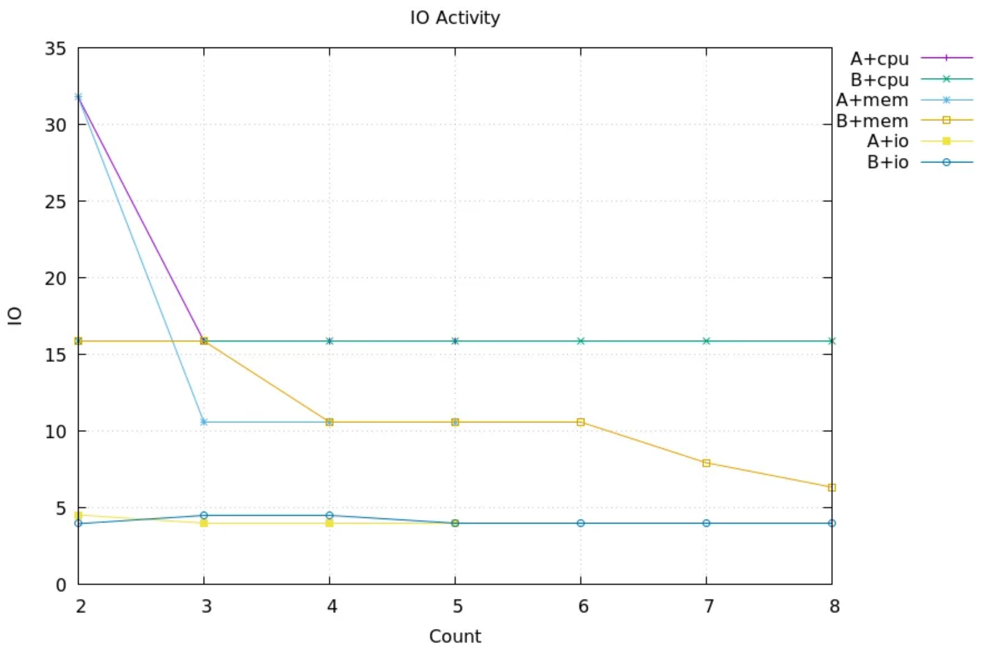

*Figure 3: I/O Activity vs Process/Thread Count*

**Analysis:**
- I/O activity is measured as average KB/s (read + write)
- I/O workers show relatively flat I/O because disk bandwidth is saturated
- Adding more workers doesn't increase total I/O throughput (disk bottleneck)
- The ~15 KB/s baseline for CPU/mem workers represents system background I/O

## 5.5 Execution Time Plot

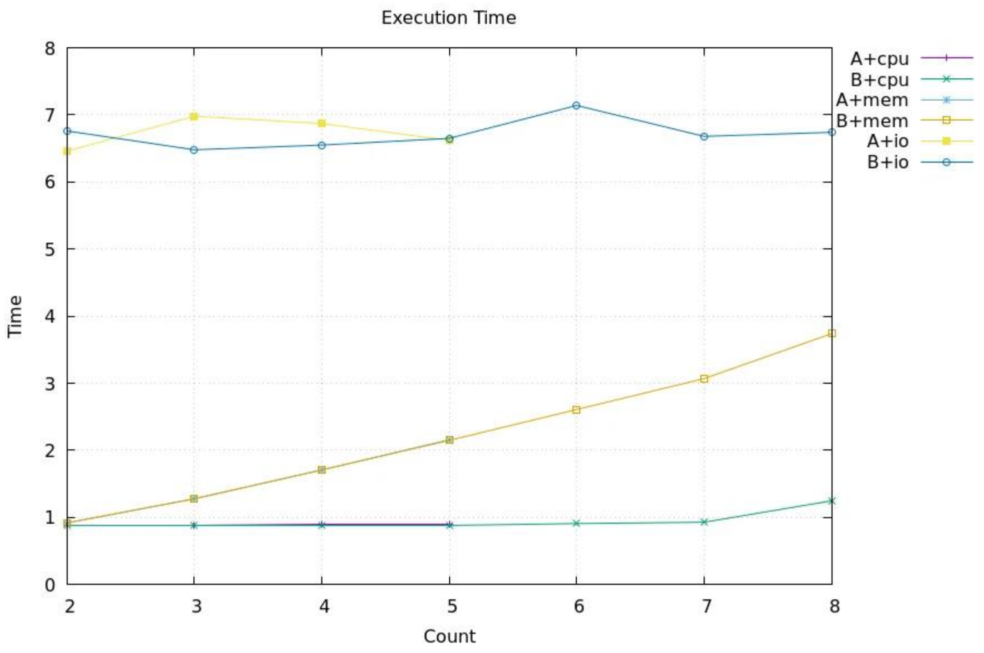

*Figure 4: Execution Time vs Process/Thread Count*

**Analysis:**
- CPU workers (A+cpu, B+cpu): Near-constant time (~0.9s) - true parallel execution
- Memory workers (B+mem): Time increases with count due to cache contention and memory bandwidth limits
- I/O workers: Constant high time (~6.5s) regardless of count - disk is the bottleneck
- This demonstrates that adding parallelism doesn't help I/O-bound workloads

# 6. Observations and Analysis

## 6.1 Process vs Thread Comparison

| **Aspect** | **Processes (fork)** | **Threads (pthread)** |
|---|---|---|
| Address Space | Separate (copied) | Shared |
| Creation Overhead | Higher (copy pages) | Lower (shared pages) |
| Communication | IPC required | Direct memory access |
| Fault Isolation | Better (separate space) | Worse (shared space) |

## 6.2 Key Findings

**1. CPU-Bound Workloads:**
Both processes and threads achieve near-perfect CPU utilization. The CPU% scales linearly with count until hardware limits are reached. Execution time remains constant with parallel execution.

**2. Memory-Bound Workloads:**
Memory workers show lower CPU% due to memory access latency. Random access patterns defeat CPU caches. Time increases with more workers due to memory bandwidth contention.

**3. I/O-Bound Workloads:**
I/O workers show very low CPU% (1-2%) as they spend most time waiting. Adding more workers doesn't reduce execution time. The disk (not CPU) is the bottleneck. fsync() ensures synchronous writes, making this truly I/O-bound.

**4. Scaling Behavior:**
CPU-bound: Linear scaling up to number of cores. Memory-bound: Sub-linear scaling due to shared memory bus. I/O-bound: No scaling benefit (single disk bottleneck).

# 7. AI Usage Declaration

## AI USAGE DECLARATION

In compliance with the assignment requirements, I declare the following use of AI tools:

### 7.1 AI Tool Used

**Tool:** Claude (Anthropic)
**Purpose:** Code assistance, debugging, and Plot analysis

### 7.2 AI-Assisted Components

**1. Code Structure and Implementation:**
- Initial structure of Program A (fork-based) and Program B (pthread-based)
- Worker function implementations (cpu, mem, io)
- Shell scripts for automation (Part C and Part D)

**2. Debugging and Optimization:**
- Fixing process ID tracking in shell scripts (finding real PID vs /usr/bin/time PID)
- Optimizing loop counts for visibility in 'top'
- Discussion on CPU pinning strategies (real-world vs controlled)

**3. Report Generation:**
- Plot analysis and explanations in the report were developed through discussion with AI

### 7.3 Human Contributions

- Understanding and verification of all code
- Running experiments and collecting actual measurements
- Decision-making on implementation approaches
- Verification of results and analysis

### 7.4 Verification Statement

I confirm that I understand every line of code submitted and can explain its functionality during the viva examination.

## 8. Conclusion

This assignment provided hands-on experience with process and thread-based parallelism in Linux. The key takeaways are:

1. Processes (fork) and threads (pthread) offer different trade-offs in terms of isolation vs. efficiency
2. CPU-bound workloads benefit most from parallelism, achieving near-linear scaling
3. Memory-bound workloads show diminishing returns due to shared memory bandwidth
4. I/O-bound workloads cannot be sped up through CPU parallelism alone
5. Proper measurement requires understanding of tools like top, taskset, iostat, and time

**GitHub Repository:** **https://github.com/WiseShukla/GRS_Assignment1**

# 9. Appendix: Complete Source Code

## 9.1 Makefile

```
CC = gcc
CFLAGS = -Wall -Wextra -O2
LDFLAGS_A = -lm
LDFLAGS_B = -lpthread -lm

all: program_a program_b

program_a: PhD25001_Part_A_Program_A.c
	$(CC) $(CFLAGS) -o program_a PhD25001_Part_A_Program_A.c $(LDFLAGS_A)

program_b: PhD25001_Part_B_Program_B.c
	$(CC) $(CFLAGS) -o program_b PhD25001_Part_B_Program_B.c $(LDFLAGS_B)

clean:
	rm -f program_a program_b *.csv *.png *.tmp
```

**Note:** Complete source code files (Program A, Program B, Shell Scripts) are included in the submission folder.