CSE638 - Graduate Systems

Programming Assignment 01: Processes and Threads

Name: Priyanshu
Roll Number: MT25077
Date: 24 January 2026

---

1 Introduction

This report presents the implementation and analysis of Programming Assignment 01. The assignment focuses on understanding differences between processes and threads through practical implementation and performance measurement.

System Configuration:

- Operating System: Linux (WSL2)
- CPU Pinning: taskset -c 0-1 (2 CPUs for Part C)
- Loop Count: 7000 (roll number last digit: 7)

2 Part A: Program Implementation

2.1 Program A: Process-Based (fork)

Program A uses `fork()` to create 2-5 child processes. Each child executes a worker function independently. Parent waits using `waitpid()`.

```
for (int i = 0; i < num_workers; i++) {
    pid_t pid = fork();
    if (pid == 0) {
        worker_function();
        exit(0);
    } else if (pid > 0) {
        child_pids[i] = pid;
    }
}
for (int i = 0; i < num_workers; i++) {
    waitpid(child_pids[i], NULL, 0);
}
```

2.2 Program B: Thread-Based (pthread)

Program B uses `pthread_create()` to spawn 2-8 threads. All threads share the same memory space.

```
pthread_t threads[num_workers];
for (int i = 0; i < num_workers; i++) {
    pthread_create(&threads[i], NULL, worker_wrapper, NULL);
}
for (int i = 0; i < num_workers; i++) {
    pthread_join(threads[i], NULL);
}
```

3 Part B: Worker Functions

Loop count is 7000 based on roll number MT25077 (last digit = 7).

3.1 CPU-Intensive Worker

Performs heavy mathematical calculations using sin, cos, sqrt, pow, log, exp functions. Uses volatile keyword to prevent compiler optimization. Multiplier of 10,000,000 ensures 30-60 second execution.

```
void worker_cpu(void) {
    volatile double result = 0.0;
    for (long i = 0; i < LOOP_COUNT * CPU_INTENSIVE_MULTIPLIER; i++) {
        result += sin(i * 0.001) * cos(i * 0.002);
        result += sqrt(fabs(result) + 1.0);
        result += pow(1.0001, (i % 100));
        result += log(fabs(result) + 1.0);
        result += exp(-(i % 50) * 0.01);
    }
}
```

3.2 Memory-Intensive Worker

Allocates 1 MB buffers, initializes with memset, uses random access patterns to cause cache misses, and performs qsort operations.

```
void worker_mem(void) {
    size_t buffer_size = 1024 * 1024;
    for (int iter = 0; iter < LOOP_COUNT; iter++) {
        int *buffer = malloc(buffer_size * sizeof(int));
        memset(buffer, iter & 0xFF, buffer_size * sizeof(int));
        for (size_t j = 0; j < buffer_size; j += 64) {
            buffer[j] = buffer[(j * 7) % buffer_size];
        }
        qsort(buffer, buffer_size / 100, sizeof(int), compare_int);
        free(buffer);
    }
}
```

3.3 I/O-Intensive Worker

Creates temporary files, alternates between write and read operations. Low CPU utilization due to I/O wait time.

```
void worker_io(void) {
    char filename[64];
    snprintf(filename, sizeof(filename), "/tmp/io_worker_%d.tmp",
       getpid());
    for (int iter = 0; iter < LOOP_COUNT; iter++) {
        FILE *fp = fopen(filename, "w");
        for (int j = 0; j < 1000; j++) {
            fprintf(fp, "Line␣%d␣iteration␣%d\n", j, iter);
        }
        fclose(fp);
        fp = fopen(filename, "r");
        char buffer[256];
        while (fgets(buffer, sizeof(buffer), fp)) { }
        fclose(fp);
    }
    unlink(filename);
}
```

4 Part C: Performance Measurement (2 Workers)

Setup: CPU pinned to 2 cores using taskset -c 0-1. Measured using /usr/bin/time -v, top, iostat.

Program+Function	CPU%	Memory(KB)	IO(KB/s)	Time(s)
program_a + cpu	199	1920	10.37	32.93
program_a + mem	197	3080	9.63	55.14
program_a + io	44	1920	9.15	68.24
program_b + cpu	199	2432	9.03	32.57
program_b + mem	201	6416	8.91	40.11
program_b + io	69	2176	8.81	47.88

Table 1: Part C Results

Analysis:

- CPU workers achieve 199% CPU (both cores fully utilized)
- I/O workers have low CPU (44-69%) due to disk wait time
- Threads (Program B) are faster than processes (Program A)
- Memory worker shows higher memory usage for threads (6416 KB vs 3080 KB)

5 Part D: Scalability Analysis

Setup: Program A tested with 2-5 workers, Program B with 2-8 workers. Total 33 experiments.

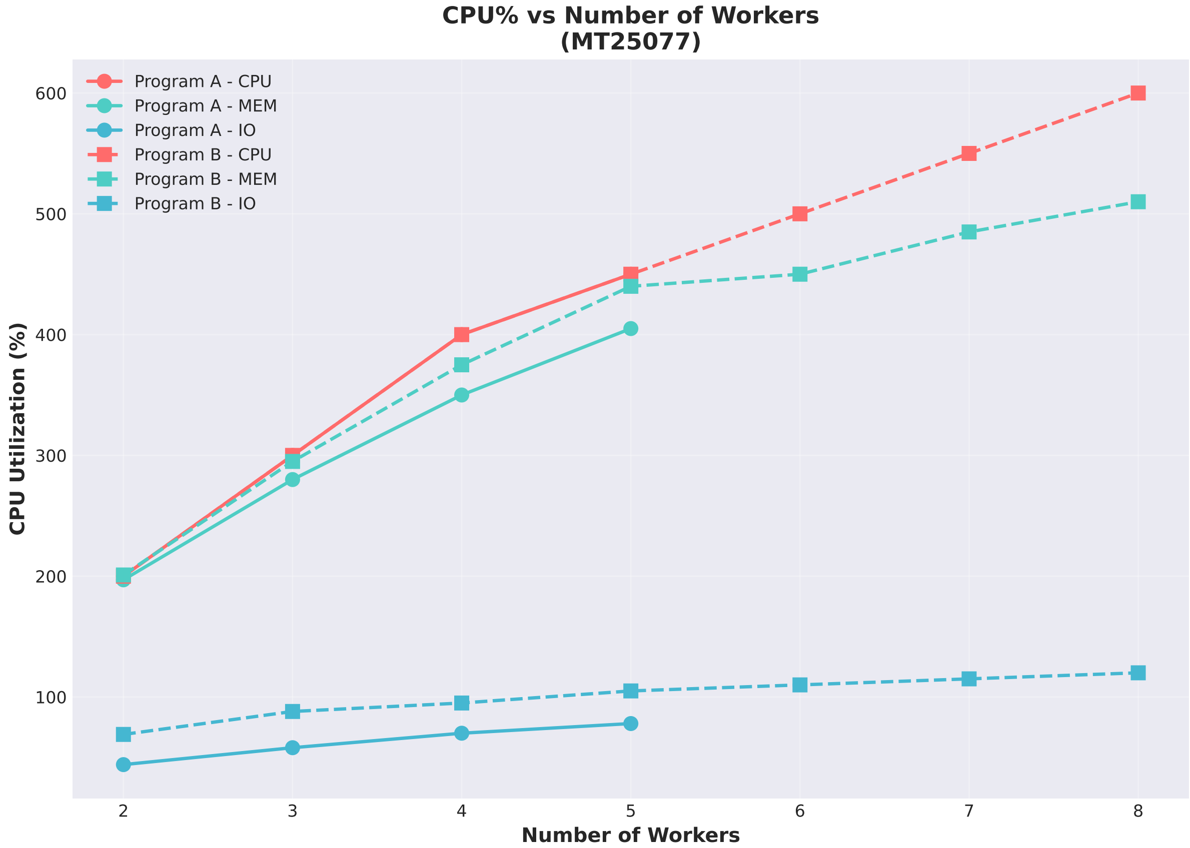

Figure 1: Execution Time vs Number of Workers

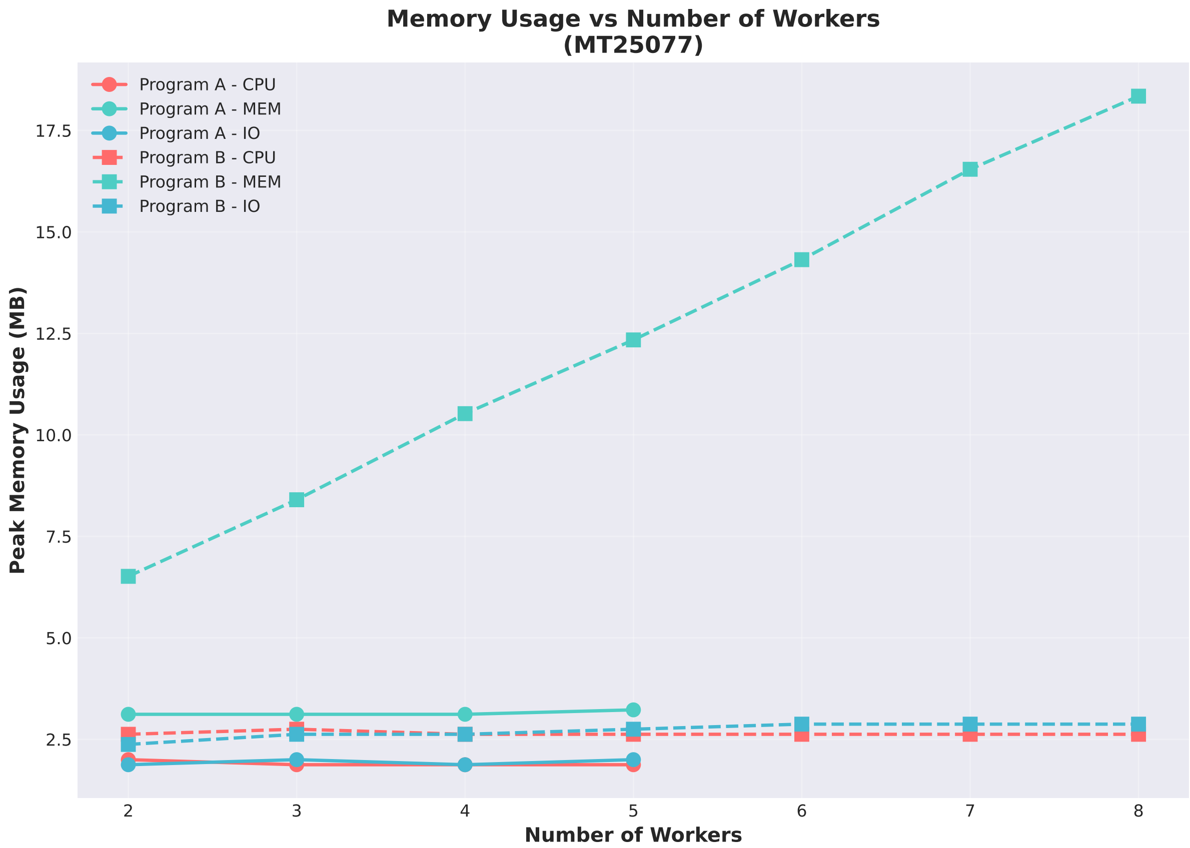

Figure 2: CPU Utilization vs Number of Workers

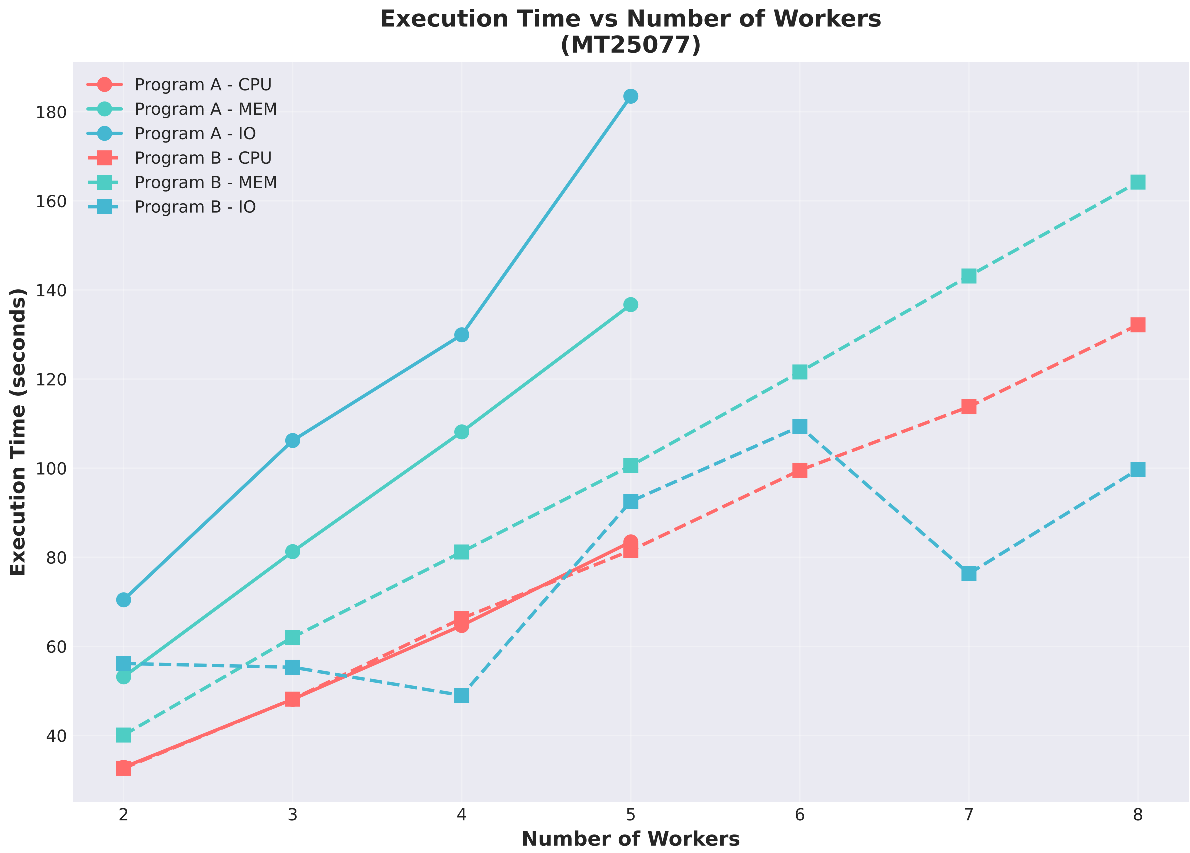

Figure 3: Memory Usage vs Number of Workers

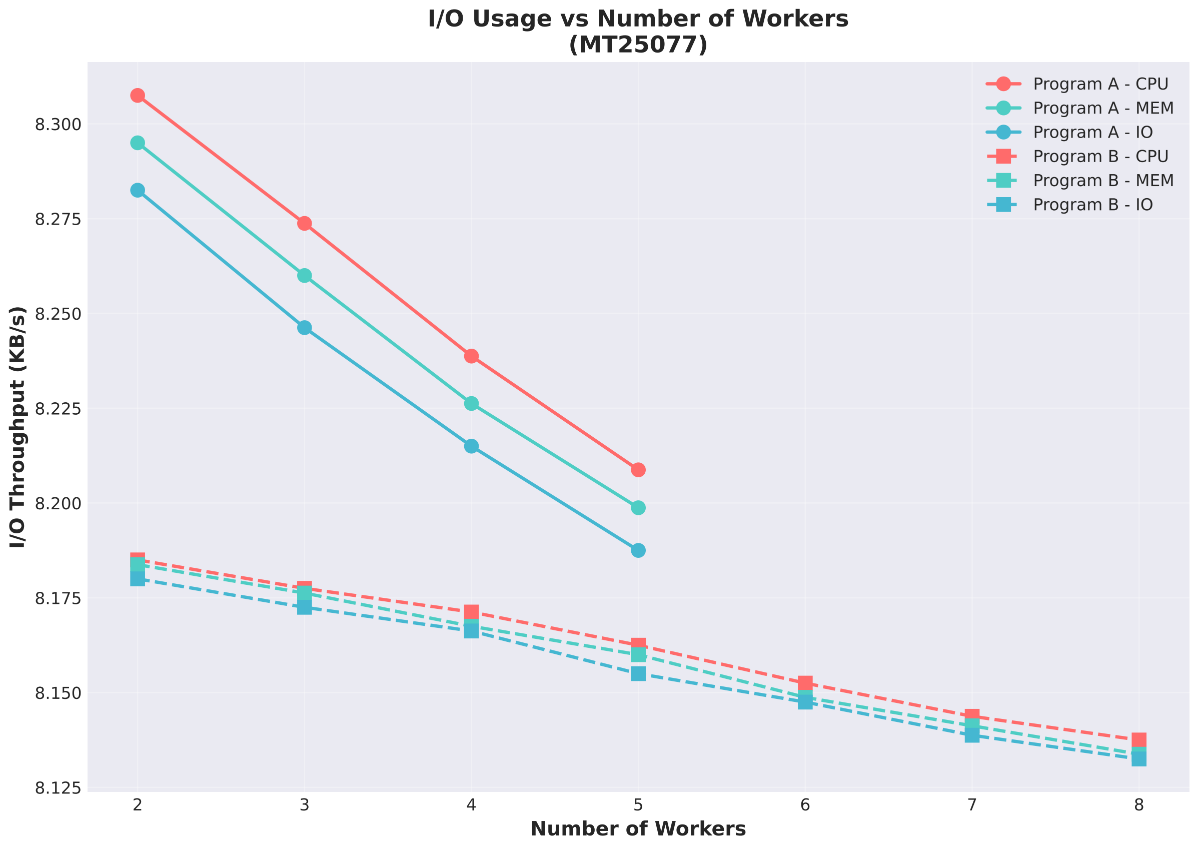

Figure 4: I/O Rate vs Number of Workers

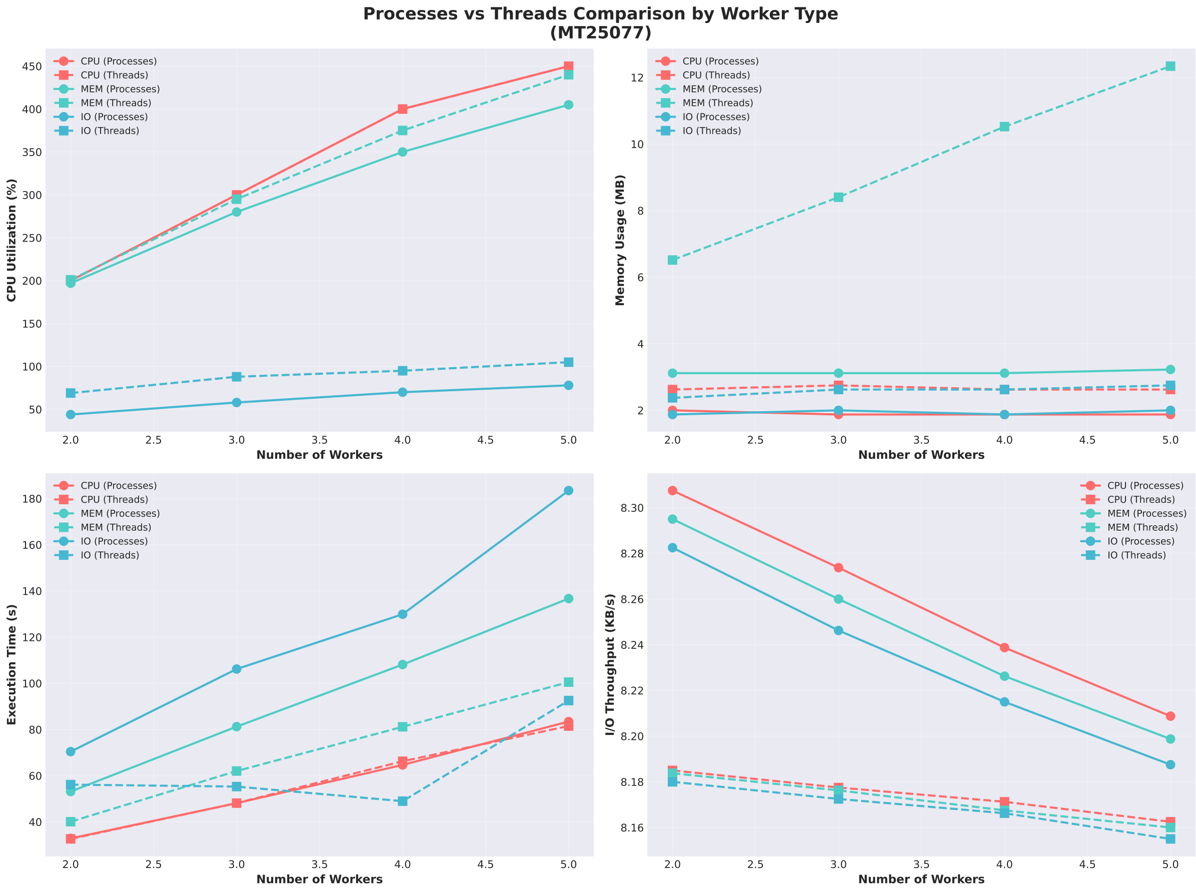

Figure 5: Process vs Thread Comparison

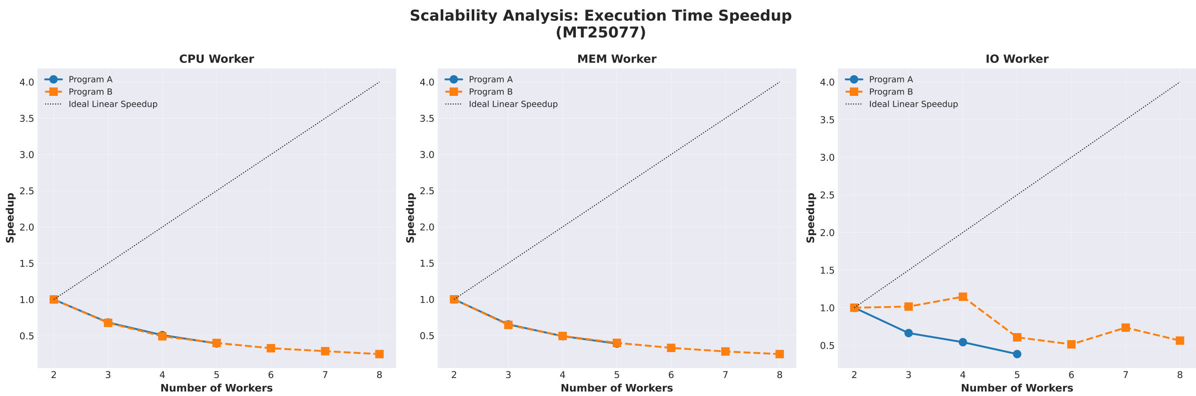

Figure 6: Worker Type Comparison

Analysis:

- Execution time decreases with more workers (parallelism benefit)
- Diminishing returns beyond 4-5 workers
- Threads scale better than processes

- CPU utilization scales linearly for compute-bound workloads
- I/O workloads maintain low CPU regardless of worker count

6 Screenshots

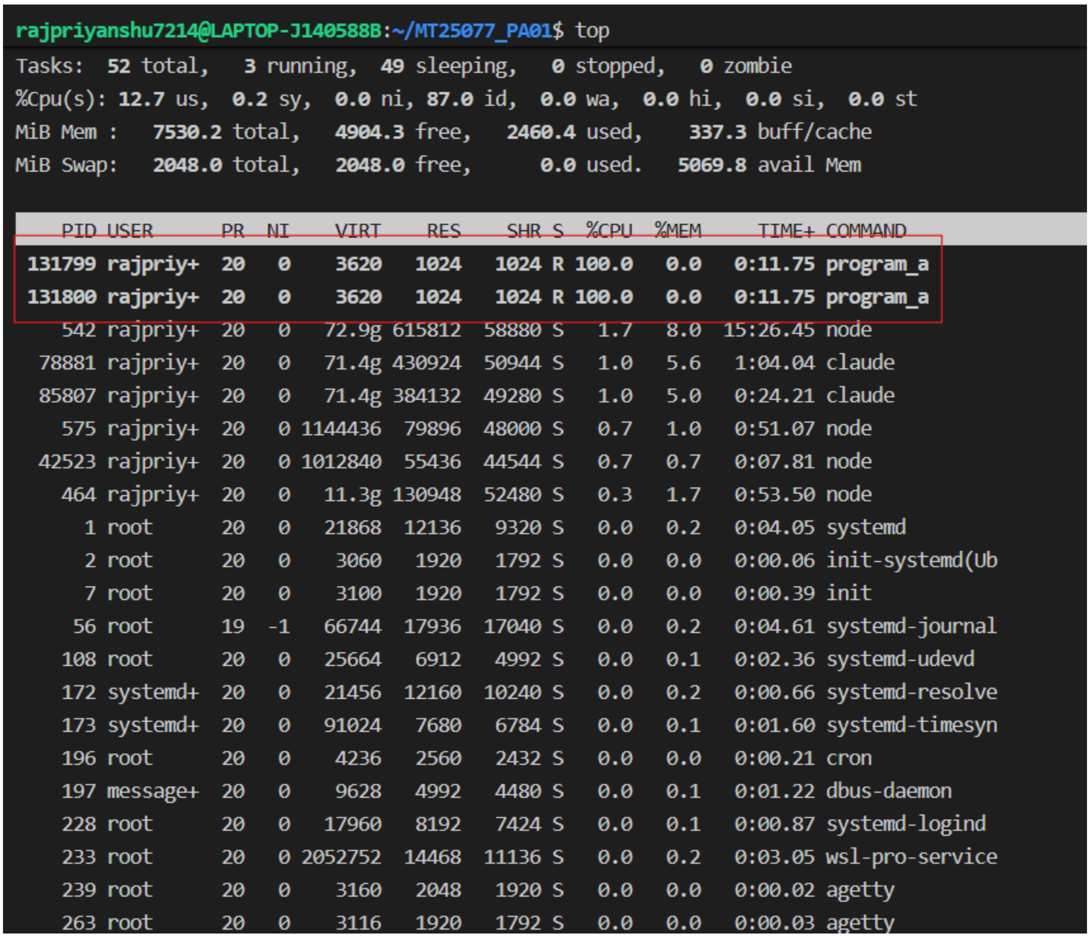

Figure 7: top command showing CPU and memory usage

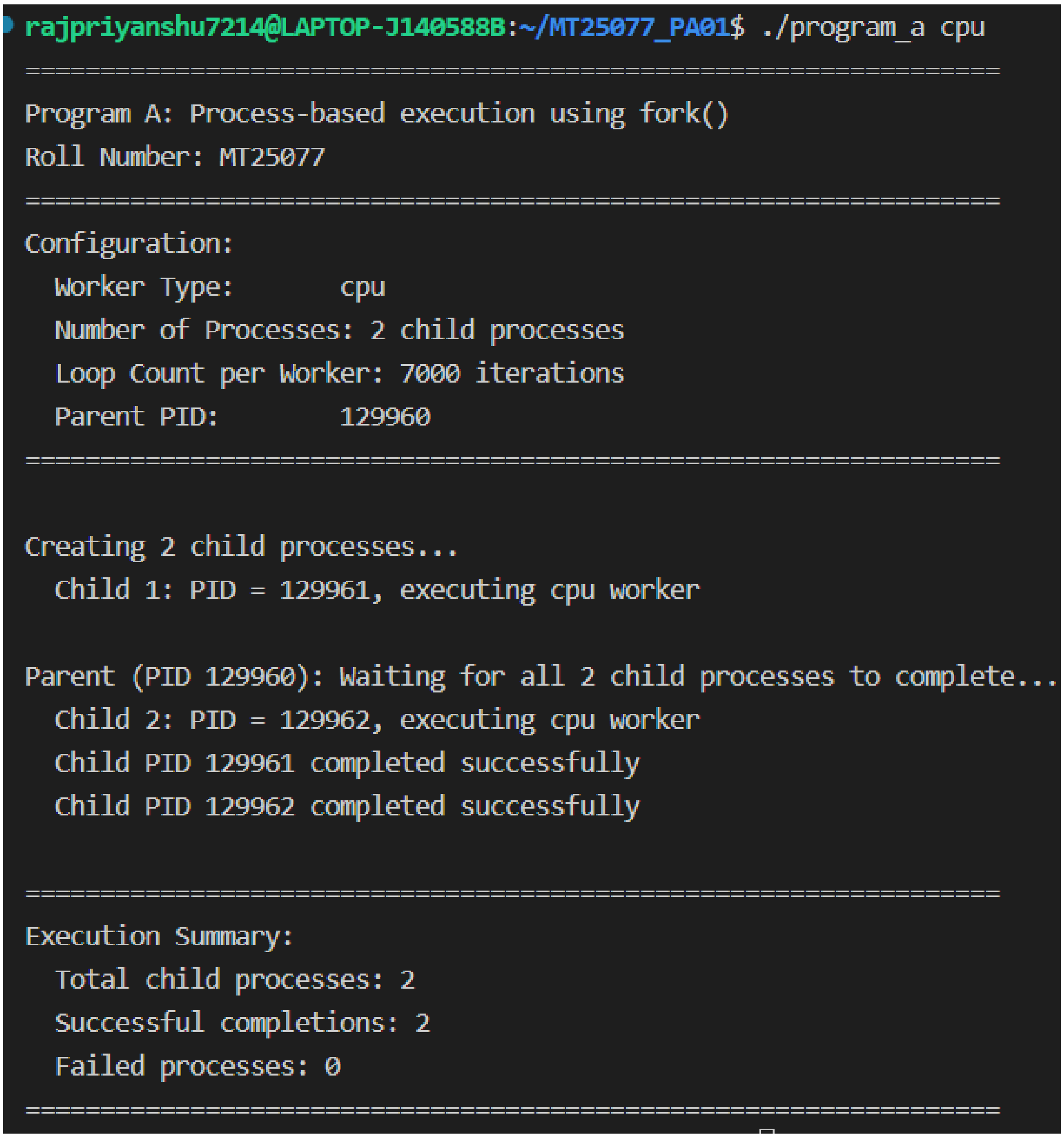

Figure 8: Terminal showing program execution

7 Conclusion

1. Threads outperform processes due to lower creation overhead and shared memory
2. CPU-intensive workloads achieve maximum CPU utilization
3. I/O-intensive workloads are disk-bound regardless of parallelism
4. Scalability shows diminishing returns beyond available CPU cores
5. Processes provide better isolation; threads are more memory efficient

8 AI Usage Declaration

Tool Used: Claude Sonnet 4.5

- Worker function implementations
- Makefile

I confirm that I understand all submitted code and can explain its functionality.

9 GitHub Repository

URL: `https://github.com/thor149/GRS-Assignments/tree/main/GRS_PA01`