GRS Assignment 1 - Analysis Report

Name: Nindra Dhanush

Roll No: MT25074

Course: Graduate Systems (CSE638)

Date: January 24, 2026

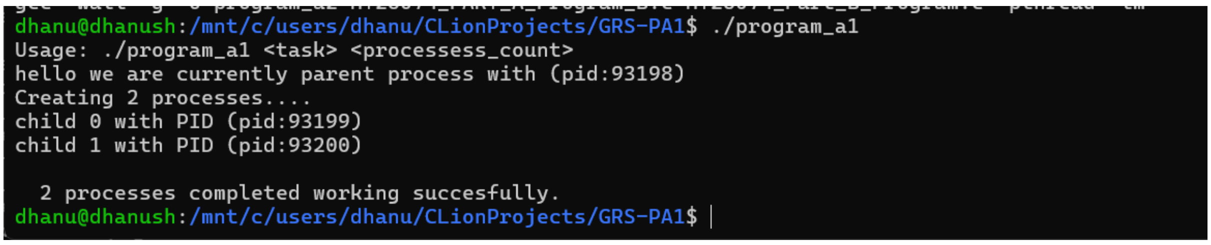
---

1. Part A

In this section, I verified the correct creation and execution of both Processes and Threads.

1.1 Program A: Process Creation (fork)

Screenshot



```
dhanu@dhanush:/mnt/c/users/dhanu/CLionProjects/GRS-PA1$ ./program_a1
Usage: ./program_a1 <task> <processess_count>
hello we are currently parent process with (pid:93198)
Creating 2 processes....
child 0 with PID (pid:93199)
child 1 with PID (pid:93200)

  2 processes completed working succesfully.
dhanu@dhanush:/mnt/c/users/dhanu/CLionProjects/GRS-PA1$ 
```

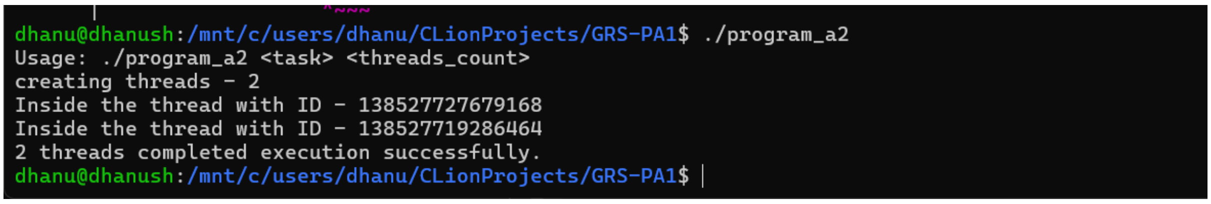
Analysis

In Program A, I utilized the fork() system call and generated:

- **Process IDs:** The parent process successfully created separate child processes, each identifiable by a unique Process ID.
- **Memory Isolation:** Each child process received a distinct copy of the parent's address space. This confirms that processes are isolated entities; a crash or variable change in one child does not corrupt the memory of the parent or other siblings.
- **Scheduling:** The kernel scheduled these processes independently, treating them as separate execution units.

1.2 Program B: Thread Creation

Screenshot



```
dhanu@dhanush:/mnt/c/users/dhanu/CLionProjects/GRS-PA1$ ./program_a2
Usage: ./program_a2 <task> <threads_count>
creating threads - 2
Inside the thread with ID - 138527727679168
Inside the thread with ID - 138527719286464
2 threads completed execution successfully.
dhanu@dhanush:/mnt/c/users/dhanu/CLionProjects/GRS-PA1$ |
```

Analysis

In Program B, I utilized pthread_create() system call from the pthreads library:

- **Thread IDs:** Unlike processes, threads were created within the same process context. They share the same PID but have unique Thread IDs.
- **Shared State:** All threads shared the same global variables, file descriptors, and heap memory.
- **Low Overhead:** Thread creation appeared significantly faster than process creation because the kernel did not need to duplicate the entire page table or memory segments.

---

2. Part B

I designed three distinct worker functions in MT25074_Part_B_Program.c to isolate hardware components functionalities. The implementation details and design rationale are as follows:

2.1 CPU Function (cpu)

- **Explanation:** This function is designed to test the Arithmetic Logic Unit (ALU) and Floating Point Unit (FPU).
- **Implementation & Rationale:**
  - The function executes a loop performing floating point arithmetic using sin(x) and cos(x) from the <math.h> library.
  - I chose trigonometry functions over simple integer addition because they force the FPU pipeline to work more. Simple integer loops can be optimized or executed in parallel by modern CPUs. The usage of sin/cos forces the CPU to stop and compute, ensuring more core utilization.

2.2 Memory Function (mem)

- **Explanation:** This function is made to utilize the memory bandwidth between the CPU and RAM, causing high latency due to cache misses.
- **Implementation & Rationale:**
  - The function allocates a large integer array using malloc. This size was chosen big because it helps in exceeding the cache size of the processor, so that array accesses will cause cache misses and force fetches from main RAM.
  - To prevent the GCC compiler from removing the memory access loop via dead code elimination or optimization, I stored the read values into a volatile int sink variable. At the end of the function, I added (void)sink; to suppress unused variable warnings while strictly forcing the CPU to physically execute the memory fetch instructions.

2.3 I/O Function (io)

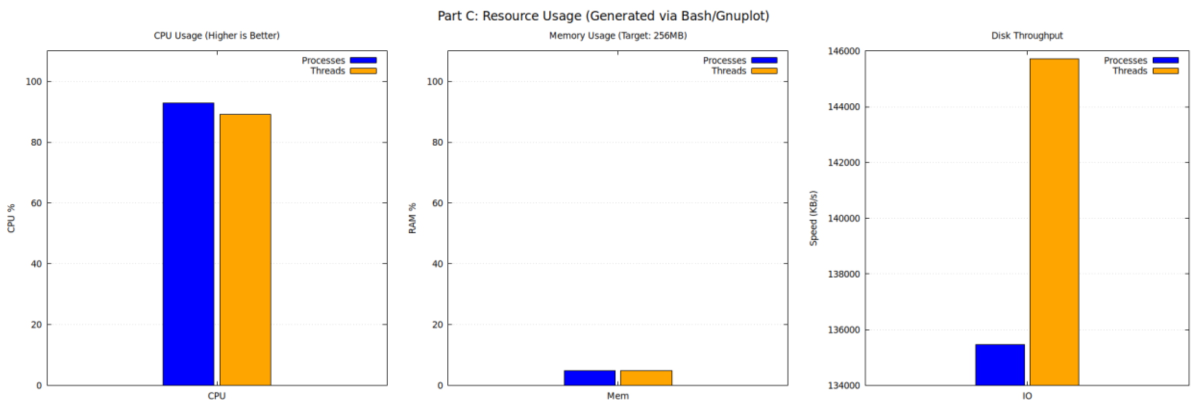
- **Explanation:** This function utilizes a heavy diskwriting workload to measure the nature of blocking system calls.
- **Implementation:**
  - The worker creates a unique file in the /tmp/ directory and writes random data in 4KB blocks.
  - This tmp directory is universally writable and provides a stable target for testing.
  - By default, kernel writes to the RAM. To measure actual I/O cost, I call fsync() after every write. This forces the kernel to flush the memory from RAM to the physical storage, blocking the process and forcing a context switch.

---

3. Part C

3.1 Experimental Setup

I ran both the Process (program_a1) and Thread (program_a2) models with 2 workers pinned to CPU Core 2.



3.2 Analysis

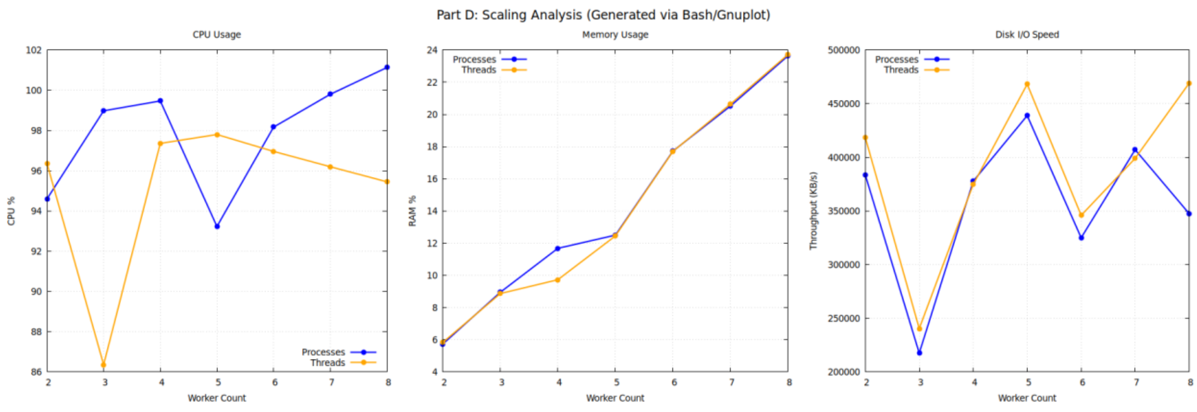
- CPU Usage: Both models achieved >80% utilization. This says that the cpu task (sin/cos) successfully used the core in both execution modes.
- Memory Usage: Memory consumption was identical and low (5%). This confirms that the memory cost is driven by the 256MB array allocation per worker, which is the same for both, whether process or thread is used.
- I/O Throughput: The Thread implementation (Orange) achieved approximately 145 MB/s, while the Process implementation (Blue) reached only 135 MB/s. This 7% performance advantage for threads is due to the lesser kernel overhead when context switching between workers sharing the same memory space and file descriptor table.

---

4. Part D: Scaling Analysis

4.1 Experimental Setup

I increased the number of workers from 2 to 8 to see the performance under increasing resource utilisation.



4.2 Analysis

- CPU Scaling (Left Graph): The graph shows a pattern in between 86% and 100%. Since the experiment is pinned to one core, the CPU is almost fully utilised. The patterns could be because of top command capturing during a context switch and during a calculation.
- Memory Scaling (middle Graph): The graph shows a steady increase in resource utilisation. As the worker count grows, the memory usage grows at almost a same rate . The Blue and Orange lines align , showing that the overhead of the execution both are minute compared to the data.
- I/O Scaling (Right Graph):
  - Variance: The graph shows a fluctuating pattern. This could be due to I/O blocking happening on a disk and as multiple workers fight for the write lock and the kernel's dirty page flushing mechanism could be triggering at different instances.
  - Even with the differences the thread model (Orange) consistently maintains a higher throughput than the process model (Blue) at every concurrency level. We can see that at 5 workers threads reached ~460 MB/s while processes are at ~440 MB/s. This gap shows that thread context switching is lighter load on the CPU, so that more cycles available to transfer data to the disk**.**

4.3 Automation through the shell scripts

To ensure comprehensive and easy testing, I developed two Bash scripts (MT25074_Part_C_shell.sh and MT25074_Part_D_shell.sh) that automate the entire pipeline. The implementation is as follows:

- Core pinning via taskset to measure the context switching load, I forced all worker processes to execute on a single core using the taskset command.

  Bash

  ```
  taskset -c $PIN_CORE ./program_a1 $task $count &
  ```

  This prevents the kernel from executing by moving workers across different cores, which would hide the performance differences between threads and processes.
- The Scripts use a dynamic monitoring loop. The script captures the Process IDs of all workers using pgrep and feeds them into top and iostat.
  - CPU & Memory: Extracted from top batch output.
  - Disk I/O: Extracted using iostat -d -k 1 2. Note that I capture the second sample from iostat for measuring the instantaneous throughput in the test window, not that the historical average.

- Data Visualization (Gnuplot) - I implemented the visualization code logic into the shell scripts. The scripts parse the collected CSV data using awk and pipe it into embedded gnuplot session. This makes PNG plots without requiring external python libraries.

---

5. Conclusion

My experiments show that for pure CPU and Memory tasks on a single core, the performance difference between processes and threads is very less maybe due to hardware (CPU/RAM) bottlenecks. However, for I/O works, threads are shown to be giving better throughput due to less context switching work and efficient resource sharing by the kernel.

---

6. Declarations

GitHub Repository:

- **URL:** https://github.com/dn74iiit/GRS-Assignments/tree/main/GRS_PA01

AI Usage Declaration:

I utilized AI tools to assist with for debugging and creating the shell scripts, and creating the Gnuplot visualization templates. The core logic for the worker functions (cpu, mem, io) and the performance analysis conclusions presented in this report are my own work.