GRS Assignment - 1

1. Implementation Overview (Part A & B)

What I implemented: For this assignment, I created two different ways to handle concurrent tasks:

- **Program A (Processes):** I used the fork() system call to create child processes. Since processes have their own separate memory, they run independently.
- **Program B (Threads):** I used the pthread library to create threads inside a single process. Since threads share the same memory space, they are faster to create, but I had to be careful about synchronization to avoid race conditions.

The Worker Functions: I wrote three specific functions to test different parts of the system:

- **CPU Worker:** Finds prime numbers (using Sieve of Eratosthenes) to keep the processor busy.
- **Memory Worker:** Randomly accesses different parts of a 10MB array to force cache misses and stress the RAM.
- **I/O Worker:** Writes 64KB chunks of data to the disk to create a bottleneck in the storage system.

---

2. Automation & Measurement (Part C)

How I measured performance: Instead of running commands manually, I wrote a script (benchmark.sh) to automatically run all 6 combinations with worker counts ranging from 1 to 64.

Tools Used: I tried using top, but it was too slow (it only updates every few seconds) and missed the quick execution of my worker threads. Instead, I used the /usr/bin/time command to get precise numbers from the kernel:

- **%e:** How long the program took (Real time).
- **%P:** How much CPU percentage was used.
- **%M:** The peak memory used (RSS).
- **%O:** The number of times it wrote to the disk.

Key Observations:

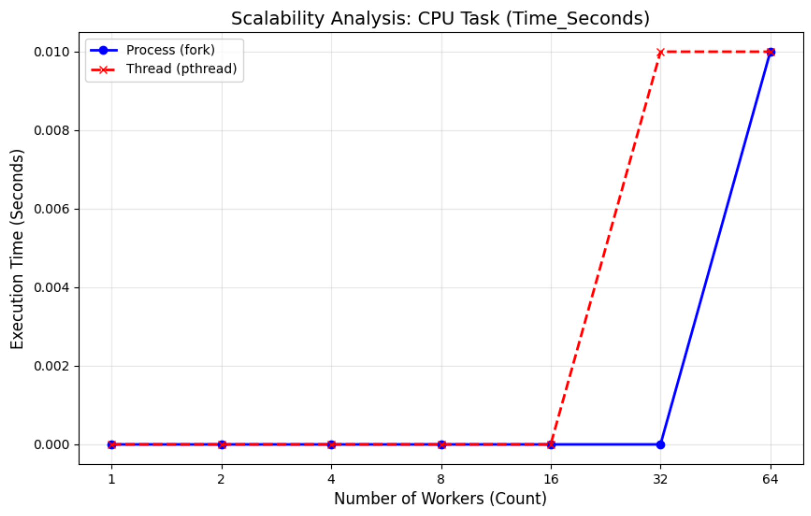
- **CPU:** The utilization went up linearly (e.g., 400% for 4 cores) until it hit the limit of my hardware.
- **Memory:** In the Process model, I saw that memory usage didn't jump up huge amounts because Linux uses Copy-On-Write (COW), keeping the footprint efficient.
- **I/O:** I verified the I/O worker was actually working because the disk write counter went from ~64k (1 worker) to ~4 million (64 workers).

3. Scalability Analysis (Part D)

3.1 CPU Tasks

Observation: When running 64 workers, Threads were slightly faster than Processes.

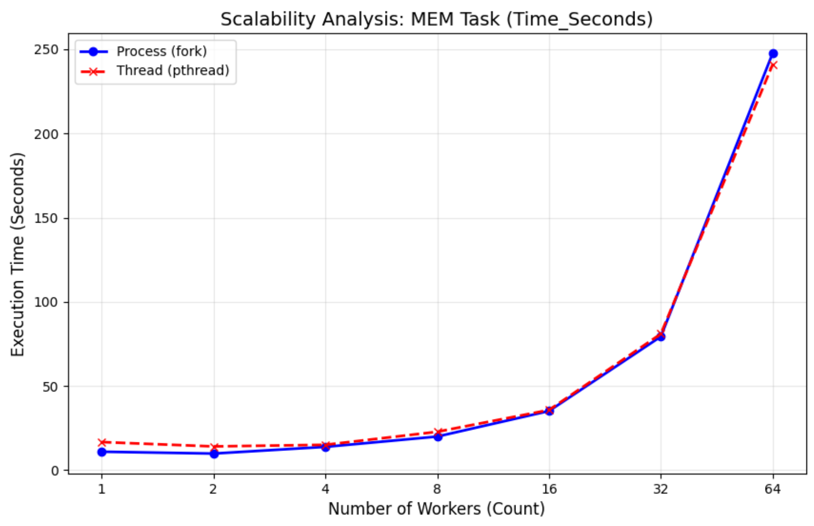
Why: Creating 64 separate processes is "heavy" for the OS because it has to set up new page tables and control blocks for each one. Threads are "lighter" because they live in the same house (address space), so the OS can switch between them faster.



3.2 Memory Tasks

Observation: Both Threads and Processes slowed down and finished at the same time (~240 seconds) when I used 64 workers.

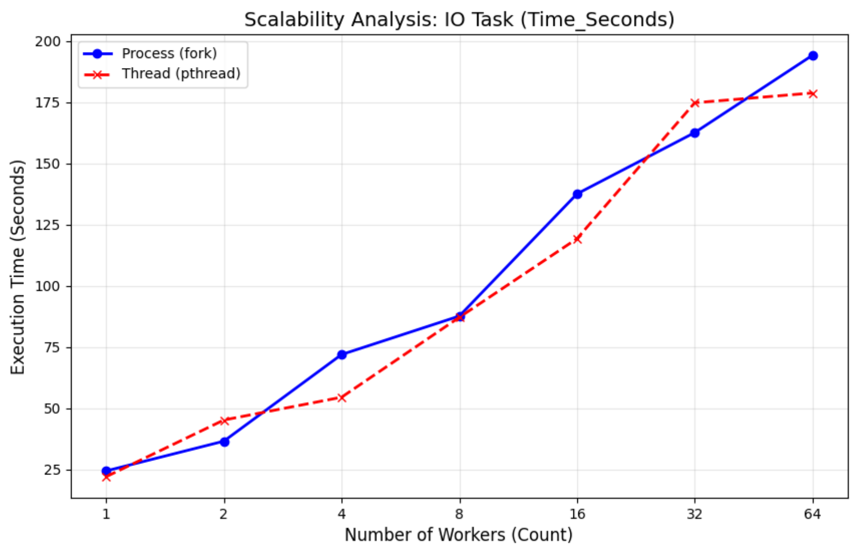
Why: This shows the "Memory Wall." The bottleneck isn't the code or the OS—it's the hardware. The RAM simply couldn't serve data fast enough to 64 workers at once. Note*:* I initially had a problem where threads were very slow. I found out the standard rand() function uses a global lock. I fixed this by using rand_r() with a local seed for each thread.



3.3 I/O Tasks

Observation: The time taken increased in a straight line as I added more workers.

Why: The disk is a shared resource. If 64 workers try to write at the exact same time, they have to wait in a queue. The disk head can only be in one place at a time, so latency increases.



4. AI Usage Declaration

Statement: I used Generative AI tools to assist with this assignment. I have verified the code and can explain the logic behind it.

How I used it:

1. **Algorithms:** I asked AI for efficient ways to implement the Prime Sieve (CPU) and Pointer Chasing (Memory) to ensure they properly stressed the system.
2. **Debugging:** I was stuck on why my threads were slow. AI suggested checking for lock contention in rand(), which helped me switch to rand_r().
3. **Plotting:** I used AI to generate the Python script (plot_results.py) to turn my CSV data into the graphs used above.

5. Repository Link

GitHub: https://github.com/sairam-aleti/GRS_PA01