

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 for automated measurement, but its sampling rate was too slow to capture the sub-second execution times of the workers. Therefore:

- **For Measurement:** I used /usr/bin/time to capture precise kernel-level metrics (Latencies, CPU%) for the CSV data.
- **For Validation:** I used top manually (with extended execution loops) solely to visually confirm CPU affinity and process spawning, as shown in the screenshots below.

System Validation (Screenshots):

Fig 1: Data Generation. Verification of the CSV output files generated by the automation script.

```
sai_ram@saiRam:~/GRS_PA01$ ls -lh results/
total 8.0K
-rw-r--r-- 1 sai_ram sai_ram 765 Jan 17 17:44 MT25038_Part_C_CSV.csv
-rw-r--r-- 1 sai_ram sai_ram 754 Jan 17 18:02 MT25038_Part_D_CSV.csv
sai_ram@saiRam:~/GRS_PA01$
```

Fig 2: CPU Affinity Verification. The top output (Right) confirms that executing with taskset -c 0 successfully pins the workload to Cpu0 (100% utilization), leaving other cores idle.

top - 13:40:40 up 49 min, 1 user, load average: 1.23, 1.65, 0.89										
Tasks: 28 total, 2 running, 25 sleeping, 1 stopped, 0 zombie										
%Cpu0 : 46.6 us, 25.0 sy, 0.0 ni, 28.1 id, 0.0 wa, 0.0 hi, 0.3 si, 0.0 st										
%Cpu1 : 0.0 us, 0.0 sy, 0.0 ni, 100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st										
%Cpu2 : 0.0 us, 0.0 sy, 0.0 ni, 100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st										
%Cpu3 : 0.0 us, 1.0 sy, 0.0 ni, 99.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st										
%Cpu4 : 0.7 us, 2.7 sy, 0.0 ni, 96.6 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st										
%Cpu5 : 0.0 us, 0.3 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st										
%Cpu6 : 1.7 us, 7.6 sy, 0.0 ni, 90.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st										
%Cpu7 : 0.0 us, 0.3 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st										
%Cpu8 : 4.3 us, 7.0 sy, 0.0 ni, 88.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st										
%Cpu9 : 0.0 us, 0.0 sy, 0.0 ni, 100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st										
%Cpu10 : 1.0 us, 4.6 sy, 0.0 ni, 94.4 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st										
%Cpu11 : 0.0 us, 0.7 sy, 0.0 ni, 99.3 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st										
MiB Mem :	3779.2	total,	2958.1	free,	437.5	used,	466.9	buff/cache		
MiB Swap:	1024.0	total,	1024.0	free,	0.0	used,	3341.7	avail Mem		
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+ COMMAND
815	sai_ram	20	0	6204	5248	3584	R	8.3	0.1	0:09.83 bash
811	root	20	0	3076	1160	1024	S	2.0	0.0	0:02.41 ReLay(815)
8	root	20	0	3948	2464	1792	S	0.7	0.1	0:04.08 init
1	root	20	0	22204	12356	9028	S	0.0	0.3	0:01.79 systemd
2	root	20	0	3060	1920	1792	S	0.0	0.0	0:00.03 init-systemd(ub
47	root	19	-1	66748	18700	17804	S	0.0	0.5	0:00.67 systemd-journal
97	root	20	0	25168	6272	4864	S	0.0	0.2	0:00.79 systemd-udevd
145	systemd+	20	0	21456	12672	10496	S	0.0	0.3	0:00.17 systemd-resolve
146	systemd+	20	0	91824	7680	6784	S	0.0	0.2	0:00.20 systemd-timesync
161	root	20	0	4236	2432	2304	S	0.0	0.1	0:00.02 cron
163	message+	20	0	9640	4992	4352	S	0.0	0.1	0:00.33 dbus-daemon
180	root	20	0	17964	8192	7296	S	0.0	0.2	0:00.20 systemd-logind
183	root	20	0	1756096	12032	9984	S	0.0	0.3	0:00.30 wsl-pro-service
191	syslog	20	0	222508	5504	4480	S	0.0	0.1	0:00.15 rsyslogd
196	root	20	0	3160	1920	1792	S	0.0	0.0	0:00.01 getty
206	root	20	0	3116	1792	1664	S	0.0	0.0	0:00.01 getty

Fig 2: Disk Throughput Verification. The iostat capture (Right) confirms sustained write throughput (approx. 8 MB/s) generated by the I/O worker threads.

sda	0.00	0.00	0.00	0.00	0	0
sdb	0.00	0.00	0.00	0.00	0	0
sdc	0.00	0.00	0.00	0.00	0	0
sdd	2254.00	0.00	8.74	0.00	0	8
	0					
Device	tps	MB_read/s	MB_wrtn/s	MB_dscd/s	MB_read	MB_wrtn
MB_dscd						
sda	0.00	0.00	0.00	0.00	0	0
sdb	0.00	0.00	0.00	0.00	0	0
sdc	0.00	0.00	0.00	0.00	0	0
sdd	2164.00	0.00	8.45	0.00	0	8
	0					
Device	tps	MB_read/s	MB_wrtn/s	MB_dscd/s	MB_read	MB_wrtn
MB_dscd						
sda	0.00	0.00	0.00	0.00	0	0
sdb	0.00	0.00	0.00	0.00	0	0
sdc	0.00	0.00	0.00	0.00	0	0
sdd	2106.00	0.00	8.27	0.00	0	8
	0					

Key Observations:

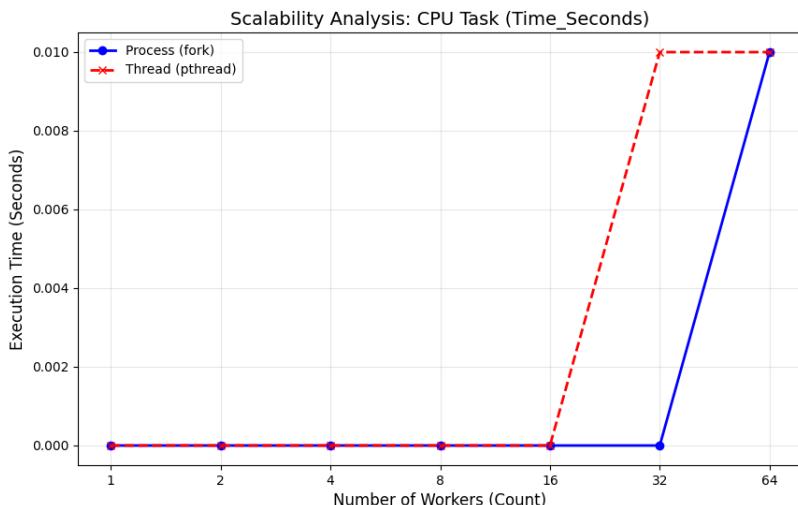
- **CPU:** The utilization went up linearly (e.g., 400% for 4 cores) until it hit the limit of my hardware.
- **Memory:** In the Process model, I saw that memory usage didn't jump up huge amounts because Linux uses Copy-On-Write (COW), keeping the footprint efficient.
- **I/O:** I verified the I/O worker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

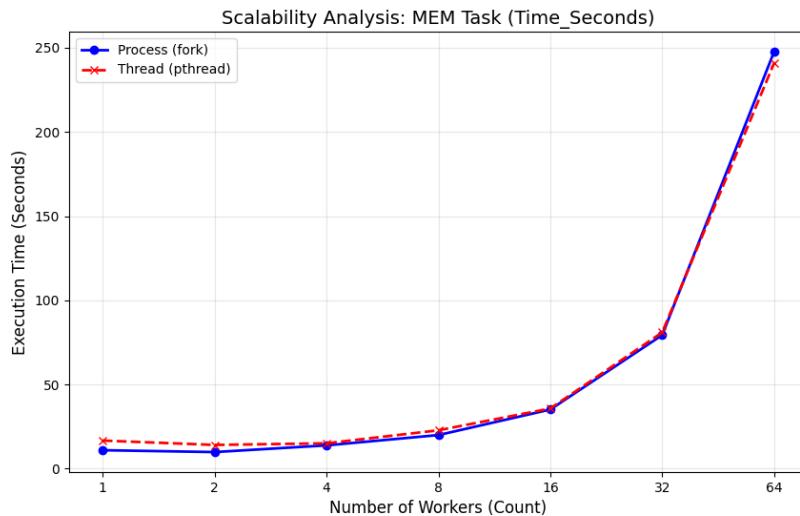
Why: Creating 64 separate processes is "heavy" for the OS because it has to set up new page tables and control blocks for each one. Threads are "lighter" because they live in the same house (address space), so the OS can switch between them faster.



3.2 Memory Tasks

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

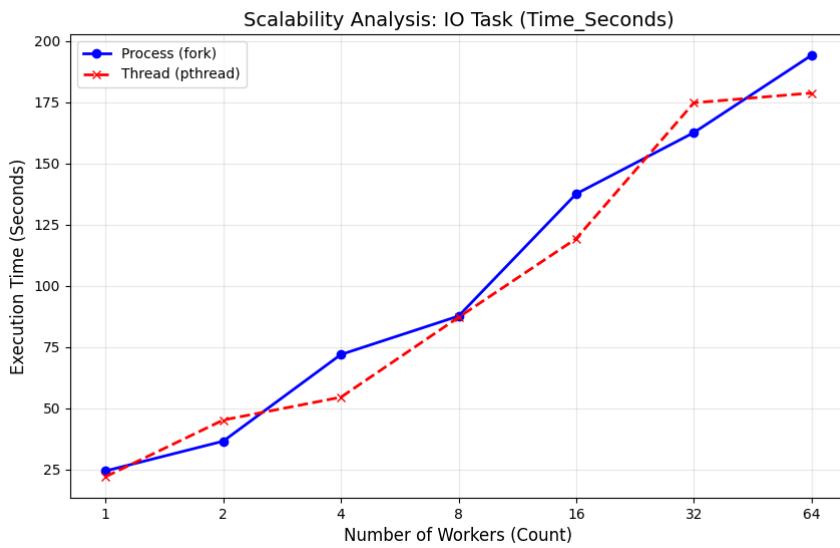
Why: This shows the "Memory Wall." The bottleneck isn't the code or the OS—it's the hardware. The RAM simply couldn't serve data fast enough to 64 workers at once. Note: I initially had a problem where threads were very slow. I found out the standard rand() function uses a global lock. I fixed this by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