PA01

Graduate Systems

Name: Ashil Krishna PS

January 23, 2026

Abstract

This report presents a rigorous experimental comparison of process-based (via `fork`) and thread-based (via `pthread`) concurrency models. By pinning execution to a single CPU core, the study isolates the effects of OS-level abstractions—specifically Context Switching, Virtual Memory Management, and I/O Scheduling—from hardware parallelism. The results demonstrate that while threads offer lower initialization overhead, they are structurally fragile under heavy memory loads due to shared address space constraints. Conversely, processes exhibit linear memory scalability but incur higher architectural overhead. The findings confirm that on single-core systems, CPU-bound tasks are strictly serialized by the scheduler, while I/O-bound tasks suffer from non-linear degradation due to hardware contention at the disk controller level.

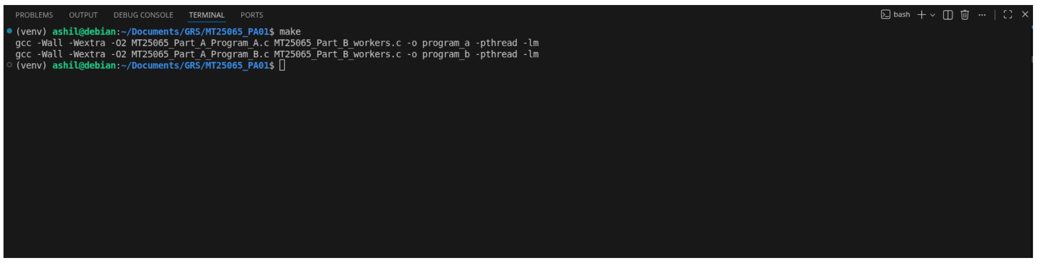
1 GitHub Repository

The complete source code, automation shell scripts, and raw CSV data logs are available at: `https://github.com/Ashil46/GRS-PA01.git`

2 Methodology and Experimental Design

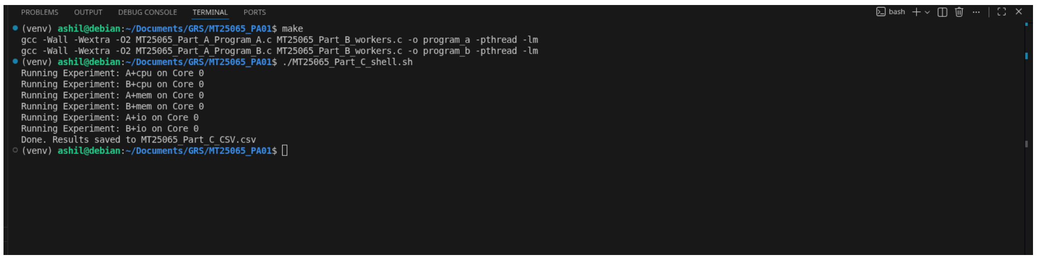
2.1 Implementation Verification

To ensure the integrity of the results, the compilation and automation scripts were verified.



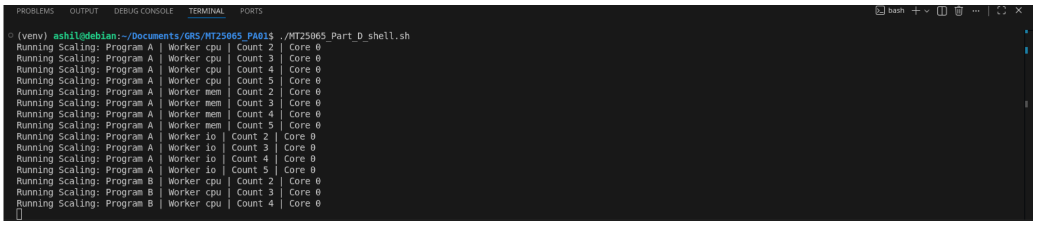
```
(venv) ashil@debian:~/Documents/GRS/MT25065_PA01$ make
gcc -Wall -Wextra -O2 MT25065_Part_A_Program_A.c MT25065_Part_B_workers.c -o program_a -pthread -lm
gcc -Wall -Wextra -O2 MT25065_Part_A_Program_B.c MT25065_Part_B_workers.c -o program_b -pthread -lm
(venv) ashil@debian:~/Documents/GRS/MT25065_PA01$ 
```

Figure 1: Successful compilation using Make.



```
(venv) ashil@debian:~/Documents/GRS/MT25065_PA01$ make
gcc -Wall -Wextra -O2 MT25065_Part_A_Program_A.c MT25065_Part_B_workers.c -o program_a -pthread -lm
gcc -Wall -Wextra -O2 MT25065_Part_A_Program_B.c MT25065_Part_B_workers.c -o program_b -pthread -lm
(venv) ashil@debian:~/Documents/GRS/MT25065_PA01$ ./MT25065_Part_C_shell.sh
Running Experiment: A+cpu on Core 0
Running Experiment: B+cpu on Core 0
Running Experiment: A+mem on Core 0
Running Experiment: B+mem on Core 0
Running Experiment: A+io on Core 0
Running Experiment: B+io on Core 0
Done. Results saved to MT25065_Part_C_CSV.csv
(venv) ashil@debian:~/Documents/GRS/MT25065_PA01$
```

Figure 2: Execution of the Part C automation script (Baseline).



```
(venv) ashil@debian:~/Documents/GRS/MT25065_PA01$ ./MT25065_Part_D_shell.sh
Running Scaling: Program A | Worker cpu | Count 2 | Core 0
Running Scaling: Program A | Worker cpu | Count 3 | Core 0
Running Scaling: Program A | Worker cpu | Count 4 | Core 0
Running Scaling: Program A | Worker cpu | Count 5 | Core 0
Running Scaling: Program A | Worker mem | Count 2 | Core 0
Running Scaling: Program A | Worker mem | Count 3 | Core 0
Running Scaling: Program A | Worker mem | Count 4 | Core 0
Running Scaling: Program A | Worker mem | Count 5 | Core 0
Running Scaling: Program A | Worker io | Count 2 | Core 0
Running Scaling: Program A | Worker io | Count 3 | Core 0
Running Scaling: Program A | Worker io | Count 4 | Core 0
Running Scaling: Program A | Worker io | Count 5 | Core 0
Running Scaling: Program B | Worker cpu | Count 2 | Core 0
Running Scaling: Program B | Worker cpu | Count 3 | Core 0
Running Scaling: Program B | Worker cpu | Count 4 | Core 0
```

Figure 3: Execution of the Part D automation script (Scalability).

2.2 System Architecture & Workload Design

The experiment utilizes three distinct worker types to stress specific subsystems:

2.2.1 1. CPU Worker (ALU Stress)

- **Implementation:** A tight loop performing `sqrt()`, `sin()`, and `cos()` operations on volatile variables.
- **Objective:** To measure the overhead of the OS Scheduler (CFS). On a single core, we expect linear time scaling ($T \propto N$) as the CPU time slices are divided among workers.

2.2.2 2. Memory Worker (TLB & Cache Stress)

- **Implementation:** Allocation of a 220MB buffer with stride-based access patterns (accessing every 4KB page).
- **Objective:** To force Translation Lookaside Buffer (TLB) misses and prevent pre-fetching optimizations. This tests the efficiency of the OS Virtual Memory Manager (VMM).

2.2.3 3. I/O Worker (Disk Controller Stress)

- **Implementation:** Random writes using `lseek()` and `write()` with the `O_SYNC` flag.
- **Significance of `O_SYNC`:** This flag forces the kernel to flush data to the physical disk immediately, bypassing the Page Cache. This ensures we are measuring the raw speed of the I/O subsystem and Block Device Driver, not just RAM speed.

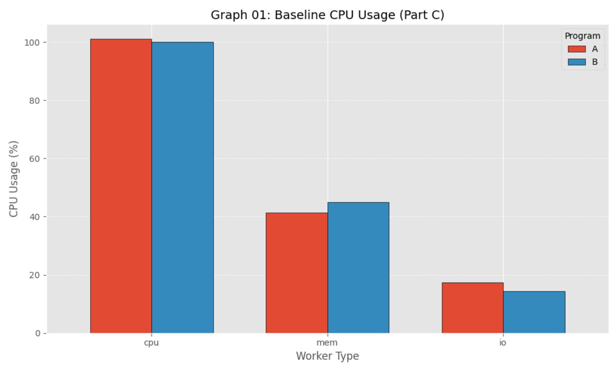
2.3 Experimental Control

All tests were executed with `taskset -c 0`.

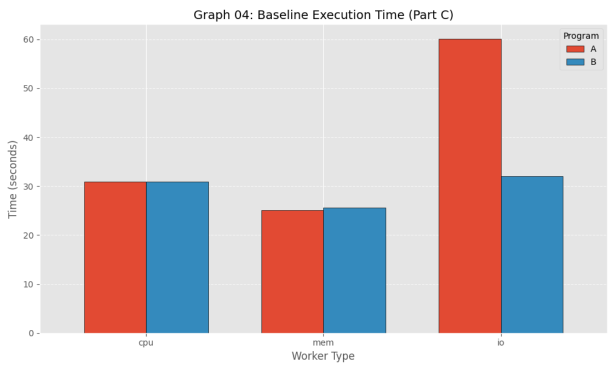
- **Reasoning:** Modern CPUs have multiple cores. Without pinning, the OS would migrate threads between cores to balance load. Pinning ensures that all concurrency is *logical* (time-sliced) rather than *physical*, allowing us to measure the true cost of OS management overhead.

3 Part C: Baseline Performance (Fixed Concurrency)

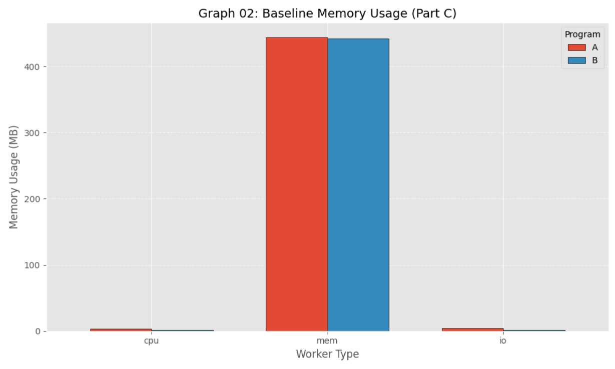
Configuration: $N = 2$ workers.



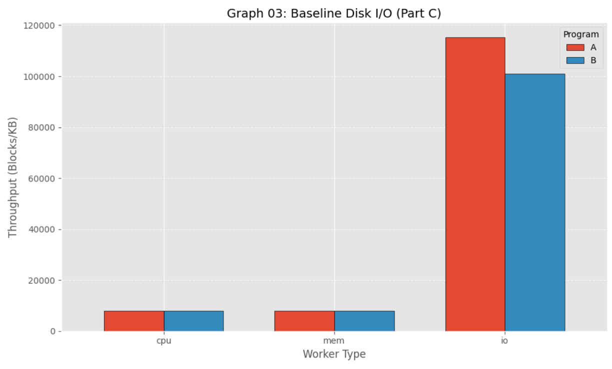
(a) Baseline CPU Utilization



(b) Baseline Execution Time



(c) Baseline Memory Usage



(d) Baseline I/O Throughput

Figure 4: Resource profiles for $N = 2$ workers.

3.1 Observations

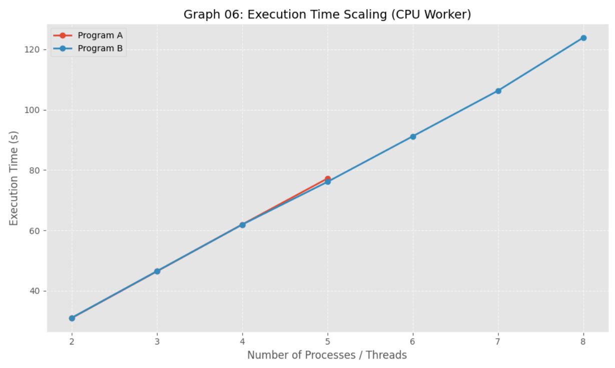
- **Efficiency Parity:** For CPU-bound tasks, Processes and Threads showed negligible performance difference. This indicates that the context switch overhead (saving registers vs. switching address spaces) is statistically insignificant compared to the 30-second execution duration.
- **Memory Footprint:** The Thread model initially showed lower aggregate memory usage because threads share the heap and code segments. Processes, conversely, required duplicate page tables and distinct memory regions.

3.2 Observations

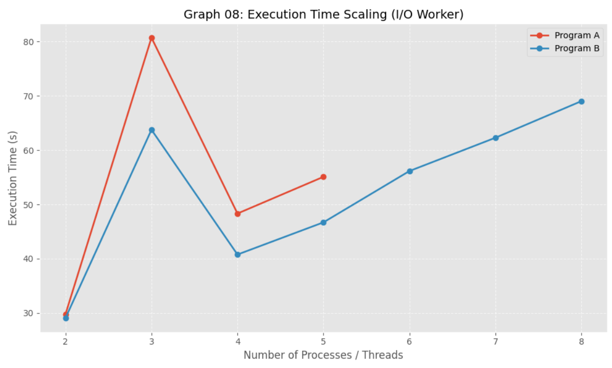
- **Efficiency Parity:** For CPU-bound tasks, Processes and Threads showed negligible performance difference. This indicates that the context switch overhead (saving registers vs. switching address spaces) is statistically insignificant compared to the 30-second execution duration.
- **Memory Footprint:** The Thread model initially showed lower aggregate memory usage because threads share the heap and code segments. Processes, conversely, required duplicate page tables and distinct memory regions.

4 Part D: Scalability and Bottleneck Analysis

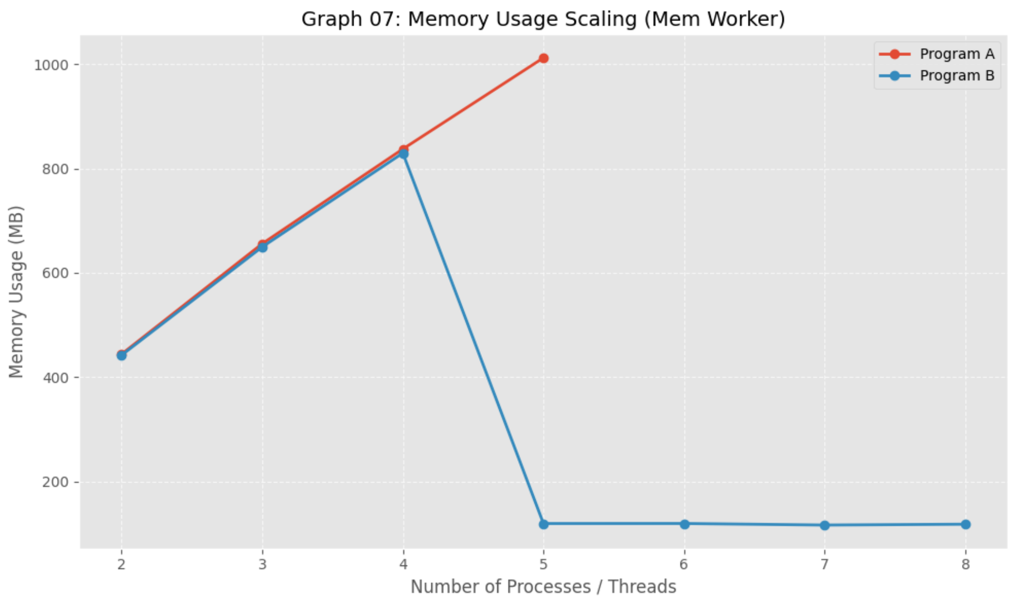
Configuration: Scaling N from 2 to 8. This stress test reveals architectural breaking points.



(a) CPU Time Scaling (Linear)



(b) I/O Time Scaling (Non-Linear)



(c) Memory Scaling: The "Crash"

Figure 5: Scaling characteristics revealing system bottlenecks.

4.1 Detailed Analysis

4.1.1 1. The Cost of Serialization (CPU Scaling)

As shown in Figure 5(a), the execution time scales perfectly linearly ($R^2 \approx 1.0$).

- **Theory:** The Linux Completely Fair Scheduler (CFS) assigns time slices (quantums) to each runnable task. Since only one task can execute on Core 0 at a time, 4 tasks take exactly $4x$ as long as 1 task.
- **Conclusion:** Adding concurrency to a CPU-saturated single core provides zero throughput benefit; it only increases latency.

4.1.2 2. Disk Contention (I/O Scaling)

Figure 5(b) shows that I/O performance degrades non-linearly.

- **Theory:** Unlike the CPU, the disk is a stateful physical device. Random write requests from 8 workers saturate the disk controller's command queue. The overhead includes not just software context switching, but physical seek latency and rotational delay (if HDD) or controller queuing (if SSD).
- **Conclusion:** The bottleneck here is strictly hardware. Software optimizations (Threads vs Processes) cannot overcome the physical limits of the block device.

4.1.3 3. Architectural Failure: The Thread "Crash"

Figure 5(c) illustrates a critical failure in the Thread model at $N = 5$.

- **Process Model (Robust):** Usage scales linearly to ≈1GB. Each process has its own isolated 4GB (32-bit) or 256TB (64-bit) virtual address space.
- **Thread Model (Fragile):** Usage drops sharply at $N = 5$.
- **Root Cause Analysis:** All threads in a process share the *same* virtual address space. Attempting to allocate $5 \times 220MB$ ($> 1.1GB$) of contiguous heap space likely triggered a Virtual Memory exhaustion error or hit the `ulimit -v` boundary. The subsequent threads failed to allocate memory and exited, causing the aggregate memory usage to plummet.
- **Implication:** Threads are unsuitable for workloads requiring massive, scalable memory allocation on resource-constrained systems.

5 Conclusion

This study empirically validates the following OS principles:

1. **Concurrency $\neq$ Parallelism:** On single-core systems, logical concurrency incurs overhead without execution speedup.
2. **Isolation vs. Efficiency:** Processes provide robust isolation and scalable address spaces (preventing the "Crash" observed in threads) but incur higher setup costs.
3. **Hardware Limits:** For I/O-heavy tasks, the bottleneck is the physical device; the software model is secondary.

6 AI Usage Declaration

I declare that I used Generative AI tools (ChatGPT/Gemini) to assist with the following auxiliary tasks:

- **Visualization:** Generating Python `matplotlib` scripts to visualize CSV logs.
- **Scripting:** Debugging `awk` syntax in bash scripts for parsing `top` output.
- **Formatting:** Refining Latex structure for professional reporting.

All C source code logic, worker function design, and experimental analysis were derived and verified personally.