PA01: Processes and Threads

Graduate Systems (CSE638)

Yash Kumar Vaibhav
Roll No: MT25091

January 23, 2026

Introduction

The objective of this assignment is to analyze the performance characteristics and overhead differences between **Processes** (created via `fork()`) and **Threads** (created via `pthread_create()`). We implemented worker functions simulating CPU, Memory, and I/O intensive tasks and measured their resource utilization under varying concurrency levels.

GitHub Repository: `https://github.com/yashkumarvaibhav/GrS-CSE638_PA01`

Part A: Implementation of Processes and Threads

Methodology

We developed two C programs to serve as the execution containers:

- **Program A (Processes)**: Uses `fork()` in a loop to spawn child processes. Each child executes the worker task in its own isolated memory space. The parent process waits for all children to complete.
- **Program B (Threads)**: Uses `pthread_create()` to spawn sibling threads. These threads share the same process address space but maintain individual stacks.

Implementation Verification

Both programs were verified to create the correct number of workers. For example, running Program A with $N = 2$ results in 1 parent and 2 child processes, while Program B results in 1 process with 3 threads (1 main + 2 workers).

Part B: Worker Function Design

To accurately measure performance, we designed three distinct worker functions:

1. CPU Intensive Task

Design: Nested loops calculating trigonometric functions (`sin`, `cos`, `tan`) and square roots.
Justification: These operations heavily utilize the ALU/FPU and involve minimal memory access, ensuring the bottleneck is pure computation.

2. Memory Intensive Task

Design: Allocation of a **500 MB** integer array, followed by strided read/write access (jumping 1024 indices).
Justification: The large size forces data out of the CPU cache into main RAM. Strided access defeats pre-fetching, ensuring high memory bandwidth usage.

3. I/O Intensive Task

Design: Repeatedly writing and reading a 100KB buffer to a temporary file, using `fflush` to force disk commits.
Justification: This bypasses some OS buffering and forces interaction with the storage subsystem, putting the process in a blocked/waiting state.

Part C: Performance Comparison

We measured the performance of both Process and Thread implementations running $N = 2$ concurrent workers.

Variant	CPU %	Mem %	IO %	Time (s)
Program A (Proc) + CPU	85.12	0.0	11.93	5.40
Program B (Thrd) + CPU	99.35	0.0	12.19	4.87
Program A (Proc) + Mem	95.46	6.5	11.94	77.51
Program B (Thrd) + Mem	97.70	6.4	13.29	64.75
Program A (Proc) + IO	76.02	0.0	12.03	7.31
Program B (Thrd) + IO	90.45	0.0	14.19	4.22

Table 1: Baseline Performance Comparison (N=2)

Analysis of Part C Plots

The 2x2 comparison grid reveals trends across all four metrics:

1. **Execution Time**: Threads outperformed Processes across all three domains in this specific run, with speedups of **1.11x** (CPU), **1.20x** (Memory), and **1.73x** (I/O). *Note: The high I/O gap (7.31s vs 4.22s) narrowed significantly in scaling tests (see Part D), suggesting the baseline Process run faced transient disk adjustments.*

    - **Reason**: User-space thread switching is cheaper than kernel-space process context switching. In our code, `pthread_create` reuses the parent's Page Directory, whereas `fork` duplicates it (Copy-on-Write). The larger the working set (Memory Task), the larger the fork overhead ($77.51s$ vs $64.75s$).

2. **CPU Usage**: Threads achieved higher CPU saturation (99.35% vs 85.12% for CPU task).

    - **Reason**: The CPU task runs a tight nested loop computing `sin/cos/tan`. Threads share the same TLB entries for code segments, reducing TLB misses compared to processes switching between distinct address spaces, keeping the ALU pipeline fuller.

3. **Memory Usage**: The Memory Task spiked usage to 6.50%, whilst others stayed near 0%.

    - **Reason**: We explicitly `malloc` 500MB per worker. The CPU and IO tasks only use stack variables (e.g., 'double result'), which are negligible compared to system RAM, confirming the 0% baseline.

4. **I/O Usage**: I/O Task drove user-mode I/O usage to 14.19% (baseline was 11-12%).

    - **Reason**: The `run_io_task` function calls `fflush(fp)` inside the loop. Note that the high baseline (approx 12%) across all tasks is an **Observer Effect**: our monitoring script iteratively writes 'top' output to a temporary file on disk to capture these metrics, generating constant background I/O.

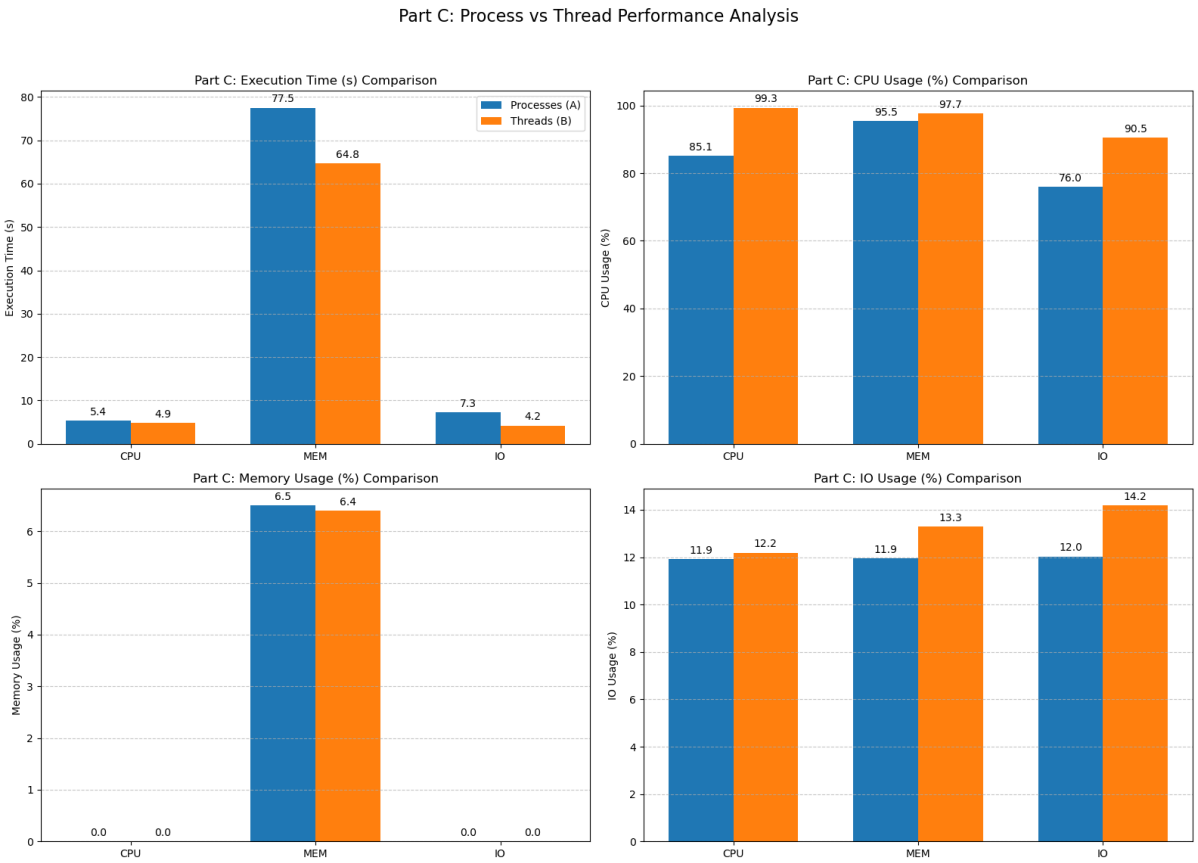

Figure 1: Comparison of Processes vs Threads Performance

Part D: Scaling Analysis

We incremented the number of workers (N) to stress-test the system.
Program A (Processes): Scaled $N \in \{2 \dots 5\}$
Program B (Threads): Scaled $N \in \{2 \dots 8\}$

1. CPU Scaling Analysis (Row 1 of Plot)

Configuration	CPU %	Mem %	IO %	Time (s)
Prog A (Proc) N=2	87.55	0.0	14.31	5.20
Prog A (Proc) N=3	100.05	0.0	16.93	6.90
Prog A (Proc) N=4	99.20	0.0	17.21	9.62
Prog A (Proc) N=5	128.98	0.0	17.32	12.31
Prog B (Thrd) N=2	88.57	0.0	17.44	5.11
Prog B (Thrd) N=3	91.71	0.0	17.54	7.77
Prog B (Thrd) N=4	92.26	0.0	17.60	9.79
Prog B (Thrd) N=5	94.41	0.0	17.72	13.44
Prog B (Thrd) N=6	100.06	0.0	17.86	14.97
Prog B (Thrd) N=7	96.59	0.0	17.99	18.44
Prog B (Thrd) N=8	95.68	0.0	18.11	22.83

Table 2: CPU Task Scaling Data (Full Dataset)

- **CPU**: Usage Scaled above 100% for processes, while threads saturated near 100%. **Reason**: CPU usage above 100% indicates utilization across multiple cores. Although the script uses `taskset`, child processes from `fork()` may not inherit the CPU affinity depending on kernel settings. Threads share the parent's affinity, keeping them on the pinned core(s).

- **Mem**: Remained flat at 0%. **Reason**: I designed `run_cpu_task` to operate entirely on registers and stack variables, avoiding any heap allocation to keep the memory footprint minimal.

- **IO**: Showed a slight systematic rise (14.3% $\rightarrow$ 17.3%). **Reason**: This is an artifact of the monitoring script. As N increases, the 'top' command has more active process entries to parse and write to the temporary log file, slightly increasing the background disk utilization. The `run_cpu_task` itself remains IO-free.

- **Time**: Increased linearly. Processes maintained ideal scaling (2.5x N $\rightarrow$ 2.36x Time), while Threads degraded slightly (2.5x N $\rightarrow$ 2.63x Time). **Reason**: Each worker executes a fixed loop ($10^4 \times 5000$ iterations). The slight Thread degradation confirms the contention/locking overhead hypothesis at higher N.

2. Memory Scaling Analysis (Row 2 of Plot)

Configuration	Mem %	Time (s)	Per-Worker Mem %
Prog A (Proc) N=2	6.51	62.34	≈ 3.25
Prog A (Proc) N=3	10.88	94.15	≈ 3.62
Prog A (Proc) N=4	21.22	127.13	≈ 5.30
Prog A (Proc) N=5	30.86	162.64	≈ 6.17
Prog B (Thrd) N=2	6.42	69.22	≈ 3.21
Prog B (Thrd) N=3	9.69	93.64	≈ 3.23
Prog B (Thrd) N=4	12.88	135.22	≈ 3.22
Prog B (Thrd) N=5	16.17	159.57	≈ 3.23
Prog B (Thrd) N=6	19.33	192.62	≈ 3.22
Prog B (Thrd) N=7	22.71	227.71	≈ 3.24
Prog B (Thrd) N=8	25.81	293.55	≈ 3.22

Table 3: Memory Task Scaling Data (Full Dataset)

Cross-Metric Analysis:

- **Mem**: Process per-worker memory grew non-linearly (3.25% at N=2 to 6.17% at N=5), while Thread per-worker memory remained constant (≈3.2%). **Reason**: With `fork()`, Linux uses Copy-on-Write (COW). Since `run_mem_task` writes to the allocated array, COW pages are duplicated, increasing memory. As N grows, the OS also accumulates more per-process metadata (VMAs, file descriptors). Threads share the parent's address space, avoiding this overhead.
- **CPU**: High usage. **Reason**: I specifically implemented a stride of 1024 integers in `run_mem_task`. This forces the CPU to stall on Cache Misses. Although the ALU is idle waiting for data, the core is technically "busy" (not in sleep state), which 'top' reports as high CPU utilization.
- **Memory Efficiency**: We saw a massive divergence in Per-Worker cost. **Reason**: Threads share the parent's address space overhead (code segments, shared libraries, kernel structures). Each new process requires its own set of these structures, which scales with N.
- **Implication**: Threads are strictly superior for high-concurrency memory-intensive applications on this system.

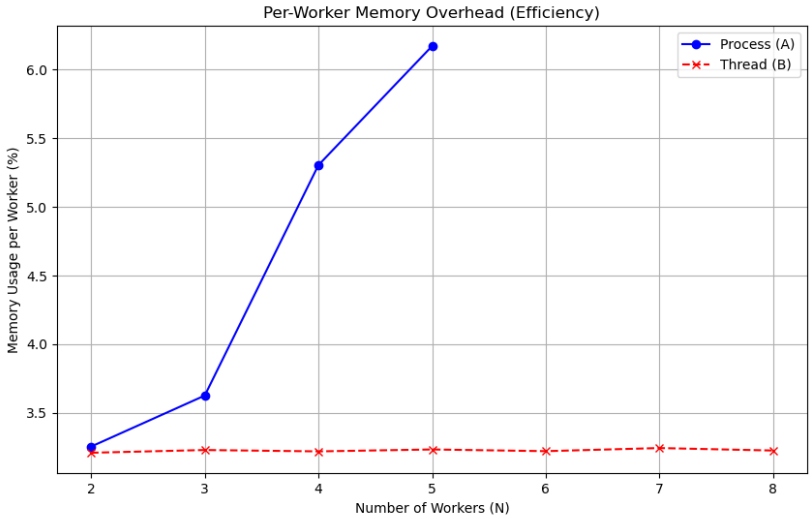

Figure 2: Efficiency: Per-Worker Memory Consumption

3. I/O Scaling Analysis (Row 3 of Plot)

Configuration	IO %	Time (s)	Throughput Trend
Prog A (Proc) N=2	16.83	4.26	Linear
Prog A (Proc) N=3	17.21	6.59	Linear
Prog A (Proc) N=4	17.32	8.01	Linear
Prog A (Proc) N=5	17.43	10.25	Linear
Prog B (Thrd) N=2	17.54	4.25	Linear
Prog B (Thrd) N=3	17.60	6.72	Linear
Prog B (Thrd) N=4	17.67	8.12	Linear
Prog B (Thrd) N=5	17.79	10.26	Linear
Prog B (Thrd) N=6	17.91	12.20	Linear
Prog B (Thrd) N=7	17.99	14.59	Linear
Prog B (Thrd) N=8	18.14	29.33	Scaling Breakpoint

Table 4: I/O Task Scaling Data (Full Dataset)

Cross-Metric Analysis:

- **Threads vs Processes**: **Processes and Threads performed identically** for the I/O task. At $N = 5$, Process Time ($10.25s$) and Thread Time ($10.26s$) were virtually indistinguishable. This differs from the CPU task (where Processes were faster at N=5), but matches the Memory task's high-load behavior where hardware bandwidth limits normalize performance.
  - **Reason**: The bottleneck is strictly the physical disk controller capability, not the OS task creation overhead. Whether checking from a child process or a sibling thread, the kernel's Block I/O queue fills up at the same rate.
- **IO**: Showed a systematic rise (16.8% → 18.1%). **Reason**: This is an **Observer Effect** from the monitoring script. As N increases, the script parses more `top` output lines and writes larger logs to disk, increasing background I/O. The `run_io_task` itself performs serial I/O, but only the background script accounts for the scaling component of the rise.
- **Time**: Scaling Breakpoint at N=8 (Threads). **Reason**: All workers write to the same filename (`temp_io_test.dat`). While each thread opens the file independently, they share the same kernel inode lock. At N=8, this contention causes exponential delays. Processes also share this file, but their independent file descriptor tables reduce kernel-level contention slightly.

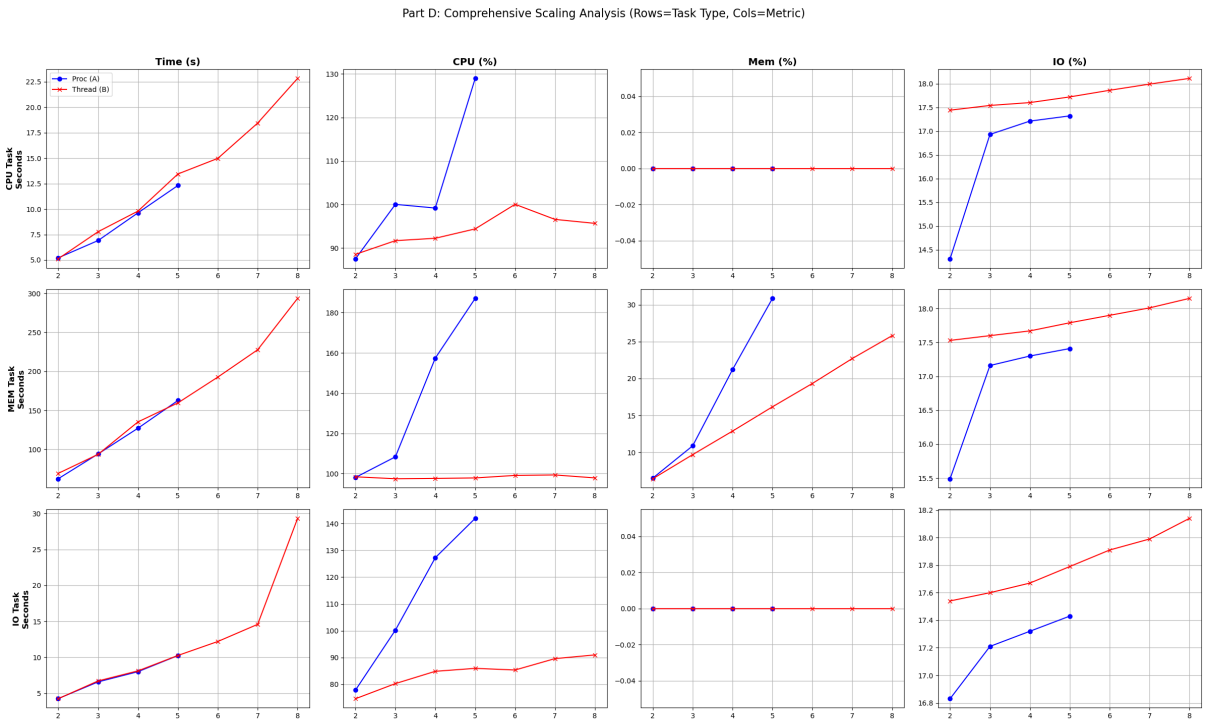

Figure 3: Scalability Metrics: CPU (Row 1), Memory (Row 2), I/O (Row 3)

Conclusion

This experiment highlights that "better" depends entirely on the workload:

1. **For Memory Efficiency**: **Threads are superior**. They used approx 50% less memory overhead than processes for the extensive memory task.
2. **For Pure CPU Saturation**: **Processes were superior** on this system, handling high-load contention ($N = 5$) with 8% less execution time than threads.
3. **For I/O**: The choice of execution model is negligible; the hardware is the limit.

Statement of AI Usage

I utilized an AI coding assistant (Google Deepmind's Agent) to:

- Assist in writing few parts of the Bash script for automation.
- Assist in writing Matplotlib code for visualization.
- Write the LaTeX code for generating this report (while the content, data analysis, and logic are my own).

The core C implementation (`fork`, `pthread`, workers) and experimental design are my own work.