PA01 - PROCESSES AND THREADS

PART A: Program Implementation

- Program A (**Multi-Process**):
  Uses the “fork()” system call to create multiple child processes. Each child process executes concurrently and operates in its independent memory space.
- Program B (**Multi-Threaded**):
  Uses POSIX threads (pthread) to create concurrent execution. All threads run in same process and share the same memory space.

PART B: Worker Function Implementation

Three different worker functions are implemented to simulate:

- CPU-intensive
- Memory-intensive
- I/O-intensive

The workers are used by both programs, A and B, in part A for evaluating system behaviour under different workloads.

Loop Count Calculation

Roll No: MT25190
Last digit:0
Loop count= 9*1000=9000 iterations (as the last digit is 0 switch it with 9)

- **CPU-intensive worker (worker_cpu):**
  This performs *prime number computation* using trial division. The numbers are checked by dividing each number by smaller number up to its square root, requiring a lot of calculations. It does not need disk operation or large amounts of memory.
  A variable (*volatile*) is used so that the compiler does not skip the calculation making it a CPU-bound workload.
- **Memory-intensive worker (worker_mem):**
  The memory-intensive worker puts load on the system’s memory by repeatedly allocating and using large chunks of memory. In each iteration, it allocates a 10 MB buffer, fills it with data, and then goes through the buffer several times to simulate heavy memory usage.
  To keep memory usage high, several buffers are kept in memory and reused in rotation. This causes high memory consumption and frequent memory access. CPU usage remains high because reading and copying data in memory requires continuous CPU work. All operations occur in RAM, with no disk I/O involved.
- **I/O-intensive worker (worker_io):**
  This repeatedly writes data to a file and reads it back from disk. In each iteration, it writes a 1 MB buffer several times and sometimes forces the data to be saved to disk using *fsync()*.
  The same buffer is reused to avoid extra memory usage. Because disk operations are slow, the program spends most of its time waiting for I/O to complete. Memory usage stays low, but execution time increases due to disk delays, making this a purely I/O-bound workload.

PART C: Performance Analysis

Program+Function	CPU%	Mem(KB)	IO(KB/s)	Time(s)
A+cpu	200.00	3652	0.14	3m32.982
A+mem	200.00	413652	0.20	2m27.430
A+io	200.00	6212	0.52	3m50.558
B+cpu	210.00	2020	0.22	3m34.654
B+mem	210.00	411540	0.58	2m19.754
B+mem	180.00	4068	0.64	3m20.037

- **A+CPU (Process-based, CPU-intensive)**
  Result: CPU = 200.00%, Memory = 3652 KB, IO = 0.14 KB/s, Time = 3m32.982s
  Analysis: The CPU usage is approximately 200%, meaning CPU is being fully utilized. Memory usage is low showing that A program performs calculations only and not store them anywhere. IO is almost zero that is expected for task that are computation heavy.

- **A+Memory (Process-based, Memory-intensive)**
  Result: CPU = 200.00%, Memory = 413652 KB, IO = 0.20 KB/s, Time = 2m27.430s
  Analysis: There is use of large amount of memory, with usage above 400MB, showing that the program puts heavy pressure on the memory. CPU usage is high as copying data in memory needs continuous CPU. IO is still very low which means complete work happens on RAM and CPU not through file operation. The program finishes faster than CPU-intensive because its speed is limited by how fast data can be moved showing that the system's memory speed matters more than the processor's computing power.

- **A+IO (Process-based, IO-intensive)**
  Result: CPU = 200.00%, Memory = 6212 KB, IO = 0.52 KB/s, Time = 3m50.558s
  Analysis: The memory usage here is low, showing that it does not store large amounts of data in RAM. Even though the amount of data written to disk per sec in not that high, the execution time is high because the program has to wait. Multiple processes add some extra overhead, because the system has to switch between processes while they are waiting for disk I/O.

- **B+CPU (Thread-based, CPU-intensive)**
  Result: CPU = 210.00%, Memory = 2020 KB, IO = 0.22 KB/s, Time = 3m34.654s
  Analysis: In thread-based, CPU is used more efficiently than in process-based. THis happens because threads are lighter than processes and have less scheduling overhead. Memory usage is low because threads have access to the same memory space. As it is CPU-intensive, the IO is very less. The execution time is little higher than the process-based, which shows that even the overhead is reduced the performance is limited by the number of available CPU cores.

- **B+Mem (Thread-based, Memory-intensive)**
  Result: CPU = 210.00%, Memory = 411540 KB, IO = 0.58 KB/s, Time = 2m19.754s
  Analysis: There is a large amount of memory usage which is expected from memory intensive workload. CPU usage remains high because copying data requires continuous CPU work. The execution time is

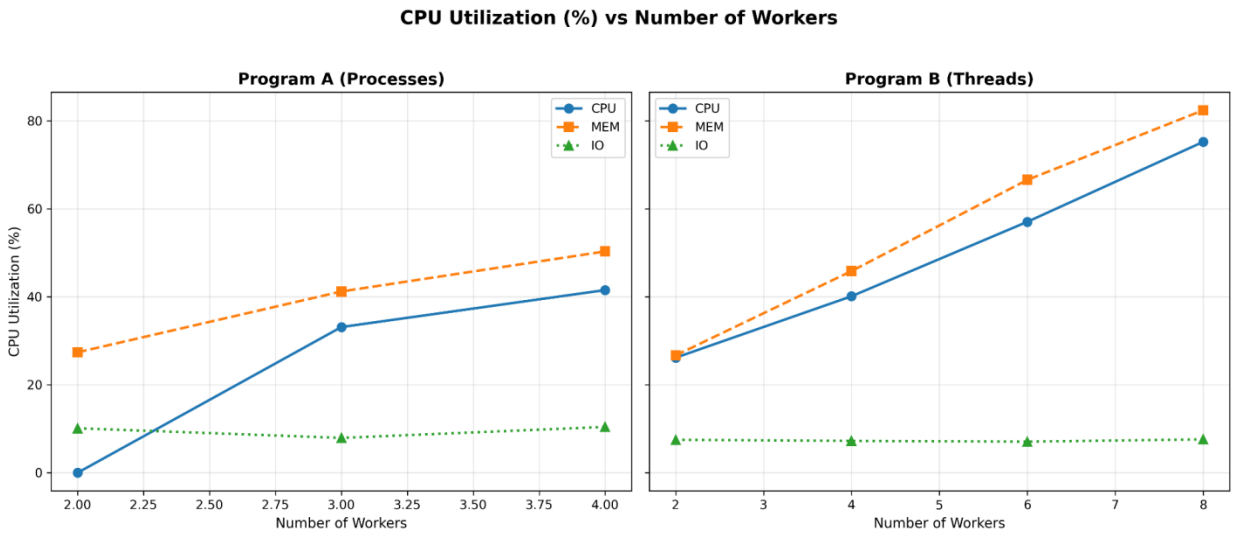
less as compared to program A memory-intensive workload because threads share the same memory space leading to reduced overhead that comes from using separate processes.

- **B+IO (Thread-based, IO-intensive)**
  Result: CPU = 180.00%, Memory = 4068 KB, IO = 0.64 KB/s, Time = 3m20.037s
  Analysis: This thread-based IO workload finishes faster than the process-based, which shows that threads handle waiting for disk operations efficiently. In this when one thread is waiting for I/O, other threads can use CPU, that is way the CPU usage is comparatively low. Memory usage is also low because much data is not stored in RAM. Shorter execution time shows that thread-based execution is much better for I/O workload than using process-based.

PART D: Scaling Analysis

This part analyse how performance changes as the number of workers increases.

Metric 1: CPU Utilization vs Number of Workers



Program A:

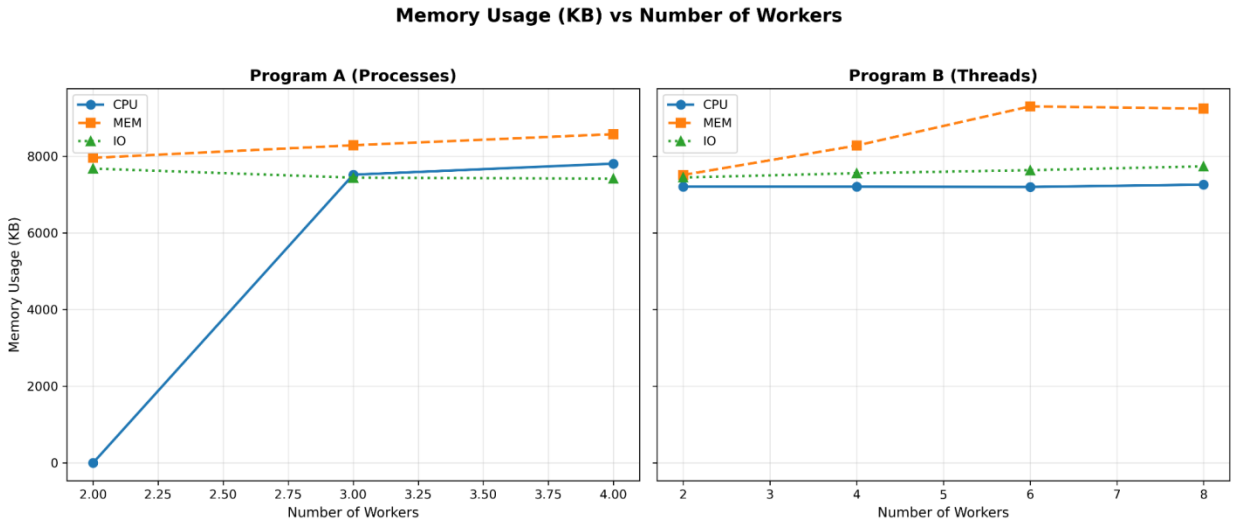
- CPU-workload: CPU utilization increases with increase in number of workers, because more processes uses more CPU cores, there is also some overhead from managing separate processes.
- MEM-workload: This has higher CPU usage than CPU workload. As memory intensive require a number of CPU Cycle for memory allocation/deallocation, memory copying operations.This shows that memory bound does not mean that the CPU is idle in multi process
- IO-workload: Remains almost flat irrespective of the number of workers. IO-bound process spend most of the time in blocked waiting, so the process usage is minimal. CPU is only used for I/O initiation and completion handling.

Program B:

- CPU-workload: increases smoothly with increase in number of workers. This is much higher than program A as threads have lower overhead,more efficient scheduling and can keep all CPu busy with less overhead

- MEM-workload: This has highest CPU utilization in all number of workers. In threaded program, memory operation become CPU-intensive due to tight memory access loop
- IO-workload: Similar to program A, I/O bound work keeps threads blocked. The flat line indicates the increasing the number of workers does not resolve the problem of I/O bottleneck.

Metric 2: Memory Usage vs Number of Workers



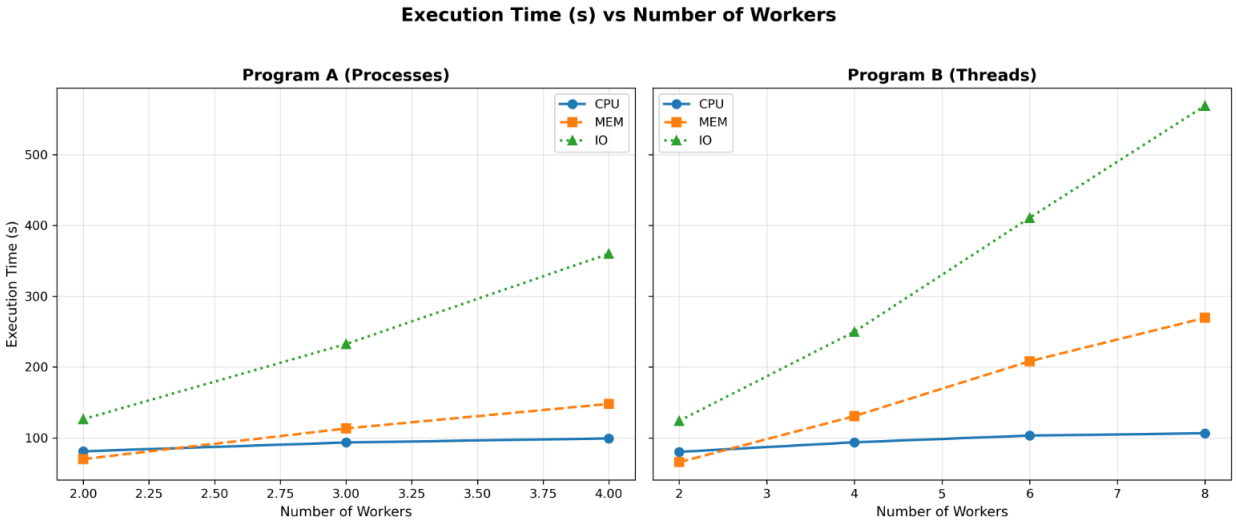
Program A:

- CPU-workload: the memory usage is high as each process has its own memory space
- MEM-workload and IO-workload: Relatively flat with increasing worker count. The memory buffers are pre-allocated. As each process has independent memory buffer size don't increase with increase in number of workers.

Program B:

For all type of workloads the graph stays almost flat or there is a slight increase in memory usage. Threads share the same memory space, so adding new workers barely uses extra memory .

Metric 3: Execution Time vs Number of Workers



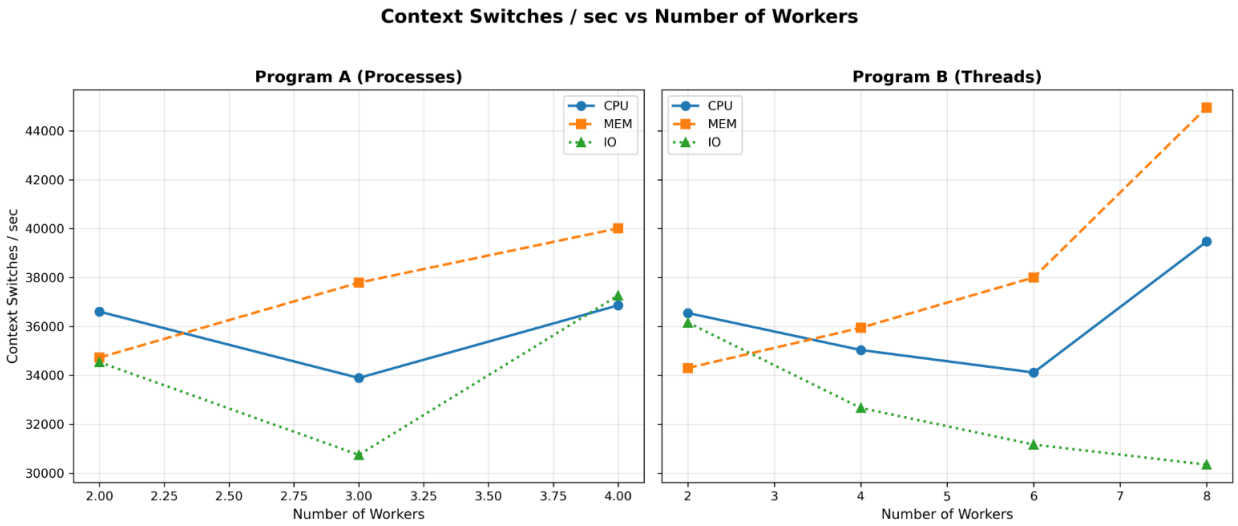
Program A:

- CPU-workload: this increases slowly because creating new processes and switching between them takes some time
- MEM-workload: linear growth in the usage with increase in workers because of severe memory bus congestion. The processes compete for memory bandwidth.
- IO-workload: Execution time increases drastically with increase in number of workers because multiple processes issuing I/O create contention and disk scheduler overhead increases the queue length

Program B:

- CPU-workload: there is very slight increase in the execution time with increase in workers. Threads are efficient for CPU work, but too many threads on few cores causes slowdown.
- MEM-workload: Graph shows significant increase in execution time. This happens because thread are mostly busy waiting for the locks and accessing the shared memory instead of doing actual work.
- IO-workload: the execution time increases drastically which is the worst case. Threads are added to queue to access the shared resources leading to lock contention. Threads are more terrible than processes for I/O scaling.

Metric 4: Context Switches vs Number of Workers



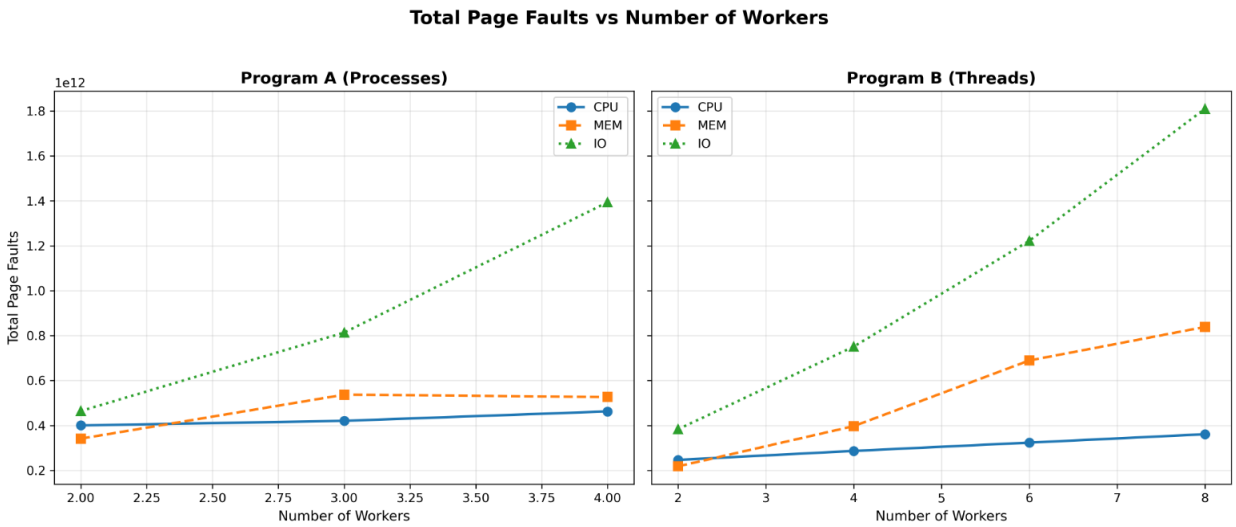
Program A:

- CPU-workload: with 2 process, there is frequent context switching. At 3 processes, system finds more optimal scheduling pattern reducing context switches. At 4 processes, there is increased contention for core which leads to increased context switches.
- MEM-workload: The graph keeps going up steadily. Each process needs its own memory space , as the worker count increase process compete for memory requiring the OS to switch between processes frequently.
- IO-workload:The graph shows V-shaped pattern. At 2 processes there is moderate blocking, At 3 processes, the process spend more time waiting rather than switching. At 4 processes, increases I/O contention forces more context switches as the scheduler manages ,multiple waiting processes.

Program B:

- CPU-workload: Context switching decreases till 6 threads but increases at 8 threads. As thread share the same address space, manking context switch cheaper. WIth 2-6 threads, the context switching decreases and the system becomes more balanced. At 8 threads, CPU oversubscription is caused leading to increased context switching.
- MEM-workload: there is dramatic increase with worker count. Thread share the memory leading to memory contention. As the thread count increases, they fight for the same memory space
- IO-workload: there is a consistent decrease in switches. Unlike processes, threads handle I/O waits more efficiently because the OS can keep other threads in the same process running and as more threads wait concurrently there is reduced overhead.

Metric 5: Total Page Faults vs Number of Workers



Program A:

- CPU-workload: Slight increase with number of workers. CPU intensive work has good locality of reference. Page fault occurs during initial loading of process code and data. The increase with number of worker reflects each process loads its page independently.
- MEM-workload: there is moderate increase in no of page faults, Memory intensive operation cause page fault through each process faulting independently (no sharing).
- IO-workload: There is steep exponential increase wth worker count leading to highest page fault rate.This indicates severe memory pressure from I/O buffers. This is because disk I/O delays cause pages to be removed and reloading on the next access. Also each process has separate page tables, so shared files are faulted multiple times.

Program B:

- CPU-workload: There is a slight increase but lower than program A. As threads have shared memory space so pages are faulted once for all threads.
- MEM-workload: It increases with worker count but it is better than processes. As shared memory means less duplication but threads compete for cache space
- IO-workload: There is a massive increase, making it the worse than processes. Threads fighting over shared I/O buffers causes cache thrashing, invalidate each other's data.
  Threads are not optimal for I/O intensive work when worker count increases.

AI Declaration

I used AI tools as learning the system commands and their syntax.
I used AI tools to generate the code for the worker functions based on my logic
I reviewed all the generated code snippets and integrated them for proper functioning.
The Makefile, Shell script has been generated using AI tools.
The analysis, report and results have been done by me completely.

Github Repository

Shagun-Kaler/GRS_PA01