

# CN Assignment-2

Shobhit Raj (2022482)

Shrutya Chawla (2022487)

## 1 Question 1

In this question, we implemented a client-server architecture using TCP sockets in C, enabling the server to provide information about the top two CPU-consuming processes to multiple clients concurrently.

### 1.1 Server Side

The server application is designed to handle multiple concurrent client connections through the use of threads. The key components of the server-side implementation are as follows :-

**Socket Creation:** A TCP socket is created using the `socket()` system call, setting up the server for incoming connections.

**Binding and Listening:** The server binds the socket to a specified port (8080) and listens for incoming connections, with a backlog of up to 3 queued connections.

**Multithreading:** Upon accepting a new client connection, the server spawns a new thread to handle the client's request. The thread continues to process the client's connection while the server remains available to accept new connections.

**Process Information Retrieval:** The server retrieves CPU usage information for processes using the `/proc` filesystem. It reads `/proc/[pid]/stat` to gather details such as the process name, PID, user CPU time, and kernel CPU time. It identifies the top two CPU-consuming processes by comparing their total CPU times (user + kernel) and stores this information for transmission. For this part, we referred to total-process-cpu-usage and `proc/pid/stat`. The information is extracted from the 52 fields in the `/proc/[pid]/stat` file, where field 1 corresponds to the PID, field 2 to the process name, and fields 14 and 15 to user and kernel CPU time, respectively.

**Response to Clients:** When a client requests the top two processes, the server formats the collected information and sends it back to the client.

**Shutting down the Server:** A signal handling mechanism has been implemented to gracefully shut down the server when Ctrl+C is pressed. This is achieved by intercepting the SIGINT signal and triggering a shutdown of the server socket, breaking out of the blocking `accept()` call, and allowing the server to terminate cleanly.

### 1.2 Client Side

The client application is designed to initiate multiple concurrent connection requests to the server. Its implementation details are :-

**Socket Creation:** Each client thread creates a TCP socket to connect to the server.

**Connection Establishment:** The client connects to the server using the server's IP address and port number. A new socket is created with 4 tuples (server IP, server listening port, client IP, client port) which is used to identify this particular socket connection. If the connection fails, appropriate error messages are displayed.

**Concurrent Requests:** The number of concurrent client connections is passed as a command-line argument. For each connection, a new thread is created, and the thread's function sends a request for the top two CPU-consuming processes.

**Response Handling:** After sending the request, each client thread reads the server's response and displays the information received about the top two processes.

### 1.3 Taskset Usage

The `taskset` command is used to set CPU affinity for processes, allowing them to execute on specified CPU cores. For this part, we referred to `taskset-linux-command`. CPU affinity can also be set programmatically using the `sched_setaffinity` system call, but as mentioned in the assignment question, we used `taskset` for executing the `./server` and `./client` applications. To run the server and client with CPU affinity, we can use the following commands :-

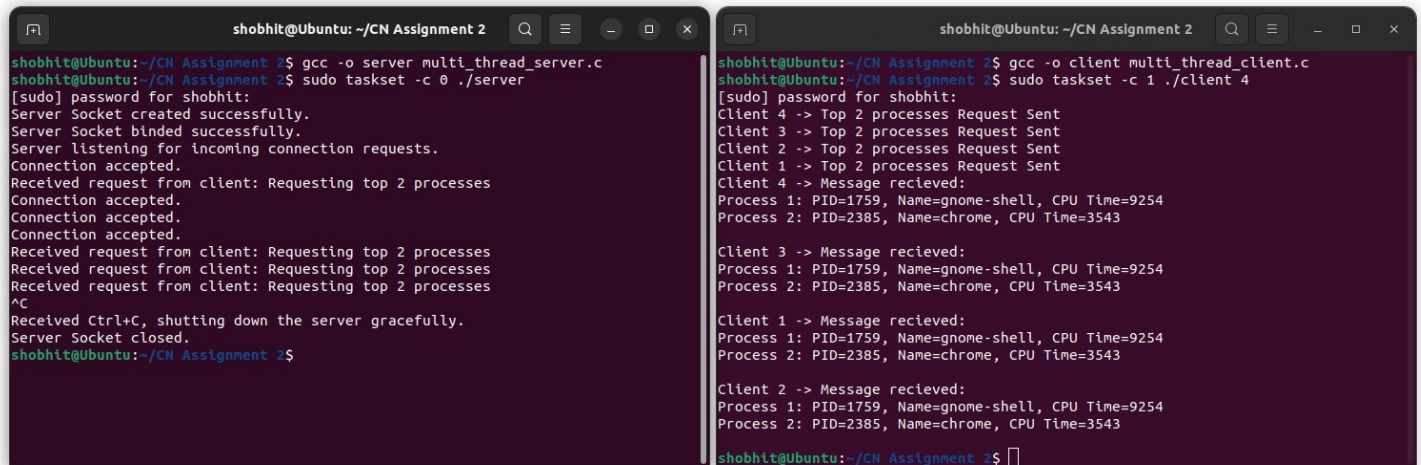
- Start the server on CPU core 0:

```
sudo taskset -c 0 ./server
```

- Start the client on CPU core 1, requesting 4 concurrent connections:

```
sudo taskset -c 1 ./client 4
```

Execution of the Server and Client Applications in Separate Terminals :-



```
shobhit@Ubuntu: ~/CN Assignment 2
shobhit@Ubuntu:~/CN Assignment 2$ gcc -o server multi_thread_server.c
shobhit@Ubuntu:~/CN Assignment 2$ sudo taskset -c 0 ./server
[sudo] password for shobhit:
Server Socket created successfully.
Server Socket binded successfully.
Server listening for incoming connection requests.
Connection accepted.
Received request from client: Requesting top 2 processes
Connection accepted.
Connection accepted.
Received request from client: Requesting top 2 processes
Received request from client: Requesting top 2 processes
Received request from client: Requesting top 2 processes
^C
Received Ctrl+C, shutting down the server gracefully.
Server Socket closed.
shobhit@Ubuntu:~/CN Assignment 2$

shobhit@Ubuntu: ~/CN Assignment 2
shobhit@Ubuntu:~/CN Assignment 2$ gcc -o client multi_thread_client.c
shobhit@Ubuntu:~/CN Assignment 2$ sudo taskset -c 1 ./client 4
[sudo] password for shobhit:
Client 4 -> Top 2 processes Request Sent
Client 3 -> Top 2 processes Request Sent
Client 2 -> Top 2 processes Request Sent
Client 1 -> Top 2 processes Request Sent
Client 4 -> Message recieved:
Process 1: PID=1759, Name=gnome-shell, CPU Time=9254
Process 2: PID=2385, Name=chrome, CPU Time=3543

Client 3 -> Message recieved:
Process 1: PID=1759, Name=gnome-shell, CPU Time=9254
Process 2: PID=2385, Name=chrome, CPU Time=3543

Client 1 -> Message recieved:
Process 1: PID=1759, Name=gnome-shell, CPU Time=9254
Process 2: PID=2385, Name=chrome, CPU Time=3543

Client 2 -> Message recieved:
Process 1: PID=1759, Name=gnome-shell, CPU Time=9254
Process 2: PID=2385, Name=chrome, CPU Time=3543
shobhit@Ubuntu:~/CN Assignment 2$
```

Figure 1: Demonstrating Concurrent Connections and CPU Usage Retrieval

## 2 Question 2

For this part, we are analyzing the performance of a TCP client-server using the perf tool in a WSL environment. Since taskset and perf are typically native to Linux, we used sudo in WSL to gain the necessary permissions and assign CPU affinity for our tests. The command we ran includes metrics such as task clock, context switches, cache misses, and CPU cycles, which were selected after running perf list to choose the most relevant counters for performance evaluation.

Commands used :-

- For measuring the performance of the server on CPU core 0:

```
sudo taskset -c 0 /usr/lib/linux-tools/5.15.0-122-generic/perf stat -e task-clock,  
context-switches,cpu-migrations,cpu-cycles,cpu-clock,page-faults,instructions,  
bus-cycles,cache-misses,branch-instructions,branch-misses,duration_time  
./multi_thread_server
```

- For measuring the performance of the client on CPU core 1, requesting 4 concurrent connections:

```
sudo taskset -c 1 /usr/lib/linux-tools/5.15.0-122-generic/perf stat -e task-clock,  
context-switches,cpu-migrations,cpu-cycles,cpu-clock,page-faults,instructions,  
bus-cycles,cache-misses,branch-instructions,branch-misses,duration_time  
./multi_thread_client 4
```

This setup allowed for detailed insights across various server implementations (single-threaded, concurrent, and select-based).

For developing the socket programming codes for this assignment, we drew inspiration from tutorial sessions provided by our teaching assistants (TAs), where sample codes were discussed and uploaded. Our implementation reflects a deep understanding of those foundational concepts while incorporating our own enhancements and adjustments to meet the specific requirements of the assignment.

## 2.1 (a) Single-threaded TCP Server-Client

```
Performance counter stats for './single_thread_server':

    2.15 msec task-clock                #    0.000 CPUs utilized
      10      context-switches          #    4.663 K/sec
       0      cpu-migrations            #    0.000 /sec
  7200631     cpu-cycles                #    3.358 GHz
    2.14 msec cpu-clock                #    0.000 CPUs utilized
       62     page-faults              #   28.910 K/sec
  6542420     instructions              #    0.91  insn per cycle
   69480      bus-cycles                #   32.398 M/sec
   36592      cache-misses
  1321288     branch-instructions       #   616.100 M/sec
   20715      branch-misses            #    1.57% of all branches
5507541483 ns  duration_time          # 2568.097 G/sec

5.507541483 seconds time elapsed

0.002795000 seconds user
0.000000000 seconds sys
```

Figure 2: Perf Stat output for Single-Threaded TCP Server

### Server Analysis

- **Task Clock (2.15 msec):** The task-clock indicates that the single-threaded server ran for approximately 2.15 milliseconds. Given that this is a single-threaded implementation, the relatively short task time indicates that the server handled the requests quickly and efficiently.
- **CPU Utilization (0.000 CPUs utilized):** This suggests that the server's CPU usage was quite minimal, likely because it was in a blocked state waiting for client requests during most of the execution. The 0.000 CPUs utilized may indicate that the task was too short to significantly register utilization on the CPU, especially since there are short durations of active work.
- **Context Switches (10):** There were 10 context switches, which means the kernel switched the CPU from one process or thread to another 10 times during the server's execution. This low number is expected in a single-threaded server, as context switching overhead would be minimal due to the lack of concurrent operations.
- **CPU Cycles (7,200,631):** The server consumed 7.2 million CPU cycles during its execution. This number is relatively high compared to the 2.15 milliseconds task time, which may indicate that while the server wasn't fully utilizing CPU time, it still executed a number of instructions during its brief active period.
- **Instructions (6,542,420) and Instructions per Cycle (0.91 IPC):** The server executed 6.54 million instructions, and the instructions per cycle (IPC) was 0.91. A value close to 1 IPC suggests that the server efficiently used the CPU, as it was close to executing one instruction per CPU cycle.
- **Branch Misses (20,715):** There were 20,715 branch misses out of 1.32 million branch instructions, leading to a branch miss rate of 1.57%. This is a reasonably low branch miss rate, suggesting that the CPU's branch predictor was working well, minimizing costly pipeline stalls.
- **Cache Misses (36,592):** The server experienced 36,592 cache misses, which is expected in a lightweight, single-threaded application. This indicates that most memory accesses were resolved without needing to fetch from slower memory levels.
- **Duration Time (5.51 seconds):** This number represents the total duration from the start to the end of the process. The server's elapsed time was 5.51 seconds, which includes time the server spent waiting for requests from the client. The high duration time compared to the task clock suggests the server was idle or waiting for client connections.

```

Performance counter stats for './single_thread_client 4':

    0.84 msec task-clock                #    0.115 CPUs utilized
      13      context-switches         #   15.496 K/sec
       0      cpu-migrations            #    0.000 /sec
  2757816     cpu-cycles                #    3.287 GHz
    0.84 msec cpu-clock                 #    0.115 CPUs utilized
      71      page-faults              #   84.635 K/sec
 1353678     instructions               #    0.49 insn per cycle
   25068     bus-cycles                 #   29.882 M/sec
   18524     cache-misses
  281235     branch-instructions        #  335.243 M/sec
   12723     branch-misses             #    4.52% of all branches
 7301323 ns   duration_time            #    8.703 G/sec

0.007301323 seconds time elapsed

0.001125000 seconds user
0.000000000 seconds sys

```

Figure 3: Perf Stat output for Single-Threaded TCP Client

### Client Analysis

- **Task Clock (0.84 msec):** The single-threaded client had a task clock of 0.84 milliseconds, which is significantly shorter than the server's overall duration. This short duration shows the client spent minimal time processing, which aligns with its role of sending requests and receiving responses.
- **CPU Utilization (0.115 CPUs utilized):** The client utilized about 11.5% of a CPU during its execution, which is a bit higher than the server's utilization. This suggests the client was more active relative to its execution time, likely due to sending multiple requests or waiting for responses from the server.
- **Context Switches (13):** The client experienced 13 context switches, slightly more than the server. This could be attributed to the client waiting on responses from the server, requiring the kernel to switch between tasks.
- **CPU Cycles (2,757,816):** The client consumed 2.76 million CPU cycles, lower than the server's 7.2 million cycles. This is expected since the client typically does less processing than the server.
- **Instructions (1.35 million) and Instructions per Cycle (0.49 IPC):** The client executed 1.35 million instructions with an IPC of 0.49. This lower IPC value indicates that the client was less efficient in using the CPU cycles compared to the server. The client likely encountered more stalls or delays, possibly due to waiting for responses or interacting with the network.
- **Branch Misses (12,723):** The client experienced 12,723 branch misses, leading to a branch miss rate of 4.52%. This rate is significantly higher than the server's branch miss rate of 1.57%, indicating that the client had more difficulty predicting branches efficiently, potentially due to the interactive nature of waiting for server responses.
- **Cache Misses (18,524):** The client experienced 18,524 cache misses, roughly half of what the server experienced. This makes sense, as the client likely performed fewer memory-intensive operations compared to the server.
- **Duration Time (7.30 milliseconds):** The client's total duration was 7.30 milliseconds, shorter than the server's but relatively long compared to its task clock. This suggests that the client spent time waiting for responses from the server.

## 2.2 (b) Multi-threaded (Concurrent) TCP Server-Client

```
Performance counter stats for './multi_thread_server':

      2.95 msec task-clock                #    0.000 CPUs utilized
        11      context-switches         #    3.743 K/sec
         0      cpu-migrations            #    0.000 /sec
    7818305     cpu-cycles                #    2.660 GHz
        2.93 msec cpu-clock              #    0.000 CPUs utilized
         78      page-faults             #   26.541 K/sec
    6988975     instructions              #    0.89  insn per cycle
        91063    bus-cycles              #   30.986 M/sec
        45160    cache-misses
    1418210     branch-instructions       #  482.581 M/sec
        23811    branch-misses           #    1.68% of all branches
  11554272933 ns    duration_time        # 3931.630 G/sec

 11.554272933 seconds time elapsed

 0.003538000 seconds user
 0.000000000 seconds sys
```

Figure 4: Perf Stat output for Multi-Threaded TCP Server

### Server Analysis

- **Task Clock (2.95 msec):** This represents the time during which the server was actively using the CPU. The low task clock suggests that the multi-threaded server ran for a very short period of time, possibly due to quick handling of the requests.
- **CPU Utilization (0.000 CPUs utilized):** This metric indicates that the server used negligible CPU resources. The result shows the server efficiently handled multiple requests in parallel, not saturating the CPU.
- **Context Switches (11):** Context switching occurs when the system switches between threads or processes. The 11 context switches are relatively few, which is good for performance as context switching can introduce overhead. This low number reflects the efficiency of the thread handling.
- **CPU Cycles (7,818,305 cycles) & Instructions (6,988,975 instructions):** The CPU executed nearly 7 million cycles to complete the work. With an instruction count close to 7 million, this suggests high computational efficiency.
- **Cycles per Instruction (CPI) (0.89):** A CPI less than 1.0 indicates that the processor is completing more than one instruction per cycle. This is a sign of efficient use of the CPU pipeline, implying the server is well optimized for multi-threaded workloads.
- **Cache Misses (45,160):** Cache misses introduce latency because the CPU has to fetch data from the main memory rather than from the faster cache. The number is relatively low given the number of instructions, which suggests that the server's code and data are cache-friendly.
- **Branch Instructions (1,418,210) & Branch Misses (23,811):** With over 1.4 million branch instructions and only about 1.68% of them resulting in branch misses, the server's branch prediction is efficient. A low branch miss rate helps reduce performance penalties from mispredictions.
- **Page Faults (78):** Page faults occur when the system accesses memory that is not loaded in physical RAM. A count of 78 page faults is quite low and would have minimal impact on performance.
- **Duration Time (11.55 seconds):** This likely represents the total time spent during the execution of the program (including idle or wait times). It might indicate waiting for client requests or time spent managing threads.



```

Performance counter stats for './multi_thread_client 4':

    1.38 msec task-clock                #    0.170 CPUs utilized
      20      context-switches          #   14.523 K/sec
      0       cpu-migrations            #    0.000 /sec
   3954970    cpu-cycles                #    2.872 GHz
      1.37 msec cpu-clock               #    0.170 CPUs utilized
      92      page-faults              #   66.805 K/sec
   1772091    instructions              #    0.45 insn per cycle
   36922     bus-cycles                 #   26.810 M/sec
   25117     cache-misses               #
   365721    branch-instructions        #  265.564 M/sec
   18241     branch-misses              #    4.99% of all branches
   8100202 ns duration_time            #    5.882 G/sec

0.008100202 seconds time elapsed

0.001762000 seconds user
0.000000000 seconds sys

```

Figure 5: Perf Stat output for Multi-Threaded TCP Client

### Client Analysis

- **Task Clock (1.38 msec):** The client's active CPU usage is also very low, indicating that it performed its operations quickly and didn't require much CPU time.
- **CPU Utilization (0.170 CPUs utilized):** Unlike the server, the client used more CPU resources relative to its task clock, reflecting the fact that the client spent more time actively sending and receiving requests from the server.
- **Context Switches (20):** The client experienced slightly more context switches than the server, possibly due to its communication with the server. However, this number is still quite low and indicates good performance.
- **CPU Cycles (3,954,970 cycles) & Instructions (1,772,091 instructions):** The client ran fewer cycles and instructions compared to the server, as expected given its simpler role in the communication process.
- **CPI (0.45):** With a CPI of 0.45, the client was even more efficient than the server in terms of instructions executed per cycle. This suggests that the client's code was very lightweight and optimized for the tasks it performed.
- **Cache Misses (25,117):** The number of cache misses is low relative to the instruction count, suggesting good data locality in the client's code.
- **Branch Instructions (365,721) & Branch Misses (18,241):** With a higher branch miss rate (around 4.99%), the client's branch prediction is less efficient compared to the server, but given the low number of instructions, the performance impact is likely minimal.
- **Page Faults (92):** A higher page fault count than the server, but still very low, indicating minimal impact from paging.
- **Duration Time (0.0081 seconds):** The client's overall runtime is very short, meaning it performed its task (sending requests and receiving responses) very quickly.

## 2.3 (c) TCP Server-Client using Select

```
Performance counter stats for './server_select':

      2.79 msec task-clock               #    0.000 CPUs utilized
        11      context-switches        #    3.952 K/sec
         0      cpu-migrations           #    0.000 /sec
    7874479     cpu-cycles               #    2.829 GHz
        2.78 msec cpu-clock             #    0.000 CPUs utilized
         62      page-faults            #   22.275 K/sec
    6681679     instructions             #    0.85  insn per cycle
     81531     bus-cycles               #   29.292 M/sec
     39706     cache-misses             #
    1349437     branch-instructions      #   484.825 M/sec
     23199     branch-misses            #    1.72% of all branches
    6304275199 ns duration_time         # 2264.995 G/sec

6.304275199 seconds time elapsed

0.001620000 seconds user
0.001873000 seconds sys
```

Figure 6: Perf Stat output for TCP Server using Select

### Server Analysis

- **Task Clock (2.79 msec):** The task clock for the server is approximately 2.79 milliseconds, indicating the total CPU time spent processing. This is close to the task clocks for the single-threaded and multi-threaded servers, which suggests that the server handled the connections quickly.
- **CPU Utilization (0.000 CPUs utilized):** The server has very minimal CPU utilization, likely because most of the time was spent in a blocking state while waiting for I/O events using select. The 0.000 CPUs utilized shows that the server was mostly idle, waiting for file descriptors to become ready.
- **Context Switches (11):** The 11 context switches indicate that the server context-switched between processes or threads 11 times during execution.
- **CPU Cycles (7,874,479):** The server consumed 7.87 million CPU cycles, which is comparable to the CPU cycles seen in the single-threaded implementation (7.20 million cycles). This suggests that even though the server was idle for much of the time, when it did process requests, it consumed a similar number of cycles as the single-threaded server.
- **Instructions (6,681,679) and IPC (0.85):** The server executed 6.68 million instructions with an IPC (Instructions per Cycle) of 0.85. This is slightly lower than the IPC in the single-threaded case (0.91), indicating a bit more inefficiency in CPU usage. The reduction in efficiency could be due to the server monitoring multiple file descriptors simultaneously, which could introduce more branching or additional checks.
- **Branch Instructions (1,349,437) and Branch Misses (23,199):** There were 1.35 million branch instructions with a 1.72% branch miss rate. This miss rate is comparable to the single-threaded server's branch miss rate of 1.57%, suggesting that branch prediction was still effective even with the added complexity of monitoring multiple file descriptors.
- **Cache Misses (39,706):** The server had 39,706 cache misses, slightly higher than the single-threaded server's 36,592 cache misses. This could be due to the additional overhead introduced by select, which may involve accessing more data structures or managing a larger number of file descriptors.
- **Duration Time (6.30 seconds):** The duration time of the server was 6.30 seconds, indicating that it took a little longer to complete than the single-threaded server (5.51 seconds). This longer duration likely reflects the time spent waiting for client requests or handling I/O events, which is expected in a select-based implementation.



```

Performance counter stats for './client_select 4':

      1.93 msec task-clock               #    0.114 CPUs utilized
         13    context-switches         #    6.739 K/sec
          0    cpu-migrations            #    0.000 /sec
    3975164    cpu-cycles                #    2.061 GHz
         1.92 msec cpu-clock             #    0.114 CPUs utilized
          80    page-faults             #   41.470 K/sec
    1513266    instructions              #    0.38  insn per cycle
     35515    bus-cycles                #   18.410 M/sec
     26981    cache-misses              #
    312132    branch-instructions       #   161.802 M/sec
     16456    branch-misses             #    5.27% of all branches
    16896518 ns  duration_time          #    8.759 G/sec

0.016896518 seconds time elapsed

0.002199000 seconds user
0.000000000 seconds sys

```

Figure 7: Perf Stat output for TCP Client using Select

## Client Analysis

- **Task Clock (1.93 msec):** The task clock for the client was 1.93 milliseconds, longer than the single-threaded client's 0.84 milliseconds. This suggests the client spent more time processing in this case, potentially due to interactions with the select-based server.
- **CPU Utilization (0.114 CPUs utilized):** The client used around 11.4% of a CPU, similar to the single-threaded client's utilization (11.5%). This suggests that the client was moderately active in its communication with the server, but still spent most of its time waiting for responses.
- **Context Switches (13):** The client experienced 13 context switches, the same as in the single-threaded client implementation. This indicates that the client's interaction with the select-based server did not introduce significantly more context-switching overhead.
- **CPU Cycles (3,975,164):** The client consumed 3.98 million CPU cycles, higher than the single-threaded client's 2.76 million CPU cycles.
- **Instructions (1.51 million) and IPC (0.38):** The client executed 1.51 million instructions with an IPC of 0.38, which is lower than the single-threaded client's 0.49 IPC. This lower IPC indicates that the client was less efficient in executing instructions per cycle, likely due to waiting for responses from the select-based server, which could have introduced additional latency or complexity.
- **Branch Instructions (312,132) and Branch Misses (16,456):** The client executed 312,132 branch instructions with a branch miss rate of 5.27%. This branch miss rate is higher than both the single-threaded client's 4.52% and the multi-threaded client's 4.99%. The higher miss rate could reflect more uncertainty in the control flow due to interactions with the select-based server, which may introduce more unpredictable branching behavior.
- **Cache Misses (26,981):** The client experienced 26,981 cache misses, which is higher than the single-threaded client's 18,524 cache misses. This increase in cache misses could be due to additional memory accesses when interacting with the select-based server, possibly related to managing file descriptors or handling I/O events.
- **Duration Time (16.89 milliseconds):** The duration time for the client was 16.89 milliseconds, which is longer than the single-threaded client's 7.30 milliseconds. This longer duration could indicate that the client spent more time waiting for responses from the select-based server, which had to monitor multiple file descriptors and respond accordingly.

## Comparative Insights and Conclusions

The “cpu-migrations” metric indicates how many times a thread has moved from one CPU core to another during execution. In our case, the output shows 0 cpu-migrations across all implementations. This is because of the use of the taskset command, which confines the execution of the server and client applications to a specific CPU core, leading to performance stability and reduced overhead.

The single-threaded approach is straightforward and incurs minimal overhead due to low context switching. However, its scalability is limited, as it can handle only one client at a time, resulting in potentially high latency under load. The long duration times relative to task clocks indicate that most of the time was spent waiting for connections and responses, rather than actively processing requests.

The multi-threaded implementation provides better handling of concurrent connections compared to the single-threaded version, allowing for reduced latency and better utilization of CPU resources. However, it introduces a slight increase in context switching and CPU cycles, indicating some overhead associated with thread management.

The select-based implementation allows a single-threaded server to manage multiple clients efficiently, but it incurs slightly higher latency and CPU cycles than both the single-threaded and multi-threaded versions. Although it effectively minimizes context switches, it suffers from increased cache misses and branch mispredictions, likely due to the complexity of monitoring multiple file descriptors.

The single-threaded approach demonstrated the least latency per operation and most cache efficiency but cannot efficiently handle multiple concurrent clients & quickly becomes a bottleneck under higher loads. The multi-threaded approach improved responsiveness and scalability but introduced some additional overhead with higher CPU cycles and context switches. The select-based implementation allows for handling multiple clients with minimal thread overhead, making it a favorable option for scenarios where scaling is essential without incurring the costs of threading.