

# Computer Networks Assignment 2

Souparno Ghose (2022506) , Stanzin Gyalpo (2022509)

September 30, 2024

## Question 1

Write a client-server socket program in C. The client process and the server process should run on separate VMs (or containers) and communicate with each other. Use “**taskset**” to pin the process to specific CPUs. This helps measure the performance. Your program should have the following features.

## Solution

This solution explains the implementation of a client-server socket program using multi-threading in C. The server is designed to handle multiple concurrent clients using separate threads for each connection. Each client requests and receives the top two CPU-consuming processes from the server.

## Program Overview

The program includes the following key features:

1. The server sets up a TCP socket and listens for incoming client connections.
2. The server handles multiple concurrent clients by creating a new thread for each connection using the **pthread** library.
3. The client initiates a TCP connection and requests the top two CPU-consuming processes from the server.
4. The server gathers this data by reading from the `/proc/[pid]/stat` file system for all running processes.
5. The server sends the requested information (process name, PID, user time, kernel time) back to the client.
6. The client receives the data and prints it to the console, then closes the connection.

## Execution Setup

To run the server and client on separate CPUs, we used the `taskset` command as follows:

- Server: `taskset -c 0 ./server_multithreaded`
- Client: `taskset -c 1 ./client_multithreaded 5`

## Server and Client Outputs

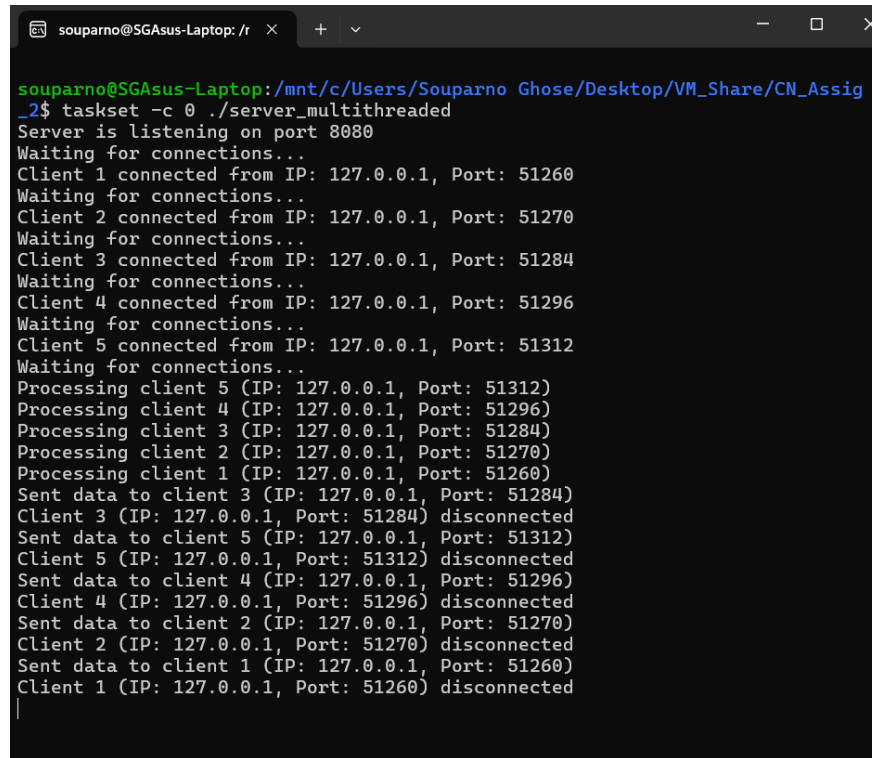
Due to multithreading, the responses from the server to the clients may not be in the same order that the connections were established. This is because each client connection is handled in a separate thread, allowing concurrent processing of multiple requests. This results in an unordered response sequence.

## Why the Top Two CPU Processes are Always the Same for Threads of Clients

The top two CPU-consuming processes tend to remain consistent across threads of clients due to the nature of how CPU usage is measured at a given point in time. When a client requests this information, the server retrieves it from the `/proc` filesystem, providing a snapshot of the system's CPU consumption. If multiple clients make requests in quick succession, the underlying process states may not change significantly, resulting in similar outputs. Additionally, certain processes may consistently exhibit high CPU usage due to their inherent workload, leading to the same processes being reported as the top consumers across multiple concurrent requests.

## Server Output

The server listens for incoming client connections and processes them concurrently. Below is the screenshot showing the server handling five concurrent client connections:

A screenshot of a terminal window titled 'souparno@SGAsus-Laptop: /r'. The terminal shows the execution of a program named 'server\_multithreaded' using 'taskset -c 0'. The server is listening on port 8080. It receives five concurrent connections from IP 127.0.0.1 on ports 51260, 51270, 51284, 51296, and 51312. The server processes these clients in a non-sequential order: client 5, then 4, then 3, then 2, then 1. For each client, it sends data and then reports that the client is disconnected. The output demonstrates that the server is handling multiple clients concurrently.

```
souparno@SGAsus-Laptop:/mnt/c/Users/Souparno Ghose/Desktop/VM_Share/CN_Assig
_2$ taskset -c 0 ./server_multithreaded
Server is listening on port 8080
Waiting for connections...
Client 1 connected from IP: 127.0.0.1, Port: 51260
Waiting for connections...
Client 2 connected from IP: 127.0.0.1, Port: 51270
Waiting for connections...
Client 3 connected from IP: 127.0.0.1, Port: 51284
Waiting for connections...
Client 4 connected from IP: 127.0.0.1, Port: 51296
Waiting for connections...
Client 5 connected from IP: 127.0.0.1, Port: 51312
Waiting for connections...
Processing client 5 (IP: 127.0.0.1, Port: 51312)
Processing client 4 (IP: 127.0.0.1, Port: 51296)
Processing client 3 (IP: 127.0.0.1, Port: 51284)
Processing client 2 (IP: 127.0.0.1, Port: 51270)
Processing client 1 (IP: 127.0.0.1, Port: 51260)
Sent data to client 3 (IP: 127.0.0.1, Port: 51284)
Client 3 (IP: 127.0.0.1, Port: 51284) disconnected
Sent data to client 5 (IP: 127.0.0.1, Port: 51312)
Client 5 (IP: 127.0.0.1, Port: 51312) disconnected
Sent data to client 4 (IP: 127.0.0.1, Port: 51296)
Client 4 (IP: 127.0.0.1, Port: 51296) disconnected
Sent data to client 2 (IP: 127.0.0.1, Port: 51270)
Client 2 (IP: 127.0.0.1, Port: 51270) disconnected
Sent data to client 1 (IP: 127.0.0.1, Port: 51260)
Client 1 (IP: 127.0.0.1, Port: 51260) disconnected
|
```

Figure 1: Server handling multiple clients concurrently

## Client Output

Each client requests the top two CPU-consuming processes from the server. Since the clients are processed in separate threads, the responses are received in no particular order. Below is a screenshot showing the client outputs:

```
souparno@SGAsus-Laptop: /r  × + ▾
_2$ taskset -c 1 ./client_multithreaded 5
Creating client 1
Creating client 2
Creating client 3
Creating client 4
Creating client 5
Client 5 attempting to connect to the server...
Client 5 connected to server
Client 4 attempting to connect to the server...
Client 4 connected to server
Client 3 attempting to connect to the server...
Client 3 connected to server
Client 2 attempting to connect to the server...
Client 2 connected to server
Client 1 attempting to connect to the server...
Client 1 connected to server
Client 3 received server response:
Process 1: (python3.10) (PID: 302), User Time: 161, Kernel Time: 32
Process 2: (python3.10) (PID: 409), User Time: 161, Kernel Time: 32

Client 3 disconnected
Client 1 received server response:
Process 1: (python3.10) (PID: 302), User Time: 161, Kernel Time: 32
Process 2: (python3.10) (PID: 409), User Time: 161, Kernel Time: 32

Client 1 disconnected
Client 2 received server response:
Process 1: (python3.10) (PID: 302), User Time: 161, Kernel Time: 32
Process 2: (python3.10) (PID: 409), User Time: 161, Kernel Time: 32

Client 2 disconnected
Client 4 received server response:
Process 1: (python3.10) (PID: 302), User Time: 161, Kernel Time: 32
Process 2: (python3.10) (PID: 409), User Time: 161, Kernel Time: 32

Client 4 disconnected
Client 5 received server response:
Process 1: (python3.10) (PID: 302), User Time: 161, Kernel Time: 32
Process 2: (python3.10) (PID: 409), User Time: 161, Kernel Time: 32

Client 5 disconnected
```

Figure 2: Client outputs with process information received from the server

## Conclusion

In this client-server program, we demonstrated how multithreading on the server allows it to handle multiple concurrent connections. By pinning the processes to specific CPUs, we can better measure performance. The unordered responses from the server show that multithreading allows concurrent processing of client requests.

## Question 2

The socket programming source code that leverages “select” system call here. Modify the server code as per Q.1. Use the perf tool to analyze the performance of the following: (a) Single-threaded TCP client-server (b) Concurrent TCP client-server (c) TCP client-server using “select”

## Part A: Single-Threaded TCP Client-Server Analysis

In this section, we analyze the performance of a single-threaded TCP client-server model using the `perf stat` tool. The server handles client requests sequentially, and both server and client processes are pinned to specific CPUs using the `taskset` command. The performance counters analyzed include CPU utilization, context switches, cache misses, and more.

### Commands Used

The following commands were used to collect performance metrics:

- **Server Side:** `sudo /usr/lib/linux-tools/5.15.0-122-generic/perf stat taskset -c 0 ./server_singlethreaded`
- **Client Side:** `sudo /usr/lib/linux-tools/5.15.0-122-generic/perf stat taskset -c 1 ./client_singlethreaded 5`

### Analysis of Performance Metrics

The following key performance metrics were observed and analyzed for the single-threaded TCP client-server model:

#### Server-Side Analysis

- **Task-clock (460.31 msec):** This metric shows that the server's CPU was active for approximately 460.31 milliseconds. Given that the process is single-threaded, the task-clock reflects the time spent waiting for the client's I/O operations, which is typical for sequential client-server communication.
- **Context Switches (27):** The server had 27 context switches. A low number of context switches is expected in a single-threaded model since the CPU is primarily focused on processing one connection at a time, minimizing the need for switching between tasks.
- **CPU Cycles (2,006,110,963 cycles):** The server consumed approximately 2 billion CPU cycles. This indicates the amount of processing effort required. The value is relatively high because single-threaded servers tend to use a lot of CPU time when handling network I/O, even though they remain idle while waiting for the client.
- **Stalled Cycles Backend (464,935,266 cycles - 23.18%):** About 23% of the backend cycles were stalled, likely due to the waiting times when handling network communication. The high proportion of stalled cycles is expected in single-threaded models that process I/O-bound tasks.
- **Instructions (1,140,659,365 instructions):** The server executed about 1.14 billion instructions, but the instructions per cycle (IPC) was 0.57, indicating a relatively low

efficiency in terms of instruction processing. This reflects the fact that the server spent significant time in I/O-bound waiting states.

- **Branch Misses (27,229,829 - 10.22%):** The branch miss rate of 10.22% indicates that there was some inefficiency in the execution path, but given the single-threaded, simple nature of the server process, this rate is within an acceptable range.
- **Page Faults (139):** The server encountered 139 page faults, which shows how often the process needed to access memory that was not in the current working set. This value is quite low, indicating efficient memory usage for a single-threaded process.

#### 0.0.1 Client-Side Analysis

- **Task-clock (3.80 msec):** The client spent only 3.80 milliseconds of active CPU time. This low task-clock value shows that the client does minimal work, sending a small number of requests to the server and waiting for responses.
- **Context Switches (23):** The client had 23 context switches, a slightly lower value than the server, reflecting the short-lived nature of client-side tasks, which mostly involve sending and receiving data.
- **CPU Cycles (8,758,323 cycles):** The client consumed approximately 8.76 million CPU cycles, much fewer than the server due to the relatively smaller processing load on the client side.
- **Stalled Cycles Backend (1,591,976 cycles - 18.18%):** The backend stalled cycles are lower than on the server side, at 18.18%, but still notable. This shows that the client also experienced some waiting time during communication with the server.
- **Instructions (2,530,021 instructions):** The client executed about 2.53 million instructions, and the IPC was 0.29. This low IPC value reflects the inefficiency typical of a client that spends most of its time waiting for server responses rather than doing active computation.
- **Branch Misses (82,827 - 13.75%):** The client had a higher branch miss rate of 13.75%, possibly due to the short bursts of activity involved in sending requests and receiving responses, which could lead to a less predictable execution path.
- **Page Faults (135):** The client encountered 135 page faults, which is similar to the server. This shows that both the client and server are managing memory fairly efficiently, without significant paging activity.

#### Screenshots

```

Performance counter stats for 'taskset -c 0 ./server_singlethreaded':

      460.31 msec task-clock                #   0.004 CPUs utilized
           27      context-switches        #   58.656 /sec
            1      cpu-migrations          #    2.172 /sec
          139      page-faults             #  301.970 /sec
    2006110963      cycles                  #    4.358 GHz
      idle 1644361      stalled-cycles-frontend #   0.08% frontend cycles
      idle 464935266      stalled-cycles-backend #  23.18% backend cycles
    1140659365      instructions           #    0.57 insn per cycle
                                     #    0.41 stalled cycles
per insn 266376821      branches           #  578.690 M/sec
      27229829      branch-misses         #   10.22% of all branches

    110.207708183 seconds time elapsed

      0.020380000 seconds user
      0.440926000 seconds sys

```

Figure 3: Performance metrics for the single-threaded server.

```

Performance counter stats for 'taskset -c 1 ./client_singlethreaded 5':

      3.80 msec task-clock                #   0.008 CPUs utilized
           23      context-switches        #    6.050 K/sec
            1      cpu-migrations          #  263.047 /sec
          135      page-faults             #   35.511 K/sec
    8758323      cycles                  #    2.304 GHz
      idle 150481      stalled-cycles-frontend #   1.72% frontend cycles
      idle 1591976      stalled-cycles-backend #  18.18% backend cycles
    2530021      instructions           #    0.29 insn per cycle
                                     #    0.63 stalled cycles
per insn 602485      branches           #  158.482 M/sec
      82827      branch-misses         #   13.75% of all branches

      0.477026229 seconds time elapsed

      0.002732000 seconds user
      0.001636000 seconds sys

```

Figure 4: Performance metrics for the single-threaded client.

## Conclusion

The single-threaded TCP client-server model exhibits typical characteristics for such setups. The server handles one client at a time, leading to significant CPU stalls due to waiting on network I/O. Both the server and the client show relatively low efficiency in terms of instructions per cycle, largely due to the sequential nature of the execution and the reliance

on network communication. While context switches are minimal, both server and client suffer from stalled backend cycles, with the server side showing higher percentages due to the longer time spent waiting for client requests.

## Part B: Multi-Threaded TCP Client-Server Analysis

In this section, we analyze the performance of a multi-threaded TCP client-server model using the `perf stat` tool. The server is now multi-threaded, handling multiple client requests simultaneously. The client, similarly, has been adapted for multi-threaded communication. Both server and client processes are pinned to specific CPUs using the `taskset` command.

### Commands Used

The following commands were used to collect performance metrics:

- **Server Side:** `sudo /usr/lib/linux-tools/5.15.0-122-generic/perf stat taskset -c 0 ./server_multithreaded`
- **Client Side:** `sudo /usr/lib/linux-tools/5.15.0-122-generic/perf stat taskset -c 1 ./client_multithreaded 5`

### Analysis of Performance Metrics

The following performance metrics were observed and analyzed for the multi-threaded TCP client-server model:

#### Server-Side Analysis

- **Task-clock (421.18 msec):** The multi-threaded server's CPU was active for approximately 421.18 milliseconds, slightly less than in the single-threaded case. Multi-threading improves CPU utilization by allowing the server to handle multiple requests concurrently, reducing idle time waiting for I/O operations.
- **Context Switches (61):** The multi-threaded server experienced 61 context switches, more than double that of the single-threaded server. This increase is expected in a multi-threaded environment, as the operating system switches between multiple threads.
- **CPU Cycles (1,852,113,659 cycles):** The server used approximately 1.85 billion CPU cycles, slightly fewer than the single-threaded server. This indicates that multi-threading allows for more efficient use of CPU cycles as tasks are distributed across threads.
- **Stalled Cycles Backend (374,134,432 cycles - 20.20%):** The backend cycles stalled at a rate of 20.20%, slightly lower than in the single-threaded case (23.18%). Multi-threading reduces waiting times as threads can continue processing while others wait for I/O operations, resulting in fewer stalled backend cycles.



- **Instructions (1,126,165,068 instructions):** The multi-threaded server executed approximately 1.13 billion instructions, similar to the single-threaded case. However, the instructions per cycle (IPC) increased to 0.61 from 0.57, indicating a more efficient execution of instructions due to the parallelism introduced by multi-threading.
- **Branch Misses (26,747,982 - 10.16%):** The branch miss rate of 10.16% remains consistent with the single-threaded case. This indicates that multi-threading did not introduce significant inefficiencies in the server's control flow.
- **Page Faults (163):** The multi-threaded server encountered 163 page faults, slightly more than the single-threaded case. This is expected, as multi-threaded programs tend to handle more data in memory, leading to more frequent memory accesses and potential page faults.

## Client-Side Analysis

- **Task-clock (4.63 msec):** The multi-threaded client used 4.63 milliseconds of active CPU time, slightly more than the single-threaded client. This increase reflects the additional overhead involved in handling multiple threads simultaneously.
- **Context Switches (24):** The client had 24 context switches, similar to the single-threaded client, as the workload of sending and receiving requests remains light even with multi-threading.
- **CPU Cycles (9,575,433 cycles):** The multi-threaded client consumed approximately 9.57 million CPU cycles, only marginally higher than the single-threaded client, suggesting that the client-side workload remains relatively light, even with multi-threading.
- **Stalled Cycles Backend (1,783,663 cycles - 18.63%):** The backend cycles stalled at a rate of 18.63%, similar to the single-threaded client. The client is still largely I/O-bound, with much of its time spent waiting for responses from the server.
- **Instructions (2,874,584 instructions):** The multi-threaded client executed about 2.87 million instructions, slightly more than the single-threaded client. The instructions per cycle (IPC) was 0.30, reflecting similar inefficiencies in instruction processing, likely due to the waiting times involved in network communication.
- **Branch Misses (93,605 - 13.67%):** The branch miss rate of 13.67% is comparable to the single-threaded client. The control flow remains similarly unpredictable due to the nature of network I/O, which can result in frequent branch mispredictions.
- **Page Faults (145):** The multi-threaded client encountered 145 page faults, similar to the single-threaded case, indicating consistent memory usage patterns across both models.

## Screenshots

```

Performance counter stats for 'taskset -c 0 ./server_multithreaded':

      421.18 msec task-clock                #   0.016 CPUs utilized
           61      context-switches        #  144.832 /sec
           1      cpu-migrations           #   2.374 /sec
          163     page-faults              #  387.011 /sec
      1852113659    cycles                  #   4.397 GHz
idle      1532642    stalled-cycles-frontend #   0.08% frontend cycles
idle      374134432  stalled-cycles-backend  #  20.20% backend cycles
      1126165068    instructions           #   0.61 insn per cycle
per insn                                     #   0.33 stalled cycles
      263142596     branches               #  624.780 M/sec
      26747982     branch-misses          #  10.16% of all branches

25.669971131 seconds time elapsed

0.035112000 seconds user
0.359902000 seconds sys

```

Figure 5: Performance metrics for the multi-threaded-server.

```

Performance counter stats for 'taskset -c 1 ./client_multithreaded 5':

      4.63 msec task-clock                #   0.010 CPUs utilized
           24      context-switches        #   5.184 K/sec
           1      cpu-migrations           #  215.983 /sec
          145     page-faults              #  31.317 K/sec
      9575433     cycles                  #   2.068 GHz
idle      224249    stalled-cycles-frontend #   2.34% frontend cycles
idle      1783663    stalled-cycles-backend  #  18.63% backend cycles
      2874584     instructions           #   0.30 insn per cycle
per insn                                     #   0.62 stalled cycles
      684776     branches               #  147.900 M/sec
      93605     branch-misses          #  13.67% of all branches

0.454399511 seconds time elapsed

0.000000000 seconds user
0.004607000 seconds sys

```

Figure 6: Performance metrics for the multi-threaded-client.

## Conclusion

In the multi-threaded TCP client-server model, we observe improved CPU utilization and increased efficiency in terms of instructions per cycle (IPC) for both the server and client.

The server benefits from reduced backend stalled cycles, as multi-threading allows it to handle multiple requests simultaneously, minimizing idle times. The number of context switches increases on the server side, as expected in a multi-threaded environment, while the client-side workload remains relatively light, with minor increases in CPU cycles and instructions.

Multi-threading has a positive impact on server performance by improving IPC and reducing stalled cycles, though the overall branch miss rate and page fault rate remain consistent with the single-threaded case. The client-side improvements are less pronounced, reflecting the fact that the client workload remains I/O-bound.

## Part C: Select-Based TCP Client-Server Analysis

In this section, we analyze the performance of the select-based TCP client-server model using the `perf stat` tool. The select system call allows the server to handle multiple client connections by monitoring multiple file descriptors, making it an efficient solution for handling I/O-bound tasks.

### Commands Used

The following commands were used to collect performance metrics:

- **Server Side:** `sudo /usr/lib/linux-tools/5.15.0-122-generic/perf stat taskset -c 0 ./server_select`
- **Client Side:** `sudo /usr/lib/linux-tools/5.15.0-122-generic/perf stat taskset -c 1 ./client_select 5`  
*5 represents the number of client to be created, implemented it in the client code using fork system call to create 5 process of client and pass it to server*

### Analysis of Performance Metrics

The following performance metrics were observed and analyzed for the select-based TCP client-server model:

#### Server-Side Analysis

- **Task-clock (3.19 msec):** The server using the `select` call shows a significantly lower task-clock value of 3.19 milliseconds, which means the server is spending less time on the CPU compared to the multi-threaded version. This suggests that the select-based approach is more efficient for handling connections since it reduces the need for constant CPU intervention.
- **Context Switches (33):** The server experienced 33 context switches, lower than the multi-threaded server. This is expected since the select model does not involve multiple threads or processes, which leads to fewer context switches between them.

- **CPU Cycles (8,217,269 cycles):** The server consumed about 8.22 million CPU cycles, much lower than the multi-threaded model, reflecting the reduced CPU workload in a select-based I/O model.
- **Stalled Cycles Backend (1,062,344 cycles - 12.93%):** The backend cycles stalled at 12.93%, lower than the multi-threaded server's 20.20%. This indicates that the select-based server is able to utilize the backend more efficiently, as it is not waiting for I/O operations as frequently.
- **Instructions (2,489,994 instructions):** The server executed approximately 2.49 million instructions, fewer than the multi-threaded server, which indicates that the select model requires fewer operations to handle connections.
- **Branch Misses (82,033 - 13.81%):** The branch miss rate was 13.81%, higher than the multi-threaded server. This increase suggests that the control flow in the select-based server might be more unpredictable, likely due to the nature of waiting for multiple events from multiple clients.
- **Page Faults (135):** The server encountered 135 page faults, similar to the multi-threaded model. This is expected, as the memory access pattern does not change significantly between the two models.

## Client-Side Analysis

- **Task-clock (4.35 msec):** The select-based client used 4.35 milliseconds of active CPU time, which is comparable to the multi-threaded client. This reflects that the client workload remains light and is primarily dependent on sending and receiving data.
- **Context Switches (27):** The client had 27 context switches, slightly more than in the multi-threaded client case. This small increase may be due to the additional handling of select-based I/O events.
- **CPU Cycles (11,246,163 cycles):** The client consumed about 11.25 million CPU cycles, slightly higher than the multi-threaded client. This suggests that the select model introduces a bit more CPU overhead on the client side, likely because the select call constantly checks for I/O readiness.
- **Stalled Cycles Backend (1,174,727 cycles - 10.45%):** The backend cycles stalled at 10.45%, similar to the multi-threaded client. The client remains I/O-bound, with much of its time spent waiting for data to be transmitted to and from the server.
- **Instructions (4,256,674 instructions):** The client executed approximately 4.26 million instructions, significantly more than the multi-threaded client. This suggests that the select model involves more instructions for checking file descriptor readiness.
- **Branch Misses (131,841 - 12.96%):** The branch miss rate of 12.96% is similar to the multi-threaded client. The control flow remains unpredictable due to the nature of network communication, leading to frequent branch mispredictions.

- **Page Faults (320):** The client encountered 320 page faults, higher than the multi-threaded model. This is likely due to the additional memory operations involved in managing select-based I/O events.

## Screenshots

```
Performance counter stats for 'taskset -c 0 ./server_select':

      3.19 msec task-clock                #   0.000 CPUs utilized
           33      context-switches      #   10.334 K/sec
           1      cpu-migrations          #  313.165 /sec
          135      page-faults            #   42.277 K/sec
      8217269      cycles                  #    2.573 GHz
idle      160422      stalled-cycles-frontend #    1.95% frontend cycles
idle     1062344      stalled-cycles-backend  #   12.93% backend cycles
          2489994      instructions        #    0.30  insn per cycle
per insn              #    0.43  stalled cycles
           594002      branches            #   186.021 M/sec
           82033      branch-misses        #   13.81% of all branches

      13.275034285 seconds time elapsed

      0.003913000 seconds user
      0.000000000 seconds sys
```

Figure 7: Performance metrics for the server using select.

```
Performance counter stats for 'taskset -c 1 ./client_select 5':

      4.35 msec task-clock                #   0.475 CPUs utilized
           27      context-switches      #    6.213 K/sec
           1      cpu-migrations          #  230.113 /sec
          320      page-faults            #   73.636 K/sec
     11246163      cycles                  #    2.588 GHz
idle      223052      stalled-cycles-frontend #    1.98% frontend cycles
idle     1174727      stalled-cycles-backend  #   10.45% backend cycles
          4256674      instructions        #    0.38  insn per cycle
per insn              #    0.28  stalled cycles
          1017262      branches            #   234.085 M/sec
          131841      branch-misses        #   12.96% of all branches

      0.009150052 seconds time elapsed

      0.005087000 seconds user
      0.000000000 seconds sys
```

Figure 8: Performance metrics for the client select.

## Conclusion

In the select-based TCP client-server model, the server shows improved CPU efficiency with a much lower task-clock and fewer CPU cycles. The select system call allows the server to handle multiple connections without resorting to multi-threading, reducing the number of context switches and backend stalls. However, this efficiency comes at the cost of a slightly higher branch miss rate due to the unpredictability of waiting for I/O readiness.

On the client side, the select-based model introduces slightly more CPU cycles and instructions compared to the multi-threaded client. The client workload remains light and primarily I/O-bound, though the select model results in slightly higher page faults and context switches due to the added complexity of managing file descriptor readiness.

Overall, the select-based approach is more efficient for handling multiple connections without the overhead of multi-threading, but it may introduce slight overheads in terms of branch mispredictions and memory operations.

## Code Repository

All the source code for the client-server models and related performance analysis can be found in the GitHub repository:

<https://github.com/SGx3377/Computer-Networks-Assignment-2.git>.

## Note

The codes for this project have been executed on two separate Windows Subsystem for Linux (WSL) environments as instructed on Google Classroom, due to the unavailability of a dedicated Linux machine. Additionally, the performance analysis using the ‘perf’ tool did not work as expected on virtual machines, which may affect the accuracy of the performance metrics presented.