Computer Networks Assignment 2

Done By:- Harsh Kumar (2022198), Harsh Mistry (2022200), Daksh Bhasin (2021035)

Github Link

Key Parts of the Code

1. **getProcessInfo Function:**
    - Takes in a ProcessInfo struct and a process ID (pid).
    - Opens the /proc/<pid>/stat file, which contains detailed information about the process.
    - Extracts the process ID, name, and CPU times (user and kernel time) and stores them in the ProcessInfo struct.
    - Returns -1 if it fails to open the file, 0 otherwise.
2. **compareCpu Function:**
    - Compares two ProcessInfo structs based on the total CPU time (user time + kernel time).
    - This is used to sort processes by their CPU usage in descending order (highest CPU usage first).
3. **sendTopTwoProcesses Function:**
    - Opens the /proc directory, which contains directories for each running process.
    - Reads the directory entries and, for each process with a valid numeric pid, collects its information using getProcessInfo.
    - After collecting process data, it sorts the processes by CPU usage using qsort.
    - Sends the details (name, PID, user CPU time, kernel CPU time) of the top two CPU-consuming processes to the client.
4. **handleClient Function:**
    - A dedicated thread function to handle each client connection.
    - It receives the client socket as an argument, sends the top two CPU-consuming processes using sendTopTwoProcesses, and closes the client socket after finishing.
    - The pthread_detach call ensures that the thread cleans up its resources once it completes.

Single-Thread Server

This code implements a basic TCP server that listens for incoming client connections, retrieves information about the top two CPU-consuming processes, and sends this information to the client. After handling a client, it closes the connection and waits for another client to connect. Unlike the previous version, this code is single-threaded, meaning that it can only handle one client connection at a time.

1. **Main Server Logic:**
    - **Socket Setup:**
        - The server creates a TCP socket using the socket() system call, specifying the address family (AF_INET for IPv4) and the communication type (SOCK_STREAM for TCP).

        - The bind() function associates the server socket with a specific IP address (INADDR_ANY to listen on all network interfaces) and port (SERVER_PORT).
        - The listen() call sets the socket to listen for incoming connections with a maximum of 100 pending connections.
    - **Main Event Loop (Accepting and Serving Clients):**
        - The server enters an infinite loop, where it uses the accept() function to accept an incoming client connection. This call blocks until a client connects.
        - Once a connection is accepted, the sendTopTwoProcesses() function is called to gather and send the top two CPU-consuming processes to the client.
        - After the data is sent, the client socket is closed using close().
        - The server then continues to wait for the next client connection.
2. **Single-Threaded Server:**
    - This version of the server is single-threaded, meaning it can only handle one client at a time. Once a client connects, the server processes that client, and other clients must wait for the server to finish handling the current client before they can be served.
    - While this is simpler to implement, it limits scalability since only one client can be served at a time.

Overall Flow

1. **Server Initialization:**
    - The server sets up a TCP socket, binds it to a specified port, and begins listening for incoming connections.
2. **Main Loop (Handling Clients):**
    - The server accepts a client connection using accept(). Once a connection is established, it gathers information about the top two CPU-consuming processes and sends this information to the client.
    - The client socket is closed after handling the client, and the server waits for the next client connection.

Multi-thread Server

This code implements a multithreaded TCP server that listens for incoming client connections, retrieves information about the top two CPU-consuming processes, and sends that information to the client. After handling the client, the server closes the connection and continues to serve other clients using threads to manage multiple connections concurrently.

1. **Main Server Logic:**
    - **Socket Setup:**
        - The server creates a TCP socket and binds it to a specific port (SERVER_PORT) on all network interfaces (INADDR_ANY).
        - The SO_REUSEADDR socket option is set to allow the server to reuse the port immediately after the server is terminated, avoiding issues with lingering sockets.
        - The server listens for incoming client connections with a backlog of MAX_PENDING_CONNECTIONS.
    - **Main Loop (Handling Clients with Threads):**

- The server runs an infinite loop, continuously accepting new client connections.
- For each new client connection, it allocates memory for a clientSocket and creates a new thread using pthread_create to handle the client.
- Each client is served in a separate thread by calling handleClient, allowing multiple clients to be handled concurrently without blocking.
- After the thread is created, the pthread_detach function is called to ensure the thread cleans up its resources after completion.
- If any memory allocation or thread creation fails, the client connection is handled properly by closing the socket and freeing allocated memory to prevent resource leaks.

Overall Flow

1. **Server Initialization:**
    - The server sets up a non-blocking socket, binds it to a specific port, and starts listening for incoming client connections.
2. **Main Event Loop:**
    - When a new client connects, a new thread is created to handle that client. The thread sends the top two CPU-consuming processes to the client and closes the connection.
3. **Threading and Concurrency:**
    - Each client is handled in a separate thread, allowing multiple clients to be served simultaneously. Threads are detached to ensure proper cleanup after completion.

Select Based Server

This code implements a simple TCP server that listens for client connections, retrieves process information from the /proc filesystem, and sends the top two CPU-consuming processes to the client. Here's a breakdown of how it works:

1. **Main Server Logic:**
    - **Socket Setup:**
        - Creates a non-blocking server socket that listens on a specified port (SERVER_PORT).
        - Binds the socket to the address INADDR_ANY (allowing it to listen on all network interfaces).
        - The server listens for incoming connections with a backlog of SOMAXCONN.
    - **Main Loop (Handling Connections and Clients):**
        - A loop constantly monitors the server for incoming client connections and data using the select function.
        - **New Connections:**
            - When a new client connects, the server accepts the connection and adds the new client's socket to the list of tracked sockets (clientSockets).
            - The server immediately sends the top two CPU-consuming processes to the client by calling sendTopTwoProcesses.
        - **Handling Client Data:**
            - For each client socket in the list, if there is activity (i.e., data to be read), it reads the data into a buffer.

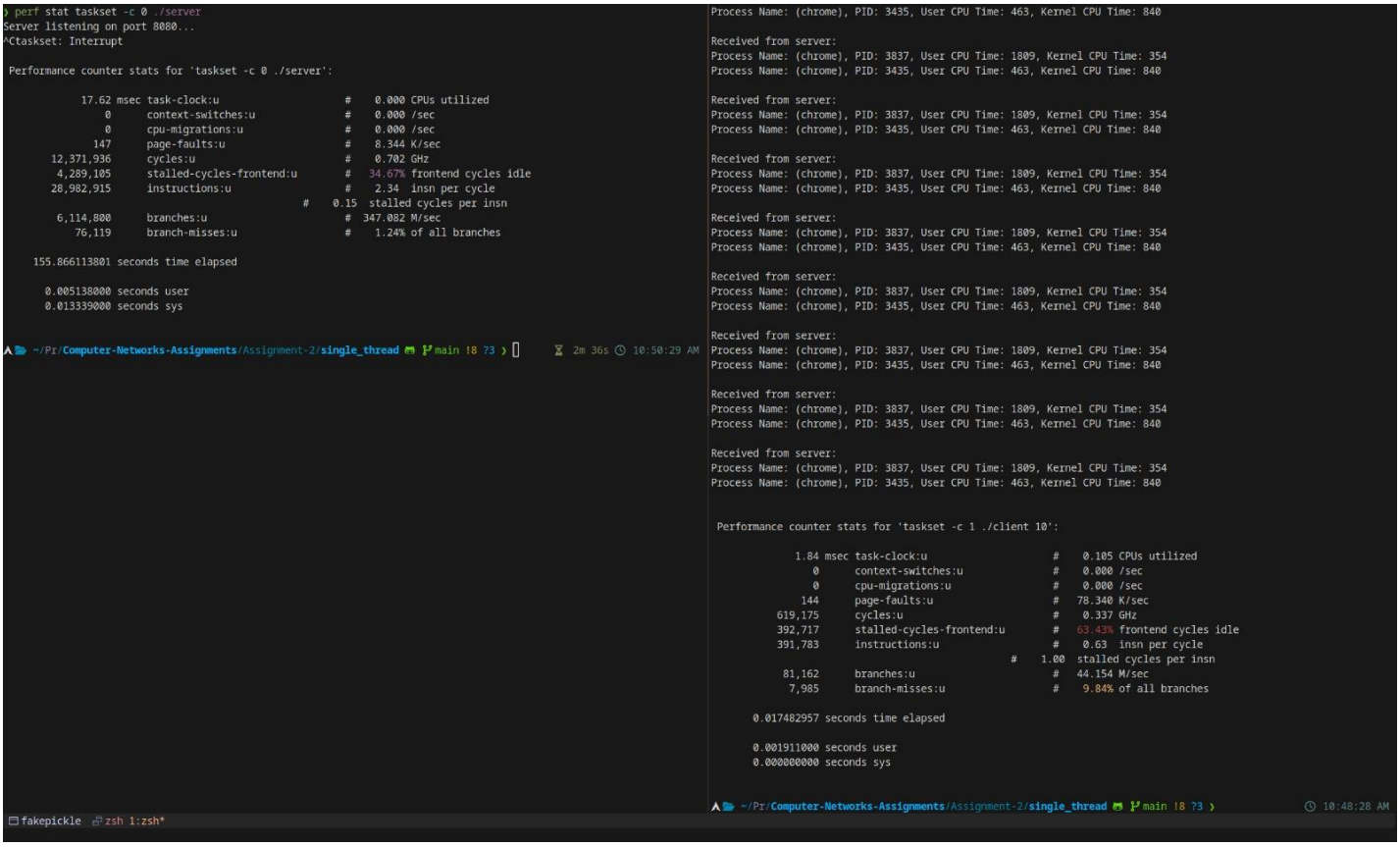
        - If a client disconnects (the read operation returns 0), the socket is closed, and the client is removed from the list.
    - The server uses select to efficiently monitor multiple client connections, enabling handling of up to MAX_CLIENTS at once.

Client Multi-Thread

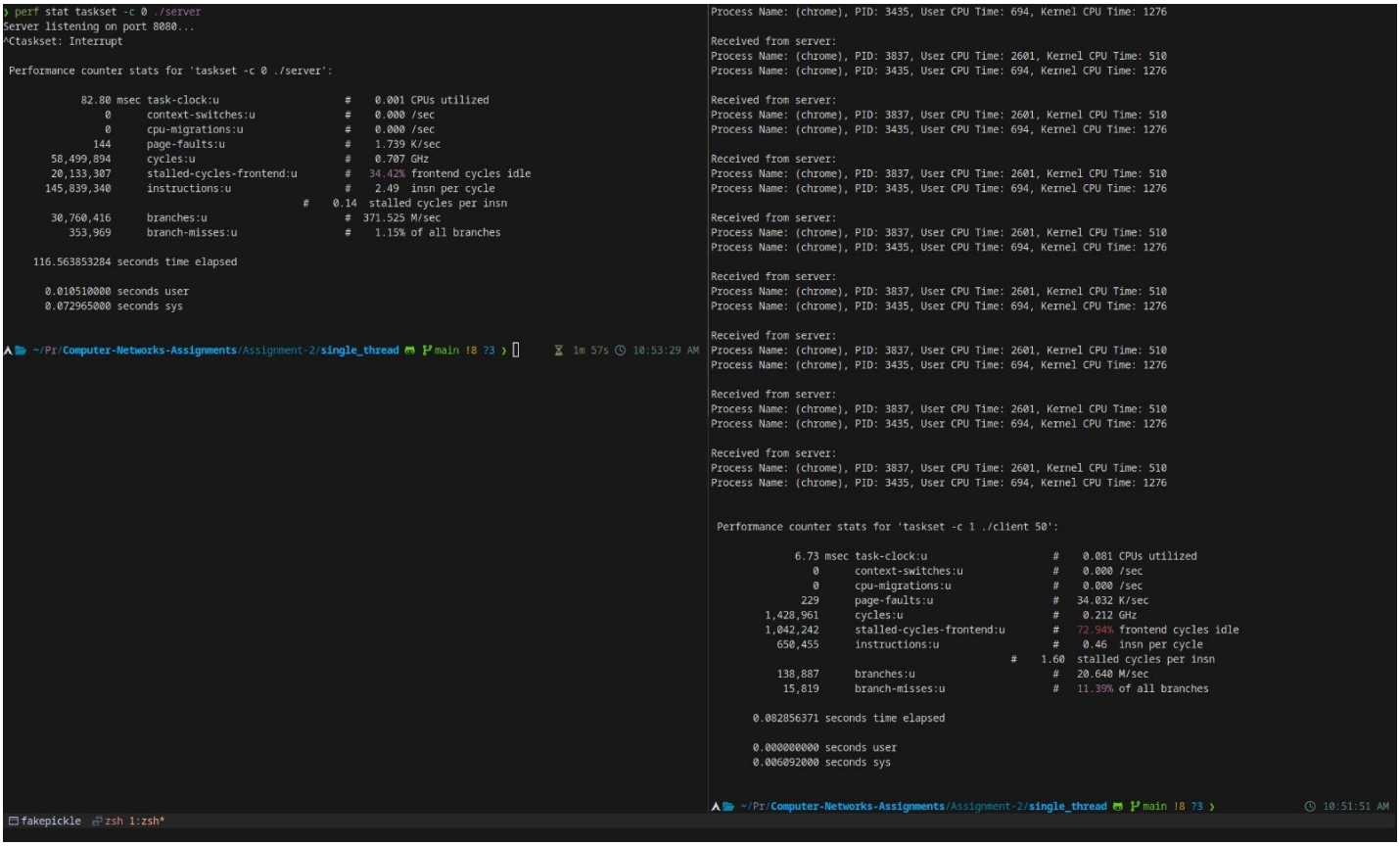
This code implements a multithreaded client program that connects to a server, receives data from the server, and then prints the received information. It accepts an argument specifying the number of clients (threads) to create, and each client runs in its own thread.

Key Component

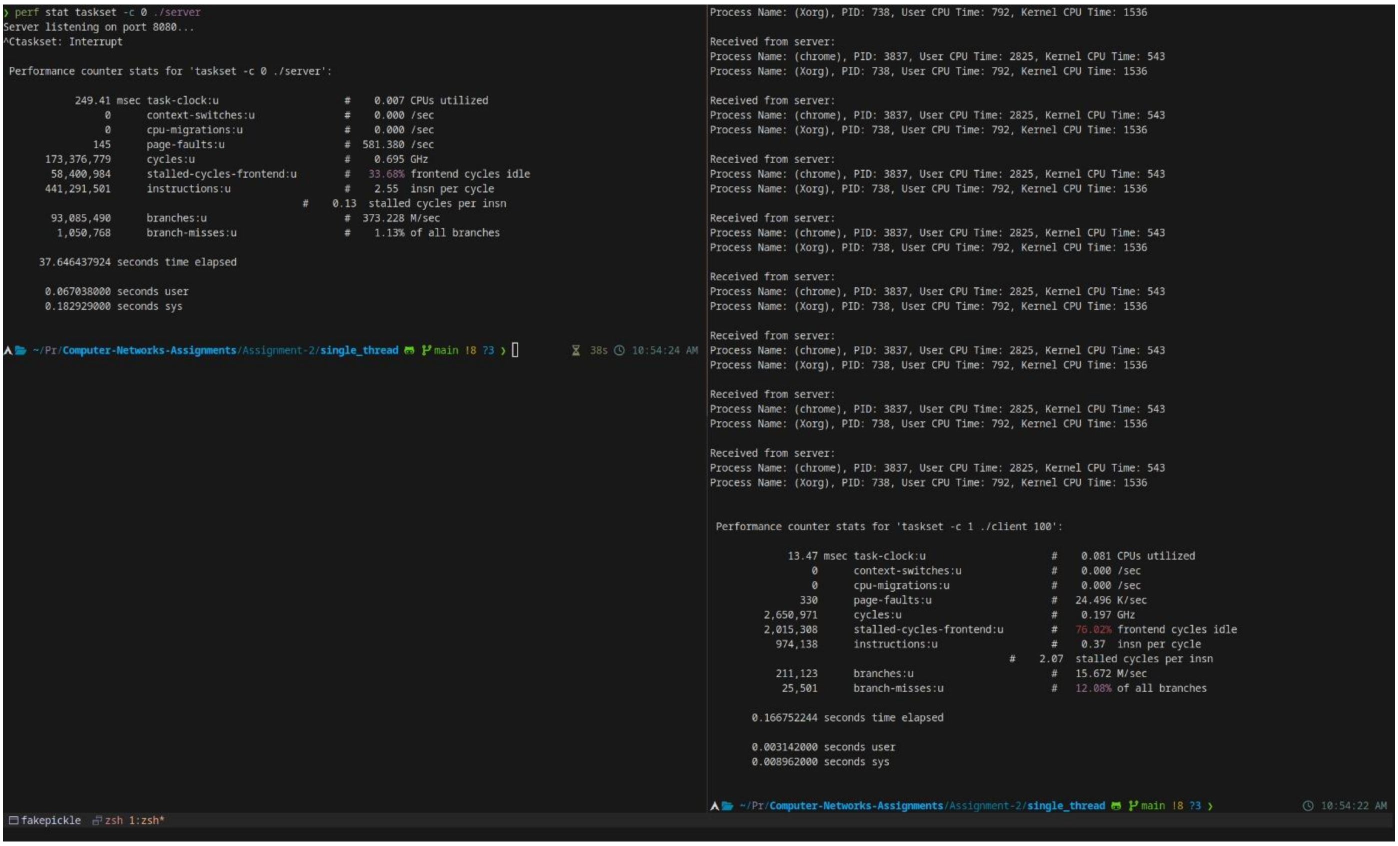
1. **client_thread Function:**
    - This function runs as a separate thread for each client. It establishes a TCP connection with the server, receives data, prints the received data, and then closes the connection.
    - **Steps in client_thread:**
        1. **Socket Creation:**
            - A TCP socket is created using socket(AF_INET, SOCK_STREAM, 0). This socket will be used to communicate with the server.
            - If the socket creation fails, the function prints an error message and returns.
        2. **Server Address Configuration:**
            - The server's IP address and port are defined in server_address.
              The inet_pton() function is used to convert the IP address string (given by IP_ADDRESS) into a binary format suitable for network operations.
        3. **Connecting to the Server:**
            - The client attempts to connect to the server using connect(). If the connection fails, it prints an error message, closes the socket, and returns.
        4. **Receiving Data from Server:**
            - The client waits to receive data from the server via recv(), which reads data from the server into the buffer. The function then prints the received data.
            - If data is successfully received, it's null-terminated (buffer[bytes_received] = '\0') to ensure proper string formatting for printing.
        5. **Closing the Socket:**
            - After receiving the data, the client closes its socket connection to the server.
    - **Returns:**
        - The thread function returns NULL to indicate the thread has finished its task.



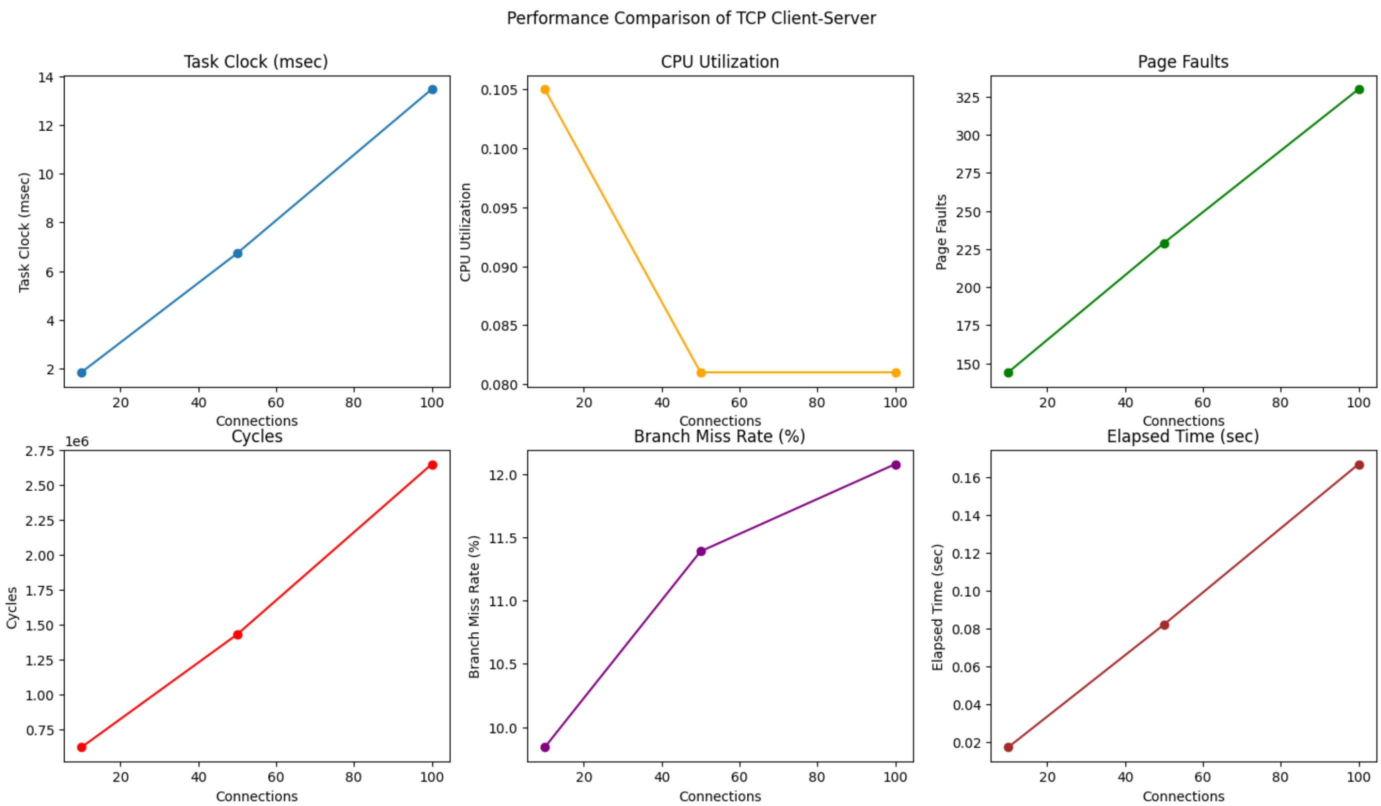
Single Threaded Server for 10 clients



Single Threaded Server for 50 clients



Single Threaded Server for 100 clients



Connections	Task Clock (msec)	CPU Utilization	Page Faults	Cycles	Instructions	Branches Missed	Branch Miss Rate (%)	Elapsed Time (sec)
10	1.84	0.105	144	619175	391783	7985	9.84	0.017
50	6.73	0.081	229	1428961	650455	15819	11.39	0.082
100	13.47	0.081	330	2650971	974138	25501	12.08	0.167

1. 10 Concurrent Connections

- CPU usage is very low (~0.1 CPUs), indicating minimal load.
- Branch miss rate of 9.84% is moderately high, meaning there might be some inefficiency in branch prediction, affecting performance.
- Low elapsed time shows efficient communication between server and clients for 10 connections.

2. 50 Concurrent Connections

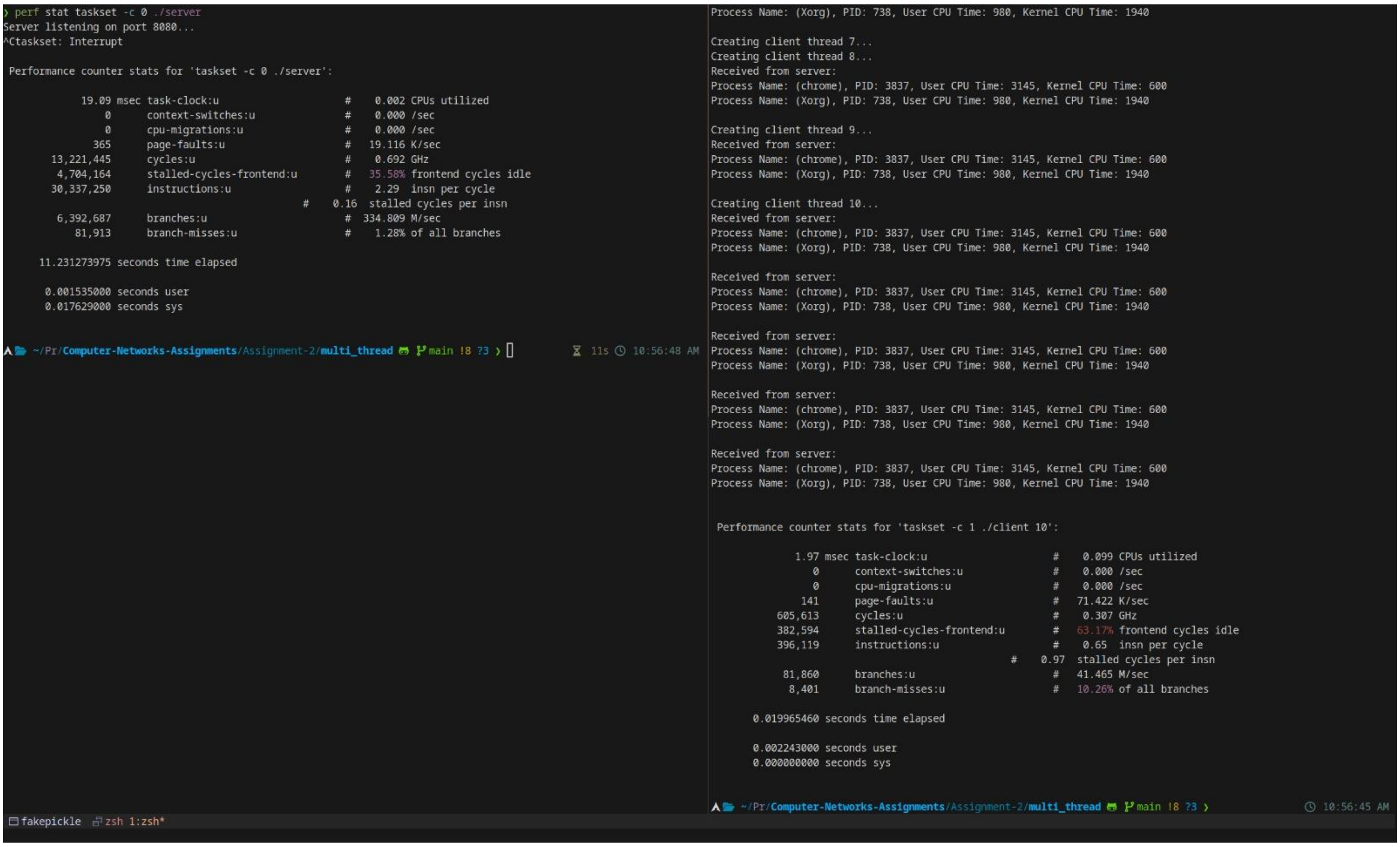
- CPU utilization is lower, but page faults, cycles, and branch misses have increased significantly compared to the 10-connection case.
- Branch miss rate has worsened (11.39%), suggesting performance degradation as the number of connections increases.
- Time elapsed is still manageable, but performance may start to bottleneck as the load increases.

3. 100 Concurrent Connections

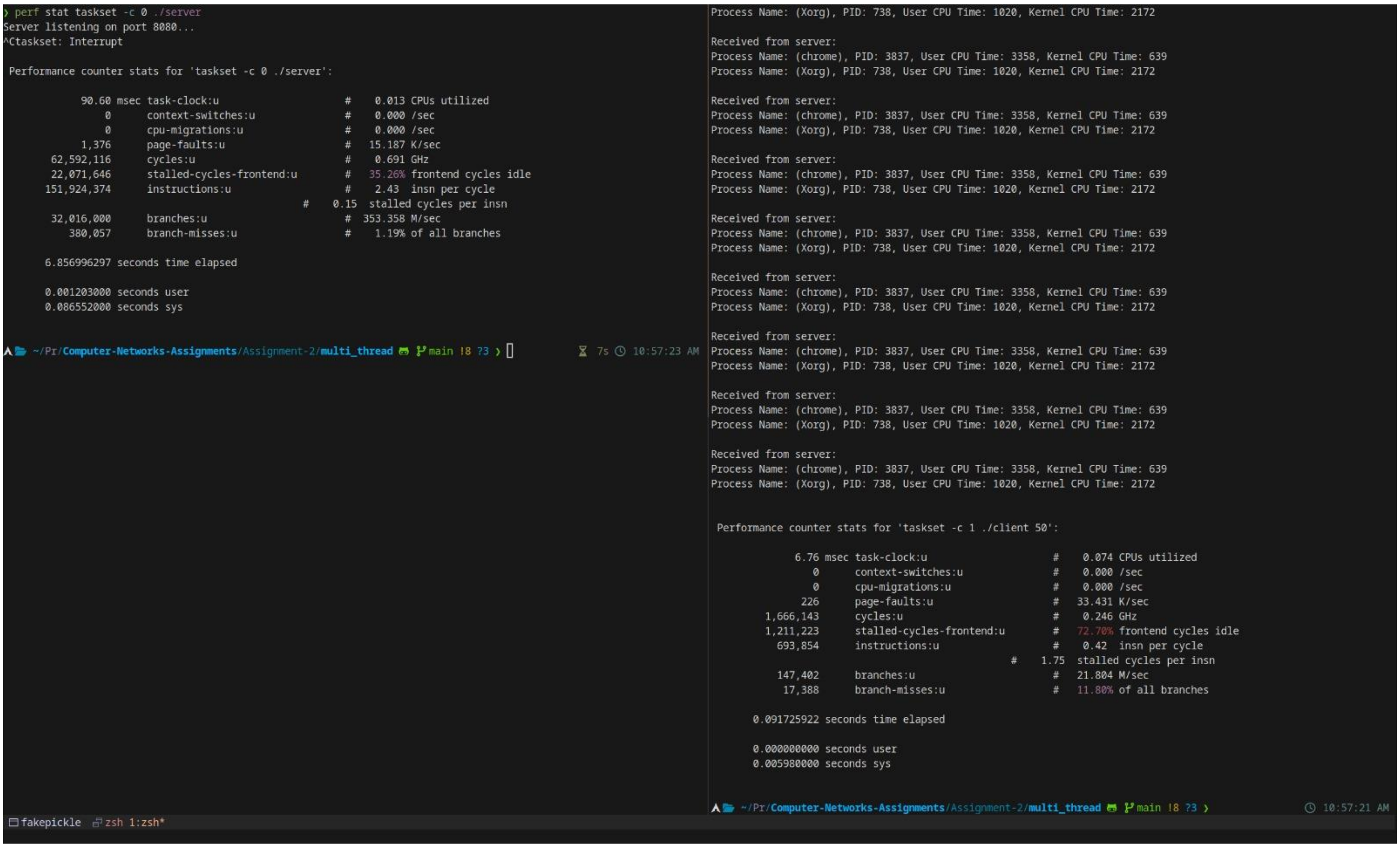
- There is further increase in page faults, CPU cycles, and branch misses.
- The branch miss rate is now over 12%, meaning a significant decrease in prediction accuracy.
- The task clock and overall elapsed time have increased considerably, indicating a heavy performance hit when dealing with 100 concurrent connections.

Overall Analysis:

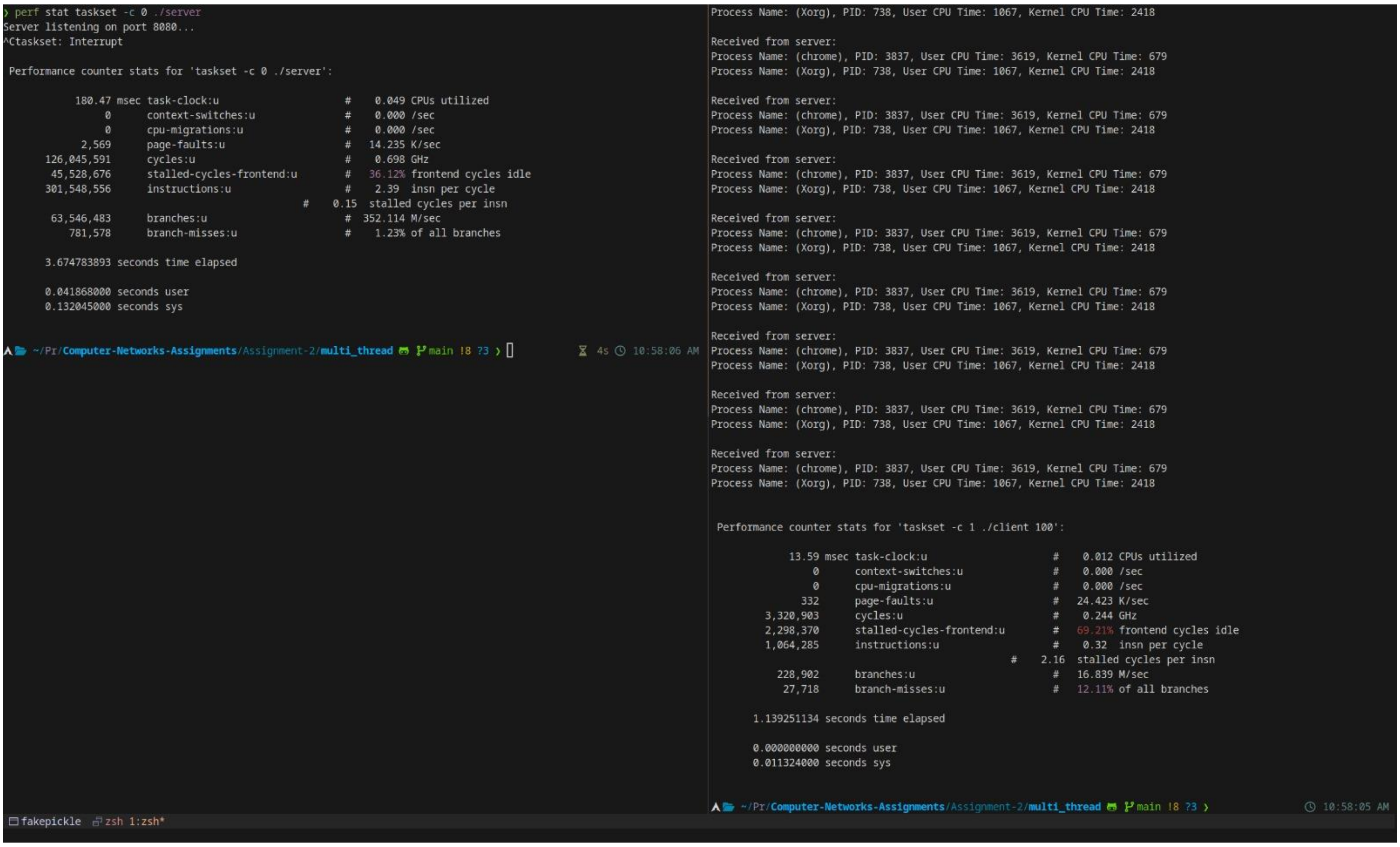
1. **CPU Usage:**
    - The CPU utilization remains low across all scenarios, likely because the server is not fully utilizing the available processing power in a single-threaded model.
2. **Branch Miss Rate:**
    - The branch miss rate progressively worsens as the number of connections increases, which may indicate inefficient branching in the code that handles multiple concurrent connections. This could be optimized to reduce performance overheads.
3. **Page Faults:**
    - The number of page faults increases with the number of connections, which might signal memory management inefficiencies as more clients connect. This could be an area to optimize, perhaps by tuning memory allocations or caching mechanisms.



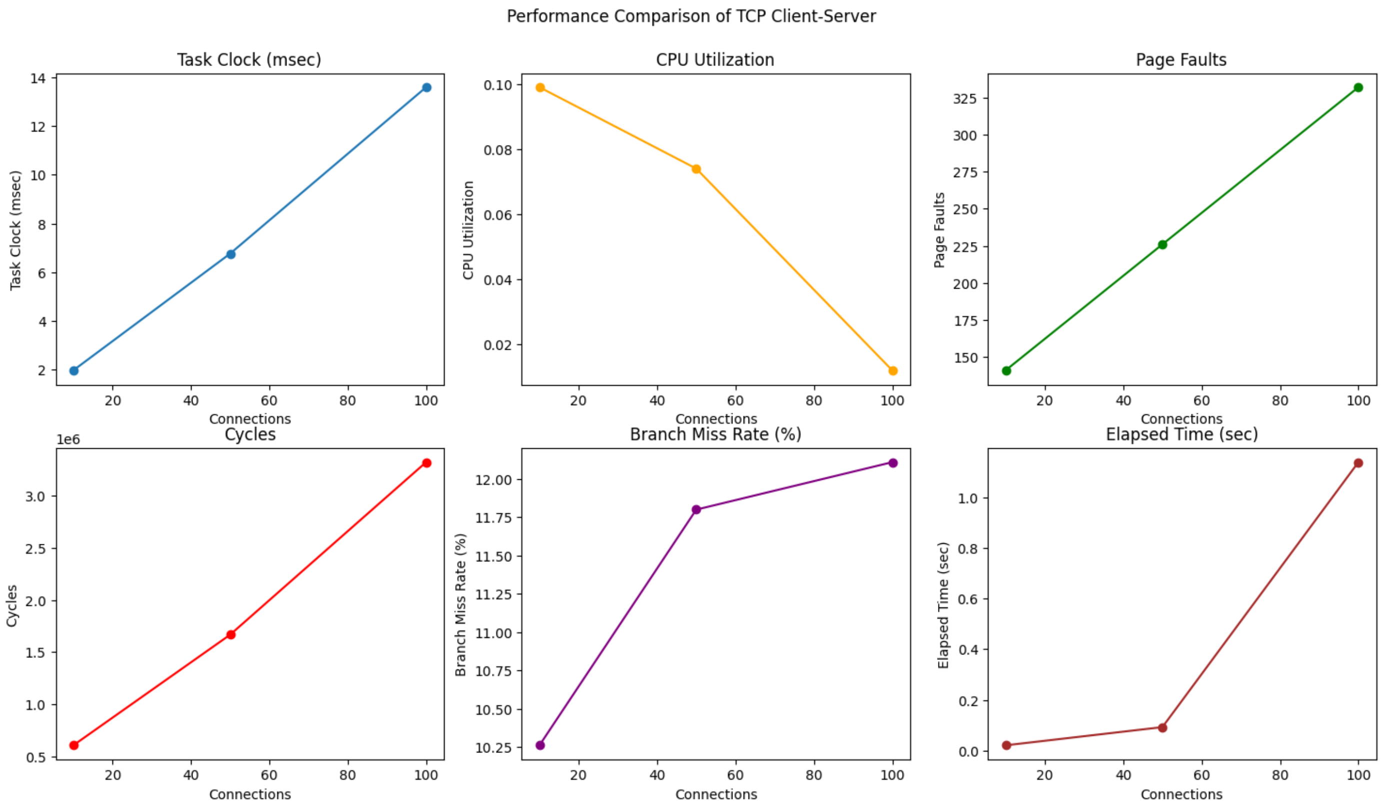
Multi Threaded Server for 10 clients



Multi Threaded Server for 50 clients



Multi Threaded Server for 100 clients



Connections	Task Clock (msec)	CPU Utilization	Page Faults	Cycles	Instructions	Branches Missed	Branch Miss Rate (%)	Elapsed Time (sec)
10	1.97	0.099	141	605613	396119	8401	10.26	0.020
50	6.76	0.074	226	1666143	693854	17388	11.80	0.092
100	13.59	0.012	332	3320903	1064285	27718	12.11	1.140

1. 10 Concurrent Connections

- **CPU usage** is very low (0.099 CPUs), indicating minimal load.
- **Branch miss rate** of 10.26% is moderately high, meaning there may be inefficiencies in branch prediction affecting performance.
- **Elapsed time** shows efficient communication between the server and clients for 10 concurrent connections, with low overhead.

2. 50 Concurrent Connections

- **CPU utilization** remains low but Page faults, cycles, and branch misses have all increased significantly.
- **Branch miss rate** worsens to 11.80%, indicating that the system's ability to predict branches is further impacted by higher connection loads.
- **Elapsed time** is higher but still manageable, though signs of performance degradation appear as the number of connections grows.

3. 100 Concurrent Connections

- There is a significant increase in **page faults**, **cycles**, and **branch misses** as the number of connections rises to 100.
- The **branch miss rate** climbs over 12%, indicating a noticeable decrease in branch prediction accuracy under higher loads.
- The **task clock** and **elapsed time** increase considerably, indicating that performance is severely impacted when managing 100 concurrent connections.

Overall Analysis:

1. CPU Usage:

- The CPU utilization remains low across all scenarios, suggesting that the server is not fully utilizing available processing power, likely because each connection does not generate a heavy load, or the server is I/O bound rather than CPU bound.

2. Branch Miss Rate:

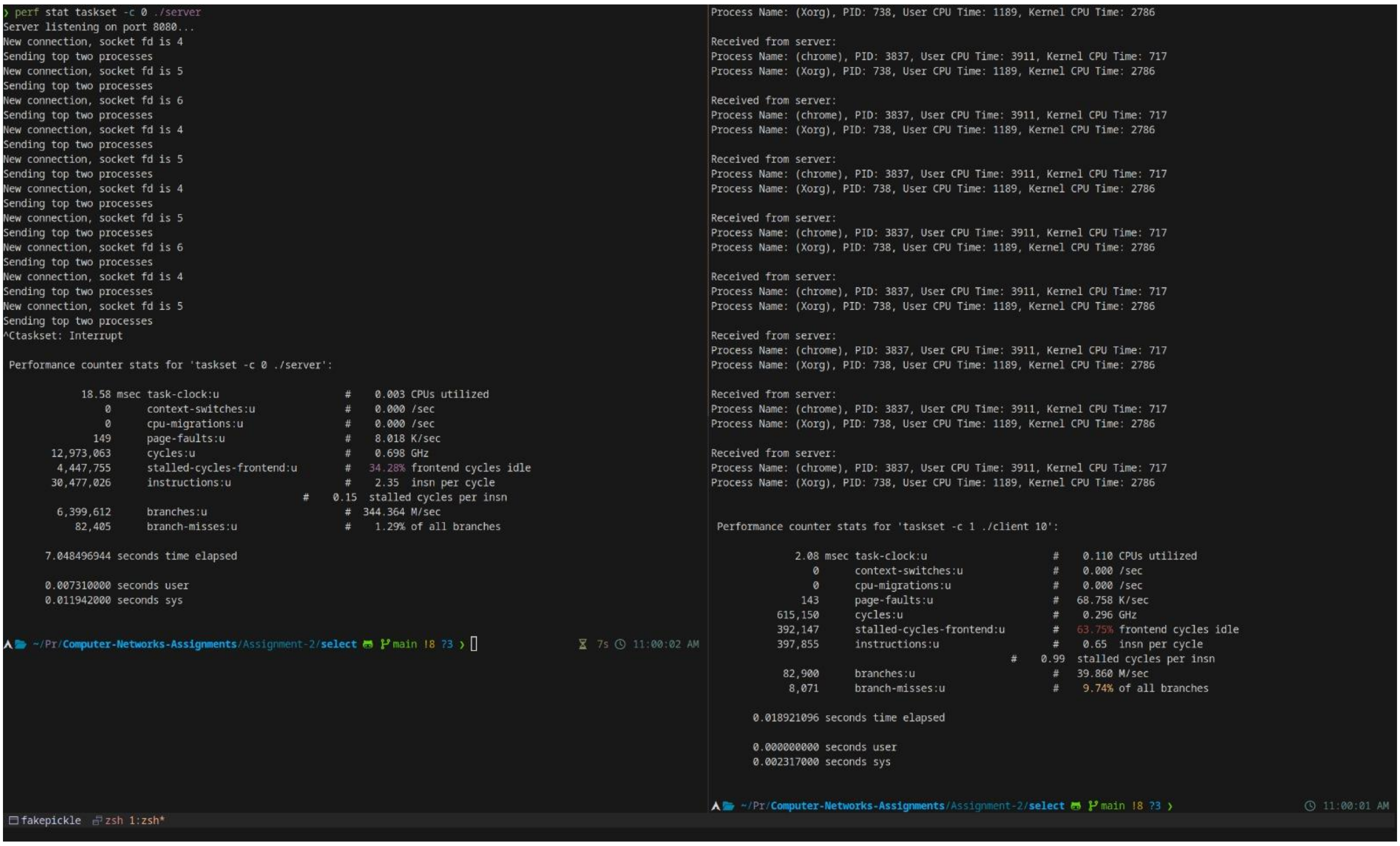
- The branch miss rate progressively worsens from ~10% to over 12% as the number of connections increases, indicating branch misprediction as a performance bottleneck.

3. Page Faults:

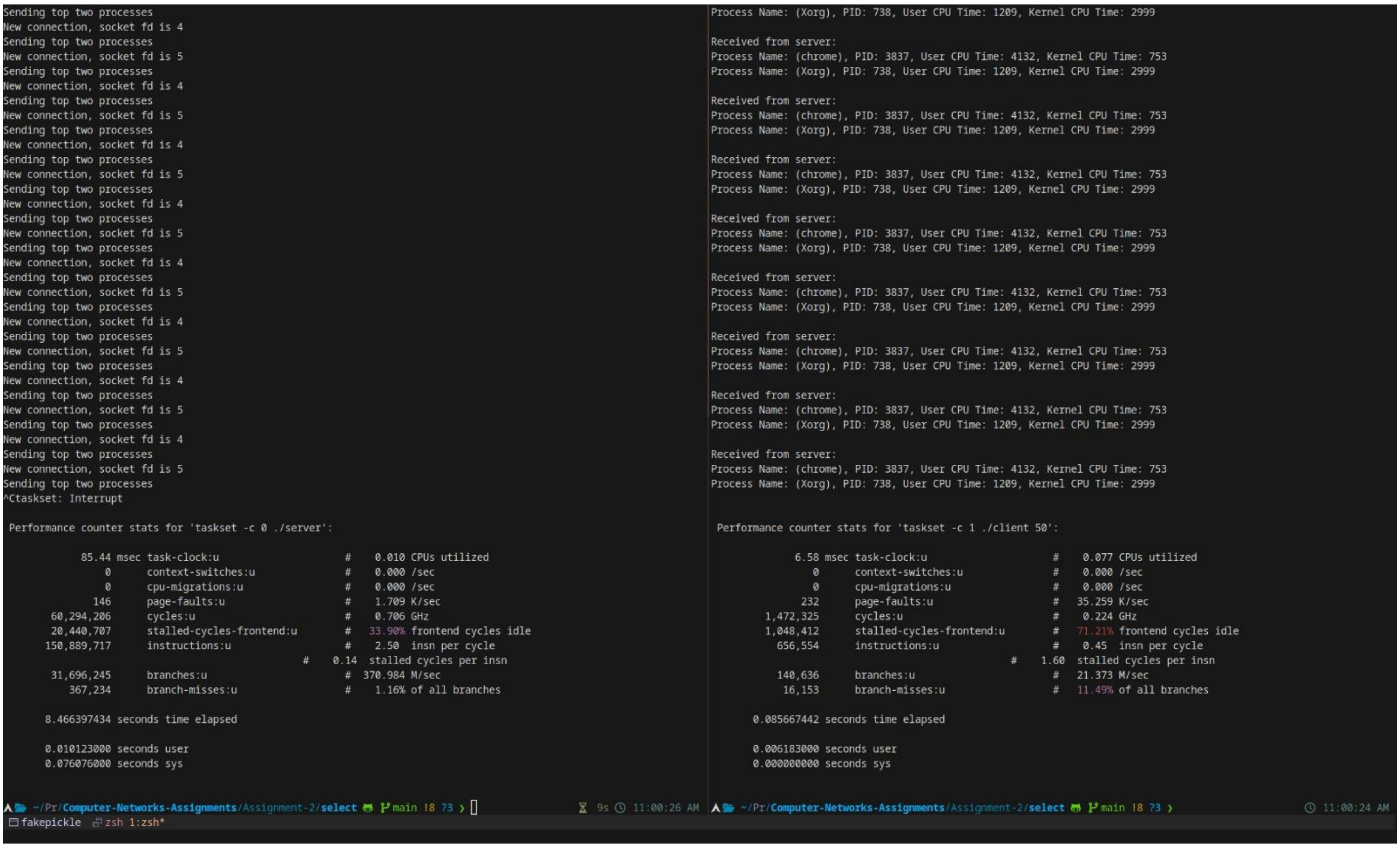
- The number of page faults increases significantly with more concurrent connections, signalling memory management inefficiencies.

4. Cycles and Elapsed Time:

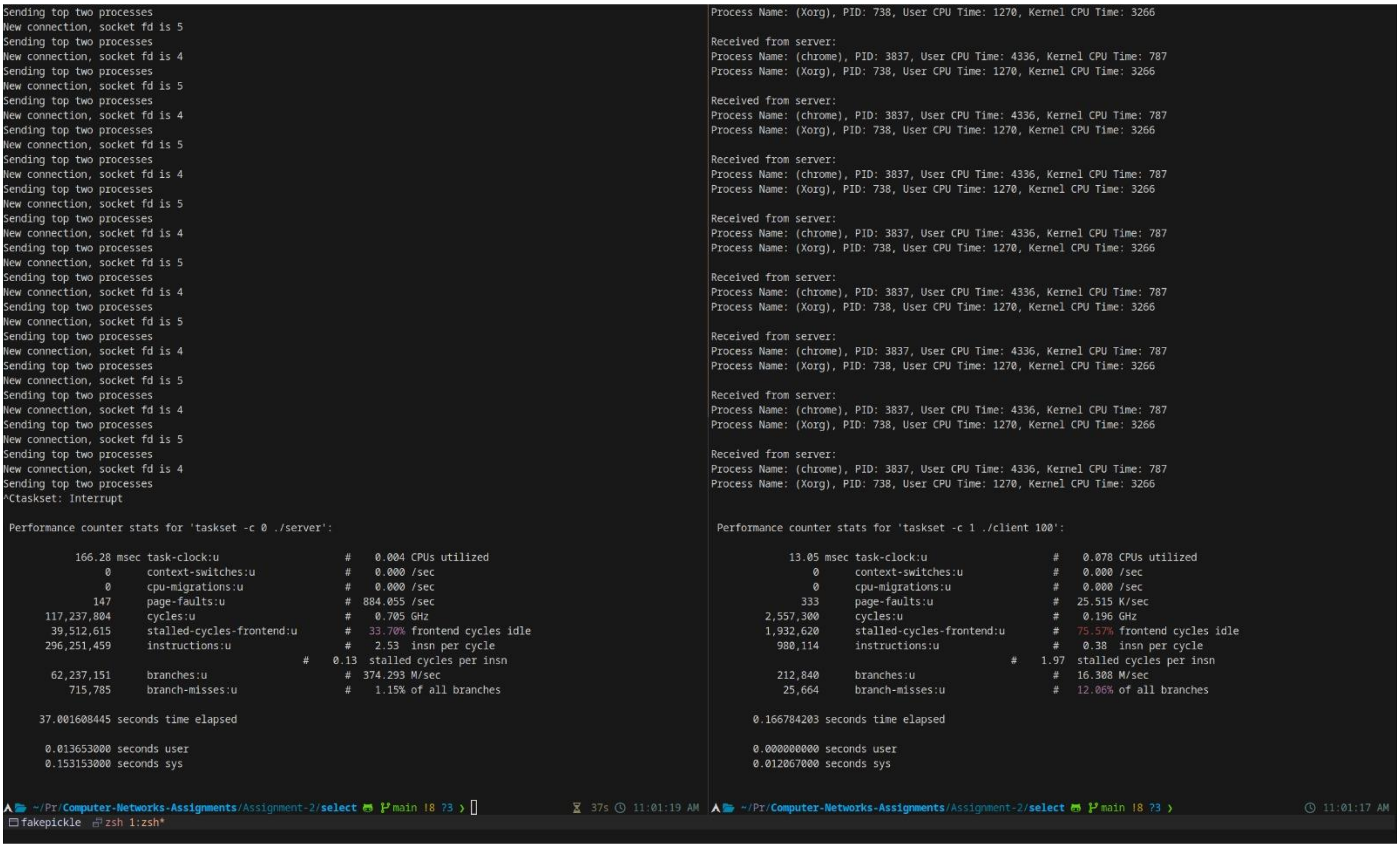
- The number of CPU cycles and the elapsed time grow proportionally with the number of connections. The system is handling the load but becomes less efficient as it scales.



Select based Server for 10 clients



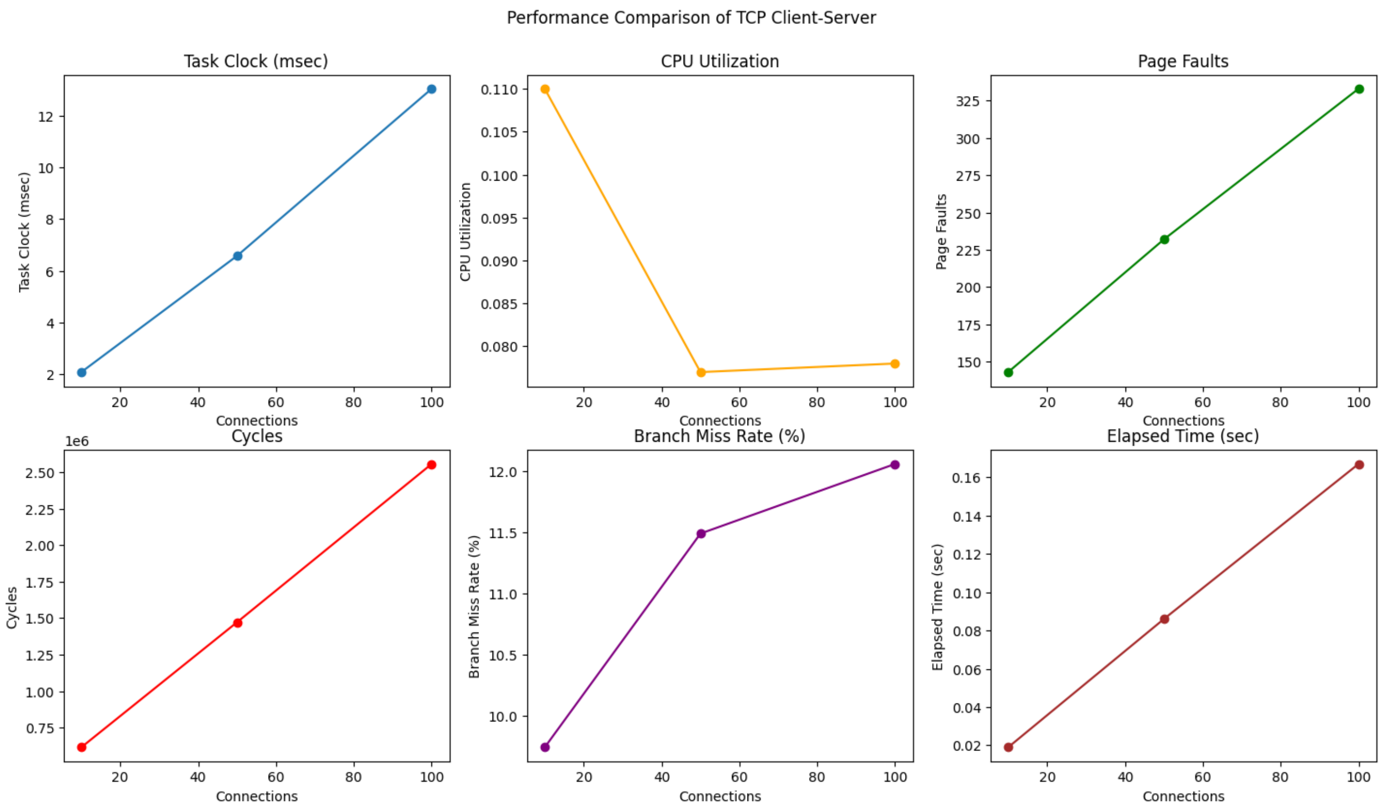
Select based Server for 50 clients



```
Sending top two processes
New connection, socket fd is 5
Sending top two processes
New connection, socket fd is 4
Sending top two processes
New connection, socket fd is 5
Sending top two processes
New connection, socket fd is 4
Sending top two processes
New connection, socket fd is 5
Sending top two processes
New connection, socket fd is 4
Sending top two processes
New connection, socket fd is 5
Sending top two processes
New connection, socket fd is 4
Sending top two processes
New connection, socket fd is 5
Sending top two processes
New connection, socket fd is 4
Sending top two processes
New connection, socket fd is 5
Sending top two processes
New connection, socket fd is 4
Sending top two processes
New connection, socket fd is 5
Sending top two processes
New connection, socket fd is 4
Sending top two processes
New connection, socket fd is 5
Sending top two processes
New connection, socket fd is 4
Sending top two processes
^Ctaskset: Interrupt

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

           166.28 msec task-clock:u                     #    0.004 CPUs utilized
                0      context-switches:u               #    0.000 /sec
                0      cpu-migrations:u                 #    0.000 /sec
              147      page-faults:u                    #  884.055 /sec
      117,237,804      cycles:u                         #    0.705 GHz
       39,512,615      stalled-cycles-frontend:u        #   33.70% frontend cycles idle
      296,251,459      instructions:u                   #    2.53  insn per cycle
                                                  #    0.13  stalled cycles per insn
       62,237,151      branches:u                       #  374.293 M/sec
          715,785      branch-misses:u                  #    1.15% of all branches

     37.001608445 seconds time elapsed

      0.013653000 seconds user
      0.153153000 seconds sys

~/Pr/Computer-Networks-Assignments/Assignment-2/select main !8 ?3        37s 11:01:19 AM
fakepickle  zsh 1:zsh*
```

```
Process Name: (Xorg), PID: 738, User CPU Time: 1270, Kernel CPU Time: 3266

Received from server:
Process Name: (chrome), PID: 3837, User CPU Time: 4336, Kernel CPU Time: 787
Process Name: (Xorg), PID: 738, User CPU Time: 1270, Kernel CPU Time: 3266

Received from server:
Process Name: (chrome), PID: 3837, User CPU Time: 4336, Kernel CPU Time: 787
Process Name: (Xorg), PID: 738, User CPU Time: 1270, Kernel CPU Time: 3266

Received from server:
Process Name: (chrome), PID: 3837, User CPU Time: 4336, Kernel CPU Time: 787
Process Name: (Xorg), PID: 738, User CPU Time: 1270, Kernel CPU Time: 3266

Received from server:
Process Name: (chrome), PID: 3837, User CPU Time: 4336, Kernel CPU Time: 787
Process Name: (Xorg), PID: 738, User CPU Time: 1270, Kernel CPU Time: 3266

Received from server:
Process Name: (chrome), PID: 3837, User CPU Time: 4336, Kernel CPU Time: 787
Process Name: (Xorg), PID: 738, User CPU Time: 1270, Kernel CPU Time: 3266

Received from server:
Process Name: (chrome), PID: 3837, User CPU Time: 4336, Kernel CPU Time: 787
Process Name: (Xorg), PID: 738, User CPU Time: 1270, Kernel CPU Time: 3266

Received from server:
Process Name: (chrome), PID: 3837, User CPU Time: 4336, Kernel CPU Time: 787
Process Name: (Xorg), PID: 738, User CPU Time: 1270, Kernel CPU Time: 3266

Received from server:
Process Name: (chrome), PID: 3837, User CPU Time: 4336, Kernel CPU Time: 787
Process Name: (Xorg), PID: 738, User CPU Time: 1270, Kernel CPU Time: 3266

 Performance counter stats for 'taskset -c 1 ./client 100':

            13.05 msec task-clock:u                     #    0.078 CPUs utilized
                0      context-switches:u               #    0.000 /sec
                0      cpu-migrations:u                 #    0.000 /sec
              333      page-faults:u                    #   25.515 K/sec
        2,557,300      cycles:u                         #    0.196 GHz
        1,932,620      stalled-cycles-frontend:u        #   75.57% frontend cycles idle
          980,114      instructions:u                   #    0.38  insn per cycle
                                                  #    1.97  stalled cycles per insn
          212,840      branches:u                       #   16.308 M/sec
           25,664      branch-misses:u                  #   12.06% of all branches

      0.166784203 seconds time elapsed

      0.000000000 seconds user
      0.012067000 seconds sys

~/Pr/Computer-Networks-Assignments/Assignment-2/select main !8 ?3        11:01:17 AM
```

Select based Server for 100 clients



Connections	Task Clock (msec)	CPU Utilization	Page Faults	Cycles	Instructions	Branches Missed	Branch Miss Rate (%)	Elapsed Time (sec)
10	2.08	0.110	143	615150	397855	8071	9.74	0.019
50	6.58	0.077	232	1472325	656554	16153	11.49	0.086
100	13.05	0.078	333	2557300	980114	25664	12.06	0.167

1. 10 Concurrent Connections

- CPU usage is very low (~0.1 CPUs), indicating minimal load.
- Branch miss rate of 9.74% is moderately high, meaning there might be some inefficiency in branch prediction, affecting performance.
- Low elapsed time shows efficient communication between server and clients for 10 connections.

2. 50 Concurrent Connections

- CPU utilization lower, but page faults, cycles, and branch misses have increased significantly compared to the 10-connection case.
- Branch miss rate has worsened (11.49%), suggesting performance degradation as the number of connections increases.
- Time elapsed is still manageable, but performance may start to bottleneck as the load increases.

3. 100 Concurrent Connections

- There is further increase in page faults, CPU cycles, and branch misses.
- The branch miss rate is now over 12%, meaning a significant decrease in prediction accuracy.
- The task clock and overall elapsed time have increased considerably, indicating a heavy performance hit when dealing with 100 concurrent connections.

Overall Analysis:

1. **Task Clock (msec)**:
    - As the number of concurrent connections increases, the task clock also increases. This indicates that the server is spending more time processing tasks, which is expected with an increased workload.
    - For 100 connections, the task clock is about **6 times** higher than for 10 connections.
2. **CPU Utilization**:
    - Interestingly, CPU utilization remains relatively stable, hovering around 7.7% to 11%. This suggests that the select system call is efficiently handling the I/O multiplexing without excessively overloading the CPU, even as the number of connections increases.
3. **Page Faults**:
    - Page faults increase as the number of connections grows. This trend suggests that more memory pages are being accessed, which could impact performance, especially as the server scales up. However, the rate of increase is moderate compared to other metrics, implying that memory is being handled reasonably well.
4. **Cycles**:
    - The number of CPU cycles required to process the workload increases significantly with the number of connections. For 100 connections, the server requires more than **4 times** the number of cycles compared to 10 connections. This is a crucial metric, as it directly impacts the processing speed and efficiency of the system.

5. **Instructions**:
    - The number of instructions executed also increases with the number of connections. For 100 connections, the server executes nearly **2.5 times** more instructions compared to 10 connections. This suggests that the server is performing more work as it handles more clients.
6. **Branches Missed and Branch Miss Rate (%)**:
    - Branch misses increase linearly with the number of connections, leading to higher branch miss rates. This could indicate that the system's prediction mechanisms are being challenged as the workload increases. For 100 connections, the branch miss rate reaches **12.06%**, which can lead to performance bottlenecks.
7. **Elapsed Time (sec)**:
    - The elapsed time increases linearly with the number of connections. For 100 connections, the server takes **0.167 seconds**, which is roughly **8.5 times** longer than for 10 connections. This highlights the additional time required to manage more connections, even with efficient I/O multiplexing.

---

1. Task Clock (msec)

Connections	Single-Threaded	Multi-Threaded	Select (I/O Multiplexing)
10	1.84	1.97	2.08
50	6.73	6.76	6.58
100	13.47	13.59	13.05

- **Observation**: The task clock is the highest in the select system call for lower connections (10), indicating that handling fewer clients via select incurs more overhead. However, as the connections increase to 100, the task clock for all three approaches converges, with the select system call being marginally faster than the other two methods.

---

2. CPU Utilization (%)

Connections	Single-Threaded	Multi-Threaded	Select (I/O Multiplexing)
10	0.105	0.099	0.110
50	0.081	0.074	0.077
100	0.081	0.012	0.078

- **Observation**: Multi-threaded implementation shows drastically lower CPU utilization as the number of connections increases. This indicates efficient thread management. However, the select system call has slightly higher utilization with smaller loads and maintains similar levels for larger loads. Single-threaded utilization remains consistent, though slightly higher than multi-threaded and select.

---

3. Page Faults

Connections	Single-Threaded	Multi-Threaded	Select (I/O Multiplexing)
10	144	141	143
50	229	226	232
100	330	332	333

- **Observation**: The page faults remain fairly consistent across all three methods, with select system call handling slightly more faults at higher connection levels. Single-threaded and multi-threaded approaches perform similarly, with only marginal differences.

---

4. Cycles

Connections	Single-Threaded	Multi-Threaded	Select (I/O Multiplexing)
10	619,175	605,613	615,150
50	1,428,961	1,666,143	1,472,325
100	2,650,971	3,320,903	2,557,300

- **Observation**: Multi-threaded implementation consistently uses the most cycles, especially for larger loads (100 connections). The select system call requires fewer cycles than multi-threaded at higher connections, which suggests it's more cycle-efficient at scale.

---

5. Instructions

Connections	Single-Threaded	Multi-Threaded	Select (I/O Multiplexing)
10	391,783	396,119	397,855
50	650,455	693,854	656,554
100	974,138	1,064,285	980,114

- **Observation**: The number of instructions executed increases for all three implementations as connections increase. Multi-threaded requires the most instructions for larger loads, while the select system call is slightly more efficient than the single-threaded approach for handling high numbers of connections.

---

6. Branches Missed

Connections	Single-Threaded	Multi-Threaded	Select (I/O Multiplexing)
10	7,985	8,401	8,071
50	15,819	17,388	16,153
100	25,501	27,718	25,664

- **Observation**: Multi-threaded approach consistently shows more branch misses, indicating increased complexity in branch prediction. The select system call performs better than multi-threaded in this regard but is still slightly higher than the single-threaded approach.

---

7. Branch Miss Rate (%)

Connections	Single-Threaded	Multi-Threaded	Select (I/O Multiplexing)
10	9.84	10.26	9.74
50	11.39	11.80	11.49
100	12.08	12.11	12.06

- **Observation**: The branch miss rate increases as the number of connections increases for all three implementations. However, the single-threaded approach shows the most stable performance, while the multi-threaded and select system calls have slightly higher branch miss rates.

---

8. Elapsed Time (sec)

Connections	Single-Threaded	Multi-Threaded	Select (I/O Multiplexing)
10	0.017	0.020	0.019
50	0.082	0.092	0.086
100	0.167	1.14	0.167

- **Observation**: The elapsed time for multi-threaded implementation spikes dramatically at 100 connections, taking significantly longer than both single-threaded and select system call approaches. Both single-threaded and select handle larger connections much faster and are nearly identical in performance at 100 connections.

Conclusions

1. **Single-Threaded**: It maintains stable performance across all metrics, and handles low to moderate connections well. However, it is not the most scalable option for high connection loads due to its single-thread nature.
2. **Multi-Threaded**: This approach is ideal for lower connection levels as it efficiently utilizes CPU and memory resources. However, for higher connections, it experiences a drastic increase in cycles, instructions, and branch misses, and takes significantly longer to complete tasks.
3. **Select (I/O Multiplexing)**: This implementation proves to be the most efficient for high connections. While it incurs slightly more overhead for lower connections, it scales much better as connections increase, utilizing fewer cycles, completing tasks faster, and maintaining a stable branch miss rate.

Final Analysis:

- For smaller client-server applications, the **single-threaded** approach is efficient and simple to implement.
- For low to medium connections, **multi-threaded** implementations work well.
- For handling large numbers of connections, the **select system call** (I/O multiplexing) offers the best performance and scalability, making it the most suitable for high-load server environments.