Sweta Snigdha
2022527

Syam Sai Santosh Bandi
2022528

CN Assignment 2

Q1

This implementation sets up a client-server socket program in C, where the server listens for multiple clients, handles them at the same time with multithreading, and sends back info on the top two processes using the most CPU on the server.

Server code (server.c):

- Socket Setup: The server makes a TCP socket with socket() and ties it to a port using bind(). Then it listens for clients with listen(). When a client connects, the server uses accept() to start handling it.
- Handling Clients: The server can work with lots of clients at the same time. It makes a new thread for each client with pthread_create(). This lets the server deal with one client, while others are still able to connect and talk with the server.
- Finding CPU Processes: The server reads from /proc/[pid]/stat to get CPU usage (stuff like time spent in user mode and kernel mode). It finds the two processes using the most CPU and sends that info back to the client.
- Sending Data: Once the server has the CPU data, it puts it into a string and sends it through the socket back to the client.

Client code (client.c):

- Socket Setup: The client makes a socket and connects to the server using a fixed IP and port (hardcoded as 8080).
- Concurrent Requests: The client can make a bunch of connections to the server. It creates multiple threads with pthread_create(), where each thread sends a request and waits for the CPU info from the server.
- Request and Print: After connecting, the client sends a simple request asking for the CPU processes using the most CPU. When it gets a reply, it prints it out to the console.

The taskset command is used so the server runs on CPU 0 and the client runs on CPU 1, making sure they're on different CPUs for testing and performance reasons.

Both the server and client use pthread for multithreading. On the server, each client gets handled in its own thread, so the server can manage a lot of clients at once. On the client, many threads are made to act like multiple clients all connecting to the server at the same time.

```
syam2004@Syam:~/CN/assignment-2$ gcc server.c
syam2004@Syam:~/CN/assignment-2$ ./a.out
pid 40977's current affinity list: 0-7
pid 40977's new affinity list: 0
fd: 6
Client: what are the top two cpu consuming processes?
fd: 5
Client: what are the top two cpu consuming processes?
fd: 4
Client: what are the top two cpu consuming processes?
fd: 7
Client: what are the top two cpu consuming processes?
fd: 5
Client: what are the top two cpu consuming processes?
fd: 4
Client: what are the top two cpu consuming processes?
fd: 7
Client: what are the top two cpu consuming processes?
fd: 6
Client: what are the top two cpu consuming processes?
```

```
syam2004@Syam:~/CN/assignment-2$ gcc client.c
syam2004@Syam:~/CN/assignment-2$ ./a.out 4
pid 42408's current affinity list: 0-7
pid 42408's new affinity list: 1
Server:
pid1:479
 user1: (node)
 user time: 8354 and kernel CPU time: 4720
pid2:1
 user2: (systemd)
 user time: 2984 and kernel CPU time: 371

Server:
pid1:479
 user1: (node)
 user time: 8354 and kernel CPU time: 4720
pid2:1
 user2: (systemd)
 user time: 2984 and kernel CPU time: 371

Server:
pid1:479
 user1: (node)
 user time: 8354 and kernel CPU time: 4720
pid2:1
 user2: (systemd)
 user time: 2984 and kernel CPU time: 371

Server:
pid1:479
 user1: (node)
 user time: 8354 and kernel CPU time: 4720
pid2:1
 user2: (systemd)
 user time: 2984 and kernel CPU time: 371
```

Q2

----

a. Single-Threaded TCP Client-Server

For the single threaded TCP Client-Server, I have assumed that the client can take n clients and run a function to handle the connections between those clients and server by looping through them without making any threads.

Server Code (server_single.c):
The server here is simple, only dealing with one connection at a time since it's single-threaded. When a client connects, the server handles it right away, grabs the top two CPU-consuming processes using get_top_two_cpu_processes, and sends that info back to the client.

Changes:

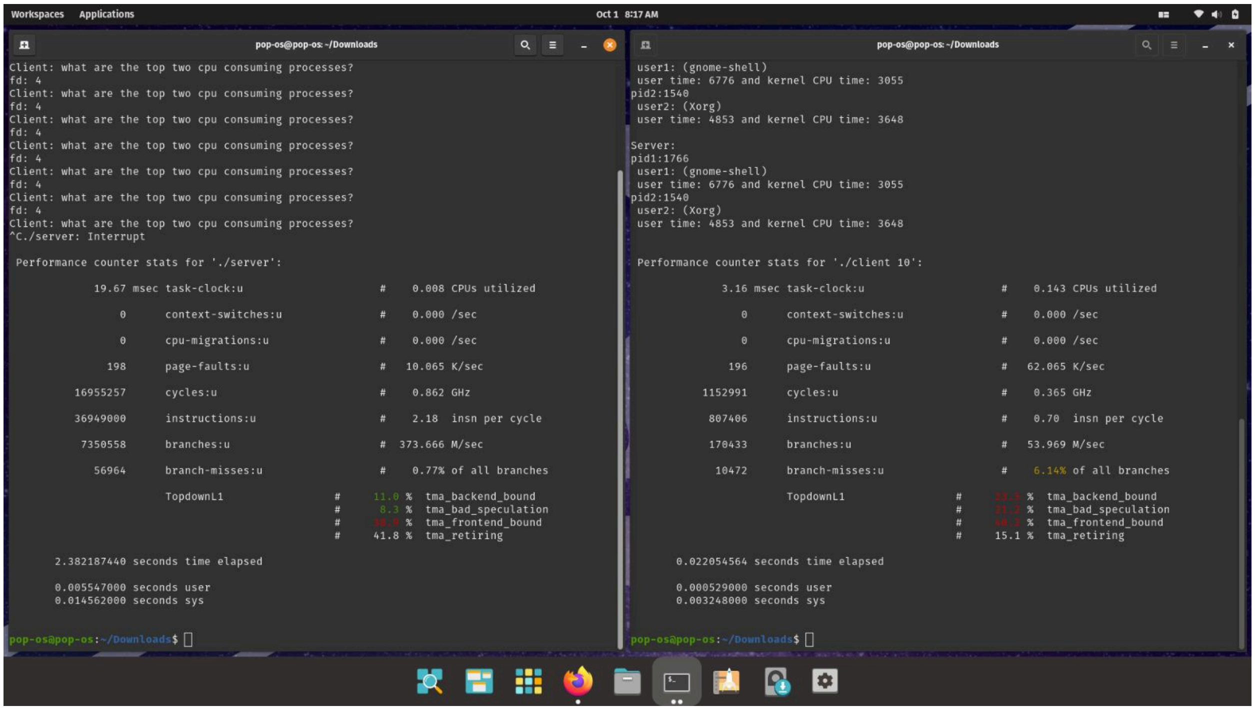
- Client Handling: The server just calls handle_client to deal with incoming client connections, without creating any new threads. This is different from multi-threaded models where each client gets a separate thread. Here, the server only deals with one client at a time, making it totally single-threaded and sequential.
- Taskset: The server gets pinned to CPU core 0 using the taskset command to keep performance consistent. This makes sure we measure performance on one core during the tests.
- Client Connections: The server accepts connections in a while loop and processes them one by one, no threads involved, which shows off true single-threaded behavior.

Client Code (client_single.c):
The client works like the one in q1, but instead of using multiple threads, it sends requests one by one.

Changes:

- Single-threaded Behavior: In client_connection, the client connects to the server and asks for the top two CPU-using processes. The server sends the data back, and the client prints it. It runs through multiple clients one by one in a loop but doesn't create threads, so it's all sequential.
- Taskset: Just like the server, the client is also pinned to CPU core 1 with taskset to keep performance steady.

Perf Analysis:



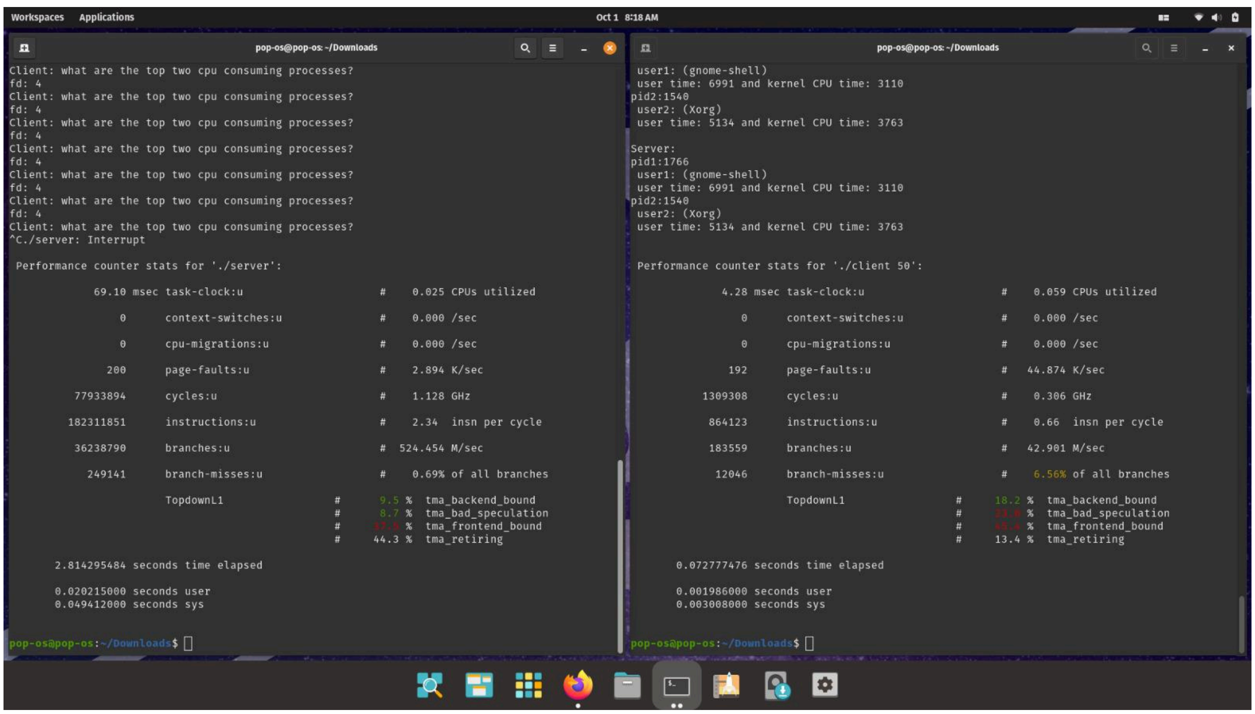
Server: perf stat ./server

- Task clock: 19.67 msec
  The CPU spent 19.67 milliseconds executing the server program, indicating quick completion of the task.
- CPUs utilized: 0.008
  Very low CPU utilization, suggesting minimal load on the CPU by the server.
- Page faults: 198
  Indicates that 198 times, the program requested data not currently in memory, requiring disk access. The number is relatively small, indicating efficient memory usage.
- Cycles: 16,955,257 (0.862 GHz)
  The server consumed around 17 million CPU cycles, with the CPU running at a low frequency of 0.862 GHz, implying the task didn't heavily tax the CPU.
- Instructions per cycle (IPC): 2.18
  The server executed 2.18 instructions per cycle, which shows good efficiency, as more than 2 instructions were processed for each cycle.
- Branches: 7,350,558
  The total number of branch instructions executed was over 7.35 million.
- Branch misses: 56,964 (0.77% of branches)
  The branch miss rate is low at 0.77%, indicating efficient branch prediction by the CPU.

Client: perf stat ./client 10

- Task clock: 3.16 msec
  The CPU spent 3.16 milliseconds executing the client, indicating a much lighter workload compared to the server.
- CPUs utilized: 0.143
  Higher CPU utilization than the server, implying the client placed more load on the CPU.

- Page faults: 196
  Similar to the server, the client incurred 196 page faults, showing similar memory-to-disk operations.
- Cycles: 1,152,991 (0.365 GHz)
  The client consumed fewer CPU cycles (about 1.15 million) and ran at a lower frequency (0.365 GHz), indicating less CPU workload.
- Instructions per cycle (IPC): 0.70
  The client was less efficient, executing fewer instructions per cycle, possibly because it was waiting on I/O or other resources.
- Branches: 178,433
  A total of 178,433 branch instructions were executed, significantly fewer than the server.
- Branch misses: 10,472 (6.14% of branches)
  The branch miss rate is higher at 6.14%, suggesting less efficient branch prediction compared to the server.



Server: perf stat ./server

- Task clock: 69.10 msec
  The CPU spent more time, 69.10 milliseconds, executing the server program, indicating a higher workload during this run.
- CPUs utilized: 0.025
  CPU utilization increased slightly, but remained very low, showing that the server did not place much load on the CPU even under heavier workload.
- Page faults: 200
  There were 200 page faults, a slight increase from the previous run, reflecting more memory-to-disk activity.

- Cycles: 77,933,894 (1.128 GHz)
  The number of cycles increased significantly to 77.93 million, and the CPU ran at a higher frequency of 1.128 GHz, indicating increased processing demands.
- Instructions per cycle (IPC): 2.34
  The IPC improved to 2.34, showing better instruction execution efficiency compared to the first run.
- Branches: 36,237,890
  The number of branches increased significantly, consistent with the heavier workload.
- Branch misses: 249,141 (0.69% of branches)
  The branch miss rate dropped to 0.69%, showing improved branch prediction compared to the first run.

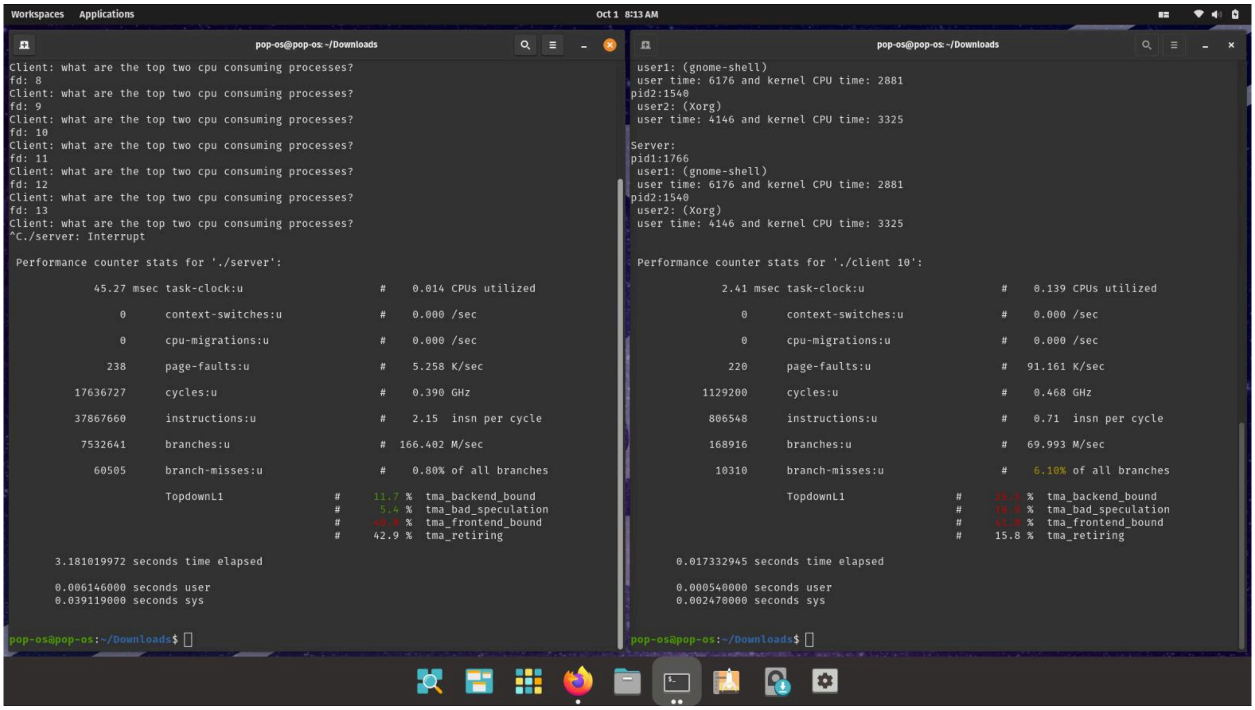
Client: perf stat ./client 50

- Task clock: 4.28 msec
  The CPU spent 4.28 milliseconds executing the client program with argument 50, indicating a slightly heavier workload than in the client 10 run.
- CPUs utilized: 0.059
  Lower CPU utilization than client 10, suggesting the client spent more time idle or waiting for resources during the task.
- Page faults: 192
  Slightly fewer page faults compared to the client 10 run, indicating consistent memory-to-disk behavior.
- Cycles: 1,309,308 (0.306 GHz)
  Fewer cycles were consumed (about 1.31 million), and the CPU frequency was lower at 0.306 GHz, implying more idle time or waiting for I/O.
- Instructions per cycle (IPC): 0.66
  The IPC is slightly lower than in client 10, reflecting reduced efficiency in processing instructions.
- Branches: 183,559
  The number of branch instructions was slightly higher than in client 10, reflecting a more complex workload.
- Branch misses: 12,046 (6.56% of branches)
  The branch miss rate was higher at 6.56%, indicating less efficient branch prediction.

b. Concurrent TCP Client-Server

For this part, we use the same server and client code as in q1, where the server can handle multiple clients at once using threads.

Perf Analysis:



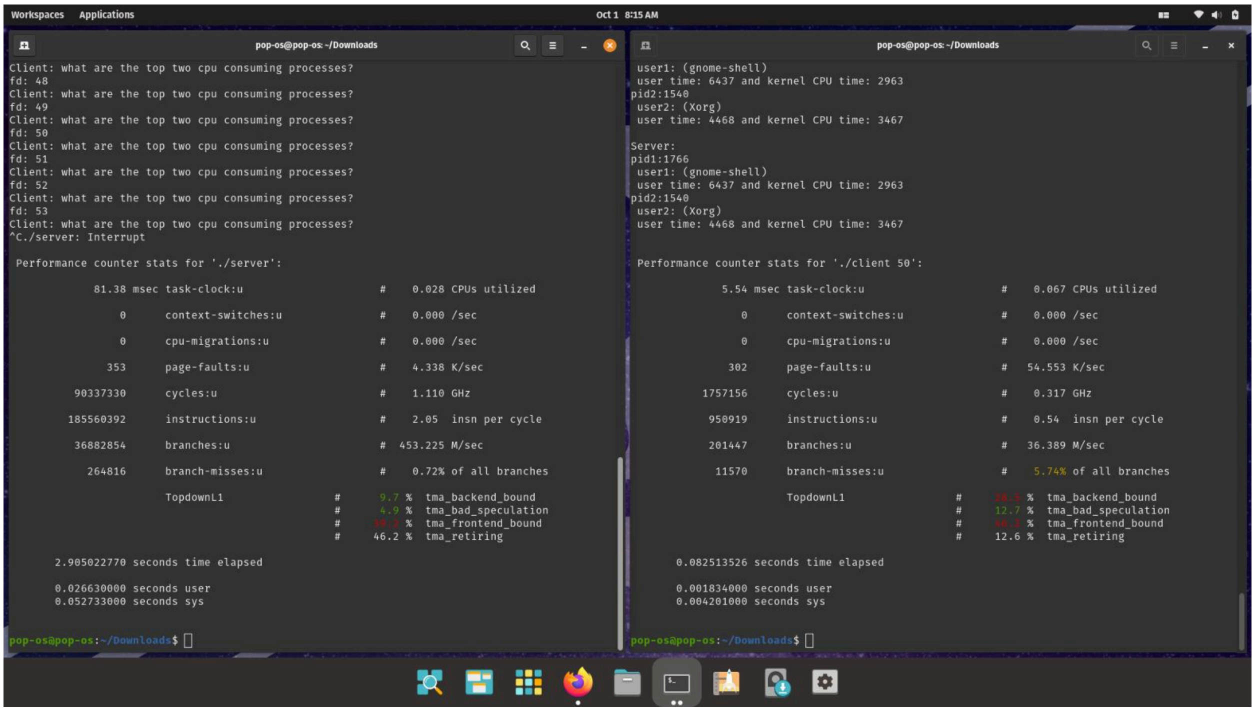
Server: perf stat ./server

- <u>Task clock:</u> 45.27 msec
  The server was running for 45.27 milliseconds, indicating a short task duration.
- <u>CPUs utilized:</u> 0.014
  The server used 1.4% of a single CPU core, which is quite low, meaning minimal load on the CPU.
- <u>Page faults:</u> 238
  There were 238 page faults, meaning the server had some memory access from disk, but not much.
- <u>Cycles:</u> 17,636,727 (0.390 GHz)
  The server used around 17.6 million cycles, with the CPU running at 0.390 GHz, indicating it wasn't working very hard.
- <u>Instructions per cycle (IPC):</u> 2.15
  The server achieved 2.15 instructions per cycle, showing good CPU efficiency.
- <u>Branches:</u> 7,532,641
  The server processed over 7.5 million branch instructions.
- <u>Branch misses:</u> 60,505 (0.80% of branches)
  The server's branch prediction was efficient, with only 0.80% branch misses.

Client: perf stat ./client 10

- <u>Task clock:</u> 2.41 msec
  The client ran for just 2.41 milliseconds, showing a light workload.
- <u>CPUs utilized:</u> 0.139
  The client used 13.9% of a CPU core, more than the server, suggesting a more active process.

- Page faults: 220
  The client had 220 page faults, similar to the server, so not much disk I/O.
- Cycles: 1,129,200 (0.468 GHz)
  The client used fewer cycles than the server, with the CPU running at 0.468 GHz.
- Instructions per cycle (IPC): 0.71
  The IPC was lower at 0.71, indicating the client wasn't as efficient as the server.
- Branches: 168,916
  The client executed 168,916 branch instructions, fewer than the server.
- Branch misses: 10,310 (6.10% of branches)
  The client had a higher branch miss rate of 6.10%, meaning branch prediction wasn't as accurate.



Server: perf stat ./server

- Task clock: 81.38 msec
  The server took longer, running for 81.38 milliseconds, showing it had a heavier workload.
- CPUs utilized: 0.028
  CPU usage increased to 2.8%, but still fairly low.
- Page faults: 353
  More page faults occurred (353), indicating increased memory access.
- Cycles: 90,337,330 (1.110 GHz)
  The CPU ran at a higher frequency (1.110 GHz), with 90.3 million cycles, showing the server was working harder.
- Instructions per cycle (IPC): 2.05
  The server's IPC remained efficient at 2.05, similar to the first result.
- Branches: 3,688,254
  The server processed more branches, reflecting its increased workload.

- Branch misses: 264,816 (0.72% of branches)
  The branch miss rate stayed low at 0.72%, indicating efficient branch prediction.

Client: perf stat ./client 50

- Task clock: 5.54 msec
  The client took longer, with a task clock of 5.54 milliseconds, indicating a larger workload than client 10.
- CPUs utilized: 0.067
  CPU usage decreased slightly to 6.7%, even though the task duration increased.
- Page faults: 302
  Fewer page faults (302) than in client 10, but still showing significant memory access.
- Cycles: 1,757,156 (0.317 GHz)
  The client's CPU ran at a lower frequency (0.317 GHz), using fewer cycles, suggesting it was more idle.
- Instructions per cycle (IPC): 0.54
  The IPC dropped to 0.54, showing the client was less efficient than client 10.
- Branches: 201,447
  The client executed more branch instructions than client 10, reflecting a more complex workload.
- Branch misses: 11,570 (5.74% of branches)
  The branch miss rate was similar to client 10 at 5.74%, meaning the inefficiency in branch prediction persisted.

c. TCP Client-Server using select

Server Code (server_select.c):
Here, the server uses the select system call to manage multiple clients at once, but in a single-threaded way. The select function lets the server watch multiple sockets (file descriptors) and checks if any of them are ready to read, write, or if there's an error. The code is pretty much based on the server code from the git repo that came with the question.

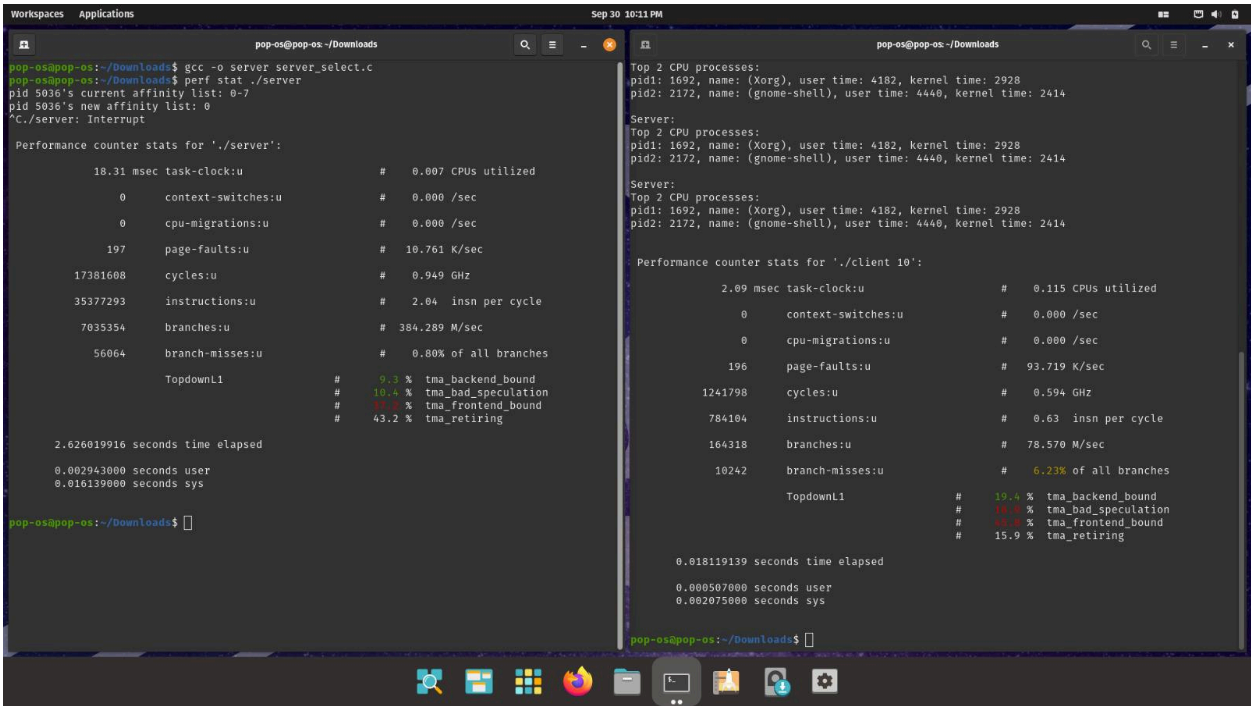
Changes:

- select System Call: Instead of making a thread for each client, the server uses select to watch all the client connections. select waits to see if there's any activity (data coming in) on any of the client sockets. It uses FD_SET, FD_ISSET, and FD_ZERO to manage these file descriptors and keep track of multiple clients, without making any new threads.
- Client Management: The server keeps track of each client using the client_sockets array. When a new client connects, its socket gets added to this array. The server watches these sockets with select to see when any client sends data, and processes it right when it arrives.
- Taskset: Like the other versions, the server is pinned to CPU core 0 using taskset to keep performance consistent during testing.

- Handling Client Requests: When select sees incoming data on any client's socket, the server reads the request and sends back the top two CPU-using processes. If a client disconnects, its socket gets removed from the client_sockets array.

Client Code:
Same client code as q2 a) is used here

Perf Analysis:



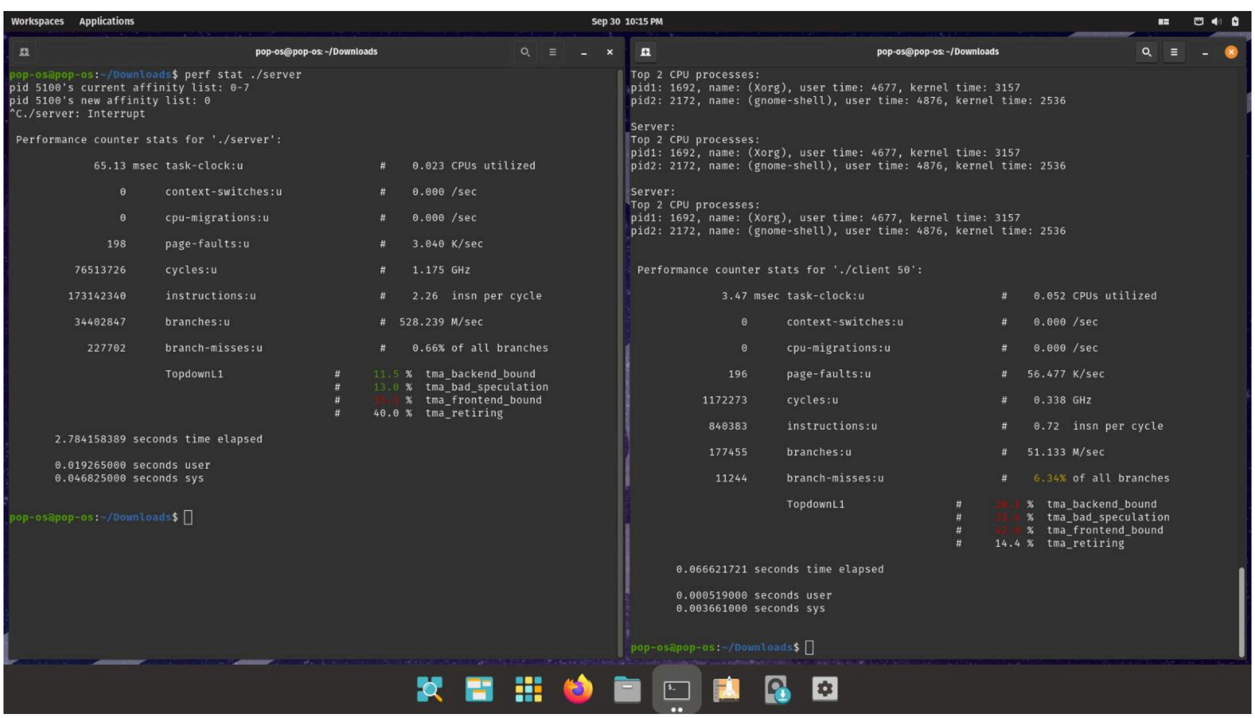
Server: perf stat ./server

- Task clock: 18.31 msec
  This is how long the CPU spent actually running the server process. A smaller number here means the program finished quicker.
- CPUs utilized: 0.007
  Shows how much of the CPU was actually being used. A low number like this means the server wasn't putting much load on the CPU.
- Page faults: 197
  This happens when the program tries to access data that's not in memory, so it has to get it from disk. 197 page faults means it's doing a bit of disk access, but not a lot.
- Cycles: 17,381,608 (0.949 GHz)
  This is how many CPU cycles it used. Less cycles usually means better performance. At 0.949 GHz, the CPU wasn't working too hard.
- Instructions per cycle (IPC): 2.04
  This shows how many instructions the CPU managed to execute for each cycle. An IPC of 2.04 is pretty good, means it's getting more than 2 instructions done per cycle.

- Branches: 7,035,354
  The total number of branch instructions executed was over 7 million.
- Branch misses: 56,064 (0.80% of branches)
  This is how many times the CPU guessed wrong when it had to pick which way the code was gonna go. 0.80% is quite low, meaning the CPU's pretty good at guessing right.

Client: perf stat ./client 10

- Task clock: 2.09 msec
  Client took less time, showing it's got a light workload.
- CPUs utilized: 0.115
  This is a bit higher than the server, so the client was putting slightly more load on the CPU.
- Page faults: 196
  Almost the same as the server, so a bit of disk I/O but not much.
- Cycles: 1,241,798 (0.594 GHz)
  Client used way fewer cycles and was running at a lower frequency, which shows it was doing less work.
- Instructions per cycle (IPC): 0.63
  This is lower than the server's IPC, so the client wasn't as efficient. Maybe it was waiting on I/O.
- Branches: 164,318
  A total of 164,318 branch instructions were executed, significantly fewer than the server.
- Branch misses: 10,242 (6.23% of branches)
  Higher than the server at 6.23%, so the client isn't as good at guessing branches.



Server: perf stat ./server

- Task clock: 65.13 msec
  The server took more time on this run, showing it was busier.
- CPUs utilized: 0.023
  CPU usage went up a little, but still low.
- Page faults: 198
  Just one more page fault compared to before, so not a big difference in memory access.
- Cycles: 76,513,726 (1.175 GHz)
  Way more cycles and a higher frequency, showing the server was working harder this time.
- Instructions per cycle (IPC): 2.26
  Better IPC than last time, so it's running more efficiently.
- Branches: 34,402,847
  The number of branches increased significantly, consistent with the heavier workload.
- Branch misses: 227,702 (0.66% of branches)
  Fewer branch misses this time, so the CPU was even better at guessing correctly.

Client: perf stat ./client 50

- Task clock: 3.47 msec
  Client took more time than when running with client 10, so it had a bigger workload.
- CPUs utilized: 0.052
  CPU usage went down compared to client 10, meaning the client was less busy even though it took longer.
- Page faults: 196
  Same page faults as before, so no big change in disk I/O.
- Cycles: 1,172,273 (0.338 GHz)
  Lower cycles and frequency, showing the client was more idle or waiting around.
- Instructions per cycle (IPC): 0.72
  Slightly better IPC than client 10, but still not super efficient.
- Branches: 177,455
  The number of branch instructions was slightly higher than in client 10, reflecting a more complex workload.
- Branch misses: 11,244 (6.34% of branches)
  Similar branch misses to before, so still some inefficiency in branching.

Key Comparisons and Analysis:

Task Time:

The concurrent model took the longest to finish tasks, with more branch misses for both server and client compared to the select model. The single-threaded model had a noticeable jump in task time with heavier workloads, mostly due to the fact that it processes client requests one at a time.

CPU Utilization:
CPU usage stayed low across all models, for both servers and clients. The concurrent model managed multiple clients more efficiently thanks to multithreading, but the select model kept similarly low utilization and scaled better without the extra complexity of threads. The single-threaded model, even though it used less CPU, really struggled when dealing with more than one client, which makes it kinda useless for real-world cases where you'd have multiple clients at once.

Memory Efficiency:
Memory usage was solid across the board, with low page faults everywhere. Not much difference between the single-threaded, concurrent, or select models when it came to memory performance, meaning they all handled memory and disk I/O well no matter how they dealt with clients.

Efficiency (IPC):
The select model showed the best instruction-per-cycle efficiency at 2.26 IPC, while the single-threaded model was close behind with 2.34 IPC in its first run. This shows the select model handles multiple clients well without needing to create a bunch of threads, while still keeping execution efficient.

Branch Prediction:
Both the single-threaded and select models kept branch miss rates low, showing good branch prediction. The concurrent model also did well, but as more clients came in, managing all those threads caused a slight bump in branch misses. Clients, in all models, had higher branch miss rates, likely 'cause they were waiting on responses and dealing with varying server times, especially in the concurrent and select models.