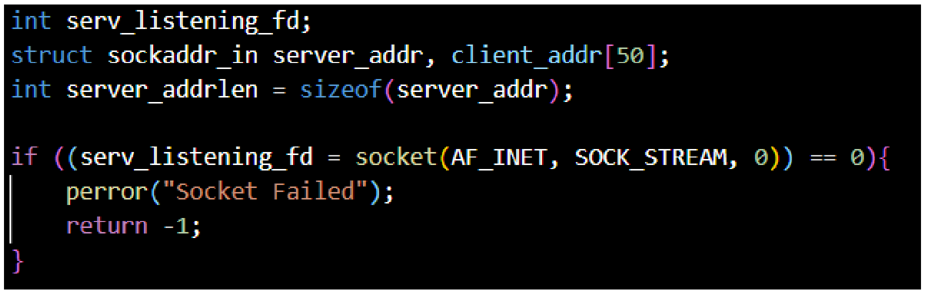
CN Assignment02

Syamantak Paliwal(2022529), Tanish Verma(2022532)

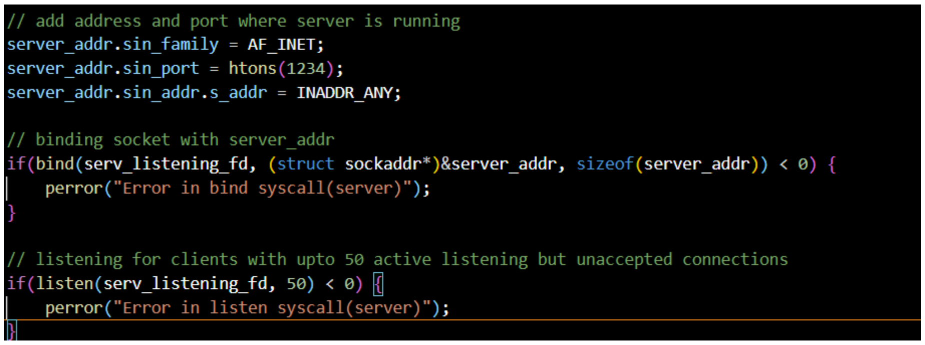
September 2024

1 Question 1

1. Using *socket()*, *bind()* and *listen()* system calls in the server.c code to set up a TCP socket. [Port- 1234]



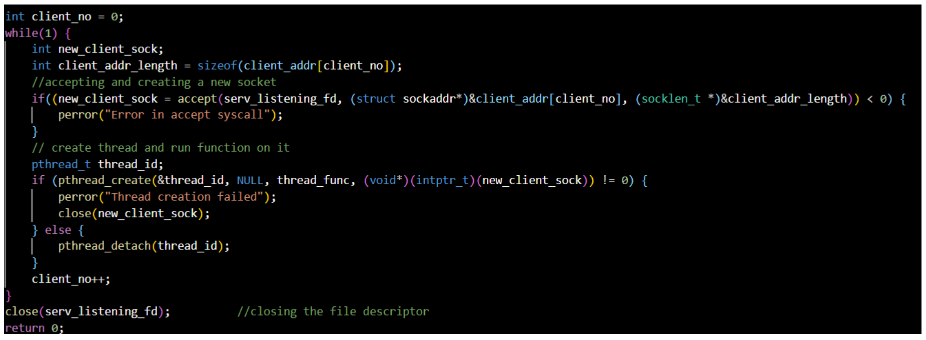
```
int serv_listening_fd;
struct sockaddr_in server_addr, client_addr[50];
int server_addrlen = sizeof(server_addr);

if ((serv_listening_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0){
    perror("Socket Failed");
    return -1;
}
```



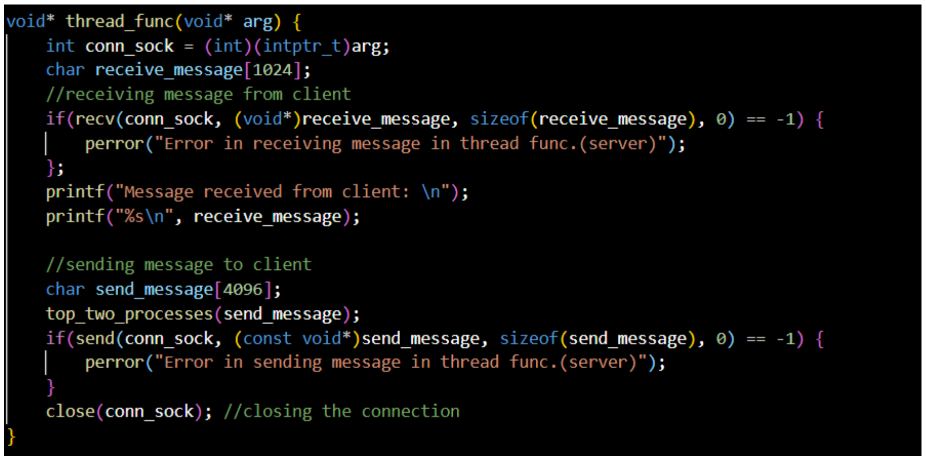
```
// add address and port where server is running
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(1234);
server_addr.sin_addr.s_addr = INADDR_ANY;

// binding socket with server_addr
if(bind(serv_listening_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
    perror("Error in bind syscall(server)");
}

// listening for clients with upto 50 active listening but unaccepted connections
if(listen(serv_listening_fd, 50) < 0) {
    perror("Error in listen syscall(server)");
}
```

2. Creating a new thread and executing it using *pthread* whenever a new client connection is established using *accept()* system call, while the server is listening forever on the port in a loop.

Each new thread created receives and sends messages, using *recv()* and *send()*

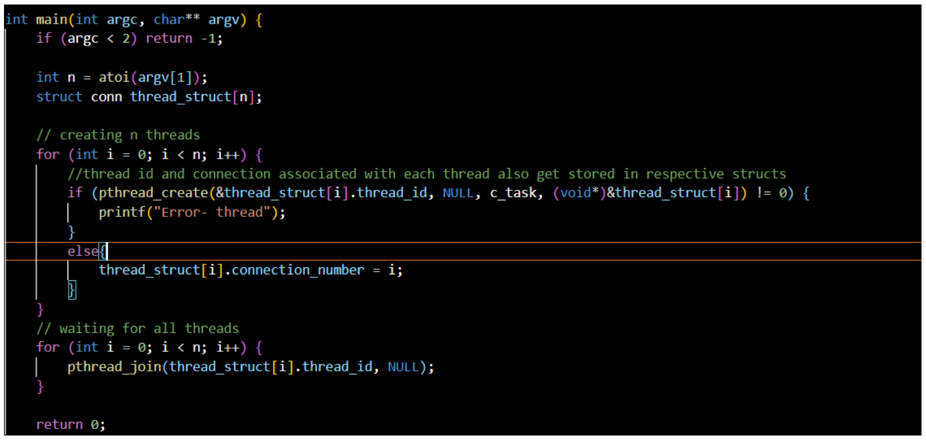


```c
int client_no = 0;
while(1) {
    int new_client_sock;
    int client_addr_length = sizeof(client_addr[client_no]);
    //accepting and creating a new socket
    if((new_client_sock = accept(serv_listening_fd, (struct sockaddr*)&client_addr[client_no], (socklen_t *)&client_addr_length)) < 0) {
        perror("Error in accept syscall");
    }
    // create thread and run function on it
    pthread_t thread_id;
    if (pthread_create(&thread_id, NULL, thread_func, (void*)(intptr_t)(new_client_sock)) != 0) {
        perror("Thread creation failed");
        close(new_client_sock);
    } else {
        pthread_detach(thread_id);
    }
    client_no++;
}
close(serv_listening_fd);          //closing the file descriptor
return 0;
```



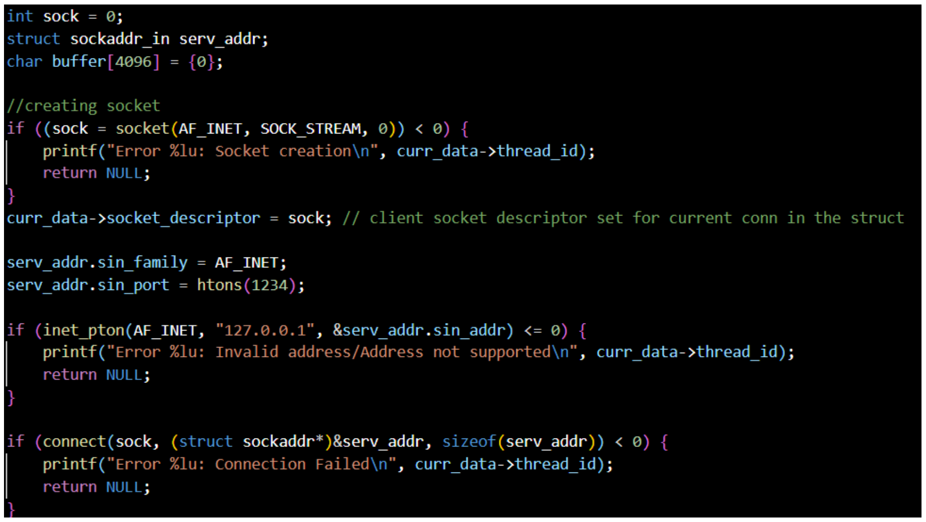
```c
void* thread_func(void* arg) {
    int conn_sock = (int)(intptr_t)arg;
    char receive_message[1024];
    //receiving message from client
    if(recv(conn_sock, (void*)receive_message, sizeof(receive_message), 0) == -1) {
        perror("Error in receiving message in thread func.(server)");
    };
    printf("Message received from client: \n");
    printf("%s\n", receive_message);

    //sending message to client
    char send_message[4096];
    top_two_processes(send_message);
    if(send(conn_sock, (const void*)send_message, sizeof(send_message), 0) == -1) {
        perror("Error in sending message in thread func.(server)");
    }
    close(conn_sock); //closing the connection
}
```

3. Implemented n concurrent client connection requests, with the help of command line arguments and pthread library.



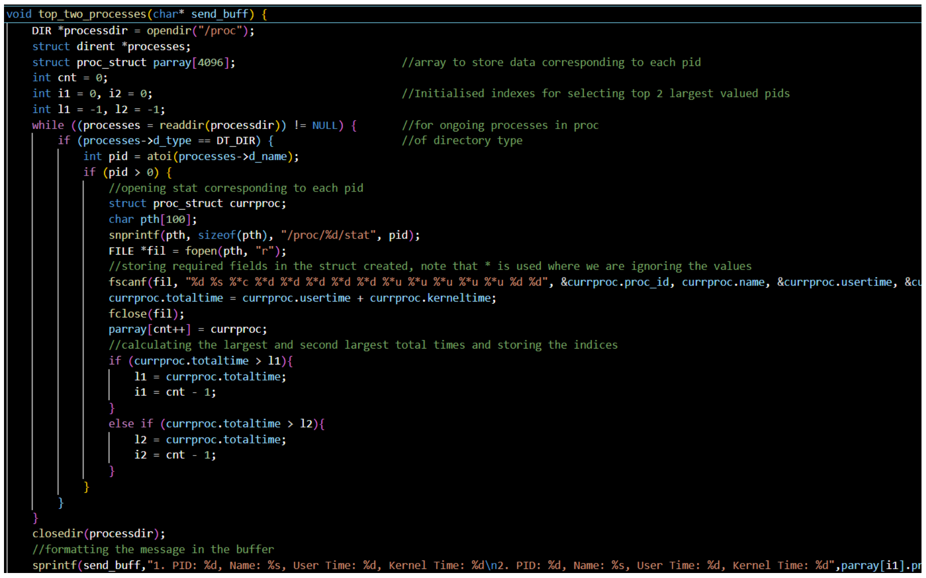
```c
int main(int argc, char** argv) {
    if (argc < 2) return -1;

    int n = atoi(argv[1]);
    struct conn thread_struct[n];

    // creating n threads
    for (int i = 0; i < n; i++) {
        //thread id and connection associated with each thread also get stored in respective structs
        if (pthread_create(&thread_struct[i].thread_id, NULL, c_task, (void*)&thread_struct[i]) != 0) {
            printf("Error- thread");
        }
        else{
            thread_struct[i].connection_number = i;
        }
    }
    // waiting for all threads
    for (int i = 0; i < n; i++) {
        pthread_join(thread_struct[i].thread_id, NULL);
    }

    return 0;
```

Socket creation and connection:



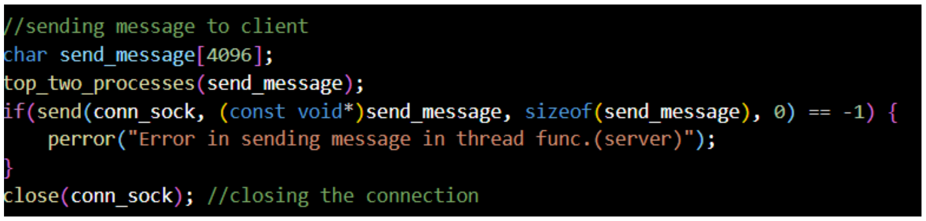
```c
int sock = 0;
struct sockaddr_in serv_addr;
char buffer[4096] = {0};

//creating socket
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    printf("Error %lu: Socket creation\n", curr_data->thread_id);
    return NULL;
}
curr_data->socket_descriptor = sock; // client socket descriptor set for current conn in the struct

serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(1234);

if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
    printf("Error %lu: Invalid address/Address not supported\n", curr_data->thread_id);
    return NULL;
}

if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
    printf("Error %lu: Connection Failed\n", curr_data->thread_id);
    return NULL;
}
```

4. Created a new function *toptwoprocesses()* to do this, by opening the *"/proc"* directory and reading the relevant information from the stat file corresponding to each pid. An algorithm to calculate the two largest times was also implemented.



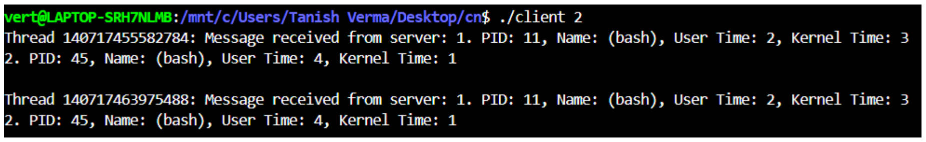
```
void top_two_processes(char* send_buff) {
    DIR *processdir = opendir("/proc");
    struct dirent *processes;
    struct proc_struct parray[4096];                         //array to store data corresponding to each pid
    int cnt = 0;
    int i1 = 0, i2 = 0;                                      //Initialised indexes for selecting top 2 largest valued pids
    int l1 = -1, l2 = -1;
    while ((processes = readdir(processdir)) != NULL) {      //for ongoing processes in proc
        if (processes->d_type == DT_DIR) {                   //of directory type
            int pid = atoi(processes->d_name);
            if (pid > 0) {
                //opening stat corresponding to each pid
                struct proc_struct currproc;
                char pth[100];
                snprintf(pth, sizeof(pth), "/proc/%d/stat", pid);
                FILE *fil = fopen(pth, "r");
                //storing required fields in the struct created, note that * is used where we are ignoring the values
                fscanf(fil, "%d %s %*c %*d %*d %*d %*d %*d %*u %*u %*u %*u %*u %d %d", &currproc.proc_id, currproc.name, &currproc.usertime, &cu
                currproc.totaltime = currproc.usertime + currproc.kerneltime;
                fclose(fil);
                parray[cnt++] = currproc;
                //calculating the largest and second largest total times and storing the indices
                if (currproc.totaltime > l1){
                    l1 = currproc.totaltime;
                    i1 = cnt - 1;
                }
                else if (currproc.totaltime > l2){
                    l2 = currproc.totaltime;
                    i2 = cnt - 1;
                }
            }
        }
    }
    closedir(processdir);
    //formatting the message in the buffer
    sprintf(send_buff,"1. PID: %d, Name: %s, User Time: %d, Kernel Time: %d\n2. PID: %d, Name: %s, User Time: %d, Kernel Time: %d",parray[i1].pr
```

5. The thread sends the information to client using *send()*



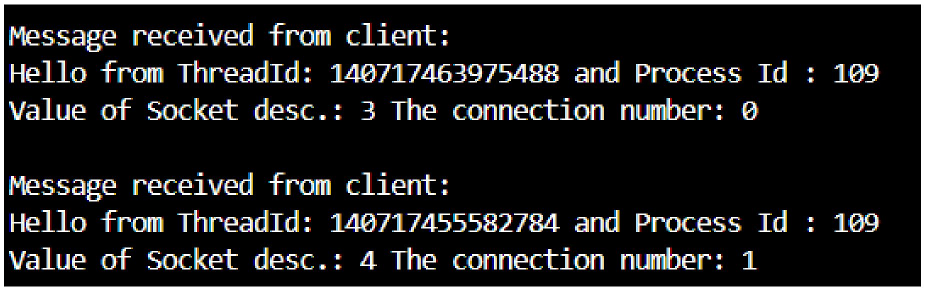
```
//sending message to client
char send_message[4096];
top_two_processes(send_message);
if(send(conn_sock, (const void*)send_message, sizeof(send_message), 0) == -1) {
    perror("Error in sending message in thread func.(server)");
}
close(conn_sock); //closing the connection
```

6. Output at client side:



```
vert@LAPTOP-SRH7NLMB:/mnt/c/Users/Tanish Verma/Desktop/cn$ ./client 2
Thread 140717455582784: Message received from server: 1. PID: 11, Name: (bash), User Time: 2, Kernel Time: 3
2. PID: 45, Name: (bash), User Time: 4, Kernel Time: 1

Thread 140717463975488: Message received from server: 1. PID: 11, Name: (bash), User Time: 2, Kernel Time: 3
2. PID: 45, Name: (bash), User Time: 4, Kernel Time: 1
```

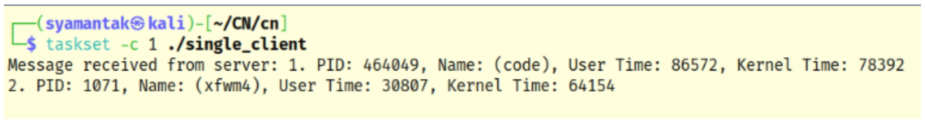
Output at server side:



```
Message received from client:
Hello from ThreadId: 140717463975488 and Process Id : 109
Value of Socket desc.: 3 The connection number: 0

Message received from client:
Hello from ThreadId: 140717455582784 and Process Id : 109
Value of Socket desc.: 4 The connection number: 1
```

2 Question 2

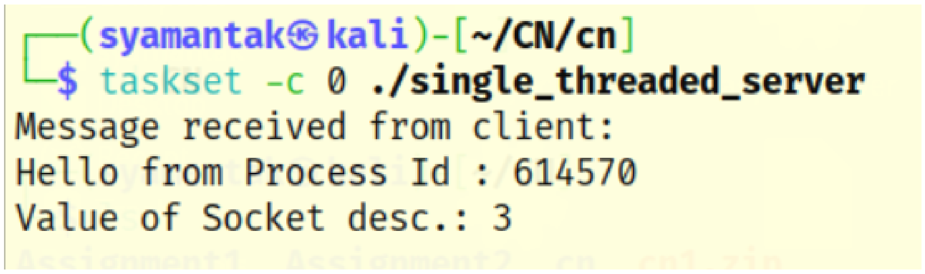
Part A
Figure 1 shows single threaded server returning information about top 2 CPU consuming processes. Their process ids, user time and kernel time are given. Note that we have fixed client to run on core 1.



```
┌──(syamantak㉿kali)-[~/CN/cn]
└─$ taskset -c 1 ./single_client
Message received from server: 1. PID: 464049, Name: (code), User Time: 86572, Kernel Time: 78392
2. PID: 1071, Name: (xfwm4), User Time: 30807, Kernel Time: 64154
```

Figure 1: Client receiving message from server

Figure 2 shows single threaded server communicating with client.

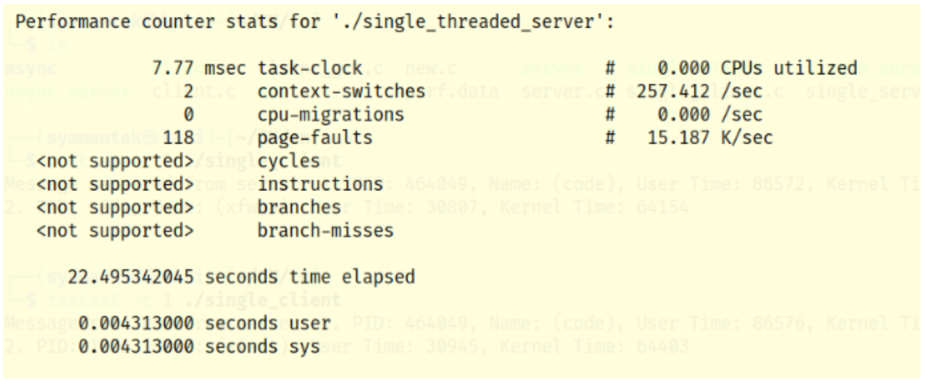


```
┌──(syamantak㉿kali)-[~/CN/cn]
└─$ taskset -c 0 ./single_threaded_server
Message received from client:
Hello from Process Id :614570
Value of Socket desc.: 3
```

Figure 2: Single Threaded Server

Figure 3 shows performance statistics of single threaded server and a single client.
The task clock specifies the CPU time spent on the task, which was 7.77 msec. Context switching happened 2 times, since no concurrency. 0 CPU migrations occurred (as the process was fixed to a specific CPU core). There were 118 page faults during the execution.
User mode time was 0.004313 seconds, and kernel mode time was also 0.004313

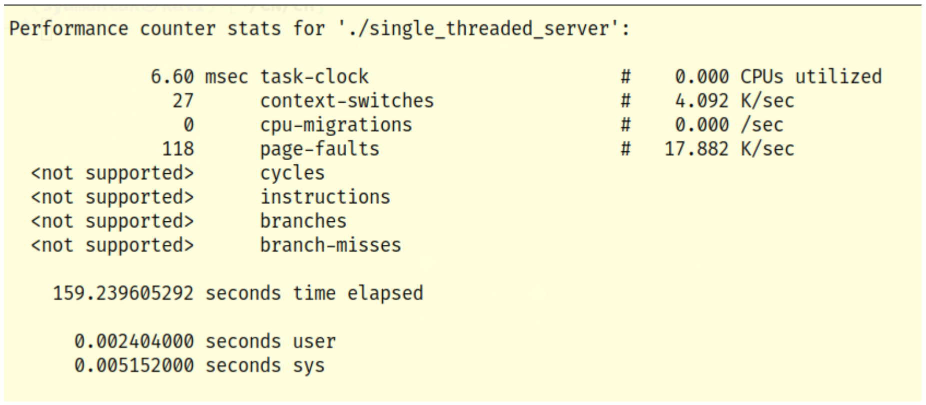
seconds. The task didn't utilize much CPU time, as it was primarily waiting for client.



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

              7.77 msec task-clock                       #    0.000 CPUs utilized
                 2      context-switches                 #  257.412 /sec
                 0      cpu-migrations                   #    0.000 /sec
               118      page-faults                      #   15.187 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

      22.495342045 seconds time elapsed

       0.004313000 seconds user
       0.004313000 seconds sys
```

Figure 3: Single Threaded Server with Single Client

In case where multiple clients are connecting to the single threaded server: Multiple clients increase the context-switching (27 now instead of 2), this is to be observed since now a single threaded server is responsible for dealing with multiple client connections, but the page faults, and CPU utilization remain the same. The kernel time increased, which could suggest more frequent kernel-level operations, likely a result of the high number of context switches. Whereas the user-space time decreased.



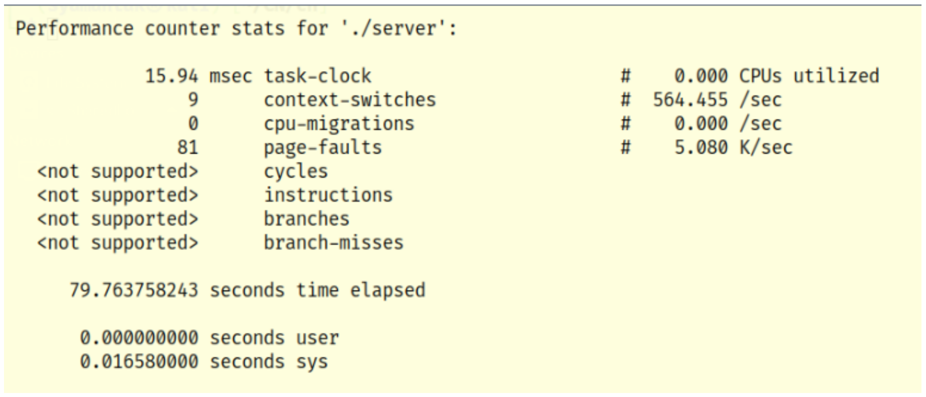
```
 Performance counter stats for './single_threaded_server':

              6.60 msec task-clock                       #    0.000 CPUs utilized
                27      context-switches                 #    4.092 K/sec
                 0      cpu-migrations                   #    0.000 /sec
               118      page-faults                      #   17.882 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

     159.239605292 seconds time elapsed

       0.002404000 seconds user
       0.005152000 seconds sys
```

Figure 4: Enter Caption

Part B

Figure 5 shows the client initiating 3 TCP connections, and the server is returning information about the top 2 CPU-consuming processes.

The performance statistics of the multi-threaded server are as follows. The task clock specifies the CPU time spent on the task, which was 15.94 msec. Context switching happened 9 times, which suggests concurrency. Along with that, 0 CPU migrations occurred (as we fixed the CPU core for the process). There were 81 page faults while running 3 TCP connections on different threads.
The user mode time was 0 seconds, and the kernel mode time was 0.01658 seconds. Since the task didn't use much CPU time, it was primarily waiting for clients.



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

           15.94 msec task-clock                       #    0.000 CPUs utilized
               9      context-switches                 #  564.455 /sec
               0      cpu-migrations                   #    0.000 /sec
              81      page-faults                      #    5.080 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

     79.763758243 seconds time elapsed

      0.000000000 seconds user
      0.016580000 seconds sys
```

Figure 5: Multi-threaded Server Performance

Part C
Performance statistics of the event-driven server.
It shows there were 4 context switches, and the task clock specifies the CPU time spent on the task, which was 21.73 msec. Context switching happened 4 times, which suggests some concurrency. As with the previous case, 0 CPU migrations occurred (since we fixed the CPU core for the process). There were 119 page faults while handling multiple TCP connections asynchronously
User mode time was 0.005375 seconds, and kernel mode time was 0.017355 seconds. The task also didn't use much CPU time, as it was primarily waiting for network events and handling connections in a non-blocking manner.

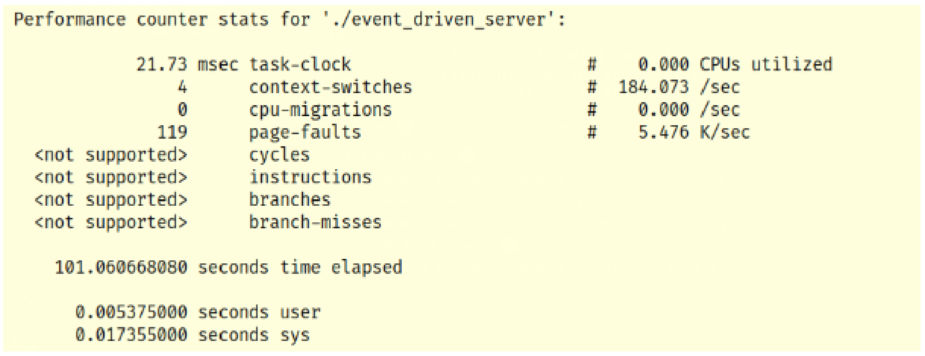

```
Performance counter stats for './event_driven_server':

          21.73 msec task-clock                #    0.000 CPUs utilized
              4      context-switches          #  184.073 /sec
              0      cpu-migrations            #    0.000 /sec
            119      page-faults               #    5.476 K/sec
  <not supported>      cycles
  <not supported>      instructions
  <not supported>      branches
  <not supported>      branch-misses

    101.060668080 seconds time elapsed

      0.005375000 seconds user
      0.017355000 seconds sys
```

Figure 6: Event-Driven (Select syscall) Performance

Overall
The single-threaded server shows no concurrency(only 2 context switches if only 1 client used, but for uniform comparison we will deal with 3 concurrent client connections) so it shows 27 context switches (the reason for higher context switches in single threaded server is that due to lack of concurrency, all work is being done on a single thread, causing more switches on it).
In contrast, the multi-threaded server manages 3 TCP connections and demonstrates more concurrency, featuring 9 context switches and its kernel mode time (0.01658 seconds) is due to the overhead from managing multiple threads, but it is higher in single threaded since it is heavily burdened.
The event-driven server, which handles connections asynchronously, has a moderate concurrency level, with 4 context switches and 119 page faults. Its task clock is longer at 21.73 msec, and the user mode time (0.005375 seconds) is slightly higher than multi-threaded server.
The event-driven server shows fewer context switches and uses fewer resources than the multi-threaded model, demonstrating its ability to handle concurrency efficiently without multiple threads. But it also has the highest kernel time, since it utilises the select system call