Day – 11 LSP : Test Questions

```
// Time : 2:10 p.m. - 3:10 p.m.
```

A. Task

1. **Design and implement a robust, distributed system using C++ that effectively leverages signals, sockets, and inter-process communication (IPC) to manage and coordinate multiple processes for a real-time data processing pipeline.**

System Requirements:

a. **Data Ingestion: Continuously receive data from multiple sources (e.g., network sockets, files, sensors) and distribute it across multiple worker processes.**

b. **Data Processing: Distribute incoming data to multiple worker processes, each responsible for specific data transformations or calculations.**

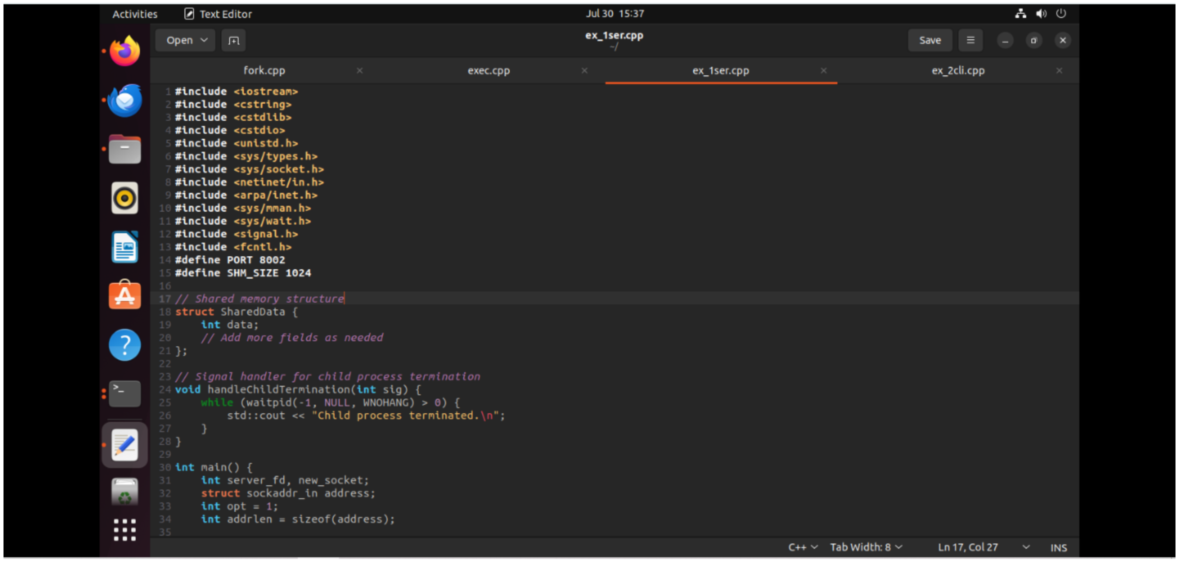
c. **Error Handling: Implement robust error handling mechanisms using signals to gracefully handle unexpected events (e.g., process termination, network failures).**

d. **Inter-Process Communication: Utilize IPC (e.g., shared memory, message queues) for efficient communication and synchronization between processes.**

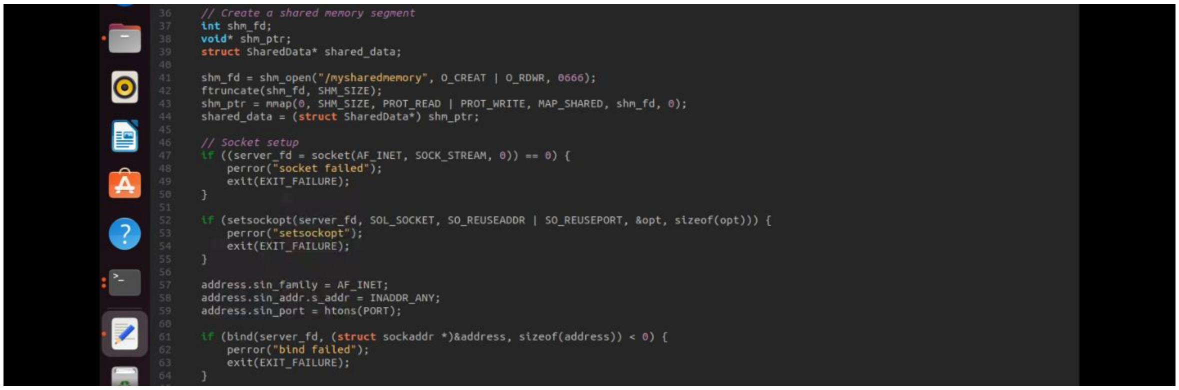
e. **Performance Optimization: Optimize the system for low latency and high throughput, considering factors like network congestion, process scheduling, and data transfer efficiency.**

f. **Scalability: Design the system to handle increasing data volumes and processing load by dynamically adjusting the number of worker processes.**

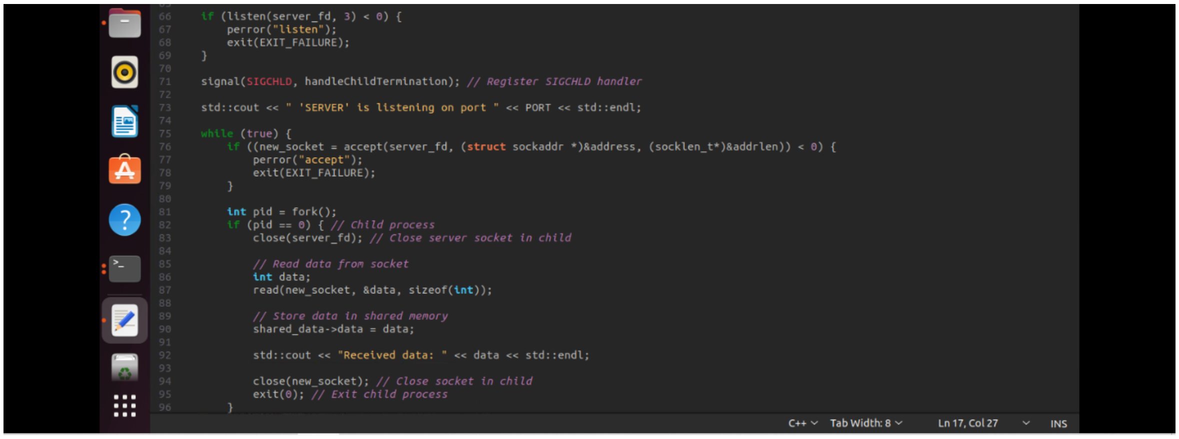
   a. **Server – side**



```cpp
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <cstdio>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <signal.h>
#include <fcntl.h>
#define PORT 8002
#define SHM_SIZE 1024

// Shared memory structure
struct SharedData {
    int data;
    // Add more fields as needed
};

// Signal handler for child process termination
void handleChildTermination(int sig) {
    while (waitpid(-1, NULL, WNOHANG) > 0) {
        std::cout << "Child process terminated.\n";
    }
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);

```

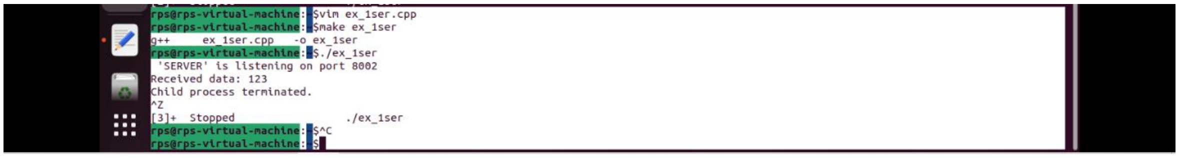


```cpp
    // Create a shared memory segment
    int shm_fd;
    void* shm_ptr;
    struct SharedData* shared_data;

    shm_fd = shm_open("/mysharedmemory", O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, SHM_SIZE);
    shm_ptr = mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    shared_data = (struct SharedData*) shm_ptr;

    // Socket setup
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))) {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
```



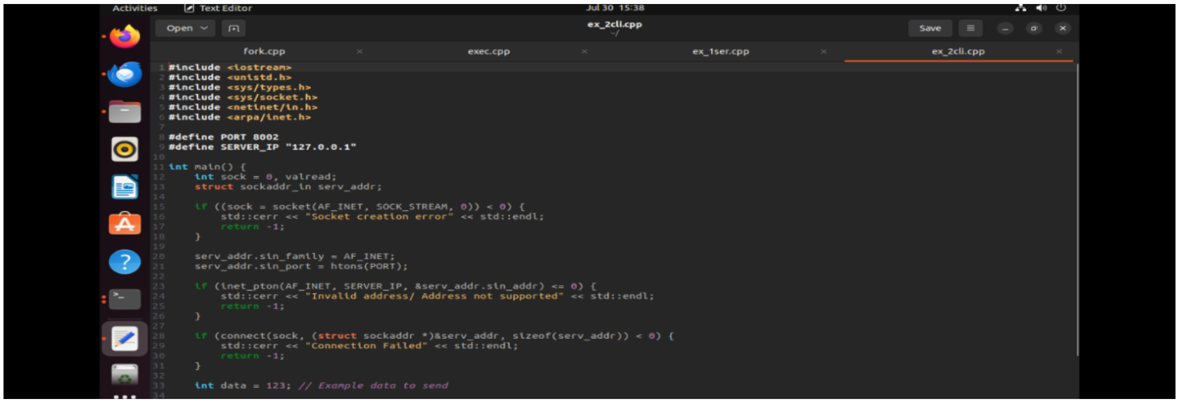
```cpp
    if (listen(server_fd, 3) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    signal(SIGCHLD, handleChildTermination); // Register SIGCHLD handler

    std::cout << " 'SERVER' is listening on port " << PORT << std::endl;

    while (true) {
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen)) < 0) {
            perror("accept");
            exit(EXIT_FAILURE);
        }

        int pid = fork();
        if (pid == 0) { // Child process
            close(server_fd); // Close server socket in child

            // Read data from socket
            int data;
            read(new_socket, &data, sizeof(int));

            // Store data in shared memory
            shared_data->data = data;

            std::cout << "Received data: " << data << std::endl;

            close(new_socket); // Close socket in child
            exit(0); // Exit child process
        }
```

Execution :

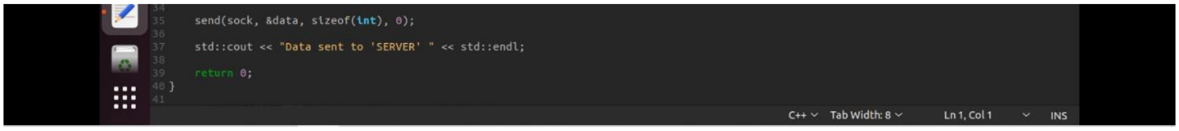


```
rps@rps-virtual-machine:~$vim ex_1ser.cpp
rps@rps-virtual-machine:~$make ex_1ser
g++     ex_1ser.cpp   -o ex_1ser
rps@rps-virtual-machine:~$./ex_1ser
 'SERVER' is listening on port 8002
Received data: 123
Child process terminated.
^Z
[3]+  Stopped                 ./ex_1ser
rps@rps-virtual-machine:~$^C
rps@rps-virtual-machine:~$
```

b. Client – side

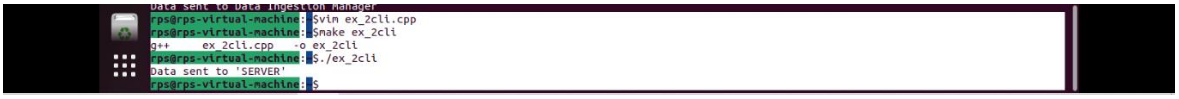


```cpp
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 8002
#define SERVER_IP "127.0.0.1"

int main() {
    int sock = 0, valread;
    struct sockaddr_in serv_addr;

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        std::cerr << "Socket creation error" << std::endl;
        return -1;
    }

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

    if (inet_pton(AF_INET, SERVER_IP, &serv_addr.sin_addr) <= 0) {
        std::cerr << "Invalid address/ Address not supported" << std::endl;
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        std::cerr << "Connection Failed" << std::endl;
        return -1;
    }

    int data = 123; // Example data to send
```



```cpp
    send(sock, &data, sizeof(int), 0);

    std::cout << "Data sent to 'SERVER' " << std::endl;

    return 0;
}
```

Execution :



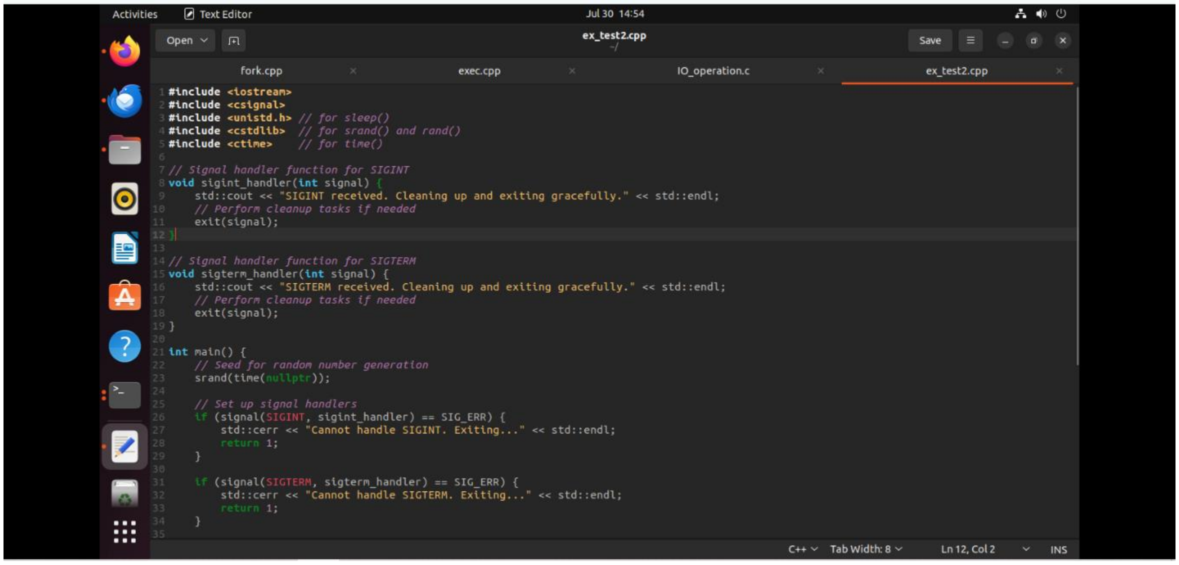
```
Data sent to Data Ingestion Manager
rps@rps-virtual-machine:~$vim ex_2cli.cpp
rps@rps-virtual-machine:~$make ex_2cli
g++     ex_2cli.cpp   -o ex_2cli
rps@rps-virtual-machine:~$./ex_2cli
Data sent to 'SERVER'
rps@rps-virtual-machine:~$
```

B. Coding Questions in C++

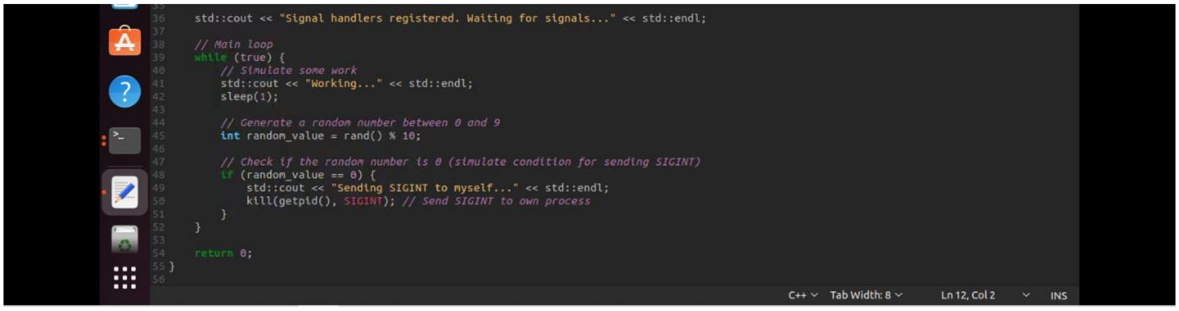
1. Signal Handling:

Write a C++ program that sets up a signal handler for SIGINT. The program should perform some tasks and print a message when SIGINT is caught, then terminate gracefully.

How would you modify your program to handle multiple different signals, each with a unique handling function?

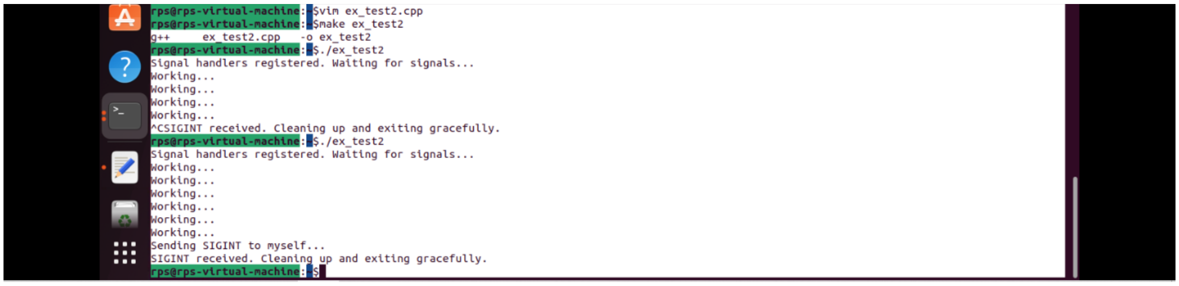


```cpp
#include <iostream>
#include <csignal>
#include <unistd.h> // for sleep()
#include <cstdlib>  // for srand() and rand()
#include <ctime>    // for time()

// Signal handler function for SIGINT
void sigint_handler(int signal) {
    std::cout << "SIGINT received. Cleaning up and exiting gracefully." << std::endl;
    // Perform cleanup tasks if needed
    exit(signal);
}

// Signal handler function for SIGTERM
void sigterm_handler(int signal) {
    std::cout << "SIGTERM received. Cleaning up and exiting gracefully." << std::endl;
    // Perform cleanup tasks if needed
    exit(signal);
}

int main() {
    // Seed for random number generation
    srand(time(nullptr));

    // Set up signal handlers
    if (signal(SIGINT, sigint_handler) == SIG_ERR) {
        std::cerr << "Cannot handle SIGINT. Exiting..." << std::endl;
        return 1;
    }

    if (signal(SIGTERM, sigterm_handler) == SIG_ERR) {
        std::cerr << "Cannot handle SIGTERM. Exiting..." << std::endl;
        return 1;
    }

```



```cpp
    std::cout << "Signal handlers registered. Waiting for signals..." << std::endl;

    // Main loop
    while (true) {
        // Simulate some work
        std::cout << "Working..." << std::endl;
        sleep(1);

        // Generate a random number between 0 and 9
        int random_value = rand() % 10;

        // Check if the random number is 0 (simulate condition for sending SIGINT)
        if (random_value == 0) {
            std::cout << "Sending SIGINT to myself..." << std::endl;
            kill(getpid(), SIGINT); // Send SIGINT to own process
        }
    }

    return 0;
}

```

Execution :

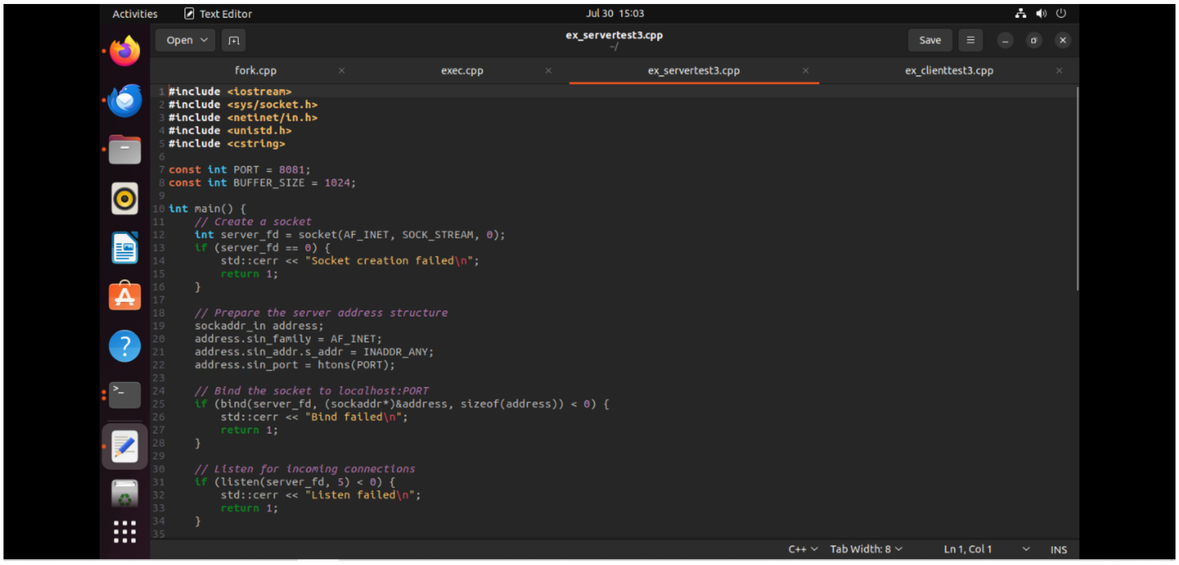


```
rps@rps-virtual-machine:~$vim ex_test2.cpp
rps@rps-virtual-machine:~$make ex_test2
g++     ex_test2.cpp   -o ex_test2
rps@rps-virtual-machine:~$./ex_test2
Signal handlers registered. Waiting for signals...
Working...
Working...
Working...
Working...
^CSIGINT received. Cleaning up and exiting gracefully.
rps@rps-virtual-machine:~$./ex_test2
Signal handlers registered. Waiting for signals...
Working...
Working...
Working...
Working...
Working...
Working...
Sending SIGINT to myself...
SIGINT received. Cleaning up and exiting gracefully.
rps@rps-virtual-machine:~$
```

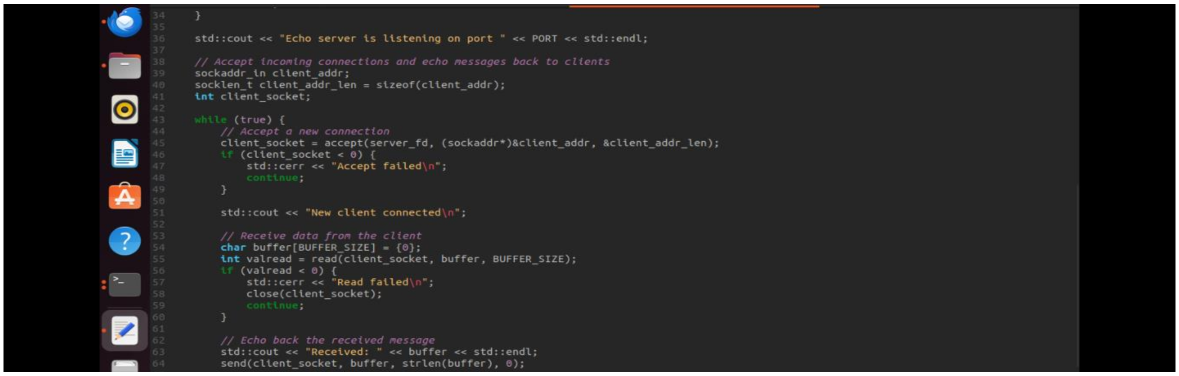
2. Sockets for Network Communication:

Implement a simple echo server in C++ that listens on a specific port, accepts client connections, and echoes back any messages received from clients.

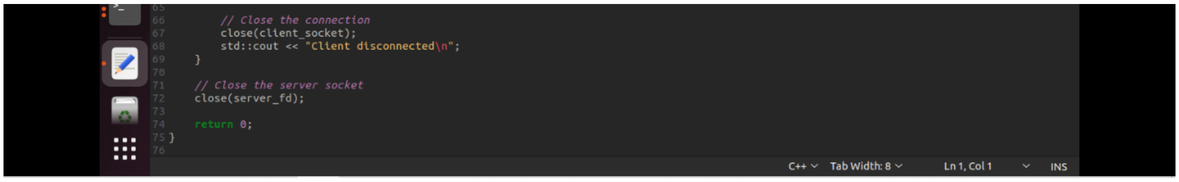
Write a client program that connects to the echo server, sends a message, and prints the echoed response. a. server - side



```cpp
#include <iostream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>

const int PORT = 8081;
const int BUFFER_SIZE = 1024;

int main() {
    // Create a socket
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == 0) {
        std::cerr << "Socket creation failed\n";
        return 1;
    }

    // Prepare the server address structure
    sockaddr_in address;
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Bind the socket to localhost:PORT
    if (bind(server_fd, (sockaddr*)&address, sizeof(address)) < 0) {
        std::cerr << "Bind failed\n";
        return 1;
    }

    // Listen for incoming connections
    if (listen(server_fd, 5) < 0) {
        std::cerr << "Listen failed\n";
        return 1;
    }

```

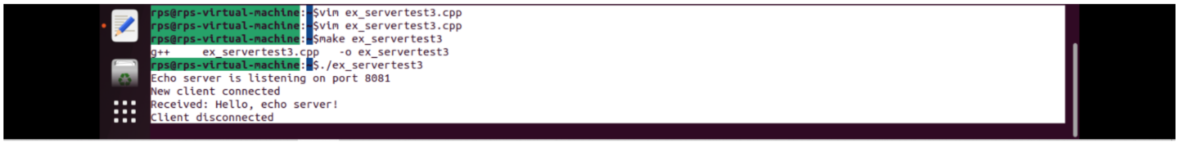


```cpp
    }

    std::cout << "Echo server is listening on port " << PORT << std::endl;

    // Accept incoming connections and echo messages back to clients
    sockaddr_in client_addr;
    socklen_t client_addr_len = sizeof(client_addr);
    int client_socket;

    while (true) {
        // Accept a new connection
        client_socket = accept(server_fd, (sockaddr*)&client_addr, &client_addr_len);
        if (client_socket < 0) {
            std::cerr << "Accept failed\n";
            continue;
        }

        std::cout << "New client connected\n";

        // Receive data from the client
        char buffer[BUFFER_SIZE] = {0};
        int valread = read(client_socket, buffer, BUFFER_SIZE);
        if (valread < 0) {
            std::cerr << "Read failed\n";
            close(client_socket);
            continue;
        }

        // Echo back the received message
        std::cout << "Received: " << buffer << std::endl;
        send(client_socket, buffer, strlen(buffer), 0);
```



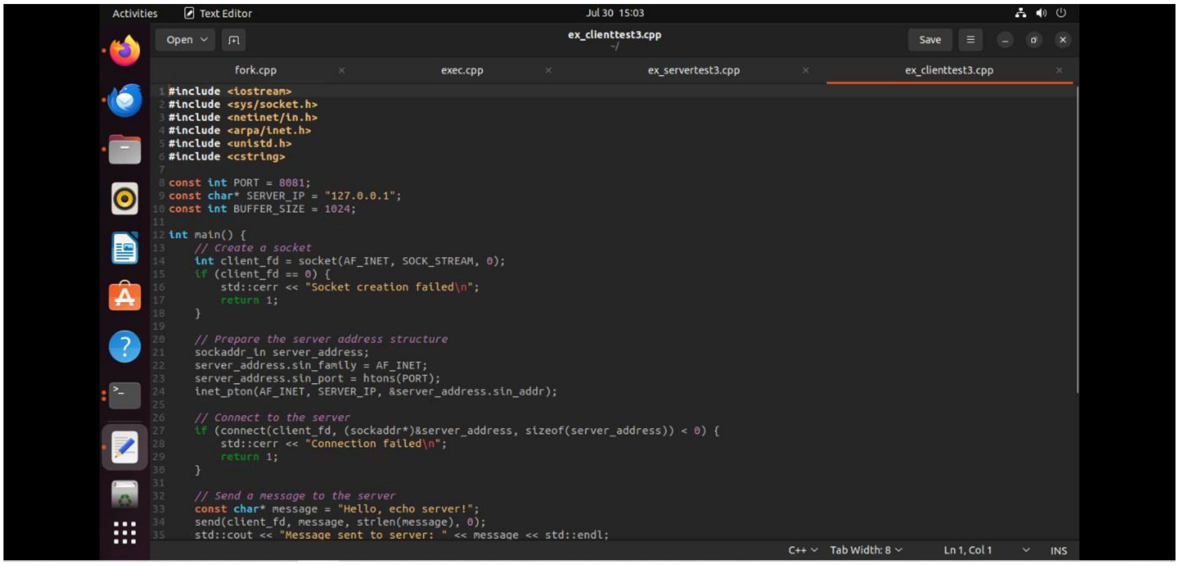
```
        // Close the connection
        close(client_socket);
        std::cout << "Client disconnected\n";
    }

    // Close the server socket
    close(server_fd);

    return 0;
}
```

Execution :

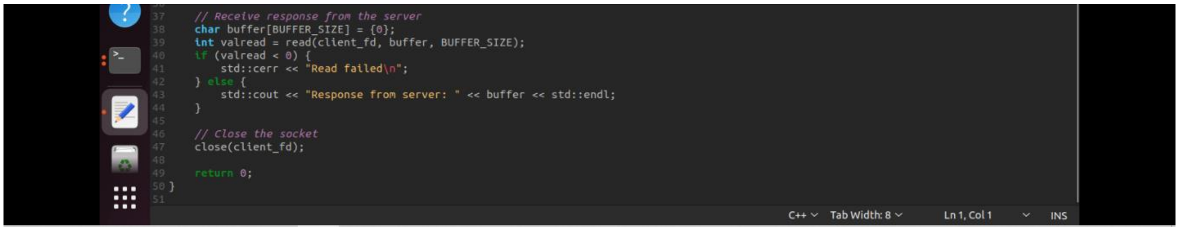


```
rps@rps-virtual-machine:~$vim ex_servertest3.cpp
rps@rps-virtual-machine:~$vim ex_servertest3.cpp
rps@rps-virtual-machine:~$make ex_servertest3
g++     ex_servertest3.cpp   -o ex_servertest3
rps@rps-virtual-machine:~$./ex_servertest3
Echo server is listening on port 8081
New client connected
Received: Hello, echo server!
Client disconnected
```

b. client - side

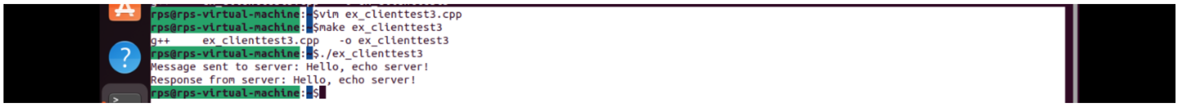


```
#include <iostream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>

const int PORT = 8081;
const char* SERVER_IP = "127.0.0.1";
const int BUFFER_SIZE = 1024;

int main() {
    // Create a socket
    int client_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (client_fd == 0) {
        std::cerr << "Socket creation failed\n";
        return 1;
    }

    // Prepare the server address structure
    sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(PORT);
    inet_pton(AF_INET, SERVER_IP, &server_address.sin_addr);

    // Connect to the server
    if (connect(client_fd, (sockaddr*)&server_address, sizeof(server_address)) < 0) {
        std::cerr << "Connection failed\n";
        return 1;
    }

    // Send a message to the server
    const char* message = "Hello, echo server!";
    send(client_fd, message, strlen(message), 0);
    std::cout << "Message sent to server: " << message << std::endl;
```



```
    // Receive response from the server
    char buffer[BUFFER_SIZE] = {0};
    int valread = read(client_fd, buffer, BUFFER_SIZE);
    if (valread < 0) {
        std::cerr << "Read failed\n";
    } else {
        std::cout << "Response from server: " << buffer << std::endl;
    }

    // Close the socket
    close(client_fd);

    return 0;
}
```

Execution :

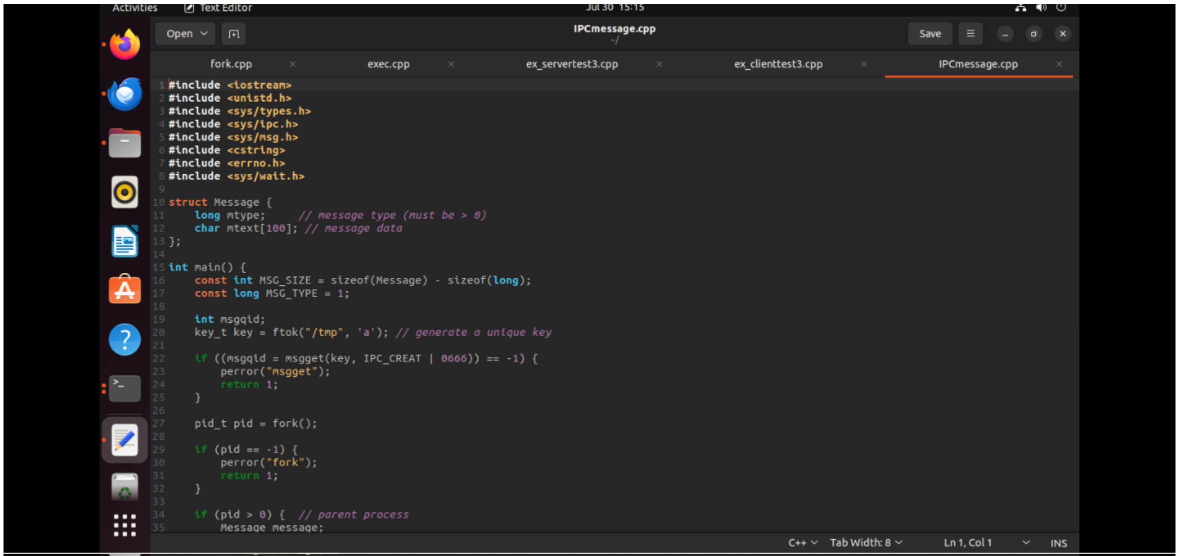


```
rps@rps-virtual-machine:~$vim ex_clienttest3.cpp
rps@rps-virtual-machine:~$make ex_clienttest3
g++     ex_clienttest3.cpp   -o ex_clienttest3
rps@rps-virtual-machine:~$./ex_clienttest3
Message sent to server: Hello, echo server!
Response from server: Hello, echo server!
rps@rps-virtual-machine:~$
```

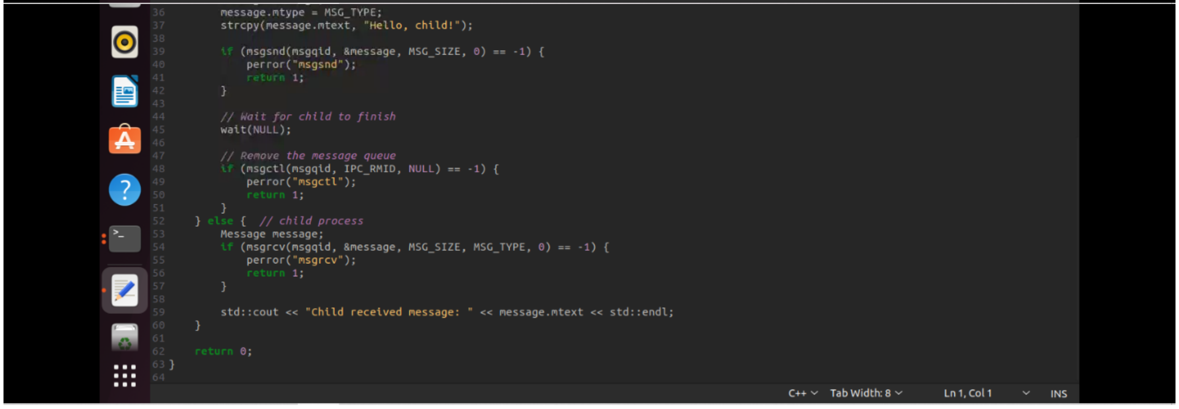
3. Inter-Process Communication (IPC):

Write a C++ program that creates a parent process and a child process. Use a pipe for IPC to send a message from the parent to the child, and have the child process print the message.

How would you modify the program to use a message queue instead of a pipe for communication between the parent and child processes?

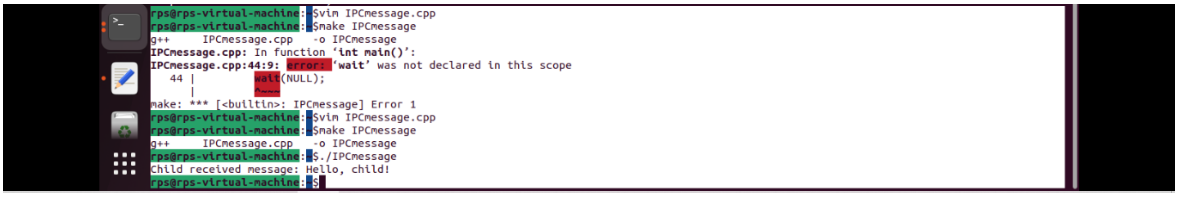


```cpp
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <cstring>
#include <errno.h>
#include <sys/wait.h>

struct Message {
    long mtype;     // message type (must be > 0)
    char mtext[100]; // message data
};

int main() {
    const int MSG_SIZE = sizeof(Message) - sizeof(long);
    const long MSG_TYPE = 1;

    int msgqid;
    key_t key = ftok("/tmp", 'a'); // generate a unique key

    if ((msgqid = msgget(key, IPC_CREAT | 0666)) == -1) {
        perror("msgget");
        return 1;
    }

    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        return 1;
    }

    if (pid > 0) {  // parent process
        Message message;
```



```cpp
        message.mtype = MSG_TYPE;
        strcpy(message.mtext, "Hello, child!");

        if (msgsnd(msgqid, &message, MSG_SIZE, 0) == -1) {
            perror("msgsnd");
            return 1;
        }

        // Wait for child to finish
        wait(NULL);

        // Remove the message queue
        if (msgctl(msgqid, IPC_RMID, NULL) == -1) {
            perror("msgctl");
            return 1;
        }
    } else {  // child process
        Message message;
        if (msgrcv(msgqid, &message, MSG_SIZE, MSG_TYPE, 0) == -1) {
            perror("msgrcv");
            return 1;
        }

        std::cout << "Child received message: " << message.mtext << std::endl;
    }

    return 0;
}
```

Execution :



```
rps@rps-virtual-machine:~$vim IPCmessage.cpp
rps@rps-virtual-machine:~$make IPCmessage
g++     IPCmessage.cpp   -o IPCmessage
IPCmessage.cpp: In function ‘int main()’:
IPCmessage.cpp:44:9: error: ‘wait’ was not declared in this scope
   44 |         wait(NULL);
      |         ^~~~
make: *** [<builtin>: IPCmessage] Error 1
rps@rps-virtual-machine:~$vim IPCmessage.cpp
rps@rps-virtual-machine:~$make IPCmessage
g++     IPCmessage.cpp   -o IPCmessage
rps@rps-virtual-machine:~$./IPCmessage
Child received message: Hello, child!
rps@rps-virtual-machine:~$
```

We can modify the program to use a message queue instead of a pipe for communication between the parent and child processes?

1. **Include necessary header for IPC messaging queue.**
2. **Define structure for the message.**
3. **Create message queue.**
4. **Fork a child process.**
5. **Parent sends a message.**
6. **Child receives the message and prints the message.**
7. **Clean up.**