Problem Statement: Socket Programming in C

Design and implement a reliable and efficient network communication system using socket programming in C to enable data exchange between two or more processes running on different machines over a network.

Specific Requirements:

Socket creation: Create appropriate socket descriptors for the desired communication protocol (TCP, UDP, etc.).

Address binding: Bind the created socket to a specific network address and port number for both client and server applications.

Connection establishment: Implement connection setup mechanisms (connect, accept) for TCP-based communication.

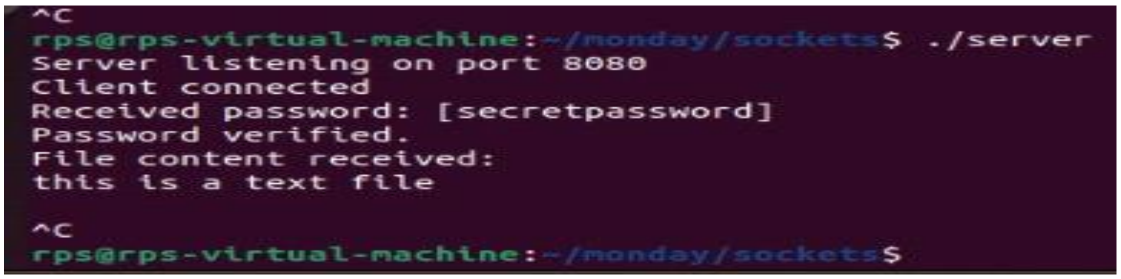
Data transfer: Develop functions for sending and receiving data over the established socket connection.

Error handling: Incorporate robust error handling mechanisms to address potential network issues and unexpected exceptions.

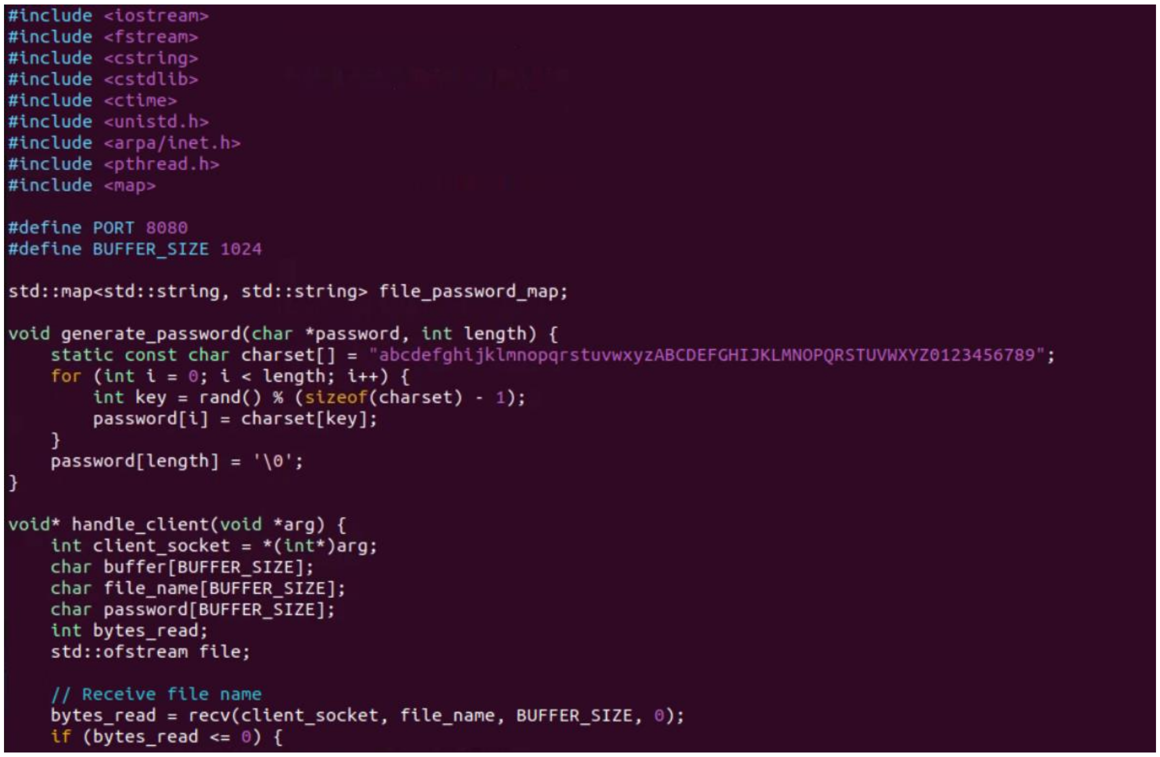
Concurrency: For server-side applications, consider handling multiple client connections concurrently using appropriate techniques (e.g., threading, forking).

Security: Implement appropriate security measures to protect data integrity and confidentiality (e.g., encryption, authentication).

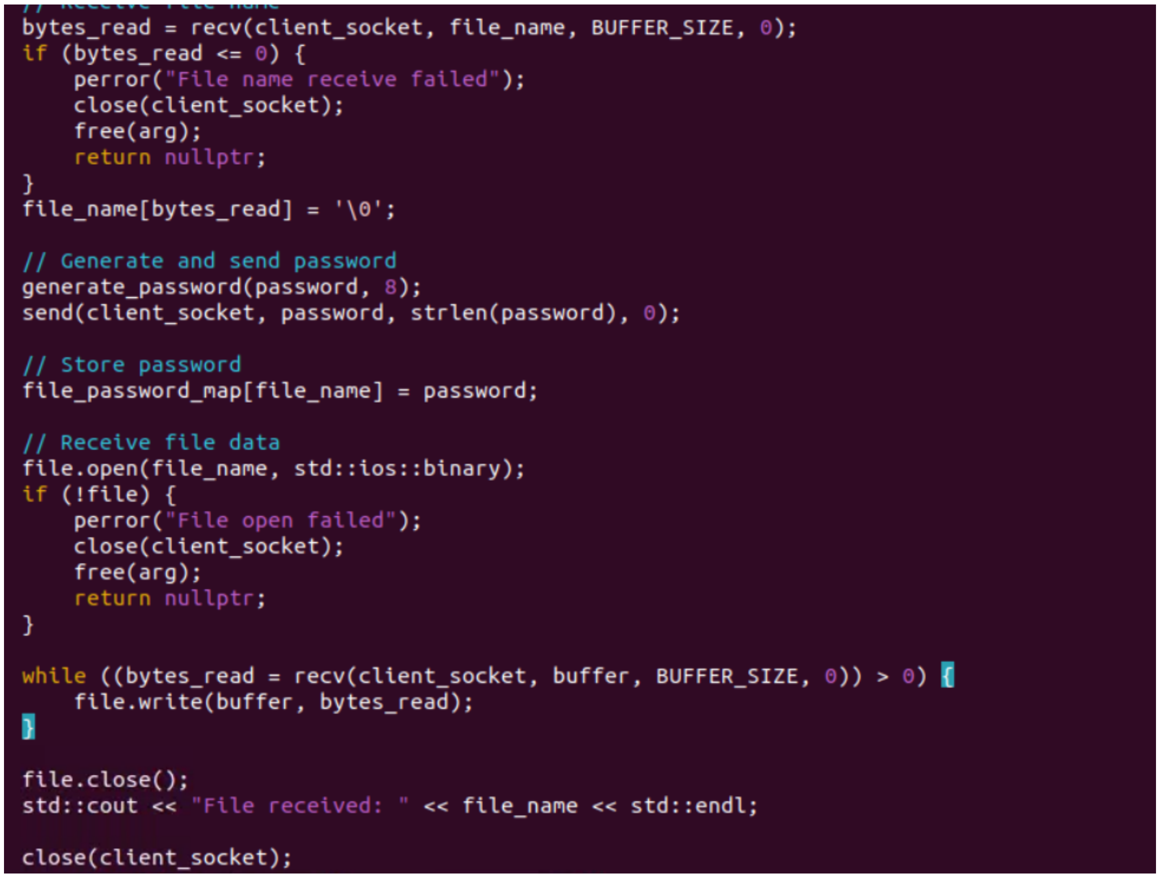
SERVER CODE :-



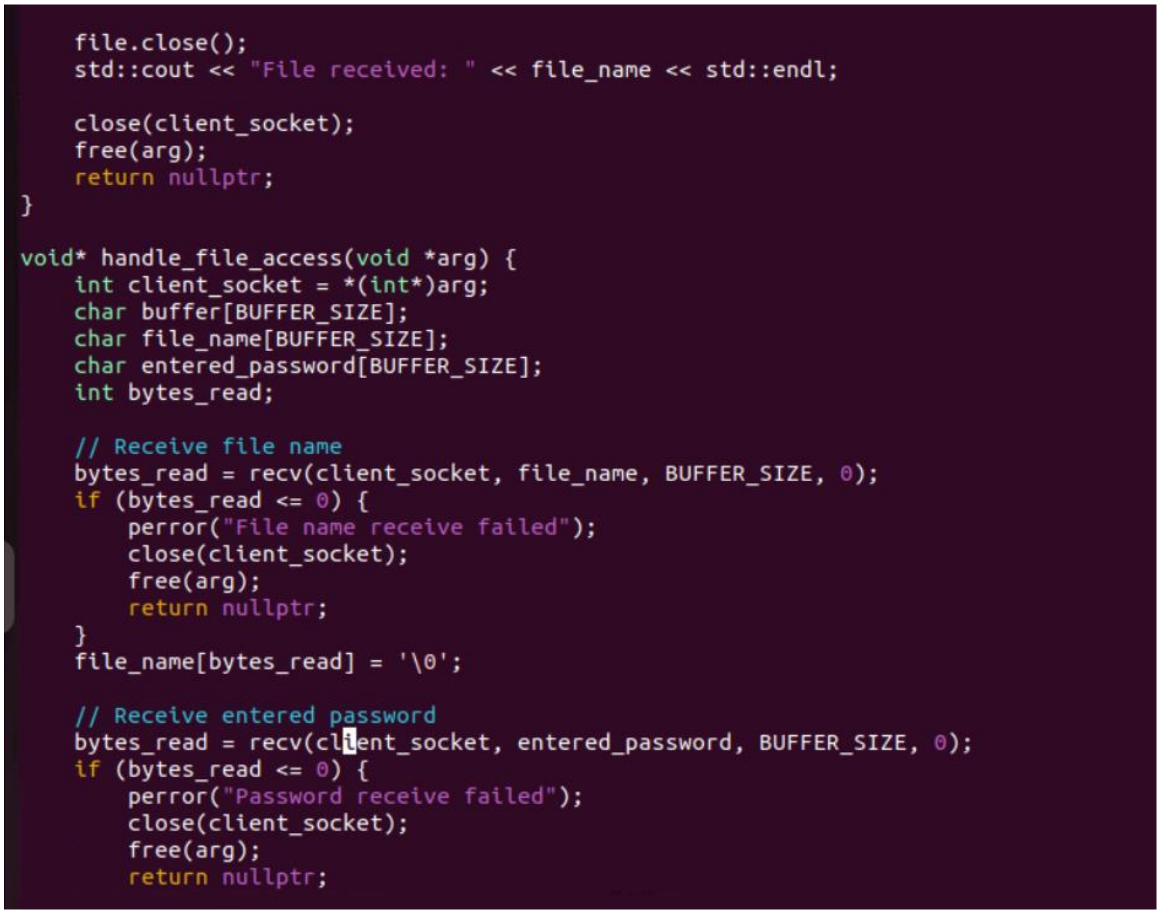
```
^C
rps@rps-virtual-machine:~/monday/sockets$ ./server
Server listening on port 8080
Client connected
Received password: [secretpassword]
Password verified.
File content received:
this is a text file

^C
rps@rps-virtual-machine:~/monday/sockets$
```



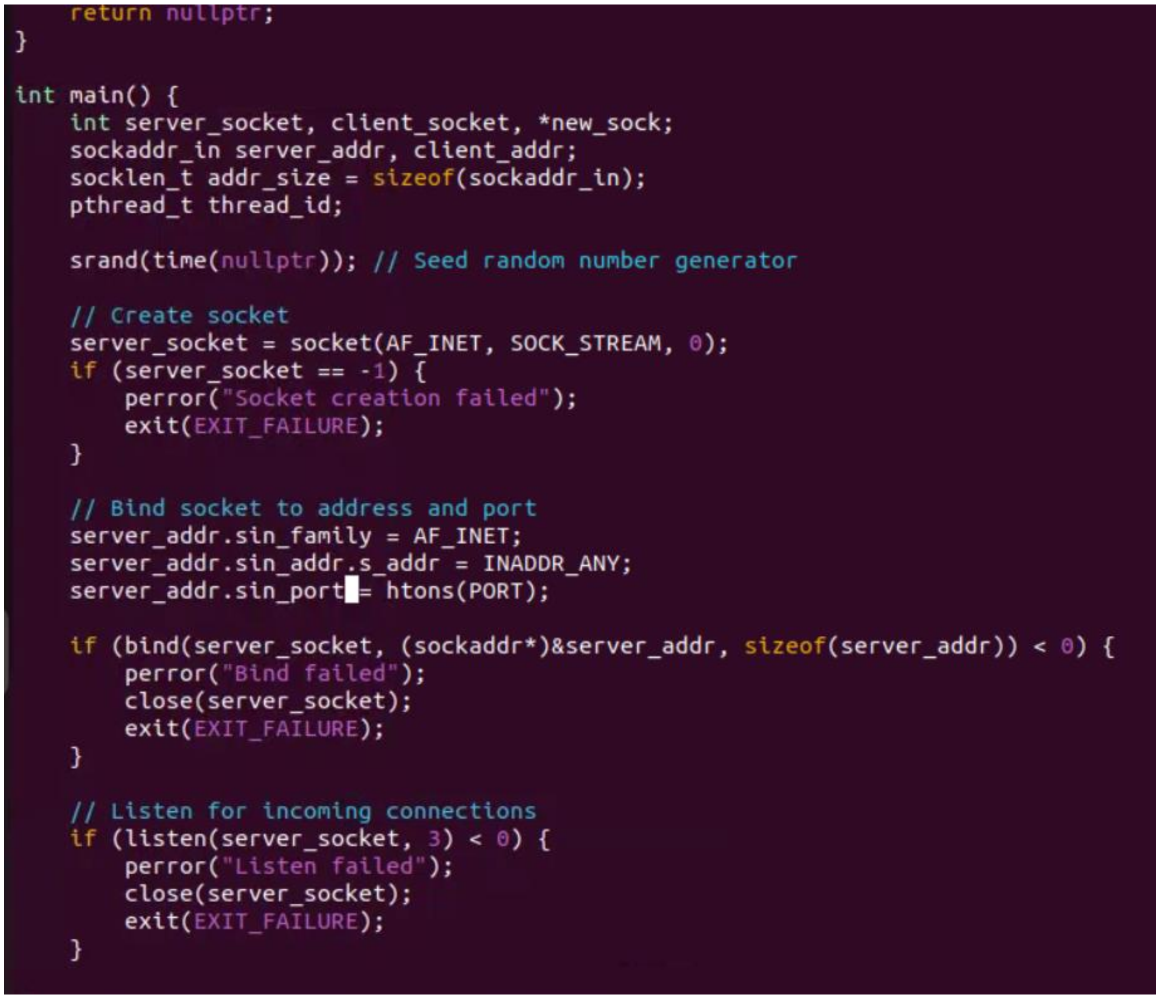
```cpp
#include <iostream>
#include <fstream>
#include <cstring>
#include <cstdlib>
#include <ctime>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <map>

#define PORT 8080
#define BUFFER_SIZE 1024

std::map<std::string, std::string> file_password_map;

void generate_password(char *password, int length) {
    static const char charset[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    for (int i = 0; i < length; i++) {
        int key = rand() % (sizeof(charset) - 1);
        password[i] = charset[key];
    }
    password[length] = '\0';
}

void* handle_client(void *arg) {
    int client_socket = *(int*)arg;
    char buffer[BUFFER_SIZE];
    char file_name[BUFFER_SIZE];
    char password[BUFFER_SIZE];
    int bytes_read;
    std::ofstream file;

    // Receive file name
    bytes_read = recv(client_socket, file_name, BUFFER_SIZE, 0);
    if (bytes_read <= 0) {
```



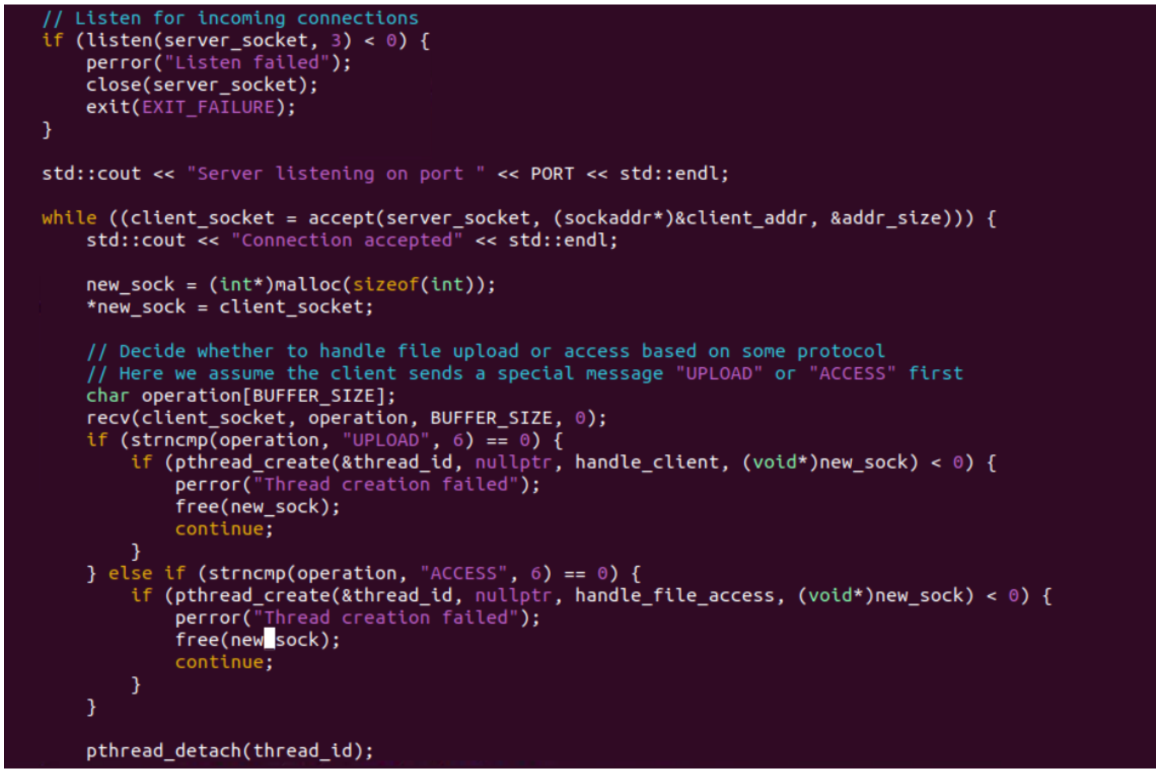
```cpp
bytes_read = recv(client_socket, file_name, BUFFER_SIZE, 0);
if (bytes_read <= 0) {
    perror("File name receive failed");
    close(client_socket);
    free(arg);
    return nullptr;
}
file_name[bytes_read] = '\0';

// Generate and send password
generate_password(password, 8);
send(client_socket, password, strlen(password), 0);

// Store password
file_password_map[file_name] = password;

// Receive file data
file.open(file_name, std::ios::binary);
if (!file) {
    perror("File open failed");
    close(client_socket);
    free(arg);
    return nullptr;
}

while ((bytes_read = recv(client_socket, buffer, BUFFER_SIZE, 0)) > 0) {
    file.write(buffer, bytes_read);
}

file.close();
std::cout << "File received: " << file_name << std::endl;

close(client_socket);
```



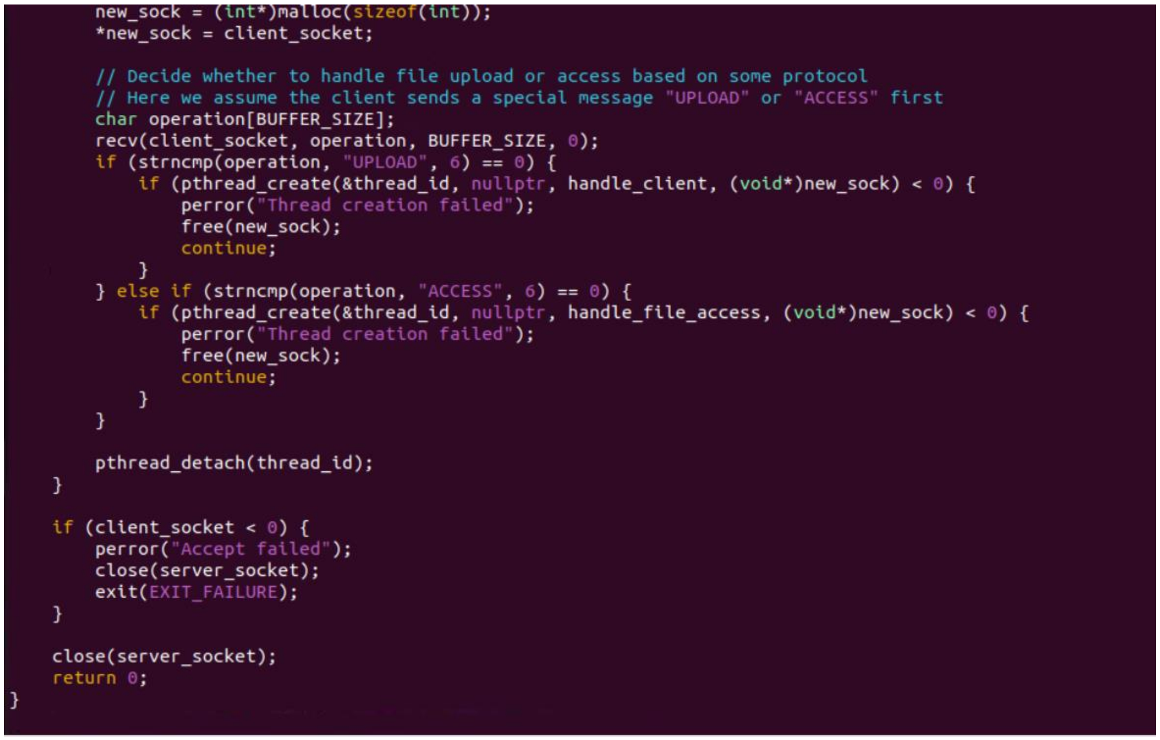
```
    file.close();
    std::cout << "File received: " << file_name << std::endl;

    close(client_socket);
    free(arg);
    return nullptr;
}

void* handle_file_access(void *arg) {
    int client_socket = *(int*)arg;
    char buffer[BUFFER_SIZE];
    char file_name[BUFFER_SIZE];
    char entered_password[BUFFER_SIZE];
    int bytes_read;

    // Receive file name
    bytes_read = recv(client_socket, file_name, BUFFER_SIZE, 0);
    if (bytes_read <= 0) {
        perror("File name receive failed");
        close(client_socket);
        free(arg);
        return nullptr;
    }
    file_name[bytes_read] = '\0';

    // Receive entered password
    bytes_read = recv(client_socket, entered_password, BUFFER_SIZE, 0);
    if (bytes_read <= 0) {
        perror("Password receive failed");
        close(client_socket);
        free(arg);
        return nullptr;
```



```
    return nullptr;
}

int main() {
    int server_socket, client_socket, *new_sock;
    sockaddr_in server_addr, client_addr;
    socklen_t addr_size = sizeof(sockaddr_in);
    pthread_t thread_id;

    srand(time(nullptr)); // Seed random number generator

    // Create socket
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Bind socket to address and port
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(server_socket, (sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("Bind failed");
        close(server_socket);
        exit(EXIT_FAILURE);
    }

    // Listen for incoming connections
    if (listen(server_socket, 3) < 0) {
        perror("Listen failed");
        close(server_socket);
        exit(EXIT_FAILURE);
    }
```

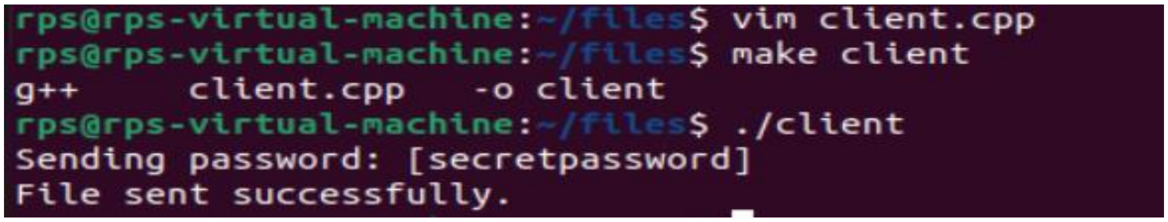


```
    // Listen for incoming connections
    if (listen(server_socket, 3) < 0) {
        perror("Listen failed");
        close(server_socket);
        exit(EXIT_FAILURE);
    }

    std::cout << "Server listening on port " << PORT << std::endl;

    while ((client_socket = accept(server_socket, (sockaddr*)&client_addr, &addr_size))) {
        std::cout << "Connection accepted" << std::endl;

        new_sock = (int*)malloc(sizeof(int));
        *new_sock = client_socket;

        // Decide whether to handle file upload or access based on some protocol
        // Here we assume the client sends a special message "UPLOAD" or "ACCESS" first
        char operation[BUFFER_SIZE];
        recv(client_socket, operation, BUFFER_SIZE, 0);
        if (strncmp(operation, "UPLOAD", 6) == 0) {
            if (pthread_create(&thread_id, nullptr, handle_client, (void*)new_sock) < 0) {
                perror("Thread creation failed");
                free(new_sock);
                continue;
            }
        } else if (strncmp(operation, "ACCESS", 6) == 0) {
            if (pthread_create(&thread_id, nullptr, handle_file_access, (void*)new_sock) < 0) {
                perror("Thread creation failed");
                free(new_sock);
                continue;
            }
        }

        pthread_detach(thread_id);
```

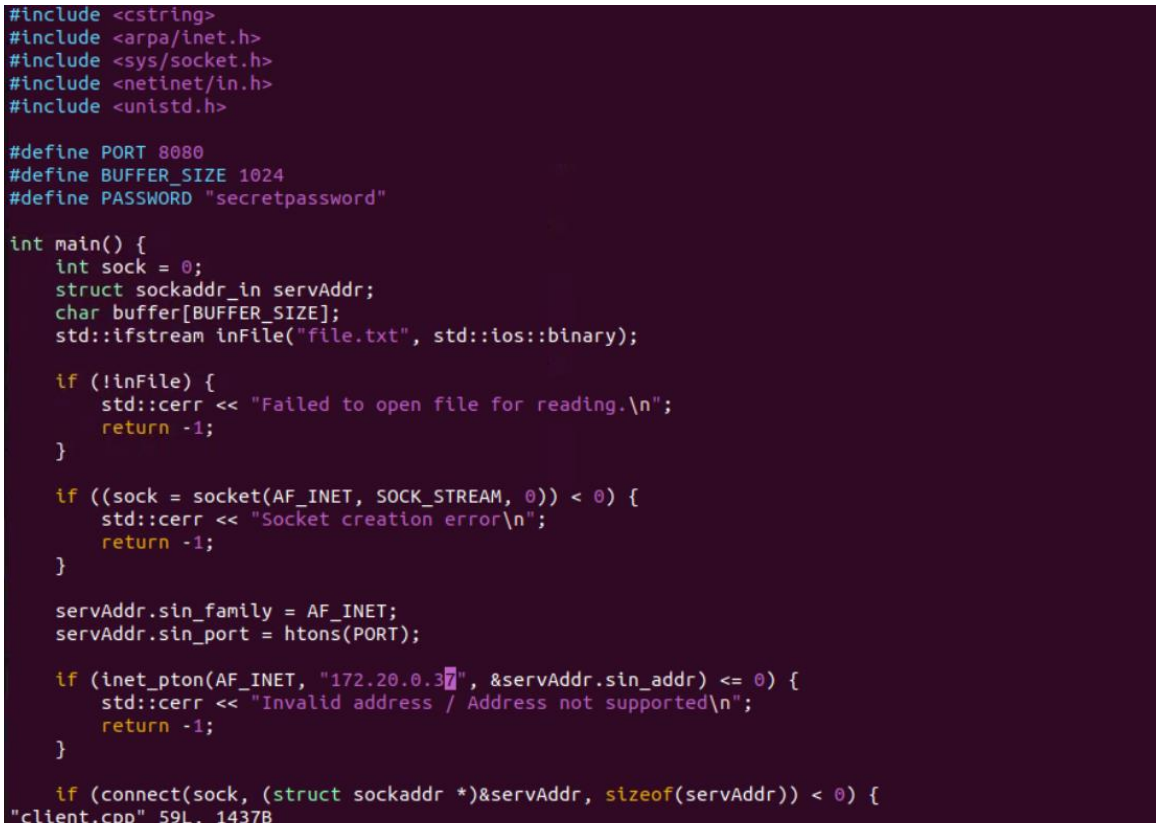


```
        new_sock = (int*)malloc(sizeof(int));
        *new_sock = client_socket;

        // Decide whether to handle file upload or access based on some protocol
        // Here we assume the client sends a special message "UPLOAD" or "ACCESS" first
        char operation[BUFFER_SIZE];
        recv(client_socket, operation, BUFFER_SIZE, 0);
        if (strncmp(operation, "UPLOAD", 6) == 0) {
            if (pthread_create(&thread_id, nullptr, handle_client, (void*)new_sock) < 0) {
                perror("Thread creation failed");
                free(new_sock);
                continue;
            }
        } else if (strncmp(operation, "ACCESS", 6) == 0) {
            if (pthread_create(&thread_id, nullptr, handle_file_access, (void*)new_sock) < 0) {
                perror("Thread creation failed");
                free(new_sock);
                continue;
            }
        }

        pthread_detach(thread_id);
    }

    if (client_socket < 0) {
        perror("Accept failed");
        close(server_socket);
        exit(EXIT_FAILURE);
    }

    close(server_socket);
    return 0;
}
```

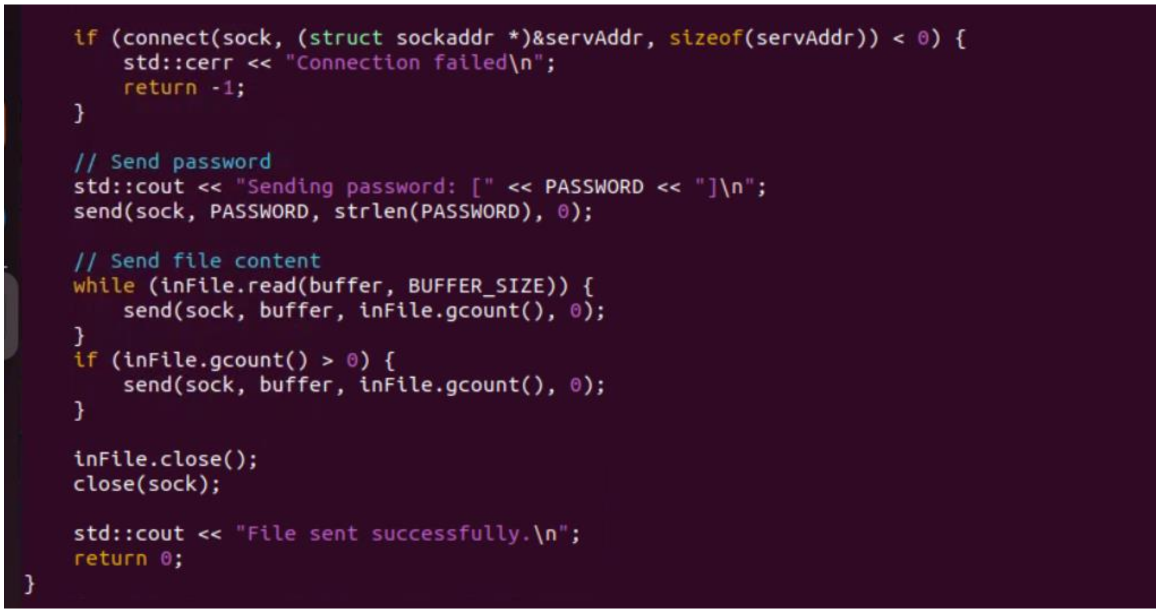
CLIENT CODE :-



```
rps@rps-virtual-machine:~/files$ vim client.cpp
rps@rps-virtual-machine:~/files$ make client
g++     client.cpp   -o client
rps@rps-virtual-machine:~/files$ ./client
Sending password: [secretpassword]
File sent successfully.
```



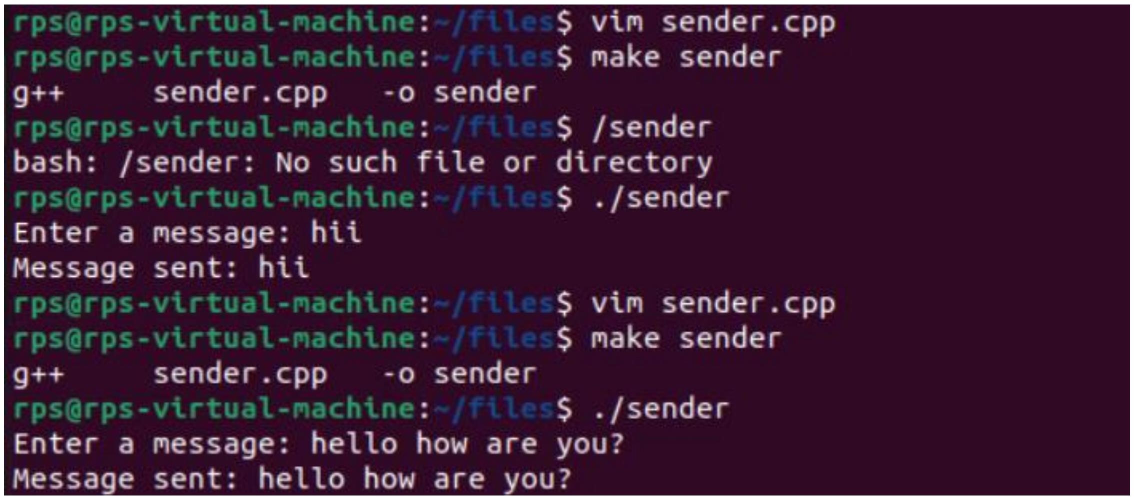
```
#include <cstring>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

#define PORT 8080
#define BUFFER_SIZE 1024
#define PASSWORD "secretpassword"

int main() {
    int sock = 0;
    struct sockaddr_in servAddr;
    char buffer[BUFFER_SIZE];
    std::ifstream inFile("file.txt", std::ios::binary);

    if (!inFile) {
        std::cerr << "Failed to open file for reading.\n";
        return -1;
    }

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

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

    if (inet_pton(AF_INET, "172.20.0.37", &servAddr.sin_addr) <= 0) {
        std::cerr << "Invalid address / Address not supported\n";
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&servAddr, sizeof(servAddr)) < 0) {
"client.cpp" 59L, 1437B
```



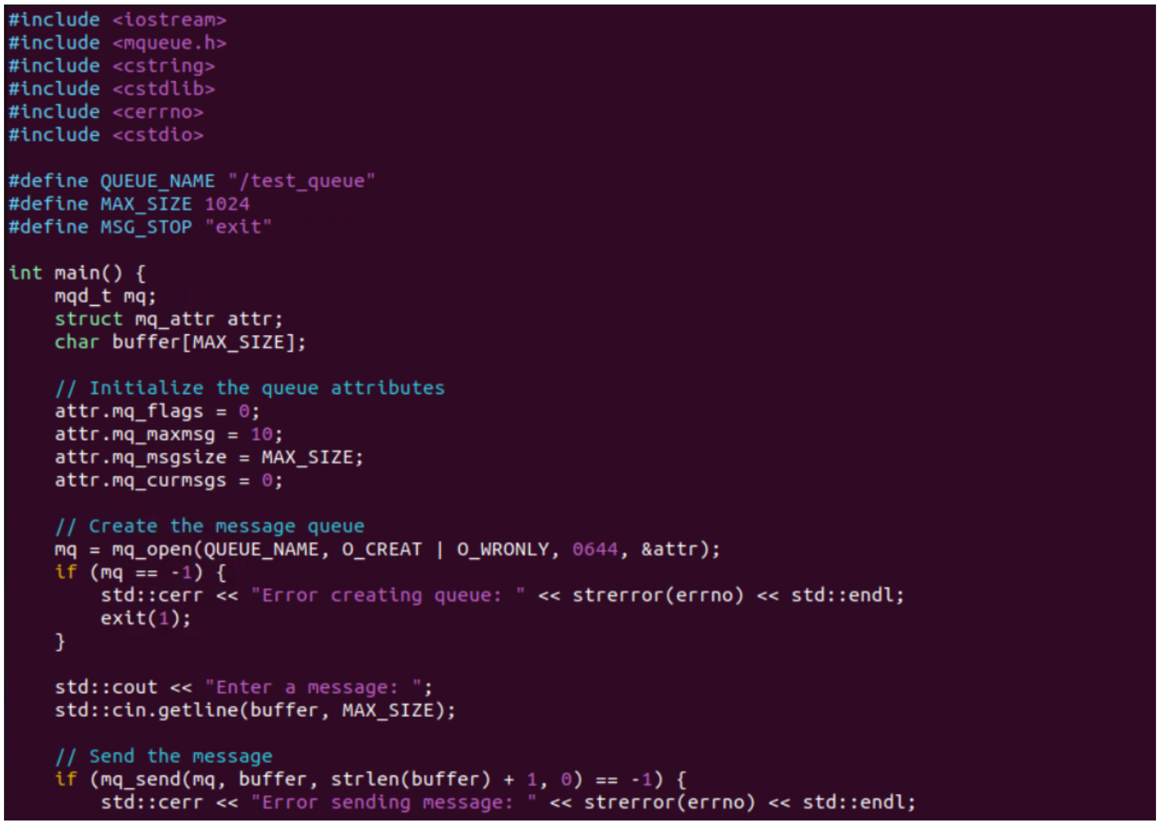
```
    if (connect(sock, (struct sockaddr *)&servAddr, sizeof(servAddr)) < 0) {
        std::cerr << "Connection failed\n";
        return -1;
    }

    // Send password
    std::cout << "Sending password: [" << PASSWORD << "]\n";
    send(sock, PASSWORD, strlen(PASSWORD), 0);

    // Send file content
    while (inFile.read(buffer, BUFFER_SIZE)) {
        send(sock, buffer, inFile.gcount(), 0);
    }
    if (inFile.gcount() > 0) {
        send(sock, buffer, inFile.gcount(), 0);
    }

    inFile.close();
    close(sock);

    std::cout << "File sent successfully.\n";
    return 0;
}
```

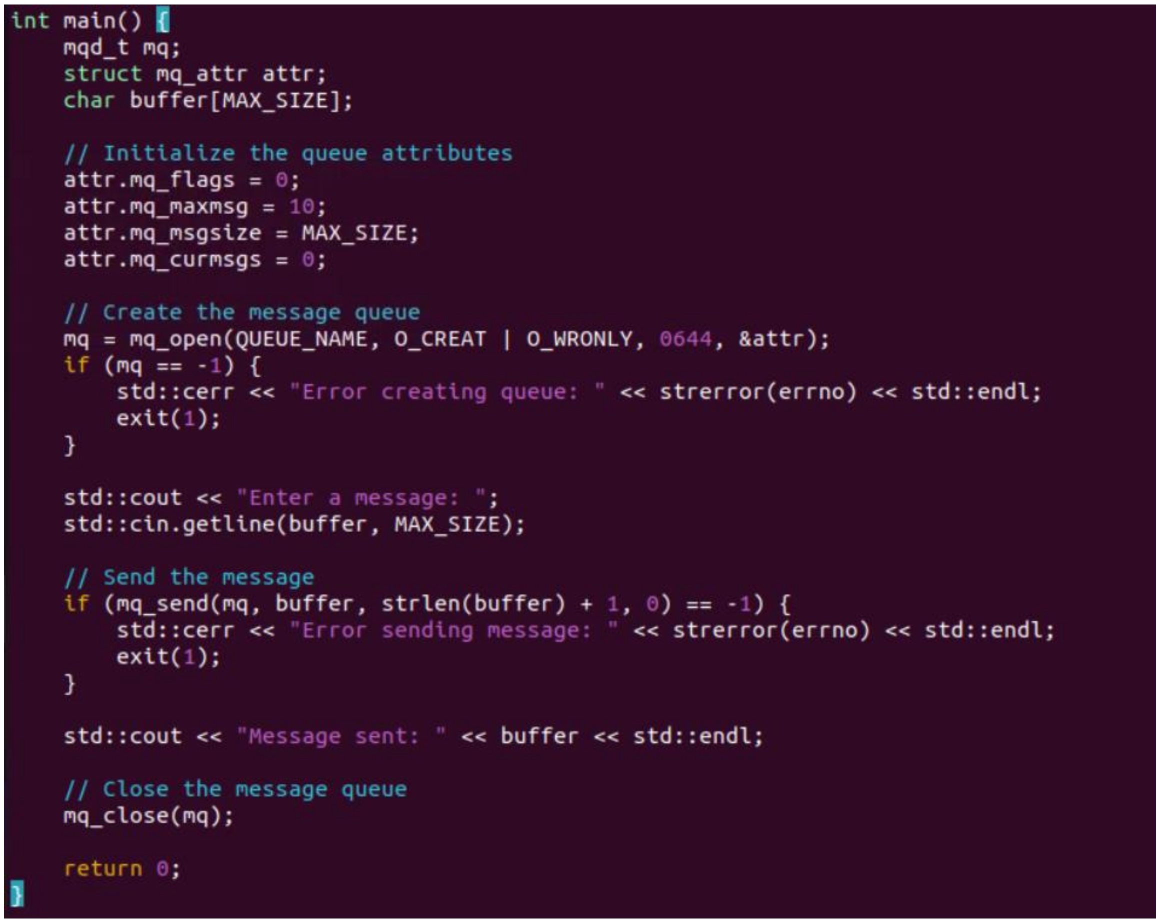
SENDING MESSAGE FROM SENDER TO RECEIVER :-

SENDER CODE :-



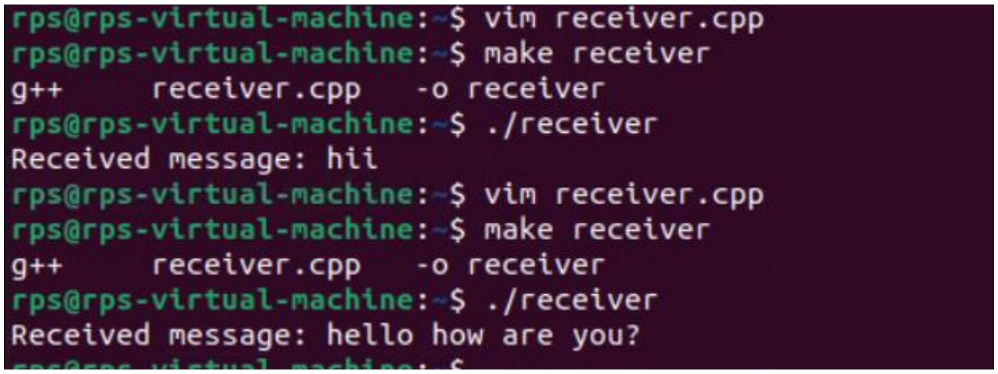
```
rps@rps-virtual-machine:~/files$ vim sender.cpp
rps@rps-virtual-machine:~/files$ make sender
g++     sender.cpp   -o sender
rps@rps-virtual-machine:~/files$ /sender
bash: /sender: No such file or directory
rps@rps-virtual-machine:~/files$ ./sender
Enter a message: hii
Message sent: hii
rps@rps-virtual-machine:~/files$ vim sender.cpp
rps@rps-virtual-machine:~/files$ make sender
g++     sender.cpp   -o sender
rps@rps-virtual-machine:~/files$ ./sender
Enter a message: hello how are you?
Message sent: hello how are you?
```


```
#include <iostream>
#include <mqueue.h>
#include <cstring>
#include <cstdlib>
#include <cerrno>
#include <cstdio>

#define QUEUE_NAME "/test_queue"
#define MAX_SIZE 1024
#define MSG_STOP "exit"

int main() {
    mqd_t mq;
    struct mq_attr attr;
    char buffer[MAX_SIZE];

    // Initialize the queue attributes
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = MAX_SIZE;
    attr.mq_curmsgs = 0;

    // Create the message queue
    mq = mq_open(QUEUE_NAME, O_CREAT | O_WRONLY, 0644, &attr);
    if (mq == -1) {
        std::cerr << "Error creating queue: " << strerror(errno) << std::endl;
        exit(1);
    }

    std::cout << "Enter a message: ";
    std::cin.getline(buffer, MAX_SIZE);

    // Send the message
    if (mq_send(mq, buffer, strlen(buffer) + 1, 0) == -1) {
        std::cerr << "Error sending message: " << strerror(errno) << std::endl;
```

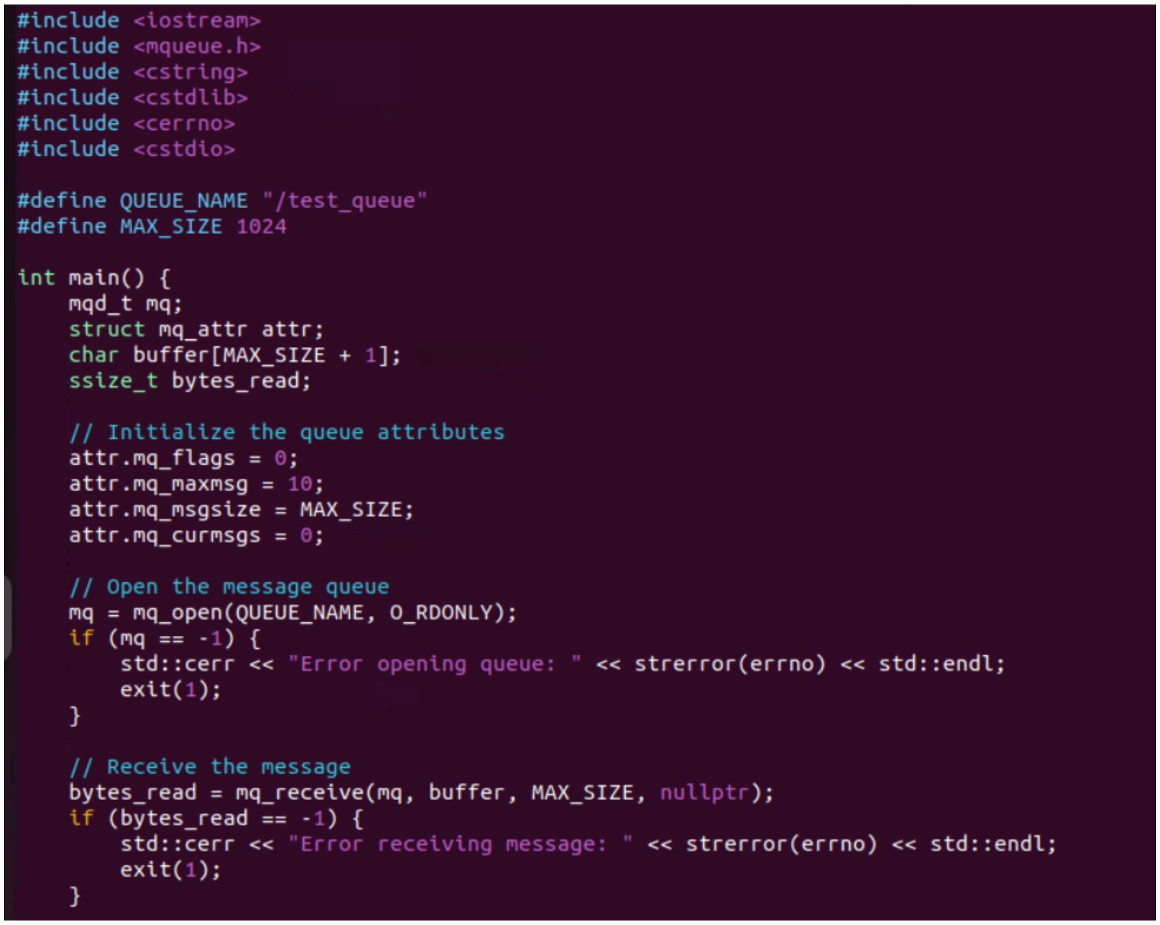


```
int main() {
    mqd_t mq;
    struct mq_attr attr;
    char buffer[MAX_SIZE];

    // Initialize the queue attributes
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = MAX_SIZE;
    attr.mq_curmsgs = 0;

    // Create the message queue
    mq = mq_open(QUEUE_NAME, O_CREAT | O_WRONLY, 0644, &attr);
    if (mq == -1) {
        std::cerr << "Error creating queue: " << strerror(errno) << std::endl;
        exit(1);
    }

    std::cout << "Enter a message: ";
    std::cin.getline(buffer, MAX_SIZE);

    // Send the message
    if (mq_send(mq, buffer, strlen(buffer) + 1, 0) == -1) {
        std::cerr << "Error sending message: " << strerror(errno) << std::endl;
        exit(1);
    }

    std::cout << "Message sent: " << buffer << std::endl;

    // Close the message queue
    mq_close(mq);

    return 0;
}
```

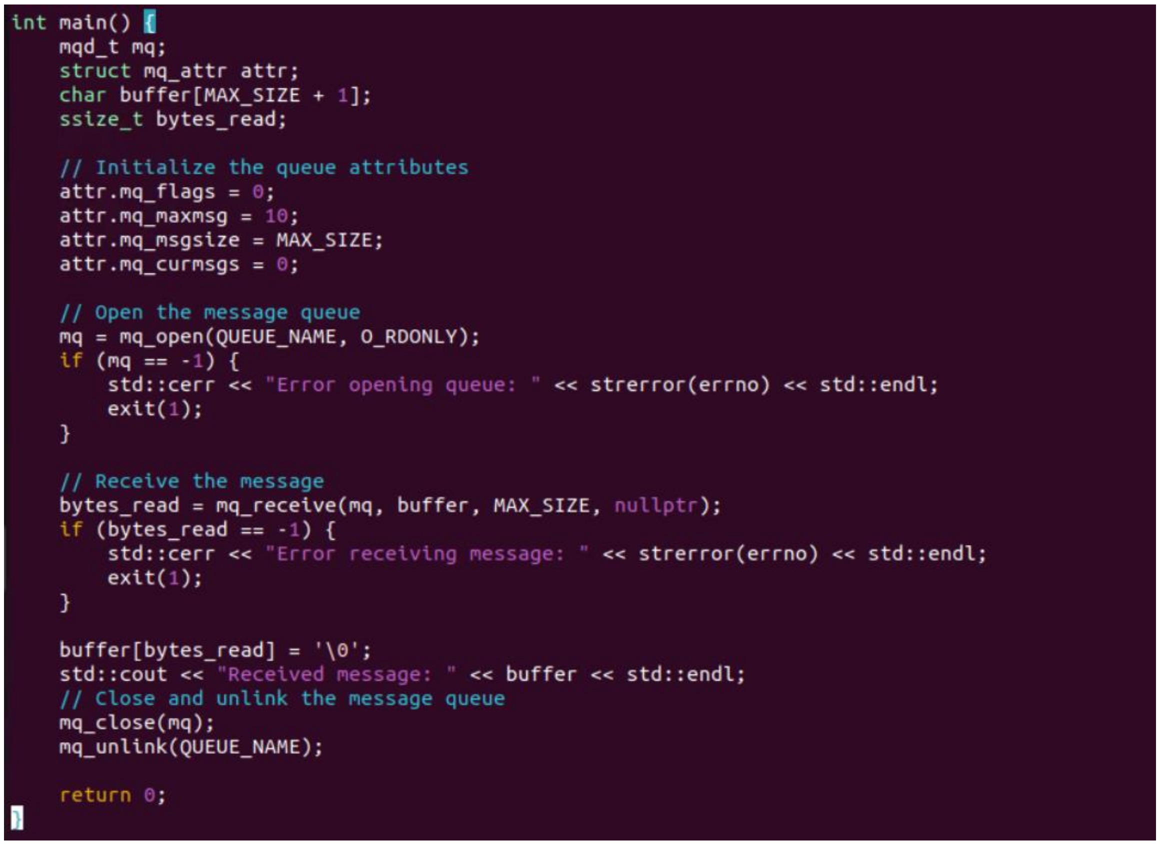
RECEIVER CODE :-



```
rps@rps-virtual-machine:~$ vim receiver.cpp
rps@rps-virtual-machine:~$ make receiver
g++     receiver.cpp   -o receiver
rps@rps-virtual-machine:~$ ./receiver
Received message: hii
rps@rps-virtual-machine:~$ vim receiver.cpp
rps@rps-virtual-machine:~$ make receiver
g++     receiver.cpp   -o receiver
rps@rps-virtual-machine:~$ ./receiver
Received message: hello how are you?
```



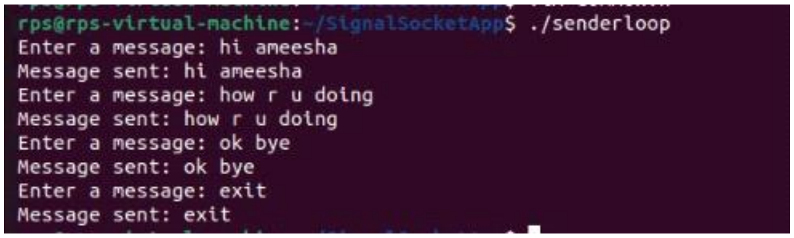
```cpp
#include <iostream>
#include <mqueue.h>
#include <cstring>
#include <cstdlib>
#include <cerrno>
#include <cstdio>

#define QUEUE_NAME "/test_queue"
#define MAX_SIZE 1024

int main() {
    mqd_t mq;
    struct mq_attr attr;
    char buffer[MAX_SIZE + 1];
    ssize_t bytes_read;

    // Initialize the queue attributes
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = MAX_SIZE;
    attr.mq_curmsgs = 0;

    // Open the message queue
    mq = mq_open(QUEUE_NAME, O_RDONLY);
    if (mq == -1) {
        std::cerr << "Error opening queue: " << strerror(errno) << std::endl;
        exit(1);
    }

    // Receive the message
    bytes_read = mq_receive(mq, buffer, MAX_SIZE, nullptr);
    if (bytes_read == -1) {
        std::cerr << "Error receiving message: " << strerror(errno) << std::endl;
        exit(1);
    }
```



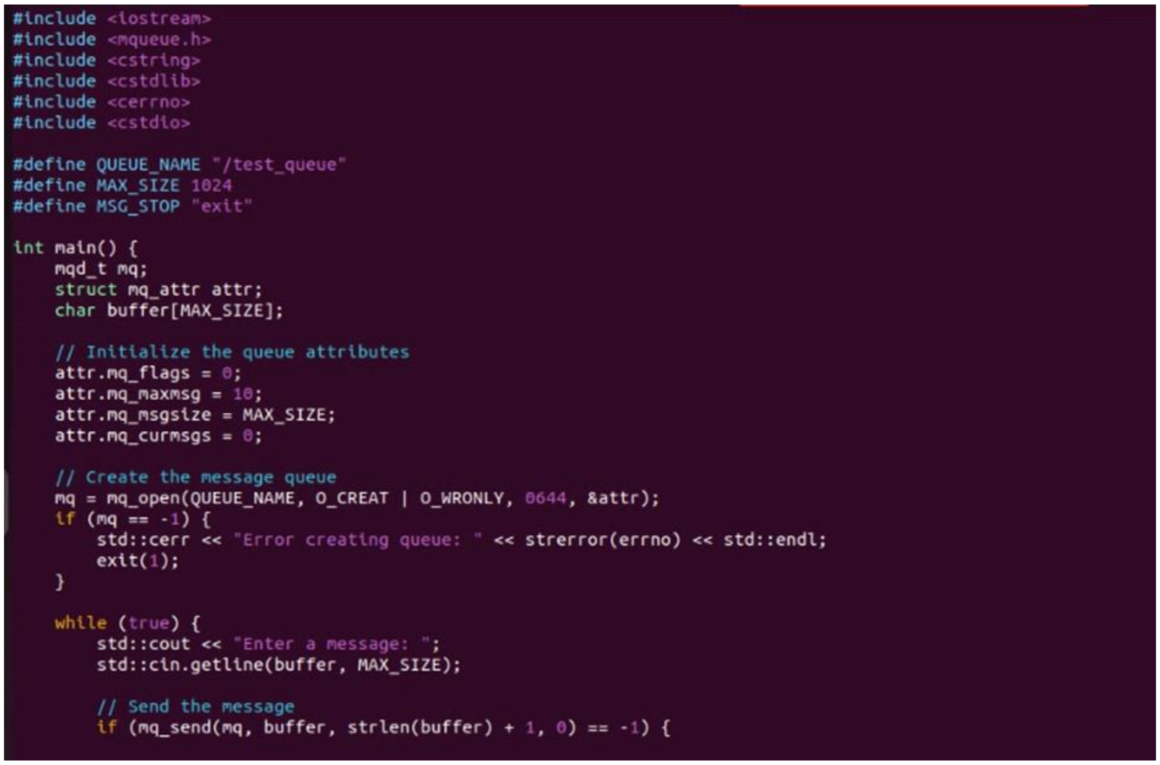
```
int main() {
    mqd_t mq;
    struct mq_attr attr;
    char buffer[MAX_SIZE + 1];
    ssize_t bytes_read;

    // Initialize the queue attributes
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = MAX_SIZE;
    attr.mq_curmsgs = 0;

    // Open the message queue
    mq = mq_open(QUEUE_NAME, O_RDONLY);
    if (mq == -1) {
        std::cerr << "Error opening queue: " << strerror(errno) << std::endl;
        exit(1);
    }

    // Receive the message
    bytes_read = mq_receive(mq, buffer, MAX_SIZE, nullptr);
    if (bytes_read == -1) {
        std::cerr << "Error receiving message: " << strerror(errno) << std::endl;
        exit(1);
    }

    buffer[bytes_read] = '\0';
    std::cout << "Received message: " << buffer << std::endl;
    // Close and unlink the message queue
    mq_close(mq);
    mq_unlink(QUEUE_NAME);

    return 0;
}
```

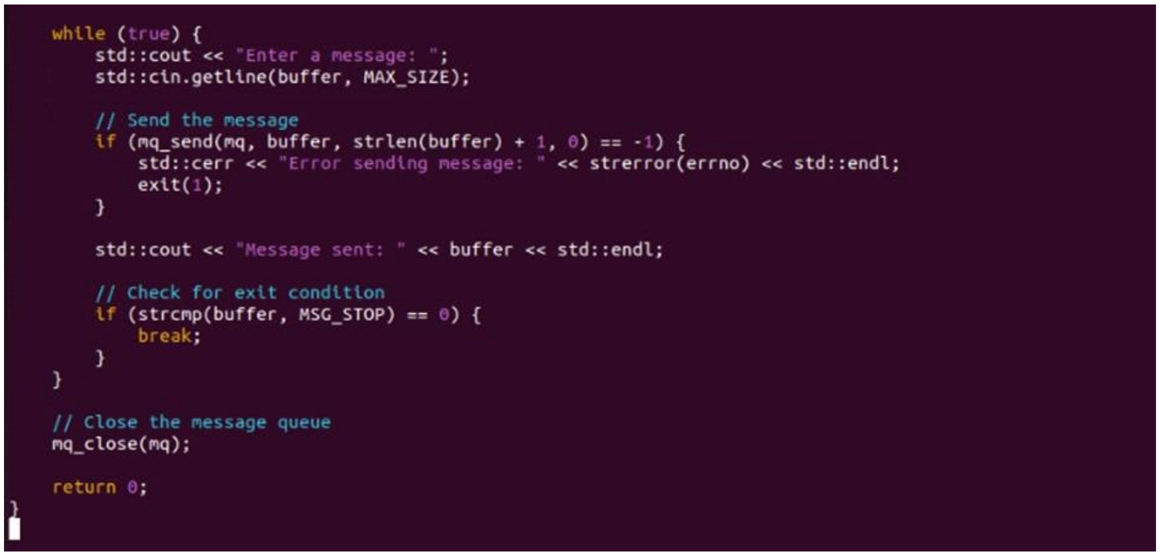
CONVERSION BETWEEN SENDER AND RECEIVER (LOOPING) :-

SENDER CODE :-

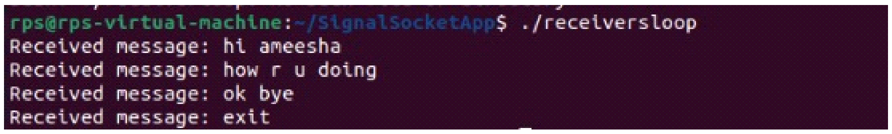


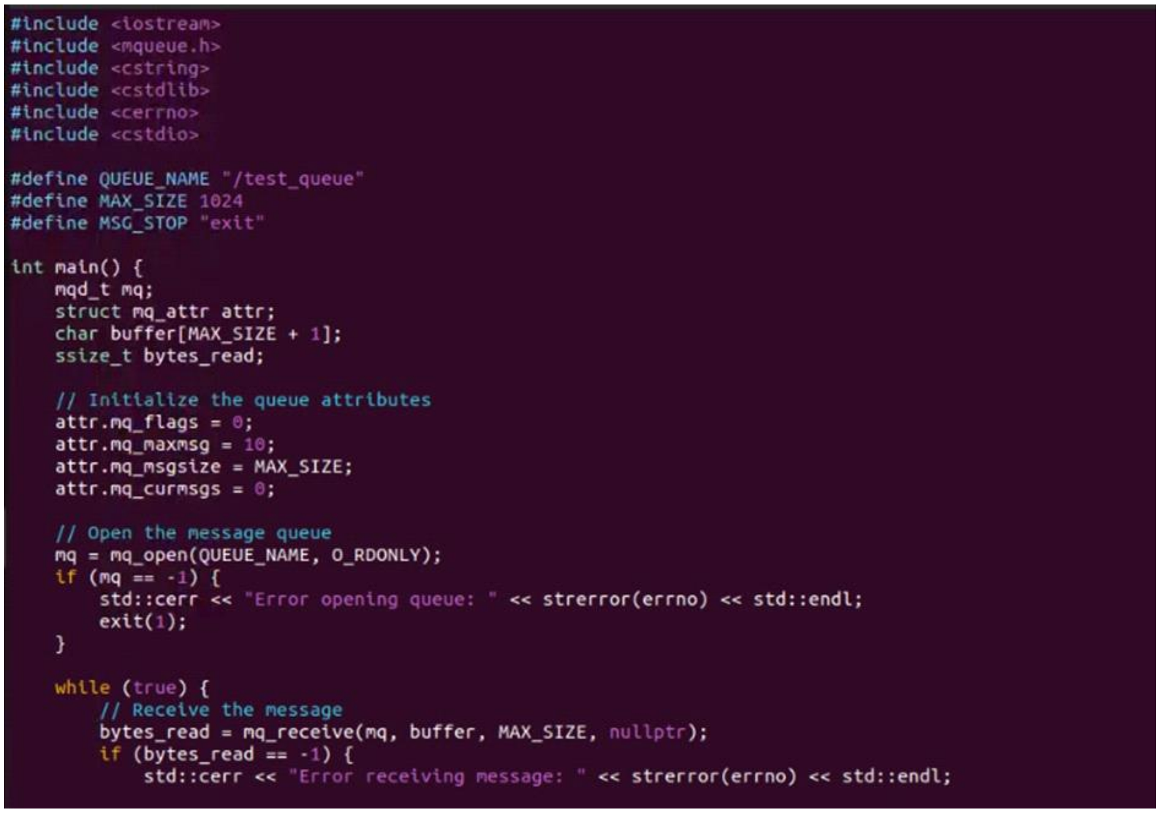
```
rps@rps-virtual-machine:~/SignalSocketApp$ ./senderloop
Enter a message: hi ameesha
Message sent: hi ameesha
Enter a message: how r u doing
Message sent: how r u doing
Enter a message: ok bye
Message sent: ok bye
Enter a message: exit
Message sent: exit
```


```
#include <iostream>
#include <mqueue.h>
#include <cstring>
#include <cstdlib>
#include <cerrno>
#include <cstdio>

#define QUEUE_NAME "/test_queue"
#define MAX_SIZE 1024
#define MSG_STOP "exit"

int main() {
    mqd_t mq;
    struct mq_attr attr;
    char buffer[MAX_SIZE];

    // Initialize the queue attributes
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = MAX_SIZE;
    attr.mq_curmsgs = 0;

    // Create the message queue
    mq = mq_open(QUEUE_NAME, O_CREAT | O_WRONLY, 0644, &attr);
    if (mq == -1) {
        std::cerr << "Error creating queue: " << strerror(errno) << std::endl;
        exit(1);
    }

    while (true) {
        std::cout << "Enter a message: ";
        std::cin.getline(buffer, MAX_SIZE);

        // Send the message
        if (mq_send(mq, buffer, strlen(buffer) + 1, 0) == -1) {
```

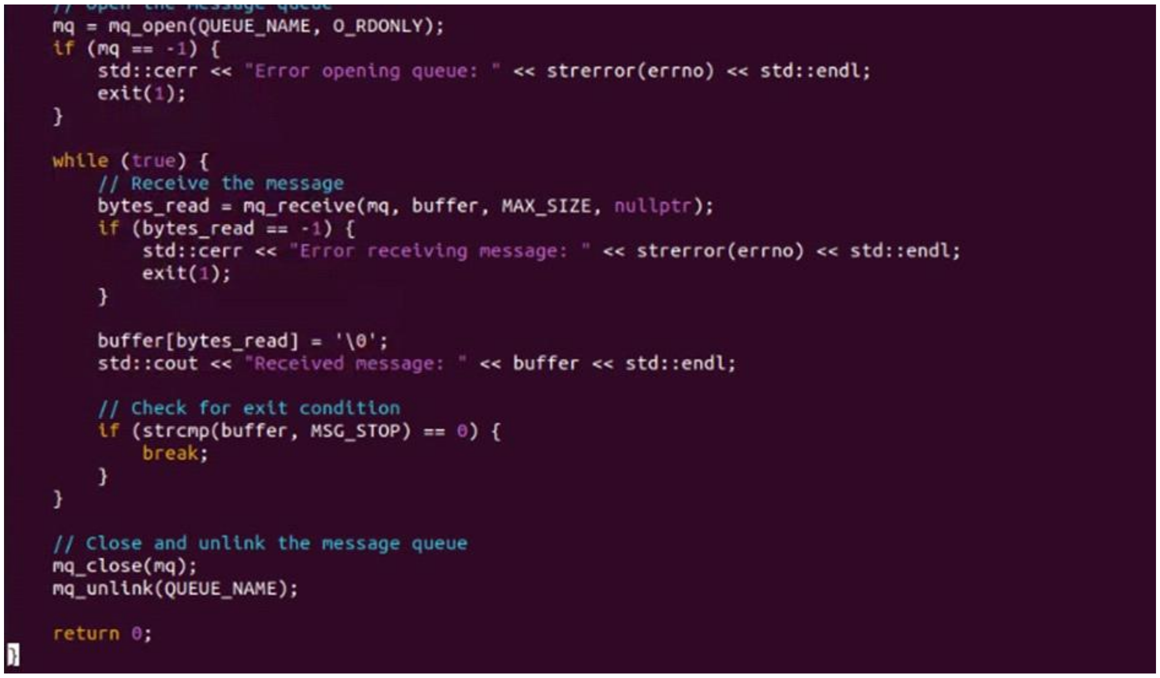

```
    while (true) {
        std::cout << "Enter a message: ";
        std::cin.getline(buffer, MAX_SIZE);

        // Send the message
        if (mq_send(mq, buffer, strlen(buffer) + 1, 0) == -1) {
            std::cerr << "Error sending message: " << strerror(errno) << std::endl;
            exit(1);
        }

        std::cout << "Message sent: " << buffer << std::endl;

        // Check for exit condition
        if (strcmp(buffer, MSG_STOP) == 0) {
            break;
        }
    }

    // Close the message queue
    mq_close(mq);

    return 0;
}
```

RECEIVER CODE :-


```
rps@rps-virtual-machine:~/SignalSocketApp$ ./receiversloop
Received message: hi ameesha
Received message: how r u doing
Received message: ok bye
Received message: exit
```



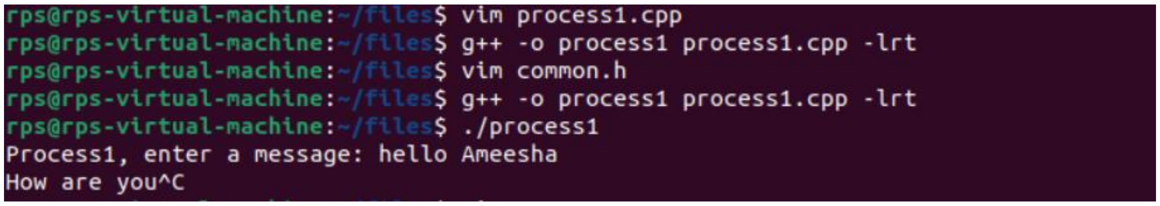
```cpp
#include <iostream>
#include <mqueue.h>
#include <cstring>
#include <cstdlib>
#include <cerrno>
#include <cstdio>

#define QUEUE_NAME "/test_queue"
#define MAX_SIZE 1024
#define MSG_STOP "exit"

int main() {
    mqd_t mq;
    struct mq_attr attr;
    char buffer[MAX_SIZE + 1];
    ssize_t bytes_read;

    // Initialize the queue attributes
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = MAX_SIZE;
    attr.mq_curmsgs = 0;

    // Open the message queue
    mq = mq_open(QUEUE_NAME, O_RDONLY);
    if (mq == -1) {
        std::cerr << "Error opening queue: " << strerror(errno) << std::endl;
        exit(1);
    }

    while (true) {
        // Receive the message
        bytes_read = mq_receive(mq, buffer, MAX_SIZE, nullptr);
        if (bytes_read == -1) {
            std::cerr << "Error receiving message: " << strerror(errno) << std::endl;
```



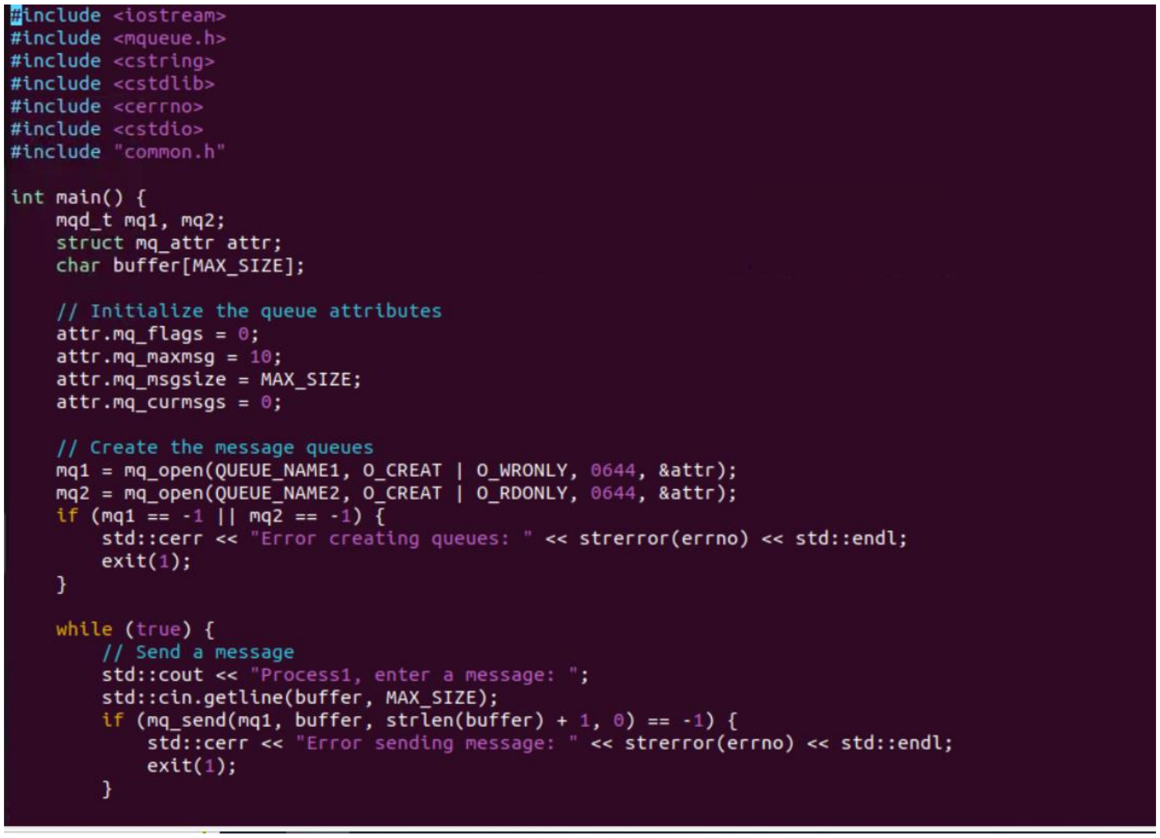
```cpp
    mq = mq_open(QUEUE_NAME, O_RDONLY);
    if (mq == -1) {
        std::cerr << "Error opening queue: " << strerror(errno) << std::endl;
        exit(1);
    }

    while (true) {
        // Receive the message
        bytes_read = mq_receive(mq, buffer, MAX_SIZE, nullptr);
        if (bytes_read == -1) {
            std::cerr << "Error receiving message: " << strerror(errno) << std::endl;
            exit(1);
        }

        buffer[bytes_read] = '\0';
        std::cout << "Received message: " << buffer << std::endl;

        // Check for exit condition
        if (strcmp(buffer, MSG_STOP) == 0) {
            break;
        }
    }

    // Close and unlink the message queue
    mq_close(mq);
    mq_unlink(QUEUE_NAME);

    return 0;
}
```

CODE FOR CONVESATION BETWEEN PROCESS1 AND PROCESS2 :-

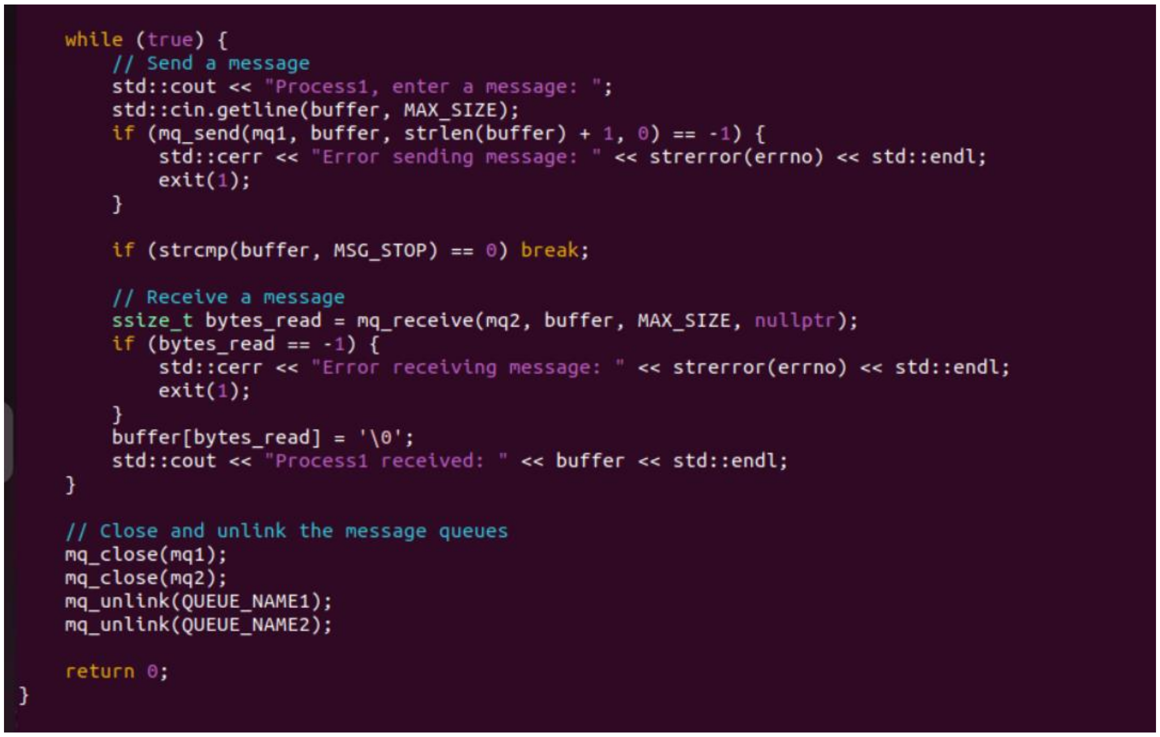
PROCESS1 CODE :-



```
rps@rps-virtual-machine:~/files$ vim process1.cpp
rps@rps-virtual-machine:~/files$ g++ -o process1 process1.cpp -lrt
rps@rps-virtual-machine:~/files$ vim common.h
rps@rps-virtual-machine:~/files$ g++ -o process1 process1.cpp -lrt
rps@rps-virtual-machine:~/files$ ./process1
Process1, enter a message: hello Ameesha
How are you^C
```

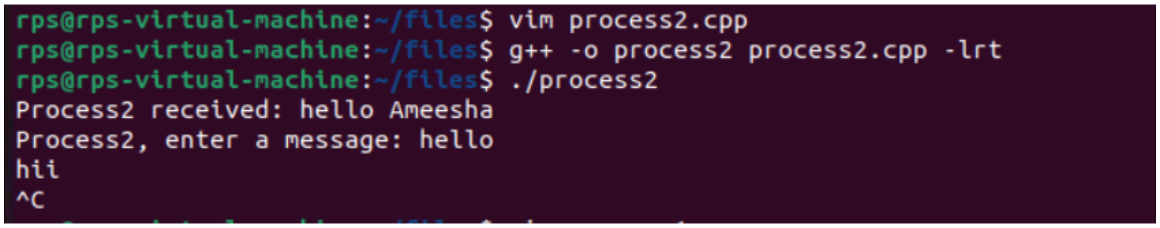


```
#include <iostream>
#include <mqueue.h>
#include <cstring>
#include <cstdlib>
#include <cerrno>
#include <cstdio>
#include "common.h"

int main() {
    mqd_t mq1, mq2;
    struct mq_attr attr;
    char buffer[MAX_SIZE];

    // Initialize the queue attributes
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = MAX_SIZE;
    attr.mq_curmsgs = 0;

    // Create the message queues
    mq1 = mq_open(QUEUE_NAME1, O_CREAT | O_WRONLY, 0644, &attr);
    mq2 = mq_open(QUEUE_NAME2, O_CREAT | O_RDONLY, 0644, &attr);
    if (mq1 == -1 || mq2 == -1) {
        std::cerr << "Error creating queues: " << strerror(errno) << std::endl;
        exit(1);
    }

    while (true) {
        // Send a message
        std::cout << "Process1, enter a message: ";
        std::cin.getline(buffer, MAX_SIZE);
        if (mq_send(mq1, buffer, strlen(buffer) + 1, 0) == -1) {
            std::cerr << "Error sending message: " << strerror(errno) << std::endl;
            exit(1);
        }
```

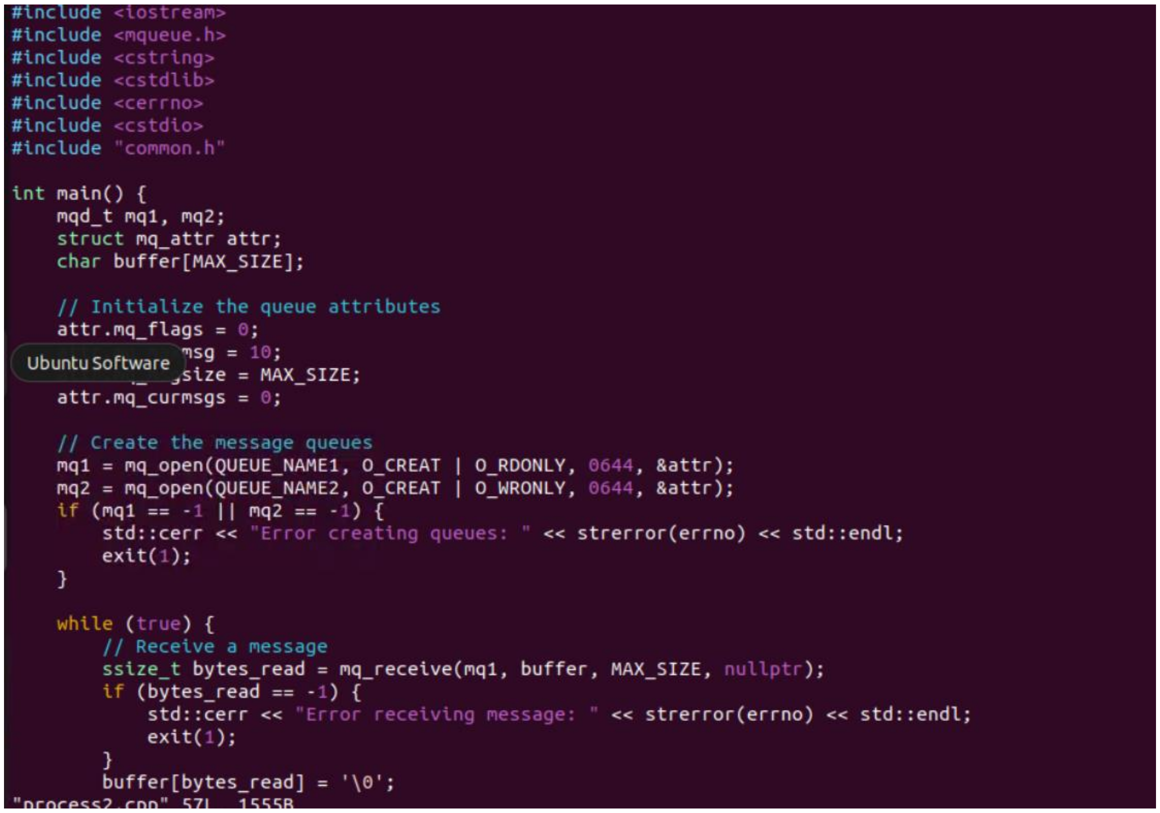


```
    while (true) {
        // Send a message
        std::cout << "Process1, enter a message: ";
        std::cin.getline(buffer, MAX_SIZE);
        if (mq_send(mq1, buffer, strlen(buffer) + 1, 0) == -1) {
            std::cerr << "Error sending message: " << strerror(errno) << std::endl;
            exit(1);
        }

        if (strcmp(buffer, MSG_STOP) == 0) break;

        // Receive a message
        ssize_t bytes_read = mq_receive(mq2, buffer, MAX_SIZE, nullptr);
        if (bytes_read == -1) {
            std::cerr << "Error receiving message: " << strerror(errno) << std::endl;
            exit(1);
        }
        buffer[bytes_read] = '\0';
        std::cout << "Process1 received: " << buffer << std::endl;
    }

    // Close and unlink the message queues
    mq_close(mq1);
    mq_close(mq2);
    mq_unlink(QUEUE_NAME1);
    mq_unlink(QUEUE_NAME2);

    return 0;
}
```

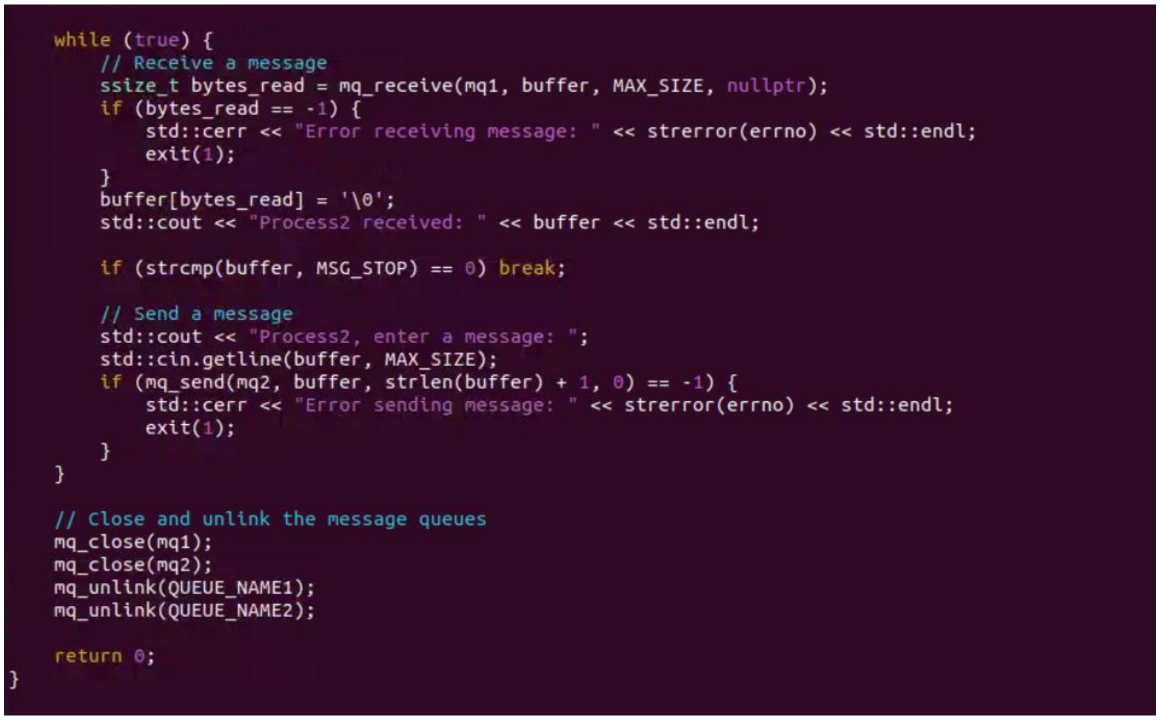
PROCESS2 CODE :-



```
rps@rps-virtual-machine:~/files$ vim process2.cpp
rps@rps-virtual-machine:~/files$ g++ -o process2 process2.cpp -lrt
rps@rps-virtual-machine:~/files$ ./process2
Process2 received: hello Ameesha
Process2, enter a message: hello
hii
^C
```

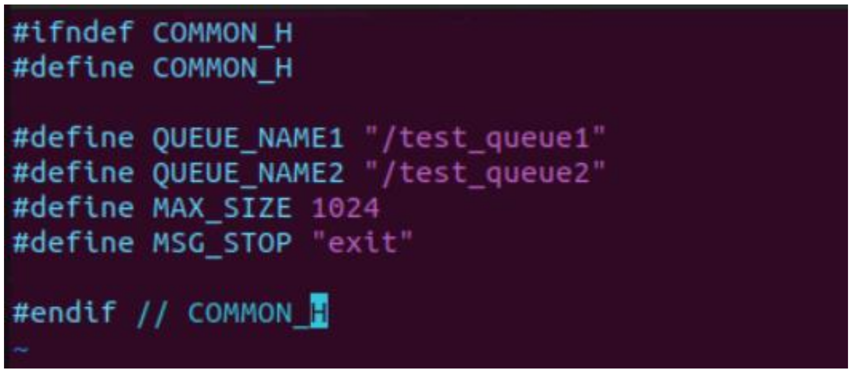


```
#include <iostream>
#include <mqueue.h>
#include <cstring>
#include <cstdlib>
#include <cerrno>
#include <cstdio>
#include "common.h"

int main() {
    mqd_t mq1, mq2;
    struct mq_attr attr;
    char buffer[MAX_SIZE];

    // Initialize the queue attributes
    attr.mq_flags = 0;
[illegible]msg = 10;
[illegible]size = MAX_SIZE;
    attr.mq_curmsgs = 0;

    // Create the message queues
    mq1 = mq_open(QUEUE_NAME1, O_CREAT | O_RDONLY, 0644, &attr);
    mq2 = mq_open(QUEUE_NAME2, O_CREAT | O_WRONLY, 0644, &attr);
    if (mq1 == -1 || mq2 == -1) {
        std::cerr << "Error creating queues: " << strerror(errno) << std::endl;
        exit(1);
    }

    while (true) {
        // Receive a message
        ssize_t bytes_read = mq_receive(mq1, buffer, MAX_SIZE, nullptr);
        if (bytes_read == -1) {
            std::cerr << "Error receiving message: " << strerror(errno) << std::endl;
            exit(1);
        }
        buffer[bytes_read] = '\0';
"process2.cpp" 57L, 1555B
```



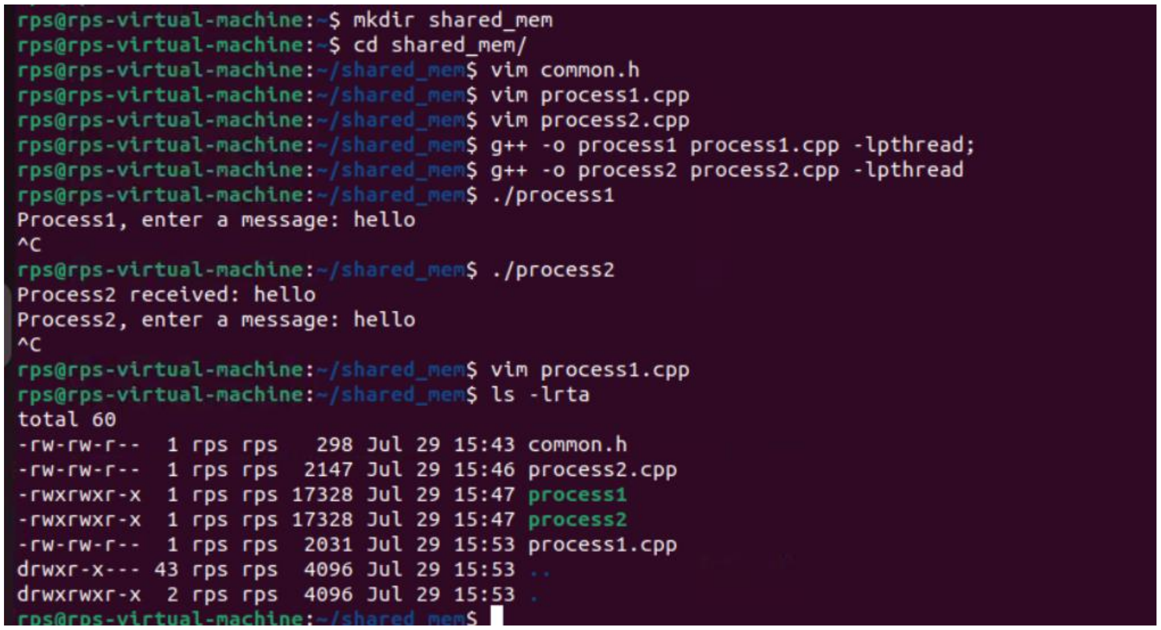
```
    while (true) {
        // Receive a message
        ssize_t bytes_read = mq_receive(mq1, buffer, MAX_SIZE, nullptr);
        if (bytes_read == -1) {
            std::cerr << "Error receiving message: " << strerror(errno) << std::endl;
            exit(1);
        }
        buffer[bytes_read] = '\0';
        std::cout << "Process2 received: " << buffer << std::endl;

        if (strcmp(buffer, MSG_STOP) == 0) break;

        // Send a message
        std::cout << "Process2, enter a message: ";
        std::cin.getline(buffer, MAX_SIZE);
        if (mq_send(mq2, buffer, strlen(buffer) + 1, 0) == -1) {
            std::cerr << "Error sending message: " << strerror(errno) << std::endl;
            exit(1);
        }
    }

    // Close and unlink the message queues
    mq_close(mq1);
    mq_close(mq2);
    mq_unlink(QUEUE_NAME1);
    mq_unlink(QUEUE_NAME2);

    return 0;
}
```

COMMON CODE :-



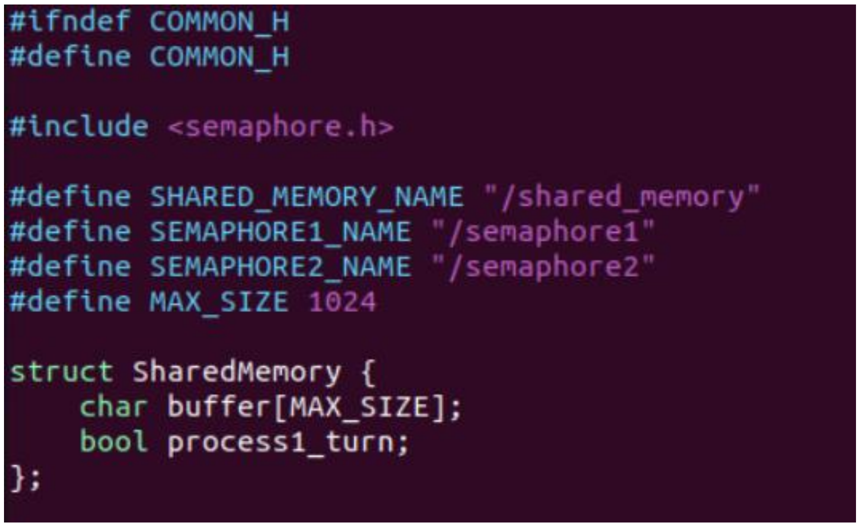
```
#ifndef COMMON_H
#define COMMON_H

#define QUEUE_NAME1 "/test_queue1"
#define QUEUE_NAME2 "/test_queue2"
#define MAX_SIZE 1024
#define MSG_STOP "exit"

#endif // COMMON_H
~
```

CODE FOR SHARED MEM :-



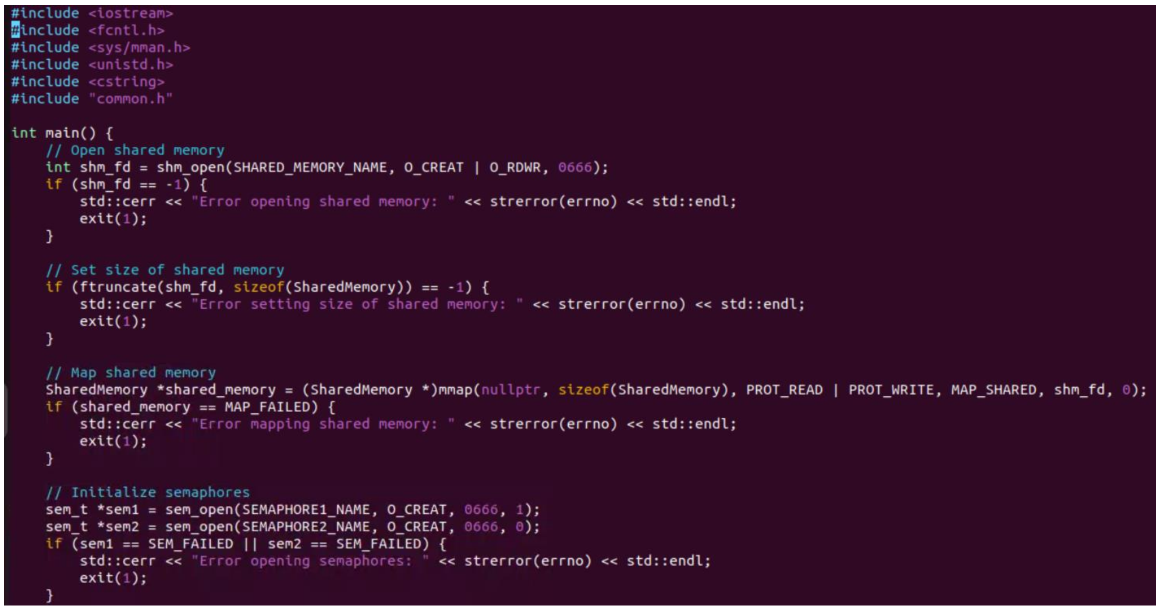
```
rps@rps-virtual-machine:~$ mkdir shared_mem
rps@rps-virtual-machine:~$ cd shared_mem/
rps@rps-virtual-machine:~/shared_mem$ vim common.h
rps@rps-virtual-machine:~/shared_mem$ vim process1.cpp
rps@rps-virtual-machine:~/shared_mem$ vim process2.cpp
rps@rps-virtual-machine:~/shared_mem$ g++ -o process1 process1.cpp -lpthread;
rps@rps-virtual-machine:~/shared_mem$ g++ -o process2 process2.cpp -lpthread
rps@rps-virtual-machine:~/shared_mem$ ./process1
Process1, enter a message: hello
^C
rps@rps-virtual-machine:~/shared_mem$ ./process2
Process2 received: hello
Process2, enter a message: hello
^C
rps@rps-virtual-machine:~/shared_mem$ vim process1.cpp
rps@rps-virtual-machine:~/shared_mem$ ls -lrta
total 60
-rw-rw-r--  1 rps rps   298 Jul 29 15:43 common.h
-rw-rw-r--  1 rps rps  2147 Jul 29 15:46 process2.cpp
-rwxrwxr-x  1 rps rps 17328 Jul 29 15:47 process1
-rwxrwxr-x  1 rps rps 17328 Jul 29 15:47 process2
-rw-rw-r--  1 rps rps  2031 Jul 29 15:53 process1.cpp
drwxr-x--- 43 rps rps  4096 Jul 29 15:53 ..
drwxrwxr-x  2 rps rps  4096 Jul 29 15:53 .
rps@rps-virtual-machine:~/shared_mem$ 
```

COMMON CODE :-

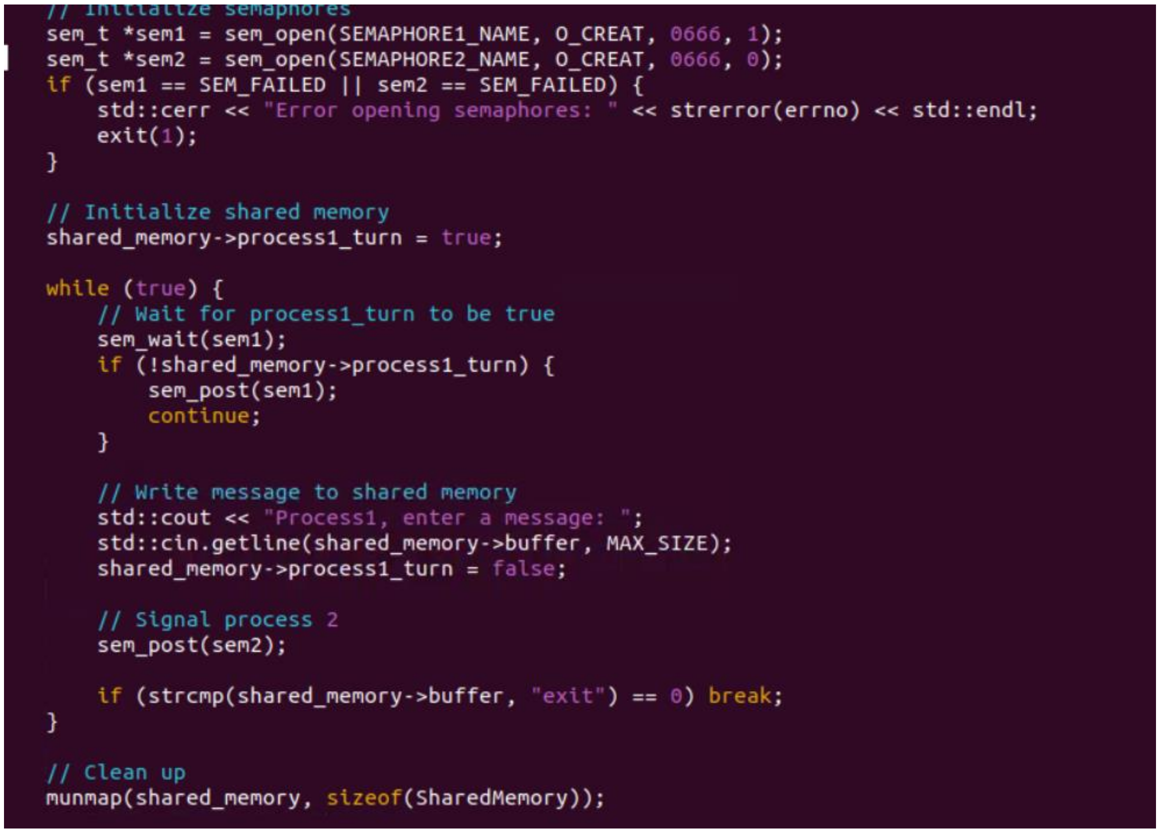


```
#ifndef COMMON_H
#define COMMON_H

#include <semaphore.h>

#define SHARED_MEMORY_NAME "/shared_memory"
#define SEMAPHORE1_NAME "/semaphore1"
#define SEMAPHORE2_NAME "/semaphore2"
#define MAX_SIZE 1024

struct SharedMemory {
    char buffer[MAX_SIZE];
    bool process1_turn;
};
```

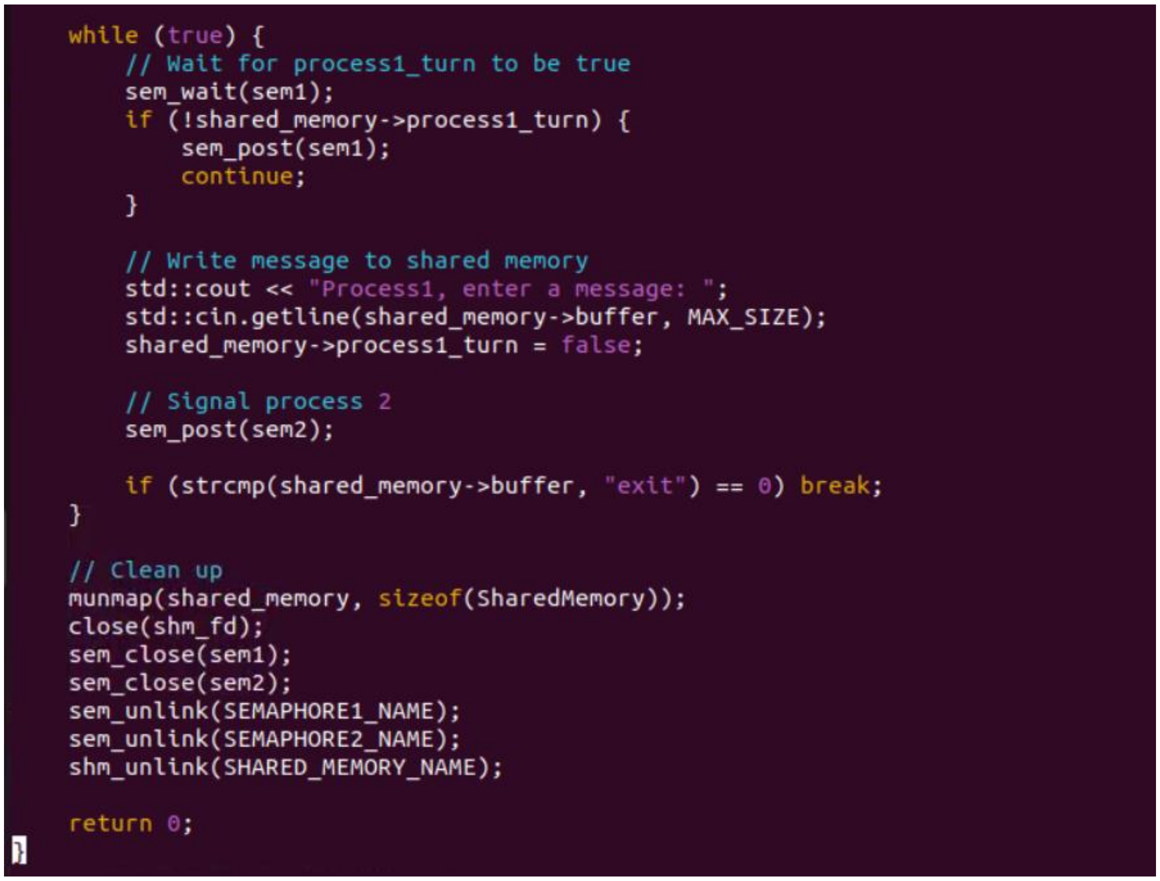
PROCESS1 CODE :-



```
#include <iostream>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <cstring>
#include "common.h"

int main() {
    // Open shared memory
    int shm_fd = shm_open(SHARED_MEMORY_NAME, O_CREAT | O_RDWR, 0666);
    if (shm_fd == -1) {
        std::cerr << "Error opening shared memory: " << strerror(errno) << std::endl;
        exit(1);
    }

    // Set size of shared memory
    if (ftruncate(shm_fd, sizeof(SharedMemory)) == -1) {
        std::cerr << "Error setting size of shared memory: " << strerror(errno) << std::endl;
        exit(1);
    }

    // Map shared memory
    SharedMemory *shared_memory = (SharedMemory *)mmap(nullptr, sizeof(SharedMemory), PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (shared_memory == MAP_FAILED) {
        std::cerr << "Error mapping shared memory: " << strerror(errno) << std::endl;
        exit(1);
    }

    // Initialize semaphores
    sem_t *sem1 = sem_open(SEMAPHORE1_NAME, O_CREAT, 0666, 1);
    sem_t *sem2 = sem_open(SEMAPHORE2_NAME, O_CREAT, 0666, 0);
    if (sem1 == SEM_FAILED || sem2 == SEM_FAILED) {
        std::cerr << "Error opening semaphores: " << strerror(errno) << std::endl;
        exit(1);
    }
```

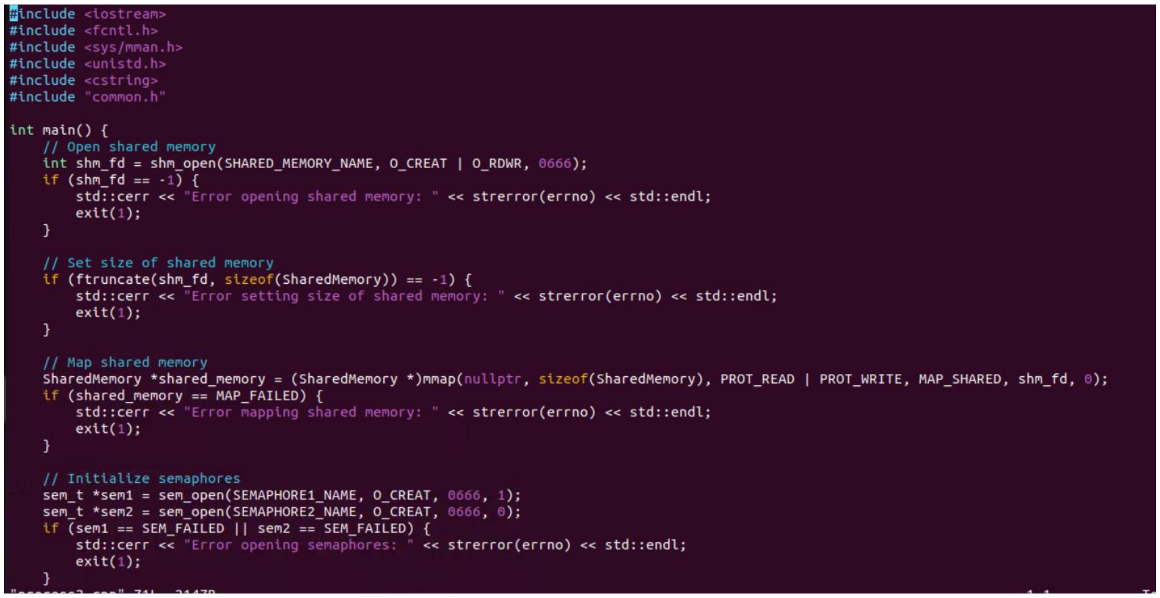


```
// Initialize semaphores
sem_t *sem1 = sem_open(SEMAPHORE1_NAME, O_CREAT, 0666, 1);
sem_t *sem2 = sem_open(SEMAPHORE2_NAME, O_CREAT, 0666, 0);
if (sem1 == SEM_FAILED || sem2 == SEM_FAILED) {
    std::cerr << "Error opening semaphores: " << strerror(errno) << std::endl;
    exit(1);
}

// Initialize shared memory
shared_memory->process1_turn = true;

while (true) {
    // Wait for process1_turn to be true
    sem_wait(sem1);
    if (!shared_memory->process1_turn) {
        sem_post(sem1);
        continue;
    }

    // Write message to shared memory
    std::cout << "Process1, enter a message: ";
    std::cin.getline(shared_memory->buffer, MAX_SIZE);
    shared_memory->process1_turn = false;

    // Signal process 2
    sem_post(sem2);

    if (strcmp(shared_memory->buffer, "exit") == 0) break;
}

// Clean up
munmap(shared_memory, sizeof(SharedMemory));
```

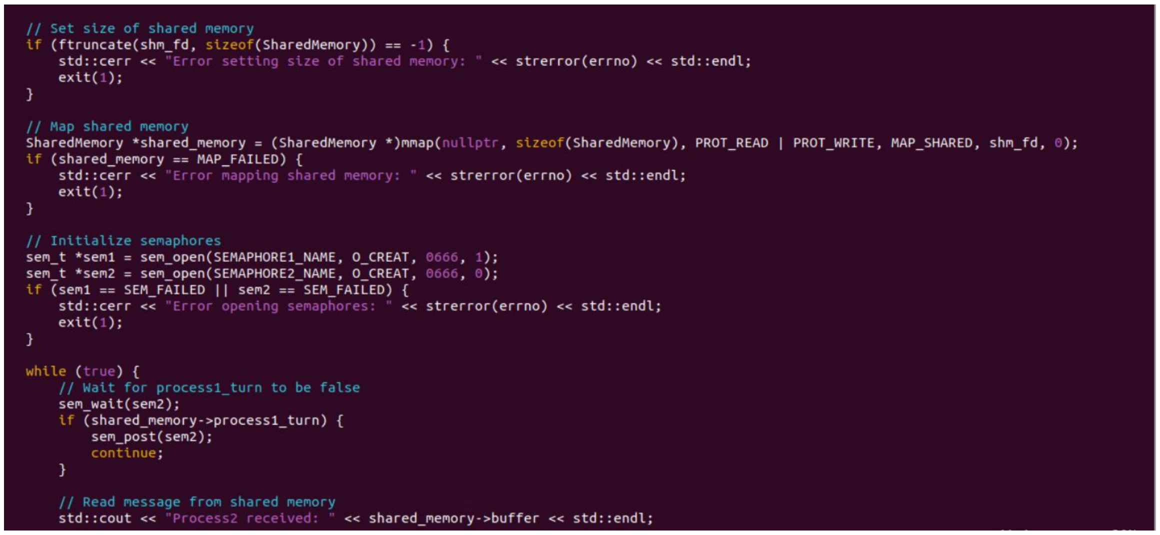


```
    while (true) {
        // Wait for process1_turn to be true
        sem_wait(sem1);
        if (!shared_memory->process1_turn) {
            sem_post(sem1);
            continue;
        }

        // Write message to shared memory
        std::cout << "Process1, enter a message: ";
        std::cin.getline(shared_memory->buffer, MAX_SIZE);
        shared_memory->process1_turn = false;

        // Signal process 2
        sem_post(sem2);

        if (strcmp(shared_memory->buffer, "exit") == 0) break;
    }

    // Clean up
    munmap(shared_memory, sizeof(SharedMemory));
    close(shm_fd);
    sem_close(sem1);
    sem_close(sem2);
    sem_unlink(SEMAPHORE1_NAME);
    sem_unlink(SEMAPHORE2_NAME);
    shm_unlink(SHARED_MEMORY_NAME);

    return 0;
}
```

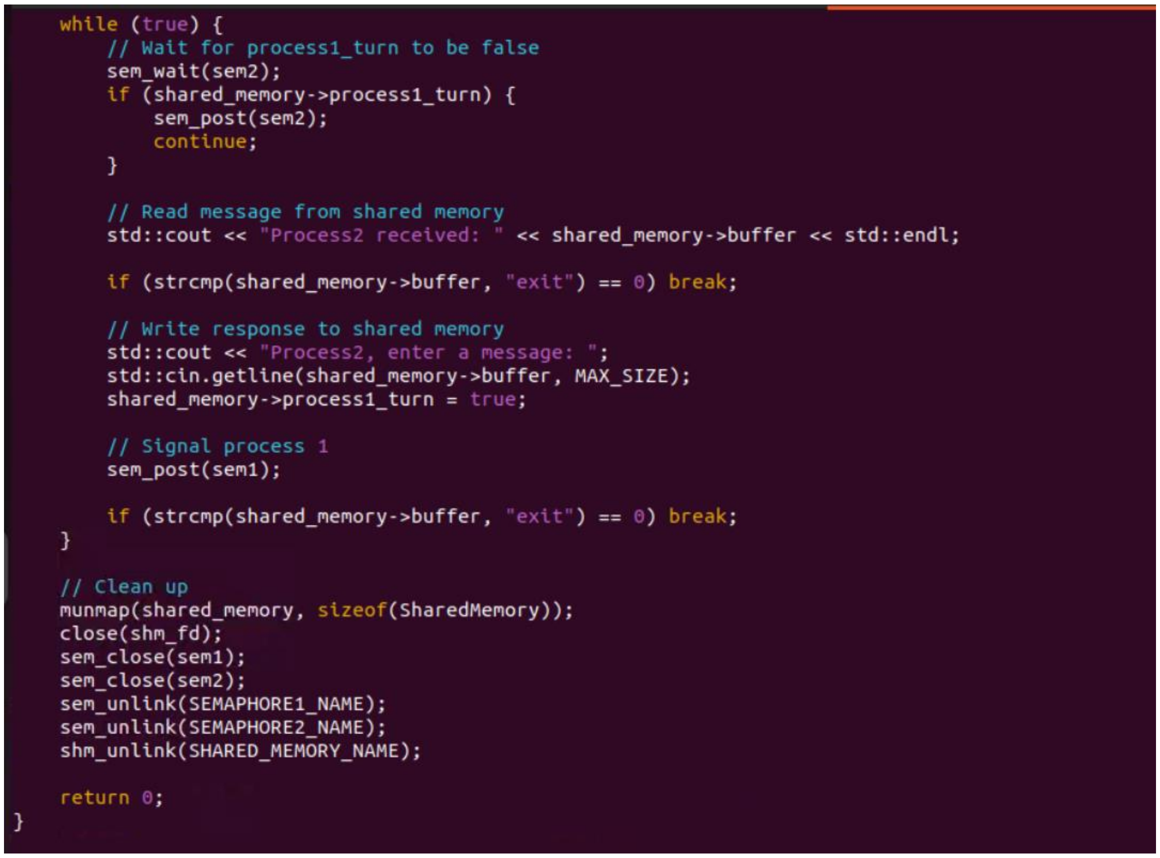
PROCESS2 CODE :-



```
#include <iostream>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <cstring>
#include "common.h"

int main() {
    // Open shared memory
    int shm_fd = shm_open(SHARED_MEMORY_NAME, O_CREAT | O_RDWR, 0666);
    if (shm_fd == -1) {
        std::cerr << "Error opening shared memory: " << strerror(errno) << std::endl;
        exit(1);
    }

    // Set size of shared memory
    if (ftruncate(shm_fd, sizeof(SharedMemory)) == -1) {
        std::cerr << "Error setting size of shared memory: " << strerror(errno) << std::endl;
        exit(1);
    }

    // Map shared memory
    SharedMemory *shared_memory = (SharedMemory *)mmap(nullptr, sizeof(SharedMemory), PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (shared_memory == MAP_FAILED) {
        std::cerr << "Error mapping shared memory: " << strerror(errno) << std::endl;
        exit(1);
    }

    // Initialize semaphores
    sem_t *sem1 = sem_open(SEMAPHORE1_NAME, O_CREAT, 0666, 1);
    sem_t *sem2 = sem_open(SEMAPHORE2_NAME, O_CREAT, 0666, 0);
    if (sem1 == SEM_FAILED || sem2 == SEM_FAILED) {
        std::cerr << "Error opening semaphores: " << strerror(errno) << std::endl;
        exit(1);
    }
```

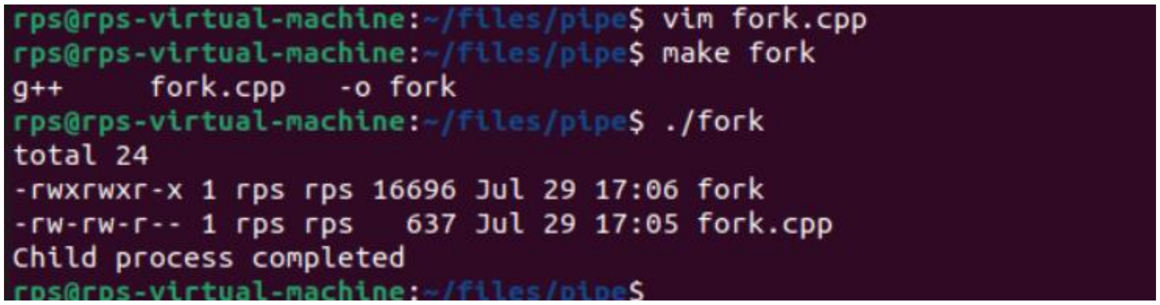


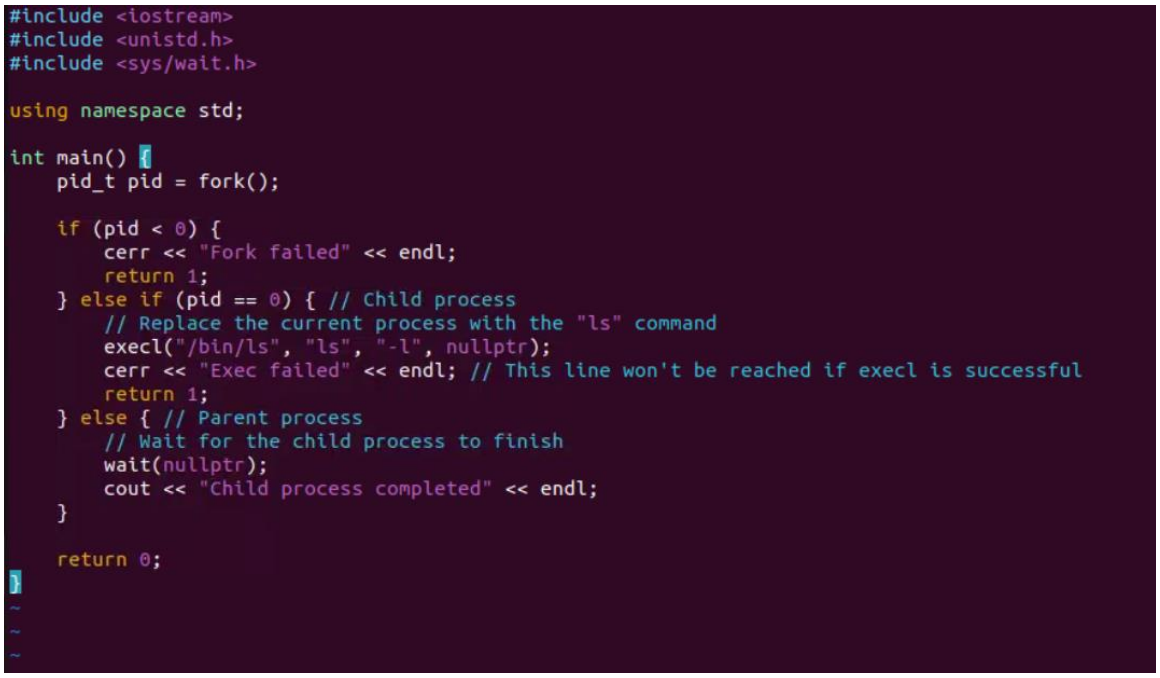
```
// Set size of shared memory
if (ftruncate(shm_fd, sizeof(SharedMemory)) == -1) {
    std::cerr << "Error setting size of shared memory: " << strerror(errno) << std::endl;
    exit(1);
}

// Map shared memory
SharedMemory *shared_memory = (SharedMemory *)mmap(nullptr, sizeof(SharedMemory), PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
if (shared_memory == MAP_FAILED) {
    std::cerr << "Error mapping shared memory: " << strerror(errno) << std::endl;
    exit(1);
}

// Initialize semaphores
sem_t *sem1 = sem_open(SEMAPHORE1_NAME, O_CREAT, 0666, 1);
sem_t *sem2 = sem_open(SEMAPHORE2_NAME, O_CREAT, 0666, 0);
if (sem1 == SEM_FAILED || sem2 == SEM_FAILED) {
    std::cerr << "Error opening semaphores: " << strerror(errno) << std::endl;
    exit(1);
}

while (true) {
    // Wait for process1_turn to be false
    sem_wait(sem2);
    if (shared_memory->process1_turn) {
        sem_post(sem2);
        continue;
    }

    // Read message from shared memory
    std::cout << "Process2 received: " << shared_memory->buffer << std::endl;
```



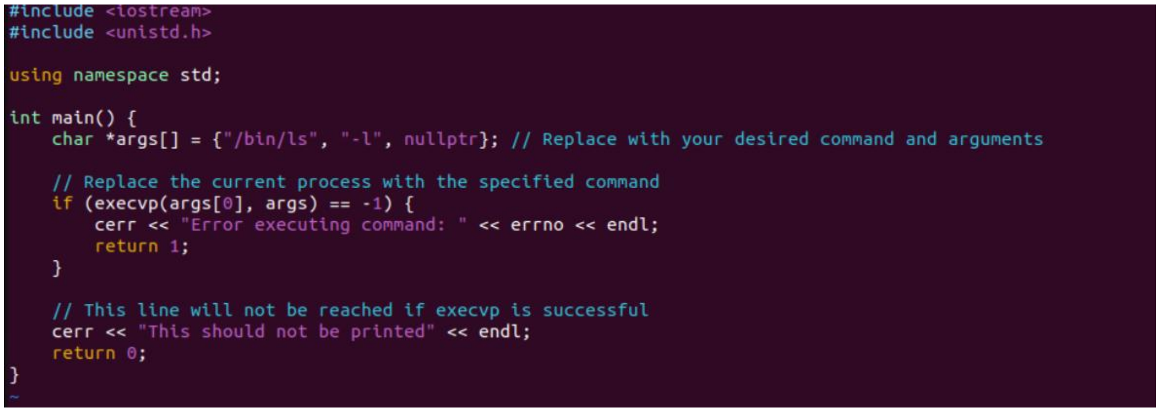
```
    while (true) {
        // Wait for process1_turn to be false
        sem_wait(sem2);
        if (shared_memory->process1_turn) {
            sem_post(sem2);
            continue;
        }

        // Read message from shared memory
        std::cout << "Process2 received: " << shared_memory->buffer << std::endl;

        if (strcmp(shared_memory->buffer, "exit") == 0) break;

        // Write response to shared memory
        std::cout << "Process2, enter a message: ";
        std::cin.getline(shared_memory->buffer, MAX_SIZE);
        shared_memory->process1_turn = true;

        // Signal process 1
        sem_post(sem1);

        if (strcmp(shared_memory->buffer, "exit") == 0) break;
    }

    // Clean up
    munmap(shared_memory, sizeof(SharedMemory));
    close(shm_fd);
    sem_close(sem1);
    sem_close(sem2);
    sem_unlink(SEMAPHORE1_NAME);
    sem_unlink(SEMAPHORE2_NAME);
    shm_unlink(SHARED_MEMORY_NAME);

    return 0;
}
```

CODE FOR FORK :-


```
rps@rps-virtual-machine:~/files/pipe$ vim fork.cpp
rps@rps-virtual-machine:~/files/pipe$ make fork
g++     fork.cpp   -o fork
rps@rps-virtual-machine:~/files/pipe$ ./fork
total 24
-rwxrwxr-x 1 rps rps 16696 Jul 29 17:06 fork
-rw-rw-r-- 1 rps rps   637 Jul 29 17:05 fork.cpp
Child process completed
rps@rps-virtual-machine:~/files/pipe$
```


```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>

using namespace std;

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        cerr << "Fork failed" << endl;
        return 1;
    } else if (pid == 0) { // Child process
        // Replace the current process with the "ls" command
        execl("/bin/ls", "ls", "-l", nullptr);
        cerr << "Exec failed" << endl; // This line won't be reached if execl is successful
        return 1;
    } else { // Parent process
        // Wait for the child process to finish
        wait(nullptr);
        cout << "Child process completed" << endl;
    }

    return 0;
}
~
~
~
```

CODE FOR EXEC :-


```cpp
#include <iostream>
#include <unistd.h>

using namespace std;

int main() {
    char *args[] = {"/bin/ls", "-l", nullptr}; // Replace with your desired command and arguments

    // Replace the current process with the specified command
    if (execvp(args[0], args) == -1) {
        cerr << "Error executing command: " << errno << endl;
        return 1;
    }

    // This line will not be reached if execvp is successful
    cerr << "This should not be printed" << endl;
    return 0;
}
~
```