

Day – 7 LSP Assignment (Task – 1)

1. 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.

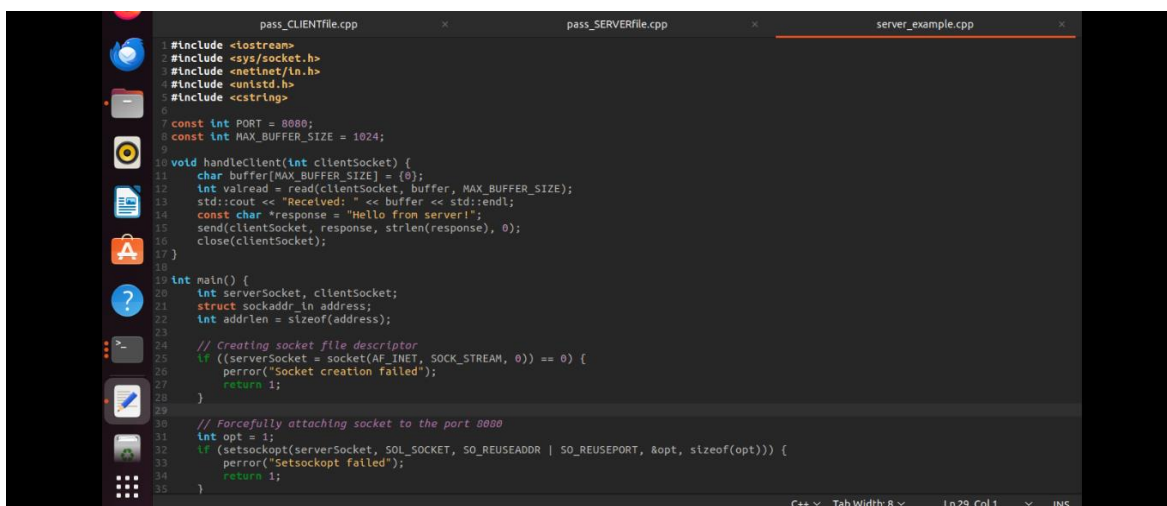
Data transfer: Develop functions for sending and receiving data over the established socket connection.

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).

a. server – side code



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

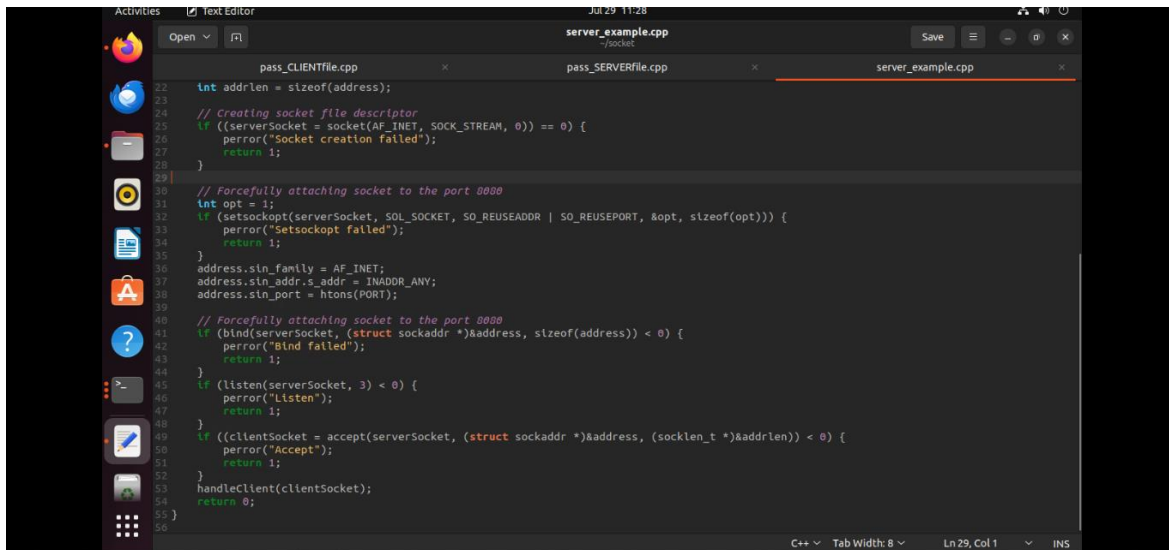
const int PORT = 8080;
const int MAX_BUFFER_SIZE = 1024;

void handleClient(int clientSocket) {
    char buffer[MAX_BUFFER_SIZE] = {0};
    int valread = read(clientSocket, buffer, MAX_BUFFER_SIZE);
    std::cout << "Received: " << buffer << std::endl;
    const char *response = "Hello from server!";
    send(clientSocket, response, strlen(response), 0);
    close(clientSocket);
}

int main() {
    int serverSocket, clientSocket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    // Creating socket file descriptor
    if ((serverSocket = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("Socket creation failed");
        return 1;
    }

    // Forcefully attaching socket to the port 8080
    int opt = 1;
    if (setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))) {
        perror("Setsockopt failed");
        return 1;
    }
}
```



```
server_example.cpp
~/socket

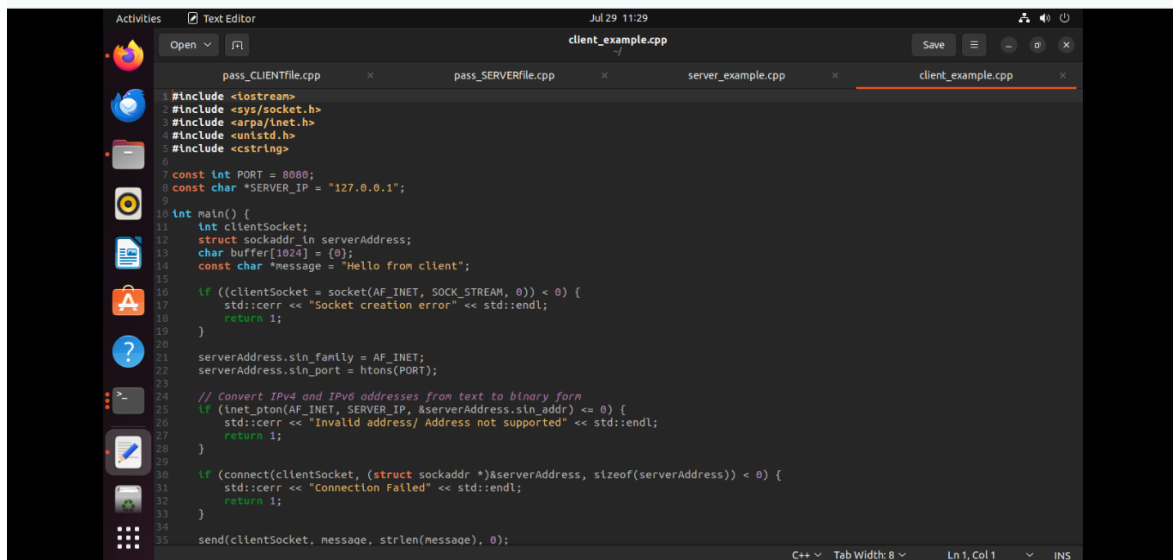
pass_CLIENTFile.cpp  pass_SERVERFile.cpp  server_example.cpp

22 int addrlen = sizeof(address);
23
24 // Creating socket file descriptor
25 if ((serverSocket = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
26     perror("Socket creation failed");
27     return 1;
28 }
29
30 // Forcefully attaching socket to the port 8080
31 int opt = 1;
32 if (setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt)) {
33     perror("Setsockopt failed");
34     return 1;
35 }
36 address.sin_family = AF_INET;
37 address.sin_addr.s_addr = INADDR_ANY;
38 address.sin_port = htons(PORT);
39
40 // Forcefully attaching socket to the port 8080
41 if (bind(serverSocket, (struct sockaddr *)&address, sizeof(address)) < 0) {
42     perror("Bind failed");
43     return 1;
44 }
45 if (listen(serverSocket, 3) < 0) {
46     perror("Listen");
47     return 1;
48 }
49 if ((clientSocket = accept(serverSocket, (struct sockaddr *)&address, (socklen_t *)&addrlen)) < 0) {
50     perror("Accept");
51     return 1;
52 }
53 handleClient(clientSocket);
54 return 0;
55 }
56
```



```
make: Nothing to be done for 'pass_CLIENTFile.cpp'.
rps@rps-virtual-machine:~/socket$ make -f Makefile
g++ -std=c++11 -c server_example.cpp -o server_example.o
g++ -std=c++11 -o server_example server_example.o
rps@rps-virtual-machine:~/socket$ ./server_example
Received: Hello from client
rps@rps-virtual-machine:~/socket$
```

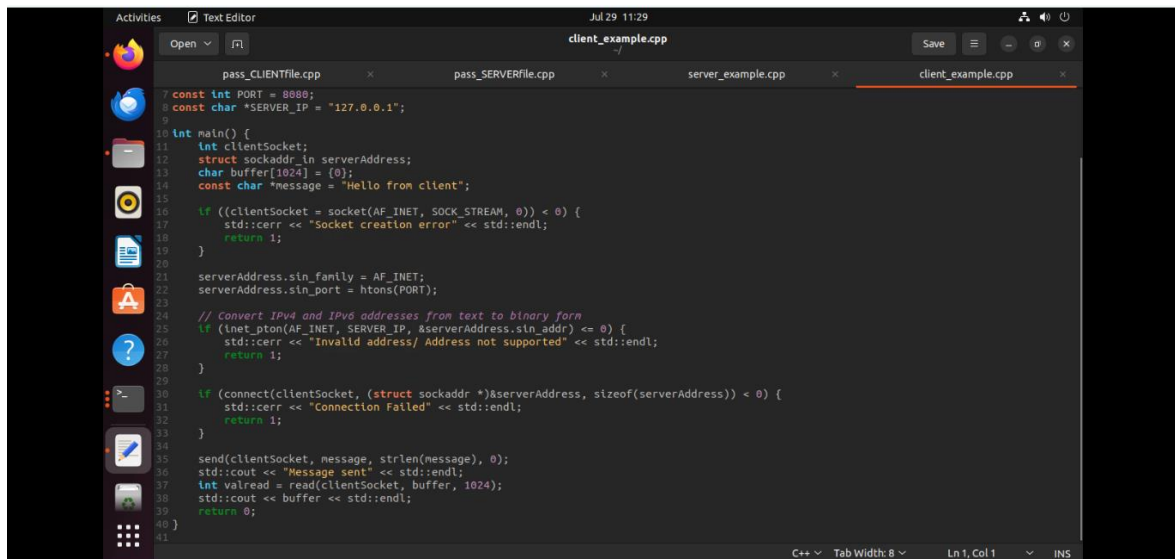
b. client – side code



```
client_example.cpp
~/

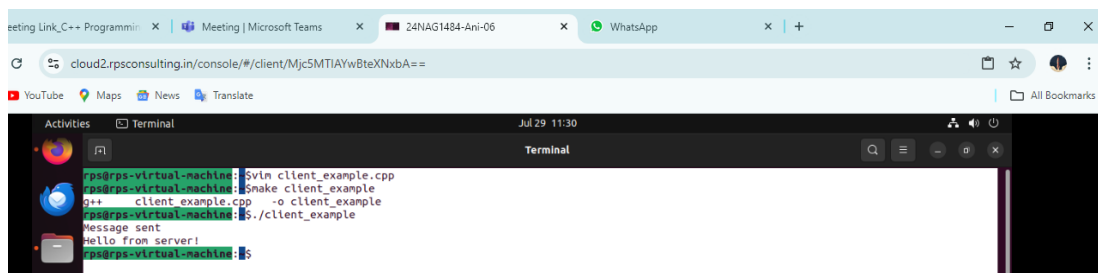
pass_CLIENTFile.cpp  pass_SERVERFile.cpp  server_example.cpp  client_example.cpp

1 #include <iostream>
2 #include <sys/socket.h>
3 #include <arpa/inet.h>
4 #include <unistd.h>
5 #include <cstring>
6
7 const int PORT = 8080;
8 const char *SERVER_IP = "127.0.0.1";
9
10 int main() {
11     int clientSocket;
12     struct sockaddr_in serverAddress;
13     char buffer[1024] = {0};
14     const char *message = "Hello from client";
15
16     if ((clientSocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
17         std::cerr << "Socket creation error" << std::endl;
18         return 1;
19     }
20
21     serverAddress.sin_family = AF_INET;
22     serverAddress.sin_port = htons(PORT);
23
24     // Convert IPv4 and IPv6 addresses from text to binary form
25     if (inet_pton(AF_INET, SERVER_IP, &serverAddress.sin_addr) <= 0) {
26         std::cerr << "Invalid address/ Address not supported" << std::endl;
27         return 1;
28     }
29
30     if (connect(clientSocket, (struct sockaddr *)&serverAddress, sizeof(serverAddress)) < 0) {
31         std::cerr << "Connection Failed" << std::endl;
32         return 1;
33     }
34
35     send(clientSocket, message, strlen(message), 0);
36 }
```



The screenshot shows a text editor window titled 'client_example.cpp' with the following code:

```
1 const int PORT = 8080;
2 const char *SERVER_IP = "127.0.0.1";
3
4 int main() {
5     int clientSocket;
6     struct sockaddr_in serverAddress;
7     char buffer[1024] = {0};
8     const char *message = "Hello from client";
9
10    if ((clientSocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
11        std::cerr << "Socket creation error" << std::endl;
12        return 1;
13    }
14
15    serverAddress.sin_family = AF_INET;
16    serverAddress.sin_port = htons(PORT);
17
18    // Convert IPv4 and IPv6 addresses from text to binary form
19    if (inet_pton(AF_INET, SERVER_IP, &serverAddress.sin_addr) <= 0) {
20        std::cerr << "Invalid address/ Address not supported" << std::endl;
21        return 1;
22    }
23
24    if (connect(clientSocket, (struct sockaddr *)&serverAddress, sizeof(serverAddress)) < 0) {
25        std::cerr << "Connection Failed" << std::endl;
26        return 1;
27    }
28
29    send(clientSocket, message, strlen(message), 0);
30    std::cout << "Message sent" << std::endl;
31    int valread = read(clientSocket, buffer, 1024);
32    std::cout << buffer << std::endl;
33    return 0;
34 }
```

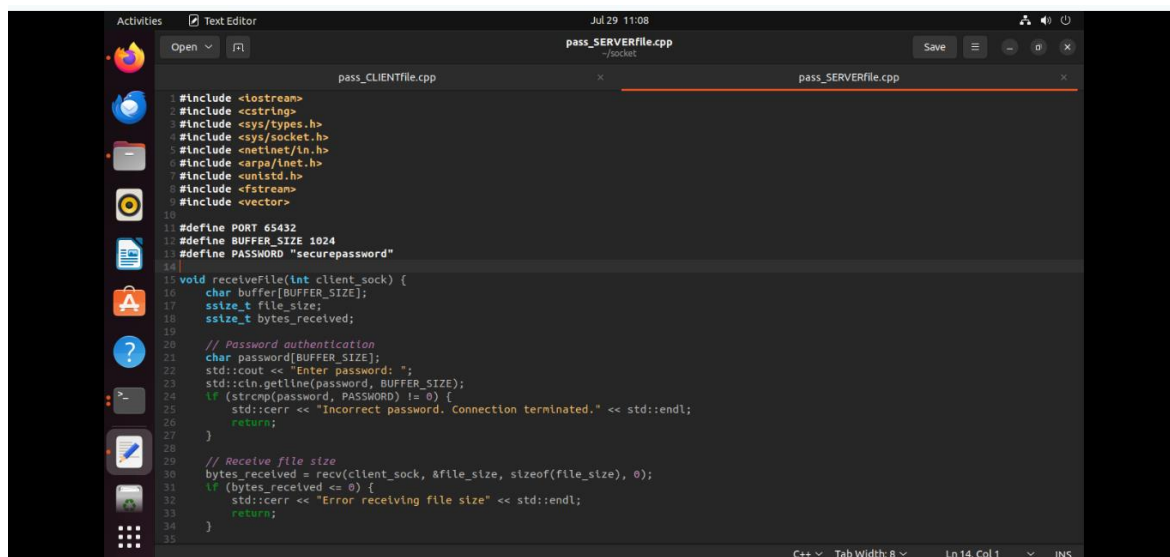


The screenshot shows a terminal window with the following commands and output:

```
ps@rps-virtual-machine:~$ g++ client_example.cpp
ps@rps-virtual-machine:~$ ./client_example
Message sent
Hello from server!
```

2. "Explain the role of password authentication and file transmission in your client-server application implementation."

a. server - side code



The screenshot shows a text editor window titled 'pass_SERVERFile.cpp' with the following code:

```
1 #include <iostream>
2 #include <cstring>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <arpa/inet.h>
7 #include <unistd.h>
8 #include <fstream>
9 #include <vector>
10
11 #define PORT 65432
12 #define BUFFER_SIZE 1024
13 #define PASSWORD "securepassword"
14
15 void receiveFile(int client_sock) {
16     char buffer[BUFFER_SIZE];
17     ssize_t file_size;
18     ssize_t bytes_received;
19
20     // Password authentication
21     char password[BUFFER_SIZE];
22     std::cout << "Enter password: ";
23     std::cin.getline(password, BUFFER_SIZE);
24     if (strcmp(password, PASSWORD) != 0) {
25         std::cerr << "Incorrect password. Connection terminated." << std::endl;
26         return;
27     }
28
29     // Receive file size
30     bytes_received = recv(client_sock, &file_size, sizeof(file_size), 0);
31     if (bytes_received <= 0) {
32         std::cerr << "Error receiving file size" << std::endl;
33         return;
34     }
35 }
```

```
pass_CLIENTfile.cpp
38 std::ofstream file("received_SERVERfile.txt", std::ios::binary);
39 if (!file.is_open()) {
40     std::cerr << "Failed to open file to write" << std::endl;
41     return;
42 }
43
44 ssize_t total_received = 0;
45 while (total_received < file_size) {
46     bytes_received = recv(client_sock, buffer, BUFFER_SIZE, 0);
47     if (bytes_received <= 0) {
48         std::cerr << "Error receiving file" << std::endl;
49         return;
50     }
51     file.write(buffer, bytes_received);
52     total_received += bytes_received;
53 }
54
55 std::cout << "File received from 'SERVER' successfully....." << std::endl;
56 file.close();
57 }
58
59 void sendFile(int client_sock, const std::string& filename) {
60     std::ifstream file(filename, std::ios::binary | std::ios::ate);
61     if (!file.is_open()) {
62         std::cerr << "Error opening file: " << filename << std::endl;
63         return;
64     }
65
66     std::streamsize file_size = file.tellg();
67     file.seekg(0, std::ios::beg);
68
69     // Convert file size to network byte order
70     file_size = htonl(file_size);
71     send(client_sock, &file_size, sizeof(file_size), 0);
72 }
```

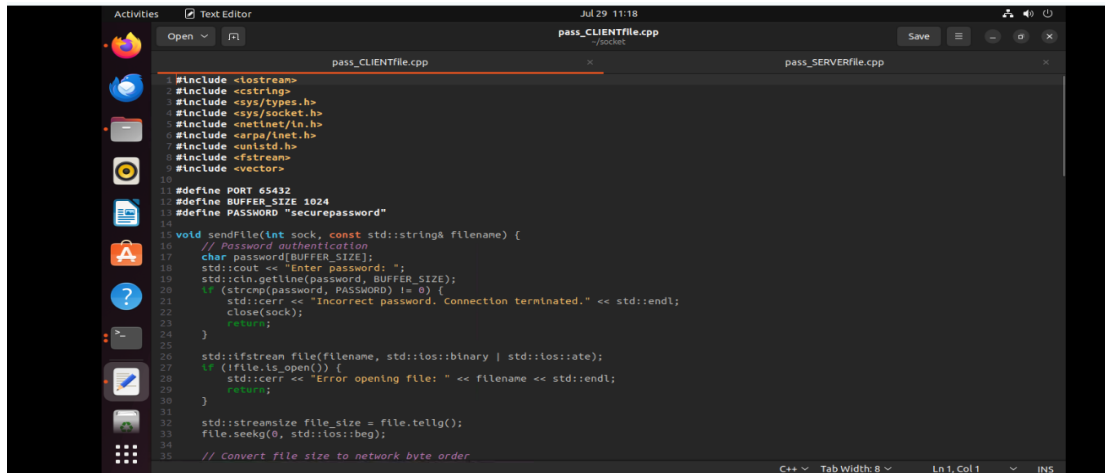
```

73 std::vector<char> buffer(BUFFER_SIZE);
74 while (!file.eof()) {
75     file.read(buffer.data(), buffer.size());
76     ssize_t bytes_read = file.gcount();
77     if (bytes_read <= 0) break;
78     send(client_sock, buffer.data(), bytes_read, 0);
79 }
80
81 file.close();
82 std::cout << "File sent from 'SERVER' successfully....." << std::endl;
83 }
84
85 int main() {
86     int server_fd, client_sock;
87     struct sockaddr_in address;
88     int opt = 1;
89     int addrlen = sizeof(address);
90
91     // Creating socket file descriptor
92     if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
93         perror("socket failed");
94         exit(EXIT_FAILURE);
95     }
96
97     // Forcefully attaching socket to the port 65432
98     if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))) {
99         perror("setsockopt");
100         exit(EXIT_FAILURE);
101     }
102     address.sin_family = AF_INET;
103     address.sin_addr.s_addr = INADDR_ANY;
104     address.sin_port = htons(PORT);
105 }
```

```

106 // Forcefully attaching socket to the port 65432
107 if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
108     perror("bind failed");
109     exit(EXIT_FAILURE);
110 }
111 if (listen(server_fd, 3) < 0) {
112     perror("listen");
113     exit(EXIT_FAILURE);
114 }
115 if ((client_sock = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen)) < 0) {
116     perror("accept");
117     exit(EXIT_FAILURE);
118 }
119
120 // Handle client connection
121 std::cout << "Client side connected" << std::endl;
122
123 // Receive a command from the client
124 char command[BUFFER_SIZE];
125 ssize_t bytes_received = recv(client_sock, command, BUFFER_SIZE, 0);
126 if (bytes_received <= 0) {
127     std::cerr << "Error receiving command" << std::endl;
128     close(client_sock);
129     close(server_fd);
130     return 1;
131 }
132
133 command[bytes_received] = '\0';
134
135 // Check the received command
136 if (strcmp(command, "get") == 0) {
137     // Client requests to receive a file
138     receiveFile(client_sock);
139 } else if (strcmp(command, "send") == 0) {
140     // Client requests to send a file
141     std::string file_to_send = "pass_SERVERfile.txt"; // Replace with actual file name
142     sendFile(client_sock, file_to_send);
143 } else {
144     std::cerr << "Invalid command received: " << command << std::endl;
145 }
146
147 close(client_sock);
148 close(server_fd);
149 return 0;
150 }
151
```

b. client – side code



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

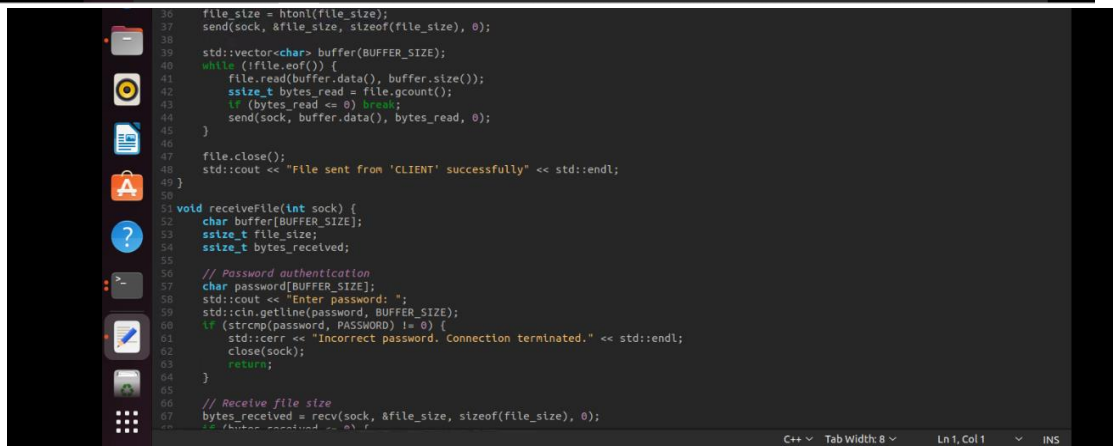
#define PORT 65432
#define BUFFER_SIZE 1024
#define PASSWORD "securepassword"

void sendFile(int sock, const std::string& filename) {
    // Password authentication
    char password[BUFFER_SIZE];
    std::cout << "Enter password: ";
    std::cin.getline(password, BUFFER_SIZE);
    if (strcmp(password, PASSWORD) != 0) {
        std::cerr << "Incorrect password. Connection terminated." << std::endl;
        close(sock);
        return;
    }

    std::ifstream file(filename, std::ios::binary | std::ios::ate);
    if (!file.is_open()) {
        std::cerr << "Error opening file: " << filename << std::endl;
        return;
    }

    std::streamsize file_size = file.tellg();
    file.seekg(0, std::ios::beg);

    // Convert file size to network byte order
```



```
file_size = htonl(file_size);
send(sock, &file_size, sizeof(file_size), 0);

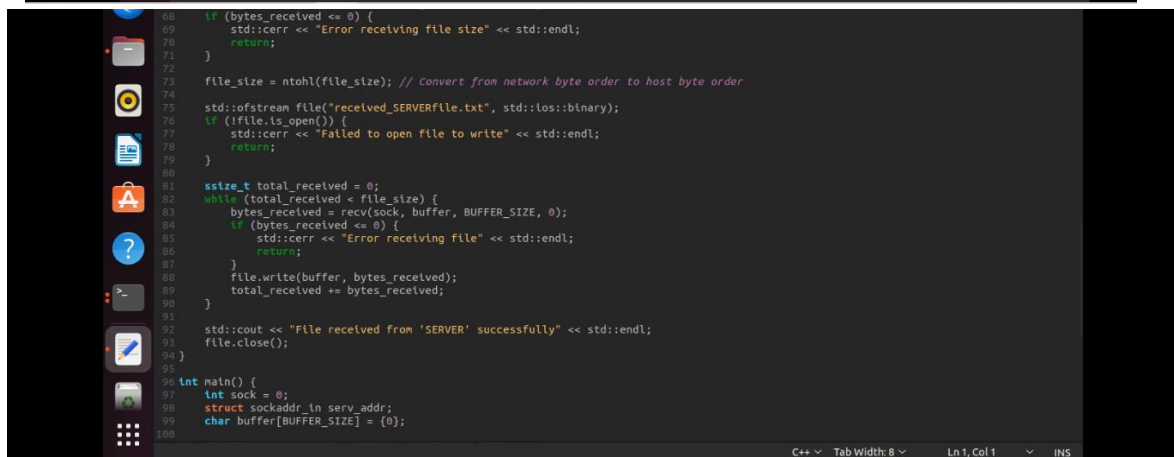
std::vector<char> buffer(BUFFER_SIZE);
while (!file.eof()) {
    file.read(buffer.data(), buffer.size());
    ssize_t bytes_read = file.gcount();
    if (bytes_read <= 0) break;
    send(sock, buffer.data(), bytes_read, 0);
}

file.close();
std::cout << "File sent from 'CLIENT' successfully" << std::endl;
}

void receiveFile(int sock) {
    char buffer[BUFFER_SIZE];
    ssize_t file_size;
    ssize_t bytes_received;

    // Password authentication
    char password[BUFFER_SIZE];
    std::cout << "Enter password: ";
    std::cin.getline(password, BUFFER_SIZE);
    if (strcmp(password, PASSWORD) != 0) {
        std::cerr << "Incorrect password. Connection terminated." << std::endl;
        close(sock);
        return;
    }

    // Receive file size
    bytes_received = recv(sock, &file_size, sizeof(file_size), 0);
```



```
if (bytes_received <= 0) {
    std::cerr << "Error receiving file size" << std::endl;
    return;
}

file_size = ntohl(file_size); // Convert from network byte order to host byte order
std::ofstream file("received_SERVERFile.txt", std::ios::binary);
if (!file.is_open()) {
    std::cerr << "Failed to open file to write" << std::endl;
    return;
}

ssize_t total_received = 0;
while (total_received < file_size) {
    bytes_received = recv(sock, buffer, BUFFER_SIZE, 0);
    if (bytes_received <= 0) {
        std::cerr << "Error receiving file" << std::endl;
        return;
    }
    file.write(buffer, bytes_received);
    total_received += bytes_received;
}

std::cout << "File received from 'SERVER' successfully" << std::endl;
file.close();
}

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[BUFFER_SIZE] = {0};
}
```

```

101 if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
102     std::cerr << "Socket creation error" << std::endl;
103     return -1;
104 }
105
106 serv_addr.sin_family = AF_INET;
107 serv_addr.sin_port = htons(PORT);
108
109 if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
110     std::cerr << "Invalid address or Address not supported" << std::endl;
111     close(sock);
112     return -1;
113 }
114
115 if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
116     std::cerr << "Connection failed" << std::endl;
117     close(sock);
118     return -1;
119 }
120
121 std::string file_to_send = "pass_CLIENTfile.txt"; // Replace with actual file name
122 sendFile(sock, file_to_send);
123
124 receiveFile(sock);
125
126 close(sock);
127 return 0;
128 }
129

```

3. Task - 3 : Implement code on message passing from client to server.

The image shows a code editor window titled "server_message.cpp" with a dark theme. The code is a C++ program that demonstrates message queue operations. It includes headers for `<iostream>`, `<queue.h>`, `<cstring>`, `<cstdlib>`, `<cerrno>`, and `<stdio.h>`. It defines a queue name "test_queue", a maximum size of 1024, and a message stop string "exit". The `main` function creates a message queue with attributes `mq_attr` and a buffer of size `MAX_SIZE`. It initializes the queue attributes, creates the queue using `mq_open`, and enters a loop where it prompts the user to enter a message. The message is sent using `mq_send` and the queue is closed using `mq_close` when the user enters "exit".

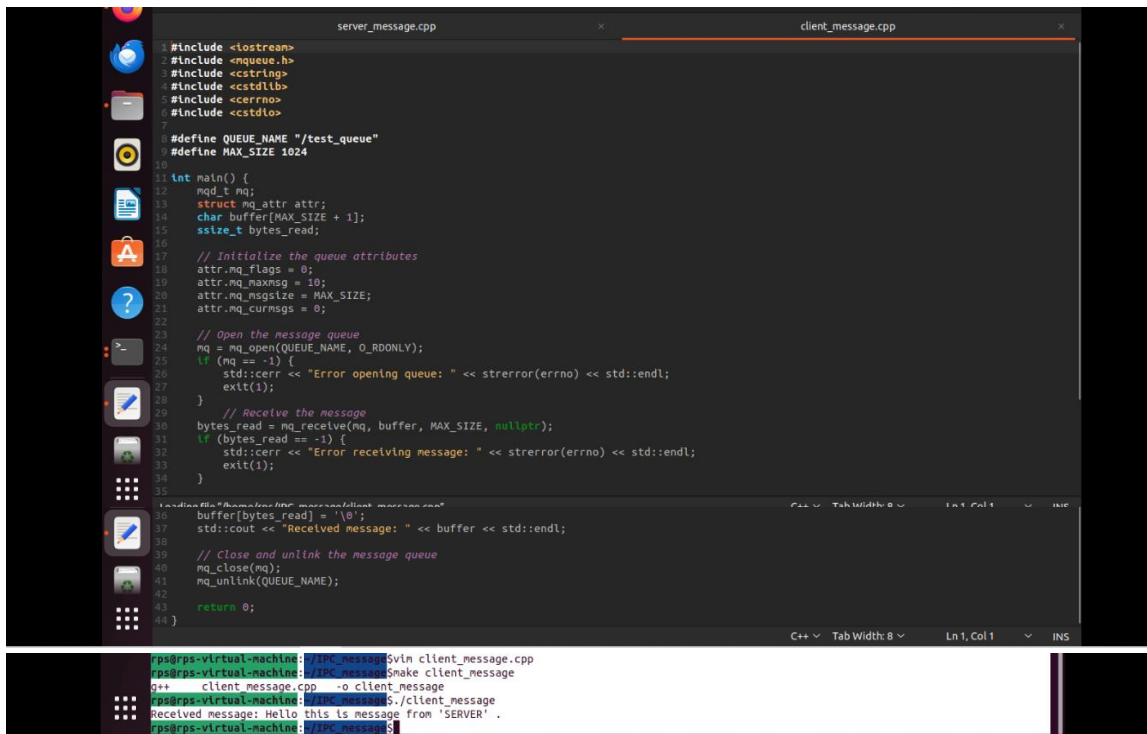
```
1#include <iostream>
2#include <queue.h>
3#include <cstring>
4#include <cstdlib>
5#include <cerrno>
6#include <stdio.h>
7
8#define QUEUE_NAME "/test_queue"
9#define MAX_SIZE 1024
10#define MSG_STOP "exit"
11
12int main() {
13    mqd_t mq;
14    struct mq_attr attr;
15    char buffer[MAX_SIZE];
16
17    // Initialize the queue attributes
18    attr.mq_flags = 0;
19    attr.mq_maxmsg = 10;
20    attr.mq_msgsize = MAX_SIZE;
21    attr.mq_curmsgs = 0;
22
23    // Create the message queue
24    mq = mq_open(QUEUE_NAME, O_CREAT | O_WRONLY, 0644, &attr);
25    if (mq == -1) {
26        std::cerr << "Error creating queue: " << strerror(errno) << std::endl;
27        exit(1);
28    }
29
30    std::cout << "Enter a message: ";
31    std::cin.getline(buffer, MAX_SIZE);
32
33    // Send the message
34    if (mq_send(mq, buffer, strlen(buffer) + 1, 0) == -1) {
35        std::cerr << "Error sending message: " << strerror(errno) << std::endl;
36        exit(1);
37    }
38
39    std::cout << "Message sent: " << buffer << std::endl;
40
41    // Close the message queue
42    mq_close(mq);
43
44    return 0;
45}
```

```

root@f0d81c1c-104b-4000-8000-000000000000:~/C++/Server-Message-IPC# g++ -std=c++11 -pthread -o server_message
root@f0d81c1c-104b-4000-8000-000000000000:~/C++/Server-Message-IPC# ./server_message
Enter a message: Hello this is message from 'SERVER' .
Message sent: Hello this is message from 'SERVER' .
root@f0d81c1c-104b-4000-8000-000000000000:~/C++/Server-Message-IPC#

```


c. client_message



The screenshot shows a code editor with two tabs: `server_message.cpp` and `client_message.cpp`. The `client_message.cpp` tab is active, displaying the following C++ code:

```
#include <iostream>
#include <mqueue.h>
#include <string>
#include <stdlib.h>
#include <cerrno>
#include <stdio.h>

#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_curnsgs = 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);
    }

    // Print the received message
    std::cout << "Received message: " << buffer << std::endl;

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

    return 0;
}
```

Below the code editor, a terminal window shows the execution of the program:

```
rp@rp-virtual-machine: ~/client_message$ g++ client_message.cpp
rp@rp-virtual-machine: ~/client_message$ ./client_message
Received message: Hello this is message from 'SERVER'.
```

2. Concept on POSIX

4. Task – 4 : The functionality and robustness of POSIX message queue-based inter-process communication program in C++ .

a. server – code



The screenshot shows a code editor with two tabs: `server_process.cpp` and `client_process.cpp`. The `server_process.cpp` tab is active, displaying the following C++ code:

```
#include <iostream>
#include <mqueue.h>
#include <string>
#include <stdlib.h>
#include <cerrno>
#include <stdio.h>
#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_curnsgs = 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);
        }
    }
}
```

```
37 if (strcmp(buffer, MSG_STOP) == 0) break;
38
39 // Receive a message
40 ssize_t bytes_read = mq_receive(mq2, buffer, MAX_SIZE, nullptr);
41 if (bytes_read == -1) {
42     std::cerr << "Error receiving message: " << strerror(errno) << std::endl;
43     exit(1);
44 }
45 buffer[bytes_read] = '\0';
46 std::cout << "Process1 received: " << buffer << std::endl;
47
48 }
49 // Close and unlink the message queues
50 mq_close(mq1);
51 mq_close(mq2);
52 mq_unlink(Queue_NAME1);
53 mq_unlink(Queue_NAME2);
54
55 return 0;
56 }
```

ps@rps-virtual-machine: ~/IPC_message\$./server_loop
Error opening queue: Bad address
ps@rps-virtual-machine: ~/IPC_message\$ vim server_loop.cpp
ps@rps-virtual-machine: ~/IPC_message\$ vim server_process.cpp
ps@rps-virtual-machine: ~/IPC_message\$ make server_process
g++ server_process.cpp -o server_process
ps@rps-virtual-machine: ~/IPC_message\$./server_process
Process1, enter a message: Hello How are you ?
Process1 received: I am fine.
Process1, enter a message: That's nice to hear.
Process1 received: and what about you ?
Process1, enter a message: I am fine too.
^Z
[20]+ Stopped ./server_process
ps@rps-virtual-machine: ~/IPC_message\$

b. client – side

```
client_process.cpp
#include <iostream>
#include <queue.h>
#include <string.h>
#include <unistd.h>
#include <cerrno>
#include <stdio.h>

// Assuming MAX_SIZE and Queue_NAME1, Queue_NAME2 are defined in "common.h"
#include "common.h"

int main() {
    mqd_t mq1, mq2;
    struct mq_attr attr;
    char buffer[MAX_SIZE + 1]; // Increase buffer size by 1 for null termination

    // Initialize the queue attributes
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = MAX_SIZE;
    attr.mq_curnsgs = 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 == (mqd_t)-1 || mq2 == (mqd_t)-1) { // Cast -1 to mqd_t for comparison
        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);
        }

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

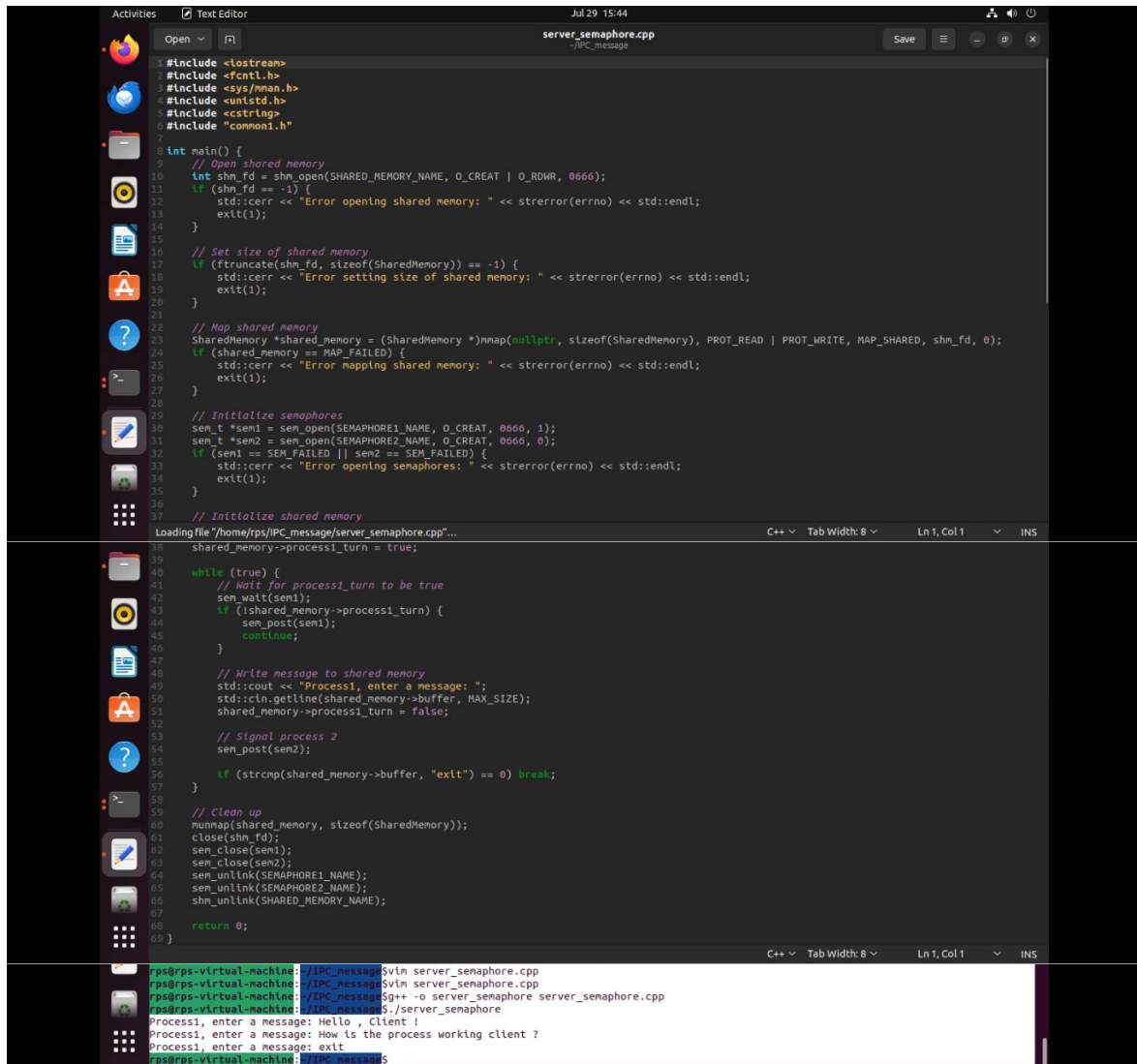
    // Close and unlink the message queues
    mq_close(mq1);
    mq_close(mq2);
    mq_unlink(Queue_NAME1);
    mq_unlink(Queue_NAME2);

    return 0;
}
```

ps@rps-virtual-machine: ~/IPC_message\$ make client_process
ps@rps-virtual-machine: ~/IPC_message\$ vim client_process.cpp
ps@rps-virtual-machine: ~/IPC_message\$ make client_process
g++ client_process.cpp -o client_process
client_process.cpp:10: fatal error: common.h: No such file or directory
9 | #include "common.h"
| ~~~~~
| compilation terminated.
make: *** [**builtin:** client_process] Error 1
ps@rps-virtual-machine: ~/IPC_message\$ vim common.h
ps@rps-virtual-machine: ~/IPC_message\$ make client_process
g++ client_process.cpp -o client_process
ps@rps-virtual-machine: ~/IPC_message\$./client_process
Process2 received: Hello How are you ?
Process2, enter a message: I am fine.
Process2 received: That's nice to hear.
Process2, enter a message: and what about you ?
Process2 received: I am fine too.
Process2, enter a message: ^Z
[6]+ Stopped ./client_process
ps@rps-virtual-machine: ~/IPC_message\$

5. Task – 5 Use of semaphore for message passing.

a. Server –side



```
#include <iostream>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <string>
#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 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);

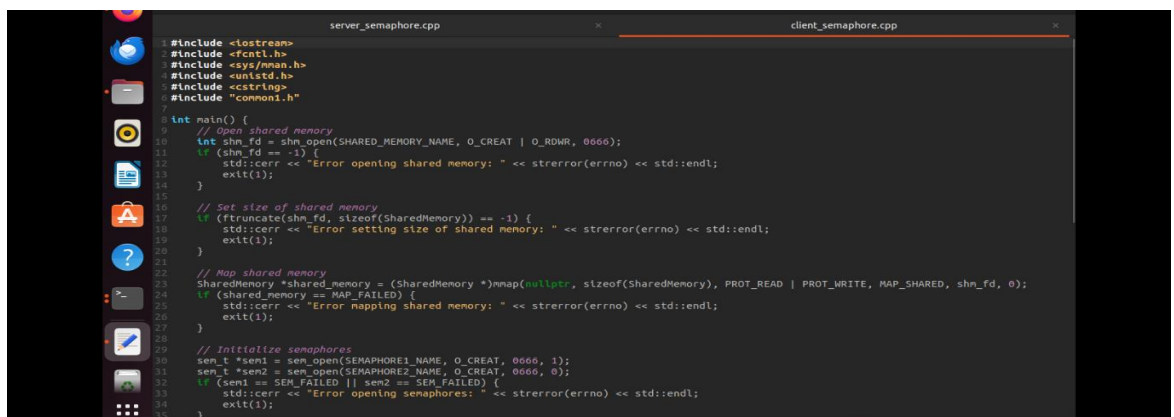
        if (strcmp(shared_memory->buffer, "exit") == 0) break;
    }

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

    return 0;
}
```

Process1, enter a message: Hello, Client!
Process1, enter a message: How is the process working client?
Process1, enter a message: exit

b. client – side



```
#include <iostream>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <string>
#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);
    }

    // Wait for process2_turn to be true
    sem_wait(sem2);

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

    // Signal process1
    sem_post(sem1);

    return 0;
}
```

The top part of the image shows a C++ IDE with two files: `server_semaphore.cpp` and `client_semaphore.cpp`. The `server_semaphore.cpp` file contains the following code:

```
37 while (true) {
38     // wait for process1_turn to be false
39     sem_wait(sem2);
40     if (shared_memory->process1_turn) {
41         sem_post(sem2);
42         continue;
43     }
44     // Read message from shared memory
45     std::cout << "Process2 received: " << shared_memory->buffer << std::endl;
46     if (strcmp(shared_memory->buffer, "exit") == 0) break;
47     // Write response to shared memory
48     std::cout << "Process2, enter a message: ";
49     std::cin.getline(shared_memory->buffer, MAX_SIZE);
50     shared_memory->process1_turn = true;
51     // Signal process 1
52     sem_post(sem1);
53     if (strcmp(shared_memory->buffer, "exit") == 0) break;
54 }
55 // Clean up
56 munmap(shared_memory, sizeof(SharedMemory));
57 close(shm_fd);
58 sem_close(sem1);
59 sem_close(sem2);
60 sem_unlink(SEMAPHORE1_NAME);
61 sem_unlink(SEMAPHORE2_NAME);
62 shm_unlink(SHARED_MEMORY_NAME);
63 return 0;
64 }
```

The bottom part of the image shows the terminal output of the program:

```
Process2 received: Hello , client !
Process2, enter a message: Hello , Server !
Process2 received: How is the process working client ?
Process2, enter a message: It's working fine.
Process2 received: exit
```

2. Concept on Pipe

a. Send – pipe

The image shows a C++ IDE with a file named `pipe_send.cpp`. The code is as follows:

```
#include <iostream>
#include <unistd.h>
#include <cstring>
#include <fcntl.h>
#include <sys/stat.h>
#include <cerrno>

#define PIPE1 "/tmp/pipe1"
#define PIPE2 "/tmp/pipe2"
#define MAX_SIZE 1024

void create_pipe(const char* pipe_name) {
    if (mkfifo(pipe_name, 0666) == -1) {
        if (errno != EEXIST) {
            std::cerr << "Error creating pipe " << pipe_name << ": " << strerror(errno) << std::endl;
            exit(1);
        }
    }
}

int main() {
    // create the pipes
    create_pipe(PIPE1);
    create_pipe(PIPE2);

    int pipe1_fd, pipe2_fd;
    char buffer[MAX_SIZE];

    while (true) {
        // Write message to PIPE1
        std::cout << "Process1, enter a message: ";
        std::cin.getline(buffer, MAX_SIZE);
        pipe1_fd = open(PIPE1, O_WRONLY);
        if (pipe1_fd == -1) {
            std::cerr << "Error opening pipe1 for writing: " << strerror(errno) << std::endl;
            exit(1);
        }
    }
}
```

```
36 read(pipe1_fd, buffer, MAX_SIZE);
37 std::cout << "Process2 received: " << buffer << std::endl;
38 close(pipe1_fd);
39
40 if (strcmp(buffer, "exit") == 0) break;
41
42 // Write message to PIPE2
43 std::cout << "Process2, enter a message: ";
44 std::cin.getline(buffer, MAX_SIZE);
45 pipe2_fd = open(PIPE2, O_WRONLY);
46 if (pipe2_fd == -1) {
47     std::cerr << "Error opening pipe2 for writing: " << strerror(errno) << std::endl;
48     exit(1);
49 }
50 write(pipe2_fd, buffer, strlen(buffer) + 1);
51 close(pipe2_fd);
52 }
53
54 std::cout << "Process2 exiting..." << std::endl;
55 return 0;
56 }
57
```

```
Process1, enter a message: exit
rps@rps-virtual-machine: ~/IPC_message$ ./pipe_send.cpp
rps@rps-virtual-machine: ~/IPC_message$ ./pipe_send.cpp
rps@rps-virtual-machine: ~/IPC_message$ ./pipe_send.cpp
rps@rps-virtual-machine: ~/IPC_message$ ./pipe_send.cpp
rps@rps-virtual-machine: ~/IPC_message$ ./pipe_send.cpp
Process1, enter a message: Hello Client !
Process1 received: Hi, Server.
Process1, enter a message: exit
rps@rps-virtual-machine: ~/IPC_message$
```

b. rcv - pipe

```
Activities Text Editor Jul 29 17:01
Open pipe_send.cpp pipe_rcv.cpp
#include <iostream>
#include <unistd.h>
#include <cstring>
#include <fcntl.h>
#include <sys/stat.h>
#include <cerrno>

#define PIPE1 "/tmp/pipe1"
#define PIPE2 "/tmp/pipe2"
#define MAX_SIZE 1024

void create_pipe(const char* pipe_name) {
    if (mkfifo(pipe_name, 0666) == -1) {
        if (errno != EEXIST) {
            std::cerr << "Error creating pipe " << pipe_name << ": " << strerror(errno) << std::endl;
            exit(1);
        }
    }
}

int main() {
    // Create the pipes
    create_pipe(PIPE1);
    create_pipe(PIPE2);

    int pipe1_fd, pipe2_fd;
    char buffer[MAX_SIZE];

    while (true) {
        // Read message from PIPE1
        pipe1_fd = open(PIPE1, O_RDONLY);
        if (pipe1_fd == -1) {
            std::cerr << "Error opening pipe1 for reading: " << strerror(errno) << std::endl;
            exit(1);
        }

        read(pipe1_fd, buffer, MAX_SIZE);
        std::cout << "Process1 received: " << buffer << std::endl;
        close(pipe1_fd);

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

    return 0;
}

Loading file "/home/rps/IPC_message/pipe_rcv.cpp"...
C++ Tab Width: 8 Ln 1, Col 1 INS

36 write(pipe1_fd, buffer, strlen(buffer) + 1);
37 close(pipe1_fd);
38
39 if (strcmp(buffer, "exit") == 0) break;
40
41 // Read message from PIPE2
42 pipe2_fd = open(PIPE2, O_RDONLY);
43 if (pipe2_fd == -1) {
44     std::cerr << "Error opening pipe2 for reading: " << strerror(errno) << std::endl;
45     exit(1);
46 }
47
48 read(pipe2_fd, buffer, MAX_SIZE);
49 std::cout << "Process2 received: " << buffer << std::endl;
50 close(pipe2_fd);
51
52 if (strcmp(buffer, "exit") == 0) break;
53 }
54
55 return 0;
56 }
57
```

```
Terminal
rps@rps-virtual-machine: ~/IPC_message$ ./pipe_rcv.cpp
rps@rps-virtual-machine: ~/IPC_message$ ./pipe_rcv.cpp
rps@rps-virtual-machine: ~/IPC_message$ ./pipe_rcv.cpp
rps@rps-virtual-machine: ~/IPC_message$ ./pipe_rcv.cpp
rps@rps-virtual-machine: ~/IPC_message$ ./pipe_rcv.cpp
Process2 received: Hello Client !
Process2, enter a message: Hi, Server.
Process2 received: exit
Process2 exiting...
rps@rps-virtual-machine: ~/IPC_message$
```

6. Task – 6 : Use of fork

The screenshot shows a C++ IDE with three tabs: `pipe_send.cpp`, `pipe_rcv.cpp`, and `fork.cpp`. The `fork.cpp` tab is active, displaying the following code:

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

using namespace std;

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

    if (pld < 0) {
        cerr << "Fork failed" << endl;
        return 1;
    } else if (pld == 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;
}
```

Below the code, the terminal output is shown, listing system resources and process details:

```
total 276
-rwxrwxr-x 1 rps rps 16976 Jul 29 14:47 client_loop
-rw-rw-r-- 1 rps rps 1367 Jul 29 14:48 client_loop.cpp
-rwxrwxr-x 1 rps rps 16936 Jul 29 14:13 client_message
-rw-rw-r-- 1 rps rps 1038 Jul 29 14:10 client_message.cpp
-rwxrwxr-x 1 rps rps 17160 Jul 29 15:10 client_process
-rw-rw-r-- 1 rps rps 1809 Jul 29 15:06 client_process.cpp
-rwxrwxr-x 1 rps rps 17336 Jul 29 15:40 client_semaphore
-rw-rw-r-- 1 rps rps 2148 Jul 29 15:39 client_semaphore.cpp
-rw-rw-r-- 1 rps rps 298 Jul 29 15:38 common.h
-rw-rw-r-- 1 rps rps 172 Jul 29 15:10 common.h
-rwxrwxr-x 1 rps rps 16096 Jul 29 17:09 fork
-rw-rw-r-- 1 rps rps 637 Jul 29 17:09 fork.cpp
-rwxrwxr-x 1 rps rps 17184 Jul 29 16:58 pipe_rcv
-rw-rw-r-- 1 rps rps 1511 Jul 29 16:58 pipe_rcv.cpp
-rwxrwxr-x 1 rps rps 17184 Jul 29 16:58 pipe_send
-rw-rw-r-- 1 rps rps 1506 Jul 29 16:52 pipe_send.cpp
-rw-rw-r-- 1 rps rps 1367 Jul 29 14:47 q
-rwxrwxr-x 1 rps rps 17064 Jul 29 14:54 server_loop
-rw-rw-r-- 1 rps rps 1066 Jul 29 14:57 server_loop.cpp
-rwxrwxr-x 1 rps rps 17024 Jul 29 14:13 server_message
-rw-rw-r-- 1 rps rps 1030 Jul 29 14:13 server_message.cpp
-rwxrwxr-x 1 rps rps 17160 Jul 29 15:11 server_process
-rw-rw-r-- 1 rps rps 1554 Jul 29 15:00 server_process.cpp
-rwxrwxr-x 1 rps rps 17336 Jul 29 15:40 server_semaphore
-rw-rw-r-- 1 rps rps 2032 Jul 29 15:39 server_semaphore.cpp
child process completed
rps@rps-virtual-machine:~/IPC_message$ man fork
rps@rps-virtual-machine:~/IPC_message$ man fork
```

7. Task – 7 : Use of exec

The screenshot shows a C++ IDE with five tabs: `pipe_send.cpp`, `pipe_rcv.cpp`, `fork.cpp`, and `exec.cpp`. The `exec.cpp` tab is active, displaying the following code:

```
#include <iostream>
#include <unistd.h>
#include <cstring>
#include <cerrno>

int main() {
    const char *args[] = {"/bin/ls", "-l", nullptr}; // Command and arguments

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

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

```
rps@rps-virtual-machine:~/IPC_message$ ls exec.cpp
rps@rps-virtual-machine:~/IPC_message$ g++ exec.cpp
rps@rps-virtual-machine:~/IPC_message$ g++ -o exec exec.cpp
rps@rps-virtual-machine:~/IPC_message$ ./exec
total 300
-rwxrwxr-x 1 rps rps 16976 Jul 29 14:47 client_loop
-rw-rw-r-- 1 rps rps 1367 Jul 29 14:48 client_loop.cpp
-rwxrwxr-x 1 rps rps 16936 Jul 29 14:13 client_message
-rw-rw-r-- 1 rps rps 1838 Jul 29 14:10 client_message.cpp
-rwxrwxr-x 1 rps rps 17168 Jul 29 15:10 client_process
-rw-rw-r-- 1 rps rps 1809 Jul 29 15:06 client_process.cpp
-rwxrwxr-x 1 rps rps 17336 Jul 29 15:40 client_semaphore
-rw-rw-r-- 1 rps rps 2148 Jul 29 15:39 client_semaphore.cpp
-rw-rw-r-- 1 rps rps 298 Jul 29 15:38 common.h
-rw-rw-r-- 1 rps rps 172 Jul 29 15:10 common.h
-rwxrwxr-x 1 rps rps 16720 Jul 29 17:15 exec
-rw-rw-r-- 1 rps rps 542 Jul 29 17:15 exec.cpp
-rwxrwxr-x 1 rps rps 16696 Jul 29 17:09 fork
-rw-rw-r-- 1 rps rps 637 Jul 29 17:11 fork.cpp
-rwxrwxr-x 1 rps rps 17184 Jul 29 16:58 pipe_rcv
-rw-rw-r-- 1 rps rps 1511 Jul 29 16:58 pipe_rcv.cpp
-rwxrwxr-x 1 rps rps 17184 Jul 29 16:58 pipe_send
-rw-rw-r-- 1 rps rps 1506 Jul 29 16:52 pipe_send.cpp
-rw-rw-r-- 1 rps rps 1367 Jul 29 14:47 q
-rwxrwxr-x 1 rps rps 17864 Jul 29 14:54 server_loop
-rw-rw-r-- 1 rps rps 1866 Jul 29 14:57 server_loop.cpp
-rwxrwxr-x 1 rps rps 17024 Jul 29 14:13 server_message
-rw-rw-r-- 1 rps rps 1030 Jul 29 14:13 server_message.cpp
-rwxrwxr-x 1 rps rps 17168 Jul 29 15:11 server_process
-rw-rw-r-- 1 rps rps 1554 Jul 29 15:00 server_process.cpp
-rwxrwxr-x 1 rps rps 17336 Jul 29 15:40 server_semaphore
-rw-rw-r-- 1 rps rps 2032 Jul 29 15:39 server_semaphore.cpp
rps@rps-virtual-machine:~/IPC_message$ man exec
rps@rps-virtual-machine:~/IPC_message$
```