Day -11 LSP : Assignment

1. **Problem Statement: Inter-Process Communication (IPC) using Pipes, Shared Memory, and Message Queues.**

   Design and implement efficient and reliable inter-process communication (IPC) mechanisms using pipes, shared memory, and message queues in C++ to facilitate data exchange and synchronization between multiple processes within a single system.

   Specific Requirements:

   a. **Pipe:** Create and manage unidirectional and bidirectional pipes for simple data transfer between related processes.
   b. **Shared Memory:** Allocate and manage shared memory segments for efficient data sharing between multiple processes.
   c. **Message Queues:** Create and utilize message queues for asynchronous communication and data exchange with message prioritization.
   d. **Synchronization:** Implement appropriate synchronization mechanisms (e.g., semaphores, mutexes) to coordinate access to shared resources and prevent race conditions.
   e. **Error Handling**: Incorporate robust error handling to manage potential IPC failures and resource leaks.

   1. A. Pipe

   a. pipe_send

```
        read(pipe1_fd, buffer, MAX_SIZE);
        std::cout << "Process2 received: " << buffer << std::endl;
        close(pipe1_fd);

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

        // Write message to PIPE2
        std::cout << "Process2, enter a message: ";
        std::cin.getline(buffer, MAX_SIZE);
        pipe2_fd = open(PIPE2, O_WRONLY);
        if (pipe2_fd == -1) {
            std::cerr << "Error opening pipe2 for writing: " << strerror(errno) << std::endl;
            exit(1);
        }
        write(pipe2_fd, buffer, strlen(buffer) + 1);
        close(pipe2_fd);
    }

    std::cout << "Process2 exiting..." << std::endl;
    return 0;
}
```

C++ Tab Width: 8 Ln 1, Col 1 INS

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

b. recv _pipe

Open pipe_rcv.cpp ~/IPC_message Save

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);
        }
```

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

```
        write(pipe1_fd, buffer, strlen(buffer) + 1);
        close(pipe1_fd);

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

        // Read message from PIPE2
        pipe2_fd = open(PIPE2, O_RDONLY);
        if (pipe2_fd == -1) {
            std::cerr << "Error opening pipe2 for reading: " << strerror(errno) << std::endl;
            exit(1);
        }
        read(pipe2_fd, buffer, MAX_SIZE);
        std::cout << "Process1 received: " << buffer << std::endl;
        close(pipe2_fd);

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

    return 0;
}
```

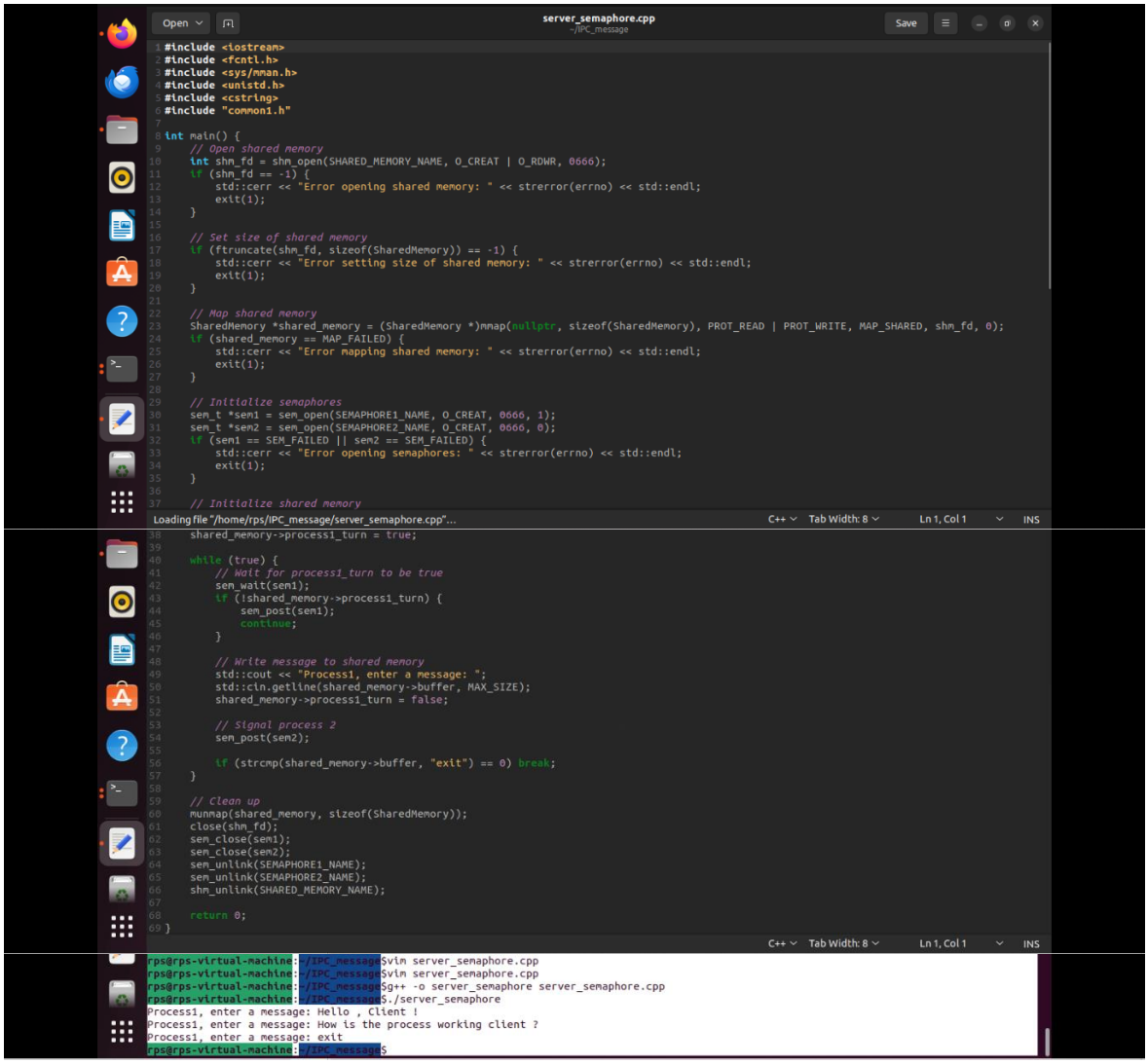
C++ Tab Width: 8 Ln 1, Col 1 INS

Terminal | Terminal

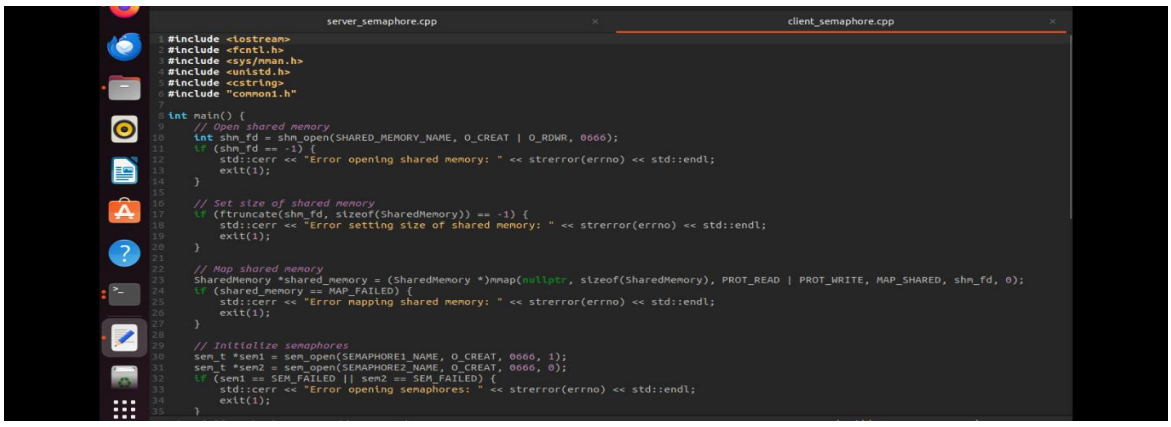
```
rps@rps-virtual-machine:~/IPC_message$vim pipe_rcv.cpp
rps@rps-virtual-machine:~/IPC_message$make pipe_rcv
g++     pipe_rcv.cpp   -o pipe_rcv
rps@rps-virtual-machine:~/IPC_message$./pipe_rcv
Process2 received: Hello Client !
Process2, enter a message: Hi, Server.
Process2 received: exit
Process2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

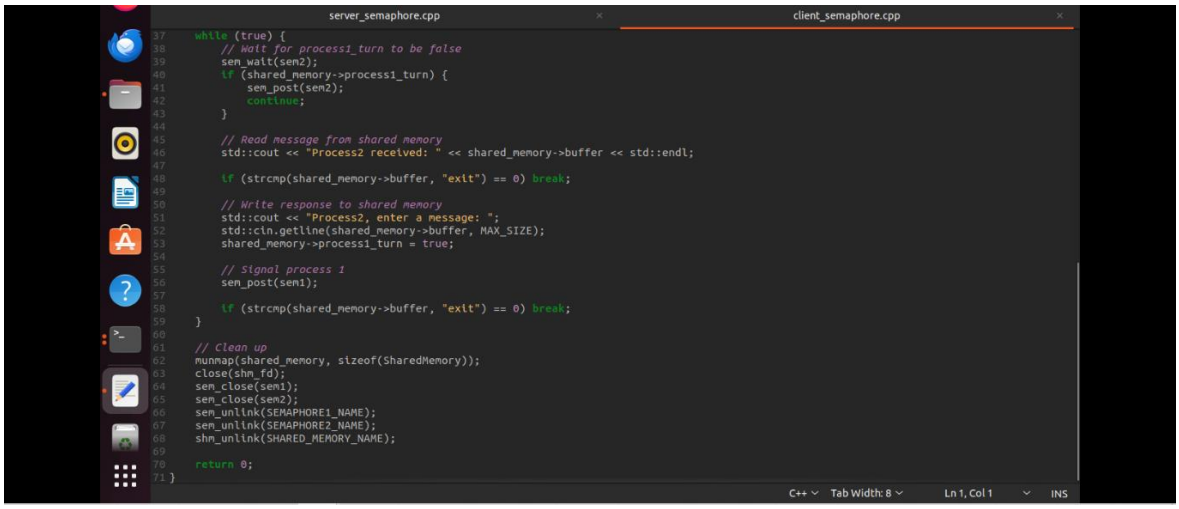
2. A. Shared Memory

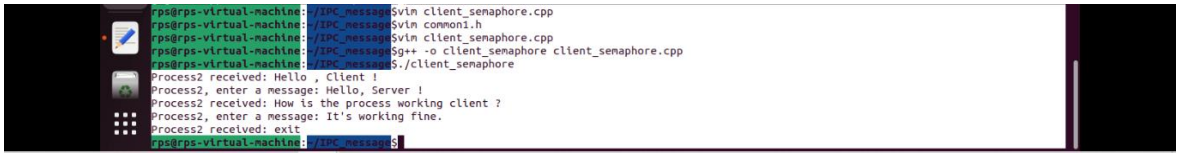
a. Server – side



b. client – side

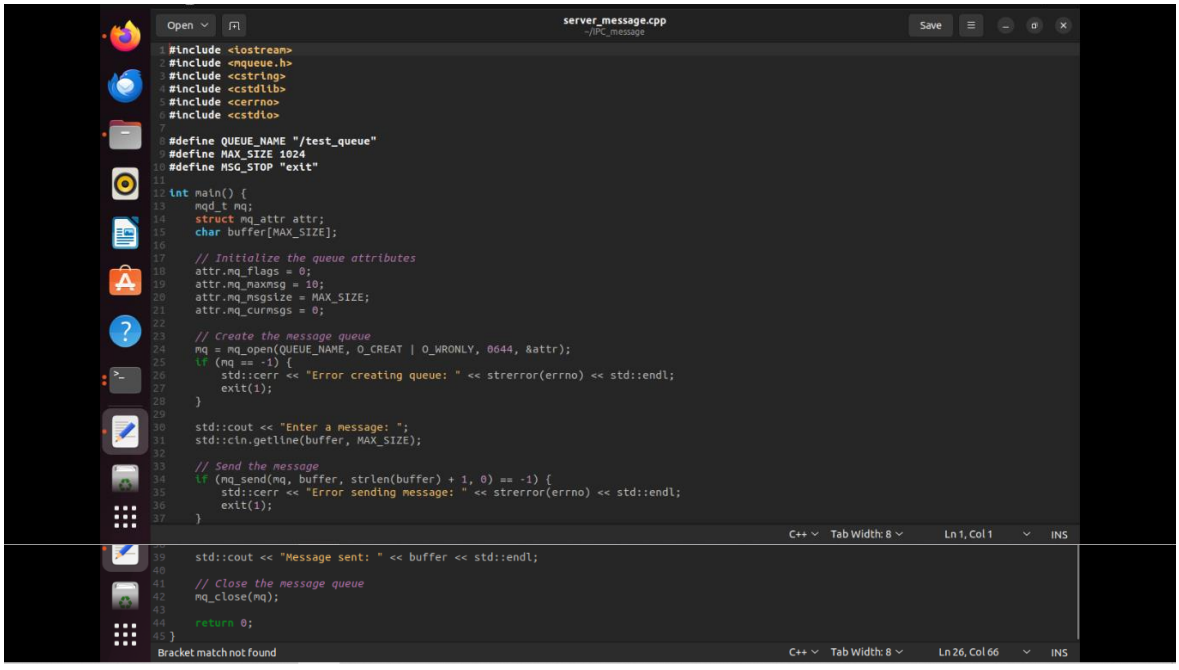




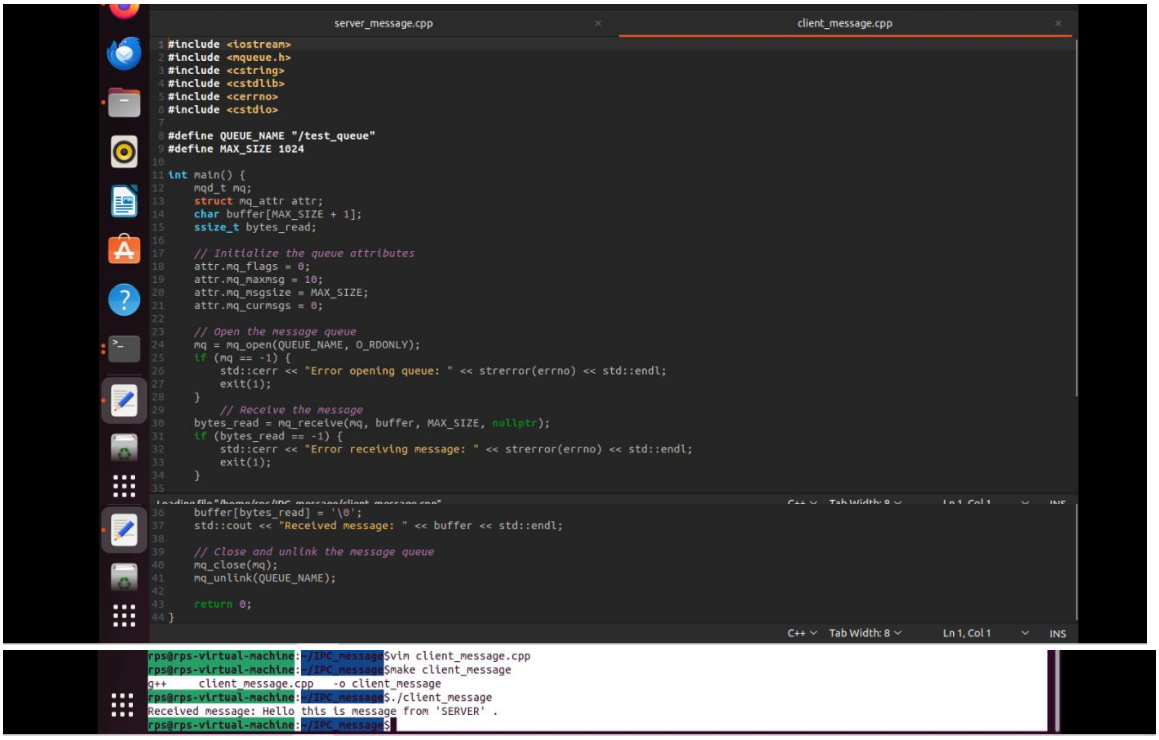


3. A. Message queue

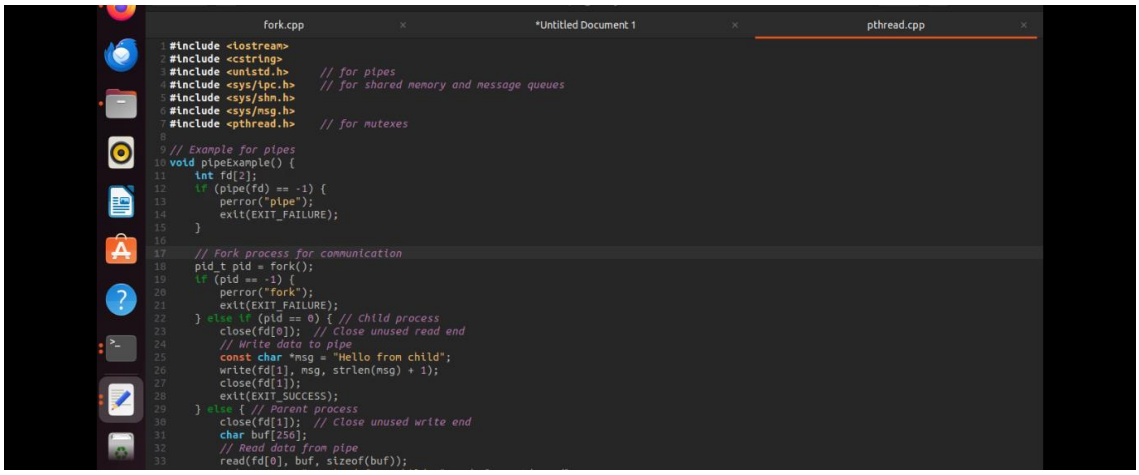
a. server – message

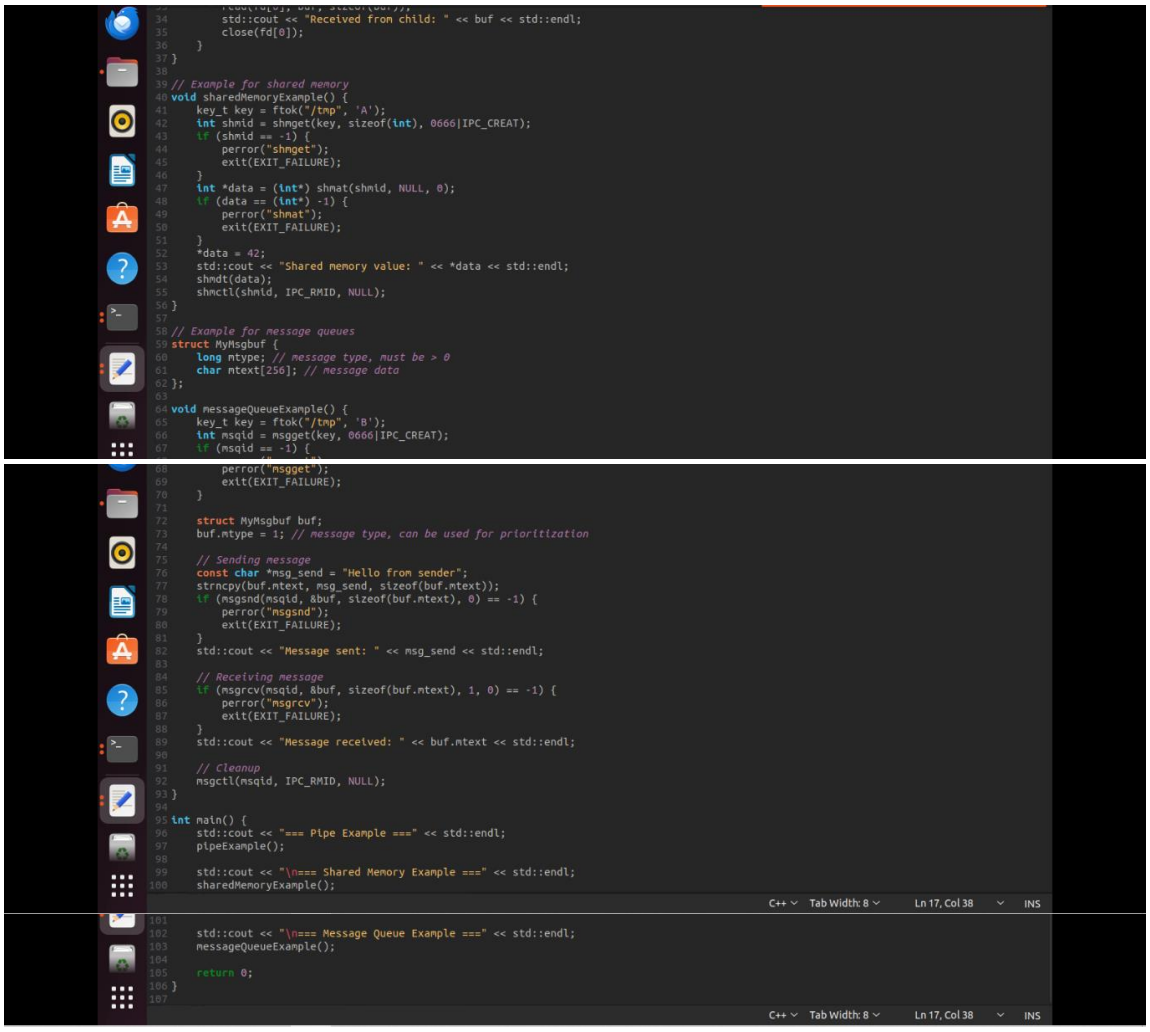


b. client message



The entire assignment code can be coded as follows,





```
        std::cout << "Received from child: " << buf << std::endl;
        close(fd[0]);
    }
}

// Example for shared memory
void sharedMemoryExample() {
    key_t key = ftok("/tmp", 'A');
    int shmid = shmget(key, sizeof(int), 0666|IPC_CREAT);
    if (shmid == -1) {
        perror("shmget");
        exit(EXIT_FAILURE);
    }
    int *data = (int*) shmat(shmid, NULL, 0);
    if (data == (int*) -1) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }
    *data = 42;
    std::cout << "Shared memory value: " << *data << std::endl;
    shmdt(data);
    shmctl(shmid, IPC_RMID, NULL);
}

// Example for message queues
struct MyMsgbuf {
    long mtype; // message type, must be > 0
    char mtext[256]; // message data
};

void messageQueueExample() {
    key_t key = ftok("/tmp", 'B');
    int msqid = msgget(key, 0666|IPC_CREAT);
    if (msqid == -1) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }

    struct MyMsgbuf buf;
    buf.mtype = 1; // message type, can be used for prioritization

    // Sending message
    const char *msg_send = "Hello from sender";
    strncpy(buf.mtext, msg_send, sizeof(buf.mtext));
    if (msgsnd(msqid, &buf, sizeof(buf.mtext), 0) == -1) {
        perror("msgsnd");
        exit(EXIT_FAILURE);
    }
    std::cout << "Message sent: " << msg_send << std::endl;

    // Receiving message
    if (msgrcv(msqid, &buf, sizeof(buf.mtext), 1, 0) == -1) {
        perror("msgrcv");
        exit(EXIT_FAILURE);
    }
    std::cout << "Message received: " << buf.mtext << std::endl;

    // Cleanup
    msgctl(msqid, IPC_RMID, NULL);
}

int main() {
    std::cout << "=== Pipe Example ===" << std::endl;
    pipeExample();

    std::cout << "\n=== Shared Memory Example ===" << std::endl;
    sharedMemoryExample();
```

C++ Tab Width: 8 Ln 17, Col 38 INS

```
    std::cout << "\n=== Message Queue Example ===" << std::endl;
    messageQueueExample();

    return 0;
}
```

C++ Tab Width: 8 Ln 17, Col 38 INS

Execution :

```
rps@rps-virtual-machine:~/IPC_message$vim pthread.cpp
rps@rps-virtual-machine:~/IPC_message$make pthread
g++     pthread.cpp   -o pthread
rps@rps-virtual-machine:~/IPC_message$./pthread
=== Pipe Example ===
Received from child: Hello from child

=== Shared Memory Example ===
Shared memory value: 42

=== Message Queue Example ===
Message sent: Hello from sender
Message received: Hello from sender
rps@rps-virtual-machine:~/IPC_message$
```

2. **Task: Create a program that replicates the functionality of the standard cp command, but without using any standard library functions related to file I/O. Instead, you must employ system calls directly to perform file operations.**

Requirements:

System calls: Utilize system calls like open, close, read, and write to interact with files.

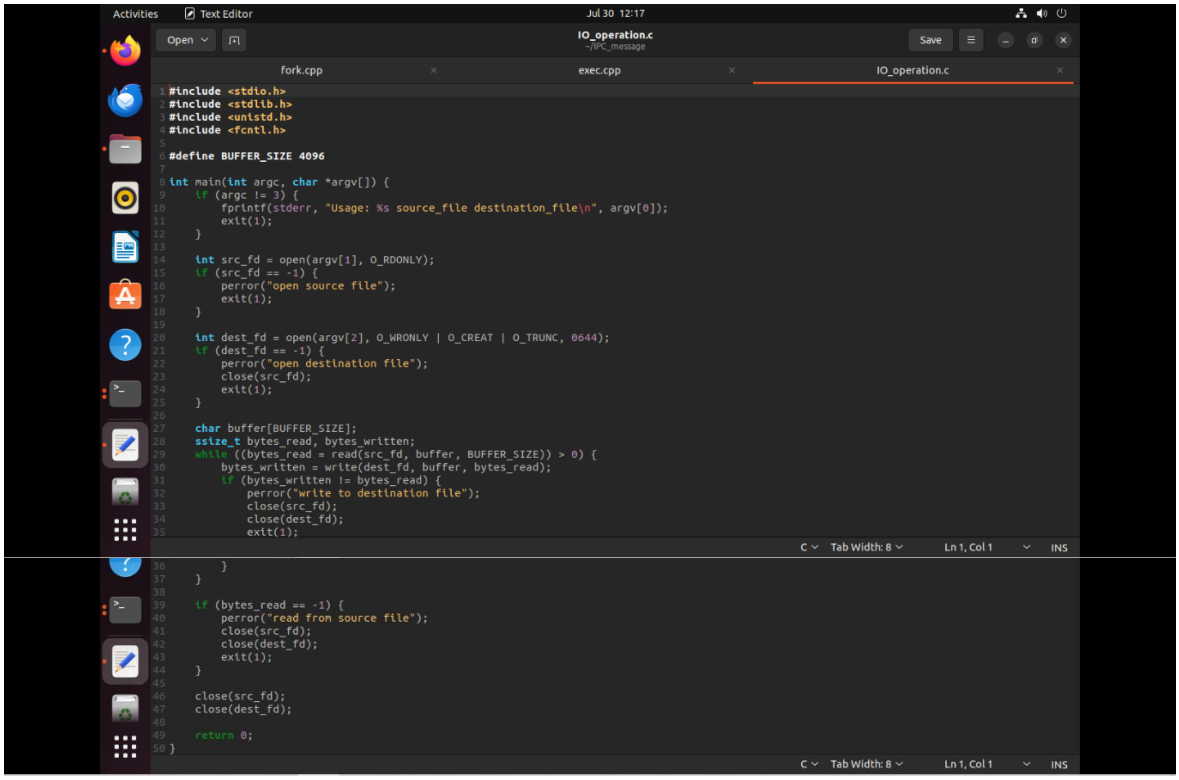
Error handling: Implement robust error handling for potential issues such as file not found, permission denied, disk full, etc.

Efficiency: Optimize the copying process for performance, considering buffer sizes and read/write operations.

Metadata: Preserve file permissions, timestamps, and other relevant metadata during the copy process.

User interface: Provide a simple command-line interface with options for source and destination file paths.

a. code



Execution :