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:

Pipe: Create and manage unidirectional and bidirectional pipes for simple data transfer between related processes.

Shared Memory: Allocate and manage shared memory segments for efficient data sharing between multiple processes.

Message Queues: Create and utilize message queues for asynchronous communication and data exchange with message prioritization.

Synchronization: Implement appropriate synchronization mechanisms (e.g., semaphores, mutexes) to coordinate access to shared resources and prevent race conditions.

Error Handling: Incorporate robust error handling to manage potential IPC failures and resource leaks.

```
rps@rps-virtual-machine:~/pipe$ vim ipc_example.cpp
rps@rps-virtual-machine:~/pipe$ g++ -o ipc_example ipc_example.cpp -lrt -pthread
rps@rps-virtual-machine:~/pipe$ ./ipc_example
Starting IPC Example...

Pipes Communication:
Child read from pipe1: Hello from parent

Shared Memory and Semaphore:
Parent read from pipe2: Hello from child
Parent read from shared memory: Hello from child

Message Queue Communication:
Parent received message: Hello from child
Child sent message.

Shared Memory and Semaphore:
Parent read from shared memory: Hello from child

Message Queue Communication:
Parent received message: Hello from child
Child sent message.
```

```
rps@rps-virtual-machine:~/pipe$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 5          rps        600        524288     2          dest
0x00000000 6          rps        606        5448960    2          dest
0x00000000 7          rps        606        5448960    2          dest
0x00000000 36         rps        600        524288     2          dest
0x00000000 52         rps        600        1048576    2          dest
0x00000000 61         rps        606        4325376    2          dest
0x00000000 62         rps        606        4325376    2          dest

rps@rps-virtual-machine:~/pipe$ ipcs -m shmid

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 5          rps        600        524288     2          dest
0x00000000 6          rps        606        5448960    2          dest
0x00000000 7          rps        606        5448960    2          dest
0x00000000 36         rps        600        524288     2          dest
0x00000000 52         rps        600        1048576    2          dest
0x00000000 61         rps        606        4325376    2          dest
0x00000000 62         rps        606        4325376    2          dest

rps@rps-virtual-machine:~/pipe$ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

CODE :-

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

#define SHM_NAME "/my_shm"
#define SHM_SIZE 4096
#define SEM_NAME "/my_sem"
#define MSG_KEY 1234
#define MSG_SIZE 256

// Message queue structure
struct message {
    long mtype;
    char mtext[MSG_SIZE];
};

void create_and_use_pipes() {
    int pipe1[2], pipe2[2];
    if (pipe(pipe1) == -1 || pipe(pipe2) == -1) {
        perror("pipe");
        exit(1);
    }

    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(1);
    }
```

```
 if (pid < 0) {
     perror("fork");
     exit(1);
 }

 if (pid == 0) {  // Child process
     close(pipe1[1]);
     close(pipe2[0]);

     char buffer[256];
     ssize_t bytesRead = read(pipe1[0], buffer, sizeof(buffer));
     if (bytesRead == -1) {
         perror("read");
         exit(1);
     }
     buffer[bytesRead] = '\0';
     std::cout << "Child read from pipe1: " << buffer << std::endl;

     const char* response = "Hello from child";
     if (write(pipe2[1], response, strlen(response)) == -1) {
         perror("write");
         exit(1);
     }
     close(pipe1[0]);
     close(pipe2[1]);
 } else {  // Parent process
     close(pipe1[0]);
     close(pipe2[1]);

     const char* message = "Hello from parent";
     if (write(pipe1[1], message, strlen(message)) == -1) {
```

```
         if (write(pipe1[1], message, strlen(message)) == -1) {
             perror("write");
             exit(1);
         }

         char buffer[256];
         ssize_t bytesRead = read(pipe2[0], buffer, sizeof(buffer));
         if (bytesRead == -1) {
             perror("read");
             exit(1);
         }
         buffer[bytesRead] = '\0';
         std::cout << "Parent read from pipe2: " << buffer << std::endl;

         close(pipe1[1]);
         close(pipe2[0]);

         wait(nullptr);  // Wait for child process to finish
     }
}

void use_shared_memory_and_semaphore() {
    sem_t* sem = sem_open(SEM_NAME, O_CREAT, 0666, 1);
    if (sem == SEM_FAILED) {
         perror("sem_open");
         exit(1);
    }

    int shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    if (shm_fd == -1) {
         perror("shm_open");
         exit(1);
    }
    if (ftruncate(shm_fd, SHM_SIZE) == -1) {
         perror("ftruncate");
```

```
    if (ftruncate(shm_fd, SHM_SIZE) == -1) {
        perror("ftruncate");
        exit(1);
    }

    void* ptr = mmap(nullptr, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) {  // Child process
        sem_wait(sem);
        const char* message = "Hello from child";
        strncpy(static_cast<char*>(ptr), message, SHM_SIZE);
        sem_post(sem);
        munmap(ptr, SHM_SIZE);
        close(shm_fd);
        exit(0);
    } else {  // Parent process
        sleep(1);  // Allow child to write first
        sem_wait(sem);
        std::cout << "Parent read from shared memory: " << static_cast<char*>(ptr) << std::endl;
        sem_post(sem);

        munmap(ptr, SHM_SIZE);
        close(shm_fd);
        shm_unlink(SHM_NAME);
        sem_close(sem);
```

```
        munmap(ptr, SHM_SIZE);
        close(shm_fd);
        shm_unlink(SHM_NAME);
        sem_close(sem);
        sem_unlink(SEM_NAME);
        wait(nullptr);  // Wait for child process to finish
    }
}

void use_message_queue() {
    int msgid = msgget(MSG_KEY, IPC_CREAT | 0666);
    if (msgid == -1) {
        perror("msgget");
        exit(1);
    }

    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) {  // Child process
        message msg;
        msg.mtype = 1;
        strncpy(msg.mtext, "Hello from child", MSG_SIZE);

        if (msgsnd(msgid, &msg, strlen(msg.mtext) + 1, 0) == -1) {
            perror("msgsnd");
            exit(1);
        }
        std::cout << "Child sent message." << std::endl;
        exit(0);
    } else {  // Parent process
        message msg;
```

```
        }
        std::cout << "Child sent message." << std::endl;
        exit(0);
    } else {  // Parent process
        message msg;
        if (msgrcv(msgid, &msg, MSG_SIZE, 1, 0) == -1) {
            perror("msgrcv");
            exit(1);
        }
        std::cout << "Parent received message: " << msg.mtext << std::endl;

        msgctl(msgid, IPC_RMID, nullptr);  // Remove the message queue
        wait(nullptr);  // Wait for child process to finish
    }
}

int main() {
    std::cout << "Starting IPC Example..." << std::endl;

    // Use pipes for communication
    std::cout << "\nPipes Communication:\n";
    create_and_use_pipes();

    // Use shared memory and semaphore for synchronization
    std::cout << "\nShared Memory and Semaphore:\n";
    use_shared_memory_and_semaphore();

    // Use message queue for communication
    std::cout << "\nMessage Queue Communication:\n";
    use_message_queue();

    return 0;
}
```

CODE FOR FILES :-

```
rps@rps-virtual-machine:~/files/pipe$ vim file2.cpp
rps@rps-virtual-machine:~/files/pipe$ make file2
g++     file2.cpp   -o file2
rps@rps-virtual-machine:~/files/pipe$ ./file2
Usage: ./file2 source_file destination_file
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFFER_SIZE 4096

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s source_file destination_file\n", argv[0]);
        exit(1);
    }

    int src_fd = open(argv[1], O_RDONLY);
    if (src_fd == -1) {
        perror("open source file");
        exit(1);
    }

    int dest_fd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (dest_fd == -1) {
        perror("open destination file");
        close(src_fd);
        exit(1);
    }

    char buffer[BUFFER_SIZE];
    ssize_t bytes_read, bytes_written;
    while ((bytes_read = read(src_fd, buffer, BUFFER_SIZE)) > 0) {
        bytes_written = write(dest_fd, buffer, bytes_read);
        if (bytes_written != bytes_read) {
            perror("write to destination file");
            close(src_fd);
            close(dest_fd);
            exit(1);
```

```c
    ssize_t bytes_read, bytes_written;
    while ((bytes_read = read(src_fd, buffer, BUFFER_SIZE)) > 0) {
        bytes_written = write(dest_fd, buffer, bytes_read);
        if (bytes_written != bytes_read) {
            perror("write to destination file");
            close(src_fd);
            close(dest_fd);
            exit(1);
        }
    }

    if (bytes_read == -1) {
        perror("read from source file");
        close(src_fd);
        close(dest_fd);
        exit(1);
    }

    close(src_fd);
    close(dest_fd);

    return 0;
}
```

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.

```
rps@rps-virtual-machine:~/files/pipe$ vim file1.cpp
rps@rps-virtual-machine:~/files/pipe$ g++ file1.cpp -o file1
rps@rps-virtual-machine:~/files/pipe$ ./file1
Usage: ./file1 <source> <destination>
```

CODE :-

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

#define BUFFER_SIZE 4096

void print_error(const char* msg) {
    std::cerr << msg << ": " << strerror(errno) << std::endl;
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <source> <destination>" << std::endl;
        return 1;
    }

    const char* source = argv[1];
    const char* destination = argv[2];

    // Open the source file
    int src_fd = open(source, O_RDONLY);
    if (src_fd < 0) {
        print_error("Error opening source file");
        return 1;
    }

    // Get the file status
    struct stat src_stat;
    if (fstat(src_fd, &src_stat) < 0) {
        print_error("Error getting source file metadata");
        close(src_fd);
```

```c
    if (fstat(src_fd, &src_stat) < 0) {
        print_error("Error getting source file metadata");
        close(src_fd);
        return 1;
    }

    // Open the destination file
    int dest_fd = open(destination, O_WRONLY | O_CREAT | O_TRUNC, src_stat.st_mode);
    if (dest_fd < 0) {
        print_error("Error opening/creating destination file");
        close(src_fd);
        return 1;
    }

    // Buffer for file data
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read, bytes_written;

    // Read from source and write to destination
    while ((bytes_read = read(src_fd, buffer, BUFFER_SIZE)) > 0) {
        ssize_t total_written = 0;
        while (total_written < bytes_read) {
            bytes_written = write(dest_fd, buffer + total_written, bytes_read - total_written);
            if (bytes_written < 0) {
                print_error("Error writing to destination file");
                close(src_fd);
                close(dest_fd);
                return 1;
            }
            total_written += bytes_written;
        }
    }
```

```c
                return 1;
            }
            total_written += bytes_written;
        }
    }

    if (bytes_read < 0) {
        print_error("Error reading from source file");
    }

    // Close file descriptors
    if (close(src_fd) < 0) {
        print_error("Error closing source file");
    }

    if (close(dest_fd) < 0) {
        print_error("Error closing destination file");
    }

    // Preserve the metadata: Set the destination file's times to match the source
    struct timespec times[2];
    times[0] = src_stat.st_atim; // Access time
    times[1] = src_stat.st_mtim; // Modification time

    if (utimensat(AT_FDCWD, destination, times, 0) < 0) {
        print_error("Error setting file timestamps");
    }

    return 0;
}
```

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

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

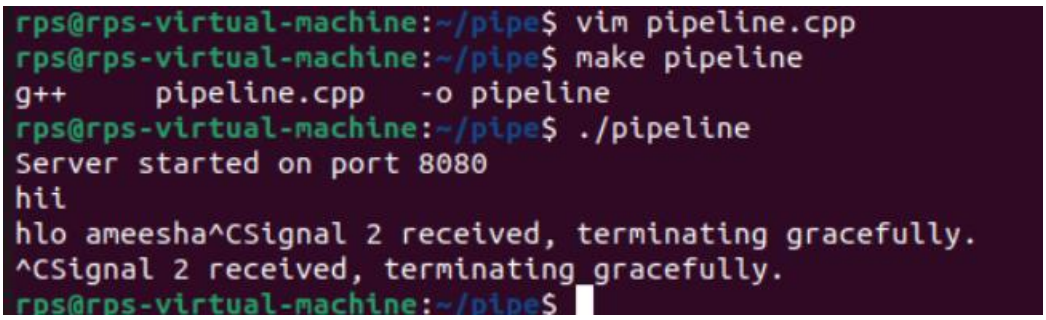
Data Processing: Distribute incoming data to multiple worker processes, each responsible for specific data transformations or calculations.

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

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

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

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



```cpp
#include <iostream>
#include <thread>
#include <vector>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <csignal>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <sys/shm.h>
#include <cstring>

#define PORT 8080
#define SHM_KEY 1234
#define SHM_SIZE 1024
#define NUM_WORKERS 4

// Global variables for IPC
std::queue<std::string> taskQueue;
std::mutex queueMutex;
std::condition_variable queueCondition;
int shmid;
char *shmaddr;

// Signal handler
void signalHandler(int signum) {
    std::cout << "Signal " << signum << " received, terminating gracefully." << std::endl;
    // Cleanup shared memory
    if (shmaddr != nullptr) {
        shmdt(shmaddr);
        shmctl(shmid, IPC_RMID, nullptr);
    }
    exit(signum);
}
```

```cpp
    exit(signum);
}

// Setup signal handling
void setupSignalHandling() {
    signal(SIGINT, signalHandler);
    signal(SIGTERM, signalHandler);
}

// Setup shared memory
void setupSharedMemory() {
    shmid = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT | 0666);
    if (shmid < 0) {
        perror("shmget failed");
        exit(EXIT_FAILURE);
    }
    shmaddr = (char*)shmat(shmid, nullptr, 0);
    if (shmaddr == (char*)-1) {
        perror("shmat failed");
        exit(EXIT_FAILURE);
    }
}

// Server for data ingestion
int setupServerSocket() {
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == 0) {
        perror("Socket creation error");
        exit(EXIT_FAILURE);
    }

    struct sockaddr_in address;
    int opt = 1;
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))) {
        perror("Setsockopt error");
```

```
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))) {
        perror("Setsockopt error");
        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);
    }

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

    return server_fd;
}

// Data distribution
void distributeData(const std::string& data) {
    std::lock_guard<std::mutex> lock(queueMutex);
    taskQueue.push(data);
    queueCondition.notify_one();
}

// Worker function
void workerFunction() {
    while (true) {
        std::unique_lock<std::mutex> lock(queueMutex);
        queueCondition.wait(lock, []{ return !taskQueue.empty(); });
```

```
void workerFunction() {
    while (true) {
        std::unique_lock<std::mutex> lock(queueMutex);
        queueCondition.wait(lock, []{ return !taskQueue.empty(); });

        std::string data = taskQueue.front();
        taskQueue.pop();
        lock.unlock();

        // Process data
        std::cout << "Processing data: " << data << std::endl;

        // Example of writing processed data to shared memory
        snprintf(shmaddr, SHM_SIZE, "Processed: %s", data.c_str());
    }
}

// Start worker processes
void startWorkers(int numWorkers) {
    std::vector<std::thread> workers;
    for (int i = 0; i < numWorkers; ++i) {
        workers.emplace_back(workerFunction);
    }
    for (auto& worker : workers) {
        worker.join();
    }
}

// Main function
int main() {
    setupSignalHandling();
    setupSharedMemory();

    int server_fd = setupServerSocket();
```

```cpp
// Main function
int main() {
    setupSignalHandling();
    setupSharedMemory();

    int server_fd = setupServerSocket();
    std::cout << "Server started on port " << PORT << std::endl;

    // Start worker processes
    std::thread workerThread(startWorkers, NUM_WORKERS);
    workerThread.detach();

    // Data ingestion
    while (true) {
        struct sockaddr_in address;
        int addrlen = sizeof(address);
        int new_socket = accept(server_fd, (struct sockaddr* )&address, (socklen_t*)&addrlen);
        if (new_socket < 0) {
            perror("Accept error");
            continue;
        }

        char buffer[1024] = {0};
        int valread = read(new_socket, buffer, 1024);
        if (valread > 0) {
            std::string data(buffer);
            std::cout << "Received data: " << data << std::endl;
            distributeData(data);
        }
        close(new_socket);
    }

    return 0;
}
```

Coding Questions in C++

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.

```
rps@rps-virtual-machine:~/files/pipe$ vim signal_handling.cpp
rps@rps-virtual-machine:~/files/pipe$
rps@rps-virtual-machine:~/files/pipe$ make signal_handling
g++     signal_handling.cpp    -o signal_handling
rps@rps-virtual-machine:~/files/pipe$ ./signal_handling
Program is running. Press Ctrl+C to trigger SIGINT.
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
^CCaught SIGINT signal (2). Terminating gracefully.
rps@rps-virtual-machine:~/files/pipe$
```

```cpp
#include <iostream>
#include <csignal>
#include <cstdlib>
#include <unistd.h> // For sleep function

// Signal handler function for SIGINT
void handleSigint(int signal) {
    std::cout << "Caught SIGINT signal (" << signal << "). Terminating gracefully." << std::endl;
    // Perform cleanup tasks here if needed
    exit(0); // Terminate the program
}

int main() {
    // Register the signal handler for SIGINT
    std::signal(SIGINT, handleSigint);

    std::cout << "Program is running. Press Ctrl+C to trigger SIGINT." << std::endl;

    // Simulate some work in the program
    while (true) {
        std::cout << "Working..." << std::endl;
        sleep(1); // Sleep for 1 second
    }

    return 0;
}
```

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

```
rps@rps-virtual-machine:~/linux30task$ vim multiple_signalhandle.cpp
rps@rps-virtual-machine:~/linux30task$ make multiple_signalhandle
g++     multiple_signalhandle.cpp    -o multiple_signalhandle
rps@rps-virtual-machine:~/linux30task$ ./multiple_signalhandle
Press Ctrl+C to send SIGINT or kill -TERM <pid> to send SIGTERM...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
Working...
^CCaught signal 2 (SIGINT). Terminating gracefully...
rps@rps-virtual-machine:~/linux30task$
```

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

// Signal handler functions
void handle_sigint(int sig) {
    std::cout << "Caught signal " << sig << " (SIGINT). Terminating gracefully..." << std::endl;
    exit(0);
}

void handle_sigterm(int sig) {
    std::cout << "Caught signal " << sig << " (SIGTERM). Terminating gracefully..." << std::endl;
    exit(0);
}

int main() {
    // Register signal handlers
    signal(SIGINT, handle_sigint);
    signal(SIGTERM, handle_sigterm);

    std::cout << "Press Ctrl+C to send SIGINT or kill -TERM <pid> to send SIGTERM..." << std::endl;

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

    return 0;
}
~
~
~
~
~
-- INSERT --
```

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.

SEVER CODE :-

```
rps@rps-virtual-machine:~/linux30task$ g++ echo_server.cpp -o echo_server
rps@rps-virtual-machine:~/linux30task$ ./echo_server
Received: Hello from client
```

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

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[BUFFER_SIZE] = {0};

    // Create socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Attach socket to the port 8080
    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);

    // Bind the socket to the network address and port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
-- INSERT --
```

```
    // Bind the socket to the network address and port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

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

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

    // Read and echo messages
    while (true) {
        int valread = read(new_socket, buffer, BUFFER_SIZE);
        if (valread > 0) {
            std::cout << "Received: " << buffer << std::endl;
            send(new_socket, buffer, valread, 0);
            std::memset(buffer, 0, BUFFER_SIZE); // Clear buffer
        } else {
            break; // Exit loop if no data is read
        }
    }

    // Close the socket
    close(new_socket);
    close(server_fd);
    return 0;
}
-- INSERT --
```

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

CLIENT CODE :-

```
rps@rps-virtual-machine:~/files/pipe$ vim client.cpp
rps@rps-virtual-machine:~/files/pipe$ make client
g++     client.cpp   -o client
rps@rps-virtual-machine:~/files/pipe$ ./client
Message sent from client
Echoed message from server: Hello from client
rps@rps-virtual-machine:~/files/pipe$
```

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

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[BUFFER_SIZE] = {0};
    const char* hello = "Hello from client";

    // Create socket
    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);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, "172.20.0.13", &serv_addr.sin_addr) <= 0) {
        std::cerr << "Invalid address/ Address not supported" << std::endl;
        return -1;
    }

    // Connect to the server
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        std::cerr << "Connection failed" << std::endl;
        return -1;
```

```cpp
    // Create socket
    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);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, "172.20.0.13", &serv_addr.sin_addr) <= 0) {
        std::cerr << "Invalid address/ Address not supported" << std::endl;
        return -1;
    }

    // Connect to the server
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        std::cerr << "Connection failed" << std::endl;
        return -1;
    }

    // Send message to the server
    send(sock, hello, strlen(hello), 0);
    std::cout << "Message sent from client" << std::endl;

    // Read echoed message from the server
    int valread = read(sock, buffer, BUFFER_SIZE);
    std::cout << "Echoed message from server: " << buffer << std::endl;

    // Close the socket
    close(sock);
    return 0;
}
```

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.

```
rps@rps-virtual-machine:~/linux30task$ g++ ipc_pipe.cpp -o ipc_pipe
rps@rps-virtual-machine:~/linux30task$ ./ipc_pipe
Child received: Hello from parent
rps@rps-virtual-machine:~/linux30task$
```

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

int main() {
    int pipefd[2];
    pid_t pid;
    char buffer[100];

    // Create pipe
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    // Fork process
    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // Child process
        close(pipefd[1]); // Close unused write end
        read(pipefd[0], buffer, sizeof(buffer));
        std::cout << "Child received: " << buffer << std::endl;
        close(pipefd[0]);
    } else {
        // Parent process
        close(pipefd[0]); // Close unused read end
        const char* message = "Hello from parent";
        write(pipefd[1], message, strlen(message) + 1);
        close(pipefd[1]);
```

```
        close(pipefd[0]);
    } else {
        // Parent process
        close(pipefd[0]); // Close unused read end
        const char* message = "Hello from parent";
        write(pipefd[1], message, strlen(message) + 1);
        close(pipefd[1]);
        wait(NULL); // Wait for child to finish
    }

    return 0;
}
```

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

```
rps@rps-virtual-machine:~/linux30task$ vim ipc_msg_queue.cpp
rps@rps-virtual-machine:~/linux30task$ g++ ipc_msg_queue.cpp -o ipc_msg_queue
rps@rps-virtual-machine:~/linux30task$ ./ipc_msg_queue
Child received: Hello from parent
rps@rps-virtual-machine:~/linux30task$
```

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

#define MSG_KEY 1234

struct MsgBuf {
    long mtype;
    char mtext[100];
};

int main() {
    pid_t pid;
    int msgid;

    // Create message queue
    msgid = msgget(MSG_KEY, 0666 | IPC_CREAT);
    if (msgid < 0) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }

    // Fork process
    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // Child process
        MsgBuf msg;
```

```
    // Fork process
    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // Child process
        MsgBuf msg;
        if (msgrcv(msgid, &msg, sizeof(msg.mtext), 1, 0) < 0) {
            perror("msgrcv");
            exit(EXIT_FAILURE);
        }
        std::cout << "Child received: " << msg.mtext << std::endl;
    } else {
        // Parent process
        MsgBuf msg;
        msg.mtype = 1;
        strcpy(msg.mtext, "Hello from parent");
        if (msgsnd(msgid, &msg, sizeof(msg.mtext), 0) < 0) {
            perror("msgsnd");
            exit(EXIT_FAILURE);
        }
        wait(NULL); // Wait for child to finish

        // Remove message queue
        msgctl(msgid, IPC_RMID, NULL);
    }

    return 0;
}
```