COMSATS UNIVERSITY ISLAMABADM, ATTOCK CAMPUS

OS PROJECT

TITLE : IPC USING PIPES AND MESSAGE PASSING

GROUP MEMBERS:

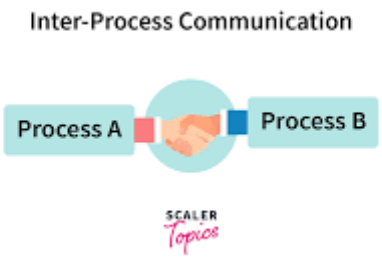
- **MUHAMMAD HASEEB (020)**
- **FARAIDON (05)**
- **JAHANZEB RAZAQ ()**
- **SYED ABDULLAH (037)**

Project Title: Implementation of Inter-Process Communication

---

1. Introduction:

Inter-Process Communication (IPC) serves as a mechanism for processes to exchange data and synchronize their actions. This project focuses on implementing IPC using two methods: Pipes and Message Passing. The primary goal is to demonstrate how processes communicate and share information using these mechanisms.



2. IPC using Pipes:

2.1. Overview:

Pipes in IPC facilitate unidirectional communication between processes within the same system. They are represented as a one-way communication channel.

2.2. Code Explanation:

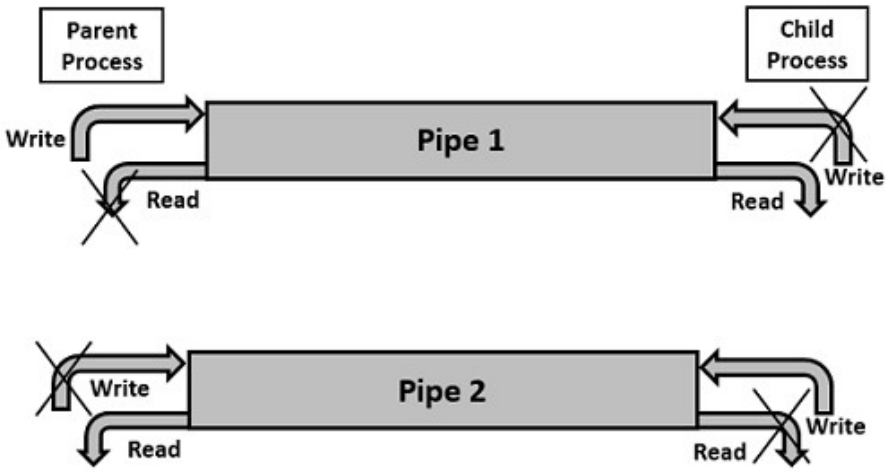
The provided C++ code demonstrates the usage of pipes for IPC. It involves the following steps:

- Creation of pipes using `pipe()` function.
- Forking a child process to establish a communication channel.
- Writing and reading messages via the pipe using `write()` and `read()` functions respectively.

2.3. Implementation Details:

```cpp
int main() {
    int pipefd[2];
    // File descriptors for the pipe:
    // pipefd[0] for reading,
    //pipefd[1] for writing
    char message[] = "Hello, IPC!"; // Message to be sent through the pipe
    char buffer[100]; // Buffer to receive the message

    if (pipe(pipefd) == -1) {
        std::cerr << "Pipe failed";
        return 1;
    }

    pid_t pid = fork(); // Create a child process

    if (pid < 0) {
        std::cerr << "Fork failed";
        return 1;
    }

    if (pid > 0) { // Parent process
        close(pipefd[0]); // Close the reading end of the pipe

        // Write the message to the pipe
        write(pipefd[1], message, sizeof(message));
        close(pipefd[1]); // Close the writing end of the pipe
    } else { // Child process
        close(pipefd[1]); // Close the writing end of the pipe

        // Read the message from the pipe
        read(pipefd[0], buffer, sizeof(buffer));
        std::cout << "Message received in child process: " << buffer << std::endl;
        close(pipefd[0]); // Close the reading end of the pipe
```

The code creates a parent-child relationship where the parent writes a message to the pipe, and the child reads the message. The management of pipe ends (`pipefd[0]` for reading and `pipefd[1]` for writing) ensures proper data transmission between processes.



3. IPC using Socket (Server-Client Model):

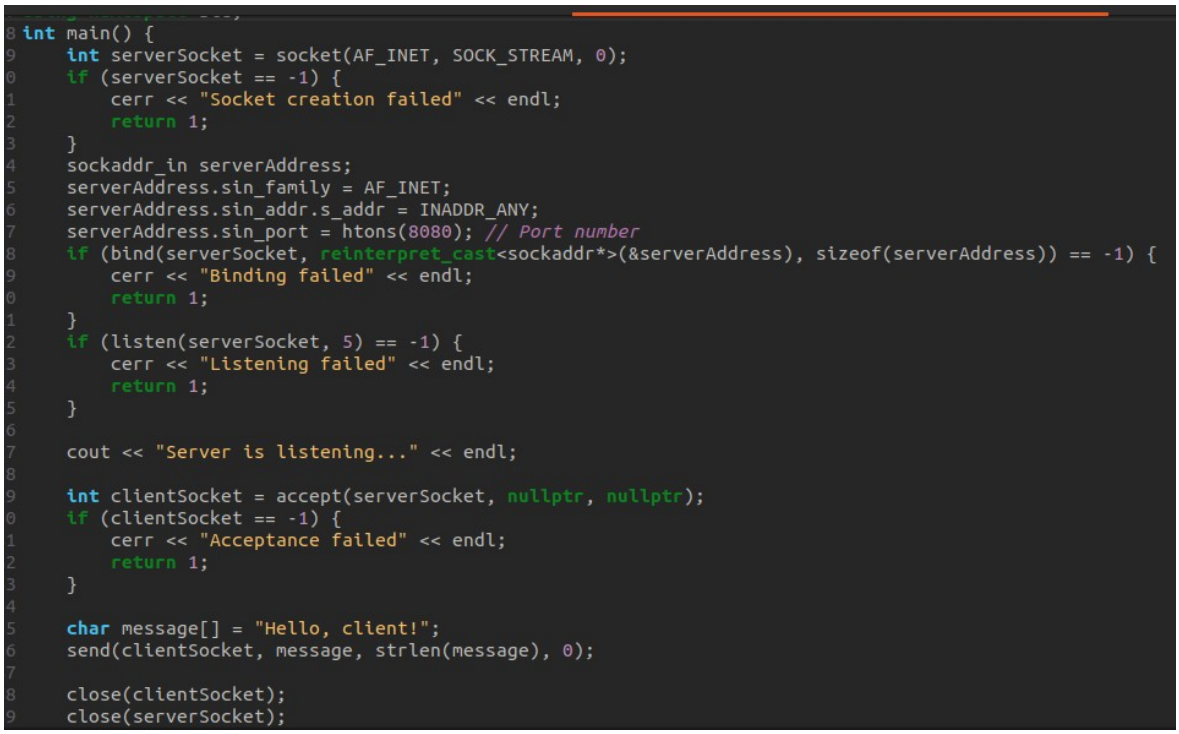
3.1. Overview:

Sockets enable bidirectional communication between processes across different systems. The server-client model establishes connections and transfers data between these processes.

3.2. Server Code Explanation:

The server-side C++ code utilizing sockets involves the following steps:

- Socket creation using `socket()` function.
- Binding the socket to an address via `bind()` function.
- Listening for incoming connections with `listen()` function.
- Accepting client connections using `accept()` function.
- Sending data to the client through the `send()` function.



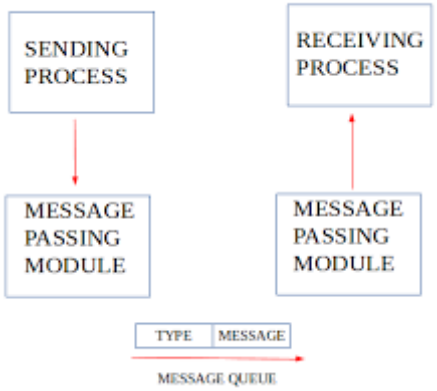
```cpp
int main() {
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket == -1) {
        cerr << "Socket creation failed" << endl;
        return 1;
    }
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY;
    serverAddress.sin_port = htons(8080); // Port number
    if (bind(serverSocket, reinterpret_cast<sockaddr*>(&serverAddress), sizeof(serverAddress)) == -1) {
        cerr << "Binding failed" << endl;
        return 1;
    }
    if (listen(serverSocket, 5) == -1) {
        cerr << "Listening failed" << endl;
        return 1;
    }

    cout << "Server is listening..." << endl;

    int clientSocket = accept(serverSocket, nullptr, nullptr);
    if (clientSocket == -1) {
        cerr << "Acceptance failed" << endl;
        return 1;
    }

    char message[] = "Hello, client!";
    send(clientSocket, message, strlen(message), 0);

    close(clientSocket);
    close(serverSocket);
```

3.3. Client Code Explanation:

The client-side C++ code utilizing sockets involves the following steps:

- Socket creation using `socket()` function.
- Specifying server address and port using `inet_pton()` and `connect()` functions.
- Receiving data from the server using `recv()` function.

```cpp
#include <cstring>

using namespace std; SC

int main() {
    int clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (clientSocket == -1) {
        cerr << "Socket creation failed" << endl;
        return 1;
    }
    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(8080); // Port number

    if (inet_pton(AF_INET, "127.0.0.1", &serverAddress.sin_addr) <= 0) {
        cerr << "Invalid address" << endl;
        return 1;
    }
    if (connect(clientSocket, reinterpret_cast<sockaddr*>(&serverAddress), sizeof(serverAddress)) == -1)
        cerr << "Connection failed" << endl;
        return 1;
    }

    char buffer[1024] = {0};
    recv(clientSocket, buffer, 1024, 0);
    cout << "Received: " << buffer << endl;

    close(clientSocket);

    return 0;
}
```



4. Conclusion:

This project successfully demonstrates IPC using Pipes and Sockets, showcasing the exchange of data between processes within the same system (Pipes) and across different systems (Sockets). Advantages include simplicity (Pipes) and versatility (Sockets) in handling IPC, while limitations involve unidirectional communication (Pipes) and the need for network connectivity (Sockets).

5. References:

- **Pipes:** Tutorialspoint - Inter Process Communication – Pipes
- **Sockets:** Opensource.com - Inter-process communication in Linux: Sockets and signals