Inter Process Communication (IPC)

Inter Process Communication (IPC) in Linux refers to a set of mechanisms that allow processes to communicate with each other, either by sharing data or by signaling events. IPC is critical in Linux and Unix-like operating systems, especially when different processes need to cooperate or exchange data. Here are the main IPC mechanisms available in Linux:

1. Pipes

- **Anonymous Pipes**: Used for communication between related processes, such as parent and child processes. Data written to one end of the pipe can be read from the other end.
  - Example:
    ```
    int fd[2]; // File descriptors for the pipe
    pipe(fd); // Creating the pipe
    // fd[0] for reading, fd[1] for writing
    ```
- **Named Pipes (FIFOs)**: Similar to anonymous pipes but can be used for communication between unrelated processes. They are created with a name in the filesystem.
  - Example:
    ```
    mkfifo myfifo  # Creating a named pipe
    ```

2. Message Queues

- Message queues allow processes to send and receive messages in a queue structure. Each message has a type identifier, which allows processes to selectively receive specific messages.
  - Example:
    ```
    #include <sys/msg.h>
    int msgid = msgget(key, IPC_CREAT | 0666);  // Create or get a message queue
    ```

3. Shared Memory

- Shared memory allows processes to share a memory region. This is the fastest form of IPC because data does not need to be copied between processes.
  - Example:
    ```
    int shmid = shmget(key, size, IPC_CREAT | 0666); // Create or get shared memory
    ```

4. Semaphores

- Semaphores are used to control access to a shared resource by multiple processes. They are typically used to synchronize processes or control access to shared resources like shared memory.
  - Example:
    ```
    int semid = semget(key, 1, IPC_CREAT | 0666); // Create or get a semaphore set
    ```

5. Sockets

- Sockets provide a mechanism for communication between processes over a network. This can be used for inter-process communication over the same machine or between machines over a network (TCP, UDP).
  - Example:
    ```
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);  // Create a socket
    ```

6. Signals

- Signals are used for simple communication between processes, mainly for sending notifications about events. Each signal has a default behavior, but processes can define custom handlers to respond to specific signals.
  - Example:
    ```
    signal(SIGINT, handler_function);  // Catch the SIGINT signal (Ctrl+C)
    ```

7. D-Bus

- D-Bus is a higher-level IPC system used for communication between applications. It's commonly used in desktop environments for event notifications, system services communication, etc.

8. Memory-Mapped Files (mmap)

- Memory mapping a file allows processes to share a file's contents by mapping the file into their virtual address spaces. This can also be used for IPC between processes.

- Example:
  void *ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

Choosing the Right IPC Mechanism:

- **Performance**: Shared memory and memory-mapped files provide the fastest communication.
- **Simplicity**: Pipes and signals are simple but may not be suitable for complex data exchange.
- **Scalability**: Message queues, semaphores, and sockets are better for large-scale, complex interactions.

Each IPC method serves different use cases, and the best choice depends on factors like data complexity, process relationship, and synchronization needs.

IPC between Processes on a Single Computer

In Linux, Inter-Process Communication (IPC) between processes on a single computer is commonly used to allow separate programs or different instances of the same program to share data and coordinate tasks. Several IPC mechanisms are optimized for communication between processes running on the same machine. Below are the main IPC methods used in such scenarios, along with examples in Linux programming.

1. Pipes

Pipes allow communication between related processes, typically parent and child processes.

Example: Anonymous Pipe

This is a unidirectional communication channel where one process writes to the pipe and another reads from it.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int fd[2];
    pipe(fd);  // Create a pipe

    if (fork() == 0) {  // Child process
        close(fd[0]);  // Close reading end
        write(fd[1], "Hello, Parent!", 15);
        close(fd[1]);  // Close writing end
    } else {  // Parent process
        char buffer[20];
        close(fd[1]);  // Close writing end
        read(fd[0], buffer, sizeof(buffer));
        printf("Received: %s\n", buffer);
        close(fd[0]);  // Close reading end
    }
    return 0;
}
```

Example: Named Pipe (FIFO)

A **named pipe** or FIFO allows communication between unrelated processes.

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

int main() {
    int fd;
    char *myfifo = "/tmp/myfifo";

    // Create the FIFO (named pipe)
    mkfifo(myfifo, 0666);

    if (fork() == 0) {  // Child process
        fd = open(myfifo, O_WRONLY);
        write(fd, "Hello from Child!", 17);
        close(fd);
    } else {  // Parent process
        fd = open(myfifo, O_RDONLY);
        char buffer[20];
        read(fd, buffer, sizeof(buffer));
        printf("Received: %s\n", buffer);
        close(fd);
        unlink(myfifo);  // Remove the FIFO
    }
    return 0;
}
```

2. Message Queues

Message queues allow processes to exchange messages in a queue structure. It is more flexible than pipes since messages can be prioritized.

Example: Message Queue

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

struct msg_buffer {
    long msg_type;
    char msg_text[100];
};

int main() {

    key_t key;
    int msgid;
    struct msg_buffer message;

    // Generate unique key
    key = ftok("progfile", 65);

    // Create message queue and return id
    msgid = msgget(key, 0666 | IPC_CREAT);

    if (fork() == 0) {  // Child process
        // Send a message
        message.msg_type = 1;
        strcpy(message.msg_text, "Hello from child process!");

        msgsnd(msgid, &message, sizeof(message), 0);
    } else {  // Parent process
```

```
        // Receive a message
        msgrcv(msgid, &message, sizeof(message), 1, 0);
        printf("Received message: %s\n", message.msg_text);

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

    return 0;
}
```

3. Shared Memory

Shared memory allows multiple processes to access the same memory segment. It is the fastest IPC method since no data needs to be copied between processes.

Example: Shared Memory

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main() {
    key_t key = ftok("shmfile", 65);  // Create unique key

    int shmid = shmget(key, 1024, 0666|IPC_CREAT);  // Create shared memory segment
    char *str = (char*) shmat(shmid, (void*)0, 0);  // Attach shared memory

    if (fork() == 0) {  // Child process
        strcpy(str, "Hello from Child!");  // Write to shared memory
        shmdt(str);  // Detach from shared memory
    } else {  // Parent process
        wait(NULL);  // Wait for child process to finish
        printf("Data read from shared memory: %s\n", str);  // Read from shared memory
        shmdt(str);  // Detach from shared memory
        shmctl(shmid, IPC_RMID, NULL);  // Destroy the shared memory
    }


    return 0;
}
```

4. Semaphores

Semaphores are used to control access to a shared resource by multiple processes. They are often used in conjunction with shared memory to ensure synchronization.

Example: Semaphore (POSIX)

```
#include <stdio.h>
#include <semaphore.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    sem_t *sem = sem_open("/mysem", O_CREAT, 0644, 1);  // Create or open a semaphore


    if (fork() == 0) {  // Child process
        sem_wait(sem);  // Lock the semaphore
        printf("Child process is in critical section.\n");
        sleep(2);
        printf("Child process is leaving critical section.\n");
        sem_post(sem);  // Unlock the semaphore
    } else {  // Parent process
        sem_wait(sem);  // Lock the semaphore
        printf("Parent process is in critical section.\n");
        sleep(2);
        printf("Parent process is leaving critical section.\n");
        sem_post(sem);  // Unlock the semaphore
        wait(NULL);  // Wait for child to finish
    }

    sem_close(sem);  // Close the semaphore
    sem_unlink("/mysem");  // Remove the semaphore
    return 0;
}
```

5. Signals

Signals are used to notify processes of system events or to send simple messages between processes. They are lightweight and typically used for signaling events like process termination.

Example: Sending Signals

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void handle_signal(int sig) {
    printf("Received signal: %d\n", sig);
}

int main() {
    signal(SIGUSR1, handle_signal);  // Setup signal handler

    if (fork() == 0) {  // Child process
        sleep(1);
        kill(getppid(), SIGUSR1);  // Send signal to parent
    } else {  // Parent process
        pause();  // Wait for signal
    }
    return 0;
}
```

6. Sockets (Local Communication)

Sockets can also be used for IPC on the same machine using Unix domain sockets. This provides a more complex but flexible method for communication.

Example: Unix Domain Sockets

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>


int main() {
    int sockfd;
    struct sockaddr_un addr;
    char buffer[100];

    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);

    addr.sun_family = AF_UNIX;
    strcpy(addr.sun_path, "/tmp/mysocket");

    if (fork() == 0) {  // Child process
        connect(sockfd, (struct sockaddr*)&addr, sizeof(addr));  // Connect to the socket
        write(sockfd, "Hello from Child!", 17);  // Write to socket
        close(sockfd);
    } else {  // Parent process
        bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));  // Bind to the socket
        listen(sockfd, 5);
        int clientfd = accept(sockfd, NULL, NULL);  // Accept a connection
        read(clientfd, buffer, sizeof(buffer));  // Read from socket
        printf("Received: %s\n", buffer);
        close(clientfd);
        close(sockfd);
        unlink("/tmp/mysocket");  // Clean up

    }

    return 0;
}
```

Summary
For processes running on the same machine, IPC methods like **pipes**, **shared memory**, **message queues**, and **signals** are typically used. Each method has its pros and cons, and the best choice depends on the nature

IPC between Processes on Different Systems

Inter-Process Communication (IPC) between processes on **different systems** in Linux typically relies on network-based mechanisms because the processes are no longer running in the same memory space.

Below are the primary methods for IPC between distributed processes across multiple machines, commonly leveraging networking protocols like TCP/IP, UDP, or Unix domain sockets.

1. Sockets (TCP/IP or UDP)

Sockets are the most commonly used IPC method for processes running on different systems, as they allow communication over a network using protocols such as TCP (reliable, connection-oriented) or UDP (unreliable, connectionless). TCP ensures data is delivered correctly, while UDP is faster but doesn't guarantee delivery.

Example: TCP Sockets (Client-Server Communication)

Server (Listening for connections):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>

#define PORT 8080
```

```
int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};

    // Create a socket file descriptor
    server_fd = socket(AF_INET, SOCK_STREAM, 0);

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;  // Accept connections from any IP
    address.sin_port = htons(PORT);

    // Bind the socket to the port
    bind(server_fd, (struct sockaddr *)&address, sizeof(address));

    // Listen for incoming connections
    listen(server_fd, 3);

    // Accept a connection from a client

    new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen);
    // Read data from the client
    read(new_socket, buffer, 1024);
    printf("Message from client: %s\n", buffer);

    // Send a response to the client
    char *response = "Hello from server";
    send(new_socket, response, strlen(response), 0);

    close(new_socket);
    close(server_fd);
    return 0;
}
```

Client (Connecting to the server):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char *message = "Hello from client";
    char buffer[1024] = {0};

    // Create a socket
    sock = socket(AF_INET, SOCK_STREAM, 0);

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

    // Convert IPv4 and IPv6 addresses from text to binary form
    inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);  // Server IP address

    // Connect to the server
    connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    // Send data to the server
    send(sock, message, strlen(message), 0);

    // Read the server's response
    read(sock, buffer, 1024);
    printf("Message from server: %s\n", buffer);

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

2. Message Queues (Distributed Systems)

Linux message queues are typically intended for local communication, but **Message Queue Servers** (such as **RabbitMQ**, **ZeroMQ**, **Apache Kafka**) allow communication between systems across a network. These tools handle distributed message queues and provide more sophisticated messaging and queue management over TCP/IP.

Example: ZeroMQ for Messaging

ZeroMQ provides an easy-to-use API for building messaging systems in distributed environments. You can create client-server applications that communicate using ZeroMQ.

Server (using ZeroMQ):

```
#include <zmq.h>
#include <string.h>
#include <stdio.h>

int main() {
    void *context = zmq_ctx_new();
    void *responder = zmq_socket(context, ZMQ_REP);  // Create a reply socket

    // Bind the socket to an IP address and port
    zmq_bind(responder, "tcp://*:5555");

    while (1) {
        char buffer[10];
        zmq_recv(responder, buffer, 10, 0);  // Receive a message
        printf("Received: %s\n", buffer);

        // Send a reply to the client
        zmq_send(responder, "World", 5, 0);
    }

    zmq_close(responder);
    zmq_ctx_destroy(context);
    return 0;
}
```

Client (using ZeroMQ):

```
#include <zmq.h>
#include <stdio.h>
#include <string.h>


int main() {
    void *context = zmq_ctx_new();
    void *requester = zmq_socket(context, ZMQ_REQ);  // Create a request socket

    // Connect to the server
    zmq_connect(requester, "tcp://localhost:5555");

    // Send a message to the server
    zmq_send(requester, "Hello", 5, 0);

    // Receive the server's reply
    char buffer[10];
    zmq_recv(requester, buffer, 10, 0);
    printf("Received: %s\n", buffer);

    zmq_close(requester);
    zmq_ctx_destroy(context);
    return 0;
}
```

3. Remote Procedure Calls (RPC)

RPC allows a program to execute a procedure (function) on a different machine. It abstracts the network communication, making it appear as if the function is being called locally.

Example: gRPC (Google Remote Procedure Call)

gRPC is an open-source framework that provides high-performance RPC for inter-process communication across distributed systems. It uses protocol buffers (protobuf) for defining service contracts.

- You would define a service with RPC methods in a .proto file.
- Then, generate client and server code, which allows communication between systems as if the functions were local.

Here's an example of what an RPC might look like using gRPC (in Python for simplicity):

```
// hello.proto
syntax = "proto3";

service HelloService {
  rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
  string name = 1;
}

message HelloResponse {
  string message = 1;
}
```

Once the .proto file is defined, you generate the server and client stubs and implement the service.

4. Distributed Shared Memory (DSM)

In distributed systems, **Distributed Shared Memory (DSM)** is an abstraction that allows processes on different machines to access shared memory, giving the illusion of a common memory space.

- Implementations such as **OpenMPI** or **TreadMarks** provide DSM features for high-performance computing.
- This mechanism is mainly used in specialized distributed environments rather than general-purpose applications.

5. File-Based IPC (NFS or Network Filesystems)

Processes running on different systems can communicate through **shared files** stored on a **network filesystem** like **NFS** (Network File System). One process writes data to a file, and another reads it, similar to file-based IPC in single-system environments.

- This approach is slower due to file system latencies and network I/O, but it is simple to implement.

6. D-Bus (Distributed)

While **D-Bus** is generally a local IPC mechanism for desktop environments, it can be extended to support **remote** communication using an underlying transport mechanism like TCP/IP. However, this is less common than using sockets or message queues.

Summary

For IPC between processes on different systems, the most common and effective methods include:

1. **Sockets (TCP/UDP)**: Standard approach for network communication.
2. **Message Queue Systems (ZeroMQ, RabbitMQ, Kafka)**: Efficient for distributed messaging.
3. **Remote Procedure Calls (RPC, gRPC)**: Makes remote communication look like local function calls.
4. **File-Based IPC (Network Filesystems like NFS)**: Simple but slower.
5. **Distributed Shared Memory (DSM)**: Advanced method for distributed computing environments.

The best choice depends on your application's needs (e.g., performance, complexity, scalability). **Sockets** and **message queues** are often the go-to solutions for communication across systems.

Unnamed Pipes Creation

To create **unnamed pipes** in Linux, the primary function used is pipe(). Here's a breakdown of the key functions involved in the process:

1. pipe()

- The pipe() system call is used to create an unnamed pipe.
- It creates a pipe that allows for one-way communication between processes, meaning one process writes to the pipe, and another reads from it.
- The function creates two file descriptors: one for reading and one for writing.

Syntax:

int pipe(int pipefd[2]);

- **Arguments**:
  - pipefd[2]: An array of two integers, where:
    - pipefd[0]: The file descriptor for the **read** end of the pipe.
    - pipefd[1]: The file descriptor for the **write** end of the pipe.
- **Return Value**:
  - Returns 0 on success.
  - Returns -1 on failure and sets errno to indicate the error.

2. fork()

- fork() is typically used in combination with pipe() to create a child process that can share the pipe with the parent process.
- After calling fork(), both the parent and child processes will have access to the pipe, allowing them to communicate.
- In the child process, you usually close the unused end of the pipe (for example, if the child writes, it closes the read end), and in the parent process, you do the same (close the unused write end).

Syntax:

pid_t fork(void);

- **Return Value**:
  - In the parent process, fork() returns the PID of the child.
  - In the child process, it returns 0.
  - If fork() fails, it returns -1.

3. read()

- read() is used to read data from the **read** end of the pipe (pipefd[0]).
- It attempts to read a specified number of bytes from the pipe into a buffer.

Syntax:

ssize_t read(int fd, void *buf, size_t count);

- **Arguments**:
  - fd: The file descriptor for the read end of the pipe (in this case, pipefd[0]).
  - buf: A pointer to the buffer where the read data will be stored.
  - count: The maximum number of bytes to read.
- **Return Value**:
  - Returns the number of bytes read on success.
  - Returns 0 when it reaches the end of the file (EOF).
  - Returns -1 on failure.

4. write()

- write() is used to write data to the **write** end of the pipe (pipefd[1]).
- It writes a specified number of bytes to the pipe from a buffer.

Syntax:

ssize_t write(int fd, const void *buf, size_t count);

- **Arguments**:
  - fd: The file descriptor for the write end of the pipe (in this case, pipefd[1]).
  - buf: A pointer to the buffer containing the data to write.
  - count: The number of bytes to write.
- **Return Value**:
  - Returns the number of bytes written on success.
  - Returns -1 on failure.

5. close()

- close() is used to close a file descriptor when it is no longer needed.
- It is important to close the unused end of the pipe in both the parent and child processes to avoid deadlocks.

Syntax:

int close(int fd);

- **Arguments**:
  - fd: The file descriptor to close (either pipefd[0] or pipefd[1]).
- **Return Value**:
  - Returns 0 on success.
  - Returns -1 on failure.

Example: Creating an Unnamed Pipe Between Parent and Child Process

Here's an example of creating an unnamed pipe to communicate between a parent and child process using pipe(), fork(), read(), write(), and close():

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>


int main() {
    int pipefd[2];  // Array to hold the read and write file descriptors
    char writeMessage[] = "Hello from parent!";
    char readBuffer[100];

    // Create the pipe
    if (pipe(pipefd) == -1) {
        perror("pipe failed");
        return 1;
    }

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


    if (pid < 0) {
        // Fork failed
        perror("fork failed");
        return 1;
    }

    if (pid == 0) {  // Child process
        // Close the unused write end
        close(pipefd[1]);

        // Read from the pipe
        read(pipefd[0], readBuffer, sizeof(readBuffer));
        printf("Child received: %s\n", readBuffer);


        // Close the read end
        close(pipefd[0]);
    } else {  // Parent process
        // Close the unused read end
        close(pipefd[0]);

        // Write to the pipe
        write(pipefd[1], writeMessage, strlen(writeMessage) + 1);

        // Close the write end
        close(pipefd[1]);
    }

    return 0;
}
```

Breakdown:

- The parent process writes a message to the pipe using write().
- The child process reads the message from the pipe using read().
- Both processes close the unused end of the pipe to avoid deadlocks and ensure proper communication.

Summary of Functions:

- **pipe()**: Creates an unnamed pipe and returns two file descriptors (read and write ends).
- **fork()**: Creates a child process that inherits the pipe file descriptors.
- **read()**: Reads data from the pipe (typically used by the child process).
- **write()**: Writes data to the pipe (typically used by the parent process).
- **close()**: Closes the file descriptors when they are no longer needed.

Named Pipes (FIFO) creation required functions

To create **named pipes (FIFOs)** in Linux, the key function used is mkfifo(). Named pipes allow communication between unrelated processes and are represented as files in the filesystem. Here's a breakdown of the main functions involved in creating and using FIFOs:

1. mkfifo()

- The mkfifo() system call is used to create a **named pipe (FIFO)**.
- This function creates a special file in the filesystem that behaves like a pipe, allowing for IPC between unrelated processes.

Syntax:

int mkfifo(const char *pathname, mode_t mode);

- **Arguments**:
  - pathname: The path of the FIFO to be created.
  - mode: The file permissions (similar to those used with open() or chmod()). This specifies the access mode (e.g., 0666 for read and write by all).
- **Return Value**:
  - Returns 0 on success.
  - Returns -1 on failure and sets errno to indicate the error.

2. open()

- open() is used to open the named pipe for reading or writing, just like any regular file.
- A process can open a FIFO in read-only or write-only mode.
- FIFOs behave like files: one process writes to the FIFO, and another process reads from it. The open operation will block if there is no other process opening the pipe in the opposite mode (i.e., writing if the pipe is opened for reading and vice versa).

Syntax:

int open(const char *pathname, int flags);

- **Arguments**:
  - pathname: The path of the FIFO to open.
  - flags: Access modes, such as:
    - O_RDONLY: Open for reading only.
    - O_WRONLY: Open for writing only.
    - O_RDWR: Open for both reading and writing (rare for FIFOs).
    - You can also use non-blocking mode with O_NONBLOCK.
- **Return Value**:
  - Returns a file descriptor on success.
  - Returns -1 on failure and sets errno to indicate the error.

3. read()

- read() is used to read data from the FIFO.
- When a process reads from a named pipe, it blocks until data is available.

Syntax:

```
ssize_t read(int fd, void *buf, size_t count);
```

- **Arguments**:
    - fd: The file descriptor for the open FIFO.
    - buf: The buffer where the read data will be stored.
    - count: The maximum number of bytes to read.
- **Return Value**:
    - Returns the number of bytes read on success.
    - Returns 0 on end-of-file (EOF).
    - Returns -1 on failure.

4. write()

- write() is used to write data to the FIFO.
- When a process writes to a named pipe, it blocks until a reading process opens the FIFO.

Syntax:

```
ssize_t write(int fd, const void *buf, size_t count);
```

- **Arguments**:
    - fd: The file descriptor for the open FIFO.
    - buf: The buffer containing the data to write.
    - count: The number of bytes to write.
- **Return Value**:
    - Returns the number of bytes written on success.
    - Returns -1 on failure.

5. close()

- close() is used to close the file descriptor when communication is finished.
- It is important to close the FIFO once the read or write operation is done to avoid resource leaks.

Syntax:

```
int close(int fd);
```

- **Arguments**:
    - fd: The file descriptor of the open FIFO.
- **Return Value**:
    - Returns 0 on success.
    - Returns -1 on failure.

6. unlink()

- unlink() is used to remove the named pipe (FIFO) from the filesystem once it's no longer needed.

Syntax:

```
int unlink(const char *pathname);
```

- **Arguments**:
    - pathname: The path of the FIFO to be removed.
- **Return Value**:
    - Returns 0 on success.
    - Returns -1 on failure.

Example: Creating and Using a Named Pipe (FIFO)

Here's a simple example of how to create and use a named pipe for communication between two unrelated processes:

Step 1: Create the FIFO (Named Pipe)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main() {
    char *fifoPath = "/tmp/myfifo";  // Path to the FIFO

    // Create the named pipe (FIFO) with read and write permissions for all
    if (mkfifo(fifoPath, 0666) == -1) {
        perror("mkfifo failed");
        return 1;
    }

    printf("FIFO created at: %s\n", fifoPath);
    return 0;
}
```

Step 2: Write to the FIFO (Writer Program)

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

int main() {
    char *fifoPath = "/tmp/myfifo";
    char message[] = "Hello, from the writer!";

    // Open the FIFO for writing
    int fd = open(fifoPath, O_WRONLY);
    if (fd == -1) {
        perror("open failed");
        return 1;
    }

    // Write to the FIFO
    write(fd, message, sizeof(message));

    // Close the FIFO
    close(fd);

    printf("Message written to FIFO.\n");
    return 0;
}
```

Step 3: Read from the FIFO (Reader Program)

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

int main() {
    char *fifoPath = "/tmp/myfifo";
    char buffer[100];

    // Open the FIFO for reading
    int fd = open(fifoPath, O_RDONLY);
    if (fd == -1) {
        perror("open failed");
        return 1;
    }
    // Read from the FIFO
    read(fd, buffer, sizeof(buffer));

    // Print the message
    printf("Message received: %s\n", buffer);

    // Close the FIFO
    close(fd);

    return 0;
}
```

Step 4: Clean up the FIFO (Optional)

After you're done using the FIFO, you can remove it using the unlink() function or manually using the rm command.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    char *fifoPath = "/tmp/myfifo";

    // Remove the FIFO
    if (unlink(fifoPath) == -1) {
        perror("unlink failed");
        return 1;
    }

    printf("FIFO removed.\n");
    return 0;
}
```

Summary of Functions:

- **mkfifo()**: Creates a named pipe (FIFO) in the filesystem.
- **open()**: Opens the FIFO for reading or writing.
- **read()**: Reads data from the FIFO.
- **write()**: Writes data to the FIFO.
- **close()**: Closes the file descriptor associated with the FIFO.
- **unlink()**: Removes the FIFO from the filesystem when it's no longer needed.

Differences between Named and Unnamed Pipes:

Feature	**Unnamed Pipe**	**Named Pipe (FIFO)**
Creation	Created using pipe()	Created using mkfifo() or mkfifo command
Naming	No name in the filesystem	Exists as a special file in the filesystem
Scope	Used for related processes (parent-child)	Can be used for unrelated processes
Lifetime	Exists only while processes are running	Persists as a file until explicitly removed
Access	Only processes sharing the pipe via inheritance	Any process with access to the file can use it

Feature	Unnamed Pipe	Named Pipe (FIFO)
Unidirectional/Bidirectional	Typically unidirectional (requires two pipes for bidirectional)	Unidirectional, but bidirectional with two FIFOs
Use case	Parent-child process communication	Unrelated process communication
Performance	Faster (in-memory)	Slower (filesystem access)

Summary:

- **Unnamed pipes** are limited to communication between related processes (like parent and child) and are not visible in the filesystem. They are faster and automatically cleaned up.
- **Named pipes (FIFOs)** allow communication between unrelated processes through a named file in the filesystem. They provide more flexibility but are slower and require explicit cleanup.

popen() and **pclose()** functions

In Linux, the **popen()** and **pclose()** functions are part of the standard library for handling **pipes to and from processes**. These functions simplify communication with other processes by opening a process for reading or writing, much like how files are opened.

1. popen(): Open a Process by Creating a Pipe

The popen() function opens a **pipe** between the calling process and another process, running a command in the shell. It creates a process and returns a **file pointer** that can be used to either read from or write to the process, depending on the specified mode.

Syntax:

FILE *popen(const char *command, const char *mode);

- **Arguments**:
    - command: A string containing the command to be executed by the shell.
    - mode: A string representing the mode for the pipe:
        - "r": Open a pipe for **reading** (the calling process reads from the command's output).
        - "w": Open a pipe for **writing** (the calling process writes to the command's standard input).
- **Return Value**:
    - On success, popen() returns a FILE* that can be used with fgets(), fputs(), fprintf(), fread(), fwrite(), and other file functions.
    - On failure, it returns NULL.

Key Features:

- **Bidirectional use**: Although popen() is used in one direction at a time (either reading from or writing to the command), it simplifies the interaction between processes.
- **Command Execution**: The command passed to popen() is executed by the shell (typically /bin/sh -c), and the output or input is directed to the calling process through the pipe.

2. pclose(): Close the Pipe and Process

The pclose() function closes the stream opened by popen() and waits for the associated process to terminate. This ensures proper cleanup of resources.

Syntax:

int pclose(FILE *stream);

- **Arguments**:
    - stream: The FILE* pointer returned by popen().
- **Return Value**:
    - Returns the **exit status** of the command run by popen().
    - Returns -1 on error (e.g., if pclose() is called with a stream that wasn't opened by popen()).

Example 1: Reading Output from a Command (using popen() in Read Mode)
This example shows how to run a shell command and read its output using popen().

```
#include <stdio.h>

int main() {
    FILE *fp;
    char path[1035];

    // Open the command for reading
    fp = popen("ls /etc", "r");
    if (fp == NULL) {
        perror("popen failed");
        return 1;
    }

    // Read the output a line at a time and print it
    while (fgets(path, sizeof(path), fp) != NULL) {
        printf("%s", path);
    }

    // Close the pipe
    pclose(fp);
    return 0;
}
```

How It Works:

- The program runs the ls /etc command and reads its output line by line using fgets().
- The pipe is opened for reading with "r", and popen() returns a FILE* that is used to read the command's output.
- pclose() closes the pipe and waits for the command to complete.

Example 2: Writing Input to a Command (using popen() in Write Mode)
This example demonstrates writing data to the input of another process (e.g., writing to a command like sort).

```
#include <stdio.h>

int main() {
    FILE *fp;

    // Open the command for writing
    fp = popen("sort", "w");
    if (fp == NULL) {
        perror("popen failed");
        return 1;
    }

    // Write some input to the command
    fprintf(fp, "Banana\n");
    fprintf(fp, "Apple\n");
    fprintf(fp, "Cherry\n");
    // Close the pipe
    pclose(fp);

    return 0;
}
```

How It Works:

- The program runs the sort command and writes several strings to it using fprintf().
- The pipe is opened in write mode with "w", and the data is written to the sort command’s standard input.
- pclose() closes the pipe and waits for the sort process to complete.

Advantages of popen() and pclose():

- **Simplicity**: These functions are easier to use than manually setting up pipes and using fork() and exec() for process creation.
- **Convenience**: They allow for file-like handling of process communication.
- **Error Handling**: pclose() ensures that the process is properly cleaned up, including waiting for the process to finish and capturing its exit status.

Disadvantages:

- **Limited Communication**: popen() only allows unidirectional communication (either reading or writing), not both simultaneously.
- **Shell Dependency**: Since the command is executed by a shell, there may be issues with shell quoting, or extra overhead depending on the shell used (/bin/sh by default).

Summary of popen() and pclose():

- **popen()**: Opens a pipe to or from a process and runs a command through the shell. It returns a FILE* that can be used for reading or writing.
- **pclose()**: Closes the pipe and waits for the process to terminate, returning the command’s exit status.

These functions are useful when you need a simple, file-like interface for communicating with external commands from within your program.