Interprocess Communication

Interprocess Communication

- **Inter-process communication** (**IPC**) is a set of methods for the exchange of data among multiple threads in one or more processes.

- Processes may be running on one or more computers connected by a network.

- Operating systems provide facilities/resources for inter-process communications (**IPC**), such as message queues, semaphores, and shared memory.

Interprocess Communication Methods

- Inter-process communication techniques can be divided into various types. These are:

1. Shared memory
2. Message Passing / Message Queue
3. Sockets
4. Pipes
5. FIFO (Named Pipes)

Interprocess Communication Methods

- **Shared memory**: Shared memory is an efficient means of passing data between programs.
- An area is created in memory by a process, which is accessible by another process.
- Processes communicate by reading and writing to that memory space.

- **Message queues**: By using this method, a developer can pass messages between messages via a single queue or a number of message queues. A system kernel manages this mechanism.

Interprocess Communication Methods

- **Sockets**: We use this mechanism to communicate **over a network, between a client and a server**. This method facilitates a standard connection that is independent of the type of computer and the type of operating system used.

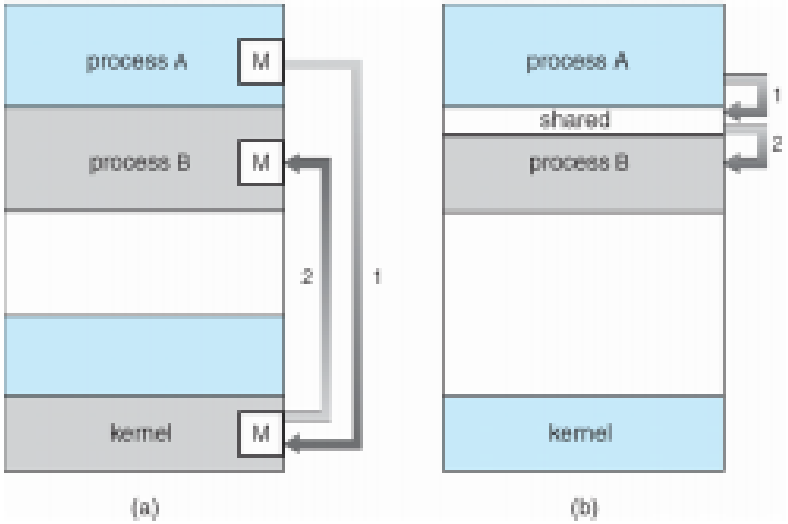
- **FIFO**: A FIFO or 'first in, first out' is a one-way flow of data.
- FIFOs are similar to pipes
- FIFOs are identified in the file system with a name.
- FIFOs are 'named pipes'.

Processes

- **Independent process** cannot affect or be affected by the execution of another process

- **Cooperating process** can affect or be affected by the execution of another process

Interprocess Communication Methods



(a) Message passing (Indirect)

(b) Shared Memory (Direct)

Shared Memory

Also known as Direct Communication

- Processes must name each other explicitly:

send (P, message) – send a message to process P

receive(Q, message) – receive a message from process Q

Properties of communication link:

- A link is associated with exactly one pair of communicating processes
- **Between each pair there exists exactly one link**
- The link may be unidirectional, but is usually bi-directional

Message Passing

Also known as In-direct Communication

- Messages are directed and received from **mailboxes** (also referred to as ports)
- Each mailbox has a unique id
- Processes can communicate only if they share a mailbox

Properties of communication link

- Link established only if processes share a common mailbox
- A link may be associated with many processes
- Link may be unidirectional or bi-directional

IPC: Message Passing

- **Operations**
- create a new mailbox
- send and receive messages through mailbox
- destroy a mailbox

Operations/ Primitives are defined as:

send(mailbox, message) – send message to mail box

receive(mailbox, message) – mailbox receives message.

Example:

- send(A, message) – send a message to mailbox A
- receive(A, message) – receive a message from mailbox A

IPC: Message Passing

Mailbox sharing

- P1, P2, and P3 share mailbox A
- P1, sends; P2 and P3 receive
- Who gets the message?

Solutions:

- Allow a **link** to be associated **with at most two processes**
- **Allow only one process at a time** to execute a receive operation
- Allow the system to **select arbitrarily the receiver**. Sender is notified who the receiver was.

IPC: Message Passing

Sending Process could be:

- **Blocking:** Sending process wait (I.e., block) for receiver

Blocking: Slows down sender

- **Non-blocking:** Requires buffering between sender and receiver

IPC: Message Passing

If P and Q wish to communicate, they need to:

- – establish a communication link between them
- – exchange messages via send/receive

Implementation of communication link

- – physical (e.g., shared memory, hardware bus)
- – logical (e.g., logical properties/ programming)
- (Programming-level mechanisms (e.g., sockets, pipes, message queues)

IPC: Message Passing

Producer-Consumer Problem :Paradigm for cooperating processes

- Producer process produces information that is consumed by a consumer process

- – **unbounded-buffer:** places no practical limit on the size of the buffer
- – **bounded-buffer:** assumes that there is a fixed buffer size

IPC: Message Passing Synchronization

- **Message passing may be either:**
  - Blocking
  - Non-blocking

1. **Blocking** is considered synchronous:

- **Blocking send:** has the sender block until the message is received
- **Blocking receive:** has the receiver block until a message is available
- Because both sender and receiver are forced to synchronize at the moment of message transfer — they must wait for each other.

IPC: Message Passing Synchronization

2. Non-blocking is considered asynchronous:

- **Non-blocking send:** has the sender send the message and continue

- **Non-blocking receive:** has the receiver receive a valid message or null

IPC: Message Passing

Buffering: When two processes communicate via message passing, the messages are stored in a **queue** attached to the communication link.
This queue is called the **buffer**.

- Queue of messages attached to the link;
- It is implemented in one of three ways:
- **1. Zero capacity** – zero messages
  - (No message can be stored. Queue is always empty.)
  - **2. Bounded capacity** – finite length of n messages
  - The buffer can hold a limited number of messages.
- **3. Unbounded capacity** – infinite length
  - The **sender never waits**, because the buffer can grow as needed.

Message Queues

- The messages that pass through the queue have two components:
- **one that describes the message (*header*) an one holding the content (*message body*).**

- There is no designated structure for the message body, so the users can define their own message types having different structure and dimension.

- The access to the queue is automatically synchronized by the operating system, such that if several processes wish to perform read and write operations upon the queue, its content remains consistent.

Communication in Client-Server Systems

Basic methods user are:

1. Sockets
2. Pipe

Communication in Client-Server Systems

Sockets

- Defined as end point of communication
- Pair of processes communicate over the network user sockets- one socket for each process
- Socket is identified by an IP address which is concatenated with port number
- Server listen to client via a port, server receives the request and complete the connection.

Communication in Client-Server Systems

Sockets

- It created using socket() system call.
- It takes:
  - Communication Domain
  - Socket type
  - Protocol to be used: as argument

Sockets

- A **network socket** is an endpoint of an inter-process communication flow across a computer network.

- **A socket address is the combination of an IP address and a port number**, much like one end of a telephone connection is the combination of a phone number and a particular extension.

Sockets

- Based on this address, internet sockets deliver incoming data packets to the appropriate application process or thread.
- A **socket API** is an application programming interface (API), usually provided by the operating system, that allows application programs to control and use network sockets. Internet socket APIs are usually based on the Berkeley sockets standard.
- An Internet socket is characterized by a unique combination of the following:
- Local socket address: Local IP address and port number
- Remote socket address
- Protocol

Sockets

There are several Internet socket types available:

- **Datagram sockets**, also known as connectionless sockets, E.g. User Datagram Protocol(UDP) (used for Broadcasting)

- **Stream sockets**, also known as connection-oriented sockets

E.g: Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP). (used for Peer to Peer)

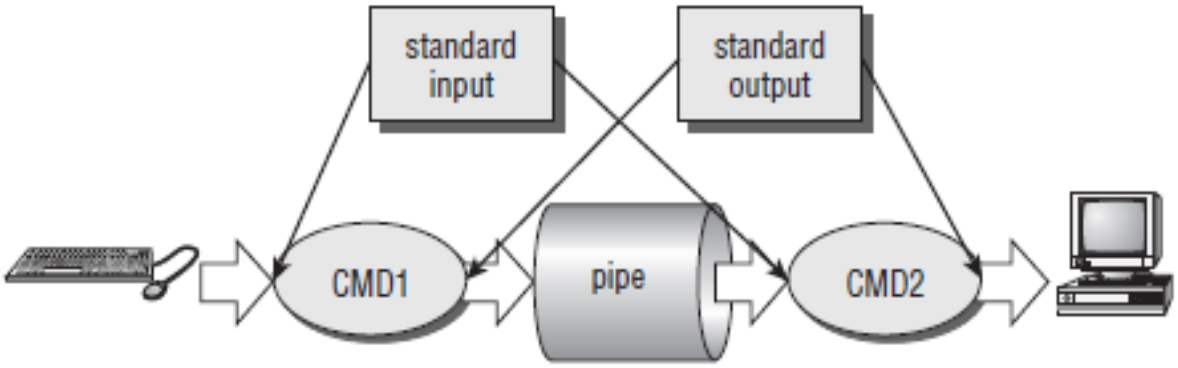
- **Raw sockets** (or *Raw IP sockets*), typically available in routers and other network equipment.

Pipe

- A pipe is a communication channel between two ends. It is mostly used to communicate between processes running within a computer.
- It is used on the command line to direct the output of a command as input of the next command.
- **A pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.**

Pipe

- linking shell commands together so that the output of one process is fed straight to the input of another.
- For shell commands, this is done using the pipe character to join the commands, such as **cmd1 | cmd2.**



Pipe

- Pipes have two limitations.

✓ They have been **half duplex** (i.e., data flows in only one direction).

✓ Pipes can be used only between processes that have a common ancestor (parent).

Pipe

- **Pipe()** is a system call that facilitates inter-process communication.
- **Pipe opens a virtual file, which is used for communication between parent and child.**
- One process can write to this "virtual file" or pipe and another related process can read from it.

Types of Pipe

- **Unnamed**
  - Virtual file used to send data between 2 processes.
  - Unnamed pipe can only work with related process.

- **Named or FIFO**
  - Created by using **mkfifo command**
  - It is a permanent file.
  - FIFO is same to unnamed file but with a facility that it can be accessed from anywhere.

Interprocess Communication

- If a process tries to read before something is written to the pipe, the process is suspended until something is written.

- Syntax :

```
#include <unistd.h>

int pipe(int filedes[2]);
```

Returns: 0 if OK, 1 on error

Interprocess Communication

- Two file descriptors are returned through the fileds argument: fd[0] is open for reading, and fd[1] is open for writing.
- The output of filedes[1] is the input for filedes[0].

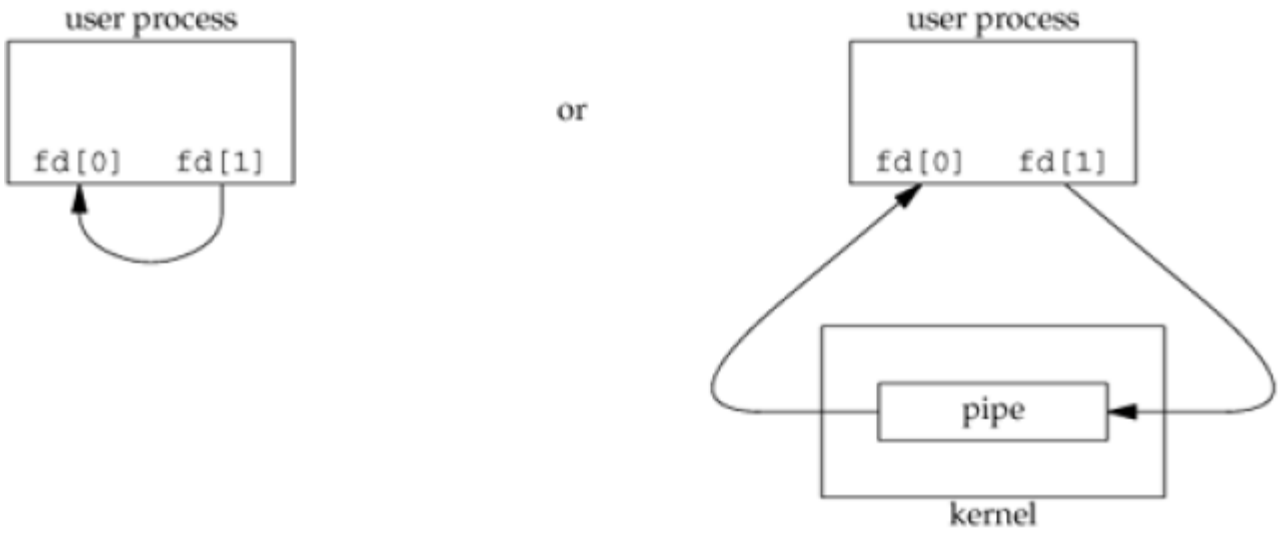
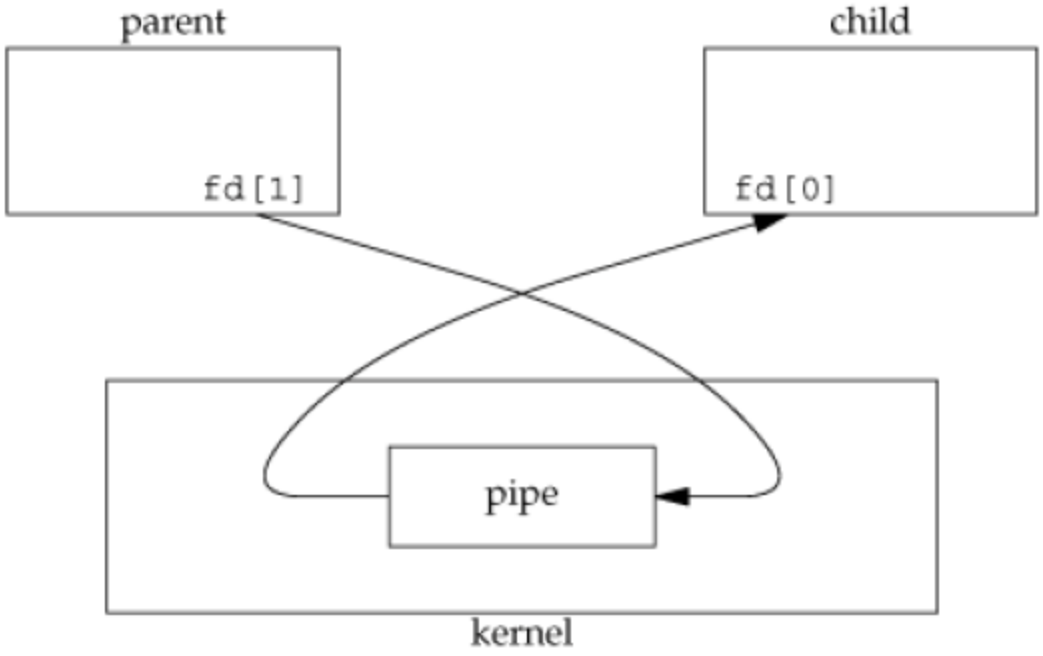


Figure : Pipe from parent to child



Example

- pip[0] - the read end of the pipe - is a file descriptor used to read from the pipe
- pip[1] - the write end of the pipe - is a file descriptor used to write to the pipe

- It is possible to have a series of processes arranged in a a pipeline, with a pipe between each pair of processes in the series.

Popen and pclose

popen() and pclose() are used to run **shell commands** from a C program **and read/write to the command through a pipe**.

C program <—— pipe ——> Shell command

popen() — open a process by creating a pipe

Syntax: FILE *fp (const char *command, const char *mode);

Parameters:

command → shell command to execute

mode → "r" to **read output** of command
"w" to **write input** to command

Use popen() when you want to:

✓ Execute a shell command
✓ Read its output or write input
✓ Easily handle the command like a file (FILE*)

Returns

FILE* stream connected to the process

NULL on error

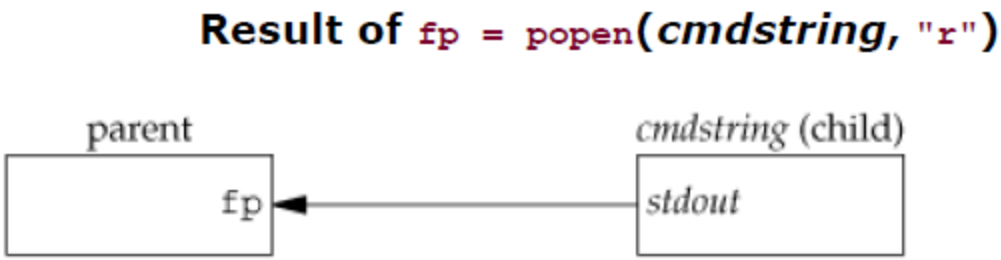
Popen and pclose

pclose() Function:

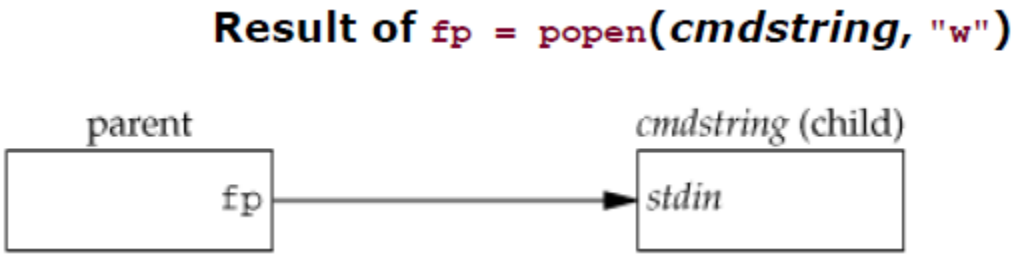
Purpose: pclose() closes a pipe that was previously opened by popen(). It waits for the associated process (the shell command) to terminate and then returns its exit status.

Popen and pclose

- If type is "r", the file pointer is connected to the standard output of cmd string. (i.e read into)



- If type is "w", the file pointer is connected to the standard input of cmd string (i.e write into)



FIFOs (Named Pipes)

- FIFOs are sometimes called named pipes.
- if two unrelated processes want to be able to exchange data?
- We do this using FIFOs.

FIFOs

- One of the major disadvantage of **pipes is that the they can not be accessed using their names** by any other process other than child and the parent.

- The work around for this problem is to create a named pipe which is also called as a FIFO, which stands for First in First out, meaning the data that is written into the pipe first will be read out first always.

- **fifo are created using the function mkfifo()**

FIFOs

- Once the file is created, it needs to be opened using the system call open()
- the data can be read and written from the file using read() and write system calls.

- Another advantage of a fifo over the pipes is that **fifo are bidirectional**, that is the same fifo can be read from as well and written into.

FIFOs

- One of the **examples you can think of using a named pipe is communication between a server and a client.**

- There are two fifos one of the server and the other of the client, then the client can send request to the server on the server fifo which the server will read and respond back with the reply on the client's fifo.

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

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

Returns: 0 if OK, 1 on error

Unnamed Pipe with popen() and pclose()

1. Write into a pipe

Syntax: File pointer = popen(“process”, “mode”);

```
rd=popen("wc -c","w");
 fwrite(buffer,sizeof(char),strlen(buffer),rd);
```

2. Read from file

```
rd=popen("ls","r");
fread(buffer, 1, 50, rd);
```

3. Close file after reading or writing

```
pclose(rd);
```

Named Pipe

1. Create a file

- int fd;
- fd = mkfifo("fifo1",0777);

2. Write into named pipe

```
fd=open("fifo1",O_WRONLY);
write(fd,"written",7);
```

3. Read from a named pipe

```
fd=open("fifo1",O_RDONLY);
nbyte=read(fd,buffer,100);
```