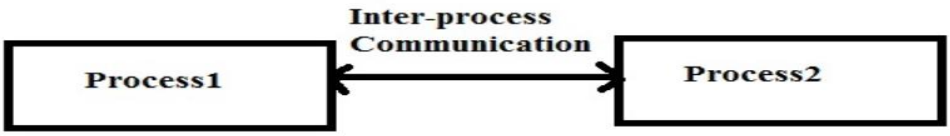
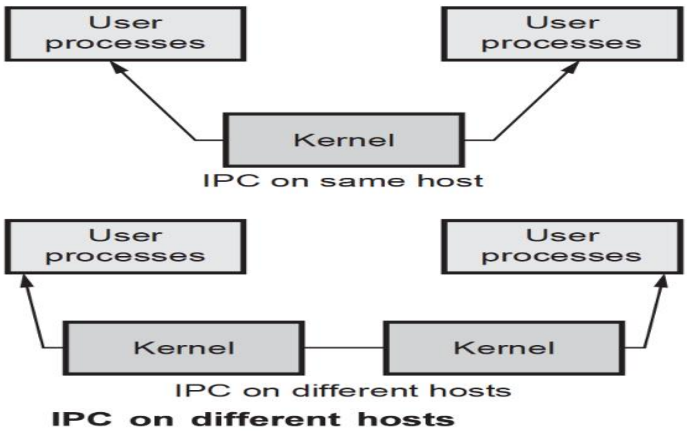
Inter Process Communication (IPC) using Pipes

Inter Process Communication (IPC) is a mechanism that involves communication of one process with another process.



Communication can be of two types

- Between related processes initiating from only one process, such as parent and child processes.
- Between unrelated processes, or two or more different processes.



Need for IPC

- Sharing of information: Inter-process communication helps in sharing information when several processes try to access a particular file concurrently. It is, therefore, necessary for the processes to cooperate with each other so that the information provided is correct.
- Speed: Dividing a big task into smaller tasks and then running them concurrently can speed up the entire functioning of the system. We need inter-process communication in such a case. The multiprocessor environment is a very good example of this.
- Modularity: When a system is divided into modules, such modules may require information sharing.
- Convenience: Allowing multiple processes to run simultaneously provides a better experience to any user

In inter-process communication (IPC), pipes are a fundamental mechanism that allows processes to communicate data with each other. One of the ways to achieve Inter Process Communication is through pipes. It is nothing but a mechanism using which the output of a single process is directed into the input of another process.

Advantages:

1. Simplicity: Pipes are a simple and straightforward way for processes to communicate with each other.
2. Efficiency: Pipes are an efficient way for processes to communicate, as they can transfer data quickly and with minimal overhead.
3. Reliability: Pipes are a reliable way for processes to communicate, as they can detect errors in data transmission and ensure that data is delivered correctly.
4. Flexibility: Pipes can be used to implement various communication protocols, including one-way and two-way communication.

Disadvantages:

1. Limited capacity: Pipes have a limited capacity, which can limit the amount of data that can be transferred between processes at once.
2. Unidirectional: In a unidirectional pipe, only one process can send data at a time, which can be a disadvantage in some situations.
3. Synchronization: In a bidirectional pipe, processes must be synchronized to ensure that data is transmitted in the correct order.
4. Limited scalability: Pipes are limited to communication between a small number of processes on the same computer, which can be a disadvantage in large-scale distributed systems

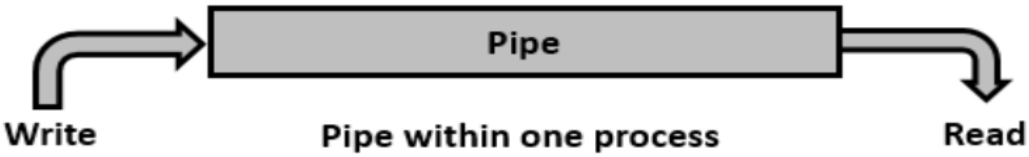
Pipes can be categorized into two types: unnamed (or anonymous) pipes and named pipes.

Unnamed Pipes or Traditional Pipes

- Unnamed pipes provide a way for processes to communicate in a unidirectional fashion, meaning data flows in only one direction. Unnamed pipes are used when communication needs to occur between a parent and its child process. The lifetime of an unnamed pipe is usually tied to the duration of the processes using it; it does not persist after the processes .

A usual pipe file has an input end and an output end. We can read from the output end and write into a pipe from the input end. There is an array of two pointers present in a pipe descriptor. One

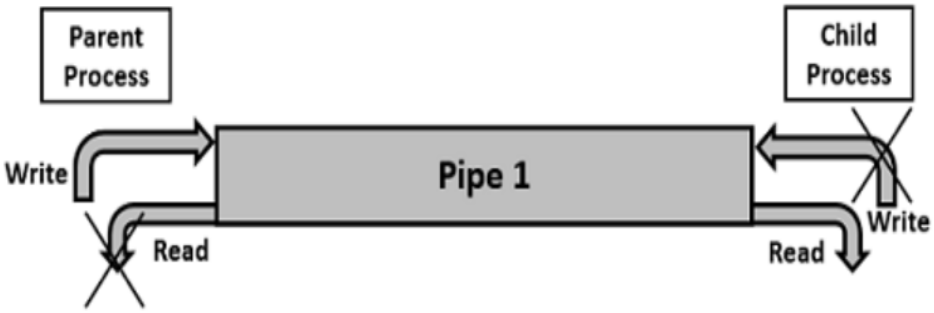
pointer is for storing the input end of the pipe and the other pointer is for storing the output end of the pipe.



Pipe within one process

Let us assume that two processes, Process One and Process Two, want to communicate with each other using pipes.

- The first Process should keep its write end of the pipe open and the read end closed.
- Process Two should keep its read end open and close its write end. On successful creation, a pipe is given a fixed size in bytes.



- If the pipe is not full and some process wants to write into the pipe, the write request is executed spontaneously. However, if the pipe is full when the process wants to write into the pipe, it is blocked until the state of the pipe changes.
- The same things happen if a reading process tries to read more bytes than what is currently in the pipe and gets blocked. On the other hand, the reading process is executed without any issues if it tries to read-only sufficient bytes that are there in the pipe. Only a single process can access a pipe at any particular point in time.

System Calls

- **int pipe (int fd[2]);** The input parameter is an array of two file descriptors fd[0] and fd[1]. A file descriptor is in fact an integer value. The system call returns a –1 in case of a failure. If the call is successful, it will return two integer values which are file descriptors fd[0]&fd[1]. In pipes, fd[0] and fd[1] are used for reading and writing, respectively.
- **Write(fd,buf,count)**

- **fd**: The file descriptor to which data is to be written. When using unnamed pipes, this is typically the write-end of the pipe.
- **buf**: A pointer to the buffer containing the data to be written.
- **count**: The number of bytes to write from the buffer.
- **Return Value**
- On success, **write()** returns the number of bytes written. This number may be less than **count** if there is insufficient space available on the pipe.
- On error, -1 is returned, and **errno** is set to indicate the error.

Read(fd,buf,count):

- **fd**: The file descriptor from which data is read. For unnamed pipes, this is typically the read end.
- **buf**: A pointer to the buffer where the read data should be stored.
- **count**: The maximum number of bytes to read.
- **Return Value**
- On successful completion, **read()** returns the number of bytes actually read and placed in the buffer. This value might be less than the number requested for several reasons, such as if fewer bytes are actually available at the moment in the pipe buffer.
- A return value of 0 indicates end-of-file (EOF); for a pipe, this means that no more data can be read because all file descriptors referring to the write end of the pipe have been closed.
- On error, -1 is returned, and **errno** is set to indicate the error.

Sample Program

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
int fd[2],n;
char buffer[100];
pid_t p;
pipe(fd); //creates a unidirectional pipe with two end fd[0] and fd[1]
p=fork();
if(p>0) //parent
{
Close(fd[0]);
printf("Parent Passing value to child\n");
write(fd[1],"hello\n",6); //fd[1] is the write end of the pipe
wait();
```

```
}
else // child
{
close(fd[1]);
printf("Child printing received value\n");
n=read(fd[0],buffer,100); //fd[0] is the read end of the pipe
write(1,buffer,n);
}
}
```

FIFO

- In computing, a named pipe (also known as a **FIFO**) is one of the methods for inter-process communication. It is an extension to the traditional pipe concept on Unix.
- A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
- Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
- A FIFO special file is entered into the filesystem by calling *mkfifo()* in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.
- The entire IPC process will consist of three programs:

Program1: to create a named pipe

Program2: process that will write into the pipe (sender process)

Program3: process that will receive data from pipe (receiver process)

//Program1: to create a named pipe

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
int main()
    {
        int res;
        res = mkfifo("fifo1",0777); //creates a named pipe with the name fifo1
        printf("named pipe created\n");
    }
        //Now compile and run this program.
```

//**Program2: Writing to a fifo/named pipe (sender.c**)

```
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>
int main()
{
        int res,n;
        res=open("fifo1",O_WRONLY);
        write(res,"Message",7);
        printf("Sender Process %d sent the data\n",getpid());
}
```

- //**Program 3**: Reading from the named pipe (receiver.c)
- #include<unistd.h>
- #include<stdio.h>

```
#include<fcntl.h>
int main()
{
        int res,n;
        char buffer[100];
        res=open("fifo1",O_RDONLY);
        n=read(res,buffer,100);
        printf("Reader process %d started\n",getpid());
        printf("Data received by receiver %d is: %s\n",getpid(), buffer);
}
```

The major differences between named and unnamed pipes are:-

1. As suggested by their names, a named type has a specific name which can be given to it by the user. Named pipe if referred through this name only by the reader and writer. All instances of a named pipe share the same pipe name.On the other hand, unnamed pipes is not given a name. It is accessible through two file descriptors that are created through the function pipe(fd[2]), where fd[1] signifies the write file descriptor, and fd[0] describes the read file descriptor.

2. An unnamed pipe is only used for communication between a child and it's parent process, while a named pipe can be used for communication between two unnamed process as well. Processes of different ancestry can share data through a named pipe.

3. A named pipe exists in the file system. After input-output has been performed by the sharing processes, the pipe still exists in the file system independently of the process, and can be used for communication between some other processes.On the other hand, an unnamed pipe vanishes as soon as it is closed, or one of the process (parent or child) completes execution.

4. Named pipes can be used to provide communication between processes on the same computer or between processes on different computers across a network, as in case of a distributed system. Unnamed pipes are always local; they cannot be used for communication over a network.
5. A Named pipe can have multiple process communicating through it, like multiple clients connected to one server. On the other hand, an unnamed pipe is a one-way pipe that typically transfers data between a parent process and a child process.

Semaphores: semget, semop, semctl,

Linux provides a **Semaphore API**, which has a set of functions that allow developers to work with semaphores in linux efficiently with their programs. Let us explore the key functions of the API,

semget

The **semget()** function is used to create a **new semaphore** or get an existing semaphore's identifier.

Syntax:

int semget(key_t key, int num_sems, int sem_flags);

It takes three arguments:

- **key:**

Used to identify the semaphore set. It can be either IPC_PRIVATE (to create a new semaphore set) or an identifier used to get an existing semaphore set.

- **num_sems:**

Number of semaphores to create in the semaphore set. Typically, this is set to 1 for single semaphores.

- **sem_flags:**

Permission flags for the semaphore linux set, specified as an octal number. It determines the read, write, and execute permissions for the semaphore set.

It returns the linux semaphore identifier (semid) if successful or -1 on failure.

semctl()

The **semctl()** function is used to control semaphore behavior.

Syntax:

int semctl(int semid, int sem_num, int command, ...);

It takes three arguments:

- **semid:**

  Semaphore linux identifier representing the semaphore set on which the operation will be performed.
- **sem_num:**

  Semaphore number within the semaphore set on which the operation will be performed. For single semaphores, this is usually set to 0.
- **command:**

  Specifies the operation to be performed on the Semaphore linux set. The most used values are:
  - **IPC_STAT:**

    Get the semaphore set status and store it in a struct semid_ds.
  - **IPC_SET:**

    Set the semaphore set status using the information from a struct semid_ds.
  - **IPC_RMID:**

    Remove the semaphore set from the system.
  - **GETVAL:**

    Get the value of the semaphore.
  - **GETPID:**

    Get the process ID of the last operation that modified the semaphore.

- **Optional Argument:**

Depending on the command used, there might be an optional fourth argument, which is a pointer to a union semun. The union semun allows specifying additional parameters for some commands, such as the new value for SETVAL or an array of semaphore values for SETALL.

This function returns the result of the specified command, depending on the operation performed. On failure, it returns -1 to indicate an error.

semop()

The **semop()** function is used to perform operations, such as locking and unlocking semaphores in linux.

Syntax:

int semop(int semid, struct sembuf *sops, size_t num_sops);

It takes three arguments:

- **semid:** The linux semaphore identifier.

- **sops:** Pointer to an array of struct sembuf, specifying the Semaphore linux operations to be performed.
- **num_sops:** Number of elements in the sops array, indicating the number of semaphore operations to be performed.

The struct sembuf data structure contains three fields:

- **sem_num:** Semaphore number within the semaphore set. For single semaphores, this is usually set to 0.
- **sem_op:** The operation to be performed on the semaphore, which can be a positive or negative value.
- **sem_flg:** Flags that control the behavior of the operation, such as SEM_UNDO (automatic semaphore release on process termination) or IPC_NOWAIT (return immediately if the semaphore is not available).

This function returns 0 if the semaphore operations are successful or -1, on error.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
int main() {
   int semaphore = semget(IPC_PRIVATE, 1, 0666 | IPC_CREAT);
   struct sembuf sem_op;

   if (semaphore < 0) {
      perror("Semaphore creation failed");
      exit(1);
   }
   printf("Semaphore created successfully.\n");
   sem_op.sem_num = 0;
   sem_op.sem_op = -1; // Lock the semaphore
   sem_op.sem_flg = 0;
   if (semop(semaphore, &sem_op, 1) == -1) {
      perror("Failed to lock semaphore");
      exit(1);
   }
   printf("Semaphore locked.\n");
   // Critical Section - Access shared resources here
   printf("In critical section.\n");
   sem_op.sem_op = 1; // Unlock the semaphore
   if (semop(semaphore, &sem_op, 1) == -1) {
      perror("Failed to unlock semaphore");
      exit(1);
```

```
    }

    printf("Semaphore unlocked.\n");
    if (semctl(semaphore, 0, IPC_RMID) == -1) {
        perror("Failed to remove semaphore");
        exit(1);
    }
    printf("Semaphore removed.\n");
    return 0;
}
```