# IPC and Pipes 1: Introduction and Basic Usage

1. **Introduction and Basic Usage**
   - **IPC mechanisms**
   - **pipes and pipe() call**
   - **pipe characteristics**
   - **standard usage pattern**
2. Usage Details
3. FIFOs

# Interprocess Communication

**Interprocess Communication** (**IPC**) refers to mechanisms for sending data from one process to another.

Linux/UNIX supports a number of IPC mechanisms:

- pipes and FIFOs
- sockets
- shared memory
- message queues
- signals

(Process **synchronization** mechanisms like **semaphores** can also be considered as IPC in a very broad sense.)

# IPC (contd.)

IPC mechanisms can differ in a number of dimensions:

- data transfer direction:
  **unidirectional** vs. **bidirectional**
- data formatting:
  **stream** (of bytes) vs. separate **messages**
- process relationship requirements:
  **related** (*share a common ancestor*) vs. unrelated
- machine(s) hosting the processes:
  *same machine only* vs. different, **networked** machines
- data read order:
  always in write order vs. allow reading "**out-of-band**" (prioritized) data first
- able to transmit special information:
  e.g., file descriptor link

# Pipes

**Pipes** are the oldest and simplest IPC mechanism, but also the most used IPC mechanism.

A pipe is a **unidirectional**, **stream-oriented** IPC mechanism that can be used between **related processes** that are running on the **same machine**.

Most users are familiar with pipes from their use in shell **pipelines**.

Consider a command like: `grep printf lab1.c | wc -l`

This will cause the shell to create a pipe and two subprocesses, with the `grep` process sending its output to the `wc` process via the pipe.

# pipe() System Call

`pipe()` is the system call to create a pipe:
`int pipe(int pipefd[2])`

- `pipefd` is a two-element `int` array into which *file descriptors* for the *read and write ends* of the pipe will be placed
- `pipefd[0]` is the *read end* FD
- `pipefd[1]` is the *write end* FD
- these FDs can be used with `read()` and `write()` to send and receive information between processes
- returns 0 on success, -1 on error

# Related Processes on Same Machine

Logically, using a pipe to send data from one process to another is equivalent to having one process write data to a *file* and the other read data from that file.

A pipe is a **kernel buffer**, however, so pipe I/O is much faster than file I/O.

Unlike a file, however, basic, *unnamed* pipes do *not* have an entry in the filesystem.

This means that the only way for a second process to get access to a pipe created by another process is by *inheriting the pipe's file descriptors* (e.g., across a `fork()` call).

Thus, pipes are limited to IPC among processes that are *"related"* (*share a common ancestor*), such as parent-child or siblings.

# Standard Usage Pattern

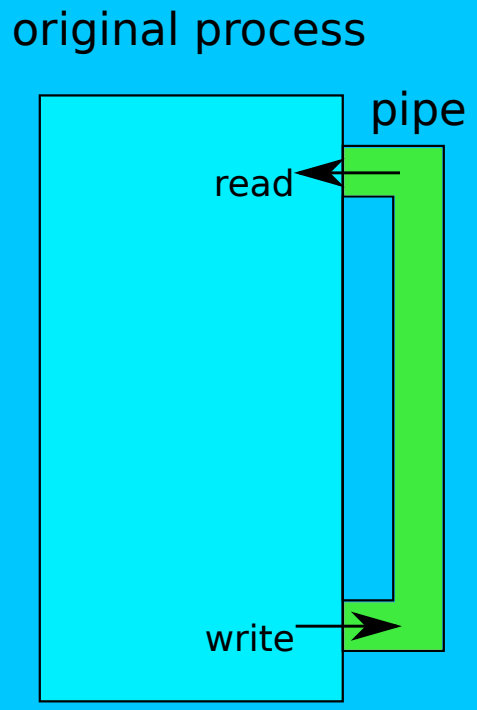
Becuase pipe file descriptors must be inherited, this virtually always means that `pipe()` will be called and only later will one or more `fork()` calls be made.

The child created by `fork()` will *inherit* the file descriptors for the read and write ends of the pipe (since it is a copy of its parent).

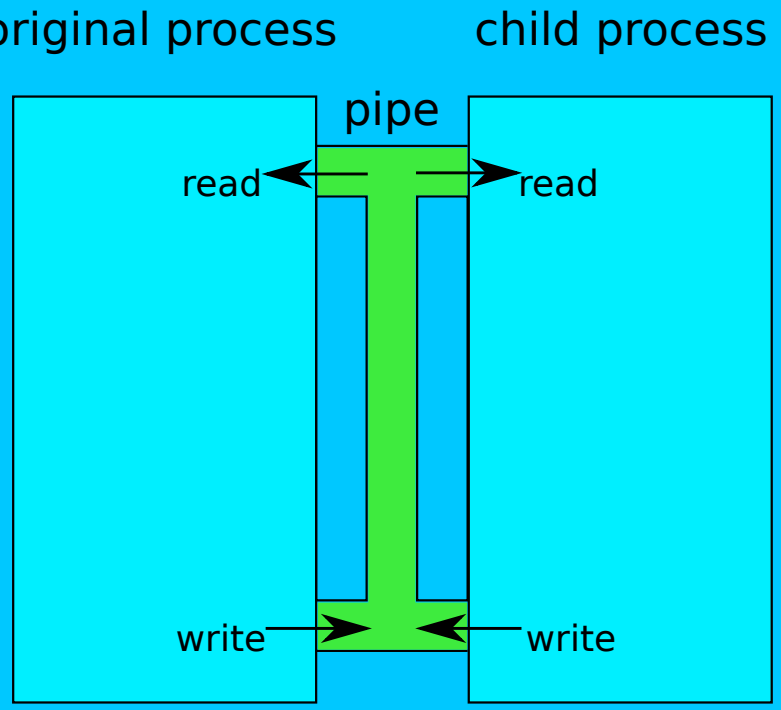
This will allow the parent and child processes to both have access to both ends of the pipe.

# Pipe Illustration

After `pipe()` call:




After later `fork()` call:



# Unidirectional Communication

Pipes are primarily intended for **unidirectional** communication between exactly *two* processes.

Consider trying to use a pipe for bidirectional communication: if P1 and then P2 each write data intended for the other into the pipe, should P1 read from the pipe first, it would get the message it had written, which was intended for P2.

Pipes are virtually always used with *only a single process reading from the pipe*, because otherwise there would be no way to control which process received the next "message."

# Unidirectional Communication (contd.)

Pipes are usually used with a *single writer*, but sometimes have multiple processes write into the pipe, e.g., several clients sending requests to a single server process.

In this case, the communication *protocol* may have to enable the server to distinguish which client each "message" originated from (e.g., by including client PID or "name" in a message *header*).

"Messages" must also be short enough to ensure they are written into the pipe with a single `write()`.

**Sockets** are generally a better choice for IPC in **client-server** systems, since they support separate connections for each client and bidirectional communication.

# Standard Usage Pattern (contd.)

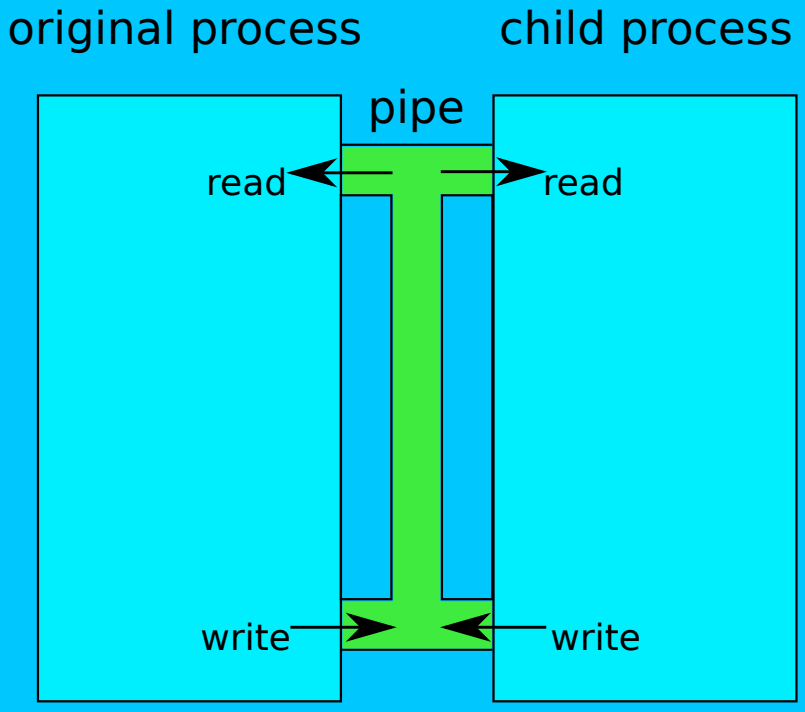
If pipes work best with one reader process and one writer process, how is this enforced after pipe FDs are inherited?

Generally, one pipe end is closed in each process, forming a unidirectional communication channel from one process to the other.

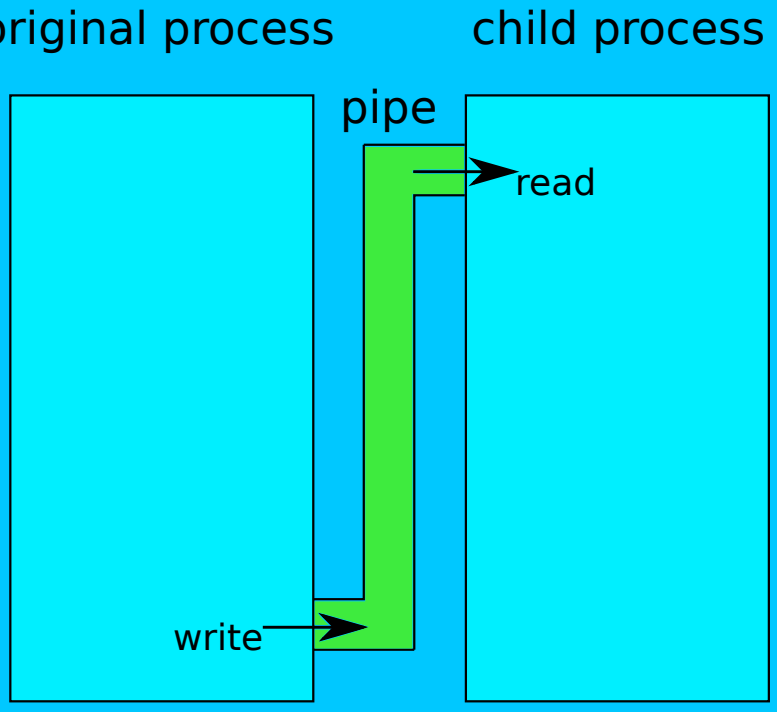
This also makes it possible for the reading process to determine when no more data will be coming—see below.

# Pipe Illustration (contd.)

Two readers & writers can cause problems:




After closing pipe ends, have unidirectional channel:



# pipe() Usage Example

Parent sends one message to child via pipe:

```
//Create pipe:  (must do before fork() so FDs inherited by child)
int pipefd[2];  //array to hold pipe FDs
pipe(pipefd);

//Create child and do its reading & printing functionality:
if (fork() == 0) {
  //In child:
  close(pipefd[1]);                 //unused in child, close
  char buff[10];                    //storage for message
  read(pipefd[0],buff,10);          //read message from parent
  buff[11] = '\0';                  //turn buff into a string
  printf("pipe read: %s\n",buff);   //print message to stdout
  exit(EXIT_SUCCESS); }             //terminate child

//In parent (after fork()), so do its writing functionality:
close(pipefd[0]);                   //unused in parent, close
write(pipefd[1],"testing #1",10);   //write message to child
wait(NULL);                         //wait for child to terminate, collect
```