System and Device Programming

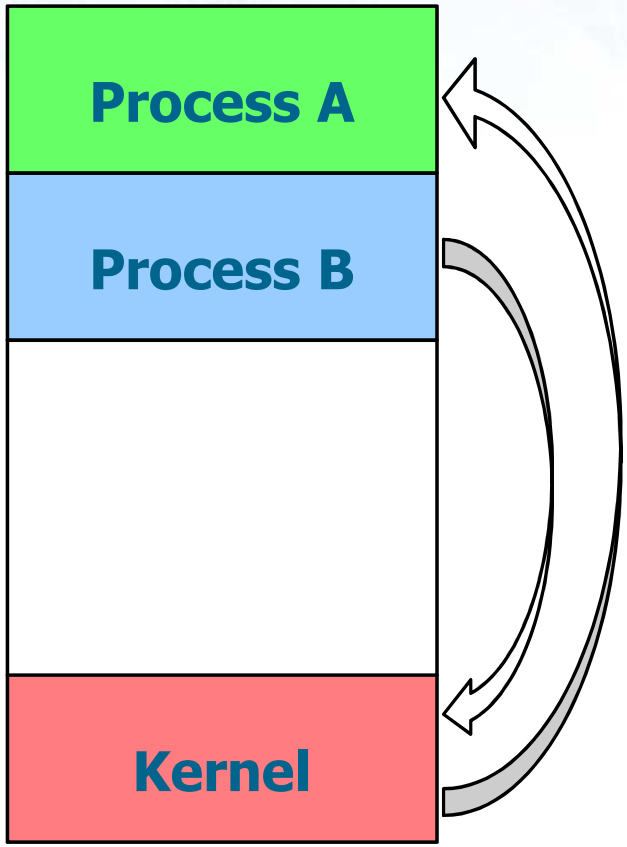
UNIX IPC

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Inter-Process Communication

- Information sharing among processes is referred to as **IPC** or **I**nter**P**rocess **C**ommunication
- The main communication models are based on
  - Message exchange
  - Shared memory

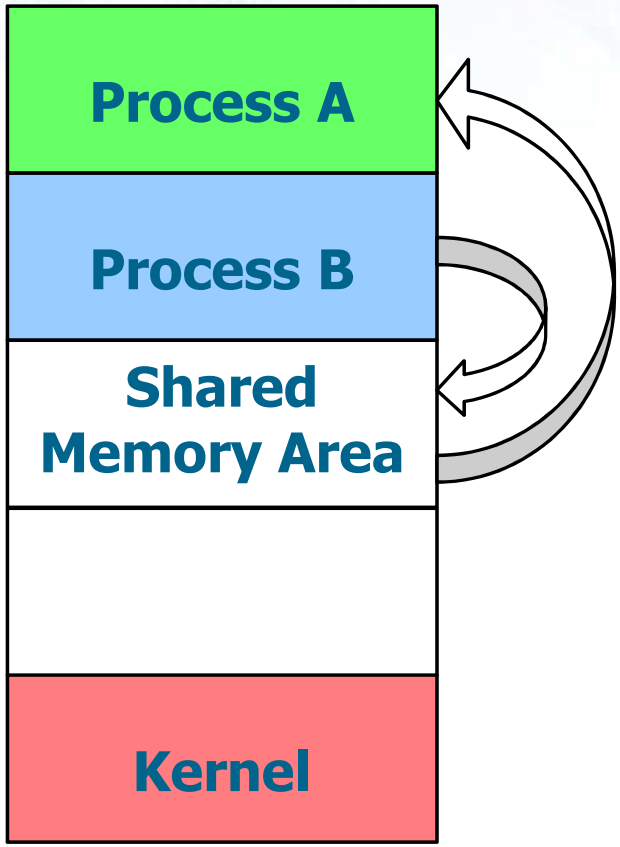
Communication models



- Message exchange
  - Communication takes place through the exchange of messages
  - Need to setup of a communication channel
  - Useful for exchanging limited amounts of data
  - Uses system calls
    - Require kernel intervention
    - Introduce overhead

Pipe: Direct, Synchronous, Limited capacity

Communication models



- Shared memory
  - Used for sharing a large amount of data
  - Based on sharing a memory area and writing of data in this area
  - Most common methods
    - **File** sharing
      - Sharing the name or the file pointer or descriptor before `fork/exec`
    - **Mapped file** in memory
      - A file mapped in memory associates a shared memory region to a file

Communication channels

- UNIX makes available
  - **Half-duplex pipes**
  - FIFOs
  - Full-duplex pipes
  - Named full-duplex pipes
  - Message queues
  - Semaphores
  - Sockets
  - STREAMS

Extensions of the original half-duplex pipes

For process synchronization

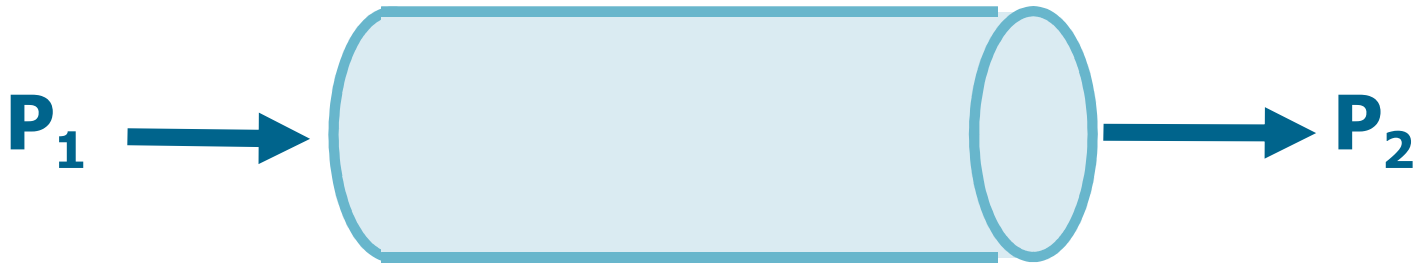
Network process communication. Each process is identified by a socket to which it is associated a network address

Not all the types of communication are supported by all the UNIX versions

Used starting from UNIX System V

Pipes

- Pipes are the oldest form of communication in UNIX-like operating systems
- Allow creating a **data stream among** processes
  - The user interface to a pipe is similar to file access
  - A pipe is accessed by means of two descriptors (integers), one for each end of the pipe
  - A process (P_1) writes to an end of the pipe, another process (P_2) reads from the other end



Pipes

- Historically, they have been
  - **Half-duplex**
    - Data can flow in both directions (from P_1 to P_2 or from P_2 to P_1), but **not** at the same time
    - Full-duplex models have been proposed more recently, but they have limited portability
  - A pipe can be used for communication among a parent and its childs, or among processes with a **common ancestor**
    - The file descriptor must be common, therefore the processes must have a common ancestor

Simplex: Mono-directional
Half-Duplex: One-way, or bidirectional, but alternate (walkie-talkie)
Full-Duplex: Bidirectional (telephone)

System call pipe

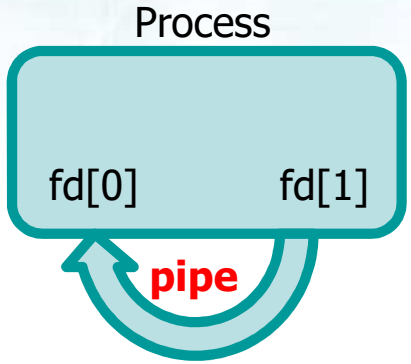
```
#include <unistd.h>

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

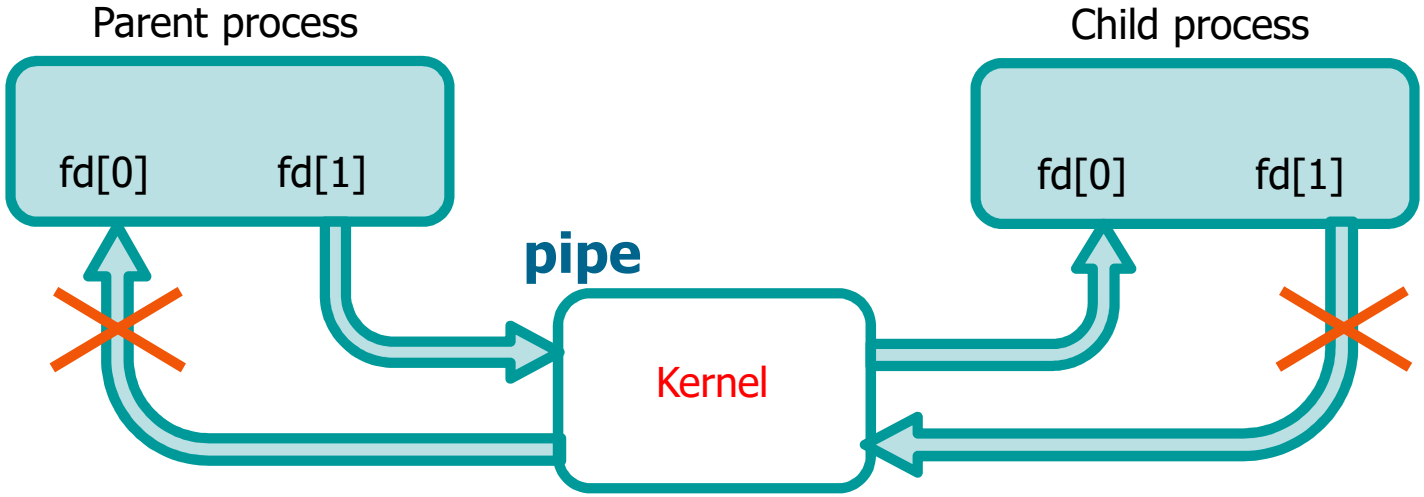
Return value:
0, on success
-1, on failure

- The system call **pipe** creates a pipe
  - A pipe allows a parent and a child to communicate
- It returns **two** file descriptors in vector **fileDescr**
  - fileDescr[0]: Typically used for reading
  - fileDesrc[1]: Typically used for writing
    - The input stream written on fileDescr[1] corresponds to the output stream read on fileDescr[0]

System call pipe



- Methodology
  - A process creates pipe
  - **Then** it performs a fork
  - The child process **inherits** the file descriptors
  - One process writes into the pipe, the other reads from the pipe
  - The unused descriptor should be closed



Pipe I/O

- The descriptor of the pipe is an integer number
- R/W on pipes do not differ to R/W on files
  - Use **read** and **write** system calls
    - Read blocks the process if the pipe is empty
    - Write blocks the process if the pipe is full
  - It is possible to have more than one reader and writer on a pipe, but
    - The standard case is to have a single writer and a single reader
    - Data can be interlaced using more than one writer
    - Using more readers, it is undetermined which reader will read the next data from the pipe

Example

Use a pipe to transfer 1 character from the father to the child

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

int main () {
  int n;
  int file[2];
  char cW = 'x';
  char cR;
  pid_t pid;
  if (pipe(file) == 0) {
    pid = fork ();
    if (pid == -1) {
      fprintf(stderr, "Fork failure");
      exit(EXIT_FAILURE);
    }
```

Firstly, create the pipe

Then, fork the process

Example

```
  if (pid == 0) {
      // Child reads
      close (file[1]);
      n = read (file[0], &cR, 1);
      printf ("Read %d bytes: %c\n", n, cR);
      exit(EXIT_SUCCESS);
    } else {
      // Parent writes
      close (file[0]);
      n = write (file[1], &cW, 1);
      printf ("Wrote %d bytes: %c\n", n, cW);
    }
  }
  exit(EXIT_SUCCESS);
}
```

Close unused end (good practice)

Child reads

Parent writes

The two process synchronize because read and write are possibly blocking

More complex data communication requires a communication **protocol**