

# Lab 2.4. Pipes

## Objectives

Pipes provide a simple and effective mechanism for process communication in the same system. In this lab, we will see the commands and system interface for pipe management and the typical communication patterns.

## Contents


- Environment Preparation
- Named Pipes
- Synchronous I/O Multiplexing
- Unnamed Pipes

## Environment Preparation

This lab only requires the development tools and environment.

## Named Pipes

Named pipes are a FIFO communication mechanism, useful for non-related processes. The calls to manage named pipes are the same as that for regular files (write, read, open...). Consult the man page of `fifo(7)`.

**Exercise 1.** Use command `mkfifo` to create a pipe. Use the file system tools (`stat`, `ls...`) to get its properties. Check its behaviour using tools to write and read files (e.g. `echo`, `cat`, `tee...`). 

**Exercise 2.** Write a program that opens the previous named pipe in write-only mode and writes in it the first argument of the program. In other terminal, read the pipe using an appropriate command.

## Synchronous I/O Multiplexing

It is common that a process reads from or writes to several I/O channels. Function `select()` allow multiplexing several I/O operations over multiple streams.

**Exercise 1.** Create another named pipe. Write a program that waits until there is data ready to be read in any of the two pipes. The program must show the pipe from where it read the data as well as the data read. Considerations:

- To optimize the operations, use a read buffer (e.g. 256 bytes).
- Use `read()` to read from the pipe and appropriately manage the length of characters read.
- Normally, opening the pipe for reading will block until it is opened also for writing (e.g. with `echo 1 > pipe`). To avoid that, use option `O_NONBLOCK` in `open()`.
- When the writer finalizes and closes the pipe, `select()` will consider that the descriptor is always ready for reading (to detect the end-of-file condition) and will not block. In this case, the pipe must be closed and opened again.

## Unnamed Pipes

Unnamed pipes are directly managed by the system kernel and are an efficient unidirectional communication mechanism for related process (parent-child). Pipe identifiers are inherited in `fork()` calls. In this case, there isn't a known path in the file system.

**Exercise 1.** Write a program that emulates the behaviour of the shell when executing a sentence like `command1 argument 1 | command2 argument2`. The program will create an unnamed pipe and will create a process. Then:

- The parent process will redirect the standard output to the write end of the pipe and will execute `command1 argument1`.
- The child process will redirect the standard input to the read end of the pipe and will execute `command2 argument2`.

Check the operation with a sentence like `./exercise1 echo 12345 wc -c`

**Note:** Before executing the corresponding command, each process must close any unneeded file descriptor.

**Exercise 2.** For bidirectional communication, it is necessary to create two pipes, one for each direction: `p_c` and `c_p`. Write a program that implements the stop-and-wait synchronization mechanism:

- The parent process will read from the standard input (terminal) and will send the message to the child process by writing to pipe `p_c`. Then, it will block waiting for the child's confirmation in the pipe `c_p`.
- Once the child reads a message from pipe `p_c`, it will write it to the standard output (terminal), will wait for 1 second, and then will send the character 'r' to the parent process to indicate that it is ready by writing to pipe `c_p`. After 10 messages, it will send the character 'q' to indicate that the parent process should finalize.