Operating systems

Interprocess communication (IPC)
PIPE and FIFO

Samuele Germiniani

samuele.germiniani@univr.it

University of Verona
Department of Engineering for Innovation Medicine

2023/2024

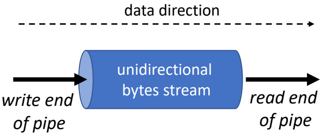
Table of Contents

PIPEs

PIPEs

Fundamental concepts

Fundamental concepts (1/2)



A *PIPE* is a byte stream (technically speaking, it is a buffer in kernel memory), which allows processes to exchange bytes.
A *PIPE* has the following properties:

- it is <u>unidirectional</u>. Data travels only in one direction through a PIPE. One end of the PIPE is used for writing, the other one for reading;
- data passes through the PIPE <u>sequentially</u>. Bytes are read from a PIPE in exactly the order they were written;
- no concept of <u>messages, or message boundaries</u>. The process reading from a PIPE can read blocks of data of any size, regardless of the size of blocks written by the writing process.

Fundamental concepts (2/2)

- Attempts to read from an empty PIPE blocks the reader until, either at least one byte has been written to the PIPE, or a no-terminating signal occurs (errno EINTR).
- If the write-end of a PIPE is closed, then a process reading from the PIPE will see end-of-file once it has read all remaining data in the PIPE.
- A write is blocked until, either sufficient space is available to complete the operation atomically[1], or a no-terminating signal occurs (errno EINTR).
- Writes of data blocks larger than PIPE_BUF[2] bytes may be broken into segments of arbitrary size (which may be smaller than PIPE_BUF bytes).

[1] On Linux, pipe capacity is 65536 bytes

[2] On Linux, PIPE_BUF has the value 4096 bytes

PIPEs

Creating and using PIPEs

Creating and using PIPEs (1/3)

The *pipe* system call creates a new PIPE.

```
#include <unistd.h>

// Returns 0 on success, or -1 on error
int pipe(int filedes[2]);
```

A successful call to *pipe* returns two open file descriptors in the array *filedes*.

- filedes[0] stores the *read-end* of the PIPE.
- filedes[1] stores the *write-end* of the PIPE.

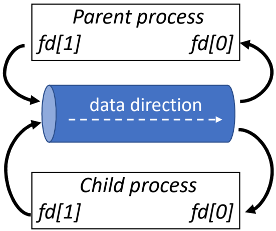
As with any file descriptor, we can use the *read* and *write* system calls to perform I/O on the PIPE.

Normally, we use a PIPE to allow communication among related processes. To connect two processes using a PIPE, we follow the *pipe* call with a call to *fork*.

Creating and using PIPEs (2/3)

```
int fd[2];
// checking if PIPE successed
if (pipe(fd) == -1)
    errExit("PIPE");
// Create a child process
switch(fork()) {
    case -1:
        errExit("fork");
    case 0: // Child
        //...child reads from
             PIPE
        // (next slide)
        break;
    default: // Parent
        //...parent writes to
             PIPE
        // (next slide)
        break;
}
```

1. *pipe(...)* creates a new PIPE.
   fd[0] is the read-end of the PIPE.
   fd[1] is the write-end of the PIPE.
2. *fork()* creates a child process, which inherits the file descriptor table of the parent process.



Creating and using PIPEs (3/3)

case 0: // child reads from PIPE

```c
char buf[SIZE];
ssize_t nBys;

// close unused write-end
if (close(fd[1]) == -1)
    errExit("close - child");
// reading from the PIPE
nBys = read(fd[0], buf, SIZE);
// 0: end-of-file, -1: failure
if (nBys > 0) {
    buf[nBys] = '\0';
    printf("%s\n", buf);
}
// close read-end of PIPE
if (close(fd[0]) == -1)
    errExit("close - child");
```

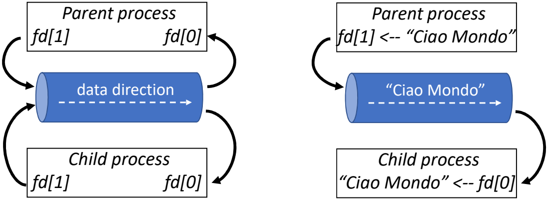
default: // parent writes to PIPE

```c
char buf[] = "Ciao Mondo\n";
ssize_t nBys;

// close unused read-end
if (close(fd[0]) == -1)
    errExit("close - parent");
// write to the PIPE
nBys = write(fd[1], buf, strlen(buf));
// checkig if write successed
if (nBys != strlen(buf)) {
    errExit("write - parent");
}

// close write-end of PIPE
if (close(fd[1]) == -1)
    errExit("close - parent");
```

Good and bad practice



After fork()

After closing unused descriptors

Why should we close the unused PIPE file descriptor?

FIFOs (named PIPEs)

FIFOs (named PIPEs)

Fundamental concepts

Fundamental concepts

A *FIFO* is a byte stream (technically speaking, it is a buffer in kernel memory), which allows processes to exchange bytes.
Semantically, a *FIFO* is similar to a *PIPE*.

The principal difference between *PIPEs* and *FIFOs* is that a *FIFO* has a name within the file system, and is opened and deleted in the same way as a regular file. This allows a *FIFO* to be used for communication between unrelated processes.

Just as with PIPEs, a *FIFO* has a write-end and a read-end, and data is read from the FIFO in the same order as it is written.

FIFOs (named PIPEs)

Creating, opening, and using FIFOs

Creating a FIFO

The *mkfifo* system call creates a new *FIFO*.

```
#include <unistd.h>

// Returns 0 on success, or -1 on error
int mkfifo(const char *pathname, mode_t mode);
```

The *pathname* parameter specifies where the *FIFO* is created. As a normal file, the *mode* parameter specifies the permissions for the *FIFO* (see chapter file system, system call *open*).

Once a *FIFO* has been created, any process can open it.

Opening a FIFO (1/2)

The *open* system call open a *FIFO*.

```c
#include <unistd.h>

// Returns file descriptor on success, or -1 on error.
int open(const char *pathname, int flags);
```

The *pathname* parameter specifies the location of the *FIFO* in the file system. The *flags* argument is a bit mask of one of the following constants that specifies the access mode for the *FIFO*.

Flag	Description
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only

Opening a FIFO (2/2)

The only sensible use of a *FIFO* is to have a reading process and a writing process on each end.

By default, opening a *FIFO* for reading (O_RDONLY flag) blocks until another process opens the *FIFO* for writing (O_WRONLY flag). Conversely, opening the *FIFO* for writing blocks until another process opens the FIFO for reading. In other words, opening a *FIFO* synchronizes the reading and writing processes.

If the opposite end of a FIFO is already open (perhaps because a pair of processes have already opened each end of the FIFO), then *open* succeeds immediately.

Creating and using a FIFO

Receiver

```c
char *fname = "/tmp/myfifo";
int res = mkfifo(fname, S_IRUSR|S_IWUSR);
// Opening for reading only
int fd = open(fname, O_RDONLY);

// reading bytes from fifo
char buffer[LEN];
read(fd, buffer, LEN);

// Printing buffer on stdout
printf("%s\n", buffer);

// closing the fifo
close(fd);

// Removing FIFO
unlink(fname);
```

Sender

```c
char *fname = "/tmp/myfifo";

// Opening for wringing only
int fd = open(fname, O_WRONLY);

//reading a str. (no spaces)
char buffer[LEN];
printf("Give me a string: ");
scanf("%s", buffer);

// writing the string on fifo
write(fd, buffer, strlen(buffer));

// closing the fifo
close(fd);
```

Statements checking errors were omitted due to lack of space.