

Inter-Process Communication



Leandro Proença

Posted on 13 de jul. de 2022 • Updated on 5 de dez. de 2022



5



3

Inter-process communication: files

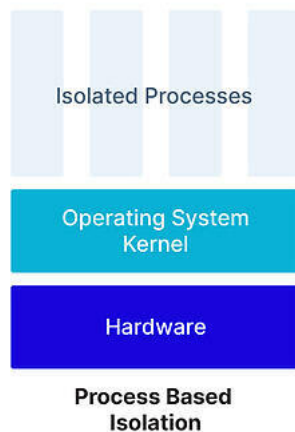
#unix #linux

Computers, OS and networking (3 Part Series)

- 1 A brief history of modern computers, multitasking and operating ...
- 2 **Inter-process communication: files**
- 3 Inter-process communication: pipes

[In the previous post](#) we learned that computers evolved to tackle **concurrency**, the reason why operating systems were created in order to share computer resources with running computer programs.

The main concurrency unit primitive of an operating system is a **process**. Therefore, OS processes are isolated, which means they have their *own memory space*.



However, eventually we *need* two different processes to communicate each other, either sending or consuming relevant data.

Because of their nature of isolation, in order to communicate, OS processes need to share a common **communication channel**.

That's where we get introduced to a technique called [IPC](#), or **Inter-Process communication**.

Disclaimer: in this article I'll focus on explaining for UNIX-like operating systems. But all the theory and elementary aspects described here, can be used somehow in other operating systems too.

IPC approaches

Historically, as computers, operating systems and runtimes evolved, approaches for IPC have been developed, namely:

- Files
- Pipes
- Sockets

...among others.

Wait...what is a **file**? Or a **socket**? *Pipes?* **No idea!**. Trust me, it's not that hard to understand.

But in this very article, we're going to focus in the **files** then **file descriptors**, leaving pipes and sockets for the next upcoming posts.

How the computer data is stored

The *computer storage* is the physical location in a computer where data is *stored*.

Eventually, we need to access some specific data, right?

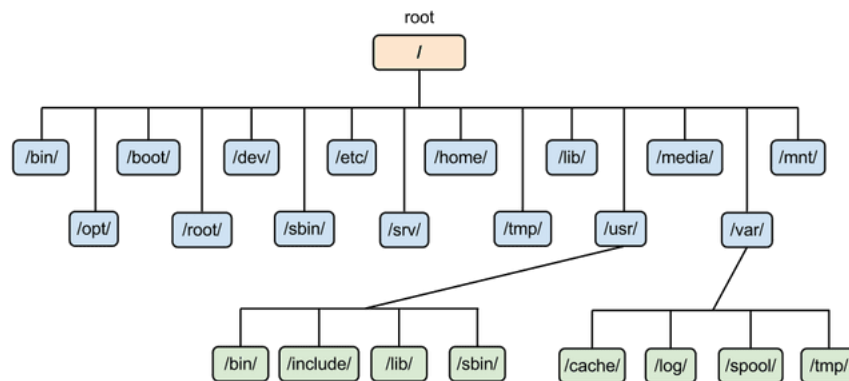
How can we locate a *specific* data in the storage?

It's easy. We need to know the range where the data **starts** and where the data **ends**. Give to this *range* a name, then you have a **file**.

Group all these files into a single system composed by *dictionaries*, and we have a **filesystem**.

Filesystem

Filesystem (or **fs**) is a place in the storage where **all** our data are stored in the computer.



Knowing that two different processes are isolated in their own *memory space* **and** they need to communicate to each other, we have no choices left for IPC but using the **fs**.

Communication channels

In order for processes to use the filesystem, they need to establish a **communication channel**. Such a channel uses a *special file* in the **fs**, called **file descriptor** (or *fd*).

File descriptors

File descriptor is a *special file*, a unique identifier stored in a *file descriptor table* pointing to other files or *I/O resources*.

Standard streams

Every UNIX process comes by default with 3 *standard* communication channels, also known as **standard streams** represented by *file descriptors*:

- `fd 0`: standard output (**STDIN**)
- `fd 1`: standard input (**STDOUT**)
- `fd 2`: standard error (**STDERR**)

Let's see them in action.

```
$ echo 'Hello'

Hello
```

It seems pretty naive, right? But let's understand what's happening under the hood:

- the program `echo` sends a message to the **STDOUT** `0` and finishes
- **STDOUT** is my computer monitor

Easy as that. Next, we'll see an example using the STDIN:

```
$ base64 # waits for data from STDIN

## Then I type 'leandro', followed by the ENTER key (CR),
#### followed by `CTRL+D` to exit from the STDIN interface

bGVhbmRybwo=
```

- the program `base64` waits for data from the **STDIN** `0`
- **STDIN** is my keyboard
- I type my name, followed by *carriage return**, followed by `CTRL+D`
- the program sends the result to the **STDOUT** `1` and finishes
- **STDOUT** is my computer monitor

Look how some programs like **base64** can interact with both streams (STDIN and STDOUT).

How about the **STDERR**?

```
$ base64 hahahaha

/usr/bin/base64: hahahaha: No such file or directory
```

As it waits for *STDIN*, the program itself raises an error to the **STDERR**, which by default in *most programs* is the **STDOUT**.

That's why we see the error in the screen monitor :)

Stream redirection

What if we wanted to redirect standard streams to another **fd**, or even **to another file**?

Yes, it is possible. We can use the operator `>` for *STDOUT/STDERR* and the operator `<` for *STDIN*. Sounds familiar?

Redirecting the **STDOUT** `1` to another file:

```
$ echo 'Hello' 1> message.txt
```

Then we can use the program `cat` to read the file:

```
$ cat message.txt

Hello
```

Needless to say that the program `cat` sends the content to the *STDOUT*!

Going back to the `base64` example, how could we redirect the **STDERR** to another file, instead of **STDOUT**? Yes, using `2> !`

```
$ base64 hahahaha 2> err.txt
```

Great, now let's "cat it":

```
$ cat err.txt

/usr/bin/base64: hahahaha: No such file or directory
```

Superb! We're almost there!

Oh, yes, I was almost forgetting. By default, the `fd` used when we redirect using only `>` is the `fd 1`, or STDOUT.

That's why we see many examples like this one:

```
$ echo 'leandro' > message.txt
$ echo 'leandro' 1> message.txt # it's the same!
```

We can even use multiple redirections at the same command:

```
$ echo 'leandro' > message.txt 2> errors.txt
```

Or, in case we want to *concentrate** errors and messages in the same place *but* the STDOUT:

```
echo 'some message' > out.log 2>&1
```

The `&1` means the current stream for the `fd 1`. In other words, we are redirecting STDERR to the same place where STDOUT **was redirected**.

We can redirect STDIN too. First, we send some message to a file:

```
$ echo 'some message' > out.txt
```

Then, in the `base64` program, we can redirect STDIN to reading directly from the file instead:

```
$ base64 0< out.txt

c29tZSBtZXNzYWdlCg==
```

UNIX allows us to remove the `0` because it's redundant, since the operator `<` is used only for *STDIN redirection*, right?

```
$ base64 < out.txt

c29tZSBtZXNzYWdlCg==
```

Okay, but how two different processes can send messages to each other using the *standard streams*?

Reading files

This is a naive approach, where a process, using stream redirection, *writes* to a file. Then another different process *reads* from this very file.

Process A:

```
echo 'hello from A!' > process_a_messages.txt
```

Then, from Process B:

```
cat process_a_messages.txt

hello from A!
```

Very naive at first, but if we analyse carefully, we can note that:

- a process called "A" using the program `echo`, sends a message to the redirected STDOUT
- another process called "B" using the program `cat` reads a message from the redirected STDIN, which is the file written by the process A

Making more sense? This conclusion is crucial:

A process's STDOUT is used to be the STDIN of another process

Wow! That's wonderful as it opens infinite possibilities to IPC, such as **UNIX pipes** which we'll see in the next post.

```
$ echo 'my precious' > rawcontent.txt
$ base64 < rawcontent.txt
```

Using other file descriptors

Apart from the standard streams, can we create more file descriptors and use them for IPC?

Yes. Let's first create a directory where we're gonna store our custom **fd's**:

```
$ mkdir /tmp/fd
```

Then, we prepare a custom file descriptor `42` that will be our *single-communication channel* between two processes. First, we open the `fd` for **writing** (STDOUT):

```
exec 42> /tmp/fd/42
```

Now, from the process "A", we send a message to the custom file descriptor using *stream redirection*:

```
$ echo 'message from the process A' >&42
```

As for the process reader, we have to open the `fd` for **reading** (STDIN):

```
$ exec 42< /tmp/fd/42
```

And from the process "B", we read the message from the *same* file descriptor using *stream redirection*:

```
$ base64 <&42
bWVzc2FnZSBmcm9tIHRobzSBwcm9jZjZlZEEK
```

We can't forget to close the `fd` for both reading and writing:

```
exec 42<&-
exec 42>&-
```

Yay! So much power using only **file descriptors** for IPC!

Conclusion

This article was a try to resume and explain the fundamentals behind using files as a primitive approach for inter-process communication.

In the upcoming articles we will see other approaches such as *pipes*, *sockets* and *message queues*. Stay tuned!

Computers, OS and networking (3 Part Series)

- 1 A brief history of modern computers, multitasking and operating ...
- 2 **Inter-process communication: files**
- 3 Inter-process communication: pipes

Top comments (2) ↕



haha512 • 29 de nov. de 22 • Edited

...



It's an error

Standard streams

Every UNIX process comes by default with 3 standard communication channels, also known as standard streams repr

```
fd 0: standard output (STDOUT) // Should be input
fd 1: standard input (STDIN)    // Should be output
fd 2: standard error (STDERR)
```

The correct one should be

```
fd 0: standard input (STDIN)
fd 1: standard output (STDOUT)
```

[wiki The file descriptor for standard input is 0 \(zero\).](#)



Leandro Proença • 5 de dez. de 22



Indeed! I appreciate your feedback, it's now fixed

[Code of Conduct](#) • [Report abuse](#)



MultiversX

PROMOTED



MultiversX

/hackathon

Calling All Code Warriors

Blockchain, AI, DeFi, Gaming,
Metaverse and more

Starting September 21st

[Join the hackathon online](#)

If you can code, join #xDayHackathon!

Rust, go, typescript, python, c/c++.

Develop tools, scripts, smart contracts, or even a fully-fledged app.

[Learn More](#)



Leandro Proença

Programmer • I occasionally write blog posts in both English and Brazilian Portuguese.

LOCATION

Brazil

EDUCATION

BSc, Information Systems

WORK

Software Developer

JOINED

8 de ago. de 2017

More from [Leandro Proença](#)

[pt-BR] Fundamentos do Git, um guia completo

#git #linux #braziliandevs

Git fundamentals, a complete guide

#git #linux

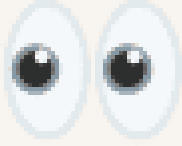
Tekton CI/CD, part IV, continuous delivery

#kubernetes #docker #linux #agile

DEV Community

...

 [Start Reaching Developers](#)



**Send this to
your marketing
team**

You can reach developers by advertising on DEV.
To find out more, head to this page below.



Explore [DEV Advertising Options](#) to start the conversation.