# Piscine — Tutorial D8 PM

version #b4173ffc5996c36bf7f9730185ebc8357500bd86


Assistants C/Unix 2024 <assistants@tickets.assistants.epita.fr>

## Contents



*https://intra.forge.epita.fr

# 1 Syscalls related to I/O

Creating, modifying and deleting a file is not possible as a simple user of the computer. For this, you have to delegate the task to the operating system's kernel. This is why the kernel provides *system calls* (or *syscalls*) as a way to perform specific operations.

It is important to understand that syscalls are really heavy operations. Indeed, when you use a syscall, the kernel will freeze your program (i.e. save all the state your program had before the call), do its job (like opening the file, reading, etc...), and then restore your process.[1]

This is why many functions that use syscalls try to limit the number of calls to the system. For instance, `printf(3)` uses `write(2)`, but uses buffers in order to reduce its system usage when multiple `printf(3)` are called.

## 1.1 `open(2) (fcntl.h)`

To open a file for reading or writing, you need to use the `open(2)` syscall. A successful call to `open(2)` returns a *file descriptor*, i.e. a positive integer serving as an identifier for the system for further read-/write operations. The file descriptor is actually the index of the file in the *File Descriptor Table*, a per-process array of open resources. With this index, you can read, write, or apply other operations to the file.

To be able to use `open(2)`, you need to include the (POSIX) header `fcntl.h`. It contains, among others, the declaration of `open(2)` and its related constants (flags).

To call `open(2)`, you need two or three arguments:

- the path, relative or absolute, to the file (or resource) you want to open
- the mode for opening the file (read, write, append, ...)
- the file permissions (optional), if it is created

Here are the different signatures for `open(2)`:

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

For `flags`, you need to specify one of the following flags:

- `O_RDONLY` for read only
- `O_WRONLY` for write only
- `O_RDWR` for read/write

You can provide other flags that are optional, like `O_CREAT` if you want to create the file, or `O_APPEND` if you want to write at the end of the file instead of replacing the contents. More flags can be found in the man page (`man 2 open`).

If the call to `open` can create the file, it is necessary to specify the permissions as the third argument. Otherwise this is an undefined behavior and the permissions will be random: you may not even be able to delete the file!

[1] To give you an example, it is a bit like scrolling a PDF to read a footnote, read it, and going back to where you were.[2]

[2] In both cases: do not overuse them!

To see the possible values for this third argument, take a look at the man page.

Last but not least, like with every syscall, it is important to check the return value of `open(2)`, as it can fail for many reasons. In case of failure, it will return `-1` and set `errno(3)` to the number of the error. You can then use `perror(3)` or `strerror(3)` to get the error message.

**Be careful!**

A file opened **MUST** be closed at some point. You must **never** leave a file opened indefinitely.

**Tips**

You can use valgrind with the `--track-fds=yes` option to list open file descriptors on exit.

### 1.2 `stat(2)`

Now that you have opened a file you might want to get some information like its size, its owner, time of last access, etc. These syscalls do exactly that.

```
int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *buf);
int lstat(const char *pathname, struct stat *statbuf);
```

You will find at the bottom of the man page an example of how to call one of the `stat(2)` functions.

**Tips**

`stat` is also a command see `stat(1)`. It shows the metadata related to the file just like `stat(2)`.

### 1.3 `close(2)`

The `close(2)` syscall is very simple to use. Unlike `open(2)`, it takes only one argument: the *file descriptor* to close. It is defined in the (POSIX) header `unistd.h`.

Here is its signature:

```
int close(int fd);
```

It returns `0` if it succeeded, `-1` otherwise (and sets `errno(3)`). For the possible errors, see `man 2 close`.

### 1.4 `read(2)`

Now that you can open and close a file, we will see how to read data from it. The file must have been opened with read permissions (`O_RDONLY` or `O_RDWR`). Once you have the file descriptor, you can call `read(2)` to read its content. Here is the signature of `read(2)`, as defined in `unistd.h`:

```
ssize_t read(int fd, void *buf, size_t count);
```

The first argument, `fd`, is the file descriptor you want to read. `buf` is a buffer (character array) in which `read(2)` will *write* the content of the file, and `count` is the maximum number of bytes to be read. The buffer must be allocated, and must be big enough to contain `count` bytes.

The return value of `read(2)` is very important. Indeed, `count` is the *maximum* number of bytes to read. The *actual* number of bytes read is given by the return value. For instance, if the file has less than `count` bytes left to be read, or for some other reason (network read, ...), `read(2)` will stop before having read `count` bytes. Thus, to read a file completely, you will need a loop.

A return value of `0` indicates the end of the file, and a value of `-1` indicates an error (see `man 2 read`).

### 1.5 `write(2)`

`read(2)` allows you to read a file, so `write(2)` will allow you to...? You guessed it. The file must have been opened with writing permissions (`O_WRONLY` or `O_RDWR`). Once you have the file descriptor, you can call `write` with this signature (defined in `unistd.h`):

```
ssize_t write(int fd, const void *buf, size_t count);
```

It is symmetrical to read: `fd` is the file descriptor, `buf` the *source* (the data to be written), and `count` the number of bytes to write. Once again, the *actual* number of bytes written is given by the return value, so you will need a loop to make sure that you wrote everything.

And again, in case of error, it will return `-1`, and `errno` will be set to an error from `man 2 write`.

### 1.6 `lseek(2)`

When you read or write in a file, a position pointer is updated. It is used as a cursor, to know where you will read/write next. Most of the time, you only need to work from the beginning to the end, but you do not have to. Indeed, this read/write head can be moved with `lseek(2)`:

```
off_t lseek(int fd, off_t offset, int whence);
```

As always, `fd` is the file descriptor. `offset` is interpreted depending on the third argument: `whence` (which means "*from where*"), indicates from where the head must be moved:

- `SEEK_SET`: Sets the file offset to `offset` bytes.
- `SEEK_CUR`: Adds (or subtract if `offset` is negative) `offset` to the current file offset.
- `SEEK_END`: Sets the file offset to the size of the file plus `offset`.

`lseek(2)` returns the position of the cursor (in bytes) from the beginning of the file. A way to get the current position of the cursor is to ask `lseek(2)` to shift the cursor by `0` bytes from the current position.

## 1.7 stdin, stdout, stderr

In most cases, when a process is created, there are 3 file descriptors already opened: *stdin*, *stdout* and *stderr*, for *standard input*, *standard output* and *standard error* (for error messages) respectively. The first one can only be read, the last 2 can only be written. The file descriptor numbers are given by `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`, in `unistd.h`.

For example, `puts(3)` ends up doing a `write(2)` on *stdout*.

## 1.8 Practice

For all the following exercises, you must use *only* syscalls.

### 1.8.1 Hello syscalls

#### Goal

Hello world, again! Use `write(2)` to display your message on stdout.

### 1.8.2 Read: write only

#### Goal

Try to `read` a file opened with write only permissions. What is the name of the constant `errno` is set to? What does `perror` display?

### 1.8.3 Read: small buffer

#### Goal

Read a file with a buffer too small (smaller than `count`). What is the name of the constant `errno` is set to? What does `perror` display?

### 1.8.4 Create file

#### Goal

Create a file with permissions 755.

### 1.8.5 Micro cat

#### Goal

Display the content of a file.

### 1.8.6 Spoilers

#### Spoilers

The goal of this exercise is to make a program which displays the last 2 characters of a file, excluding whitespace characters. You must submit a file with a main function. The two characters must be followed by a line feed.

Whitespace characters:

- ' ' space
- '\f' feed
- '\r' carriage return
- '\n' newline
- '\t' horizontal tab
- '\v' vertical tab

```
42sh$ ls
enemy.txt champignius_brain.txt
42sh$ cat enemy.txt
We all hate Champignius
```

Only 1 argument must be passed to your program otherwise display *Invalid number of arguments* to stderr and return 1. If the file is too short you must display *File is not long enough* to stderr and return 1. If no error occurred return 0. If the file does not exist or if you can't access it you should print *Could not open file* to stderr and return 1.

```
42sh$ ./spoilers enemy.txt
us
42sh$ ./spoilers enemy.txt champignius_brain.txt
Invalid number of arguments
42sh$ ./spoilers
Invalid number of arguments
42sh$ ./spoilers champignius_brain.txt
File is not long enough
42sh$ ./spoilers champignus_pride.txt
Could not open file
```

### 1.8.7 Write after EOF

#### Goal

Write **after** the end of a file. Is there an error? If so, which one? Otherwise, how was the file modified?

### 1.8.8 One out of two

#### Goal

Write an executable that takes a filename as an argument, and displays every other character of a file, i.e. skips one character every two characters.

You should exit with a return value of *1* when the number of arguments given to your executable is not exactly one. You should also exit *1* on any error.

# 2 `strace(1)`

## 2.1 Introduction

`strace(1)`, for "linux **s**yscall **trace**r", is a program that runs the given executable until it exits. While executing, it will print all syscalls made with their arguments, return value and the value of `errno` when an error occurs. `strace(1)` is a useful tool for debugging as well as reverse-engineering programs.

There are two ways to use `strace(1)`:

- Trace a new program

```
42sh$ strace command args
```

- Trace an existing program

```
42sh$ strace -p pid
```

If you try to strace `ls(1)`, you can see at the end of the trace one or several calls to `write(2)` which are creating the output.

```
42sh$ ls
file1 file2 file3
42sh$ strace ls
...
write(1, "file1  file2  file3\n", 20)   = 20
...
```

You can see that it is writing on file descriptor 1, which is `stdout`, the 20 characters long string `file1 file2 file3\n`, and that `write(2)` returns 20.

**Tips**

Most of the time, problems in your code involving syscall can be found and fixed using `strace(1)`.

**Going further...**

You can filter `strace(1)` output with `-e` option see `strace(1)`

**Going further...**

`strace(1)` also allows you to see signals that your program receives.

```
42sh$ strace ./a.out
execve("./a.out", ["./a.out"], 0x7ffec1829720 /* 45 vars */) = 0
...
--- SIGSTOP {si_signo=SIGSTOP, si_code=SI_USER, si_pid=42, si_uid=1000} ---
--- stopped by SIGSTOP ---
```

**Going further...**

`ltrace(1)` is a program very similar to `strace(1)`. Their usage are almost identical. The only difference between the two is that `strace(1)` tracks system calls whereas `ltrace(1)` tracks calls to functions in dynamic libraries. It is therefore possible to track calls to any function in the C library.

## 2.2 Exercises

### 2.2.1 Strace

#### Goal

Find the `read` buffer size for the `cat` command.

### 2.2.2 Strace me

#### Goal

Using the given file `strace_me` you will have to build a folder architecture in which this program will run properly. In your git you have to submit the whole architecture but not the binary (it will be considered as a trash file).

Example of usage once the folder is properly created :

```
42sh$ cd strace_me
42sh$ ./strace_me
42sh$ echo $?
0
```

The binary must return 0.

# 3 Signals

Signals are a type of *IPC* (Inter Process Communication): they are used for basic and simple interactions between different processes. They are also used by the kernel to trigger actions on processes.

## 3.1 Explanation

Signals are an implementation of software interrupt sent to processes, they are used by the kernel to report exceptional situations, like errors or asynchronous events.

There are numbers defined by the kernel with common names (and macros in C) to identify them. Take a look at `signal(7)` for the full list.

**Be careful!**

Never use the numerical value as it is OS dependent.

Fortunately the kernel is not the only one who is able to send signals, but it is always in charge of the delivery, thus you are able to send signals *through* the kernel.

When a signal is sent to a process, it is also sent to all the processes in its process group e.g. its children.

Again, go read the manual of `signal(7)` to know more about them.

## 3.2 Sending signals

To send a signal you need two things: the signal and the pid of the process you want to send it to. Remember, the pid is the unique process identifier on Unix systems.

### 3.2.1 In shell

To send a signal to a process in shell you can use the command `kill(1)`.

```
42sh$ kill -s SIGINT $my_process_pid
```

To know a process' pid you can use the command `pgrep(1)`, it gives the pids of all the processes matching the given criteria.

```
42sh$ pgrep my_process
1042
42sh$ kill -s SIGINT 1042
```

There is an easier and faster way to do this: `pkill(1)` which works like `kill(1)` but you can give the matching criteria you would give to `pgrep(1)`.

```
42sh$ pkill pattern --signal SIGINT
```

**Be careful!**

`pkill(1)` sends the given signal to every process matching the pattern. Be very careful using it.

#### 3.2.2 In `C`

Sending a signal in *C* is *almost* as easy as in *shell*. You have to use the syscall `kill(2)` giving it the target's pid and the signal number.

Unfortunately there is no easy way in C to get a process' pid like in shell. Thus daemon usually store their pid in a file for other process to send them signals, as it is one of the only way to communicate with them.

### 3.3 Catching signals

Signals all have a default behavior. Often it is ignored and does nothing, but it can also terminate the program with or without a core dump depending on the signal used.[1]

But you can choose to *catch* signals and change the behavior of your program. It is mainly used to add cleanup behavior (deleting temporary files, closing sockets, ...) when receiving a termination signal. But it can be also used as a way to interact with your program.

**Be careful!**

There are two signals that you **cannot** catch:

- SIGKILL
- SIGSTOP

#### 3.3.1 The syscalls

There are three functions to set a *signal handler*: `signal(2)`, `sigaction(2)` and `signalfd(2)`.

- `sigaction(2)`: it is more recent and the one you should use.
- `signal(2)`: it is the older function and has more overall compatibility, but it is less powerful and less friendly to use than `sigaction(2)`.
- `signalfd(2)`: this one is synchronous, its peculiarity is that you read the signal queued from a file descriptor with `read(2)` whereas sigaction would call the handler when the signal is raised.

In short, if you have the choice, do not use `signal(2)`.

`signalfd(2)` has been designed to be used with `select(2)` or `poll(2)`. It is harder and longer to set up correctly a program that catches signals with `signalfd(2)`.

This is the reasons why you will only work with `sigaction(2)` in this tutorial.

[1] Again, see the manual pages for `signal(7)` and `core(5)` for more information.

### 3.3.2 sigaction(2)

This function allows you to associate a *handler* with a given signal. This *handler* is called when the signal is received. The *handler* must be a function whose prototype follows the `sighandler_t` typedef[2] (see the manual for `signal(2)`)

```
void handler(int signum);
```

`signum` is the number corresponding to the received signal.

In fact `sigaction(2)` uses structures to get additional information about what to do when the signal is received. Here is an example of how to use it:

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int signum)
{
    switch(signum)
    {
    case SIGINT:
        // Caught a SIGINT
        break;
    case SIGUSR1:
        // Caught a SIGUSR1
        break;
    case SIGUSR2:
        // Caught a SIGUSR2
        break;
    default:
        // Not expecting to catch this signal
        break;
    }
}

int main(void)
{
    struct sigaction sa;
    sa.sa_flags = 0; // Nothing special to do
    sa.sa_handler = handler;

    // Initialize mask
    if (sigemptyset(&sa.sa_mask) < 0)
    {
    // Handle error here
    }

    if (sigaction(SIGINT, &sa, NULL) < 0)
    {
    // Handle error here
    }
```

(continues on next page)

[2] You may find it weird seeing a typedef like `sighandler_t`, but this is actually how you define a *function pointer type* using `typedef`.

(continued from previous page)

```
    if (sigaction(SIGUSR1, &sa, NULL) < 0)
    {
    // Handle error here
    }
    if (sigaction(SIGUSR2, &sa, NULL) < 0)
    {
    // Handle error here
    }

    while(1);
}
```

Here, `handler` is the function being called upon receiving `SIGINT`, `SIGUSR1` or `SIGUSR2`.

Do not forget to check for errors and to read the man pages to know the different error codes possible.

**Be careful!**

As described in `man 2 sigaction`, you need to enable `feature_test_macros(7)` to compile your code. You can either define the corresponding macro using the `#define` directive before other includes or using the `-D` GCC option.

### 3.4 Guided exercise

1. a. Write a program that goes into infinite loop.

   b. Send a signal to your program from your shell. Try to send the followings:

      - `SIGINT`
      - `SIGUSR1`
      - `SIGUSR2`
      - `SIGSEGV`
      - `SIGKILL`

3. a. Change your program with the following instructions:

   - **It contains a *handler* named `handler_1`.**
       - `handler_1` prints `"Hello user!\n"`.
       - It must be triggered when the program receive `SIGUSR1` or `SIGUSR2`.
   - **It contains another *handler* named `handler_2`.**
       - If `handler_2` received `SIGSEGV` it must print `"SIGSEGV\n"`.
       - If `handler_2` received `SIGINT` it must print `"SIGINT\n"`.
       - `handler_2` prints `"Unknown\n"` otherwise.
       - It must be triggered when the program receive `SIGSEGV` or `SIGINT`.

**Tips**

You can use the same sigaction structure multiple times. Simply change the handler, call `sigemptyset` and then assign signals with `sigaction`.

b. Now try to send the following signals to your program:

- `SIGINT`
- `SIGUSR1`
- `SIGUSR2`
- `SIGSEGV`
- `SIGQUIT`

## 4 ltrace(1)

Unix systems rely widely on dynamic libraries (.so files) which are binaries in the ELF format containing code and symbols, like executable files, but no entry point. When a process makes use of a dynamic library, its content is loaded into the process' address space at a variable location. The process has a mechanism that allows it to find the location of the library's functions so it can call them when needed. This is for example the case for the functions `printf`, `atoi`, `puts`, `strlen`, ...

`ltrace` is a program that exploits this mechanism to intercept and record every call a program does to such functions. Let's take the following example:

**File:** `example1.c`

```c
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(void)
{
    char buffer[] = "Test.";
    char output[32];
    size_t len;

    strcpy(output, buffer);
    len = strlen(output);
    printf("%s (%zu)", output, len);
    putchar('\n');

    int fd = open("/dev/null", 0);
    close(fd);
}
```

If we launch this program with `ltrace`, we get the following output:

```
42sh$ ltrace ./test
__libc_start_main(0x4005e6, 1, 0x7fff79bf4198, 0x400670 <unfinished ...>
```

(continues on next page)

(continued from previous page)

```
strcpy(0x7fff79bf4080, "Test.") = 0x7fff79bf4080
strlen("Test.") = 5
printf("%s (%zu)", "Test.", 5) = 9
putchar(10, 0x400706, 0x7fb8a6b50a00, 0x7ffffff7Test. (5)
) = 10
open("/dev/null", 0, 012) = 3
close(3) = 0
```

`ltrace` just executed our program and told us every call it made to a dynamic library's function. Its output has the following syntax:

> function(parameters...) = return-value

Let's analyze the output line by line, we notice three important things:

- the call to `__libc_start_main`. This function is called right before your `main` function. Its purpose is to initialize the process and to end it properly when the `main` function returns. The vast majority of programs are compiled with the `libc` (this is what we call the "standard library"), thus, you will see this call almost every time;
- the `putchar` function is defined as `putchar(char c)`, but here, `ltrace` shows incoherent information. In order to display calls to functions correctly, `ltrace` uses some files (`/etc/ltrace.conf` and `$HOME/.ltrace.conf`) to find the prototypes of these functions. Here, the prototype of `putchar` is not provided, so `ltrace` tries to guess what parameters it could take. We notice, however, that the first parameter, "10" is the ASCII code of '\n', the character we gave as a parameter to `putchar`;
- `open` and `close` are **syscalls**. They are not functions that come from a dynamic library, but they appear in the output of `ltrace`. In practice, the `libc` provides an abstraction layer to many *syscalls*. This abstraction consists of a wrapper function that will in turn do the real *syscall*. These lines thus refer to calls to these wrapper rather than the real *syscalls*.

*I must not fear. Fear is the mind-killer.*