Piscine — Tutorial D9

version **#v3.2.0**

Assistant C/UNIX 2021 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

1 Syscalls related to I/O **3**
- 1.1 `open(2) (fcntl.h)` . . . 3
- 1.2 `stat(2) (unistd.h)` . . . 4
- 1.3 `close(2) (unistd.h)` . . . 4
- 1.4 `read(2) (unistd.h)` . . . 4
- 1.5 `write(2) (unistd.h)` . . . 5
- 1.6 `lseek(2) (unistd.h)` . . . 5
- 1.7 stdin, stdout, stderr . . . 5
- 1.8 Practice . . . 6
- 1.9 Goal . . . 7

2 Processes **7**
- 2.1 Definition . . . 7
- 2.2 Process lifecycle . . . 8
- 2.3 Starting a subprocess . . . 9
- 2.4 Executing another program . . . 10
- 2.5 Guided exercise . . . 10
- 2.6 Creating a daemon . . . 12

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

1 Syscalls related to I/O

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

It is important to understand that syscalls are really heavy operations. Indeed, when you call 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.

...with (optionally) other flags, 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 file may be created, it is necessary to specify the third argument, for the permissions (otherwise the permissions are random, you may not even be able to delete it!). To see the values possible for this third argument, see `man 2 open`.

[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 : don't abuse it!

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.

1.2 `stat(2)` (`unistd.h`)

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

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

1.3 `close(2)` (`unistd.h`)

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

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)` (`unistd.h`)

Now that you can open and close a file, we will see how to read data from it. The file has to 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)` (`unistd.h`)

`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)` (`unistd.h`)

When you read or write in a file, a position pointer is updated. It serves as the 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`: Set the file offset to `offset` bytes.
- `SEEK_CUR`: Adds (or subtract if `offset` is negative) `offset` to the current file offset.
- `SEEK_END`: The file offset is set 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

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

1.8.2 Read: write only

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

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

Create a file with permissions 755.

1.8.5 Micro cat

Display the content of a file.

1.8.6 Spoilers

The goal of this exercise is to make a program which displays the last 2 characters of a file, excluding whitespace characters.

Whitespace characters:

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

```
42sh$ ls
example1.txt example2.txt example3.txt
42sh$ cat example1.txt
This is a simple line.
```

Only 1 argument must be passed to your program otherwise display *Invalid argument number.* to stderr.

```
42sh$ ./spoiler example1.txt
e.
42sh$ ./spoiler example1.txt example1.txt
Invalid argument number.
42sh$ ./spoiler a b c d e f g h
Invalid argument number.
42sh$ ./spoiler
Invalid argument number.
```

1.8.7 One out of two

1.9 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.

1.9.1 Write after EOF

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

2 Processes

2.1 Definition

A process is an instance of a program in execution. The process contains the program state (including its code and data). When you launch a program, a process that contains this information is started. Processes are like cells in some way: they are forked by their parent process, they have their own life, they optionally generate one or more child processes, and eventually, they die. Each process also has a unique Process Identifier, or **PID**[1], associated with it.

[1] `credentials(7)` is a great source of information about PIDs.

2.2 Process lifecycle

Processes on a computer form a tree hierarchy: each process has a parent process and, optionally, child processes. The root of the *process tree* is the process with the PID **1**, called `init`, which is launched by the kernel when the machine boots. A process whose parent dies before itself becomes the child of `init`. You can view the process tree with the command `pstree(1)`:

```
42sh$ pstree -T
init-+-5*[agetty]
  |-2*[alacritty---bash]
  |-alacritty---bash---pstree
  |-chrome-+-chrome---chrome---25*[chrome]
  |        `-4*[chrome]
  |-chronyd
  |-console-kit-dae
  |-crond
  |-2*[dbus-daemon]
  |-dbus-launch
  |-dconf-service
  |-dhcpcd
  |-dockerd---containerd
  |-emacs
  |-gvfsd-+-gvfsd-network
  |       `-gvfsd-trash
  |-i3bar---i3blocks---cpu_usage---mpstat
  |-login---bash-+-mega-cmd-server
  |              `-startx---xinit-+-X
  |                               `-i3
  |-lvmetad
  |-pulseaudio---gsettings-helpe
  |-sshd
  |-syslog-ng---syslog-ng
  |-udevd
  |-wpa_cli
  `-wpa_supplicant
```

Of course, the output of `pstree` might be very different on your machine. Notice that `init` is indeed at the root of the tree. Once a process dies, it does not get removed from the process tree immediately. To remove a dead process from the process tree, the process must have its termination information read (using the `wait(2)` or `waitpid(2)` syscalls) by its parent. A *dead process* whose state has not been read by its parent becomes a **zombie**[2].

Going further...

Note that `init` will always read its child's termination information.

[2] A *zombie* still has an entry in the process tree but has finished its execution. As such it can be considered dead and alive at the same time, hence the *zombie* name.

2.2.1 Daemons

Some processes can run without direct user interaction, as background processes. These are called **daemons**. To communicate with a daemon, you can send it signals as it is not interactive. A daemon is a process whose parent died and is attached to the first user process: `init`.

2.3 Starting a subprocess

In `C`, to start a new subprocess in your program, you need to use the syscall `fork(2)`. A successful call to `fork(2)` returns a *PID*, i.e. an integer that represents the id of a process. When it succeeds in creating a new process, `fork(2)` returns:

- The **child's PID** in the parent process.
- **0** in the child process.

If it fails to create a new process, `fork(2)` returns *-1*, the process is not created, and `errno(3)` is set to indicate the kind of error encountered by `fork(2)`.

`fork(2)` duplicates the parent process to create the child process. The only differences between the parent and the child is that the child runs in a separate memory space and has a different *PID*.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
  pid_t cpid;

  cpid = fork();

  if (cpid == -1)
  {
    // Failed to create new process
    return 1;
  }
  else if (!cpid)
  {
    // Subprocess successfully created
    // We're in the child
    printf("Hello I'm the child!\n");
  }
  else
  {
    // Subprocess successfully created
    // We're in the parent
    printf("Hello I'm the parent and my child is %d\n", cpid);
  }
  return 0;
}
```

From this code, it is easy to create a daemon. Indeed, in order to make a daemon, you only have to exit in the parent and make the child do stuff.

It can also be interesting to keep both parent and child and make the child run another program. Then the exit state of the child would be used by the parent.

If you did not create a daemon you will have to get the termination information of the child (we also say the parent **waits** for the child). To do this you have to use the syscalls `wait(2)` or `waitpid(2)` we have told you about before.

2.4 Executing another program

In `C`, to execute a program you have to use the syscall `execve(2)`. This syscall does not launch a new process. Instead, it replaces the current process' execution flow with the one in the targeted executable.

You rarely call `execve(2)` directly. Instead, you should use the *libc* function `execvp(3)`, which is a wrapper around `execve(2)`. It uses the `PATH` environment variable to find the executable if you do not give its full path. `execvp(3)` also needs less arguments than `execve(2)`:

```c
int execvp(const char *file, char *const argv[]);
```

Be careful!

When you build your arguments for `execvp(3)`, do not forget to add a null pointer at the end of your array.

2.5 Guided exercise

1. a. Write a program that calls `ls(1)` using `execvp(3)`.

   b. Add some instructions at the end of your program (after the call to `execvp(3)`).

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

int main(void)
{
    char *args[] = { "ls", "-a", NULL};
    execvp(args[0], args);

    // Example of something that can be added to see that this is not printed
    printf("Yay!\n");
}
```

Here you can see that the end of your execution flow is ignored. Thus, you will need to create a process with `fork(2)` that will then call `execvp(3)`.

2. Call `execvp(3)` in a subprocess.

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    pid_t cpid = fork();

    if (cpid == -1)
    {
        /* Handle error here */
    }
    else if (!cpid)
    {
        char *args[] = { "ls", "-a", NULL};
        execvp(args[0], args);
    }

    printf("Yay!\n");
    return 0;
  }
```

3. With this update, it is possible to continue the execution flow in the parent, but there are still two issues:

   - We do not know if the execution in the child has succeeded
   - We do not wait for the child to exit when the father executes the code after fork

   Using `waitpid(2)` will resolve these issues. You have to check if the child exited and if its exit status is 0. Write the code.

Going further...

`wait(2)` can also be used in this case. `waitpid(2)` will return when the child process with the given *PID* dies. `wait(2)` returns when **any** of the child processes dies.

Tips

- You should read the man page of `waitpid(2)`.
- WIFEXITED(status) returns *true* if child process exited normally (end of main function, call to `exit(3)` or `_exit(2)`).
- WEXITSTATUS(status) returns the exit status of child process. Be sure WIFEXITED returned *true* before calling this macro.

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void)
{
    pid_t cpid = fork();
```

(continues on next page)

(continued from previous page)

```
    if (cpid == -1)
    {
        /* Handle error here */
    }
    else if (!cpid)
    {
        char *args[] = { "ls", "-a", NULL};
        return execvp(args[0], args);
    }

    int cstatus = 0;
    if (waitpid(cpid, &cstatus, 0) == -1)
    {
        /* Handle error here */
    }

    if (WIFEXITED(cstatus) && WEXITSTATUS(cstatus) == 0)
    {
        /* Success */
    }
    else
    {
        /* Failure */
    }

    return 0;
}
```

In order to truly *launch* a program, you will have to first `fork` and then call `execve(2)` or `execvp(3)` in the child. To make sure that the child has ended its execution flow without error, you should always check it in the parent using `waitpid(3)`.

2.6 Creating a daemon

2.6.1 Explanations

In the case of a daemon, the code is almost the same. To create a simple daemon, it is possible to simply call `exit(3)` or return from the main function. This is the equivalent of using `daemon(3)`.

A real daemon would also create a new session by calling `setsid(2)` before calling `fork(2)` a second time.

Going further...

You should take a look at the `daemon(3)` and `credentials(7)` manual pages.

2.6.2 Guided exercise

Create a daemon that writes its pid in a file (use `getpid(2)`) and sleeps for 60 seconds.

Once your daemon is running, you can get the pid of the daemon in the file and check that it is running with `ps aux | grep $(cat file)`.

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

int main(void)
{
    int ds = daemon(1, 0);

    if (ds == -1)
    {
        // Handle error here
    }

    char buf[6];
    sprintf(buf, "%d\n", getpid());

    int fd = open("truc.txt", O_CREAT | O_TRUNC | O_WRONLY, S_IRUSR | S_IWUSR);

    if (fd == -1)
    {
        // Handle error here
    }

    int wc = write(fd, buf, sizeof(buf) - 1);
    if (wc < 0)
    {
        // Handle error here
    }

    if (close(fd) == -1)
    {
        // Handle error here
    }

    sleep(60);

    return 0;
}
```

The only way out is through