Piscine — Tutorial D10

version **#b4173ffc5996c36bf7f9730185ebc8357500bd86**

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

Copyright

Contents

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

1 Processes

1.1 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.

1.2 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 `systemd`, which is launched by the kernel when the machine boots. A process whose parent dies before itself becomes the child of `systemd`. You can view the process tree with the command `pstree(1)`:

```
42sh$ pstree -T
systemd-+-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 `systemd` 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

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

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 `systemd` will always read its child's termination information.

1.2.1 Daemons

Some processes can run without direct user interaction, as background processes. These are called *daemon*. To communicate with a `daemon(7)`, you can send it signals as it is not interactive. A `daemon(7)` is a process whose parent died and is attached to the first user process: `systemd`.

1.3 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*. 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 are in the child
        printf("Hello! I am the child!\n");
```

(continues on next page)

[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.

(continued from previous page)

```c
    }
    else
    {
        // Subprocess successfully created
        // We are in the parent
        printf("Hello! I am the parent and my child is %d\n", cpid);
    }
    return 0;
}
```

It can 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.

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 mentioned earlier.

1.4 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.

1.5 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("This line should never be reached\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.

```
#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("This line should be reached\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.

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

```

(continues on next page)

(continued from previous page)

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

    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)`.

1.6 Creating a daemon

1.6.1 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.

1.6.2 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;
}
```

1.6.3 Exercise: My Daemon

Goal

This exercice is combining subprocesses and signals to create a simple daemon.

As you may know, a daemon is a process that runs in the background, without user interaction.

This one will be a simple daemon with which you will interact using signals. First you will have to create the daemon and then catch four signals:

- `SIGUSR1`
- `SIGUSR2`
- `SIGTERM`
- `SIGINT`

For each signal your daemon will have to do something.

SIGUSR1

Upon receiving this signal the daemon will print the number of signals it has caught since its birth. It shall print "Received `X` signals." with a line feed at the end and `X` replaced by the number of signals received.

SIGUSR2

Upon receiving this signal the daemon will print its current generation number. It shall print "Current generation: `X` " with a line feed at the end and `X` replaced by its generation number. Generation number starts at 0 with the initial daemon.

SIGTERM

Upon receiving this signal the daemon will create another daemon, and then exit. This action increases the next daemon's generation number by one and reset the counter for the number of signal caught.

SIGINT

Upon receiving this signal, the daemon will exit gracefully, that means by returning from the main function.

Example

```
42sh$ ./my_daemon
42sh$ pgrep my_daemon
22469
42sh$ pkill my_daemon --signal SIGUSR1
Received 1 signals.
42sh$ pkill my_daemon --signal SIGUSR1
Received 2 signals.
42sh$ pkill my_daemon --signal SIGUSR2
Current generation: 0
42sh$ pkill my_daemon --signal SIGUSR1
Received 4 signals.
42sh$ pkill my_daemon --signal SIGTERM
42sh$ pkill my_daemon --signal SIGUSR1
Received 1 signals.
42sh$ pkill my_daemon --signal SIGUSR2
Current generation: 1
42sh$ pkill my_daemon --signal SIGTERM
42sh$ pkill my_daemon --signal SIGUSR2
Current generation: 2
42sh$ pkill my_daemon --signal SIGINT
42sh$ pkill my_daemon --signal SIGUSR2
42sh$ pgrep my_daemon
42sh$
```

Be careful!

Your code will be compiled using `-D_POSIX_C_SOURCE`

2 Job control

2.1 Signals

A signal is a kind of notification that you can send to processes to specify an action to be taken. For instance sending a `SIGTERM` signal to a process is used to tell it to terminate itself. Another example is when pressing `Ctrl + C`, you just send a `SIGINT` (interruption signal) signal to the running process.

Most signals can be ignored if needed, but two specific signals cannot be caught: `SIGKILL` and `SIGSTOP`.

To send a specific signal to a process, use the builtin `kill(1)`.

```
42sh$ kill -s SIGKILL 123 # 123 is the pid of a command
```

More information can be found on the dedicated man page (`whatis signal`). To get the pid of a process, you can use `pgrep(1)`.

2.2 Jobs

First, if this is not already done, download this tutorial on your computer. Then open it in a terminal with a pdf reader like `evince`.

```
42sh$ evince tutorial.pdf
tutorial.pdf
```

As you can see, you cannot use your terminal while your pdf reader is active.

This is because your pdf reader is said to be in *foreground*.

Now come back to your terminal where your reader is running and press `Ctrl + Z`.

```
42sh$ evince tutorial.pdf
^Z
[1]+  Stopped                 evince tutorial.pdf
```

Pressing `Ctrl + Z` sends a "terminal stop" signal (named `SIGTSTP`) to the foreground task. As printed in your terminal, this stopped your pdf reader but as you probably see, the window did not close, it is now just unusable. Why? Because stopping does not mean terminating, your process is just paused and now waiting for a `SIGCONT` signal (continue signal) to resume its execution.

Now that your terminal is usable, let us resume our task. Two commands are available:

- `bg` -> put your task in background
- `fg` -> put back your task in foreground

When a task is running in background, your terminal is usable but note that the running task is still attached to your terminal. This means that if you close your terminal, any attached background task will also terminate.

Currently running tasks attached to your terminal can be displayed with the command `jobs`.

```
42sh$ jobs
[1]-  Stopped                 evince tutorial.pdf
[2]+  Stopped                 vim
[3]   Running                 emacs &
```

In the example above, `evince` and `vim` are currently stopped and `emacs` is running in background.

Finally, to run a command in background you do not need to first run it in foreground, stop it and resume it with `bg`. You can just run it with an ampersand (`&`) at the end of your command.

```
42sh$ jobs # no current job
42sh$ evince tutorial.pdf &
[1] 13343
42sh$ jobs # your terminal is usable, evince runs in background
[1]+  Running                 evince tutorial.pdf &
42sh$
```

Going further...

The command `ps` will, by default, display the tasks attached to your terminal. Also, it has options to print a lot more of information about the processes running on your system.

3 Standard library functions

The (far from exhaustive) list of syscalls we saw is not the only way to access files. Higher level functions, from the standard *C* library provide some similar functionalities with some advantages. Most of the functions we will now see are declared in `stdio.h`.

Instead of using file descriptors, these functions work with a pointer to a `FILE`. `FILE` is a structure which has the corresponding file descriptor as one of its fields. However, `FILE` is implementation defined, so depending on which standard library you are using, the fields may not be the same. Only the behavior of the functions is guaranteed.

3.1 Predefined FILE variables

The standard library defines three macros corresponding to the standard stream, already open by the time your `main` function is called. They are `stdin`, `stdout`, and `stderr`, for your program's standard input, standard output, and standard error output respectively.

3.2 `fopen(3)`

The equivalent of `open(2)` is the `fopen(3)` function. It opens a file and returns a `FILE *`. However, the signature is quite different:

```
FILE *fopen(const char *path, const char *mode);
```

`path` is, surprisingly, the path to the file (absolute or relative to `$PWD`). The second one however, is a string rather than an `int`. This string describes the mode for opening the file, the possible values being listed in the `fopen(3)` man page, among which:

- `"r"` is read-only.
- `"r+"` is read-write.
- `"a"` is append (i.e. write only, but with the cursor at the end of the file).

The `FILE *` returned is important to be able to use the other library functions.

3.3 `fclose(3)`

`fclose(3)` closes a file opened with `fopen(3)`:

```
int fclose(FILE *fp);
```

The only parameter is the `FILE *` to close and free. The return value is `0` if all goes well. See the man page for possible errors.

3.4 `fread(3)`

The `fread(3)` function allows to read from a file, like `read(2)`. However, its use can *seem* more complex. Let us have a look at its signature:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Here, `ptr` is like `buf` for read, the buffer into which we will read the data. `stream` is the `FILE *` from which we will read. However, where `read(2)` only took one parameter to indicate the number of bytes to read, `fread(3)` takes two: `size` and `nmemb`. The first one is the size, in bytes, of one *element*, and `nmemb` is the *number of elements* to read. For instance, if you have an array of 23 `int` to read, `size` will be `sizeof(int)`, and `nmemb` will be 23.

Why the difference? Because `fread(3)` returns the number of *elements* read, not the number of bytes.

3.5 `fwrite(3)`

As you might have guessed, `fwrite(3)` is to `fread(3)` what `write(2)` is to `read(2)`. The signature is like so:

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

The parameters are the same as `fread(3)`, only the `ptr` buffer contains the data to be written to `stream`.

`fwrite(3)` returns the number of elements written.

3.6 `fseek(3)` and `ftell(3)`

For `fseek(3)`, the signature is the same as `lseek(2)`, except that the file descriptor has been replaced by a `FILE *`:

```
int fseek(FILE *stream, long offset, int whence);
```

To get the read/write head position, instead of calling `fseek(3)` with dummy arguments, you can use `ftell(3)`.

3.7 `getline(3)` and `getdelim(3)`

As you saw `fread(3)`, is well-suited for binary streams. Most commonly however you will want to read a file line by line. These functions are exist for this purpose:

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
ssize_t getdelim(char **lineptr, size_t *n, int delim, FILE *stream);
```

Reading the man page carefully is mandatory in order to use the functions correctly.

Be careful!

Don't forget to define `_POSIX_C_SOURCE 200809L` macro in order to use `getline(3)` and `getdelim(3)`.

3.8 Exercises

3.8.1 Hidden message

Goal

The provided file `hidden_file` contains a hidden message. The caracters composing the message are the fourth character of the file and each fourth character following a semicolon (`;`). There can not be more than 32 characters between two semicolons.

Write a program that accepts a file as an argument, extracts the hidden message and prints it in the stdout followed by a newline.

Use `fopen`, `fseek`, `getdelim` and `fwrite`. Return `1` if any call to one of those functions fails, `0` otherwise. You do not have to handle bad file structures such as a semicolon in the last four characters.

Example

```
42sh$ gcc -Wall -Wextra -Wvla -Werror -std=c99 -pedantic -o hidden_message hidden_message.c
42sh$ ./hidden_message hidden_file | cat -e
I must not fear. Fear is the mind-killer.$
```

4 Redirections

4.1 Preamble

For this part, you need to have a clear understanding of *file descriptors*.

A *file descriptor* is an integer representing a file or other kind of object such as the ones you will see in the next part.

In *Unix* the following file descriptors are already bound to streams:

- `0`: Bound to the standard input (what enters the program).
- `1`: Bound to the standard output (what is going out of the program).
- `2`: Bound to the standard error output (what is going out of the program, but seen as an error).

4.2 Basics

Redirection is the process of changing the input *or* the output of a command. The basic syntax is pretty simple:

- `command > file`: `command` output is written to the file. If this file does not exist, it is created. If it already exists, its content is overwritten.
- `command >> file`: appends `command` output to the file. If the file does not exist, it is created.
- `command < file`: the content of file is used as a standard input for `command`.

This syntax is simple because it hides what is really behind it. For instance, the `command > file` form is a syntactic sugar for `command 1> file`. What this means is that the standard output of `command` (file descriptor `1`) will be redirected into file.

```
42sh$ ls
script.sh
42sh$ cat script.sh
#!/bin/sh

echo out # Prints on standard output
ech oout # Prints on standard error
42sh$ ./script.sh 1> file1 2>> file2
42sh$ ./script.sh 1> file1 2>> file2
42sh$ ls
file1 file2 script.sh
42sh$ cat file1
out
42sh$ cat file2
./script.sh: 4: ./script.sh: ech: not found
./script.sh: 4: ./script.sh: ech: not found
```

Let us have a look at these commands. The `script.sh` echoes an error message to the standard error and 'out' to the standard output. Then, we redirect the standard output to file1 two times and append the standard error to file2. As a result, we see that 'out' is written once to file1 (as it has been overwritten) and that the standard error is written two times to file2 (as it has been appended).

```
42sh$ cat script.sh
#!/bin/sh

echo out # Prints on standard output
ech oout # Prints on standard error
42sh$ ./script.sh &> file
42sh$ cat file
out
./script.sh: 4: ./script.sh: ech: not found
```

This syntax `command &> file` is actually pretty simple to understand. It simply means to redirect both standard output and standard error to file.

Be careful!

The `command &> file` syntax is not POSIX compliant. You should use `command >file 2>&1` in a portable shell script

4.3 Redirecting to file descriptors

You have seen how to redirect to files, the syntax to redirect to a file descriptor is pretty much the same, except that you have to put an ampersand `&` before the file descriptor.

```
42sh$ cat script.sh
#!/bin/sh

echo err >&2 # Redirecting the standard output to standard error
echo out
42sh$ ./script.sh 1> file1 2> file2
42sh$ cat file1
out
42sh$ cat file2
err
```

4.4 Standard input redirection

Instead of reading from the standard input, it is possible to read from something else, a file for example.

```
42sh$ cat file
hello world
42sh$ wc -w # Count number of words
hello world
2
42sh$ wc -w < file
2
```

Here, we use the `wc` command that displays the number of lines, words, and bytes contained in each input file, or standard input.

For more information: `man wc`.

Going further...

You can use different type of redirections for a single command:

```
42sh$ wc -w < input-file > output-file
```

4.5 Pipes

Pipes redirect standard output from the left command to the standard input of the right command.

```
42sh$ ls
file1 file2 file3
42sh$ wc -l # Count number of lines
hello
world
2
```

(continues on next page)

(continued from previous page)

```
42sh$ ls | wc -w # Count files in a directory
3
```

Here, we use `wc` to count the number of files from the input given by `ls` thanks to the pipe.

4.6 Here strings

Be careful!

Please, note that here strings are not POSIX compliant and **should not** be used in a portable shell script.

Here strings redirect strings to the standard input.

```
42sh$ cat file
hello world
42sh$ wc -w # Count number of words
hello world
2
42sh$ wc -w <<< 'hello world'
2
```

Going further...

Here docs are similar to here strings, but with the ability to have a custom delimiter. Escaping can become useless as the delimiter is user defined.

```
42sh$ wc -l <<'EOF' # EOF  is our delimiter here
this
will
print on
four lines
EOF
4
```

Going further...

Sometimes, you are in the need of redirecting a stream to several outputs at the same time. The program `tee` allows you to do so.

```
42sh$ echo "out" | tee file1 file2
out
42sh$ cat file1
out
42sh$ cat file2
out
```

4.7 Exercises

4.7.1 Goal

Write a script that takes anything as a parameter and redirects it is execution's standard error to the standard output.

```
42sh$ lls
bash: lls: command not found
42sh$ lls 2> file1
42sh$ cat file1
bash: lls: command not found
42sh$ ./script.sh lls
script.sh: line 3: lls: command not found
42sh$ ./script.sh lls 2> file2
script.sh: line 3: lls: command not found
42sh$ cat file2
42sh$
```

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