

Elements of Operating Systems

Chapter 5

Programs versus Processes

J. Fornés M. Farreras Y. Becerra

Barcelona School of Informatics
Universitat Politècnica de Catalunya

Bachelor's Degree in Informatics Engineering — Operating Systems

Preview for readers · October 8, 2026



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

© Universitat Politècnica de Catalunya

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International licence (CC BY-NC-SA 4.0):

«By Universitat Politècnica de Catalunya — BarcelonaTech (UPC), Any 2026»



<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en>

This text was written with the help of Claude, Anthropic's language model. Any errors that remain are the authors'.

5 Programs versus Processes

There are two words that everybody uses as if they were the same one, and they are not. A program is one thing; a process, another. The difference is not a matter of terminology: it is the difference between a sheet of music and a concert.

5.1 A file and a verb

A program is a sequence of instructions and data saved in an ordinary file with execute permission. That permission is the only thing that makes it executable: remember from Chapter 3 that in Unix a filename means nothing*.

Executable files come from two places. From a compiler, as in Chapter 4: `gcc` makes a file with the machine code and some headers that tell the system where everything starts. Or from a text editor, if what you are writing is a shell script: a text file with commands and a `chmod u+x` on top. The first is fast and can do anything; the second takes two minutes to write and can only do what the commands already know how to do.

In both cases, though, what you have on disk is a still thing. A file does nothing.

A **process** is the operating system's representation of a program in execution. The program is what is on the disk; the process is what happens in the machine while it runs, with its own memory, its own registers, its own open files and its own position inside

*A file called `program.exe` without execute permission does not run. One called `notes.txt` with execute permission does — and the system will look at what is inside to work out how.

the code.

From this comes the first consequence, and it is no small one: of the same program you can have as many processes running as you like, and each one is independent of the others. Three windows with the same editor are one program and three processes. What tells them apart is not the code — which is identical — but everything the system has added to it.

5.2 What it takes to set a program going

Running a program means, at the very least:

- **Giving it memory** for the code, for the data and for the stack.
- **Preparing the CPU's registers**, and in particular the program counter, so that it starts running in the right place.
- **Giving it access to the devices**, which the process cannot touch directly because they live in kernel mode: it will only reach them through system calls.
- **Remembering all of this**, because ten milliseconds from now the process will leave the CPU and we have to be able to put it back exactly where it was.

That last point is what forces there to be a data structure.

5.2.1 The PCB

Everything the system needs to know about a process lives in a structure called the *Process Control Block*, or PCB. Exactly what is in it depends on the system and on the machine, but three groups can always be told apart:

- **The address space**: the description of the process's memory regions — code, data, stack — and where they really are. Chapter 8 deals with this.

- **The software context:** the PID, the information the scheduler needs, which devices it has open, the usage statistics.
- **The hardware context:** the contents of the registers and whatever else of the machine — the page table, the program counter — is needed for the process to carry on exactly where it was.

i Careful

The PCB belongs to the kernel, not to you. There is no way for a process to read it or write it directly; what there is are system calls that look at, or change, particular pieces of it, and every one of them checks that you have the right to do so. That boundary is, literally, the operating system.

5.3 Concurrency and parallelism

On an ordinary machine there are dozens of processes at a time, and it is not a whim. While one process waits for the disk to bring it data the CPU has nothing to do; if there is another process ready, the CPU goes to that one. With enough processes the machine never stops, and every user has the impression of having it all to themselves.

But “at a time” means two different things, and they should not be confused.

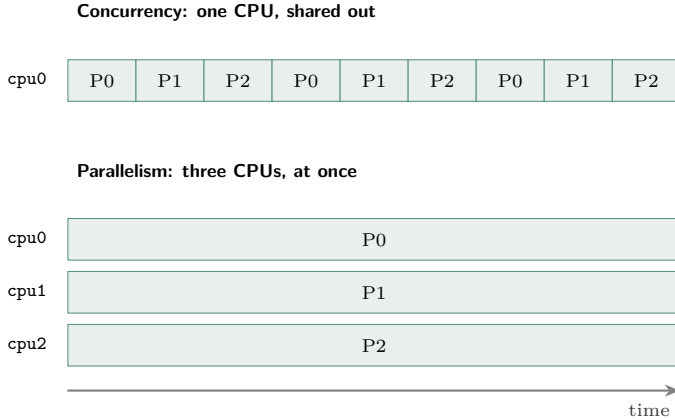


Figure 5.1. The same three processes, seen with one CPU and with three. At the top they take turns; at the bottom they really do advance at the same time.

Several processes are **concurrent** if they have the ability to run in parallel, whether the machine lets them or not. There is **parallelism** when they really do run at the same time, which takes more than one core.

Concurrency is a property of the program: you have it or you don't, depending on how you wrote it. Parallelism is a property of the execution, and it depends on the machine, on the operating system and on whatever else is running at that moment. The same concurrent program, on a single-core machine, will have none of it.

The opposite of concurrent is **sequential**: the processes run one after another because the programmer has forced them to, by synchronising them. We will see this a few pages from now, and the way to do it will be to put a `waitpid()` in the right place.

5.4 The states of a process

If the system has to decide who to give the CPU to, it needs to know who wants it. And not all of them do: a process waiting for a keystroke is no use to the scheduler. That is why the kernel keeps its processes sorted according to what they are doing, and that is what is called the process's **state**.

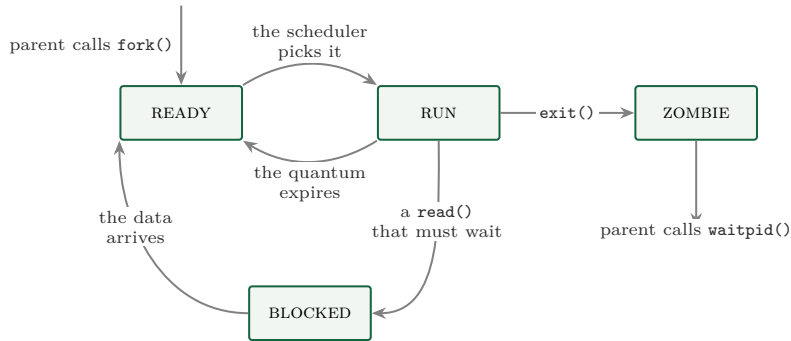


Figure 5.2. The states of a process on Linux, simplified. Each arrow carries the event that causes it.

- **RUN:** the process has a CPU assigned and is running on it.
- **READY:** it is ready to run and is only waiting for its turn.
- **BLOCKED:** it cannot make progress, because it is waiting for data from a device or for an event to arrive. Here it consumes no CPU, and it matters that it doesn't.
- **ZOMBIE:** it has finished, but it has not yet disappeared from the kernel's structures. Why this state is needed we will discuss when we get to `exit()`.

This graph is not universal: it is a consequence of how the system is made. A system that was not multiprogrammed would not need to tell RUN from READY; one with no blocking input/output would not need BLOCKED. When you see a state graph, what you

are looking at is that system's list of design decisions.

Question 1

The step from RUN to READY is the only arrow in the picture that the process does not ask for. Who causes it, and how can the kernel run anything at all if the process has the CPU?

Answer

The scheduler causes it, and it is reached through the clock interrupt. Every few milliseconds the hardware interrupts whatever is running and hands control to the kernel, which takes the chance to see whether that process has used up its turn. Without that interrupt, an infinite loop in a user program would leave the machine hung forever, which is exactly what happened on the systems that had none. Chapter 6 comes back to it.

5.5 Identity, environment and context

A process is not just the program it runs: it is everything that tells one run of that program from the one next to it. On Linux, these properties fall into three groups.

The **identity** says who the process is and, therefore, what it can do. There is the *process id*, or PID, a number the kernel assigns to each new process and which identifies it across the whole system: it is what you will type when you want to send it something. And there are the credentials — the user and the groups the process belongs to —, which are what decide the files and the resources it will be able to reach. We come back to the credentials in the last section of the chapter.

The **environment** is what you gave it when it was born: the command-line parameters, which reach it as the **argv** of Chapter 4, and the environment variables of Chapter 3, which it inherits from whoever created it.

The **context** is all the rest: which state it is in, which files it has open, how much CPU it has used so far, what is in its registers. It is what changes from one instant to the next, and it is what takes up most of the PCB.

```
alumne@labS0:~$ ps -o pid,ppid,user,stat,cmd
  PID   PPID  USER   STAT  CMD
  2841   2839  alumne   Ss    -bash
  3112   2841  alumne   R+    ps -o pid,ppid,user,stat,cmd
```

Look at this output carefully, because everything is in it. `ps` has PID 3112 and its parent is 2841, which is the shell you launched it from. Both are yours. And the **STAT** column says that the shell is asleep (**S**) waiting for `ps` to finish, while `ps` is running (**R**). Everything that follows is how to bring this about from a program.

5.6 Four calls and no more

Unix solves process management with a surprisingly small set of calls:

I want to...	Call
create a process	<code>fork()</code>
change its executable	<code>execvp()</code>
finish	<code>exit()</code>
wait for a child to finish	<code>waitpid()</code>
know my own PID	<code>getpid()</code>
know my parent's PID	<code>getppid()</code>

Of these, `waitpid()` is **blocking**: it can make the process leave the **RUN** state and go into one from which it cannot run, until what it is waiting for happens. This is no misfortune — on the contrary, it is what lets the CPU do something else meanwhile —, but you need to know which calls block, because they are the ones that move you around the picture on the previous page.

5.6.1 `fork()`: a process that makes another

System call `fork`

```
pid_t fork(void);
```

`fork()` creates a new process that is a clone of the one that called it*. A relationship is set up that we will call **parent** and **child**, and since the child can call `fork()` too, what gets generated is a tree.

The child gets a private copy of the parent's memory: the same code, the same data with the same values, the same stack. The physical memory is new†, but the contents are identical. And so is the program counter: the child starts running exactly where the parent was, that is, just on its way out of `fork()`.

*Literally, and the name is no accident: on Linux the one that does the work is `clone()`, with the `SIGCHLD` flag. The kernel has a single call for making new threads of execution—we will see it in Chapter 12—and `fork` is its “copy everything and let a process come out of it” case. That is why in `strace` you will read `clone`.

†Conceptually, at least. Making a real copy would be a waste, and in Chapter 8 we will see the trick every Unix of today saves itself with.

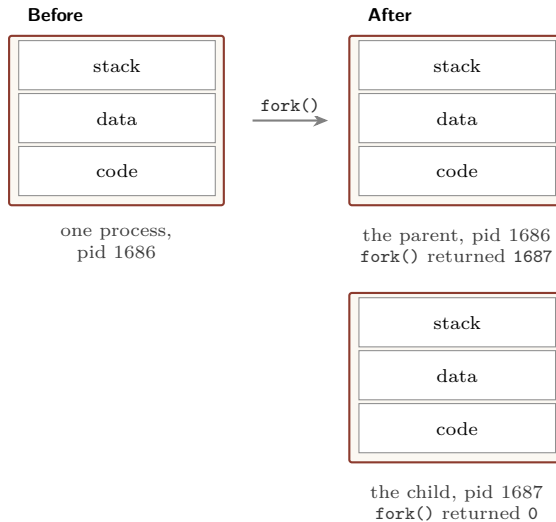


Figure 5.3. A `fork()` seen from memory. Everything is the same on both sides but one thing: the value the call returned to each.

And here comes the one oddity of the mechanism, which is also what makes it work. If the two processes are identical and both carry on from the same line, how can they do different things? Because the call does not return the same thing to both:

- to the **parent** it returns the child's PID;
- to the **child** it returns `0`;
- and if it failed, it returns `-1` and there is no child.

`fork()` is called once and returns twice. That sentence sounds like a joke and it is the literal description of what happens.

The shortest program that shows it is this one:

Listing 5.1. It is called once and writes two lines

```

1  int main(void)
2  {
3      char buf[128];
4      pid_t me, ret;
5
6      me = getpid();                                ❶
7
8      ret = fork();                                  ❷
9      if (ret < 0)
10         error_and_exit("fork", 1);
11
12     sprintf(buf, "was %d; I am %d; parent %d; fork() returns ↵
13         %d\n", (int)me, (int)getpid(), (int)getppid(), (int)ret);
14     write(1, buf, strlen(buf));                    ❸
15     return 0;
16 }

```

- ❶ The PID from before the call. We keep it so that we can compare.
- ❷ From here on there are two processes running this same line of code.
- ❸ And so this `write()` runs twice: once in each process.

```

alumn@labS0:~/S3$ ./proc-fork
was 1686; I am 1686; parent 1607; fork() returns 1687
was 1686; I am 1687; parent 1686; fork() returns 0

```

The second line is the child's, and everything shows in it: its PID is new, its parent is the original process, and the variable `me` — which the parent filled in before the `fork()` — still carries the old value, because the child inherited a copy of it.

Going further

A C function cannot return two values. The trick is that `fork()` does not return two: it returns *one to each process*, and the two processes

do not share registers.

When a program enters the kernel, the kernel saves the program's registers so as to give them back on the way out, and it saves them with the process —it is the PCB's hardware context. A call's return value goes nowhere else either: the kernel writes it into that copy, in the result register of the table in Chapter 2 (Intel's `rax`, ARM's `x0`), and it is on the way out that it reaches the real register.

`fork()` makes the child by copying, among other things, this copy of the parent's registers. And then it makes the one difference: in the parent's copy it leaves the child's PID, and in the child's it puts a zero. On Linux it is one line of the `copy_thread` function, inside each architecture's own code.

The child is not running yet. When the scheduler picks it for the first time, the kernel restores its registers as it would for any other process, the zero reaches the real register, and the child's program carries on just after the call with a `fork()` that has returned 0. Nobody ever ran the `fork()` inside the child: the child is born halfway out.

From this comes the skeleton of everything that follows:

```
ret = fork();
if (ret < 0)                /* it couldn't: there is no child */
    error_and_exit("fork", 1);
if (ret == 0) {
    /* only the child runs this */
} else {
    /* and only the parent, this */
}
/* and both of them, this */
```

5.6.2 What is inherited and what is not

The child is not a copy of everything.

It **inherits** — that is, it gets a private copy of — the whole address space, the devices the parent had open, the user and group credentials, the environment variables and the table that says what is to be done with each signal (Chapter 7).

It **does not inherit** what is by definition its own and nobody else's: the PID and the parent's PID, the CPU usage counters — which start at zero — and any signals and alarms the parent had pending. An event addressed to the parent is addressed to the parent; the child is another process.

i Careful

That the child inherits the *open devices* and not the *memory* is the hardest distinction of the lot, and it is the one that makes Chapter 10 possible. Of the memory it gets a copy: if the parent changes a variable, the child never hears about it. Of the open file descriptors it does not get a copy: they share the read position, so that if one reads, the other carries on where the first left it.

5.6.3 `exit()` and `waitpid()`: finishing, and hearing of it

System call `exit` and `waitpid`

```
void exit(int status);
pid_t waitpid(pid_t pid, int *status, int options);
```

A process can finish voluntarily, by calling `exit()`, or involuntarily, if a signal kills it*. In both cases the system frees its memory, closes its devices and takes back everything it had given it.

All but one thing. Because the process that finishes has something to say — the exit status it passes to `exit()`, which by convention is 0 if all went well —, and that information has to be kept somewhere until somebody collects it. The place is the PCB, and the one who collects it is the parent, with `waitpid()`.

*You will not find these two names in `strace` either. `exit` still has work to do before it clocks off —it runs whatever you left arranged with `atexit` and flushes the buffers— and it ends in an `exit_group()`, which on Linux is what kills the whole process; `waitpid` ends in a `wait4()`.

A process that has finished but whose exit status the parent has not collected yet is a **zombie**. It has no memory, no CPU, nothing: all that is left is the PCB, waiting.

Zombies are amusing, but they take up entries in a kernel table that is finite, and that is why you should always `waitpid()` for the children you create. If a process dies leaving live children or zombies behind, the system gives them a new parent — `init`, process number 1 —, whose jobs include collecting the dead that nobody has claimed.

The two usual uses of `waitpid()` are these:

```
waitpid(-1, NULL, 0);    /* any child          */
waitpid(child, NULL, 0); /* that child and no other */
```

And if instead of `NULL` you pass it the address of an integer, it will leave the exit status there. Careful, though: that integer is not the number the child passed to `exit()`. Two different things are packed into it — how it ended and with what value —, and they have to be unpacked with macros:

```
if (WIFEXITED(status))    /* did it end by itself? */
    code = WEXITSTATUS(status); /* with what value      */
else
    which = WTERMSIG(status); /* which signal killed it */
```

5.6.4 `execvp()`: changing program without changing process

With `fork()` alone you get nowhere: all you can do is run the code you already had a few more times. To run another program you need the call that is missing:

System call `execlp`

```
int execlp(const char *file, const char *arg0, ...);
```

The last thing passed to it has to be a `(char *)NULL`, because that is how it knows where the list ends.

`execlp()` replaces the process's executable with another one*. Note carefully what it does *not* say: it creates no process. The process is the same one, with the same PID and the same parent; what changes is the program inside it.

So the whole address space changes — code, data and stack of the new program — and so does the program counter, which goes back to the first instruction. Nothing that makes up the identity of the process changes: the PID, the credentials, the usage counters, the pending signals. And the signal dispositions go back to their defaults, which makes every sense: the code that set them up is no longer there.

i Careful

If `execlp()` works, it **does not return**. There is nothing to return to: the next instruction belonged to the old program, which no longer exists. So the only line that can come after an `execlp()` is the error handling. If it runs, the call failed.

🔗 Going further

The letters at the end are not decoration. `exec()` is a family of six functions that all do the same thing and differ only in how you hand them the work:

*POSIX defines six exec calls; the Linux kernel, one: `execve()`. The other five are library and they all end up in it. Which they are and how they differ, a few pages further on.

letter	what it means
l	the arguments one by one, ended with a <code>NULL</code> (<i>list</i>): this is the case of this chapter's code
v	the arguments in an array (<i>vector</i>): this is what you need when you don't know how many there will be until the user tells you
p	it looks for the executable along the <code>PATH</code> instead of demanding the whole pathname of you (<i>path</i>)
e	you pass it the environment yourself instead of it inheriting one (<i>environment</i>)

Combining the letters gives `execl()`, `execv()`, `execlp()`, `execvp()`, `execle()` and `execve()`. And all six end up in the same one: the last, `execve()`, is the only one the kernel ever sees, and the other five lay the arguments out for it properly before calling it. You can check this yourselves with `strace`, and Appendix A does.

There is more than one way of making children too. `vfork()` is a `fork()` that copies nothing and leaves the parent stopped until the child does its `exec()`: it came from the days when copying the address space really was expensive, and today, with the lazy copy of Chapter 8, it no longer pays for what it costs to use properly. And `clone()` is the generalisation of both, which is the subject of Chapter 12.

5.6.5 And with that we have a shell

In Chapter 3 we defined the shell as a loop that reads lines and runs programs, and we promised it would fit in thirty lines of C. You now have the pieces to see how it does it.

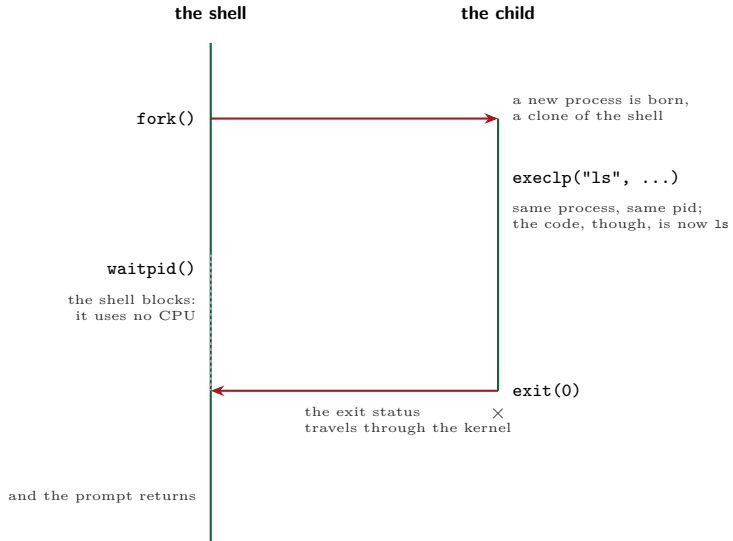


Figure 5.4. What really happens when you type `ls -l`: the shell clones itself, the clone turns into `ls`, and the original waits.

Listing 5.2. The shell, cut to the bone

```

1  int main(void)
2  {
3      pid_t ret;
4      char buf[80];
5
6      ret = fork();
7      if (ret < 0)
8          error_and_exit("fork", 1);
9
10     if (ret == 0) {
11         execlp("ls", "ls", "-l", (char *)NULL);    ❶
12         error_and_exit("execlp", 1);               ❷
13     }
14
15     waitpid(ret, NULL, 0);                          ❸
16     sprintf(buf, "and the prompt comes back\n");

```

```
17     write(1, buf, strlen(buf));
18     return 0;
19 }
```

- ❶ This line never returns, if it works. From here on the child process is no longer running this program but `ls`.
- ❷ And that is why no `exit()` goes here: if control reaches this point it is because `execlp()` failed, and what is needed is to say so.
- ❸ The parent waits for that one child. Without this line the prompt would come back before `ls` had written anything.

⚠ Question 2

In Listing 5.2, what would happen if we took the `if` out and both processes did the `execlp()`?

💡 Answer

The parent would turn into an `ls` as well, and the `waitpid()` on the next line would never run because that line would no longer exist. You would see two directory listings and no prompt: the shell would have vanished, replaced by `ls`. Trying it out from a window you don't care about is instructive.

5.7 Two patterns, and the line that separates them

With these four calls you can already write the two ways of sharing work out among processes. Compare them: the code is the same, and the only difference is where the `waitpid()` goes.

Listing 5.3. Sequential: one at a time

```
1 int main(void)
2 {
```

```

3     pid_t ret;
4     int i;
5
6     for (i = 0; i < NCHILDREN; i++) {
7         if ((ret = fork()) < 0)
8             error_and_exit("fork", 1);
9         if (ret == 0) {                                ❶
10             task(i);
11             exit(0);                                    ❷
12         }
13         waitpid(-1, NULL, 0);                            ❸
14     }
15     return 0;
16 }

```

- ❶ The child goes in here and the parent does not.
- ❷ Without this `exit()` the child would carry on round the loop and make children of its own. We talk about that in the next section.
- ❸ Here, inside the loop: the parent does not create the next child until the previous one has finished.

Listing 5.4. Concurrent: all at once

```

1  int main(void)
2  {
3      pid_t ret;
4      int i;
5
6      for (i = 0; i < NCHILDREN; i++) {
7          if ((ret = fork()) < 0)
8              error_and_exit("fork", 1);
9          if (ret == 0) {
10             task(i);
11             exit(0);
12         }
13     }
14     while (waitpid(-1, NULL, 0) > 0)                    ❶
15         ;
16     return 0;
17 }

```

- ❶ The same wait, but outside the loop and repeated until no child is left. `waitpid()` returns `-1` when there are none, and the loop stops.

```
alumni@labS0:~/S3$ time ./proc-sequential
child 0 (pid 1689) working
child 1 (pid 1690) working
child 2 (pid 1691) working
real    0m3.004s
alumni@labS0:~/S3$ time ./proc-concurrent
child 0 (pid 1693) working
child 1 (pid 1694) working
child 2 (pid 1695) working
real    0m1.003s
```

Each child sleeps for a second. Three seconds against one: that is the whole difference between the two versions, and it is why you have to understand where the line goes.

5.8 The tree that comes out of a loop

Go back to Listing 5.3 and look at balloon ❷. That `exit()` looks like a formality and it is not. Consider this loop, which is the same one without it:

```
for (i = 0; i < 3; i++) {
    ret = fork();
    if (ret == 0)
        doTask();           /* ...and if there is no exit() in ↵
    here? */
}
```

If `doTask` ends with an `exit()`, the child dies right there and the parent creates three children: one per iteration. If it isn't there, the child returns from `doTask`, finds itself inside the loop, and carries on with the iterations the parent had left — making children of its own,

which in turn will do the same.

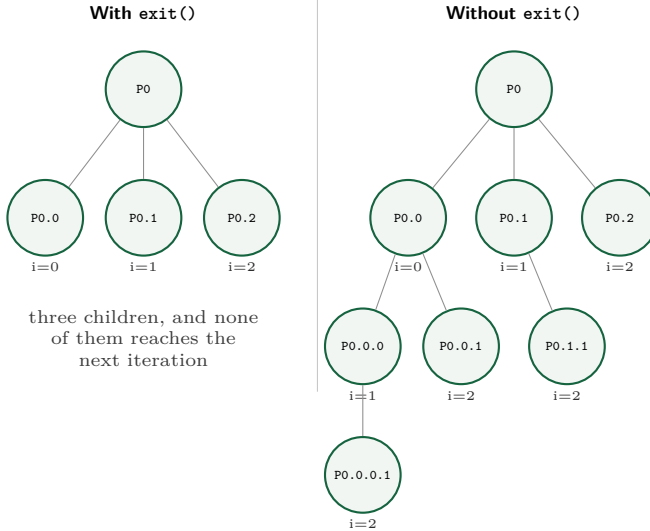


Figure 5.5. The same loop of three `fork()`s, with the child's `exit()` and without it. On the left there are four processes; on the right, eight.

⚠ Question 3

On the right there are eight processes, and the loop goes round three times. Where does the eight come from? And how many would there be with a loop of ten?

💡 Answer

From 2^3 . At every iteration *all* the processes there are at that moment run the `fork()` — the children included, which got there on the turn after the one that created them — and so they double. With a loop of ten there would be $2^{10} = 1024$ of them, which is more than the machine's administrator will let you have.

5.9 A one-line shell

Let's put it all together. This program takes a list of commands, runs them all at the same time and reports how each one ends.

Listing 5.5. fork + exec + waitpid, all together

```
1 static void launch(char *command)           ❶
2 {
3     pid_t ret = fork();
4
5     if (ret < 0)
6         error_and_exit("fork", 1);
7     if (ret == 0) {
8         execlp(command, command, (char *)NULL);
9         error_and_exit("execlp", 1);
10    }
11 }
12
13 static void report(pid_t pid, int status)    ❷
14 {
15     char buf[128];
16
17     if (WIFEXITED(status))
18         sprintf(buf, "process %d ended with exit status %d\n",
19                 (int)pid, WEXITSTATUS(status));
20     else
21         sprintf(buf, "process %d was killed by signal %d\n",
22                 (int)pid, WTERMSIG(status));
23     write(1, buf, strlen(buf));
24 }
25
26 int main(int argc, char *argv[])
27 {
28     pid_t pid;
29     int i, status;
30
31     if (argc < 2)
32         Usage(argv[0]);
33
34     for (i = 1; i < argc; i++)
35         launch(argv[i]);
36 }
```

```

37     while ((pid = waitpid(-1, &status, 0)) > 0)      ❸
38         report(pid, status);
39     return 0;
40 }

```

- ❶ Create and mutate, in five lines. Note that the parent does not wait: it goes straight back to launching the next command.
- ❷ Unpacking the exit status. A program that ends well and one that has been killed are not reported in the same way.
- ❸ The wait, once they have all been launched. The order they finish in has nothing to do with the order you wrote them in.

```

alumne@labS0:~/S3$ ./proc-launcher date whoami
alumne
Wed Sep  2 17:14:49 UTC 2026
process 1701 ended with exit status 0
process 1700 ended with exit status 0

```

Look at the order of the first two lines: `whoami` wrote before `date`, even though it was launched after. There is no mistake here. They are two concurrent processes, and which of them writes first depends on the scheduler, not on you. If your program needs things to be otherwise, it has to say so; and saying so means synchronising them, which is what Chapter 7 is about.

🔗 Going further

The same question always comes up here, and it is a good one: **why two calls?** If what everybody wants is to start a program, why not a single one that does the lot?

There is one, and it is called `posix_spawn()`. It takes the program, the arguments and the environment, and hands you back the PID of the child, already started.

But look at what there is between the `fork()` and the `exec()` of this chapter's programs: *code of your own* fits in there, running inside the child already and still with your program in memory. Those few

lines are where the shell closes file descriptors, redirects them, changes directory, sets the *nice* value or lets go of privileges —and the whole of Chapter 9 lives in that gap. With a single call there is no gap: everything you want to do in between you have to know how to ask for as a parameter, and `posix_spawn()` has a long list of them for exactly that reason, one that will never be as long as “anything that might occur to you”.

So the split is not an oversight from the seventies: it is what makes the child programmable. The price is the copy of the address space, and you have already seen that today it is lazy and costs next to nothing. Both are POSIX functions: the standard fixes the interface and never says a word about how it is implemented. That is why in this book both of them are system calls. Underneath, though, each kernel has decided for itself which of them deserves an entry in the system call table:

Really a system call?

	Linux	macOS
<code>fork()</code>	no: libc does a <code>clone()</code> with <code>SIGCHLD</code>	yes: number 2
<code>posix_spawn()</code>	no: libc does a <code>clone()</code> and an <code>execve()</code>	yes: number 244

And this difference explains something too. On macOS, where a process is a Mach task and making one is expensive, the fast path is the single call; on Linux, where `fork()` was cheap already, there was no need for it. The same POSIX on top, two opposite decisions underneath.

And there is a third path, the one half the world runs on today, and it did not have to be invented either. The same `clone()` that sits under `fork()` has a second family of flags that say not what is *shared* with the parent but what stops being *visible*: the process table, the file tree, the network. A program started that way is still an ordinary `fork()` and `exec()`, and everybody calls it a container. Section 14.6 covers both halves of it.

5.10 What the kernel does, meanwhile

So far, the mechanism seen from outside. Now to see, if only in outline, what happens on the other side of each call.

fork() looks for a free PCB and reserves it; initialises in it what is new (the PID, the counters at zero); applies the memory policy to give it an address space with a copy of the parent's; updates the input/output structures so that the child has the same file descriptors open; and adds the process to the queue of those that want CPU.

execlp() replaces the address space with the new program's, and reinitialises the PCB fields tied to it: the signal dispositions, the registers, the program counter. It neither touches the PCB as a table entry nor takes it out of any queue: the process is the same one and still wants CPU.

exit() frees all the resources — memory, devices —, saves the exit status in the PCB and takes the process out of the queue of runnable ones, so that it will not be given CPU again. The PCB, though, stays. Then it calls the scheduler, because the CPU has just been left free.

waitpid() looks for the child it was asked about. If it was a zombie already, it collects its exit status, frees the PCB and returns straight away. If it is still alive, it blocks the parent — from RUN to BLOCKED, the arrow in Figure 5.2 — and calls the scheduler.

Three things turn up there that this chapter has kept leaving for later: the queues, the scheduler and the memory policy. The first two are Chapter 6; the third, Chapter 8.

5.11 Who can do what

One piece of the process's identity is left hanging, and it is the one that decides whether all these calls will let you do anything at all.

Every process carries a user and some groups, inherited from whoever created it, and the kernel uses them to answer the same question every time: can this process touch this object? For files, the answer comes from the permissions we read in Chapter 3 and that Chapter 11 explains in full. For processes, the same rule holds: you can only send signals to, or look inside, the processes of your own user.

The exception is **root**, who gets everywhere and can carry out the system's privileged operations. Hence the practical problem: there are things an ordinary user has to be able to do that mean touching files that are not theirs. Changing your own password, for instance, means writing to `/etc/shadow`, which belongs to **root** and which it does you no good at all for anybody else to own.

Unix solves this with a bit in the executable file's permissions, the *setuid*: when a file carries it, the process that comes out of it runs not with the credentials of whoever launched it but with those of the program's owner.

```
alumni@labS0:~$ ls -l /usr/bin/passwd
-rwsr-xr-x. 1 root shadow 89184 May  5 12:38 /usr/bin/passwd
```

That **s** where you expected the **x** is the whole mechanism*. **passwd** belongs to **root**, and so, while it runs, your process *is* **root** and can write wherever it has to†.

Careful

A *setuid* program is, by definition, a controlled hole in the system's protection. If it has a bug, the bug runs with the owner's privileges, not with yours. A good many of the classic attacks against Unix

*The dot just after the permissions has nothing to do with it: it says the file carries an extra security label (SELinux's), which we will not be using in this book.

†In the lab it will not get your password changed. The `/etc/passwd` and `/etc/shadow` of those machines only hold the local users: your accounts are validated on another server, and the disk is reinstalled every time the computer is switched on. The mechanism is the same; the file it would write to is not.

are exactly that: finding a *setuid* **root** program that can be made to fail in a convenient way. That is why there have to be few of them, and they have to be short.

Going further

The *setuid* bit is not magic: what it does is give the process **two numbers instead of one**. The *real UID* is who you are—who launched the program—and the *effective UID* is who the system pretends you are when it decides whether to let you touch a file. Normally they are the same number; in a *setuid* program, they are not. `getuid()` tells you the first and `geteuid()` the second:

System call `getuid` and `geteuid`

```
uid_t getuid(void);      /* who you are */
uid_t geteuid(void);     /* who you may be */
```

And there is a third one, `setuid()`, which is there for the most important thing a well-written privileged program does: **letting go of the privileges once it no longer needs them**. The `codi-en/proc-credentials.c` is an eight-line program that prints both of them, lets them go and prints them again. To try it you have to be the machine's administrator, which in the lab you are not; this session is from a machine where you are:

```
$ sudo chown root proc-credentials
$ sudo chmod u+s proc-credentials
$ ./proc-credentials
before:  real=1000  effective=0
after:   real=1000  effective=1000
```

The effective one on the first line is a zero: for as long as it lasts, that process *is* **root**. The one on the second is not any more, and **there**

is no way back: when the effective one is zero, `setuid()` changes all three UIDs—real, effective and saved—and shuts the door. That is the virtue, not the limitation: a well-written `passwd` opens `/etc/shadow` straight away and goes back to being you before it looks at anything you have typed, so that whatever bug it has further down no longer runs as `root`.

Protection, in any case, is an internal problem: it is about deciding who can touch what inside a machine that works. Security is another thing, and a bigger one: it takes in the physical place the machine is in, the people who have access to it, the operating system and, above all today, the network. This book will give you only the first.

5.12 Exercises

1. Run `proc-fork` a few times in a row. The PIDs change; does the difference between the parent's and the child's always change? And the order the two lines come out in?
2. How many processes does this fragment create? Draw its tree.

```
fork();  
fork();  
fork();
```

3. Modify `proc-fork` so that the child writes `getppid()` after a `sleep(2)` and the parent does nothing but finish immediately. What does the child's `getppid` answer if you wait long enough? (Hint: look at `ps` while it happens.)
4. Take the `waitpid()` out of `proc-sequential` and leave the `exit()` in. Which of the two programs do you get? And if you take the `exit()` out and leave the `waitpid()` in?
5. Make a child of `proc-launcher` finish with an exit status other than 0 — by asking it to run a command that does not exist,

for instance — and see what it reports. Then kill one of them from another terminal with `kill` and compare.

6. Write a program that creates N processes in a chain: each child creates a child and waits for it to finish, down to depth N . How many lines' difference is there from `proc-sequential`?