Chapter 5: Processes, Kernel, System Calls

In Unix, every application program runs in a dedicated execution environment called a **process**. The OS gives each process its own virtualized processor and its own virtualized memory space. Within this safe sandbox, the program has no restrictions on how it uses their CPU and RAM. However, any interaction with the outside world must go through the OS: showing text on screen, reading a file, opening a network connection, etc.
All these **Input**/**Output** operations are performed by the operating system's **kernel**, a special program with complete control over everything in the system.
The process must use a special CPU instruction called a **system call**, typically through a wrapper function.

1 Warm-up, simple syscalls: sleep() and getpid()

Exercise The `sleep()` syscall pauses execution for a given number of seconds. A unix command named `sleep` exists to offer this feature directly in the shell (it's mostly useful in script programs). Try it with `sleep 5` then read the doc with `man 1 sleep` or just `man sleep`.

Exercise We now want to implement a similar command, which also prints its process identifier (PID) and the remaining duration every second. The expected behavior is illustrated below. Read the two syscall docs with `man 3 sleep` and `man getpid`. Write a `countdown.c` program which expects just one command-line argument and sleeps that many times for one second. To convert `argv[1]` to an integer, use the `atoi()` function from `stdlib.h`.

```
$ ./countdown 5
41098: start
41098: 5
41098: 4
41098: 3
41098: 2
41098: 1
41098: end
```

2 Creating processes with fork()

Exercise Write a `forky.c` program which prints its PID, then spawns a child process, then prints its PID again (from both processes). The expected output is illustrated below.

```
$ ./forky
54365: hello world
54365: I am the parent
54366: I am the child
```

Exercise Add a global `int` variable to `forky` and print its value (with `"%d"`) and address (with `"%p"`) before the call to `fork()`. After the fork, decrement the variable in the parent and increment it in the child. Then, print its value and address again (in both processes). Observe how the two processes see distinct values at the same memory address. Do the same but with a local variable. Ask us questions until you're comfortable with what you're observing.

Exercise Read the code below without executing it. How many times does it print “X” in total ?

```
main()
{
  fork();
  if ( fork() )
  {
     fork();
  }
  printf("X\n");
}
```

3 Executing programs with exec()

From within a process, one can ask the kernel to change the program that is currently executed. This can be done by using the `exec` system call. This is available with different functions, each of which with slightly different interfaces and behavior: execl, execle, execlp, execv, execvp ... This call never returns: on the contrary, the process simply forget everything it was doing, and starts executing the newly designated program, from the start. It’s nothing like a temporary replacement. It’s final: when the new program will terminate, the process will terminate as well and will *never* go back to execute the previous program.

Exercise What does this program print ? If unsure, type it in and execute it.

```
int main(void)
{
    printf("A\n");
    execl("./countdown", "./countdown", "5", NULL);
    printf("A\n");
    return 0;
}
```

Remarks

- The code above uses the `exec` syscall. Go and read the documentation, so type `man 3 exec` or go and see the page here: `https://www.gnu.org/software/libc/manual/html_node/Executing-a-File.html`
- Our example uses `execl()`, whose parameters should be:
  - the path to the new executable file, either absolute (eg . `"/dir/prog"`) or relative (eg `"./prog"` or `"../dir/prog"`)
  - command line arguments (including argument number 0 which, by convention is the *name* of the program),
  - and a NULL pointer to mark the end of the argument list.

Exercise Write a `doublecount.c` program which forks into two processes, and then each process `exec()`utes program `countdown` with a different parameter e.g. 2 and 4, or 5 and 1. Swap these parameters and observe how the shell always waits for the “main” process (its own child) but not for the child process (the shell’s grandchild), as illustrated below on the right.

```
$ ./doublerebours
5338: start
5338: 2
5337: start
5337: 4
5338: 1
5337: 3
5338: finish
5337: 2
5337: 1
5337: finish
$
```

```
$ ./doublerebours
8599: start
8598: start
8599: 5
8598: 1
8598: finish
8599: 4
$ 80599: 3
8599: 2
8599: 1
8599: finish
```

Remarks

- If the parent process terminates *before* the child process, your program will output something like the listing on right, in the above picture: the shell is able to print its own prompt before our command is actually done executing.
- To clean the terminal, you can type `ENTER` several times to force it to show its prompt correctly again.
- This is not a bug: the process executes *in the background* compared to the shell. Some commands in the linux environment are even designed to be used in this way.
- However, this is not the expected behavior for our program. The next exercise explains how to solve this.

4 Waiting for a child with wait()

The goal of this part is to implement a C program that will be called `parexec`. This program takes as argument from the command line the name of another program `prog`, followed by an arbitrary long list of arguments. It executes `prog` in parallel (in separate processes) over each of its arguments. For example, typing `./parexec gzip file1 file2 ...  fileN` from the command line will launch in parallel the commands `gzip file1`, `gzip file2` ... `gzip fileN`.

Exercice Write a `parexec.c` program. You will use the system calls we have seen so far. The name of the program (`prog`) to be executed in parallel as well as all corresponding arguments are received by the program from the command line (through `argc`, as we have seen in the previous chapter). Beware, we want `parexec` to give back control to the shell only *after* all executions of `prog` are over. This is illustrated below with `countdown`. To help you, please also read the remarks below.

```
% ./parexec ./rebours 4
28393: start
28393: 4
28393: 3
28393: 2
28393: 1
28393: finish
%
```

```
% ./parexec ./rebours 3 6
41035: start
41035: 6
41034: start
41034: 3
41035: 5
41034: 2
41034: 1
41035: 4
41034: finish
41035: 3
41035: 2
41035: 1
41035: finish
%
```

```
% ./parexec ./rebours 1 2 3
57371: start
57372: start
57371: 2
57372: 3
57370: start
57370: 1
57371: 1
57370: finish
57372: 2
57371: finish
57372: 1
57372: finish
%
```

Remarks

- To wait for the child processes to terminate, use the `wait()` system call. You'll find its documentation by typing `man 2 wait` or on the web here: `https://www.gnu.org/software/libc/manual/html_node/Process-Completion.html#index-wait`
- Beware: `wait()` *only* works on direct children processes only, not on other descendants (eg grand-children).
- Note that our usage of `wait` doesn't require any argument, so you can simply write `wait(NULL);`
- Beware as well of the fact that, as said in the documentation, this primitive "*is used to wait until any one child process terminates*". Since you eventually want to wait for the termination of several children processes, you will need to call `wait(NULL)` the right number of times.