Chapter 6: 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 pass through the OS: showing text on screen, reading a file, opening a network connection, talking to other processes, etc. All these so-called **Input**/**Output** operations are performed by the **OS kernel**, a special program with complete control over the machine. To request such a service from the the kernel, a process must use a special CPU instruction named a **system call** or syscall. It would be quite hard to do this by hand, but fortunately all programming languages provide convenience wrappers for all syscalls. In C, they look like ordinary function calls.

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

Exercise 1 The `sleep()` syscall **pauses execution** of the calling process 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 2 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`. Then, write a `countdown.c` program which expects just one command-line argument and sleeps that many times for one second. To interpret the string `argv[1]` as 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
```

Exercise 3 Process termination happens through the `exit()` system call. Observe how you can call it from anywhere in your program to end execution. This is equivalent to (and implemented in the same way as) simply returning from the `main()` function.

Exercise 4 The argument to `exit()` is refered to as the **exit status**, an unsigned 8-bit value that will be passed back to the shell. The convention is that returning zero means a success, any non-zero value indicates an error. Make sure your program from the previous question produces a recognizable exit status, e.g. 42. Run your program, and then observe its exit status from the shell with the special command `echo $?`.

Remarks If you want to learn more,
- read the `EXIT STATUS` portion of `man bash`
- or visit `https://en.wikipedia.org/wiki/Exit_status#POSIX`

2 Creating processes with fork()

Process creation in Unix is a bit counter-intuitive, because creating new processes and executing new programs actually happen via two distinct system calls, respectively named `fork()` and `exec()`. Calling `fork()` causes the **current process to be duplicated**. After the syscall is done, there are two distinct processes executing the same program, hence the saying «call once, return twice».

Exercise 5 Read each program below and guess how many messages it will print. Then, check your answers by typing in, compiling, and then running them. Note: the `fork()` syscall is provided by `unistd.h`. Go read `man 2 fork` while you're at it.

```
main()
{
   fork();
   printf("hi !\n");
   exit(0);
}
```

```
main()
{
   fork();
   fork();
   printf("hi !\n");
   exit(0);
}
```

```
main()
{
   fork();
   fork();
   fork();
   printf("hi !\n");
   exit(0);
}
```

When a process calls the `fork()` system call, the kernel creates a so-called **child process** which starts as an exact copy of the **parent process**. This operation duplicates the whole **execution state**: the child process begins its life by returning from `fork()`, not from the top of the program. To help the code know by which process it is being executed, the system call returns different values:

- in the newly created process (aka the child) the `fork()` syscall returns zero.
- in the original process (aka the parent) the `fork()` syscall returns the PID of the child.

Exercise 6 Write a `forky.c` program which prints its PID, then spawns a child process, then prints its PID again, but from both processes. The expected output is illustrated below. Note how we don't care about the **ordering between parent and child**. Actually, we don't even have a way to control it: every process gets its own virtual CPU, so on a modern (multicore) machine, both processes might very well be executed simultaneously ! This is an example of **concurrency**.

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

```
% ./forky
12629: hello world
12631: I am the child
12629: I am the parent
```

Exercise 7 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). Do the same but with a local variable. Observe how the two processes see distinct values at the same memory address. This is possible because the kernel maintains a unique **virtual memory space** for each process. Ask us questions until you're comfortable with what you're observing.

Exercise 8 Remember that in C, booleans are implemented as integers with False = zero and True = anything non-zero. As a result, writing `int x = f(); if(x != 0) ...` is exactly equivalent to `int x = f(); if(x) ...` and also to just writing `if(f()) ...`

Now 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 by the process. This happens through the `exec()` system call, available in many variations with slightly different interfaces and behavior: execl, execle, execlp, execv, execvp ... Today we will use `execl()` because it best matches our use-case.
A call to `exec()` never returns: the process simply **forgets everything** it was doing before, and **starts executing** the newly designated program file, from the top. This is not a "temporary" replacement, or a sub-program etc: if the new program terminates, then our process terminates. In any case, we will *never* return back to the original call site in the previous program.

Exercise 9 What does this program print ? If unsure, type it in and execute it. To help you, please also read the remarks below.

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

Remarks

- Type `man 3 exec` and/or read the docuemntation online: `https://www.gnu.org/software/libc/manual/html_node/Executing-a-File.html`
- Our example uses `execl()`, whose parameters should be:
  - first, the path to the new executable file, either absolute (eg . *"/dir/prog"*) or relative (eg *"./prog"* or *"../dir/prog"*)
  - then, its command line arguments (including argument number 0 which, by convention is the *name* of the program),
    - This means that the first two parameters of exec are typically identical, like in our example.
  - and finally a NULL pointer to mark the end of the argument list.

Exercise 10 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. To help you, please also read the remarks below.

```
% ./doublecount
5338: start
5338: 2
5337: start
5337: 4
5338: 1
5337: 3
5338: finish
5337: 2
5337: 1
5337: finish
%
```

```
% ./doublecount
8599: start
8598: start
8599: 5
8598: 1
8598: finish
8599: 4
% 8599: 3
8599: 2
8599: 1
8599: finish
```

Remarks

- If the parent process terminates *before* the child process, then you will observe something like the listing on right: the shell prints its next prompt (illustrated here by a percent sign) before our command is actually done executing.
- This is not a bug: a the process can execute *in the background* compared to the shell. Some Unix commands are even designed to be used in this way.
- However, this is not always what we want. In the next exercise you will learn how to control this feature.
- To clean your terminal, you can press `ENTER` several times, which will force the shell to show its prompt again, or use the `clear` command, or press `ctrl-L`.

4 Waiting for a child with wait()

The goal of this part is to implement a “parallel execution” tool, which we will name `parexec`. On the command-line, `parexec` expects the name of another program e.g. `prog`, followed by an arbitrary long list of arguments. Its purpose is to execute `prog` in parallel (in separate child processes) for every argument. 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 Implement the `parexec.c` command. The name `prog` of the program to be executed, as well as all corresponding arguments are received from the command line through `argc/argv`. Please note that we want `parexec` to wait until all executions of `prog` are over, *before* returning to the shell. This is illustrated below with `countdown`. To help you, please also read the remarks below.

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

```
% ./parexec ./count 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 ./count 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.