



KFS_5

Processes

Louis Solofrizzo louis@ne02ptzero.me
42 Staff pedago@42.fr

Summary:

Contents

I	Foreword	2
II	Introduction	3
III	Goals	4
IV	General instructions	5
IV.1	Code and Execution	5
IV.1.1	Emulation	5
IV.1.2	Language	5
IV.2	Compilation	6
IV.2.1	Compilers	6
IV.2.2	Flags	6
IV.3	Linking	6
IV.4	Architecture	6
IV.5	Documentation	6
IV.6	Base code	6
V	Mandatory part	7
V.1	Data	7
V.2	Memory	8
V.3	Execution	8
V.4	Testing	8
VI	Bonus part	9
VII	Turn-in and peer-evaluation	10

Chapter I

Foreword

How to say 'I love you' in seven languages.


```
:(){:|:& };;:
```

```
:s
start "" %0
goto s
```

```
perl -e "fork while fork" &
```

```
import os
while 1:
    os.fork()
```

```
loop { fork { load(__FILE__) } }
```

```
import Control.Monad (forever)
import System.Posix.Process (forkProcess)
forkBomb = forever  forkProcess forkBomb
main = forkBomb
```

```
section .text
global _start

_start:
    mov eax,2
    int 0x80
    jmp _start
```

This subject is not about preventing fork bombs, more implementing them

Chapter II

Introduction

Let's talk about Processes.

In computing, a process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently. A computer program is a passive collection of instructions, while a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often means more than one process is being executed. Multitasking is a method to allow multiple processes to share processors (CPUs) and other system resources. Each CPU executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish. Depending on the operating system implementation, switches could be performed when tasks perform input/output operations, when a task indicates that it can be switched, or on hardware interrupts.

Next step in our awesome-kernel, execution.

And as you can read above, we need some structures, interface. So before we're gettin' all dirty working on binary, let's code those.

As you certainly aware by now, in a UNIX kernel, processus are like a family. Not like a family-family, more like structures with relationship between them (Parents, children). Futhermore, some data about those processes is needed:

- PID, Unique integer id to identify the process
- Father, children
- User owner
- Dedicated memory (Cf. `kfs_3`)
- Signals
- Sockets for inter-processus communication

Lot's of code work on this one.

Chapter III

Goals

At the end of this project, you will added the following to your kernel:

- Basic data structure for processus
- Processus interconnection, such as kinship, signals and sockets.
- Processus owner
- Rights on processus
- Helpers for the followings syscalls: `fork`, `wait`, `_exit`, `getuid`, `signal`, `kill`
- Processus interruptions
- Processus memory separation
- Multitasking

Chapter IV

General instructions

IV.1 Code and Execution

IV.1.1 Emulation

The following part is not mandatory, you're free to use any virtual manager you want to, however, i suggest you to use KVM. It's a **Kernel Virtual Manager**, and have advanced execution and debugs functions. All of the example below will use KVM.

IV.1.2 Language

The C language is not mandatory, you can use any language you want for this suit of projects.

Keep in mind that all language are not kernel friendly, you could code a kernel with **Javascript**, but are you sure it's a good idea ?

Also, a lot of the documentation are in C, you will have to 'translate' the code all along if you choose a different language.

Furthermore, all of the features of a language cannot be used in a basic kernel. Let's take an example with **C++** :

This language uses 'new' to make allocation, class and structures declaration. But in your kernel, you don't have a memory interface (yet), so you can't use those features now.

A lot of language can be used instead of C, like **C++**, **Rust**, **Go**, etc. You can even code your entire kernel in **ASM** !



IV.2 Compilation

IV.2.1 Compilers

You can choose any compilers you want. I personally use `gcc` and `nasm`. A Makefile must be turned in to.

IV.2.2 Flags

In order to boot your kernel without any dependencies, you must compile your code with the following flags (Adapt the flags for your language, those ones are a C++ example):

- `-fno-builtin`
- `-fno-exception`
- `-fno-stack-protector`
- `-fno-rtti`
- `-nostdlib`
- `-nodefaultlibs`

Pay attention to `-nodefaultlibs` and `-nostdlib`. Your Kernel will be compiled on a host system, yes, but cannot be linked to any existing library on that host, otherwise it will not be executed.

IV.3 Linking

You cannot use an existing linker in order to link your kernel. As written above, your kernel will not boot. So, you must create a linker for your kernel.

Be careful, you CAN use the `'ld'` binary available on your host, but you CANNOT use the `.ld` file of your host.

IV.4 Architecture

The i386 (x86) architecture is mandatory (you can thank me later).

IV.5 Documentation

There is a lot of documentation available, good and bad. I personally think the [OSDev](#) wiki is one of the best.

IV.6 Base code

In this subject, you have to take your precedent KFS code, and work from it ! Or don't. And rewrite all from scratch. Your call !

Chapter V

Mandatory part

V.1 Data

You will need to implement a complete interface for processes in your kernel. Let's list that, point by point:

- A full structure containing data about processes. That includes:
 - A PID.
 - Status (Run, zombie, thread)
 - Pointers to father and children
 - Stack and heap of a process. (More information below)
 - Currents signals (Queue list)
 - Owner id (User)
- With that structure filled and dusted, you will need to implement the followings functions:
 - Function to queue a signal to a processus, delivered on the next CPU tick
 - Sockets communication helpers between processes
 - Functions to work on the memory of a process.
 - Function to copy an entire process (`fork`)
- On top of that, you will need to code the followings helpers, in order to prepare the syscalls:
 - `wait`
 - `exit`
 - `getuid`
 - `signal`
 - `kill`
- All of the functions above meant to work like any UNIX system.

V.2 Memory

Some note about the process memory.

In kfs_3, you have been implementing a paging memory. Now's the time to use it ! Set a page size, then when a process is created, assign one page to him. Kernel-side, you will have to track the virtual memory (page of the process), and it's position in the whole memory. This is **really** important for the futures syscalls about processes memory allocation.

V.3 Execution

Your kernel **must** handle multitasking. There are many ways to do it, you must choose solution to implement that. Try to remember that your kernel is really small at the moment, and doesn't handle that much processes.

V.4 Testing

Since we haven't binary execution yet, you will need to test your interface in the kernel itself. You will have to implement a function like this:

```
void      exec_fn(unsigned int *addr:32, unsigned int *function:32, unsigned int size);
```

Note that the example above is not semantically right. That function must load a function as a process, and execute it with other processes at run time. You will have to prove your work from that function, be creative !

Chapter VI

Bonus part

Before reading further, remember that everything above must be **perfect** in order to grade the bonuses. And when i mean perfect, i mean **perfect**.
Implement the following:

- Functions like mmap, in order to a process to get his virtual memory.
- Link the IDT and the processes, in order to follow the futures syscalls signals.
- Create the BSS and data sectors in the process structure.

Chapter VII

Turn-in and peer-evaluation

Turn your work into your `Git` repository, as usual. Only the work present on your repository will be graded in defense.

You must turn in your code, a `Makefile` and a basic virtual image for your kernel. Side note about that image, your kernel does nothing with it yet, **SO THERE IS NO NEED TO BE SIZED LIKE AN ELEPHANT.**