# INFO0940-1: Operating Systems (Groups of 2)

## Research Project 1: Adding System Calls

Prof. L. Mathy - G. Gain - V. Rossetto

Submission before Wednesday, April 19, 2023 (23:59:59)

## 1 Introduction

Virtual memory is an abstraction which allows to create the illusion of a very large main memory. To manage memory efficiently, the kernel divides the virtual addressing of a process into various blocks of fixed size (by default 4KB), called pages.

In Linux, when a process creates a copy of itself, the pages in memory that are to be modified either by the process or by its copy are marked Copy-on-Write (CoW).

In this assignment, you will implement 3 system calls to retrieve information about CoW page faults of specific processes in the Linux kernel.

## 2 Assignment

For this project, you will develop several features within the Linux kernel that implement the following system calls as an interface for user space applications.

```c
long pf_set_param(const char *process_name, size_t name_len);
long pf_get_info(const char *process_name, size_t name_len, struct
    pf_stat *pf);
long pf_cleanup(void);
```

We provide a user space program[1] to test this interface. You can bring some slight modifications to the code but do not modify directly the syscalls number otherwise the program will not work anymore.

[1] available on eCampus

### 2.1 `pf_set_param`

This system call will initialise a global data structure (if not already set) and will track a particular process (according to the `process_name` argument) in order to monitor specific information that will be retrieved later via the system call `pf_get_info`. Note that several processes (with different names) can be tracked simultaneously. Furthermore, calling this system on an already tracked process will have no effect.

**Arguments and return value specification:**

- `process_name`: pointer to a string that represents the name of the process(es) to analyze.
- `name_len`: length of the `process_name` string.
- *Return value:* On success, it returns 0. On failure, it returns a specific error code according to the error caused (see Section 2.6).

### 2.2 `pf_get_info`

This system call will allow to retrieve a `pf_stat` structure that will be set by the kernel. This structure contains 5 different fields and has the following form:

```

// Per simplicity, a maximum size is imposed
#define MAX_SIZE_PF 128

struct pf_stat{
 // Number of VMA of a specific process
 int nb_vma;
 // Number of CoW page faults of a specific process
 long cow_page_faults;
 // Array that contains CoW page faults per VMA
 long vma_fault[MAX_SIZE_PF];
 // Array that contains the start address of each VMA
 unsigned long vma_list_start[MAX_SIZE_PF];
 // Array that contains the end address of each VMA
 unsigned long vma_list_end[MAX_SIZE_PF];
};
```

This structure contains memory-specific information about a particular process such as the number of Virtual Memory Areas (VMAs), the total number of CoW page faults, the number of CoW page faults per VMA, etc. Some of this information will be populated when the page fault is processed by the page fault handler (see Section 2.5). Note that values of the `pf_stat` structure are reset if a process with the same name is executed several times.

In this case, the structure contains the information of the last executed process.

Note that a VMA is processor-independent structure that consists of a number of pages, each of which has an entry in the page table. The kernel uses VMA to keep track of the process's memory mappings; for example, a process has one VMA for its code, one for the heap, etc. Each VMA has a start address and an end address.

The `vma_fault` field represents the number of CoW page faults per VMA. For example, if we have a process that has 5 VMAs, and its first three VMAs contain 1, 3 and 7 CoW page faults respectively, we will have the following array: $\{1, 3, 7, 0, 0\}$. A more detailed example is given in the Appendix A.

**Arguments and return value specification:**

- `process_name`: pointer to a string that represents the name of the process(es) to analyze.
- `name_len`: length of the `process_name` string.
- `pf_stat`: pointer to a structure that contains memory-specific information about a particular process.
- *Return value:* On success, it returns 0. On failure, it returns a specific error code according to the error caused (see Section 2.6).

## 2.3 `pf_cleanup`

This system call free up any kernel resources that have been previously allocated by the previous system call(s).

**Arguments and return value specification:**

- This system call takes no argument.
- *Return value:* On success, it returns 0. On failure, it returns a specific error code according to the error caused (see Section 2.6).

## 2.4 System Call Numbers

The system call number assignment is as follows (it is not allowed to change them):

```
385: pf_set_param
386: pf_get_info
387: pf_cleanup
```

### 2.5 Page fault handler

To intercept CoW page faults of a specific process, you must investigate in order to find the page fault handler (function) of the underlying architecture (x86 32bits). Once the page fault handler is set up, it will be necessary to process CoW page faults of specific processes and save them in a global data structure that can be manipulated within the system call `pf_get_info`.

### 2.6 Errors Handling

In the kernel, it is necessary to use specific errors so that users can know what causes a system call not to be completed. To do this, the variable `errno` is used. Depending on the error, your implementation must return an appropriate error code. For instance, a memory allocation error will return an -*ENOMEM* error code (out of memory).

Furthermore, your implementation must display error messages with `KERN_ERR` and by printing the following prefix "`[INFO940][ERROR]`" on the kernel log. It is necessary to trace **all** errors. It is up to you to know which one is the most suitable according to the error encountered[2]. In addition, the given helper program can help a lot.

## 3 Generic Requirements

The following requirements also have to be satisfied:

- You can use any type of data structures such as list, tree, and hash tables but do not reinvent the wheel.
- You must only consider **x86** architecture (32bits);
- Your code must be readable. Use common naming conventions for variable names and functions. In addition, your patch(es) must also be clean and must be easy to read it/them;
- Your code must be robust and must not crash. Cleaning must be properly managed;
- You can use the kernel log for debugging but do not forget to remove all `printk` that are not related to errors otherwise it will generate some overhead in the kernel;
- Do not consider huge pages, consider only 4KB pages. Also do not consider kernel page faults, only user x86 page faults must be taken into account.

[2] `https://elixir.bootlin.com/linux/v4.15/source/include/uapi/asm-generic/errno-base.h`

## 4 Report

You are asked to write a very short report (**max 2 pages**) in which you explain your implementation. In addition, briefly explain how the memory of a process is managed by the kernel.

Note that, you can draw diagrams to illustrate your explanations but note that these ones **will not** be considered as appendices.

Finally, we would also appreciate an estimation of the time you passed on the assignment, and what were the main difficulties that you have encountered.

## 5 Evaluation and tests

Your program can be tested on the submission platform. A set of automatic and basic tests will allow you to check if your program satisfies the basic requirements. Note that this automatic feedback is limited and other criteria such as the structure of your code, the memory management, ... are not taken into account by it. You are however **reminded** that the platform is a **submission** platform, not a test platform.

## 6 Submission

Projects must be submitted before the deadline. After this time, a penalty will be applied to late submissions. This penalty is calculated as a deduction of $2^N - 1$ marks (where $N$ is the number of started days after the deadline).

You must submit patches that contain diffs from linux-4.15. The patch file(s) has/have to be generated by the `git format-patch` command. Your submission will include your **clean** patch files in a directory named `patch`. In addition, add also your report in `.pdf` or `.txt` within the `patch` directory.

Submissions will be made as a `patch.tar.gz` archive, on the submission system. Failure to compile or to apply patches will result in an awarded mark of 0.

**Bon travail...**

## A Example

```
> ./test -o set_param -p "fork"
pf_set_param performed successfully.
> ./fork
...
> ./test -o get_info -p "fork"
Number of VMA       : 17
Number of COW faults: 32
All VMA:
 [ 0]: 0x8048000 - 0x8049000 (size: 0x1000): 0 cow faults
 [ 1]: 0x8049000 - 0x804a000 (size: 0x1000): 2 cow faults
 [ 2]: 0x804a000 - 0x804b000 (size: 0x1000): 4 cow faults
 [ 3]: 0x903e000 - 0x905f000 (size: 0x21000): 5 cow faults
 [ 4]: 0xb7d20000 - 0xb7d21000 (size: 0x1000): 2 cow faults
 [ 5]: 0xb7d21000 - 0xb7ed1000 (size: 0x1b0000): 2 cow faults
 [ 6]: 0xb7ed1000 - 0xb7ed2000 (size: 0x1000): 2 cow faults
 [ 7]: 0xb7ed2000 - 0xb7ed4000 (size: 0x2000): 2 cow faults
 [ 8]: 0xb7ed4000 - 0xb7ed5000 (size: 0x1000): 5 cow faults
 [ 9]: 0xb7ed5000 - 0xb7ed8000 (size: 0x3000): 0 cow faults
 [10]: 0xb7ede000 - 0xb7edf000 (size: 0x1000): 2 cow faults
 [11]: 0xb7edf000 - 0xb7ee2000 (size: 0x3000): 0 cow faults
 [12]: 0xb7ee2000 - 0xb7ee4000 (size: 0x2000): 0 cow faults
 [13]: 0xb7ee4000 - 0xb7f07000 (size: 0x23000): 0 cow faults
 [14]: 0xb7f07000 - 0xb7f08000 (size: 0x1000): 2 cow faults
 [15]: 0xb7f08000 - 0xb7f09000 (size: 0x1000): 4 cow faults
 [16]: 0xbfc2e000 - 0xbfc4f000 (size: 0x21000): 0 cow faults
> ./test -o cleanup
pf_cleanup performed successfully.
```

Concerning the `pf_get_info` system call, you might have slightly different results at each run. This is due to the volatility of the main memory.