Gaulthier Gain & Vincent Rossetto

INFO0940
Operating Systems

Project #1

Academic year 2022-2023

Your first project

You will hack the kernel

1. You will implement 3 syscalls in order to retrieve specific information about processes in the Linux kernel.
2. You will mainly deal with some stats about virtual memory of particular processes(es).
3. All will be done within the kernel space. We provide a user-space program[1] in order to test your syscalls.

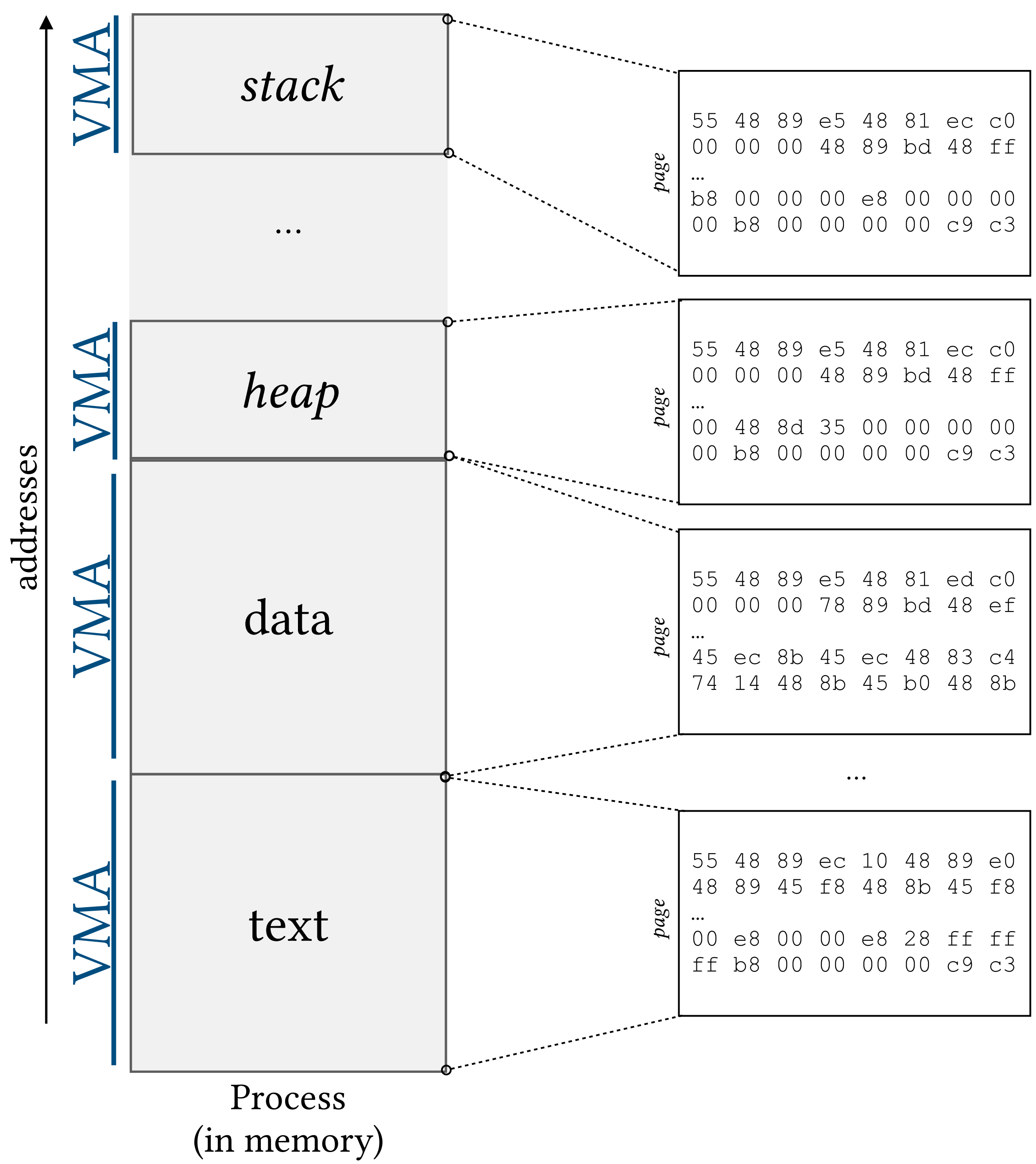
[1] see the tester program is available on eCampus

YOUR FIRST PROJECT

It is a **research** project…

1. This means you have to do some research yourself to understand how things work.
2. It is not a waste of time. If you understand how memory works (page fault, pages, …), you might pass the oral exam if you have a similar question.
3. Basically, it seems complicated but you will see that after some investigation, all will be clear.

DEALING WITH VIRTUAL MEMORY



- Virtual memory[1] 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**.
- When manipulating memory, the kernel first needs to consult the **page table**[1] which is used by virtual memory to store the mapping between virtual addresses and physical addresses.
- The page table (managed by the kernel) also contains the permissions associated to pages.
- A Virtual Memory Area (VMA) is a kernel data structure that allows the kernel to keep track of the process's memory mappings[2].

[1] see theoretical courses
[2] see practical session 5

Copy-on-Write (CoW) and Page Fault

In Linux:

- A new process is created via the `fork()` system call[1];
  - calling `fork()` allows both parent and child processes to **initially** share the same pages in memory.
  - Both process have the same page-table and every page is marked as read-only.
- When a process tries to write to a shared page:
  - There is a page-fault (CoW page-fault).
  - The OS make a copy of the page and update the page table.

CoW allows more efficient process creation as only modified pages are copied.

[1] see theoretical courses

High overview (1)

You need to implement the 3 following syscalls:

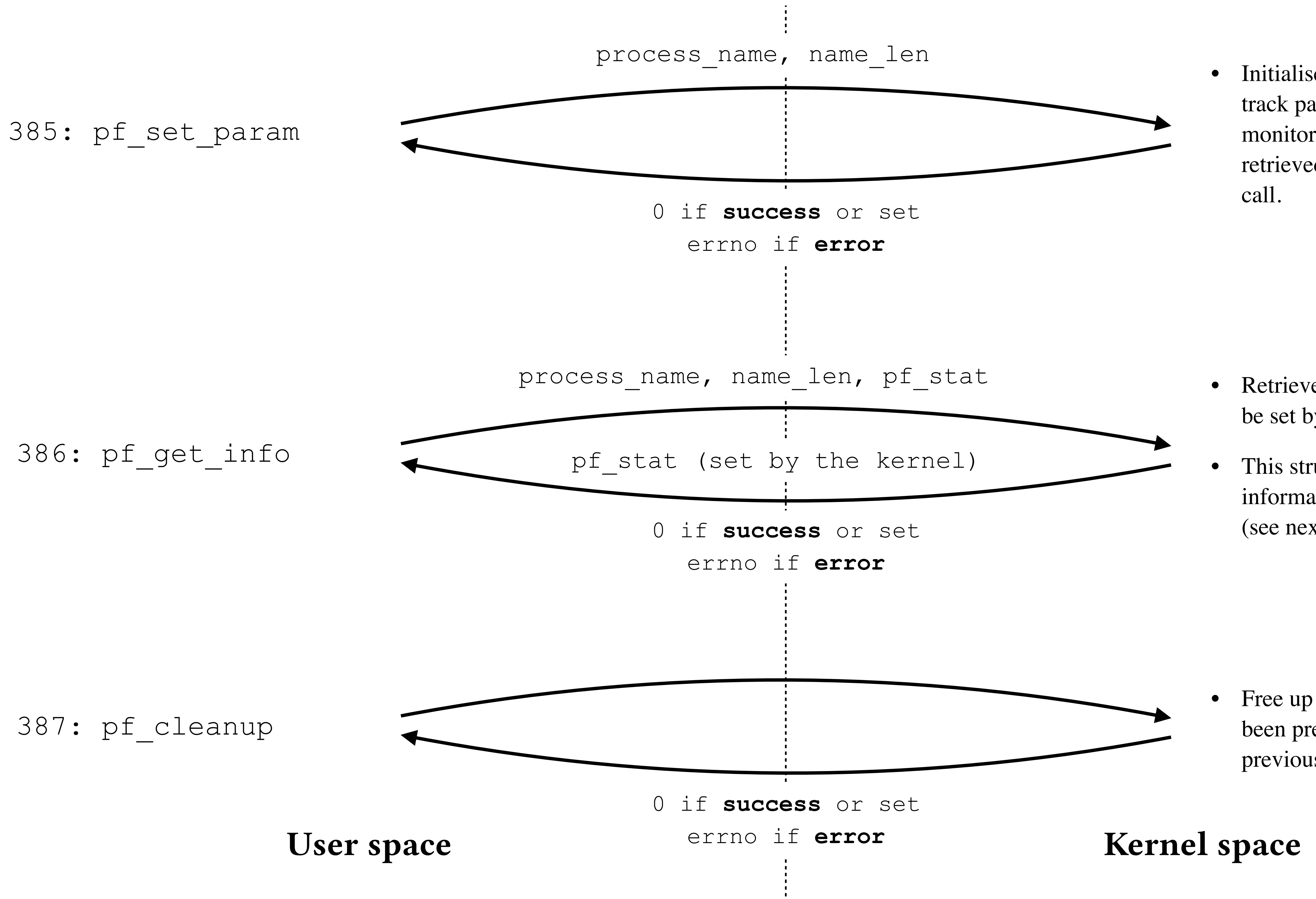
```
long pf_set_param(const char *process_name, size_t name_len);
long pf_get_info(const char *process_name, size_t name_len, struct pf_stat *pf);
long pf_cleanup(void);
```

You will test your system calls with a test program available on eCampus.

The system call number assignment is as follows (**do not change it**).

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

HIGH OVERVIEW (2)



- Initialise a kernel data structure and track particular process(es) in order to monitor specific information that will be retrieved by the `pf_get_info` system call.

- Retrieve a `pf_stat` structure that will be set by the kernel.
- This structure contains memory-specific information about a particular process (see next slide).

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

[1] see statements on eCampus for further information

The pf_stat structure

```c
#define MAX_SIZE 128

/* Do not modify this structure */
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];
  // Array that contains the start address of each VMA
  unsigned long vma_list_start[MAX_SIZE];
  // Array that contains the end address of each VMA
  unsigned long vma_list_end[MAX_SIZE];
};
```

- This structure contains memory-specific information about a particular process:
  - the number of Virtual Memory Areas (VMAs),
  - the total number of CoW page faults,
  - the number of CoW page faults per VMA,
  - ...
- Some of this information will be populated when the page fault is handled by the page fault handler (see next slide).

The 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/retrieved within the system call `pf_get_info`.

ERRORS HANDLING

- In the kernel, it is necessary to use specific errors[1][2] 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
- Furthermore, your implementation must display error messages with `KERN_ERR` and by printing the following prefix "`[INFO940][ERROR]`" on the kernel log.

[1] https://elixir.bootlin.com/linux/v4.15/source/include/uapi/asm-generic/errno-base.h
[2] the given helper program can help a lot.

Demonstration

General hints

For this project, you need to do **some research** but here are some hints to help you:

- You need to add 3 syscalls… think that we **only** consider **x86 32bits** architecture.
- You can investigate within the Linux source code and analyse existing system calls such as `read`, `write`, `fork`, … (+ official documentation).
- Do not forget to check user inputs (see session 4).
- Use `printk` to debug and `goto` for errors handling (see session 4).
- Check if your kernel is upgraded (`uname -a`).
- You can use `make` twice to be sure that you see all the errors (if any).

HOW TO START?

- Add 3 system calls respecting the imposed prototypes. To define the `pf_stat` structure in the kernel, you can refer to the system call `getrusage` to see where it defines its associated structure (`struct rusage *usage`).
- The next step is to find the page fault handler for the 32-bit architecture. Once found, it will be necessary to "filter the page faults" by tracked process and detect CoW page faults.
- Each VMA has a start and an end address, and their sizes are always a multiple of the page size (4KB).
- As an entry point, it will be interesting to have a look at these files: include/linux/sched.h and include/linux/mm_types.h then do some googling.
- It is related to memory management so have also a look within the `mm/` subfolder.
- You can check all VMAs associated with a process through the `/proc/<pid>/maps` file (see tutorial 3).

MAIN DIFFICULTIES

- (1) Know where to declare/define the `pf_stat` structure in the kernel.
  - Have a look at the `getrusage` system call.
- (2) Define a data structure that is manipulated by your system calls and the page fault handler.
  - The session 4 may help a lot.
- (3) Determine the page faults that are CoW page faults.
  - Need a bit of research but you will find how to do by looking at the kernel sources.

SUBMISSION

- Submit **all patches** that have diffs from Linux-4.15 (the first commit).
- Patches should be made by `git format-patch` (refer to the slides/tutorial page to know how to use it). Note that we setup a test environment on the submit platform (see the last tutorial).
- The patch file(s) should be saved in a directory named `patch`, and it should be compressed into an archive called `patch.tar.gz`
- As you have seen the submission platform might crash sometimes. Try to submit as soon as possible (submissions take time!).

The 32bits reference machine

Use the provided reference machine (32bits) to test your project.

- Do not implement your syscalls on another version of Linux than the one provided with the reference machine.
- Otherwise your program may fail some tests on the submit platform…
- Please refer to the online tutorial to setup the required environment.

REQUIREMENTS

Additional Information:

- Group of **two** that you will **keep** the whole semester.
- Submit a *tar.gz* archive on the submission platform (**clean** patch(es) & report).
- You are asked to write a **very short** report (max **2** pages) in which you briefly explain your implementation.
- Further information in the statements available on eCampus.

Do not forget: We want clean code and patches, without error.

Do not forget too: We detect **plagiarism** so don't try…

Plagiarism = **0 for the course!**

<u>Deadline: 19th April 2023</u>

Happy Coding!