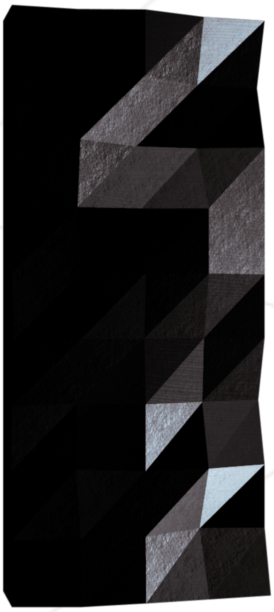


Linux Kernel: Process and Memory

LK1

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

Summary: Syscall ? Syscall.

Contents

Chapter I

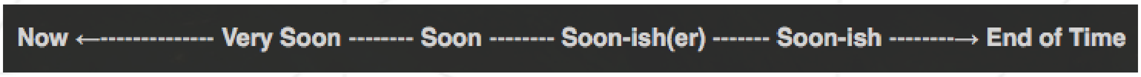
Foreword

I.1 Soon

Soon: Copyright 2004-2016 Blizzard Entertainment, Inc. All rights reserved. "Soon" does not imply any particular date, time, decade, century, or millennia in the past, present, and certainly not the future. "Soon" shall make no contract or warranty between Blizzard Entertainment and the end user. "Soon" will arrive some day, Blizzard does guarantee that "soon" will be here before the end of time. Maybe. Do not make plans based on "soon" as Blizzard will not be liable for any misuse, use, or even casual glancing at "soon."

I.2 Very Soon

Another common term implemented by Blizzard often misleading players into excitement for future content. "Very Soon" is guaranteed to arrive between now and the end of time with a higher chance of arriving on the "now" half of the time table. Although this means closer to now than "soon" there is no guarantee that you will live long enough to see the content finally release.



Now ←-------------- Very Soon -------- Soon -------- Soon-ish(er) ------- Soon-ish --------→ End of Time

I.3 Soon-ish

On Febuary 19, 2010, Nethaera used the term "Soon-ish" with regards to the next release of information pertaining to Cataclysm. Current speculation believes "Soon-ish" to exist between "Soon" and "End of Time."

Chapter II

Introduction

Welcome to LK: Process & Memory. In this subject, you will write a working syscall, and play with user and kernel memory. Exciting, eh ?! Wait until you break everything...

Syscalls are important. Like really important. They are the only interface existing between Kernel and Userspace. Each call you make to `malloc(3)` or `printf(3)` for instance hides a syscall. Get it ? Writing a kernel is only a part of writing an operating system, and to do so, one needs a robust and reliable interface between kernelspace and userspace. This interface is the syscalls.

For instance, one can write text to a tty using the syscall `write`, request a free memory page using `mmap`, interacting with a file on the disk using `open`, `read`, and `write`, etc.

These functions are part of the `libc` but are, in fact, wrappers for the syscall. This is mostly for readability purposes. Syscalls are nothing more than integers in kernel memory. Calling the `libc` `write` function will actually call something like `syscall(4, arg1, arg2, arg3)`. So yeah, calling `write` is waaaay more readable.

The goal of this project is to write the following not-yet-existing syscall `sys_get_pid_info` (Note the `sys_` prefix here, all syscalls in the kernel code will be prefixed the same way). The purpose of this syscall is to gather information about a process, store it, and return it to user space.

Chapter III

Goals

Adding a new syscall to the Linux kernel requires to understand what a syscall is, and how it is implemented kernel side. If you play along with this project, you will learn to :

- Find and understand the syscalls table
- Find and understand the syscalls header file.
- Add a custom syscall in the Linux Kernel. In our case, a syscall that takes a PID as its argument, fill a structure with informations regarding that PID and send it back to the caller.
- Integrate your code in the Linux code base.
- Handle correctly the differences between the kernel and userland.

Chapter IV

General instructions

- You must use your custom linux distribution, made in the `ft_linux` project. If it doesn't work like it should, too bad. Go back to `gcc` compilation pass 3, or go home.
- You must use a kernel version 4.x. Stable or not, as long as it's a 4.x version.
- All memory allocated in kernel space must be properly passed to user space. NO SEGV !
- A Makefile must be turned in.
- That Makefile must copy the right files from your git repository to `/usr/src/linux-$(VERSION)`, compile the kernel, move the kernel to the right place, and reboot the computer.
- You must also provide a working example of your syscall in a C file to prove your work during defense.

Chapter V

Mandatory part

V.1 Structure

You must write and add a new syscall to your Linux distribution. This syscall must be named `get_pid_info` user side, and `sys_get_pid_info` kernel side. The function must be prototyped as follows:

```
long    sys_get_pid_info(struct pid_info *ret, int pid);
```

The parameters are:

struct pid_info *ret : Output parameter. The structure containing the requested informations for a given PID.

int pid : Input parameter. The PID to get infos from.

The `struct pid_info` must contain the following informations:

- PID
- State of the process (Between 3 values)
- Pointer to process' stack. (RO in user space)
- Age of the process (Since execution, not boot-time)
- An array of child processes (PIDS).
- Parent PID
- Root path of the process
- Current PWD of the process (Full path)

V.2 Proof

For your peer-evaluation, you must be able to demonstrate that your code is working. Write a C example file that handle the following:

- Print info about a PID (Relative information to the structure returned by the syscall).
- Iterate over Parents / Children while printing information about them.
- That C example file must use your syscall, not read on /proc/pid.

Chapter VI

Bonus part

Some ideas for bonuses:

- Recode the `kill` and `wait` syscalls.
- Recode `mmap` (relative to the pages allocated by the process)
- Recode `fork`. If you think this one is difficult, wait till you discover that it's going to be part of a full scale project later...

Honestly, have a look in the Linux scheduler, and take some ideas from there.

Chapter VII

Turn-in and peer-evaluation

Turn your work in using your GiT repository, as usual. Only work present on your repository will be graded during defense.

Your code will be running on your custom linux distribution. Keep in mind the necessary setup to migrate your code from your Mac to your VM.