Operating Systems

Laboratory Exercises

Semester 2, 2024-25

Course Organizer
Dr. Antonio Barbalace

Teaching Assistants
Amir Noohi
Karim Manaouil
Alan Nair

Contents

1 Know your process

1.1 Experience with procfs

procfs is a special filesystem in Unix-like operating systems. It presents information about processes and other system information in a hierarchical file organization. For each process there is a directory /proc/<PID>/ where <PID> is its process id. A shortcut for the current executing process exists, and is /proc/self/.

The information available for each process is vast, and includes the command line, the executable name, its open files, etc. The information can be accessed using

```
ls --color /proc/self
```

Note that while in this example we use /proc/self, assuming the user has enough privileges procfs enables such user to access the information of each process in the system knowing its pid.

Basics

To know more about a process, we could first print the name of its executable as follow:

```
cat /proc/self/comm
```

Question: Before actually invoking the command, try to think whom is self – you have all the information to answer to this question already.

Secondly, the command line, which includes the arguments passed to the executable:

```
cat /proc/self/cmdline
```

Question: Before invoking the command, try to think what the output could be.

Lastly, let's have a look at the pid of self, and figure out whom is its parent process and some more information. Use the following command:

```
cat /proc/self/status
```

Note that the parent process is PPid. We can use that value to discover whom is the parent process in this way:

```
cat /proc/<PPid>/comm
```

It may also be interesting to discover what are the open files of an application from here. That can be done with:

```
ls -l /proc/self/fd
```

The kernel bit

procfs also provides information regarding the machine and the Linux kernel in use. For example, you can access the Linux kernel version simply with the following command:

```
cat /proc/version
```

Similarly to processes, the command line arguments used to launch the OS kernel can be visualized by using this command:

```
cat /proc/cmdline
```

Information regarding the CPUs on the system can instead be accessed via

```
cat /proc/cpuinfo
```

1.2 Creating Threads and Processes

Linux support kernel-level threading, i.e., the kernel is aware of all threads. Indeed, the Linux kernel supports the `fork` system call discussed in class. To create a thread Linux implements the `clone` system call. To avoid repetitive code, because these system call do a lot of similar work, `fork` (and even `vfork`), and `clone` are implemented by a single kernel level function called `kernel_clone`, which can be found here: https://elixir.bootlin.com/linux/v6.1.74/source/kernel/fork.c#L2641.
Question: Can the student identify the difference between the different syscalls in the source file?

Now that the student is familiar with the Linux source code to start a process or a thread, it is important to understand what are the actual arguments passed to the `clone` syscall to start a thread. This can be found in the source code that implements the `pthread_create` function. Specifically, here: https://elixir.bootlin.com/glibc/glibc-2.17/source/nptl/sysdeps/pthread/createthread.c#L146.
Question: What part of the process resources are copied to create a new thread for the same process?

2 Know your thread

In this section we will use GDB together with QEMU to identify what is the current task (thread) running. We will use the information available in the Linux kernel – this is what the student can exploit when debugging the kernel code.

Please refer to the related tutorial slides or to the previous tutorial and lab for the process involved to launch the Linux kernel provided in QEMU, and attach to it the GDB debugger. When starting GDB make sure to start it passing the Linux kernel image (`vmlinux`) as an argument. Then type the following command:

```
(gdb) target remote localhost:YYYYY
```

Where YYYYY is the same number specified during the invocation of QEMU – e.g., `-gdb tcp::YYYYY`. Also QEMU should be started with `-S`.

2.1 First Attempt

Before starting to trying to understand how we can devise information on the current task executing at anytime in the Linux kernel, let's make GDB a little bit more user friendly with the following commands:

```
(gdb) layout next
(gdb) layout next
(gdb) layout next
```

Indeed, the same command need to be inputted three times.

Successively, let's put a breakpoint at the function `get_current()`, and let the kernel continue executing:

```
(gdb) break get_current
(gdb) c
```

Soon QEMU will block and give the control back to GDB because the any of the QEMU CPUs reached the `get_current()` function.

The definition of the `get_current()` can be found here for Intel/AMD CPUs: https://elixir.bootlin.com/linux/v6.1.74/source/arch/x86/include/asm/current.h#L13, but basically it is a `mov` instruction with an offset to the `%gs` CPU register. Specifically, an assembly instruction like the following can be found:

```
mov %gs:0x1ad00,%rdx
```

Note that at the end of this instruction a poiter to `struct task_struct` is saved in the `rdx` CPU register.

Hence, we could use GDB to access the `tgid`, `pid`, and `comm` fields of such structure:

```
(gdb) print ((struct task_struct*)($rdx))->tgid
(gdb) print ((struct task_struct*)($rdx))->pid
(gdb) print ((struct task_struct*)($rdx))->comm
```

2.2 Second Attempt

In the same run, let's delete any existent breakpoint and add a new breakpoint to the function `try_to_wake_up`. This can be done with the following commands, the last continue execution:

```
(gdb) delete
(gdb) break try_to_wake_up
(gdb) c
```

Differently from the first attempt, here we show that the student doesn't need to master assembly language to debug the Linux kernel.

Once the execution stops at `try_to_wake_up`, the GDB is back in charge and the variable `p` is a pointer to a `struct task_struct`. Therefore, we can use gdb to print the same information (`tgid`, `pid`, and `comm` fields) as before:

```
(gdb) print p->tgid
(gdb) print p->pid
(gdb) print p->comm
```

Note that this attempt is more familiar than the previous one.

2.3 More

GDB can be scripted to automate operations. This is very much necessary when debugging complex pieces of code. We will show a simple example here.

Imagine we would like to see all the processes (implicitly) calling the function `try_to_wake_up`. We could do that manually, but we can also automate that with the following commands:

```
(gdb) delete
(gdb) break try_to_wake_up
(gdb) command
> print p->comm
> cont
> end
(gdb) c
```

Note that the `delete` command here helps to make sure there are no other breakpoints. The last command, `c`, starts the execution. The rest of the code tells to GDB that anytime it stops it should first print `p->comm` and then continue execution.

2.4 Python in GDB

The GDB interpreter/shell has been considered not very efficient, so GDB can be scripted in Python.

The Linux kernel comes with a series of GDB scripts written in Python that enable an easier debugging of the Linux kernel. They can be found in `scripts/gdb/linux/constants.py`. Potentially, you may need to patch this file. A simple patch that worked before is indicated in the companion slides (the slides of the associated tutorial). To load the Python scripts, when in running GDB in the root directory of the Linux kernel, the following GDB command should be inputted:

```
(gdb) source vmlinux-gdb.py
```

Commands and functions can be listed with:

```
(gdb) apropos lx
```

The slides of the associated tutorial guide the student on using such commands. It is suggested to explore how to use `lx_current` to obtain the current task.

3 Tracing

Linux enables function tracing with `ftrace`, which is a component of the Linux kernel itself. There are several command line and graphical tools to be used together with `ftrace`, including:

- trace-cmd (command line)
- kernel-shark (GUI)
- Lttng (GUI)

While `ftrace` can be used also without the help of any of such tools, the use of any of such tools simplify the usage. Herein we will focus on `trace-cmd`.

`trace-cmd` may not be installed in the guest VM with your prototype Linux kernel. To install it is necessary to use the following command:

```
# apt-get install trace-cmd
```

3.1 First Attempt

A trace should be first recorded, and then it can be visualized. A trace can be related to a specific process, and this is what we are going to show here. To invoke the recording for a specific process (`ls`), the following command can be used:

```
# trace-cmd record -e ext4 ls
```

This example records all events that matches "ext4". A set of traces files are saved as a result of the trace recording.

To read the trace the following command can be used in the same directory:

```
# trace-cmd report
```

3.2 Second Attempt

Events related to scheduling can be recorded with the following command:

```
# trace-cmd record -e sched_switch -e sched_wakeup -e irq sleep 10
```

In this case we are tracing the command `sleep`. Moreover, this trace highlights all context switches.