PRACTICE 2: MODIFICATIONS TO THE KERNEL

Presentation

The practices of the subject Operating Systems Design aim for the student to be able to understand portions of source code of a current version of the *kernel* of Linux and to introduce localized modifications to the functionalities of the *kernel* .

The practices of the subject are two:

- The first practice describes how to prepare the development environment used in the practices (compilation of the *kernel* and booting a machine using the *kernel* generated) and how to navigate within the directory structure that make up the source code of the *kernel* of Linux. It also presents the mechanism of the modules because it will be used in the second practice to introduce new functionalities in the code of the*kernel* .
- The second practice proposes a series of quite localized modifications to the code of the *kernel* of Linux. For example, go through the page table structure of a process, create a driver for a new device, or get information about the file system.

The previous knowledge necessary for the development of these practices is:

- Algorithm.
- Programming in a high level language (preferably in C language). Use of pointers.
- Data structures (double-stranded lists, hash tables).
- Theoretical concepts of the subject Operating Systems.
- Using Linux from the shell.
- Understand small snippets of code written in i386 assembler.
- Those presented in practice 1.

This document contains the statement of the second practice.

The second practice of the subject proposes that the student write code that becomes part of the *kernel* of Linux. To perform this task, the development environment presented in the previous practice (machine*host* and machine *guest*) and the mechanism of the modules. To do this practice you need to download the filebase2.zip of [1] and unpack it with the command unzip.

Skills

Transversal:

- Ability to adapt to technologies and future environments by updating professional skills
- Ability for written communication in the academic and professional field

Specific:

- Ability to analyze a problem at the level of abstraction appropriate to each situation and apply the skills and knowledge acquired to address and solve it
- Ability to design and build computer applications using development, integration and reuse techniques

Targets

The objectives of this practice are that the student ...

- know the steps to follow so that the *kernel* may offer a new call to the system.
- implement the *driver* for a new device.
- explore the source code of *kernel* to study code snippets similar to the ones you will need to implement.

In order for the teacher to assess the degree to which these goals have been achieved, students will need to resolve *individually* of the exercises.

Description of the Practice

1 Exercise 1: Add a new call to the system

Linux 4.19.206 consists of approximately 380 system calls. All calls are associated with a numeric identifier (can be found in the filearch / x86 / include / generated / asm / syscalls 32.h).

To add a new system call to *kernel* it can be done in two ways: by compiling the *kernel* or by creating a module. Although the first form would be the most generic solution, in this practice it is proposed to use the second form. Take advantage of the fact that some of the call identifiers in the system are not actually used.

Before starting the exercise you need to compile the *kernel* to allow modules to reference the symbols of the *kernel* sys call table i sys or syscall. To export sys or syscall it is necessary to proceed similarly to what you had to do in example 3 of practice 1 to export *kernel* ia32 sys call table.

1.1 Example 5: Empty system call

We provide you with the source code for a module (modules / example5 / newsyscall.c) which adds a new call to the system. The module, when installed·perform the following tasks:

1. Look for an unused entry in the table sys call table. To find the ticket you have to go through the tablesys call table and look for any entry that is initialized with the value sys or syscall. The entries in this table are of type unsigned and the table has NR syscalls tickets.

2. Show (using the routine printk) the index (position number) corresponding to the found entry. This index will be the identifier that will be associated with the new system call. Under normal circumstances, this index should be 17.

3. Replace the contents of the unused entry of sys call table by the memory address of a routine (sys newsyscall, implemented in the module) which will be the one to serve the new call to the system. In this example, the routine printsHello world followed by a parameter.

Uninstalling the module will restore the contents of the entry used in the table sys call table. _

To get a user program to invoke the new system call, the indirect system call must be used syscall (). It is parameterized with the numeric identifier of the call to the system that we want to invoke and the list of parameters of this call to the system. For example, we could invoke the callread (fd, & c, 1) through syscall (NR read, fd, & c, 1). The user programmodules / example5 / test5.c invokes the new system call (assuming it has been installed·lat in position 17).

The order make to the machine *host* compile the test module and program. The filesnewsyscall.koi test5 they must be copied to the machine *guest* . Once the module is installed, run test5 on the machine*guest* should cause the message to appear Hello world shown by the routine that serves the new call to the system. In addition, it will also display the parameter specified on the command line.

An example of the installation is shown·operation of the example module.

```
root@virtual:~/modules/example5#
root@virtual:~/modules/example5# insmod newsyscall.ko
New syscall installed. Identifier = 17
root@virtual:~/modules/example5# ./test5
Hello world, parameter 0
Return 29
root@virtual:~/modules/example5# ./test5 45
Hello world, parameter 45
Return 29
root@virtual:~/modules/example5# rmmod newsyscall
New syscall removed
root@virtual:~/modules/example5#
root@virtual:~/modules/example5# _
```

1.2 Call the system to implement

Once you understand the example above, you are asked to modify it so that the new call to the system implements a more useful service.

The service to be implemented will be to calculate the number of sister or child processes of a specified process. The system call will have two inbound input parameters: the first parameter corresponds to a process identifier and the second to the operation type. If the value of the second parameter isNUM CHILDREN(0), will return the number of child processes of the process; if it isNUM SIBLINGS (1) will return the number of sister processes in the process. If the specified process does not exist, it will return the error code -ESRCH; if the type of operation is incorrect it will return the error code -EINVAL.

To calculate the number of child processes and sister processes of a process it is necessary to access its data structure of type task struct which contains the process PCB. This structure has a lot of fields, but the relevant ones in this exercise arechildren i sibling, both types list head.Each of these fields is just a double-stranded list that allows you to get the list of child processes and sister processes respectively.

In Example 2 of Practice 1 you can see how to get the task struct of a process. With the toolcscopeyou can find out how to go through the lists children i sibling.

To be able to test the module, the user program modules / newsyscall / testnew.c invokes the new call to the system several times and performs some tests to determine its proper operation. If the program execution does not reach its final statement, it indicates that your module has an error.

1.3 Deliveries corresponding to this exercise

The deliveries corresponding to this exercise are:

- The source code of the module that implements the new call to the system.
- Brief explanation of the work done (format pdf, including screenshot of the test program result).

2 Exercise 2: Implementing a *driver*

The didactic module *Input / output management* of the documentation of the subject [2] describes the concept of *driver* (Chapter 1) and Device Descriptor (Chapter 5.3.2). The*drivers* in Linux they use a data structure similar to a device descriptor. Therefore, the implementation of calls to the incoming / outgoing system is structured in two levels: one device-independent and one device-dependent.

The section 2.1 presents the source code of a module that registers a new one *driver* . After studying this source code, the section 2.2 raises the exercise to be solved in this practice.

2.1 Example 6: *driver*

To the directors modules / example6 the source code of a module is presented (abc.c) which dynamically discharges a new one *driver* and Linux. This*driver* returns the letters of the alphabet in an orderly fashion. A program is available to test the module (test6.c) which tests the system calls implemented by *driver* . The order make compiles both the module and the user program. Once compiled, they must be transferred to the machine *guest* to be tested.

Below we will describe the source code of the module with the *driver* . The module installation routine tells the operating system that you want to register a new one *driver* . This is done with routine register chrdev ().This routine is parameterized with the *major* of the device, the name of the device and with a pointer to a type structure struct file operations.

- He *major* isis an integer that identifies the device. The*majors* used are defined in the fileinclude / uapi / linux / major.h. You need to choose a free identifier and assign it to the macro MAJOR DRIVER, file abc.c, this value. We choose the 231.
- Device name is only one *string* to be used when listing active devices (file /proc / devices).
- The pointer to a type structure struct file operations will allow the *kernel* find out the specific routines for accessing your device. The structure has pointers to operationsopen (), read (), write (), llseek (), ioctl (), ... device specifications; all these routines have a defined interface. Before invoking the routineregister chrdev (), éIt is necessary to initialize the structure.

Once the module is installed (and the *driver*) on the machine *guest*, you need to create a device type file that is associated with *major* which you specified as the first parameter in the routine register chrdev ().One possible way to do this is to use the command mknod / dev / ex6 c 231 0 on the machine *guest*.From now on, all incoming / outgoing operations (system calls open (), read (), write (), ...) made on this file will be served by the routines specified in the third parameter of the routine register chrdev ().

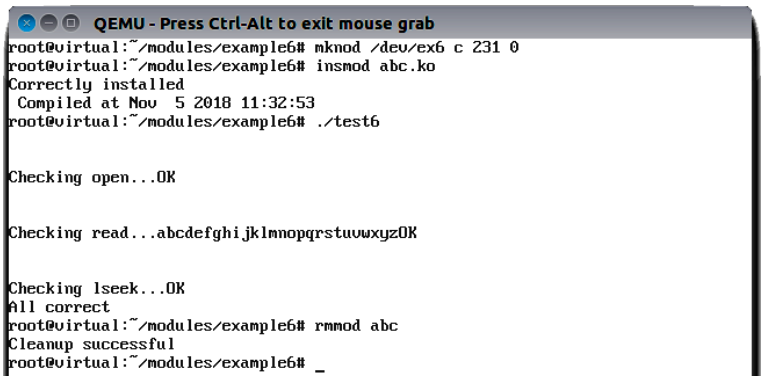
The routine open () check that the file has been opened in mode O RDONLY (rural area f flagsof the second parameter of the routine) and returns 0, otherwise returns the result -EACCES (negative results indicate that an error has occurred and, in absolute terms, what the error code is; results greater than or equal to zero indicate that the call was successful).

The routine read () of the device receives four parameters: a pointer to a structure struct file, the memory address of the user buffer where the read characters are to be written, the maximum number of characters to read, and the memory address where the read / write pointer is stored on the file. The second and third parameter of the callread () of the device match the second and third parameters of the system call read () invoked by the user program. Depending on the value of the read / write pointer on the file (parameterf pos), writes the corresponding characters on the user buffer using the routine copy to user. Finally, the routine returns the number of characters that have been read. Plus, this one*driver* example limits the maximum number of characters that can be read in each reading to three.

The routine llseek () of the device updates the read / write pointer according to the settingsoffset i orig. Routine returns the new value of the read / write pointer (or -EINVAL in case of error).

When uninstalling the module you need to unregister the *driver* using the routine unregister chrdev ().

An example of the installation is shown·operation of the example module.



2.2 *driver* to implement

Once you understand the sample code of *driver* presented in the previous section, you can solve the following exercise.

You are asked to transform the module from the previous exercise (which added a new call to the system that returned the number of children / siblings in a process) so that no you will interact with the module using the new system call but you will do so using the device file /dev / relatives. To do this, you can take advantage of the code *driver* for example abc.c.

He *driver* will need to respond to the following system calls:

- open (): In order to access the data, you will need to open the device using the system callopen () in mode O RDONLY. If you try to open the device in any mode other thanO RDONLY, the call must return the error code EACCES. If the device is already open, the call will return the error code EBUSY. To ensure that the implementation of this call is successful, it is necessary to mutually access a variable using, for example, kernel routines. DEFINE SEMAPHORE, up, down; research how to use them.
- read (): The system call must be used to read the data read (int fd, char * buf, int num).

  We will interpret the read / write pointer associated with the device file as *pid* of the process from which we want to obtain the number of children / siblings.

  The third parameter of the call read () will be interpreted as the maximum number of processes from which we want to obtain the information (not bytes because we are in an architecture where integers occupy 32 bits). For example, if the read / write pointer points to process 54 and the number of counters to read is 3, we will try to read the information from processes 54, 55, and 56, that is, 12 bytes. Some of these processes may not exist.

  The second parameter of the call read () will be interpreted as the memory address of the user memory space from which the information obtained will be left (-1 in case the process does not exist).

  A side effect of this call will be to increase the read / write pointer on the table.
- lseek ():

  The call lseek will change the read pointer on the device in a similar way to an ordinary file. In this way, we will be able to position ourselves to read the data relating to a specific process. We will accept travelSEEK CUR i SEEK SET, but no SEEK END.
- ioctl (): The call ioctl will support two commands (defined in newsyscall2.h):
  - NUM CHILDREN: It will return the number of children (default value).
  - NUM SIBLING: The number of siblings will return.
- write (): No need to implement call write.
- close (): The call close will release the device so that it can be opened by other processes.

Observations:

- Like the code of the *driver* is part of the *kernel* , isIt is very important to perform a thorough error check and return the appropriate error code in each case.
- You can use the routine printk to print "xivatos" in the module code.
- A test program is available (modules / driver / testdriver.c) to check the correct operation of the *driver* . To generate the executable you need to run make. If the trial program aborts before the last sentence, it means yours *driver* is incorrect.
- If you have any questions about the behavior of calls to the system or the error codes that need to be returned, please refer to the test set code.

2.3 Deliveries corresponding to this exercise

The deliveries corresponding to this exercise are:

- The source code of the module that implements the *driver* .
- Brief explanation of the work done (format pdf, including screenshot of the test program result).

Resources

Basic Resources

[1] Sample files and shellscripts for the second practice.
URLhttp://einfmlinux1.uoc.edu/aso/base2.zip

Complementary Resources

[2] T. Jové, JL Marco, D. Royo, E. Morancho, Operating Systems Design.

Evaluation criteria

The first exercise weighs 40% and the second 60%.

Delivery format and date

Answers to this practice must be submitted in a single file zip o tar result of compressing two directories. Each directory will correspond to an exercise and will contain the files to be delivered in the exercise.

This file must be submitted to the continuous assessment register before the deadline established in the teaching plan (24:00 on December 17).