CS 251/EE 255: Real-Time Embedded Systems
University of California, Riverside

HW Project #1: Introduction to Linux Kernel Programming

Due: 2/4/2023, Saturday, 11:59 p.m. (Pacific time)

1. Introduction

You should learn the following from this project:

- Creating kernel modules for the Linux kernel
- Adding new system calls to the Linux kernel.

Before you start hacking:

- Follow all the steps in the **Development Environment Setup** document.
- **Read the entire handout** before starting work: writeup questions and the tips section aim to give you hints about design and implementation.
- Before proceeding, please complete all steps in Part 1 to setup your development environment.
- Whenever the handout asks you to do something without explaining how to accomplish it, please consult references on kernel development. That's by design.
- **To be awarded full credit, your kernel must *not* crash or freeze under any circumstances.**

2. Background Information

2.1. Loadable Kernel Modules (LKMs)

An LKM is simply a chunk of binary code that can be loaded and unloaded into the kernel on demand. Kernel modules are used to extend the functionality of the kernel without the need to reboot the system. The counterpart in user-space programs are shared libraries (aka. DLLs). For example, one type of module could be a character device driver which enables the kernel to interact with a particular hardware connected to the system. Loadable kernel modules keep kernel source modular and allow for third party components distributed in binary form (without the source code). Essential utilities for working with modules:

- `lsmod`: list currently loaded modules
- `insmod`: load a module by a path to a .ko file
- `rmmod`: unload a module by name
- `modprobe`: load a module by name together with its dependencies
- `modinfo`: display information about a module

For further information about loadable kernel modules, see Chapters 1 and 2 of [2] (online).

2.2. System calls

System calls provide an interface between userspace processes and the kernel. For example, the I/O interface is provided by the system calls `open`, `close`, `read`, and `write`. A userspace process cannot access kernel memory and it cannot (directly) call kernel functions. It is forced to use system calls.

To make a system call, the process loads the arguments into registers, loads the system call number into a register, and then executes a special instruction (called a *software interrupt* or *trap*), which jumps to a well-defined location in the kernel. The hardware automatically switches the processor from user-mode execution to restricted kernel mode. The trap handler checks the system call number, which in turn represents what kernel service the process requested. This service in turn will correspond to a particular OS function that needs to be called. The OS looks at the table of system calls to determine the address of the kernel function to call. The OS calls this function, and when that function returns, returns back to the process.

In case of 64-bit x86 architectures, the system call *table* is defined in `arch/x86/entry/syscall_64.c` and the system call *numbers* (and their associated function names) are declared in `arch/x86/entry/syscalls/syscall_64.tbl`. During the kernel compilation process, the tbl file is read to automatically generate actual header files needed for the kernel. For example, if you want to look at where the system call `sched_setparam`'s number is declared, type `'grep -rnI sched_setparam arch/x86/'`. This will display `syscall_64.tbl` and other auto-generated files (those in arch/arm/include/generated). The implementation of the system call function can be found in a different directory, e.g., `sched_setparam` is defined in `kernel/sched/core.c`)

IMPORTANT: When adding a new system call, do **NOT** modify those auto-generated files (i.e., anyfile in arch/x86/include/generated). Their contents may be silently modified or replaced by the kernel complication process. Also, their changes are not tracked by `git`.

2.3. Processes and Threads

In the Linux kernel, the kernel object corresponding to each process/thread is a *task*. Every process/thread running under Linux is dynamically allocated a `struct task_struct` structure, called Task Control Block (TCB). The TCB is declared in `<kernel_srcs>/include/linux/sched.h`.

The Linux kernel assigns a task ID to each task. Hence, both processes and threads have task IDs and each task's ID is stored in its TCB, in a member variable `pid_t pid` (not 'tid' due to historical reasons). In a single-threaded process, the task ID is equal to the process ID. In a multi-threaded process, all threads have the same process ID, but each one has a unique task ID. The process ID is also called a *thread group leader ID* (TGID) and it is stored in `pid_t tgid` of a TCB.

You can use `ps` and `top` commands in shell to view the list of running processes and (optionally) threads. You can also see the family relationship between the running processes in a Linux system using the `pstree` command.

For example, type "`ps -eLfc`" in the command line.

```
$ ps -eLfc
UID        PID  PPID   LWP NLWP CLS PRI STIME TTY          TIME CMD
root         1     0     1    1 TS   19 Nov29 ?        00:00:09 /sbin/init
root         2     0     2    1 TS   19 Nov29 ?        00:00:00 [kthreadd]
...
root        10     2    10    1 TS   19 Nov29 ?        00:00:00 [rcu_bh]
root        11     2    11    1 FF  139 Nov29 ?        00:00:00 [migration/0]
root        12     2    12    1 TS   19 Nov29 ?        00:00:00 [cpuhp/0]
...
pi         804   560   804    3 TS   19 Nov29 ?        00:00:00 /usr/lib/gvfs/gvfsd-trash...
pi         804   560   809    3 TS   19 Nov29 ?        00:00:00 /usr/lib/gvfs/gvfsd-trash...
pi         804   560   810    3 TS   19 Nov29 ?        00:00:00 /usr/lib/gvfs/gvfsd-trash...
...
```

This command prints all tasks including processes and threads running in the system. The second column, `PID`, displays the process ID of a task, and the fourth column, `LWP`, shows the actual task ID. See lines in cyan and yellow. `PID` and `LWP` fields are the same. That means it is either a single-threaded process or the leader thread (or also called *master* or *main* thread) of a multi-threaded process. In fact, PID 804 is a multi-threaded process. You can see two green lines, where PID is also 804 but LWP is different. This means these two tasks, 809 and 810, are the child threads of the process 804.

The TCBs of tasks are maintained in a doubly-linked list in the kernel. To find a task's TCB with a task ID (`pid`), you can use: `find_task_by_pid_ns(<pid>, &init_pid_ns);`

Each task is associated with a priority level. In Linux, the priority value range is divided into general-purpose priorities and real-time priorities (see `rt_priority` field in `struct task_struct`). All real-time priorities supersede all general-purpose priorities. A real-time task is a task whose scheduling class is a real-time class and `rt_priority` value is within the real-time priority value range (1 to 99). The `chrt` command can be used to give a process/thread real-time priority in a shell terminal. `sched_setscheduler()` is a system call to set real-time priority in C programs.

Let's take a look at the above "`ps -eLfc`" example again. The scheduling class and priority of tasks are shown in the `CLS` and `PRI` columns, respectively. For convenience, Linux displays the priorities of time-sharing scheduling class (e.g., SCHED_OTHER) in the range of 0 to 40 and those of real-time scheduling class (e.g., SCHED_FIFO, SCHED_RR) in the range of 41 to 139 (meaning real-time priority 1 to 99). See the line in cyan, PID 11. Its CLS is FF, meaning its scheduling class is SCHED_FIFO; its PRI is 139, indicating its real-time priority is 99 (the highest priority level). You can verify this observation by '`chrt -p <pid>`' in a shell.

2.4. Miscellaneous character drivers

A device driver is a part of the kernel that implements an interface to a hardware device. A characteristic feature of Linux and other Unix-like systems is that this interface is usually represented in a file abstraction. This is what you can see from the `/dev` directory – this is the virtual file system that the kernel maintains to present devices to the user. Each device is identified by two numbers (major and minor numbers; you can see them by running "`ls -l /dev/`").

Miscellaneous character drivers (aka. misc drivers) are simple driver interfaces provided by the Linux kernel. In most cases, they are owned by kernel modules and used to implement small driver code and interact with the user-space program. Misc drivers can be registered and deregistered by `misc_register()` and `misc_deregister()`.

3. Build Setup

You need to create a subfolder `proj1/` in the top-level directory of your repository (e.g., `~/rtes/proj1)`. Your source files to be implemented will fall into three categories:

Category	**Location in kernel repo**	**Build method**
built-in kernel code	`proj1/kernel`	built together with the kernel image
loadable kernel modules	`proj1/modules`	standalone using the kernel build system
user-space applications	`proj1/apps`	standalone using user space libraries

The following sections describe how each kind of source is to be built. All builds must be automated via makefiles as shown in the following sections. If needed, a reference to the tool `make` is [3]. The kernel should be buildable with its standard existing Makefile. For each module and user-space application (or app), **you must add a Makefile so that running ‘`make`’ in the respective subfolder builds the module or app.** This is necessary for graders to build your source.

3.1. Built-in kernel code

Source code files in this category need to be compiled together with the kernel image, when running `make` in the top-level directory. The following explains how to do so.

1. Create `proj1/kernel` directory in the repo (e.g., `~/rtes/proj1/kernel`).
2. Create `proj1/kernel/`**`Kbuild`** file where you will add each source file (not header file) that you'd like to be compiled as part of the kernel image. For example, to include `somefile.c`, add a line in `proj1/kernel/Kbuild`:

   ```
   obj-y += somefile.o
   ```

3. Edit **`Makefile`** in the top directory of the repository (e.g., `~/rtes/Makefile`) to add `proj1/kernel/` to the `core-y` list:

   ```
   ...
   core-y += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ proj1/kernel/
   ...
   ```

   Please do not modify the root kernel makefile beyond this.

3.2. Loadable kernel modules

To build with a simple `make` from the module's directory, create a **`Makefile`** in the module's directory (e.g., `proj1/modules/hello/Makefile`):

```
1  obj-m += hello.o
2  hello-y += another_source_file.o         # not needed if you have only one C file
3
4  all:
5          make -C ../../.. M=$(PWD)
```

By `Makefile` syntax, the commands (line 5) begin with a mandatory ***tab***, not spaces. Here, `hello` would be your module name. Line 2 is optional; this kind of line is needed only when the module (`hello.ko`) consists of other source files and can be repeated to add more files. In other words, if your module has only one C file, skip line 2

Note that each module must be in a separate directory (e.g., `proj1/modules/hello`) and needs to have at least one main source file.

3.2. Userspace applications

Userspace applications should be compilable with `make`. Create a `Makefile` in the application's directory (e.g., `proj1/apps/hello/Makefile`) in the following form:

```
CC=gcc

all:
        $(CC) -o someapp someapp.c

```

where `someapp.c` is your source file name.

4. Assignment

4.1. Hello World Kernel Module (10 points)

4.1.1. User-space C application (5 points)

Source code location: `<your_repo>/proj1/apps/hello/usrhello.c`

Write a simple application in C that prints to the console:

```
Hello world! team00 in user space
```

You must change `team00` to your team name (same as your repo name). Build it, run it, and check the output.

Be sure to add `Makefile` and source code to your team's git repo.

4.1.2. Loadable Kernel Module (5 points)

Source code location: `<your_repo>/proj1/modules/hello/hello.c`

Write a loadable kernel module (LKM) which upon loading prints the following to the kernel log (using `printk`):

```
Hello world! team00 in kernel space
```

You can check kernel messages by typing `dmesg`.

To load the kernel module, use `insmod` (e.g., `sudo insmod hello.ko`). The module should be removable by `rmmod.` Check if you can load the module again after `rmmod`.

4.2. Virtual Device (40 points)

4.3.1. Write a misc device driver (20 points)

Source code location: `<your_repo>/proj1/modules/rtesdev/rtesdev.c`

Create a module with a **misc device** that when read will output a list of real-time tasks (`rt_priority` > 0) with their task IDs (`pid`), process IDs (`tgid`), real-time priorities, and command names. Once this module is loaded, the misc device should appear as `/dev/rtesdev`. Implement the `read` function for this device -- reading from your device with e.g. `sudo cat /dev/rtesdev` should output kernel messages, for example:

```
$ sudo cat /dev/rtesdev
$ dmesg
...
tid pid pr name
102 102 90 adbd
103 103 80 pigz
104 103 80 pigz
```

The device must be deregistered when the module is unloaded.

4.3.2. Support IOCTL (20 points)

Source code location: `<your_repo>/proj1/modules/rtesdev/rtesdev.c` (add this feature to the same file)

`IOCTL` is an effective way for a user-level process to communicate with a device driver. In your misc device driver, implement the `unlocked_ioctl` function to handle IOCTL commands sent from a user-level process. The function should handle two different IOCTL commands defined below:

```
#define IOCTL_PRINT_HELLO _IO(0, 0)

#define IOCTL_GET_TIME_NS _IO(0, 1)
```

`IOCTL_PRINT_HELLO` simply prints the same string as in 4.1.2 (`Hello world! team00 in kernel space`) to the kernel log. The user program does not pass any argument other than this command.

`IOCTL_GET_TIME_NS` causes the device to return the current system time in nanoseconds to the user-space program. Use `ktime_t time_get(void)` to obtain the system time in the kernel (include `linux/time.h` in the module source file). `ktime_t` is just a signed 64-bit integer (`typedef s64 ktime_t`). The user program passes a pointer as an argument to store the current time. For example, the user program can do:

```
int fd;
long current_time;
fd = open("/dev/rtesdev", O_RDONLY);
ioctl(fd, IOCTL_GET_TIME_NS, &current_time);
printf("Current time in ns: %ld\n", current_time);
```

4.3. System Calls (40 points)

4.3.1. Write a process information system call (20 points)

Source code location: `<your_repo>/proj1/kernel/count_tasks.c`

Add a new system call `count_rt_tasks` that can be used to obtain an integer value that equals the total number of real-time processes or threads that currently exist in the system. The system call should return success (0) when the call is successful; fail (-1) when it fails (e.g., input parameter is not valid).

The `count_rt_tasks` system call must use **449** as the system call number and have the following format:

```
count_rt_tasks(int* result)
```

Keep the type of input parameter. The violation of this format will make the system call unusable. Of course, your kernel should never crash even if the user gives a wrong parameter (e.g., NULL pointer).

4.3.2. Write a count_tasks user application (5 points)

Source code location: `<your_repo>/proj1/apps/test_count_tasks/test_count_tasks.c`

Write a user-level application that makes the `count_tasks` system call, and outputs the result of the system call:

```
$ ./test_count_tasks
3
```

4.3.2. Override syscall at runtime (15 points)

Source code location: `<your_repo>/proj1/modules/mod_count_tasks/mod_count_tasks.c`

Write an LKM that upon loaded modifies the system call table and replaces the behavior of the `count_rt` system call such that it counts the number of real-time processes or threads whose **RT priorities are higher**

than 50. When this module is unloaded by `rmmod`, the `count_tasks` system call should behave normally. For example,

```
# before loading mod_count_tasks
$ ./test_count_tasks
3
# after loading mod_count_tasks; assume there is only one task w/ rtprio > 50
$ ./test_count_tasks
1
# after unloading mod_count_tasks
$ ./test_count_tasks
3
```

The system call table is defined as a read-only variable (`const`) in the Linux kernel. Change this so that your LKM can update it at runtime. (Note: the standard Liuux kernel also adds other features to protect the system call table from modifications for security purposes. But the kernel code used in this course has disabled such features).

4.4. Writeup (10 points)

Writeup location: `<your_repo>/proj1/proj1_team00.pdf` or `proj1_team00.txt`

Submit a **plain text** or **PDF** document with brief answers to the following questions:

1. How does a system call execute? Explain the steps from making the call in the userspace process to returning from the call with a result.
2. What does it mean for a kernel to be *preemptive*? Is the Linux kernel you are hacking on preemptive?
3. When does access to data structures in userspace applications need to be synchronized?
4. What synchronization mechanisms can be used to access shared kernel data structures safely? List them.
5. Take a look at the `container_of` macro defined in `include/linux/kernel.h`. In rough terms, what does it do and how is it implemented?
6. Give a brief summary of the contributions of each member.

5. How to submit?

To submit your work for (future) homework projects, **(1)** push your commits to the **`main`** branch of GitHub, and **(2)** submit the **SHA hash of the commit** that you want to be graded to eLearn (first 7 hexdigits are enough, e.g., `dee1237`). Only one member of the team needs to submit the SHA in plain text (.txt) to eLearn. The SHA hash of each commit can be checked by "`git log`" command or from your repo's GitHub page. For example,

```
$ git log
commit ff66847a00ac27d8d94b3664ec156a195dbf3676 (HEAD -> main, origin/main,
origin/HEAD)
Author: xxx
Date:   xxx

    Just testing

commit dee123781efab5effe4f9a7583c0d71608b6977f
Author: xxx
Date:   xxx
```

```
    Final version
...
```

If your git log is like above and you want the highlighted one to be graded, submit `dee1237` to eLearn.

Add write-ups to your repo in either **plain text (.txt)** or in **PDF**. If you use MS Word or other tools, convert it to PDF and then submit. **We will not accept any other format than txt of pdf.**

Add only the source files, headers, and other relevant files you have modified or created (e.g., .c, .h, Makefile, write up). Object files (e.g., .o, bzImage, vmlinuz) or auto-generated headers as part of the compilation (e.g., include/generated/*) must **not** be added to the repository.

Do NOT "`git add *`" unless you know what you are doing.

The code that will be graded has to be pushed to the remote origin (GitHub) by the deadline. If you miss any file and submit it late, there will be a **late submission penalty**.

The fresh clone of the submitted version of your repository should be compilable on the grader's machine. **Do NOT use absolute file paths for header file inclusion and in Makefile** (e.g., `#include "/home/user/foo/bar/file.h"` – this will likely cause compile errors when compiled on a different machine). Use relative paths! If you are unsure: (1) clone your team's repo from the server in a new directory, and (2) check if all files exist and your code is compiled properly.

IMPORTANT: Late submission will be determined by the time when you submit the SHA to eLearn. If you submit SHA multiple times, the last one will be used. We will **NOT** use git repo's commit time since it only reflects your VM's local time (and can be easily faked).

For grading, we will create a snapshot of your **main** branch with the specified **SHA** . So make sure your final version corresponding to the SHA is in the main branch (check `git branch`) and it is pushed to the GitHub server. This also means that we will not check your other branches or any other commits.

6. Tips

6.1. General

- Use `man` command to get help on standard Linux utilities like `grep`.
- `printk()` is your best friend in the kernel world.
- Use `dmesg` to see kernel messages and errors. You can also read `/var/log/kern.log`
- Use `grep` to navigate through the code. For example, to see the usages of `SYSCALL_DEFINE`, change to the kernel directory (e.g., `cd ~/rtes`) and type:

  ```
  $ grep -rnI -C3 SYSCALL_DEFINE . | less
  ```

- You can never access kernel memory space directly from the user level. To copy between kernel memory space and user memory space, make use of the kernel functions **`copy_to_user`** and **`copy_from_user`**. If success, they return 0.
- You can browse the kernel source code and search definitions/references of variables/functions/macros at https://elixir.free-electrons.com/linux/v5.11.22/source

- Pay attention to kernel compile logs. Check if your added source file appears in the log and if it is properly compiled without errors.

6.2. System Calls

- Review the difference between **declarations** and **definitions** in C. You may need both, each in the right place. Also review what ‘`extern`’ does.
- What makes your kernel function a syscall? Look into **`syscall_64.tbl`** and auto-generated header files in the x86 subdirectory.
- Whenever lost, take an existing syscall, e.g. `sched_setparam` as an example.
- Make use of **`SYSCALL_DEFINEx`** macros when defining your syscall function (e.g., SYSCALL_DEFINE2). Note that this macro prepends “`sys_`” to your syscall name. So if your systemcall name declared in `syscall_64.tbl` is `sys_foo` and has two integer arguments, you have to define the systemcall function by: `SYSCALL_DEFINE2(foo, int, arg1, int, arg2)`. Use `grep` to check how these macros are used by the kernel code.
- You need to include the correct header files in order to call kernel functions or to use macros defined in different files (this is obvious; the kernel is a C program). For instance, to use `SYSCALL_DEFINEx` macros, include `<linux/syscalls.h>`.
- **`EXPORT_SYMBOL()`** is a kernel macro to expose the variables or functions of the kernel image to loadable kernel modules. Recall that modules are not part of the kernel image (they are linked dynamically when loaded). So if your module code needs to access any kernel variable/function that is not exported, you will see a compile error or loading error.
- How to test your system call? Type '`man syscall`' in your terminal.

6.3. Misc Drivers

- For some example code, see https://embetronicx.com/tutorials/linux/device-drivers/misc-device-driver/

References

1. Built Your Own Kernel. https://wiki.ubuntu.com/Kernel/BuildYourOwnKernel
2. Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, Linux Device Drivers, Third Edition, O'Reilly, 2005. https://lwn.net/Kernel/LDD3/
3. An Introduction to Makefiles. https://www.gnu.org/software/make/manual/make.html#Introduction
4. Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel," O'Reilly Media, ISBN: 978-0596005658, 2005 (3rd Edition).
5. Robert Love, "Linux Kernel Development," Addison-Wesley Professional, ISBN: 978-0672329463, 2010 (3rd Edition).