Lab 1: Introduction to Kernel Programming

18-648: Embedded Real-Time Systems

Out: September 12, 2016, 12:00:00 AM EDT
Due: September 28, 2016, 11:59:59 PM EDT

1 Introduction

You should learn the following from this lab:

- Running native application-level C code on Android.
- Creating kernel modules for the Linux kernel
- Adding new system calls to the Linux kernel.
- Accessing process information in the kernel data structures.

Before you start hacking:

- Read the entire lab handout before starting work: writeup questions and the tips section aim to give you hints about design and implemention.
- If you complete any work separately, make sure to explain to each other how things work in detail, incl. the writeup questions. During demo we may ask any teammate about any part of the lab.
- Before proceeding, please complete all steps to setup your development environment [3].
- Whenever the lab 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. You are an RTOS vendor!

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 applications 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 (some are from Busybox):

- `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 [1] (online).

2.2 Character Device Drivers

A device driver is a part of the kernel that implements an interface to a hardware device. A characteristic feature of *nix systems is that this interface is usually a file abstraction. A Linux character device is an abstract file that supports stream access but not random access. The driver implements the supported file operations, including `open`, `close`, and `read`.

Each character device driver implements a *device type*. To access a device, an instance of the particular type needs to be created. The `mknod` utility can be used to create a device instance within the `/dev/` path.[1]

2.3 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, 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. The system call table is defined in `calls.S`.

2.4 Processes and Threads

The kernel object corresponding to a program execution flow is a *thread*. One or more threads that share an address space, file descriptors, and other resources compose a *process*. Each thread is identified by a thread ID (TID) and each process is identified by a process ID (PID) also known as a thread group leader ID (TGID). Every process in the system, except the initial process, has a parent process.

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

Every thread running under Linux is dynamically allocated a `struct task_struct` structure. The task structure is declared in `sched.h`. The set of threads on the Linux system is stored in a circular, doubly-linked list.

Each process is associated with a priority level that controls the share of the CPU resource that the process gets relative to other processes. The priority value range is divided into general-purpose priorities and real-time priorities. All real-time priorities supercede all general-purpose priorities. A *real-time* process is a process whose priority value is within the real-time priority value range. The `chrt` tool can be used to give a process real-time priority.

[1]`mknod` is part of Busybox. See Section 8 in [3] if you haven't installed it yet.

3 Build setup

Your source files will fall into three categories:

Category	Location in kernel repo	Build method
built-in kernel code	`rtes/kernel`	together with the kernel
kernel modules	`rtes/modules`	standalone using the kernel build system
userspace apps	`rtes/apps`	standalone using a toolchain for user space

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 make is [4]. The kernel should be buildable with its standard existing Makefile. For each module and app you will need to add a Makefile so that running `make` in the respective subfolder builds the module or app. This is necessary for the class server to build your source.

3.1 Built-in kernel code

1. Create `rtes/kernel/Kbuild` file into which you will add each source file (not header file) which you'd like to be included into the kernel binary. For example, to include `somefile.c` add a line:

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

2. Edit `Makefile` in top directory of the kernel source tree to add `rtes/kernel/` to the list of subdirectories assigned to `core-y` list on the line that looks like:

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

Please do not modify the root kernel makefile beyond this.

Tip: Remember to have the `ARCH` and `CROSS_COMPILE` environment variables set when running make. See Section 9 in [3]. You could add these `exports` to `~/.bashrc` to have them set automatically in your shells, however that might lead to confusion in the occassional case when you don't want them set.

3.2 Kernel modules

Since kernel modules are built using the kernel build system, each module will also need a `Kbuild` file in the module directory. The file contents needs to be of the form

```
obj-m += modulename.o
modulename-y += another_source_file.o
```

where `modulename` would be your module name and the second kind of line can be repeated to add more source files (your code is modular, isn't it?). Note that each module should be in a separate directory (within `rtes/modules`) and needs to have one main source file named `modulename.c`.

To build it, run from the top-level kernel directory

```
    $ make M=rtes/modules/modulename
```

The `M` argument is a path to the module directory.

To build with a simple `make` from the module's directory, create a Makefile in the module's directory with the similar `M` argument but with a `-C` to have make navigate into the top-level kernel directory:

```
all:
	make -C ../.. M=$(PWD)
```

Note that by makefile syntax, the commands begin with a **mandatory tab**, not spaces.

Tip: As above, remember `ARCH` and `CROSS_COMPILE`.

3.3 Userspace apps

To compile userspace, we need a *sysroot* (see Section 5.3 in [3]). The sysroot includes kernel headers, core library headers, and core library binaries. Earlier you've compiled Busybox using the sysroot shipped with the Android NDK. The kernel headers that were used were the headers for the stock kernel included with the NDK. However, now you need to use the headers for *your* kernel. Otherwise, how would the userspace know about your cool new syscalls?[2]

First, make sure `ARCH` environment variable is set as usual and export the headers for your kernel (into `usr` directory at to level of kernel tree):

```
$ make headers_install
```

Note that this is not a plain copy of `.h` files: the raw header code needs to be processed, and only a subset of all files needs to be selected based on architecture.

Tip: Remember to re-export your headers whenever you change a userspace-facing header.

Now, you can use the NDK toolchain with the cross-compilation command similar to that of Busybox, but precede the NDK include paths with the path to the exported headers so that it overrides the stock kernel headers. Note that you still need the other include paths since kernel headers are just one piece of the sysroot. For example, to compile and link an app from `hello.c`, create a Makefile containing:

```
EXTRA_CFLAGS=see Busybox compilation in Section 8 of Dev Env setup
EXTRA_LDFLAGS=see Busybox compilation in Section 8 of Dev Env setup
EXP_HDRS=$(HOME)/kernel/usr/include
CC=arm-linux-androideabi-gcc

hello: hello.c
	$(CC) -o hello -isystem $(EXP_HDRS) $(EXTRA_CFLAGS) $(EXTRA_LDFLAGS) hello.c
```

Note that by makefile syntax, the commands begin with a **mandatory tab**, not spaces.

4 Assignment

4.1 Writeup (10 points)

Submit a plain text or PDF document with **brief** answers to the following questions to the `writeups` repo.

1. (2 points) How does a system call execute? Explain the steps in detail from making the call in the userspace process to returning from the call with a result.
2. (1 point) Define re-entrancy and thread-safety.
3. (1 point) What does it mean for a kernel to be *preemptive*? Is the Linux kernel you are hacking on preemptive?
4. (2 points) When does access to data structures in userspace need to be synchronized?
5. (2 points) What synchronization mechanism can be used to access shared kernel data structures safely on both single- and multi-processor platforms?
6. (2 points) What is the `container_of` macro used for in the kernel code? In rough terms, how is it implemented?

[2]No, magic numbers are not an adequate solution.

4.2 Native Android Application (5 points)

Source code location: `kernel/rtes/apps/hello/hello.c`

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

```
Hello, world! It's warm and cozy here in user-space.
```

Build it (Section 3.3), copy the compiled binary onto the device using `adb push`, run it in the shell, and check its output.

4.3 Loadable Kernel Module (10 points)

Source code location: `kernel/rtes/modules/hello/hello.c`

Write a loadable kernel module which upon loading on the target device prints the following to the kernel log:

```
Hello, world! Kernel-space -- the land of the free and the home of the brave.
```

4.4 Character Device Driver (15 points)

Source code location: `kernel/rtes/modules/psdev/psdev.c`

Create a module with a device that when read will output a list of *real-time* threads with their thread IDs, process IDs, real-time priorities, and command names. Reading from your device with e.g. `cat /dev/psdev` should print, for example:

```
tid    pid    pr    name
102    102    90    adbd
103    103    80    pigz
104    103    80    pigz
```

The user should be able to create and use ***multiple, independent*** device instances. The number of file descriptors that can be associated with one device instance at a time should be limited to one. When the limit is exceeded, the `open` system call must fail with `EBUSY`. Operations that are part of the file abstraction, but do not make sense for your device should fail with `ENOTSUPP`. As with any file, the user should be able to read the contents in chunks over multiple read operations. Note that the file abstraction allows the user to issue concurrent reads on the same file descriptor.

4.5 Linux System Calls (30 points)

4.5.1 Write a calculator system call (5 points)

Source code location: `kernel/rtes/kernel/calc.c`

Add a new system call called `calc` that takes four input parameters: two *non-negative rational* numbers as character strings, an operation (`-`, `+`, `*`, or `/`) as a character, and a buffer to store the result of applying the given arithmetic operation to the two numbers as a character string.[3] Support a maximum number magnitude of at least 16-bits and a resolution of at least 10-bits after the decimal point. Fail with `EINVAL` when the result is not a number or the input parameters are not valid.

The system call should have the following format:

```
sys_calc(const char* param1, const char* param2, char operation, char* result)
```

[3]In-kernel arithmetic will come in handy for later labs.

4.5.2 Write a calculator application (5 points)

Source code location: `kernel/rtes/apps/calc/calc.c`

Write a user-level application that takes two non-negative rational numbers and a character operation from the command line (escaped with single quotes when needed), makes the `calc` system call, and outputs to `stdout` nothing but the result as a number in base 10 or `nan`:

```
$ ./calc 3 + 0.14
3.14
$ ./calc 3.14 / 0
nan
```

Note: Please strictly follow the format specified in the example for the parameters.

4.5.3 Override syscalls at runtime (5 points)

Source code location: `kernel/rtes/modules/cleanup/cleanup.c`

Write an LKM that when loaded intercepts when a process exits and reports the paths of files it left open to the kernel log. Implement this strictly by overriding relevant syscalls without changing the process exiting logic. The override should be active only while the module is loaded.

Allow limiting the effect of your file-close-policing module to only processes whose name (`comm`) contains a substring passed as a *kernel module parameter*, specified as follows:

```
$ insmod cleanup comm="sloppyapp"
```

A report should be visible in `dmesg` after a process exits without having closed one or more files. There should be no output at all if no files were left open. The output should be formatted as follows:

```
cleanup: process 'sloppyapp' (PID 123) did not close files:
cleanup: /path/to/file1
cleanup: /path/to/file2
```

4.5.4 Process Information Syscalls (15 points)

Source code location: `kernel/rtes/kernel/ps.c`

Add a new system call `count_rt_threads` that can be used to obtain an integer value that equals the total number of *real-time* threads that currently exist in the system (see Section 2.4).

Add a new system call `list_rt_threads` that can be used to obtain a data structure with the list of real-time threads that currently exist in the system. The following attributes should be available for each thread: TID, PID, real-time priority and name (command).

4.6 RTPS (20 points + 5 bonus)

Source code location: `kernel/rtes/apps/rtps/rtps.c`

Write an application called `rtps` that displays a list of real-time threads that exist on the device in descending order of real-time priority. Make use of your syscalls created in Section 4.5.4. The application must not quit until you hit Ctrl-C and must print updated information every 2 seconds. See Section 3.3 for building.

4.6.1 Bonus: persistent terminal interface (5 points)

The terminal screen must *not* scroll each time it updates. The updated information should cleanly overwrite the old information, much like the program `top`. Only as many processes as fit into the current terminal size should be displayed.

Tip: Look into the `ncurses` library. You can compile it with the NDK toolchain (with a sysroot), much like Busybox. Then, you can pass the resulted compiled binary to the linker (when you build your dependent app) using `-L` and `-l` arguments.

4.7 Demo (10 points)

Each group will also have to *demonstrate* their lab work in action to the TAs. The date and schedule of the demos will be announced via Piazza. The server will build all binaries from your submitted source prior to the demo.

5 How to Submit?

You must use `git` to submit all your work for this lab, including the answers to the writeup questions. Please make sure all your commits to the source code and the writeup are pushed into the master branch of the respective repos. The writeup can be plain text or a PDF.

Make sure you don't miss any files (incl. makefiles) by checking `git status` before and after your commits. Finally, make sure your commits are pushed into the master branch of the remote repo by checking `git log origin/master`. The code that will be graded and demoed has to be in the remote repo master by the deadline. This means: push frequently as you complete each part, but always make sure that what is in the remote master builds successfully. If you want to share unfinished code, push into another branch of your making.

See Section 5.4 in Development Environment Setup handout for an example work flow.

6 Tips

6.1 General

- Use `man` *`command`* on your host to get help on standard Linux utilities like `grep`.
- GNU `screen` utility is one way to not drown in a sea of terminal windows.
- See Section 3.3 for building the userspace applications assigned for you to write.
- `printk()` is your best friend.
- Use `dmesg`. Read `man dmesg` on your host.
- See `/proc/last_kmsg` after a crash.
- Use `grep` to navigate through the code,

  ```
  $ grep -rnI -C3 --color=always string folders
  ```

  for convenience you could:

  ```
  $ echo 'alias sgrep="grep␣-rnI␣-C3␣--color=always"' >> ~/.bashrc
  $ . ~/.bashrc
  ```

  then, e.g. to see usages of `SYSCALL_DEFINE`:

  ```
  $ sgrep SYSCALL_DEFINE kernel/ include/ | less
  ```

- If you find yourself tempted to include headers with `#include "asystemheader.h"` instead of `#include <asystemheader.h>`, then your cross-compilation environment with sysroot is not setup properly. See Section 3.3.
- The OS cannot and should not trust that users will do the right thing. In fact, the OS should assume that the users can be malicious or totally stupid, and provide invalid arguments all the time. Your system call must check whether it is safe to write into user-space memory before writing to it. Otherwise, you're asking for *panic*.
- 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 function `copy_to/from_user`.
- Remember to protect accesses to shared data structures. Some of the ones you will need are protected by the RCU mechanism [2]. When in doubt, look at how the needed data structure is accessed elsewhere in kernel code.
- Feel free to split your implementation across multiple files for modularity, but (1) make sure all files are committed, (2) added to makefile and (3) keep the name of the main file and the corresponding compiled binary as specified in Source Code Location entries throughout the assignment.
- As you work on the labs, consider keeping as much as possible of your code in LKMs instead of incorporating it directly into the kernel binary. Consider only inserting hooks into the needed places in the kernel and link those hooks to function handlers using your own custom *dispatch* table. The table could be dynamically populated with function pointers when your module is loaded and unloaded. This will minimize the recompiles, relinks, and reboots that would need to happen as you edit your code. Similarly, consider storing your own fields in a separate struct and only pointing to it from the task struct – this would minimize changes to `sched.h`, which triggers a rebuild of large portion of the kernel due to its being low in the dependency graph. These are just some completely optional ideas for your consideration.

6.2 Character Devices

- The driver should be contained within an LKM, which should be cleanly loadable and unloadable.
- The opposite of `mknod` is `rm`.
- The kernel should allow unloading the module only when there are no device instances present.
- Make use of the EOF byte in your implementation.

6.3 Linux System Calls

- Review the difference between *declarations* and *definitions* in C: you will need both, each in the right place. You will also declare the syscall handler and the syscall number separately.
- Make use of `SYSCALL_DEFINE*` macros when defining your syscalls.
- Lookup what `asmlinkage` is for.
- Declare your syscalls (see note below regarding the `?`):
  - in `?/syscalls.h`, and
  - in `?/unistd.h` for use from userspace

- There are multiple `syscalls.h` and `unistd.h` files in the kernel tree. Understand why and understand what is the meaning of each parent directory that has one (hint: architecture). Then, you should be able to figure out in which one of them your syscall (function and number) declarations belong and which (exported) one you should include from userspace to get those declarations (hint: which parts of your declaration are arch-dependent and which aren't?).
- What makes your function a *syscall*? Look into the `calls.S` file which is inside the kernel for the ARM architecture.
- Whenever lost, take an existing simple syscall, e.g. `sys_exit` as an example.
- Can anything go wrong inside your syscall implementation? Does your design allow for detecting and reporting all failure conditions? Investigate how a syscall implementation can cause libc syscall wrappers to set `errno`.
- If to override a system call, you are tempted to type an explicit memory address into your source code, consider what would be enough to break that "solution"? Consequently, this temptation has to be resisted to receive credit.
- Read about `EXPORT_SYMBOL` (included via `linux/module.h`)

6.4 In-kernel arithmetic

- Oh, no! What happened to `float`?
- Keywords: fixed-point, bitshift operators.

6.5 Process information

- Look into `sched.h` for task structure definition.
- `grep` inside the core kernel folder for the macro `for_each_process`.
- Use the syscall `count_rt_threads` first to find out the total number of processes and allocate the appropriate amount of memory using `malloc` or `calloc`. You can then pass the pointer returned by `malloc` to the syscall `list_rt_threads`. Remember: the process count can change between the two calls, so use a crash-proof mechanism in `list_rt_threads` to avoid a race-condition.

7 CIT Plagiarism Policy

Please read and understand the CMU-CIT Plagiarism Policy. All work submitted for grading are subject to the policy, which will be strictly enforced.

References

[1] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux Device Drivers*, Third Edition, O'Reilly, 2005 http://lwn.net/Kernel/LDD3/

[2] Paul McKenney. What is RCU, Fundamentally? in Linux Weekly News (LWN), December 2007. http://lwn.net/Articles/262464/

[3] Development Environment Setup Guide on Piazza (`devenv.pdf`).

[4] An Introduction to Makefiles. http://www.gnu.org/software/make/manual/make.html#Introduction