OS Project 1 Report

ZHANG Chi 518021910395

Date: April 25, 2020

Contents

1 Introduction

Here I highlights some works I've done in this project.

- I tried to build x86 kernel module with latest SDK. (Section 2)
- I figured out where `0×c000d8c4` comes from, and automatically extracted it from kernel source. (Section 2.5)
- I made intensive tests on my kernel module, where I tried to attack kernel, and issued concurrent requests to kernel. (Section 3)
- I found out why we can't access user data when holding read lock, which may relate to lazy page allocation. (Section 4)

- With the help of condition variable, cashiers can know which customer they're serving. (Section 5)
- Burger Buddies' Problem output can be verified automatically. (Section 6)

1.1 TL;DR How to test this project

After starting up emulator, just run these three commands, and everything will be automatically tested and run.

```
make all
make test_bbc
make run_bbc BBC_PARAMETER="2 4 41 10"
make rmmod
```

For other things you can play with (or documentation), refer to README.md.

2 Use x86 emulator and latest SDK

I have a weird hobby, that is to say, using latest version of every tool. Upon figuring out that we're working on Linux 3.4 and 24.x SDK, I came with the idea to upgrade to latest SDK. Although I succeeded in running modified kernel on latest emulator, I still did this project on tools that our teachers provide. But from my perspective, this experience may benefit students who attend this course next year, as x86 emulator is much faster than current arm one.

2.1 Compile goldfish 3.10

After downloading latest Android SDK and NDK from Android Development website, we can compile goldfish 3.10 kernel with kernel module enabled.

```
git clone https://aosp.tuna.tsinghua.edu.cn/kernel/goldfish/ \
    -b android-goldfish-3.10 --depth=1
```

Then we may run `make x86_64_ranchu_defconfig` to obtain a default `.config` file. Add these four lines in it to enable kernel module support.

Aside: what is "ranchu"? According to AOSP doc[1], it is a new virtual QEMU board.

```
CONFIG_MODULES=y
CONFIG_MODULE_FORCE_LOAD=y
CONFIG_MODULE_UNLOAD=y
CONFIG_MODULE_FORCE_UNLOAD=y
```

Now we may build goldfish 3.10. This kernel is supported by latest Android emulator.

Note: we may need to download the course-provided version of NDK, as the latest version provides LLVM instead of GNU toolchain.

```
export PATH="$ANDROID_NDK_HOME/toolchains/x86_64-4.9/prebuilt/linux-x86_64/bin":
    $PATH
make -j4 ARCH=x86_64 CROSS_COMPILE=x86_64-linux-android-
```

We'll get `bzImage` afterwards.

2.2 Build x86 Kernel Module

Use this Makefile to build kernel module in x86 mode.

```
obj-m += android_module.o
KID := ~/goldfish
CROSS_COMPILE=x86_64-linux-android-
CC=$(CROSS_COMPILE)gcc
LD=$(CROSS_COMPILE)ld

all:
    make -C $(KID) M=$(shell pwd) ARCH=x86_64 modules

clean:
    make -C $(KID) M=$(shell pwd) ARCH=x86_64 clean
```

Place `android_module.c` in the same directory.

```
#include "linux/module.h"
#include "linux/kernel.h"

int init_module(void)
{
  printk(KERN_INFO "Hello android kernel...\n");
  return 0;
}

void cleanup_module(void)
{
  printk(KERN_INFO "Goodbye android kernel...\n");
}
```

Run the following command to compile.

```
export ARCH=x86_64
export CROSS_COMPILE=$ANDROID_NDK_HOME/toolchains/x86_64-4.9/prebuilt/linux-x86_64/
    bin/x86\_64-linux-android-
make
```

And we'll get `android_module.ko` in current directory.

2.3 Test Kernel Module

Configure AVD with AVD Manager in Android Studio or in whatever way, with name `OsPrj`.

```
emulator -avd OsPrj -kernel bzImage -show-kernel -no-window
```

Push kernel module to the VM, and install it.

```
adb shell
insmod android_module.ko
lsmod
dmesg
```

This completes the development setup. x86 emulator boots in 5 secs, which is much faster.

2.4 Difference between x86 and arm

When I was building a x86 kernel module on Linux 3.10, I found there're several differences from arm.

- **Page Permission** x86 version of Linux makes the page storing syscall table read-only. Therefore, we had to patch page table before registering a new syscall.
- **Syscall Number** x86 has fewer syscalls than arm. Therefore, syscall number range is limited.
- **Syscall Table Location** x86 saves syscall table in a different location, and has two separate syscall table for 32-bit applications and 64-bit applications. (There's only one in arm, and syscall table is at 0xc000d8c4). Therefore, I wrote a script to automatically extract syscall location from kernel source.

As there're many difference between x86 and arm, at last, I gave up using x86 for programming and debugging, and traded speed for a more reliable project environment.

In module directory, `gen.sh` contains script to extract syscall table location from `System.map`.

3 Make Syscall Robust

It's important to make a kernel module safe. If a kernel module is prone to user space attack, then every application on device may gain whole control of computer system. This is very dangerous.

In this project, I enforced kernel module safety by accessing user space safely, and writing "destructive" tests.

3.1 Access User Space Safely

When we're writing kernel code, we should never trust user parameters. Therefore, I use `copy_to_user` to copy data to user buffer, and use `get_user` and `put_user` to access user variable nr. These functions have signature like this.

```
unsigned long copy_to_user(void __user *to, const void *from,
                           unsigned long count);
unsigned long copy_from_user(void *to,
                           const void __user *from, unsigned long count);
```

We can check the return value. If it is -EFAULT, the user may have not properly allocate memory for buffer array, or he or she is attacking the kernel. In this situation, we should free all resources and return to user space.

3.2 Write Test to Ensure Safety

In ptree_test directory I made some unit tests on kernel module. Tests include:

- **Safety test** I request ptree syscall with NULL or pointer to kernel space, and the kernel module should reject this request.
- **Functionality Test** I use array smaller than number of tasks in kernel, use negative nr, etc. to test the functionality of kernel module.

3.3 Concurrent is Fun

Our syscall should handle concurrent requests. Therefore, I also included a test that issues 50 syscalls at a time in test script.

4 Investigate Unexpected Page Faults

When I was testing my kernel module, I found that in some situation, kernel may get into unexpected page fault. From my observation, this will happen when the following conditions are met.

- User allocates a very large array (more than 4 pages).
- Using read-lock in kernel space.

These clues quickly remind me of how paging works in kernel. Though I have no experience with Linux kernel programming before, I suspect that user pages are not immediately granted. Instead, they're allocated lazily. When using malloc or creating a very big array, the kernel just marks these pages as "phantom" pages, and will actually allocate the page only when the user writes data into it.

And in this situation, if I enabled read-lock in kernel, the kernel cannot turn this kernel thread into sleeping state. That is to say, the kernel cannot allocate memory for these "phantom" user pages, when we access user buffer in kernel.

The solution is easy. Just use kmalloc to create a buffer in kernel before locking task list, and after releasing the lock, copy kernel buffer back into user buffer.

From Linux Device Drivers book[2], I found the recommended way of using kmalloc in buffer. The programming pattern can be concluded as follows.

```c
int* buffer = kmalloc(100 * sizeof(int), GFP_KERNEL);

if (doA()) goto error1;
int* buffer2 = kmalloc(100 * sizeof(int), GFP_KERNEL);
if (doB()) goto error2;

// success
return 0;

// different failure point
error2:
kfree(buffer2);
error1:
kfree(buffer);
return -EFAULT;
```

In this way, we can free kernel resource before returning to user space when faulting. Edsger Dijkstra said that ”Go To Statement Considered Harmful”. In modern programming language, there’re simpler and safer way to handle resource allocation when faulting. For example, we can use RAII in C++/Rust, or use `defer` statements in Go language, thus eliminating the need of using `goto`.

This issue was also made into a test case in `ptree_test`.

5 I know who comes

I’ve implemented burger buddies with condition variable (`burger_buddies_cond`). Customer will notify cashier that he/she has come, and cashier will notify a customer that he/she has prepared the burger. In this way, cashier itself can know which customer is ordering. On contrast, in the semaphore version, only customers know which cashier is serving, but the cashier can’t know who is being served (as we cannot query who acquired the lock).

This is very similar to IPC on operating systems, that one process wake up another process waiting on pipe. Here condition variable (along with mutex) is such a ”pipe”, and message is written and read from a single variable bound to that condition variable.

6 Testing Burger Buddies

By going through log entries from Burger Buddies, we can automatically check if there’s anything wrong with the program. Here I do four checks.

1. **Check log entries size** There must be at least 100 entries in a second, or there might be deadlock.
2. **Burger on Rack** Burgers made should never exceed rack size, and it should never be negative.

3. **Customers come and leave** If a customer come and come without leaving or being served, there must be issue.
4. **Cashiers wait and serve** Cashiers must wait until there're customers, and serve them if there's burger on rack.

References

[1] AOSP docs on android kernel
`https://android.googlesource.com/platform/external/qemu/+/emu-master-dev/android/docs/ANDROID-KERNEL.TXT`

[2] Linux Device Drivers, 3rd edition
`https://lwn.net/Kernel/LDD3/`