VE482 — Introduction to Operating Systems

OS challenges

Manuel — UM-JI (Fall 2021)

Challenge setup

- No deadline
- Optional but open to all
- Rewarded by a bonus on the final grade

1 General setup

1.1 Goal

The goal of this challenge is to learn more about the Linux kernel and do some actual programming at the operating system level. The challenge consists of a slightly adjusted version of the Eudyptula challenge. Students reaching task 10 can become part of the Linux kernel contributors!

1.2 Rules

List of rules to follow:

- The tasks are to be completed in increasing order;
- Unless specified otherwise, kernel 5.2 should be used as reference kernel;
- When asked to build a module, or run a specific kernel, provide clear evidences of the work accomplished;
- All the code should be tested and working on the bear device, i.e. not only in a virtual machine;
- Patches are to be written and submitted in a way that is acceptable for merging in the kernel source tree;
- Each completed task results in a bonus of 0.4 mark, up to task 10. Starting from task 11 each one is worth 0.5 mark; If less than five tasks are completed then **no bonus** is awarded;

Note: besides providing the source code, decide on the best (simple and clear) way to prove a task has been completed.

2 Tasks

Task 1

Write a Linux kernel module, and stand-alone `Makefile`, that when loaded prints to the kernel debug log level, "Good morning ve482!". Be sure to make the module be able to be unloaded as well.

The `Makefile` should build the kernel module against the source for the currently running kernel, or, use an environment variable to specify what kernel tree to build it against.

Task 2

The tasks for this round is:

- Download Linus's latest git tree from git.kernel.org.
- Build it, install it, and boot it. Ensure `CONFIG_LOCALVERSION_AUTO=y` is enabled.

Hints: look into the `make localmodconfig` option, and base your kernel configuration on a working distro kernel configuration; Write a simple to answer all the different kernel configuration options;

Task 3

The task for this round is to take the kernel git tree from Task 2 and modify the `Makefile` as well as the `EXTRAVERSION` field. Do this in a way that the running kernel (after modifying the `Makefile`, rebuilding, and rebooting) displays "ve482challenge" in the version string. Prepare a patch for the modified `Makefile`.

Hint: refer to the file `Documentation/SubmittingPatches`.

Task 4

Part of the job of being a kernel developer is recognizing the proper *Linux kernel coding style*. The full description of this coding style can be found in the kernel itself (`Documentation/CodingStyle`). There is also a tool (`checkpatch.pl` in the `scripts` directory) available from the kernel source tree.

The task for this round consists in refactoring the module from the first task such that it meets the requirements of the Linux kernel coding style.

Note: it is important that everyone follow the same standard so that the patterns become consistent. In other words, you want to make it really easy for other people to find the bugs in your code, and not be confused and distracted by other unimportant things (e.g. prefer spaces instead of tabs for indentation).

Task 5

Take the kernel module from task 1, and modify it so that when a USB keyboard is plugged in, the module gets automatically loaded by the correct user-space hotplug tool (`udev`, now part of `systemd`).

Hint: refer to chapter 14 of *Linux Device Drivers*, 3rd edition.

Task 6

The task this time is as follows:

- Take the kernel module you wrote for task 1, and modify it to be a misc char device driver.
- The misc device should be created with a dynamic minor number.
- The misc device should implement the read and write functions.
- The misc device node should show up in `/dev/challenge`.
- When the character device node is read from, it returns "ve482challenge!".

- When the character device node is written to, the data sent to the kernel needs to be checked. If it matches the string "ve482challenge", then return a correct write return value. Otherwise return the "invalid value" error value.
- The misc device should be registered when the module is loaded, and unregistered when it is unloaded.

Note: the misc interface is a very simple way to be able to create a character device, without having to worry about all of the `sysfs` and character device registration mess.

Hint: no need to reserve a real minor number for the test module.

Task 7

The tasks this round consist in downloading today's "linux-next kernel", build it, and boot it.

Hint: refer to the Linux kernel is development process in `Documentation/development-process/` in the kernel source itself.

Task 8

In this task we focus on the `debugfs` user/kernel interface. It is rumored that the creator of `debugfs` said that there is only one rule for `debugfs` use: "There are no rules when using debugfs." So let's take them up on that offer and see how to use it.

First ensure `debugfs` is (i) enabled in the currently running kernel (`CONFIG_DEBUG_FS` option) and (ii) mounted (usually mounted on `/sys/kernel/debug/`).

The task is described as follows:

- Take the kernel module from task 1, and modify it to create a `debugfs` subdirectory called `ve482challenge`. In that directory, create three virtual files called `challenge`, `jiffies`, and `info`.
- The file `challenge` should operate just like it did in task 6; reuse the same logic, and ensure the file is readable and writable by any user.
- The file `jiffies` is to be read by any user, and when read, should return the current value of the jiffies kernel timer.
- The file `info` needs to be writable only by root, but readable by anyone. When writing to it, the value must be stored, up to one page of data. On a read the file content should be displayed.
- When the module is unloaded, all of the `debugfs` files must be cleaned up, and any memory allocated freed;

Hint: properly handle the fact that someone could be reading from the file `comment` while someone else is writing to it.

Note: never use `/proc/`, that is reserved for processes, when printing out debugging information, rather use `debugfs`.

Task 9

Another "interface" for interaction between the kernel and the user is `sysfs`. For this task adapt the code from task 8 to `sysfs`. In particular, put the `ve482challenge` directory under `/sys/kernel/`.

Hint: this is not a simple task; refer to `Documentation/kobject.txt`to understand how to use `kobjects` and `sysfs`.

Task 10

For this task, go back to the "linux-next" tree from task 7. Update it, and then do the following:

- Create a patch that fixes one coding style problem in any of the files from `drivers/staging/`
- Submit the code to the maintainer of the `driver/subsystem`.

Hints: remember the `checkpatch.pl` script to ensure the correctness of the patch; finding the proper name and mailing lists to send it to can be done by running `scripts/get_maintainer.pl` on the patch.

Note: if this patch is accepted into the kernel tree, you become an "official" kernel developer!

Task 11

After working with `kobject` and `sysfs` in task 9, we move one level up the tree and start to mess with devices and not raw `kobjects`.

For this task:

- Write a patch against any driver that you are currently using on your machine.
- In that driver, add a `sysfs` file to show up in the `/sys/devices/` tree for the device that is called "id". As you might expect, this file follows the same rules as task 09 as for what you can read and write to it.
- The file is to show up only for devices that are controlled by a single driver, not for all devices of a single type (like all USB devices).

Hint: First you have to figure out which drivers you are using, and where the source code in the kernel tree is for that driver.

Task 12

Now let's step back from drivers, and focus on the kernel core. To do that, we need to go way back to the basics: linked lists.

The kernel has a unique way of creating and handling linked lists, that is quite different than the "textbook" way. But, it turns out to be faster, and simpler, than a "textbook" would describe, so that's a good thing.

For this task, write a kernel module, based on your cleaned up one from task 4. It should do the following:

- You have a structure, named `identity`, with three fields:

```
1   char   name[20];
2   int    id;
```

```
3   bool  busy;
```

- Your module has a static variable that points to a list of these `identity` structures.
- Write a function `int identity_create(char *name, int id)` which creates the structure `identity`, copies in the `name` and `id` fields and sets `busy` to false. Proper error checking for out of memory issues is required. Return 0 if everything went OK, and an error value if something went wrong.
- Write a function `struct identity *identity_find(int id);` which takes a given `id`, iterates over the list of all ids, and returns the proper `struct identity` associated with it. If the identity is not found, return `NULL`.
- Write a function `void identity_destroy(int id);` which given an `id`, finds the proper `struct identity` and removes it from the system.
- Your `module_init()` function should look much like the following:

```
1   struct identity *temp;
2
3   identity_create("Alice", 1);
4   identity_create("Bob", 2);
5   identity_create("Dave", 3);
6   identity_create("Gena", 10);
7
8   temp = identity_find(3);
9   pr_debug("id 3 = %s\n", temp->name);
10
11  temp = identity_find(42);
12  if (temp == NULL)
13    pr_debug("id 42 not found\n");
14
15  identity_destroy(2);
16  identity_destroy(1);
17  identity_destroy(10);
18  identity_destroy(42);
19  identity_destroy(3);
```

Note: do not forget to properly checking return values of the above functions.

Task 13

Those lists are used all over the kernel in lots of different places.

Now that we are allocating a structure that we want to use a lot, we might want to start caring about the speed of the allocation, and not have to worry about the creation of those objects from the "general" memory pools of the kernel.

This task consists in taking the code written in task 12, and cause all memory allocated from the `struct identity` to come from a private slab cache.

Instead of using `kmalloc()` and `kfree()` in the module, use `kmem_cache_alloc()` and `kmem_cache_free()`

instead. You are free to name your memory cache whatever you wish, but it should show up in the `/proc/slabinfo` file.

Hint: do not forget to initialize your memory cache properly when the module is loaded.

Note: Do not send the whole module for this task, but only a patch with the diff from task 12.

Also show the output of `/proc/slabinfo` with your module loaded.

Task 14

Now that you have the basics of lists, and we glossed over the custom allocators it's time to move on to something a bit more old-school: tasks.

For this task:

- Add a new field to the core kernel task structure called `id`;
- When the task is created, set the `id` to your student ID;
- Add a new `proc` file for every task called, `id`, located in the `/proc/$PID/` directory for that task;
- When the `proc` file is read from, have it print out the value of your `id`, and then increment it by one, allowing different tasks to have different values for the `id` file over time as they are read from;

As you are touching files all over the kernel tree, a patch is the required result to be sent in here.

Task 15

For this task we want to create a new system call:

- Add a new system call to the kernel called `sys_ve482challenge`";
- The system call number needs to be the next syscall number for the architecture you test it on;
- The syscall should take two parameters: `int high_id`, `int low_id`;
- The syscall will take the two values, mush them together into one 64bit value (`low_id` being the lower 32bits of the id, `high_id` being the upper 32bits of the id); - If the id value matches your student IDthen the syscall returns success. Otherwise it returns a return code signifying an invalid value was passed to it.
- Write a userspace C program that calls the syscall and properly exercises it (valid and invalid calls need to be made).

Note: document the arch you are using and why you picked this number in the module.

Task 16

Install the tool 'sparse'. It was started by Linus as a static-analysis tool that acts much like a compiler. The kernel build system is set up to have it run if you ask it to, and it will report a bunch of issues in C code that are really specific to the kernel.

When you build the kernel, pass the "C=1" option to the build, to have sparse run on the `.c` file before `gcc` is run. Depending on the file, nothing might be printed out, or something might. Here's an example of it being run on the `ext4` code:

```
ve482sh: make C=1 M=fs/ext4
  CHECK   fs/ext4/balloc.c
  CC      fs/ext4/balloc.o
  CHECK   fs/ext4/bitmap.c
  CC      fs/ext4/bitmap.o
  CHECK   fs/ext4/dir.c
  CC      fs/ext4/dir.o
  CHECK   fs/ext4/file.c
  CC      fs/ext4/file.o
  CHECK   fs/ext4/fsync.c
  CC      fs/ext4/fsync.o
  CHECK   fs/ext4/ialloc.c
  CC      fs/ext4/ialloc.o
  CHECK   fs/ext4/inode.c
  CC      fs/ext4/inode.o
  CHECK   fs/ext4/page-io.c
  CC      fs/ext4/page-io.o
  CHECK   fs/ext4/ioctl.c
  CC      fs/ext4/ioctl.o
  CHECK   fs/ext4/namei.c
  CC      fs/ext4/namei.o
  CHECK   fs/ext4/super.c
  CC      fs/ext4/super.o
  CHECK   fs/ext4/symlink.c
  CC      fs/ext4/symlink.o
  CHECK   fs/ext4/hash.c
  CC      fs/ext4/hash.o
  CHECK   fs/ext4/resize.c
  CC      fs/ext4/resize.o
  CHECK   fs/ext4/extents.c
  CC      fs/ext4/extents.o
  CHECK   fs/ext4/ext4_jbd2.c
  CC      fs/ext4/ext4_jbd2.o
  CHECK   fs/ext4/migrate.c
  CC      fs/ext4/migrate.o
  CHECK   fs/ext4/mballoc.c
fs/ext4/mballoc.c:5018:9: warning: context imbalance in 'ext4_trim_extent' - unexpected unlock
  CC      fs/ext4/mballoc.o
  CHECK   fs/ext4/block_validity.c
  CC      fs/ext4/block_validity.o
  CHECK   fs/ext4/move_extent.c
  CC      fs/ext4/move_extent.o
  CHECK   fs/ext4/mmp.c
  CC      fs/ext4/mmp.o
  CHECK   fs/ext4/indirect.c
  CC      fs/ext4/indirect.o
  CHECK   fs/ext4/extents_status.c
  CC      fs/ext4/extents_status.o
  CHECK   fs/ext4/xattr.c
  CC      fs/ext4/xattr.o
  CHECK   fs/ext4/xattr_user.c
  CC      fs/ext4/xattr_user.o
  CHECK   fs/ext4/xattr_trusted.c
```

```
  CC      fs/ext4/xattr_trusted.o
  CHECK   fs/ext4/inline.c
  CC      fs/ext4/inline.o
  CHECK   fs/ext4/acl.c
  CC      fs/ext4/acl.o
  CHECK   fs/ext4/xattr_security.c
  CC      fs/ext4/xattr_security.o
  LD      fs/ext4/ext4.o
  LD      fs/ext4/built-in.o
  Building modules, stage 2.
  MODPOST 0 modules
```

As you can see, only one warning was found here, and odds are, it is a false-positive, as the ext4 developers know what they are doing with their locking functions...

The task is as follows:

- Run sparse on the drivers/staging/ directory;
- Spot a warning that looks interesting;
- Write a patch that resolves the issue;
- Submit the code to the maintainer of the driver/subsystem;

Note: make sure the patch is correct by running it through `scripts/checkpatch.pl`.

Hint: run the tool `scripts/get_maintainer.pl` on the patch to find the proper name and mailing lists to send the patch to.

Task 17

It is time to start putting the different pieces of what we have done in the past together, into a much larger module, doing more complex things. Much more like what a "real" kernel module has to do.

Go dig up your code from task 6 (the misc `char` device driver) and make the following changes:

- Delete the read function. You don't need that anymore, so make it a write-only misc device and be sure to set the mode of the device to be write-only, by anyone. If you do this right, `udev` will set up the node automatically with the correct permissions.
- Create a wait queue, naming it `wee_wait`;
- In your module `init` function, create a kernel thread named of "ve482challenge";
- At this point the thread's main function should not do anything, except ensuring to shutdown if asked to, and wait on the `wee_wait` wait-queue;
- In the module `exit` function, shut down the kernel thread you started up;

Note: to "prove" that the module works properly check that the thread is created, can be found in the process list, and that the device node is created with the correct permission value.

Task 18

Base all of this work on your task 17 codebase.

Go back and dig up task 12's source code, the one with the list handling. Copy the structure into this module, as well as the `identity_create()`, `identity_find()`, and `identity_destroy()` functions.

Write a new function, `identity_get()`, that looks like:

```
struct identity identity_get(void);
```

It should return the next `identity` structure that is on the list, and remove it from the list. If nothing is on the list, return `NULL`.

Then, hook up the misc `char` device `write` function to do the following:

- If a write is larger than 19 characters, truncate it at 19;
- Take the write data and pass it to `identity_create()` as the string, and use an incrementing counter as the `id` value;
- Wake up the `wee_wait` queue;

In the kernel thread function:

- If the `wee_wait` queue wakes us up, get the next identity in the system with a call to `identity_get()`;
- Sleep (in an interruptible state, such as not to increase the system load in a bad way) for 5 seconds;
- Write out the `identity` name, and `id` to the debug kernel log and then free the memory.

Note: when the module exits, clean up the whole list by using the functions given, no fair mucking around with the list variables directly.

This shows the basics of (i) taking work from userspace, (ii) quickly returning to the user, (iii) going off, (iv) doing something else with the data, and (v) cleaning everything up. It's a common pattern for a kernel, as it's really all that a kernel ends up doing most of the time (e.g. get a disk block, write a disk block, handle a mouse event, etc.)

Hint: you can test with the following script:

```
echo "Alice" > /dev/challenge
echo "Bob" > /dev/challenge
sleep 15
echo "Dave" > /dev/challenge
echo "Gena" > /dev/challenge
rmmod task18
```

Task 19

We now focus on the network side of the kernel, as that is a huge reason for why Linux has taken over the world.

For this task, write a `netfilter` kernel module that does the following:

- Monitors all IPv4 network traffic that is coming into the machine;
- Prints the `id` to the kernel debug log if the network traffic stream contains your student id;
- Properly unregisters you from the `netfilter` core when the module is unloaded;

Task 20

This task requires you to work on the fat filesystem code:

- Add an `ioctl` to modify the volume label of a mounted fat filesystem.
- Provide a userspace .c program to test this new ioctl.

Note: be sure to handle both 16 and 32 bit fat filesystems.

Hint: do this work on either a loop-back fat filesystem on your "normal" filesystem, or on a USB stick. Watch out for locking issues, as well as dirty filesystem state problems.