Linux Kernel Programming Labs

Setup

Simple

- Use Elixir: https://elixir.bootlin.com/linux/v6.6.135/source
- Use grep: `git grep -rn "iterate_supers"`

Modern

- Use modern IDE (nvim, vscode…) with LSP support
- Install clangd (and use the kernel script)
- Install fuzzy finders

virtme-ng

Successor to virtme.
It's a script that allows building and running VMs very quickly and conveniently.

Commands:

- `vng -bv` to compile
- `vng -v` to run a lightweight vm

Git blame

Sometimes the documentation is in the `commit` itself!

Use git blame to understand kernel code.

Debugging

Config options to enable:

- DEBUG_KERNEL
- KGDB
- KGDB_KDB
- MAGIC_SYSRQ
- DYNAMIC_DEBUG
- DETECT_HUNG_TASKS (30s)

Debugging

virtme-ng:

- `vng --gdb` and `vng --debug`
- `vng -p 1 --no-virtme-ng-init --disable-microvm -v --qemu-opts="-serial tcp::1234,server,nowait" -a kgdboc=ttyS1 -a kgdbwait -a nokaslr`

The 8 Kernel Log Levels

- **Viewing logs:** `dmesg --level=err,crit,alert,emerg`
- **Config** In the kernel cmdline: `loglevel=X`

Level	Macro	Severity	Meaning
0	KERN_EMERG	Emergency	System is unusable; usually right before a crash.
1	KERN_ALERT	Alert	Action must be taken immediately (e.g., database corruption).
2	KERN_CRIT	Critical	Critical conditions (e.g., hard drive failure).
3	KERN_ERR	Error	Error conditions (e.g., hardware errors).
4	KERN_WARNING	Warning	Warning conditions that may indicate future problems.
5	KERN_NOTICE	Notice	Normal, but significant conditions (e.g., security events).
6	KERN_INFO	Info	General informational messages (e.g., hardware detected).
7	KERN_DEBUG	Debug	Detailed debug-level messages for developers.

Decode backtrace

```
  [   18.160593] RIP: 0010:print_stats+0xc0/0x1a0 [kcpustat]
  [   18.160687] Code: 49 c7 c1 30 60 37 c0 48 89 41 08 48 89 08 48 89 3b 48 8b 35 02 1e 00 00 4c 89 43 08 4c 39 ce 0f 84 c8 00 00 00 31 c0 0f 1f 00 <48>
8b 4c 06 10 48 01 0c 02 48 83 c0 08 48 83 f8 50 75 ed 48 8b 0e
  [   18.160955] RSP: 0018:ff3cc51501143e80 EFLAGS: 00010246
  [   18.160955] RAX: 0000000000000000 RBX: ff269453430c4300 RCX: ff269453430c4240
  [   18.160955] RDX: ff3cc51501143e80 RSI: dead000000000100 RDI: dead000000000100
  [   18.160955] RBP: 000000000000000a R08: dead000000000122 R09: ffffffffc0376030
  [   18.160955] R10: ff3cc51501143ed0 R11: ff269453430c4360 R12: ff2694534850e400
  [   18.160955] R13: ff26945344392d80 R14: 0000000000000000 R15: ffffffffc0374330
  [   18.160955] FS:  0000000000000000(0000) GS:ff2694537df80000(0000) knlGS:0000000000000000
  [   18.160955] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
  [   18.160955] CR2: 0000563060c70970 CR3: 000000000d03e002 CR4: 0000000000771ee0
  [   18.160955] PKRU: 55555554
  [   18.160955] Call Trace:
  [   18.160955]  <TASK>
  [   18.160955]  my_kcpustat_fn+0x53/0x70 [kcpustat]
  [   18.160955]  kthread+0xe5/0x120
  [   18.160955]  ? \_\_pfx_kthread+0x10/0x10
  [   18.160955]  ret_from_fork+0x31/0x50
  [   18.160955]  ? \_\_pfx_kthread+0x10/0x10
  [   18.160955]  ret_from_fork_asm+0x1b/0x30
  [   18.160955]  </TASK>
  [   18.160955] Modules linked in: kcpustat
```

Decode backtrace

```
  [   18.160593] RIP: 0010:print_stats+0xc0/0x1a0 [kcpustat]
  [   18.160687] Code: 49 c7 c1 30 60 37 c0 48 89 41 08 48 89 08 48 89 3b 48 8b 35 02 1e 00 00 4c 89 43 08 4c 39 ce 0f 84 c8 00 00 00 31 c0 0f 1f 00 <48>
8b 4c 06 10 48 01 0c 02 48 83 c0 08 48 83 f8 50 75 ed 48 8b 0e
  [   18.160955] RSP: 0018:ff3cc51501143e80 EFLAGS: 00010246
  [   18.160955] RAX: 0000000000000000 RBX: ff269453430c4300 RCX: ff269453430c4240
  [   18.160955] RDX: ff3cc51501143e80 RSI: dead000000000100 RDI: dead000000000100
  [   18.160955] RBP: 000000000000000a R08: dead000000000122 R09: ffffffffc0376030
  [   18.160955] R10: ff3cc51501143ed0 R11: ff269453430c4360 R12: ff2694534850e400
  [   18.160955] R13: ff26945344392d80 R14: 0000000000000000 R15: ffffffffc0374330
  [   18.160955] FS:  0000000000000000(0000) GS:ff2694537df80000(0000) knlGS:0000000000000000
  [   18.160955] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
  [   18.160955] CR2: 0000563060c70970 CR3: 000000000d03e002 CR4: 0000000000771ee0
  [   18.160955] PKRU: 55555554
  [   18.160955] Call Trace:
  [   18.160955]  <TASK>
  [   18.160955]  my_kcpustat_fn+0x53/0x70 [kcpustat]
  [   18.160955]  kthread+0xe5/0x120
  [   18.160955]  ? \_\_pfx_kthread+0x10/0x10
  [   18.160955]  ret_from_fork+0x31/0x50
  [   18.160955]  ? \_\_pfx_kthread+0x10/0x10
  [   18.160955]  ret_from_fork_asm+0x1b/0x30
  [   18.160955]  </TASK>
  [   18.160955] Modules linked in: kcpustat
```

```
# write the panic message to a file `bug` and execute the script
./scripts/decode_stacktrace.sh vmlinux . . < bug
```

Introduction to drivers

A **driver** is the code to interact with **devices**.

Introduction to drivers

A **driver** is the code to interact with **devices**.

At its root, a **driver** is simply kernel code accessing the registers/ memory of a device.

Introduction to drivers

A **driver** is the code to interact with **devices**.

At its root, a **driver** is simply kernel code accessing the registers/ memory of a device.

Traditionally, this is kernel code for security reasons (though user-space drivers do exist).

Introduction to drivers

Drivers can define an interface, so that user programs can communicate with the device through files.

Introduction to drivers

Drivers can define an interface, so that user programs can communicate with the device through files.

When exposing such drivers, one needs to specify:

- a major number
- a set of operations

Introduction to drivers

1. Define the operations

```
static const struct file_operations my_driver_fops = {
  .owner          = THIS_MODULE,
  .open           = my_driver_open,
  .read           = my_driver_read,
  .write          = my_driver_write,
  .unlocked_ioctl = my_driver_ioctl,
  .release        = my_driver_close,
};
```

2. Register the driver

```
register_chrdev(major, "my_driver", &my_driver_fops);
```

What the heck is ioctl

An `ioctl` is a syscall just like `read` or `open`.
It's used by drivers to implement complex APIs.

What the heck is ioctl

An `ioctl` is a syscall just like `read` or `open`.
It's used by drivers to implement complex APIs.

```
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long op, ...);
```

What the heck is ioctl

An `ioctl` is a syscall just like `read` or `open`.
It's used by drivers to implement complex APIs.

```
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long op, ...);
```

- The advantage over read/write is the ability to send complex commands.
- No new syscall is needed to implement complex functions with arguments.
- The ioctl can implement many functions, identified using a `MAGIC_NUMBER`.

Implement your ioctl

We need to define a header file used by both the user program and the kernel.

```
#ifndef _HELLO_IOCTL_H
#define _HELLO_IOCTL_H

#include <linux/ioctl.h>

#define HELLO_IOCTL_MAGIC   'N'
#define HELLO_IOCTL_HELLO   _IOR(HELLO_IOCTL_MAGIC, 0, char *)

#endif
```

Implement your ioctl

```
long hello_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
  char buf[256] = "Hello ioctl!";
  void __user *argp = (void __user *)arg;

  switch (cmd) {
  case HELLO_IOCTL_HELLO:
                // return the string to the user... USE copy_to_user() !!
        default:
                // you should return -ENOTTY
  }

  return 0;
}

static const struct file_operations hello_ops = { .unlocked_ioctl = hello_ioctl };
```

Create the dev/ file

Once a driver is registered, the file can be created under `/dev/`

```
mknod /dev/my_driver c 247 0
```

Create the dev/ file

Once a driver is registered, the file can be created under `/dev/`

```
mknod /dev/my_driver c 247 0
```

The same **major** number must be used. It links the fops structure with the file.

Create the dev/ file

Once a driver is registered, the file can be created under `/dev/`

```
mknod /dev/my_driver c 247 0
```

The same **major** number must be used. It links the fops structure with the file.

Modern drivers don't manually create the file for every driver. Rather, they rely on `udev`.

A bit of history

- Pre 2003: every single combination of file was created in userspace using a script `MAKEDEV`, so the `/dev` directory would contain thousands of entries.

A bit of history

- Pre 2003: every single combination of file was created in userspace using a script `MAKEDEV`, so the `/dev` directory would contain thousands of entries.
- Around kernel 2.4: the kernel managed `/dev/` by itself but it was buggy and bloated

A bit of history

- Pre 2003: every single combination of file was created in userspace using a script `MAKEDEV`, so the `/dev` directory would contain thousands of entries.
- Around kernel 2.4: the kernel managed `/dev/` by itself but it was buggy and bloated
- Since the end of 2003: we use `udev`.

A bit of history

- Pre 2003: every single combination of file was created in userspace using a script `MAKEDEV`, so the `/dev` directory would contain thousands of entries.
- Around kernel 2.4: the kernel managed `/dev/` by itself but it was buggy and bloated
- Since the end of 2003: we use `udev`.

Udev

Udev listens to a **netlink** kernel socket

Udev

Udev listens to a **netlink** kernel socket

Every time a driver is registered, the kernel writes a “uevent” in a socket

Udev

Udev listens to a **netlink** kernel socket

Every time a driver is registered, the kernel writes a “uevent” in a socket

Udevd listens to the socket and creates the device file in `/dev/`

Udev

Udev listens to a **netlink** kernel socket

Every time a driver is registered, the kernel writes a “uevent” in a socket

Udevd listens to the socket and creates the device file in `/dev/`

We can check this socket with `udevadm monitor`

Summary

> Usually, there is a module with a set of static functions used internally, alongside other exported functions for the kernel, and a `fops` structure that exposes a subset of this API to userspace.
>
> — What you should remember