Linux Device Drivers 101

Environment Used: multipass as a hypervisor for Linux on MacOS, Linux or Windows.

Packages as Requirements on Linux (Debian Based): `sudo apt install -y build-essential linux-headers-$(uname -r) kmod`

What is Kernel and Device Driver

Kernel schedules processes and manages hardware at a very basic level.

A hardware device has a kernel that manages all the hardware-related resources and for all the devices in that hardware device, drivers are used to configure them. For example, a camera needs a kernel to run it while requires drivers to run its USB ports and display, etc.

When a computer system boots up, the ROM on the motherboard is first storage accessed by the CPU that initialises some essential hardware like the DRAM and storage mediums. Once the DRAM is loaded, a bootloader is fetched from the storage which is responsible for fetching the kernel into DRAM as well as a data structure containing a list of all the devices on the computer called dtb (device tree blob). The CPU is acknowledged about the address of dtb and the kernel and finally, the kernel takes over and the system boots. The dtb is used to

acknowledge the kernel about devices present and load specific drivers to control them.

Kernel Space Vs. User Space

The Kernel works in privileged mode and has access to all the resources of the hardware. User-Space on the other hand is in unprivileged mode and doesn't have direct access to resources. For accessing resources, it needs to go via the kernel which ensures that all the applications in the User Space fairly use resources. For example, reading from an SSD. Kernel ensures that all the applications are making fair use of accessing the storage and not locking it (dominating it) and starving other processes.

Kernel on ARM-A (V8) Processers

The ARM-A (V8) processor lines have execution levels, namely EL0, EL1, EL2 and EL3. The Kernel lives in EL1 where it has access to the status of the CPU whereas User Space lives in EL0 where it doesn't have direct access to the status of registers.

System Calls

To access the resources to which the kernel has access, the User Space uses system calls to communicate with the kernel and make a certain operation happen like read or write to a drive. And to create system calls by developers, it is required to create drivers that will be plugged into the kernel to create those desired system calls with which, a User Space application can access hardware resources.

Types of Drivers

When the system boots up, Kernel has some drivers built into it called the built-in drivers (or say boot time drivers). These drivers are pre-installed and exist in the kernel while it is booted up. On the other hand, drivers that are developed externally, out of the kernel community need to be loaded at runtime on a live system. These are called loadable kernel modules/drivers. These have two methods built into them called a constructor and a destructor. As the name

suggests, the constructor initialises specific things while it is loaded whereas the destructor is executed when the driver is plugged out of the kernel.

Categories of System Calls

Since the kernel manages hardware and files, we are going to particularly work on file operations. Device Drivers can be classified into 3 categories:

- Char: Read and Write operations.
- Block: Storage Devices like SSD and filesystems
- Network: Network-related configurations like ethernet and wireless Wi-Fi cards

Now since everything in Linux is a file, we most need to deal with Char to read and write files in Linux to communicate with the device driver. For example, consider a UART communication device. It is connected to Linux on User Space in a file path, suppose /dev/ttyACMO. You can have a driver which performs read-and-write system calls while you read and write in that particular file location. The driver will be responsible for managing the UART interface and working as per the requirements of the User Space.

This file in the Linux Kernel contains a definition of file operations.

https://github.com/torvalds/linux/blob/master/include/linux/fs.h

A Basic Driver for Linux Kernel

Linux Kernel Drivers are written in C and also in Rust. Here we are gonna focus on C Language.

```
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");                  // * Important since it
MODULE_AUTHOR("Aditya Patil");              // Optional
MODULE_DESCRIPTION("My first linux kernel driver"); // Optional

static int driver_init(void) {
```

```
    printk("Driver has been activated!\n");
    return 0;
}

static void driver_exit(void) {
    printk("Exiting Linux Kernel!\n");
}

module_init(driver_init);   // Macro, not a function (constructo
module_exit(driver_exit);   // Macro, not a function (destructor
```

Also, you need to create a Makefile for automating the build process for the drivers.

```
obj-m += ldd.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) module

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

To load driver into system: `sudo insmod <driver>.ko`

To view kernel log buffer: `sudo dmesg`

To clear kernel log buffer after viewing: `sudo dmesg -c`

To unload driver from the system: `sudo rmmod <driver>.ko`

The Make Utility

Make utility is used for automating the builds from commands. It makes use of a Makefile or .mk files for recipies to build the given project.

Basic Commands

`make`

`make <target>`

`make -f <file>.mk <target>`

Basic Structure of Makefile

```
target1: dependencies
    command1
    command2
    ....


target2: dependencies
    command1
    command2
```

For example:

```
obj-m += ldd.o

all: ldd.c
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) module

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Make sure that tabs are used for spacing commands. Spaces don't work in this case.

lsmod command

The `lsmod` command lists all the drivers installed in the kernel. This is useful for making sure that the device driver has been loaded into the kernel. Using dmesg is not always the most realiable way to make sure that the driver is loaded. Since the constructor may end up returning -1, the destructor will not be called and hence, dmesg will not log the destructor message and hence, can fail to let you know that the driver is unloaded. So use `lsmod` for listing the all the loaded drivers.

insmod command

The `insmod` command is used to load driver to the kernel.

1. Calls `init_module` to hint the kernel that a module insertion is attempted.
2. Tranfers control to the kernel.
3. Kernel execute `sys_init_module`.
4. Verifies permissions.
5. `load_module` function is called:
    a. Checks the sanity of the .ko.
    b. Creates memory.
    c. Copies from user space to kernel space.
    d. Resolves symbols.
    e. Returns a reference to the Kernel.
6. Adds a reference to a linkedlist that has all the loaded modules.
7. `module_init` listed function.

rmmod command

1. `rmmod` calls `delete_module()` which hints the kernel that a module is to be removed.
    a. Control is transferred to the kernel.
2. Kernel executes `sys_delete_module()`
    a. Checks the permissions of the one requesting.
    b. Checks if any other loaded module needs the current module.
    c. Checks if the module if actually loaded!
    d. Executes the function provided in `module_exit`
    e. `free_module`
        i. Removes references and kernel object references.

ii. Performs any other cleanup.
iii. Unloads the module.
iv. Changes the state in list.
v. Removes it from the list and frees the memory.
