Linux Network Device Drivers: an Overview

T.A. Gonsalves, IIT Mandi

21st April 2020

Table of Contents

1 Device Drivers

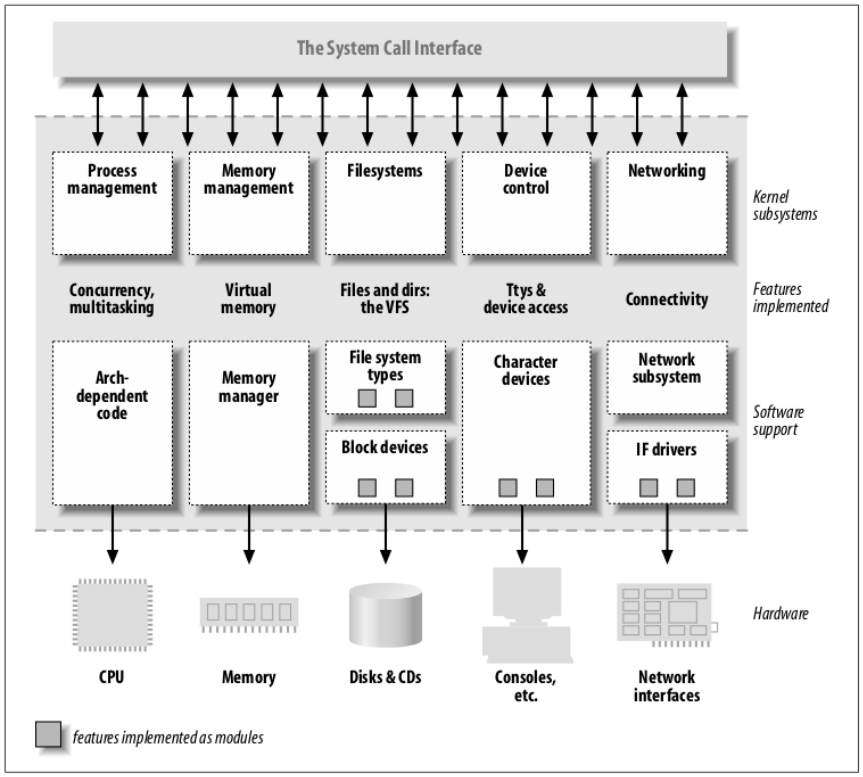

Figure 1: Structure of the Linux Kernel (Fig. 1-1 from [2])

A device driver is code in the OS kernel that directly interacts with hardware devices (Fig. 1). Device drivers are an important part of the kernel. They interact directly with the hardware so are complex, they are a potential security risk, and they must be efficient. In Linux, device drivers account for about 75-80% of the kernel code (Ch. 2 in [1]).

Linux has 3 classes of devices (Ch. 7 in [1], Ch. 1 in [2]):

1. **Character devices**: these are devices that receive/send data serially as a stream of bytes. E.g. keyboard, display, printer, mouse.
2. **Block devices**: these deal with chunks of data that can be accessed randomly. E.g. disks which consist of sectors, usually of size 512 bytes, that can be accessed in arbitrary order.
3. **Network devices**: these are similar to character devices as they deal with data serially. However, they have different characteristics such as highly variable delay, variable error rate, complex network protocols, etc. Data is often read/written in units of one variable-length packet. With most character and block devices, data in read only when

an application process opens the device. A network device receives packets asynchronously to any process. For these reasons, Linux treats network devices as a separate class. E.g. Ethernet, dial-up modem, Wifi.

Every device driver implements a function call interface that is used by other parts of the kernel to read/write from/to the device. The functions include `init()`, `open()`, `close()`, `read()`, `write()`, etc. Most hardware devices operate at speeds much slower than the CPU and RAM. Hence, I/O operations with a device are normally done asynchronously. The device driver issues a command to the hardware device adapter and then continues with other processes. When the I/O command completes, the adaptor interrupts the device driver.

2 Linux Network Stack

The Linux kernel includes the transport layer (TCP and UDP), the network layer (IP), and device drivers for various MAC layers (Ethernet, Wifi, loopback, etc) (Fig. 2).[1]

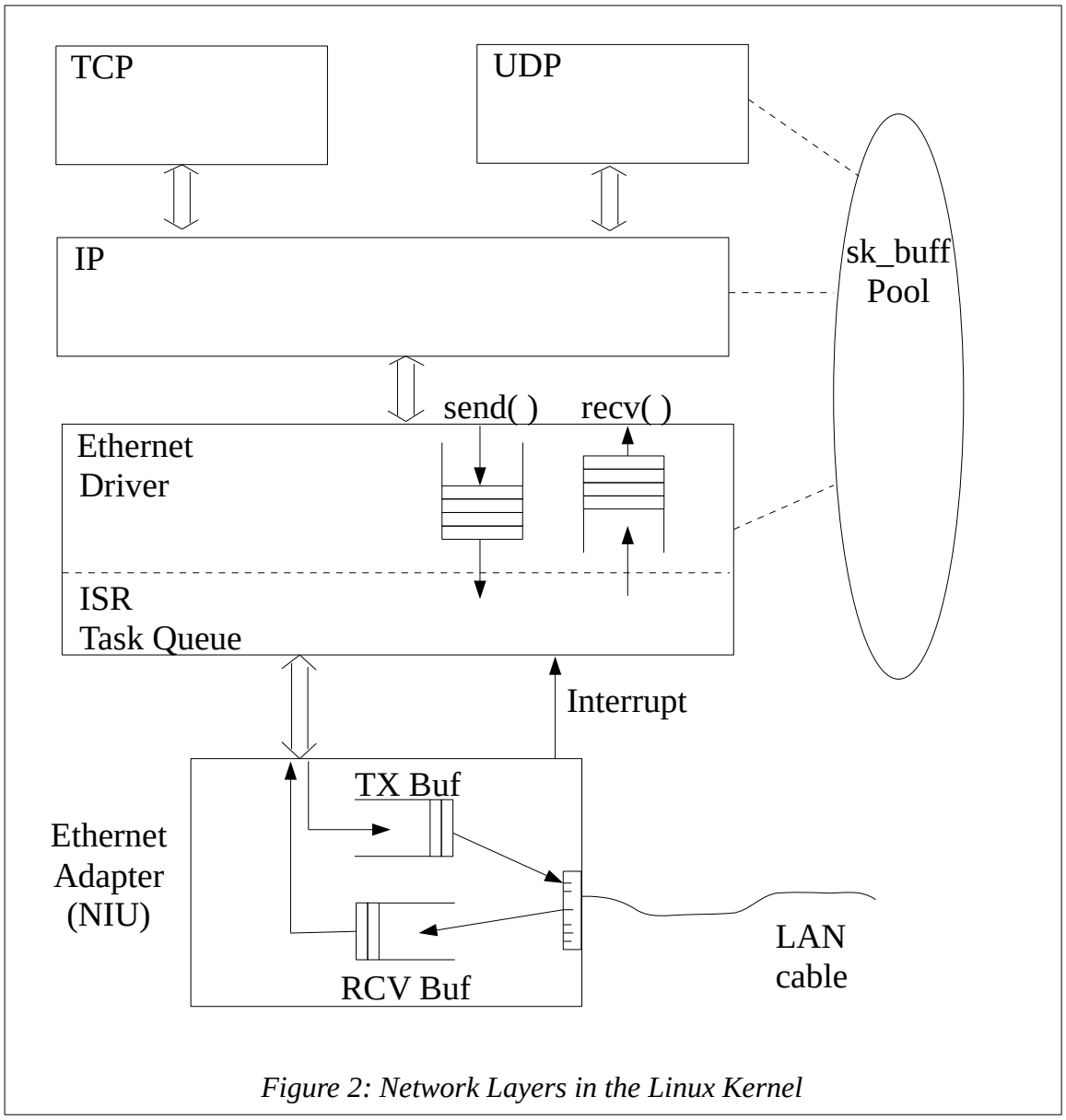

Figure 2: Network Layers in the Linux Kernel

1 The Linux networking design accommodates assorted transport and network protocols. As TCP, UDP and IP are the most widely used, we refer only to these here.

When a process, such as a web browser, wants to send or receive data over the network, it opens a *socket*. The socket descriptor is a kernel data structure that contains relevant information about the network connection and the network device.

The layered structure in Fig. 2 implies that when an application, say, a webserver has a page to be sent to a browser in response to a HTTP request, the data is copied to TCP. From there, the data is copied to the IP layer, once more copied to the Ethernet driver and finally copied to the Ethernet adapter from where it is transmitted. All this data copying is expensive. For efficiency, the network stack is designed to avoid copying data. The TCP layer allocates a `struct sk_buff` that holds the data to be sent plus assorted control information. The `sk_buff` is passed by reference to the IP and Ethernet layers. By judicious violation of the layering principle, the network stack in Linux can operate at data rates of Gb/s.

3 Network Device Drivers[2]

A network driver provides a function call interface to the IP layer (Ch. 17 in [2]). Some of the important functions are:

`open()`: called by `ifconfig <dev> up`

`hard_start_xmit()`: hand a packet to the NIU to start transmission

`stop()`: called by `ifconfig <dev> down`

The upper half of the driver runs when it is called by the IP layer on behalf of a process such as a browser. It executes in the context of the browser process. The context includes the memory, open file handles, environment variables, etc.

The interrupt service routines (ISR) in the lower half of the driver run when in interrupt context when an interrupt occurs. An ISR must be short. To cater to lengthy computations involved in some network activities, the ISR can place a request for the longer processing in a *task queue*. Tasks in the queue are executed in process context at a later time.

The network stack makes use of **hook functions**. A hook function is an *up-call* mechanism by which a lower layer invokes the next upper layer. This is used when the lower layer encounters an asynchronous event that needs to be processed by the upper layer, even though the upper layer has not made a call to the lower layer. Hooks are used for reception of packets which are asynchronous events. Note that up-calls are a violation of the strict layering principle in which the upper layer always calls the lower layer.

3.1 Packet Transmission

We consider the steps involved in transmission of a packet. E.g. `ping` uses UDP to transmit a single `ECHO_REQUEST` packet.[3]

1. `ping` uses the `sendto()` system call. The data is copied from the user address space of ping to an `sk_buff` in the kernel address space.
2. UDP adds its header fields and calls a `send()` function of the IP layer, passing a pointer to the `sk_buff`
3. IP fills its header in the packet and calls `hard_xmit_start()` in the Ethernet driver.

2 The functions mentioned in this document are only approximate. To thoroughly understand the network stack or specific drivers, refer to the source code and developer documentation for the current kernel.

3 For ease of understanding, we present only a simplified view of the common case. Refer to [2] and the source code for full details and optimisations.

4. The Ethernet driver copies the packet from the `sk_buff` to the TX buffer in the NIU. It issues the transmit command to the NIU and then blocks on the end-of-transmit event.
5. After completion of transmission, the NIU issues a hardware interrupt to the Ethernet driver.
6. The ISR wakes up the blocked `hard_xmit_start()`. The calls return back up the layers to the `ping` process.

Steps 1-4 run in the context of the `ping` process. Hence, the `ping` process is blocked in step 4 and cannot proceed until woken up in step 6. Note also that the data of the packet, which could be up to 1,500 B, is copied only twice: once from user space to kernel space in step 1, and then from kernel space to network adapter buffer in step 4.

3.2 Packet Reception

After successful transmission, ping needs to receive the ECHO_RESPONSE packet. Since packets are received asynchronously even when no process expects them, the logic is more complex than transmission. The steps involved are:

1. `ping` uses the `recvfrom()` system call.
2. UDP calls the `receive()` function of IP
3. On initialisation, IP registers `ip_rcv()` as a *hook function* with the Ethernet driver. The IP `receive()` function checks for a packet in its receive queue. If one is available, it is returned. Otherwise, it blocks waiting for a packet to be received.
4. When the NIU receives a packet, it interrupts the Ethernet driver.
5. The ISR copies the packet from the hardware buffer to an `sk_buff` in the receive queue of the driver. The driver calls hook function `ip_rcv()` passing the pointer to `sk_buff`.
6. `ip_rcv()` adds the packet to the IP receive queue and unblocks the waiting `ping` process. The call return back up the layers to the `ping` process. Before it completes, the `recvfrom()` system call copies the packet from the `sk_buff` to the user space buffer and frees the `sk_buff`.

When the Ethernet driver is initialised (i.e. it goes to the `up` state, say by `ifconfig eth0 up`), it sets up the receive queue and issues a command to the NIU to enable packet reception.

4 MACVLAN

A MACVLAN is a pseudo-device [3]. The network driver for a MACVLAN is inserted between the IP layer and a physical network driver, such as the Ethernet driver in Fig. 1. To the IP layer, it looks like any other network driver. It does not directly control any hardware. Instead, it treats the Ethernet driver as a virtual network adapter. Several MACVLAN drivers can be inserted side-by-side above the same Ethernet driver, thus creating several pseudo-interfaces. This is useful for creating several execution containers such as Dockers on a single Linux system [4].

5 Traffic Classifier, `tc`

The Linux `tc` module makes use of the hook mechanisms in the Linux kernel. It registers itself as a hook for packets in each interface for which it ios configured. When a packet is

passed to its hook function, `tc` applies the queueing discipline, delay, loss, and bandwidth limit configured for that interface to the packet before passing it back to the network stack in the Linux kernel.

References

[1] M. Beck, et al., *Linux Kernel Programming, 3rd ed.*, Pearson, 2002.

[2] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd ed.*, O'Reilly, 2005.

[3] Hangbin Liu, "Introduction to Linux interfaces for virtual networking", https://developers.redhat.com/blog/2018/10/22/introduction-to-linux-interfaces- for-virtual-networking/ macvlan, Accessed: 7 April, 2020

[4] Anoushka Banerjee & Shaifu Gupta, "*CS549: Quick Guide to Virtual Networking"*, Moodle, IIT Mandi, April 2020.