LDD3, Ch1

ECEN 427

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

Why do we care about writing device drivers?

"There are a number of reasons to be interested in the writing of Linux device drivers. The rate at which new hardware becomes available (and obsolete!) alone guarantees that driver writers will be busy for the foreseeable future. Individuals may need to know about drivers in order to gain access to a particular device that is of interest to them. Hardware vendors, by making a Linux driver available for their products, can add the large and growing Linux user base to their potential markets. And the open source nature of the Linux system means that if the driver writer wishes, the source to a driver can be quickly disseminated to millions of users."

(page 1)

Mechanism vs Policy

As a programmer, you are able to make your own choices about your driver, and choose an acceptable **trade-off between the programming time required and the flexibility of the result**.

Though it may appear strange to say that a driver is “flexible”, we like this word because it emphasizes that the role of a device driver is **providing mechanism, not policy**.

The distinction between mechanism and policy is one of the best ideas behind the Unix design. Most programming problems can indeed be split into two parts:

- “what capabilities are to be provided” (the ***mechanism***), and
- “how those capabilities can be used” (the ***policy***).

If the two issues are addressed by different parts of the program, or even by different programs altogether, the software package is much easier to develop and to adapt to particular needs. (Pages 2-3)

Mechanism vs Policy

Mechanism :

- What capabilities will be provided
- What can be done with the hardware

Policy

- How the capabilities will be used
- How to use the hardware in applications

Example: Disk Driver

	Mechanism	Policy
1. Reads/Writes data		
2. Error Handling		
3. Bad Sector Handling		
4. Data Caching		
5. Power Management		

Drivers Provide Mechanism, not Policy

When writing drivers, a programmer should pay particular attention to this fundamental concept: write kernel code to access the hardware, but don't force particular policies on the user, since different users have different needs.

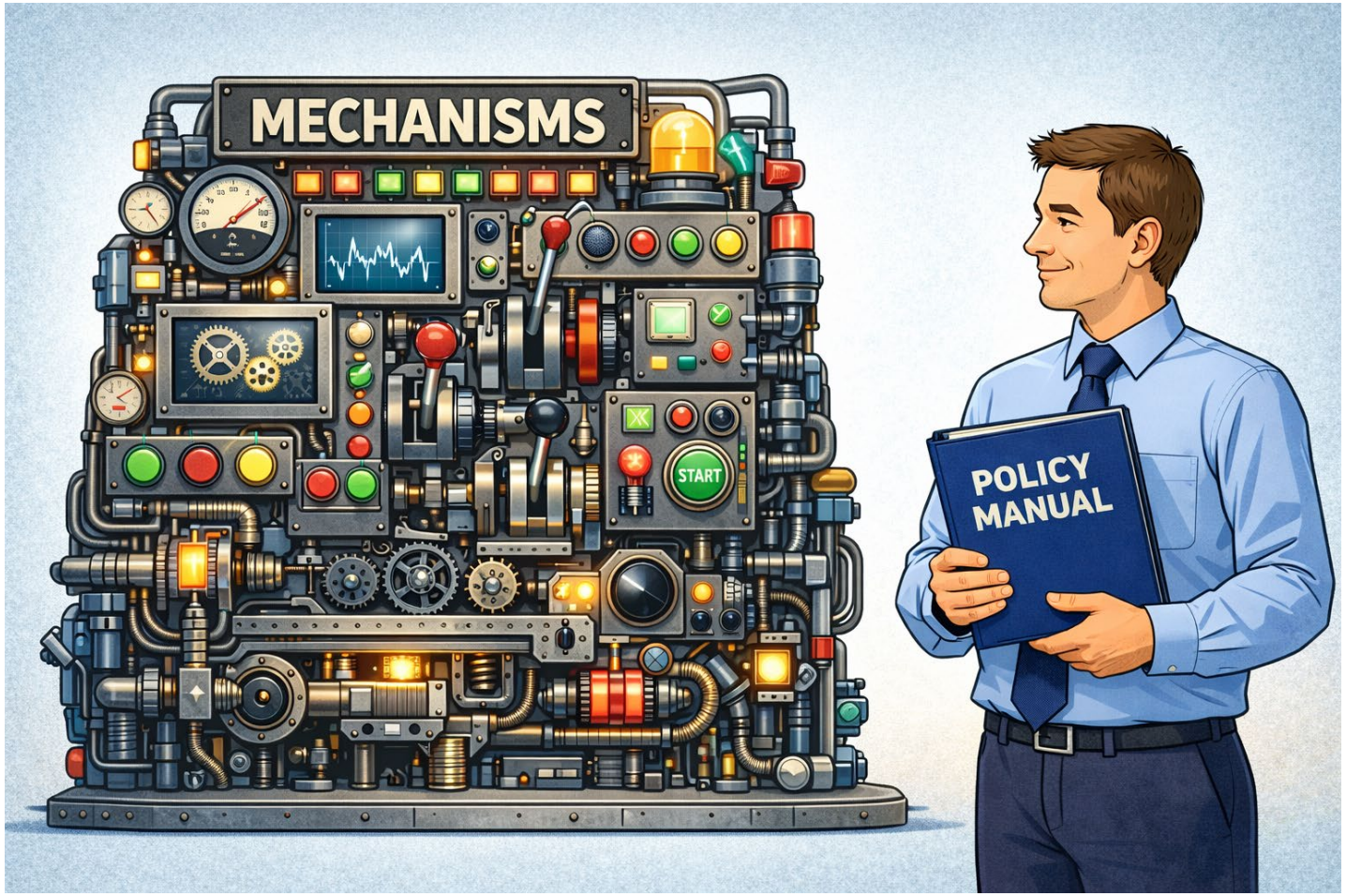
The driver should deal with making the hardware available, leaving all the issues about how to use the hardware to the applications. A driver, then, is flexible if it offers access to the hardware capabilities without adding constraints.

(Page 3)

Lab 2 – Intc Driver

Did our *intc* driver do a good job of providing mechanism, not policy?

What is a policy decision that you might be tempted to add?



Policy-free drivers have a number of typical characteristics. These include support for both synchronous and asynchronous operation, the ability to be opened multiple times, the ability to exploit the full capabilities of the hardware, and the lack of software layers to “simplify things” or provide policy-related operations. Drivers of this sort not only work better for their end users, but also turn out to be easier to write and maintain as well.

…The actual driver design should be a balance between many different considerations. For instance, a single device may be used concurrently by different programs, and the driver programmer has complete freedom to determine how to handle concurrency…One major consideration is the trade-off between the desire to present the user with as many options as possible and the time you have to write the driver, as well as the need to keep things simple so that errors don’t creep in.

(Page 3)

- What is synchronous vs asynchronous operation?
- What is concurrency? What sort of issues might arise when trying to make a concurrent driver?

Synchronous

The operations cause the process to “block” and wait for them to complete.

Common when reading a file from disk

read() / write() may take some time to return

```c
int fd;
ssize_t bytes_read;
char buffer[100];

// Open a file or device in blocking mode
fd = open("/dev/some_device", O_RDONLY);
if (fd < 0) {
    perror("Failed to open the device");
    return EXIT_FAILURE;
}

// Read data (blocking)
bytes_read = read(fd, buffer, sizeof(buffer));
if (bytes_read < 0) {
    // Error occurred during read
    perror("Failed to read from the device");
} else if (bytes_read == 0) {
    // End of file reached or no data to read
    printf("End of file reached or no data to read.\n");
} else {
    // Data was read
    printf("Read %ld bytes: %s\n", bytes_read, buffer);
}

close(fd);
```

Asynchronous

The process doesn't block, but the operations aren't guaranteed to have completed.

Common when interacting with network sockets/devices.

read() / write() will return quickly, but:

- May not have read/written any data
- May have read/written a portion of the data

```c
// Open a file or device in non-blocking mode
fd = open("/dev/some_device", O_RDONLY | O_NONBLOCK);
if (fd < 0) {
    perror("Failed to open the device");
    return EXIT_FAILURE;
}

while (1) {
    // Attempt to read data
    bytes_read = read(fd, buffer, sizeof(buffer));

    // Check if read was successful
    if (bytes_read > 0) {
        // Data was read
        printf("Read %ld bytes: %s\n", bytes_read, buffer);
        break;
    } else if (bytes_read < 0) {
        // Check if the read operation would block
        if (errno == EAGAIN) {
            // No data available right now, try again later
            printf("No data available, retrying...\n");
            usleep(100000); // Sleep for 100 milliseconds before r
            continue;
        } else {
            // An actual error occurred
            perror("Failed to read from the device");
            break;
```

Concurrency

- PYNQ uart example (opening /dev/ttyUSBX multiple times)

The Kernel

In a Unix system, several concurrent processes attend to different tasks. Each process asks for system resources, be it computing power, memory, network connectivity, or some other resource. The kernel is the big chunk of executable code in charge of handling all such requests. Although the distinction between the different kernel tasks isn't always clearly marked, the kernel's role can be split (as shown in Figure 1-1) into the following parts:

(Page 4)

The Kernel

Process management

The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output). Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel. In addition, the scheduler, which controls how processes share the CPU, is part of process management. More generally, the kernel's process management activity implements the abstraction of several processes on top of a single CPU or a few of them.

Memory management

The computer's memory is a major resource, and the policy used to deal with it is a critical one for system performance. The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple malloc/free pair to much more complex functionalities.

The Kernel

Filesystems

Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file. The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. In addition, Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium. For example, disks may be formatted with the Linux-standard ext3 filesystem, the commonly used FAT filesystem or several others.

Device control

Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a device driver. The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape drive. This aspect of the kernel's functions is our primary interest in this book.

The Kernel

Networking

Networking must be managed by the operating system, because most network operations are not specific to a process: incoming packets are asynchronous events. The packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity. Additionally, all the routing and address resolution issues are implemented within the kernel.

- What are the 5 responsibilities of the kernel?

Loadable Modules

One of the good features of Linux is the ability to **extend at runtime** the set of features offered by the kernel.

Each piece of code that can be added to the kernel at runtime is called a **loadable module**.

The Linux kernel offers support for quite a few different types (or classes) of modules, **including, but not limited to, device drivers**.

Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the **insmod** program and can be unlinked by the **rmmod** program.

(Page 5)

Types of Device Drivers

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

3 Classes of Devices

1. Character devices

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls.

The text console (/dev/console) and the serial ports (/dev/ttyS0 and friends) are examples of char devices, as they are well represented by the stream abstraction.

Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0.

The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially.

There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using mmap or lseek.

2. **Block devices**

Like char devices, block devices are accessed by filesystem nodes in the /dev directory.

A block device is a **device (e.g., a disk) that can host a filesystem**.

In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length. Linux, instead, allows the application to read and write a block device like a char device—it permits the transfer of any number of bytes at a time.

As a result, block and char devices **differ only in the way data is managed internally by the kernel**, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.

3. **Network interfaces**

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are, usually, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets.

Not being a stream-oriented device, a network interface isn't easily mapped to a node in the filesystem, as /dev/tty1 is. The Unix way to provide access to interfaces is still by assigning a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the filesystem. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of read and write, the kernel calls functions related to packet transmission.