IO Devices

At some point, all persistent data is eventually stored on an I/O device. There are many types of devices. Devices can connect to the CPU using a variety of interfaces. OS code chooses among several strategies when interacting with devices. The goal of this chapter is to understand the landscape of choices.

- What choices do we make when a device is slow? (e.g., how do we connect to it, how do we communicate with it?n)
- What choices do we make when a device is fast? (e.g., how do we connect to it, how do we communicate with it?n)
- How does the Linux software storage stack's design accommodate such a wide variety of devices?

Connections

Different elements of the memory hierarchy are connected to the CPU using different physical interfaces. The physical interfaces often support different software interfaces (commands) for communicating with the devices and/or have different maximum throughputs.

- Main memory (RAM) is connected to the CPU via a memory bus.
- Up to a few fast devices (e.g., GPU) are connected to the CPU via an I/O bus (e.g., Peripheral Component Interconnect Express or PCIe).
- NVMe is a relatively new I/O interface that was designed specifically for high-end non-volatile memory devices (i.e., expensive low-latency SSDs). Since NVMe is relatively new, we will not go into much detail. The important takeaways are that NVMe devices connect directly to the PCIe bus, and NVMe is optimized specifically for NVM storage devices.
- Then there are potentially many slower devices connected to the peripheral bus. In terms of storage devices, hard drives are often connected via SCSI or SATA interfaces to the peripheral bus.
- **Questions**

1. Why isn't everything just connected to the fastest bus? What are the practical/design constraints that determine how many and which devices are connected in which ways?

Devices

Each device exports some hardware interface so that the OS may communicate with it. Devices also have their own internal structures (some devices have their own CPUs or memory inside them, some have moving parts, etc.). Firmware is software that runs *inside* a device to implement the storage abstractions over its internal structure. Firmware details are often hidden from devices users (e.g., proprietary code) and firmware is often fixed (e.g., once you buy a device you can't update its firmware).

- **Questions**

1. What are some example tasks that firmware on a storage device might need to perform?
2. What types of devices might have complex firmware?
3. In comparison, what types of devices might be "dumb"?

Interacting with Devices

- **Polling** is a common technique for waiting on I/O. When polling, the OS repeatedly checks the status of a device, waiting for a particular state or action.
- **Interrupts** allow the CPU to switch tasks while waiting for a slow I/O device to complete an operation. Some OS code issues an I/O request, then the OS switches to another process while the I/O device does some work. When the I/O device is completes the request, the I/O device issues a hardware interrupt, and the CPU stops its current task and jumps to the designated interrupt handler (i.e. code registered by the OS to finish any work necessary to complete/verify the I/O request). Then it can wakes up the original task to resume working.
- Coalescing is when the device waits a bit before issuing a hardware interrupt so that a single interrupt may be used to "complete" multiple requests. In this way, the OS is interrupted a fewer number of times to complete the same amount of work.
- **Questions**

1. What is a major disadvantage of polling?
2. Interrupts solve one problem with polling: the CPU can do other work while waiting for an I/O to complete, rather than *busy-waiting* for the device. When might polling actually be more efficient than using interrupts?
3. When would you want to use interrupts, and when would you want to use polling? Is there a way to get the best of both worlds?

Device Drivers

To keep the OS as general as possible, device drivers occupy the lowest level of the Linux software storage stack. There are advantages and disadvantages to an OS designed in this way.

- One *advantage* is that relatively little code needs to be changed to support a new device (or a new class of devices).
- The kernel is comprised of generic layers with software *hooks* (a well-defined set of functions called at specific code points that let designers insert/change functionality) that the device driver overrides with its own implementation. The device driver then registers its specific functions with the OS.
- When a user compiles their kernel, they only need to include the device drivers for the specific devices that they own (or wish to support). This lessens software bloat.
- One *disadvantage* is that device drivers, since they are isolated code that is very specific to a piece of hardware, are often sources of bugs. Since device drivers are part of the OS kernel, they run at the highest privilege levels. A device driver bug can compromise your entire system.
- **Questions**

1. If you look at the Linux software stack diagram (OSTEP figure 36.3, page 418), file systems sit above the generic block layer, and device drivers sit below it. 1. What is one advantage of the OS's strict layering? 2. What is one disadvantage of the OS's strict layering?