Chapter 10

Input / output

I wouldn’t trust any critical program created by someone who has never written an interrupt service routine.

Chandu Thekkath

Definition 10.1 (I/O subsystem). Every OS has an I/O subsystem, which handles all interaction between the machine and the outside worlds. The I/O subsystem abstracts individual hardware devices to present a more or less uniform interface, provides a way to name I/O devices, schedules I/O operations and integrates them with the rest of the system, and contains the low-level code to interface with individual hardware devices.

Remarks:

• The I/O subsystem usually divides neatly into device drivers and generic functionality.

• In some OSes, like Linux, the I/O subsystem evolved organically over time and the boundary between it and the rest of the OS is sometimes a little blurred. In others, like MacOS, it’s clearly delineated (the I/O subsystem in MacOS is called DeviceKit).

• Not all the I/O subsystem is necessarily in the kernel. It is possible to run most of it (including the device drivers) in user space, as in a microkernel.

We’ll start with recapping the basic concepts of I/O.

Definition 10.2 (Device). To an OS programmer, a device is a piece of hardware visible from software. It typically occupies some location on a bus or I/O interconnect, and exposes a set of hardware registers which are either memory mapped or (in the case of x86 machines) in I/O space. A device is also usually a source of interrupts, and may initiate Direct Memory Access (DMA) transfers.
Remarks:

- Hopefully, the above definition should not be new to you (if you have taken the ETH Systems Programming course).

- This definition doesn’t say a lot about what a device is physically. In practice this is difficult: in the past, a device really was just that: a piece of metal with connectors on it. Today, it might be some bundle of functionality somewhere on a chip. Anything that isn’t a processor, RAM, or interconnect is often a device.

The software component of the OS that talks to a particular device is called the driver.

**Definition 10.3 (Device driver).** The **device driver** for a particular device is the software in the OS which is understands the specific register and descriptor formats, interrupt models, and internal state machines of a given device and abstracts this to the rest of the OS.

Remarks:

- The driver can be thought of as sitting between hardware and rest of the OS.

- A given OS has a **driver model** which defines the kinds of abstractions a device driver will provide.

- In Unix, drivers run in the kernel for the most part, but there is not necessary reason for this.

- The concept of a driver long predates object-oriented program. This can lead to some confusion, since the term can equally refer to the **body of code** which is written to manage a particular piece of hardware (or a group of similar models of hardware device), and the runtime **software object** which manages a single device.

In the rest of this chapter, we’re going to talk about “devices” or “the device” at lot, when we actually mean “the representation inside the OS that corresponds to a hardware device”. To the OS, what matters is what it thinks a device is, rather than what the device physically is. Indeed, we’ll see pseudo-devices which don’t physically exist at all.

### 10.1 Devices and data transfer

The whole of this section should be familiar to you from the “Systems Programming and Computer Architecture” course.

**Definition 10.4 (Device registers).** A **device register** is a physical address location which is used for communicating with a device using reads and writes.
Remarks:

- A hardware register is not memory, but it sits in the physical address space. There is no guarantee that reading from a device register will return the same value that was last written to it.

- Reading a device register can return device input data, or status information about the device.

- Writing a device register can send it data to be output, or configure the device.

- In addition to the above, however, both read and writing to the register can trigger actions in the device hardware.

**Definition 10.5 (Programmed I/O).** *Programmed I/O consists of causing input/output to occur by writing data values to hardware registers from software (output), or reading values from hardware registers into CPU registers in software (input).*

**Algorithm 10.6 Programmed I/O input**

```plaintext
1: inputs
2: l: number of words to read from input
3: d: buffer of size l
4: d ← empty buffer
5: while length(d) < l do
6: repeat
7: s ← read from status register
8: until s indicates data ready
9: w ← read from data register
10: d.append(w)
11: end while
12: return
```

Remarks:

- You can construct the corresponding output algorithm by symmetry arguments.

- Programmed I/O is the simplest way for software on a core to communicate with a device.

- It is fully synchronous: the CPU has to write to the register to cause output to happen there and then, and must read from the register for input. There is no buffering.

- It is also polled: there is no way for a device to signal that it has data ready, or it is ready to send data.

**Definition 10.7 (Interrupt).** *An interrupt is a signal from a device to a CPU which causes the latter to take an exception and execute an interrupt service routine (ISR), also known as an interrupt handler.*
Algorithm 10.8 Interrupt-driven I/O cycle

1. Initiating (software)
   1: Process A performs a blocking I/O operation
   2: OS initiates an I/O operation with the device
   3: Scheduler blocks Process A, runs another process

2. Processing (hardware)
   4: Device performs the I/O operation
   5: Raises device interrupt when complete, or an error occurs

3. Termination (software)
   6: Currently running process is interrupted
   7: Interrupt handler runs, processes any input data
   8: Scheduler makes Process A runnable again

4. Resume (software)
   9: Process A restarts execution.

Remarks:

- Interrupts solve the polling problem, and so decouple the software and hardware to some extent.

- The CPU still has to copy output data to the device when it initiates the request, and/or copy input data from the device when the operation completes.

- To further decouple the two, we need to allow the device to do the copy itself.

Definition 10.9 (Direct Memory Access). Using Direct Memory Access or DMA, a device can be given a pointer to buffers in main memory and transfer data to and from those buffers without further involvement from the CPU.

Remarks:

- DMA in its simple form mean that the processor’s involvement at the start and end of an I/O operation is minimized, since the data itself does not need to be copied.

- DMA is typically performed by the device, though in older hardware a separate DMA engine was provided which essentially performed programmed I/O on a device while the CPU did something else.

- DMA saves bus bandwidth, since the data doesn’t need to be copied to through the CPU’s registers.

- Typically only a single interrupt is needed, so signal the end of a complete DMA copy.
• The real value of DMA comes when the device uses it both to transfer data to and from main memory, but also to read new I/O operations from a list in memory, and write completion information back to main memory. This is the concept of “descriptor rings” that you saw last year, and further decouples CPU and device.

• DMA is, for the most part, physical (not virtual) access to memory (but see IOMMUs, below). This means that a virtual address in a user program or the kernel (if the kernel runs in virtual address space, which is common) must be translated to a physical address before being handed to the device.

• DMA transfers to and from main memory may or may not be coherent with processor caches. If not, device drivers must take great care to flush (clean) and invalidate processor caches before and after DMA transactions.

10.2 Dealing with asynchrony

Device drivers have to deal with the fundamentally asynchronous nature of I/O: the system must respond to unexpected I/O events, or to events which it knows are going to happen, but not when. Input data arrives without warning, and an input operation takes an unknown period of time. A busy device becomes capable of accepting more output data at an unspecified time in the future, and it’s not clear when an output operation is going to complete.

Definition 10.10 (First-level interrupt service routine). The First-level Interrupt Service Routine or FLISR is the code that executes immediately as a result of the interrupt.

Remarks:

• An FLISR runs regardless of what else is happening in the kernel (unless interrupts are disabled).

• As a result, it can’t change much since the normal kernel invariants might not hold: it can’t allocate memory from the kernel heap (if the kernel has one), it can’t acquire or release locks, and it should not take too long to finish.

Since I/O is for the most part interrupt-driven, but data is transferred to and from processes which perform explicit operations to send and receive it. Consequently, data must be buffered between the process and the interrupt handler, and the two must somehow rendezvous to exchange data.

There are three canonical solutions to this problem: deferred procedure calls, driver threads, and non-blocking kernels.

Definition 10.11. Deferred procedure calls. A Deferred procedure call, sometimes known as a 2nd-level interrupt handler, a soft interrupt handler, or a slow interrupt handler, is a program closure created by the 1st-level interrupt handler. It is run later (hence the name) by any convenient process, typically just before the kernel is exited.
Remarks:

- DPCs are extremely efficient, and a common solution to the rendezvous problem (e.g. in VMS, Windows, BSD Unix, etc.).

- The closure itself is small (a few words), and can be statically allocated (since you only ever need one outstanding per-device) so doesn’t need to be dynamically allocated inside the FLISR.

- DPCs do need to be queued up for execution once the interrupt context has finished, so we need a lock-free queue to hold them.

**Definition 10.12** (Driver thread). A *driver thread*, sometimes called an *interrupt handler thread*, serves as an intermediary between interrupt service routines and processes. The thread starts blocked waiting for a signal either from the user process or the ISR. When an interrupt occurs or a user process issues a request, the thread is unblocked (this operation can be done inside an ISR) and it performs whatever I/O processing is necessary before going back to sleep.

Remarks:

- Driver threads are heavyweight: even if they only run in the kernel, the still require a stack and a (same address space) context switch to and from them to perform any I/O requests. They therefore take time (cycles) and space (memory). The latter also translates into time (more cache misses).

- They are conceptually simple, and can be understood more intuitively than DPCs. Consequently, if many kernel and/or driver developers are of intermediate skill and ability (such as with Linux), this may be the preferred option from an engineering perspective.

The third alternative, used in microkernels and exokernels, is to have the FLISR convert the interrupt into a message to be sent to the driver process. This is conceptually similar to a DPC, but is even simpler: it simply directs the process to look at the device. However, it does require the FLISR to synthesize an IPC message, which might be expensive. In non-preemptive kernels which only process exceptions serially, however, this is not a problem, since the kernel does not need locks.

**Definition 10.13** (Bottom-half handler). The part of a device driver code which executes either in the interrupt context or as a result of the interrupt (like a DPC) is the *bottom half*.

**Definition 10.14** (Top-half handler). The part of a device driver which is called “from above”, i.e. from user or OS processes, is the *top half*.

Remarks:

- Note that the top half can be scheduled, and so the time it uses can be accounted to some process, whereas the bottom half either isn’t scheduled (the FLISR) or is run by whatever is handy (the driver thread or the current process).
• Note that this is not the Linux terminology, but it is the one used by pretty much all other OSes (including other UNIX systems, all of which predate Linux). In Linux, for unknown reasons, the “top half” is the FLISR, while the DPC or driver thread is the “bottom half”.

10.3 Device models

Definition 10.15 (Device model). The device model of an OS is the set of key abstractions that define how devices are represented to the rest of the system by their individual drivers.

Remarks:
• The device model fulfills the role of device abstraction in the system. It includes the basic API to a device driver, but goes beyond this: it encompasses how devices are named throughout the system, and how their interconnections are represented as well. It also specifies the relationship between physical devices and device drivers.

As a rough example, we’ll discuss the UNIX device model here. UNIX divides devices into three classes, character, block, and network devices.

Definition 10.16 (Character devices). A character device in UNIX is used for “unstructured I/O”, and presents a byte-stream interface with no block boundaries.

Remarks:
• Character devices are accessed by single byte or short string get/put operations.
• In practice, buffering implemented by libraries
• Examples include keyboards, serial lines (UARTS), mice, USB-controlled missile launchers, etc.

Definition 10.17 (Block devices). A block device in UNIX is intended for “structured I/O”, and deals with “blocks” of data at a time - for example, disk blocks.

Remarks:
• Block devices often resemble files more than simply in being named by the filing system: storage devices like disks or SSDs are seekable and mappable like files. Access to them often uses the UNIX buffer cache. We’ll see more of block devices as a basis for file systems in Chapter 19
• In practice, the distinction between character and block devices is somewhat arbitrary.

Definition 10.18 (Network devices). A network device in UNIX corresponds to a (real or virtual) network interface adapter. It is accessed through a rather different API to character and block devices.
Remarks:

- Arguably, network interfaces don’t fit nicely into either of these models (they came along much later in the development of Unix), and indeed NICs per se in Unix are generally not abstracted as device. Instead, streams of packets can be send and received through sockets or special character devices like `/dev/tun`.
- We’ll look at network devices in more detail later in Chapter ??.

**Definition 10.19** (Pseudo-devices). A *pseudo-device* is a software service provided by the OS kernel which it is convenient to abstract as a device, even though it does not correspond a physical piece of hardware.

**Example 10.20.** Unix systems have a variety of pseudo-devices, such as:

- `/dev/mem` A character device corresponding to the entire main memory of the machine
- `/dev/random` Generates random numbers when read from.
- `/dev/null` Anything written is discarded, read always returns end-of-file.
- `/dev/zero` Always reads as zeroes.
- `/dev/loop` Block device for making a file look like an entire file system.

How are devices identified inside the kernel?

**Example 10.21** (Traditional Unix device configuration). In older Unix systems, devices were named inside the kernel by a pair of bytes: the major and minor device numbers.

The **Major device number** identified the class of device (e.g., disk, CD-ROM, keyboard).

The **Minor device number** identified a specific device within a class.

In addition, a third “bit” determined whether the device was a character or block device.

As a naming scheme, this was fine when there were very few different devices and device types, but things have changed a lot since then. Not only are there a large number of different models of device that might be plugged into a computer (literally tens of millions), they can also be connected in a variety of different ways.

**Definition 10.22** (Device discovery). Most modern OSes (with the exception of some small embedded systems) perform **device discovery**: the process of finding and enumerating all hardware devices in the system and storing metadata about them (where and what they are) in some kind of queryable data store.

**Example 10.23** (Linux *sysfs*). Modern versions of Linux store this information in the kernel, but expose it to curious user programs via a “pseudo file system” (i.e. something that looks like a file system but isn’t) called *sysfs*.
CHAPTER 10. INPUT / OUTPUT

While `sysfs` is a hierarchical directory structure, devices appear in it multiple times, organized by type, connection, etc.

`sysfs` is a very strange way to build a database (which is what it essentially is), but does fit in the Unix philosophy of “everything is a file”.

When devices are plugged or unplugged, the contents of `sysfs` change. User programs can get notification of these changes by listening on a special socket.

Linux device drivers are also more dynamic than in the old days: they are typically implemented as loadable “kernel modules”, to be loaded on demand when the kernel discovers a device that needs a particular driver. The initial list of drivers is given at boot time, but a daemon can load more on demand if required.

10.4 Device configuration

In addition to simply discovering a device, and finding out how to access it, the OS often has to configure the device and other devices in order to make it work.

Example 10.24 (USB hotplug). When a USB device (such as a USB thumb drive) is plugged in, a number of different devices are involved, at the very least:

- The USB drive itself

- The USB host bus adapter or HBA, which interfaces the USB device network to the rest of the computer.

- The USB hub that the device was plugged into. This can easily not be a physically separate hub, but one integrated onto the motherboard or built into another device (such as the HBA).

Broadly speaking, when the device is plugged in, the HBA notifies the OS that something has been plugged in. The HBA driver then talks to the HBA to enumerate the devices attached to it, including the hubs – USB is organized approximately as a tree of devices where the non-leaf nodes are hubs. The HBA adapter then has to assign new bus and device identifiers to anything that has changed and reconfigure the HBA and switches. It also discovers the new device by finding out what it is – USB devices, like PCI devices, describe themselves with a 4-byte code.

After this, the OS can start the appropriate driver for the device, and tell it where in the USB hierarchy the new device is.

Sometimes, more complex resource allocation is required.

Example 10.25 (PCI configuration). Configuration of PCI (or, today, PCI Express) devices at a high level looks very similar to that of USB, except that the “PCIe Root Complex” is used instead of the HBA, and “PCIe bridges” are used instead of USB hubs.

However, unlike in USB, where a driver for something like a USB stick talks to the physical device by sending it messages via the HBA, in PCIe all devices are memory mapped. The regions of memory they need to be mapped have to be allocated by the OS from the physical address space of the machine. This is a complex process, which continues to cause problems for OS developers.
Moreover, almost all devices require interrupt routing.

**Definition 10.26 (Interrupt routing).** **Interrupt routing** is the process of configuring interrupt controllers in the system to ensure that when a device raises an interrupt, it is delivered to the correct vector on the correct core (or group of cores).

**Remarks:**

- Interrupt routing is one of the things that is getting much more complex over time. It is not unusual for a modern PC to have 4 or 5 interrupt controllers between a device and the CPUs, and even phones can have 3 or more.
- This is also a resource allocation problem: vector numbers and intermediate interrupt numbers between interrupt controllers must all be allocated.
- This problem, like discovery and other kinds of configuration, is a generic issue: it is not the job of a single device driver, but instead the function of the common part of the I/O subsystem (though the PCI bridge drivers, USB HBA drivers, etc. may also play a role).

### 10.5 Naming devices

Once configured, an OS needs a way to refer to devices (again, driver instances really) from user space (for both user programs and system daemons). This is, of course, a naming problem, and it is important to understand what kind of problem. For example, it’s useful if the same device has the same name on different computers, rather than giving every device in the world a unique name.

**Example 10.27.** In older versions of UNIX, where every device was identified by a (major, minor) pair of integers, devices were named using the file system by creating a special kind of file to represent each device, using the `mknod` command.

We’ll cover the UNIX file system later on in Chapters 18 and 19 but as a preview, the major and minor device numbers were stored in the `inode`, meaning the “device file” took up very little space.

Devices are traditionally grouped in the directory `/dev`. For example:

- `/dev/sda`: First SCSI/SATA/SAS disk
- `/dev/sda5`: Fifth partition on the above
- `/dev/cdrom0`: First DVD-ROM drive
- `/dev/ttyS1`: Second UART

In the truly old days, all drivers were compiled into the kernel. Each driver probed the hardware itself for any supported devices, and the system administrator populated `/dev` manually using `mknod` when a new device was connected.

Modern hardware trends have resulted in a huge explosion of devices, and this approach is unworkable. People simply want to plug a device in and have it work.
Example 10.28. In modern versions of Linux (including Android), /dev still contains files corresponding to all the devices, but /dev itself is no longer a “real” file system (i.e. residing on storage). Instead, it is an illusion created by a device discovery process (called udev) which repeatedly polls sysfs.

10.6 Protection

Another function of the I/O subsystem is to perform protection:

- Ensuring that only authorized processes (such as user-space drivers, or the kernel) can directly access devices.
- Ensuring that only authorized processes can access the services offered by the device driver
- Ensuring that a device cannot be configured to do some malicious to the rest of the system

There are a number of mechanisms for achieving this. Putting device drivers in the kernel makes it easy to control access to the hardware, but you have to trust the device drivers to do the right thing since they are now part of the kernel.

UNIX controls access to the drivers themselves by representing them as files, and thereby leveraging the protection model of the file system.

The last point is more difficult. DMA-capable devices are in principle capable of writing to physical memory anywhere in the system, and so it is important to check any addresses passed to them by the device driver. Even if you trust the driver, it has to make sure that it’s not going to ask the device to DMA somewhere it shouldn’t.

One approach is to put a memory management unit (MMU) on the path between the device and main memory, in addition to each core having one. Such a unit is called an IOMMU, and its main purpose is to provide I/O virtualization to virtual machines, which we’ll cover in a later chapter.

10.7 More on I/O

There’s a lot more to say about I/O, since it’s one of the most important things the OS does. However, we’ll see more of this later when we look at virtual memory [16], paging [17], storage [19] and networking ??.