Part III: Persistence

**Chap 35: A Dialogue on Persistence**

Making information persist, despite computer crashes, disk failures, or power outages is a tough and interesting challenge.

**Chap 36: I/O Devices**

We will discuss **input/output (I/O) device** and how OS interact with such identity.

**36.1 System Architecture**

Diagram

Description automatically generated

The CPU is attached to the main memory of the system through the **memory bus** or interconnect. Some devices (e.g., graphics) are connected to the system via a general **I/O bus**, which many systems called **PCI**. Lower down, we have a peripheral bus, such as SCSI, SATA, USB. These connect slow devices to the system, including disks, mice, and keyboards.

We put them in a hierarchical structure because of physics and cost. The faster the bus, the shorter it must be. Thus, high-performance memory bus does not have much room to plug devices and such into it. In addition, high performance bus is costly. Thus, components that demand high performance are nearer to the CPU. The further away, the lower the performance.

Figure 36.2 shows the diagram of Intel Z270 chipset. The CPU connects most closely to the memory system and the graphics card. The CPU connects to an I/O chip via **DMI (Direct Media Interface)**. The rest are connected to this I/O chip via different interconnects (eSATA or external SATA, ATA or AT attachment, SATA or serial ATA).

Below the I/O chip are a number of **USB (Universal Serial Bus)** connections, which in this depiction enable a keyboard and mouse to be attached to the computer.

On the left of the system is where higher performance devices can be connected via **PCIe (Peripheral Component Interconnect Express)**.

Diagram

Description automatically generated

**36.2 A Canonical Device**

Diagram, text

Description automatically generated

A canonical device has two important components. The first one is the hardware **interface** that represents to the rest of the system. Just like a piece of software, hardware must also present some kind of interface that allows the system software to control its operation. The second part is its **internal structure**. This part of the device is implementation specific and is responsible for implementing the abstraction the device presents to the system. Simple devices will have one or a few hardware chips to implement their functionality. More complex devices will include a simple CPU, some general purpose memory, and other device-specific chips to get their job done.

In modern controllers, they have hundreds of thousands of lines of **firmware** (software within a hardware device) to implement the chips’ functionality.

**36.3 The Canonical Protocol**

The interface consists of **status register** to read to see the current status of device, **command register** to tell the device to perform a certain task and **data register** to pass data to the device or get data from the device. The protocol is described as follows:

Text

Description automatically generated

First, the OS wait until the device is ready to receive a command by checking the status register. We call this **polling** the device. Then, the OS sends some data down to the data register. When the main CPU is involved with the data movement, we called this **programmed I/O (PIO)**. Next, the OS writes a command to the command register. This simply lets the device know that both the data is present, and it should begin working on the command. Finally, the OS waits for the device to finish by polling it.

This protocol is simple and working. However, there are some inefficiencies and inconveniences. The first thing is that it wastes a lot of CPU time just waiting for the device.

**36.4 Lowering CPU Overhead With Interrupts**

To deal with the above problem, we use interruption. Instead of polling the device repeatedly, the OS can issue a request, put the calling process to sleep, and context switch to another task. When the device is finished worth the operation, it will raise a hardware interrupt, causing the CPU to jump into the OS at a predetermined **interrupt service routine (ISR)** or an **interrupt handler**, which is just a piece of OS code that will finish the request and wake the process waiting for the I/O and proceed.

By using this solution, the timeline having an interruption is described as follows:

A picture containing graphical user interface

Description automatically generated

However, we can utilize the idle time like this:

Graphical user interface, application

Description automatically generated

We can have process 2 to run while the disk services process 1’s request.

Interrupt is not always the best solution. For example, if a device performs task very quickly, the first poll usually finds the device to be done with task. Using an interrupt in this case will actually slow down the system: switching to another process, handling the interrupt, and switching back to the issuing process is expensive.

A safe solution is a hybrid of polling and interrupt where sometimes we use polling, sometimes we use interrupt.

Another reason to not use interrupt is **networks**. When a huge stream of incoming packets each generate an interrupt, it is possible for the OS to live-lock, that is, find itself only processing interrupts and never allowing a user-level process to run and actually service the requests.

One interrupt-based optimization is **coalescing**. In such a setup, a device which needs to raise an interrupt first waits for a bit before delivering the interrupt to the CPU. While waiting, other requests may soon complete, and thus multiple interrupts can be coalesced into a single interrupt delivery, thus lowering the overhead of interrupt processing.

**36.5 More Efficient Data Movement With DMA**

Problem might arise when PIO transfer a large chunk of data to a device. The CPU will be overburdened with a rather trivial task and wastes a lot of time and effort that could better be spent running other processes.

Table

Description automatically generated

The c represents the steps to copy the data from memory to the device explicitly. This can waste some CPU time.

A solution to this problem is **Direct Memory Access (DMA)**. A DMA engine is essentially a very specific device within a system that can orchestrate transfers between devices and main memory without much CPU intervention.

Graphical user interface, text, application

Description automatically generated

To transfer data to the device, the OS would program the DMA engine by telling it where the data lives in memory, how much data to copy and which device to send it to. At that point, the OS is done with the transfer and can proceed with other work. When the DMA is complete, the DMA controller raises an interrupt, and the OS thus knows the transfer is complete.

**36.6 Methods Of Device Interaction**

The first method of device communication is to have explicit **I/O instructions**. These instructions specify a way for the OS to send data to specific device registers and thus allow the construction of the protocols described above.

For example, on x86, the in and out instructions can be used to communicate with devices. For example, to send data to a device, the caller specifies a register with the data in it, and a specific port which names the device. Executing the instruction leads to the desired behavior.

Such instructions are usually **privileged**.

The second method to interact with devices is known as **memory-mapped I/O**. With this approach, the hardware makes device registers available as if they were memory locations. To access a particular register, the OS issues a load (to read) or store (to write) the address; the hardware then routes the load/store to the device instead of main memory. This approach does not introduce new instructions.

**36.7 Fitting Into The OS: The Device Driver**

Again, we will solve this through the technique of **abstraction**. At the lowest level, a piece of software in the OS must know in detail how a device works. We call this piece of software a **device driver**, and any specifics of device interaction are encapsulated within.

Table

Description automatically generated

A file system is completely oblivious to the specifics of which disk class it is using; it simply issues block read and write requests to the generic block layer, which routes them to the appropriate device driver, which handles the details of issuing the specific request.

The diagram shows a **raw interface** to devices that enables special applications to directly read and write blocks without using the file abstraction. Most systems provide this type of interface to support these low-level storage management applications.

**36.8 Case Study: A Simple IDE Disk Driver**

We will examine an IDE disk drive. The source code is shown as follows:

Text, table

Description automatically generated with medium confidence

An IDE disk presents a simple interface to the system, consisting of four types of register: control, command block, status, and error.

The basic protocol to interact with the device is as follows:

* **Wait for drive to be ready**: read status register (0x1F7) until drive is ready and not busy.
* **Write parameters to command registers**
* **Start the I/O**
* **Data transfer (for writes)**
* **Handle interrupts**
* **Error handling**

Most of this protocol is found in the xv6 IDE driver:

A picture containing table

Description automatically generated

The first is ide\_rw(), which queues a request or issues it directly to the disk. In either case, the routine waits for the request to complete, and the calling process is put to sleep. The second is ide start\_request(), which is used to send a request to the disk. The start request routine uses the third function, ide\_wait\_ready(), to ensure the drive is ready before issuing a request to it. Finally, ide\_intr() is invoked when an interrupt takes place.