I/O Systems

Hello and welcome to this lesson on IO Systems and device drivers.

We will start by getting an overview of how the IO system and the control of devices fit into a typical operating system.

We will see that some part of the device control is performed by device independent management layers and some other by device dependent components, so called device drivers.

We will then have a closer look at device controllers and how they interact with device drivers.

We will explore the space of such interactions, and we will compare polling with interrupt driven control, and programmed I/O with I/O based on so-called direct-memory-access channels, short DMA.

We will then so-to-say design a little IO architecture that reflects what some of the more modern operating systems have. In fact, we will borrow some ideas from windows.

From this discussion of modern IO architectures, a generic structure for device drivers naturally emerges, and we will look at the various parts of such a device driver.

After this lesson you will understand how devices are controlled by the OS software, and you will understand the structure and function of a device driver.

Let's begin.

Structure of I/O System

Let's look at a system, with the user on top and devices at the bottom. Here we have a keyboard and a printer.

Between user and devices is the kernel, with the system call interface on top and the SW/HW interface at the bottom.

The programming interface to devices is provided by the device controller, where we have one for each device.

The IO system provides an IO device interface to the user, which provides generic device access functions, such as open/close, get and put data, and io_control.

A generic device management layer connects the user to the device-specific device driver, which is connected to the device controller, in this case for the keyboard, and here for the printer.

[Cont.]

In a real system, there are multiple classes of devices, such as block-based devices, for example hard disk and DVD writer. Such devices are called “block-based” because one reads/writes blocks at a time, often in a random-access mode. This is different from streaming based devices, such as keyboards, mice, printers, and many other devices, where one reads or writes streams of data.

A third family of devices, which are often treated as stream-based devices, are network devices.

All these devices are often managed by the following layered system, where the user sees a generic IO interface, which sits on top of a device-Independent IO management layer.

This one interacts with the device drivers, which provide the device-dependent IO management layer.

Device Controllers

Let’s focus on the interface between device controller and device driver.

We have a device, a printer in this case.

The programming interface to the device is provided by the so-called “device controller”.

The user interface of the controller is a collection of device registers.

Of which, the “opcode register” is one. The opcode register is where the user writes the command that is to be executed by the device.

The operand registers are used to store parameters for the command. The user typically deposits the parameters in the operand registers and then initiates the command by writing the appropriate code of command into the opcode register.

The user then checks whether a command that has been completed by reading the content of the “busy register”. (5)

If the content of the busy register indicates that the operation has completed, the user can check the status register for additional information about the execution of the operation, such as error codes.

Once the operation is completed, any return data is available in the data buffer, which the device driver can read to copy data into main memory.

In some devices, the data buffer can also be used to transfer data from main memory to the device.

The device driver interacts with the device controller by writing to the opcode register and the operand registers reading from the busy and status register, (10) and reading or writing from or to the data registers, depending on the operation.

Let's focus a bit more closely on how the device driver interacts with the device controller.

[cont.]

When we design an IO system, we need to address three questions.

1. **First, how do we access the device registers?**
2. **Second, when we start the device on an operation, how do we know when the controller is done?**
3. **And finally, how do we move the data between the controller data buffer and the main memory?**

Let's focus on how to access the device registers first.

Explicit vs. Memory-Mapped Device Interfaces

One way is to explicitly access device registers through specific IO operations.

This is popular on many architecture, for example Intel, where the cores have an I/O bus, which is different from the memory bus.

In general, IO devices are controlled through some form of io_store operations, where the content of a CPU register is written to a particular device register of a device with a given number.

On the x86, this is realized through the "out" instruction, which writes a particular value to a given so-called "port", which is identified by a number.

One way to implement a write to disk connected to an ATA controller, would be, to send a control command to the disk controller to position the head. In this case this is done by writing the 28-bit block number, eight bit at a time, to port numbers hex 1F1 to hex 1F6.

In hex 1F6, we also write the disk identifier for this particular controller, since ATA controllers can manage a master and a slave device each.

We then specify the desired operation by writing the appropriate code to port hex 1F7. This writes the command into the opcode register of the device.

Now we can start writing the data to the data buffer of the controller, by writing the data, two byte at a time, to port hex 1F0.

This example illustrates how we can control a device by explicitly writing (and later reading) to and from the device registers.

[cont.]

Alternatively, device registers can be accessed in a memory-mapped manner.

In this approach, a portion of the memory is mapped to the device registers, and the user accesses them by reading and writing from and to locations in the mapped memory.

In many x86 systems, for example, the VGA display is mapped in this way to the 128kB of memory starting at address 640kB.

Device Controllers

Now that we know about explicit and memory mapped access to device registers, let's address how the device driver knows when the device controller is done with the requested operation.

Programmed I/O with Polling

One way for the device driver to monitor the progress of the device operation is through polling.

In this approach, the CPU explicitly checks the status of the device until the device indicates that it is done.

Let's see how this works.

The device driver writes the parameters for the operation into the operand registers.

It then writes the code, the requested operation, into the opcode register.

Now the controller triggers the device to execute the requested operation. At the same time, the "busy" register is set, to indicate to the device driver that the device is busy.

The device driver periodically checks the value of the busy register, waiting until the "busy" register is clear.

Once it is, and the CPU reads a "not-busy" value,

The CPU continues by checking the status register whether there has been an error, and if there is none, it initiates the data transfer between the data buffer and the main memory.

Programmed I/O with Interrupts

How does the same apply when we use interrupts?

Similar to before, the device driver initiates the operation by writing the operation parameters into the operand registers, and the operation code into the opcode register.

The device controller starts the operation on the device, which may take some time.

Once the operation is done, the device controller raises an interrupt, (5) which causes the CPU to start the interrupt service routine associated with this device.

The interrupt service routine in turn checks the status register, and if everything is ok, start transferring the data between device and main memory.

In some operating systems the part of the device driver that sets up the device and sends the parameters and operation code to the device, “top half” of the device driver. The “top half” performs the “synchronous” part of the device control.

The part that is handled asynchronously in the interrupt service routine, on the other hand, is sometimes called the “bottom half” of the device driver.

Device Controllers

Once we know how to let the device driver know when the device controller is done, we still need to find a way to best move the data between device data buffer and main memory.

Data Transfer: Programmed I/O

One way is to have the CPU transfer the data explicitly, one word at a time, between device buffer and main memory.

This is very simple.

The problem with this approach is that it puts a lot of stress on the CPU. We need to find a way to transfer the data without direct CPU involvement.

Data Transfer: Direct Memory Access (DMA)

This is typically done using so-called “direct-memory-access” or DMA channels.

Once the device is ready, and an interrupt is issued to tell the device driver, for example, the CPU checks the status register, and then initializes a DMA channel to transfer the data between the data buffer and a specified portion of main memory.

Once the DMA transfer is set up, the CPU is free to do other things.

Well, it is not totally free because it has to compete with the DMA transfer for access to memory.

In practice this is not too much of a problem, (5) as the cache reduces the number of requests that actually go to memory, and therefore there is less contention.

Modern Driver Architectures

Now that we know how the interaction between device driver and device controller looks, let’s investigate how modern IO systems are architected. For this we loosely follow the Windows IO architecture.

Let’s simplify the problem by focusing on the hard disk device, and let’s further focus on the case where disk operations are triggered by the file system.

Therefore, we ignore the demands of the virtual memory manager.

We also eliminate the interface level, to simplify the discussion.

Let’s turn our device-independent block device manager into a generic IO manager, and let’s move the disk a bit over to make some space.

[cont.]

Let’s be a bit creative, and let’s think of the file system as a device that responds to file read and write operations.

The file system operations are accessed through the file system driver.

[cont.]

When the user now issues a read or write request to a file, the request is intercepted by the IO manager.

A so-called IO Request Block (short IRB) is created, which contains all the information about the read or write request.

The IRB is forwarded to the file system driver, which figures out which device is to handle the request. Once it knows that, it sends the IRB to the IO manager, which routes it to the appropriate device driver.

The driver issues the request to the device controller, and when the response is ready, it updates the IRB and sends it back to the file system driver.

This return the IRB to the IO handler, which triggers a return from the system call.

[cont.]

This architecture of having the IO manager route IRB requests back and forth between file system driver and device driver of the storage devices is very flexible. In particular it is amenable to what is called “device-driver layering”, which we illustrate here.

Let’s assume that we want to store the data of a file system transparently on a collection of drives rather than on a single drive.

We don’t particularly want the file system to know about this. The disks can be connected to the same device driver, or each disk may be connected to its own device driver as well; it does not matter.

This architecture allows us to add a new “virtual” device, which we call a “volume”. A volume represents one or more disks. What is special about it is that it looks like a disk driver to the file system and like a file system to the actual disk driver.

When a file IO request now comes in, the IO manager creates the IRB and forwards it to the file system driver, just as before.

The driver now identifies the hard drive where the data is stored, (5) and forwards the IRB to what it thinks is a disk driver. The volume now identifies the actual disk that has the data stored, and forwards the IRB to it driver.

After the disk has handled the request, the IRB is updated, and send back to what the disk driver thinks is the file system.

The volume driver forwards it on to the actual file system driver, which returns it to the IO manager.

The IO manager returns from the system call.

This architecture has other benefits as well.

-

For example, it is easy to add a power manager, which can generate IRB's to send to device drivers to control their power saving modes.

Similarly, a plug-and-play manager would keep the various components informed when new devices are plugged in or existing devices removed from the system. Whenever the plug-and-play manager is informed by device drivers, it would update other components, such as the volume manager for example when a new disk is connected to the system.

Generic Device Driver Structure

The device driver architecture for such an IO system may look as follows.

The top end would have the routine "start IO" which would load the operand registers and the opcode register to initiate the IO operation.

After that the bottom end would take over.

The interrupt service routine of this device driver would be called whenever the operation would complete on the device controller, and an interrupt would be raised.

A special class of "bottom end" routines are the so-called "deferred procedure call" or DPC routines. Because the execution of interrupt service routines is so disruptive to the rest of the system, device driver designers would like to separate time-critical operations in the bottom end from not-so-time critical ones.

The time critical operations are packed into the interrupt service routine which executes immediately when the interrupt is raised. The remaining operations are packed into the DPC routine, which in turn is queued up to be executed by the scheduler later after the interrupt service routine returns. When the system is done executing interrupt service routines it checks whether there are DPC to execute.

The device architecture may require additional device management routines to be present, for example, an initialization routine which is called whenever the device needs to be initialized, an "add-device" routine, which would be triggered by the plug-and-play framework, and one or more power-management routines.

Finally, the device driver may have to respond to IRB that are forwarded by the IO manager.

This is done by what in Windows parlance is called "dispatch routines".

The device driver designer develops the device driver by populating these routines with the functionality necessary to control the device.

I/O Systems Conclusion

With this, we have come to the conclusion of our exploration of I/O systems and device drivers.

We started by looking at the overall structure of IO systems.

We then had a closer look at how device drivers interact with device controllers.

We explored polling and interrupt driven i/o. One point that we did not discuss with polling was that the polling frequency needs to be carefully chosen. If we poll too frequently, we burn CPU bandwidth. If, on the other hand, we don't poll frequently enough, this leads to significantly increased IO latencies or even missed IO operations. Given these problems, one may wonder if and why systems still implement polling. One big advantage of polling is that the CPU bandwidth devoted to IO is controlled by the system, and not by the device. Polling-based systems are therefore better protected against devices that malfunction and that would flood the system with interrupts (so called "babbling idiots").

We compared programmed IO with DMA based IO, where special Direct memory access channels relieve the CPU from having to shuffle data between memory and devices.

After this close look on the interaction with device controllers we zoomed out and looked at modern IO architectures. These architectures are characterized by an object based approach to IO handling. IO requests are packaged into so-called IO Request Blocks, which are then routed between device drivers.

This architecture gave rise to a generic structure for device drivers, which consists of "top half" and "bottom half" routines, device management routines to connect to power and plug-and-play managers, and so-called "dispatch routines" to hook into the IO manager infrastructure.

We genuinely hope that you enjoyed this exploration of IO systems and device drivers.

Modern IO architectures provide an easy way to expand the functionality of operating systems by simply adding new devices that take on functionalities that one may not at first associate with devices proper, such as cloud-based services, privacy, and others.

Thank you for watching.