LECTURE 5: INPUT/OUTPUT MANAGEMENT

One of the main functions of the operating system is to control all the computers I/O devices. It must issue commands to the devices, catch interrupts and handle errors. It should also provide an interface between the devices and the rest of the system that is simple and easy to use. I/O component represents a significant fraction of the total operating system.

4.1 I/O Hardware

I/O devices can be roughly divided into two categories: **block devices** and **character devices.** Block devices store information in fixed-size blocks, each with its own address. The essential property of block devices is that it is possible to read or write each block independently of all other. Disks are the most common block devices. A disk is a block addressable device because no matter where the arm currently is, it is always possible to seek for another cylinder and then wait for the required block to rotate under the head.

The other type of I/O device is the **character device**. A character device delivers or accepts streams of characters, without regard to any block structure. It is not addressable and does not have any seek operation. Printers, Mice, keyboards and other devices that are not disk-like can be seen as character devices.

This classification is not perfect, as some devices do not fit in either. Clocks, for example, are not block addressable, neither do they generate or accept character streams. There are other classifications that have also been used, for example, those that can be accessed sequentially or randomly, those that transfer data synchronously or asynchronously. Dedicated or shared. Operating systems need to handle all devices despite these variations. One key goal of an operating system's I/O subsystem is to provide the simplest interface possible to the rest of the system.

A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with computer via a connection point termed as a **port**. If one or more devices use a common set of wires, the connection is called a **bus**.

4.1.2 Device Controllers

I/O units typically consist of a mechanical component and an electronic component. The electronic component is called the **device controller** or adapter. A device controller is a collection of electronics that can operate a port, a bus or a device. On PCs, it often takes the form of a printed circuit card that can be inserted into a slot on the computer's **parent board.** The mechanical component is the device itself. The controller card usually has a connector on it, into which a cable leading to the device itself can be plugged. Many controllers can handle a several identical devices. The interface between the controller and the device is standard interface.

The operating system deals with the controller and not the device. To encapsulate the details and oddities of different devices, the operating system is structured to use device **driver modules**. The device drivers present a uniform device access interface to the I/O subsystem, much as system calls provide a standard interface between the application and the operating system.

Each controller has a few registers that are used for communicating with the CPU. On some computers these registers are part of the regular memory address space. This scheme is called **memory-mapped I/O**.

In addition to I/O ports, many controllers use interrupts to tell the CPU when they are ready for their registers to be read or written. An interrupt is an electrical event. Hardware Interrupt ReQuest line (IRQ) is a physical input to the interrupt controller. Some controllers are hardwired onto the system parent board, for example the keyboard controller.

The operating system performs I/O by writing commands onto the controller's registers. The floppy disk controller, for example, accepts 15 different commands, such as READ, WRITE, SEEK, FORMAT, etc. When a command has been accepted, the CPU can leave the controller alone and do other work. When the command has been executed, the controller causes an interrupt in order to allow the operating system gain control of the CPU and test the results of the operation. The CPU gets the results and device status by reading from the controller's registers.

4.1.3 Polling

Communication between the host and controller can take the simple protocol of handshaking. We assume that 2 bits are used to co-ordinate the controller and the host. The controller indicates its state through the **busy bit** and **status register**. The controller sets the busy bit when it is busy working. And clears it when it is ready to accept the next command. The host sets the command-ready bit when a command is available for the controller to execute. For example, the host writes output through a port, coordinating with the controller by handshaking as follows:

1. Host repeatedly reads the busy bit until it becomes clear.
2. Host sets the write bit and writes a byte to the **data out** register
3. Host sets the command ready register
4. When the controller notices that the command ready bit is set, it sets the busy bit
5. Controller reads the command register and sees the write command. It reads the data out register to get the byte and does the I/O to the device
6. Controller clears the command ready bit and bust bit to indicate that it is finished.

This loop is repeated for each byte. In step 1, the host is busy-waiting or *polling*.

4.1.4 Direct Memory Access

Many controllers, especially those for block devices, support Direct Memory Access (DMA). When DMA is not used, the following happens. First the controller reads the block from the drive serially, bit by bit, until the entire block is in the controller's internal buffer. Next, it computes the checksum bit to verify that no read errors have occurred. Then the controller causes an interrupt. The operating system can read the disk block from the controller's buffer byte by byte or a word at a time by executing a loop, with each iteration reading one byte or word from the controller's device register and storing it in memory.

DMA was invented to the CPU from the low level work of continuously looping to read bytes of data. The CPU gives the controller two items of information, in addition to the disk address of the block: the memory address where the block is to go and the number of bytes to store. After the controller has read the entire block from the device into its buffer and verified the checksum, it copies the first byte or word into the main memory at the address specified by the DMA memory address. Then it increments the DMA address and decrements the DMA count by the number of bytes just transferred. This process is repeated until the DMA count becomes zero at which time the controller causes an interrupt.

4.2 I/O Software

The basic idea of I/O software is to organize it as a series of layers, with the lower layers concerned with the hiding of peculiarities of the hardware from the upper ones and the upper ones concerned with presenting a clean, nice and regular interface to the users.

A key concept in I/O software design is **device independence**. This means that it should be possible to write programs that can read files on a floppy disk, hard disk or a CD-ROM without having to modify the programs for each different device type. It is up to the operating system to take care of the problems caused by the differences in media types, which may need different device drivers.

Closely related to device independence is the goal of **uniform naming**. The name of a file or device should simply be a string of characters or an integer not dependent on the device in any way. For example in UNIX, all disks are integrated in the file system hierarchy in ways that are transparent to the user. For instance, a floppy disk can be mounted on top of the directory */usr/ast/backup/* so that copying a file */usr/ast/backup/file1* copies the file to the floppy disk. In this way all the files and devices are addressed in the same way: by path name.

Another important issue in I/O software is error handling. Errors should be handled as close to the hardware as possible. If the controller discovers a read error, it should try to correct the error if it can. If it can't then the device driver should handle it, perhaps by trying to read the block again. Most errors are transient and if the operation is repeated they go way. Upper layers only handle errors that cannot be addressed by lower layers.

Another key issue is synchronous (blocking) versus asynchronous (interrupt driven) transfers. Most physical I/O is asynchronous – CPU starts the transfer and goes to do something else until the interrupt arrives. User programs are much easier to write if the I/O operations are blocking – after a READ operation the program is suspended until the data is available in the buffer. It is up to the operating system to make operations that are interrupt driven look blocking to the user programs.

Another concept is sharable versus dedicated devices. Some I/O devices such as disks and printers can be used by many users at the same time. There is no problem with multiple users having open files on the same disk at the same time. Other devices like tape drives have to be dedicated to a single user. The operating system should be able to handle shared and dedicated devices.

These issues ca be achieved in an efficient way by structuring I/O software in four layers:

1. Interrupt Handlers
2. Device Drivers
3. Device-independent Operating System Software
4. User-level Software

4.2.1 Interrupt Handlers

Interrupts should be hidden from users by the operating system. The best way to hide them is to have every process starting an I/O operation block until the I/O has completed and the interrupt occurs. The process can block itself by doing a DOWN on a semaphore, WAIT on a condition variable or a RECEIVE on a message, for example.

When the interrupt happens, the interrupt procedure does whatever it has to do in order to unblock the process that started it. It might do an UP on a semaphore or a SIGNAL on a condition variable. The process that was blocked will now be able to run.

4.2.2 Device Drivers

All the device-dependent code goes in the device drivers. Each device driver handles one device type or at most one class of related devices. For example, it would be good to have a single terminal driver, even if the system supported several different brands of terminals, all slightly different. We have seen that device controllers have registers used to give it commands. Device drivers issue these commands and check that they are carried out properly. Thus the disk driver is the only part of the operating system that knows how many registers that disk controller has and what they are used for.

The work of a device driver is to accept requests from the device-independent software above it and ensure that the request is executed. A typical request is to read block *n.* If the driver is idle at the point of the request, it carries it out immediately. If not, it queues it in its queue of pending requests to be dealt with as soon as possible. The first step in carrying out the request for a disk, for instance, is to translate it from abstract to concrete terms. This means the device driver has to locate the block on disk, check if the drive motor is running, and determine if the arm is positioned on the proper cylinder and so on. That is, it must determine what controller operations are required and in what sequence.

The device driver then issues the commands. In most cases, it blocks until an interrupt comes to unblock it. In other cases it doesn't have to block. Once the operation has been completed, it checks for errors. If everything is okay, it might have some data to pass to the device-independent software. If there are more requests, it processes them; otherwise, it blocks waiting fro the next request.

4.2.3 Device Independent I/O Software

The boundary between drivers and device independent software is system dependent, because some functions that could be done in a device-independent way may be done in the drivers for efficiency or other reasons. The following functions are normally done in device-independent software.

1. Uniform interfacing for device drivers
2. Device naming and protection
3. Providing a device-independent block size
4. Buffering
5. Storage allocation on block devices
6. Allocating and releasing dedicated drivers
7. Error reporting

The basic function of the device-independent software is to perform the I/O functions that are common to all devices and provide a uniform interface to the user-level software. A major issue in operating system is how files and I/O devices are named. The device-independent software takes care of mapping symbolic device names to the proper drivers. Closely related to naming is

protection. How does the system prevent users from accessing files and devices they are not entitled to access? In UNIX, this is done through the use of *rwx* bits.

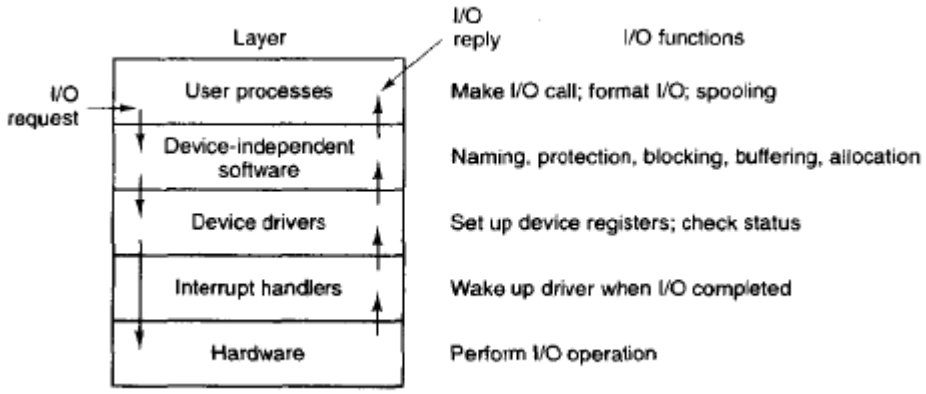
Different disks may have different sector sizes. It is up to the device-independent software to hide this fact and provide a uniform block size to upper layers, for example treating several sectors as a single logical block. Also some devices deliver their data one byte at a time (modems), while others deliver theirs as a block (NIC). These differences must be hidden. Buffering is also an issue for both character and block devices.
When a file is created and filled with data, the new disk blocks have to be allocated to the file. To perform this, the operating system needs a list or bit map of free blocks per disk, but the allocation algorithm is device independent and can be done above the level of the driver. Some devices can only be used by one process at a time. It is up to the operating system to examine requests and for device usage and accept or reject them depending on whether the device is available or not. The device driver does error handling. The driver reports the error to the system dependent software then how it is treated from there on is device independent.

4.2.4 User Space I/O Software

Although most of the I/O software is within the operating system, a small portion of it consists of libraries linked together with user programs and other programs running outside the kernel. System calls, including the I/O system calls are normally made by library procedures.

Not all user-level I/O software consists of library procedures. Another important category is the spooling system. Spooling is a way of dealing with dedicated I/O devices in a multiprogramming system Consider a typical spooled device: a printer. What is normally done is to create a special process called a **daemon**, and a special directory called a **spooling directory**. To print a file, a process normally generates the entire file to be printed and puts it in the spooling directory. It is up to the daemon which is the only process having permission to use the printer's special directory to print the files in the directory. Spooling is also used in the network by a network daemon. To send a file somewhere, a user puts it in a network spooling directory. Later on, the network daemon takes it out and transmits it.

The figure below summarizes the I/O system, showing all the layers and the principal functions of each layer. Staring at the bottom, the layers are: hardware, interrupt handlers, device drivers, device independent software and finally the user processes.



I/O software layers