Operating Systems (CS1022)

Input/Output

Yagiz Özbek Evren

Content

Operating systems control the Input/Output devices of computers. Operating systems are to rovide a simple and an easy interface to use between the IO devices and the system. This interface should be device independent, which means, it shouldn't vary between different devices. It must be the same for all. To demonstrate IO devices, here are some of example devices for both input and output:

Input devices:
Cameras, keyboard, mouse, disks, microphones, touchpads, etc.
Output devices:
Monitors, printers, speakers, head phones, projectors, etc.
Both Input/Output devices:
Modems, network cards, touch screen, etc.

In following pages, we will look into principles of IO hardware and software to understand what are the roles of these things, and how do they work/fit together.

1 Principles of IO Hardware

Different professions examine IO hardware in different ways which related to their profession. For example, electrical engineers examine it in terms of physical components of hardware like chips, wires, power supplies etc. On the other hand, programmers examine it in terms of interface, errors and functions. We are interested in the second examination, which programmers do. We will look the way hardwares have being programmed, and how do they work inside.

1.1 IO Devices

IO devices are divided into two categories; block devices and character devices. Block devices are stoing the information in fix-sized blocks, which has their unique addresses. These blocks' sizes are between 512B and 65536B. The important property of these devices are, blocks are not dependent to each other. This means, read and write processes in blocks are being done independently from the other blocks. We can count hard-disks and flash-disks (USB sticks) as block devices.

Character devices are able to deliver and accept character streams. It is being done irrespective of block structures. On the contrary, these devices are not addressable nor have seek operations. For this category, we can count printers and network interfaces as character devices.

Altough we divide IO devices into two cateories called block and character devices, this is not always perfect way to consider them. There are some devices, which we can not consider under these categories. For example, clocks. They are not addressable, nor they can not deliver or accept character streams. Their job is causing interrups. There are some more example devices which we can not consider under these categories, like, memory-mapped screens and touch screens. Altough there are

some exceptions, this categorization is quite enough.

Speed range of IO devices is huge. It makes the role of software development in terms of performance. In the following table, you can see the data rate of some common devices, also with behaviors and partners of them. Since we are always looking for more speed with improving technology, some devices' speed getting faster over years.

Device	Behavior	Partner	Data rate(KB/sec)
Keyboard	input	human	0.01
Mouse	input	human	0.02
Voice input	input	human	0.02
Scanner	input	human	400.00
Voice output	output	human	0.60
Line printer	output	human	1.00
Laser printer	output	human	200.00
Graphics display	output	human	60,000.00
Modem	input or output	machine	2.00-8.00
Network/LAN	input or output	machine	500.00-12,500.00
Floppy disk	storage	machine	100.00
Optical disk	storage	machine	1000.00
Magnetic tape	storage	machine	2000.00
Magnetic disk	storage	machine	2,000.00-10,000.00

Table 1

1.2 Device Controllers

IO units are mostly composed of two components, which are mechanical, and electronical components. We can consider these two parts for more understandable and generalized designs. One of them, electronical component, called device controller (adapter). The other component, mechanical component, is the device. Following figure shows this organization:

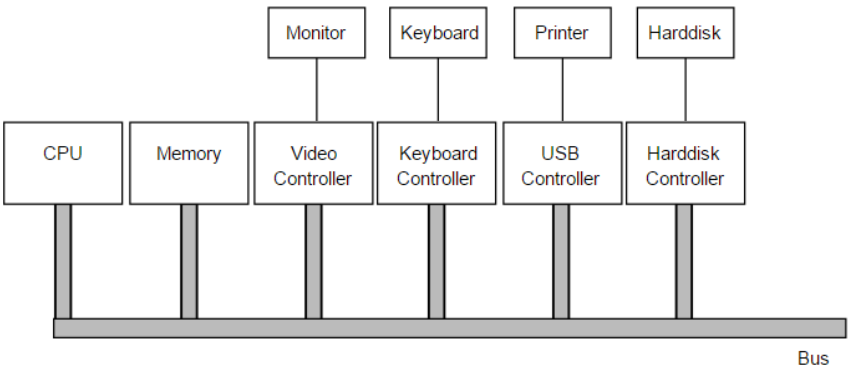

Figure 1

Mostly, there is a connector on the controller cards. Device can be plugged to this connector. Most of the controllers are able to handle several devices which are identical. If the interfacec between controllers and devices is standardized, companies can produce both controllers and devices that may fit with these interfaces. To give an example, we can say that many companies produce disk drives compatible with SATA or Thunderbolt interfaces. Following figure shows one example of common interfaces.

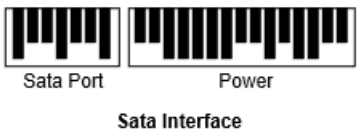

Figure 2

The interfaces which is used between controller and device are generally low level interfaces. For instance, we can format a disk composed of 2 million sectors with 512B each track. The things comes off the drive is a serial bit stream. Bit stream includes, in order, preamble, 4096 bits per sector, and checksum or Error-Correcting code.

In this process, controllor's job is converting this bit stream to some blocks of bytes. After that, it performs an error correction, if it is needed. Checksum is done to be declare the block as error free. After checksum is done, it is being copied to the main memory. This controller is for disk. Another controller we can illustrate is display monitor's controller. It is also works the same, works as a bit serial device. Level of it is equally the as low as disk controller. It read bytes includes characters to generate signals for modification of pixels will be displayed on the screen.

1.3 Memory-Mapped IO

Every controller has some registers. These registers are used in the communication with central process unit (CPU). Writing in registers let operating system control/command the device. It commands device to deliver or accept datas, turning it on or off, or to perform any other action if it is necesseary. Reading from registers let operating system get information about the device. Operating system can learn which state device is on in that time, or can learn if device is ready or not for another command by reading from registers. Most of the devices have a data buffer. Data buffer lets operating system to be able to read or write to the control registers.

Now, we face with the problem, how will CPU communicate with these device data buffers and control registers? There are two ways for this. One of them is, control register assigned with 8-16bit integer. This number called IO port number. IO

ports form a new thing called IO port space. IO port space is protected. Only operating system can access it. CPU can read in control register *"PORT"* and store the result in CPU register *"REG"*, using a special instruction, *"IN REG,PORT"*. Writing process of CPU is also similar. *"OUT PORT,REG"* instruction is used for this. CPU writes content from *"REG"* to control registers. In this communication, address spaces and IO are different. Following figure illustrates this scheme.

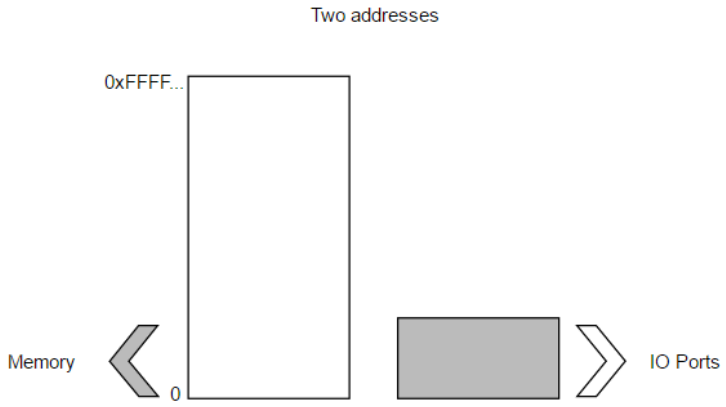

Figure 3

Second way of communication between CPU and devices is the method called memory-mapped IO. It is being done by mapping all control registers into memory spaces. Every control register assigned to special memory addresses. Assigned addresses are generally at the top of the address spaces in most of the systems. Following figure illustrates memory-mapped IO approach.

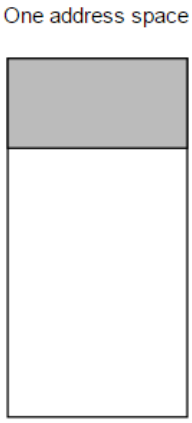

Figure 4

The practical work is being done with the following process. When CPU start reading data from memory or from IO port, bus' address line filled with address which CPU needs. Then, CPU sends *"READ"* signal to the bus' control line. Either if memory or IO ports get the request, they respond it. This process is not ambiguous since no addresses are assigned to memory or IO device.

Since there are different types of this communication process, naturally they have some advantages and disadvantages over each other. Since IO processes need read and write onto device control registers, you can programm them with assembly code. It is not possible to execute in and out processes without assembly code. In memory-mapped IO, since device control registers are variables, it can be programmed in C completely. In conclusion, memory-mapped IO lets you use programming languages like C, instead programming with assembly.

Again in memory-mapped IO, there is no need for an extra protection mechanism to keep user away from control registers' addresses. Operating system simply can avoid giving the address space of control registers to the usage of user.

In memory-mapped IO, instructions can refer both to memory, and control registers. For example, if an instruction checks the memory for a specific word, it may also check the control register. Therefore, you will need 2 instructions instead of 1. It slows down the responses from test procedure. This checking process can be done by a loop written in assembly.

Most of the computers have a caching form of memory words. It may do some caching process to the device control register. It will be really problematic. This is one of the disadvanteges of memory-mapped IO method. The assembly loop we talked before runs a loop to check a specific word. When it finds the word, it goes out of the loop. If this caching mechanism cached the wanted word from memory, loop will last forever, since software will never find the word and the loop ends with goto statement.

Since this problematic situation is critic, we have to avoid it. There is one way to prevent this situation. Hardware should selectively disable caching on per-page basis. This will make our hardware and operating system more complicated.

1.4 Direct Memory Access

Independently from CPU's IO method used, it always will be in a need of addressing the device controllers to communicate with them (read – write). CPU is able to make requests from IO controller. But it takes time. It can ask one byte at a time. To prevent wasting of CPU's time, there is an alternative method is in use. It is called Direct Memory Access (DMA). Operating system is able to use DMA, if there is an existing DMA controller on the hardware. DMA controllers are able to regulate transferings into multi-devices. It happens concurrently often.

Irrelevant to its location, DMA controllers are always connected to the bus, without any connection existing between itself and CPU. DMAs has some registers inside, which CPU can perform read and write processes. These registers are; byte count register, memory address register, and at least one control registers. The control

register is responsible for some actions. These are; specification of the IO ports will be used, transfer direction, quantity of the byte or word that will be trasfered in one time, and the quantity of bytes that will be transfered.

Following figure illustrated how DMA works:

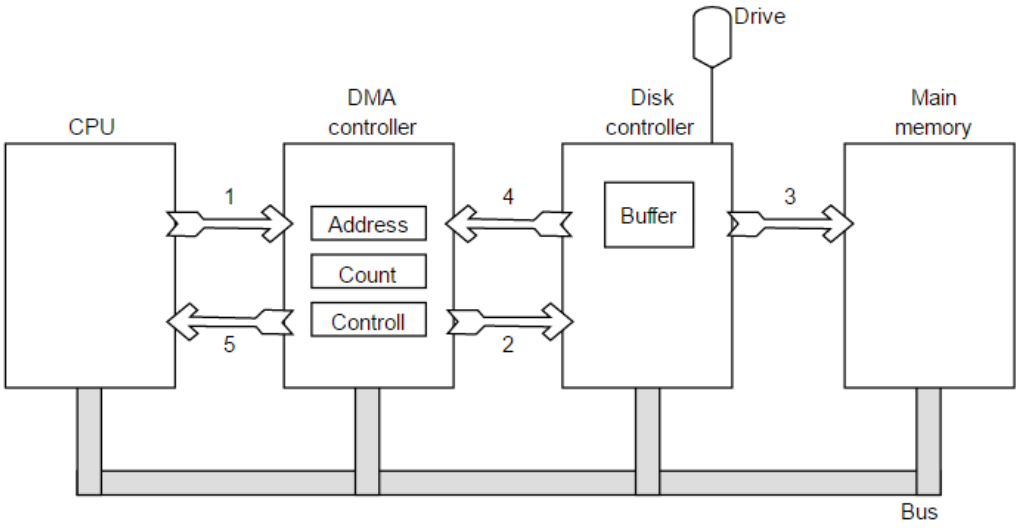

Figure 5

(1)First, CPU sets the registers inside DMA. Therefore, it will know what will be transfered, and where will it transfered. (2)Then DMA sends a read request to the disk controller. (3)Afterwards, write request sent to main memory. When writing process to main memory is over, (4)disk controller sends an acknowledgment signal to DMA's controller. Then, DMA's controller increases the memory address, and descreases the byte count. Until byte count goes under zero, DMA keeps sending read requests to the disk controller and the same process is done. When it is done, (5)DMA interrupts CPU. Then CPU understands that transfer process is over. While operating system is starting, everything will be ready. Disk blocks will be in memory, there will be no need for copy process.

This kind of DMA controllers are simple ones. There are more complex DMA controllers, which can perform more than one transfer process at the same time. But since complex DMAs can perform many transfer processes, there may be ambiguous situation since acknowledgement sent on same bus. To avoid this situation, often acknowledgement signal is sent on different acknowledgement lines for each DMA channel.

Most of the buses has two different modes to work. One of them is word at a time mode, and the other one is block mode.

The DMA controller wants to use the bus, and transfers a word back to the CPU. The DMA controller re-requests for each word until all data has been

transferred. In the same time, CPU wants to use the bus, but it has to wait. This mechanism called cycle stealing. It is called cycle stealing since controller steals use of bus from CPU for a while. In block mode, DMA controller start using bus by telling the device to take use of bus. When it is done with the bus, it releases the bus. This action is called burst mode. It is much more useful and efficient in practice than cycle stealing. There is also one more situation called fly-by mode. Data directly sent to main memory. This means, data being moved from one location to another one without passing through the DMA, it is not stored in DMA.

1.5 Interrupts

IO devices causes interrupts when their work is done. Interrupt is being done by signals sent to bus line. Interrupt controller detects the signal. Following figure illustrates the intterrupt operation.

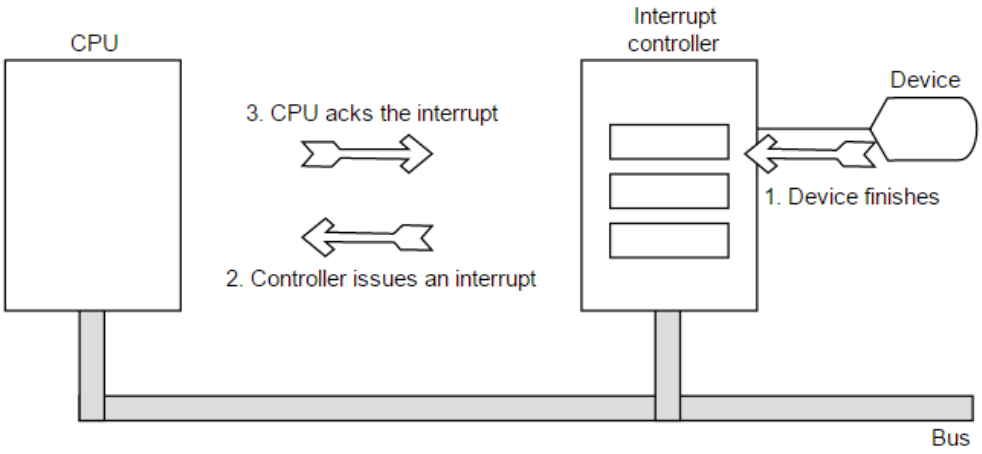

Figure 6

Controller handles the interrupt by putting a number to the address line with specifying which device wants to interrupt the CPU. After interruption signal, CPU stops its process and starts another process. The number that controller put into address line is indexing a table which is called interrupt vector.

Shortly, hardware work interrupts can be specified as the prevention of bad use of the CPU's waiting time for external events in its polling cycle. When the CPU is performing an event, the work is suspended with the interrupt signal, and the CPU is allowed to perform another task.

Work interruptions can be performed in hardware on control lines or on different systems, or can participate in work interrupt memory subsystems. If the interrupt is being executed in hardware, the Programmable Interrupt Controller (PIC) or Advanced Programmable Interrupt Controller (PIC) is connected to the interrupt pin of the processor along with the interrupt device. If the execution is part of the memory controller, the interrupts are designed in the system memory address space.

Common uses of job cuts are system timers, disk IOs, and power interrupt signals. In addition to these, UART and Ethernet use work interruptions in data transfer. Disk business interrupt signals are complementary in transferring data from peripherals to diskette or diskette peripherals. Power-off interruptions indicate that power-loss will occur or not. This allows the computer staff to systematically stop their work.

Precise & Imprecise Interrupts

When an interruption happens in a well-defined state, it is called prcise interrupt. In this intterruption, program counter saved in a known place. All existing instructions before interruption are done. Conversely, all the other insructions after interruption are not being executed. Execution of the instruction which is pointed by program counter is permitted. Generally in IO interrupts, instruction is not started. But if this is a page fault or a trap, program counter points the instruction which may cause this fault. Therefore, it can be started again later on. Following figure illustrated precise interrupt. Each instruction fully executed until the interrupt. On the other hand as we mentioned, following instructions did not start.

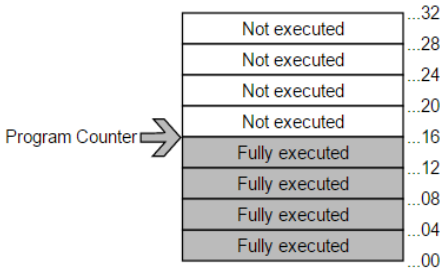

Figure 7

Conversely, imprecise interrupts are does not fit with the information for precise interrupts. Operating system programmer has to understand what is happening in this situation. It is undesirable situation. Instructions before program counter is neither fully executed, nor un-executed. Also completion rate of instructions has no order. Following figure illustrates this situation.

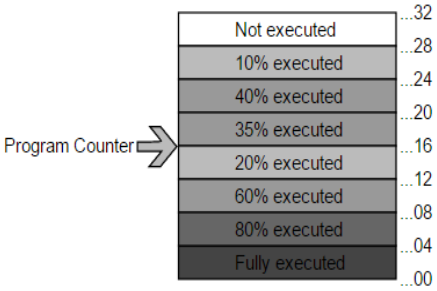

Figure 8

2 Principles of IO Software

2.1 Goals of IO Software

The most important thing is, again, independency. In IO software design, it is known as device independence. For example, if a program is able to read the input, it must read input from harddisk, USB stick, DVD or from any other device without any modification.

The name of any file should be simply strings, or integers which are not dependent on any device. This is uniform naming. Shortly, all of the files and the devices (e.g. USB stick) have the same way to be addressed. It is “path name”.

Another important goal of IO software is of course error handling. If errors can be handled as close as can it be to hardware, it is good. Otherwise, device driver is to handle it.

One of the most important issues is synchronous vs. asynchronous transfers. In synchronous data transfer, sender and reciever use the same clock signal. Conversely, in asynchronous data transfer, sender provides a synchronization to the reciever before transfer. It is up to operating system to choose one of them.

Buffering is also another issue of IO software. Generally, datas cannot stored in their final target places, which are come from a device. For instance, operating system can not know where to put the packet coming from a network. It will understand where to put it when operating system stores packet somewhere and examine. Buffering has major effect on IO performance.

The last issue is sharable vs. dedicated devices. An example to sharable devices is disks. More than one user can access disks and operate actions at the same time. This is why its name is sharable. It won't produce any problem. An example to dedicated devices is printers. Only one user can use it at the same time. Introducing dedicated devices comes with some problems. Deadlock is one of the problems. Operating systems are to handle both shared and dedicated devices to handle probable problems that may occur.

2.2 Programmed IO

There are three methods to perform IO. One of them is called programmed IO. Programmed IO is the simplest form of IO, where CPU does the all work.

Following figure illustrates programmed IO. In this figure, user wants to print a string on the printer. First, string assembled in the buffer in user space. And then operating system copies the buffer with string to kernel space. Then operating systems check if printer is ready or not. It waits until it gets ready. When printer

become available, operating system copies the first char of the string to printer's data register. This last action makes printer get activated. After the first character printed on the paper, operating systems again start checking if printer is ready for the next character. It happens until the string ends.

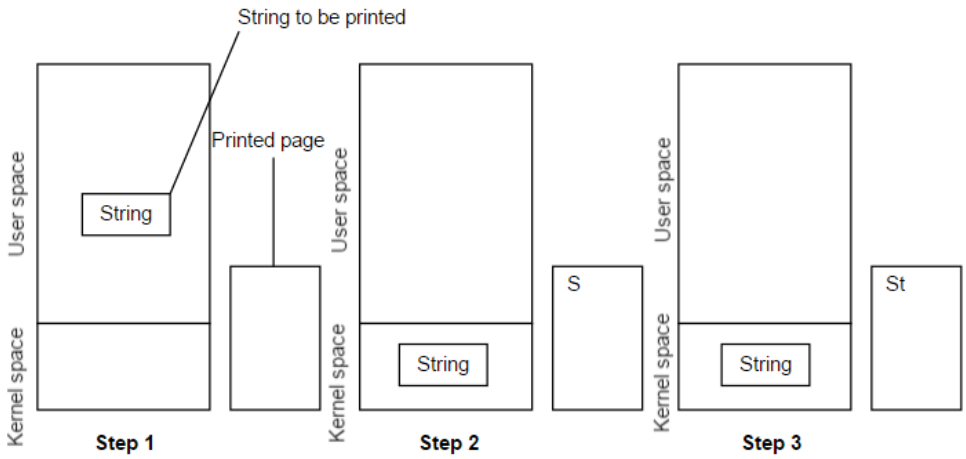

Figure 9

In short, first, data copied to kernel. Afterwards operating system starts a tight loop. This process outputs characters one at a time. This process known as polling, or busy waiting.

```
copy_from_user(buffer, p, count);              //p is the kernel buffer
for(i = 0; i < counter; i++) {                 //make loop for every character
      while (*printer_status_reg != READY);    //make loop until it is ready
            *printer_data_register = p[i];     //output a character
}
return_to_user();
```

Figure 10

Disadvantage of this IO method is, it block the CPU all the time, until IO process finishes. If printing takes small period of time, it is acceptable to use this method. But in complex systems, where any waiting is important, we need better method rather than programmed IO which blocks the CPU for a long time.

2.3 Interrupt-Driven IO

We can move on from the same printer example that we discussed in programmed IO to describe interrupt-driven IO. It does not buffer characters but prints them as they arrive. Let's say that, printer has a speed of printing characters. Let's say that it can print a character in 10 millisecond. CPU will wait in a loop of 10 millisecond to be able to output a character.

While waiting, if printing a character process is already done, interrupt is being sent to CPU to finish current process. Printing process of the current character ends,

and it generates an interrupt. Afterwards, it will stop the current process, save the state, and starts new process of printing.

2.4 IO Using DMA

Since we talked about interrups before, it is obvious to see the disadvantage of interrupt-driven IO. For every character is going to be printed, interrupt occurs. This is unwanted situation. Inturrputs take much time that we can not ignore. It takes to much time from CPU. This method is a solution for that.

IO using DMA is simply the same thing with programmed IO. The only difference is, DMA controller does all the work, instead of CPU. Therefore, CPU is not being bothered by this work. Compared to CPU, DMA is slow, but most of the time, DMA worth it to use. Following figure illustrates IO Using DMA.

3 IO Software Layers

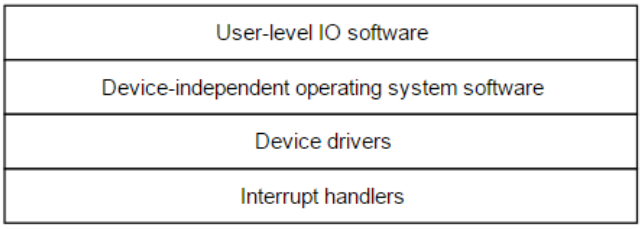

Figure 11

As shown in the figure above, IO software orianted in four layers. These layers has their unique functions. These functionalities and interfaces are different between systems. These layers are not one machine specific.

3.1 Interrupt Handlers

For most of the IOs, intterrupts are inevitable. There are no way to avoid some of them. These interrupts must be hidden somewhere in operating system, so that just a small portion of operating system can know about them. We can hide them by starting an IO operation block.

Interrupt procedure can does anything to handle the interruption, when interrupt happens. Afterwards, unblocks the waiting driver. The effect of interrupt is, driver which interrupted (blocked) before, is going to be able to run after new interrupt occurs in all cases. This model is the best, if the structure of the driver is kernell process.

It is easy to write and talk about, but actually processing interrupts is hard topic. It is not simply matter of taking the interrupt. Operating system deals with the most of the hard work. There are some steps for this work that will be done in software

after hardware interruption is done. Following steps are not independent between systems, some of them may be unnecasseary for some systems. Also order of them can be different system to system.

Steps are:

1	Saving all the registers that have been not saved by interrupt hardware
2	For the interrupt-service procedure, setting up a context
3	For the interrupt-service procedure, setting up a stack
4	Sending ackowledge signal for interrupt controller
5	Copying registers to the process table
6	Running the interrupt-service procedure
7	Deciding the process which will be run next
8	For the next process will be run, setting up the MMU context
9	Loading new process' registers
10	Starting the new process

Table 2

Interrupting process takes much CPU time with many instructions.

3.2 Device Drivers

We learned that device controllers' job is, with the use of their device registers, they control the status of the device, or give some commands to device. It can also perform both. Also, as we learned, number of these device registers are not standard. Devices can have different numbers of device registers. Hence, drivers of different devices are very different from each other. It is obvious to see when we compare a mouse device with a disk device.

Every IO device is connected to computer and need specific code to being managed. This code is known as device driver. Device drivers generally written by the manufacturer of the devices. There are several operating systems, hence, IO devices need different drivers to be managed by those different operating systems. Manufacturers provide different drivers for different operating systems (not for all of them).

Generally, drivers are able to manage different devices which belongs to same kind. For instance, one driver for disks can be used to manage same kind of disks. However, it is not a good idea to write a driver which provides management for different kinds of devices. For example, mouse's and keyboard's driver should be different.

With the USB technology, different kinds of devices can rely on the same driver. For example, as we mentioned mouse and keyboard example, however, if these IO devices connected via USB, same driver can be applied. The trick here is, USB drivers are stacked. But, we still have different device drivers.

To access hardware of devices, device drivers should be a part of operating system kernel. But it is possible to construct drivers which can run in user space. This kind of designs seperate drivers from each other, also seperate drivers from kernel, which can avoid system crashes. This is really important to make reliable system construction. But most of the operating systems expect drivers to run in kernel.

Operating system designers do not write the drivers, as we mentioned before. Since manufacturers write code for drivers, operating system must be able to allow these drivers to be installed. Device drivers' position is illustrated in the following figure.

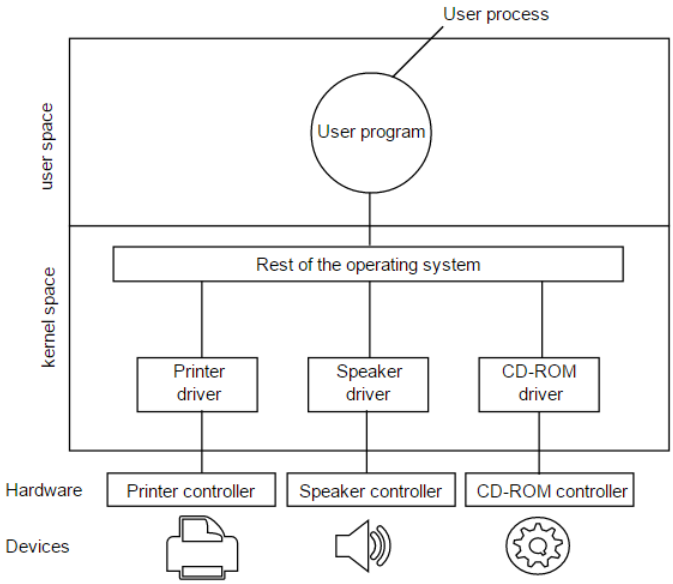

Figure 12

Drivers are categorized by the operating system into categories. Some of them are; block devices, which we talked before (e.g. Disks), and character devices (e.g. Printers, Keyboards etc.). Many operating systems have the standard interfaces for both block and character devices.

Some operating systems like UNIX, used to be single binary programs which contains every device drivers inside. Since those machines used by computer centers, IO devices are stable, they do not needed to be changed for long time. If there will be a new IO device will be added, system admin re-compile the kernel with new driver. PCs (Personal Computers) stoped the use of this model. If this operating system model will be used, it will be challenging situation or most of the users, since they are not able to re-compile kernel for new IO devices.

One of the most important functions of device drivers is, accepting read and write requests. There are also another functions such as, initializing the device when it is necesseary. Most of the drivers have similiar structure. For instance, they start checking the input parameters, and decided if they are ready or not. Also, driver checks if the device is in use or not. If it is, request will be queued to be processed later on.

Drivers are the places where sequence of commands are being decided. This is drivers' job, deciding the sequence of commands. Since driver decides which command will be issued, it starts writing it to controller's device register. After this process, it is sometimes needed to check if controller is okay with the command have been send, and to check if it is ready to get another command. After this processes are done, one of the following situations will be applied. One of them is, device driver waits until controller finish some of its work. Afterwards, it blocks itself. Other one is, it does not block itself. In first situation, driver is awakened with an interruption later on. In the second situation, driver never sleeps. In both situation, driver's work is completed. Now, it has to check for errors. If there are no errors, driver sends the data to software. Since this is a good aproximation, there are more factors which make the code more complicated.

Users can unplug devices suddenly, and it may damage the kernel. To prevent this, system must inform that IO device is suddenly removed from computer. It may led some crashes tough. For instance, in windows, every user may experience the issue of unplugging USB mouse IO device may led bluescreen error.

3.3 Device-Independent IO Software

Device-Independent IO software is, performing the IO functions in common to all devices, and providing a uniform interfaces. All functions of it are as follows:

Uniform interface
Buffering
Error reporting
Allocating-releasing dedicated devices
Providing device-independent block size

Table 3

Uniform Interface

Interfaces for different devices must be the same to prevent need of re-compile process of kernel. If their interfaces are different, everytime when a new device is added, kernel needs to be modified. Interfaces can not be the exactly same, but more or less they should be the same to prevent this situation. It is illustrated in the following figure.

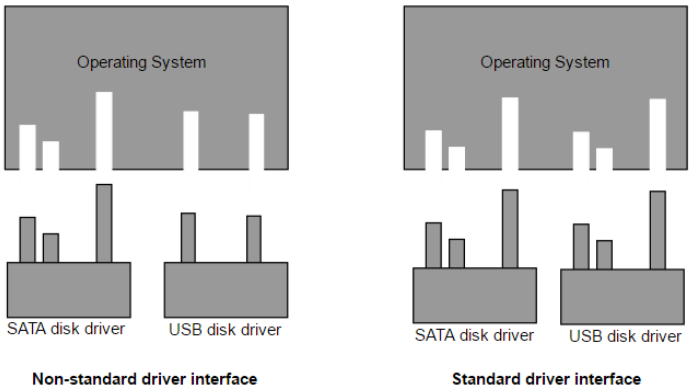

Figure 13

As you see in the Figure 16, first driver interfaces are not standardized, which is an unwanted situation. In the second one, it is standardized, therefore drivers can match with the interfaces. Another way for uniform interfaces is naming devices.

Buffering

There are two kinds of important buffering styles. One of them is double buffering, and the another one is circular buffeirng.

Double buffering comes up with the problem, since one buffer is full, there are no place to put characters that have been arrived. For example, in a modem, character comes to user space and buffered there with one buffer. Interrupt handler put the characters into the buffer which is in kernel. And when when this one also filled up, second kernel buffer is in charge. This is illustrated with the following figure.

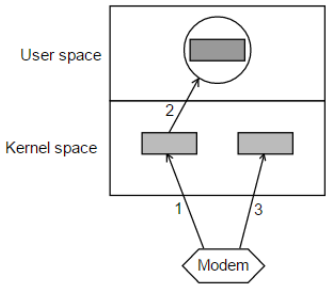

Figure 14

Circular buffering is different. It composed of a region and pointers. One of the pointers point the next free word, and the other one is pointing the first word. These words are data in the buffer. Circular buffering is illustrated in following figure.

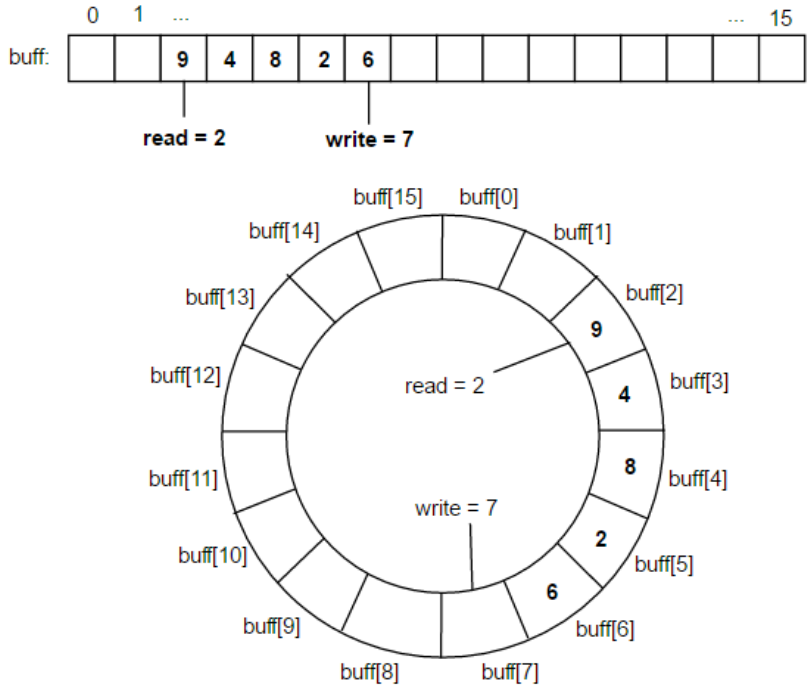

Figure 15

Error Reporting

When errors occur, they must be handled by the operating system. Most of the errors are specific to devices. They must be handled by the proper device driver. There are two types of IO errors. One of them is programming errors. These errors occur when some impossible requests appear. Handling of these errors are easy. Only thing should be done is, error report to the caller.

Second IO error type is actual IO errors. For instance, if disks tries to write data to damaged block, it is an actual error. In this situation, driver decides what will be done. If it is not aware of what should be done, it passes the problem to device-independent software. Software does, if it is simple error, asks user what to do with some GUI. If there is nothing that user can do, then system call returns an error code.

Allocating-releasing dedicated devices

Some of the IO devices, for example printers, just can be used for one process in one time. Operating system decided which operation (request) will be performed at that time. If device is in use, operating system rejects the request. Otherwise, it will accept that. A great approach is having a mechanism for allocating and releasing dedicated device mechanism. When request comes while device is working, caller is

being blocked. That request have been put into queue. After device becomes available, queued request(s) being performed. Following figure illustrates this mechanism.

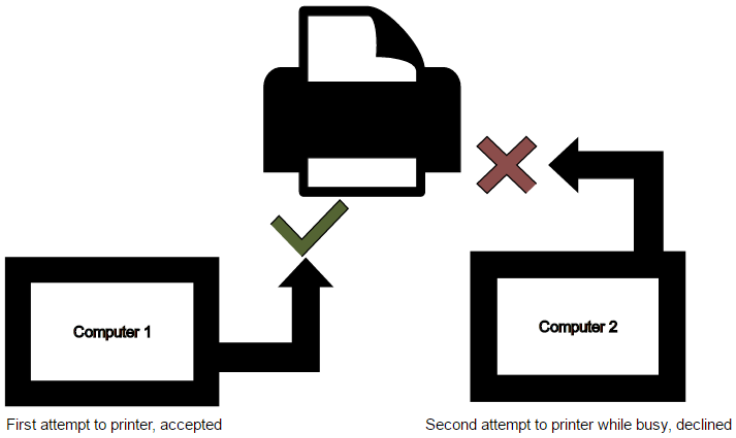

Figure 16

Device-Independent Block Size

This function is used when there are different sector sized disks, for example. Device-independent software provides uniform block size by treating sectors as a single block.

3.4 User-Space IO Software

In short, user-space IO software is libraries, which makes it much more easier to access kernel's functionality with simplified interface. Many of the user-level IO softwares composed of library procedures. These library procedures have some exceptions, which is spooling system. This is the way of dealing with dedicated IO devices.

To provide an interface to operating system, IO libraries such as *"stdio"* are in user-space. We can count printf( ), scanf( ) as examples of user level IO which are available in library "stdio" in the programming language C.

Following, and the last, figure illustrates the working principle of layers.

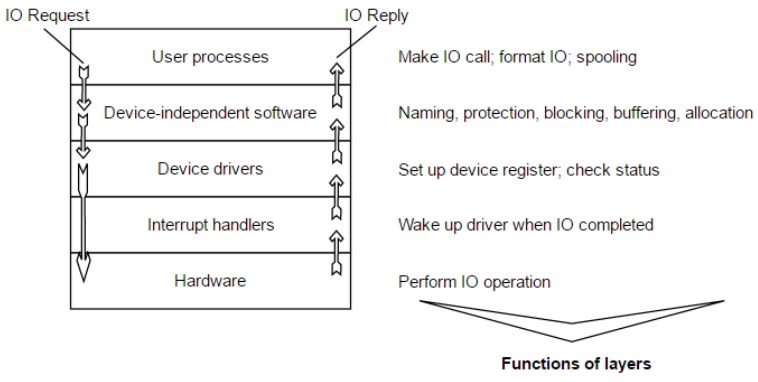

Figure 17