Input/Output

Week 10 - Lecture 1

Team

Instructors

Giancarlo Succi

Joseph Brown

Teaching Assistants

Vladimir Ivanov

Stanislav Litvinov

Alexey Reznik

Munir Makhmutov

Hamna Aslam

Sources

- These slides have been adapted from the original slides of the adopted book:

- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013

  Prentice-Hall, Inc.

  and customized for the needs of this course.

- Additional input for the slides are detailed later

Introduction to Input/Output (1)

- In addition to providing abstractions such as processes, address spaces, and files, an **operating system** also controls all the **computer's I/O (Input/Output) devices**.

- Operating system issue commands to the devices, catch interrupts, and handle errors.

Introduction to Input/Output (2)

- OS should also provide an interface between the devices and the rest of the system that is simple and easy to use.
- To the extent possible, the interface should be the same for all devices (device independence).
- The I/O code represents a significant fraction of the total operating system.

I/O Devices (1)

I/O devices can be roughly divided into **two** categories:

1. **Block devices.**
2. **Character devices**.

I/O Devices (2)

Block devices:

- Stores information in fixed-size blocks each one with its own address.
- Common block sizes range from 512 to 65,536 bytes.
- Transfers are in units of entire blocks.
- It is possible to read or write each block independently of all the other ones.
- Example: Hard disks, Blu-ray discs etc.

I/O Devices (3)

Character devices:

- Delivers or accepts stream of characters, without regard to block structure.
- Not addressable, does not have any *seek* operation.
- Example: Printers, mice(for pointing) etc.

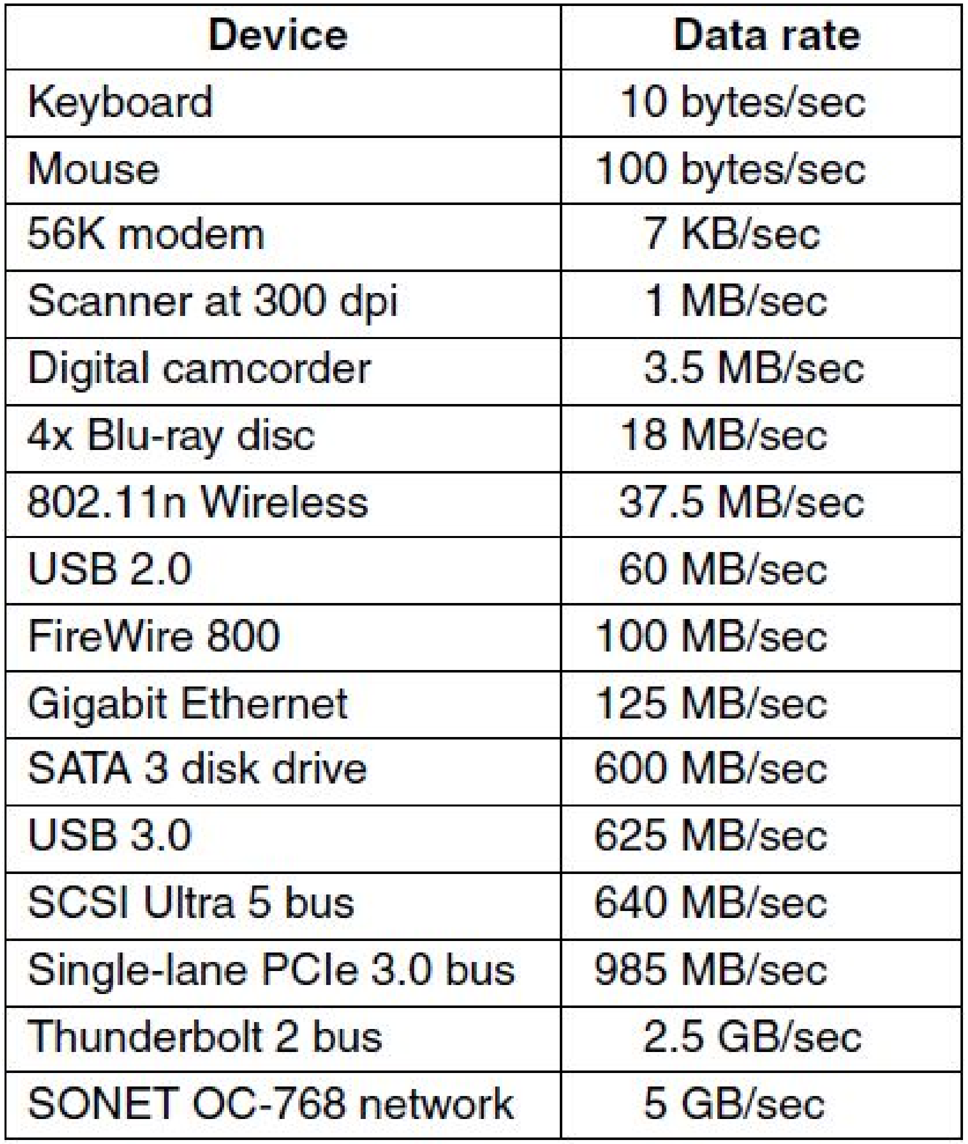
I/O Devices (4)

This classification scheme is **not perfect**. Some devices do not fit in. For example,

- Clocks are not block addressable. Nor do they generate or accept character streams.

- Memory-mapped screens and touch screens do not fit the model well either.

I/O Devices (5)



Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner at 300 dpi	1 MB/sec
Digital camcorder	3.5 MB/sec
4x Blu-ray disc	18 MB/sec
802.11n Wireless	37.5 MB/sec
USB 2.0	60 MB/sec
FireWire 800	100 MB/sec
Gigabit Ethernet	125 MB/sec
SATA 3 disk drive	600 MB/sec
USB 3.0	625 MB/sec
SCSI Ultra 5 bus	640 MB/sec
Single-lane PCIe 3.0 bus	985 MB/sec
Thunderbolt 2 bus	2.5 GB/sec
SONET OC-768 network	5 GB/sec

Figure 5-1. Some typical device, network, and bus data rates.

Device Controllers (1)

I/O units often consist of a mechanical component and an electronic component.

- The electronic component is called the **device controller** or **adapter**. Such as:
  On personal computers, it often takes the form of a chip on the parentboard or a printed circuit card that can be inserted into a (PCIe) expansion slot.

- The mechanical component is the device itself.

Device Controllers (2)

The **interface** between the controller and the device is often a very low-level one. For example,

A disk, might be formatted with 2,000,000 sectors of 512 bytes per track. What actually comes off the drive, however, is a serial bit stream, starting with a **preamble**, then the 4096 bits in a sector, and finally a checksum, or **ECC** (**Error-Correcting Code**).

Device Controllers (3)

- The preamble is written when the disk is formatted and contains the cylinder and sector number, the sector size, and similar data, as well as synchronization information.
- The controller’s job is to convert the serial bit stream into a block of bytes and perform any error correction necessary.
- The block of bytes is typically first assembled, bit by bit, in a buffer inside the controller.
- After its checksum has been verified and the block has been declared to be error free, it can then be copied to main memory.

Memory-Mapped I/O (1)

Each **controller** has few **registers** that are used for communicating with the CPU.

•By **writing into these registers**, the operating system can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action.

•By **reading from these registers**, the operating system can learn what the device's state is, whether it is prepared to accept a new command.

Memory-Mapped I/O (2)

In addition to the control registers, many devices have a **data buffer** that the operating system can read and write.

For example:

A common way for computers to display pixels on the screen is to have a video RAM, which is basically just a data buffer, available for programs or the operating system to write into.

Memory-Mapped I/O (3)

The issue thus arises of how the CPU communicates with the control registers and also with the device data buffers.

Two alternatives exist:

First Approach:

Each control register is assigned an **I/O port** number, an 8- or 16-bit integer.

The set of all the I/O ports form the **I/O port space**, which is protected so that ordinary user programs cannot access it (only the operating system can).

Memory-Mapped I/O (4)

First Approach

- Using a special I/O instruction, the CPU can **read** in control register PORT and store the result in CPU register REG:

```
IN REG,PORT,
```

- The CPU can **write** the contents of REG to a control register:

```
OUT PORT,REG
```

- In this scheme, the address spaces for memory and I/O are different, as shown in Fig. 5-2(a).

Memory-Mapped I/O (5)

First Approach

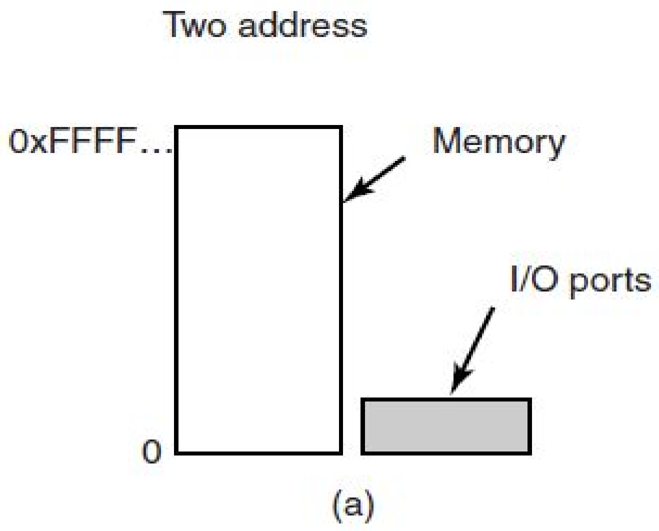


Figure 5-2. (a) Separate I/O and memory space.

Memory-Mapped I/O (6)

First Approach

- The instruction: IN R0,4

  reads the contents of I/O port 4 and puts it in R0.

- The instruction: MOV R0,4

  reads the contents of memory word 4 and puts it in R0. The 4s in these examples refer to different and unrelated address spaces.

- The 4s in these examples refer to different and unrelated address spaces.

Memory-Mapped I/O (7)

Second Approach

The second approach:

This approach maps all the control registers into the memory space, as shown in Fig. 5-2(b). Each control register is assigned a unique memory address to which no memory is assigned. This system is called **memory-mapped I/O**.

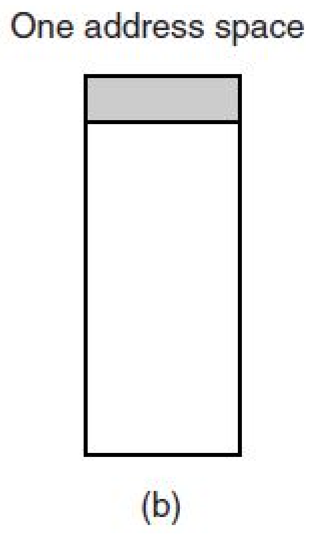


Figure 5-2. (b) Memory-mapped I/O.

Memory-Mapped I/O (8)

Second Approach

A hybrid scheme, with memory-mapped I/O data buffers and separate I/O ports for the control registers, is shown in Fig. 5-2 (c).

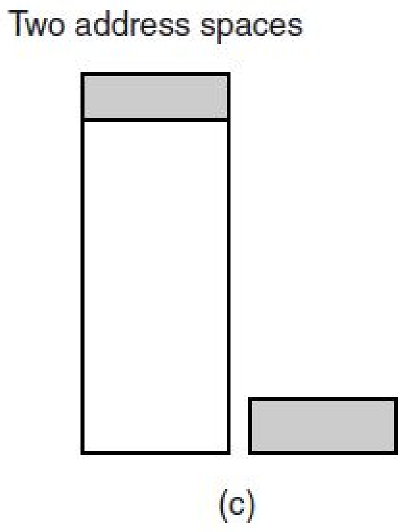


Figure 5-2. (c) Hybrid.

Memory-Mapped I/O (9)

How do these schemes actually work

- In all cases, when the CPU wants to read a word, either from memory or from an I/O port, it puts the address it needs on the bus’ address lines and then asserts a READ signal on a bus’ control line.

- A second signal line is used to tell whether I/O space or memory space is needed.

- If it is memory space, the memory responds to the request.

Memory-Mapped I/O (10)

How do these schemes actually work

•If it is I/O space, the I/O device responds to the request.

•If there is only memory space[as in Fig. 5-2(b)], every memory module and every I/O device compares the address lines to the range of addresses that it services.

•If the address falls in its range, it responds to the request.

•Since no address is ever assigned to both memory and an I/O device, there is no ambiguity and no conflict.

Memory-Mapped I/O (11)

Advantages

- If special I/O instructions are needed to read and write the device control registers, access to them requires the use of assembly code since there is no way to execute an IN or OUT instruction in C or C++. Calling such a procedure adds overhead to controlling I/O.

- With memory-mapped I/O, device control registers are just variables in memory and can be addressed in C the same way as any other variables.

Memory-Mapped I/O (12)

Advantages

- With memory-mapped I/O, an I/O device driver can be written entirely in C. Without memory-mapped I/O, some assembly code is needed.

- With memory-mapped I/O, no special protection mechanism is needed to keep user processes from performing I/O. All the operating system has to do is refrain from putting that portion of the address space containing the control registers in any user's virtual address space.

Memory-Mapped I/O (13)

Advantages

- If each device has its control registers on a different page of the address space, the operating system can give a user control over specific devices but not others by simply including the desired pages in its page table.

- With memory-mapped I/O, every instruction that can reference memory can also reference control registers.

Memory-Mapped I/O (14)

Advantages

For example,

If there is an instruction, TEST, that tests a memory word for 0, it can also be used to test a control register for 0, which might be the signal that the device is idle and can accept a new command.

```
LOOP: TEST PORT 4          // check if port 4 is 0
BEQ READY                  // if it is 0, go to ready
BRANCH LOOP                // otherwise, continue testing
READY:
```

Memory-Mapped I/O (15)

Disadvantages

Consider again the assembly-code loop:

```
LOOP: TEST PORT 4        // check if port 4 is 0
BEQ READY                // if it is 0, go to ready
BRANCH LOOP              // otherwise, continue testing
READY:
```

The first reference to PORT 4 would cause it to be cached. Subsequent references would just take the value from the cache and not even ask the device. Then when the device finally became ready, the software would have no way of finding out. Instead, the loop would go on forever.

Memory-Mapped I/O (16)

Disadvantages

To prevent this situation with memory-mapped I/O, the hardware has to be able to selectively disable caching, for example, on a per-page basis. This feature adds extra complexity to both the hardware and the operating system, which has to manage the selective caching.

Memory-Mapped I/O (17)

Disadvantages

Second, if there is only one address space, then all memory modules and all I/O devices must examine all memory references to see which ones to respond to. If the computer has a single bus, as in **Fig. 5-3(a),** having everyone look at every address is straightforward.

However, the trend in modern personal computers is to have a dedicated high speed memory bus, as shown in **Fig. 5-3(b).**

Memory-Mapped I/O (18)

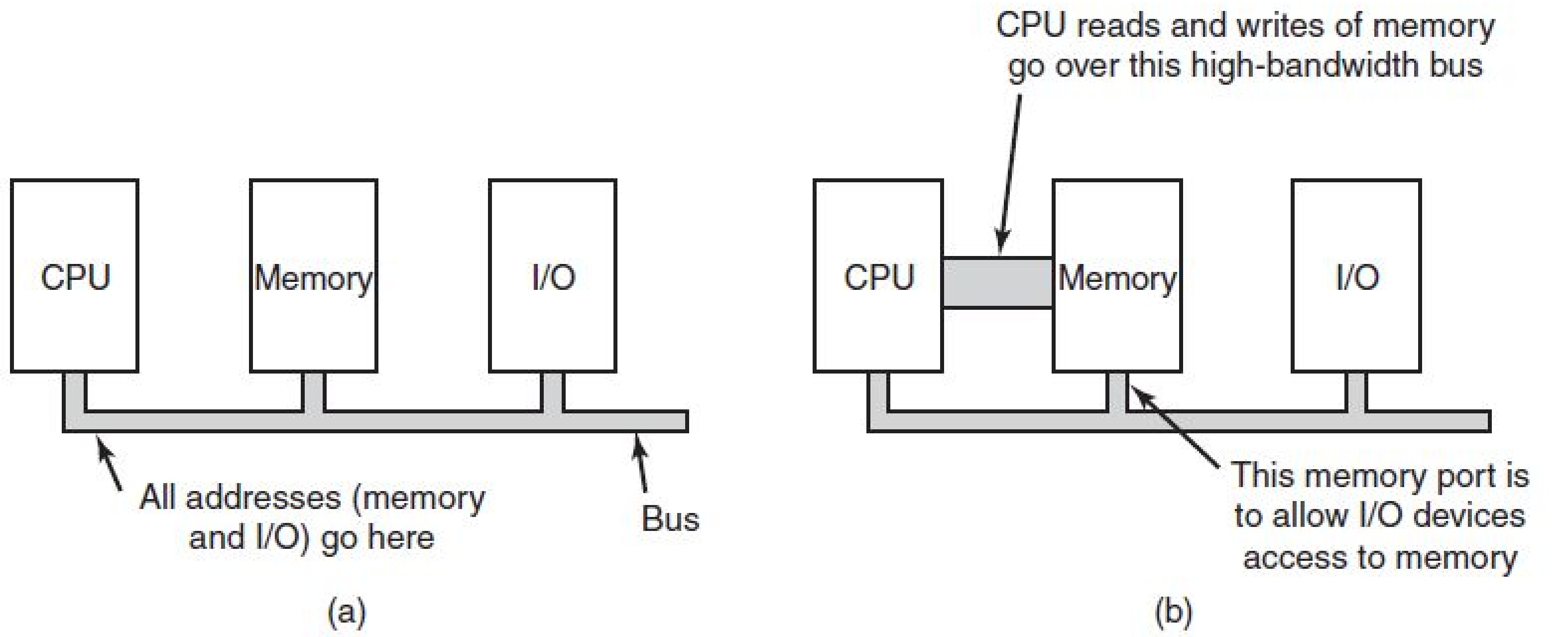

CPU
Memory
I/O
All addresses (memory and I/O) go here
Bus
(a)
CPU reads and writes of memory go over this high-bandwidth bus
CPU
Memory
I/O
This memory port is to allow I/O devices access to memory
(b)

Figure 5-3. (a) A single-bus architecture.
(b) A dual-bus memory architecture.

Direct Memory Access (1)

No matter whether a CPU does or does not have memory-mapped I/O, it needs to address the device controllers to exchange data with them.

The CPU can request data from an I/O controller one byte at a time, but doing so wastes the CPU’s time, so a different scheme, called **DMA** (**Direct Memory Access**) is often used.

we assume that the CPU accesses all devices and memory via a single system bus that connects the CPU, the memory, and the I/O devices, as shown in **Fig. 5-4**.

Direct Memory Access (2)

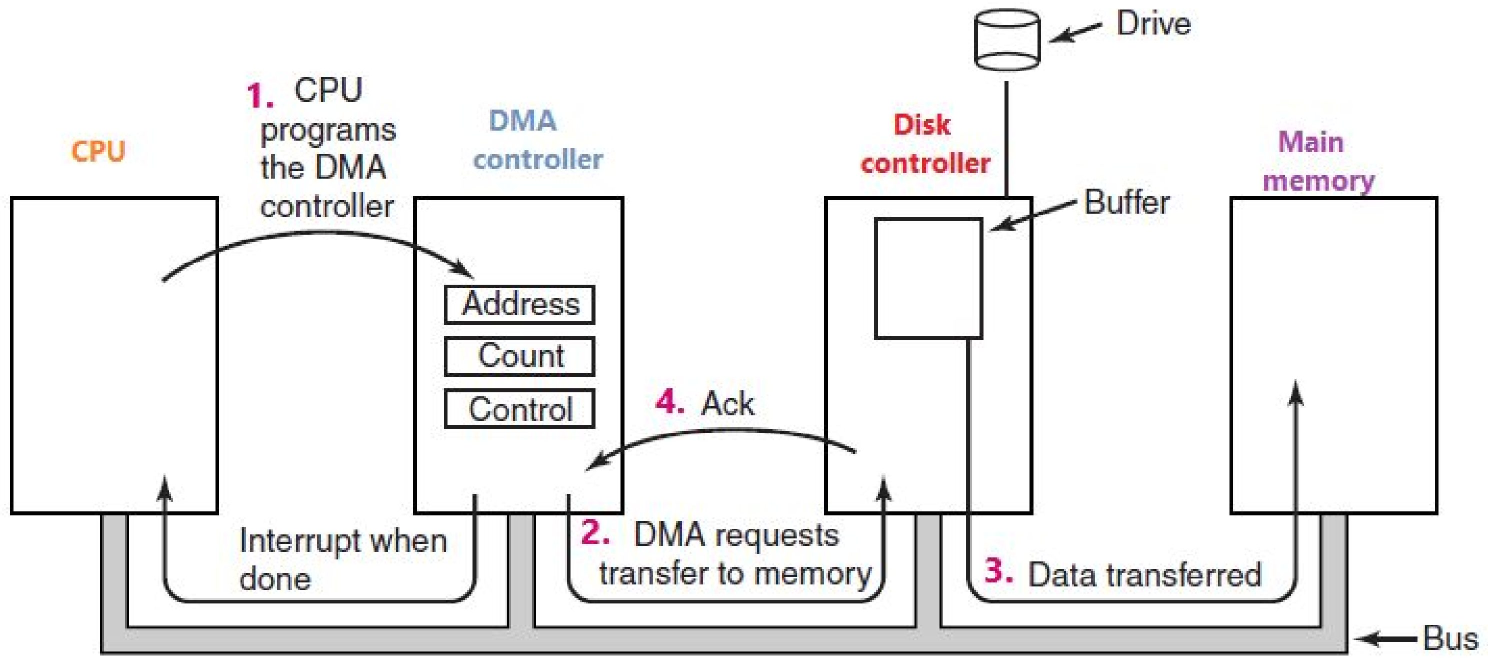


Figure 5-4. Operation of a DMA transfer.

Direct Memory Access (3)

- The DMA controller has access to the system bus independent of the CPU, as shown in **Fig. 5-4**.
- It contains several registers that can be written and read by the CPU.
- These include a memory address register, a byte count register, and one or more control registers.
- The control registers specify the I/O port to use, the direction of the transfer (reading from the I/O device or writing to the I/O device), the transfer unit (byte at a time or word at a time), and the number of bytes to transfer in one burst.

Direct Memory Access (4)

How Disk Read Occur when DMA is NOT used

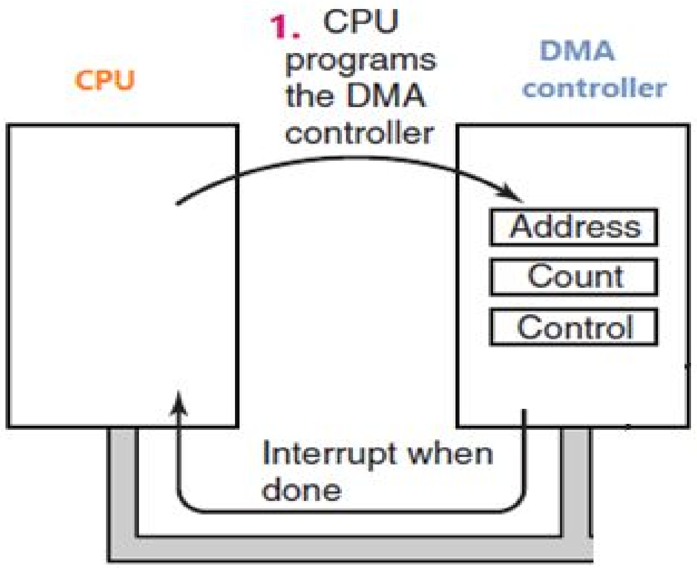
- First the disk controller reads the block (one or more sectors) from the drive serially, bit by bit, until the entire block is in the controller's internal Buffer.
- Next, it computes the checksum to verify that no read errors have occurred.
- When the OS starts running, it can read the disk block from the controller's buffer a byte or a word at a time by executing a loop, with each iteration reading one byte or word from a controller device register and storing it in main memory.

Direct Memory Access (5)

How Disk Read Occur with DMA

When DMA is used, the procedure is different.

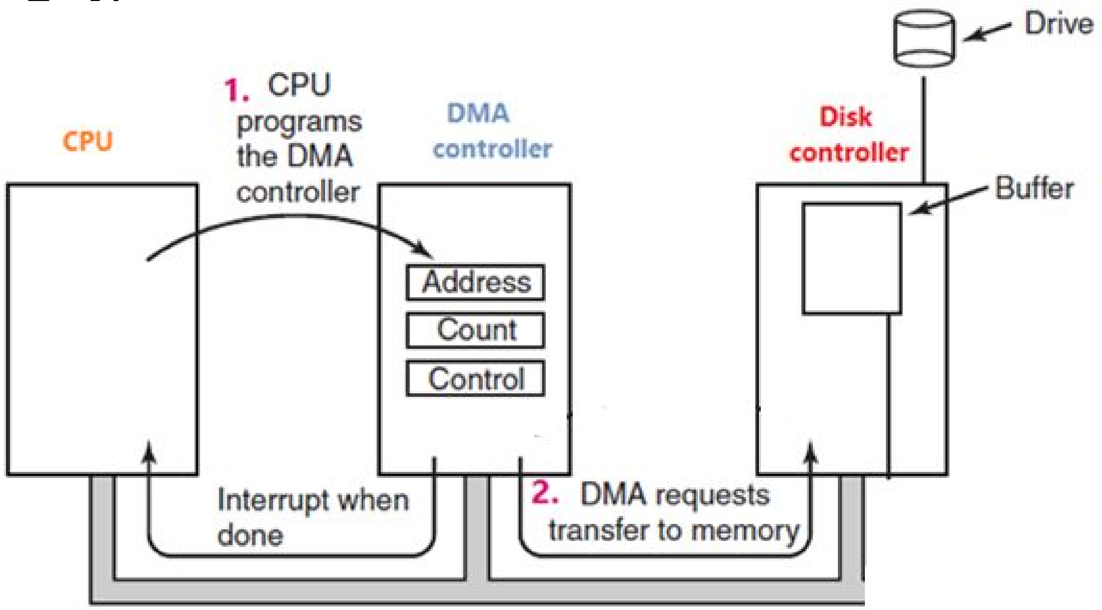
•First the CPU programs the DMA controller by setting its registers so it knows what to transfer where (**step 1** in Fig. 5-4).



Direct Memory Access (6)

How Disk Read Occur with DMA

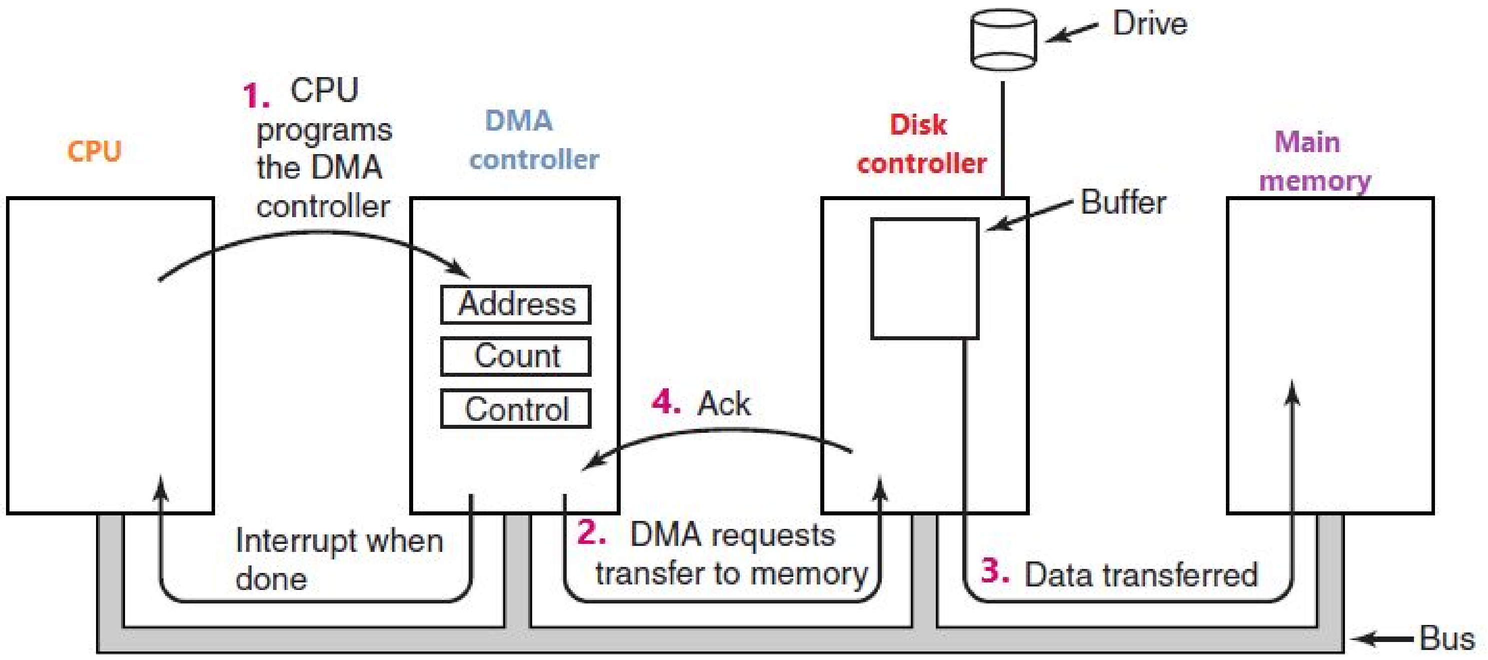
- It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum. When valid data are in the disk controller's buffer, DMA can begin. The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller (**step 2** in Fig



Direct Memory Access (7)

How Disk Read Occur with DMA

- The write to memory is another standard bus cycle **(step 3).** When the write is complete, the disk controller sends an acknowledgement signal to the DMA controller, also over the bu



Direct Memory Access (8)

How Disk Read Occur with DMA

- The DMA controller then increments the memory address to use and decrements the byte count.
- If the byte count is still greater than 0, steps 2 through 4 are repeated until the count reaches 0.
- At that time, the DMA controller interrupts the CPU to let it know that the transfer is now complete.
- When the operating system starts up, it does not have to copy the disk block to memory; it is already there.

Direct Memory Access (9)

Complex DMA Controllers

- DMA controllers vary considerably in their sophistication. More complex ones can be programmed to handle multiple transfers at the same time.

- Multiple requests to different device controllers may be pending at the same time, provided that there is an unambiguous way to tell the acknowledgements apart. Often a different acknowledgement line on the bus is used for each DMA channel for this reason.

Direct Memory Access (10)

Bus Modes

Word-at-a-time mode:

•The DMA controller requests the transfer of one word and gets it.

•If the CPU also wants the bus, it has to wait. The mechanism is called **cycle stealing** because the device controller sneaks in and steals an occasional bus cycle from the CPU once in a while, delaying it slightly.

Direct Memory Access (11)

Bus Modes

Block mode:

- The DMA controller tells the device to acquire the bus, issue a series of transfers, then release the bus.
- This form of operation is called **burst mode**.
- It is more efficient than cycle stealing because acquiring the bus takes time and multiple words can be transferred for the price of one bus acquisition.
- The down side to burst mode is that it can block the CPU and other devices for a substantial period if a long burst is being transferred.

Direct Memory Access (12)

Bus Modes

Fly-by mode:

- The DMA controller tells the device controller to transfer the data directly to main memory.
- An alternative mode that some DMA controllers use is to have the device controller send the word to the DMA controller, which then issues a second bus request to write the word to wherever it is supposed to go.
- This scheme requires an extra bus cycle per word transferred, but is more flexible in that it can also perform device-to-device copies and even memory-to-memory copies.

Direct Memory Access (13)

Transfer Using Physical Memory

Most DMA controllers use physical memory addresses for their transfers.

- Using physical addresses requires the operating system to convert the virtual address of the intended memory buffer into a physical address and write this physical address into the DMA controller's address register.
- An alternative scheme used in a few DMA controllers is to write virtual addresses into the DMA controller instead.
- Then the DMA controller must use the MMU to have the virtual-to-physical translation done.

Interrupts Revisited (1)

In a typical personal computer system, the interrupt structure is as shown in **Fig. 5-5.**

At the hardware level, interrupts work as follows:

- When an I/O device has finished the work given to it, it causes an interrupt (assuming that interrupts have been enabled by the operating system). It does this by asserting a signal on a bus line that it has been assigned.

- This signal is detected by the interrupt controller chip on the parentboard, which then decides what to do.

Interrupts Revisited (2)

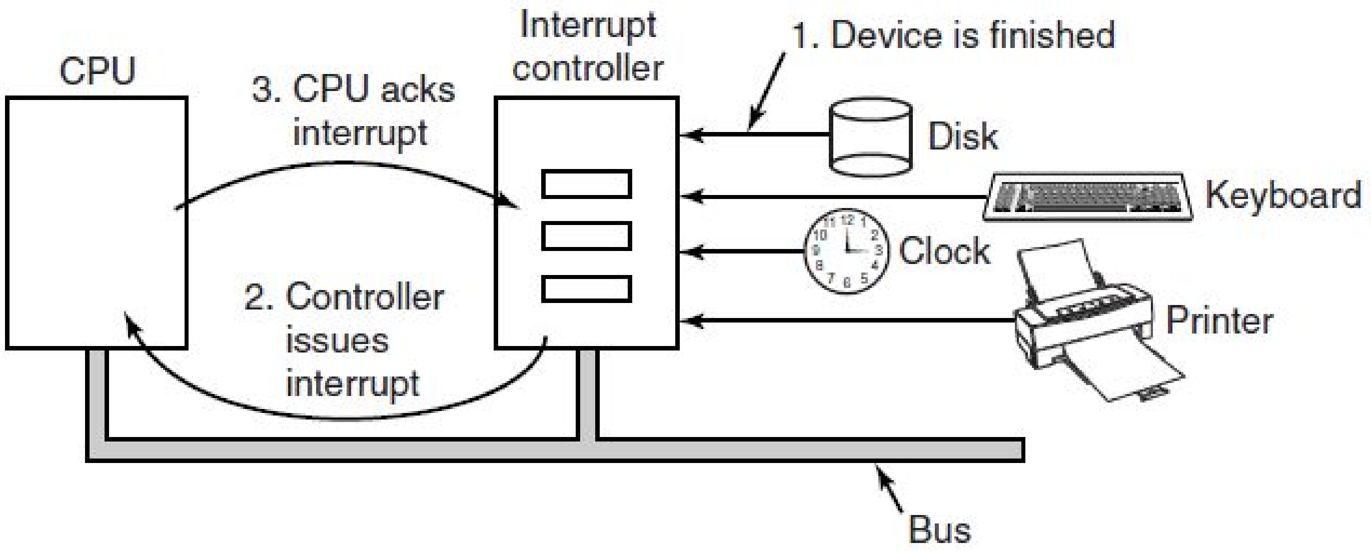


Figure 5-5. How an interrupt happens. The connections between the devices and the interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

Interrupts Revisited (3)

Interrupt Handling

- To handle the interrupt, the controller puts a number on the address lines specifying which device wants attention and asserts a signal to interrupt the CPU.
- The interrupt signal causes the CPU to stop what it is doing and start doing something else.
- The number on the address lines is used as an index into a table called the **interrupt vector** to fetch a new program counter.
- The number on the address lines is used as an index into a table called the **interrupt vector** to fetch a new program counter. This program counterpoints to the start of the corresponding interrupt-service procedure.

Precise Interrupt (1)

An interrupt that leaves the machine in a well-defined state is called a precise interrupt.

Precise Interrupt (2)

Four properties of a precise interrupt:

1. The PC is saved in a known place.
2. All instructions before the one pointed to by the PC have completed.
3. No instruction beyond the one pointed to by PC has finished.
4. The Execution state of the instruction pointed to by the PC is known.

Imprecise Interrupt

An interrupt that does not meet the requirements such as all instructions up to the program counter have completed and none of those beyond it have started, is called an imprecise interrupt.

Precise vs. Imprecise (1)

The situation of **Fig. 5-6(a)** illustrates a precise interrupt. All instructions up to the program counter (316) have completed and none of those beyond it have started.

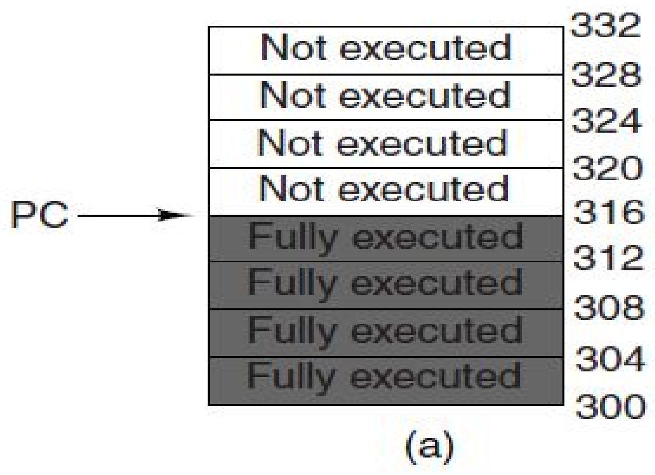


Figure 5-6. (a) A precise interrupt.

Precise vs. Imprecise (2)

Fig. 5-6(b) illustrates an imprecise interrupt, where different instructions near the program counter are in different stages of completion.

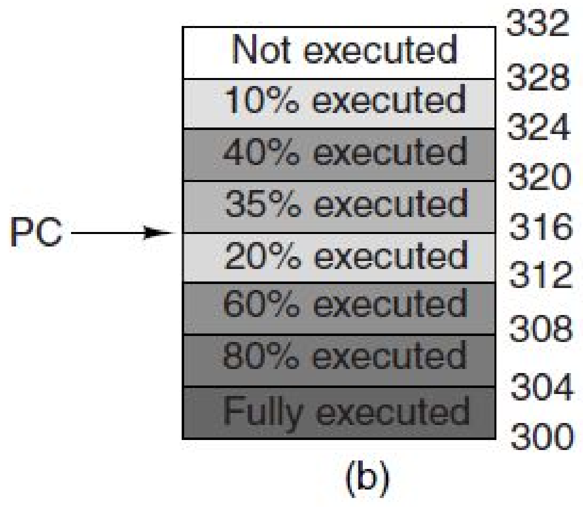


Figure 5-6. (b) An imprecise interrupt.

Goals of the I/O Software (1)

- Device independence
- Uniform naming
- Error handling
- Synchronous versus asynchronous
- Buffering.

Goals of the I/O Software (2)

Device Independence

we should be able to write programs that can access any I/O device without having to specify the device in advance. Such as:

- A program that reads a file as input should be able to read a file on a hard disk, a DVD, or on a USB stick without having to be modified for each different device.
- Similarly, one should be able to type a command such as

sort <input >output

and have it work with input coming from any kind of disk or keyboard and the output going to any kind of disk or the screen.

Goals of the I/O Software (3)

Uniform Naming

The name of a file or a device should simply be a string or an integer and not depend on the device in any way.

Example: A USB stick can be **mounted** on top of the directory:

/usr/ast/backup

So that copying a file to:

/usr/ast/backup/monday

copies the file to the USB stick. In this way, all files and devices are addressed the same way such as by a path name.

Goals of the I/O Software (4)

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 itself if it can. If it cannot, then the device driver should handle it, perhaps by just trying to read the block again.

•In many cases, error recovery can be done transparently at a low level without the upper levels even knowing about the error.

Goals of the I/O Software (5)

Asynchronous vs. Synchronous Transfers

Asynchronous (Interrupt Driven):

- Most physical I/O is asynchronous.
- The CPU starts the transfer and goes off to do something else until the interrupt arrives.

Synchronous (Blocking):

- User programs are much easier to write if the I/O operations are blocking.
- After a read system call the program is automatically suspended until the data are available in the buffer.

Goals of the I/O Software (6)

Buffering

- Often data that come off a device cannot be stored directly in their final destination. **For Example:**

- when a packet comes in off the network, the operating system does not know where to put it until it has stored the packet somewhere and examined it.

- Buffering involves considerable copying and often has a major impact on I/O performance

END

Week 10 – Lecture 1

References

- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013

  Prentice-Hall, Inc.
- http://rodrev.com/graphical/programming/Deadlock.swf
- http://www.utdallas.edu/~ilyen/animation/cpu/program/prog.html
- https://courses.cs.vt.edu/csonline/OS/Lessons/Processes/index.html
- http://inventwithpython.com/blog/2013/04/22/multithreaded-python-tutorial-with threadworms/