Chapter 7

INPUT-OUTPUT

Table of contents:

7.1	DEVICE CONTROLLERS	7-1
7.2	BASIC APPROACHES	7-2
7.3	DIRECT MEMORY ACCESS	7-3
7.4	I/O BUFFERING	7-5
7.5	DISK CACHE	7-7
7.6	SYNCHRONOUS VS. ASYNCHRONOUS I/O	7-7
7.7	I/O ARCHITECTURE	7-8
7.8	DISK DRIVE AND CONTROLLER	7-9
7.8.1	Disk Geometry	7-9
7.8.2	Structure of a Sector	7-10
7.8.3	Physical and Logical Sector Addresses	7-11
7.8.4	Logical Block Number	7-12
7.8.5	Disk Scheduling	7-13

Copyright © 1998-2002 by Marko Vuskovic

DEVICE CONTROLLERS

Device controllers are hardware units that interface peripheral device(s) with the rest of the computing system (usually via the system bus). Device controllers can be physically part of the device, or a separate unit (device interface card). Normally, a device controller can monitor several devices.

Device controllers accept commands and data from CPU, and monitor data transfer from/to device. The state of the device is reported by status register. The data sent to device, or received from device are latched into data registers. Some controllers, that perform input and output can have two data registers (input and output data register), or can share only one register. The data registers are sometimes called data buffers. Commands from CPU are latched in control, or command register. Most of the device controllers generate an interrupt signal after the data transfer between the device and the controller is completed.

CPU has access to device controller registers. Usually the parts of the program which access device controller's registers are grouped into a software module called device driver.

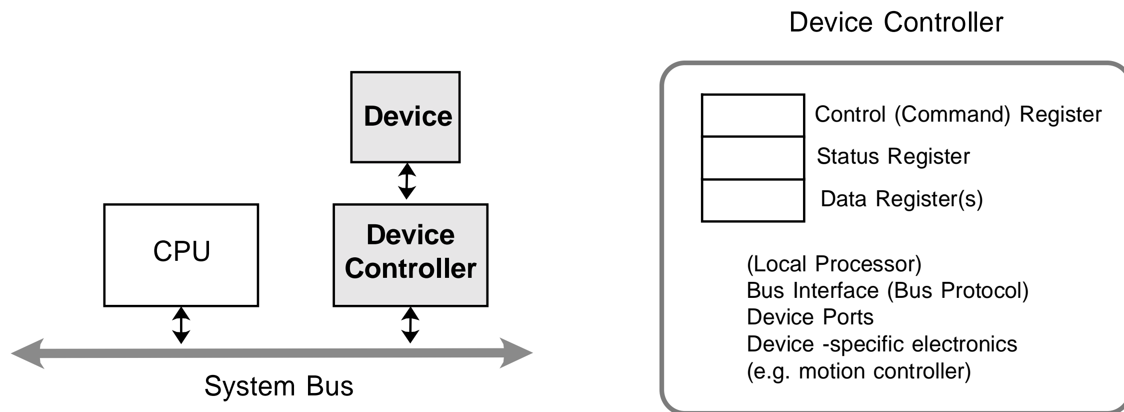
There are two general styles to access device controller registers: via special machine instructions (INP, OUT), or via reading/writing from/to special memory locations. The latter style is called memory-mapped I/O. In memory--mapped I/O the controller registers are mapped into the systems's address space. These locations are absolute physical locations which are located in protected area (system space), and can be accessed only in supervisory mode.

There are basically two types of devices:

Character-oriented devices:
keyboards, mice, modems, printers, plotters, monitors, joysticks

Block-oriented devices:
hard disk, floppy disk, tape, CD-ROM, network (Ethernet, Token Ring, ATM)

Character-oriented devices transfer a byte of data at each I/O request, while the block-oriented devices transfer a block (or packet) of data. Blocks for disks have usually 512 byte, while network packets can have sizes between 64 and 1518 bytes (Ethernet).



Copyright © 1998-2002 by Marko Vuskovic

BASIC APPROACHES

There are three basic approaches in I/O:

Programmed I/O

(Also called direct I/O, polled I/O)
P issues an I/O request, than enters a loop in which periodically interrogates the device controller's status register to see whether the data transfer is completed. This form of waiting is called: busy waiting.

Simple, inefficient (waste of the CPU time, OK in single-user system, unacceptable in multiprogrammed systems.)

Interrupt-Driven I/O

(Also called overlapped I/O. WARNING: in NT terminology overlapped I/O means asynchronous I/O)
P issues an I/O request and gets blocked by the OS. Another P can now use CPU. After the I/O transfer (between the device and the device controller) is completed, device controller issues an interrupt to CPU. An interrupt handler (Interrupt Service Routine - ISR) will move data to the user's area and will deblock the P which has issued the I/O request.

Essentially less efficient than programmed I/O since each block or character transfer causes an interrupt and possibly a process/thread switch.
However the CPU is better utilized, a must in multiprogrammed systems.

Direct Memory Access (DMA)

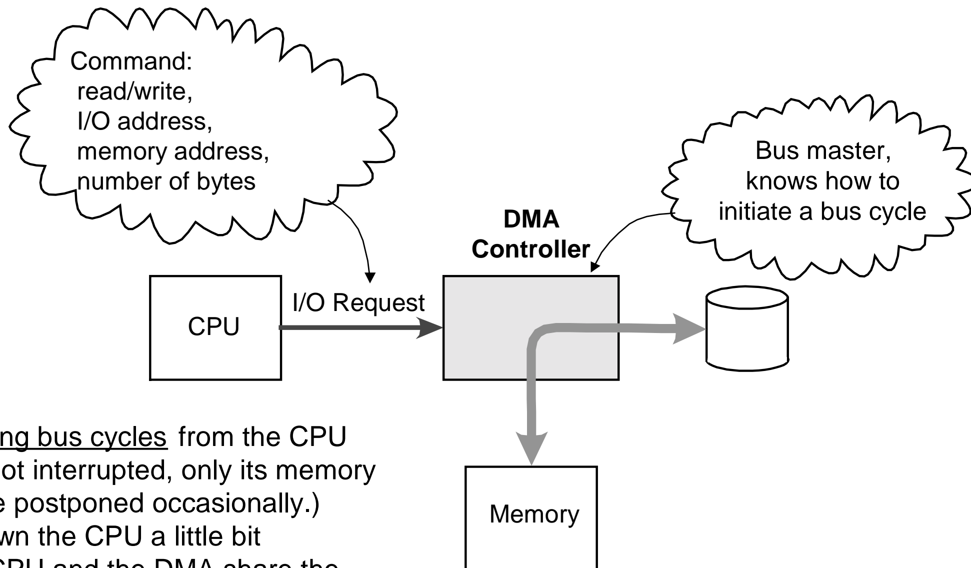
(Also called autonomous I/O, or intelligent I/O)
Device controller has its own (degenerate) processor which is capable to be a bus master, to arbitrate the bus with the CPU and to perform data transfer between device controller and memory. This relieves the CPU from data transfer (CPU needs only to issue the I/O request and delegate the rest of the job to DMA.)

Very efficient, specially in longer data transfers. An I/O request can be formulated as multi character or multi block transfer, which will prevent interrupts after each character or block. (See details later.)

A generalization of DMA is I/O Processor, or I/O Channel. The Device controller employs a special-purpose processor (microcontroller) which has more functionality than DMA. I/O request can be at higher level (including error handling, mapping of logical to physical blocks, scheduling of multiple disk I/O requests, and some functions of the file system.)

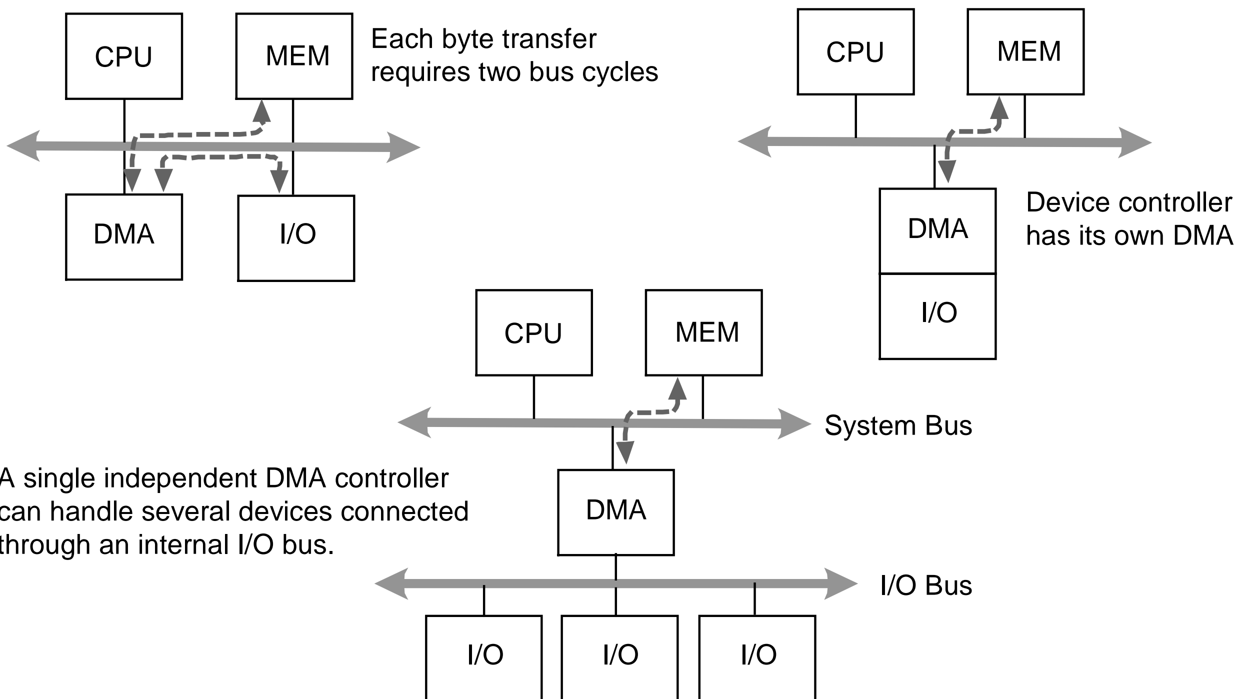
Copyright © 1998-2002 by Marko Vuskovic

DIRECT MEMORY ACCESS



DMA is stealling bus cycles from the CPU
(The CPU is not interrupted, only its memory references are postponed occasionally.)
This slows down the CPU a little bit because the CPU and the DMA share the same bus, and normally DMA has higher priority over the CPU.

DMA controller can be part of the device controller, or can be independent and used with any device that supports DMA. Some topologies require stilling two bus cycles instead of one.



Copyright © 1998-2002 by Marko Vuskovic

DIRECT MEMORY ACCESS (Cont.)

DMA and Virtual Memory

What memory address should use DMA, logical address (LA) or physical address (PA)?

LA - Needs address translation
PA - What if block of data crosses a page boundary?
(frames are not consecutive in general case)

Solutions:

(A) DMA works with LA. It has its own page registers (a small number of page table entries, like TLB). The address translation is done by the CPU at the I/O request.

(B) DMA works with PA. Block transfer is broken into several parts, each within a page boundary. The transfers are chained together and handled by the DMA. If DMA does not support chaining, the OS can request each transfer individually.

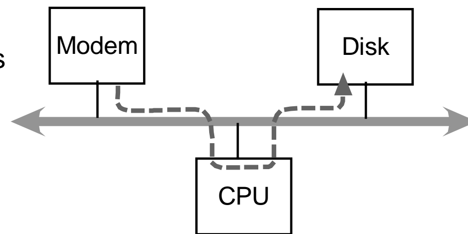
Copyright © 1998-2002 by Marko Vuskovic

I/O BUFFERING

Buffering is a common techniques to improve efficiency of the system in data transfers which involve several I/O devices with different speeds and different transfer data sizes (bytes, blocks).

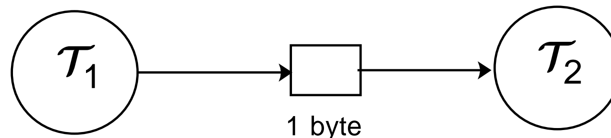
Suppose a file is received via modem (FTP for example). This involves two devices: a character device (modem) and a block-oriented device (disk), and suppose that two threads, $\mathcal{T}_1$ and $\mathcal{T}_2$ are carrying the job.

Modem: 1 KB/sec
File: 5 KB = 5120 bytes



Disk: 10000 KB/sec
File: 10 blocks (sectors)

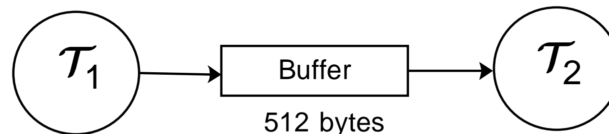
Unbuffered I/O



5120 I/O requests (made by $\mathcal{T}_1$)
5120 thread switches ($\mathcal{T}_1$ -> $\mathcal{T}_x$ -> $\mathcal{T}_1$)
5120 interrupts

5120 I/O requests (made by $\mathcal{T}_2$)
5120 thread switches ($\mathcal{T}_2$ -> $\mathcal{T}_x$ -> $\mathcal{T}_2$)
5120 interrupts

Single Buffered I/O

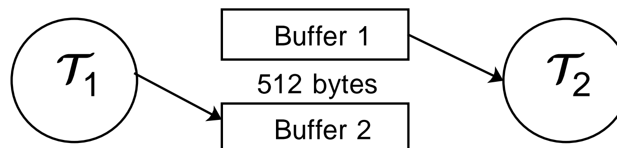


10 I/O requests (made by $\mathcal{T}_1$)
10 thread switches ($\mathcal{T}_1$ -> $\mathcal{T}_x$ -> $\mathcal{T}_1$)
5120 interrupts

10 I/O requests (made by $\mathcal{T}_2$)
10 thread switches ($\mathcal{T}_2$ -> $\mathcal{T}_x$ -> $\mathcal{T}_2$)
10 interrupts

Once the buffer is filled, $\mathcal{T}_1$ must wait until $\mathcal{T}_2$ empties the buffer.

Double Buffered I/O

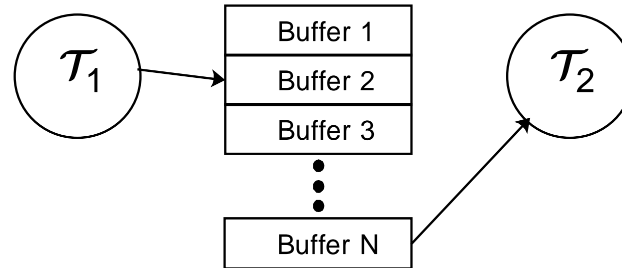


About the same ammount of thread switches and interrupts,
only $\mathcal{T}_1$ does not have to wait. $\mathcal{T}_2$ empties the buffer 1 while $\mathcal{T}_1$ is filling the buffer 2.
Buffers are switched subsequently.

Copyright © 1998-2002 by Marko Vuskovic

I/O BUFFERING (Cont.)

Circular Buffering



Circular buffers (multi-slot buffers, bounded buffers) are generalization of double buffers. They can improve independency of two threads. Controlling the buffers requires thread synchronization (see chapter "Synchronization")

Buffers are allocated in the system space and they are completely controlled by the OS (they are transparent to the user).

The scheme above can be applied to a single device. For example, instead of transferring data to a file on disk, they can be transferred to an array in memory. The buffering is specially helpful in networking where we deal with messages of different sizes which have to be assembled and disassembled. Buffers can be used to help this.

NOTICE : Buffering is a techniques that smooths out peaks in I/O demand. However no amount of buffering will allow an I/O device to keep pace with a thread indefinitely when the average demand of the thread is greater than the I/O device can service. Even with multiple buffers, all of the buffers will eventually fill up and the thread will have to wait after processing each chunk of data. In multiprogramming environment buffering is a tool that can increase the efficiency of the OS and the performance of individual threads. [Stallings, OS, 1998]

Buffering also helps virtual memory. Suppose we transfer data from the modem to a location in user space The target page must not be out-swapped, otherwise data can be lost. Transferring data from device to a buffer in system space (which is memory resident all the time) will alleviate the problem.

Buffering also supports copy semantics. Suppose an application has filled a buffer ready to send to a file on disk. For that the write() system call is used and a pointer to buffer is passed. While system is performing disk write (using the pointer), application wants to change the buffer. The integrity of data sent to disk will be thus compromised. A clean way is that the system service write() moves data to be sent from user's space to a buffer in system space and to perform data transfer from there. The application is then free to modify the user's buffer as soon as the system call returns.

Copyright © 1998-2002 by Marko Vuskovic

DISK CACHE

A copy of some disk blocks can be kept in primary memory. When a file I/O is performed, I/O system checks first if the requested sector is in memory. If yes, it passes a pointer to memory, so no I/O transfer is necessary. If not, the referenced sector is brought into memor and further on referenced from there. This is justified by locality of references.

If disk cache is limited in number of blocks and when a new block has to be brought into memory, an old block has to be replaced. Popular replacement strategies are:

LRU (Least recently used - with stack implementation)
LFU (Least frequently used - counter used with each block)

Windows NT uses technique called mapped file I/O which works jointly between I/O system and VMM (virtual memory manager). It is used automatically by the OS for file caching and for image activation (loading and running executable images). The mapped file I/O is also available as an explicit system service.

In mapped file I/O program accesses the file as a large array in memory. The VM uses the paging mechanism to load the correct page from the file if it is not present yet. If the array is modified the VM takes care of rewriting the modified page back into file. This technique improves the efficiency considerably at multiple file accesses.

SYNCHRONOUS vs. ASYNCHRONOUS I/O

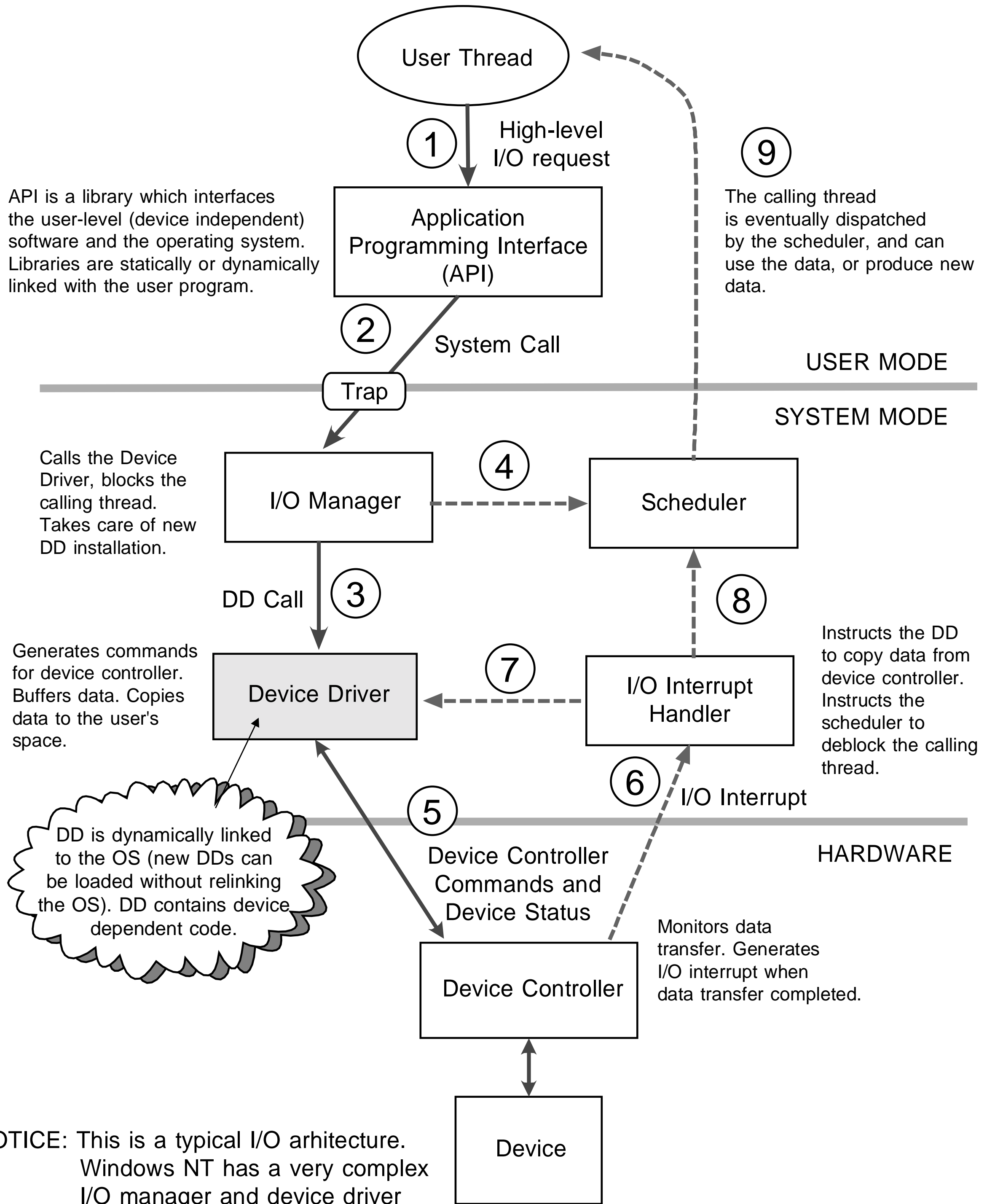
Synchronous I/O (also called: blocking I/O) is the most common form of I/O. The application issues an I/O request by calling a system service and gets blocked. When the data transfer is completed an I/O interrupt causes the blocked application to be ready and subsequently running.

The asynchronous I/O (non-blocking I/O) doesn't block the calling application. After issuing the I/O request, the application can continue to execute while the I/O transfer progresses. When I/O is completed, the application can be notified somehow. The application must synchronize its execution with the I/O and not to access data before the I/O transfer is completed.

Asynchronous I/O is extensively used in Windows NT (called: overlapped I/O in Win32 terminology).

Copyright © 1998-2002 by Marko Vuskovic

I/O ARCHITECTURE

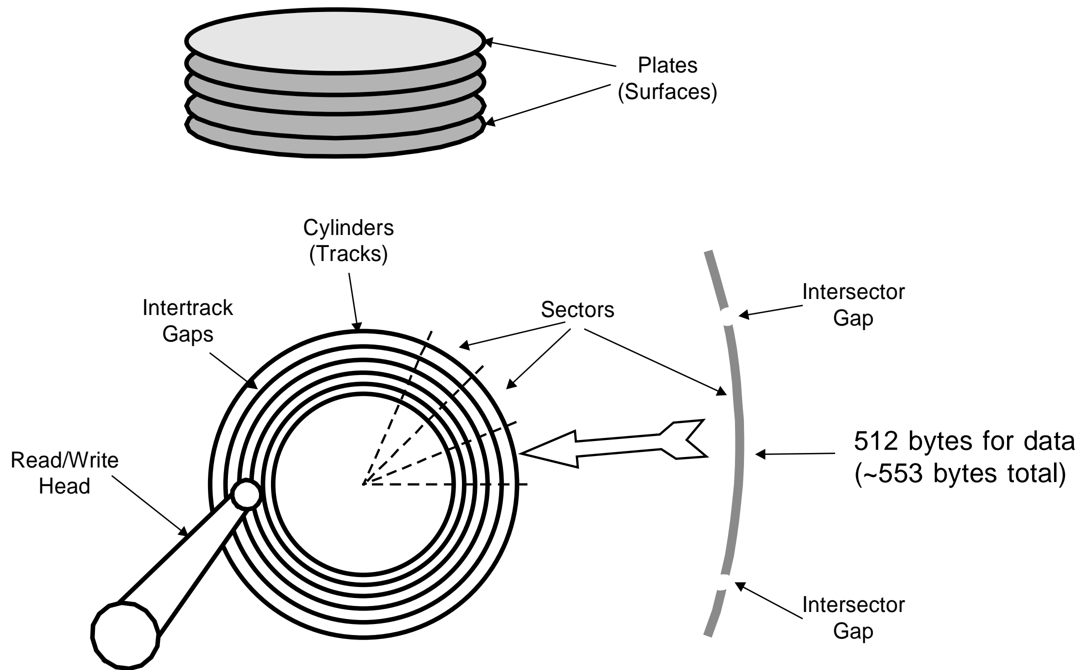


NOTICE: This is a typical I/O arhitecture. Windows NT has a very complex I/O manager and device driver (which will be discussed in case study).

Copyright © 1998-2002 by Marko Vuskovic

DISK DRIVE AND CONTROLLER

Disks are the most important peripheral components from the OS point of view (Files/file system, backing store/virtual memory)



Average Rotational Delay: $T_R = 0.5 * 60/V;$

Transfer Time: $T_T = 60*b/(VN);$

Average Access Time: $T_A = T_s + T_c + T_R + T_T;$

V - spindle speed [RPM]
b - number of bytes to transfer
N - number of bytes per track
T_s - Average seek time [sec]
T_c - Controller overhead [sec]

Example (MAXTOR, DiamondMax 2880):

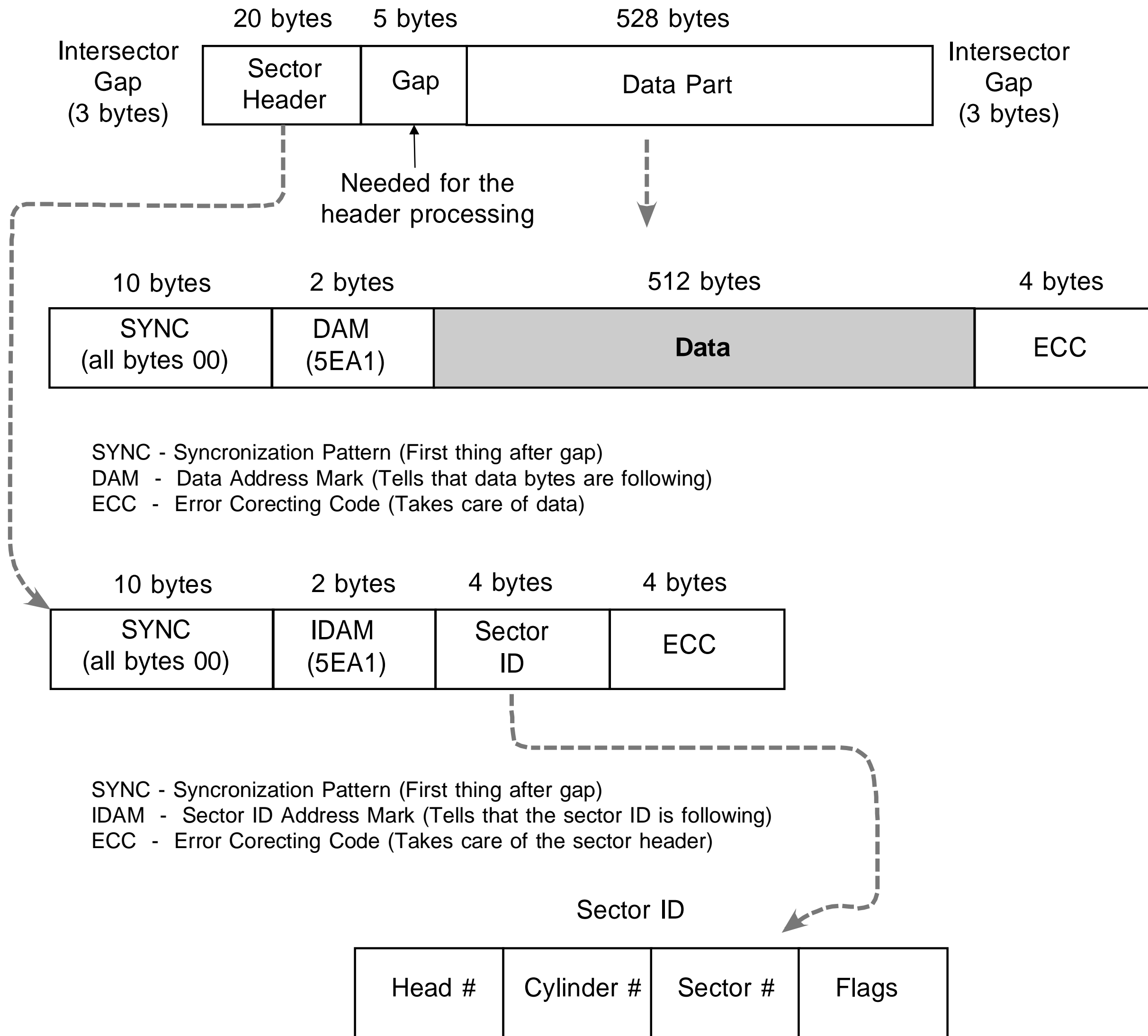
Formatted capacity:	4.3 GB
Plates:	3
Cylinders:	10,022
Sectors:	200-330
Bytes per sector:	512
Seek Times:	1 ms (track-to-track), 9 ms (average)
Spindle Speed:	5,400 RPM
Controller Overhead:	0.5 ms
Data Transfer Rate:	33 MB/sec (to/from controller), 15.2 MB/sec (to/from drive)

Copyright © 1998-2002 by Marko Vuskovic

DISK DRIVE/CONTROLLER (Cont.)

Structure of a Sector

(Example of IBM formatting. Floopy disk and hard disks have similar format)



Flags:
- bit 0: Bad sector
- bit 1: Bad track without reassignment
- bit 2: Track ID points to alternative track
- bit 3: Alternative track
- bit: 4-7: Reserved

Controller electronics is checking validity of data (and possibly correcting the data) using ECC. Bad sectors/tracks are either skipped or reassigned.

Copyright © 1998-2002 by Marko Vuskovic

DISK DRIVE/CONTROLLER (Cont.)

Physical and Logical Sector Address

Sector is a minimal addressable unit on a disk (like byte in internal memory).

Physical sector address for a given drive reflects the disk geometry:

<Head #, Cylinder #, Sector #>

NOTICE: Strictly speaking the physical sector address is not an actual address of a sector. For example sectors can be reassigned if the original sector is bad.

Logical sector address is a linear mapping of the physical sector address onto the range of integers 0..N-1 (N is total number of sectors of a drive). Logical sector addresses simplify addressing from the OS point of view (see chapter File System).

Example (high density, double sided floppy disk):

Head #	Cylinder #	Sector #	Logical sector #
0	0	1	0
....	....	....	....
0	0	18	17
1	0	1	18
....	....	....	....
1	0	18	35
0	1	1	36
....	....	....	....
0	1	18	53
....	....	....	....
1	79	18	2879

The drive has two sides (two heads), 80 tracks (cylinders) per side, 18 sectors per track, and 512 data bytes per sector. Total capacity of the formatted disk is: 2*80*18*512 = 1474560 bytes = 1440 KB

Similar mapping exist for hard drives. Only it is more complicated since newer hard drives have a variable number of sectors per track depending whether it is an inner or an outer track (zoning).

Copyright © 1998-2002 by Marko Vuskovic

DISK DRIVE/CONTROLLER (Cont.)

Logical Block Number

Data transfer is more efficient if it extends to a larger number of consecutive bytes per I/O request. This will minimize delays due to seek time and rotational latency. Transfer of larger blocks of consecutive data also reduces number of I/O interrupts. On the other hand, larger blocks increase the internal fragmentation of the disk space. For example, suppose a block size of 32 KB and a small file of less than 1 KB. This would waste a space of 31 KB, which is about 97%. Systems usually have many small 1 KB files.

The choice of the block size is a matter of compromize between the transfer rate and the space efficiency - problem similar to the choice of the page size in memory management.

Since the minimal addressable unit of a disk is sector (512 bytes), a block of disk data contains a group (<u>cluster</u>) of one or more consecutive disk sectors. The choice of the block size is primarily decided by the operating system. In most cases the block size can be configured for a given system. Here are typical block sizes for some systems:

SGI (BSD UNIX) - 512 bytes (1 disk sector)
HPUX - 1 KB (2 disk sectors)
Windows NT - 4 KB (8 disk sectors)
Solaris 2 - 8 KB (16 disk sectors)

Addressing of the disk space as viewed by the file system is done by block numbers - <u>Logical Block Number (LBN)</u>.

Copyright © 1998-2002 by Marko Vuskovic

DISK DRIVE/CONTROLLER (Cont.)

Disk Scheduling

Disk DD (device driver) gets I/O requests from the I/O manager.
In case of a multiprogrammed system, there could be several I/O requests queued in the disk DD. Because each I/O request results in a head movement which takes a considerable amount of time (seek time, about 1 ms for cylinder-to-cylinder movement - enough time to execute about 10,000 machine instructions) it is important to schedule the head movements in a most efficient way. For example, if there are two I/O requests: one at cylinder #330, and another at cylinder #10, and the current position of the head is at cylinder #5, it is obviously more efficient if the driver schedules the second request first, then the first request. However, this may change if there are other additional requests.

NOTICE: Disk DD can sometimes delegate the task of disk scheduling to the disk controller, if the latter has enough intelligence.

There are available several disk scheduling policies. They will be examined for the same sequence of disk I/O requests (the numbers are the corresponding cylinder numbers, suppose 200 total of cylinders and suppose that the head is initially at cylinder 100):

55, 58, 39, 18, 90, 160, 150, 38, 184. [Stallings, OS, Prentice Hall, 1998]

FCFS (First Come First Served)

Requests served: 55, 58, 39, 18, 90, 160, 150, 38, 184
Number of cylinders traversed: 45, 3, 19, 21, 72, 70, 10, 112, 146
Average seek length: 55.3

Simple, fair but very inefficient - too much head movements.

SSTF (Shortest Seek Time First)

Requests served: 90, 58, 55, 39, 38, 18, 150, 160, 184
Cylinders traversed: 10, 32, 3, 16, 1, 20, 132, 10, 24
Average seek length: 27.5

If head is at cylinder 100, then the closest request is at cylinder 90, from there the closest request is at 58, then at 55, etc.
Reduces the seek time. Not necessarily optimal.
Like SJF (in thread scheduling) can cause starvation
(imagine a person waiting for elevator and wants to go to the 88-th floor while there is a crowd of people going between the floors 3 to 20 to lobby.)

Copyright © 1998-2002 by Marko Vuskovic

DISK DRIVE/CONTROLLER (Cont.)

SCAN (Elevator algorithm)

Request sequence: 55, 58, 39, 18, 90, 160, 150, 38, 184
Requests served: 150,160, 184, 90, 58, 55, 39, 38, 18
Cylinders traversed: 50, 10, 24, 94, 32, 3, 16, 1, 20
Average seek length: 27.8

Head is moving in one direction and serves all requests in that direction. When it reaches the last request it reverses the direction of movement and serves all request in the reversed direction until it reaches the last request there, then reverses the movement again... In the example above the initial movement of the head is towards higher cylinder numbers starting from cylinder 100.

No starvation, retains the property of SSTF.
Problem: suppose the head reached the right end and reverses the motion.
Suppose there is only one request in the new direction, which is far at the left side.
At that moment several requests come in, which are all right to from the head - they have all to wait for relatively long time.

C-SCAN Algorithm (Cyclic scan)

Solves the problem of the SCAN algorithm and provides more uniform wait times.
Always goes in one direction: after it reaches the last request in one direction immediately returns to the first request at the opposite end (scans a circular list of cylinders.)

Request sequence: 55, 58, 39, 18, 90, 160, 150, 38, 184
Requests served: 150, 160, 184, 18, 38, 39, 55, 58, 90
Cylinders traversed: 50, 10, 24, 166, 20, 1, 16, 3, 32
Average seek length: 35.8

Copyright © 1998-2002 by Marko Vuskovic