Managing Secondary Storage

Things we've seen:

- why we need secondary storage
- that secondary storage is, when used unintelligently, too slow
- the physical characteristics of secondary storage devices and the performance limits these characteristics impose
- that the limits are magically unnoticeable when we `read` from or `write` to files inside our programs

Demystifying what goes on in the dark area between application program and device:

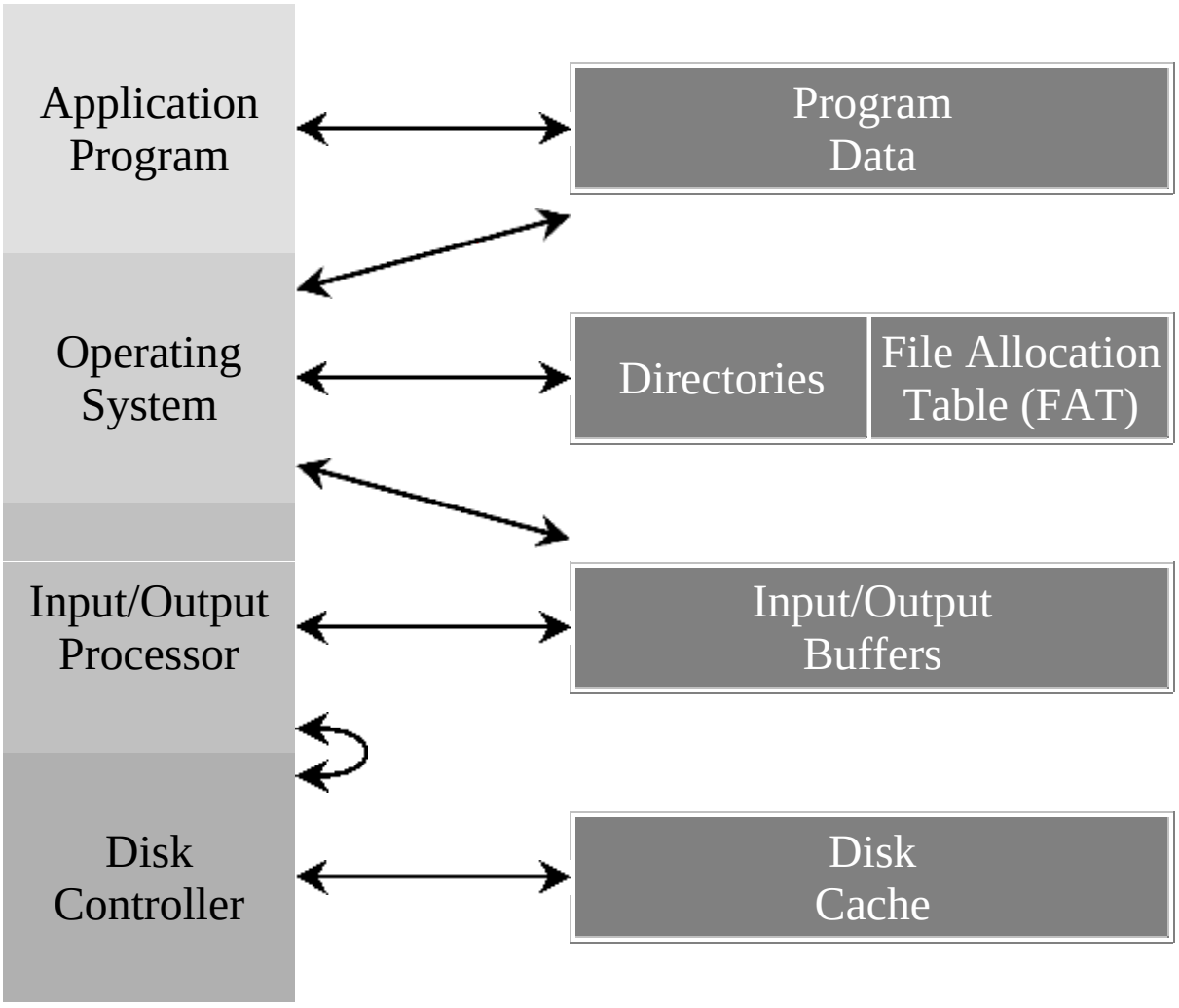
Topic	***Folk & Zoellick***
• "Journey of a Byte"	§ 3.5
• Directory structure/The FAT	§ 3.1.3, 3.7.1, 3.7.2
• Buffering (part I)	§ 3.6

N.B. *What we'll see is simplified, and not necessarily exactly how any one file system is organized, but the concepts are gold!*

Software Architecture

What are the different pieces of software involved in accomplishing a particular task?

What data do the different pieces of software access for the purposes of file management?



Application Program

Here is a fragment of a C program that assigns a value to a 1-byte variable and writes the value out at the end of a file.

C program

```
char ch;
FILE *ofil;

...
ofil = fopen("foo.bar", "a");
ch = 75;
fwrite(&ch, 1, 1, ofil);
...
```

Program Data

Assuming the address of the variable `ch` is `0x0092` (speaking in C, `&ch == 0x92`), here is a peek at part of RAM used for the data for our little C program after the assignment statment `ch = 75;`.

Program data

...							...
0x0080							...
0x0090			75				...
0x00A0							...
0x00B0							...
...							...

Opening a File

The statement `ofil = fopen("foo.bar", "a");` in our little C program gets translated to a request to the operating system to open the file `foo.bar` for writing (at eof). The operating system must do several things to open the file:

1. Find the named file in the directory
2. Create an entry in the *Open File Table*
3. Fill a `FILE` structure with information about the open file
4. Return the address of the `FILE` structure to the program

The Directory

File Name	*Attributes*	*Time and Date*	*First Cluster*	*File Size*	*etc.*
...	...	...	...	...	...
foo.bar	r/w	0x73C0261A	0x0006	42000	...
...	...	...	...	...	...
...	...	...	...	...	...

Opening a File (cont.)

The operating system takes some of this information to put in the `FILE` structure to be passed back to the program.

The *Open File Table* keeps information about files that are currently open (in use by programs).

The Open File Table

Number	*Mode*	*Sharing*	*Current Position*	*Pointer to FAT*	*etc.*
...	...	...	...	...	...
3	...	...	...	...	...
4	w	...	42000	0x0006	...
5	...	...	...	...	...
...	...	...	...	...	...

Cluster, Sectors, Bytes Revisited

Recall that the smallest unit of disk accessed by the operating system is a cluster. The number of sectors in a cluster is fixed, and usually depends on the size of the disk.

In the *Windows NT File System* (NTFS), the number of sectors per cluster is calculated as follows:

```
if DiskSize > 2^(N-1) megabytes and ≤ 2^N megabytes, then
   SectorsPerCluster is 2^(N-9)
```

Most disks have 512 bytes per sector.

□

Q: In NTFS, how many sectors per cluster are there for a 10 GB (10,240 MB) disk? How many bytes per cluster?

A:

Q: In NTFS, if we have 16 sectors per cluster, what happens if the most frequent file size is 25,000 bytes?

A:

Q: What are the advantages and disadvantages of large clusters? Small clusters?

A:

The File Allocation Table

Remember the directory entry for our file `foo.bar`?

File Name	*Attributes*	*Time and Date*	*First Cluster*	*File Size*	*etc.*
`foo.bar`	`r/w`	`0x73C0261A`	`0x0006`	`42000`	`...`

Since we opened the file for *append*, the current file position (from the *open file table*) is 42000. But where *on disk* is the 42000th position in `foo.bar`?

The directory entry tells us that the first cluster for `foo.bar` is cluster number 6. If our NTFS disk is 5 GB ($< 2^{13}$ MB but $> 2^{12}$ MB) then we have

$$2^{13 - 9} = 2^4 = 16 \text{ sectors per cluster.}$$
$$\text{That's } 16 \times 512 = 8192 \text{ bytes per cluster.}$$

So if `foo.bar` has 42000 bytes, then it occupies $\lceil 42000 \div 8192 \rceil = 6$ clusters on disk. The 42000th position is somewhere in the last cluster.

□

So where is the last cluster?

The File Allocation Table (cont.)

Every cluster on disk has a corresponding entry in the FAT. (For a 5 GB disk in Windows NT that's 655,360 entries in the FAT!).

Each entry in the FAT identifies the *next* cluster (after this one) occupied by the file. So if the value of the 3rd entry in the FAT is 25, we have three pieces of information:

1. The current file occupies cluster number 3 on disk
2. The next cluster occupied by the file is cluster number 25
3. The 25th entry in the FAT will tell us what is the next cluster occupied by the file (after cluster 25)

There are some special values in the FAT:

- 0x0000 in the Nth entry in the FAT means that cluster N on disk is available (not occupied by any file).
- 0xFFFF in the Nth entry in the FAT means that cluster N is the last cluster occupied by the current file.

The File Allocation Table (cont.)

The File Allocation Table (FAT)

...	...	...	...	...	...	0x0007	0x0008
0x001B	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...
...	...	...	0x001C	0x001D	0xFFFF	...	...
...	...	...	...	...	...	...	...

According to the picture, the file `foo.bar` occupies clusters 0x0006, 0x0007, 0x0008, 0x001B, 0x001C and 0x001D on disk; our byte (`ch == 75`) will be placed in position 1040 (42000 mod 8192) in cluster 0x001D.

The Input/Output Buffers

The operating system is just a program running on the machine like any other program (though its jobs are very low level). Like any other program, it cannot physically access a disk drive; it must work with RAM.

The operating system maintains special areas in RAM (not part of the application program data area) called *buffers*. The buffers are exact "images" of the sequence of bytes in a cluster on disk. The system maintains *at least* two buffers (one for input, one for output), often many more.

To write our single byte `75` to position 1040 in cluster 0x001D on disk, the operating system has only a few tasks left:

1. Ensure that the bytes from cluster 0x001D are copied into a buffer.
    - If they are not, request that the I/O processor load cluster 0x001D into a buffer.
2. Place our byte (`75`) at position 1040 *in the buffer*.
3. Update the current file position in the *open file table*.
4. If the buffer is full, request that the I/O processor write the contents of the buffer to disk.

The Input/Output Processor and The Disk Controller

I/O Processor

A separate chip in the computer (not on the disk's circuit board inside the disk case).

The I/O processor runs separately from the CPU so the CPU can go about its business without waiting for I/O processes to complete.

Disk Controller

A separate chip on the disk's circuit board. (Sometimes refers to the whole board).

The disk controller runs the disk motors and does the bottom-level reading and writing of bytes from and to disk. It maintains its own kind of buffer (called a disk cache, usually up to 1MB) right on the disk board.

The Final Act

Now that our byte is in a buffer, we're almost home free:

1. The operating system tells the I/O processor to send the contents of the buffer to cluster 0x001D on disk.
2. The I/O processor asks the disk controller if it is accepting data for writing.
3. The disk controller accepts the request if it is ready and receives the data and disk address from the I/O processor.
4. Arms move, platters spin, heads switch on and off and our `75` becomes a splattering of polarized magnetic particles!

Giddyup!

Back in the FAT

Let's look at the FAT for our file `foo.bar` again, this time showing some chains for other files:

0x0001	0x0002	0x0003	0x0004	0x0005	0xFFFF	0x0007	0x0008
0x001B	0x000A	0x000B	0x000C	0x000D	0x000E	0x000F	0x0010
0x0011	0x0012	0x0013	0x0014	0x0015	0x0016	0x0017	0x0018
0x0019	0x001A	0xFFFF	0x001C	0x001D	0xFFFF	0x001F	0x0020
0x0021	0xFFFF	0x0000	0x0000	0x0000	0x0000	0x0000	...

Q: The last byte in `foo.bar` was at position 1040 in cluster 0x001D. What would happen if we wrote another 8000 bytes to `foo.bar`?

A:

Q: Our FAT pictures show each cluster entry as two bytes (called a 16-bit FAT). This is supposed to be a 5 GB disk with 8192 bytes per cluster. Can this FAT picture be accurate?

A:

Q: How big a disk could you have (under NTFS) with a 32-bit FAT (assuming 512 bytes per sector)?

A:

FAT Trivia

- The chain formed by the "pointers-to-next-cluster" is called a *chain*!
- DOS used to keep *two* copies of the FAT on disk and religiously updated both every time any file changed. But it never did any checks to make sure the two FATs were the same, and it never used the second FAT.