Computer Basics II: Disks and File Systems

CS 585

Fall 2009

Storing Bits

- In the CPU/cache/memory:
  - bits persist only as long as power is applied
- Disks and flash technology:
  - bits persist across removal/reapplication of power
- Disk storage medium:

  Platter: non-magnetic material (aluminum alloy or glass)

  Coating: thin layer of magnetic material

  Typically 10-20 nm thick (smallest bacteria: ~200nm)

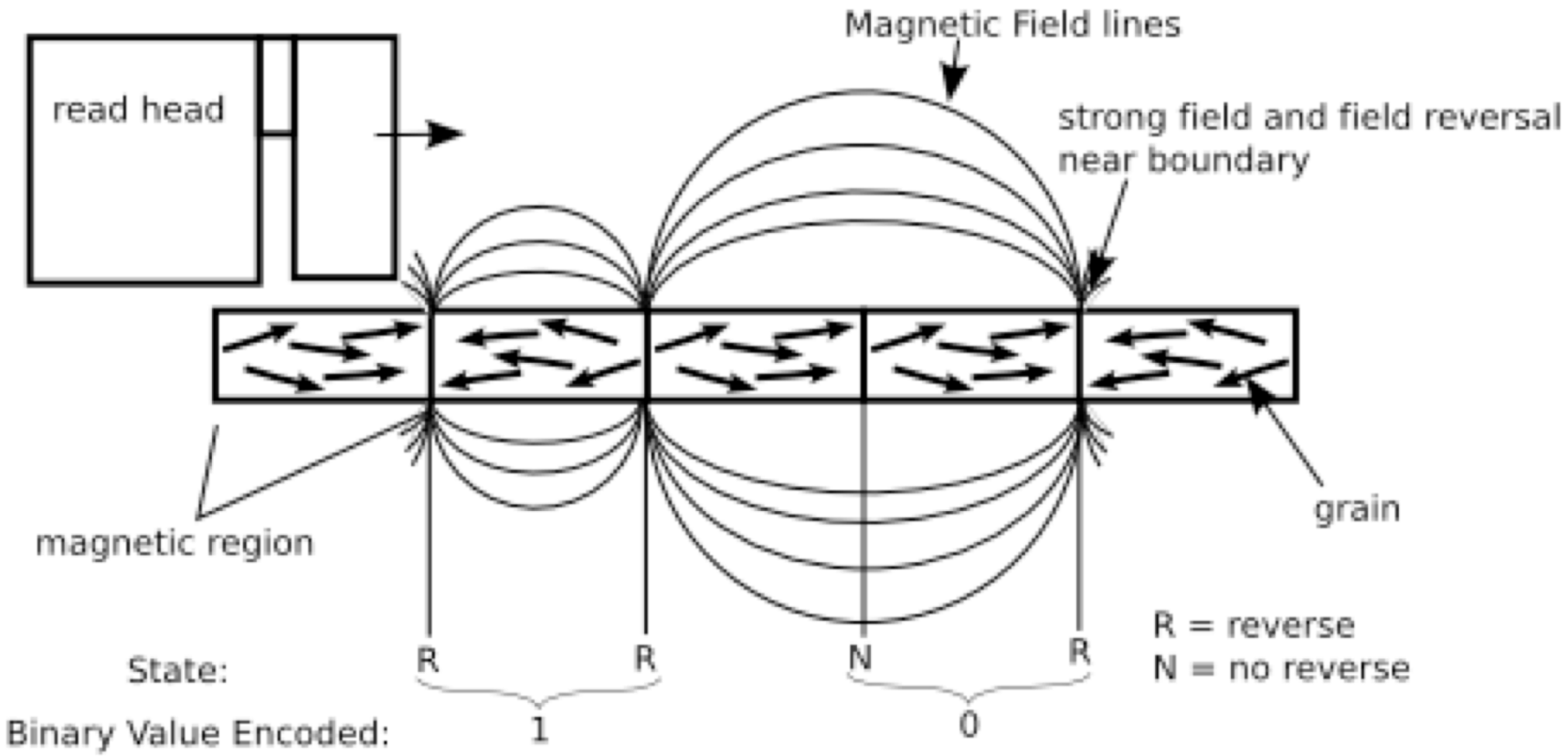
  Outer carbon coating

Storing Bits

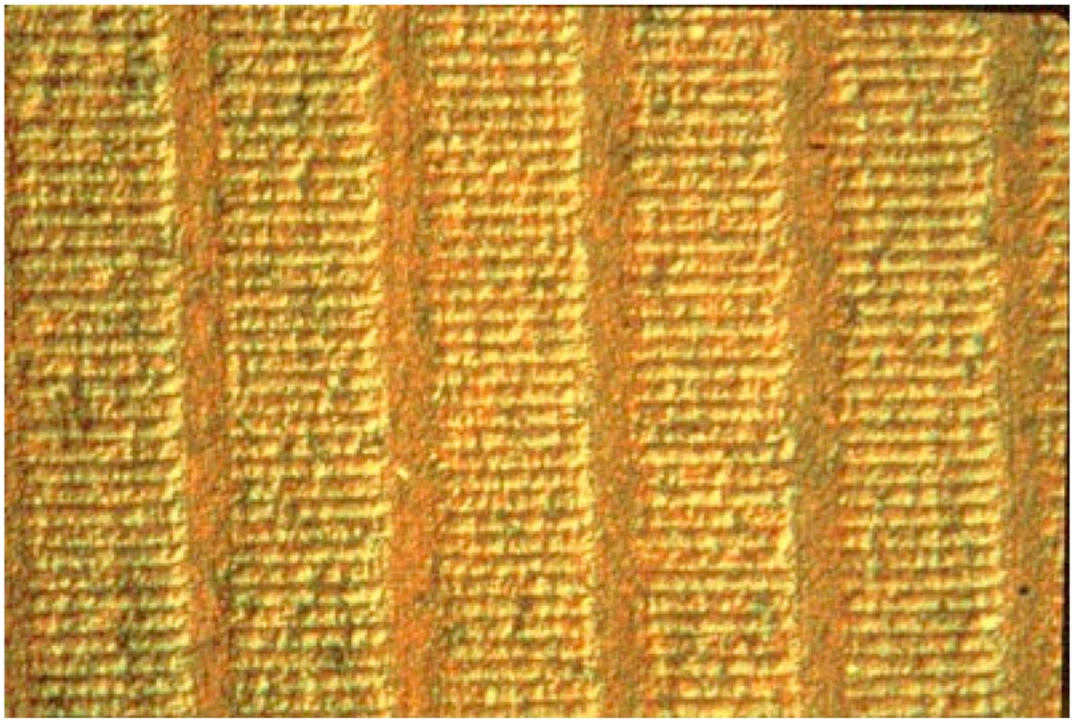
- Platter rotates at high speeds
  - 5400, 7200, or 10000 RPM are common
  - 2.5” platter @ 7200 RPM: outer edge moves ~53 mph
- Read/write head “flies” just above the surface
- Bits are represented by direction of magnetic flux stored in discrete magnetic regions
  - Each region contains ~100 magnetic grains
  - Typical region dimensions (2006):
    200-250nm (radial) x 25-30nm (circumferential)
  - Available density today: 3 to 4 x 10^{11} bits/in^2

Disk Hardware

Disk Storage Technology



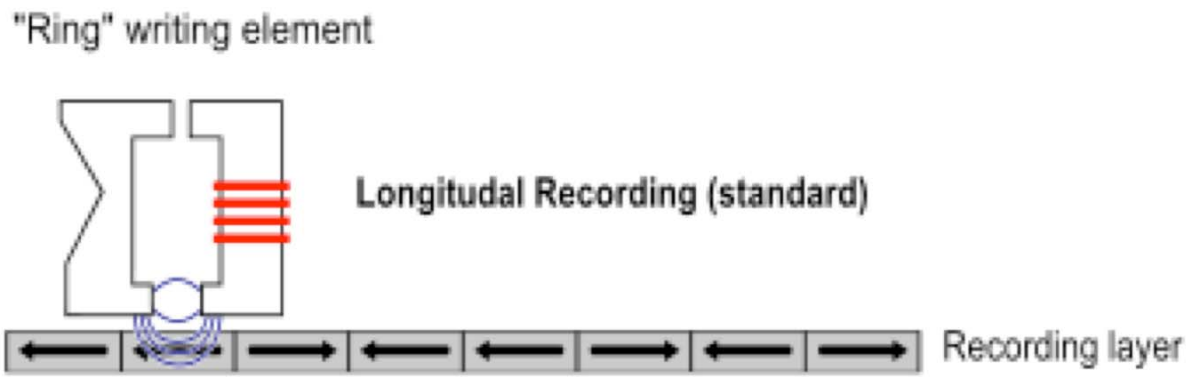
Disk Storage Technology

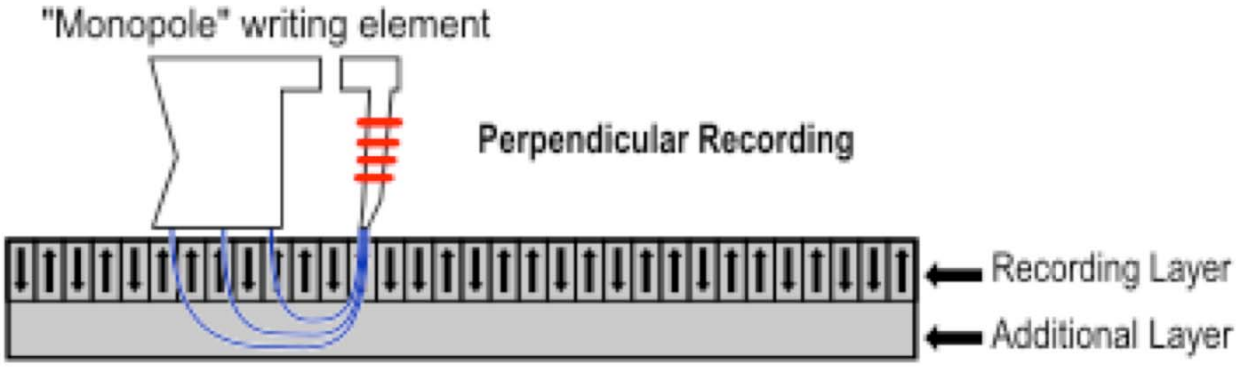


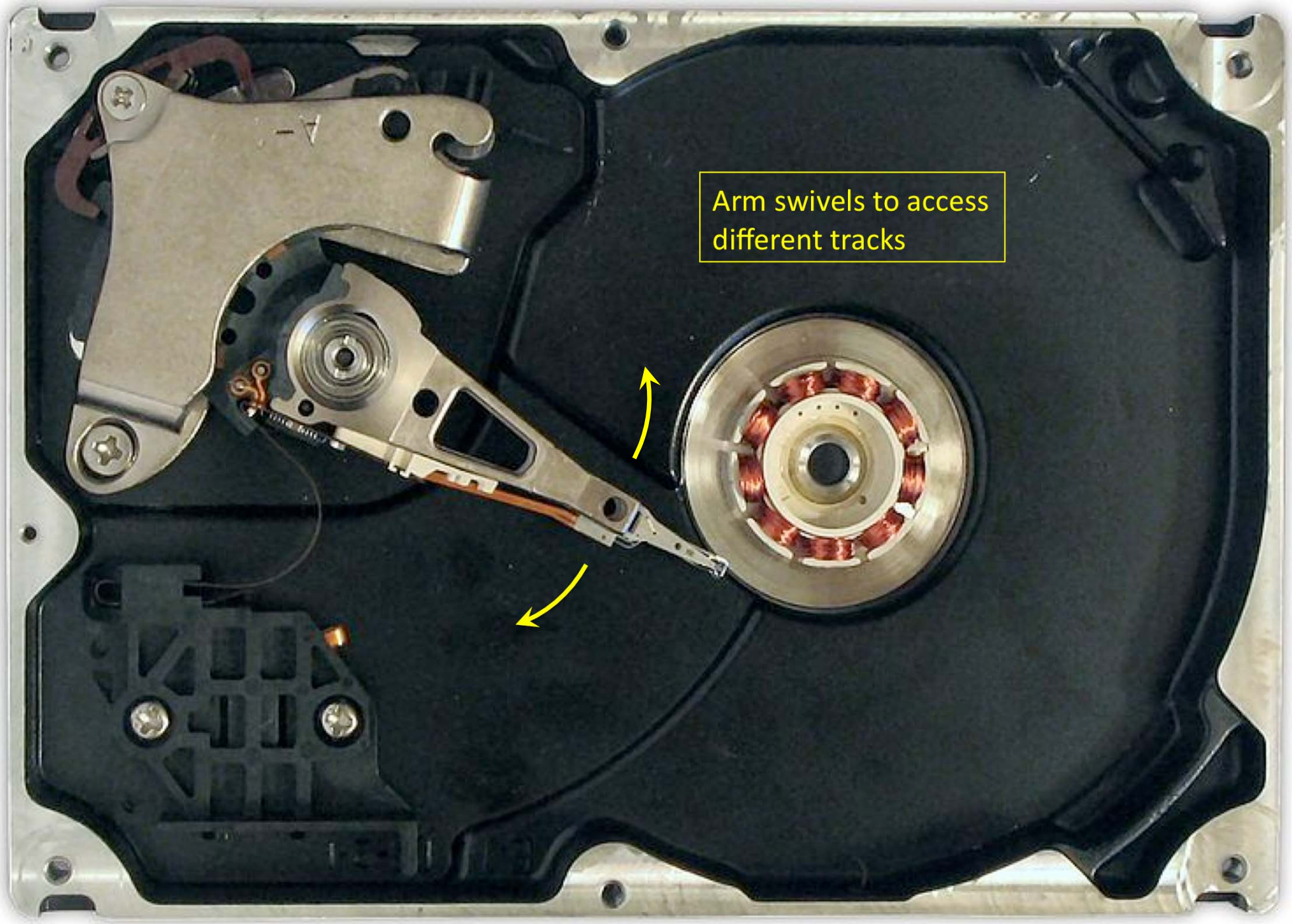
Magnetic regions in five tracks on an IBM 3380 disk surface.

The surface held 1774 concentric tracks, about 22 million bits/in^2.

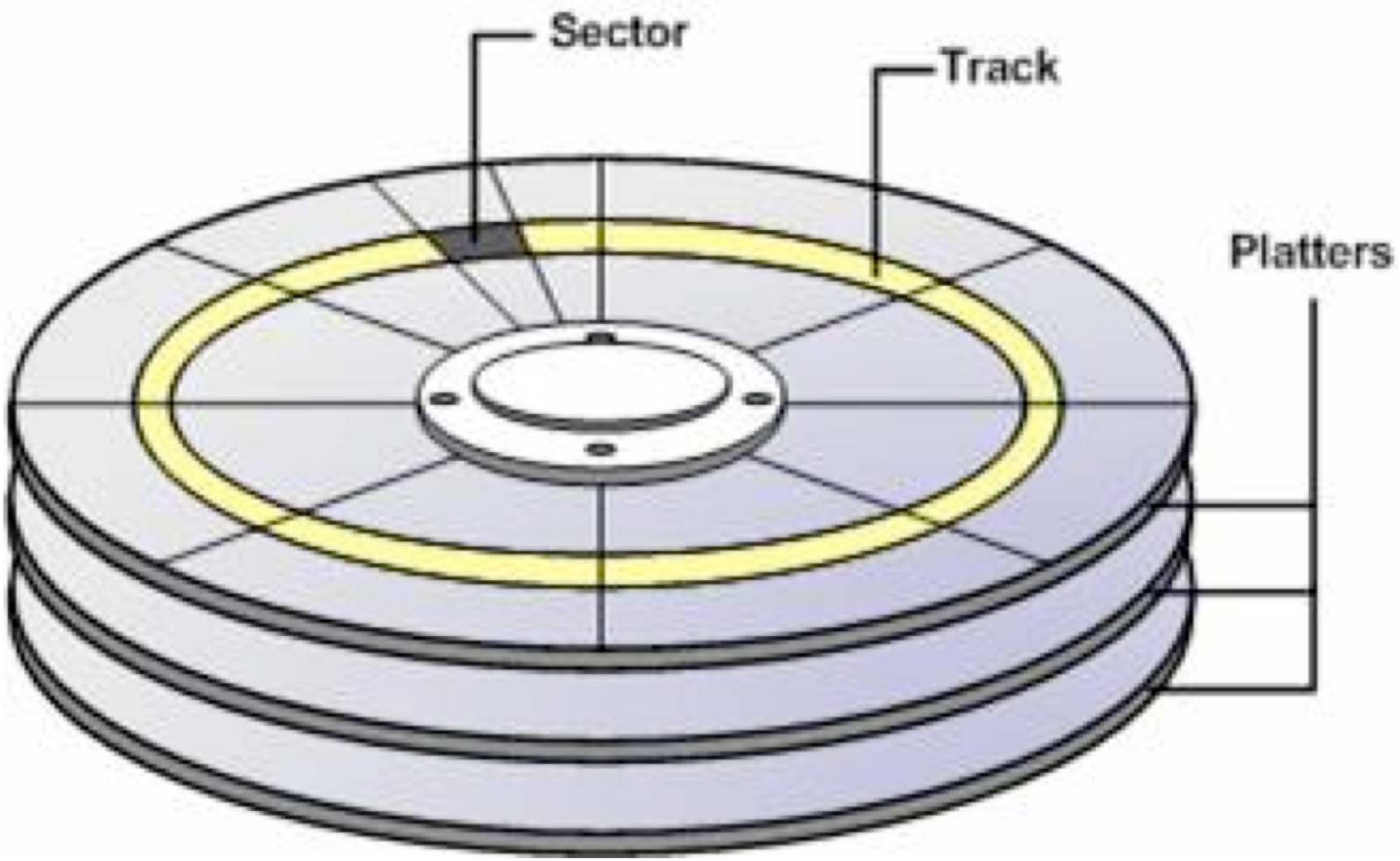
“Perpendicular” Regions: Higher Density






Arm swivels to access
different tracks

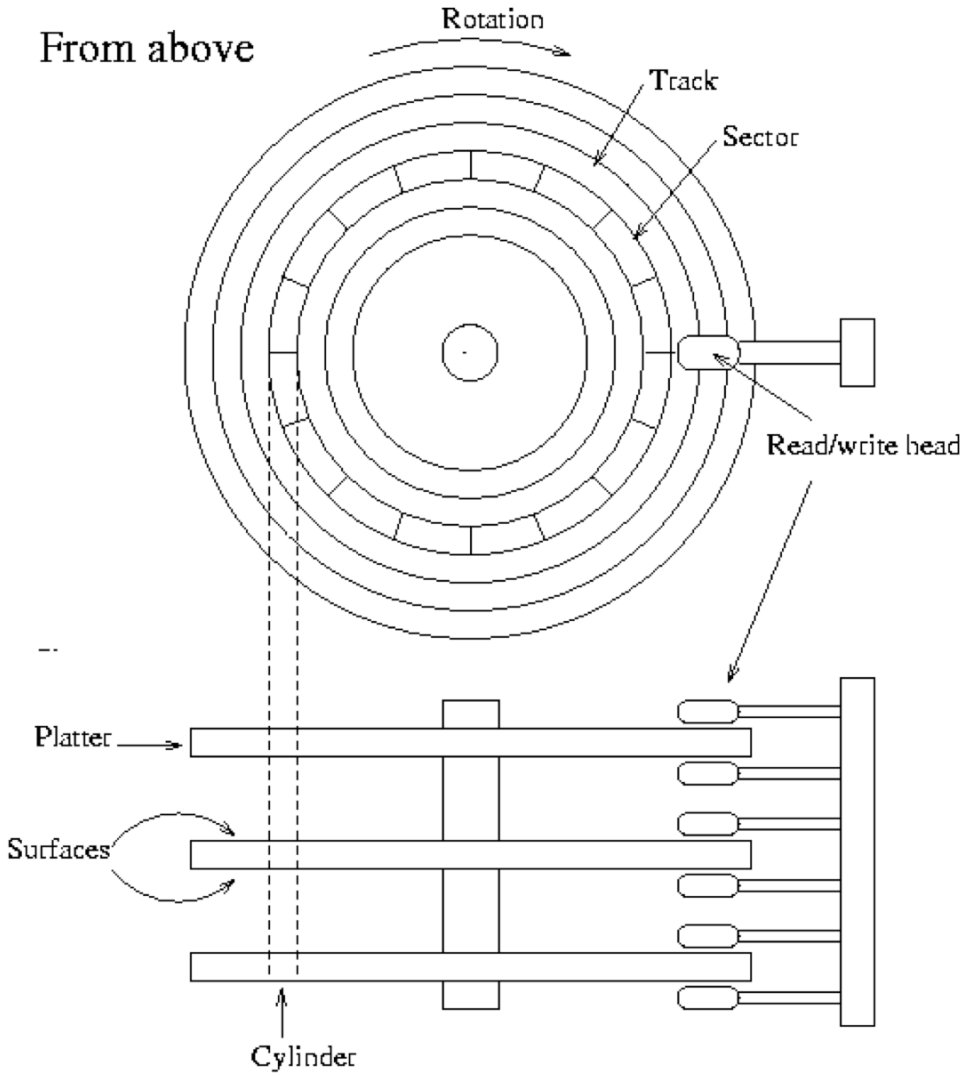
Disk Organization



Disk Geometry

- Sector = minimum unit of data transfer
  - Generally 512 bytes
  - Note: typical sector contains far more than 2^{12} bits
    - Synchronization bits – indicate beginning of track
    - Redundant bits for error correction coding
- Track = ring of sectors on a platter
  - Note: inner tracks are shorter than outer tracks!
- Head = platter surface (on multi-platter drives)
- Cylinder = all tracks at the same radius

Disk Geometry

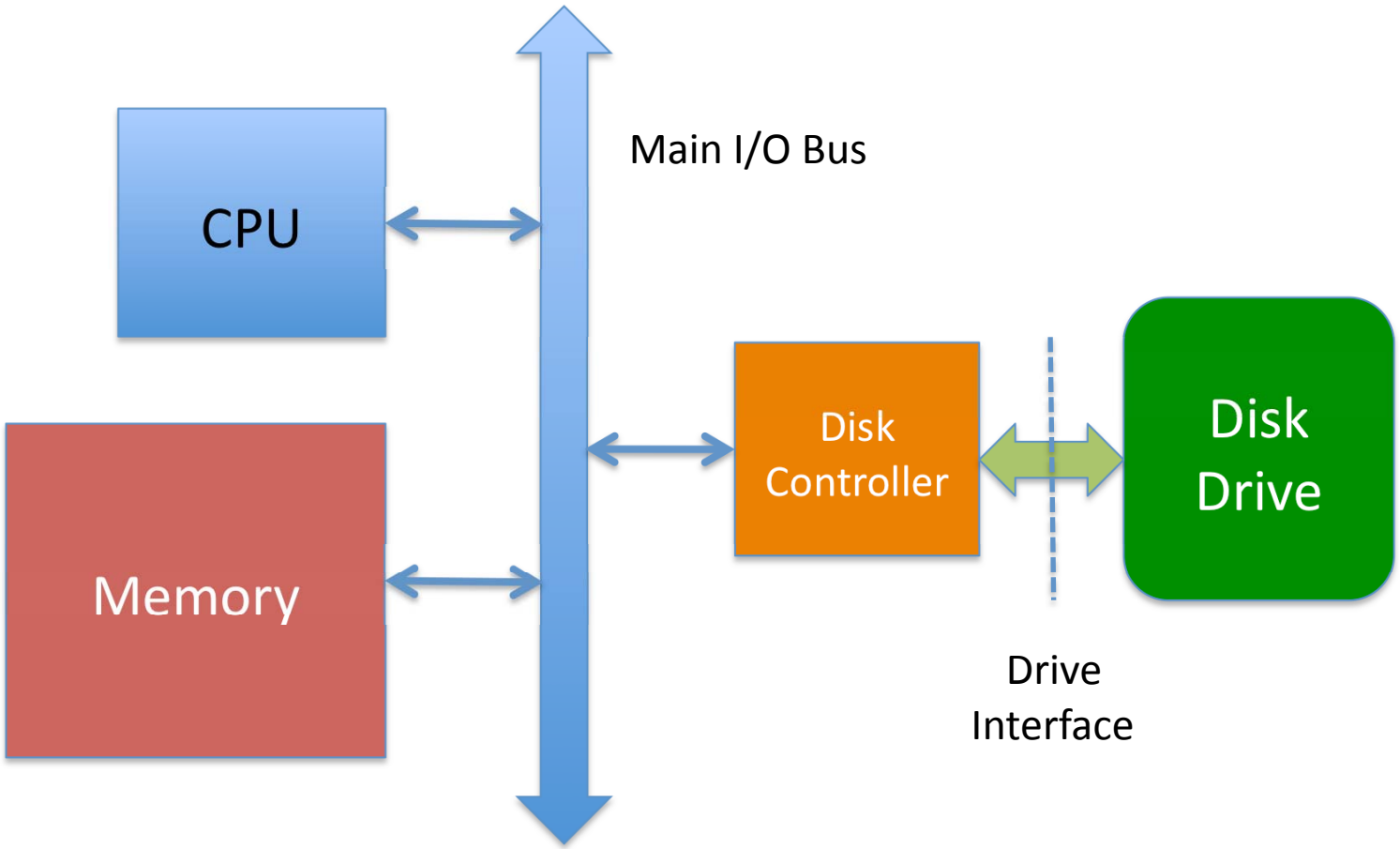


Identifying Blocks

- Drives are usually characterized by "geometry:"
  - Number of cylinders (C)
  - Number of heads (H) (= tracks/cylinder)
  - Number of sectors per track (S)

    C x H x S = number of addressable sectors
  - In modern disks these are a convenient fiction
    - Drive smarts hide the actual arrangement
- Sectors can be addressed by (C, H, S) triple
- Logical Block Addressing (LBA): treat the disk as a linear array of sectors
  - Blocks can be addressed by simple index
  - Drive smarts keep track of the actual layout

Drives in the System



Disk Drive Interfaces

- Early days (ST-506, Shugart/Seagate)
  - Dumb drive, separate controller
  - Drive controlled by low-level commands
    - Move head, select head, start transfer, etc.
  - Controller translates request for physical block into low-level commands
- Integrated Drive Electronics (IDE, Western Digital)
  - Controller inside drive package handles low-level details
  - 40-wire parallel connector
  - Evolved into AT Attachment (ATA, ANSI X3 std)

Disk Drive Interfaces

- ATA evolved through several stages
  - ATAPI, 1998
  - “Ultra DMA” transfer modes – required 80-wire cable
  - 28-bit addressing (max addressable ~137 GB)
    - ATA-6 went to 48-bit addressing (limit 144 petabytes)
- Serial ATA introduced in 2003
  - Smaller, longer cables
  - Hot-swappable
- Small Computer System Interface (SCSI)
  - 1980’s – present
  - More sophisticated interface – required more expensive drive hardware

Example: IDE

40-pin connector:

pin	Name
1	RESET
2,19,22,24, 26,30,40	GROUND
3-17 odd	D7-D0 (low data byte)
4-18 even	D8-D15 (high data byte)
20	[blocked; physical alignment]
21	DDREQ (handshaking)
27	IOCHRDY (handshaking)
29	DDACK (handshaking)

Example: IDE

40-pin connector:

pin	Name
23	WR (write control signal)
25	RD (read control signal)
28	ALE (?)
31	IRQ (Interrupt Request)
32	IO16 (8/16-bit control)
35,33,36	A0,A1,A2 (Register select)
34	DMA?
37, 38	CS1, CS3 (Register select)
39	ACTIVITY (“busy”)

IDE Drive Registers

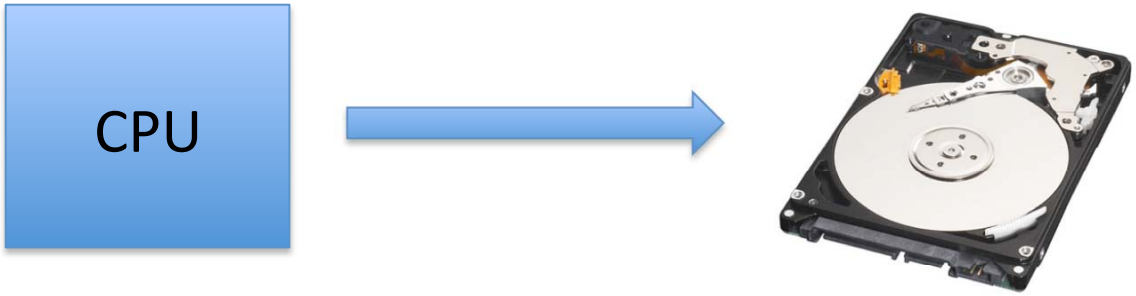
- Data I/O (16 bits)
- Sector Counter (8 bits)
- Start Sector (8 bits – range = 1..256)
- Cylinder Low byte (8 bits)
- Cylinder High bits (2-8 bits – traditional IDE only allows 1024 Cylinders
- Head and Device (8 bits)
    - Selects Master/Slave drive
- Comand/Status register (8 bits)
    - Different functions read/write

Example: Simple PIO Read

0. Select Cylinder Low register using A0-A3, CS1, CS3
1. Put low byte of Cylinder number on D0-D7
2. Strobe WR; Await /ACTIVITY
3. Select Cylinder High register using A0-A3, CS1, CS3
4. Put high byte of Cylinder number on D0-D7
5. Strobe WR; Await/ACTIVITY

6-12. Repeat for Head, Sector registers

7. Select Command register using A0-A3, CS1, CS3
8. Put 20h ("Read Sectors with retry" on D0-D7, strobe WR)



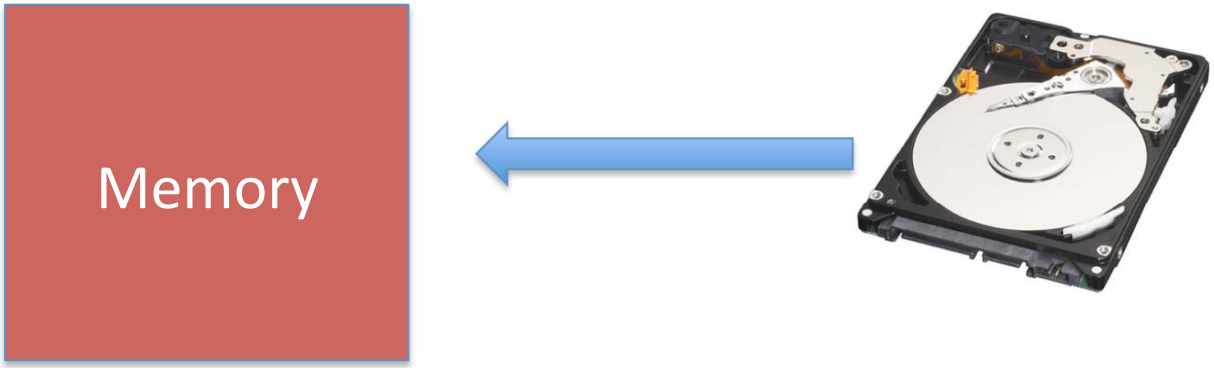
Note: this is a rough sketch, many details omitted or mangled.

Example: Simple PIO Read

9. Read data (8 or 16 bits at a time, depending on IO16) into memory (using handshake lines).

Note: this PIO mode transfer requires that the CPU read a word from the location in memory corresponding to the data lines D0-D15, and write it into a different memory location (buffer). With DMA mode transfers, the drive handles transferring bytes into the buffer.

10. Monitor "Status" register contents to determine completion status.



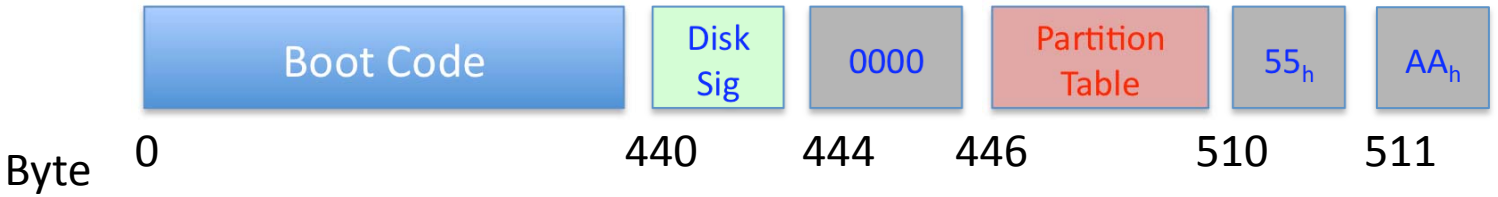
Note: this is a rough sketch, many details omitted or mangled.

File Systems and Partitions

- File System = way of organizing data for storage on disk
  - Provides named file and named directory abstractions
  - Examples: FAT-16, FAT-32, NTFS, Ext3, HPFS, …
  - To get useful information from a drive, need to understand its file system organization
- A single physical drive may be partitioned
  - Operating System treats it like multiple disk drives
  - Why do this?

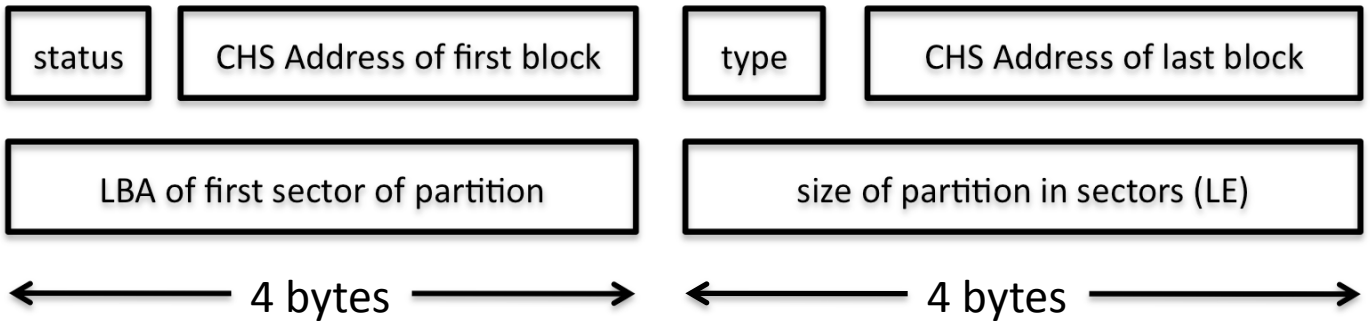
Master Boot Record

- First addressable block (LBA 0) on the disk
  - Convention: Machines read this block first when booting from disk
- Contains:
  - Bootstrapping code (up to ~440 bytes)
  - Partition table
  - MBR Magic Number: 0x55, 0xAA



Partition Table

- By convention: exactly four entries
  - Some may be unused
- Each entry tells:
  - Addresses of first and last blocks in the file system
  - Type of file system used in that partition (code #)



Partition Types

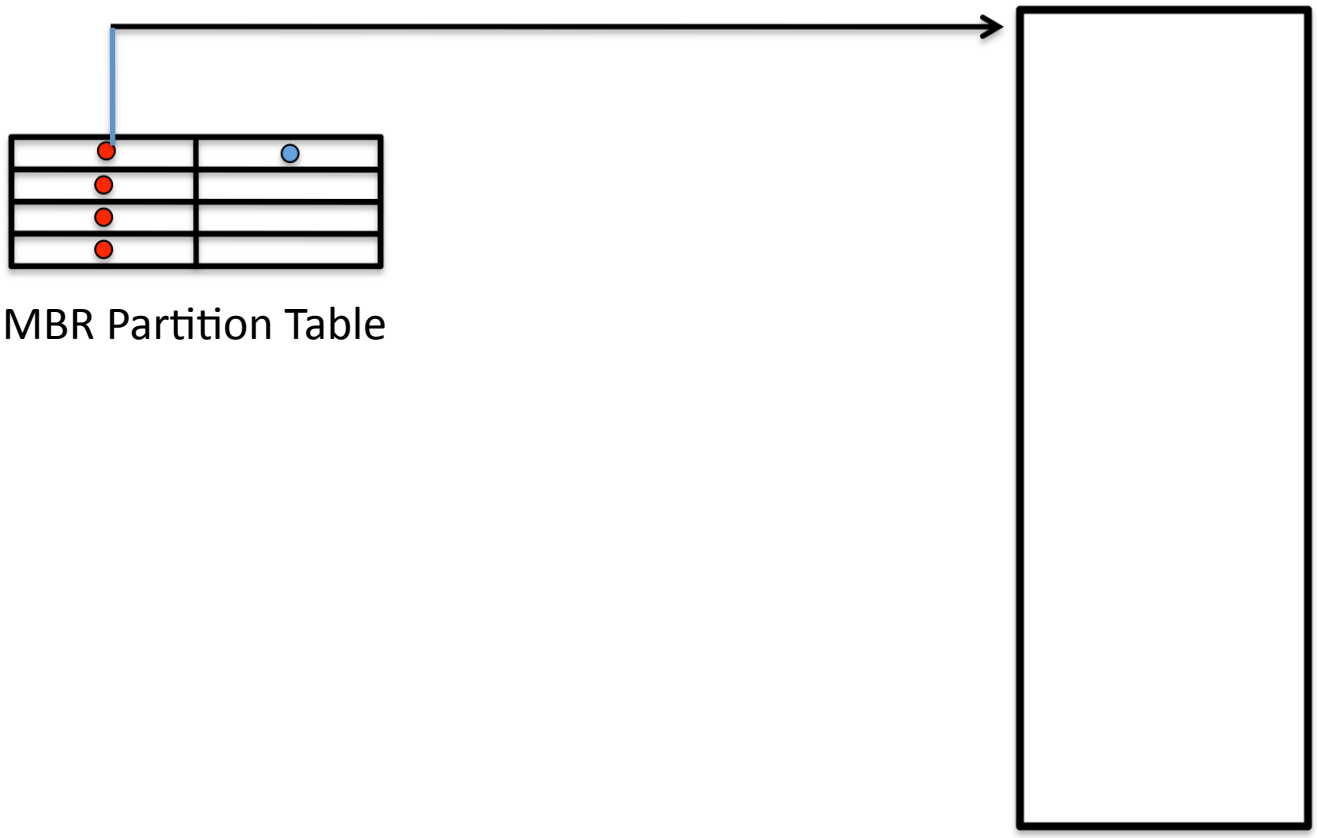
- No “official” assignment of types
  - No guarantee that two different types don’t use the same number!
- 0 = unused partition
- 04_h, 06_h = FAT
- 07_h = NTFS or HPFS
- 83_h = linux file system: ext2, ext3, reiserfs, ...
- 82_h = linux swap space
- 05_h or $0F_h$ = extended partition

Extended Partitions

- What if you want to have more than four partitions on a physical disk?
- Solution: Use one entry (max) in the partition table as an extended partition
  - Specified like any other partition
  - Type code 0x05 or 0x0e or 0x0f
  - First sector in the partition contains another partition table with two entries
    - First entry describes one (the first) “logical” (sub)-partition in the usual way (= start sector + length in sectors)
    - Second entry is a “pointer” to the next partition table (0 if there is none)

Extended Partition Example

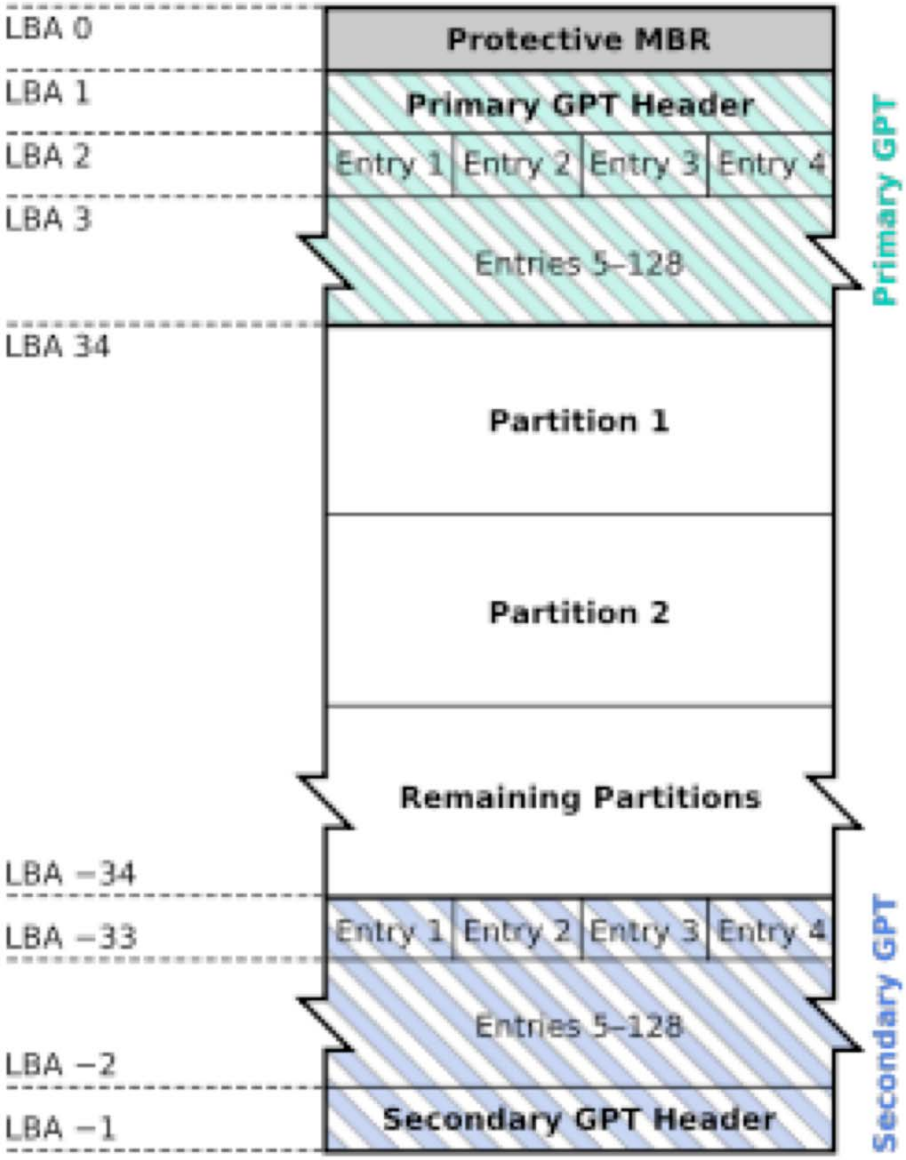
(Note: this shows only the start and length fields of each partition table entry)



Modern Partition Tables

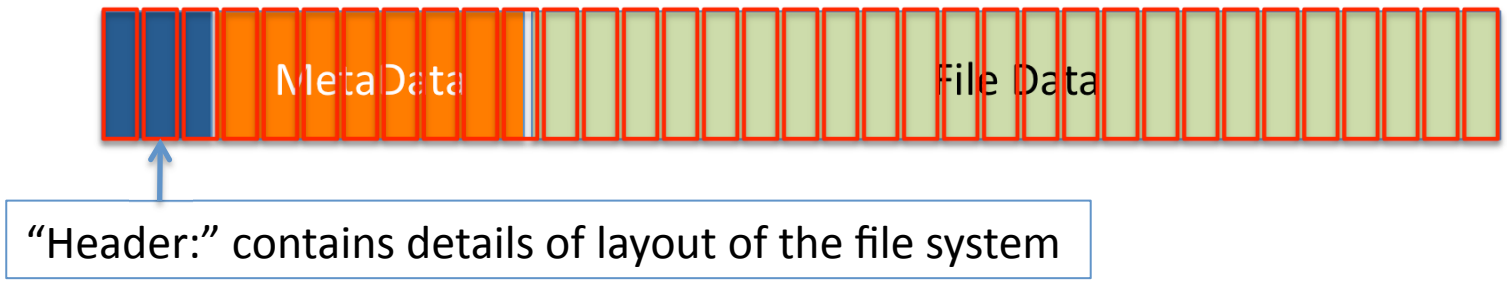
- 2^{32} sectors x 2^{9} bytes/sector = 2 Terabytes
  - Need bigger partition tables!
- Globally Unique ID (GUID) Partition Table (GPT)
  - Part of Intel's Extensible Firmware Interface (EFI) initiative to replace the PC BIOS
  - Many additional features besides bigger addresses
    - drops CHS addressing altogether
    - Checksum for detecting corruption
    - Variable-sized partition tables
    - Backup partition tables stored on disk
    - GUID for each disk
- Still need a "Legacy MBR" PT in LBA 0
  - Configured as a single partition, type 0xEE = GPT
  - OS's that know about GPT ignore size in Legacy MBR

GUID Partition Table Scheme



File Systems

- What they provide:
  - Higher-level abstractions: named files and directories
  - Management of free space
  - Hiding complexity of locating information on disk
- Principle: some of the "space" on the disk is used up to provide these services



The “File” Abstraction

- File = named, finite, sequence of bytes
  - Bytes encode information of some type (BG Part I)
- Name = sequence of characters
  - Characters are encoded as bytes (of course)
  - Name provides a (unique) identifier for the file
- Directory = special file that “contains” other files
  - Creates a hierarchical structure: directory tree
  - “Root” directory: entry point of tree

Program Access to File Data

Standard OS abstraction:

```
fd = open("myFile.txt",O_RDWR|O_CREAT);
rv = lseek(fd,1000000,SEEK_SET);
...
rv = write(fd,buffer,BUFSIZE);
...
rv = lseek(fd,0,SEEK_SET);
...
rv = read(fd,rbuf,BUFSIZE);
...
close(fd);
```

Program Access to File Data

OS has to:

- Remember that this program has “opened” the file
- Keep track of “current location” in the file
  - Initially at beginning of file
  - Can’t read past the end of the file!

File system has to:

- Find the data (disk sectors) associated with filename
- Locate and allocate new blocks as the file grows
- De-allocate and reclaim blocks belonging to deleted files

Buffering

- Operating System keeps a pool of buffers containing copies of some blocks in a file system
  - Each buffer is a block of memory the same size as the minimum-sized unit of disk allocation (typically: sector)
  - OS keeps track of which block is in which buffer
  - When a program needs to read block j, first check whether a copy is already in a buffer
    - If so: copy from the OS buffer into the program's buffer
    - If not: find a "free" buffer; send a request to the disk to read the corresponding sector(s) into that buffer

Buffering

- Number of buffers << Number of sectors on disk
  - Need algorithm to decide which buffer to “evict” when all are busy (typically: LRU)
- Writing to block j:
  - First get the block into a buffer
  - Write into the buffer in memory
    - Disk not yet changed!
  - Eventually, flush the buffer by writing its contents back to (the appropriate block on the) disk
    - Need algorithm to decide when to flush “dirty” buffers

How to do it?

- Information to be associated with filename:
  - Data (zero or more disk sectors)
  - ...
- The way this association is accomplished varies with the file system type
- General Principle:

  Directories associate file names with file information

File Allocation Table

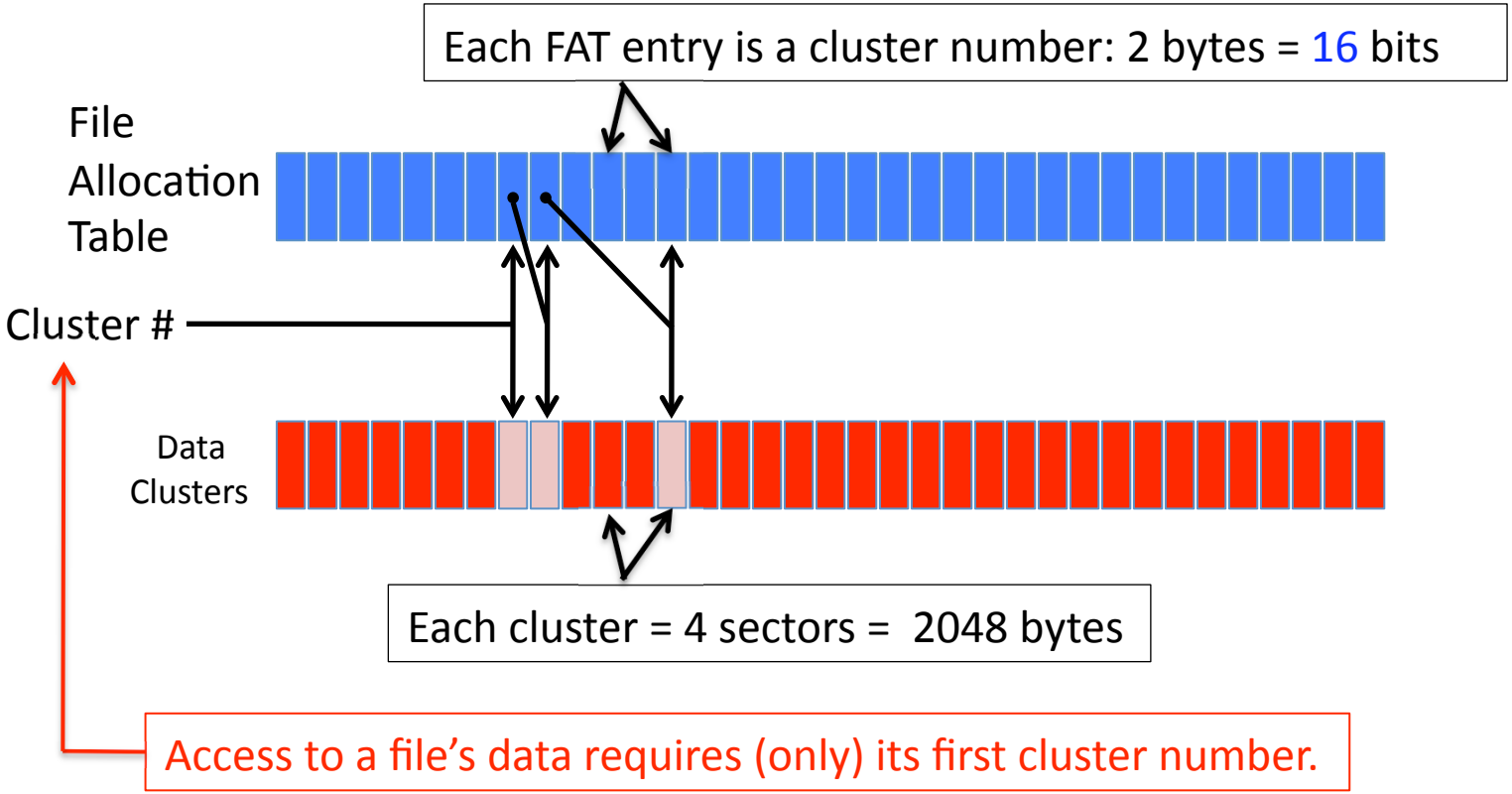
- File System from Microsoft
  - Dates from MS-DOS
  - Often used on USB “Thumb drives”
  - FAT12, FAT16, FAT32
    - Evolved to deal with ever-larger drives
    - Features like long filenames added (backward compatible)
- Primitive, not terribly efficient

General FAT Principles

- Data is allocated in clusters, which consist of 2^k sectors, where k = 0, 1, 2, 3, 4, 5, 6, or 7
  - k is fixed for the entire filesystem
  - Sectors are almost always 512 bytes, but in theory could be otherwise
    - Example: for k=6, the smallest nonempty file uses 32KB
  - As the file grows, more clusters are allocated
- Disk volume (partition) divided into FAT and data (cluster) areas

FAT: Chaining Clusters

Example: FAT16, cluster Size = 2K



FAT Contents

- Each entry in the FAT table is a 12-, 16- or 32-bit number (depending on type)
- Possible values:
  - 0: denotes a free cluster
  - FFF, FFFF, 0FFFFFFF: End-of-Chain mark
  - FF7, FFF7, 0FFFFFF7: Bad Cluster mark (not used)
    - Note: no FAT32 volume should ever be configured so that FFFFFF7 is an allocatable cluster number
- Note: No list of free clusters is stored (!)
  - To find a free cluster, scan list until finding a 0 entry

Directory Entry– Short Filenames

- Designed for MS-DOS “8+3” filenames
  - E.g. EVIDENCE.TXT, COMMAND.COM, WINDOWS.EXE, etc.
- Directory Entry: 32-byte structure:

```
Name              11 bytes    // “dot” is implicit, both parts padded
                              //  Name[0] = 0xE5 means “free”
Attr              1 byte      // Bit flags: readonly, sys, etc.
Rsvd              1 byte      // Reserved for NT
CreatTimeTenth    2 bytes     // Tenths part of create time
CreateDate        2 bytes     //  Date created
LastAccessDate    2 bytes     //  Date of last access
FirstClusterHi    2 bytes     //  1st cluster index, high word
WriteTime         2 bytes     //  Time last written
WriteDate         2 bytes     //  Date last written
FirstClusterLo    2 bytes     //  1st cluster index, low word
FileSize          4 bytes     //  size in bytes
```

Short Directory Entry Examples

“foo.bar”	FOO□□□□□BAR	(□=space)
“FOO.BAR”	FOO□□□□□BAR	
“Foo.Bar”	FOO□□□□□BAR	
“foo”	FOO□□□□□□□□	
“foo.”	FOO□□□□□□□□	
“pickle.a”	PICKLE□□A□□	
“prettybg.big”	PRETTYBGBIG	
“.tar”	(illegal: Name[0] cannot be 20_h=space)	

Short Directory Entry

- Letters, digits, and other characters with code points greater than 127 (e.g., Japanese)
  - Space (20_h) is allowed (in the “8” part)
- Always mapped to upper case
  - This is problematic for some charsets!
- Name[0]=$E5_h$ indicates free entry
  - Name[0] = 0 indicates free and all subsequent entries are free
  - Hack: $E5_h$ is a valid Japanese character; use 05_h instead
- Max path length: 80 characters
  - includes trailing null
  - 64 path + 3 drive letter + 11 + 1 = 79 (?)

Supporting Long Filenames

- Goal: Support more general filenames
  - up to 255 characters
  - larger character sets (Unicode – 2 bytes/char)
- In a backward-compatible fashion
  - Must not confuse older disk-checking programs
    - They use raw device access
    - Must not jeopardize integrity of existing file data
- Basic idea: use multiple 32-byte entries to store long filenames
  - Combination of attribute flags indicates long entry

    RDONLY, HIDDEN, SYSTEM, VOLUME_ID
  - Each entry looks (more or less) like a valid short entry to legacy software

Long Filenames: Principles

- For each file with a long name, there is a regular (short) entry containing a unique, automatically-generated short filename to which the long name maps
  - e.g. PROGRA~1.EXE
- Auto-generated short names are unique
- The long name and corresponding short name are stored in a contiguous sequence of 32-byte directory entries
- Case is not distinguished in long names
  - Cannot create foobar if FooBar exists, even with long names

Long Name Directory Entry

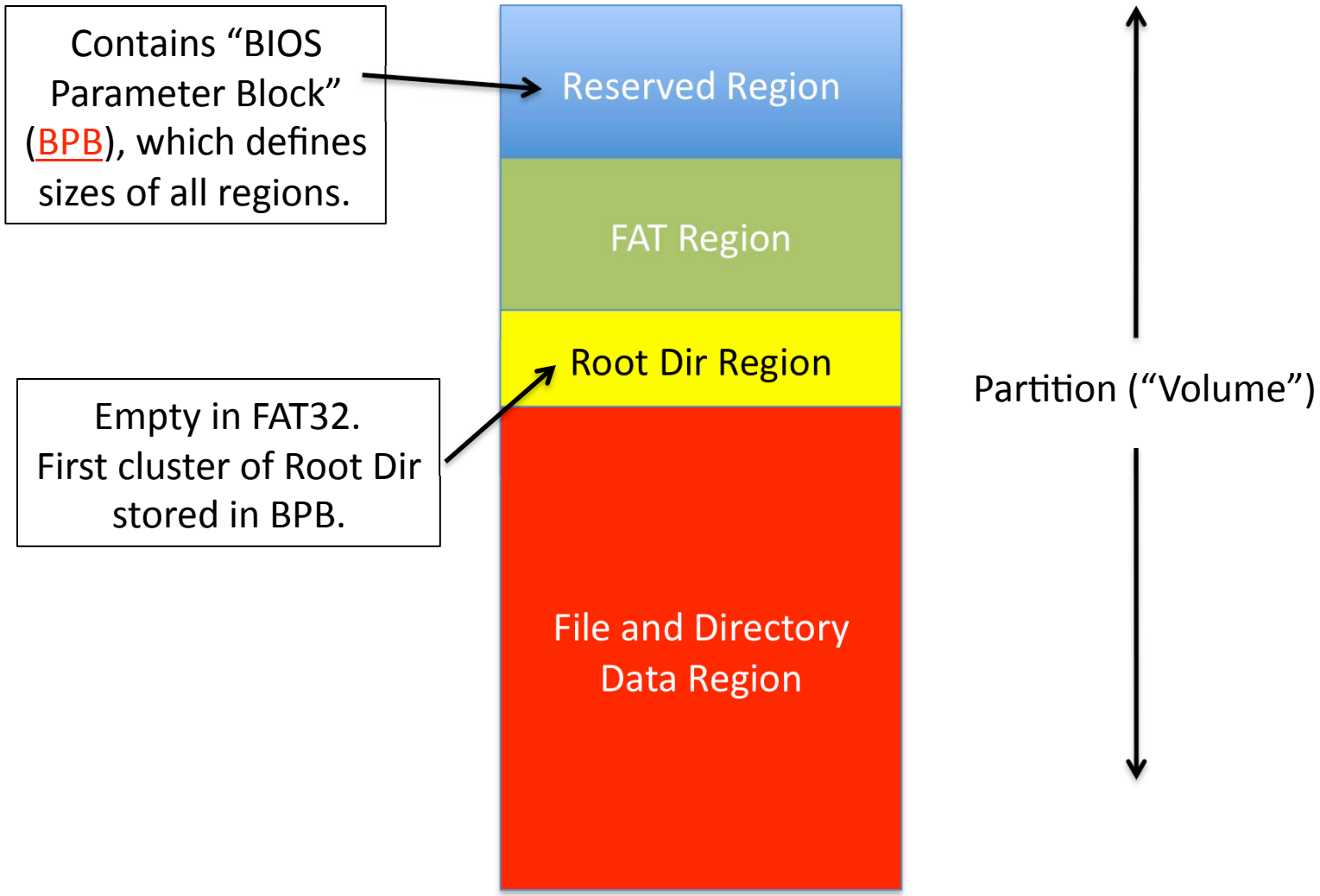
Bit 0x40 set in first byte to indicate “last” [= first] in set

Long Name Directory Entry

- Directory Entry: 32-byte structure:

Ordinal	1 byte	// This is the Nth dir entry
Name1	10 bytes	// First 5 chars (unicode)
Attributes	1 byte	// Always RDONLY|HIDDEN|SYSTEM|VOL_ID
Type	1 byte	// 0 means part of long name
Checksum	2 bytes	// f(short name)
Name2	10 bytes	// Next 5 chars of name
FirstClusterLow	2 bytes	// Always 0
Name3	4 bytes	// Next 2 chars of name

The Big FAT Picture



Where's the Root?

- To find a file's directory entry from its pathname:
  - Recursively search directories for components of pathname, starting from root
  - Final component is filename
- Must be able to find the information for the root directory
  - FAT12, FAT16:
    - Root directory immediately follows the FAT
    - Size (# of entries) contained in BPB

BIOS Parameter Block (Common)

Field Name		Size (bytes)	Comment
jmpBoot		3	// always EB, ??, 90 or
			// E9, ??, ??
OEMName		8	// “MSWIN4.1” typical
BytesPerSector		2	// 512, 1024, 2048 or 4096
SectorsPerCluster		1	// 1, 2, 4, 8, 16, 32, 64, or 128
ReservedSecCount	2		// size of Reserved Region
			// should be 1 (12,16)
NumFATs		1	// should be 1 or 2
RootEntryCount		2	// Size of Root Region(12,16)
			// must be 0 for (32)
TotalSectorsCount	16	2	// total # of sectors or 0 (32)
FATSize16		2	// #sectors used for FAT (12,16)
...			
FATSize32		4	//#sectors for FAT (32)

BIOS Parameter Block
FAT12, FAT16

Field Name	Size (bytes)	Comment
BootSig	1	// always 29_h
VolumeID	4	// Serial Number
VolumeLabel	11	// Matches label in root dir
FileSysType	8	// “FAT12□□□” or
		// “FAT16□□□” or
		// “FAT□□□□□”

BIOS Parameter Block
FAT32

Field Name	Size (bytes)	Comment
FATSize32	4	// size of FAT in sectors
ExtFlags	2	// blah
FSVersion	2	// minor, major
RootCluster	4	// First cluster of Root dir
FSInfo	2	
BkBootSect	2	// loc. of bk-up boot sec (usu 6)
...		
VolumeID	4	
VolumeLabel	11	// same as before
FileSysType	8	// Always “FAT32□□□”

Is it FAT12, FAT16, or FAT32?

- RootDirSectors = ((RootEntryCount * 32) + (BytesPerSec – 1)) / BytesPerSec;
- DataSectors = TotalSectors – (ReservedSecCount + (NumFATs * FATSize) + RootDirSectors)
- ClusterCount = DataSectors/SectorsPerCluster
- If (ClusterCount < 4085) FAT12
  else if (ClusterCount < 65525) FAT16
  else FAT 32

Tools for Examining Disks

Warning! In general it is not “safe” to plug a drive that you want to preserve into a Windows (or any other) system

- If automounted, FS info may be altered
- “Autorun” code may be started ...

Having said that…

You may not have a choice if you don’t have a write-blocker.

(Why write-blockers are a great idea!)

Tools for Examining Disks

- Device Files
  - Most systems have pseudo-files that provide block-level access to each drive/partition
    - MacOS: /dev/disk0 = entire drive
      /dev/disk0s1 = first partition
      /dev/disk0s2 = second partition ...
  - File systems are mounted from these pseudo-files
- Raw Device
  - Treats entire drive as linear sequence of uninterpreted bytes
    - Read 512 bytes starting at offset 0 ➔ Master Boot Record
  - Names prefixed with “r”, e.g. /dev/disk0 ➔ /dev/rdisk0

Tools for Examining Disks

- fdisk: partitioning tool
- dd: disk-to-disk copy
  - Reads blocks of bytes from input file, writes to output file
  - Suitable for block-by-block copying from raw device
    - Can be used to make an “image file” of a drive
    - Note: to be safe, engage write lock
- diskutil (MacOS)
- fsck, fsdb – old Unix file system utilities

Unix File System

- Based on the original Bell Labs Unix file system
  - Developed at Berkeley in the late 1980's
  - Refinements to make it more efficient for larger files and file systems
  - AKA "UFS", "FFS", "Berkeley Fast File System"
- Pre-LBA
  - Attempts to reduce the amount of arm movement
  - Reduces amount of space wasted due to breakage
    - That is: using only a small part of a large cluster

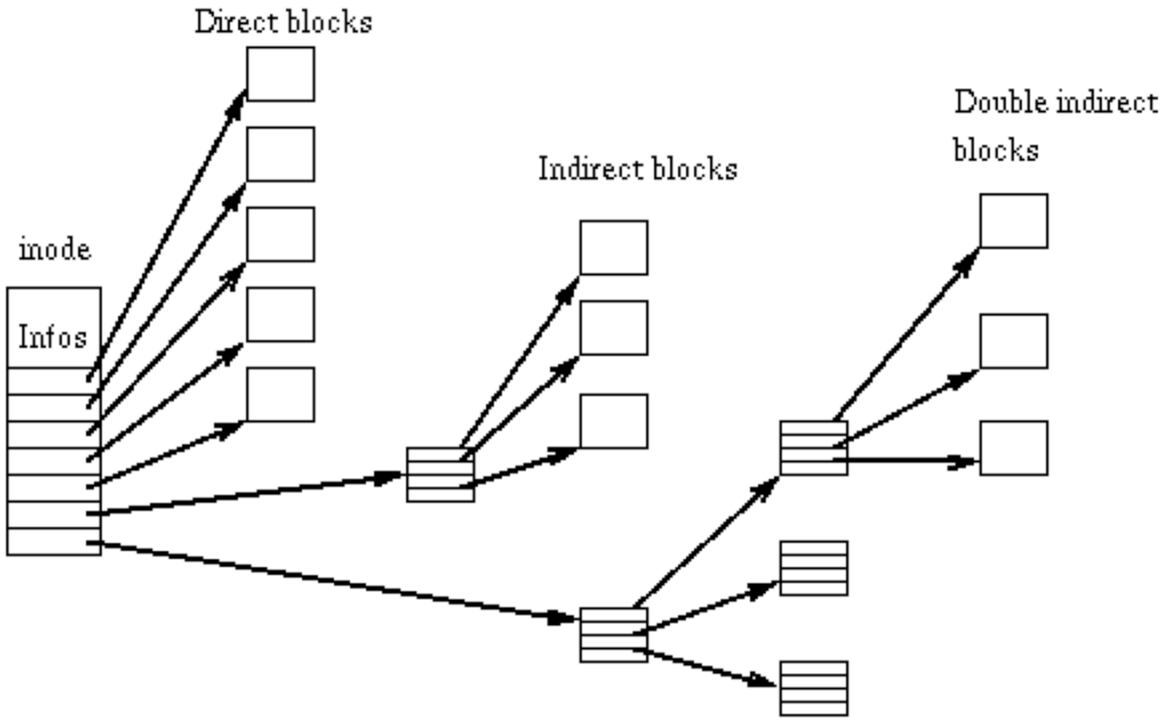
Unix Filesystem

- Inode: fixed-size structure; repository of all meta-information associated with a file
  - Locations of data blocks
  - File size in bytes
  - Owner, group UIDs
  - Access control information (permissions)
    - Bitmap: (Owner, Group, World) x (read, write, execute)
    - Additional: “sticky” bit, setuid, setgid
  - Creation, Modification, Access times
  - Link count (≈ number of names for this file)
- Directories contain (name, inode #) pairs

Storing Data Block Locations

Inode contains an array of 15 LBAs

- 12 Direct – the blocks referenced contain data
- 1 indirect, 1 doubly-indirect, and 1 triply-indirect



Traditional Unix File System

- Superblock
  - Contains meta-information for the file system
    - Similar to the BPB in FAT
  - Blocksize, number of inodes, etc.
  - “Cylinder group” locations
    - Idea: put inode and file data close to each other on disk; avoid seeking!
  - Locations of backup superblock copies (blocks in data area)
  - Root directory inode pointer, free block bitmap

FFS Features

(These are shared by all modern file systems, including NTFS)

- Multiple names for the same file/directory
  - More than one directory entry can have the same inode in it
  - All of the names have exactly the same status
  - When the link count in the inode goes to zero, OS frees the inode and associated resources.
- “Symbolic link” – a special file that contains a pathname (= pointer to another file)
  - Creates the possibility of dangling pointers

FFS Features

- File Locking: file system provides exclusive access (via OS API!) to file data