Chapter 39: Interlude: Files and Directories

- Virtualization of the CPU, Virtualization of memory, and persistent storage make up the key components of an operating system
- A persistent storage device stores information permanently/for a long time

39.1: Files and Directories

- **File:** simply a linear array of bytes each of which you can read or write
- **Low-level name:** usually a number of some kind -> inode number
- The OS does not know much about the structure of a file (file format like image, or txt, etc).
    - the responsibility of the file system is simply to store such data persistently on the disk
- **Directory:** Also has a low level name (inode), but its contents include
    - a list of (user-readable name, low-level) name pairs
    - By placing directories within other directories, users are able to build an arbitrary directory tree
- **absolute pathname:** /foo/bar.txt is an example

39.3: Creating Files

- Creating a file can be accomplished through using the open() system call, passing in O_CREAT
    - int fd = open("foo", O_CREAT|O_WRONLY | O_TRUNC. S_IRUSR|S_IWUSR)
- **File descriptor:** a pointer to an object of type file -> then you can call methods such as read() and write()

39.4: Reading and Writing Files

- ex: echo hello > foo cat foo
    - How does the program cat access the file foo?
- Linux uses strace, Mac uses dtruss, and truss on some older UNIX variants
- Provides a way t see what programs are up to
    - sees what system calls a program makes, the arguments, and return codes, etc.
- program opens the first available file descriptor (0 for STDIN, 1 for STDOUT, 2 for STDERR)

39.5: Reading and Writing, But Not Sequentially

- Example: Reading from a particular offset like looking up a specific word in a text document
- off_t lseek(int fd, off_t offset, int whence)
    - **offset positions the file offset to a particular location within the file**
    - **whence determines exactly how the seek is performed**
    - SEEK_SET sets the offset to offset bytes
    - SEEK_CUR sets the offset to its current location + offset bytes
    - SEEK_END sets the offset to the size of the file plus offset bytes

- Calling lseek() does not perform a disk seek
    - simply changes a variable in OS memory that tracks, for a particular process, at which offset its next read or write will start
    - A disk seek occurs when a read or write issued to the disk is not on the same track as the last read or write, and thus necessitates a head movement
    - lseek does not necessarily cause disk I/O itself
- lseek is different from the seek operation, which moves the disk arm

39.6: Writing Immediately with fsync()

- Most times when a program calls write(), it is just telling the file system to write the data to the persistent storage at some point in the future
    - This means that although to the user, it seems as if the write completes quickly, it is not exactly done immediately
- Some applications may require that a write is done immediately
    - In a database management system, development of a correct recovery protocol requires the ability to force writes to disk from time to time
- When a process calls fsync() the file system responds by forcing all dirty (not yet written) data to disk, for the file referred to by the specified file descriptor
- In summary, fsync makes sure that the writes are forced immediately onto disk

39.7: Renaming Files

- rename (char* old, char* new)
- rename() is implemented as an atomic call with respect to system crashes
    - if the system crashes during renaming, the file will either be named the old name or the new name -> no inbetween

39.8: Getting Information about Files

- Information about each file that is being stored in the file system is called the metadata
- use the stat() or fstat() command to see the metadata of the file

39.9: Removing Files

- the system call that the UNIX command rm uses to remove a file is the unlink() call

39.10: Making Directories

- You can never write to a directory directly because the format of the directory is considered file system metadata and you can only update a directory indirectly by, creating files, directories, or other object types within it
- mkdir() system call
    - empty directory has two entries: one that refers to itself and one entry that refers to its parent (./ and ../)

39.11: Reading Directories

- when we do the ls command on UNIX we are really making calls to opendir(), readdir(), and closedir()

39.12: Deleting Directories
- rmdir() call to remove directories and it must be an empty directory in order for you to be able to delete it

39.13: Hard Links
- the link() system call lets you make an entry in the file system tree
- link() system call takes two params, an old pathname and a new one
    - when you link a new file name to an old one, you essentially create another way to refer to the same file
- The way link works is that it simply creates another name in the directory you are creating the link to, and refers it to the same inode number of the original file
    - you simply have two names that refer to the same inode number, or the same low-level name/file
- When you create a file you are really doing two things
    - First, you create a structure (inode) that will track all relevant information about the file
    - Second, you link that inode to the file and putting that file in the directory
- ex: ln file file2
    - if you rm file -> you can still access contents through using cat file2
- When the file system unlinks file, it checks a reference count within that inode number
    - Only when the reference/link count reaches 0, does the inode itself get removed as well
    - you can view the reference count of the file through the stat() command as well

39.14: Symbolic Links
- Hard links are somewhat limited: you can’t create one to a directory and you can’t hard link to files in other disk partitions because inode numbers are only unique within a particular file system, not across file systems
- use ln -s flag
- Symbolic link is actually a file itself, of a different type
- The way a symbolic link is formed is by holding the pathname of the linked-to file as the data of the link file
- Possibility of a **dangling reference**
    - removing a soft link causes the link to point to a pathname that no longer exists

39.15: Making and Mounting a File System
- How to assemble a full directory tree from many underlying file systems
- To make a file system, most file systems provide a tool, usually referred to as mkfs (make fs), that performs exactly this task.

- give the makefs command an input and a file system type and it creates an empty file system
- ex: Imagine we have an unmounted ext3 file system, stored in device partition /dev/sdal
    - has a root directory which contains two sub-directories, a and b, each of which in turn holds a single file named foo
    - We wish to mount this file system at the mount point /home/users
    - mount -t ext3 /dev/sdal /home/users

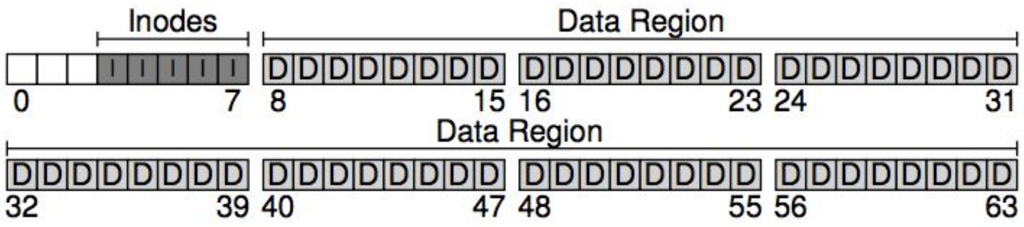
Chapter 40: File System Implementation

- Very Simple File System: This file system is a simplified version of a typical UNIX file system
- The file system is pure software, we do not need to introduce new hardware support unlike when we virtualized the CPU and memory

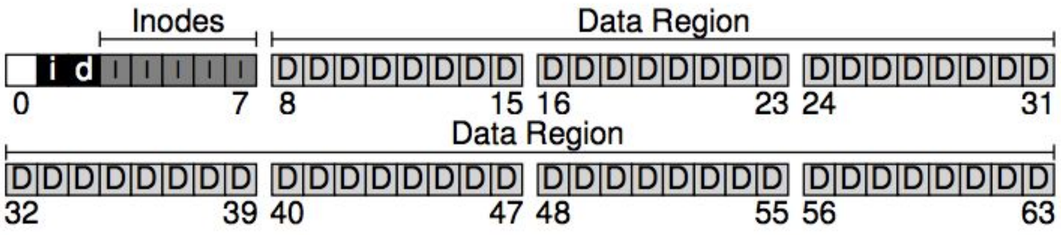
40.1: The Way to Think

- **Data structures:** What types of on-disk structures are utilized by the file system to organize its data and metadata
- **Access methods:** How does it map the calls made by a process, such as open(), read, write(), etc onto its structures?
    - Which structures are read during the execution of a particular system call? Which are written? How efficiently are these steps performed?

40.2: Overall Organization

- Divide the disk into **blocks;** simple file systems use just one block size (commonly used such as 4KB)
- Most of the space in any file system is user data
- File systems use inodes to stores key pieces of metadata such as the size of the file, access and modify times, and other kinds of info
- 

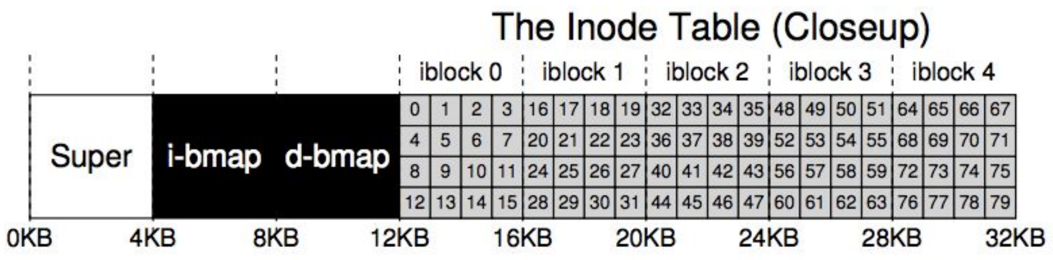
- Therefore, our file system thus far has I (inodes) D (data blocks)
- **Allocation structures:** ways to keep track of whether inodes or data blocks are free
    - we could use something like a free list that points to the first free block, which then points to the next free block and so forth.
    - **Bitmap:** each bit is used to indicate whether the corresponding object/block is free (0) or in-use (1)



-
- **Superblock:** Contains information about the particular file system, including how many inodes and data blocks are in the file system, and so forth
  - Likely includes a magic number of some kind to identify the file system type
- Thus, when mounting a file system, the operating system will read the superblock first, to initialize various parameters, and then attach the volume to the file-system tree

40.3: File Organization: The Inode

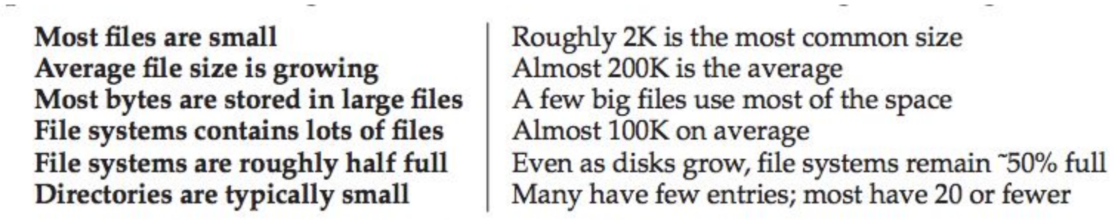
- inode is short for index node
- Each inode is implicitly referred to by a number (called the inumber)
- in vsfs, given an i-number, you should be able to directly calculate where on the disk the corresponding inode is located
- Ex: 20KB in size (5 4KB-blocks) and thus consists of 80 inodes;
  - assume each inode region starts at 12KB (first 12KB reserved for superblock, i-bmap, and d-bmap)



-
- Above shows the layout for the beginning of the file system
- To read inode number 32, the file system would first calculate the offset into the inode region (32*sizeof(inode) or 8192), add it to the start of the inode table (12KB), and thus arrive upon the correct byte address of the desired block of inodes -> 20KB
  - Disks are not byte addressable, but consist of a large number of addressable sectors, usually 512 bytes -> we need to fetch the block of inodes that contains inode number 32
  - blk = (inumber*sizeof(inode_t))/blockSize
  - sector = ((blk*blockSize) + inodeStartAddr) /sectorSize
- Inside each inode is virtually all of the info you need about a file such as its type, its size, the number of blocks allocated to it, protection information, time information, where its data blocks reside on disk, etc.

The Multi-Level Index

- To support bigger files, file system designers have had to introduce different structures within inodes.
    - **Indirect pointer:** Instead of pointing to a block that contains user data, it points to a block that contains more pointer, each of which point to user data
    - An inode can therefore have some fixed number of direct pointers to user data, but with one indirect pointer
    - If a file grows large enough, an indirect block is allocated (from the user data block region of the disk) and the inode's slot for an indirect pointer is set to point to it
- To support even larger files, you can add another pointer to the inode -> **double indirect pointer**
    - This pointer refers to a block that contains pointers to indirect blocks, each of which contain pointers to direct pointers to data blocks
    - You may even want more -> triple indirect pointer
    - This approach is referred to as the multi-level index approach to pointing to file blocks
- Aside: Extents
    - an extent is simply a disk pointer plus a length (in blocks)
    - instead of requiring a pointer for every block of a file, all one needs is a pointer and a length to specify the on-disk location of a file
- Why use an imbalanced tree?
    - most files are small -> with a small number of direct pointers, an inode can directly point to 48KB of data, needing one or more indirect blocks for larger files



Most files are small	Roughly 2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of the space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ˜50% full
Directories are typically small	Many have few entries; most have 20 or fewer

Figure 40.2: **File System Measurement Summary**

-

40.4: Directory Organization

- In vsfs (as in many file systems), directories have a simple organization; basically just contains a list of (entry name, inode number) pairs
- For each file or directory in a given directory, there is a string and a number in the data blocks of the directory
- Deleting a file (e.g. calling unlink()) can leave an empty space in the middle of the directory, and hence there should be some way to mark that as well (e.g. with a reserved inode number such as 0)
- Where exactly are directories stored?
    - File systems treat directories as a special type of file

    - Thus, a directory has an entry somewhere in the inode table labelled as directory instead of regular file

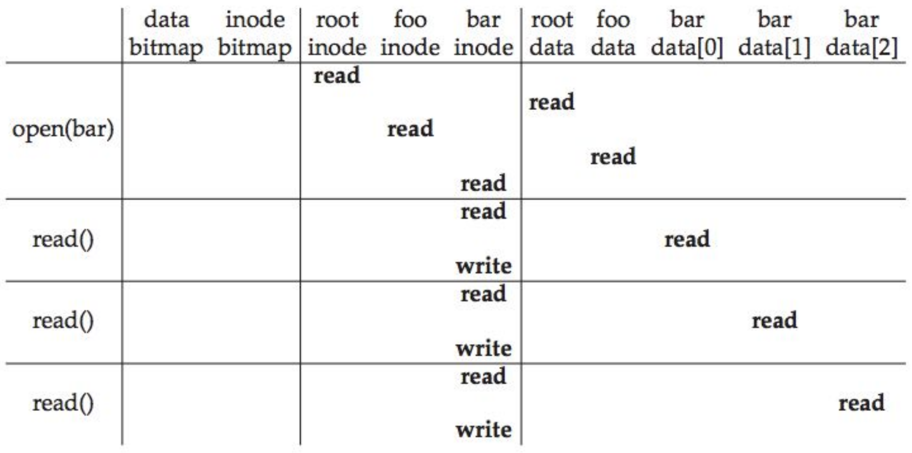
40.5: Free Space Management

- A file system must track which inodes and data blocks are free, and which are not, so that when a new file or directory is allocated, it can find space for it
- When we create a file, we have to allocate an inode for that file
    - The file system will thus search through the bitmap for an inode that is free, and allocate it to the file -> then the file system will have to mark the inode as used
    - Eventually must update the on-disk bitmap with the correct information
- Linux systems look for a sequence of blocks (say 8) that are free when a new file is created and needs data blocks
    - by finding a sequence of blocks and then allocating them to the newly created file, the file system guarantees that a portion of the file will be contiguous on the disk -> improves performance -> **pre-allocation policy**

40.6: Access Paths: Reading and Writing

- For the following examples, assume that the file system has been mounted and that the superblock is already in memory, but everything else is still on disk
- **Reading a File From Disk**
    - assume that we want to simply open a file, read it, and then close it
    - when you issue an open call, the file system first needs to find the inode for the file, to obtain some basic info about the file
    - The file system must traverse the file pathname and thus locate the desired inode
    - The first thing that the file system will read is the **root directory called / and thus find the inode of the root directory**
    - The FS must know what the root inumber is when the file system is mounted. In most UNIX file systems, the root inode number is 2
    - Once the inode is read in, the FS can look inside of it to find pointers to data blocks, which contain the contents of the root directory
    - The FS will use these pointers to look for the specific entry and therefore also find the inode number
    - We must then recursively traverse the pathname until the desired inode is found
    - Finds the file, calls open() to read the inode into memory -> does permissions check -> allocates a file descriptor for this process in the per-process open-file table -> returns to user
- **Aside: Reads don’t access allocation structures**
    - When you are simply reading a file, the bitmap is not consulted because allocation structures are only used when allocation is needed
        - Inodes, directories, and indirect blocks already have all the information they need to complete a read request
        - No need to make sure a block is allocated when the inode already points to it

    - No disk I/O takes place during this time



	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			**read**							
						read				
				read						
							read			
					read					
read()					**read**					
								read		
					write					
read()					**read**					
									read	
					write					
read()					**read**					
										read
					write					

Figure 40.3: **File Read Timeline (Time Increasing Downward)**

    -
    - the write in the read() updates the inode's last accessed time field with a write
    - The amount of I/O generated by the open is proportional to the length of the pathname -> for each additional directory in the path, we have to read its inode as well as its data -> problems can occur with the presence of large directories
- **Writing to Disk**
    - Unlike reading, writing to the file may also allocate a block (unless the block is being overwritten)
    - When writing out a new file, each write not only has to write data to the disk, it also has to decide which block to allocate to the file and thus update structures of the disk accordingly (e.g. data bitmap and inode)
    - Thus, each write to a file logically generates five I/Os:
        - one to read the data bitmap (which is then updated to mark the newly allocated block as used)
        - one to write the bitmap (to reflect its new state to disk)
        - two more to read and then write the inode
        - and one to write the actual block itself
    - The amount of traffic gets worse when one considers a simple operation such as creating a file
        - must allocate an inode, but also allocate space within the directory containing the new file
            - one read to the inode bitmap (to find a free inode)
            - one write to the inode bitmap (to mark allocated)
            - one write to the new inode itself (to initialize)
            - one to the data of the new directory (link high-level name of the file to its inode number)
            - one read and write to the directory inode to update it

40.7: Caching and Buffering

- Reading and writing files can be expensive, incurring many I/Os to the slow disk
- Most file systems aggressively use system memory (DRAM) to cache important blocks
- **Fixed size cache to hold popular blocks**
    - strategies such as LRU and different variants would decide which blocks to keep in the cache
    - Would usually be allocated at boot time to be roughly 10% of the total memory
- Because static partitioning of memory can be very wasteful, modern systems have implemented a dynamic partitioning approach
    - Many modern OS integrate **virtual memory pages and file system pages** into a **unified page cache**
    - This way, memory can be partitioned fairly across virtual memory pages and file system pages depending on which needs more memory
- Example: file open
    - first open may generate some I/O traffic but subsequent calls to the same file will result in cache hits, significantly speeding up the process -> no I/O needed
- **Write buffering**
    - Delaying writes allows the file system to do batch updates into a smaller set of I/Os
    - ex: if an inode bitmap is updated when one file is created and then updated moments later as another file is created, the file system saves an I/O by delaying the write after the first update
    - By buffering a number of bytes in memory, the system can then **schedule** the subsequent I/Os and thus increase performance
    - Some writes can be avoided altogether
        - if a write is done and then immediately is deleted -> delaying ensures that the file creation is just avoided to improve performance
    - Tradeoff: If there is a power loss/system crashes, the updates are lost if they are not propagated to disk
    - Databases do not want this tradeoff because they want to simply force writes onto disk by calling fsync()
        - using direct I/O interfaces that work around the cache, or by using a **raw disk interface** and avoiding the file system altogether
        -
- Tip: Advantages of Static vs Dynamic Partitioning
    - Advantages of Static
        - ensures each user receives some share of the resource, predictable performance, and easier to implement
    - Advantages of Dynamic
        - achieves better utilization
    - Disadvantages of Static
        - Can lead to fragmentation/wasting memory -> if not all of it is used it is wasted
    - Disadvantages of Dynamic

        - performance issues + complexity

File Types and Attributes

- Files are often described as byte streams or one-dimensional arrays

Ordinary Files

- **Text file:** a byte stream, but when we process it, we generally break it into lines through \n and render it as characters
- **Archive (zip/tar)** is a single file that contains many others. It is an alternating sequence of headers (that describe the next file in the archive) and data blobs
- **A load module:** similar to an archive (alternating sequence of section headers and contents) but the different sections represent different parts of a program (code, initialized data, symbol table, etc.)
- **MPEG stream:** sequence of audio, video, frames, containing compressed program information, which require considerable processing in order to reconstruct the encoded program
- Ordinary file is a blob of ones and zeroes that don't have meaning unless rendered by a program that understands the data format

Data Types and Associated Applications

- We must find the right program to interpret each file (or byte stream)
- require the user to specifically invoke the correct command to process data
    - ex: vi filename and compile with gcc filename
- Consult a registry that associates a program with a file type
    - may be a system-wide registry that associates programs with file types (e.g. Windows)
    - there may be a program-specific registry that associates programs with file types (e.g. configuring browser plug-ins)
    - the owning program may be an attribute of the file (Mac OS)
- We must know what type the file is
    - simplest approach is based on file name suffix (e.g. png txt, etc)
    - A magic number at the start of each file -> each type of file begins with a reserved and registered magic number that identifies the file's types
    - In systems that support **extended attributes**, the file type can be an attribute of the file

File Structure and Operations

- In some cases, the structure of the data is not merely an implementation decision, but fundamental to the manner in which the data is intended to be used
    - earliest databases were **indexed sequential files:** organized into records, each with a unique index key.
    - Relational databases

    - The complexity and non-scalability of SQL databases gave rise to much simpler key-value stores accessed only by get, put, and delete
- Implemented by some middleware layer on top of byte streams

Directories
- Directories do not contain blobs of client data -> rather they use name-spaces, or associations of names to blobs of data
- Similar to key value stores but instead of being restricted to a single user, directories contain files owned by numerous users, each of whom want to impose different sharing/privacy constraints on the access
- All directory operations tend to be implemented within the OS
    - mkdir, rmdir, link, unlink, create, and open -> involves considerable permission checking to ensure the integrity and security of the directory structure
- Directories can still be accessed with open(2) and read(2)

Inter-Process Communication Ports
- Data is exchanged via write(2) and read(2) system calls on file descriptors that can be manipulated with the dup(2) and close (2) operations. In the case of named pipes, they can be accessed via the open(2) system call
- With directories, we saw something that was implemented as on-disk byte streams but accessed through the OS in complex operations
- With IPC Ports, the implementation is very different but the access is the same as a regular file I/O

I/O Devices
- I/O devices connect a computer to the outside world.
- Many sequential access devices (e.g. keyboards and printers) are fit naturally into byte-stream read(2)/write(2) model
- Communication interfaces often behave like byte streams, which we can handle with ioctl(2) operations
- Not all I/O devices can be fit into a byte-stream model
    - Ex: 3D rendering engine comprised of a few thousand GPUs
    - Map gigabytes of display memory and control registers into our address space and manipulate them directly

File Attributes (Metadata)
System Attributes
- Unix/Linux Files all have a standard set of attributes
    - type: regular, directory, pipe, device, symbolic link, etc
    - ownership: identity of the owning user and owning group
    - protection: permitted access
    - when the file was created/updated/last accessed
    - size/ the number of bytes in the file

- All files have these attributes
- the OS depends on the se attributes to correctly implement access control
- the OS maintains these attributes

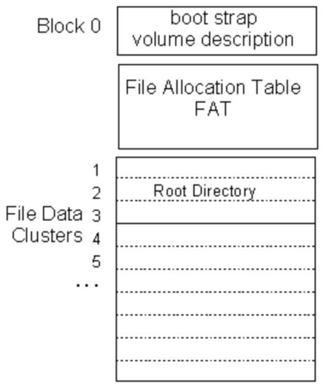
Extended Attributes

- There may be additional info that is vitally important to correct file processing
    - if the file has been encrypted or compress, by what algorithm?
    - If a file has been signed, what is the associated certificate?
    - If a file has been check-summed, what is the correct check-sum?
    - If a program has been internationalized, where are its localizations?
- Metadata that is not part of the file's contents but rather descriptive information that may be necessary to properly process the files' contents
    - associate a limited number/size of name=value attributes with each file
    - pair each file with one or more shadow files (resource forks) that contain additional resources and information

Introduction to DOS FAT Volume and File Structure

Structural Overview

- All file systems include a few basic types of data structures
    - **bootstrap**: code to be loaded into memory and executed when the computer is powered on. MVS volumes reserve the entire first track of the first cylinder for the boot strap
    - **volume descriptors:** information describing the size, type, and layout of the file system + in particular, how to find the other key metadata descriptors
    - **file descriptors:** information that describes a file (ownership, protection, etc.) and points where the actual data is stored in the disk
    - **free space descriptors:** lists of blocks of currently unused space that can be allocated to the files
    - **file name descriptors:** data structures that user-chosen names with each file
- DOS FAT file systems divide the volume into fixed-sized physical blocks, which are grouped into larger fixed-sized block clusters

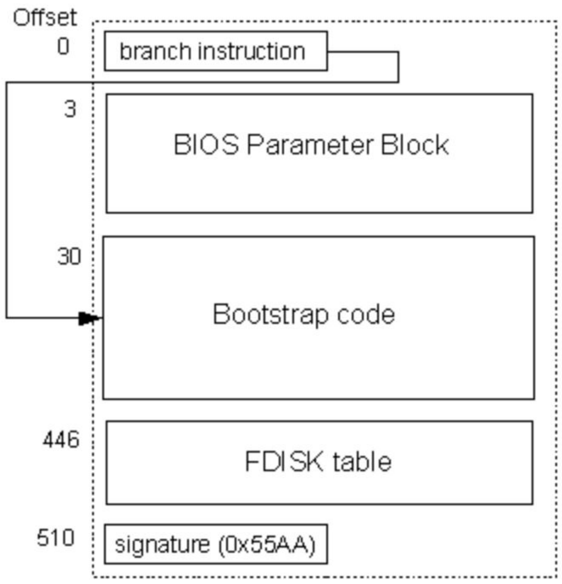


    -

    - File Allocation Table is used, both as a free list, and to keep track of which blocks have been allocated to which files

Boot Block BIOS Parameter Block and FDISK Table

- Most file systems separate the first block (mostly pure bootstrap code) from volume description information -> however, DOS file systems often combine these into a single block
- Boot record:
    - begins with a branch instruction (to the start of the real bootstrap code)
    - followed by a volume description (BIOS parameter block)
    - real bootstrap code
    - followed by an optional disk partitioning table
    - followed by a signature (for error checking)



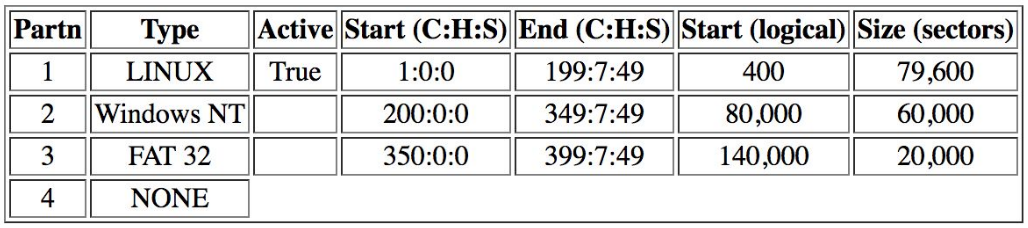
-

BIOS Parameter Block

- Comes after the branch instruction, containing a brief summary of the device and file system, describes basic device geometry
    - number of bytes per physical sector
    - number of sectors per track
    - number of tracks per cylinder
    - total number of sectors on the volume
- Also describes the way the file system is laid out on the volume
    - number of sectors per (logical) cluster
    - the number of reserved sectors (not part of the file system)
    - the number of Alternate File Allocation Tables
    - the number of entries in the root directory
- These parameters enable the OS to interpret the remainder of the file system

FDISK Table

- Small partition table called the FDISK table added to the end of the bootstrap block
    - used to put multiple file systems on the same disk
- Four entries, each capable of one disk partition
    - A partition type (Primary DOS partition, UNIX partition)
    - An ACTIVE indication
    - The disk address where that partition starts and ends
    - The number of sectors contained within that partition

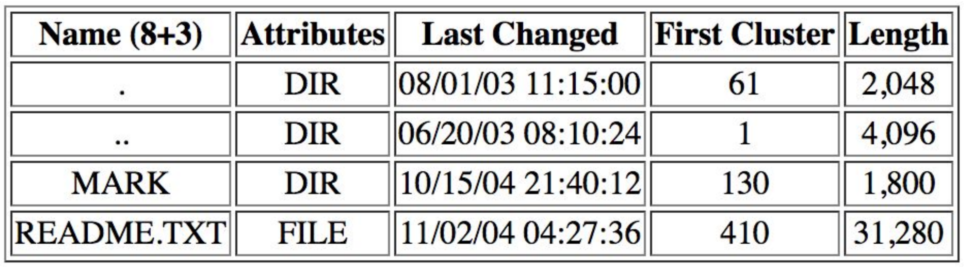


Partn	Type	Active	Start (C:H:S)	End (C:H:S)	Start (logical)	Size (sectors)
1	LINUX	True	1:0:0	199:7:49	400	79,600
2	Windows NT		200:0:0	349:7:49	80,000	60,000
3	FAT 32		350:0:0	399:7:49	140,000	20,000
4	NONE					

-
- The addition of disk partitioning also changed the structure of the boot record
- First sector of a disk contains the Master Boot Record which contains the FDISK table, and a bootstrap that finds the active partition, and reads in the first sector
- Most people (except Bill Gates) make their MBR bootstrap ask what system you want to boot from, and boothe active one by default after a few seconds

File Descriptors

- In keeping with their desire for simplicity, DOS file systems combine both file description and file naming into a single file descriptor
    - A DOS directory is a file that contains a series of fixed sized (32 byte) directory entries
    - each entry describes a single file
        - an 11-byte name
        - a byte of attribute bits for the file
            - is this a file, or a sub-directory
            - has this file changed since the last backup
            - is this file hidden
            - is this file read-only
            - is this a system file
            - does this entry describe a volume label
        - times and dates of creation and last modification, and date of last access
        - a pointer to the first logical block of the file
        - the length in the file
    - If the first character of a file's name is NULL, the directory entry is unused
- DOS stores file modification times and dates as a pair of 16-bit numbers
    - 7bits of year, 4 bits of month, 5 bits of day of month
    - 5 bits of hour, 6 bits of minute, 5 bits of second

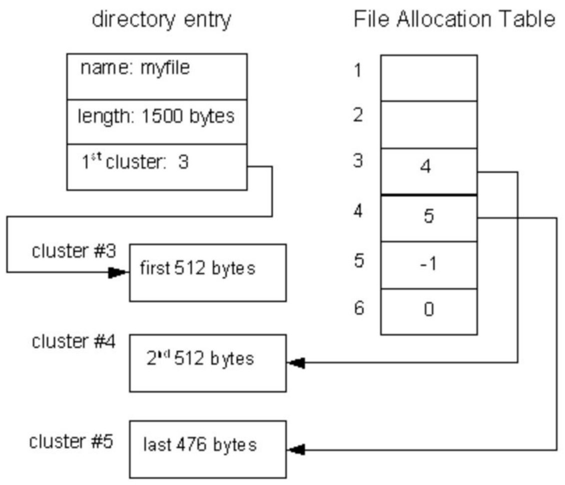
- 

Name (8+3)	Attributes	Last Changed	First Cluster	Length
.	DIR	08/01/03 11:15:00	61	2,048
..	DIR	06/20/03 08:10:24	1	4,096
MARK	DIR	10/15/04 21:40:12	130	1,800
README.TXT	FILE	11/02/04 04:27:36	410	31,280

Links and Free Space (File Allocation Table)
- Many file systems have very compact (e.g. bitmap) free lists, but most of them use some per-file data structure to keep track of which blocks are allocated to which file
- DOS File Allocation Table
  - contains one entry for each logical block in the volume
  - if a block is free, this is indicated by the FAT entry.
  - If a block is allocated to a file, the FAT entry gives the logical block number of the **next** logical block in the file

Cluster Size and Performance
- Space is allocated to files, not in physical blocks, but in logical multi-block clusters
- Allocating space to files in larger chunks improves I/O performance, by reducing the number of operations required to read or write a file
  - Comes at the cost of higher internal fragmentation
- The max number of clusters a volume can support depends on the width of the FAT entries
  - An 8-bit wide FAT entry would have been too small (256*512 - 128KB) to describe even the smallest floppy, but a 16-bit wide FAT entry would have been too large
  - Microsoft compromised by creating 12-bit wide FAT entries (FAT-12)
  - Also created 2 byte-wide (FAT-16) and 4-byte wide (FAT-32) file systems

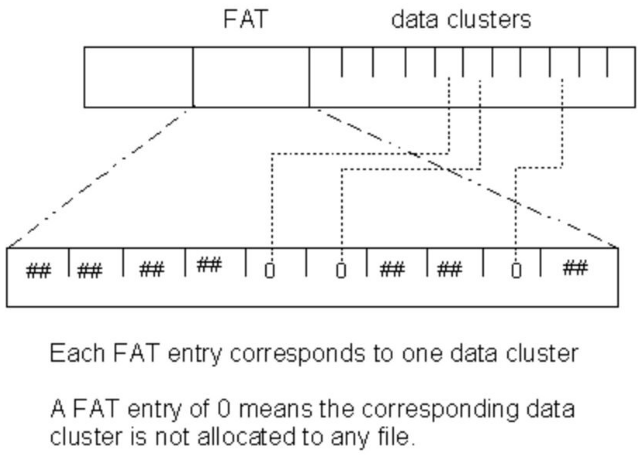
Next Block Pointers
- the file allocation table entry for that cluster tells us the cluster number next cluster in the file
- When we get to the final cluster of the file, its FAT entry will contain a -1, indicating that there is no block next



-
- The “next block”: organization of the FAT means that in order to figure out what physical cluster is the third logical block, we must know the physical cluster number of the second logical block
- If we had to go to disk to re-read the FAT each time we needed to figure out the next block number, the file system would perform very poorly
  - FAT is so small (e.g. 512 bytes per megabyte of file system) that the entire FAT can be kept in memory as long as a file system is in use
  - This means successor blocks can be looked up in memory without the need to do I/O

Free Space

- DOS reserves a value 0 to indicate that the cluster is free
- To find a free cluster, one has but to search the FAT for an entry with the value -2
- If we want to find a free cluster near the clusters that are already allocated to the file, we can start our search with the FAT entry after the entry for the first cluster in the file


FAT
data clusters
##
##
##
##
0
0
##
##
0
##
Each FAT entry corresponds to one data cluster
A FAT entry of 0 means the corresponding data cluster is not allocated to any file.

-

Garbage Collection

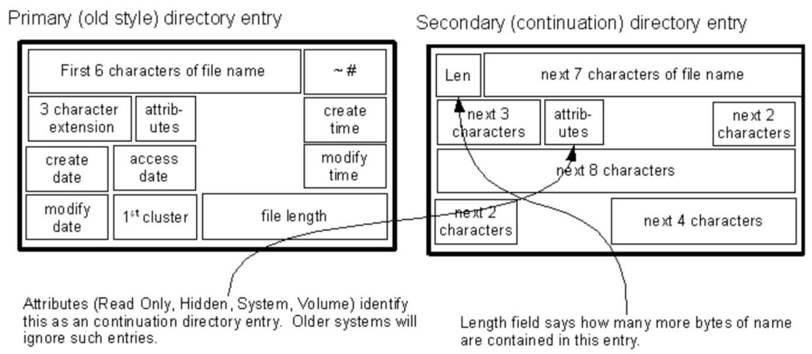
- Older versions of FAT file systems did not bother to free blocks when a file was deleted
- Starting from the root directory, DOS would find every “valid” entry
    - Follow the chain of next block pointers to determine which clusters were associated with each file, and recursively enumerate the contents of all sub-directories
    - Any cluster not found in some file was free, and marked them as such in the File Allocation Table
- Was possible to recover the contents of deleted files for a while because the enumeration of all allocated clusters/garbage collection
    - Because clusters are not freed when files are deleted, they could not be reallocated until after GC

Summary of FAT (File Allocation Tables): The table contains entries for a contiguous area of disk storage (cluster). Contains a pointer to the next cluster, or a marker indicating the end of the file, unused disk space, or special reserved areas of the disk. Describes the usage status of disk blocks

Descendants of the DOS file system

- Newer DOS file systems can support wider FAT entries to hold larger files and therefore additional features can be implemented

Longer File Names

- The 32 Byte directory entries didn’t have enough room to hold longer names, and changing the format of DOS directories could break hundreds of thousands of applications
- The DOS diskettes are commonly used to carry files between various systems -> means that old systems still had to be able to read the new directories
    - must support upwards compatibility
- Solution: Extended filenames in additional directory entries
    - Supplementary directory entries w/ extensions of the file names
- 
Primary (old style) directory entry
First 6 characters of file name
~#
3 character extension
attrib-utes
create time
create date
access date
modify time
modify date
1st cluster
file length
Secondary (continuation) directory entry
Len
next 7 characters of file name
next 3 characters
attrib-utes
next 2 characters
next 8 characters
next 2 characters
next 4 characters
Attributes (Read Only, Hidden, System, Volume) identify this as an continuation directory entry. Older systems will ignore such entries.
Length field says how many more bytes of name are contained in this entry.

Alternate/Back-up FATs

- The FAT is a very concise way of keeping track of all of the next-block pointers in the file system
    - If anything were to happen to the FAT, the results would be disastrous
    - The directory entries would tell us where the first blocks of all the files were, but we would have no way of figuring out where the remainder of the data was
- To add support for corruptions of the file allocation table, Microsoft added alternate FATs
    - Periodically, the FAT would be copied to one of the pre-reserved alternate FAT locations

6.3: ISO 9960

- Everyone recognized the importance of a single standard file system format because dueling file system formats would raise the cost of producing new products -> loss for everyone
- The International Standards Organization chartered a sub-committee to propose such a standard
- The most idiomatic features of the DOS file systems were irrelevant to a CDROM file system
    - We don't need to keep track of the free space on a CD ROM. We write each file contiguously, and the next file immediately goes after the next one
    - Because files can be written contiguously, we don't need any "next block" pointers
- It was decided that ISO 9660 file systems would have tree structured directory hierarchies, and that (like DOS) each directory entry would describe a single file (rather than having auxiliary data structures like and inode do this).
- 9660 directory entries contain
    - file names
    - file type
    - location of file's first block
    - number of bytes contained in the file
    - time and date of creation
- Advantages over DOS/DOS's mistakes
    - Realizing that new information would be added to directory entries over time, they made them variable length. Each directory begins with a length field
    - They also made the filename field a variable length field
    - Recognizing that over time, people would want to associate a wide range of attributes with files, they also created a variable length extended attributes section after the file name
        - Defined several new attributes for files
            - file owner
            - owning group
            - permissions
            - creation, notification, effective, and expiration times
            - record format, attributes, and length information

Summary of DOS

- DOS file systems are extremely simple
    - don’t support multiple links to a file, or symbolic link, or even multi-user access control
    - economical in terms of the space it takes up
        - free block lists and file block pointers are combined into a single compact FAT (file allocation table)
        - File descriptors are incorporated into directory entries
- Microsoft was finally forced to change the file system format to get past the 8.3 uppercase file name limitations -> used a kludgy but upwards compatible solution using additional directory entries per file

Key-Value Database (wiki)

- A data storage paradigm designed for storing, retrieving, and managing associate arrays, a data structure commonly known as hash
- RDBs vs Key-value databases
    -