## File System Layout

* Sector 0 of the disk → MBR (Master Boot Record)
* used to boot computer.
* End of MBR → partition table.
* Gives starting & ending addresses of each partition.
* One of the partitions marked as active.

# File System Layout

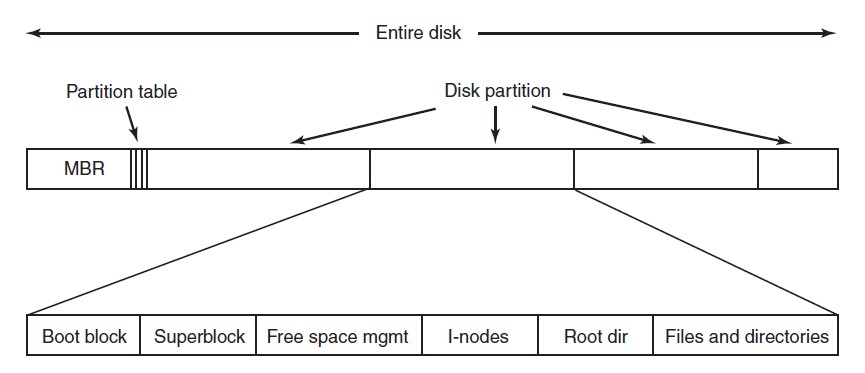


Fig. A possible file system layout.

## File System Layout (cont’d)

* When a computer is booted:
* BIOS reads in and executes Master Boot Record (MBR).
* The first thing the MBR program does is locate the active partition, read in its first block, which is called the boot block, and execute it.
* The program in the boot block loads the operating system contained in that partition.
* For uniformity, every partition starts with a **boot block**, even if it does not contain a bootable OS. Besides, it might contain one in the future.
* Other than starting with a boot block, the layout of a disk partition varies a lot from file system to file system.

### Items in a possible layout of a disk partition

* **Superblock**
* contains all key parameters about file system
* read into RAM when computer booted or file system first touched.
* typical information includes a magic number to identify file-system type, number of blocks in file system and other key admin info.
* **Information about free blocks in the file system**
* e.g. in the form of a bitmap or a list of pointers.
* **I-nodes**
* an array of data structures, one per file, telling all about the file.
* **Root directory**
* contains the top of the file-system tree.
* **All the other directories and files**

## Implementing Files

* Most important issue: keep track of which disk blocks go with which file.
* Various methods are used in different OSs.
* We examine a few of them in the coming slides:
* Contiguous Allocation
* Linked-List Allocation
* Linked-List Allocation Using a Table in Memory • I-nodes

### Implementing Files Contiguous Layout

* Simplest allocation scheme
* Stores each file as a contiguous run of disk blocks.

### Contiguous Layout (cont’d)

* Advantages
  1. **Simple to implement**: keeping track of where a file’s blocks → remembering: disk address of first block and # blocks occupied by file.
  2. **Excellent read performance**
* Only 1 seek needed (to first block).
* No more seeks or rotational delays after that → data come in at full bandwidth of the disk.
* Drawback
* Over course of time, disk becomes **fragmented**: the disk ultimately consists of files and holes (see Fig).
* Compacting disk → prohibitively expensive (moving millions of blocks).
* Reuse free space in the holes → doable but impractical (whenever a new file created, need to know its final size in order to choose hole of correct size to place it in).

#### Linked List Allocation

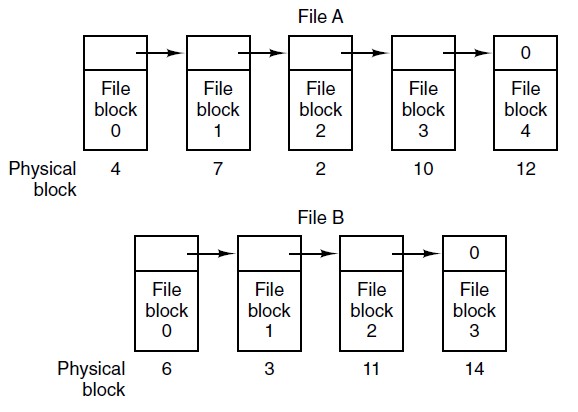


Fig. Storing a file as a linked list of disk blocks.

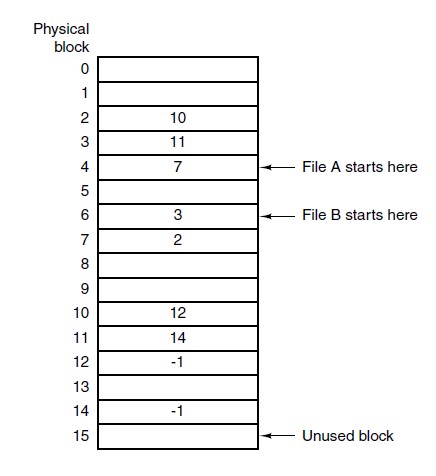
#### Linked List Allocation

* Advantages
* Unlike contiguous allocation, **every disk block can be used**.
* No space lost to disk fragmentation (except for internal fragmentation)
* Enough to store disk address of first block.
* Reading a file is straightforward
* Disadvantages
* **Random access extremely slow.**
* To get to block *n*, OS has to start at beginning and read *n*-1 blocks prior to it, one at a time. Doing so many reads painfully slow.
* Amount of data storage no longer a power of 2 (since pointer takes up a few bytes).
* Not fatal but many programs read/write in blocks whose size power of 2
* With few bytes occupied by pointer, reads of full block size require acquiring and concatenating info from 2 disk blocks → **extra overhead due to copying.**

#### Linked List – Table in Memory

* Eliminate both disadvantages of linked list allocation.
* Take pointer word from each disk block and put it in a table in memory → **File Allocation Table** (**FAT**).
* See Fig next slide.
* Advantages
* All the blocks available for data.
* Random access much easier.
* Although chain must still be followed, chain in entirely in memory → can be followed without making any disk references.
* Sufficient for directory entry to keep single integer (starting block number) and can locate all blocks, no matter the file size.

#### Linked List – Table in Memory

•Two files.

•File A uses blocks 4,7,2,10,12

•File B uses blocks 6,3,11,14 •Both chains terminated with special marker (“-1”).

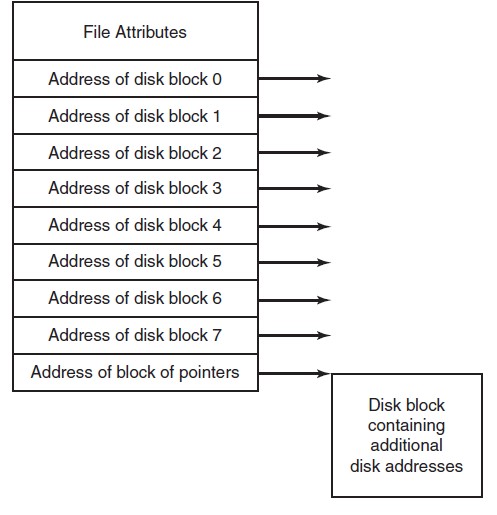
Fig. Linked list allocation using a file allocation table in main memory.

#### Linked List – Table in Memory (cont’d)

* Disadvantage
* **Entire table must be memory all the time**.
* E.g. how much does FAT table take in RAM **given a 200 GB disk and 1-KB block size**.
* The table needs 200 million entries (one for each of 200 million disk blocks)
* (200x109) / (1x103) = 200x106
* Each entry should be 4 bytes.
* log(200x10^6) / log(2) = 27.58 bits = 27.58 / 8 = 3.45 bytes
* **Table will take up 800 MB of RAM.**
* (200x10^6) entries x4 bytes = 800x10^6 bytes = 800 MB
* **Not practical and does not scale well to large disks.**

#### Implementing Files I-nodes

* Associate with each file a data structure called **i-node** (**index-node**).
* Lists attributes and disk addresses of the file’s blocks.
* See Fig next slide.
* Given the i-node of a file, can find all blocks of the file.
* Advantage over linked list allocation with in-memory table is that i-node need only be in memory when corresponding file is open.
* If each i-node occupies *n* bytes and max *k* files may be open at once, total memory needed = *kn* bytes only.
* Only this much space need be reserved in advance.
* Far smaller than FAT which grows proportionally with disk size.
* Memory required by i-node is independent of disk size.



I

-

nodes

Figure 4

-

13

. An

example i

-

node.

#### Implementing Files I-nodes (cont’d)

* Problem with i-nodes
* If each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit?
* Solution
* Reserve the last disk address not for a data block but instead for address of a block containing more disk block addresses.

# Implementing Directories

* Before file is read, must be opened.
* When file opened, OS uses path name supplied by user to locate directory entry.
* Directory entry provides info needed to find disk blocks.
* This info may be:
* The disk address of entire file (with contiguous allocation) • The number of the first block (both linked list schemes)
* The number of the i-node.
* **Main function of directory system is to map ASCII name of file onto info needed to locate the data.**

## Example: reading a file

* Assume you want to open “/foo/bar”, read and close it.
* For simplicity, assume its size is 4KB (i.e. 1 block).
* File system (FS) first needs to inode for the file “bar”
* to obtain some basic information about the file (permissions info, file size, etc.).
* To do so, FS must be able to find the inode, but all it has right now is the full pathname.
* FS must traverse pathname and locate desired inode.
* All traversals begin at the root of the file system, in the root directory which is simply called “/”.

## Example: reading a file (cont’d)

* Thus, first thing FS will read from disk is the inode of the root directory.
* But where is this inode? To find an inode, we must know its i-number. Usually, we find the i-number of a file or directory in its parent directory; the root has no parent (by definition).
* Thus, the root inode number must be “well known”; the FS must know what it is when the file system is mounted. In most UNIX file systems, the root inode number is 2.
* Thus, to begin the process, the FS reads in the block that contains inode number 2 (the first inode block).

## Example: reading a file (cont’d)

* Once the inode is read in, FS can look inside of it to find pointers to data blocks, which contain the contents of the root directory.
* FS will read through these looking for an entry for foo
* Once found, FS will also have found the inode number of foo (say it is 44) which it will need next.
* The next step is to recursively traverse the pathname until the desired inode is found.
* FS reads the block containing the inode of foo and then its directory data, finally finding the inode number of bar.
* Then contents of bar can be read from following pointers to data blocks inside the inode of bar.

Tanenbaum & Bo, Modern Operating

Systems:4th ed., (c) 2013 Prentice-

Journaling File SystemsHall, Inc. All rights reserved.

Steps to remove a file in UNIX:

* 1. Remove file from its directory.
  2. Release i-node to the pool of free i-nodes.
  3. Return all disk blocks to pool of free disk blocks.
* What happens if system crashes after 1st step?
* i-node and file blocks won’t be accessible from any file but will also not be available for reassignment
* just in limbo somewhere → decreasing available resources.
* Keep a log of what file system is going to do *before* it does it.
* If system crashes before it can do its planned work, upon rebooting, system can look in the log to see what was going on at the time of the crash and finish the job.

# Virtual File Systems

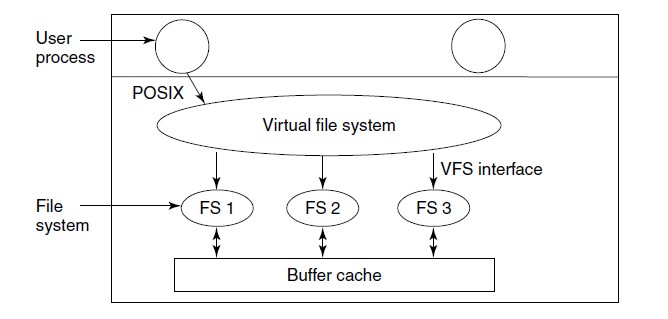


Figure 4-18. Position of the virtual file system.

# Keeping Track of Free Blocks

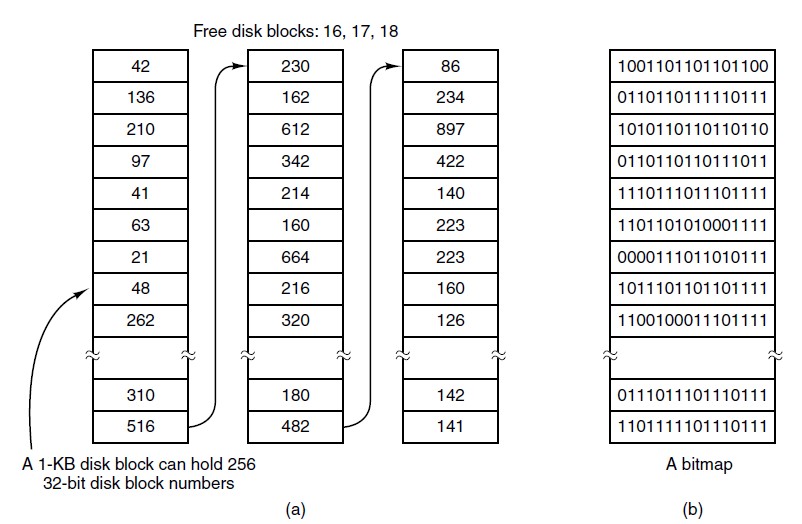
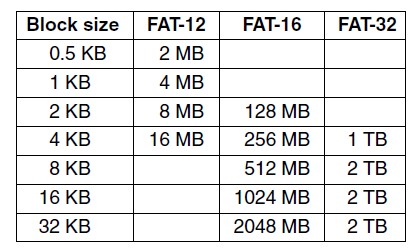


Fig. (a) Storing the free list on a linked list. (b) A bitmap.

# The MS-DOS File System (2)

Fig. Maximum partition size for different block sizes. The empty boxes represent forbidden combinations.

## Windows File Systems

## FAT16 and Extended FAT16

* Extended FAT16 evolved from FAT16 used in earlier versions of MS-DOS and Windows (3.x/95/98/Me)
* In extended FAT16:
* Maximum size of a volume is 4GB
* Maximum size of a file is 2GB
* Has been around for awhile and can be read by non-Windows operating systems like UNIX/Linux
* Considered a stable file system
* Long filenames (LFNs) can be used
* Can contain up to 255 characters
* Not case sensitive

# FAT32

* Support for FAT32 started with Windows 95 Release 2 • Designed to accommodate larger capacity disks
* FAT32:
* Root folder does not have to be at the beginning of a volume
* Can use disk space more efficiently than FAT16 (because it uses smaller cluster sizes)
* Largest volume that can be formatted is 32 GB
* Maximum file size is 4 GB
* Offers fast response on small 1 or 2 GB partitions

# FAT64

* FAT64 is also known as exFAT
* Proprietary file system introduced by Microsoft for mobile personal storage
* Good choice for USB flash devices that may store large files (such as pictures, videos, etc…)
* Available in Service Pack 1 for Windows Vista, Windows

7, and Windows Server 2008, Mac OS X Snow Leopard

* Support is available for Linux from a third party

# NTFS

* NTFS – dominant Windows file system for all Windows operating systems starting with Windows 2000
* Uses a Master File Table (MFT) instead of FAT tables
* The MFT and related files take up about 1 MB of disk space
* When a file is created, a record for that file is added to the

MFT

* Contains additional attributes such as security settings, ownership, and permissions

# NTFS

* The MFT record reflects the sequence of disk blocks that a file uses
* It is possible to have multiple filenames that refer to the same file
* A technique known as hard linking
* This feature is also available in UNIX/Linux file systems
* Windows Vista, Server 2008, and 7 use NTFS version 6
* Windows XP and Server 2003 use NTFS version 5
* Windows NT 4.0 used NTFS 4

# NTFS

* Basic features of NTFS:
* Long filenames
* Built-in security features
* Better file compression than FAT
* Ability to use larger disks and files than FAT
* File activity tracking for better recovery and stability than FAT
* **Portable Operating System Interface for Unix (POSIX)** support
* Volume striping and volume extensions
* Less disk fragmentation than FAT

# NTFS

* NTFS is equipped with security features that meet the US government’s C2 security specifications
* Refers to high-level, “top-secret” standards for data protection, system auditing, and system access
* Examples:
* System files can be protected so only the server administrator has access
* A folder of databases can be protected with read access, but no access to change data
* Public folder can give users in a designated group access to read and update files, but not to delete files

# NTFS

* Some files can be compressed by more than 40%, saving disk storage for other storage needs
* NTFS has the ability to keep a log or journal of file system activity (called **journaling**)
* Makes it possible for files to be restored in the event of a power failure
* NTFS supports **volume striping**
* A striped volume uses more than one physical hard disk to create a bigger volume.
* Faster than simple volume because reads and writes happen across multiple disks at the same time.
* Increased risk of catastrophic failure leading to data loss
* NTFS has **hot fix** capabilities
* If a bad disk area is detected, automatically copies the information to another disk area that is not damaged

CC216 Lecture 3 - File Systems

# NTFS

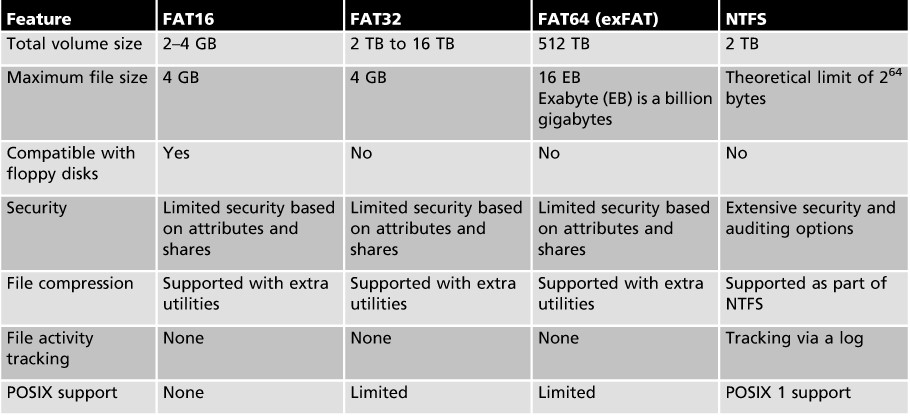


Fig: FAT16, FAT32, FAT64, and NTFS compared

## The UNIX File System

* Both ufs and ext use the same structure
* Built on the concept of information nodes (or **inodes**)
* Each file has an inode and is identified by an inode number • An inode contains general information about that file such as:
* User and group ownership, permissions, size and type of file, date the file was created, and the date the file was last modified and read
* Each disk is divided into logical blocks
* The **superblock** contains information about the layout of blocks, sectors, and cylinder groups on the file system
* The inode for a file contains a pointer (number) that tells the OS where to find a file on the hard disk (based on logical blocks)
* Inode 0 contains the root of the folder structure and is the jumping off point for all other inodes

CC216 Lecture 3 - File Systems

## Summary

* When seen from the outside, a file system is a collection of files and directories, plus operations on them.
* Files can be read and written, directories can be created and destroyed, and files can be moved from directory to directory.
* Most modern file systems support a hierarchical directory system in which directories may have subdirectories and these may have subsubdirectories ad infinitum.
* When seen from the inside, a file system looks quite different. The file system designers have to be concerned with how storage is allocated, and how the system keeps track of which block goes with which file.

## Chapter Summary

* Possibilities include contiguous files, linked lists, fileallocation tables, and i-nodes.
* Different systems have different directory structures.
* Disk space can be managed using free lists or bitmaps.
* The main file systems used in Windows since Windows 2000 are extended FAT16, FAT32, and NTFS.
* NTFS is the native file system for Windows 2000 and after with the advantage of better security, larger disk and file sizes, better management tools, and greater stability than FAT16 and FAT32.
* UNIX and Linux support many support many different file systems but typically employ ufs or ext.
* ufs and ext use information nodes (inodes) to organize information about files.