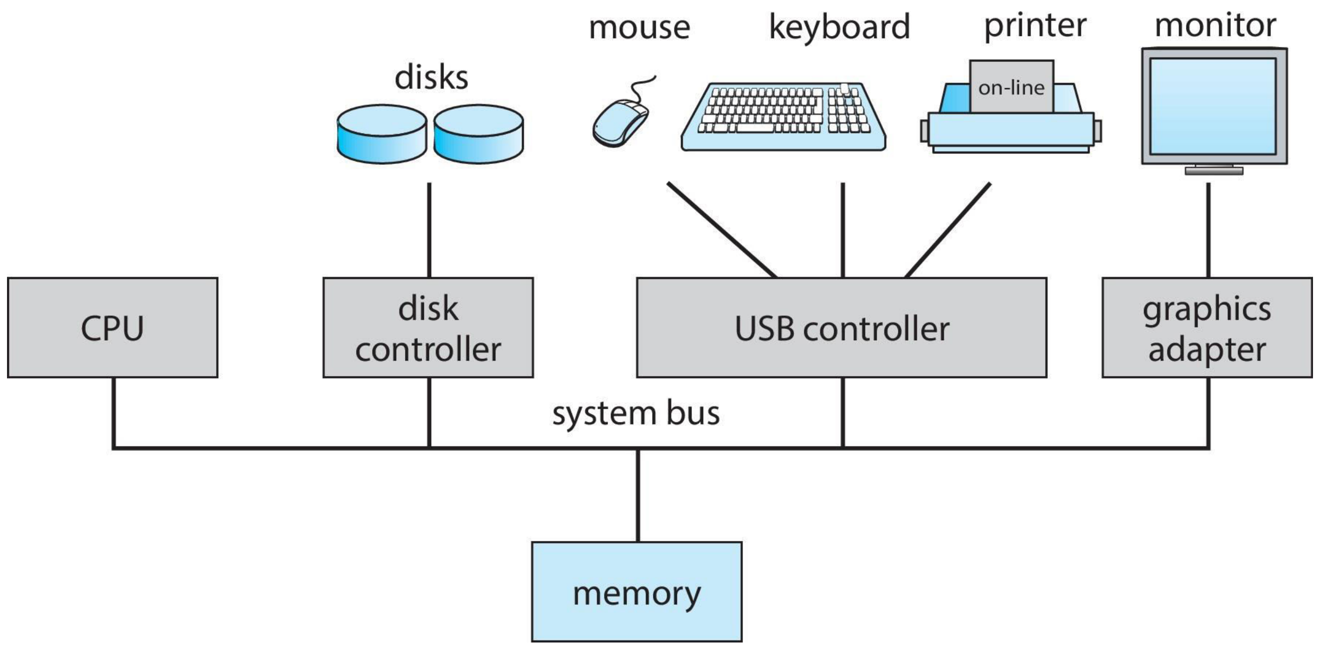
File systems

- Secondary storage devices
  - Types of storage devices
  - Disk structure
  - Disk formatting
- Files
- File systems
  - Definition of “file system”
  - Address mapping
  - Strategies for allocating disk space to files
- Windows file systems
  - FAT 12,16, 32
  - NTFS
- Linux file systems
  - Linux file structure on disk
  - ext2, ext3, ext4, nfs
- Boot sequence

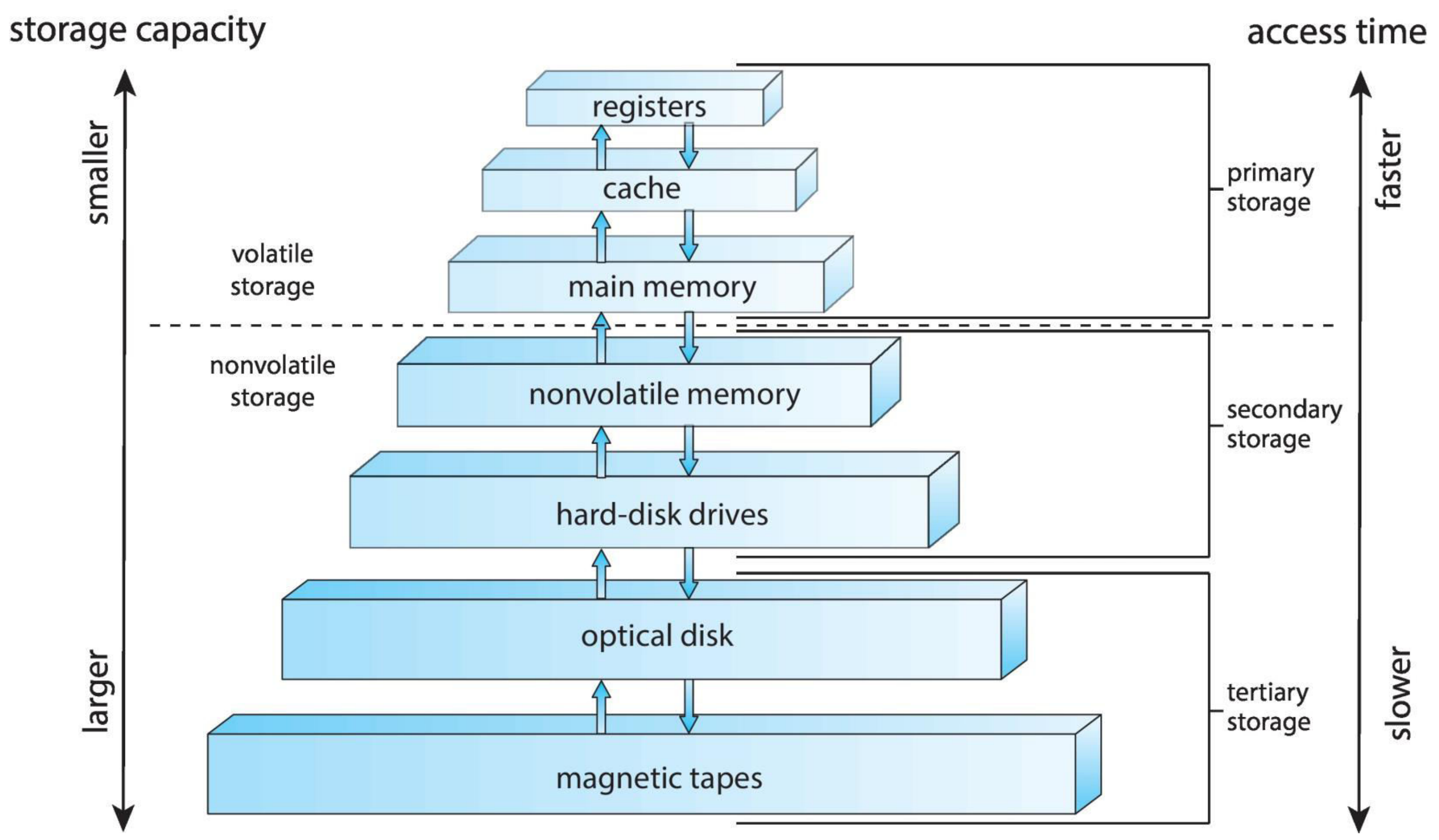
- This section of the course is based on chapters 11, 13, 14 and 15 in the book Operating System Concepts, tenth ed., and chapter 4 in the book Modern Operating Systems, third ed.

Computer System Hardware

- Computer-system hardware
  - One or more CPUs, device controllers connect through common bus providing access to the memory and other I/O devices



Storage-device hierarchy

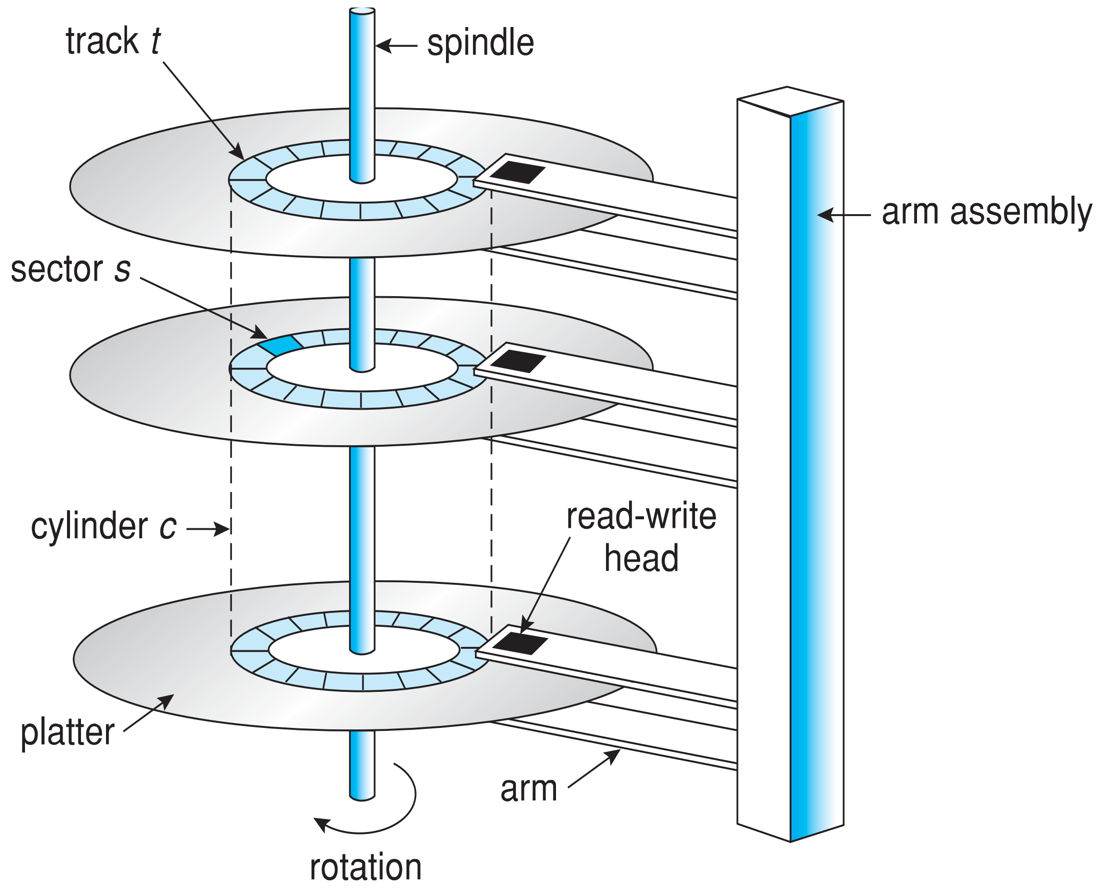


Mass storage devices

- hard disk drives (HDDs) and
- nonvolatile memory (NVM) devices, mainly solid-state disks (SSDs)

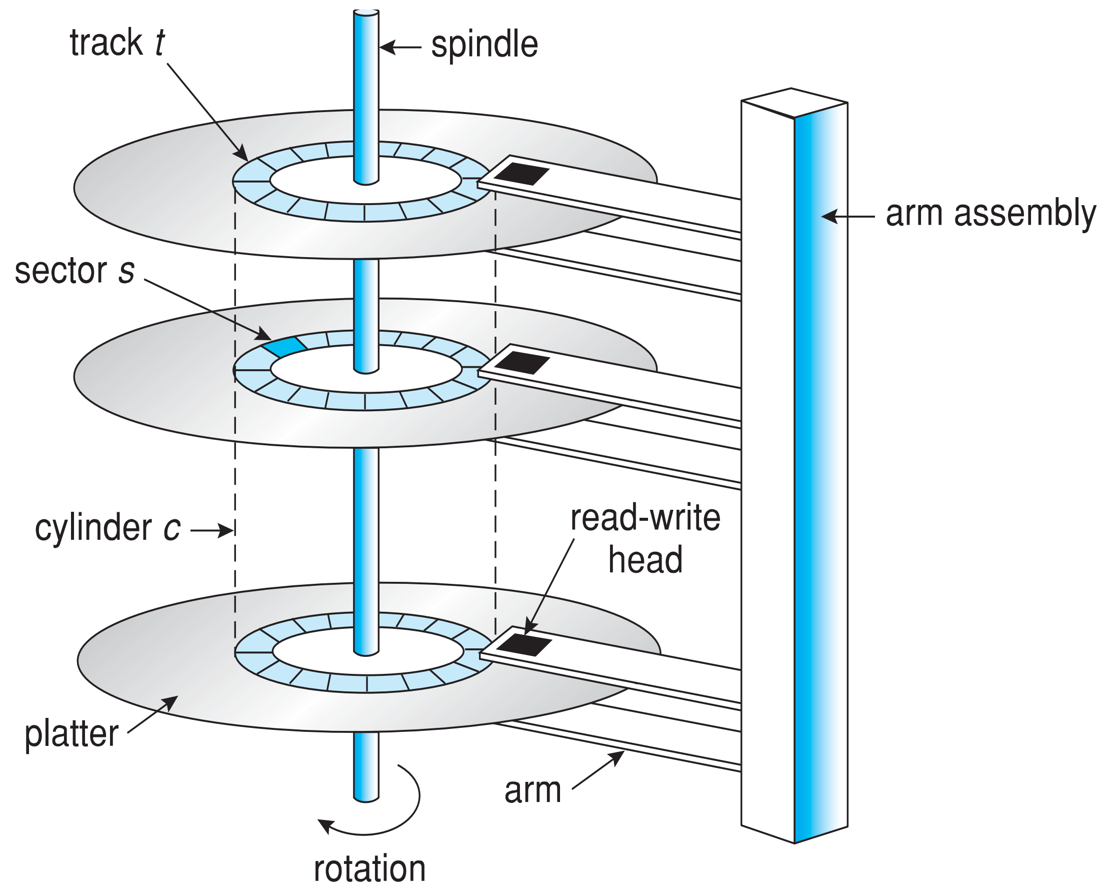
Hard disk drives

- HDDs spin platters of magnetically-coated material under moving read-write heads
- Components:
    - arm
    - read-write head
    - platter
    - track
    - cylinder
    - sector



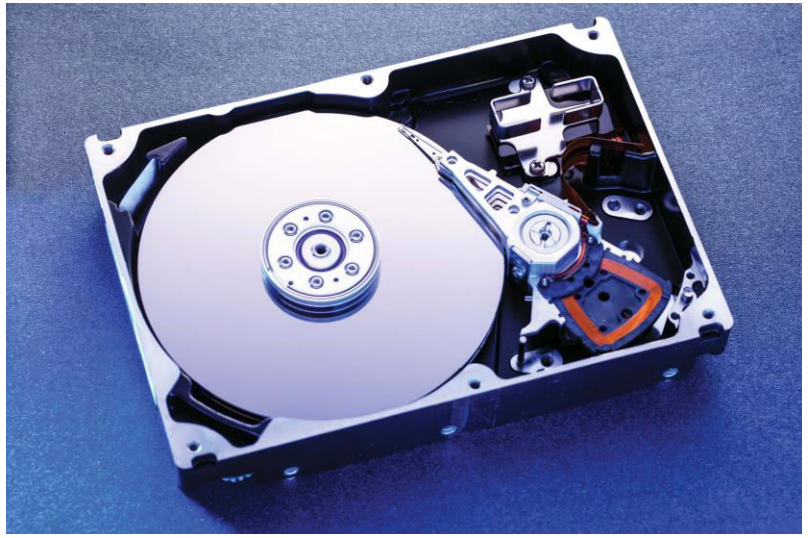
Hard disk drives

- Drives rotate at 60 to 250 times per second
- Transfer rate: rate at which data flow between drive and computer
- Positioning time (random-access time):
  - time to move disk arm to desired cylinder (seek time) and
  - time for desired sector to rotate under the disk head (rotational latency)



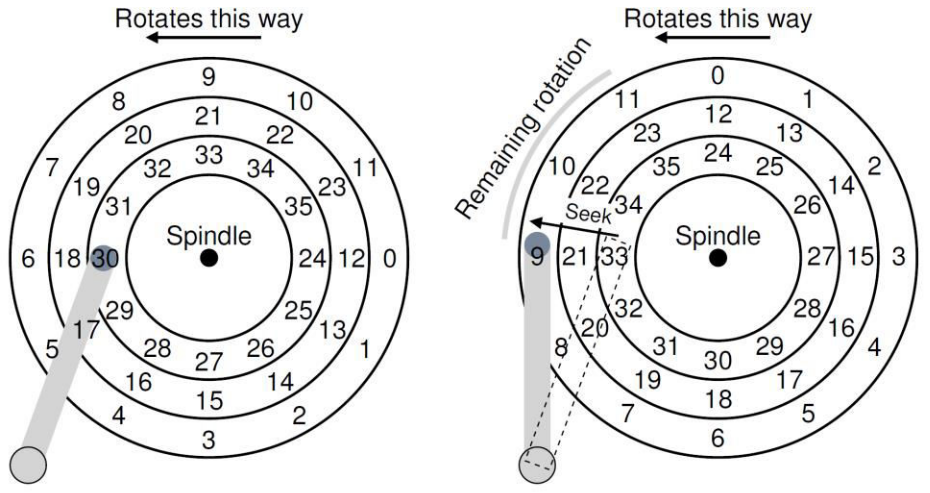
Hard Disk Drives

- Platters range from .85" to 14" (historically)
  - Commonly 3.5", 2.5", and 1.8"
- Range from 30GB to 18TB per drive
- Performance
  - Transfer Rate – theoretical – 6 Gb/sec
  - Effective Transfer Rate – real – 1Gb/sec
  - Seek time from 3ms to 12ms – 9ms common for desktop drives
  - Average seek time measured or calculated based on 1/3 of tracks
  - Latency based on spindle speed
    - 1 / (RPM / 60) = 60 / RPM
  - Average latency = ½ latency

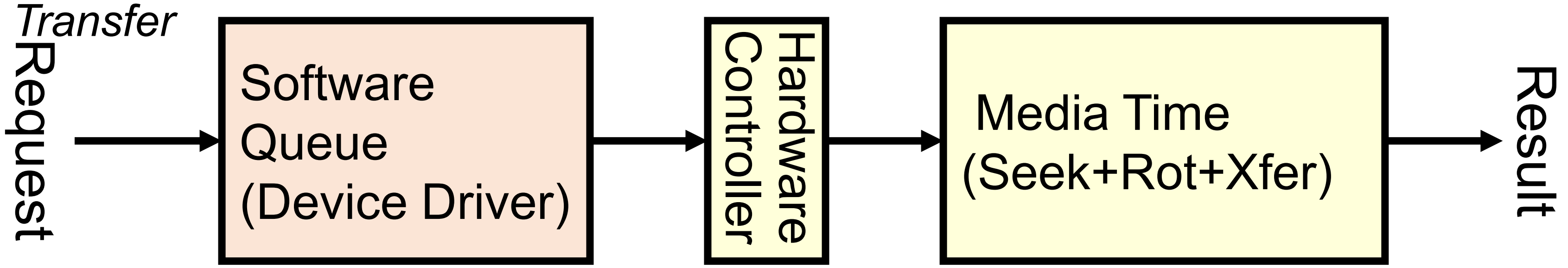


Performance of HDDs

- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track
  - Rotational latency: wait for desired sector to rotate under r/w head
  - Transfer time: transfer a block of bits (sector) under r/w head



Request Time = Queueing Time + Controller Time + Seek + Rotational + Transfer



Example of Current HDDs

- Seagate Exos X18 (2020)
  - 18 TB hard disk
    - 9 platters, 18 heads
    - Helium filled: reduce friction and power
  - 4.16ms average seek time
  - 4096 byte physical sectors
  - 7200 RPMs
  - Dual 6 Gbps SATA /12Gbps SAS interface
    - 270MB/s MAX transfer rate
    - Cache size: 256MB
  - Price: $ 562 (~ $0.03/GB)

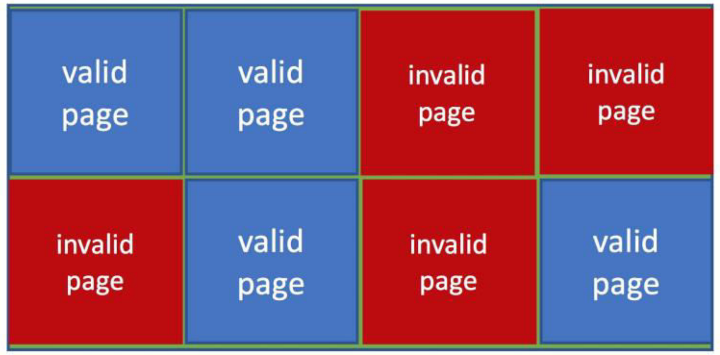
- IBM Personal Computer/AT (1986)
  - 30 MB hard disk
  - 30-40ms seek time
  - 0.7-1 MB/s (est.)
  - Price: $500 ($17K/GB, 340,000x more expensive !!)

Nonvolatile memory (NVM) devices

- If disk-drive like, then called **solid-state disks** (**SSDs**)
- Other forms include **USB drives**
- Can be more reliable than HDDs
- More expensive per MB
- May have a shorter life span – need careful management
- But much faster
- Busses can be too slow -> connect directly to PCI for example
- No moving parts, so no seek time or rotational latency

Nonvolatile Memory Devices

- Based on NAND semiconductors technology, have some characteristics that present their own storage and reliability challenges
- Read and written in “page” increments (think sector) but can’t overwrite in place
  - Must first be erased, and erases can only be executed in ”block” increments
  - Can only be erased a limited number of times before wearing out – ~ 100,000
  - Life span measured in **drive writes per day** (**DWPD**)
    - A 1TB NAND drive with a rating of 5DWPD is expected to have 5TB per day written within the warranty period without failing
- With no overwrite, pages end up with mix of valid and invalid data

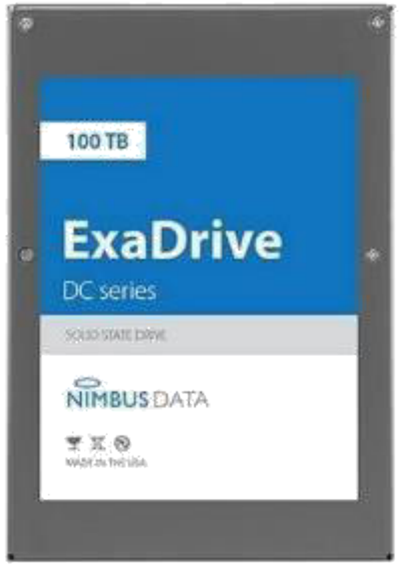


NAND block with valid and invalid pages

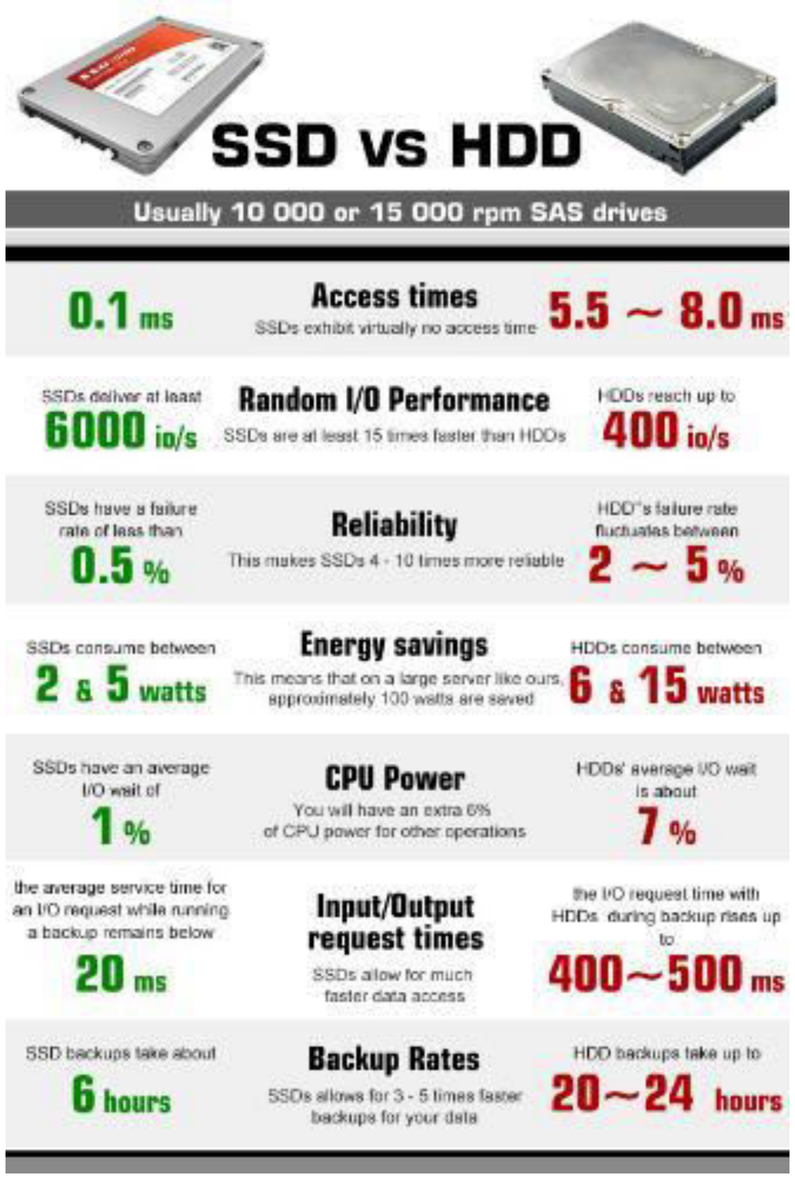
Some “Current” (large) 3.5in SSDs

- Seagate Exos SSD: 15.36TB (2017)
  - Seq reads 860MB/s
  - Seq writes 920MB/s
  - Price (Amazon): $5495 ($0.36/GB)

- Nimbus SSD: 100TB (2019)
  - Seq reads/writes: 500MB/s
  - Random Read Ops (IOPS): 100K
  - *Unlimited writes for 5 years!*
  - Price: ~ $40K? ($0.4/GB)
    - However, 50TB drive costs $12500 ($0.25/GB)



HDD vs. SSD Comparison



SSD vs HDD

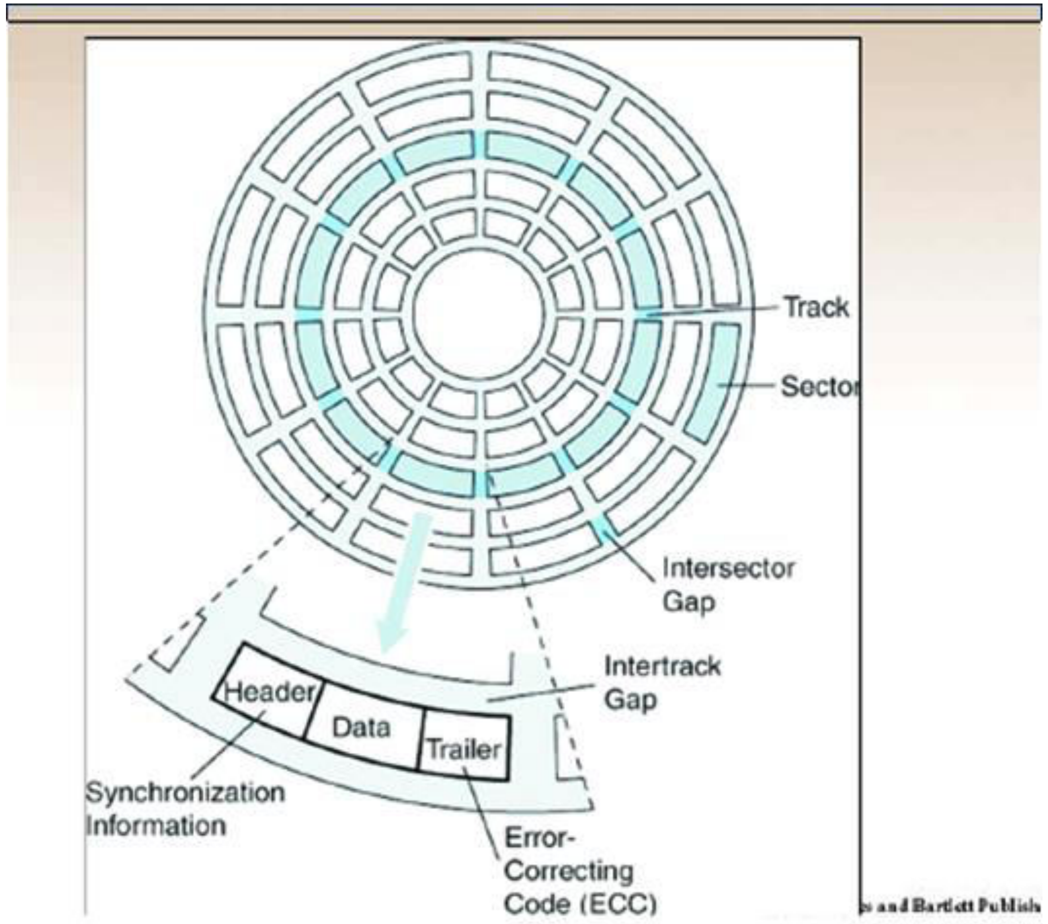
Usually 10 000 or 15 000 rpm SAS drives

0.1 ms	**Access times** SSDs exhibit virtually no access time	5.5 ~ 8.0 ms
SSDs deliver at least 6000 io/s	**Random I/O Performance** SSDs are at least 15 times faster than HDDs	HDDs reach up to 400 io/s
SSDs have a failure rate of less than 0.5 %	**Reliability** This makes SSDs 4 - 10 times more reliable	HDD"s failure rate fluctuates between 2 ~ 5 %
SSDs consume between 2 & 5 watts	**Energy savings** This means that on a large server like ours, approximately 100 watts are saved	HDDs consume between 6 & 15 watts
SSDs have an average I/O wait of 1 %	**CPU Power** You will have an extra 6% of CPU power for other operations	HDDs' average I/O wait is about 7 %
the average service time for an I/O request while running a backup remains below 20 ms	**Input/Output request times** SSDs allow for much faster data access	the I/O request time with HDDs during backup rises up to 400~500 ms
SSD backups take about 6 hours	**Backup Rates** SSDs allows for 3 - 5 times faster backups for your data	HDD backups take up to 20~24 hours

HDD	SDD
Require seek + rotation	No seeks
Not parallel (one head)	Parallel
Brittle (moving parts)	No moving parts
Random reads take 10s milliseconds	Random reads take 10s microseconds
Slow (Mechanical)	Wears out
Cheap/large storage	Expensive/smaller storage

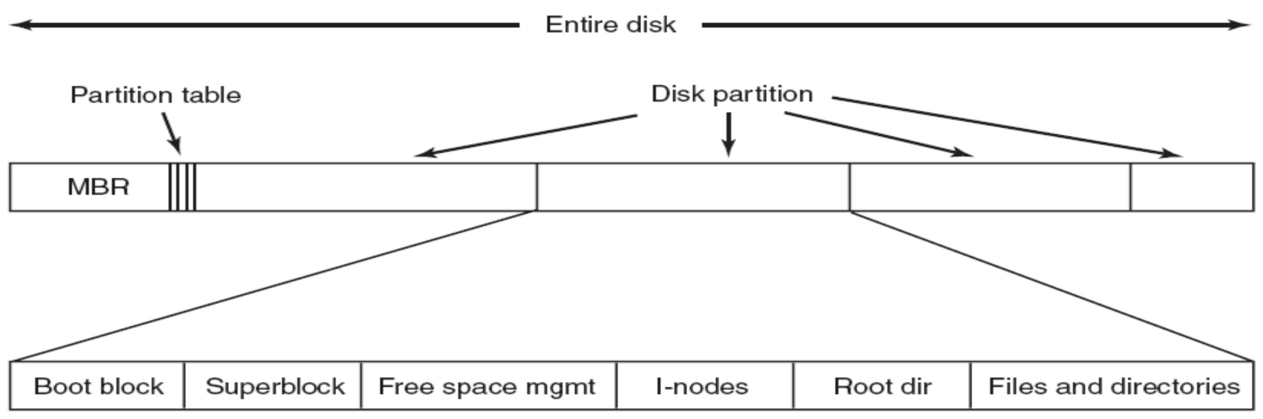
Disk formatting

- A new storage device is a blank slate: it is just a platter of a magnetic recording material (HDD) or a set of uninitialized semiconductor storage cells (SSD)
- Before a storage device can store data, it must be divided into sectors that the controller can read and write, this is called low-level formatting
- **Low-level formatting, or physical formatting** — Dividing a disk into sectors that the disk controller can read and write
  - Each sector can hold header information, plus data, plus error correction code (**ECC**)
  - Usually 512 bytes or 4KB
- Most drives are low-level-formatted at the factory as a part of the manufacturing process.



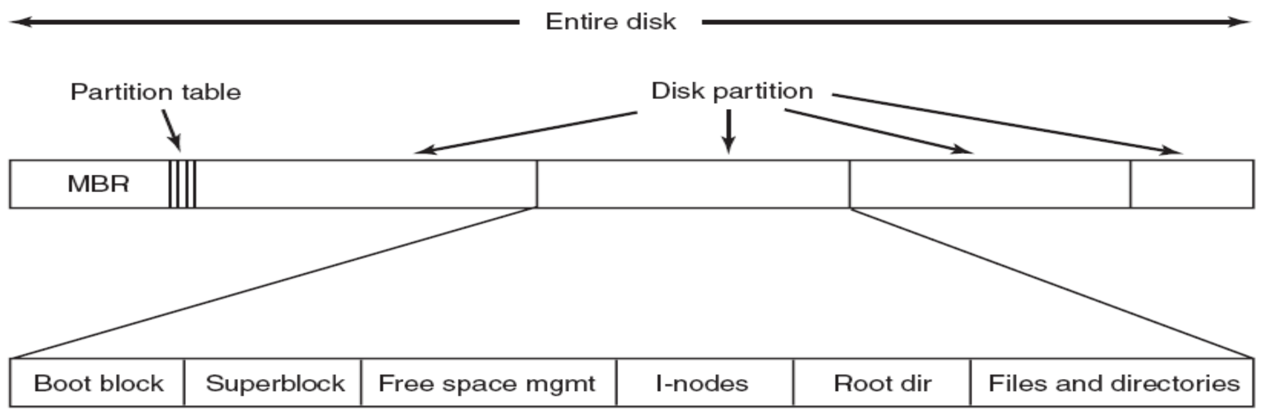
Disk formatting

- The next step is disk partition.
- **Partition:** the disk is partitioned into one or more groups of cylinders, each treated as a logical disk
- For instance, one partition can hold a file system containing a copy of the operating system's executable code, another the swap space, and another a file system containing the user files
- The partition information is written in a fixed format at a fixed location on the storage device



Disk formatting

- The last step is logical formatting or creation of a file system
- In this step, the operating system stores the initial file-system data structures:
  - **Superblock**. Contains a magic number to identify the file-system type, the number of blocks in the file system, and other key administrative information
  - free blocks in the file system in the form of a bitmap or a list of pointers
  - i-nodes / FAT
  - Root directory of the file system



Files

- **Files** are logical units of information created by processes.
- When a process creates a file, it gives the file a **name**.
- When the process terminates, the file continues to exist and can be accessed by other processes using its name.
- So, files provide a way to store information on the disk and read it back later.

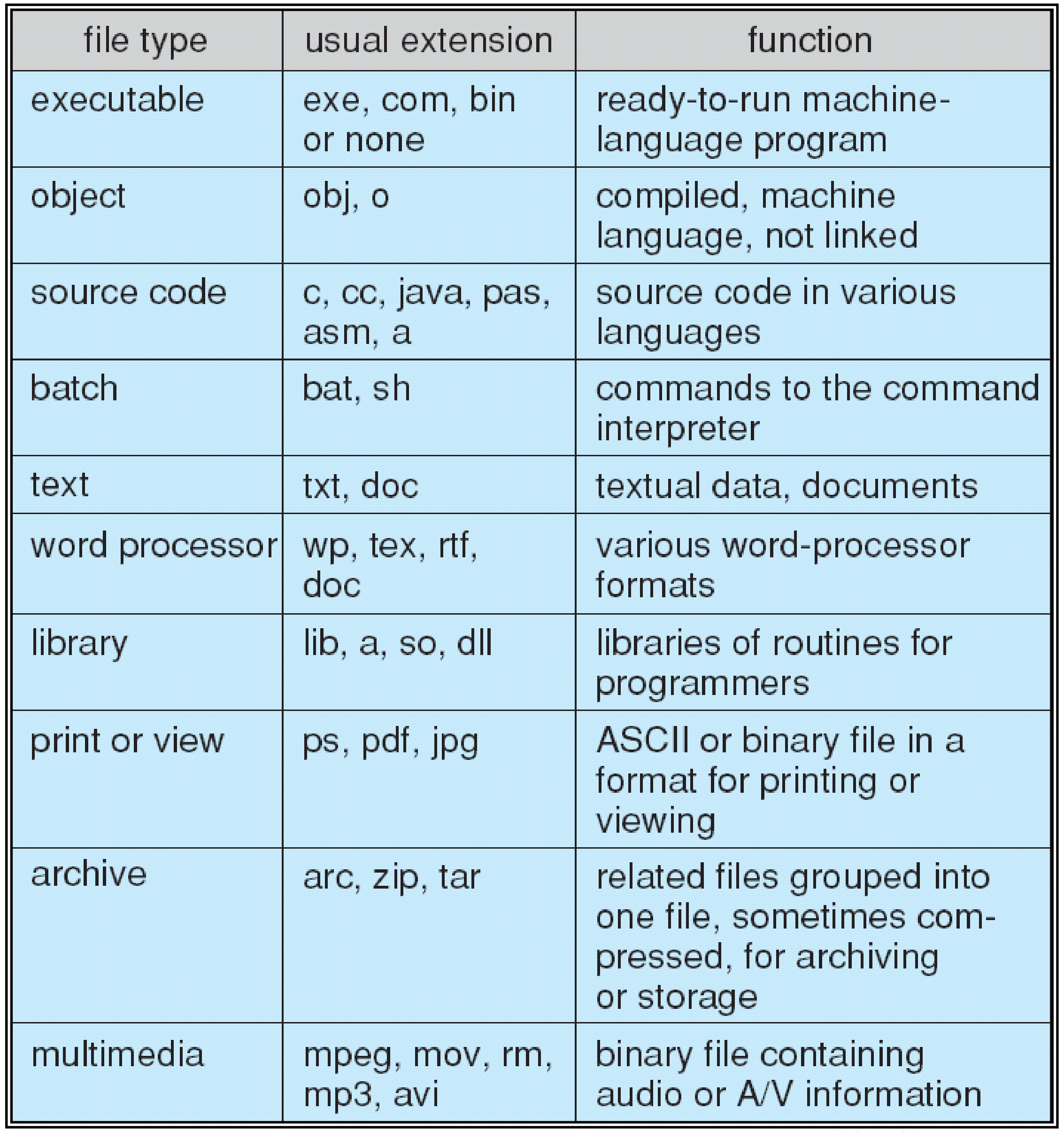
File naming

- The exact rules for file naming vary from system to system, but all current file systems allow at least strings of one to eight characters as legal file names.
    - *andrea*, *bruce*, and *cathy* are possible file names.
- Digits and special characters are also permitted, like *2*, *urgent!*, and *Fig.2-14*
- Many file systems support names as long as 255 characters.
- Some file systems distinguish between upper- and lowercase letters (UNIX), whereas others do not (MS-DOS)

File naming

- Many file systems support two-part file names, with the two parts separated by a period, as in *prog.c*.
- The part following the period is called the **file extension** and usually indicates something about the file.
- In MS-DOS, for example, file names are 1 to 8 characters, plus an optional extension of 1 to 3 characters.
- In UNIX, the size of the extension, if any, is up to the user, and a file may even have two or more extensions: *homepage.html.zip*
- UNIX file extensions are just conventions and are not enforced by the operating system
- Windows is aware of the extensions and assigns meaning to them. When a user double clicks on a file name, the program assigned to its file extension is launched with the file as a parameter.
    - For example, double-clicking on *file.docx* starts Microsoft *Word* with *file.docx* as the initial file to edit.

File types, extensions



file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes com-pressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

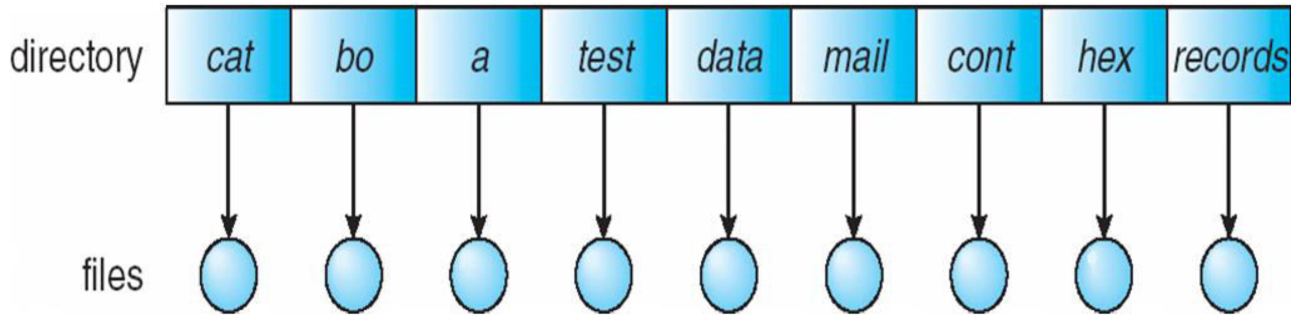
File attributes

- Every file has a name and its data.
- In addition, all file systems associate other information with each file, the date and time the file was last modified and the file's size.
- These extra items are the file's **attributes** or **metadata**.
- The list of attributes varies considerably from system to system

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

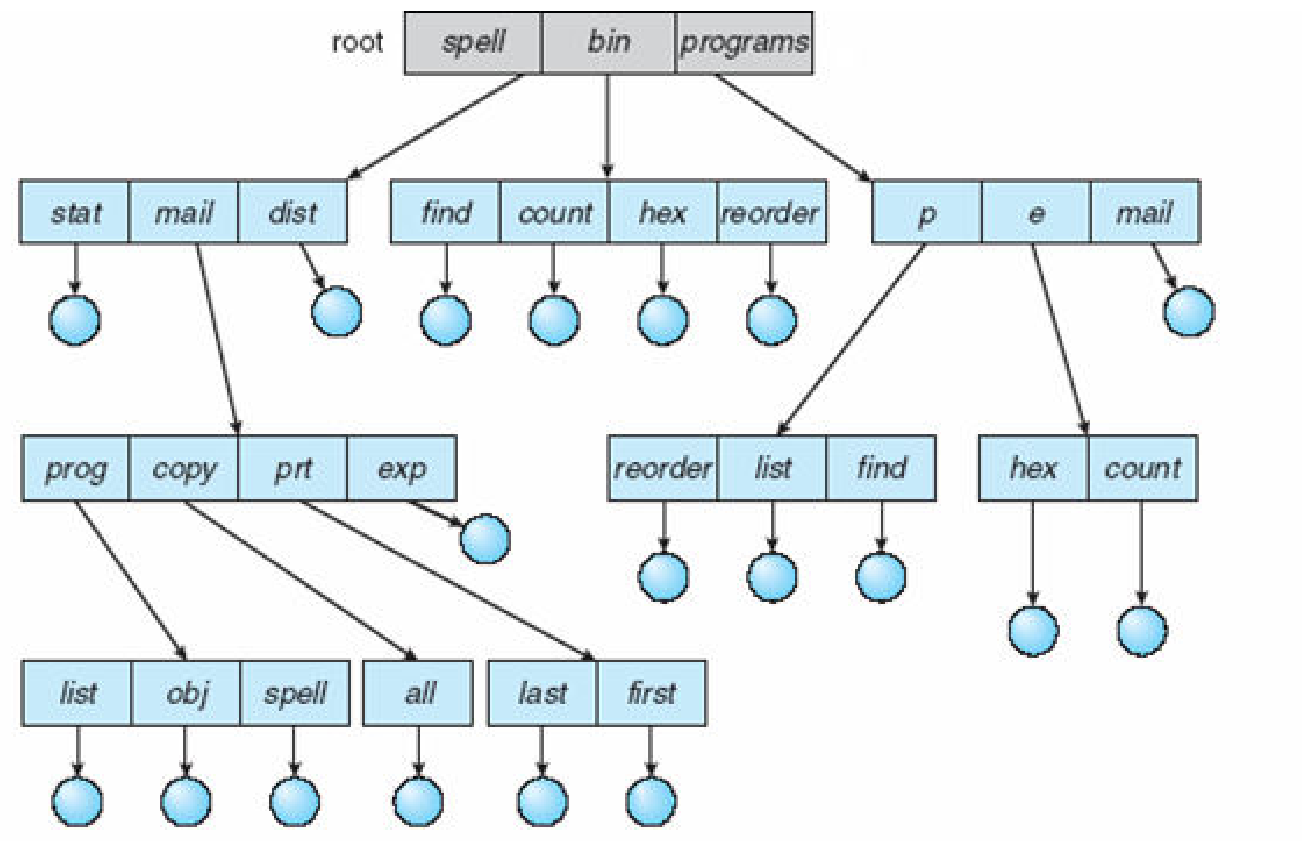
Directories

- To keep track of files, file systems normally have **directories**, which are themselves files.
- A directory is a file with a special structure
- Two types of directories: **Single-Level Directory and Hierarchical directory**
- **Single-Level Directory Systems:** one directory containing all the files
    - Was common in early personal computers



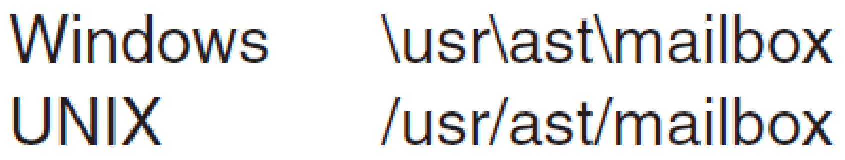
Hierarchical directory systems

- It is a tree where leaves are data-files and internal nodes are directory-files
- Each directory-file entry points a mix of data-files and subdirectories
- The tree has a root node which is a directory, the root directory



Path names

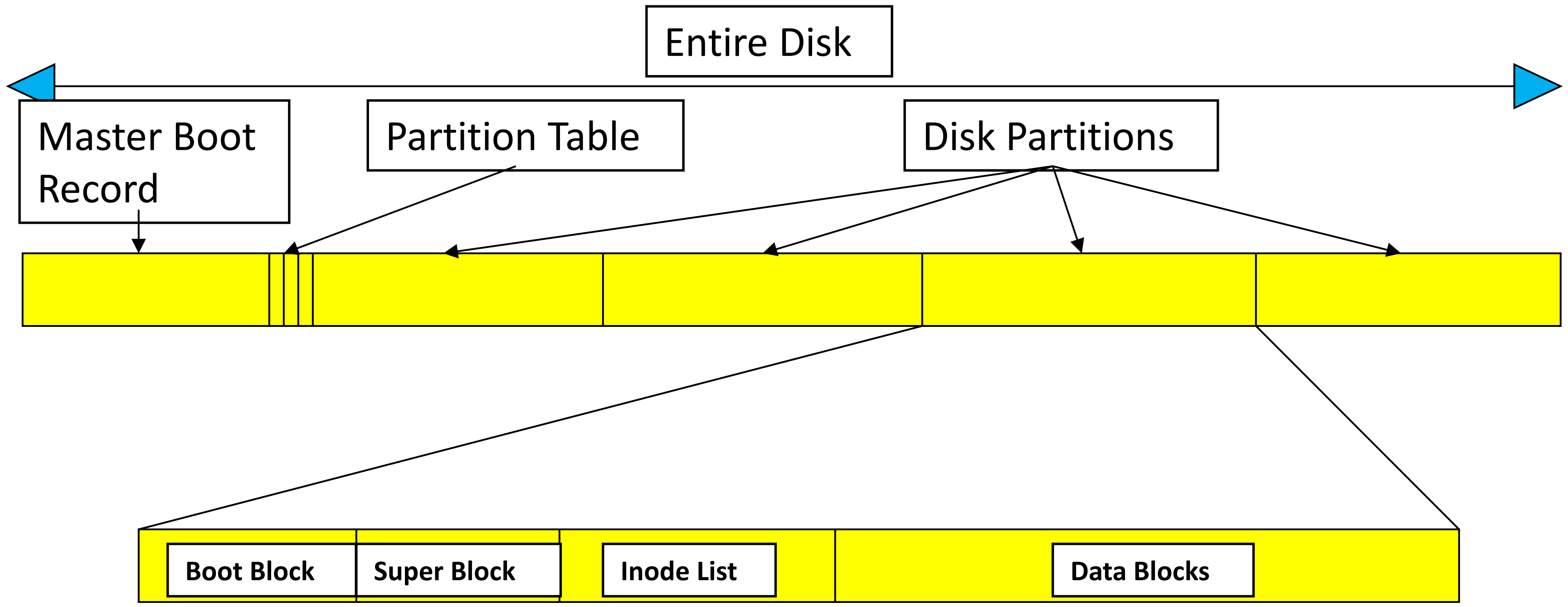
- When the file system is based on a directory tree, a file is identified using its name and a path in the tree.
- Two different methods:
  - **Absolute path name** consisting of the path from the root directory to the file. As an example, the path */usr/ast/mailbox*
  - **Relative path name**. In conjunction with **working directory** (also called the **current directory**). Path names beginning with the working directory



Windows	\usr\ast\mailbox
UNIX	/usr/ast/mailbox

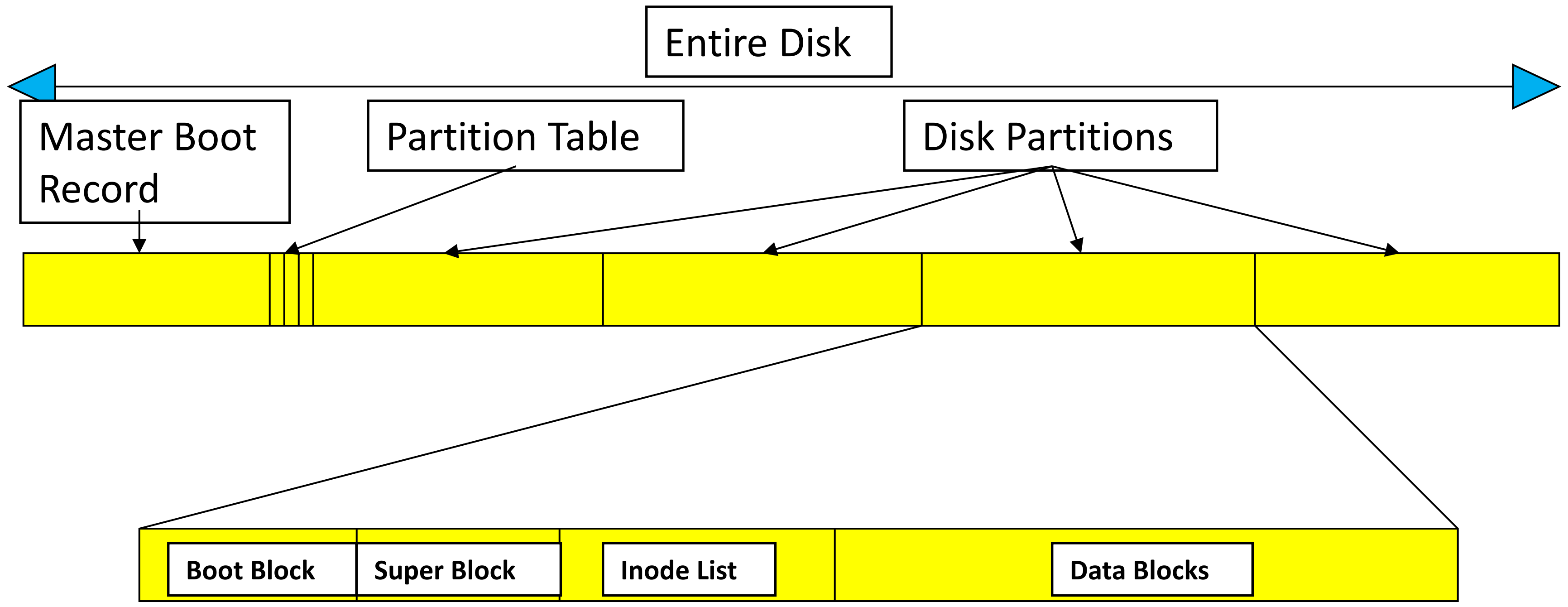
File systems

- File systems describe how files are stored, and retrieved in a partition of the hard disk
- As a partition is a contiguous set of blocks on a drive that are treated as an independent disk, there must be a file system for each partition



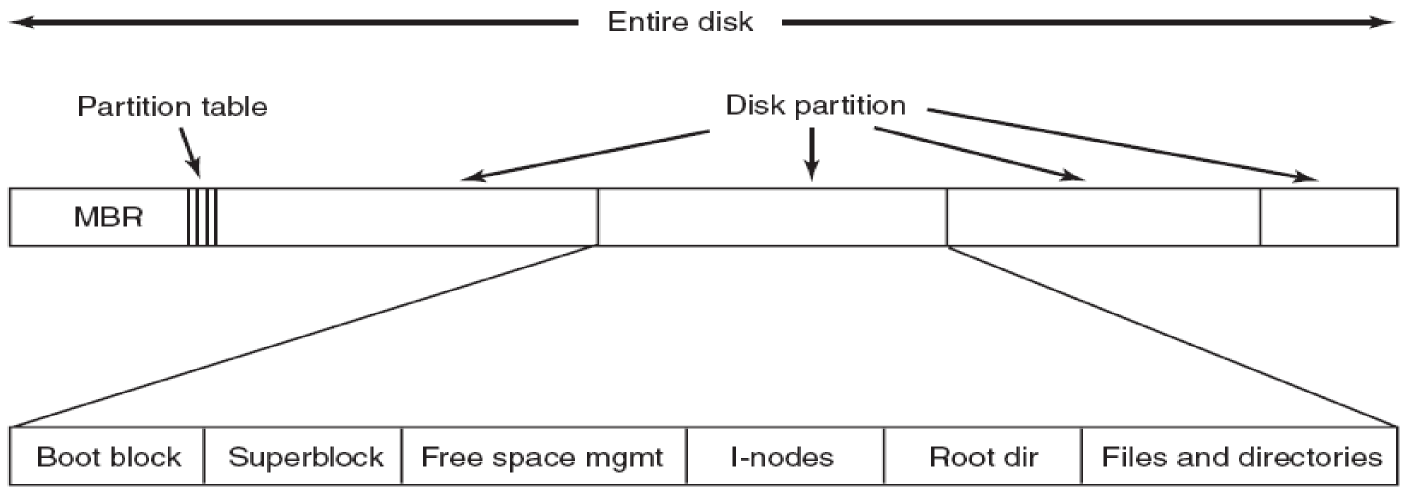
File system

- Information about a file system is contained in the superblock.
- The superblock is read into memory when the computer is booted or the file system is first touched.
- Typical information in the superblock includes a magic number to identify the file-system type, the number of blocks in the file system, and other administrative information.



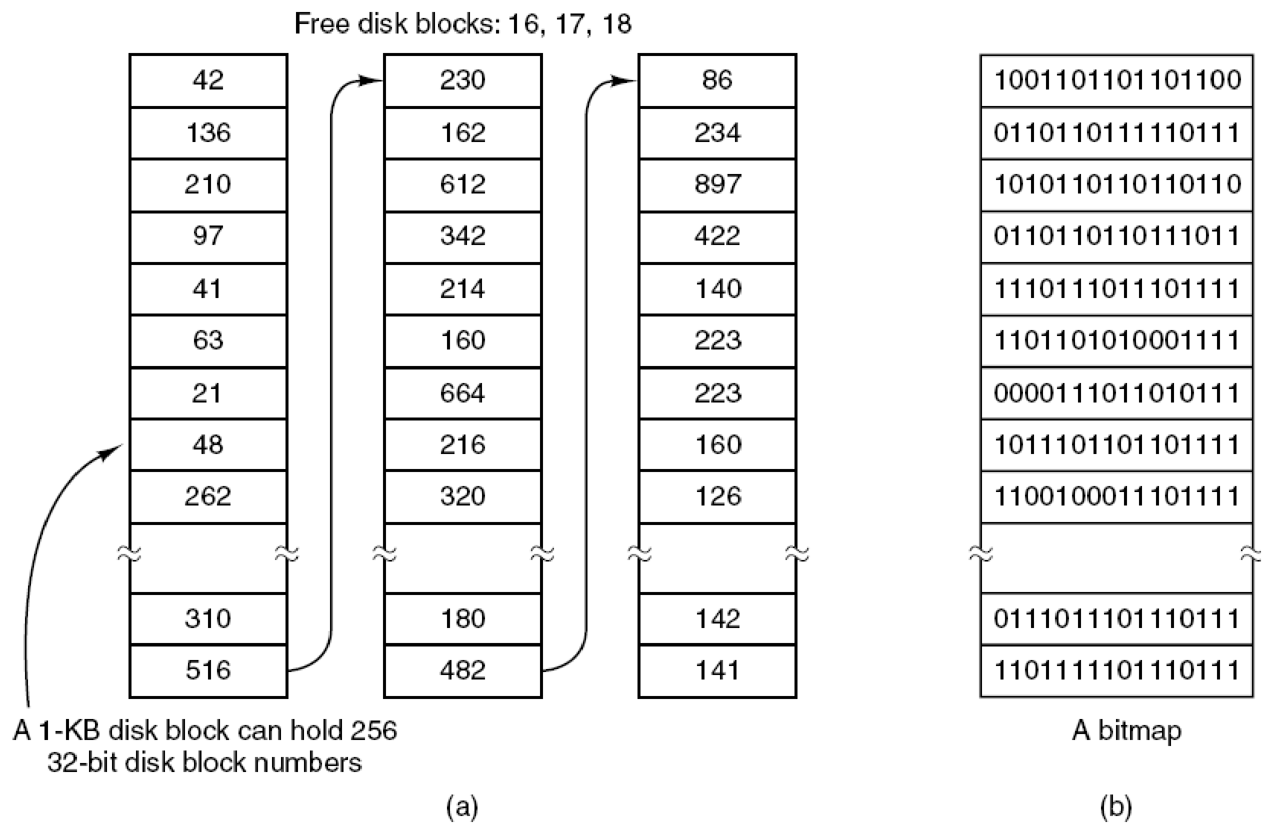
File system

- After the superblock comes information about free blocks in the file system, for example in the form of a bitmap or a list of pointers.



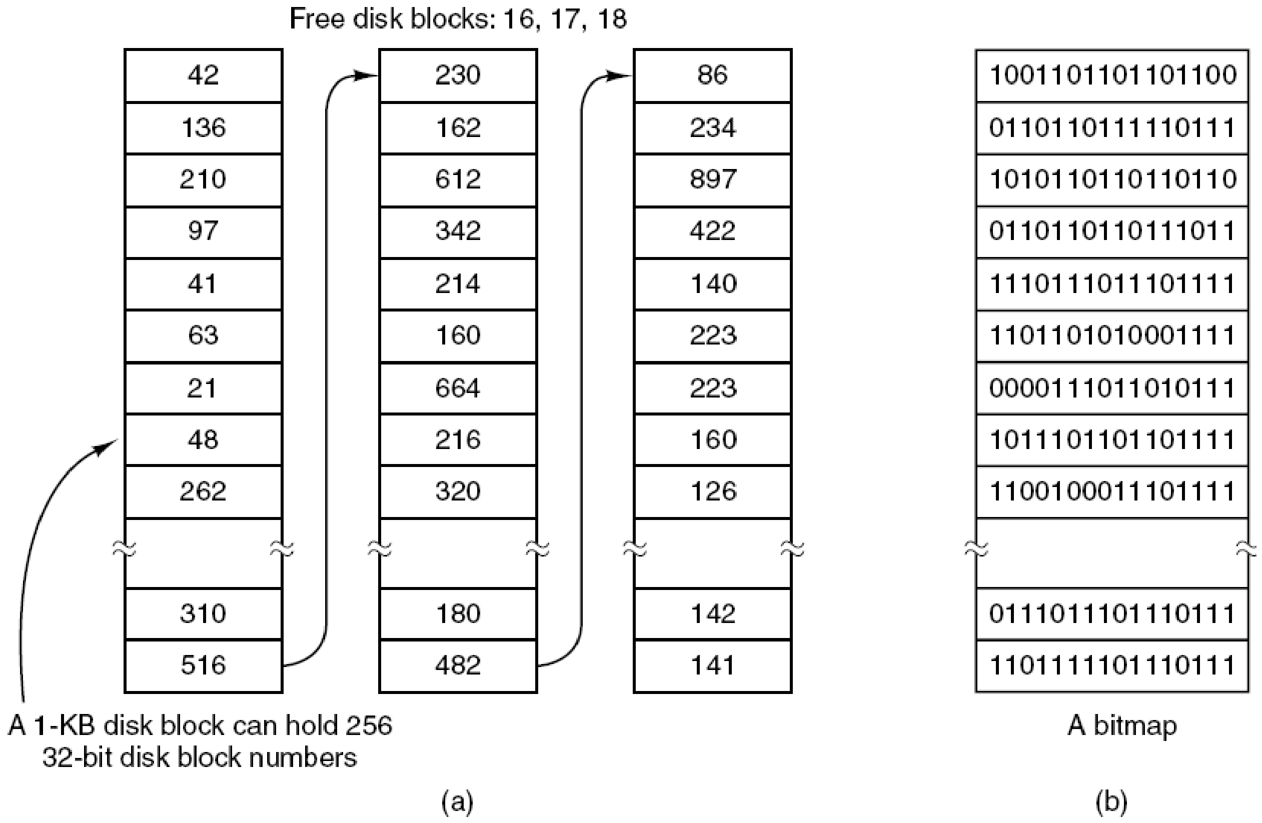
Keeping track of free blocks

- 1- use a linked list of disk blocks, each block holding the address of free disk block numbers:
  - With a 1-KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is required for the pointer to the next block.)
  - A 1-TB disk requires about 4 million blocks.



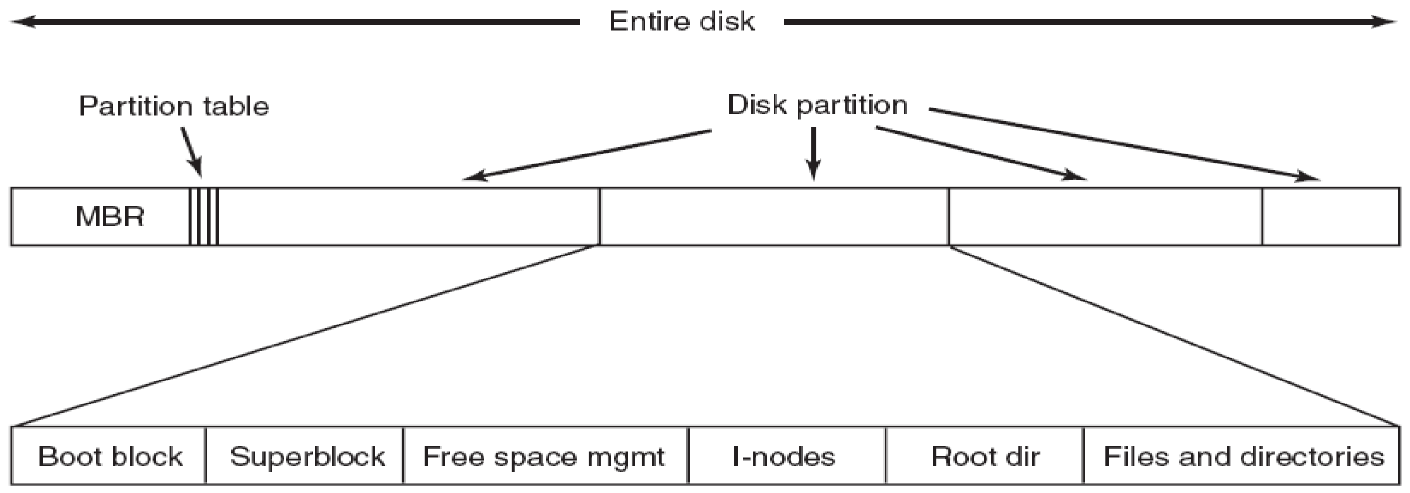
Keeping track of free blocks

- 2- Use a bitmap. A disk with *n* blocks requires a bitmap with *n* bits.
- Free blocks are represented by 1s in the map, allocated blocks by 0s
  - 1-TB disk, we need 1 billion bits for the map, which requires around 130,000 1-KB blocks to store



File system

- After the superblock and the management of free blocks in the file system come
  - i-nodes, an array of data structures, one per file, telling all about the file
  - or a file allocation table (FAT)
  - the root directory, which contains the top of the file-system tree.
  - the remainder of the disk contains all the other directories and files.

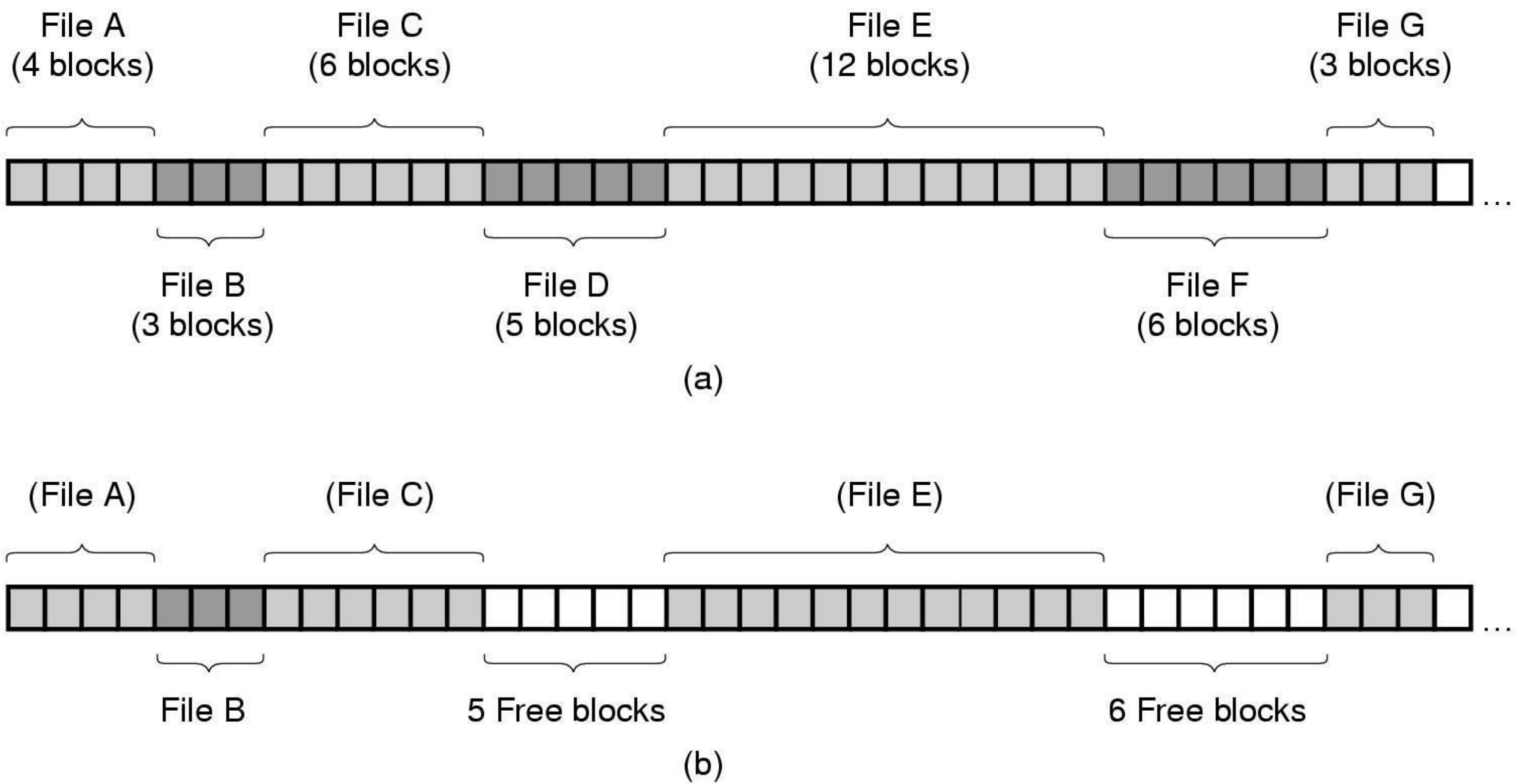


Allocating physical storage to files

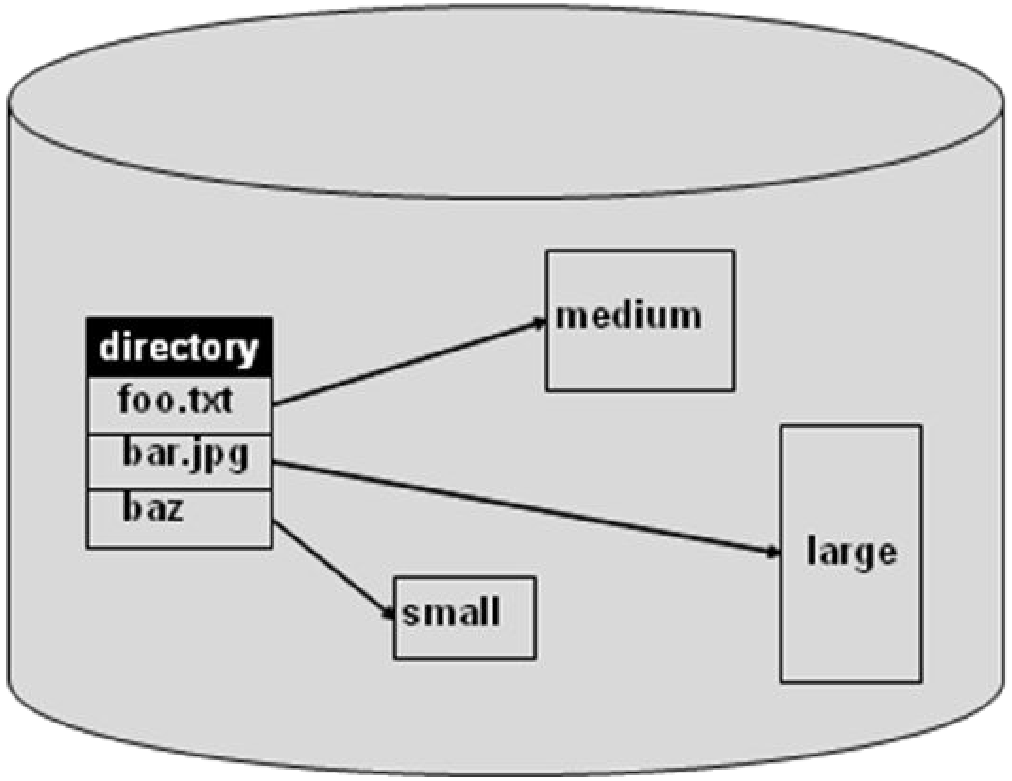
- Files are stored in blocks (sectors) of the disk, so there must be a method to allocate blocks to files (similar to allocate main memory to processes)
- Allocation methods:
  - Contiguous blocks
  - Linked list of blocks
  - Linked list using table
  - I-nodes

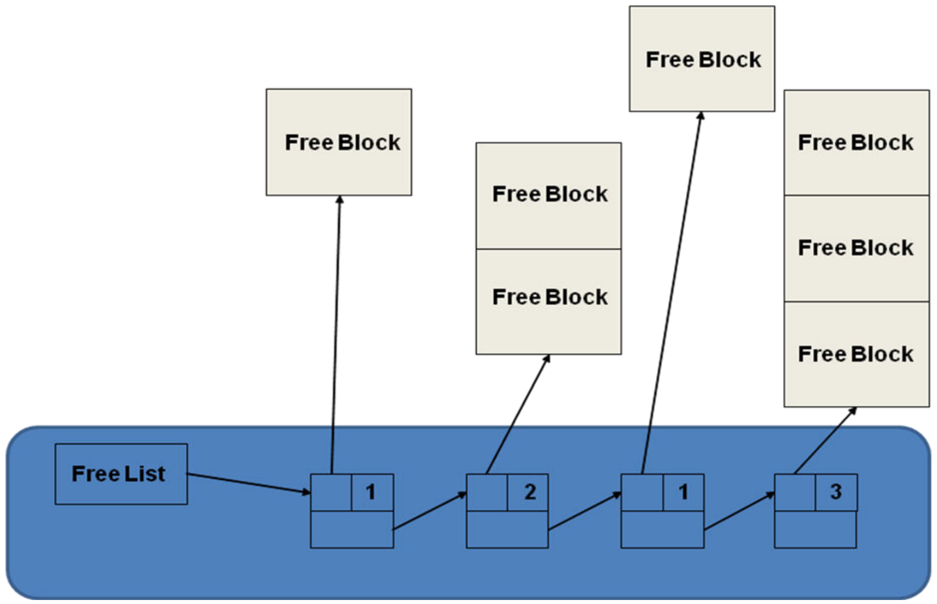
Contiguous allocation

- The simplest allocation scheme is to store each file as a contiguous sequence of disk blocks.
- With 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks
- The directory entry for the file only needs the address of the first block
- Drawback: over time, the disk becomes fragmented.



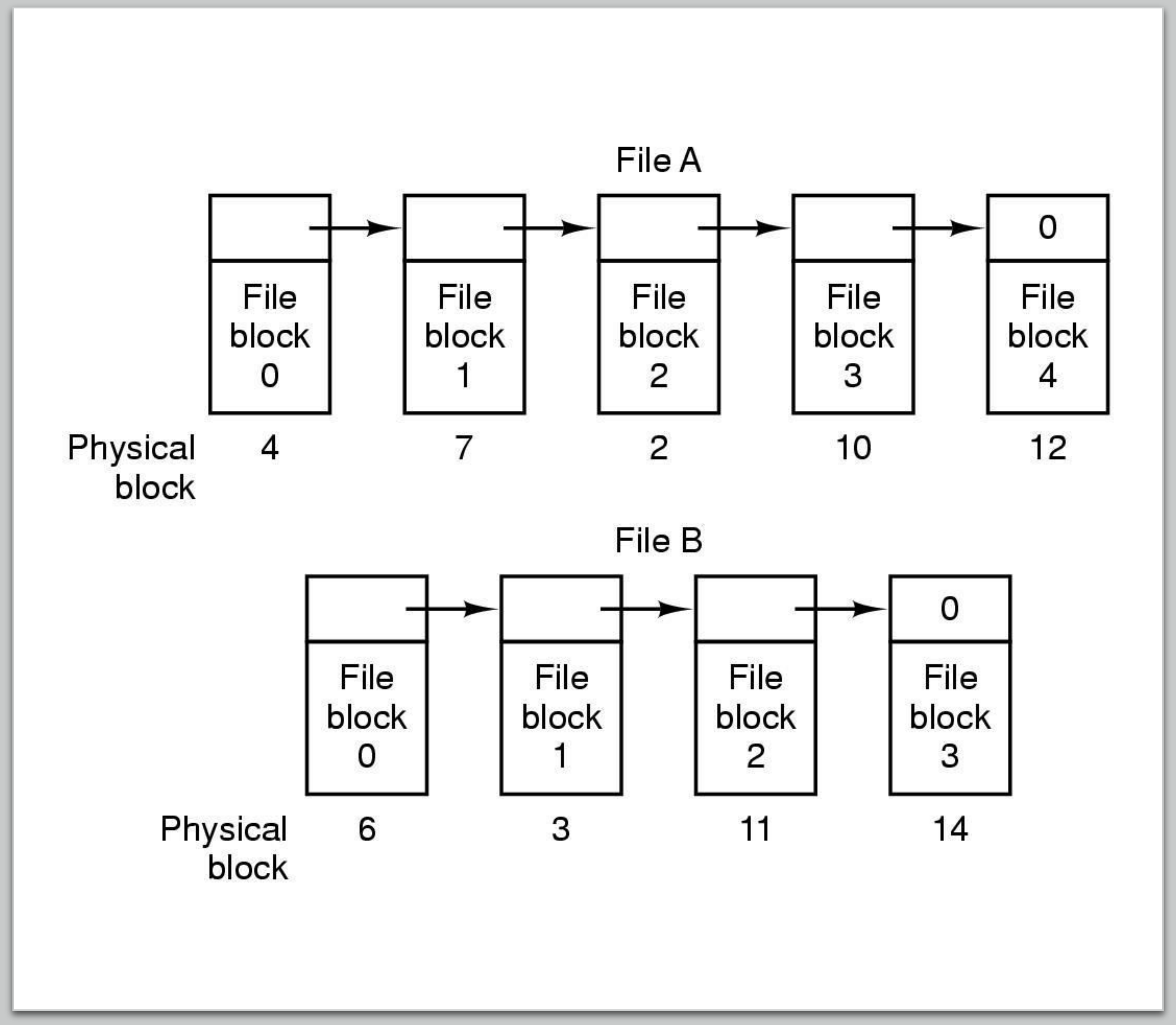
- **Contiguous Allocation**
  - At file creation time, a sequence of blocks is allocated, only remember the address of the first block
  - File cannot grow beyond that size
  - Fragmentation a problem
- Free list
  - Allocation may be by first or best fit
  - Requires periodic compaction





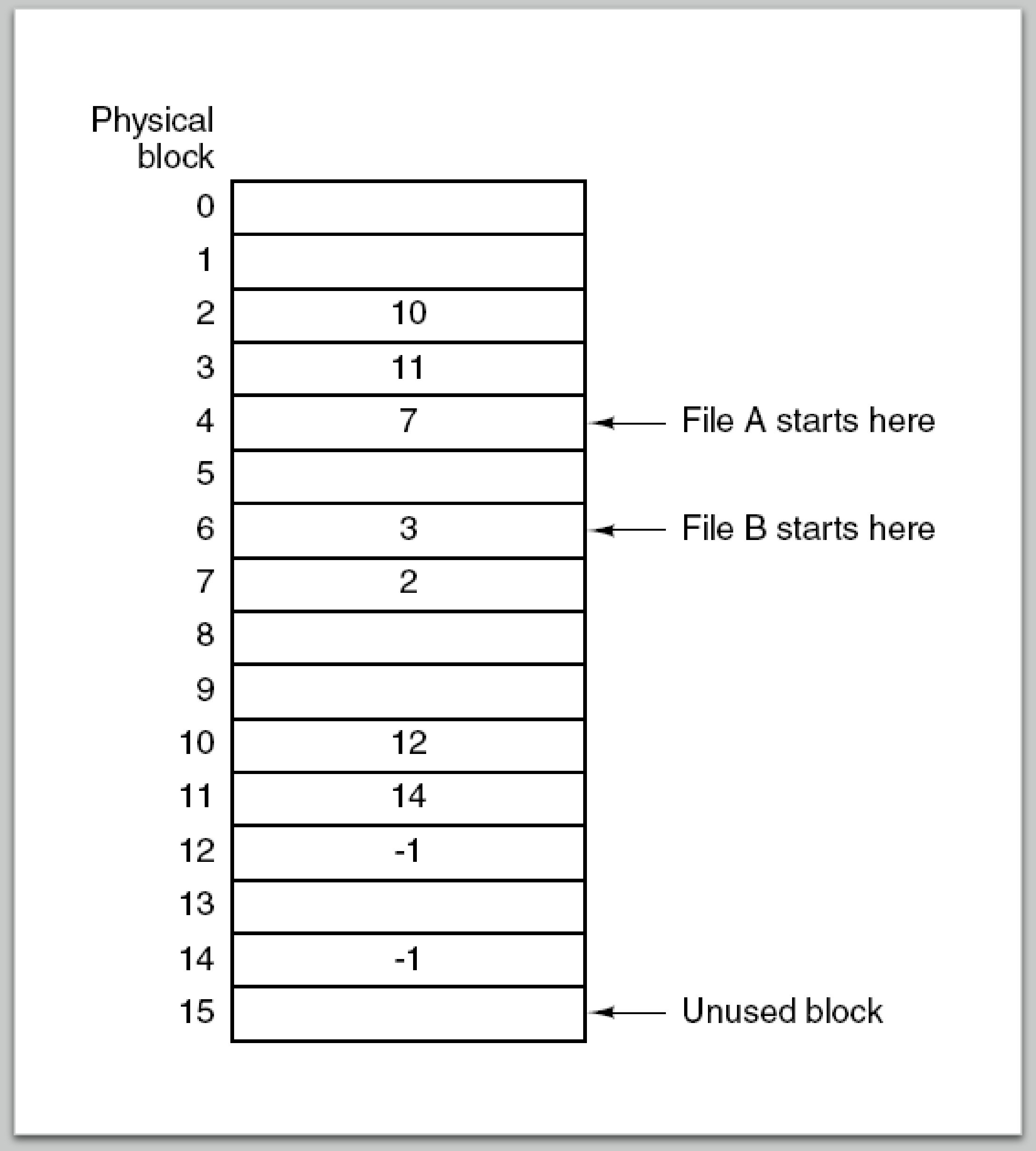
Linked list allocation

- Here each file is stored as a linked list of disk blocks.
- The first word of each block is used as a pointer to the next one.
- The rest of the block is for data.
- The directory entry only need the address of the first block
- Random access is slow, need to chase all the pointers

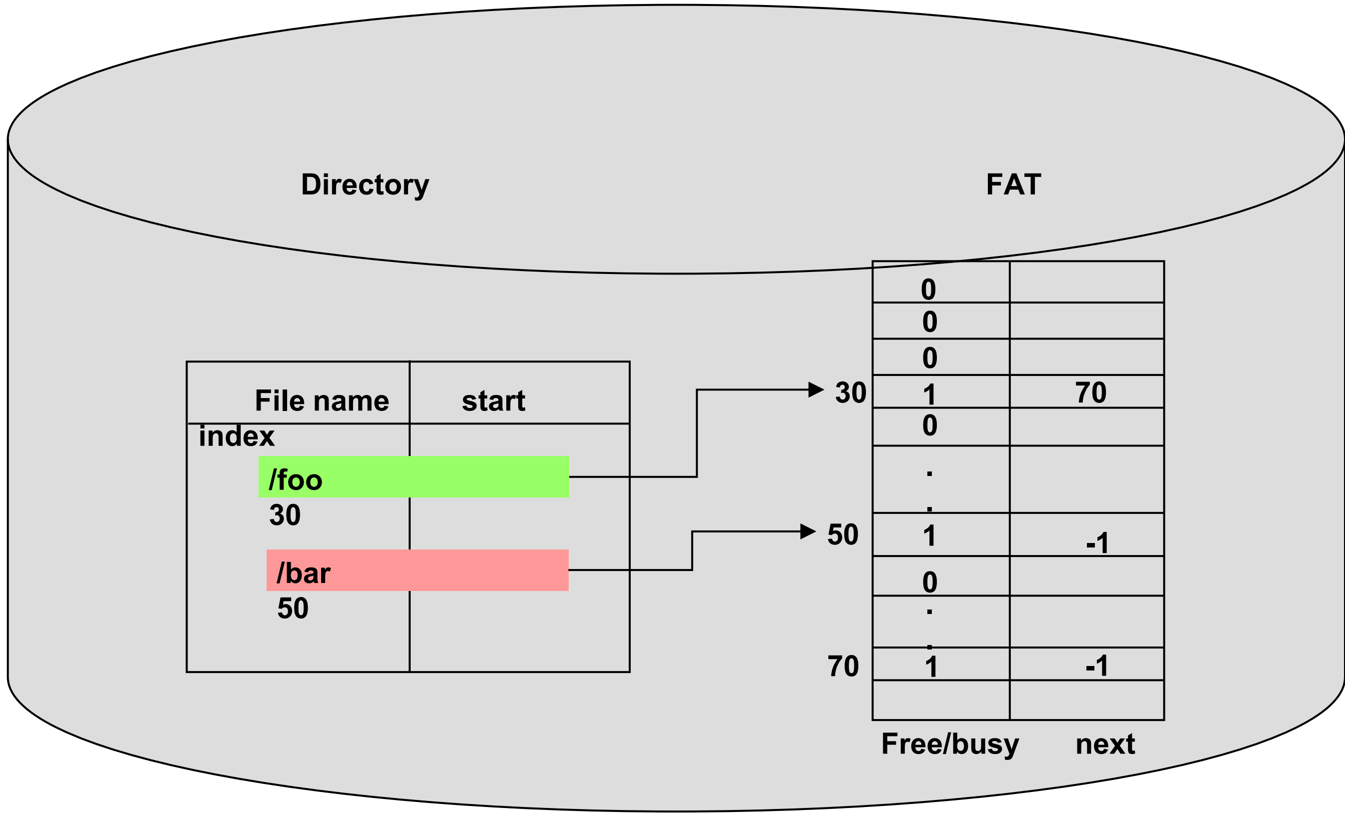


Linked list allocation using table

- Have a table in main memory of the same size as the number of blocks on the partition
- Store the address of the next block in the table
- Inconvenient: Table too big, with a 1-TB partition and a 1-KB block size, the table needs 1 billion entries, one for each of the 1 billion disk blocks
- This “file allocation table” (FAT) come in different forms in MS_DOS and Windows, FAT-12, FAT-16, FAT-32

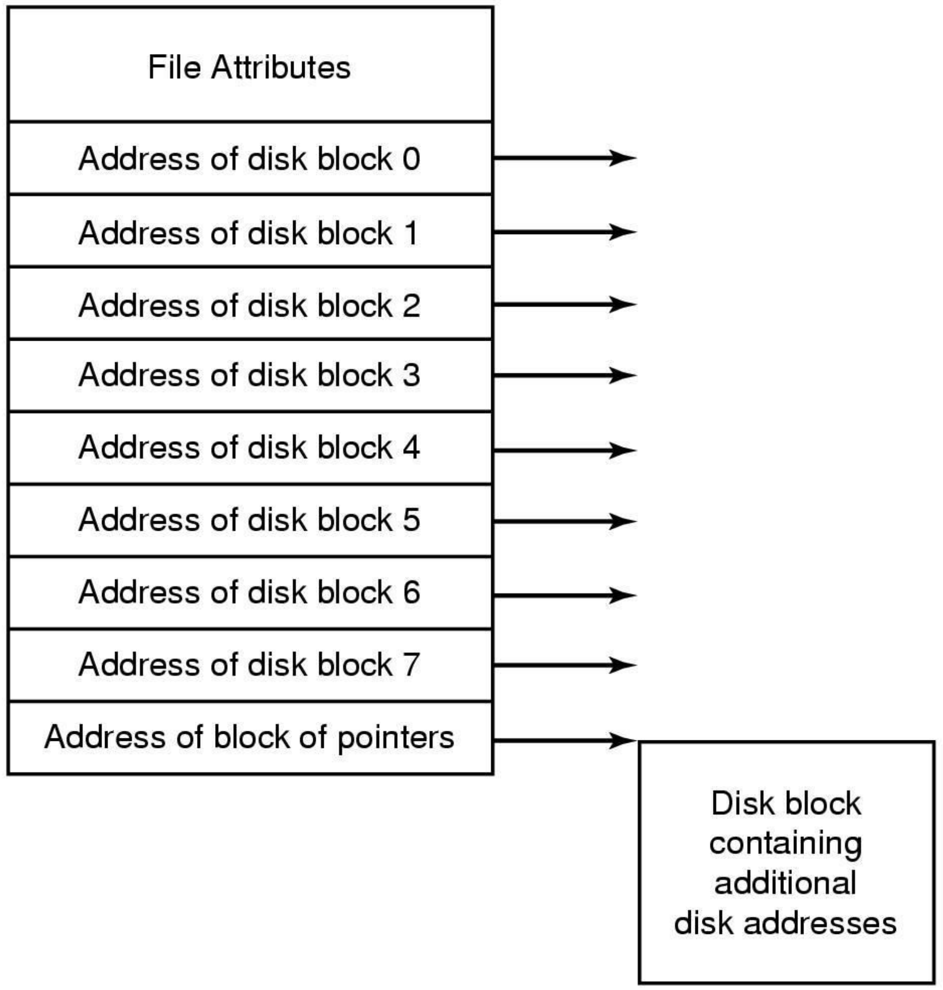


- **File Allocation Table (FAT)**
  - FAT in main memory
  - The directory has a pointer into the starting block entry in the FAT for each file.
  - FAT becomes big, use too much main memory

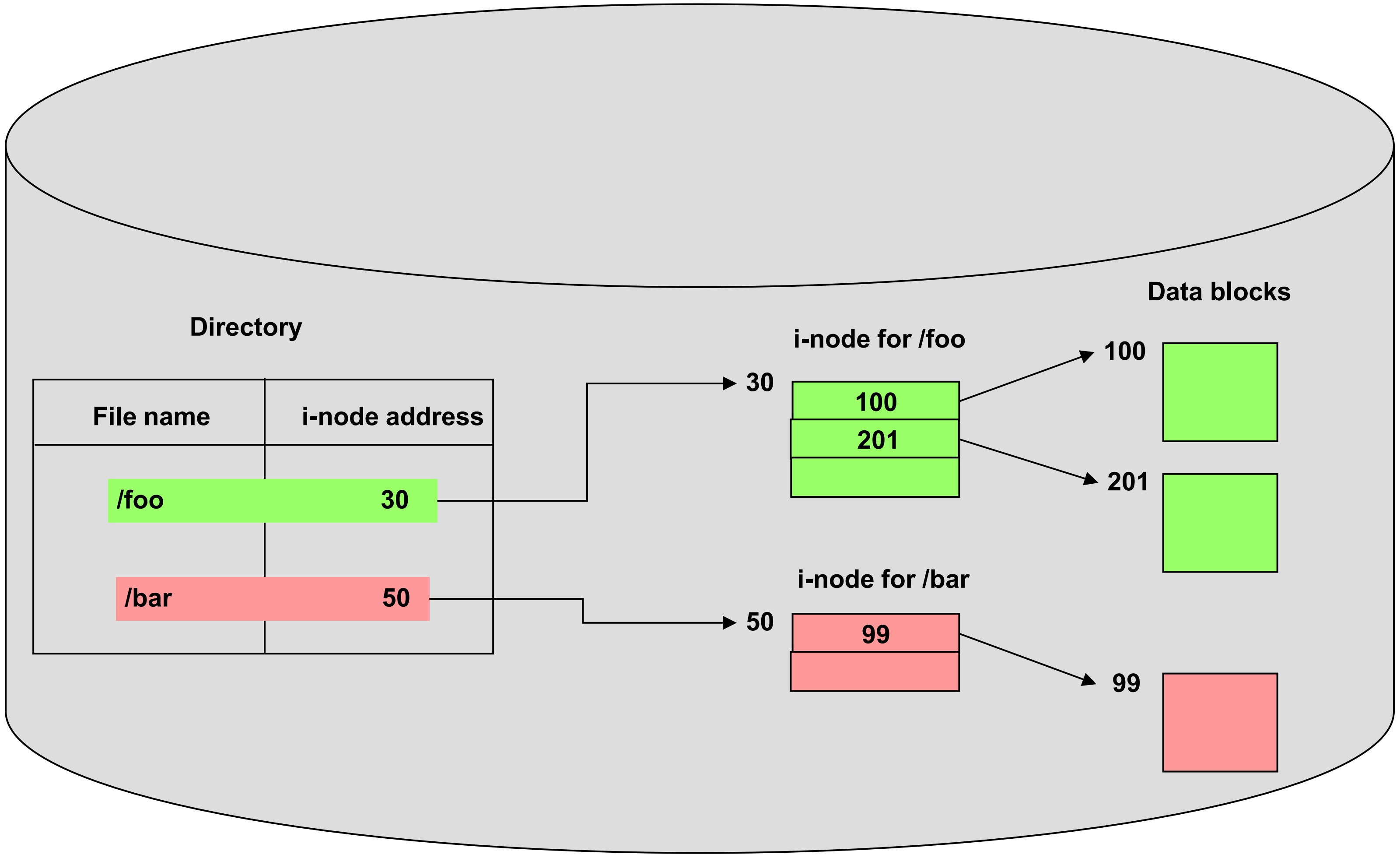


I-nodes

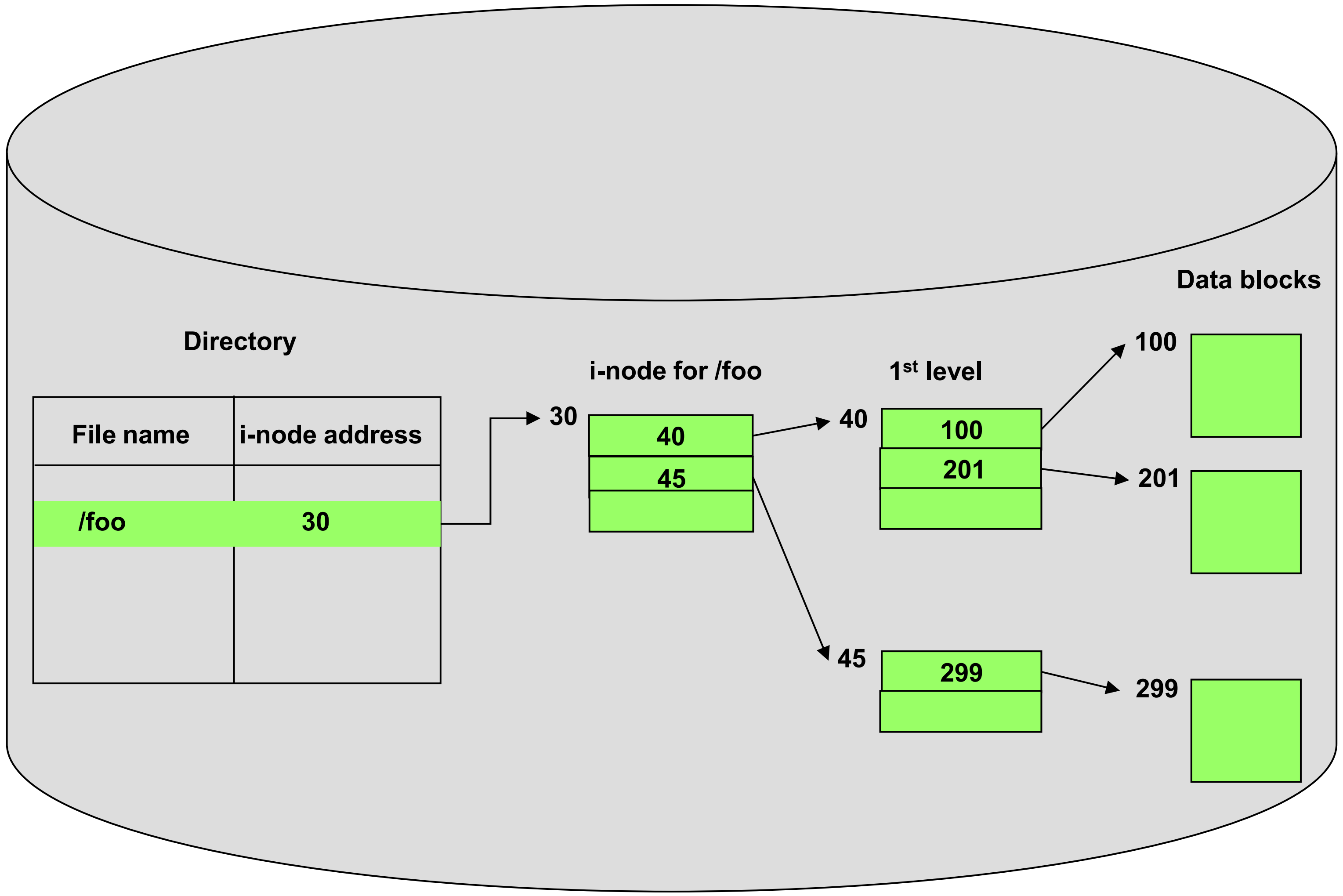
- Lists the attributes and disk addresses of the file’s blocks
- There is one i-node per file, i-nodes are stored on a specific set of blocks on the disk, when a file is opened, they are loaded in main memory
- i-nodes contain a fix number of disk addresses, if file is big, one address is kept to store the address of a block that contain more disk address
- i-nodes is the method in Unix/Linux operating systems, Windows has a similar system called NTFS



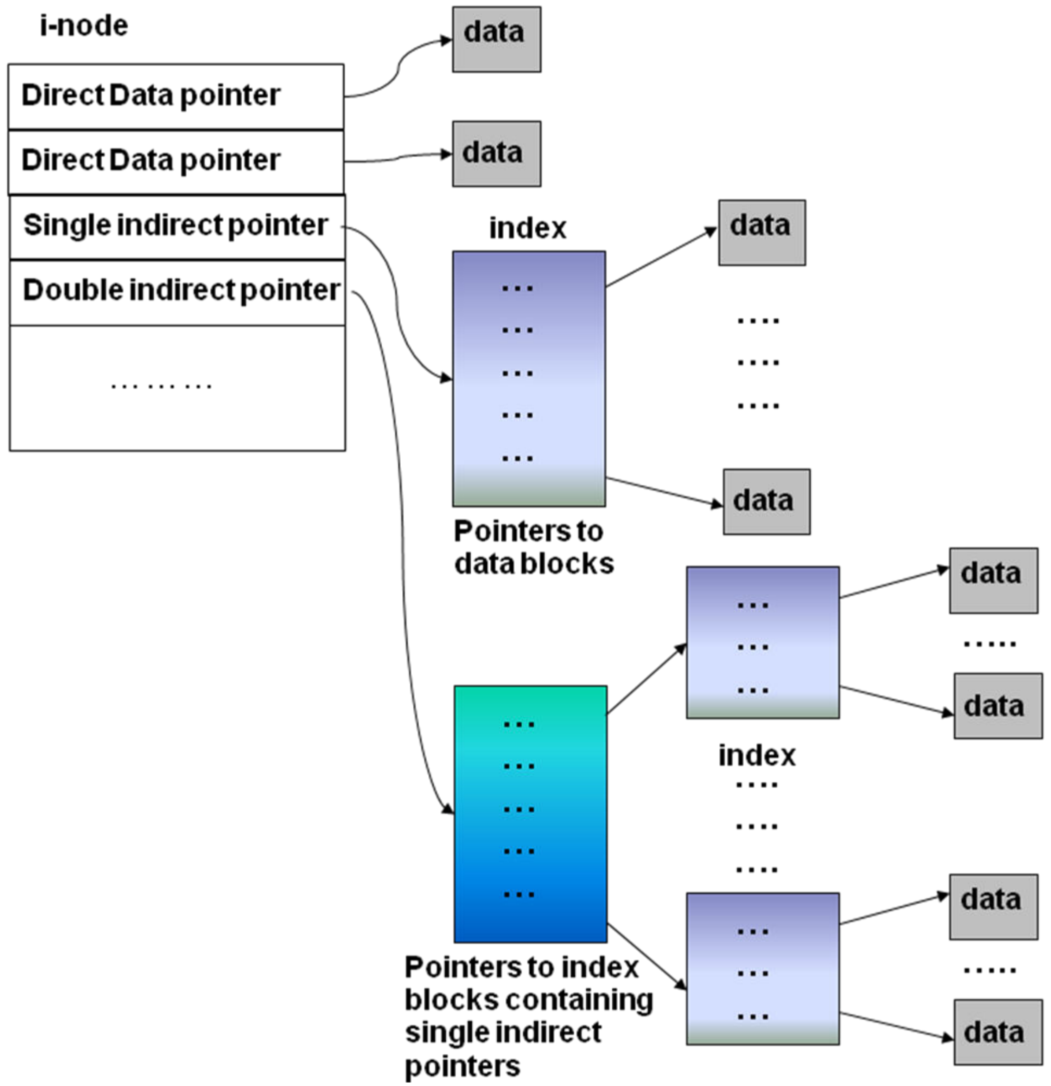
- **Indexed Allocation**
  - Directory entries now point to the i-node for that file
  - Since the i-node is a fixed size there is a maximum file size



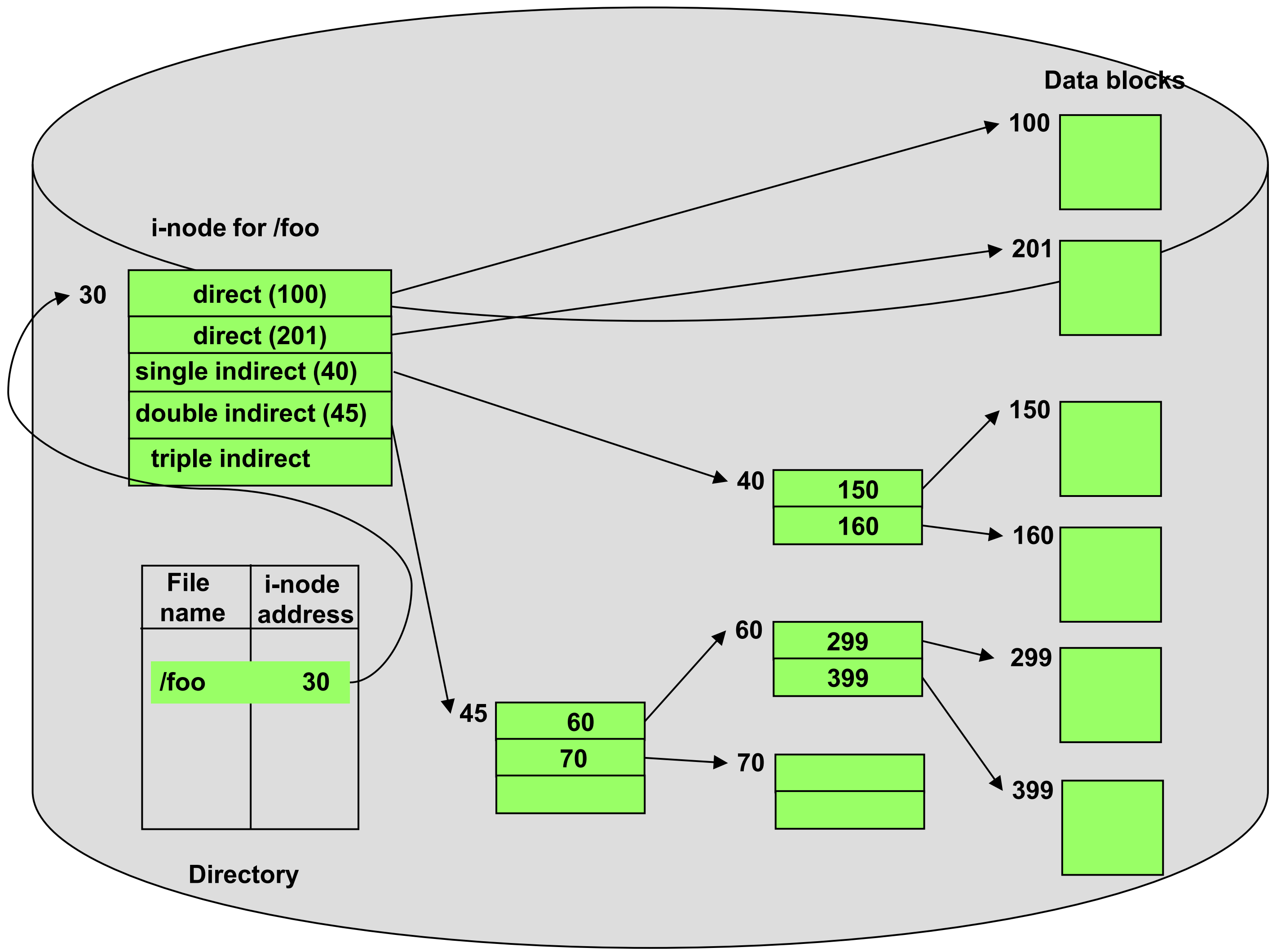
- **Multilevel Indexed Allocation**
  - Make the i-node point to index blocks which point to the files (first-level indirection)
  - May be extended to two-level (and beyond) indirection
  - Problem: Accessing even a small file requires a lot of indirection



- **Hybrid Indexed Allocation**
  - Two direct pointers for small files
  - One single indirect
  - One double indirect
  - One triple indirect

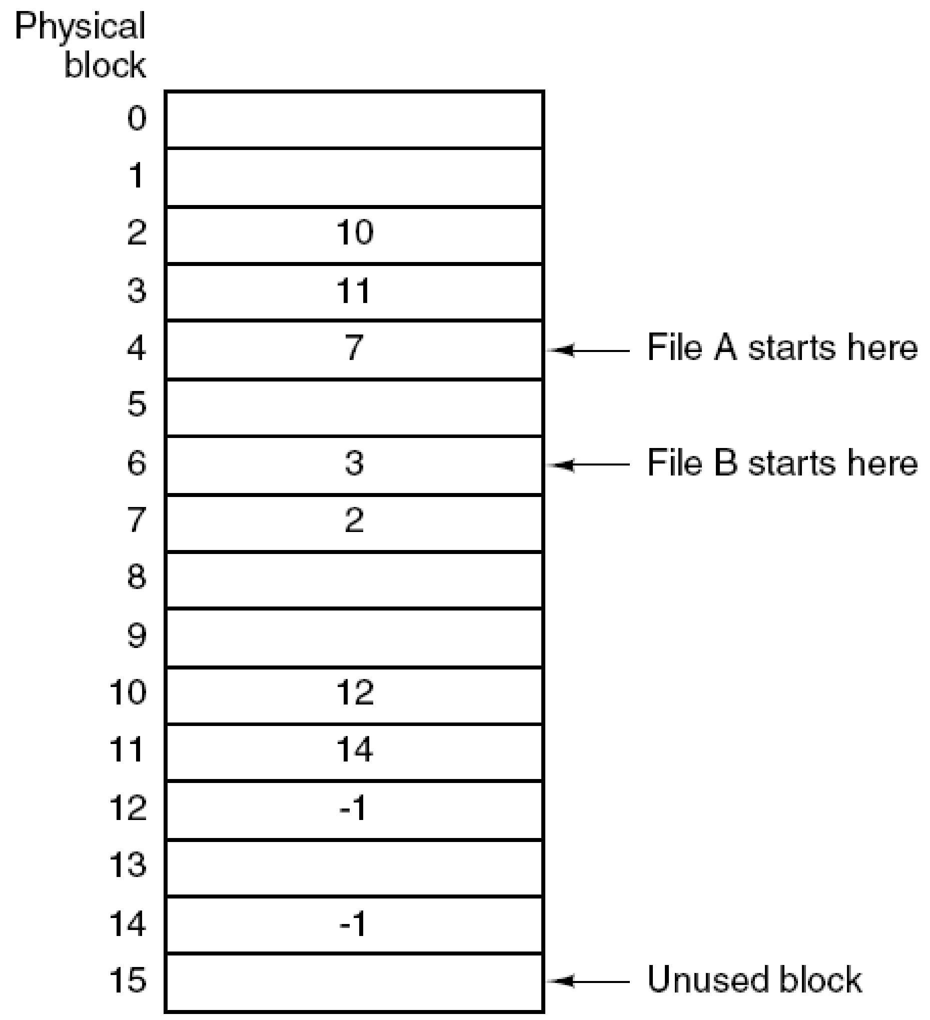


Hybrid indexed allocation for a directory



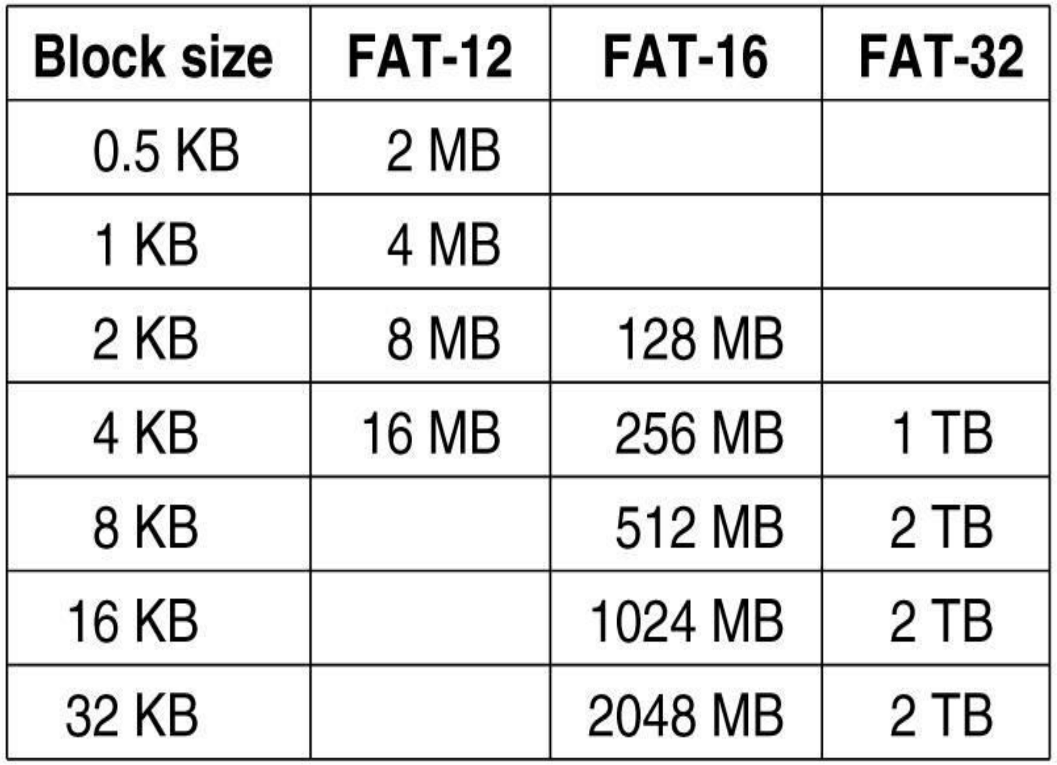
Windows file systems

- FAT12, 16, 32, NTFS
- FAT means files are allocated blocks through a table
- Each partition holds a FAT for mapping files to physical blocks
- The number after FAT indicated the number of bits used to represent an entry in the table
- Implicitly this number defines the size of the FAT table as well as the size of the file system (partition)
- FAT file systems run under the MS-DOS operating system, the ancestor of Windows



FAT 12

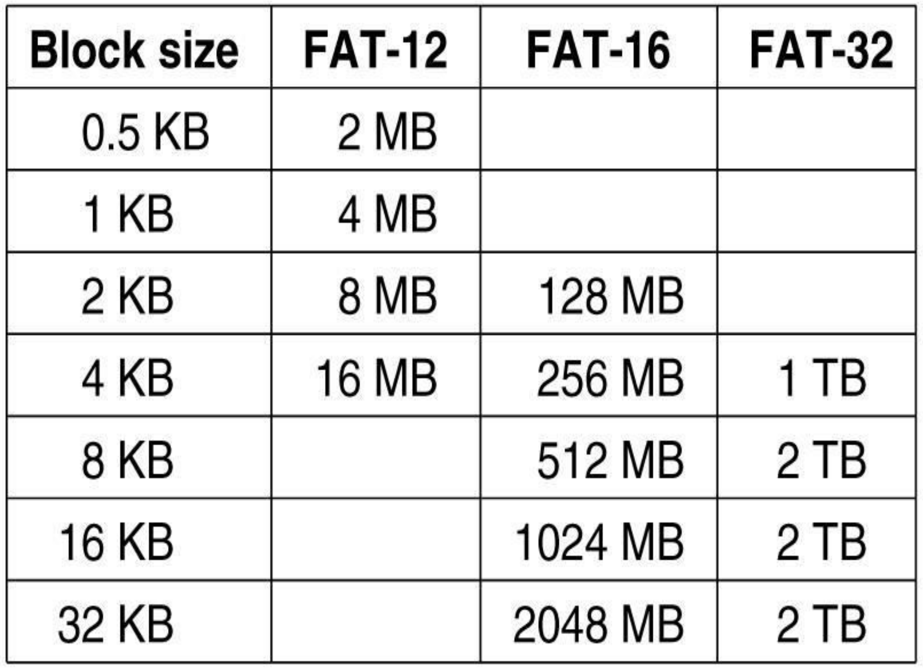
- Designed for floppy disks, 1.44MB capacity, block = sector = 512bytes, 2812 sectors
- Need 12 bits to address any block
- If FAT 12 used with larger drives, need to increase block sizes
- FAT 12 was allowed to have new block sizes of 1KB, 2KB and 4 KB
  - 2^{12} = 4096 is the largest number that can be represented with 12 bits
- If block size is 4KB, 4 * 2^12 = 16MB, each partition can be 16MB
- MS-DOS and Windows only supports 4 disk partitions, C:, D:, E:, F:, thus FAT 12 can address disk drives up to 64 MB



Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

FAT 16, FAT 32

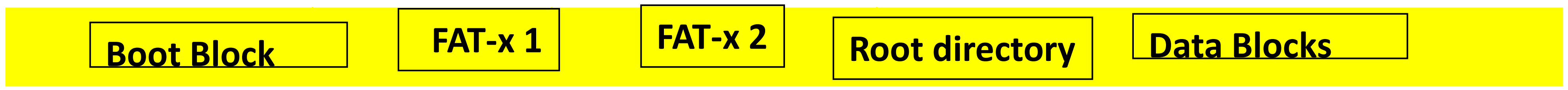
- FAT-16 was introduced to handle larger disk drives
- Was allowed block sizes 8KB, 16KB and 32KB
- FAT-16 uses 128KB of main memory per partition
  - 2^{16} = 65536 entries in the FAT, each entry is 2 bytes long
- The largest possible disk partition was 2GB
  - 65536 blocks * 32KB per block = 2097152 bytes
- 2GB is just enough for 8 minutes of video!
- Could address hard drives of 8GB
- Starting with Windows 95, FAT-32 file system was introduced



Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

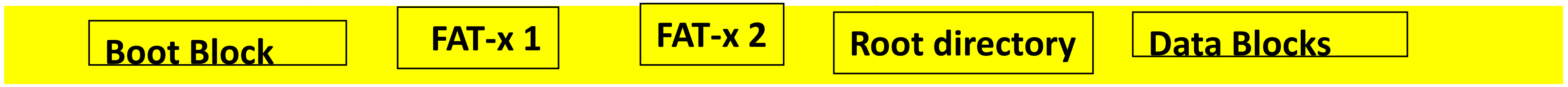
FAT structure on-disk

- Boot block, followed by
  - File Allocation Table (FAT)
    - 12, 16, and 32 bit FAT
    - Typically two copies of FAT
  - Root Directory
    - Fixed size
    - 32 byte Directory Entries
  - Data area
    - Once only for file data, later expanded to include subdirectories



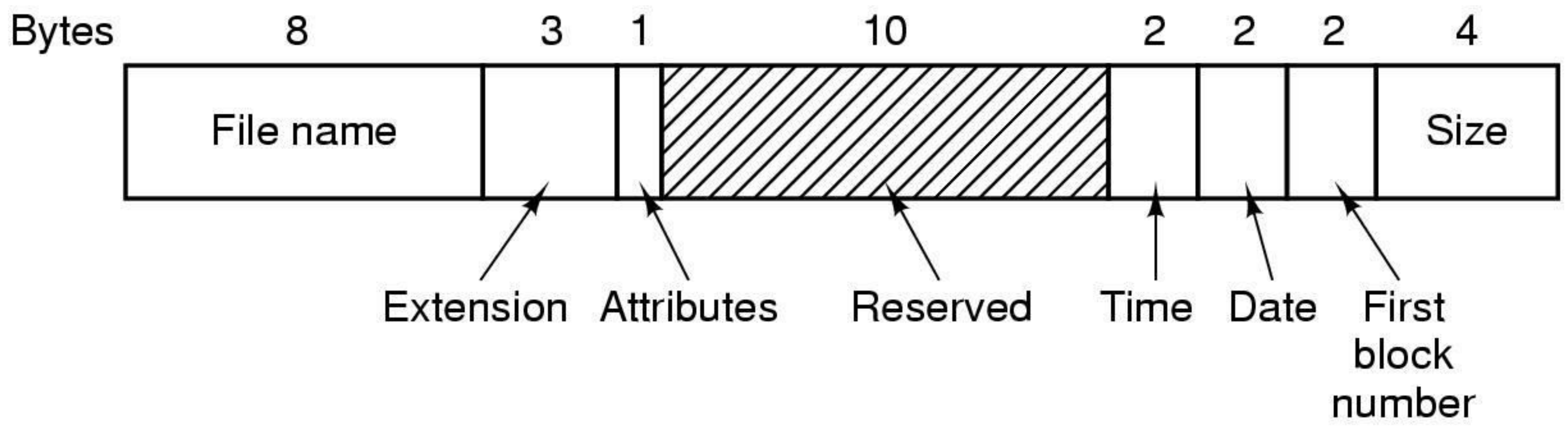
FAT structure on-disk

- The boot block contains information about the file system, including how many copies of the FAT tables are present, how big a sector is, how many sectors in a block
- The root directory, unlike directories in the data area, has a finite size
  - for FAT12, 14 sectors
  - Each sector is 512 bytes, and each directory entry is 32 bytes
  - 16 directory entries per sector = 224 entries
- Other directories have variable lengths, but each entry is 32 bytes long as well



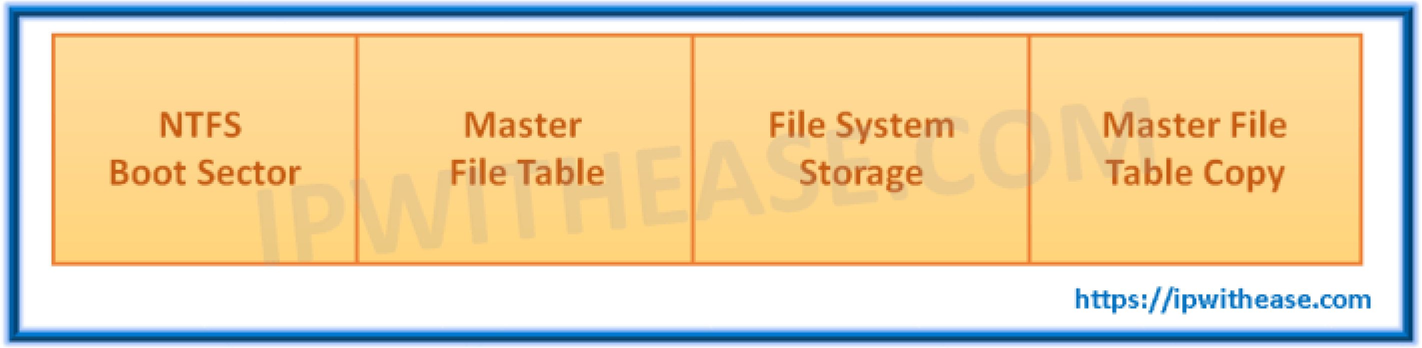
Structure of directory entries

- Directory entries are 32 bytes long
- A directory entry has the format below:
  - Attributes: a collection of bit flags:
    - read only,
    - hidden,
    - system file,
    - root directory,
    - sub-directory,
    - file has been archived,
    - last 2 unused



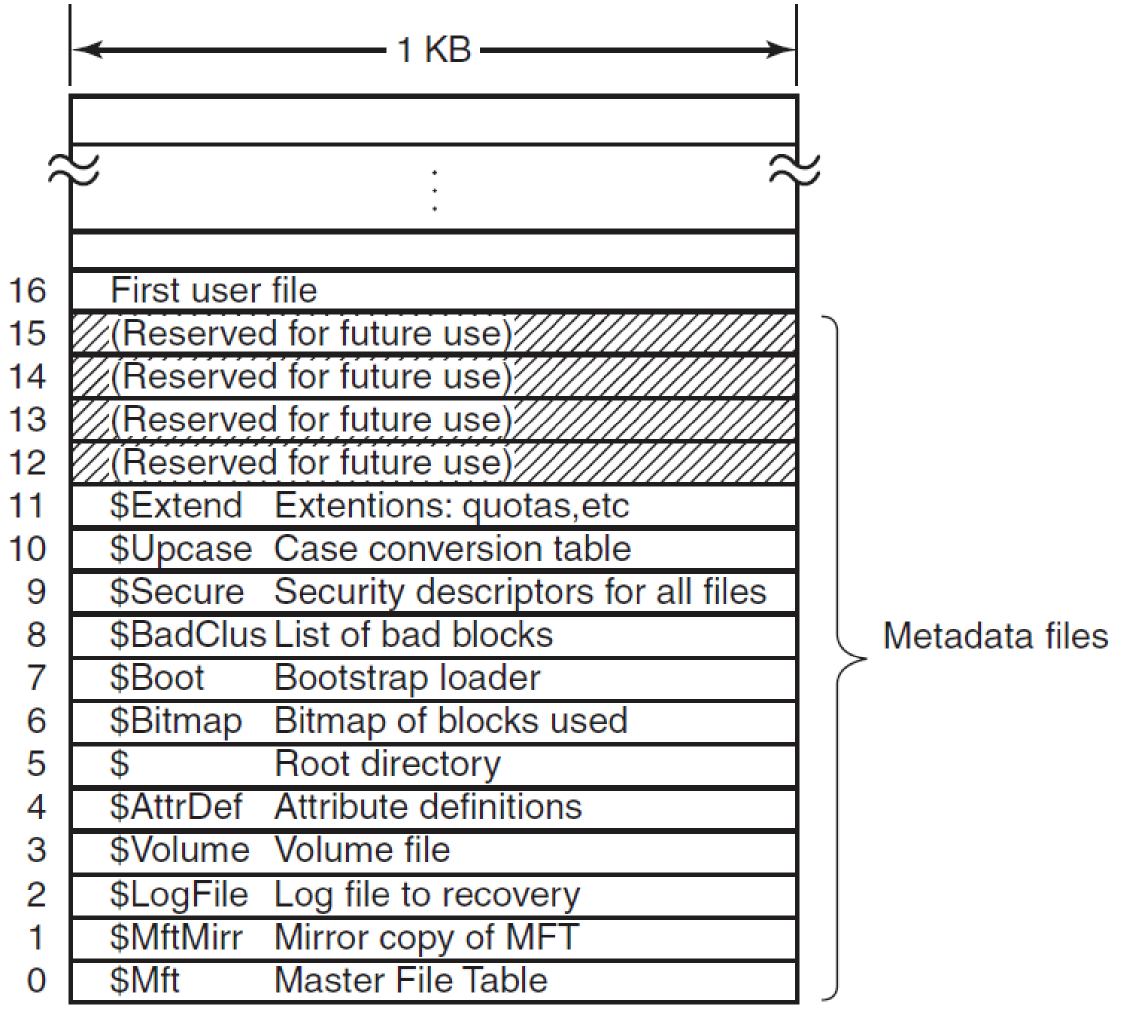
Windows NTFS

- NTFS is the file system currently uses by Windows
- Each partition is organized as a linear sequence of blocks, with the block size being fixed for each partition, ranging from 512 bytes to 64 KB, depending on the partition size. Most NTFS disks use 4-KB blocks.
- Blocks are referred to by their offset from the start of the partition using 64-bit numbers.
- Using this scheme, the system can calculate a physical storage offset (in bytes) by multiplying the block number by the block size.
- Each partition is structured as below



NTFS Master file table

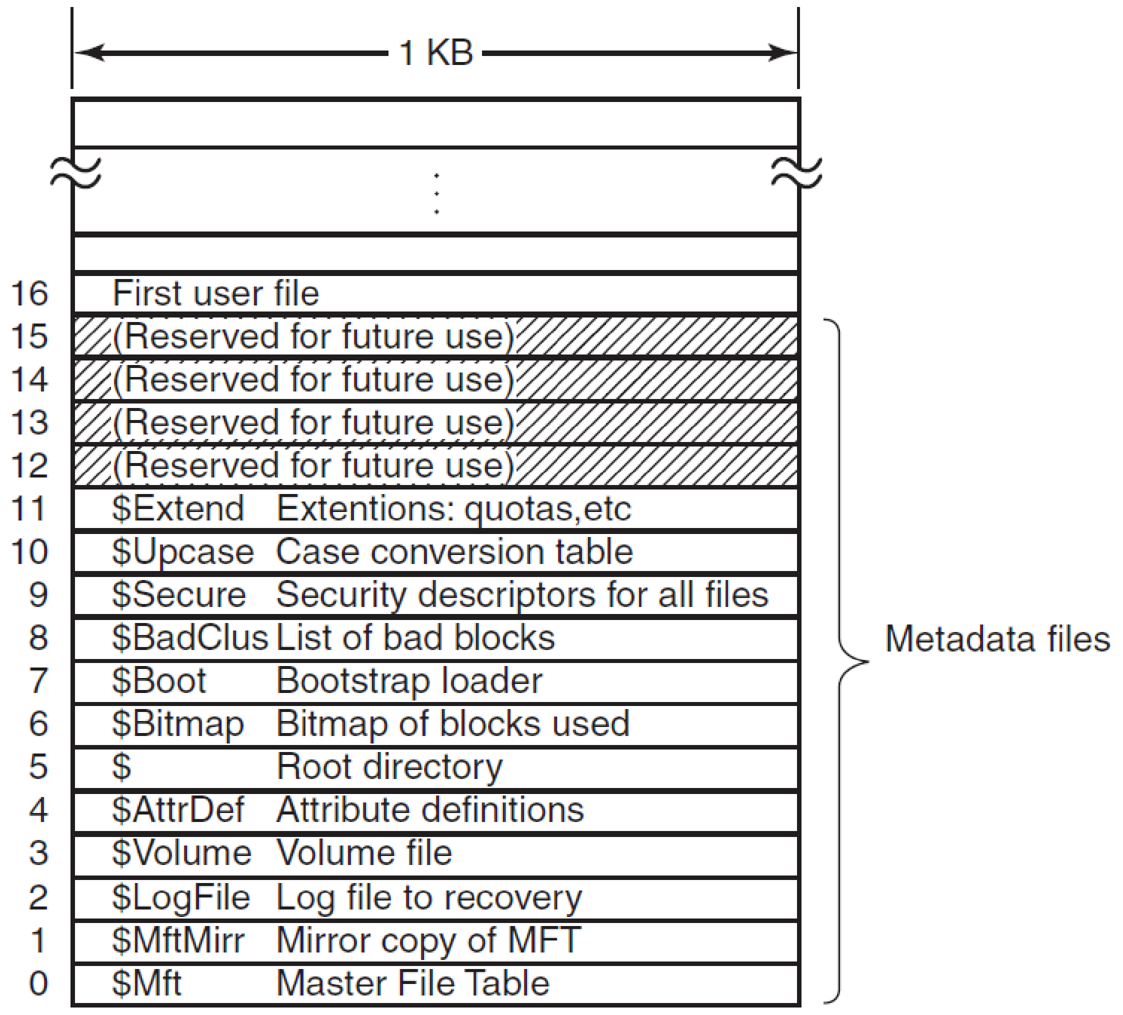
- The main data structure in each partition is the **MFT** (**Master File Table**), an array of fixed-size 1-KB entries.
- Each MFT entry describes one file or one directory.
- It contains the file's attributes, such as its name and timestamps, and the list of disk addresses where its blocks are located.
- The MFT is itself a file, the file can grow as needed, up to a maximum size of 2^{48} entries.



16	First user file	
15	(Reserved for future use)	
14	(Reserved for future use)	
13	(Reserved for future use)	
12	(Reserved for future use)	
11	$Extend	Extentions: quotas,etc
10	$Upcase	Case conversion table
9	$Secure	Security descriptors for all files
8	$BadClus	List of bad blocks
7	$Boot	Bootstrap loader
6	$Bitmap	Bitmap of blocks used
5	$	Root directory
4	$AttrDef	Attribute definitions
3	$Volume	Volume file
2	$LogFile	Log file to recovery
1	$MftMirr	Mirror copy of MFT
0	$Mft	Master File Table

NTFS Master file

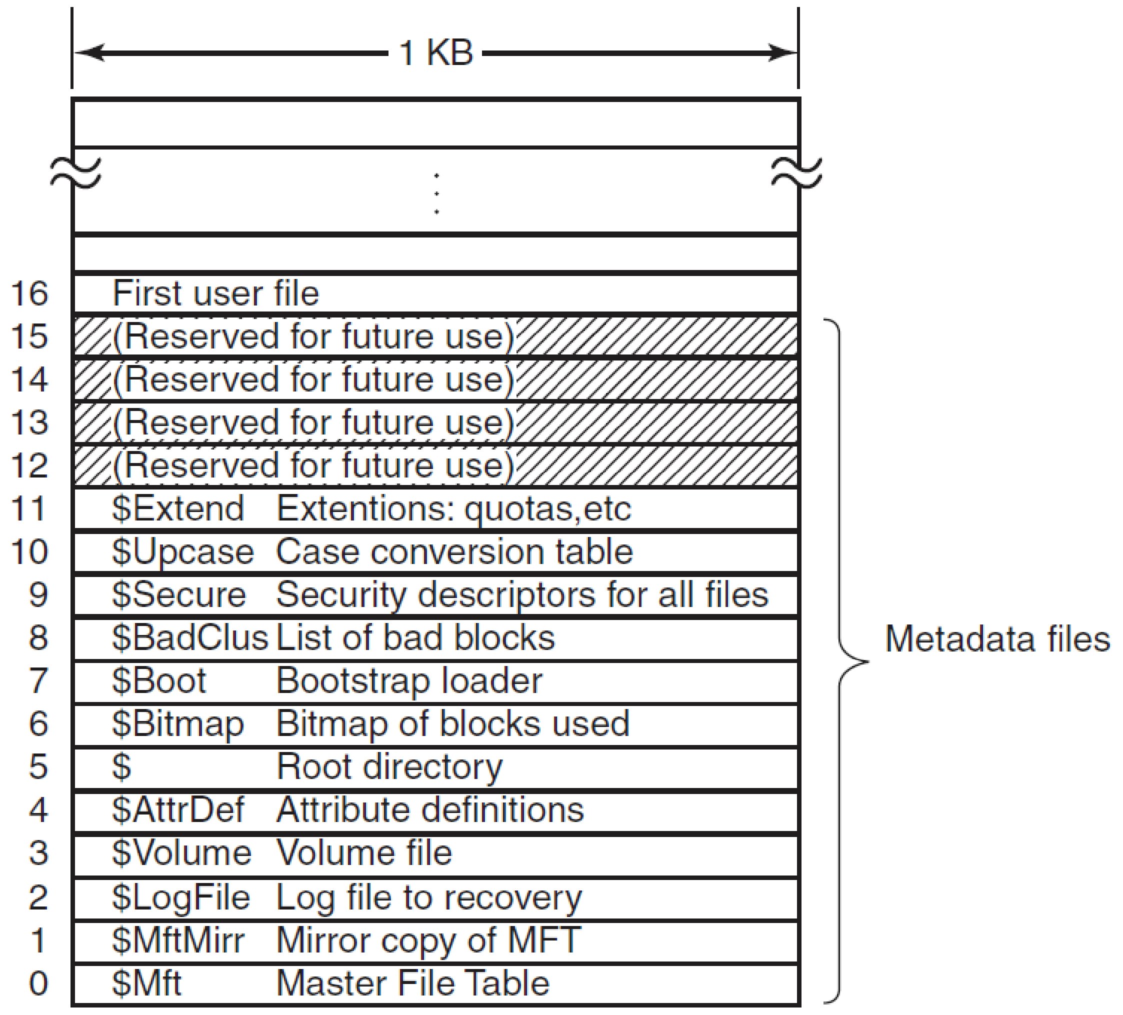
- The first 16 MFT entries are reserved for NTFS metadata files
- Each of these 16 entry describes a normal file that has attributes and data blocks, just like any other file.
- Each of these files has a name that begins with a dollar sign to indicate that it is a metadata file.
- Entry 0 describes the MFT file itself. In particular, it tells where the blocks of the MFT file are located so that the system can find the MFT file.
- The address of the first MTF block is in the boot block



16	First user file	
15	(Reserved for future use)	
14	(Reserved for future use)	
13	(Reserved for future use)	
12	(Reserved for future use)	
11	$Extend	Extentions: quotas,etc
10	$Upcase	Case conversion table
9	$Secure	Security descriptors for all files
8	$BadClus	List of bad blocks
7	$Boot	Bootstrap loader
6	$Bitmap	Bitmap of blocks used
5	$	Root directory
4	$AttrDef	Attribute definitions
3	$Volume	Volume file
2	$LogFile	Log file to recovery
1	$MftMirr	Mirror copy of MFT
0	$Mft	Master File Table

Some metadata Master file entries

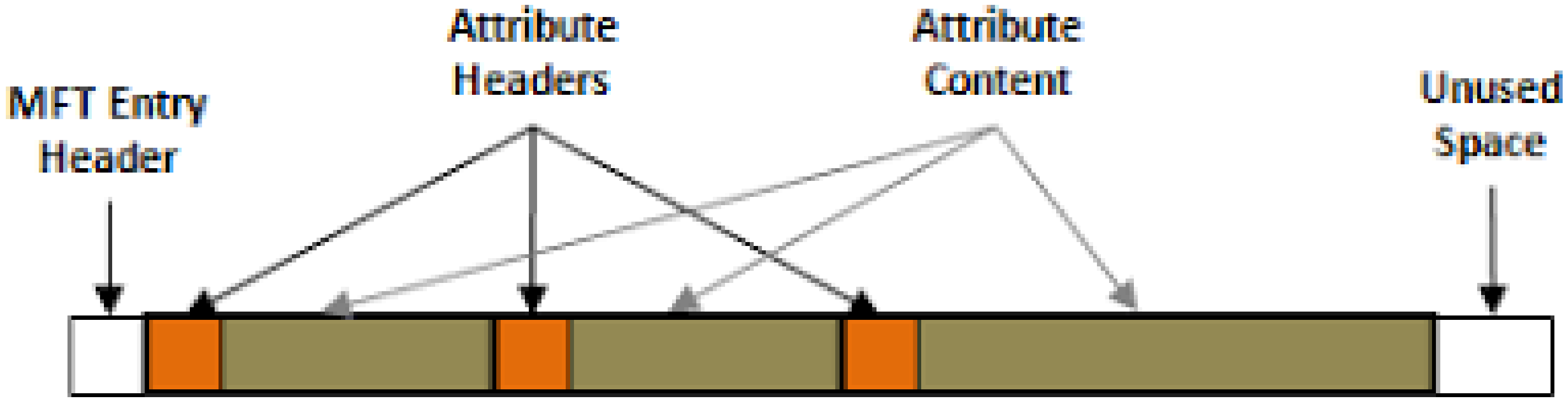
- MTF entry 1 is a duplicate of entry 0.
- MTF entry 3 contains information about the partition, such as its size, label, and version.
- MTF entry 5: the root directory, which itself is a file and can grow to arbitrary length.
- Free space on the partition is kept track of with a bitmap. The bitmap is itself a file, and its attributes and disk addresses are given in MFT entry 6.
- MTF entry 11 is a directory containing miscellaneous files for things like disk quotas.


1 KB
Metadata files

	⋮	
16	First user file	
15	(Reserved for future use)	
14	(Reserved for future use)	
13	(Reserved for future use)	
12	(Reserved for future use)	
11	$Extend	Extentions: quotas,etc
10	$Upcase	Case conversion table
9	$Secure	Security descriptors for all files
8	$BadClus	List of bad blocks
7	$Boot	Bootstrap loader
6	$Bitmap	Bitmap of blocks used
5	$	Root directory
4	$AttrDef	Attribute definitions
3	$Volume	Volume file
2	$LogFile	Log file to recovery
1	$MftMirr	Mirror copy of MFT
0	$Mft	Master File Table

Structure of master file entries

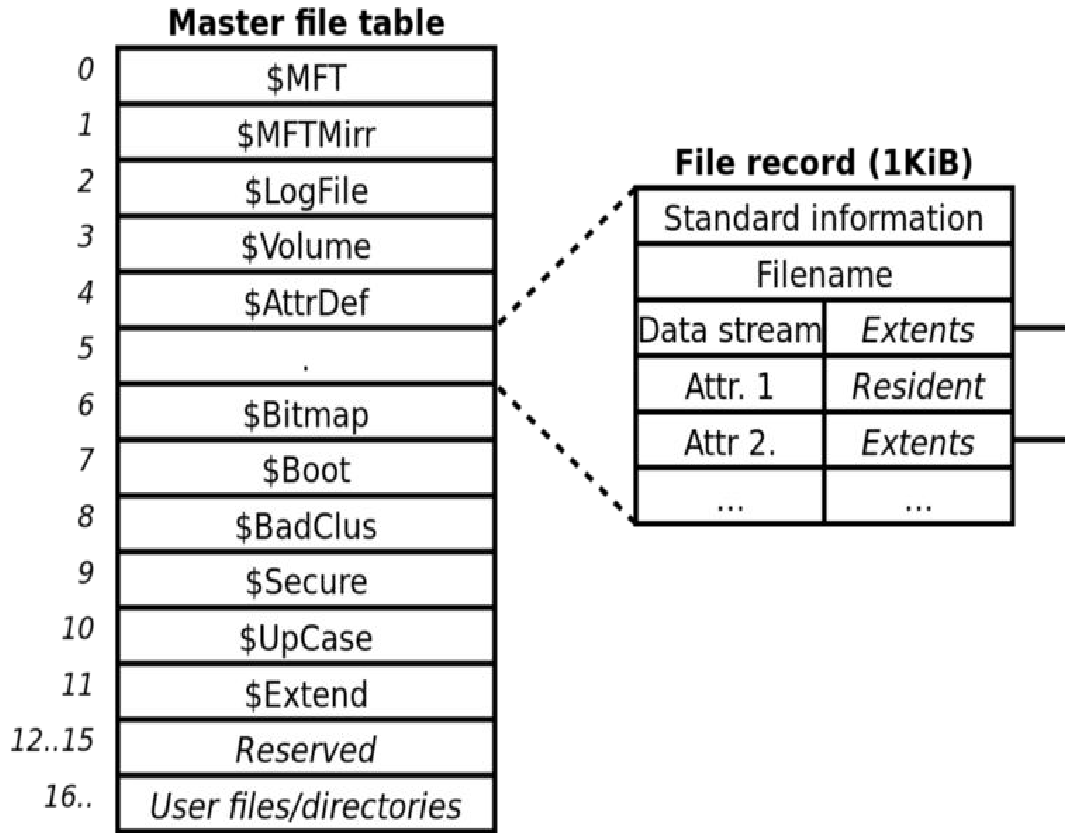
- **The first 42 bytes store the header. The header contains 12 fields.**
- **The other 982 bytes do not have a fixed structure, and are used to keep attributes.**
- Each MFT entry consists of a sequence of (attribute header, value) pairs.
- Each attribute begins with a header telling which attribute this is and how long the value is.
- If the attribute value is short enough to fit in the MFT entry, it is placed there.
- If it is too long, it is placed elsewhere on the disk and a pointer to it is placed in the MFT entry



Structure of master file entries

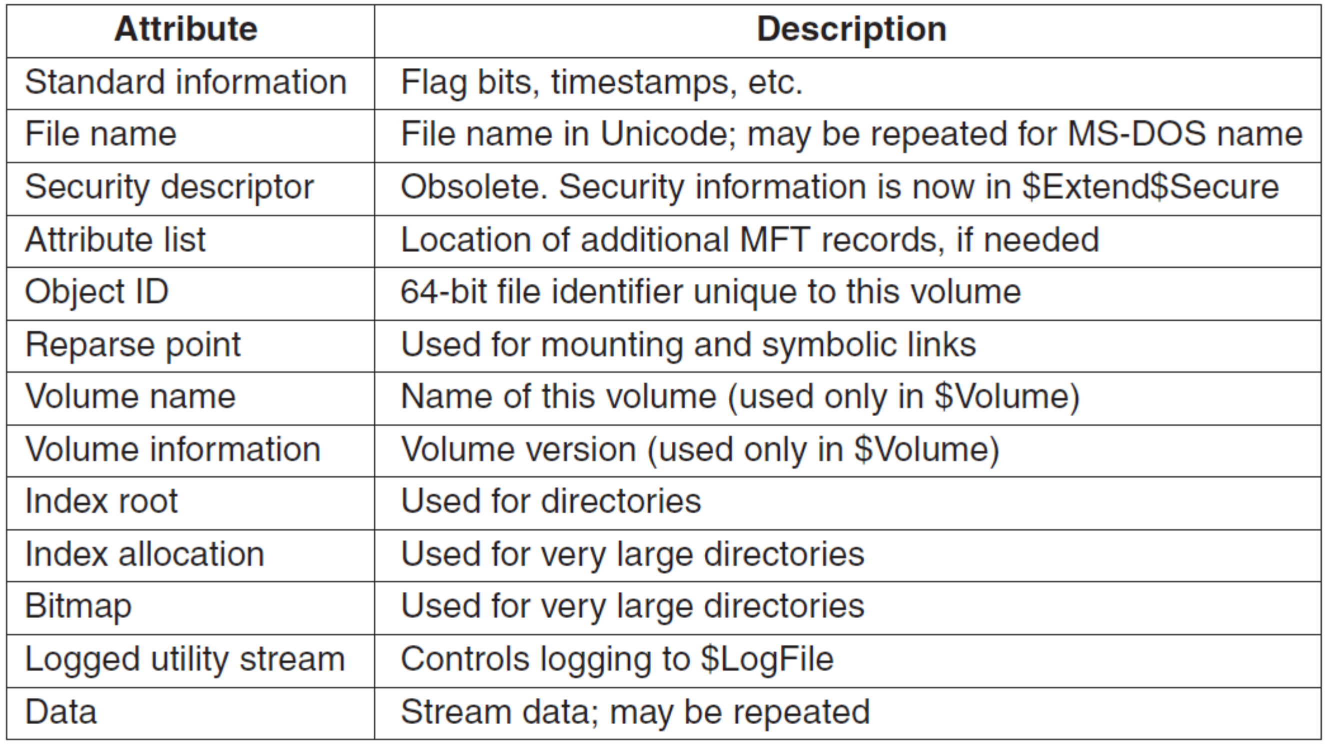
- NTFS defines 13 attributes that can appear in MFT entries. Each attribute header identifies the attribute and gives the length and location of the value field

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated



Structure of master file records

- The standard information field contains the file owner, security information, the timestamps. This field is always present.
- The value of the file name is obviously the name of the file
- NTFS files have an ID associated with them that is like the i-node number in UNIX. Files can be opened by ID
- An NTFS file has one or more data streams associated with it. For each stream, the stream name, if present, goes in this attribute header. Following the header is either a list of disk addresses telling which blocks the stream contains, or for streams of only a few hundred bytes (and there are many of these), the stream itself.



Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated

Structure of master file entries

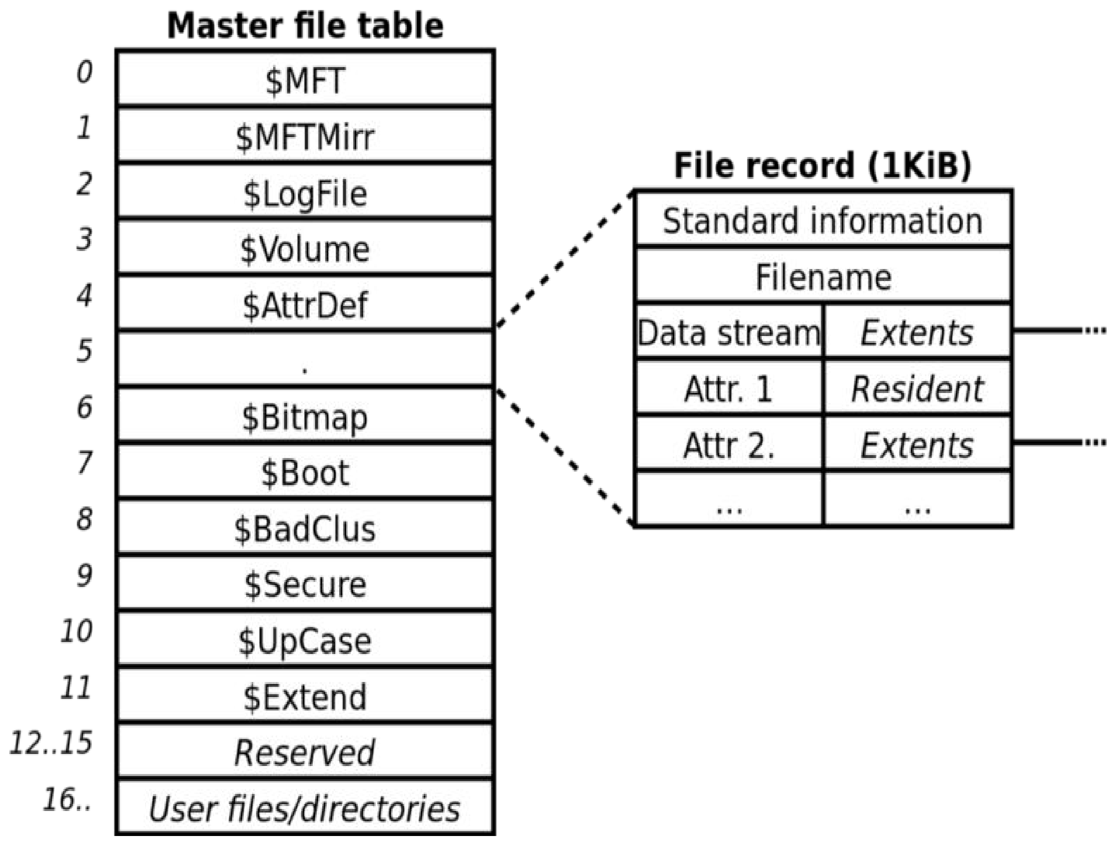
- Usually, attribute values follow their attribute headers directly, but if a value is too long to fit in the MFT entry (like data), it may be put in separate disk blocks.
- Such an attribute is said to be a **nonresident attribute**.

- The headers for resident attributes are 24 bytes long
- The headers of nonresident attributes are longer because they contain information about where to find the attribute on disk.

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated

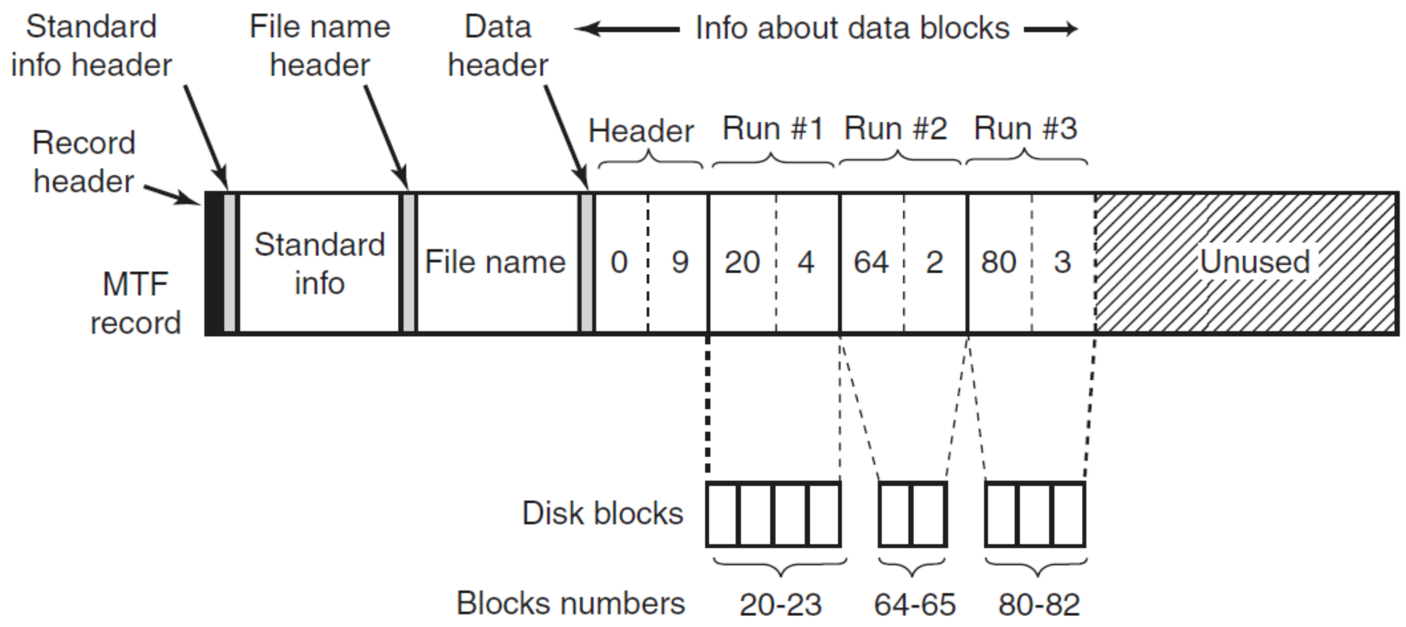
NTFS file organization

- An NTFS file is not just a linear sequence of bytes, as UNIX files
- Instead, a file consists of multiple attributes, each represented by a stream of bytes.
- Most files have a few short streams, such as the name of the file and its 64-bit object ID, plus one long (unnamed) stream with the data
- However, a file can also have two or more (long) data streams as well.
- Each stream has a name consisting of the file name, a colon, and the stream name, as in *foo:stream1*.



Storage allocation

- The blocks in a stream are described by a sequence of records, each one describing a sequence of logically contiguous blocks
- Each record begins with a header giving the offset of the first block within the stream. Next is the offset of the first block not in the record
- Each record header is followed by one or more pairs, each giving a disk address and run length. The disk address is the offset of the disk block from the start of its partition; the run length is the number of blocks in the run



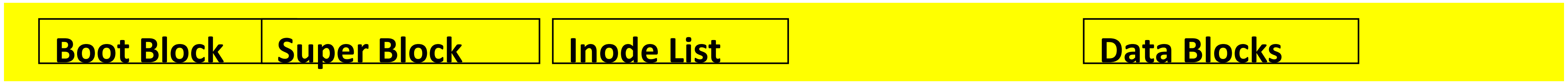
Unix/Linux files systems

- Unix/Linux structure on disk
- System calls related to files and directories
- Linux ext2, ext3, ext4
- The network file system (NFS)

Linux structure on disk: boot block

- Boot Block
- Super Block
- Inode List
- Data Block

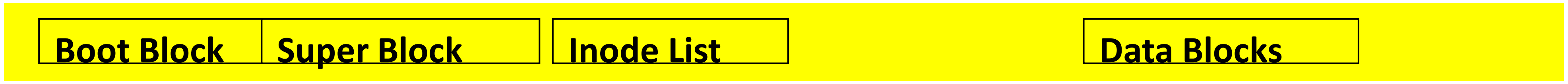
  - Like for Windows file systems, the boot block occupies the beginning of a file system, the first sector
  - May contain the bootstrap code that is read into the machine at boot time
  - Only one boot block is required to boot the system, but every file system may contain a boot block



Super block

- Boot Block
- Super Block
- Inode List
- Data Block

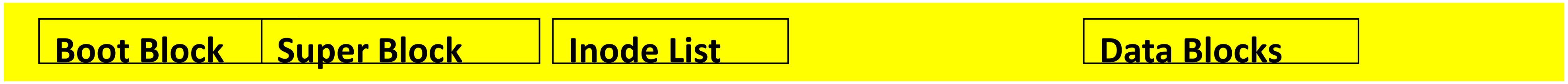
- The super block describes the file system:
  - Number of bytes in the file system
  - Number of files it can store
  - Where to find free space in the file system
  - Additional data to manage the file system



i-nodes

- Boot Block
- Super Block
- Inode List
- Data Block

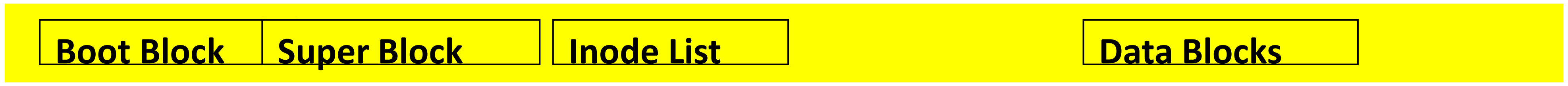
- Inode List:
  - Each i-node is the internal representation of a file, i.e. the mapping of the file to disk blocks
  - The attributes of the file: owner, permissions, date of creation, etc
- The i-node list contains all of the i-nodes present in an instance of a file system



Data blocks

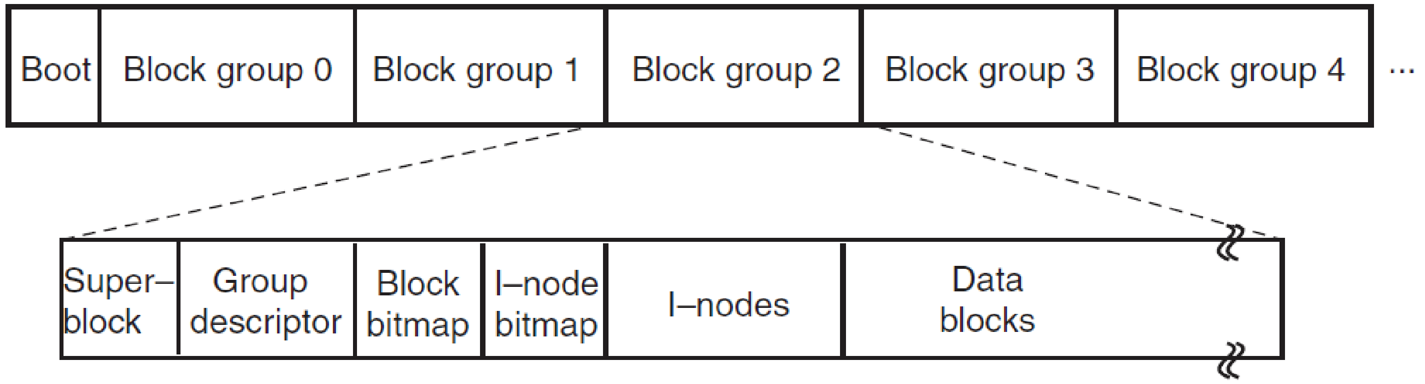
- Boot Block
- Super Block
- Inode List
- Data Block

  - Data Blocks:
    - Contain the file data in the file system
    - Additional administrative data
    - An allocated data block can belong to one and only one file in the file system



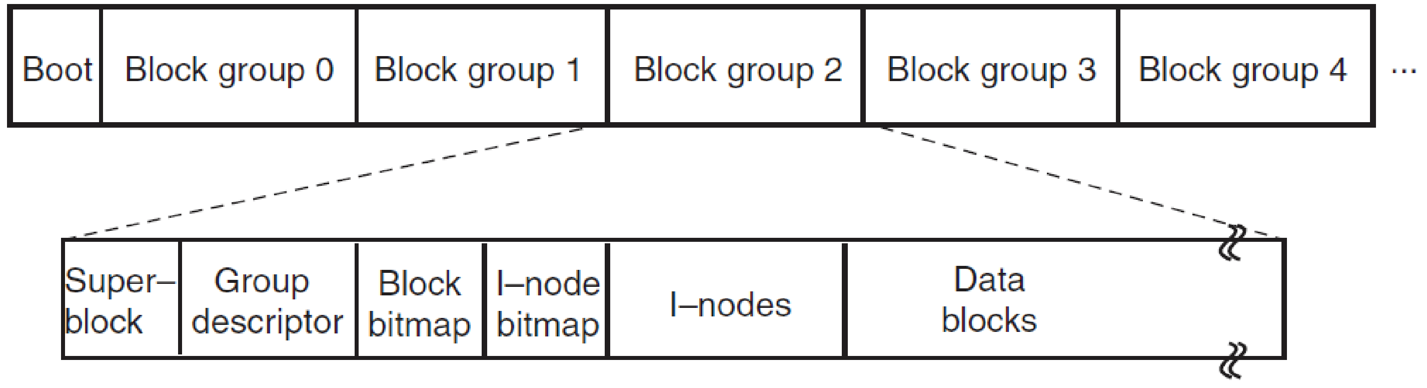
The Linux ext2 file system

- Linux supports over 130 different file systems, the standard Linux file system is known as the **extended file system**, with the most common versions being ext2, ext3 and ext4
- ext2 divides the logical partition that it occupies into Block Groups, the file system has the disk layout has shown below
- Each group includes **data blocks and inodes** stored in adjacent tracks.
- This structure allow files stored in a single block group to be accessed with a lower average disk seek time



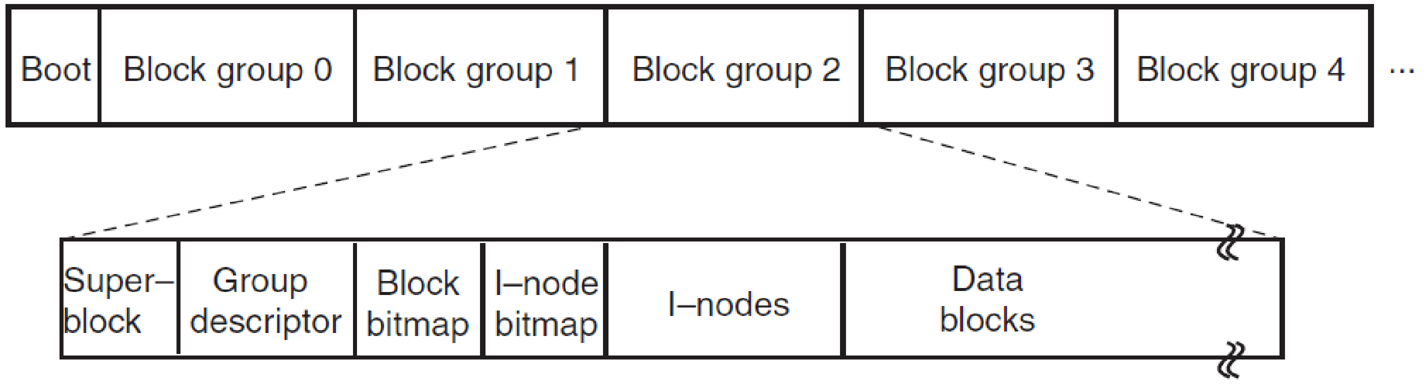
The Linux ext2 file system

- Block 0 (boot block) is not used by Linux, contains code to boot the computer
- All block groups in the filesystem have the same size and are stored sequentially
- The block groups have the layout as below:
  - Superblock: Contains a copy of the of the filesystem's superblock for the partition
  - Group descriptor: Contains information about the location of the bitmaps, the number of free blocks and i-nodes in the group, the number of directories in the group



The Linux ext2 file system

- Two bitmaps are used to keep track of the free blocks and free i-nodes
  - Block bitmap identifies the free blocks inside the group
  - i-node bitmap identifies the free i-nodes inside the group
- Each map is one block long. With a 1-KB block, this design limits a partition to 8192 blocks and 8192 i-nodes
- i-nodes are 128 bytes long



Structure of the directory entries

- The file must be opened using the file's path name. The path name is parsed to extract individual directories.
- Each directory entry consists of four fixed-length fields and one variable-length field.

- The first field is the i-node number, 19 for the file *colossal*, 42 for the file *voluminous*, and 88 for the directory *bigdir*.
- Next comes a field rec len, telling how big the entry is (in bytes),
- Next, the type field: file, directory
- Last field is the length of the actual file name in bytes, 8, 10, and 6 in this example

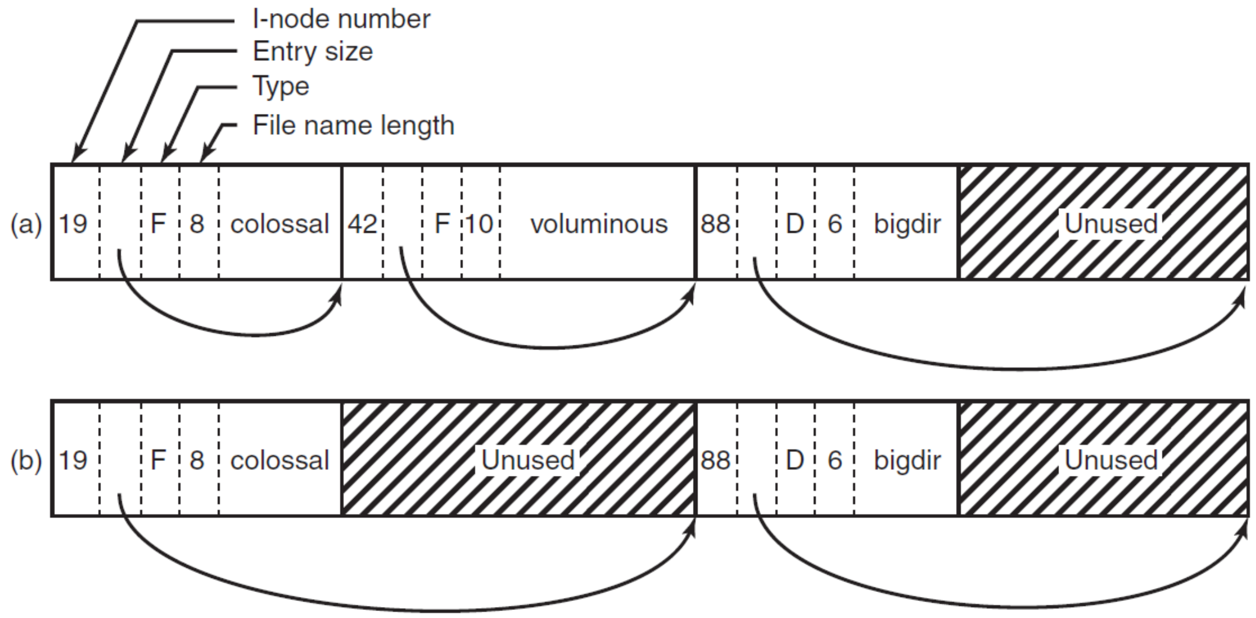
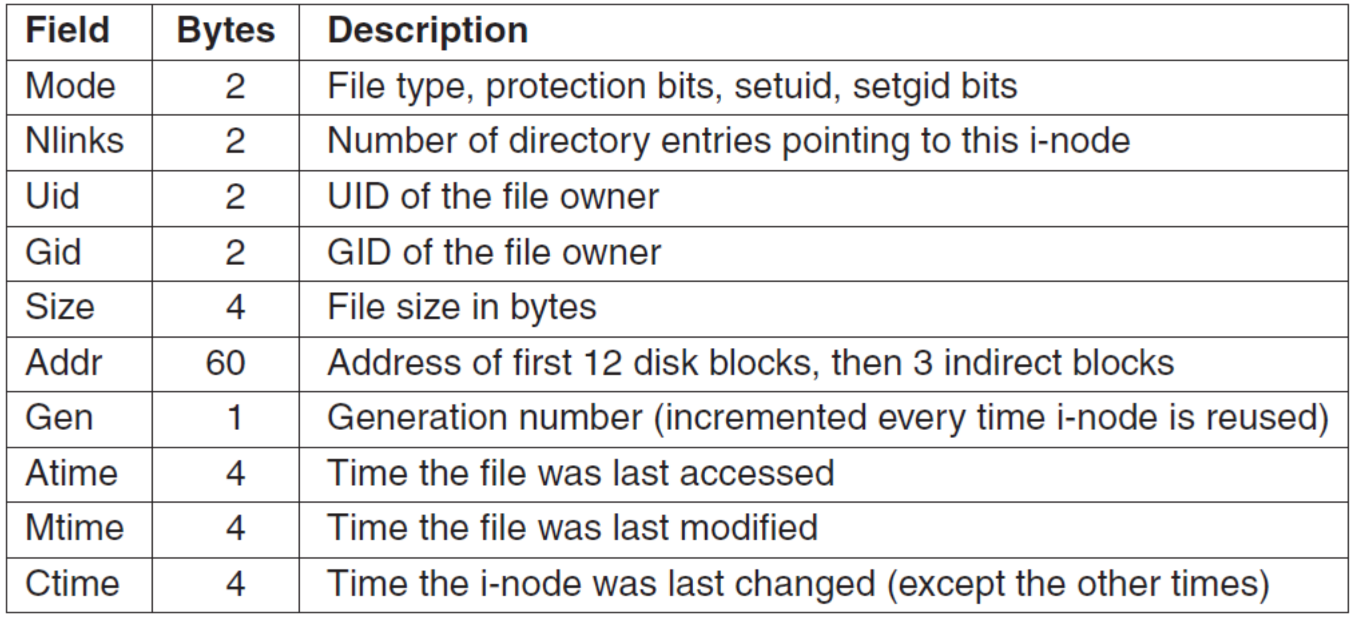


Figure 10-32. (a) A Linux directory with three files. (b) The same directory after the file *voluminous* has been removed.

Structure of i-nodes

- The i-node number of a file is used as an index into the i-node table (on disk) to locate the corresponding i-node and bring it into memory.
- The i-node is put in the **i-node table**, a kernel data structure that holds all the i-nodes for currently open files and directories. The format of the i-node entries, must contain all the fields returned by the stat system call



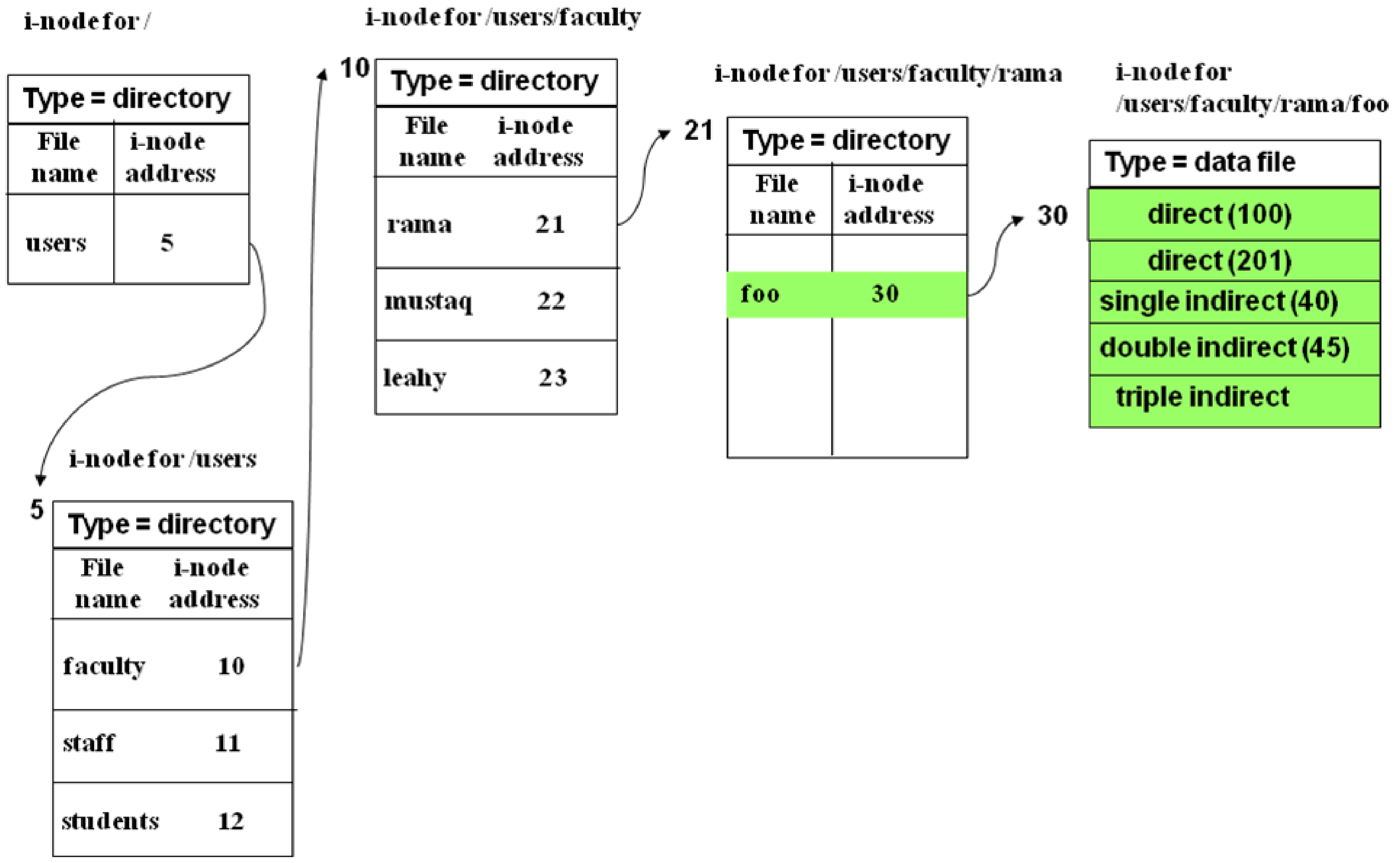
Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	60	Address of first 12 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

Structure of i-nodes

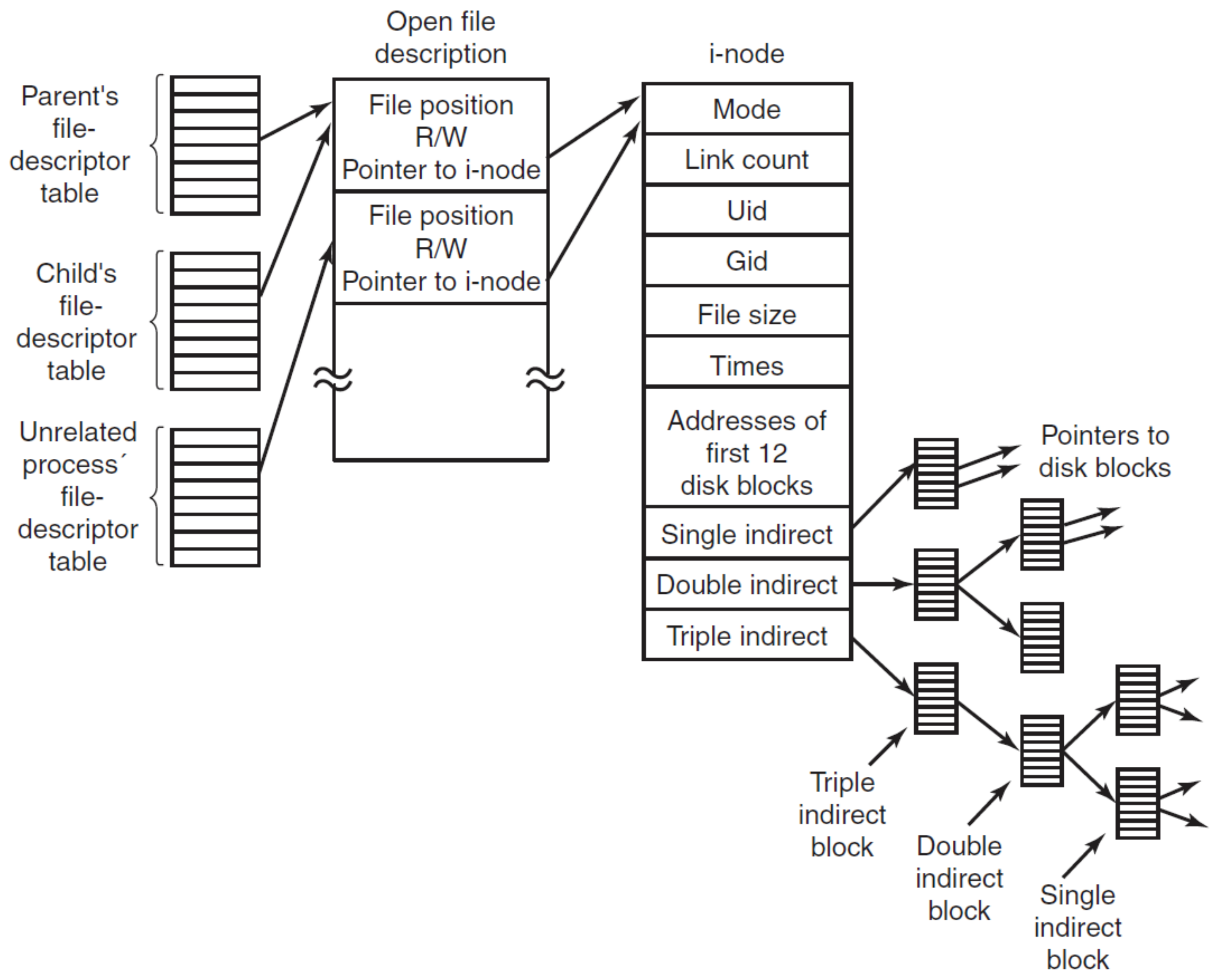
- Not shown in the previous table is the pointers to the blocks holding the file
- The i-node contains the disk addresses of the first 12 blocks of a file.
- If the file position pointer falls in the first 12 blocks, the block is read and the data are copied to the user.
- For files longer than 12 blocks, a field in the i-node contains the disk address of a **single indirect block**. This block contains the disk addresses of more disk blocks.
- For example, if a block is 1 KB and a disk address is 4 bytes, the single indirect block can hold 256 disk addresses. Thus this scheme works for files of up to 268 KB.
- Beyond that, a **double indirect block** is used. It contains the addresses of 256 single indirect blocks, each of which holds the addresses of 256 data blocks. This mechanism is sufficient to handle files up to 10 + 216 blocks (67,119,104 bytes)
- **Triple indirect block**. can handle file sizes of 16 GB for 1-KB blocks. For 8-KB block sizes, the addressing scheme can support file sizes up to 64 TB.

Unix/Linux hierarchical file naming

- Linux hierarchical naming:
- /users/faculty/rama/foo
- Each part of the file name corresponds to an i-node which form part of a tree like structure where all but the leaf nodes are directory files (which are i-nodes)
- Each directory entry contains a type which indicates if it is a directory or a data file



Relation between file descriptors and i-nodes: Linux



Implementing directories

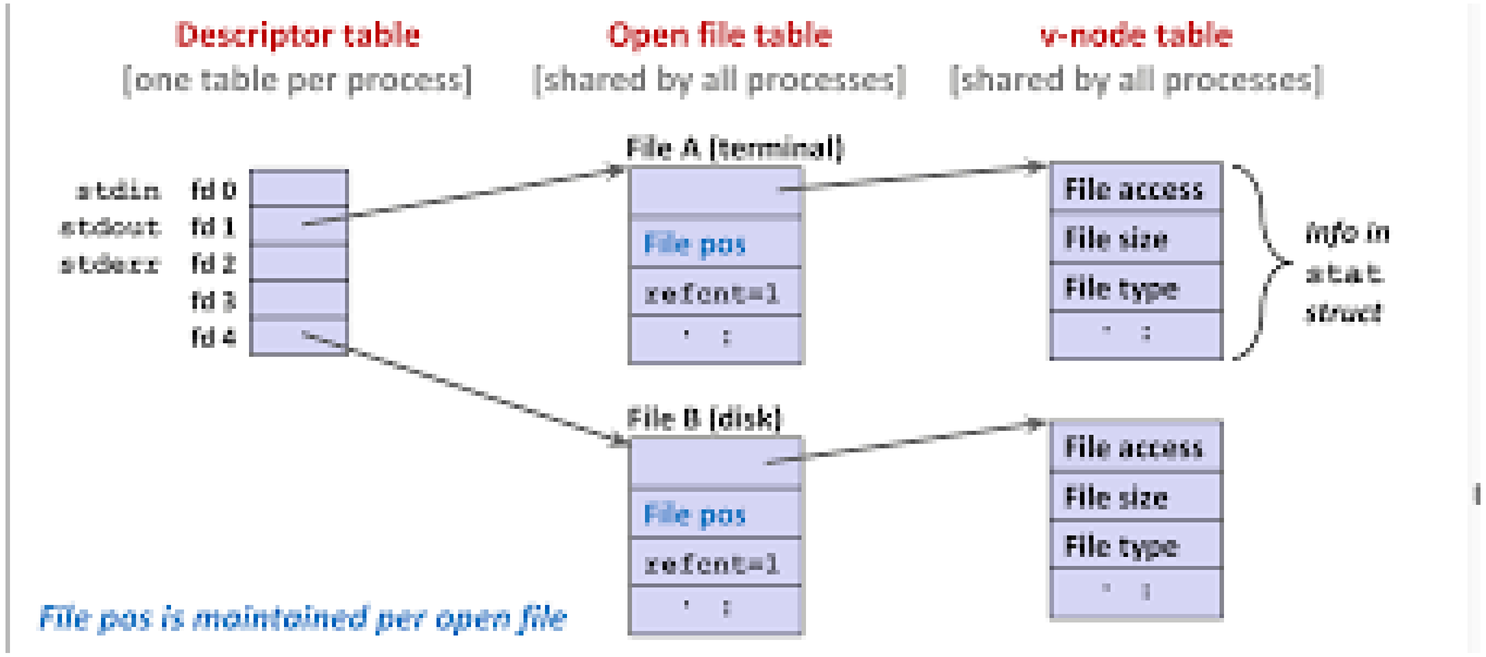
- Before a file can be read, it must be opened.
- When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry on the disk.
- Then the directory is loaded in main memory where it can be read.
- The directory entry provides the information needed to find the disk blocks of the file.
- Depending on the system this information may be:
    - the disk address of the entire file (with contiguous allocation),
    - the number of the first block (both linked- list schemes),
    - or the number of an i-node.
- In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.

Open Files

- Several pieces of data are needed to manage open files:
  - Open-file table: tracks open files
  - File pointer: pointer to last read/write location, per process that has the file open
  - File-open count: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information

File descriptor (fd)

- File descriptors index into a per-process **file descriptor table**, which indexes into a system-wide table of files opened by all processes, called the **file table**.
- The file table records the *mode* with which a file has been opened: reading, writing, appending and indexes into a third table called inode table (v-node)
- When a file is opened, the path name must be used to the file on disk. After that the read or write operations only need the fd to find the file.
- Linux file descriptors of a process can be accessed in `/proc/PID/fd/`



Opening a file

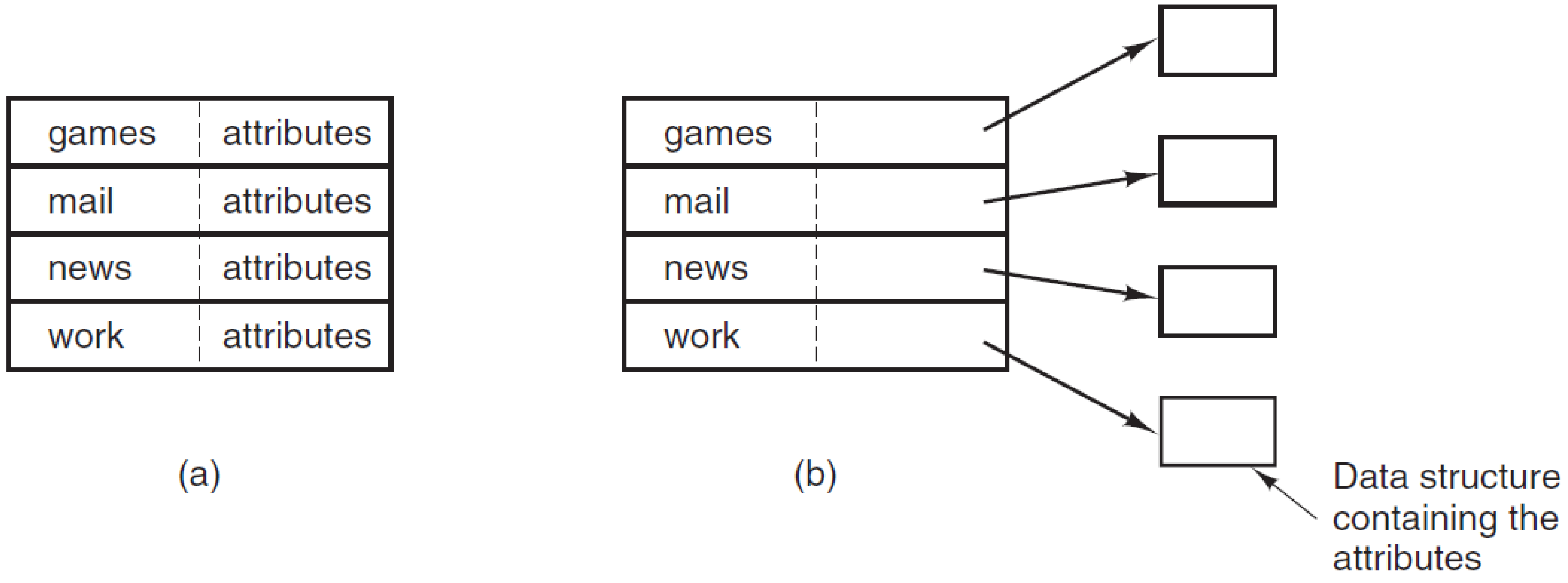
- The open() system call passes a file name to the file system.
- The open() system call first searches the open-file table to see if the file is already in use by another process.
- If it is, an entry is created into the process file descriptor table pointing to the existing open-file table.
- If the file is not already open, the directory structure is searched for the given file name.
- Once the file is found, its i-node is copied in main memory and an entry in the open-file table is created pointing to the new i-node
- Next, an entry is made in the process file descriptor, with a pointer to the newly created entry in the open-file table
- The open() call returns a pointer to the appropriate entry in the file descriptor of the process.
- All other file operations (read, write, ect.) are then performed via this pointer.

Opening a file: example

- Open(/usr/me/mailbox), not already open
  - 1-find entry of “usr” in / and get the i-node of usr on disk
  - 2-load i-node in main memory
  - 3-find blocks of the directory file usr on disk
  - 4-load the directory usr in main memory
  - 5-read usr until find entry of directory “me” and read the number of the corresponding i-node
  - 6-find i-node of “me” on disk and load it in main memory
  - 6.1-read the directory file for me into main memory
  - 7-read the directory “me” until string “mailbox” is found
  - 8-read the i-node number of file “mailbox”
  - 9-find the i-node on disk and load it in main memory
  - 10-create an entry in the open file table and makes it point to the i-node for the mailbox file
  - 11-create an entry in the file descriptor table and makes it point to the corresponding entry in the open file table
  - 12 returns the entry number to the calling process
- The system usually don’t need the absolute path, rather the relative path from the current working directory is enough

Implementing directories

- Another issue is where the attributes should be stored.
- Every file system maintains various file attributes, such as each file's owner and creation time, and they must be stored somewhere.
- One obvious possibility is to store them directly in the directory entry.
- For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries

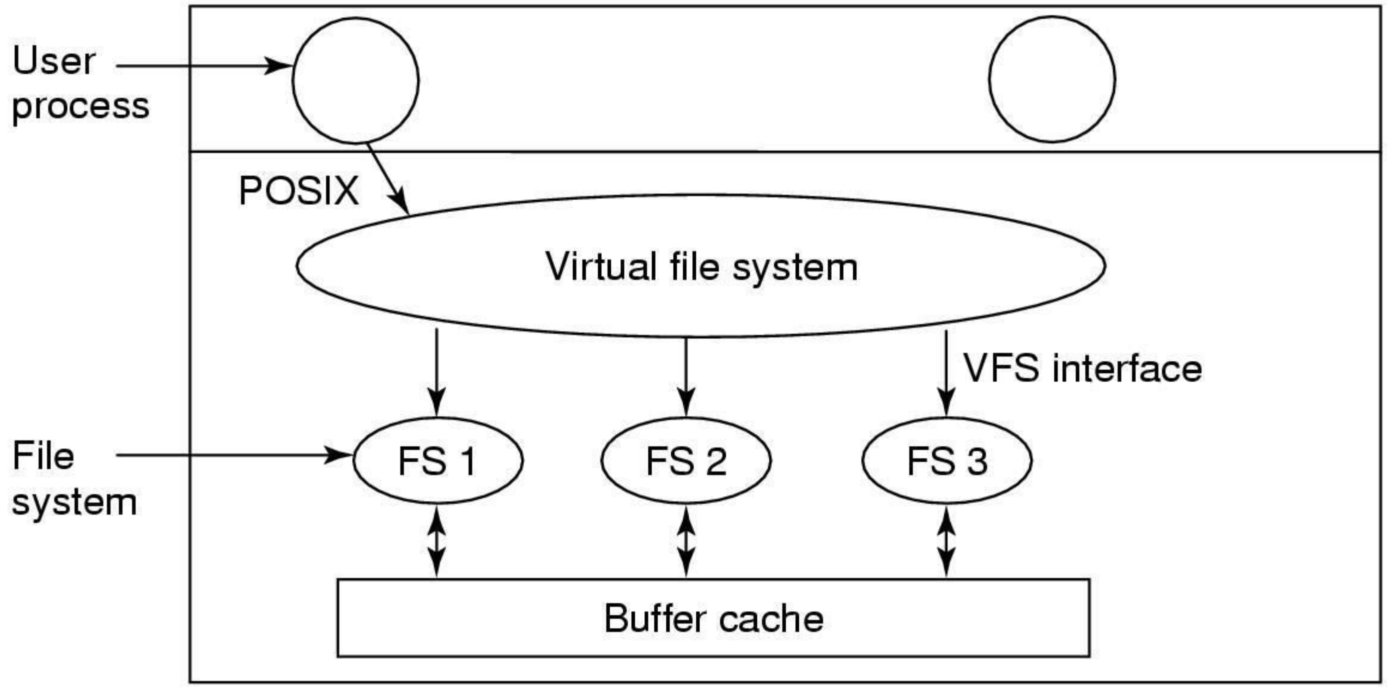


Multiple file systems

- A computer or an operating system could use different file systems
- Windows may have a main NTFS file system, but also a legacy FAT -32 or FAT -16 drive or partition that contains old, but still needed, data, and from time to time a flash drive, an old CD-ROM or a DVD
- Windows handles these disparate file systems by identifying each one with a different drive letter, as in *C:*, *D:*, etc.
- When a process opens a file, the drive letter is explicitly or implicitly present so Windows knows which file system to pass the request to.
- Windows does not have a virtual file system

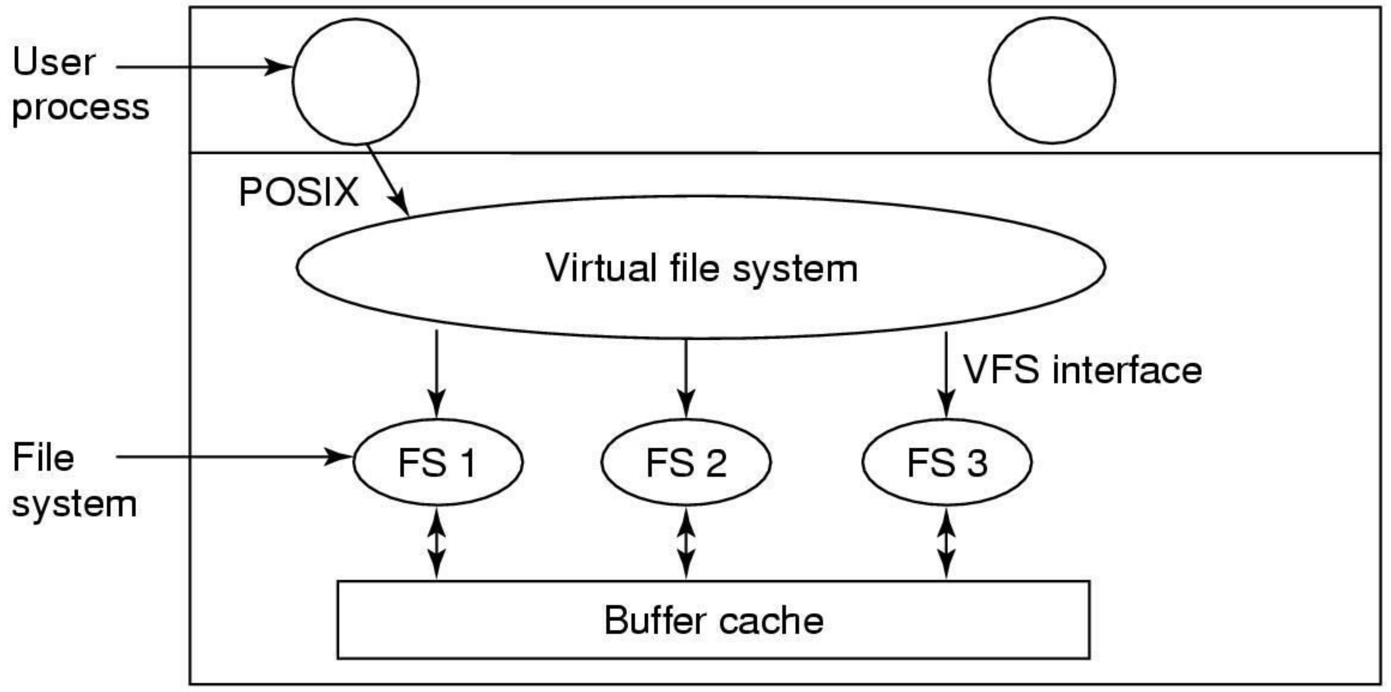
Virtual file systems

- UNIX systems integrate multiple file systems into a single VFS structure.
- For example, a Linux system could have ext2 as the root file system, with an ext3 partition mounted on */usr* and a second hard disk with a ReiserFS file system mounted on */home* as well as an ISO 9660 CD-ROM temporarily mounted on */mnt*.
- POSIX system calls such as open, read, write, lseek, are directed to the virtual file system for initial processing



Virtual file systems

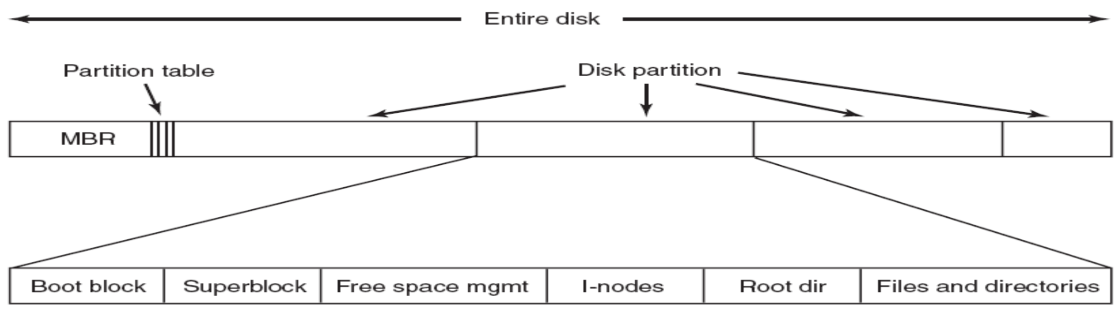
- Most file systems under the VFS are partitions on the local disk.
- They can also be a remote file systems using the **NFS** (**Network File System**) protocol.



How VFS works

- After a file system has been mounted, it can be used. For example, if a file system has been mounted on */usr* and a process makes the call
  open("/usr/include/unistd.h", O RDONLY)

- Parsing the path, the VFS sees that a new file system has been mounted on */usr* and locates its superblock by searching the list of superblocks of mounted file systems.
- The VFS find the root directory of the mounted file system and look up the path *include/unistd.h* there.



Why disk partitioning

- Partitioning allows the use of different filesystems to be installed for different kinds of files.
- Separating user data from system data can prevent the system partition from becoming full and rendering the system unusable.
  - With DOS, Microsoft Windows, and OS/2, a common practice is to use one primary partition for the active file system that will contain the operating system, the page/swap file, all utilities, applications, and user data. On most Windows consumer computers, the drive letter C: is routinely assigned to this primary partition.
  - On Unix-like operating systems it is possible to use multiple partitions on a disk device.
  - Multiple partitions allow directories such as /boot, /tmp, /usr, /var, or /home to be allocated their own filesystems. Such a scheme has a number of advantages:

•If one file system gets corrupted, the data outside that filesystem/partition may stay intact, minimizing data loss.
•Specific file systems can be mounted with different parameters, e.g., read-only, or with the execution of setuid files disabled.
•A runaway program that uses up all available space on a non-system filesystem does not fill up critical filesystems.
•Keeping user data such as documents separate from system files allows the system to be updated with lessened risk of disturbing the data.
A common minimal configuration for Linux systems is to use three partitions: one holding the system files mounted on "/" (the root directory), one holding user configuration files and data mounted on /home (home directory), and a swap partition.

Unix-like disk partitioning

- On Unix-like operating systems it is possible to use multiple partitions on a disk device.
- Multiple partitions allow directories such as /boot, /tmp, /usr, /var, or /home to be allocated their own filesystems.
- Such a scheme has a number of advantages:
  - If one file system gets corrupted, the data outside that filesystem/partition may stay intact, minimizing data loss.
  - Specific file systems can be mounted with different parameters, e.g., read-only, or with the execution of setuid files disabled.
  - A runaway program that uses up all available space on a non-system filesystem does not fill up critical filesystems.
  - A common minimal configuration for Linux systems is to use three partitions: one holding the system files mounted on "/" (the root directory), one holding user configuration files and data mounted on /home (home directory), and a swap partition.