File systems 2

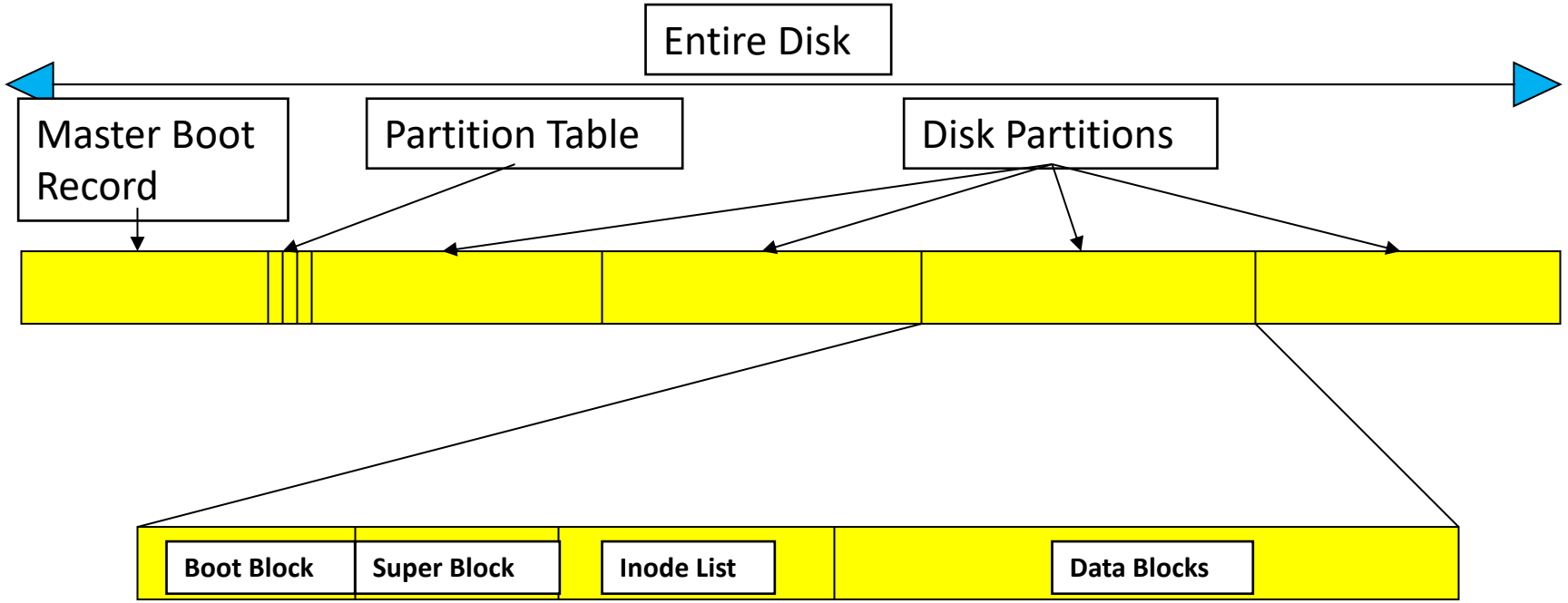
- Definitions of “file system”
- Disk partition
- Strategies for allocating disk space to files
- Windows file systems
  - FAT 12,16, 32
  - NTFS
- Linux file systems
  - Linux file structure on disk
  - ext2, ext3, ext4, nfs
  - Virtual file system
- Boot sequence

File system

- The term can be used to mean any of the following:
  - A specific implementation such as ext2, NTFS or nsf (disk)
  - A file system type residing at a location such as /dev/hda4 (tree)
  - The methods and data structures to map files on a disk or partition (OS)

Partitions

- File system instances reside on partitions
- Partitioning divides a single hard drive into many logical drives
- A partition is a contiguous set of blocks on a drive that are treated as an independent disk
- A partition table is an index that relates sections of the hard drive to partitions

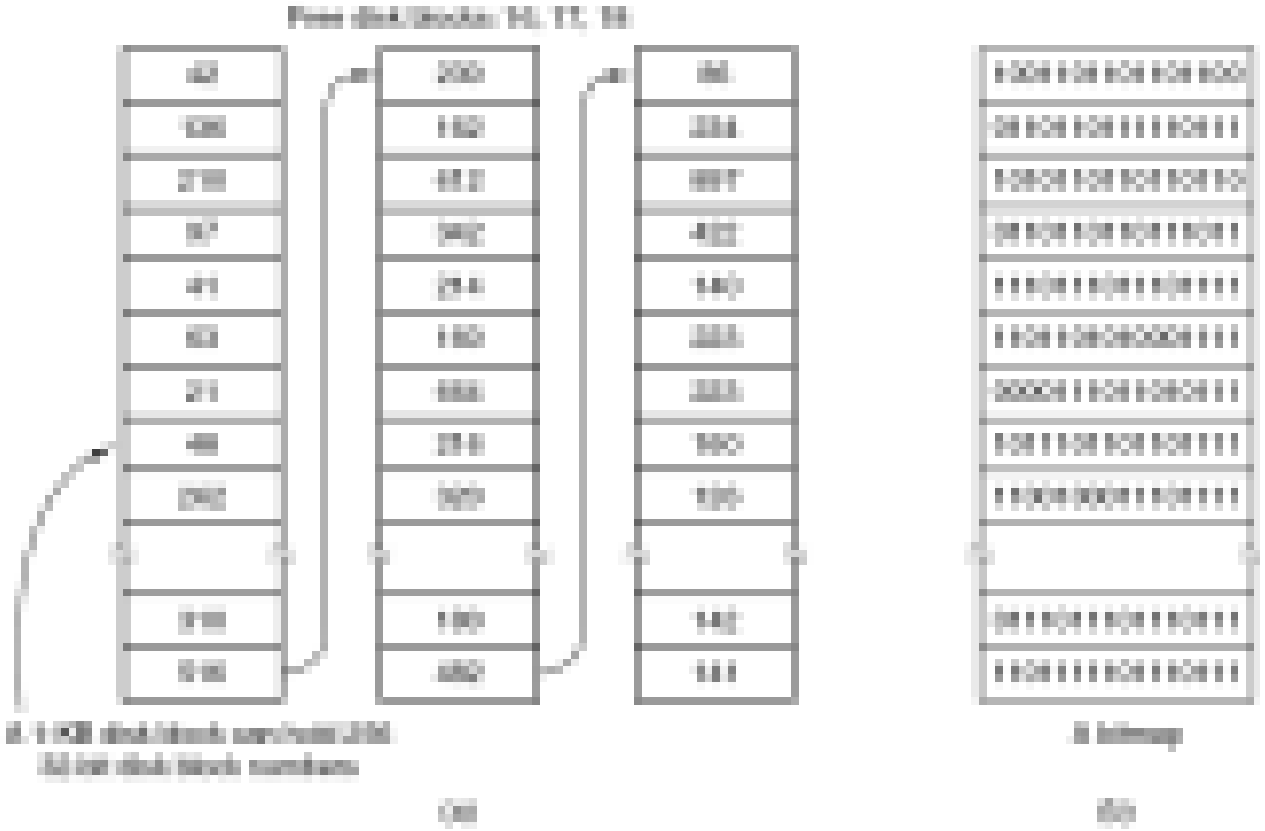


Why multiple partitions

- 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.
- If one file system gets corrupted, the data outside that filesystem/partition may stay intact, minimizing data loss.
  - A runaway program that uses up all available space on a non-system filesystem does not fill up critical filesystems
- Specific file systems can be mounted with different parameters, e.g., read-only, or read-write, or executable

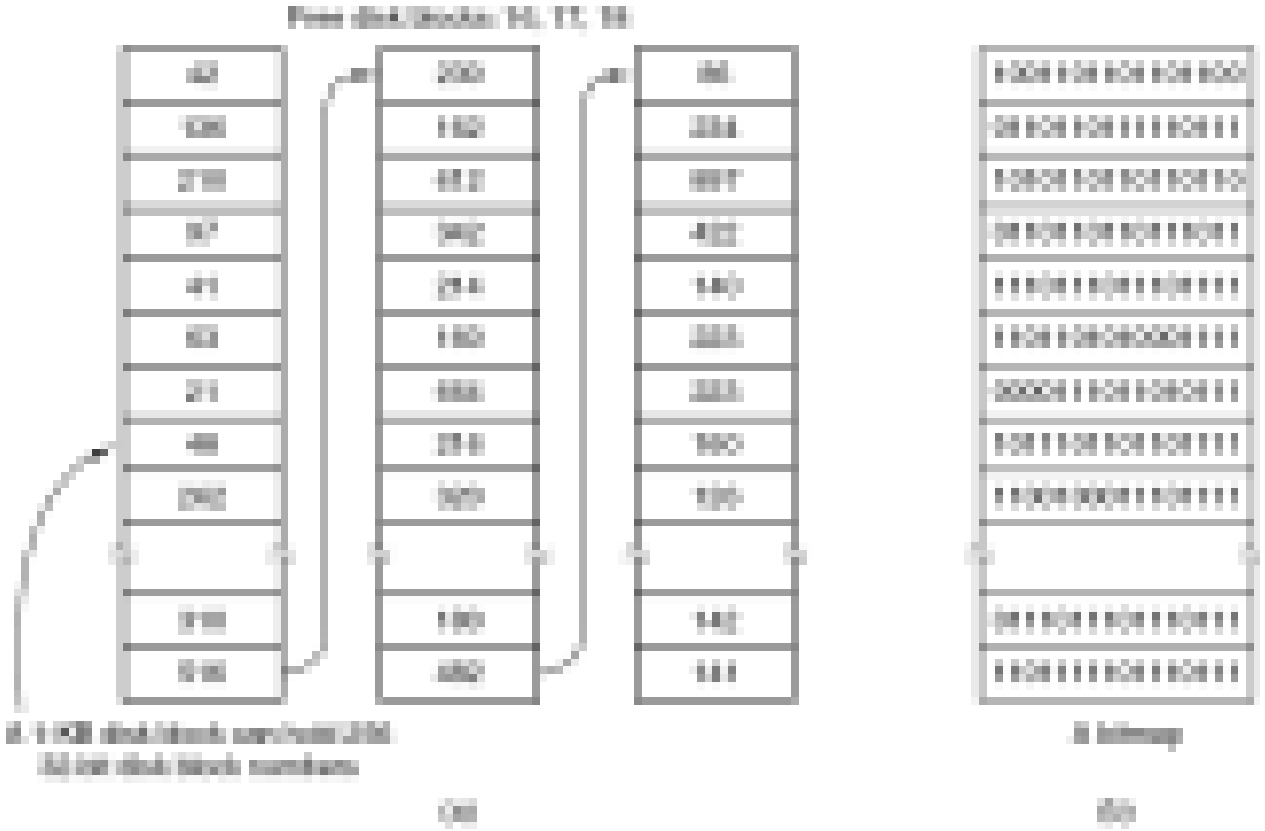
Keeping track of free blocks

- Prior to allocate blocks, we must know which blocks are free
- Free blocks recording: many approaches exist, 2 describe here
- 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.

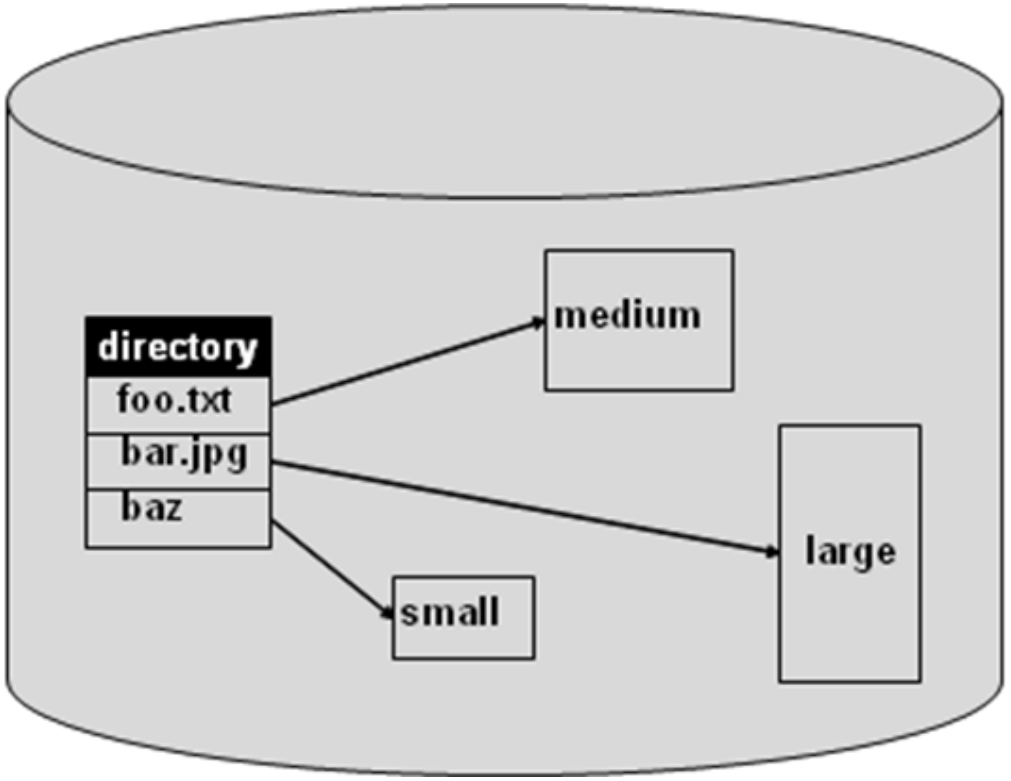


Keeping track of free blocks

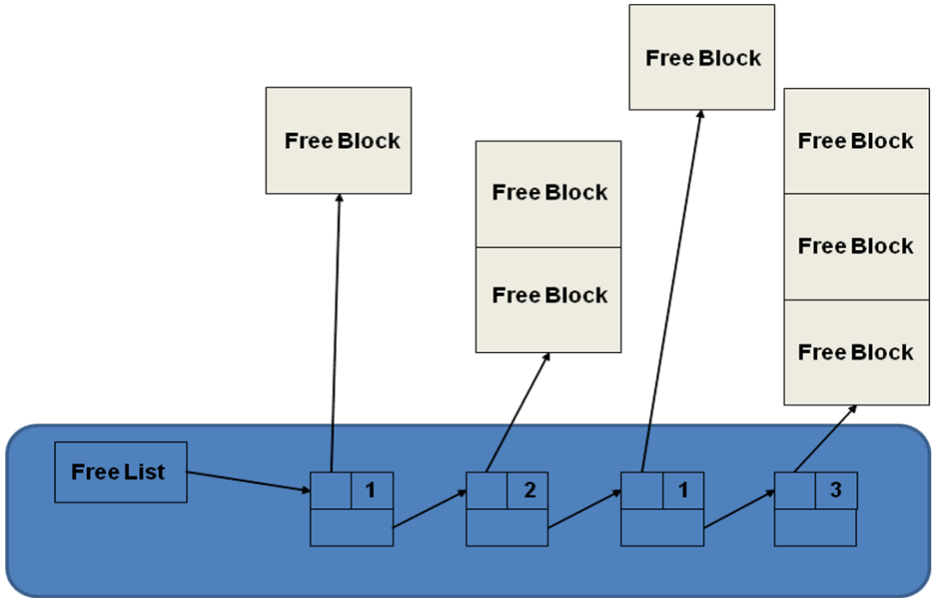
- 2- Use a bitmap. A disk with *n* blocks requires a bitmap with *n* bits.
- Free blocks are represented by 1s in the map, allocated blocks by 0s
  - 1-TB disk, we need 1 billion bits for the map, which requires around 130,000 1-KB blocks to store



Strategies for disk space allocation to files

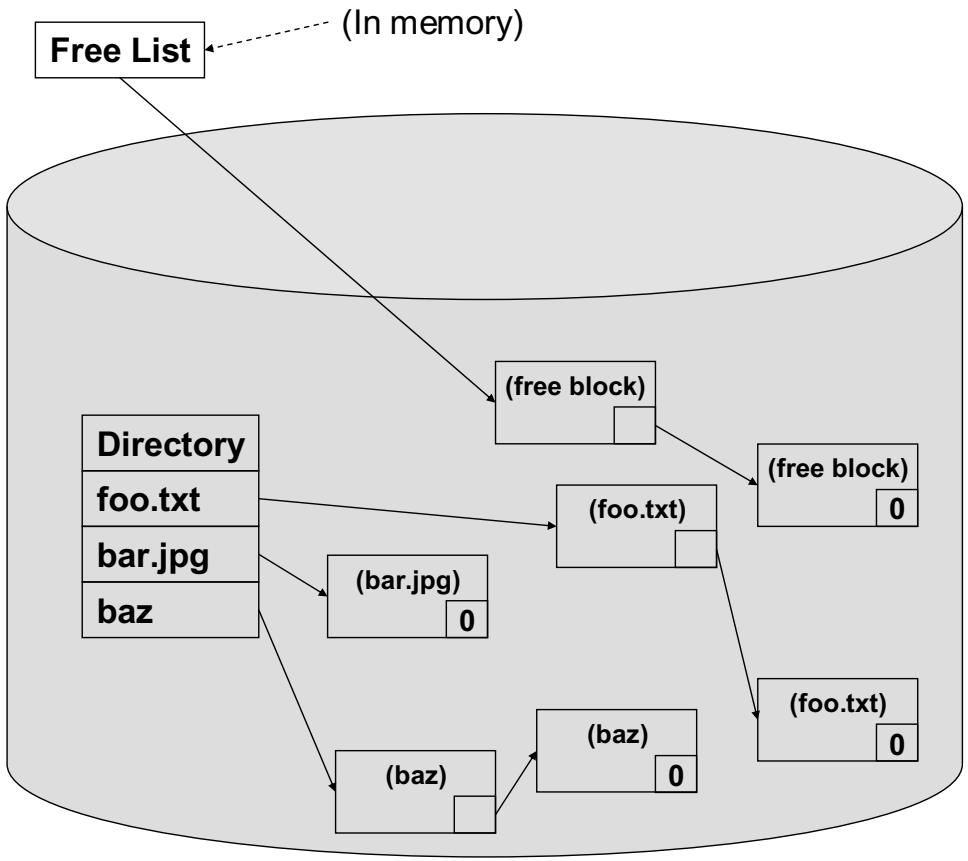


- **Contiguous Allocation**
  - At file creation time, a sequence of blocks is allocated, only remember the address of the first block
  - File cannot grow beyond that size
  - Fragmentation a problem
- Free list
  - Allocation may be by first or best fit
  - Requires periodic compaction



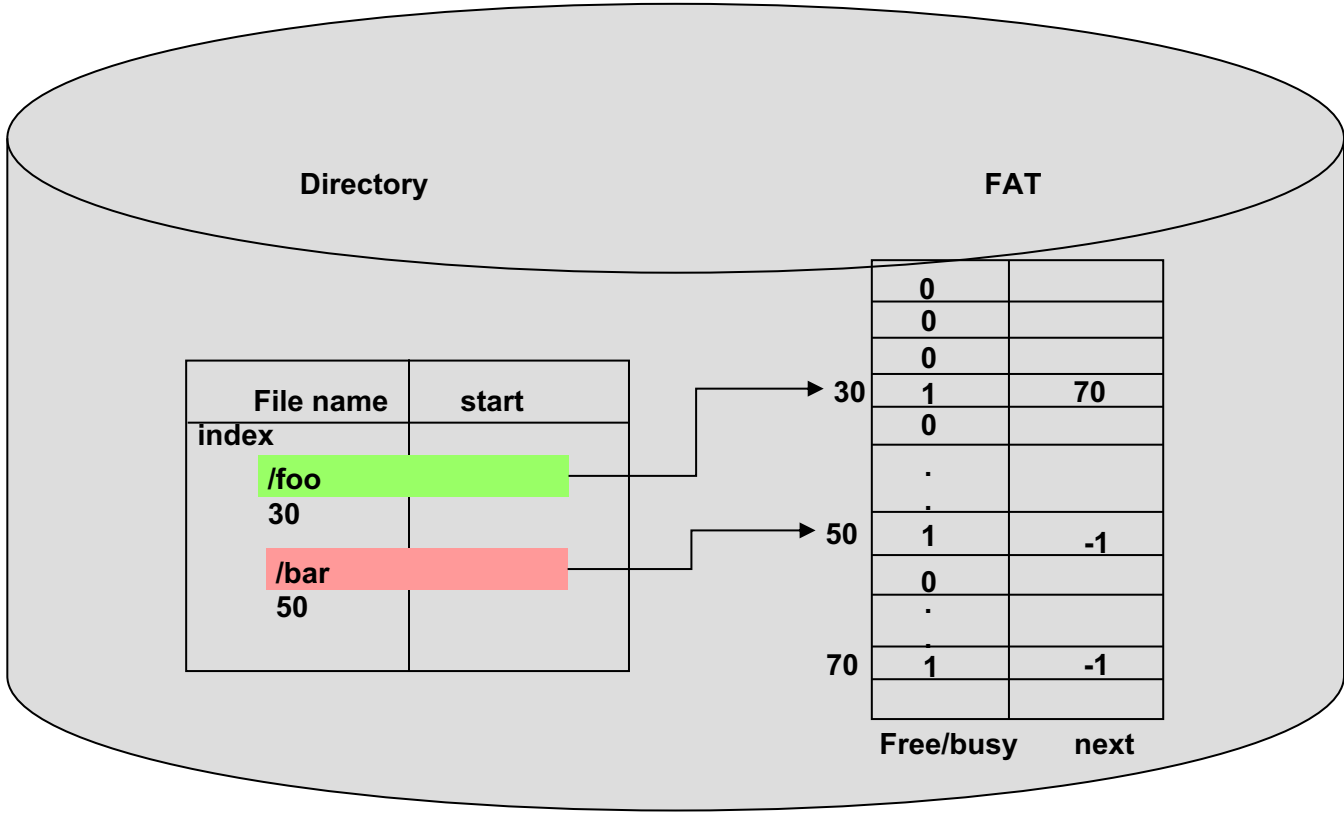
Strategies for disk space allocation to files

- **Linked list Allocation**
  - Directory only need to store the address of the first block
  - No compaction required
  - Sequential access is poor
  - Need to use space in each block to store the address of the next block allocated to the file
  - The list of free blocks can be built in the same way



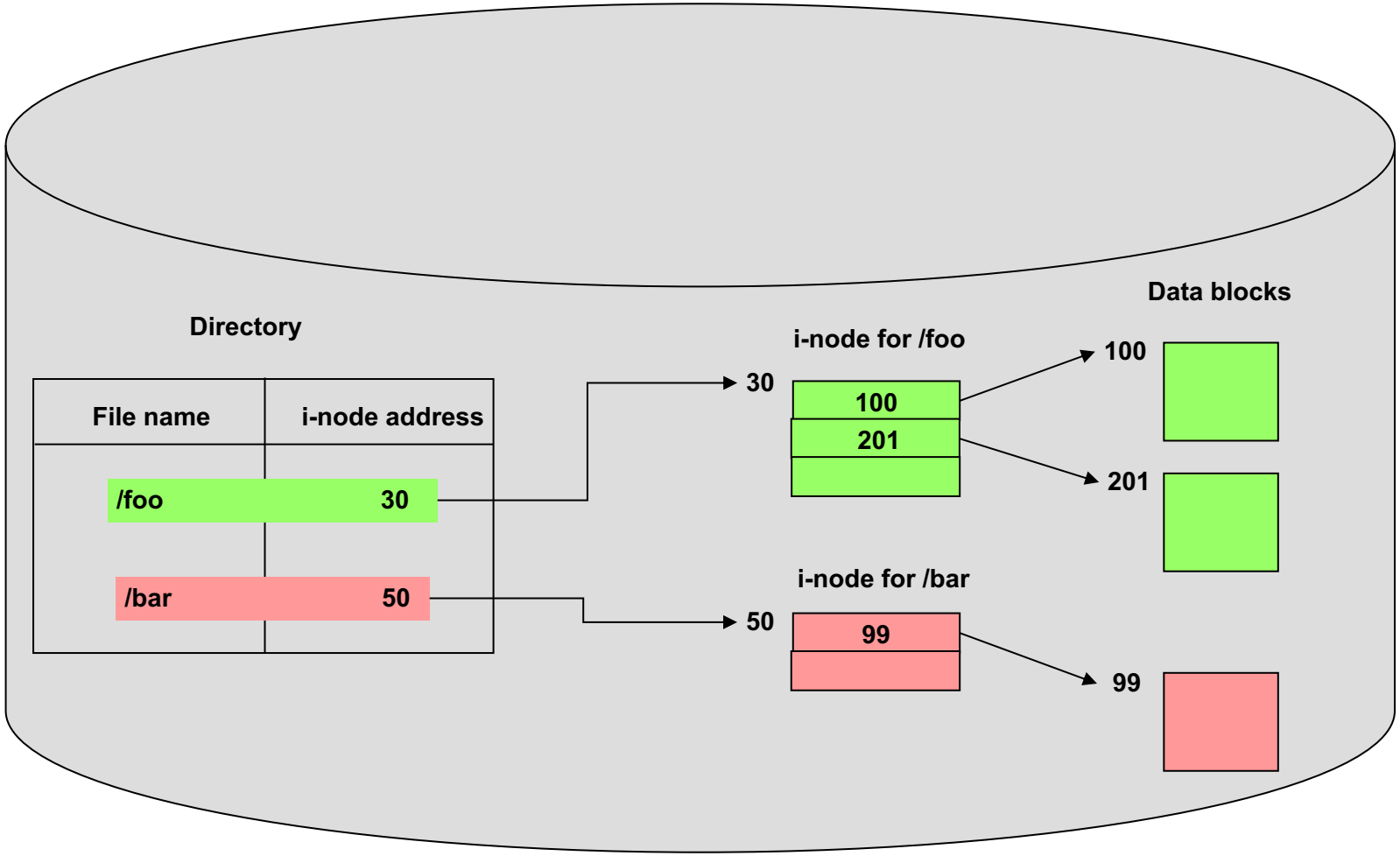
Strategies for disk space allocation to files

- **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



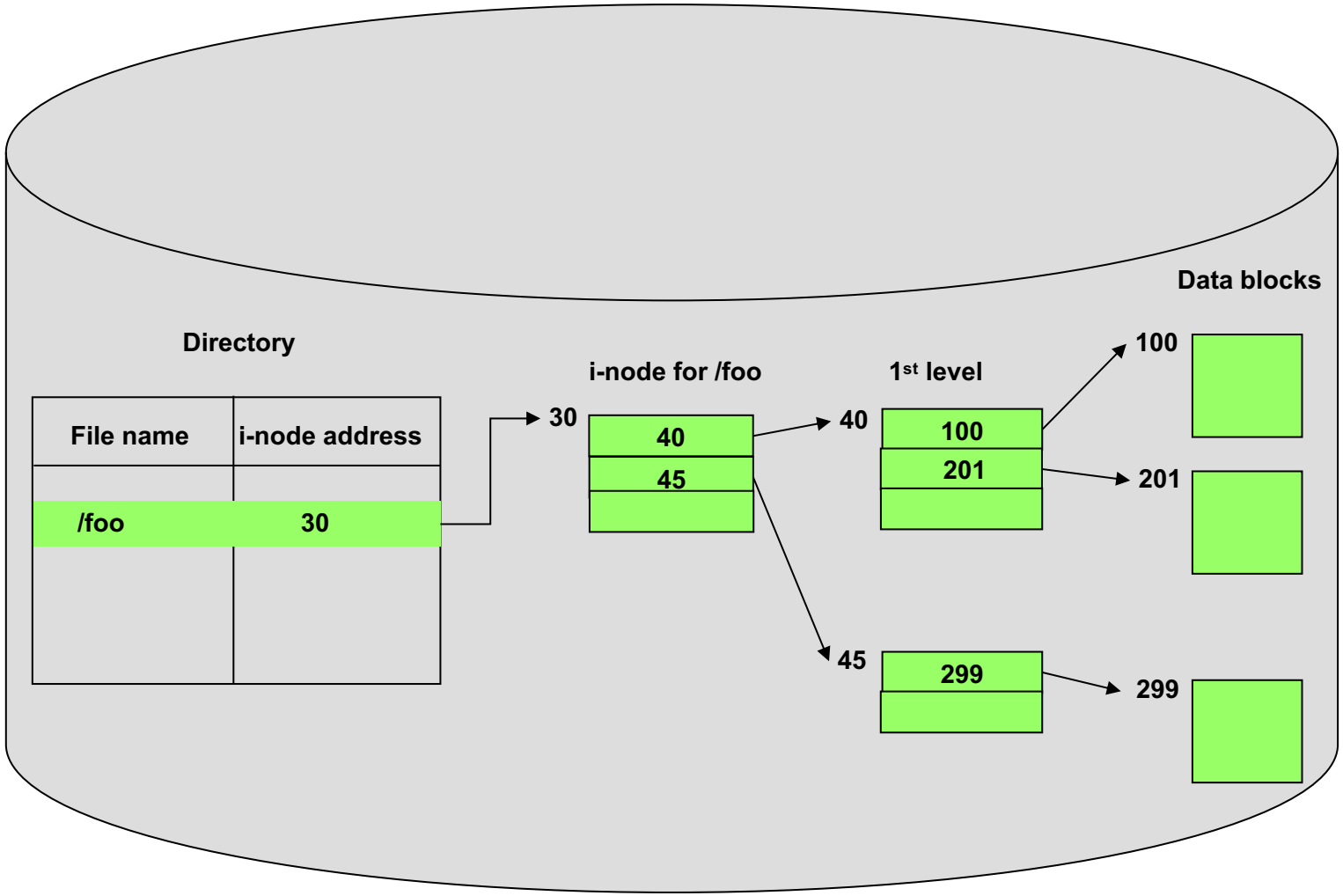
Strategies for disk space allocation to files

- **Indexed Allocation**
  - Breaks up FAT into one data structure per file
  - Allocate an index disk block for each file called an i-node
  - Directory entries now point to the i-node for that file
  - Maintain free list as bit vector
  - Since the i-node is a fixed size there is a maximum file size



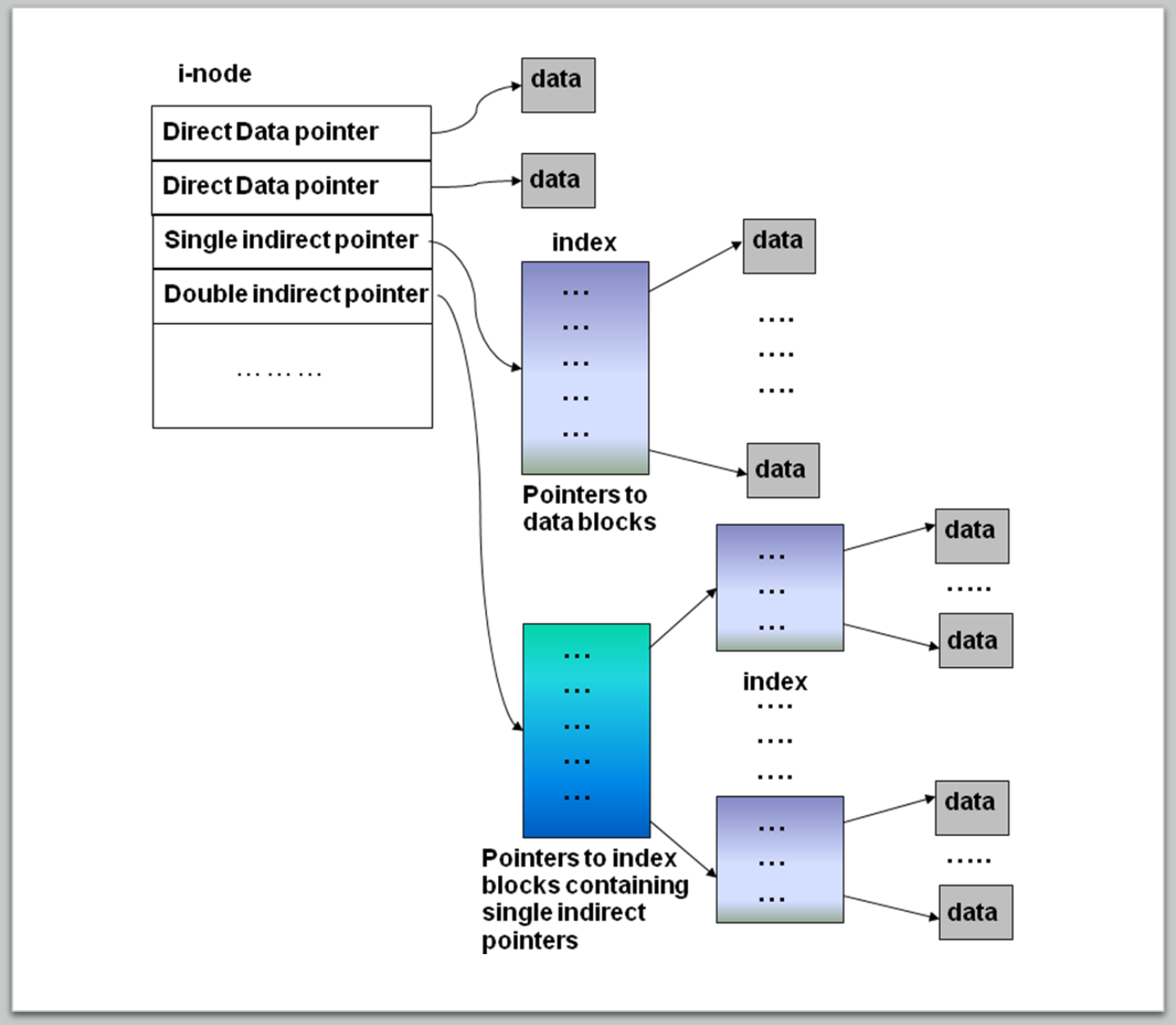
Strategies for disk space allocation to files

- **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



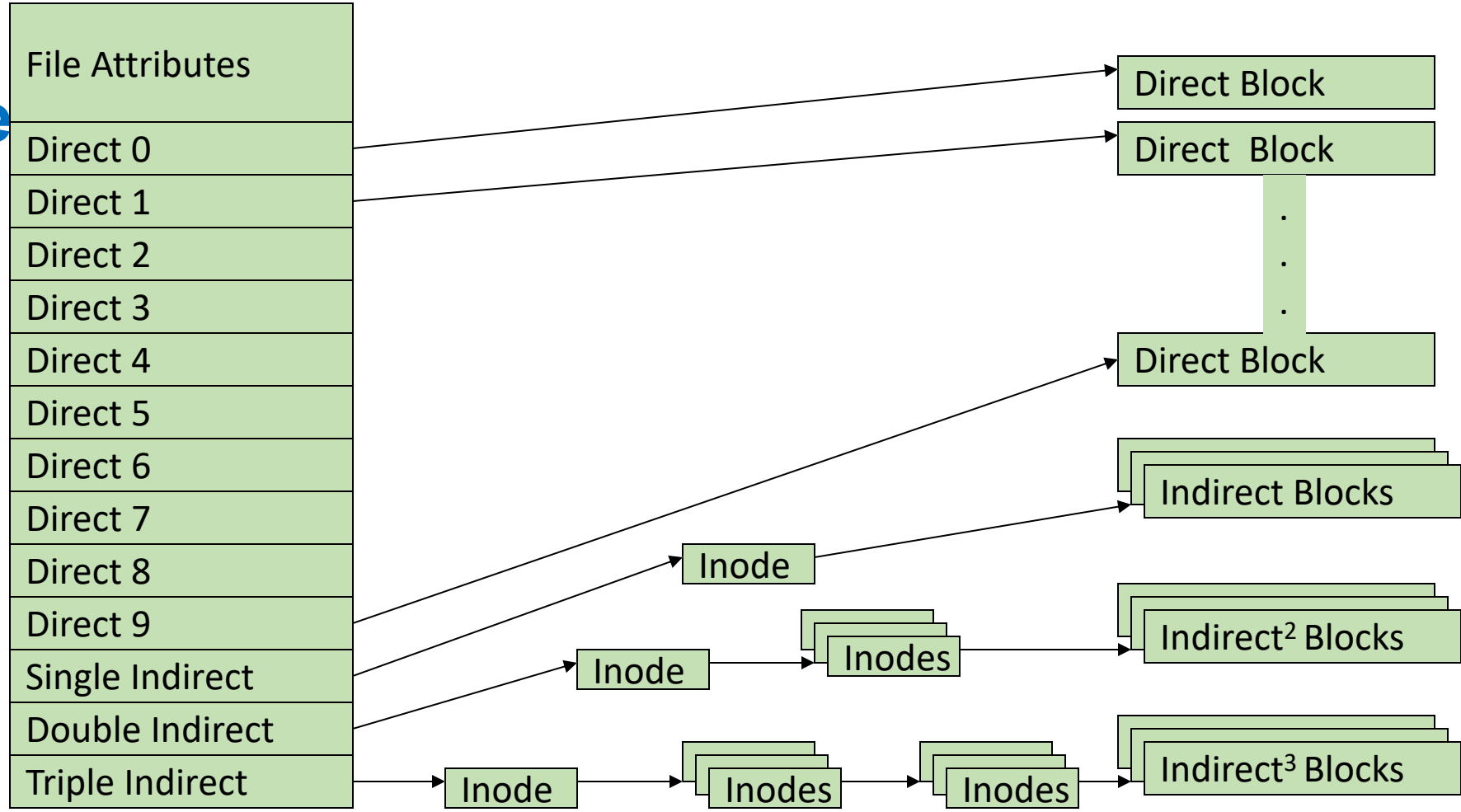
Strategies for disk space allocation to files

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

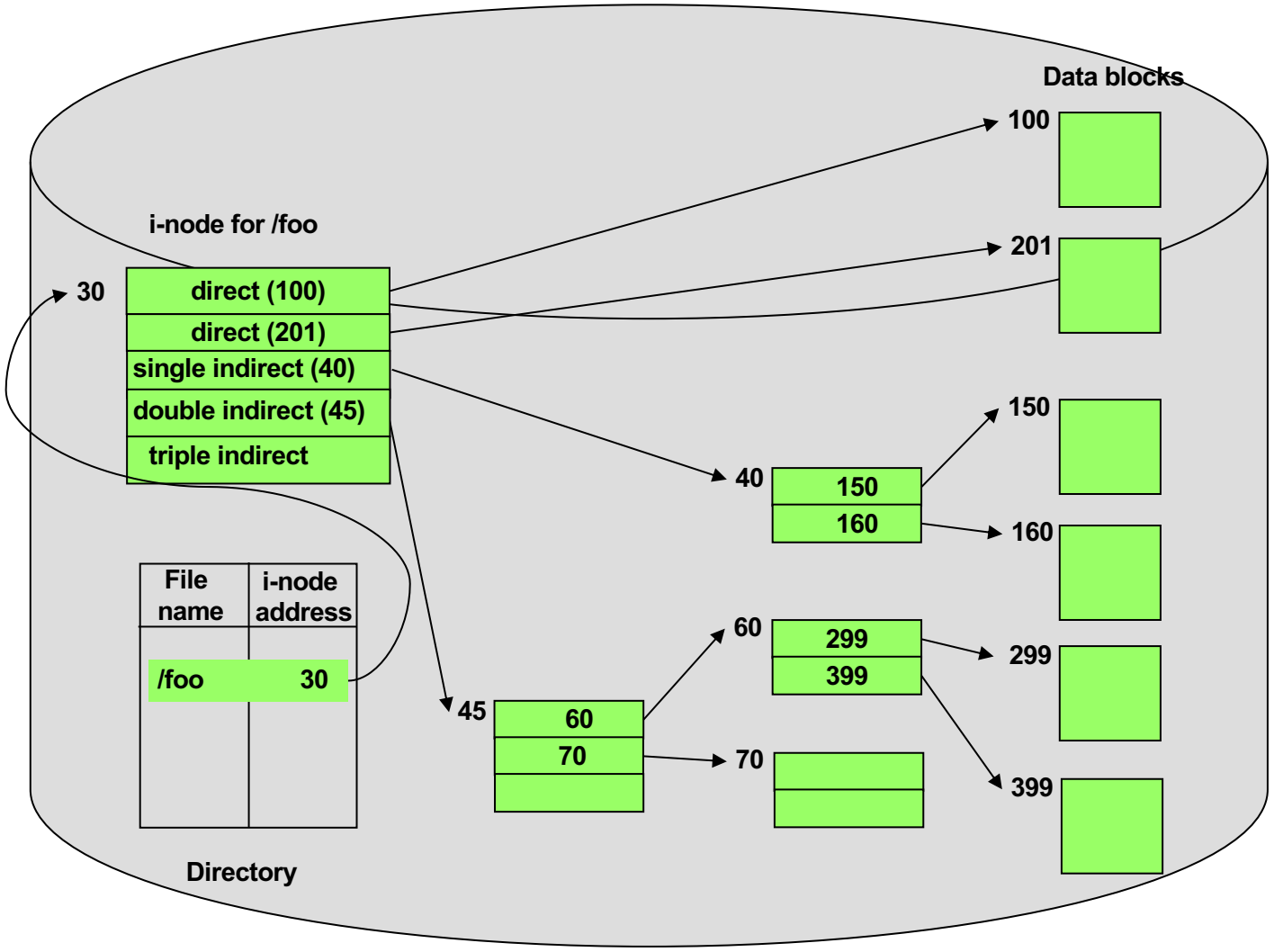


Strategies for disk space allocation to file

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

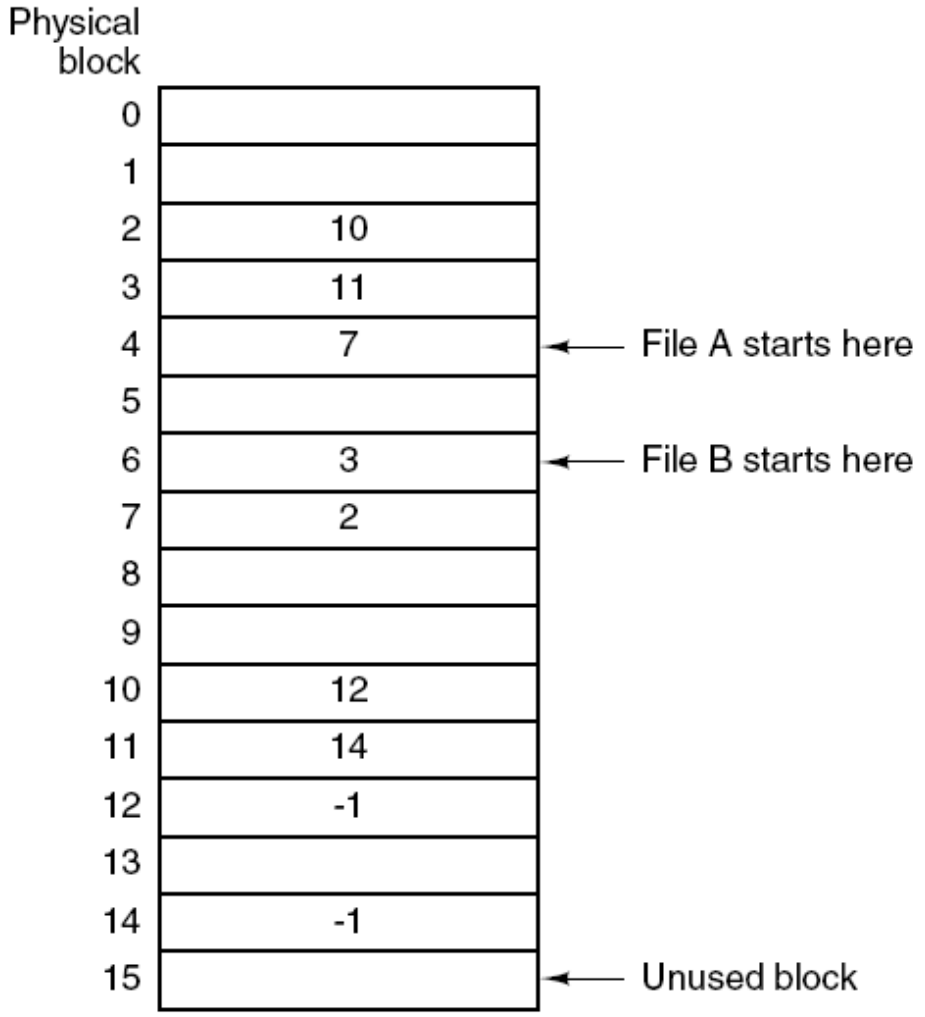


Hybrid indexed allocation for a directory



Windows file systems

- FAT12, 16, 32, NTFS
- FAT stands for file allocation table, which means files are allocated blocks through a table
- Each partition hold 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
  - 2^{32} = 32,768
- FAT-16 uses 128KB of main memory per partition
- The largest possible disk partition was 2GB
- Could address hard drives of 8GB
- 2GB is just enough for 8 minutes of video!
- Starting with Windows 95, FAT-32 file system was introduced

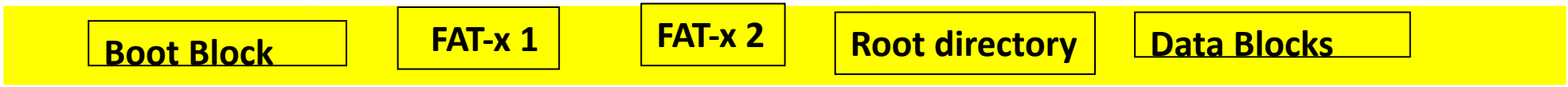
Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

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

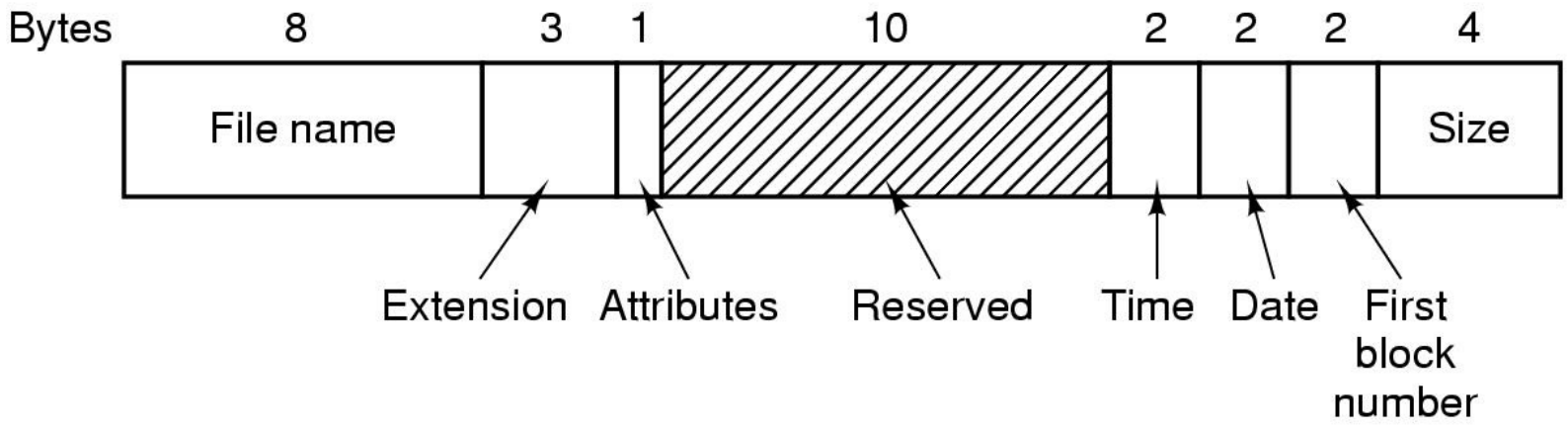
FAT structure on-disk

- 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 * 16 directory entries per sector = 224 entries
- Other directories have variable length, but each entry is 32 bytes long



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

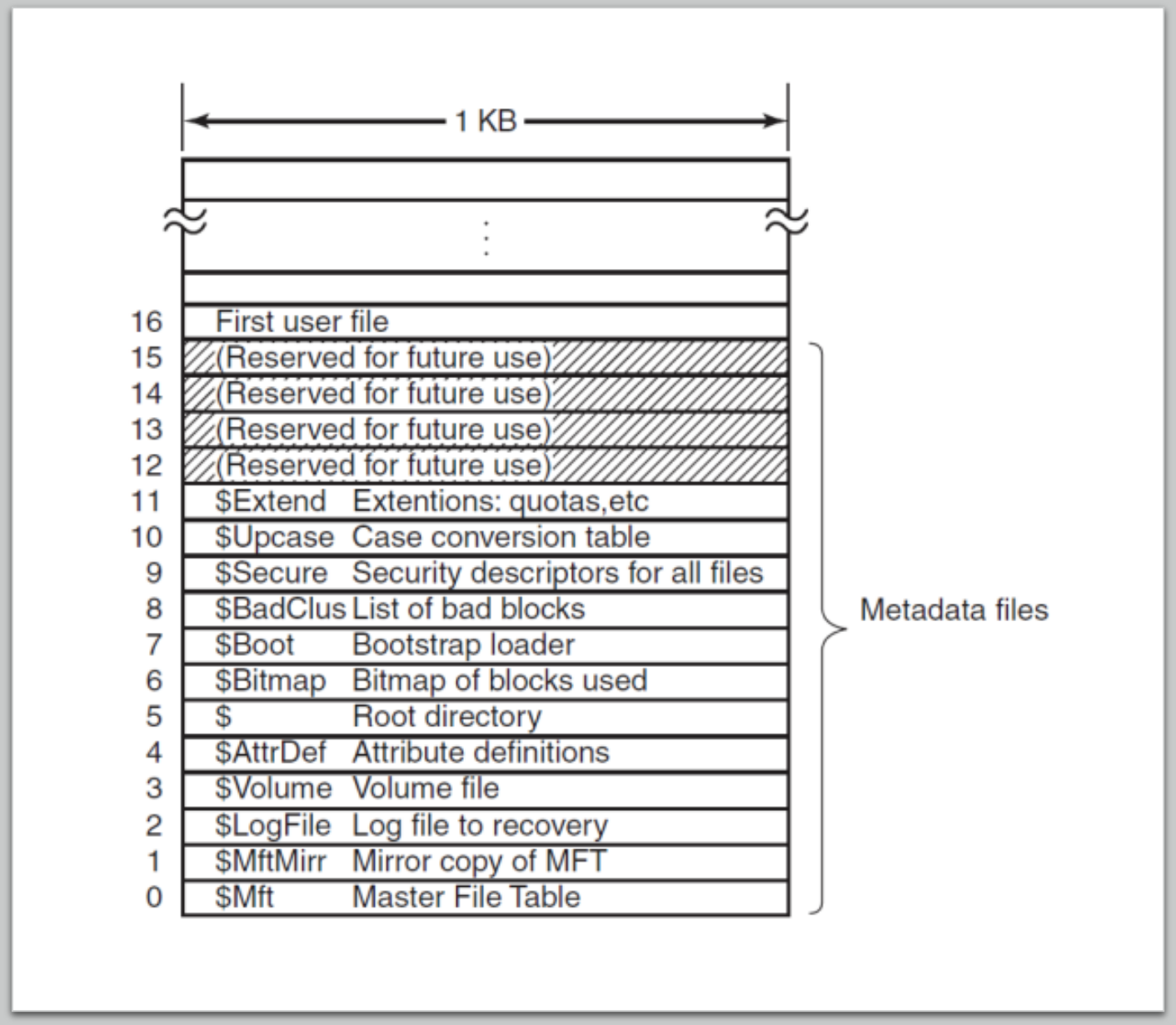


Windows NTFS

- 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

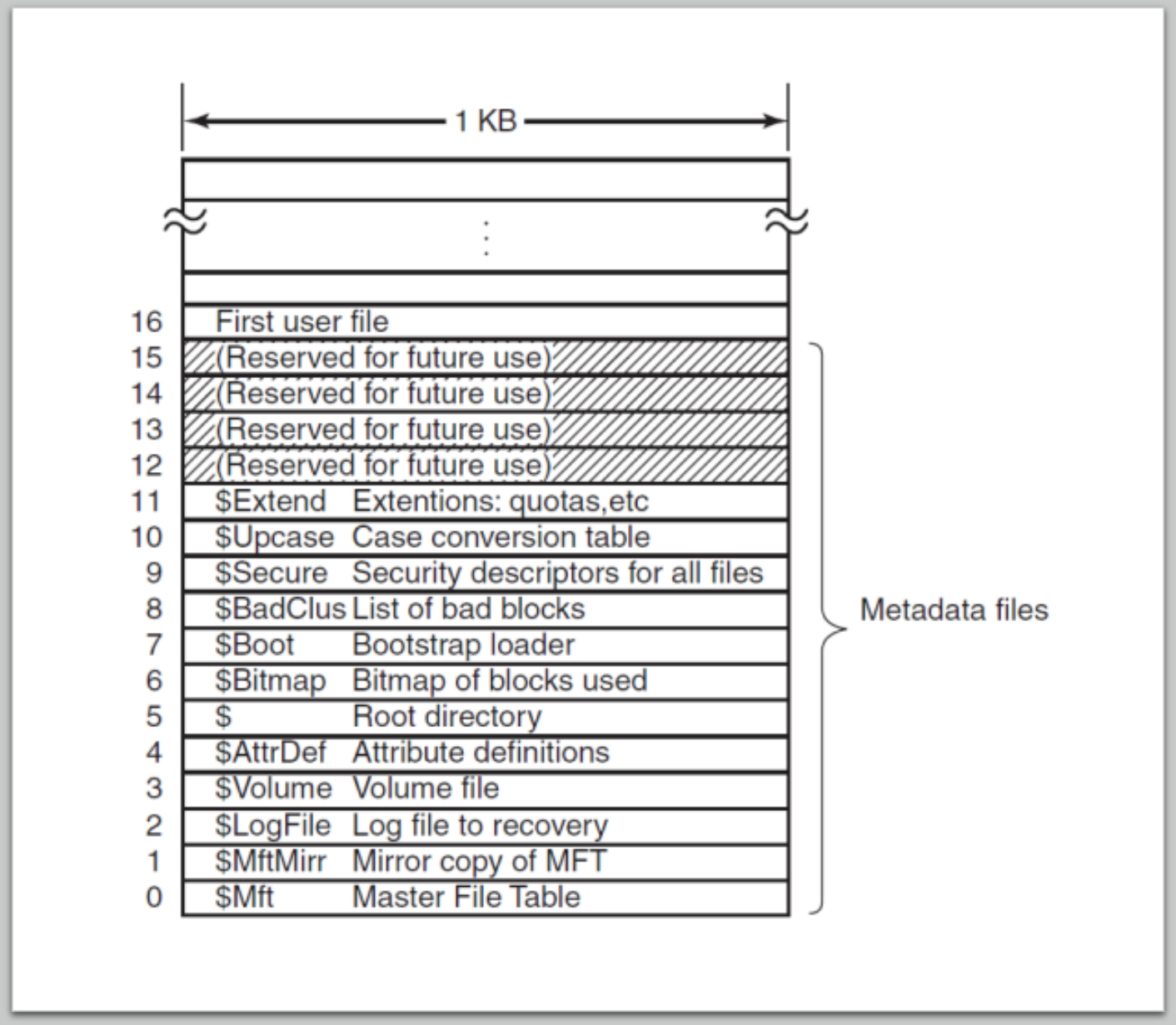
NTFS Master file table

- 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.



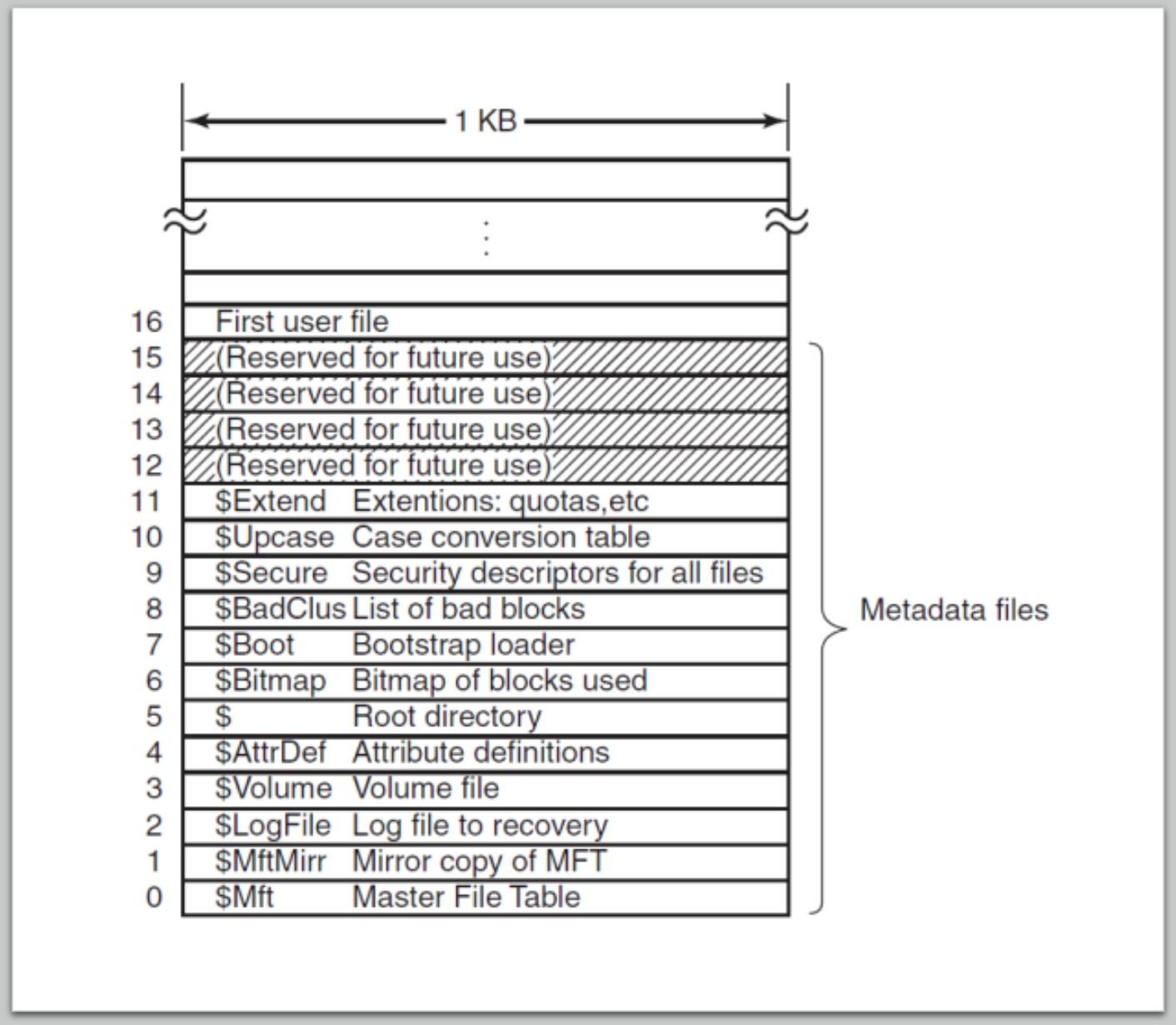
NTFS Master file

- 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



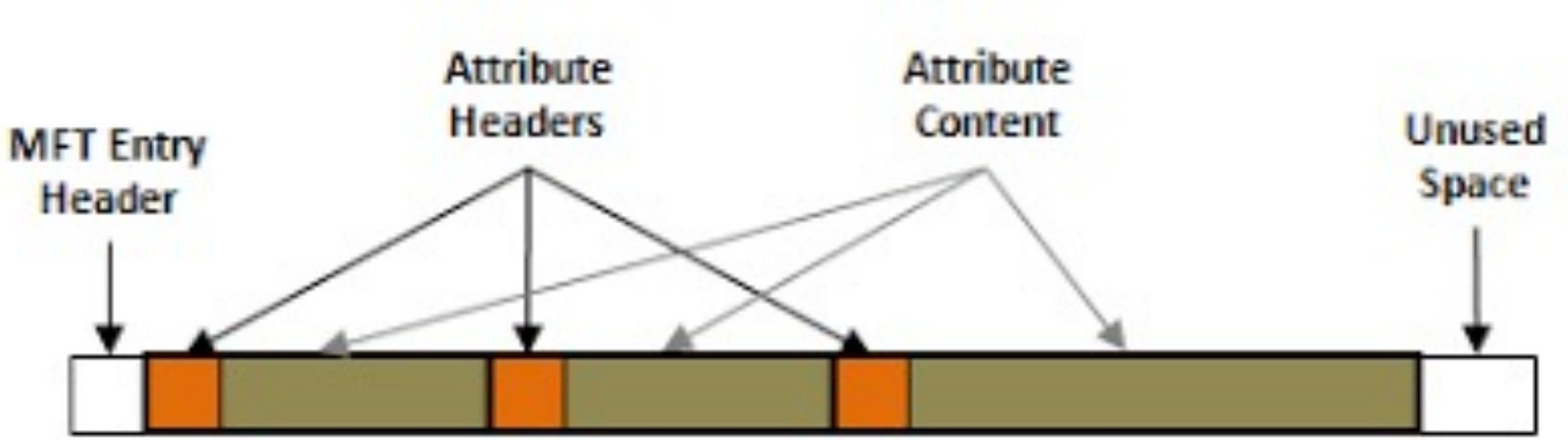
Some metadata Master file entries

- 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.



Structure of master file entries

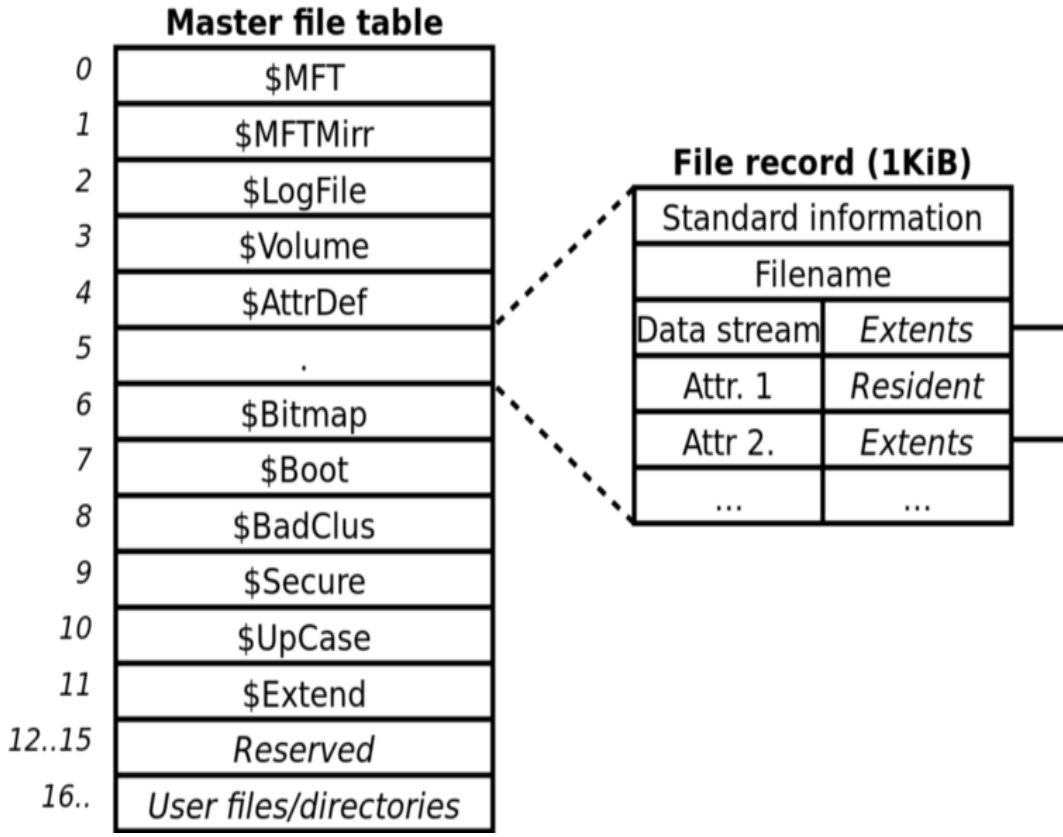
- The first 42 bytes store the header. The header contains 12 fields.
- The other 982 bytes do not have a fixed structure, and are used to keep attributes.
- Each MFT entry consists of a sequence of (attribute header, value) pairs.
- Each attribute begins with a header telling which attribute this is 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

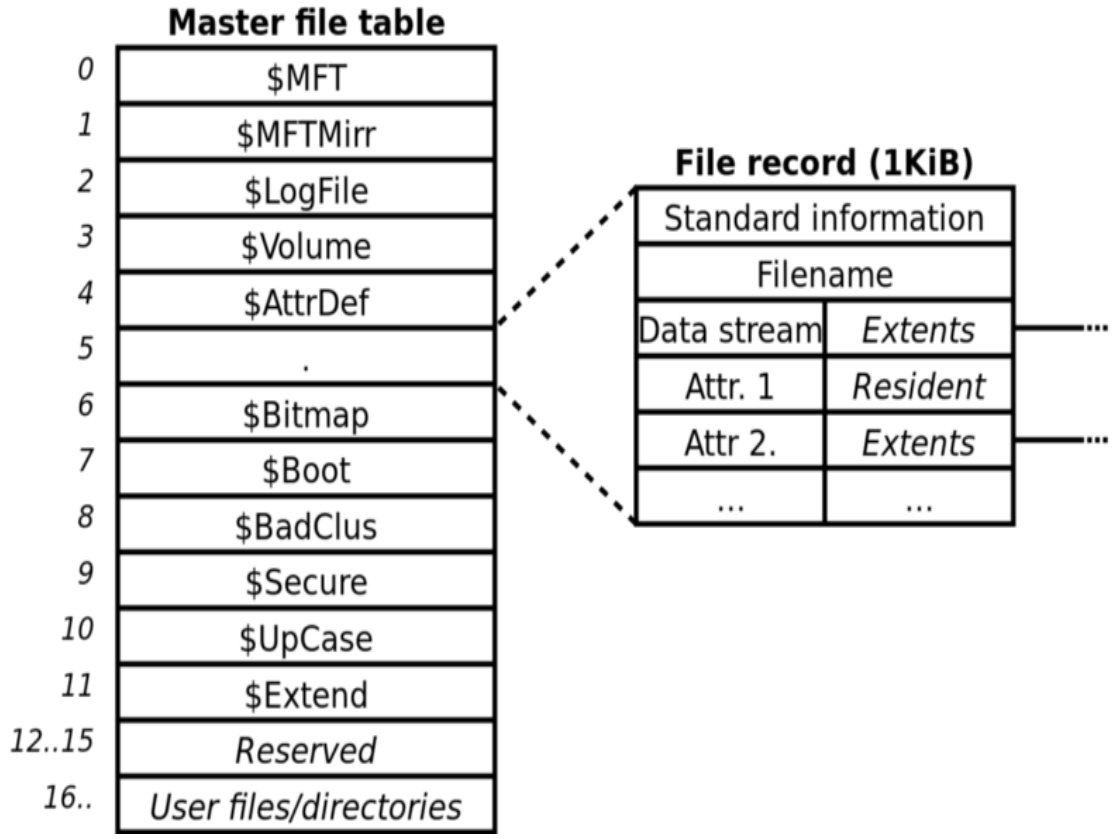
- Usually, attribute values follow their attribute headers directly, but if a value is too long to fit in the MFT entry (like data), it may be put in separate disk blocks.
- Such an attribute is said to be a **nonresident attribute**.

- 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

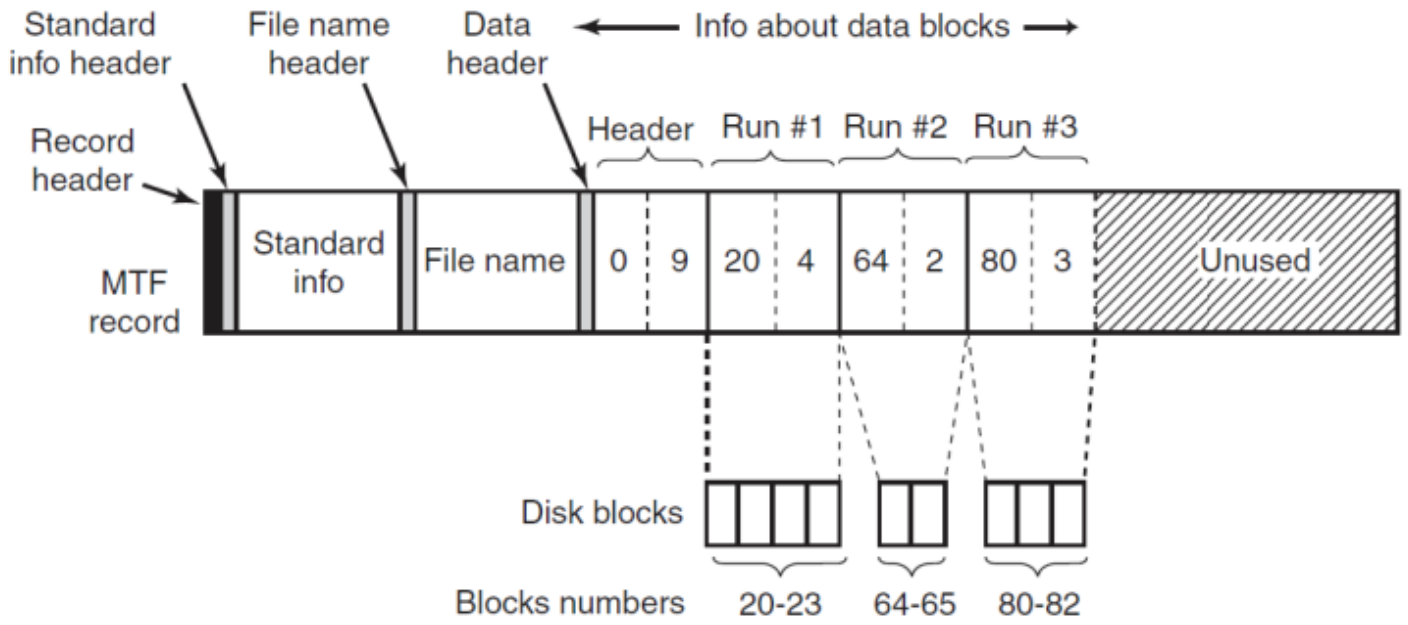
NTFS file organization

- 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



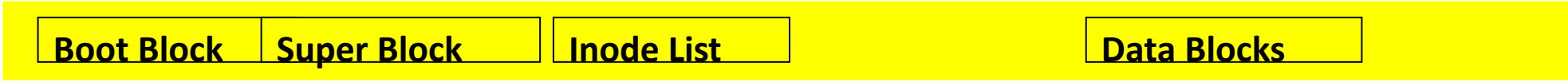
Unix/Linux files systems

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

Linux structure on disk: boot block

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

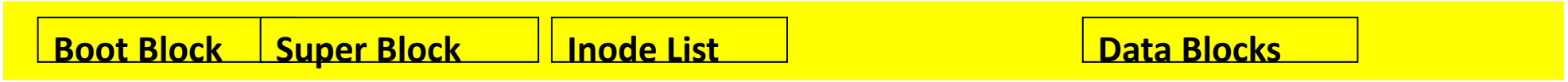
  - Like for Windows file systems, the boot block occupies the beginning of a file system, the first sector
  - May contain the bootstrap code that is read into the machine at boot time
  - Only one boot block is required to boot the system, but every file system may contain a boot block



Super block

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

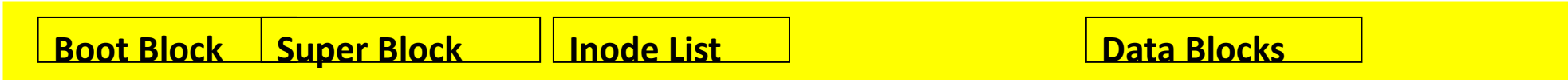
  - The super block describes the file system:
    - Number of bytes in the file system
    - Number of files it can store
    - Where to find free space in the file system
    - Additional data to mange the file system



i-nodes

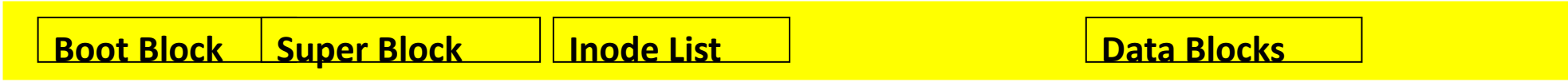
- Boot Block
- Super Block
- Inode List
- Data Block

  - 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



System Calls related to files

- Linux system calls relate to files and the file system

System call	Description
fd = creat(name, mode)	One way to create a new file
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information
s = fstat(fd, &buf)	Get a file's status information
s = pipe(&fd[0])	Create a pipe
s = fcntl(fd, cmd, ...)	File locking and other operations

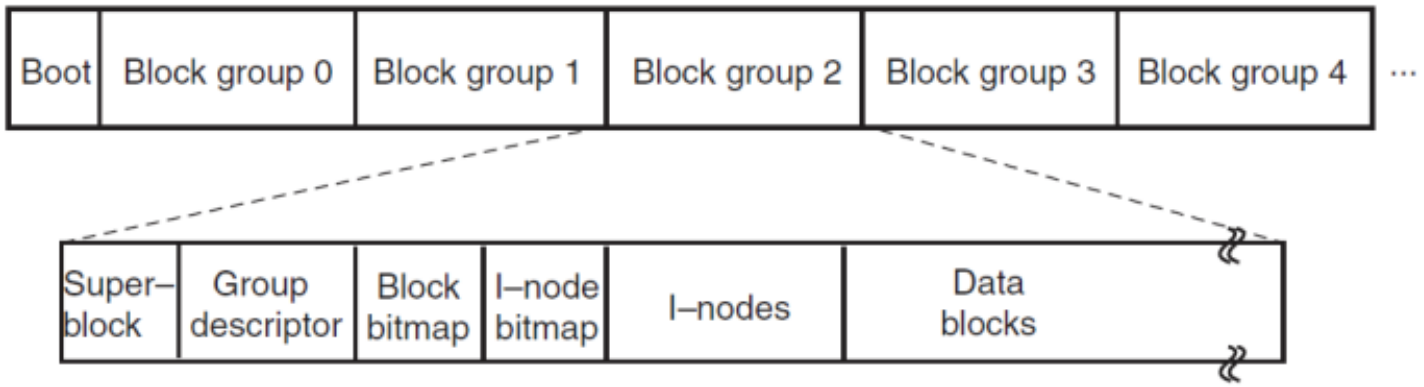
System Calls related to directories

- Link system call: linking to a file creates a new directory entry that points to an existing file
- Directory entries are removed with unlink.
- When the last link to a file is removed, the file is automatically deleted. For a file that has never been linked, the first unlink causes it to disappear

System call	Description
s = mkdir(path, mode)	Create a new directory
s = rmdir(path)	Remove a directory
s = link(oldpath, newpath)	Create a link to an existing file
s = unlink(path)	Unlink a file
s = chdir(path)	Change the working directory
dir = opendir(path)	Open a directory for reading
s = closedir(dir)	Close a directory
dirent = readdir(dir)	Read one directory entry
rewinddir(dir)	Rewind a directory so it can be reread

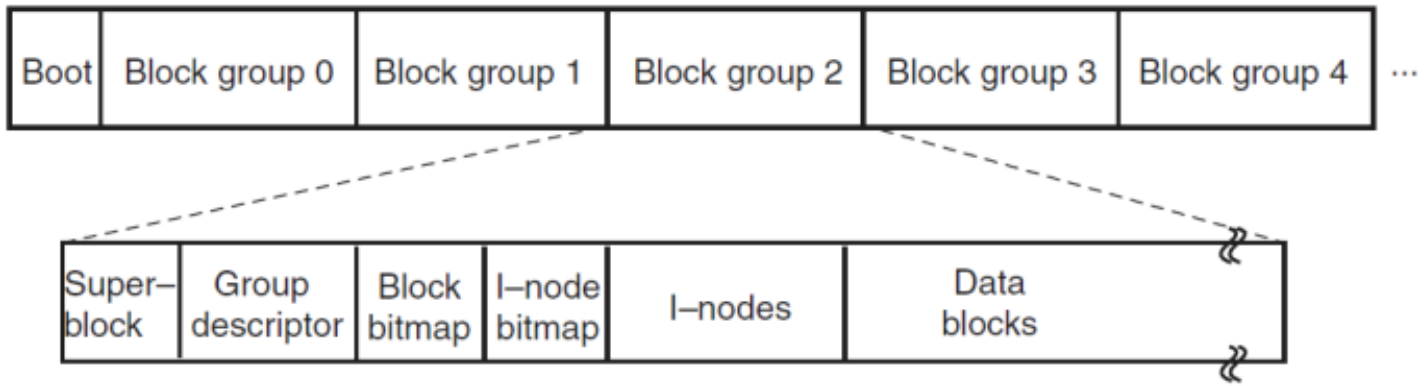
The Linux ext2 file system

- 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



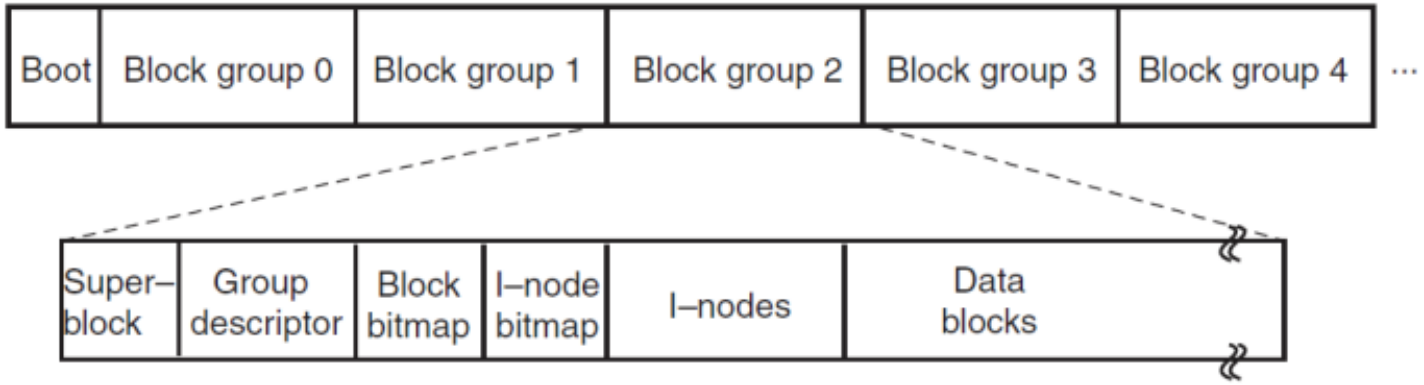
The Linux ext2 file system

- 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

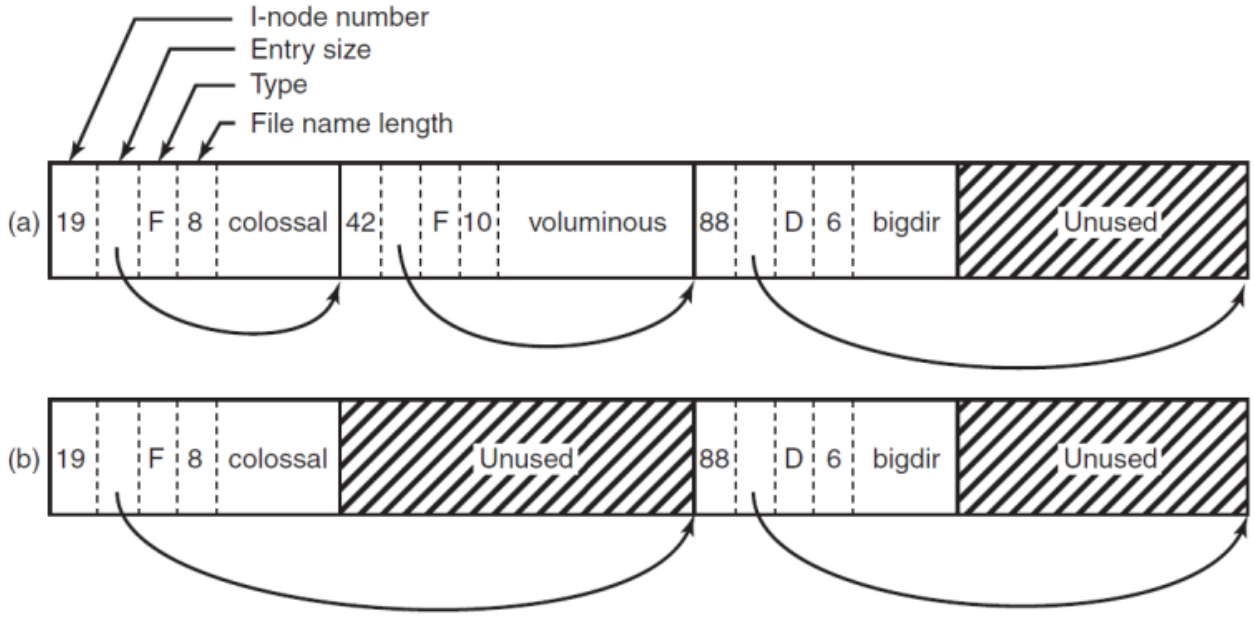


Figure 10-32. (a) A Linux directory with three files. (b) The same directory after the file *voluminous* has been removed.

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)

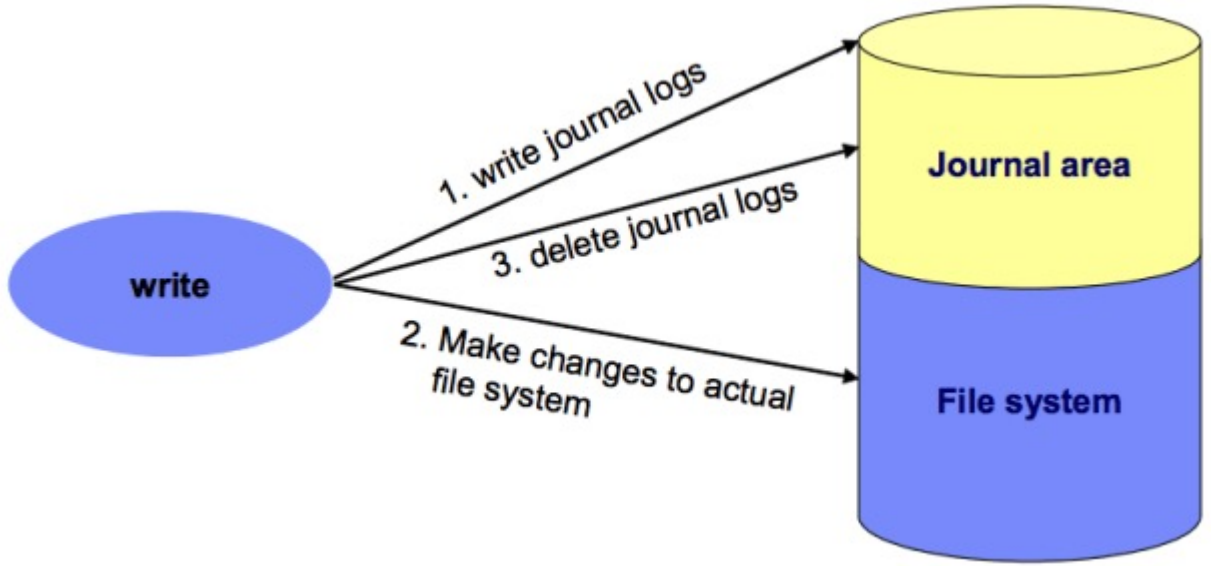
Structure of i-nodes

- 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.

ext3 and ext4

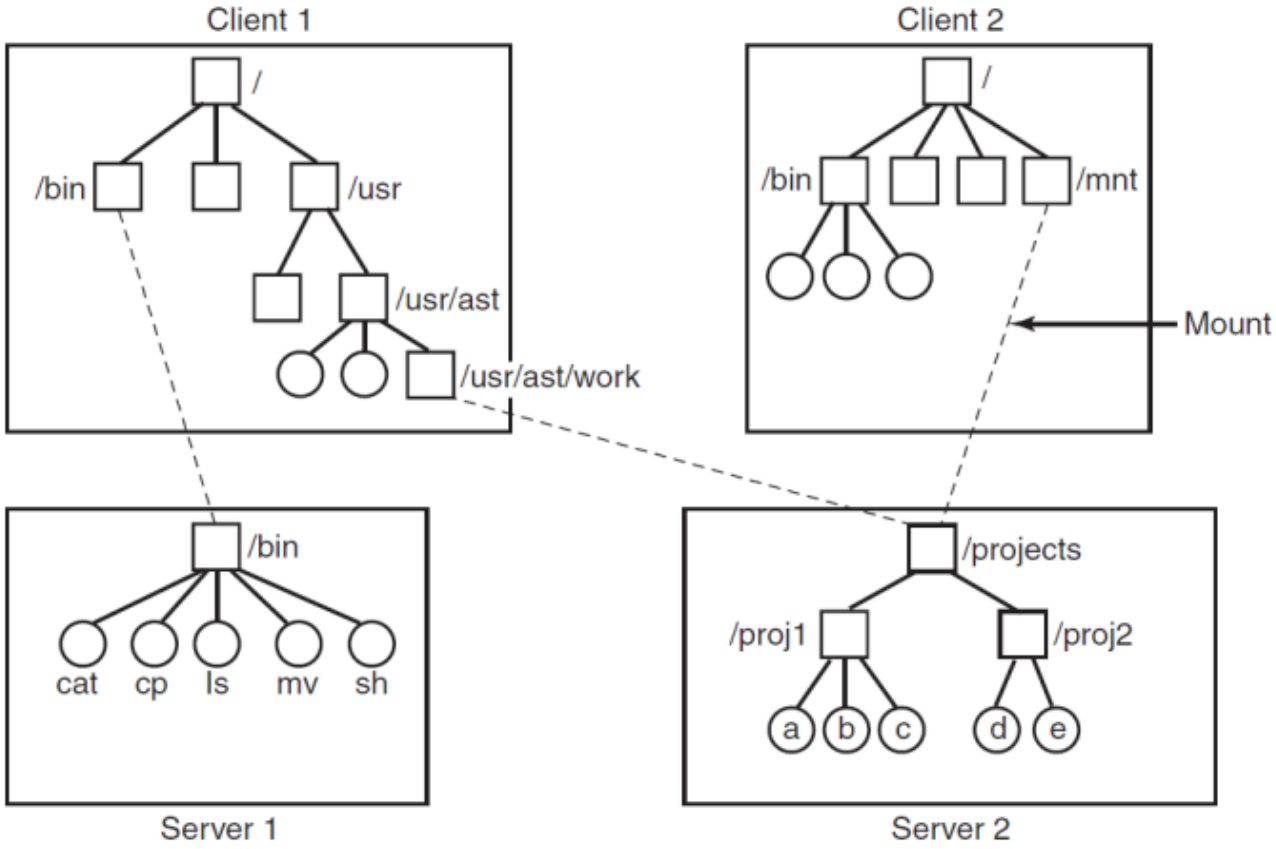
- Like Windows NTFS, ext3 and ext4 are “journaling file systems”
- For example, deleting a file on a Unix file system involves three steps:
  - 1- Removing its directory entry.
  - 2- Releasing the inode to the pool of free inodes.
  - 3- Returning all disk blocks to the pool of free disk blocks.
- If a crash occurs after step 1 and before step 2, there will be an orphaned inode; if a crash occurs between steps 2 and 3, then the blocks previously used by the file cannot be used for new files

- A journaled file system allocates a special area—the journal—in which it records the changes it will make ahead of time. After a crash, recovery simply involves reading the journal from the file system and replaying changes from this journal until the file system is consistent again.



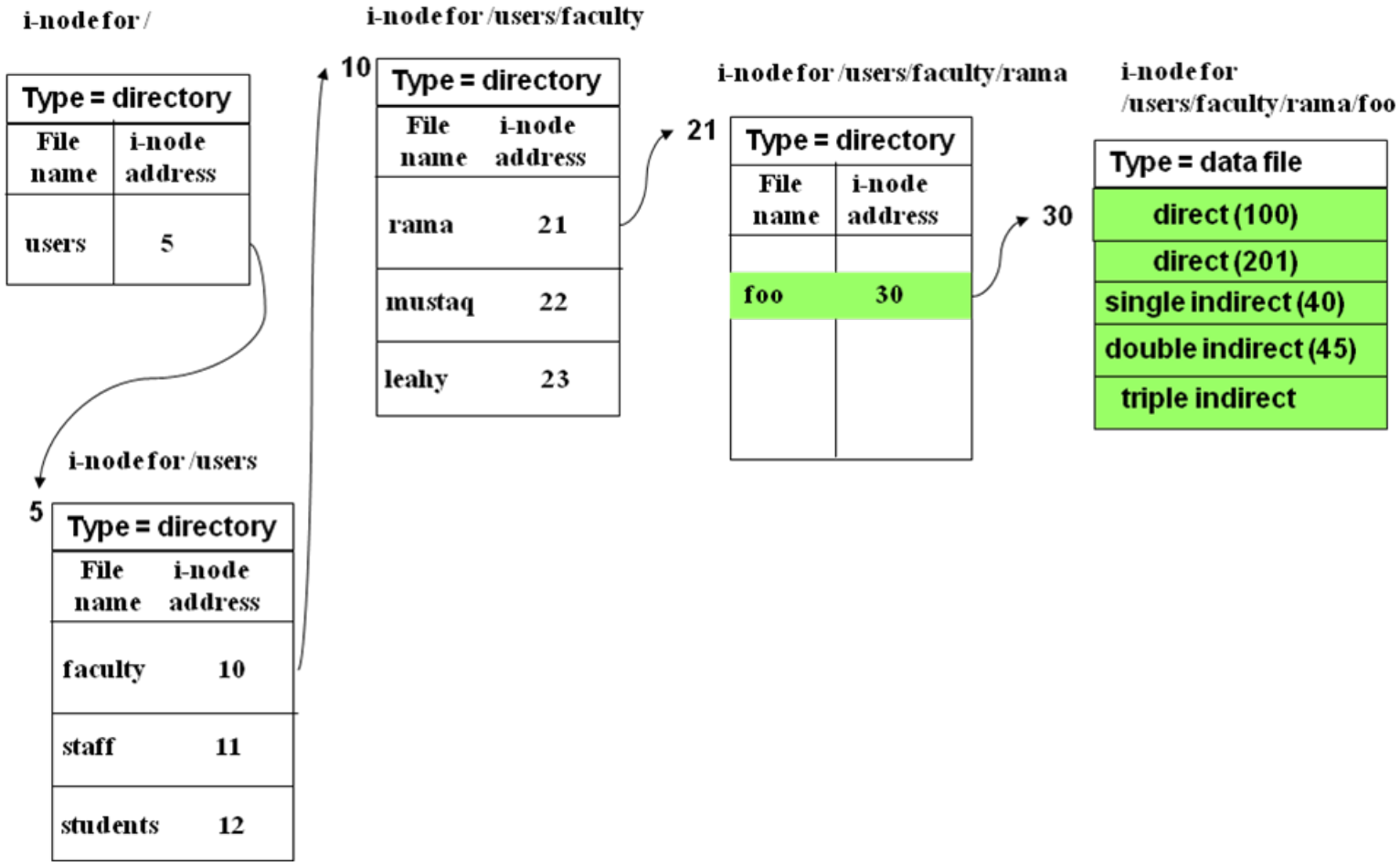
NFS: The Network File System

- Is used to join the file systems on separate computers into one logical file system.
- The basic idea behind NFS is to allow an arbitrary collection of clients and servers to share a common file system

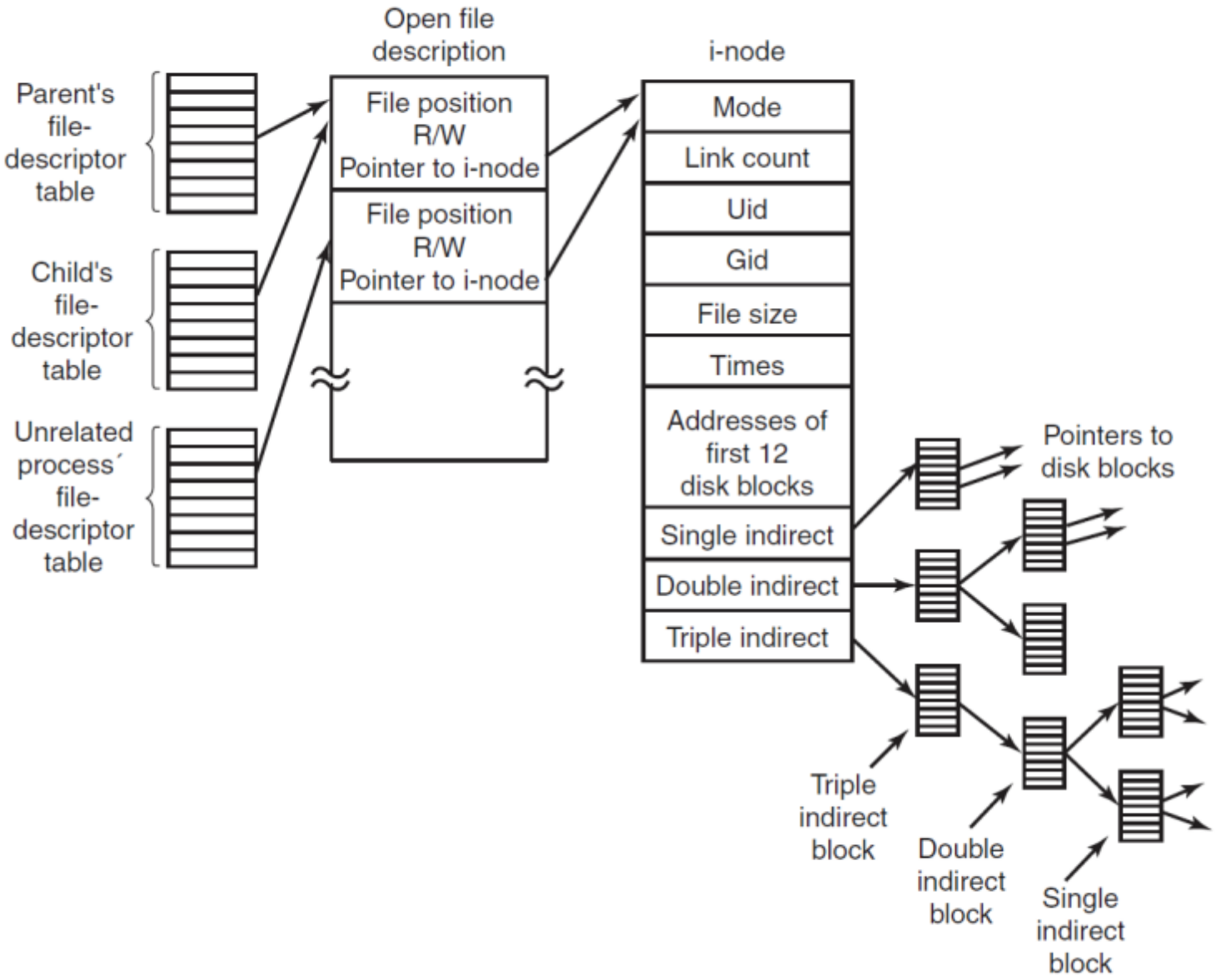


Unix/Linux hierarchical file naming

- 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

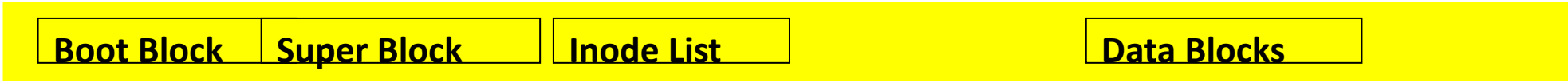


Relation between file descriptors and i-nodes: Linux



Summary: On disk organization of file systems

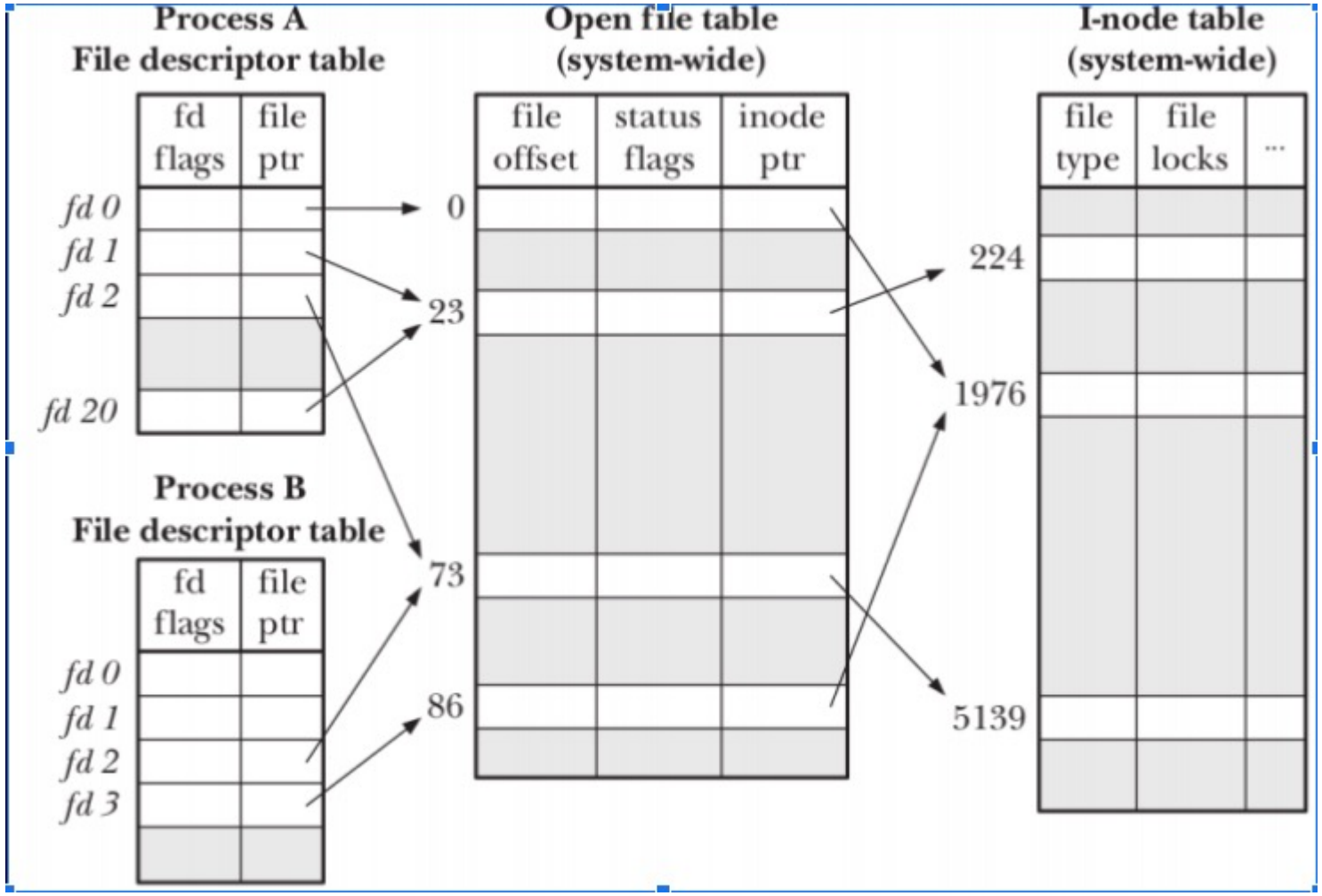
- Boot block: MS-DOS and Unix/Linux: Boot block; NTFS: Boot sector
- Control block: partition details such as number of blocks in partition, size of the blocks, number of free-blocks, pointers to free-blocks: NTFS: Master file table; Linux: Superblock
- A mapping of files to blocks: NTFS: Master file table; Linux: i-nodes; FAT-x: File allocation table



Summary: in-memory file system data structures

- Main memory keeps information about the file system for both file-system management and performance improvement via caching
  - An in-memory **mount table** contains information about each file system mounts, mount points, file system types, etc
  - The **system-wide open-file table** contains a pointer to the i-node of each open file, as well as other information
  - The **per-process open-file** table (Linux file descriptor, fd) (Windows file handle) which contains pointers to the appropriate entries in the system-wide open-file table, as well as other information, for all files the process has opened
  - **v-nodes table**, which has one entry per i-node of open files, it contains the information stored in the i-nodes copied from the disk as well as some other information

Some of the in-memory file system data structures



General file systems

- A general-purpose computer system can have multiple storage devices:
  - HDD (or SSD), DVD drive, floppy, USB drive, network servers, etc
- Those devices can be sliced up into partitions, which in turn hold file systems
- Computer systems may also have varying numbers of file systems, and the file systems may be of varying types

Different Solaris file systems

- **tmpfs**—a "temporary" file system that is created in volatile main memory and has its contents erased if the system reboots or crashes
- **procfs**—a virtual file system that presents information on all processes as a file system
- **ufs, zfs**—general-purpose file systems

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	proc
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fd
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs

Figure 15.2 Solaris file systems.

General file systems

- The question arises of how to handle these different file systems.

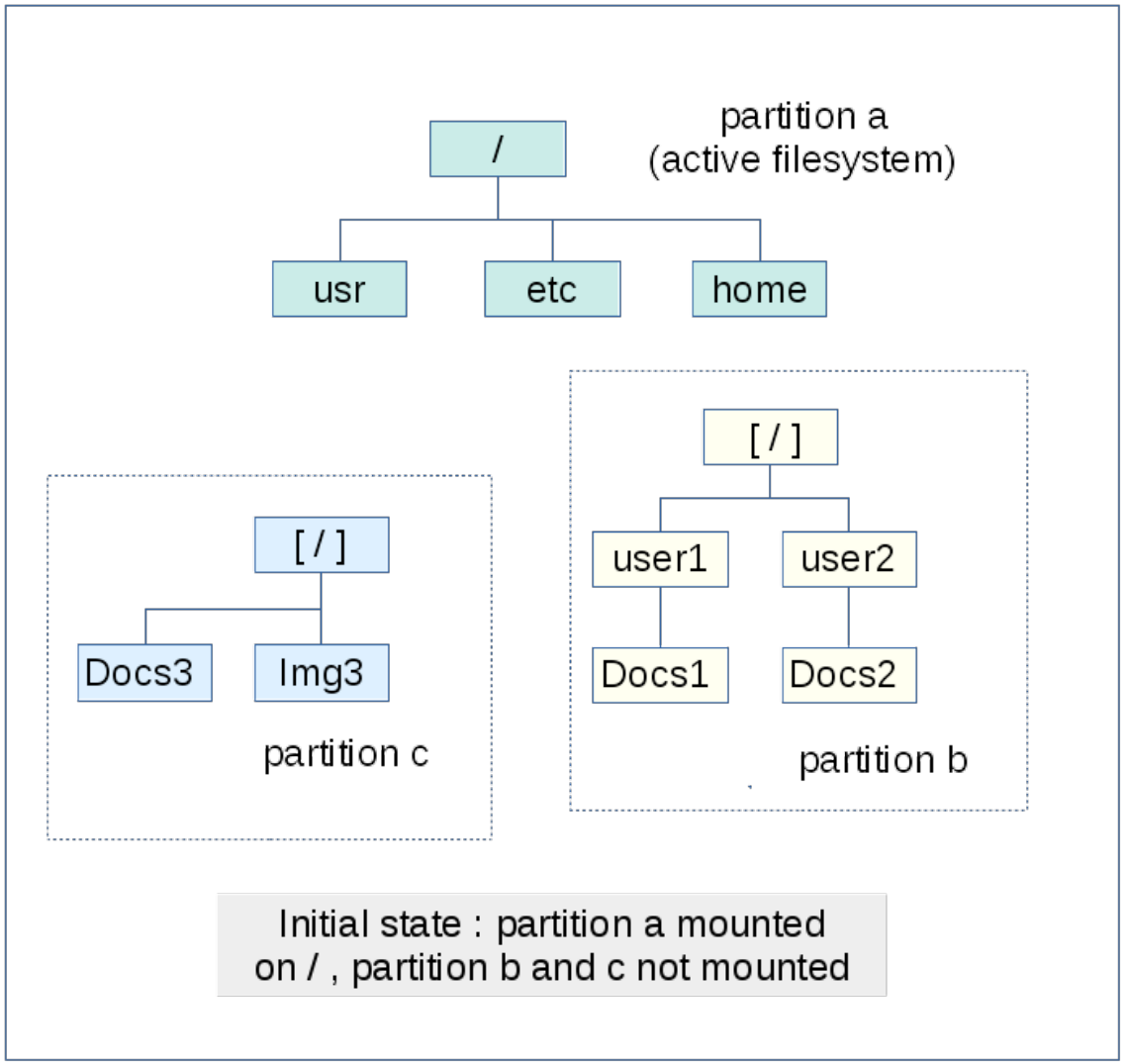
- One solution is to put a self-contained file system on each device/partition and just keep them separate.
  - This is what Windows does, by identifying each device/partition with a different drive letter, as in *C:*, *D:*, etc.
  - When a process opens a file, the drive letter is explicitly or implicitly present, so Windows knows which file system to pass the request to.
- The Unix/Linux solution is to allow one device/partition to be mounted in another device/partition's file tree
- The user sees a single file tree, and doesn't have to be aware of which file resides on which device

File system mounting

- Whether it is Unix/Linux or Windows OS system, a file system must be mounted before it can be available to processes of the OS
- The mount procedure is straightforward.
- The operating system is given the name of the device and the **mount point**, i.e. the location within the file structure where the file system is to be attached
- Typically, a mount point is an empty directory.

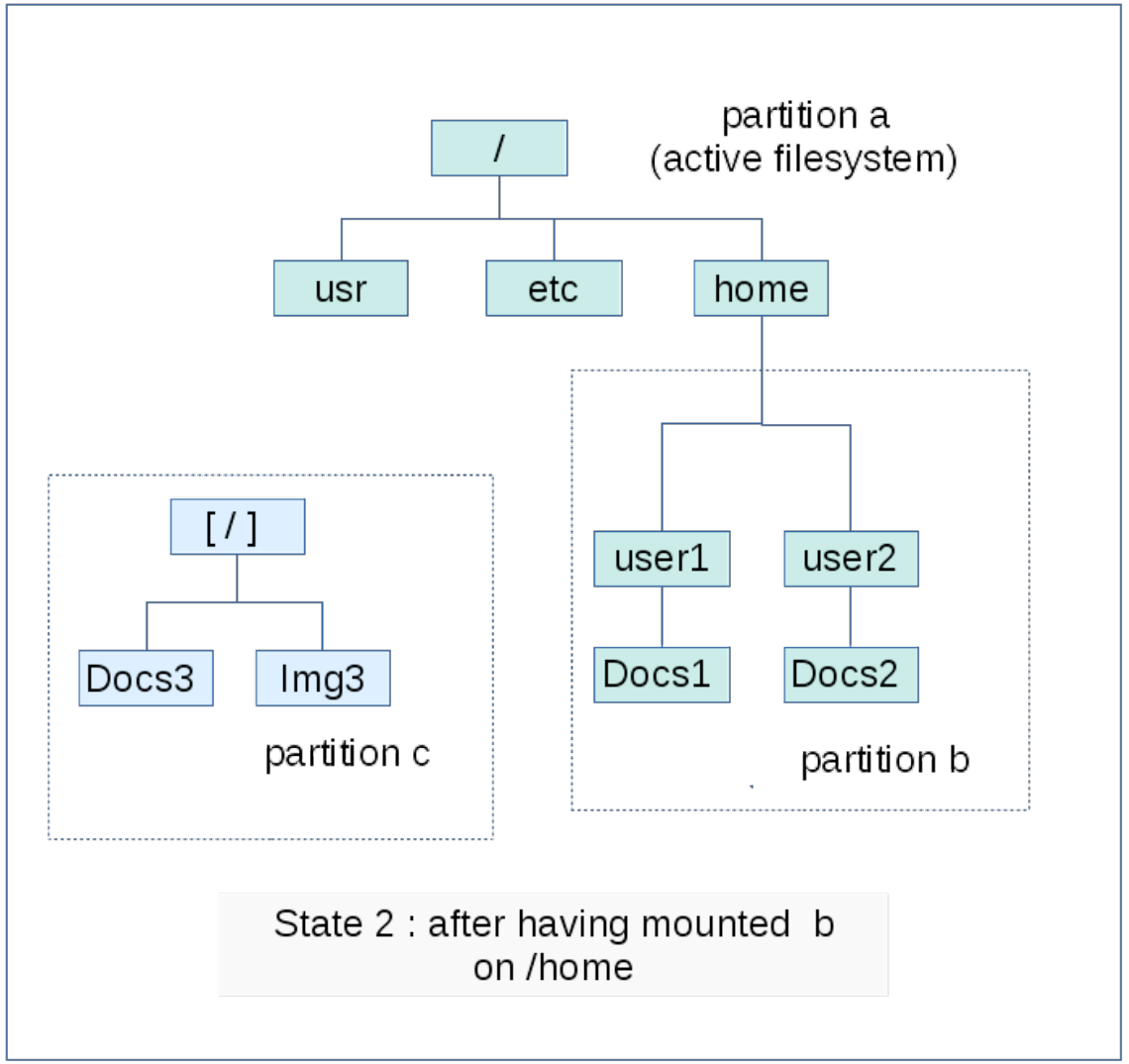
Mounting a file system

- Assume we want to mount the file system on one of the disk partitions.
- First a mount point must be selected, i.e. a directory in the filesystem.
- The side picture describe the file systems on 3 partitions



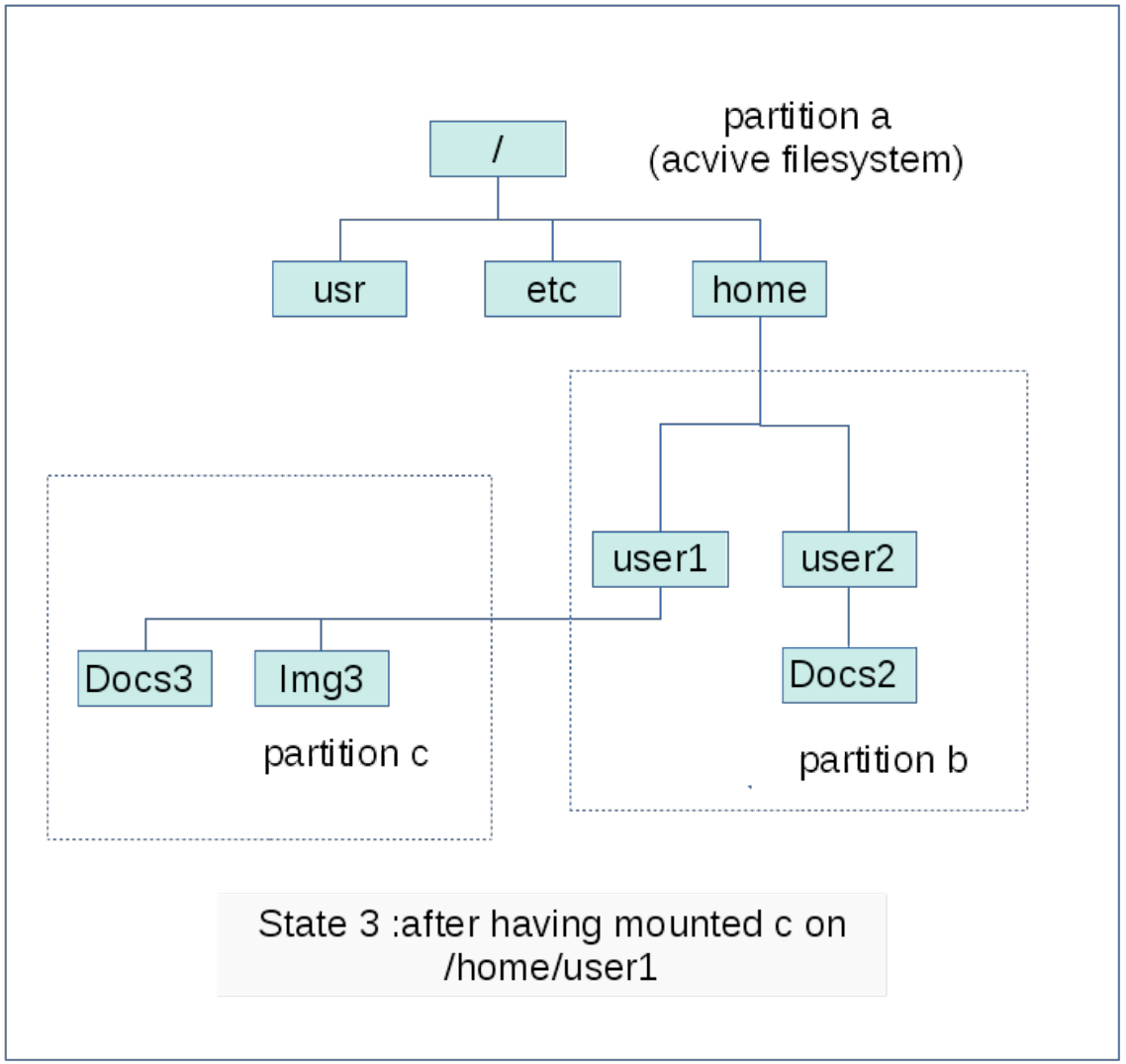
Mounting a file system

- The file system of partition b has been mounted in the directory /home



Mounting a file system

- Here the file system of partition c has been mounted in /home/user1
- Note, the subdirectory Docs1 has disappeared, this is what happens when a file system is mounted in a directory that is not empty



Mounting a file system: the device side

- Beside the mounting point, the operating system must be given the name of the device (partitions, USB keys, etc)
- In Linux, the names of the devices are in /dev
  - sda for the disk partitions, sda1, sda2 for partitions 1 or 2,
  - sdb for USB key
  - cdrom
  - fd for floppy disks
- To mount a USB key, the device /dev/sdb of the file system is combined with the mount point /mnt/media directory using:

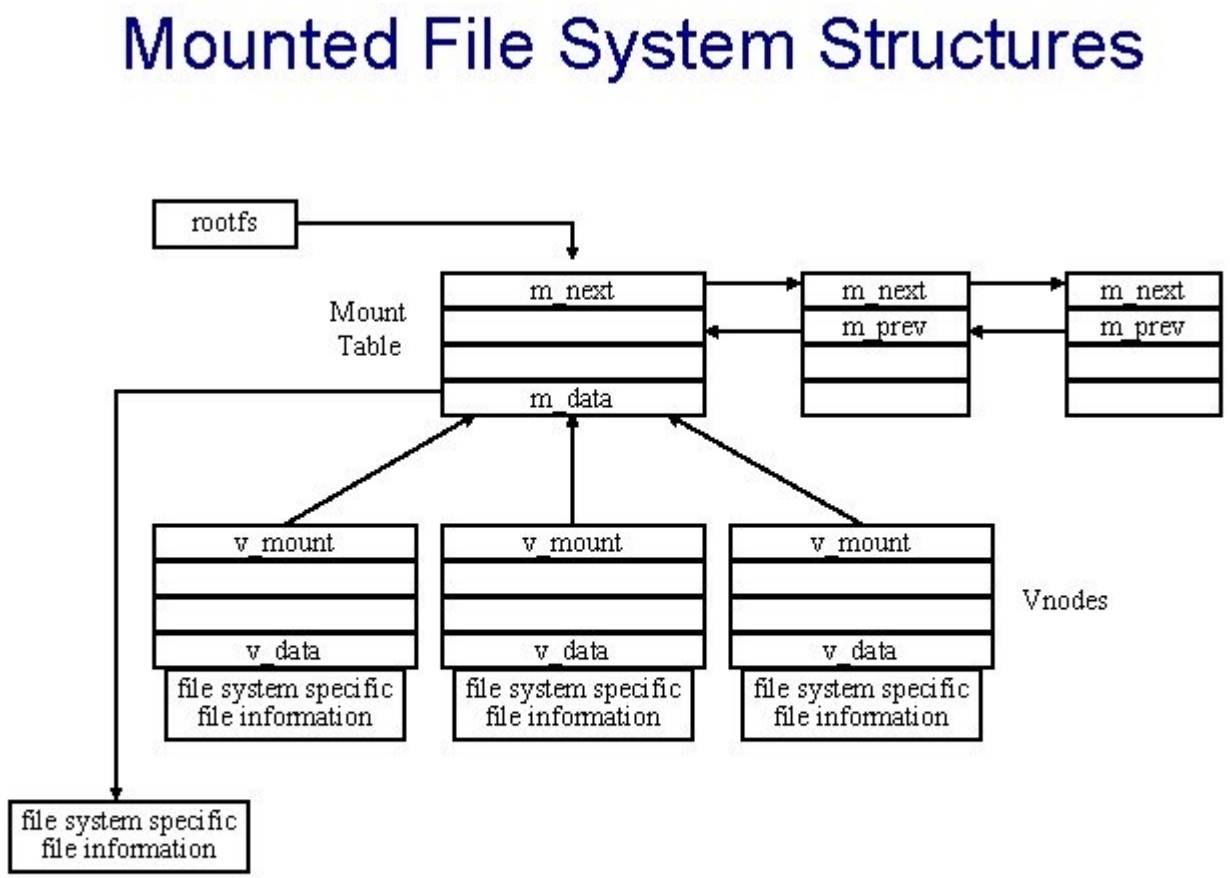
  sudo mount /dev/sdb /mnt/media

Windows: mounting file system

- Windows–based systems mount each partition with a separate name, denoted by a letter and a colon such as C:
- To record that a file system is mounted at C:, for example, the operating system places a pointer to the file system in a field of the mount table corresponding to C:.
- When a process specifies the driver letter, the operating system finds the appropriate file-system pointer and traverses the directory structures on that device to find the specified file or directory.
- One can find all the Windows partitions by right-click the Start button and select Disk Management.
  - 2 hidden partitions that holds information about the system boot file and the disk boot sector in case of system failure. It is the part of the hard disk that is not displayed or used directly under normal conditions.

Unix/Linux: Mounting file system

- File systems can be mounted at any directory.
- Mounting is implemented by setting a flag in the in-memory copy of the inode (v-node) for that directory.
- The flag indicates that the directory is a mount point.
- A field then points to an entry in the mount table, indicating which device is mounted there.
- The mount table entry contains a pointer to the superblock of the file system on that device.

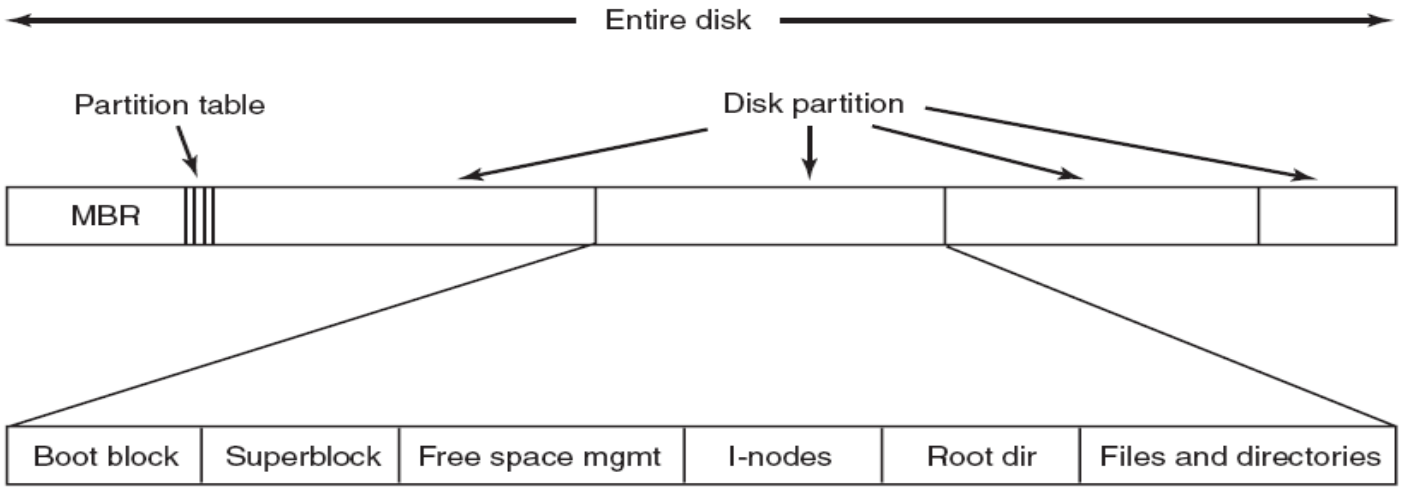


Boot sequence

- CPU reads and executes a sequence of instructions from the BIOS, a program on a small memory cheap sitting on the motherboard
- Among these instructions, some request the CPU to load the first sector of the hard disk, the 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

Boot sequence

- The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition.
- One of the partitions in the table is marked as active.



Boot sequence

- 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
- Usually, the file system of the boot partition, the one selected by the boot block, is mounted at boot time.
- Other partitions can be automatically mounted at boot or manually mounted later