UNIX File Systems

How UNIX Organizes and Accesses Files on Disk

Why File Systems

- File system is a service which supports an abstract representation of the secondary storage to the OS
- A file system organizes data logically for random access by the OS.
- A virtual file system provides the interface between the data representation by the kernel to the user process and the data presentation to the kernel in memory. The file and directory system cache.
- **Because of the performance disparity between disk and CPU/memory, file system performance is the paramount issue for any OS**

Main memory vs. Secondary storage

- Small (MB/GB)
- Expensive
- Fast (10-6/10-7 sec)
- Volatile
- Directly accessible by CPU
  - Interface: (virtual) memory address

✓ Large (GB/TB)

✓ Cheap

✗ Slow (10^{-2}/10^{-3} sec)

✓ Persistent

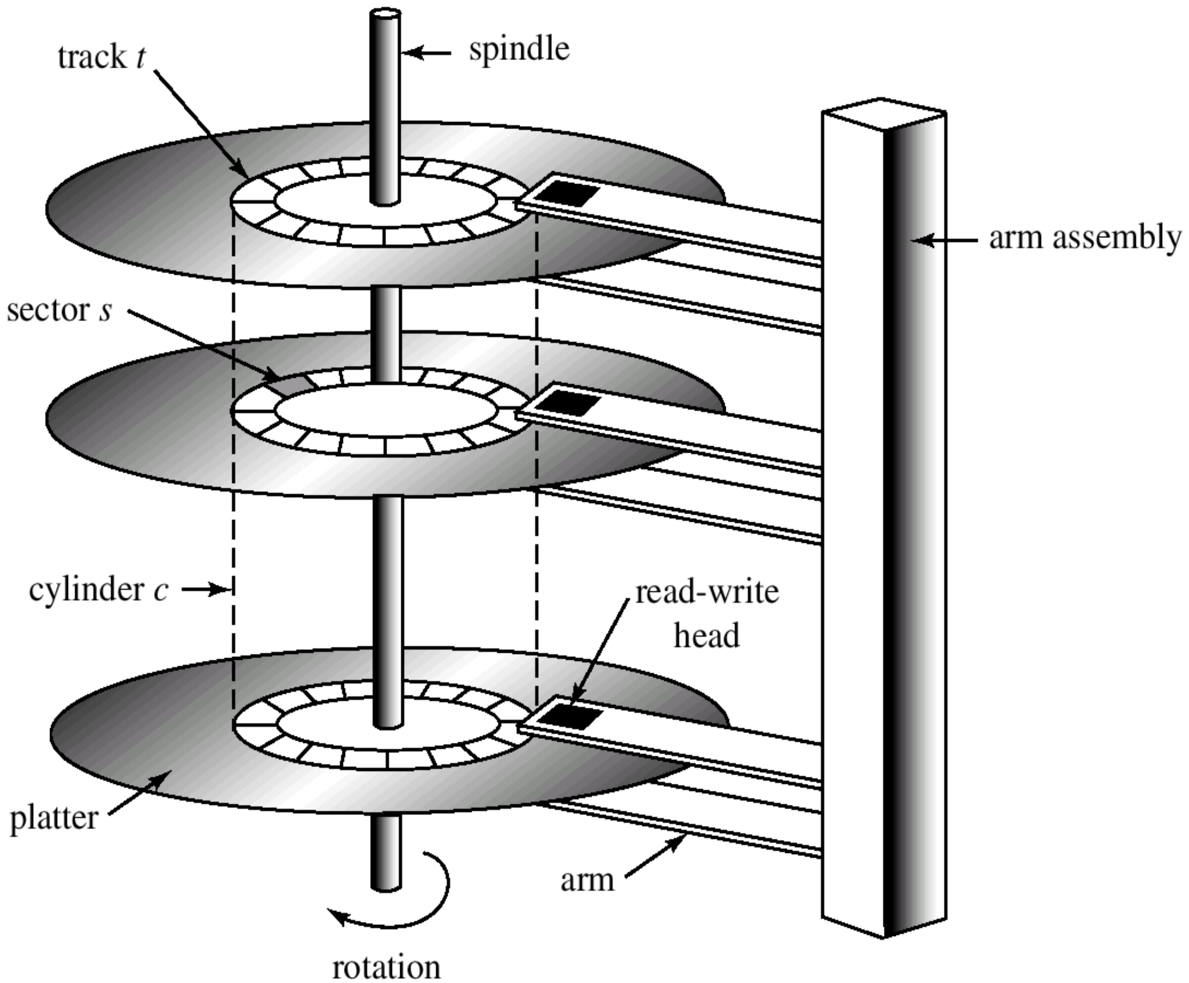
✗ Cannot be directly accessed by CPU
  - Data should be first brought into the main memory

Secondary storage (disk)

- A number of disks directly attached to the computer
- Network attached disks accessible through a fast network - Storage Area Network (SAN)
- Simple disks (IDE, SATA) have a described disk geometry. Sector size is the minimum read/write unit of data (usually 512Bytes)
  - Access: (#surface, #track, #sector)
- Smart disks (SCSI, SAN, NAS) hide the internal disk layout using a controller type function
  - Access: (#sector)
- Moving arm assembly (*Seek*) is expensive
  - Sequential access is x100 times faster than the random access

Internal disk structure

- Disk structure



User Process Accessing Data

- Given the file name. Get to the file's FCB using the file system *catalog (Open, Close, Set_Attribute)*

- The catalog maps a file name to the FCB
  - Checks permissions

- *file_handle=open(file_name)*:
  - search the catalog and bring FCB into the memory
  - UNIX: in-memory FCB: in-core i-node

- Use the FCB to get to the desired offset within the file data:
  (*CREATE, DELETE, SEEK, TRUNCATE)*

- *close(file_handle)*: release FCB from memory

Catalog Organization (Directories)

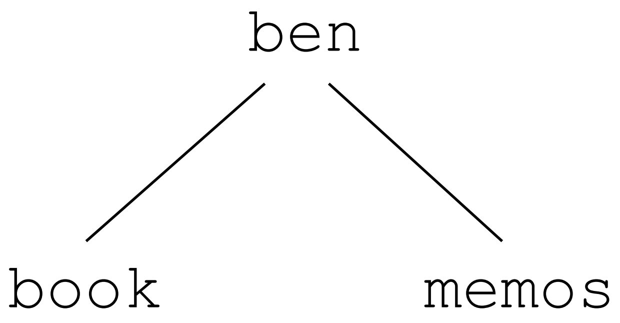
- In UNIX, special files (not special device files) called *directories* contain information about other files. A UNIX directory is a file whose data is an array or list of (filename, i-node#) pairs.
  - it has an owner, group owner, size, access permissions, etc.
  - many file operations can be used on directories
  - As a file, a directory has an I-node type structure.
  - A flag in the structure indicates its type.
- Unlike other files, the kernel imposes a structure on directory files using *mkdir*.
- A directory is a sequence of lines , a sequence of *directory entries* of variable length where each line contains an *i-node number* and a *file name* mapping: **<filename, inode #>**
- Directory data is stored as binary, cannot use cat. But some older UNIXs allow: **od -c dir-name.**
- Although directories are files, UNIX permissions – rwx- have slightly different meanings:
  - r, lists directoy contents
  - w, add a file to the directory
  - x, cd to the directory

Subdirectories

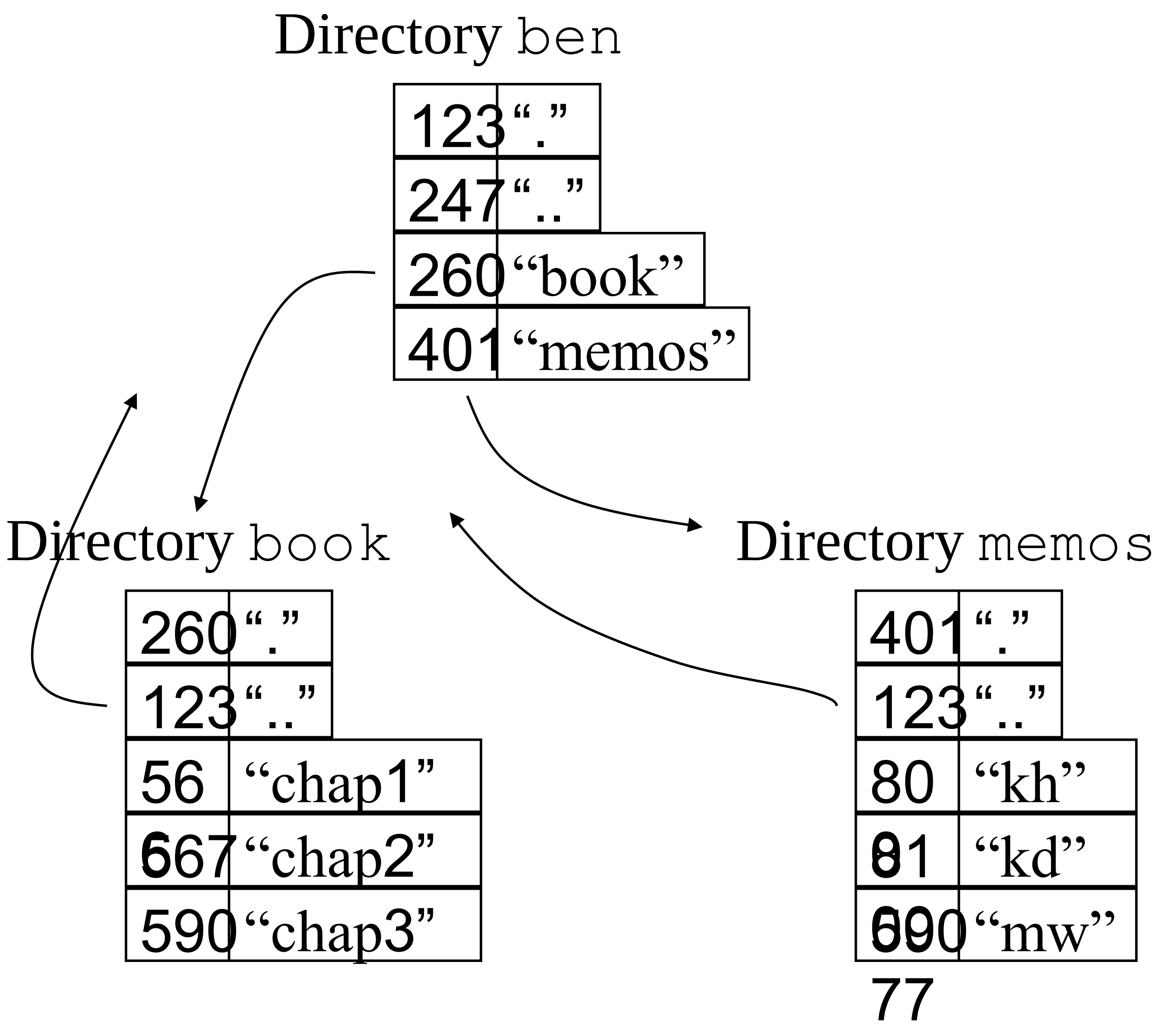
- *mkdir subdir* causes:
  - the creation of a subdir directory file and an i-node for it
  - an i-node number and name are added to the parent directory file

120	“fred.html”
207	“abc”
135	“bookmark.c”
201	**“subdir”**

- “.” and “..” are stored as ordinary file names with i-node numbers pointing to the correct directory files.



Subdirectories in more detail:



Directory ben

123	“.”
247	“..”
260	“book”
401	“memos”

Directory book

260	“.”
123	“..”
56	“chap1”
567	“chap2”
590	“chap3”

Directory memos

401	“.”
123	“..”
80	“kh”
81	“kd”
590	“mw”

77

Directory block or chunk

- Directories are allocated in *chunks.* Each chunk can be read/written in a single I/O operation, Why? Performance.

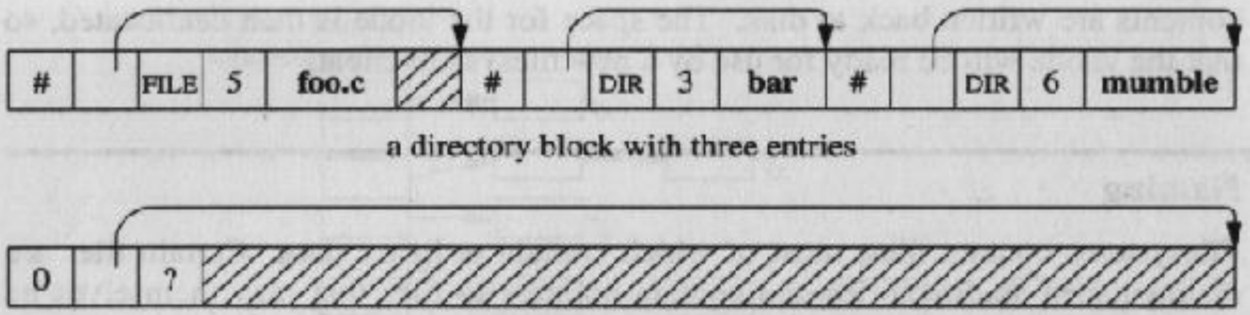


Figure 7.5 Format of directory chunks.

Regular Files and I-nodes

- Information about each regular local file is contained in a structure called an INODE. Containing the file FCB information.

- There is 1-to-1 mapping between the INODE and a file. However a file may have multiple INODES. Large files may in fact multiple layers of INODE's using *indirection* to keep track of data blocks.

- Each INODE is identified through its *number, a n*on-negative integer as an index into -

- The INODE hash array is a list of *allocated* INODE's located at the beginning of the file system

- INODE structures (UNIX i-node list) are stored on the file system block device (e.g., disk) in a predefined location on the disk. UNIX: the i-node list. Where it is exactly is file system implementation specific.

- INODE numbers have only local meaning (to each file system)

- One file system per device, one INODE table per file system.

- Hierarchical structure: Some FCBs are just a list of pointers to other FCBs (i.e. indirection)

- To work with a file (through the descriptor interface) the I-node of the file is brought into the main memory as an in-core INODE (V-Node).

INODE Table

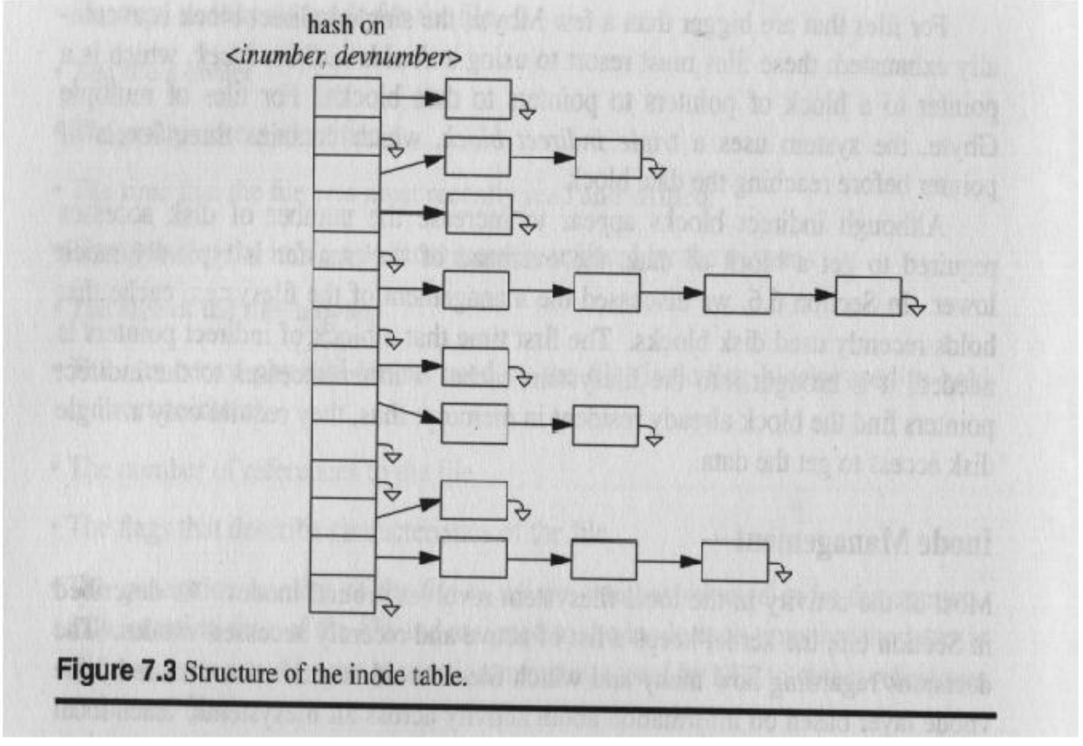


Figure 7.3 Structure of the inode table.

INODE Structure

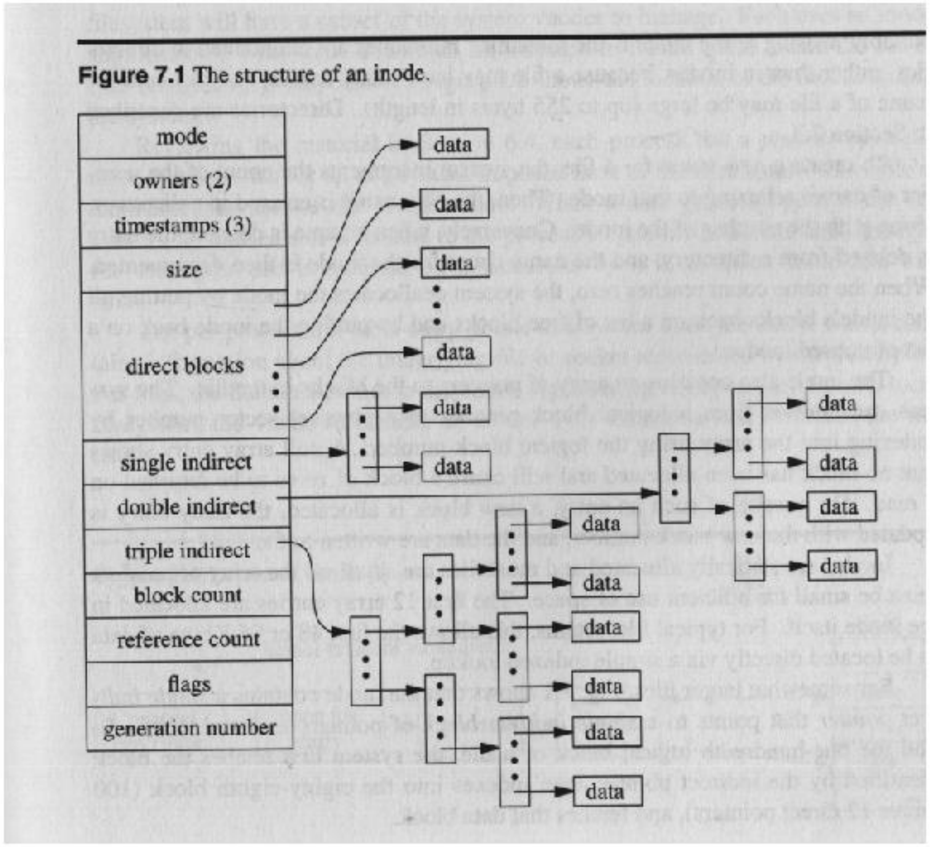


Figure 7.1 The structure of an inode.

File System Directory Structure

- The file system directory structure is a link between the INODE hash list and the directory files.
- The root "/" INODE is always #2. The root "/" directory file data block is located thru INODE #2.
- Directory file entries include each entry **<filename, inode #>.** Where *filename* is the local (unqualified) *directory* or *regular* filename within the directory and *inode#* is an index into the INODE hash array.
- The (*inode#)* INODE in the hash list entry has pointer to the data block of the subsequent *regular* file or *directory* file data block.
- If a directory file, the directory data blocks are read for the next **<filename, inode #>,** and the process repeats.
- If a regular file, the data blocks are located and read.
- This process also explains the difference between hard links and soft links. A hard link directory entry is a direct pointer to a file INODE. A soft link is a pointer to another directory entry.
- In a link, *rm* clears the directory record. Usually means that the i-node number is set to 0 but the file may not be affected
- The file i-node is only deleted when the last link to it is removed; the data block for the file is also deleted (reclaimed).

Hard links

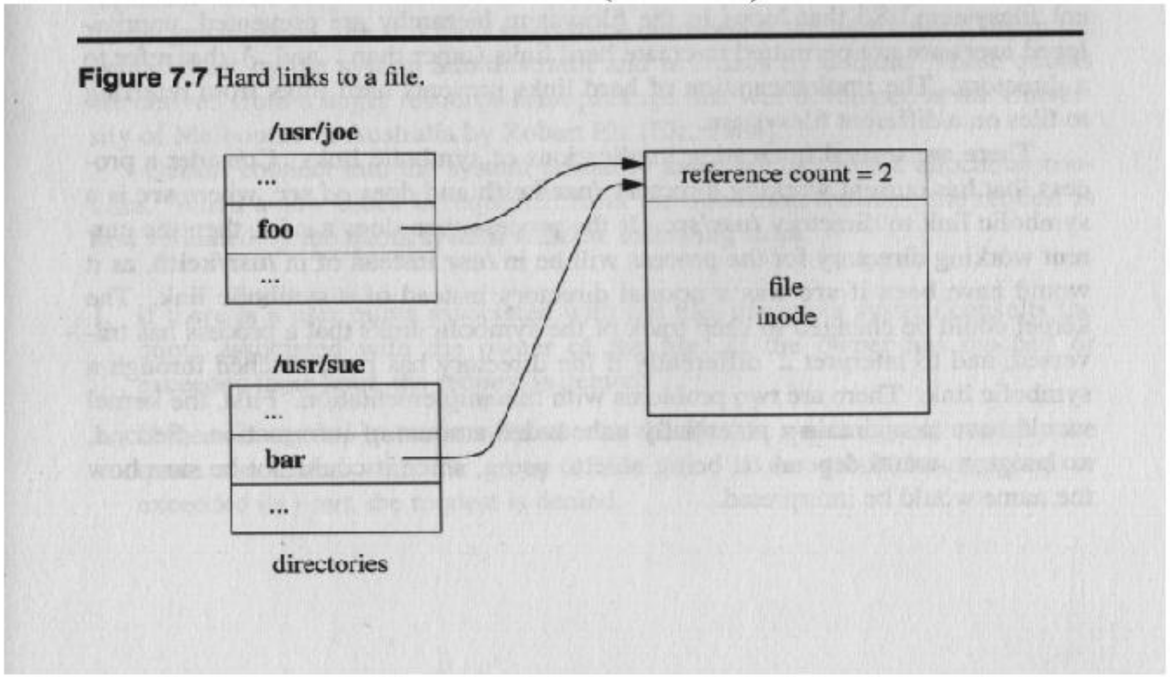


Figure 7.7 Hard links to a file.

Hard links cannot span different file systems (local meaning only)

Soft links

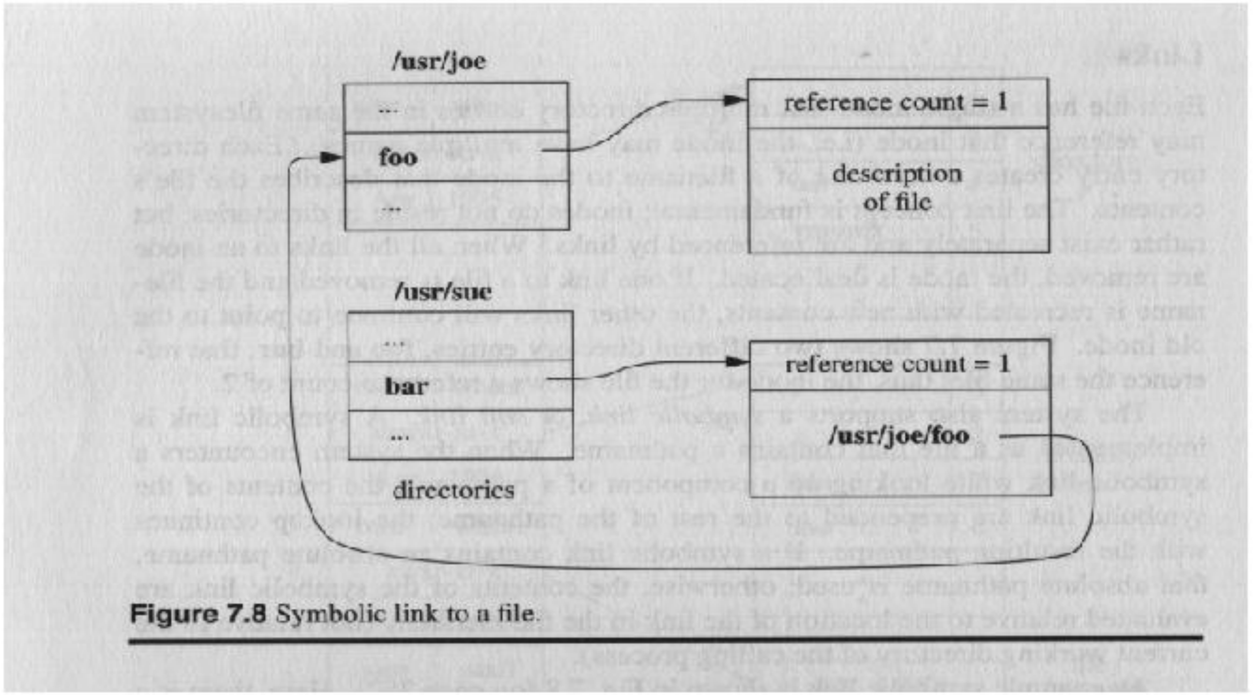


Figure 7.8 Symbolic link to a file.

Soft links can span the device boundaries

Example: Finding /usr/bin/vi

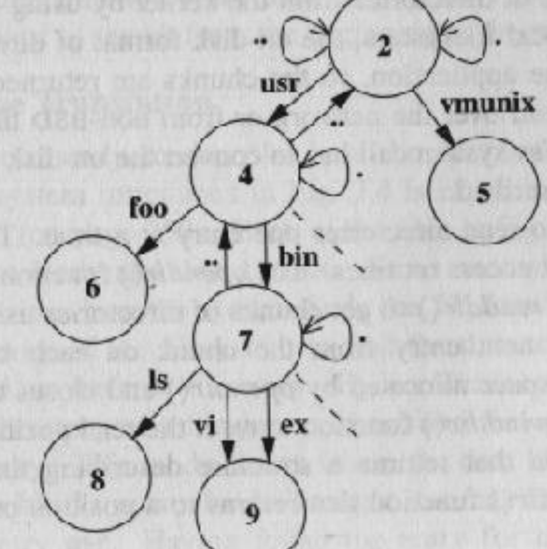
- **/usr/bin/vi** => i-node of **/usr/bin/vi**

- Get the root i-node:
  - The i-node number of ‘/’ is pre-defined (#2)
- Use the root i-node to get to the ‘/’ data
- Search (**usr**, i-node#) in the root’s data
- Get the **usr**’s i-node
- Get to the **usr**’s data and search for (**bin**, i-node#)
- Get the **bin**’s i-node
- Get to the **bin**’s data and search for (**vi**, i-node#)
- Get the **vi**’s i-node, find data block

- Permissions are checked all along the way
  - Each dir in the path must be (at least) executable

Finding /usr/bin/vi

Figure 7.4 A small filesystem tree.



File System Directory Structure

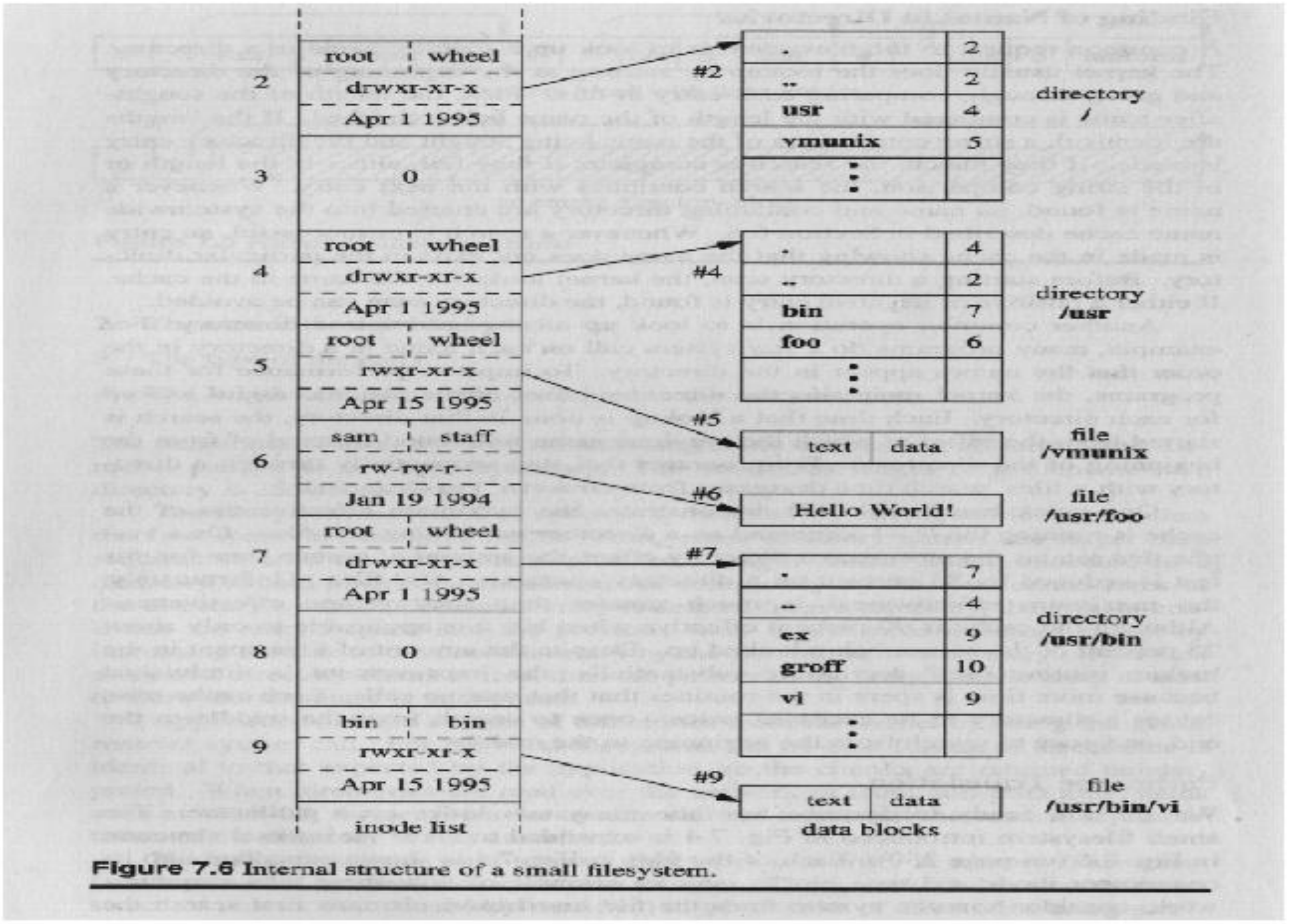
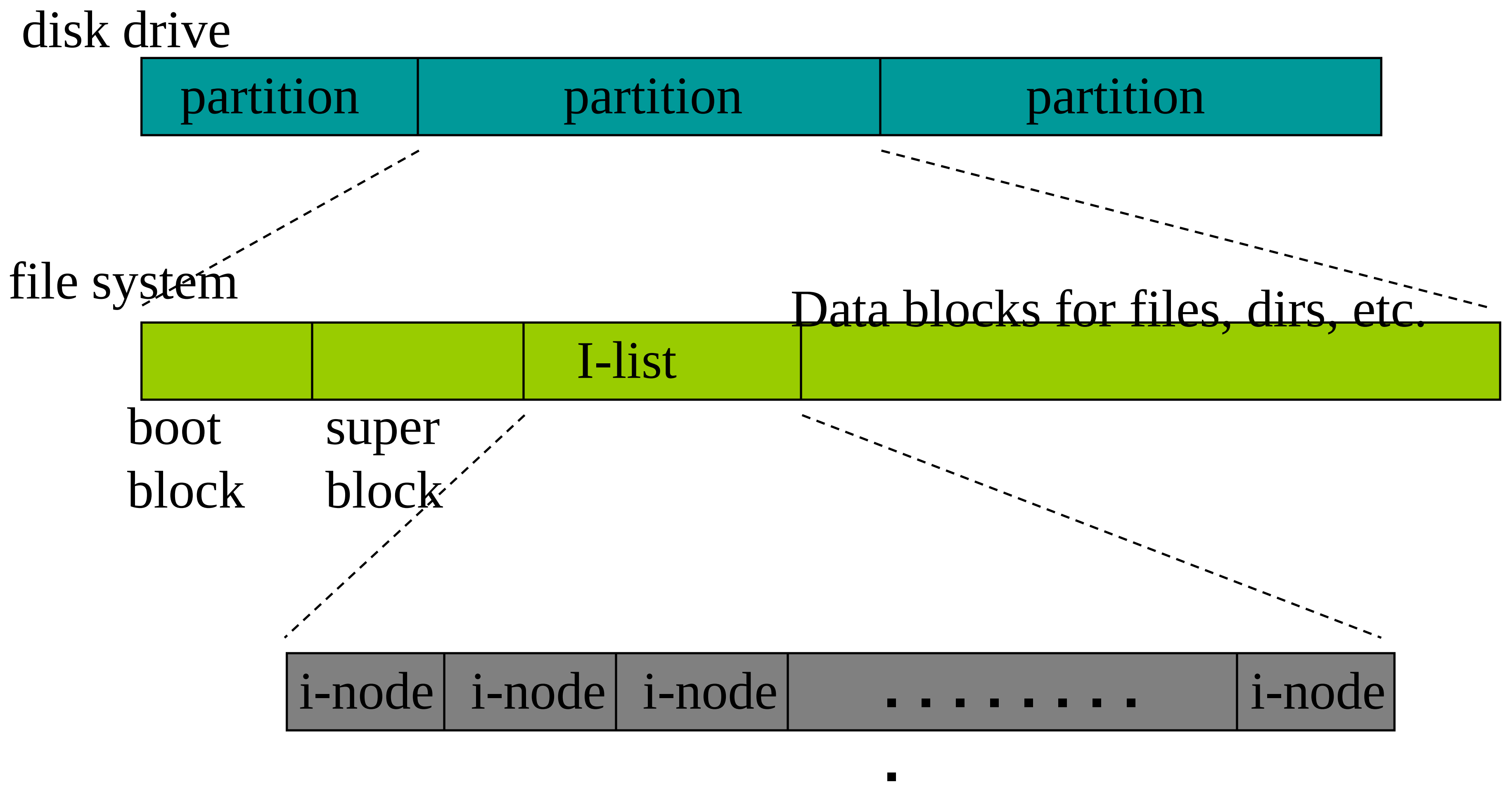
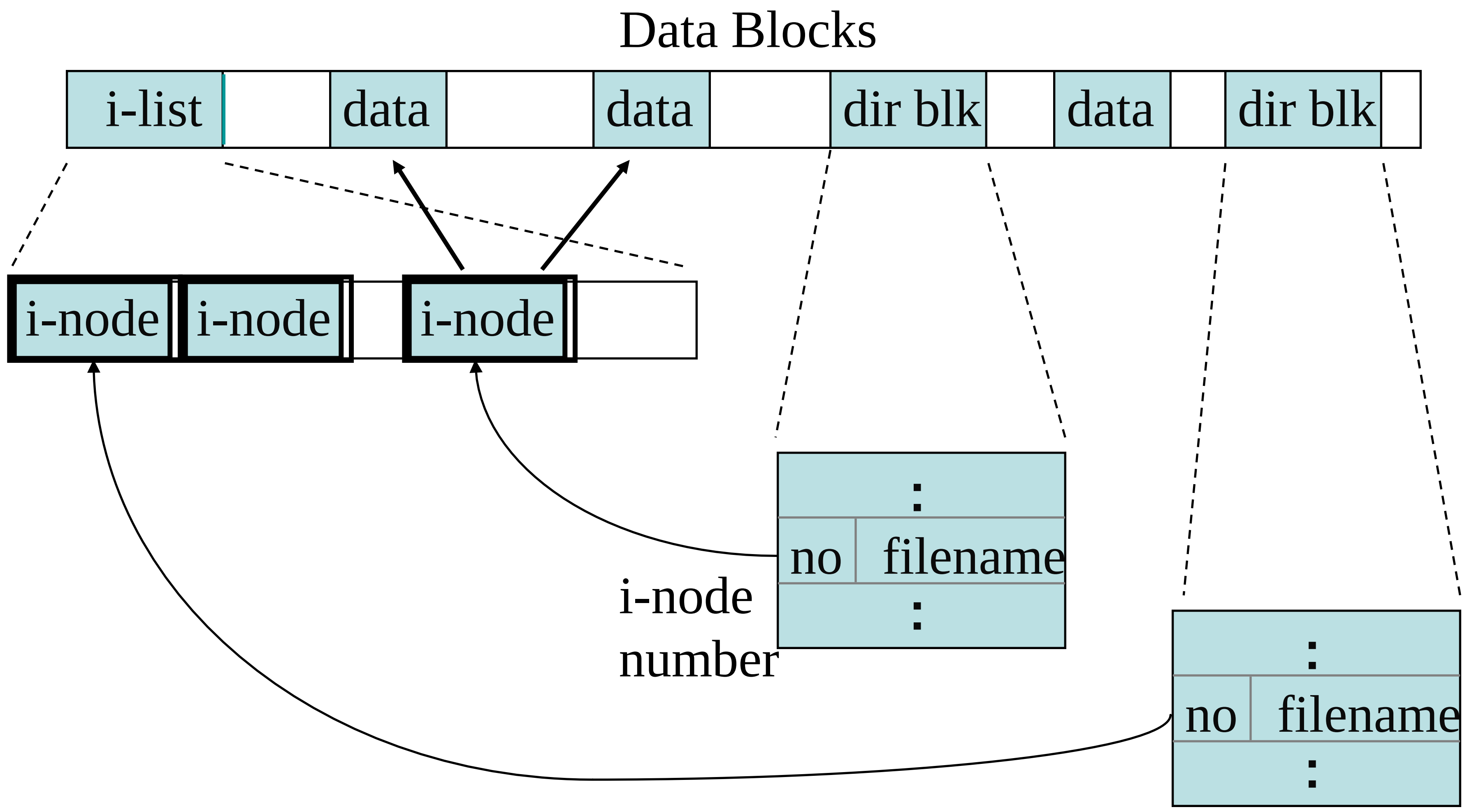


Figure 7.6 Internal structure of a small filesystem.

File System Data Structure



File System in More Detail



Classical UNIX File System

- Sequentially from a predefined disk addresses (cylinder 0, sector 0):

  – Boot block (Master Boot Record)
  – Superblock
  – I-node hash-array
  – Data blocks

- *Boot block*: a hardware specific program that is called automatically to load UNIX at system startup time.

- *Super block* -- it contains two lists:
  - a chain of free data block numbers
  - a chain of free i-node numbers

Superblock

- One per filesystem
- Contains:

  – Size of the file system;
  – The number of free blocks in the file system;
  – Size of the logical file block;
  – A list of *free data blocks* available for file allocation;
  – Index of the next free block on the list;
  – The size of I-node list;
  – The number of free I-nodes on the system;
  – The list of *free INODES* on the file system;
  – The index of the next free I-node on the list.

- tune2fs –l /dev/sdax to display superblock info
- df command output
- sync command
- du command

INODE Allocation

- As long as there is a free I-node – allocate it.
- Otherwise scan the I-node list linearly, and enter into the super-block list of free I-nodes as many numbers as possible.
- Remember the highest free I-node number.
- Continue with step 1.
- Next time start scanning from the remembered Inode number; when at the end – go back to the beginning of the I-node list.

Data Block Allocation

- Static and Contiguous Allocation (simple filesystem)

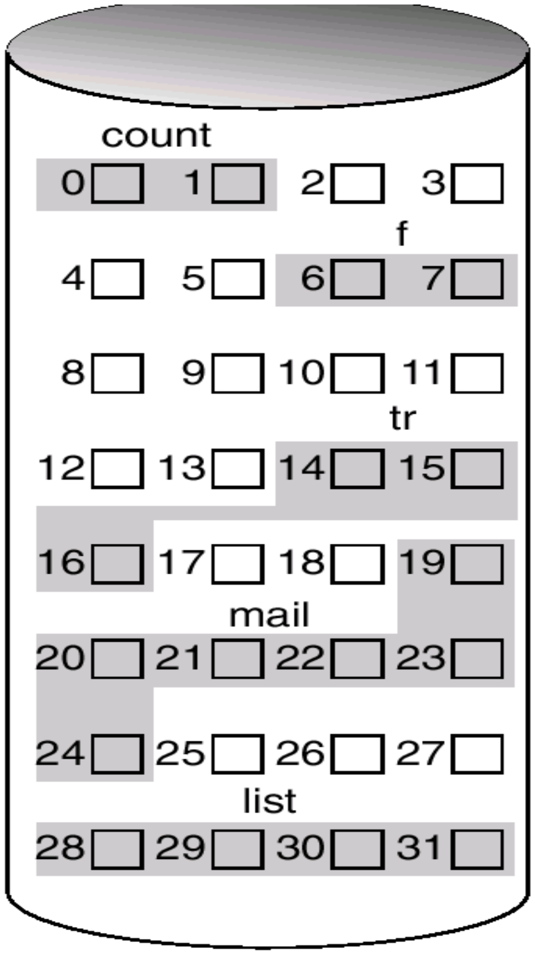
  - Allocates each file a fixed number of blocks at the creation time
  - Efficient offset lookup as only the block # of the offset 0 is needed
  - Inefficient space utilization, internal, external fragmentation
  - Fixed size, no support for dynamic extension

- Block allocation (UNIX, DOS, Ext, Ext2, Ext3)

  -Assumes unstructured files as an array of bytes
  - Efficient offset -> disk block mapping
  - Efficient disk access for both sequential and random patterns by minimizing number of seeks due to sequential reads
  - Efficient space utilization, minimizing external/internal fragmentation

- Extent allocation (FFS, NTFS, EXT4)
  - File get blocks in contiguous chunks called *extents*, multiple contiguous allocations
  - high performance for large files
  - larger block size = larger files

Static, Contiguous Allocation



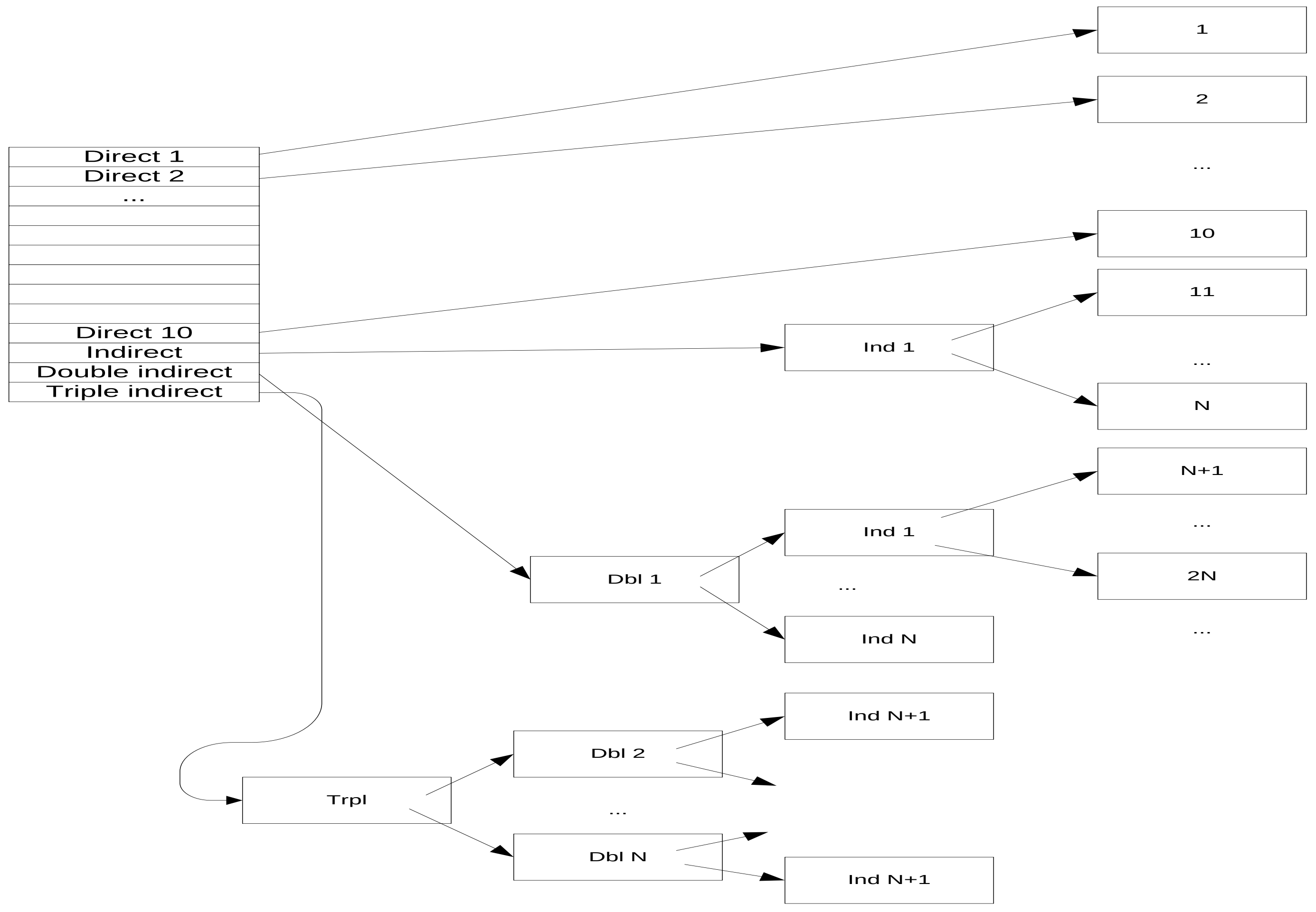
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Block Allocation in UNIX

- Classic UNIX Block and DOS FAT Linked file systems
- Extent-based allocation with a *fixed extent size of one disk block*. File blocks are scattered anywhere on the disk. Inefficient sequential access (*i.e. fragmentation)*.
- Dynamically expandable; accommodates large(r) file sizes better.
- No external fragmentation
- Optimized for small(er) files. Outdated empirical studies indicate that 98% of all files are under 80 KB
- Poor performance for random access of large files
- Wasted space in pointer blocks for large sparse files (indirection)
- UNIX block allocation (see slide)
  - 10 direct pointers
  - 1 single indirect pointer: points to a block of N pointers to blocks
  - 1 double indirect pointer: points to a block of N pointers each of which points to a block of N pointers to blocks
  - 1 triple indirect pointer…
  - Overall addresses $10+N+N^2+N^3$ disk blocks
  - N value determined by filesystem type. Determines maximum file size on a specific filesystem.

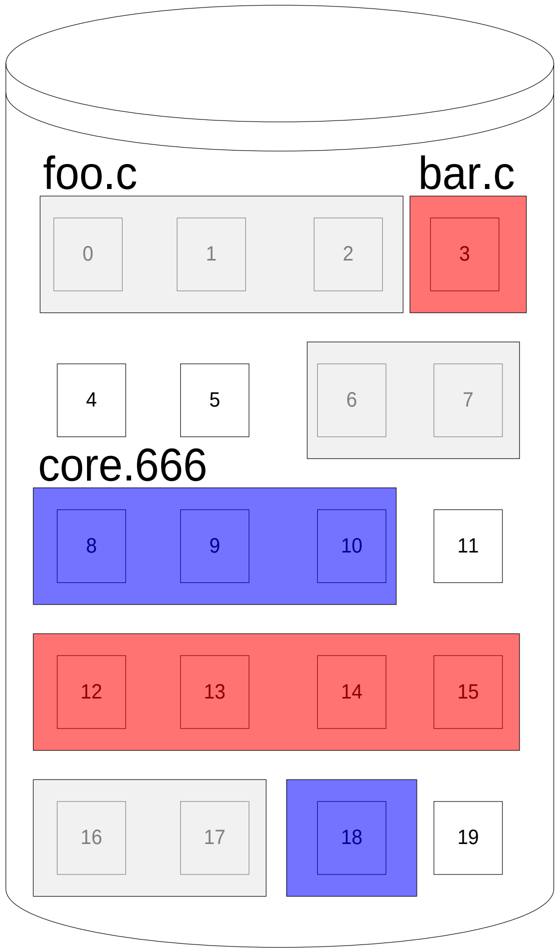
Block Allocation in UNIX



Extent-based, Multi-Block Allocation

- Modern UNIX implementations use the multi-block, extent-based allocation for performance
- File get blocks in contiguous chunks called *extents*. Multiple block with contiguous allocations.
- For large files, B-tree is used for efficient offset lookup.
- Efficient offset lookup and disk access. Uses newer “smart” disk sequential geometry (sector #)
- Support for dynamic growth/shrink
- Dynamic memory allocation techniques are used (e.g., first-fit)
- Suffers from external fragmentation - use compaction

Extent-based allocation



Catalog

foo.c	(0,3)	(7,2)	(16,2)
bar.c	(3,1)	(12,4)	
core.666	(8,3)	(18,1)	

DOS File Allocation Table (FAT)

- A section at the beginning of the disk is set aside to contain the table
  - Indexed by the block numbers on disk
  - An entry for each disk block (or for a cluster thereof)

- Blocks belonging to the same file are chained
  - The last file block, unused blocks and bad blocks have special markings

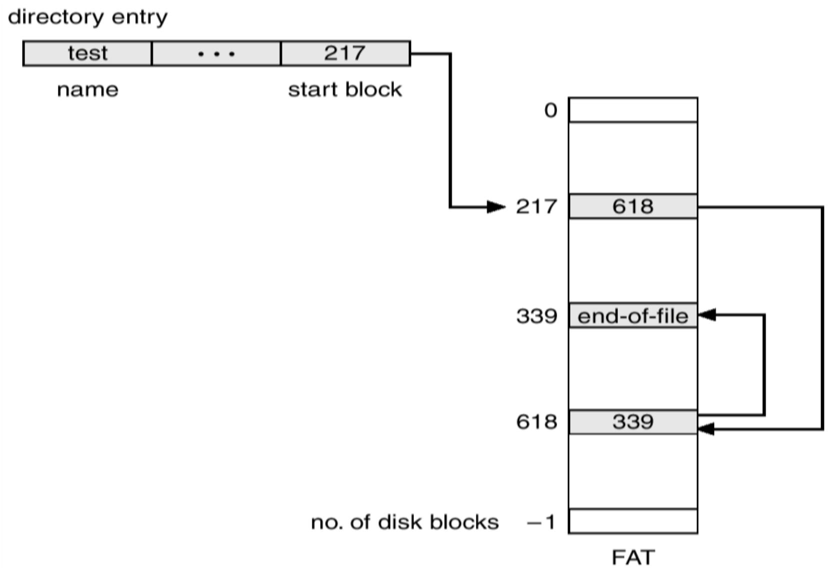
- Each file is a linked list of disk blocks

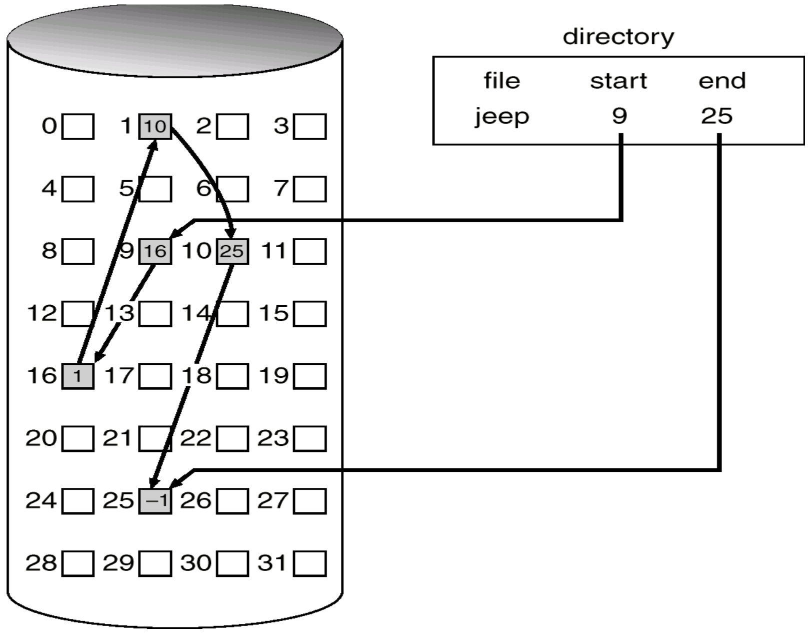
- Offset lookup:
  - Efficient for sequential access
  - Inefficient for random access

- Access to large files may be inefficient as the blocks are scattered
  - Solution: block clustering

- No fragmentation, wasted space for pointers in each block

DOS / FAT - Directory pointer and linked list block allocation





New UNIX File Support

- Issues with the old UNIX file system – fragmentation, performance, reliability..
- BSD Fast File System (FFS) - BSD
- Journaled File Systems (JFS) - AIX
- Log-based file systems (LFS) – Various OS
- Network File System (NFS) - Solaris