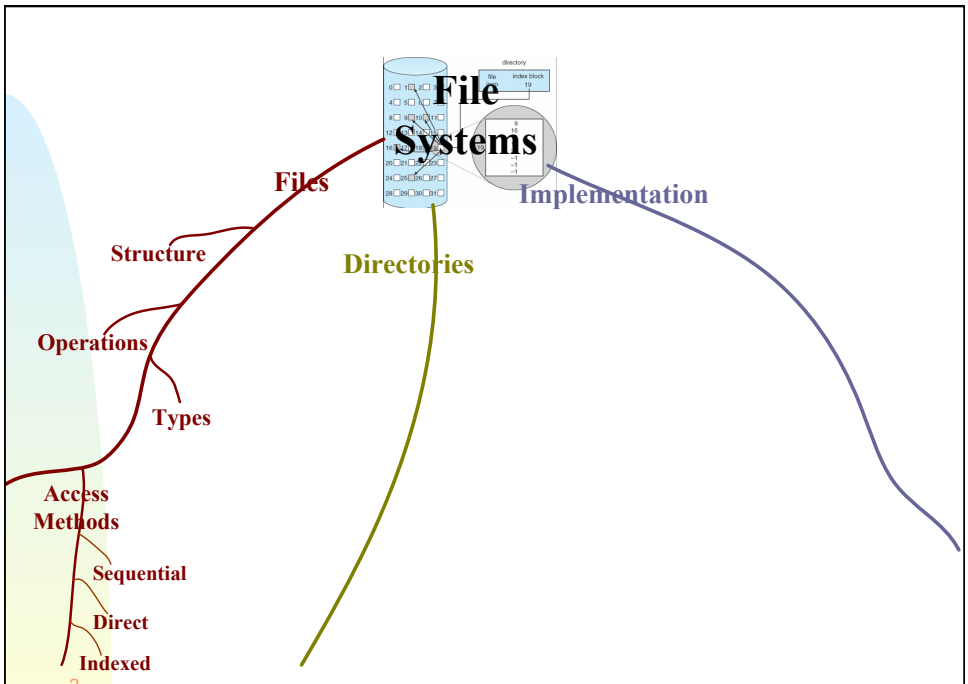
Module 9: File-Systems

Reading: Chapter 10 and 11

Objectives

- **File System Interface (Chap 10)**
  - Explain the function of file systems.
  - Describe the interfaces to file systems.
  - Discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures.
  - Explore file system protection.
- **File System Implementation (Chap 11)**
  - Describe the details of implementing local file systems and directory structures.
  - Describe the implementation of remote file systems.
  - Discuss block allocation and free-block algorithms and trade-offs.

1



2

File-System Interface

What is File-System?

- Organized data in files
- Files organized in directories, and the whole associated machinery

What are the main File-System concepts?

- Files
- Directories

What do we expect from a File-System?

- Efficient and convenient access methods
- Convenient way to organize data - Directory structure
- Transparent handling of several devices - File-System Mounting
- File Sharing
- File Protection

3

File Concept

What is a file?

- Named collection of related information
- Can be seen as collection of records
- Basic unit for storing data

What is the structure of a file?

- Simple structure - sequence of words, bytes
- Simple record structure
  - Lines
    - Fixed length
    - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file (executable)

4

File Structure

At the lowest level, a file is simply collection of bytes/words.

Interpretation of the file's contents is up to the

- **Operating system**
- **Program accessing the file**
- **Combination – some OS support different file types**

Unix:

- **Treats files as a sequence of bytes, it is up to the user/ program to interpret file's content**

Windows:

- **Recognizes several file types (using the file extension) and automatically launches/suggests the program that knows how to interpret the files of this type**

Databases:

- **Have specific needs, usually creates and interprets their own database files (e.g. indexed files)**

5

File Attributes

What needs to be remembered about each file?

- Name **– only information kept in human-readable form**
- Identifier **– unique tag (number) identifies file within file system**
- Type **– needed for systems that support different types**
- Location **– pointer to file location on device**
- Size **– current file size**
- Protection **– controls who can do reading, writing, executing**
- Time, date, and user identification **– data for protection, security, and usage monitoring**

Where is all this information kept?

- **in the directory structure, we will talk about that later**

6

File Operations

File can be seen as an abstract data type **supporting operations:**

- **Basic Operations**
  - **Create**
  - **Write: Pointer for write position**
  - **Read: Read pointer**
  - **Reposition within file (seek)**
  - **Truncate (clear): sets file size to zero, but maintains all attributes**
  - **Delete: Releases space**
- **Other Operations**
  - **Rename,**
  - **Copy**
  - **Change attributes: owner, permissions, etc.**
  - **…**

7

File Operations

Do we need to specify with each file operation the file name and a location within the file (i.e. for read or write)?

- **No**
  - **File Descriptor: we ask the file system to open the file and get a file descriptor (file handle)**
  - **The file operations (read/write/seek) then use this handle to identify the file**
  - **File Pointer: there is also a file pointer maintained that points to the current location within the file, to be used for the next read or write**
- **Why?**
  - **Would be inefficient to do otherwise**

8

Open Files

- open(fileName, mode)
  - Searches the directory structure to find the record containing the file attributes for file *fileName*, then loads this record into memory and returns a descriptor that references it
  - Mode specifies additional information, i.e. open for read only, …
- close(file_descriptor)
  - Clears the structures kept about this open file
- What needs to be in the structure for each open file?
  - File pointer: pointer to the location of the next read/write
    - Per file or per process? Per process
  - Location on the disc (to know where to physically read/ write the data)
  - File-open count
    - Per file or per process? Per file
  - Access rights
    - Per file or per process? Both

9

Open Files

- What needs to be in the structure for each open file?
  - File pointer
  - Location on the disc
  - Access rights
  - Can a file be opened by several processes simultaneously?
  - Yes. Do we maintain separate file attribute records in memory for each process the file is opened?
    - Yes - for file pointer, access rights
    - No – for location on the disc
  - If several processes open the same file, and then some of them close it, how do we know when can we release the record from the memory?
    - Keep the counter of processes that opened this file
  - Locks for controlling concurrent access to the file

10

Open File Locking

- Provided by some operating systems and file systems
- Mediates access to a file
- Mandatory or advisory:
  - Mandatory – access is denied depending on locks held and requested
  - Advisory – processes can find status of locks and decide what to do
- Reader-Writer Locks
- Recall synchronization concepts

11

File Types

- Certain operating systems use the extension of the file to identify its type.
  - Microsoft: an executable file must extension .EXE, .COM, or .BAT (otherwise OS refuses execution).
- The type is not defined for certain OS's
  - UNIX: the extension is used (and recognized) by applications only.
- For other OS's, the type is an attribute
  - MAC-OS: the file has an attribute that contains the name of the program that generated it (ex: document Word Perfect)

12

File Types – Name, Extension

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 compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

13

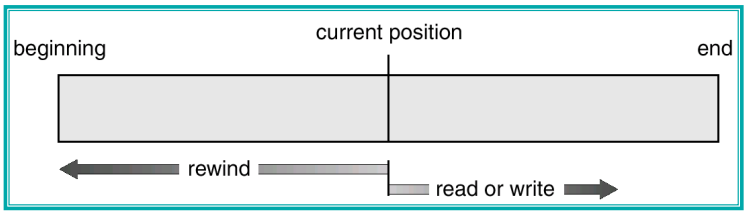
File Access Methods

- The access method is determined by the logical structure
- OS on « mainframes » usually support many access methods (one per file type)
  - Because they support many file types (e.g. indexed)
- But many modern OS (Unix, Linux, MS-DOS…) basically support very few access methods (ex: sequential and direct) since all files are of the same type (ex: byte streams)
- Ex: Data Base Management Systems (DBMS) almost always support indexed access of files even when the OS does not support this.
  - In that case the DBMS uses the basic random-access support of the OS to provide indexed file access to the DBMS user
  - But the same DBMS on a main-frame OS would directly use the index file access method provided by the OS

14

Sequential Access

- The most common method and the simplest method
- Records can be accessed one after the other in their sequential order of storage
  - Based on the tape model
- The operation read_next() reads the next record and advance the pointer to the next record
  - Operation write_next() appends a record at the end
- We may also come back to the beginning (rewind)
- It is sometimes possible to jump n records (backward or forward)



15

Direct Access

- Based on disk model containing blocks of data
- A direct access file consists of a set of logical blocks
  - Their size is the same as those of physical blocks
  - They are numbered from 0 to k (for a file of k+1 blocks)
    - But logical blocks are positioned on arbitrarily-chosen physical blocks on disks according to the file allocation method used (see later).
  - Each logical block can be directly (and independently) accessed
- Each logical block consists of R records of the same size
  - Hence, we may have internal fragmentation
  - The first logical block contains the first R records. The next group of R records is in the second logical block, an so on…
- Access to record number N is done by first extracting logical block number ⌊N/R⌋ from secondary memory and bringing it into main memory
  - This is the logical block containing the desired record

16

Direct and Sequential Access

- Not all OS's offer both sequential and direct access
  - Easy to simulate sequential access with direct access
    - Maintain a pointer *cp* that indicates the current position in a file.
  - The reverse is very difficult and inefficient.

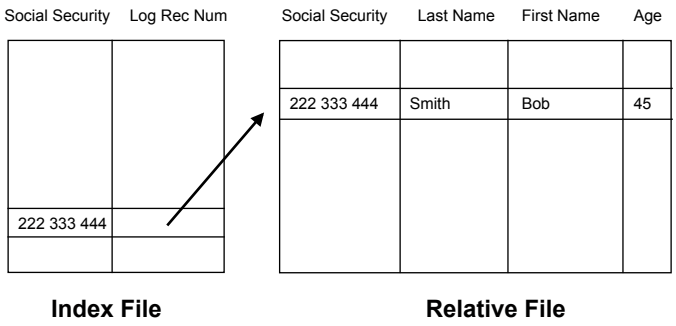
17

Indexed Files

- It is a direct access method performed by using an index
  - Heavily used by DBMS
- We need to use two files (per data file):
  - A relative file : a direct access file containing the data (in logical blocks)
  - An index file : containing the indices
- An index consists of a key and a pointer
  - Value of the pointer is the logical block number containing the record identified by the key
  - The key is one field in record of the relative file. Its value provides a unique identifier for each record.
    - It is not permitted to have two different records of the same file having the same key value
    - Example of key: social insurance number

18

Indexed Files (cont.)



- The index file is an ordered list (by key value)
- We simply perform a binary search whenever the index file can fit entirely into main memory
  - If not, then we can create an index for the index file!
    - The primary index file is first consulted to find the logical block of the secondary index file containing the desired key value
    - That logical block is then searched to find the logical block containing the desired record

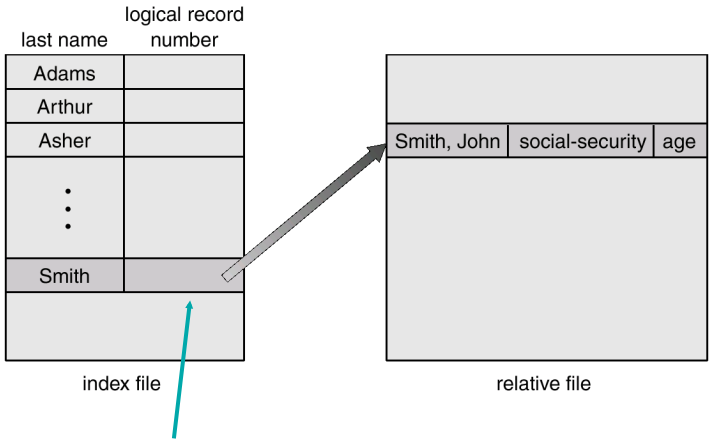
19

Indexed Files (continued)

- An index can also lead to a point close to the desired record.
  - The keys in the index file contain references (pointers) to certain important points in the relative file (e.g., beginning of names that start with the letter S, beginning of the Smiths, beginning of serial numbers that start with 8)
  - The index file provides the means to access rapidly a point in the relative file; the search then continues from that point.
  - Such a file could also index another index file, i.e. serves as the primary file described in the previous slide.

20

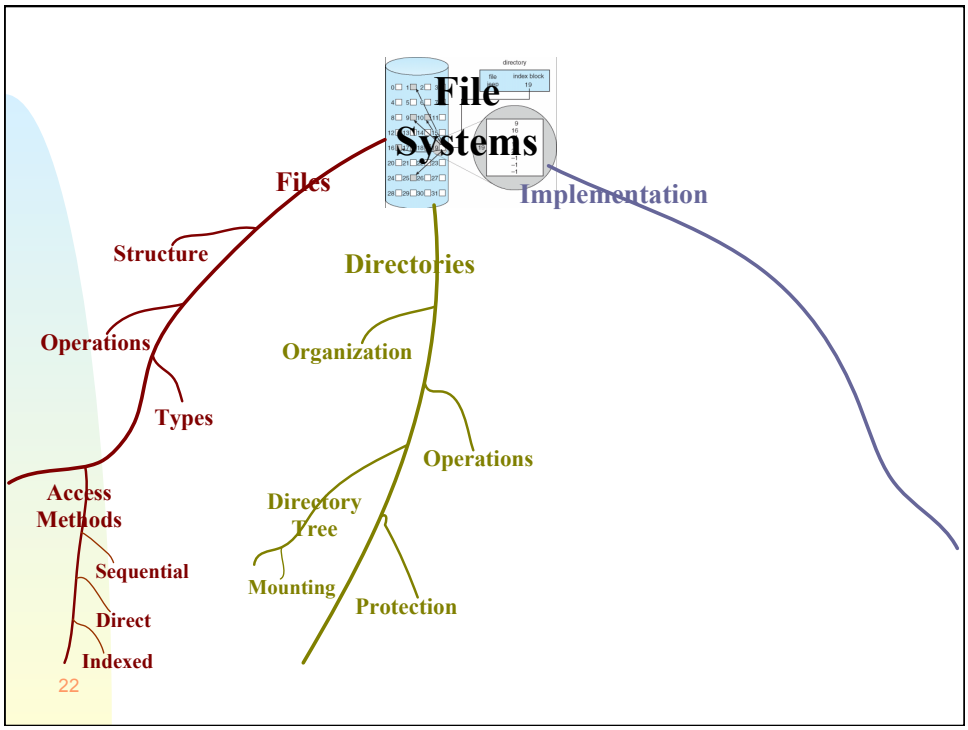
Example of Indexing



Points to the start of the Smiths (there are many)

The relative file must support direct access.

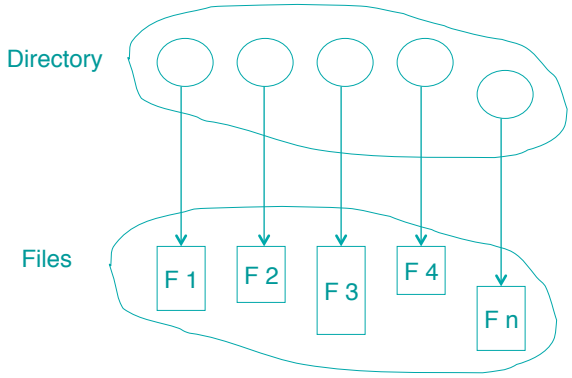
21



22

Directory Structure

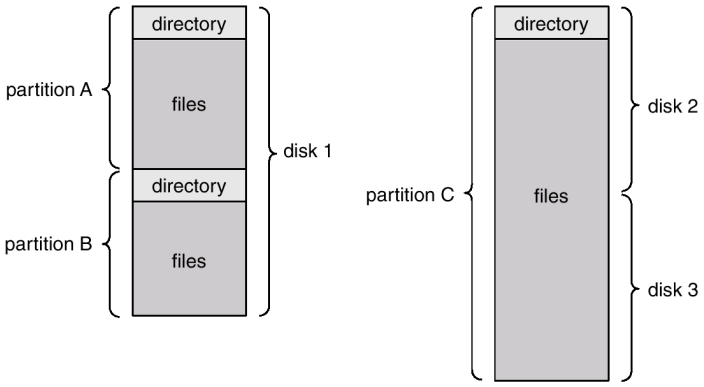
- A collection of nodes containing information about all files



Both the directory structure and the files reside on disk

23

Typical Organization of a File-System



24

Information in a Directory

- File names
- Type
- Location on disk (or other device)
- Current length
- Maximum length
- Date last accessed
- Date last modified
- Owner
- Protection

25

Operations Performed on Directory

What are the operations performed on a directory?

- Search for a file
- Create a file
- Delete a file
- Rename a file
- List a directory
- Create/Delete/Rename directory
- Traverse the file system

26

Why do we use directories?

- Efficiency – locating a file quickly
- Naming – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)

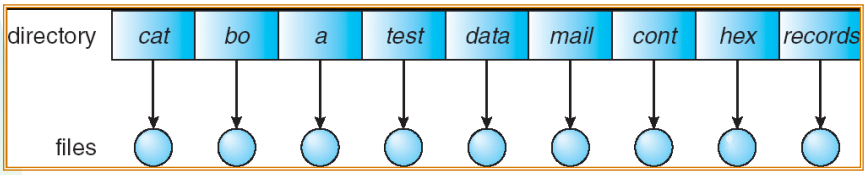
Each directory holds directory entries for the files it contains

- We will discuss later file system implementations

27

Single-Level Directory

- A single directory for all users



Any problems?

With many files becomes total mess

Naming conflicts between users

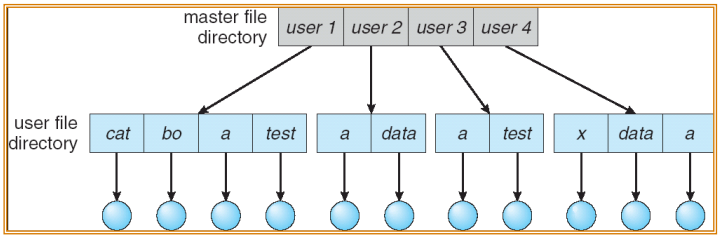
…

Primitive and not practical

28

Two-Level Directory

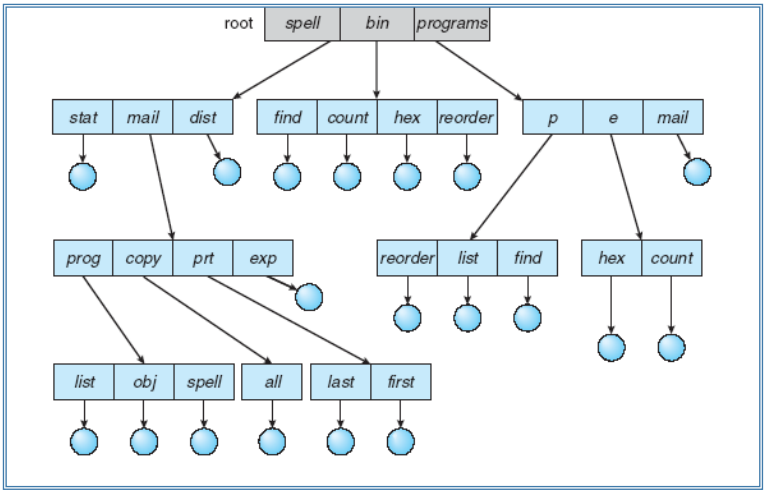
- **Separate directory for each user**



- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

29

Tree-Structured Directories



30

Tree-Structured Directories (Cont)

- **Efficient searching**
- **Grouping Capability**
- **Useful concept: current/working directory**
  - **cd /spell/mail/prog**
  - **type list**

31

Tree-Structured Directories (Cont)

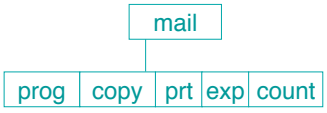
- Absolute **or** relative **path name**
- **Creating a new file is done in current directory**
- **Delete a file**

  rm <file-name>
- **Creating a new subdirectory is done in current directory**

  mkdir <dir-name>

  Example: if in current directory /mail

  mkdir count

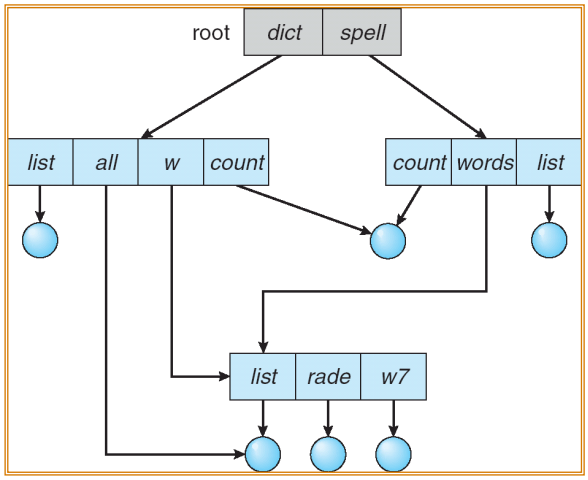


Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"

32

Acyclic-Graph Directories

- Idea: **Have shared subdirectories and files**



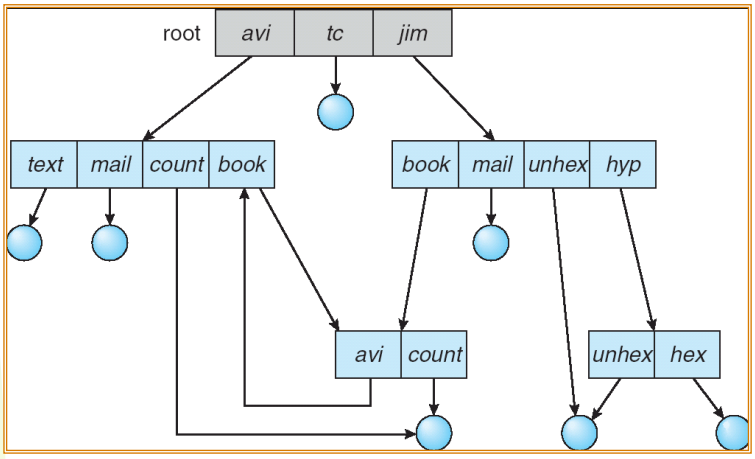
33

Acyclic-Graph Directories (Cont.)

- **How do we achieve sharing?**
  - **Have a special directory entry, not containing the file attributes, but containing the name of another file**
    - **Called symbolic (soft) link**
  - **Having two different directory entries that reference the same file attribute information (i.e. the file control block – FCB)**
    - **Called hard link (FCB contains a count of links to the file)**
- **What happens when a file being referenced by a symbolic link is deleted?**
  - **Dangling pointer**
- **What happen when a file referenced by a hard link is deleted?**
  - **The link is deleted**

34

General Graph Directory



35

General Graph Directory (Cont.)

- The tree-structured (easily searched) directory can be destroyed
  - When searching the directory, can search sub-directories multiple times.
- Cycles in the graph can occur.
  - When searching the directory, can end up with looping (infinite searching).
- When a link to a directory is removed, how to release the files contained in the subdirectory (link count is non zero)
- But how to allow or omit cycles?
  - Allow only links to files, not subdirectories
  - Every time a new link to a subdirectory is added use a cycle detection algorithm to determine whether it is OK
    - Complex and costly.
  - Allow only symbolic links to subdirectories and do not follow symbolic links
  - Use garbage collection to clean up the directory when a subdirectory is deleted (this is expensive).

36

Combining Several File Systems

- Directory tree residing on the particular device/partition

Why combining?

- Several HD partitions, floppy/ZIP disks, CDROM, network disks
- Uniform view/access to them

How?

- Mount a file system into particular node of the directory tree
- Windows: 2 level system – automatically mounts into drive letters; path to specific file: drive-letter:\path\to\file
- Unix: explicit mount operation, can mount anywhere

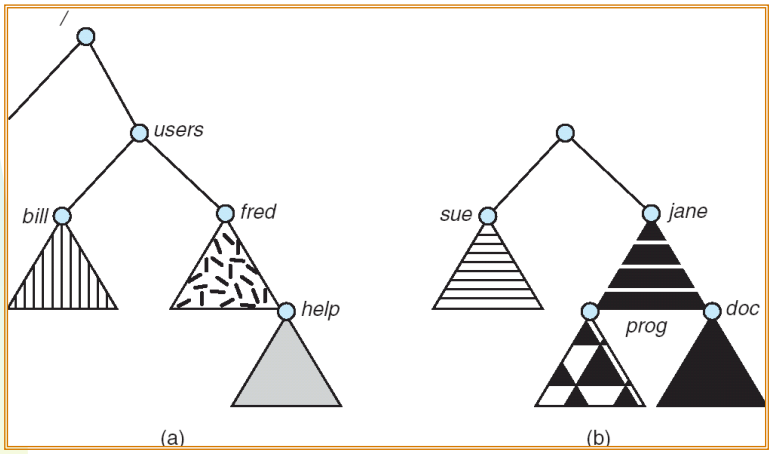
37

File System Mounting

- A *mount point* is a directory or file at which a new file system, directory, or file is made accessible. To mount a file system or a directory, the mount point must be a directory; and to mount a file, the mount point must be a file.
- **A file system must be** mounted **before it can be accessed**
- **An unmounted file system is mounted at a** mount point

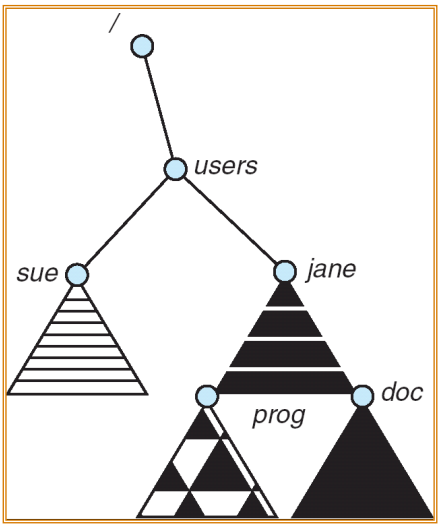
38

(a) Existing. (b) Unmounted Partition



39

Mount Point



40

Protection

- **File owner/creator should be able to control:**
  - **what can be done to the file**
  - **by whom**
- **Types of access**
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List** (list names and attributes in a subdirectory)

41

Access Lists and Groups - UNIX

- **Mode of access: read, write, execute**
- **Three classes of users**

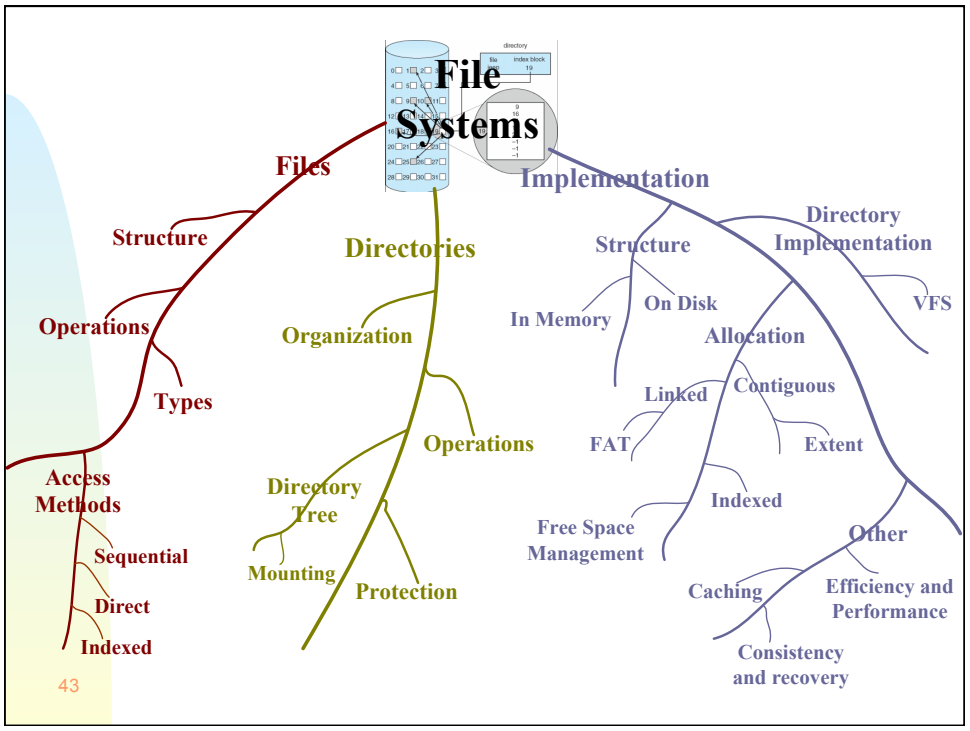
			RWX
a) owner access	**7**	⇒	**1 1 1**
b) group access	**6**	⇒	**1 1 0**
c) public access	**1**	⇒	**0 0 1**

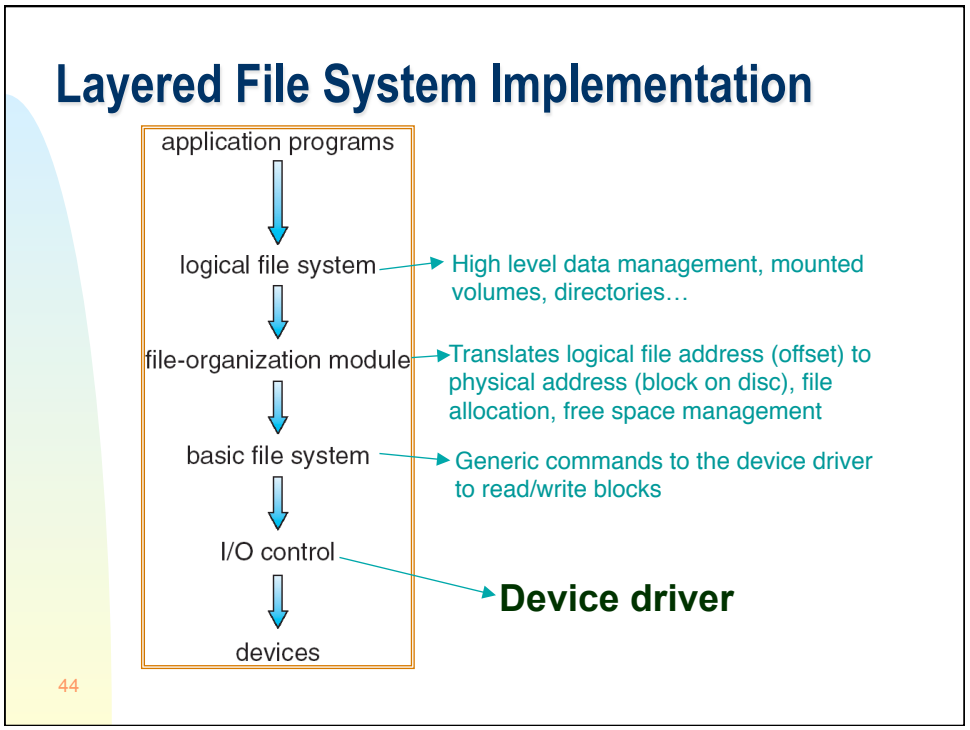
- **Ask manager to create a group (unique name), say G, and add some users to the group.**
- **For a particular file (say *game*) or subdirectory, define an appropriate access.**

Attach a group to a file

chgrp G game

42

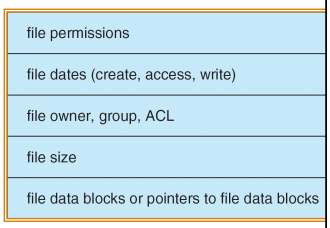

File Systems
Files
Structure
Operations
Types
Access Methods
Sequential
Direct
Indexed
Directories
Organization
Operations
Directory Tree
Mounting
Protection
Implementation
Structure
In Memory
On Disk
Allocation
Linked
Contiguous
FAT
Extent
Indexed
Free Space Management
Directory Implementation
VFS
Other
Caching
Efficiency and Performance
Consistency and recovery
43


Layered File System Implementation
application programs
logical file system
High level data management, mounted volumes, directories…
file-organization module
Translates logical file address (offset) to physical address (block on disc), file allocation, free space management
basic file system
Generic commands to the device driver to read/write blocks
I/O control
Device driver
devices
44

On-Disc File System Structures

What FS data are stored on the disc?

- **Secondary memory is subdivided into blocs and each I/O operation is performed in terms of blocs (to be studied later).**
- **Directory structure**
- **Information about each file**
  - **File control blocks/inodes (FCB)**
- **Per volume (partition)**
  - **Boot control block**
    - **Contains the code for starting the OS (can be empty)**
  - **Volume control block (also called the superblock)**
    - **# of blocks, size of blocks, # of free blocks, pointer to free blocks**



45

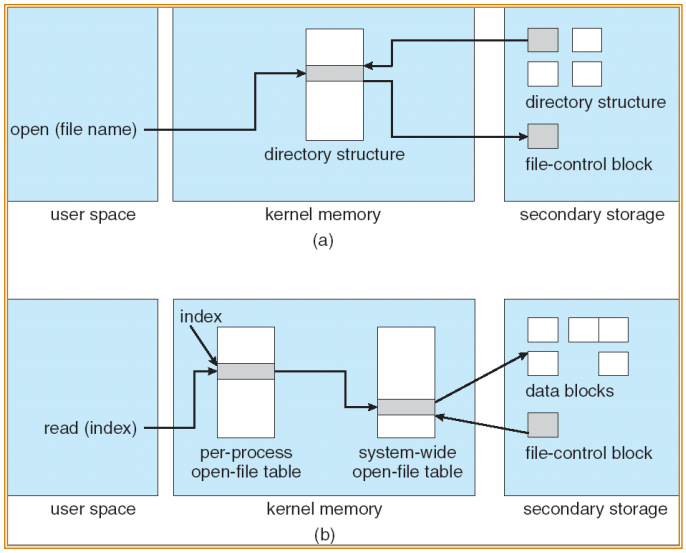
In-Memory File System Structures

What FS data are stored in main memory?

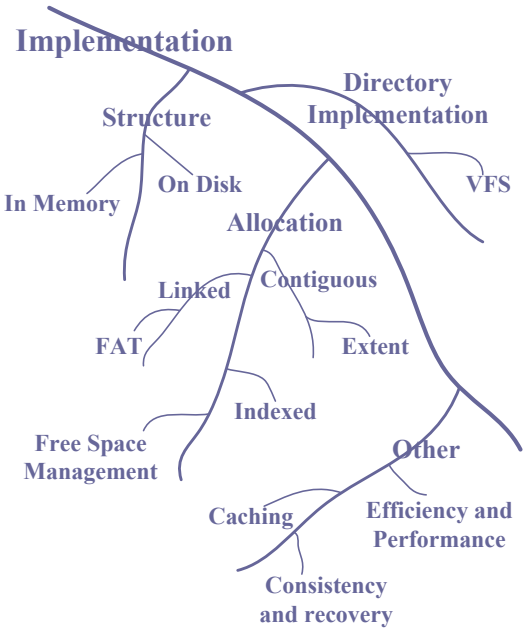
- **Information about each mounted volume**
- **Cache of the FS data, esp. directory structure**
- **System-wide open file table**
  - **FCB of each open file, file locks**
- **Per-process open file table**
  - **File pointer, access rights**

46

In-Memory File System Structures

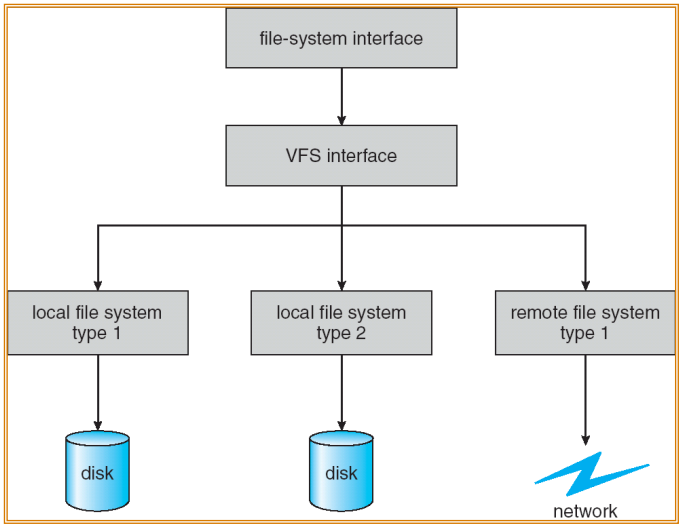


47



48

Virtual File System



49

Virtual File Systems

- **Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.**
- **The API is to the VFS interface, rather than any specific type of file system.**

Example:

- **In Linux there are four main object types:**
  - ***inode*** **(info about individual file)**
  - ***file*** **(open file)**
  - ***superblock*** **(entire file system of one volume)**
  - ***dentry*** **(individual directory entry)**
- **Each of them is required to implement the needed methods, but different devices/FSs can implement them differently**

50

Directory Implementation

What does a directory contain?

- The directory consists of a collection of entries that associate "names" to the files (and also subdirectories) represented by the FCB's
  - Must be organized in a tree-structure
  - Hard links – entries that reference FCB's
  - Soft links – entries that reference other entries.

How to store the directory entries for the files within one directory?

- Linear list **of file names with pointer to the data blocks.**
  - **simple to program**
  - **Problems: time-consuming to perform search**
- Hash Table **– linear list with hash data structure.**
  - **decreases directory search time**
  - **Problems: fixed size**

Note: **As each opening of a file involves many directory searches, they have to be really efficient.**

51

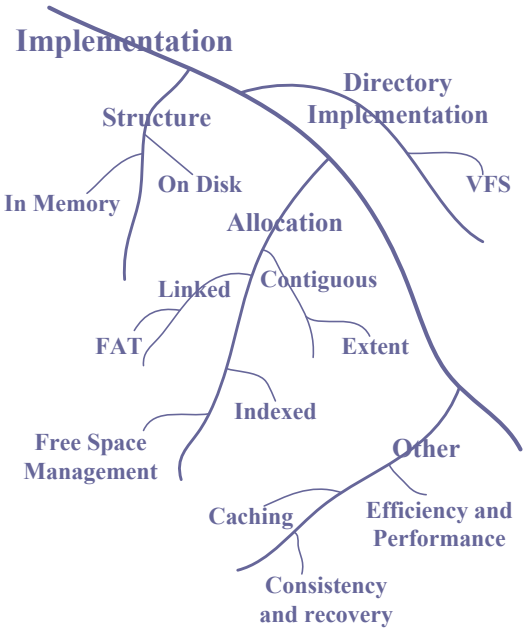
Directory Implementation

What should a directory entry contain?

- **file name (must be)**
- **+ all other info in the FCB (DOS)**
- **or pointer to the FCB/inode containing the info**
- **or symbolic link – pointer to another directory entry**
- **might have a flag indicating that this is a mounting point, and a pointer to the entry in the mount table with the info about the volume mounted here**

Note: **DOS used fixed length field (8+3 bytes) within the directory entry to store the filename+extension**

52



53

Allocation Methods

Before we start:

- **The disc is block-oriented device, basic unit of access is a sector of (almost always) 512 bytes**
- **The FS might for efficiency reasons group several sectors into blocks (e.g. of size 8kb) and use those as basic units**
- **Notion of clusters equivalent to the block.**

How to store a file on a disc?

- **Good news:**
  - **The ideas from memory management approaches are being reused**
- **Bad news:**
  - **There are differences**

So, how to allocate a space on disc for a file?

- **Contiguous allocation**
- **Linked allocation**
- **Indexed allocation**

54

Contiguous Allocation

Idea: **Each file occupies a set of contiguous blocks on the disk**

How much to allocate when a file is created?

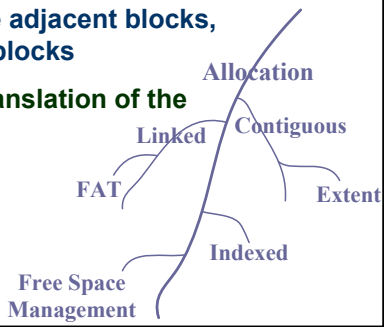
- **difficult to guess in general**
- **too little – difficult to grow, to much – wasting space**

How to grow a file?

- **into preallocated free space, free adjacent blocks, relocate into bigger hole of free blocks**

What information is needed to allow translation of the logical to physical address?

- **initial block and size, in the FCB**



55

Contiguous Allocation

Benefits:

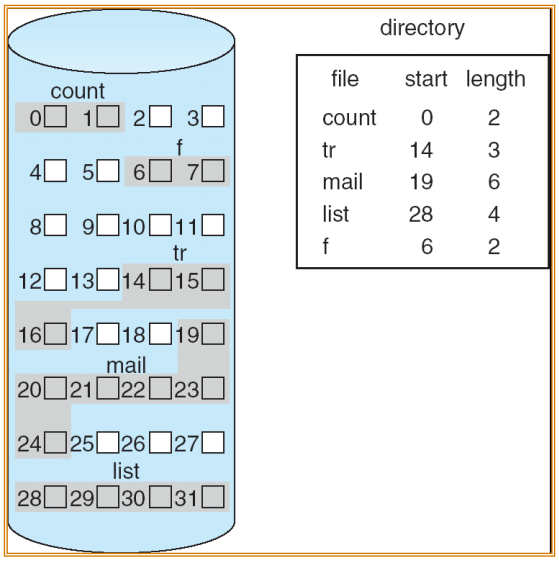
- **Simple to implement**
  - **ph. block = start block + logical address/ block size**
  - **offset = logical address modulo block size**
- **Efficient random access**
- **Good locality of reference**

Drawbacks:

- **Wasteful of space (fragmentation)**
- **Files cannot grow**
- **Cannot easily add to data in the middle of the file**
- **Needs compaction**

56

Contiguous Allocation Example



57

Extent-Based Systems

- **Many newer file systems (I.e. Veritas File System) use a modified contiguous allocation scheme**
- **Extent-based file systems allocate disk blocks in** extents
- **An** extent **is a contiguous set of blocks on disk**
  - **Extents are allocated to files**
  - **A file consists of one or more extents.**
- **Analogous to segments in memory management**

58

Linked Allocation

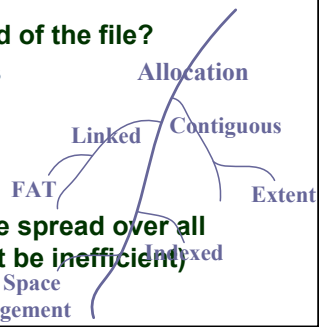
Idea: **Each block of the file contains a pointer pointing to the next block of that file. The directory entry of the file points to the first block of the file.**

Benefits:

- **simple**
- **no external fragmentation**
- **easy to grow files (a free block anywhere on the disc is linked to the end of the file)**

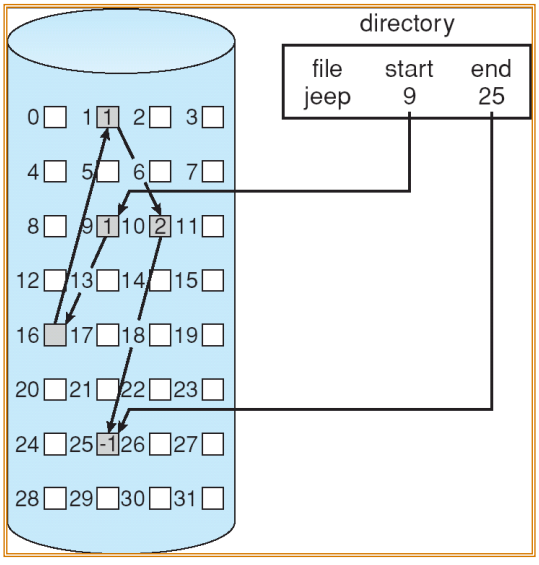
Drawbacks:

- **What happens if you want to reach the end of the file?**
  - **Has to read it all, to follow the pointers**
- **Very inefficient random access**
- **Wasted space for pointers**
  - **Using larger bloc size helps**
- **Poor locality of reference (the file might be spread over all disc, so even reading it sequentially might be inefficient)**



59

Linked Allocation



60

Linked Allocation + FAT

Idea:

- **Store the pointers to the next block separately in a special table (called File Allocation Table - FAT)**
- **One table for all files is sufficient**

Benefits:

- **The table or at least significant part of it might fit into memory**
- **Seeking within the file now means traversing the pointers in the memory, not on the disc**

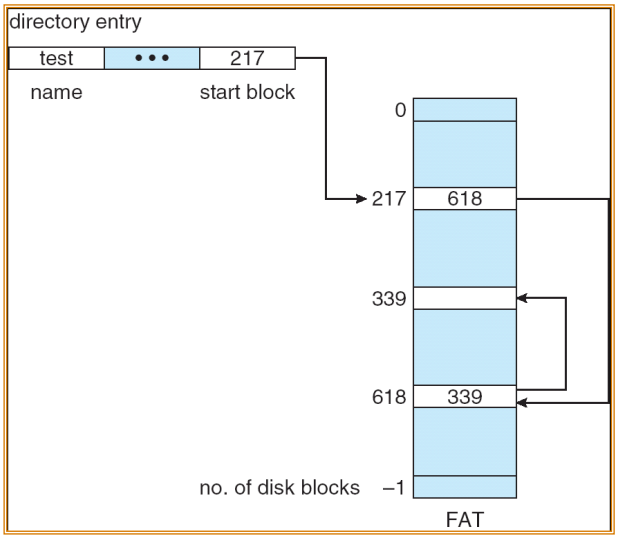
Note: **Used in DOS, OS/2, and other Windows OS's**

FAT size vs cluster size:

- **# of blocks on disc = FAT entries * cluster size**
- **With large discs, FAT16 (2^{16} entries) must use large clusters**
  - **Lots of internal fragmentation with small files**
- **FAT32 solves this problem (but the FAT itself is quite beefy)**

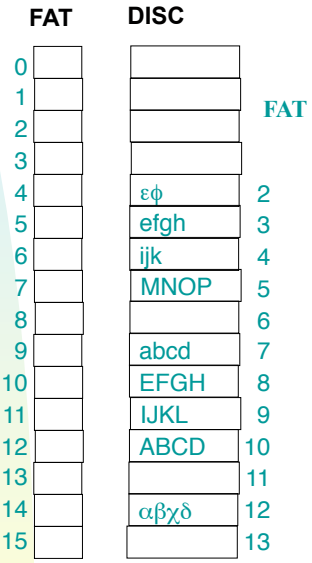
61

File-Allocation Table



62

Exercise



How does the FAT table look, assuming each block can hold only 4 entries and only the following files are in the system?

File A = abcdefghijk

File B = ABCDEFGHIJKLMNOP

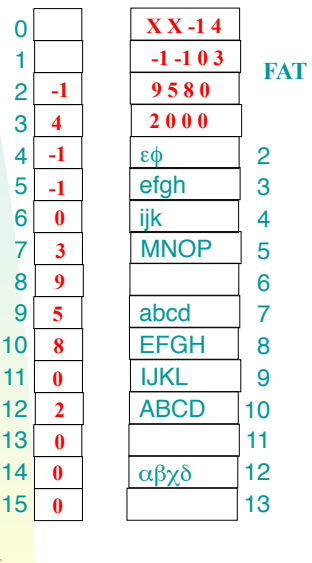
File C = αβχδεφ

Directory entries:
- File A: start block: length:
- File B: start block: length:
- File C: start block: length:

Assume the FAT is stored in blocks 0..3
Block numbers start at 2 in the Data Region of the file system

63

Exercise



How does the FAT table look, assuming each block can hold only 4 entries and only the following files are in the system?

File A = abcdefghijk

File B = ABCDEFGHIJKLMNOP

File C = αβχδεφ

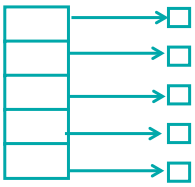
Directory entries:
- File A: start block: 7 length: 11
- File B: start block: 10 length: 16
- File C: start block: 12 length: 6

Assume the FAT is stored in blocks 0..3
Block numbers start at 2 in the Data Region of the file system
0 indicates a free block

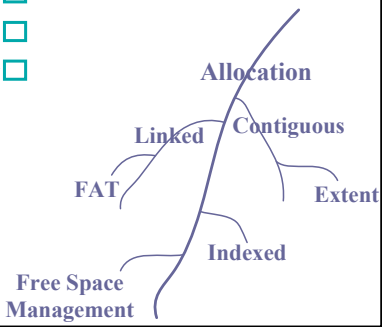
64

Indexed allocation – similar to paging

- All pointers to blocs (bloc numbers) are placed in a table (index bloc).



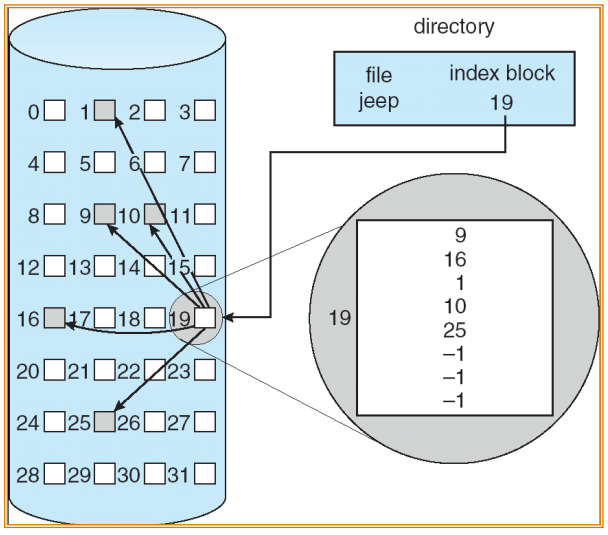
index table



65

Indexed Allocation

Idea: **Use an *index block* to store all the block pointers of the file**



66

Indexed Allocation (Cont.)

Benefits:

- **easy random access**
- **no external fragmentation**

Drawbacks:

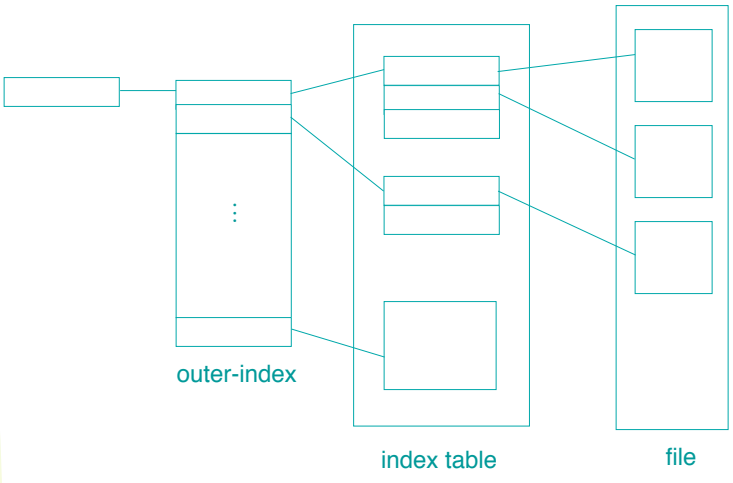
- **need index table blocks**
- **poor locality of reference**

What if the file needs more blocks than the index block can reference?

- **i.e. 512 byte block with 256 entries, 256x512 ⇒ maximum files size of 128 Kbytes**
- **Implement Index Table as**
  - **linked list of blocks**
    - **poor for random access**
  - **hierarchical index tables**
    - **too much overhead for small files**
  - **combined scheme**

67

Hierarchical Indexed Allocation



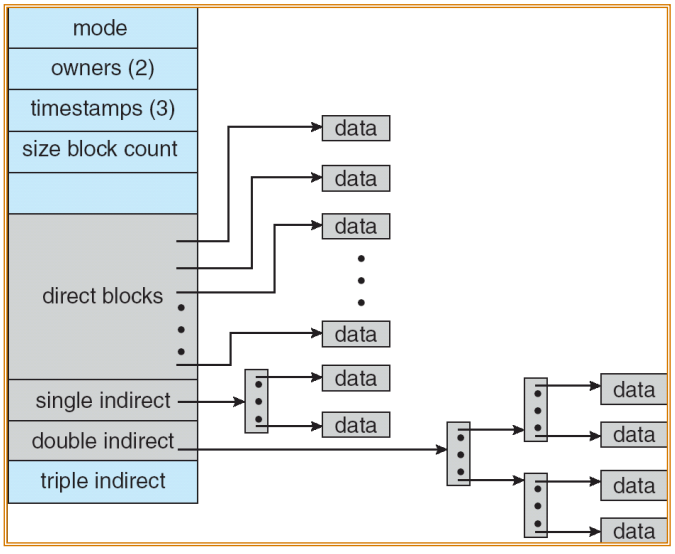
68

Indexed Allocation Used in Unix

- Each directory entry contains the address of an index node called inode
- The inode contains info about the type of file, its restrictions,…, and 13 addresses (block numbers)
  - The first 10 addresses point directly on the first 10 blocks of the file
  - If the file needs more blocks, then…
    - The11th address (single indirect) points to a block containing pointers to up to 256 additional blocks
  - If the file needs more blocks, then…
    - The 12th address (double indirect) points to a block containing pointers to 256 blocks of additional pointers
  - If the file needs more blocks, then…
    - Triple indirection (see next figure)

69

Combined Scheme – UNIX inode



70

Exercise

DISC

0	
1	
2	
3	
4	εφ
5	efgh
6	ijk
7	MNOP
8	
9	abcd
10	EFGH
11	IJKL
12	ABCD
13	
14	αβχδ
15	

Same exercise, but with simple indexed allocation.

File A = abcd efgh ijk

File B = ABCD EFGH IJKL MNOP

File C = αβχδ εφ

How do the index tables of files A,B,C look? (We have to also choose where on disc are those tables located.)

File A index table: at block:
File B index table: at block:
File C index table: at block:

71

Exercise

DISC

0	9 5 6 -1
1	12 10 11 7
2	14 4 -1 -1
3	
4	εφ
5	efgh
6	ijk
7	MNOP
8	
9	abcd
10	EFGH
11	IJKL
12	ABCD
13	
14	αβχδ
15	

Same exercise, but with simple indexed allocation.

File A = abcd efgh ijk

File B = ABCD EFGH IJKL MNOP

File C = αβχδ εφ

How do the index tables of files A,B,C look? (We have to also choose where on disc are those tables located.)

File A index table: 9 5 6 -1 at block: 0
File B index table: 12 10 11 7 at block: 1
File C index table: 14 4 -1 -1 at block: 2

72

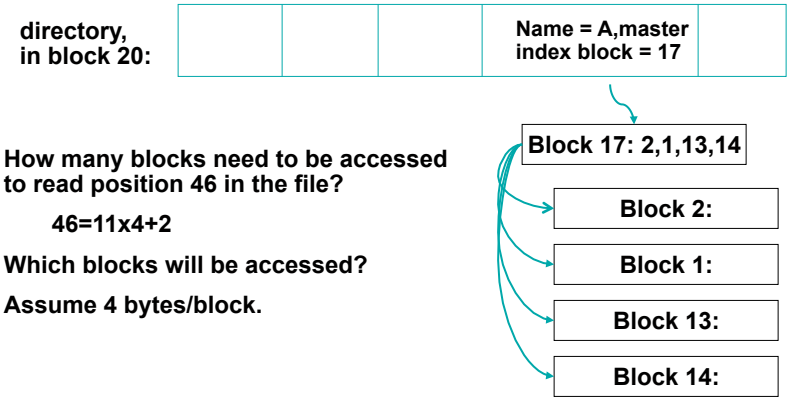
Exercise 2 – Allocating Larger Files

Assume each index block can hold at most 4 index entries.

The blocks containing file A: 4..12, 62..65, 35 (14 blocks)

How is file A allocated?

Hierarchical allocation:



How many blocks need to be accessed to read position 46 in the file?

46=11x4+2

Which blocks will be accessed?

Assume 4 bytes/block.

73

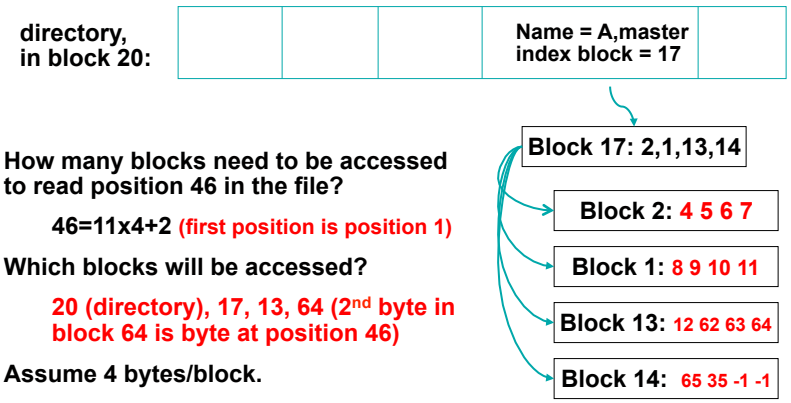
Exercise 2 – Allocating Larger Files

Assume each index block can hold at most 4 index entries.

The blocks containing file A: 4..12, 62..65, 35 (14 blocks)

How is file A allocated?

Hierarchical allocation:



How many blocks need to be accessed to read position 46 in the file?

46=11x4+2 (first position is position 1)

Which blocks will be accessed?

20 (directory), 17, 13, 64 (2nd byte in block 64 is byte at position 46)

Assume 4 bytes/block.

74

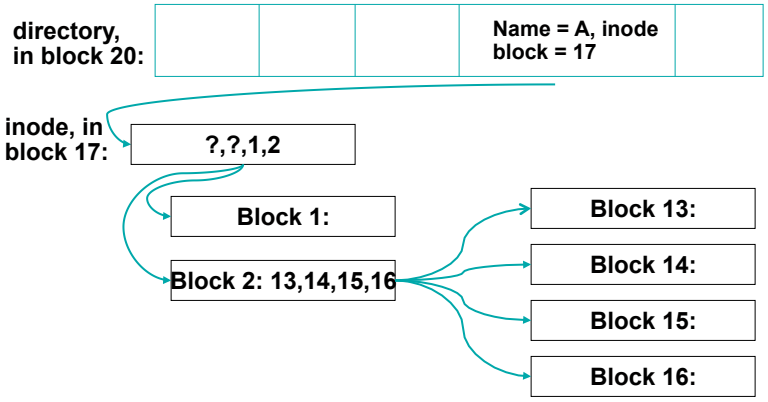
Exercise 2 – Representing Larger Files

Assume each index block can hold at most 4 index entries.

The blocks containing file A: 4..12, 62..65, 35 (14 blocks)

How is file A allocated?

Unix like allocation:



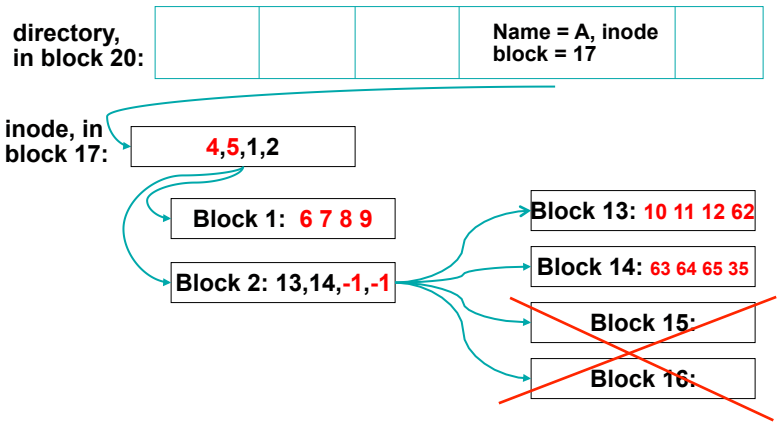
Exercise 2 – Representing Larger Files

Assume each index block can hold at most 4 index entries.

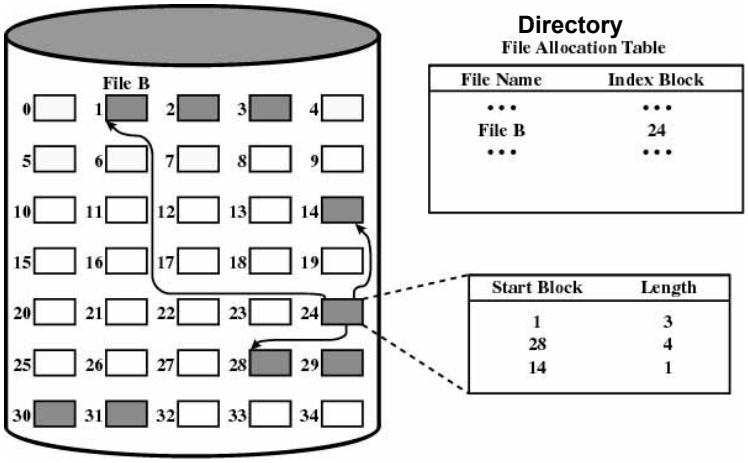
The blocks containing file A: 4..12, 62..65, 35 (14 blocks)

How is file A allocated?

Unix like allocation:



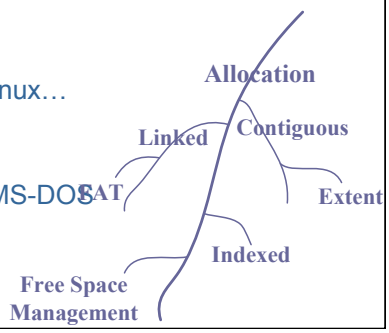
Indexed Allocation with variable-size portions



77

Free-Space Management

- When creating/growing file, we need free blocs.
- How to find free blocs in fast and efficient manner?
  - Have a data structure storing them:
    - Bit Vector (Map)
      - Ex: Windows 2000, MacOS, Linux…
    - Linked List
      - Ex: Unix SVR4, Windows 9x, MS-DOS



78

Bit Vector (or Bit Map)

- Each existing block is represented by a bit
  - The bit is 1 if the block is free
  - The bit is 0 if the block is allocated to a file
  - The number of bits of the vector = number of existing blocks
- Example of Bit Vector where blocks 3, 4, 5, 9, 10, 15, 16 are free:
  - 00011100011000011…
- If the whole bit vector fits into memory, we can reasonably fast locate the first 1, representing a free block
  - First find the first non-zero byte
  - Then the first 1 on that byte
    - Most processors provide HW support for that

79

Bit Vector (or Bit Map)

Can we expect to have the whole bit vector in memory?

Let's see:

- 160GB harddrive, 512 byte bloc = 320 Mbits = 40Mbytes
  - Ouch!
- With 16kb bloc, its still over 1Mbyte
- Sequential search in the bitmap for the first non-0 word will not do any more, need smarter algorithms/data structures.

Can the bitmap of the free blocks be computed from the directory/file mapping information?

- Yes, but we need to traverse the whole structure, whatever was not used is free.
- So, is the bit vector stored only in memory or is it stored also on HD?
  - Well, do you want to scan the whole FS on each boot?

80

Linked List

Idea:

- **Each free block points to the next free block**
- **A designated place on the disc stores the first free block (also cached in memory)**

Bad news:

- **Scanning the free blocks is very costly (lots of disc I/O)**

Good news:

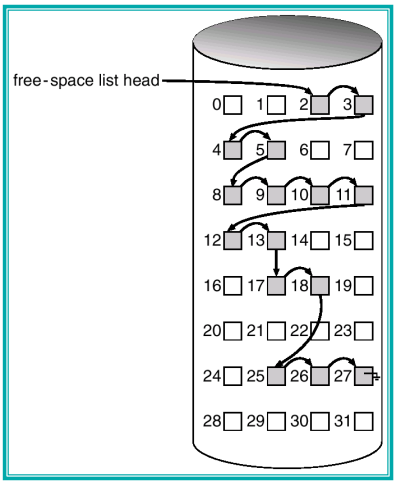
- **Usually we just want a free block – we get it immediately from the head of the list, just have to move the head**

Problem:

- **Want several blocks**

Solution:

- **Have a list of contiguous blocks of free blocks: head -> (2,4) -> (8,6) ->(17,2)->(25,3)**



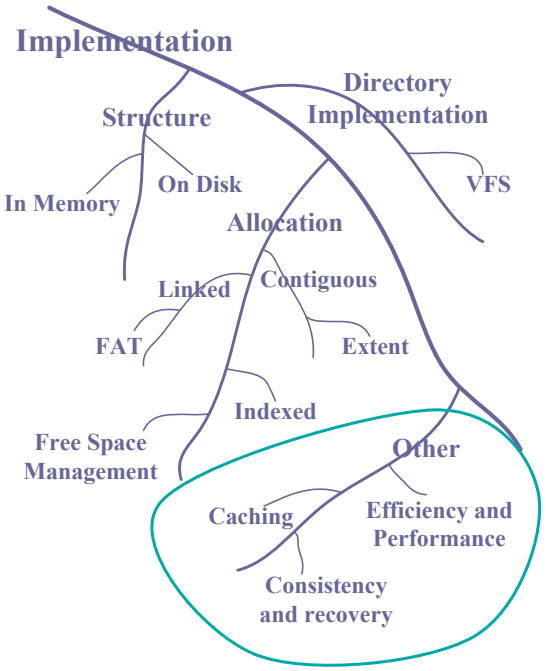
81

FAT

What structure do we need for representing free blocks if we use FAT?

- **The FAT itself works a bit as a bit map:**
  - **An entry containing 0 means that this block is empty.**

82



83

Efficiency and Performance

How much of the disc's memory is wasted/unusable?

- Why would be some disc space wasted?
  - Internal fragmentation -> want smaller clusters
  - FAT/inodes/bitmaps -> want larger clusters
  - Fixed width directory entries -> small = too limiting, large = waste
  - Overhead for supporting variable-width fields (e.g. for file names)
- What happens if all meta-data information is at the beginning of the disc?
  - Accessing/creating file involves lots of disc head movement
  - You want to have the meta-data (inode, directory entry) about the file close to it
- What kind of data to keep?
  - Writing into file means updating not only the file itself, but also meta-data: last modification time
  - Should we maintain last access time?
    - Also reading would involve write into meta-data

84

Efficiency and Performance

What is a universal solution for slow memory? (Note that the disc is secondary memory)

- **Caching**

Ideas:

- **Dedicate part of the main memory for caching frequently used blocks (exploits temporal locality)**
- **Can also use *read-ahead*, betting that the file is accessed sequentially (exploits spatial locality)**
- **Might use *free-behind* (releasing the just read bloc as soon as the next bloc is requested) – again betting on sequential access)**
- **Unified Virtual Memory, unified buffer cache**

Other speedup techniques:

- **Synchronous vs asynchronous writes**

85

Implementing Caching

Caching for what?

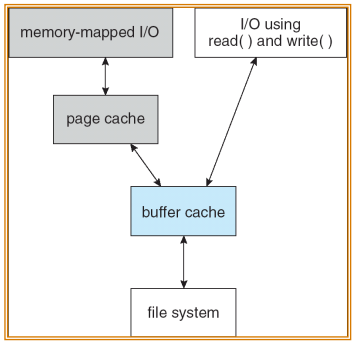
- **open/read/write accesses (buffer cache)**
- **memory-mapped files (page cache)**

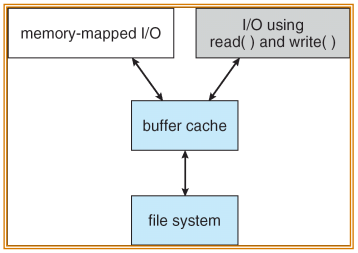
Can have separate caches for each of them

- **but leads to double caching and inefficiencies**

Better to have a unified cache for both disk blocks and pages

- **Use the page cache for both - unified virtual memory**
- **Unified buffer cache**





86

Consistency and Recovery

When should we write the meta-data to the disc?

- Whenever they change (i.e. if file is modified, update the inode with the new modification time)
  - Slow (additional disc access, possibly involving head movement)
  - Wasteful (lots of writes, do it only once when the write burst finishes)
- On system shutdown
  - Minimizes disc traffic
  - But what if abnormal (power off, crash) termination?
    - Inconsistent data on the disc

87

Consistency and Recovery

What to do?

- Write the meta-data less frequently
  - Always writing on each read/write is unacceptable from performance point of view
  - Still might need to always write the file creation/rename/growth
- But that means we must be able to deal with the inconsistencies

Consistency checking

- compares the data in the directory structure/FAT/other meta-data structures with the data blocks on the disc and tries to detect and fix inconsistencies
- *fsck* in UNIX, *chkdsk* in DOS

Making back-ups is always good idea

- Not always 100% possible to recover the data
- Especially if HW failure

88

Example: Free-Space Management

Special care must be taken with the order of operations

- First write the meta-data change, then perform the change, then update the in-memory tables

Consider free-space management using bitmap

- We want to have the bitmap in the memory for fast searches and updates
  - Cannot allow for block[*i*] to have a situation where bit[*i*] = 1 in memory and bit[*i*] = 0 on disk
    - If crash comes, the allocated block will be lost
  - Solution:
    - Set bit[*i*] = 1 in disk
    - Allocate block[*i*]
    - Set bit[*i*] = 1 in memory

89

Log Structured File Systems

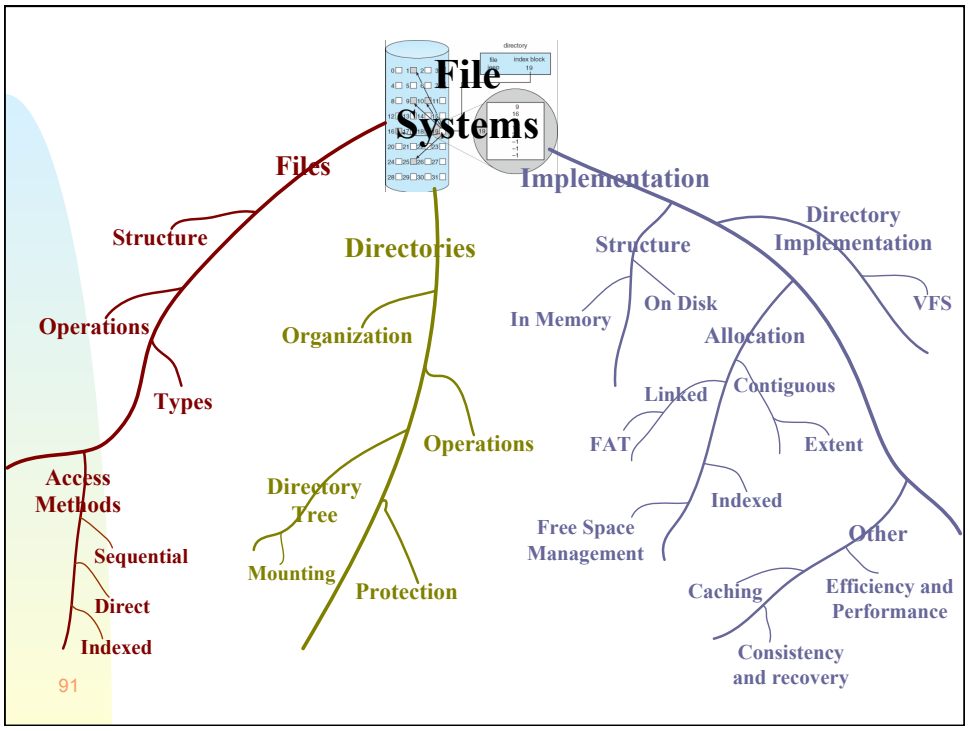
What do we want to achieve with log structured FS?

- To implement crash-tolerant file system in an efficient way.

Idea:

- *Log structured* (or journaling) file systems record each update to the file system as a transaction
- All transactions are written to a log on the disc
  - Efficient write, as the log file is accessed sequentially
  - A transaction is considered committed once it is written to the log
  - However, the file system data on the disc may not yet be updated
- The transactions in the log are asynchronously written to the file system
  - When the file system is modified, the transaction is removed from the log
- What to do after a system crash?
  - Read the log and perform all transactions remaining there

90


File Systems
Files
Structure
Operations
Types
Access Methods
Sequential
Direct
Indexed
Directories
Organization
Operations
Directory Tree
Mounting
Protection
Implementation
Structure
In Memory
On Disk
Allocation
Linked
Contiguous
FAT
Extent
Indexed
Free Space Management
Directory Implementation
VFS
Other
Caching
Efficiency and Performance
Consistency and recovery
91