CS 422/522 Design & Implementation of Operating Systems

Lectures 16-17: Files and Directories

Zhong Shao
Dept. of Computer Science
Yale University

1

The big picture

- Lectures before the fall break:
  - Management of CPU & concurrency
  - Management of main memory & virtual memory
- Current topics --- "Management of I/O devices"
  - Last week: I/O devices & device drivers
  - Last week: storage devices
  - This week: file systems
    - File system structure
    - Naming and directories
    - Efficiency and performance
    - Reliability and protection

2

This lecture

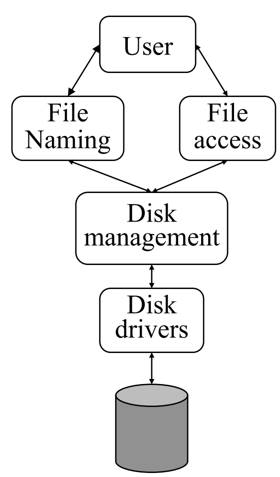
- Implementing file system abstraction

Physical Reality	File System Abstraction
block oriented	byte oriented
physical sector #'s	named files
no protection	users protected from each other
data might be corrupted if machine crashes	robust to machine failures

3

File system components

- Disk management
  - Arrange collection of disk blocks into files
- Naming
  - User gives file name, not track or sector number, to locate data
- Security / protection
  - Keep information secure
- Reliability/durability
  - When system crashes, lose stuff in memory, but want files to be durable



4

User vs. system view of a file

- User's view
  - Durable data structures
- System's view (system call interface)
  - Collection of bytes (Unix)
- System's view (inside OS):
  - Collection of blocks
  - A block is a logical transfer unit, while a sector is the physical transfer unit. Block size >= sector size.

5

File structure

- None - sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex structures
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters.
- Who decides:
  - Operating system
  - Program

6

File attributes

- **Name** – only information kept in human-readable form.
- **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.
- Information about files are kept in the directory structure, which is maintained on the disk.

7

File operations

- create
- write
- read
- reposition within file – file seek
- delete
- truncate
- open(F_i) – search the directory structure on disk for entry F_i, and move the content of entry to memory.
- close (F_i) – move the content of entry F_i in memory to directory structure on disk.

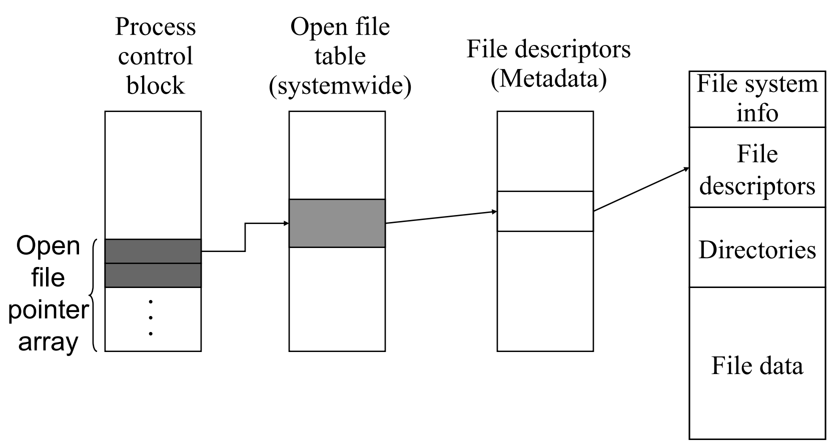
8

File types – name, extension

File Type	Usual extension	Function
Executable	exe, com, bin or none	ready-to-run machine-language program
Object	obj, o	complied, machine language, not linked
Source code	c, p, pas, 177, 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, rrf, etc.	various word-processor formats
Library	lib, a	libraries of routines
Print or view	ps, dvi, gif	ASCII or binary file
Archive	arc, zip, tar	related files grouped into one file, sometimes compressed.

9

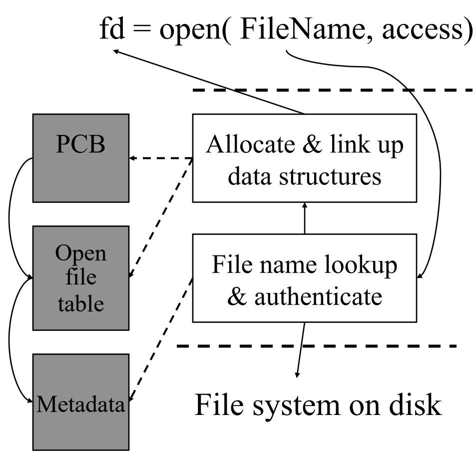
Data structures for a typical file system



10

Open a file

- File name lookup and authenticate
- Copy the file descriptors into the in-memory data structure, if it is not in yet
- Create an entry in the open file table (system wide) if there isn't one
- Create an entry in PCB
- Link up the data structures
- Return a pointer to user



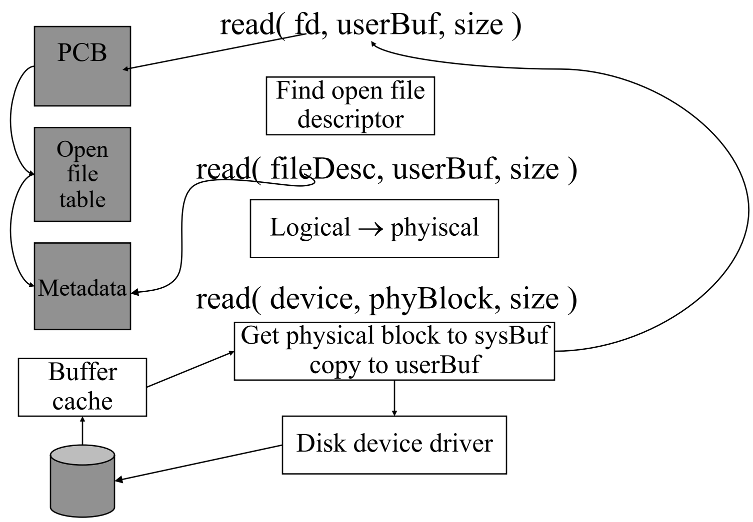
11

Translating from user to system view

- What happens if user wants to read 10 bytes from a file starting at byte 2?
  - seek byte 2
  - fetch the block
  - read 10 bytes
- What happens if user wants to write 10 bytes to a file starting at byte 2?
  - seek byte 2
  - fetch the block
  - write 10 bytes
  - write out the block
- Everything inside file system is in whole size blocks
  - Even getc and putc buffers 4096 bytes
  - From now on, file is collection of blocks.

12

Read a block



13

File system design constraints

- For small files:
  - Small blocks for storage efficiency
  - Files used together should be stored together
- For large files:
  - Contiguous allocation for sequential access
  - Efficient lookup for random access
- May not know at file creation
  - Whether file will become small or large

14

File system design

- Data structures
  - Directories: file name -> file metadata
    - Store directories as files
  - File metadata: how to find file data blocks
  - Free map: list of free disk blocks
- How do we organize these data structures?
  - Device has non-uniform performance

15

Design challenges

- Index structure
  - How do we locate the blocks of a file?
- Index granularity
  - What block size do we use?
- Free space
  - How do we find unused blocks on disk?
- Locality
  - How do we preserve spatial locality?
- Reliability
  - What if machine crashes in middle of a file system op?

16

File system design options

	FAT	FFS	NTFS	ZFS
Index structure	Linked list	Tree (fixed, assym)	Tree (dynamic)	Tree (COW, dynamic)
granularity	block	block	extent	block
free space allocation	FAT array	Bitmap (fixed location)	Bitmap (file)	Space map (log-structured)
Locality	defragmenta tion	Block groups + reserve space	Extents Best fit defrag	Write-anywhere Block-groups

17

Named data in a file system

18

A disk layout for a file system

Boot block	Super block	File descriptors (i-node in Unix)	File data blocks

- Superblock defines a file system
  - size of the file system
  - size of the file descriptor area
  - free list pointer, or pointer to bitmap
  - location of the file descriptor of the root directory
  - other meta-data such as permission and various times

19

File usage patterns

- How do users access files?
  - Sequential: bytes read in order
  - Random: read/write element out of middle of arrays
  - Content-based access: find me next byte starting with "CS422"
- How are files used?
  - Most files are small
  - Large files use up most of the disk space
  - Large files account for most of the bytes transferred
- Bad news
  - Need everything to be efficient

20

Data structures for disk management

- A file header for each file (part of the file meta-data)
  - Disk sectors associated with each file
- A data structure to represent free space on disk
  - Bit map
    - 1 bit per block (sector)
    - blocks numbered in cylinder-major order, why?
  - Linked list
  - Others?
- How much space does a bitmap need for a 4G disk?

21

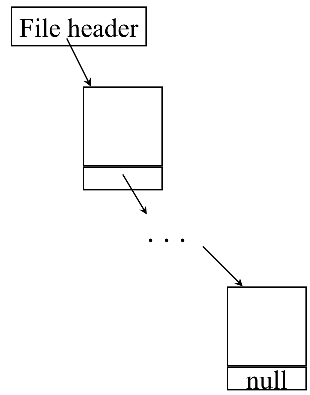
Contiguous allocation

- Request in advance for the size of the file
- Search bit map or linked list to locate a space
- File header
  - first sector in file
  - number of sectors
- Pros
  - Fast sequential access
  - Easy random access
- Cons
  - External fragmentation
  - Hard to grow files

22

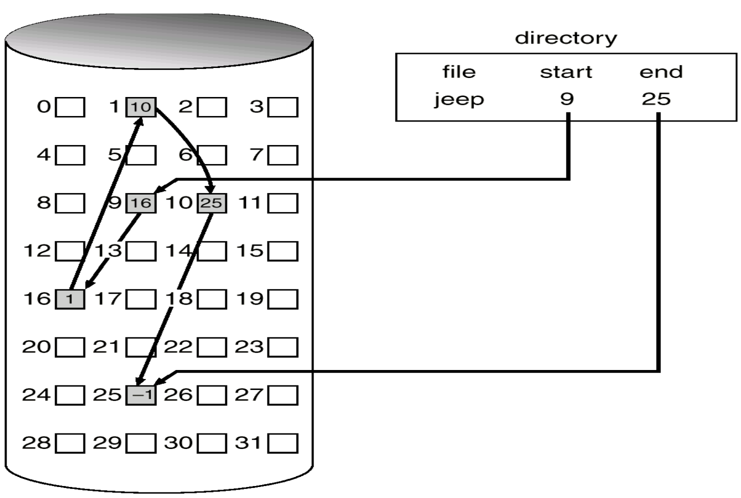
Linked files

- File header points to 1st block on disk
- A block points to the next
- Pros
  - Can grow files dynamically
  - Free list is similar to a file
  - No waste of space
- Cons
  - random access: horrible
  - unreliable: losing a block means losing the rest



23

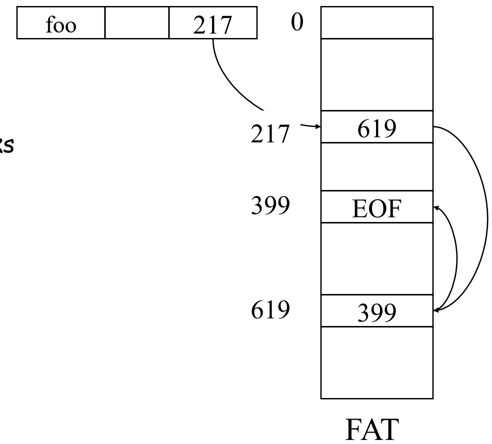
Linked files (cont'd)



24

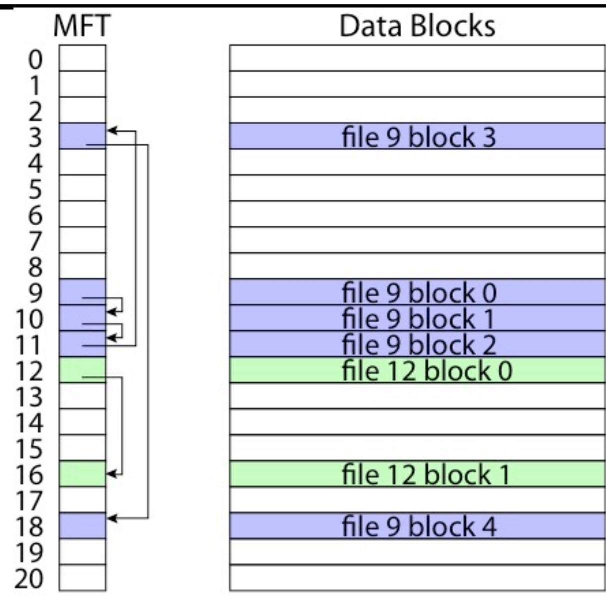
File Allocation Table (FAT)

- Approach (used by MSDOS)
  - A section of disk for each partition is reserved
  - One entry for each block
  - A file is a linked list of blocks
  - A directory entry points to the 1st block of the file



25

FAT



26

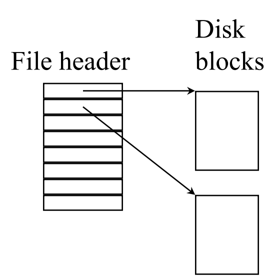
FAT

- Pros:
  - Easy to find free block
  - Easy to append to a file
  - Easy to delete a file
- Cons:
  - Small file access is slow
  - Random access is very slow
  - Fragmentation
    - * File blocks for a given file may be scattered
    - * Files in the same directory may be scattered
    - * Problem becomes worse as disk fills

27

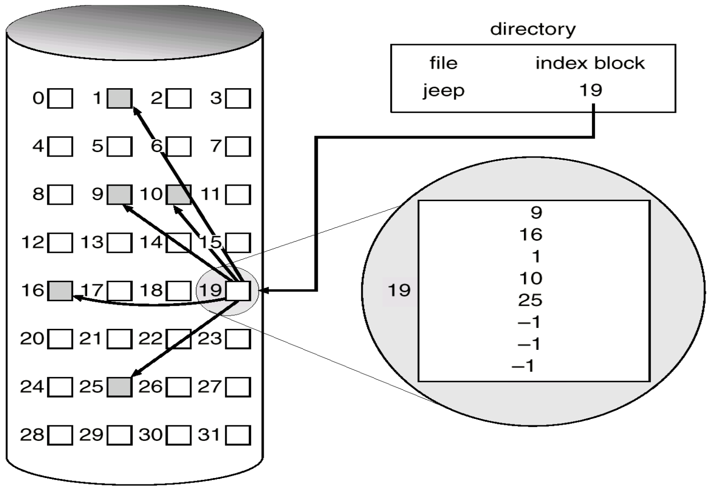
Single-level indexed files

- A user declares max size
- A file header holds an array of pointers to point to disk blocks
- Pros
  - Can grow up to a limit
  - Random access is fast
  - No external fragmentation
- Cons
  - Clumsy to grow beyond the limit
  - Still lots of seeks



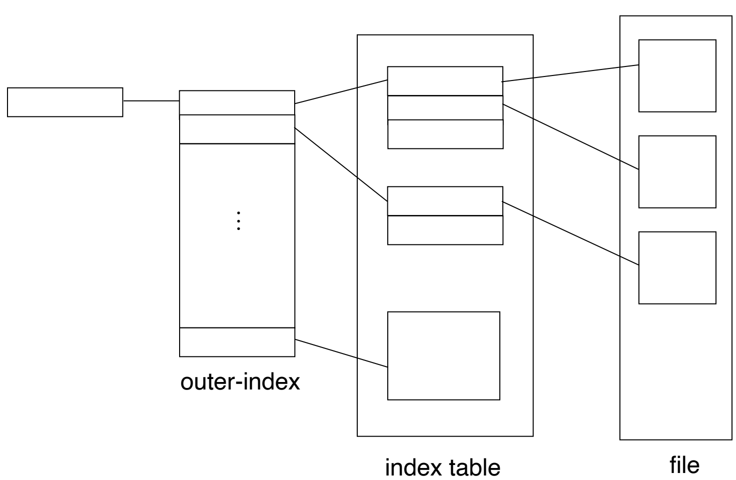
28

Single-level indexed files (cont'd)



29

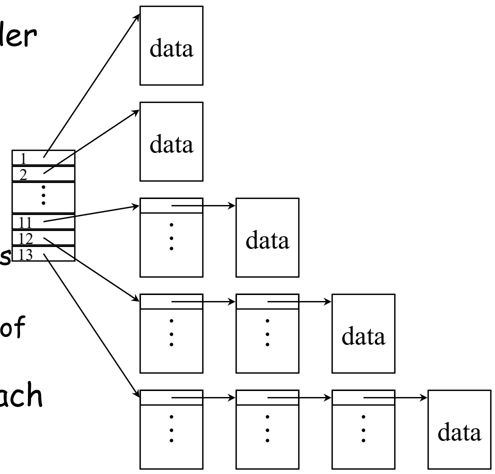
Multi-level indexed files



30

Combined scheme (Unix 4.1, 1 KB / block)

- 13 Pointers in a header
  - 10 direct pointers
  - 11: 1-level indirect
  - 12: 2-level indirect
  - 13: 3-level indirect
- Pros & Cons
  - In favor of small files
  - Can grow
  - Limit is 16G and lots of seek
- What happens to reach block 23, 5, 340?



31

Berkeley UNIX FFS (Fast File System)

- inode table
  - Analogous to FAT table
- inode
  - Metadata
    - File owner, access permissions, access times, …
  - Set of 12 data pointers
  - With 4KB blocks => max size of 48KB files

32

FFS inode

- Metadata
  - File owner, access permissions, access times, …
- Set of 12 data pointers
  - With 4KB blocks => max size of 48KB files
- Indirect block pointer
  - pointer to disk block of data pointers
- Indirect block: 1K data blocks => 4MB (+48KB)

33

FFS inode

- Metadata
  - File owner, access permissions, access times, …
- Set of 12 data pointers
  - With 4KB blocks => max size of 48KB
- Indirect block pointer
  - pointer to disk block of data pointers
  - 4KB block size => 1K data blocks => 4MB
- Doubly indirect block pointer
  - Doubly indirect block => 1K indirect blocks
  - 4GB (+ 4MB + 48KB)

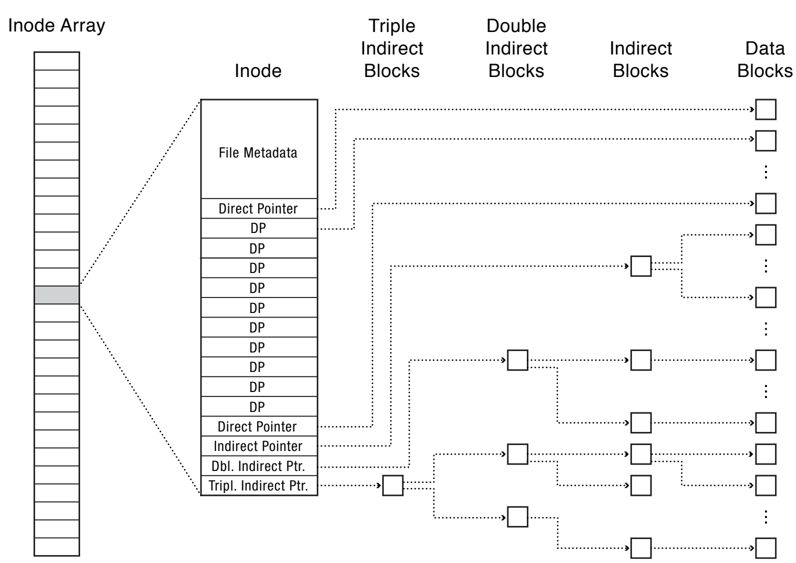
34

FFS inode

- Metadata
  - File owner, access permissions, access times, …
- Set of 12 data pointers
  - With 4KB blocks => max size of 48KB
- Indirect block pointer
  - pointer to disk block of data pointers
  - 4KB block size => 1K data blocks => 4MB
- Doubly indirect block pointer
  - Doubly indirect block => 1K indirect blocks
  - 4GB (+ 4MB + 48KB)
- Triply indirect block pointer
  - Triply indirect block => 1K doubly indirect blocks
  - 4TB (+ 4GB + 4MB + 48KB)

35

FFS inode (cont'd)



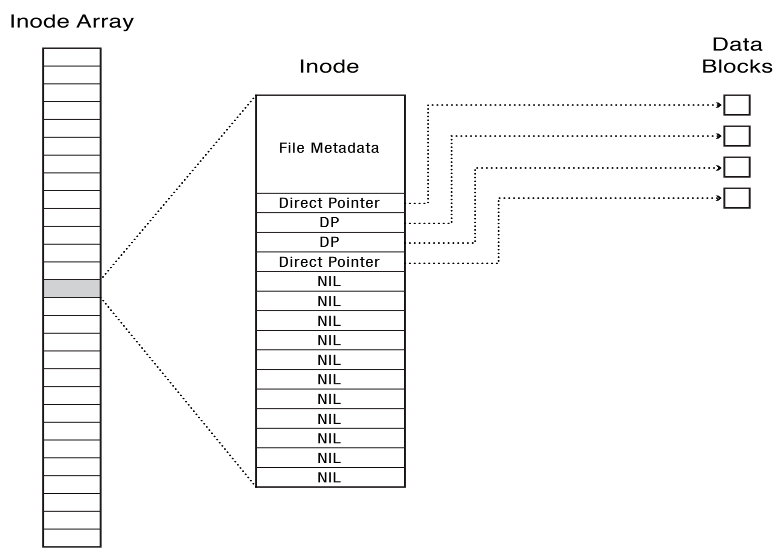
36

FFS asymmetric tree

- Small files: shallow tree
  - Efficient storage for small files
- Large files: deep tree
  - Efficient lookup for random access in large files
- Sparse files: only fill pointers if needed

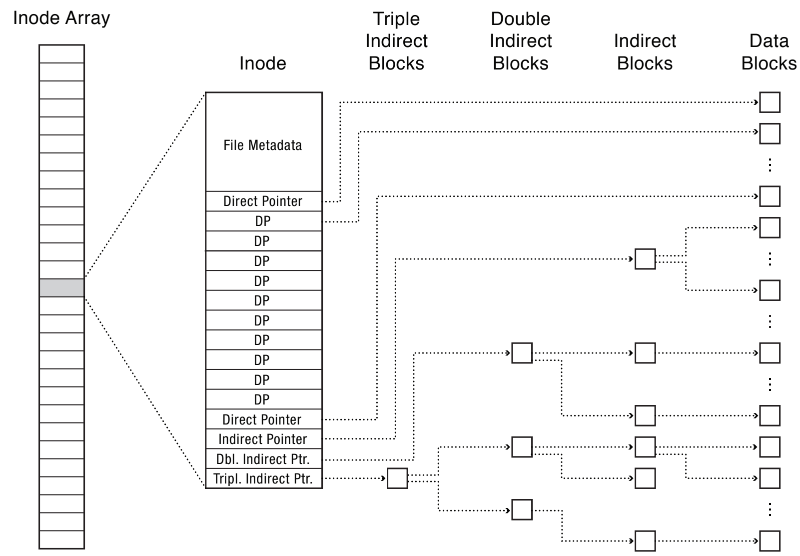
37

FFS small files: shallow tree



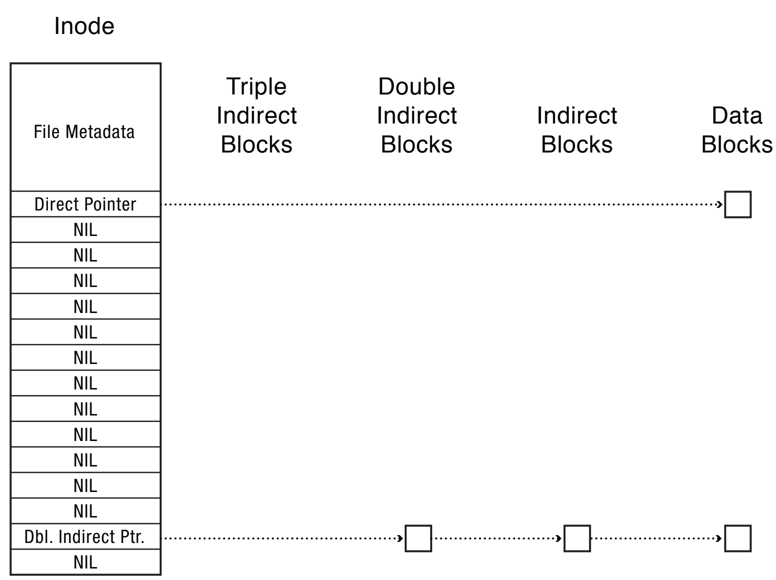
38

FFS large files: deep tree



39

FFS sparse files: only fill pointers if needed



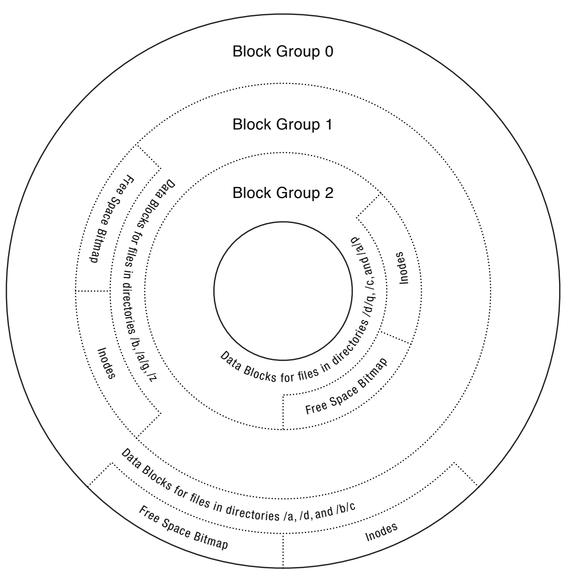
40

FFS locality

- Block group allocation
  - Block group is a set of nearby cylinders
  - Files in same directory located in same group
  - Subdirectories located in different block groups
- inode table spread throughout disk
  - inodes, bitmap near file blocks
- First fit allocation
  - Small files fragmented, large files contiguous

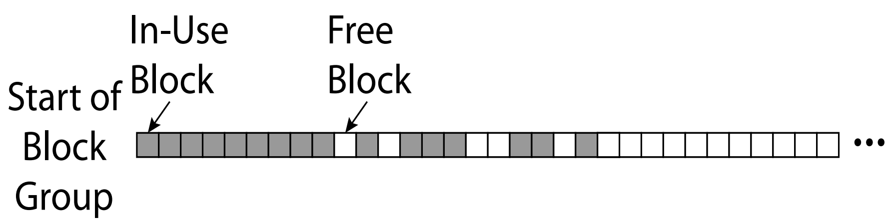
41

FFS block groups for better locality



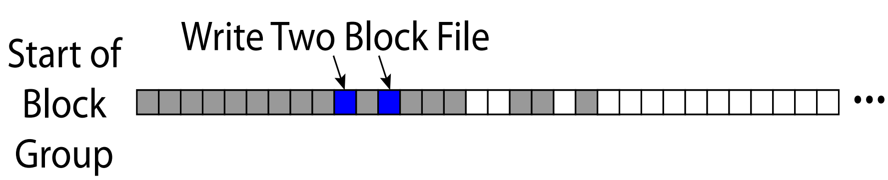
42

FFS first fit block allocation



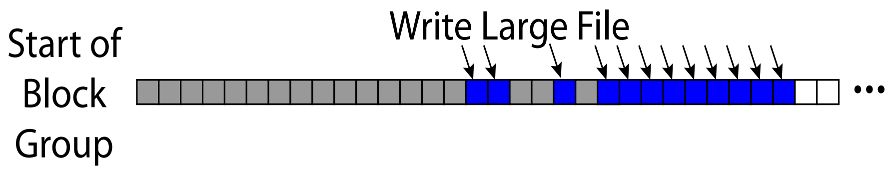
43

FFS first fit block allocation



44

FFS first fit block allocation



45

FFS

- Pros
  - Efficient storage for both small and large files
  - Locality for both small and large files
  - Locality for metadata and data
- Cons
  - Inefficient for tiny files (a 1 byte file requires both an inode and a data block)
  - Inefficient encoding when file is mostly contiguous on disk (no equivalent to super pages)
  - Need to reserve 10-20% of free space to prevent fragmentation

46

File header storage

- Where is file header stored on disk?
  - In (early) Unix & DOS FAT file sys, special array in outermost cylinders
- Unix refers to file by index into array --- tells it where to find the file header
  - "i-node" --- file header; "i-number" --- index into the array
- Unix file header organization (seems strange):
  - header not anywhere near the data blocks. To read a small file, seek to get header, seek back to data.
  - fixed size, set when disk is formatted.

47

File header storage (cont'd)

- Why not put headers near data?
  - Reliability: whatever happens to the disk, you can find all of the files.
  - Unix BSD 4.2 puts portion of the file header array on each cylinder. For small directories, can fit all data, file headers, etc. in same cylinder → no seeks!
  - File headers are much smaller than a whole block (a few hundred bytes), so multiple file headers fetched from disk at same time.
- Q: do you ever look at a file header without reading the file ?
  - Yes! Reading the header is 4 times more common than reading the file (e.g., ls, make).

48

Naming and directories

- Options
  - Use index (ask users specify inode number). Easier for system, not as easy for users.
  - Text name (need to map to index)
  - Icon (need to map to index; or map to name then to index)
- Directories
  - Directory map name to file index (where to find file header)
  - Directory is just a table of file name, file index pairs.
  - Each directory is stored as a file, containing a (name, index) pair.
  - Only OS permitted to modify directory

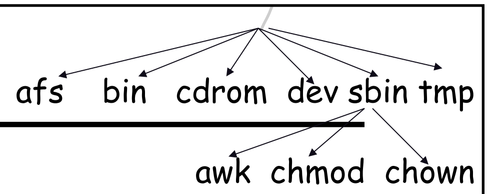
49

Directory structure

- Approach 1: have a single directory for entire system.
  - put directory at known location on disk
  - directory contains <name, index> pairs
  - if one user uses a name, no one else can
  - many older personal computers work this way.
- Approach 2: have a single directory for each user
  - still clumsy. And ls on 10,000 files is a real pain
  - many older mathematicians work this way.
- Approach 3: hierarchical name spaces
  - allow directory to map names to files or other dirs
  - file system forms a tree (or graph, if links allowed)
  - large name spaces tend to be hierarchical (ip addresses, domain names, scoping in programming languages, etc.)

50

Hierarchical Unix



- Used since CTSS (1960s)
  - Unix picked up and used really nicely.
- Directories stored on disk just like regular files
  - inode contains special flag bit set
  - user's can read just like any other file
  - only special programs can write
  - file pointed to by the index may be another directory
  - makes FS into hierarchical tree (what needed to make a DAG?)

<name, inode#>
<afs, 1021>
<tmp, 1020>
<bin, 1022>
<cdrom, 4123>
<dev, 1001>
<sbin, 1011>
...

- Simple. Plus speeding up file ops = speeding up dir ops!

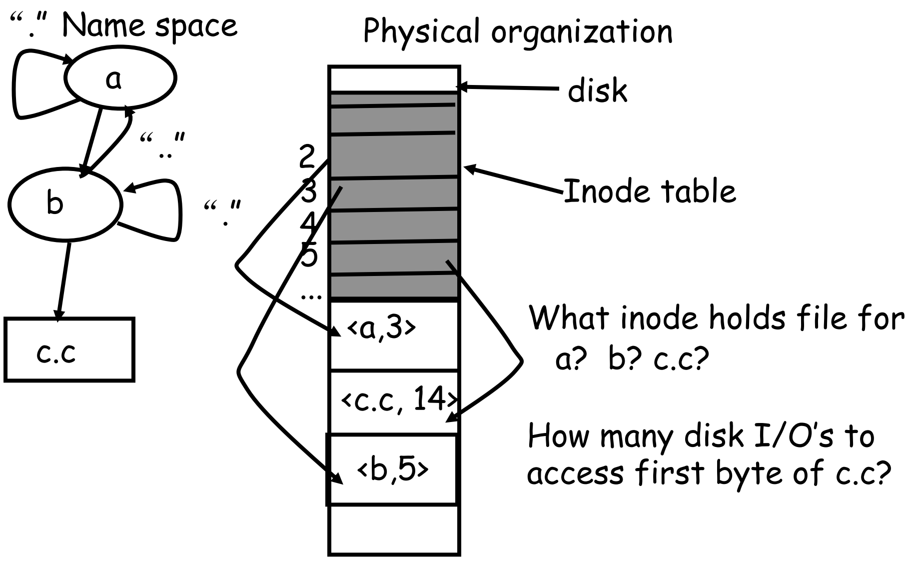
51

Naming magic

- Bootstrapping: Where do you start looking?
  - Root directory
  - inode #2 on the system
  - 0 and 1 used for other purposes
- Special names:
  - Root directory: "/" (bootstrap name system for users)
  - Current directory: "."
  - Parent directory: ".." (otherwise how to go up??)
  - user's home directory: "~"
- Using the given names, only need two operations to navigate the entire name space:
  - cd 'name': move into (change context to) directory "name"
  - ls : enumerate all names in current directory (context)

52

Unix example: /a/b/c.c



53

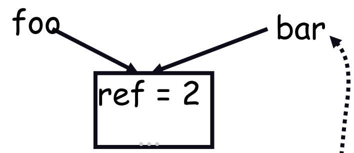
Default context: working directory

- Cumbersome to constantly specify full path names
  - in Unix, each process associated with a "current working directory"
  - file names that do not begin with "/" are assumed to be relative to the working directory, otherwise translation happens as before
- Shells track a default list of active contexts
  - a "search path"
  - given a search path { A, B, C } a shell will check in A, then check in B, then check in C
  - can escape using explicit paths: "./foo"

54

Creating synonyms: hard and soft links

- More than one dir entry can refer to a given file
  - Unix stores count of pointers ("hard links") to inode
  - to make: "ln foo bar" creates a synonym ('bar') for 'foo'



- Soft links:
  - also point to a file (or dir), but object can be deleted from underneath it (or never even exist).
  - Unix builds like directories: normal file holds pointer to name, with special "sym link" bit set

  - When the file system encounters a symbolic link it automatically translates it (if possible).

55

Example: basic system calls in Unix

What happens when you open and read a file?

- open
- read
- close
- lseek
- create
- write

56

Example: the open-read-close cycle

1. The process calls `open ("DATA.test", RD_ONLY)`
2. The kernel:
   - Get the current working directory of the process:
     Let's say "/c/cs422/as/as3
   - Call "namei":
     Get the inode for the root directory "/"

     For (each component in the path) {
       can we open and read the directory file ?
       if no, open request failed, return error;
       if yes, **read** the blocks in the directory file;
         Based on the information from the I-node, read through the directory file
         to find the inode for the next component;
     }
     At the end of the loop, we have the inode for the file DATA.test

57

Example: open-read-close (cont'd)

1. The process calls `open ("DATA.test", RD_ONLY)`
2. The kernel:
   - Get the current working directory of the process:
   - Call "namei" and get the inode for DATA.test;
   - Find an empty slot "fd" in the file descriptor table for the process;
   - Put the pointer to the inode in the slot "fd";
   - Set the initial file pointer value in the slot "fd" to 0;
   - Return "fd".
3. The process calls read(fd, buffer, length);
4. The kernel:
   - From "fd" find the file pointer
   - Based on the file system block size (let's say 1 KB), find the blocks where the bytes (file_pointer, file_pointer+length) lies;
   - Read the inode

58

Example: open-read-close (cont'd)

4. The kernel:
   - From "fd" find the file pointer
   - Based on the file system block size (let's say 1 KB), find the blocks where the bytes (file_pointer, file_pointer+length) lies;
   - Read the inode
   - For (each block) {
     * If the block # < 11, find the disk address of the block in the entries in the inode
     * If the block # >= 11, but < 11 + (1024/4): read the "single indirect" block to find the address of the block
     * If the block # >= 11+(1024/4) but < 11 + 256 + 256 * 256: read the "double indirect" block and find the block's address
     * Otherwise, read the "triple indirect" block and find the block's address }
   - Read the block from the disk
   - Copy the bytes in the block to the appropriate location in the buffer
5. The process calls close(fd);
6. The kernel: deallocate the fd entry, mark it as empty.

59

Example: the create-write-close cycle

1. The process calls `create ("README");`
2. The kernel:
   - Get the current working directory of the process:
     Let's say "/c/cs422/as/as3
   - Call "namei" and see if a file name "README" already exists in that directory
   - If yes, return error "file already exists";
   - If no:
     Allocate a new inode;
     Write the directory file "/c/cs422/as/as3" to add a new entry for the ("README", disk address of inode) pair
   - Find an empty slot "fd" in the file descriptor table for the process;
   - Put the pointer to the inode in the slot "fd";
   - Set the file pointer in the slot "fd" to 0;
   - Return "fd";

60

Example: create-write-close (cont'd)

3. The process calls `write(fd, buffer, length);`
4. The kernel:
   - From "fd" find the file pointer;
   - Based on the file system block size (let's say 1 KB), find the blocks where the bytes (file_pointer, file_pointer+length) lies;
   - Read the inode
   - For (each block) {
     * If the block is new, allocate a new disk block;
     * Based on the block no, enter the block's address to the appropriate places in the inode or the indirect blocks; (the indirect blocks are allocated as needed)
     * Copy the bytes in buffer to the appropriate location in the block }
   - Change the file size field in inode if necessary
5. The process calls close(fd);
6. The kernel deallocate the fd entry --- mark it as empty.

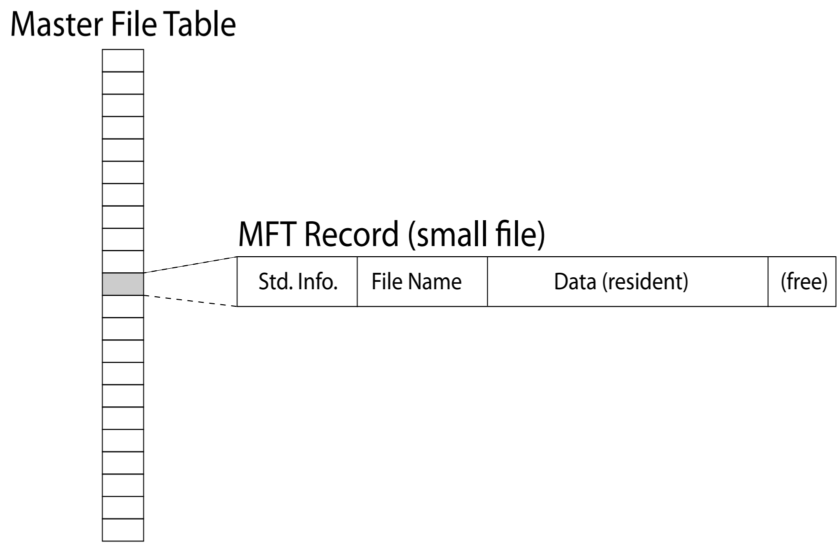
61

NTFS

- Master File Table (MFT)
  - Array of 1KB MFT records for metadata and data
- Extents
  - Block pointers cover runs of blocks
  - Similar approach in linux (ext4)
  - File create can provide hint as to size of file
- Journaling for reliability

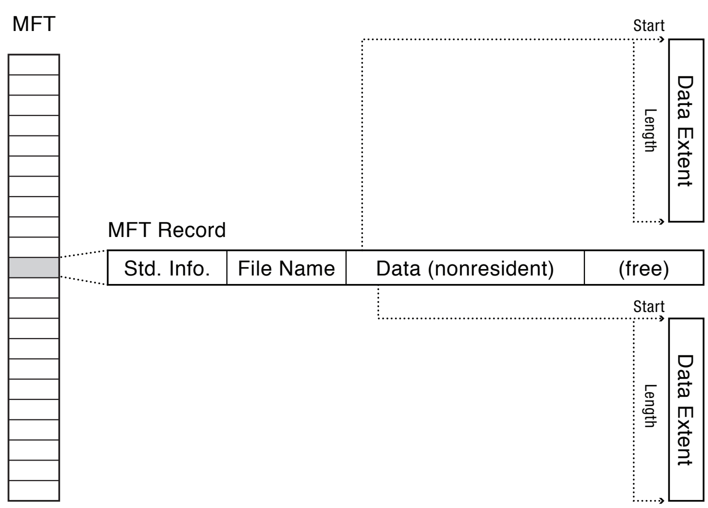
62

NTFS small file



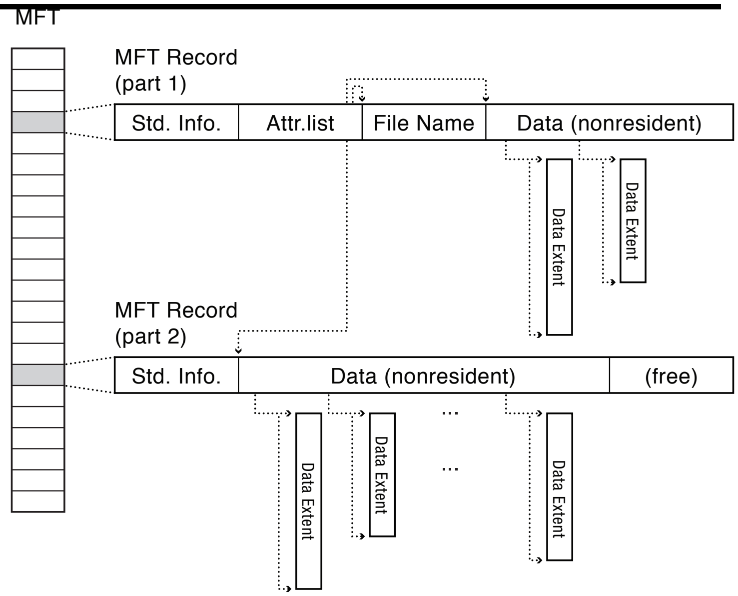
63

NTFS medium-sized file



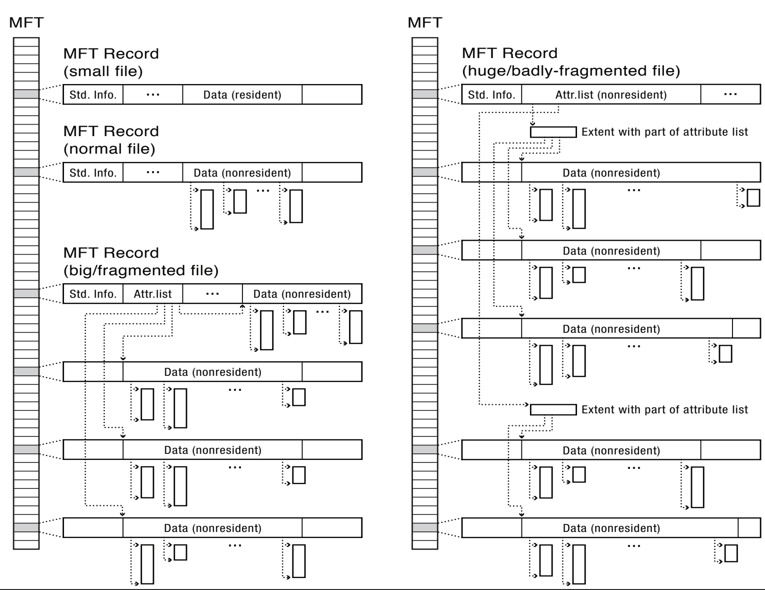
64

NTFS indirect block



65

NTFS files in four stages of growth



66