File Systems

Arvind Krishnamurthy

Spring 2004

File Systems

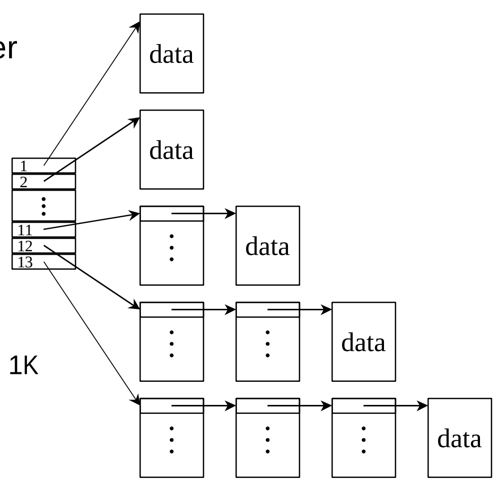
- Implementing file system abstraction on top of raw disks
- Issues:
  - How to find the blocks of data corresponding to a given file?
  - How to organize files?
  - How to enforce protection?
- Performance issues: need to minimize the number of “non-local” disk accesses
  - Try to keep related information together on the disk

File Blocks Organization

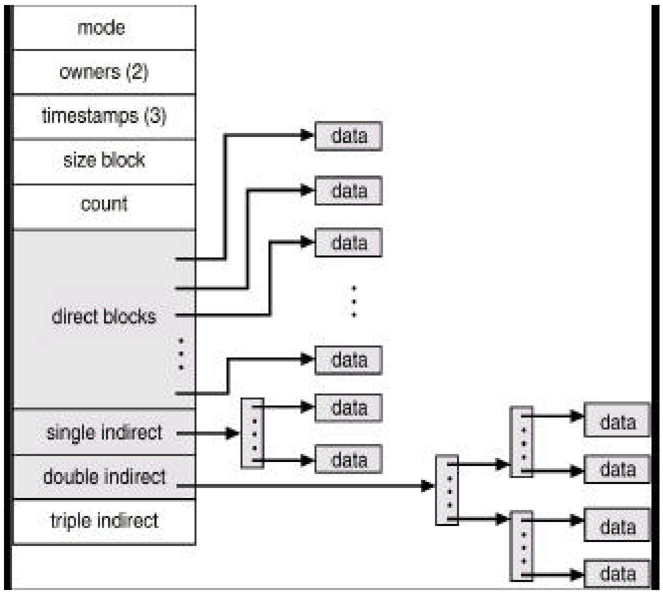
- Approaches:
  - Contiguous allocation
    - A file is stored on a contiguous set of blocks
    - Prevents incremental growth and complicates allocation
  - Linked list allocation
    - A file header points to the first block of the file
    - Each block of the file points to the next block
    - If blocks are dispersed across disk ➔ horrible performance for both sequential access and random access
    - Random access can be made faster by separating the next block pointers from the data and storing it at a centralized place (FAT)
  - Indexed files
    - File header stores pointers to file blocks
    - Multi-level indexing required for large files

Hybrid Multi-level Indexing Scheme

- Used in Unix 4.1
- 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 (assuming 1K blocks)



Unix file header (I-node)



Disk Layout

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

- File headers (I-nodes) are identified by a number (I-number)
  - Can translate the I-number to a location on the disk
  - Headers are either located together as one group or spread across the disk in predetermined fashion
- 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

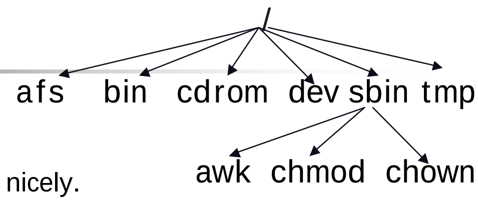
Naming and directories

- 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

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
- 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.)

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
  - users 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 is needed to make a DAG?)

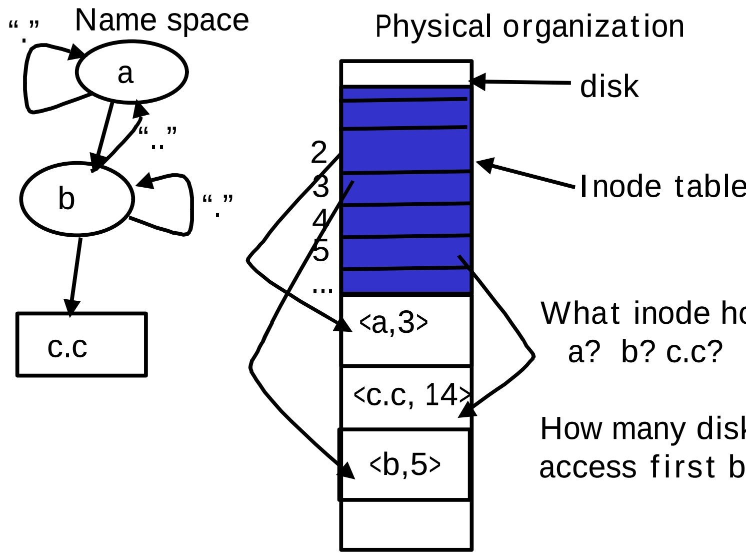
<name, inode#>
<afs, 1021>
<tmp, 1020>
<bin, 1022>
<cdrom, 4123>
<dev, 1001>
<sbin, 1011>

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

Naming

- 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: “..”

Unix example: /a/b/c.c



What inode holds file for a? b? c.c?

How many disk I/O's to access first byte of c.c?

Announcements

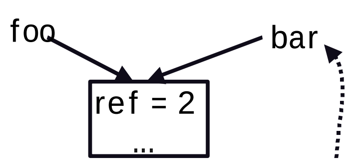
- Paper reviews due on Wednesday for:
  - Fast File Systems
  - Log structured file systems
- Next assignment will be online by tomorrow

Outline

- Topics covered so far in file systems:
  - Data blocks
  - File headers
  - Directories
  - File system superblocks
- Remaining topics:
  - Hard and soft links
  - Permissions

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).
  - 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).

Protection

- Goals:
  - Prevent accidental and maliciously destructive behavior
  - Ensure fair resource usage
- A key distinction to make: policy vs. mechanism
  - **Policy**: what is to be done
  - **Mechanism**: how something is to be done

Access control

- Domain structure
  - Access/usage rights associated with particular domain
  - Example: user/kernel mode → two domains
  - Unix: each user is a domain; super-user domain; groups of users (and groups)
- Type of access rights
  - For files: read/write/execute
  - For directories: list/modify/delete
  - For access rights themselves
    - Owner (I have the right to change the access rights for some resource)
    - Copy (I have the right to give someone else a copy of an access right I have)
    - Control (I have the right to revoke someone else's access rights)

Access control matrix

- Conceptually, we can think of the system enforcing access controls based on a giant table that encodes all access rights held by each domain in the system

Example:

	File1	File2	File3	Dir1	Dir2	...
UserA	rw	r	rwx	lmd	l	...
GroupB		r	rw		lm	...
...	...	...	...	...	...	...

The access control matrix is the "policy" we want to enforce;

Mechanisms: (1) access control lists

(2) capability lists

Access control lists vs. capability lists

- **Access control lists** (ACL): keep lists of access for each domain with each object:

  File3: User A: rwx

  Group B: rw

  …………

- **Capability lists** (CAP): keep lists of access rights for each object with each domain

  User A: File1: rw

  File2: r

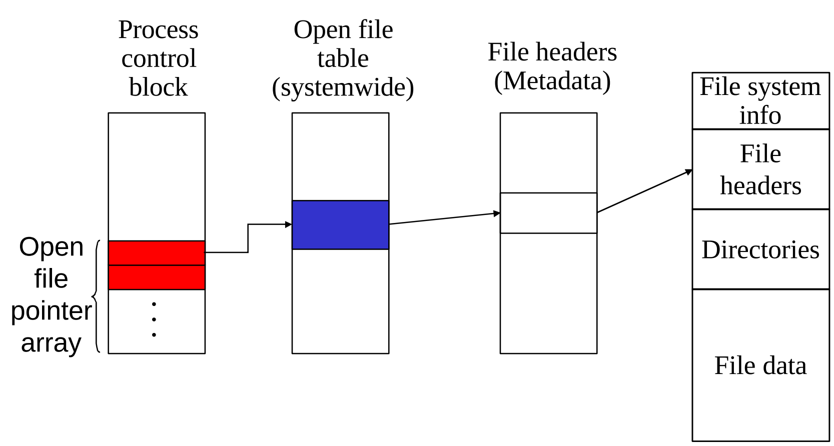
  …………

- Which is better?
  - ACLs allow easy changing of an object's permissions
  - Capability lists allow easy changing of a domain's permissions

A combined approach

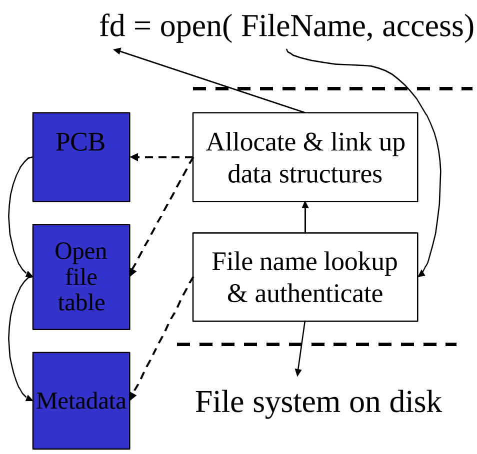
- Objects have ACLs
- Users have CAPs, called "groups" or "roles"
- ACLs can refer to users or groups
- Change permissions on an object by modifying its ACL
- Change broad user permissions via changes in group membership

Data structures for a typical file system

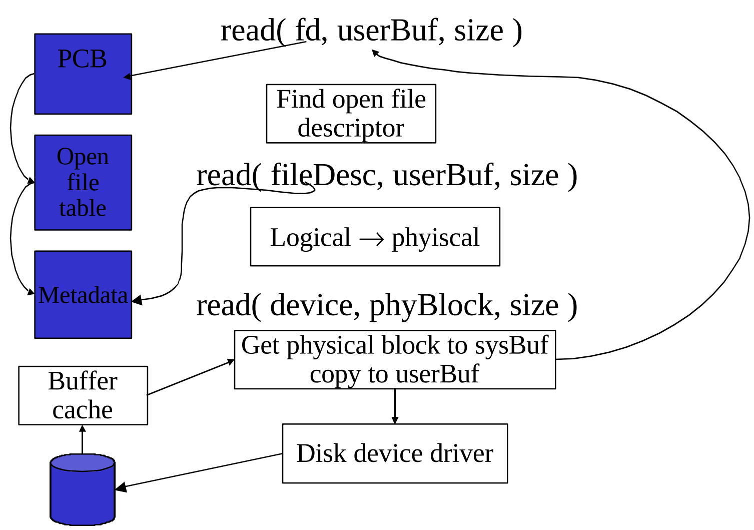


Appendix: Open a file

- 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



Read a block



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

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

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.

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";

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.