File Systems (1)

May 28, 2018

Byung-Gon Chun

Acknowledgments. Slides and/or picture in the following are adapted from UW, Columbia, and UC Berkeley slides

So far

- Operating system kernel - user space/kernel space
- Computation abstraction – process, thread, synchronization
- Memory abstraction – address translation, caching, virtual memory

File Systems

- File systems
  - FFS, EXT*x*, LFS
- Linux VFS
- RAID, Flash, …

Outline

- File system concepts
  - What is a file?
  - What operations can be performed on files?
  - What is a directory and how is it organized?
- File implementation
  - How to allocate disk space to files?

CSE 컴퓨터공학부
Department of Computer Science & Engineering

What Is A File?

- User view
  - Named byte array
  - Permanently and conveniently available

- OS view
  - Map bytes as collection of blocks on physical storage
    - Stored on nonvolatile storage device: Magnetic Disks, SSDs
    - Persistent across reboots and power failures

Role of File System

- Naming
  - How to “name” files
  - Translate “name” + offset -> logical block #
- Reliability
  - Must not lose file data\
- Protection
  - Must mediate file access from different users
- Disk management
  - Fair, efficient use of disk space
  - Fast access to files

File Metadata

- Metadata: additional system information associated with each file
  - Name – only information kept in human-readable form
  - Type of file
  - Location – pointer to file location on device
  - Identifier – unique tag (number) identifies file within file system (inode number in UNIX)
  - Size – current file size
  - Time – creating, access, modification
  - Protection – controls who can do reading, writing, executing
  - Owner and group id
  - Special file? (directory? Symbolic link?)
- Meta-data is stored on disk

CSE 컴퓨터공학부 Department of Computer Science & Engineering

UNIX File Operations

- int creat(const char* pathname, mode_t mode)
- int unlink(const char* pathname)
- int rename(const char* oldpath, const char* newpath)
- int open(const char* pathname, int flags, mode_t mode)
- int read(int fd, void* buf, size_t count);
- int write(int fd, const void* buf, size_t count)
- int lseek(int fd, offset_t offset, int whence)
- int truncate(const char* pathname, offset_t len)
- ...

Everything as a File

- A core UNIX tenet from the early days
  - Block devices (disks, graphics cards in /dev)
  - Character devices (USB devices, network cards in /dev)
  - IPC: Pipes, Network sockets
  - Accessing kernel data structures (/proc, /sys)
  - Setting kernel configuration
  - Volatile filesystems in RAM (e.g., tmpfs)
  - Shared memory (based on tmpfs/shmfs)
  - Remote files (NFS, SMB, AFP, …)
  - Even normal local files
- Implications
  - Everything accessed using common API (open, read, write)
  - Implementation may be totally different
  - OS must support some measure of object orientedness

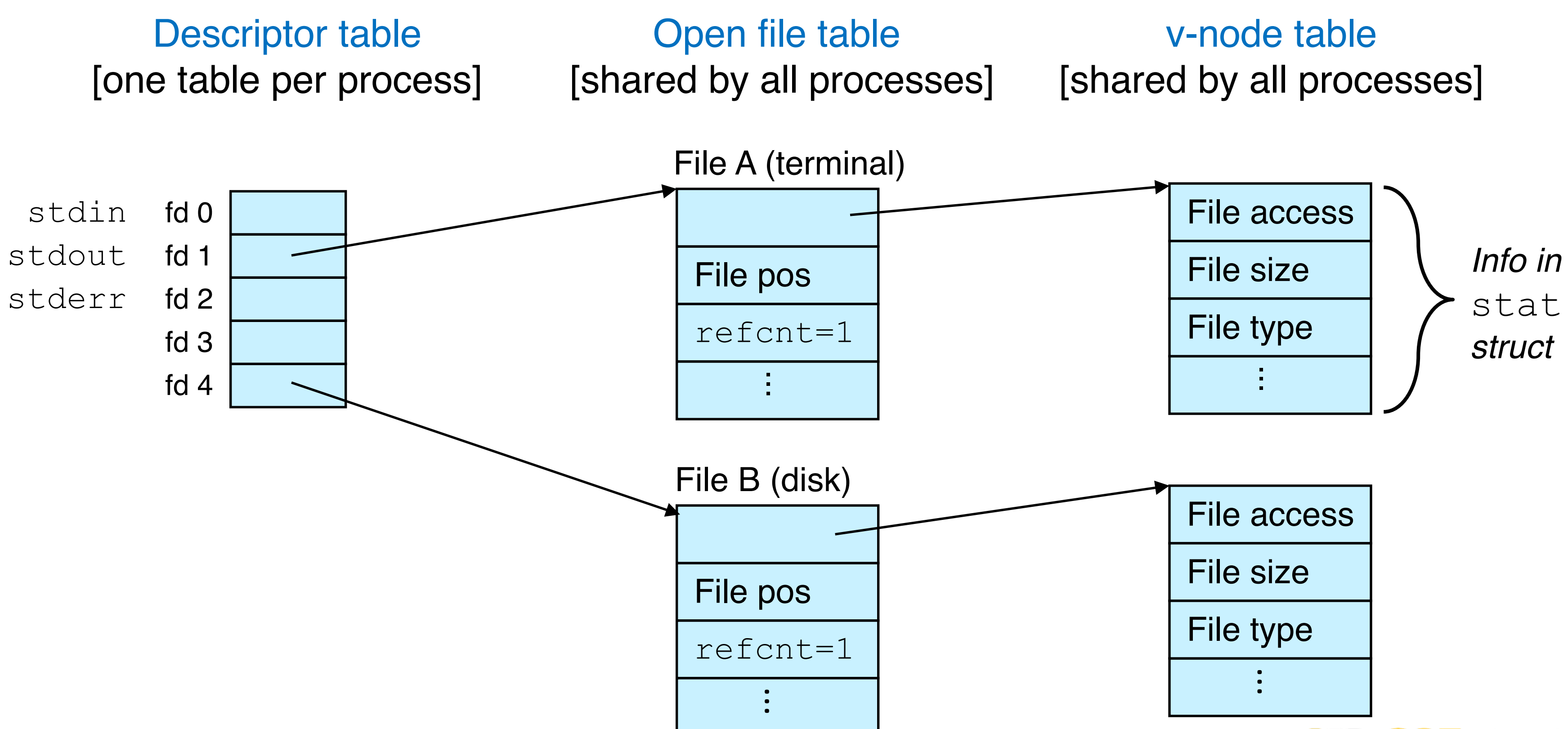
Open Files

- Problem: expensive to resolve name to identifier on each access
- Solution: open file before access
  - Name resolution: search directories for file name and check permission
  - Read relevant file metadata into open file table in memory
  - Return index in open file table (file descriptor)
  - Application pass index to OS for subsequent access

- System-wide open file table shared across processes
- Per-process file table stores current pointer position and index to system-wide open file table

How the Unix Kernel Represents Open Files

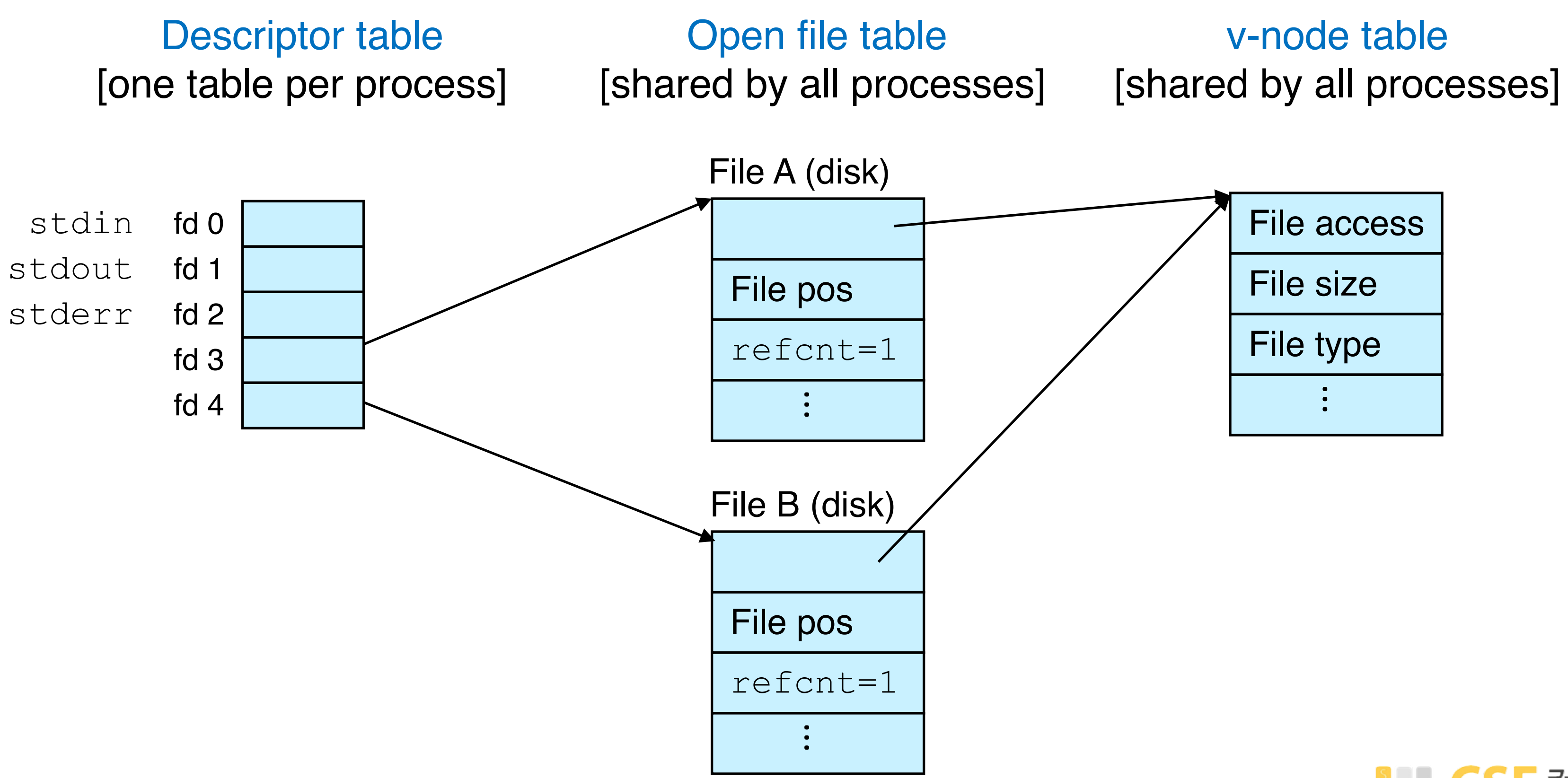
- Two descriptors referencing two distinct open disk files. Descriptor 1 (stdout)
  points to terminal, and descriptor 4 points to open disk file
  "Vnodes: An Architecture for Multiple File System Types in Sun Unix"



CSE 컴퓨터공학부 Department of Computer Science & Engineering

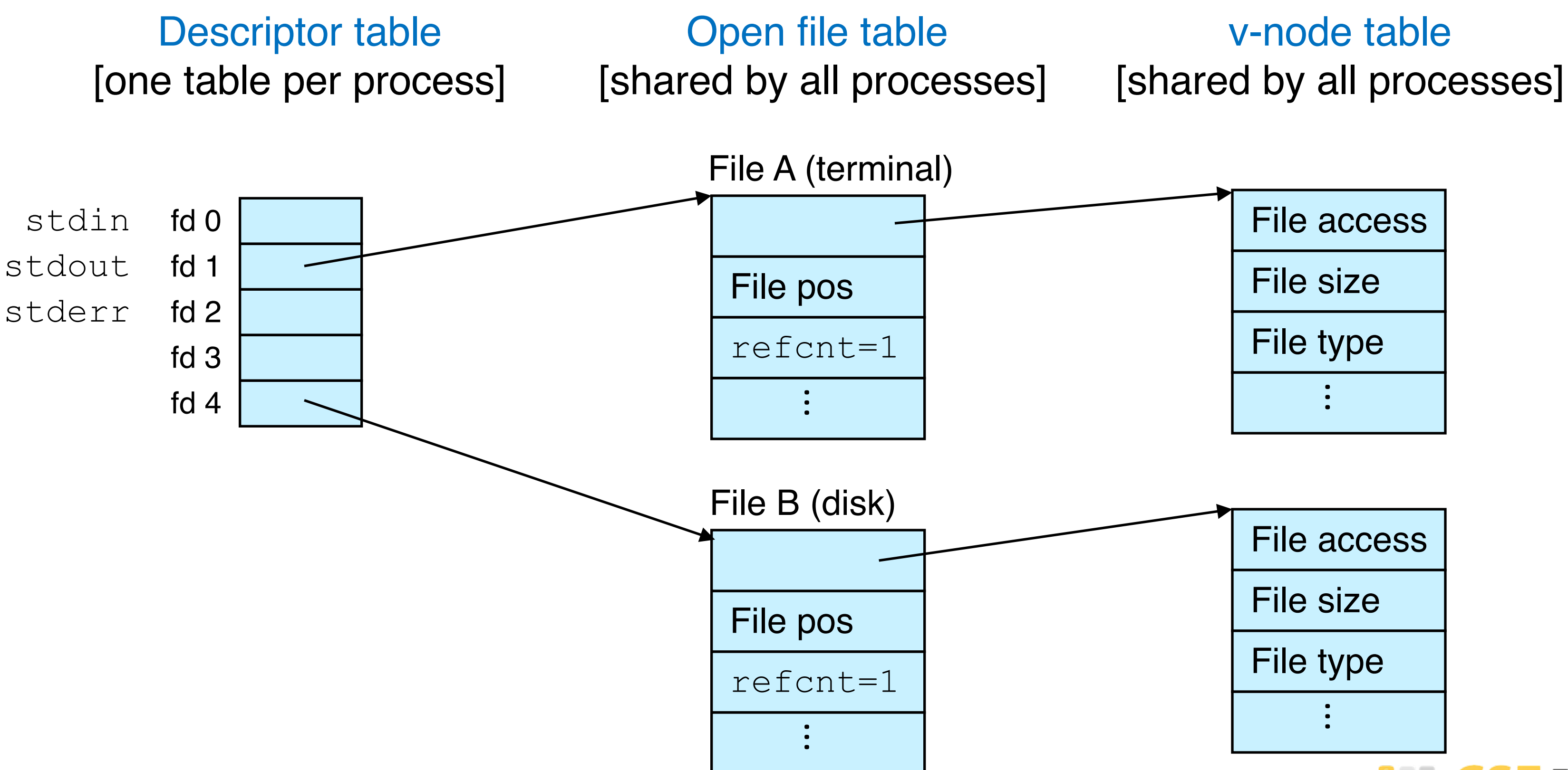
File Sharing

- Two distinct descriptors sharing the same disk file through two distinct open file table entries
  - E.g., Calling open twice with the same filename argument



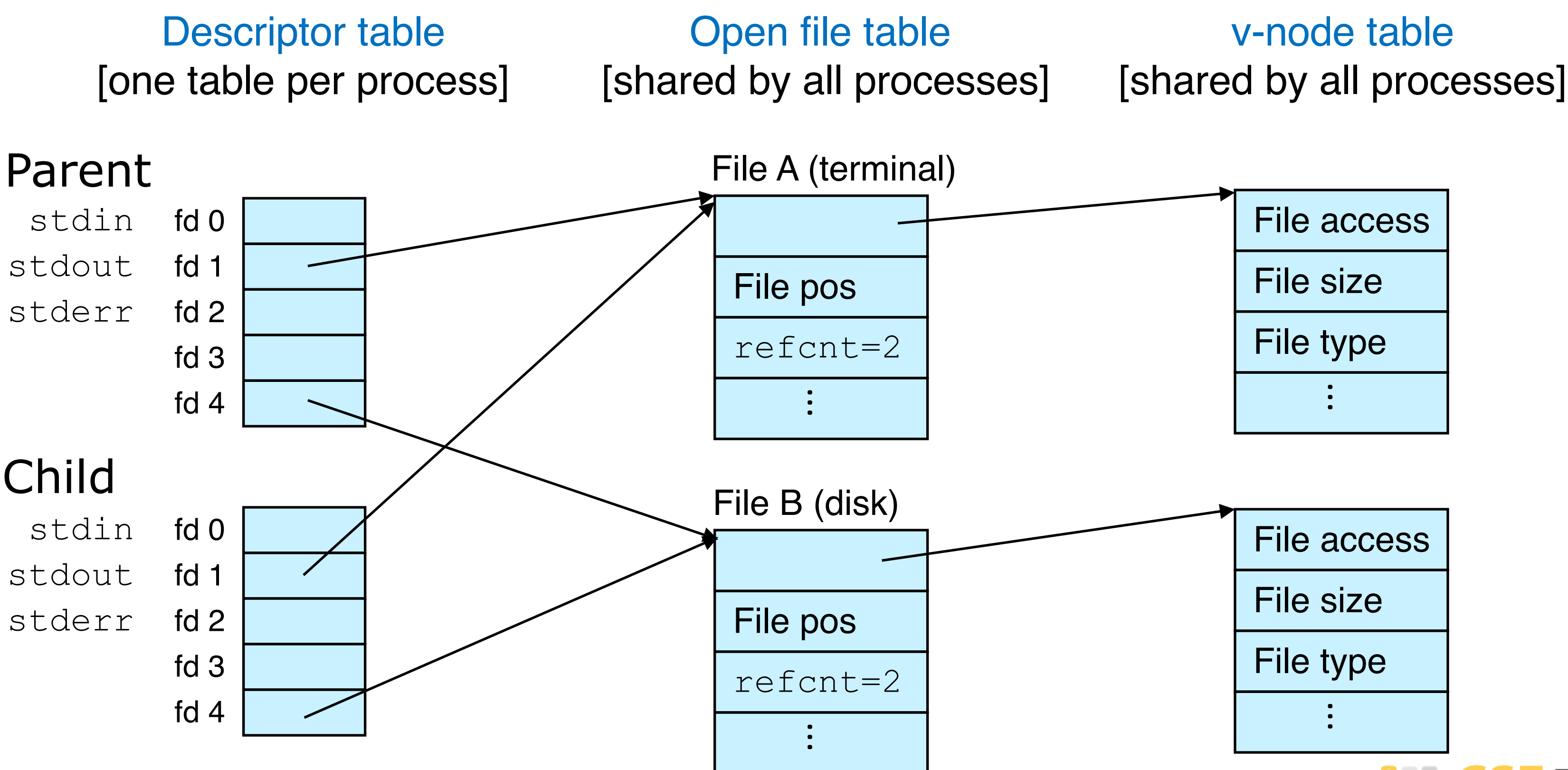
How Processes Share Files: Fork()

- A child process inherits its parent's open files
- Before fork() call:



How Processes Share Files: Fork()

- A child process inherits its parent's open files
- After fork() call:



Directories

- Organization technique
  - Map file name to location on disk
  - Also stored on disk
- Single-Level directory
  - Single directory for entire disk
    - Each file must have unique name
  - Not very usable
  - Special part of disk holds directory listing
- Two-level directory
  - Directory for each user
  - Still not very usable

Tree-Structured Directory

- Directory stored on disk just like files
  - Data consists of <name, index> pairs
    - Index points to file identifier (inode)
    - Name can be another directory
  - Designated by special bit in meta-data
  - Reference by separating names with slashes
  - Operations
    - User programs can read (readdir())
    - Only special system calls can write
- Special directories
  - Root (/): fixed index for metadata
  - . : this directory
  - .. : parent directory

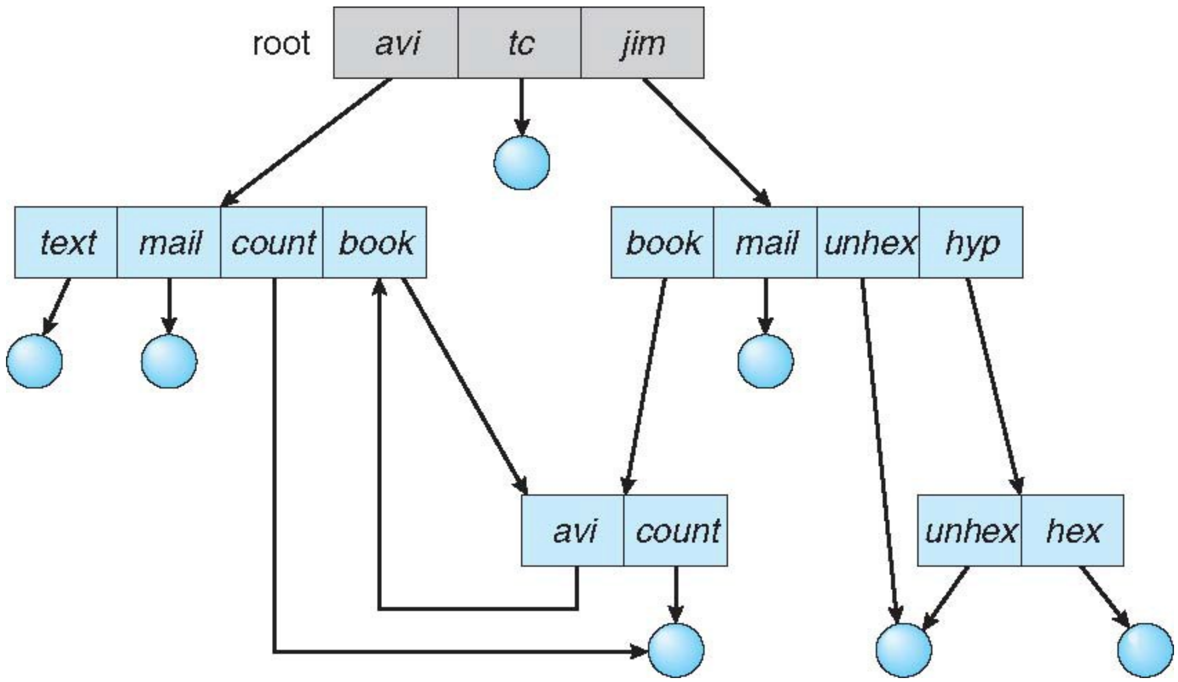
Tree-Structured Directory

- Example: `mkdir /a/b/c`
  - Read meta-data 2, look for “a”: find <“a”, 5>
  - Read 5, look for “b”: find <“b”, 9>
  - Read 9, verify no “c” exists; allocate c and add “c” to directory

Acyclic-Graph Directories

- More general than tree structure
  - Add connections across the tree (no cycles)
  - Create links from one file (or directory) to another
- Two types of links
  - Symbolic link
    - Special file, designated by bit in meta-data
    - File data is name to another file
  - Hard link
    - Multiple directory entries point to same file
    - All hard-links are equal: no primary
    - Store reference count in file metadata
    - Cannot refer to directories; why?

General Graph Directory and Cycles



- Cycles cause problems with reference counts
- E.g., a cycle that isn't accessible through root
- Need garbage collection

Acyclic-Graph Directories

- Hard link: "`ln a b`" ("a" must exist already)
  - Idea: Can use name "a" or "b" to get to same file data
  - Implementation: Multiple directory entries point to same meta-data
  - What happens when you remove a? Does b still exist?
    - How is this feature implemented???
  - Unix: Does not create hard links to directories. Why?

Acyclic-Graph Directories

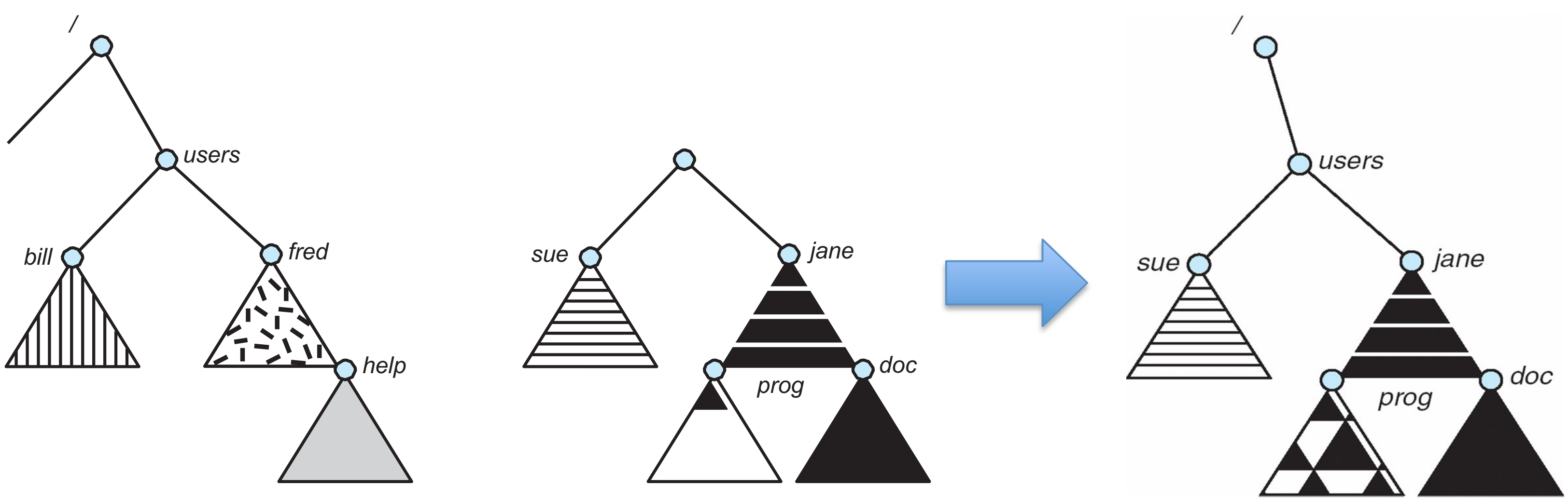
- Symbolic (soft) link: "`ln -s a b`"
  - Can use name "a" or "b" to get to same file data, if "a" exists
  - When reference "b", lookup soft link pathname
  - b: Special file (designated by bit in meta-data)
    - Contents of b contain name of "a"

Path Names

- Absolute path name (full path name)
  - Start at root directory
    - E.g. /home/html
- Relative path name
  - Full path is lengthy and inflexible
  - Give each process current working directory
  - Assume file in current directory

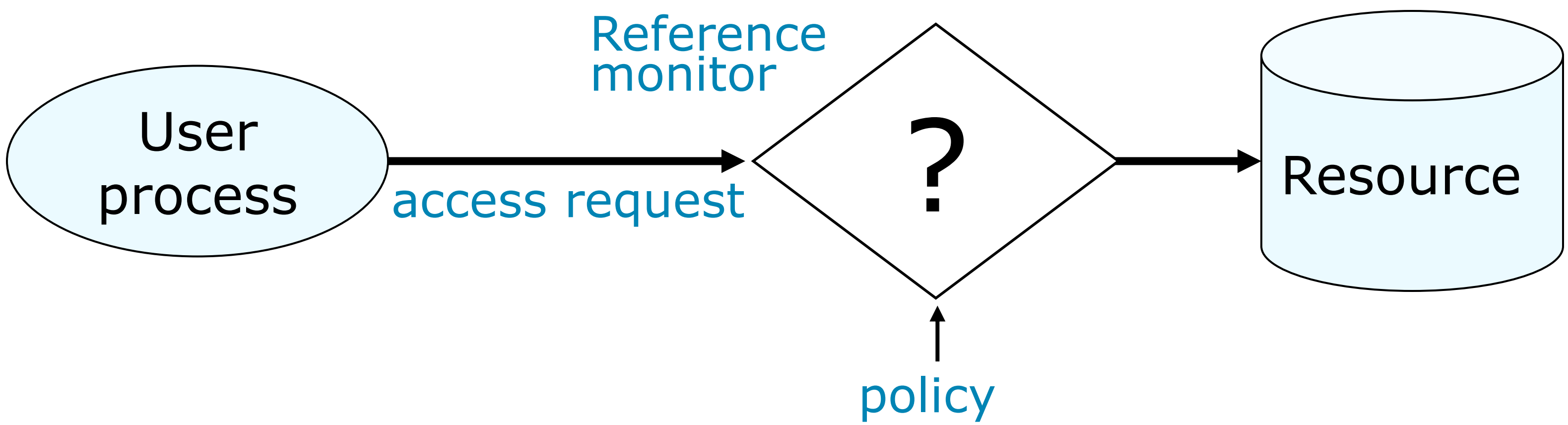
File System Mounting

- Start off with root filesystem
- New file systems can be mounted into an existing directory (mount point)
- E.g., mount –o opts –t ext2 /dev/hda3 /users



Access control

- Assumptions
  - System knows who the user is
    - Authentication via name and password, other credential
  - Access requests pass through gatekeeper (reference monitor)
    - System must not allow monitor to be bypassed



Access Control [Lampson]

- The guard evaluates a function:
  permissions = policy(subject, object)

- If functions are too mathematical, call it an access matrix (Lampson 1971)

Access control matrix [Lampson]

Objects

Subjects

	File 1	File 2	File 3	…	File n
User 1	read	write	-	-	read
User 2	write	write	write	-	-
User 3	-	-	-	read	read
…					
User m	read	write	read	write	read

Implementation concepts

- Access control list (ACL)
  - Store column of matrix with the resource
- Capability
  - User holds a "ticket" for each resource
  - Two variations
    - store row of matrix with user, under OS control
    - unforgeable ticket in user space

	File 1	File 2	…
User 1	read	write	-
User 2	write	write	-
User 3	-	-	read
…			
User m	Read	write	write

Access control lists are widely used, often with groups

Some aspects of capability concept are used in many systems

ACL vs Capabilities

- Access control list
  - Associate list with each object
  - Check user/group against list
  - Relies on authentication: need to know user
- Capabilities
  - Capability is unforgeable ticket
    - Random bit sequence, or managed by OS
    - Can be passed from one process to another
  - Reference monitor checks ticket
    - Does not need to know identify of user/process

Protection

- Type of access
  - Read, write, execute, append, delete, list, …
- Access control list
  - Associate lists of users with access rights for every file
  - Advantage: complete control
  - Disadvantage
    - Tedious to construct list (may not know in advance for all users)
      - Require variable-size information
- <u>Classify users</u>
  - Assign a owner and group to each file
  - Different permissions based on who is accessing: owner, group, other
  - Advantage: easier to implement
  - Disadvantage: no fine grained control

CSE 컴퓨터공학부 Department of Computer Science & Engineering