CSC369: Operating Systems

Fall 2014

Andrew Petersen

Anonymous Says!

vague - that is my description of the instructions for this assignment.

- I agree!
- What do you find most frustrating (or liberating) about that?

Anonymous Says!

vague - that is my description of the instructions for this assignment.

- Yes.
- ... and wait until you see A3.
- We will ask for an outcome. We will not tell you what you *must* do to make that happen.

Speaking of which, questions on A2?

The File Abstraction

- Users interact with the filesystem using the abstraction of the *file*.
- This is similar to the *process* abstraction: a file is the OS's unit of storage.

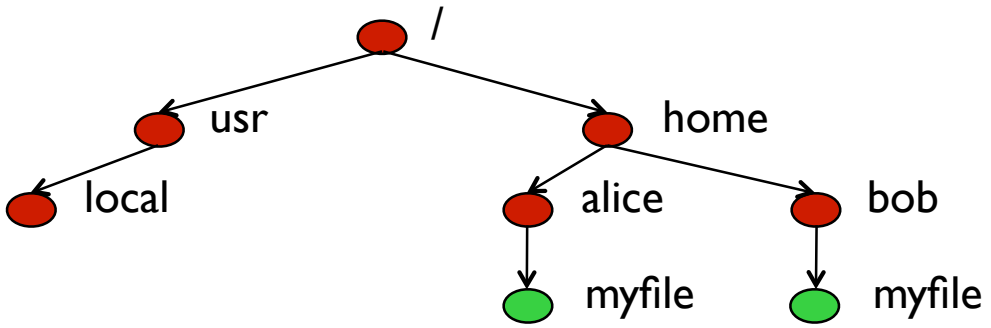
- First, though, we need *directories*.

Outline

- Directories
- File Operations
- Inodes
- Traversing Paths

Directories

- Directories provide logical structure to file systems
  - For users, they provide a means to organize files
  - For the file system, they provide a convenient naming interface
    - Allows the implementation to separate logical file organization from physical file placement
  - Use to store information about files (owner, permission, etc.)
- Most file systems support multi-level directories
  - Naming hierarchies (/, /usr, /usr/local/, /home, …)



Directory Structure

- A directory is a list of entries
  - Each entry contains a name and associated metadata
  - The metadata describes properties of the file (size, protection, location, etc.)

- The list is usually unordered (effectively random)
  - Entries are usually sorted by the program that reads the directory
- Directories are typically stored as files
  - Only need to manage one kind of secondary storage unit

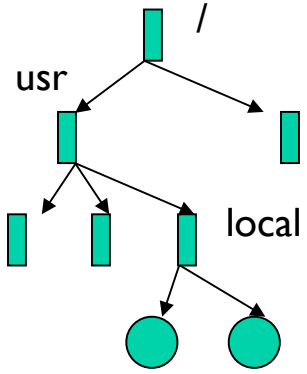
Operations on Directories

- Search
  - Find a particular file within directory
- Create file
  - Add a new entry to the directory
- Delete file
  - Remove an entry from the directory
- List directory
  - Return file names and requested attributes of entries
- Update directory
  - Record a change to some file's attributes

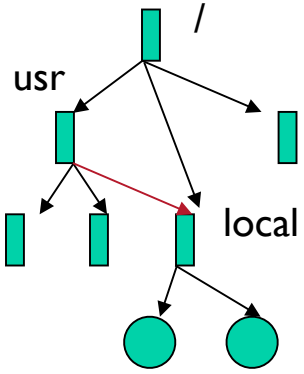
Directory Implementations

- Single-level, two-level, or tree-structured
- Acyclic-graph directories: allows for shared directories
  - The same file or subdirectory may be in 2 different directories

Tree-structured:

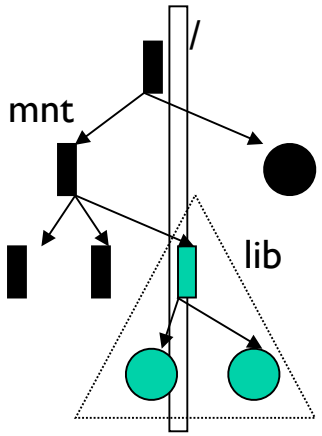


Acyclic graph:



File System Mounting

- File system "namespace" may be built by gluing together subtrees from multiple physical partitions
  - Each device (or disk partition) stores a single file system
  - Mount point is an empty directory in the existing namespace
  - Parent directory notes that a file system is mounted at dir



File Links

- Sharing can be implemented by creating a new directory entry called a link: a pointer to another file or subdirectory
  - Symbolic, or soft, link
    - Directory entry refers to file that holds "true" path to the linked file
  - Hard links
    - Second directory entry identical to the first

'/' directory

File Name	Start Block	Type
...	...	...
local	42	dir
usr	150	dir

'usr' directory (hard link)

File Name	Start Block	Type
...	...	...
local	42	dir
...	...	...

'usr' directory (soft link)

File Name	Start Block	Type
...	...	...
local	215	link
...	...	...

Issues with Links

- With links, a file may have multiple absolute path names
  - Traversing a file system should avoid traversing shared structures more than once
- Maintaining consistency is a problem
  - How do you update permissions in directory entry with a hard link?
- Deletion: When can the space allocated to a shared file be deallocated and reused?
  - Somewhat easier to handle with symbolic links
    - Deletion of a link is OK; deletion of the file entry itself deallocates space and leaves the link pointers dangling
  - Keep a reference count for hard links
- Sharing: How can you tell when two processes are sharing the same file?

File Sharing

- File sharing is incredibly important for getting work done
  - Basis for communication and synchronization
  - Uh-oh ... there's that word again ...

- Two key issues when sharing files
  - Semantics of concurrent access
    - What happens when one process reads while another writes?
    - What happens when two processes open a file for writing?
  - Protection

Outline

- Directories
- File Operations
- Inodes
- Traversing Paths

File Operations

- Creation
  - Find space in file system, add entry to directory mapping file name to location and attributes
- Writing
- Reading
  - Dominant abstraction is "file as stream"
  - Repositioning within a file
- File removal
- Truncation and appending
  - May erase the contents (or part of the contents) of a file while keeping attributes

Example File Operations

Unix: fcntl.h

- creat(name)
- open(name, mode)
- read(fd, buf, len)
- write(fd, buf, len)
- sync(fd)
- seek(fd, pos)
- close(fd)
- unlink(name)

NT

- CreateFile(name, CREATE)
- CreateFile(name, OPEN)
- ReadFile(handle, ...)
- WriteFile(handle, ...)
- FlushFileBuffers(handle, ...)
- SetFilePointer(handle, ...)
- CloseHandle(handle, ...)
- DeleteFile(name)
- CopyFile(name)
- MoveFile(name)

Handling File Operations

- Must search the directory for the entry associated with the named file
- When the file is first used, store its attribute info in a system-wide open-file table
  - The index into the open-file table is used on subsequent operations, so no searching is required

Unix example (open, read, write are syscalls):

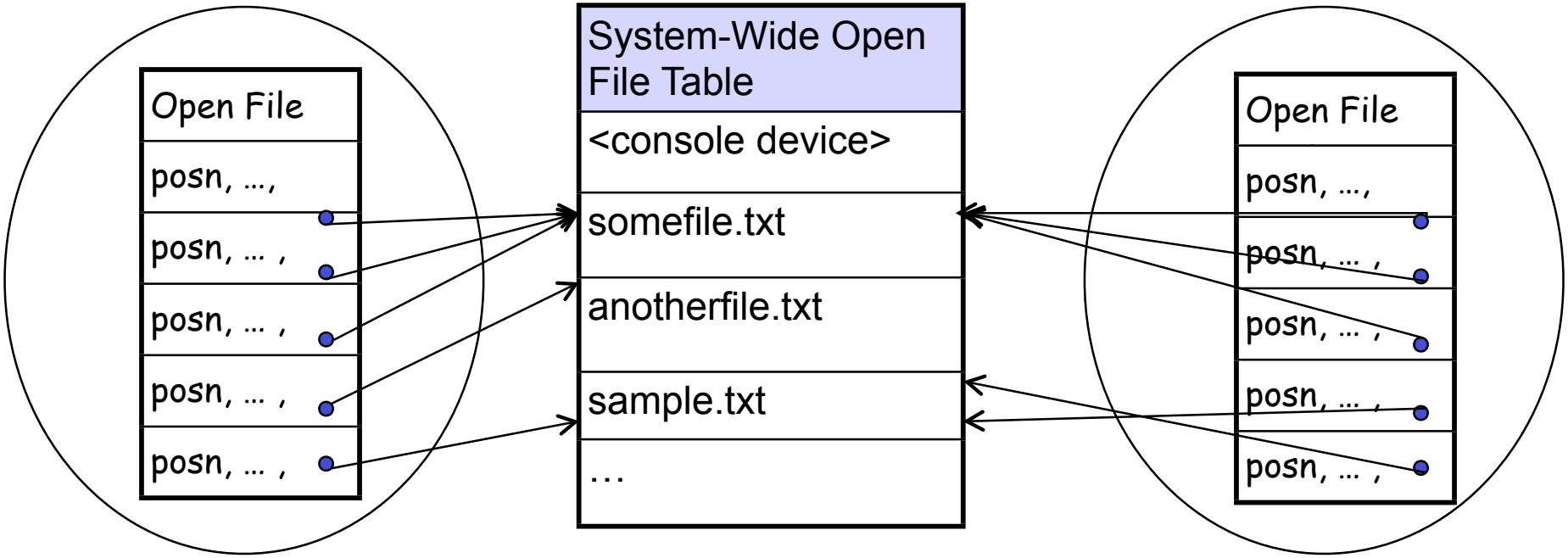
```
main() {
     char onebyte;
     int fd = open("sample.txt", "r");
     read(fd, &onebyte, 1);
     write(STDOUT, &onebyte, 1);
     close(fd);
}
```

Open File Table
<console device>
…
sample.txt
…
…

Shared Open Files

- Actually, we use two levels of internal tables

  1. A per-process table of all files that each process has open that contains the current file position for that process
  2. Each entry in the per-process table has a pointer to an entry in the system-wide open-file table for process independent info



File Access Methods

- General-purpose file systems support simple methods
  - Sequential access – read bytes one at a time, in order
  - Direct access – random access given block/byte number

- Database systems support more sophisticated methods
  - Record access
  - Indexed access

- Older systems provide more complicated methods
- Why do modern systems typically only support simple access?