File System

File system name space; file format and type; operations on files; file data and meta-data (attributes); file open and usage information; file descriptors & file system data structures; directory structures & symbolic/hard links

OS6: 20/2/2018
Textbook (SGG): Ch. 10.1, 10.2, 10.3.2-10.3.7, 10.6, 11.2.1

Note on Ubuntu/Unix Commands

- We will use Ubuntu commands (work for Unix too) to illustrate concepts
  - cat <file> (display the content of a text file)
  - rm <file> (delete a file)
  - mkdir/rmdir <dir> (create/delete a subdirectory)
  - cd <dir> (change to a directory)
  - . or .. in a path name (current or parent directory)
  - find (traverse a file system – see Slide 10.16 for some examples)
  - ln <source-file> <link> (create a hard link from <link> to <source-file>)
  - ln -s <source-file> <link> (create a symbolic link from <link> to <source-file>)

What is a file?

- Different kinds of files
  - Regular files
    - Store data – user data (your pics, music, emails, programs) or system data
  - Directories or folders
    - Organize files in a structured *name space* e.g., /Users/john/Desktop/work
- Data usually stored in secondary storage (especially disks)
  - But not always, e.g., ramdisk (main memory) – new memory technologies (e.g., NAND flash) blur distinction between main memory and disks
  - Non-volatile (persists across shutdowns, etc); can be large, even huge
  - Memory-cached for performance

File structure or format

- None – file is an *uninterpreted* sequence of words/bytes
- Simple record structure
  - Lines (e.g., text files)
  - Fixed length records (e.g., structured database)
  - Variable length records (e.g., unstructured database)
- Complex structures
  - Executable files (e.g., ELF format)
- Who interprets the format? Three possibilities:
  - Operating system kernel
    - Unix/Linux implements directories as special files, knows the difference between directory & regular files; supports executable files but doesn't otherwise require/interpret any formats of regular files
  - System programs (e.g., linker or loader knows ELF)
  - User/application programs (e.g., web browser understands HTML)

File types and extensions

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

File as abstract data type

- Files can be viewed as an abstract data type, like a class of objects in the OOP sense
- As usual, class/object has two key parts
  - State
    - Information about the *files themselves* – file data and meta-data (attributes like file name, creation time, etc) (Slide 10.7)
    - Information about *usage* of the files (Slide 10.9)
  - Interface (set of methods on the objects, Slide 10.8)

File attributes (meta-data)

- **Name** – main id information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different file types
- **Location** – pointer to file's location on disk
- **Size** – current file size
- **Protection** – who can read, write, execute?
- **User id, date, time, etc** – who owns it? when created? when last accessed or modified?
- Meta-data about files usually kept in the *directory* structure, which (like the file's data) is maintained on disk

File interface (methods)

- Main operations on files
  - Create
  - Open
  - Read/write
  - Reposition within file (e.g., **lseek()** system call changes the **cp** on Slide 10.10)
  - Memory map (map file into process's address space)
  - Delete
  - Truncate (remove data, keep attributes)
- Before you can use a file, you must *open* it
  - Begins a *usage session* for the file, subject to access rights, etc
  - Often by explicit open() system call, but can be by other methods too
- After you have used a file, you should *close* it
  - Ends usage session for the file
  - If program crashes (or when it terminates), opened files are usually automatically closed by OS

File usage information: opening files

- Not good to pass file name to every file system operation (e.g., **read()**)
  - Name can be long and of variable length
  - Mapping of name to internal data structures takes time
- Program translates name into succinct *file descriptor* ("handle" to the file) at the beginning of a usage session
  - fd is (integer) index into *per-process* **file-descriptor (fd) table** (see Slide 10.11)
  - Index has meaning only in context of its process
- Each fd table entry points to *system-wide* **open file table** about *usage* of the opened file (see Slide 10.11), e.g.,
  - Current file offset (**cp**): pointer to current read/write location (byte offset, starting from 0)
  - Access status: mode granted like read, write/append, execute
  - Open count: how many fd table entries point to it – e.g., can't remove open file table entry if reference count is positive

Basic Open-Read-Write-Close Paradigm

Content of file “string.txt”: abcdefghij…z (list of characters from a to z)

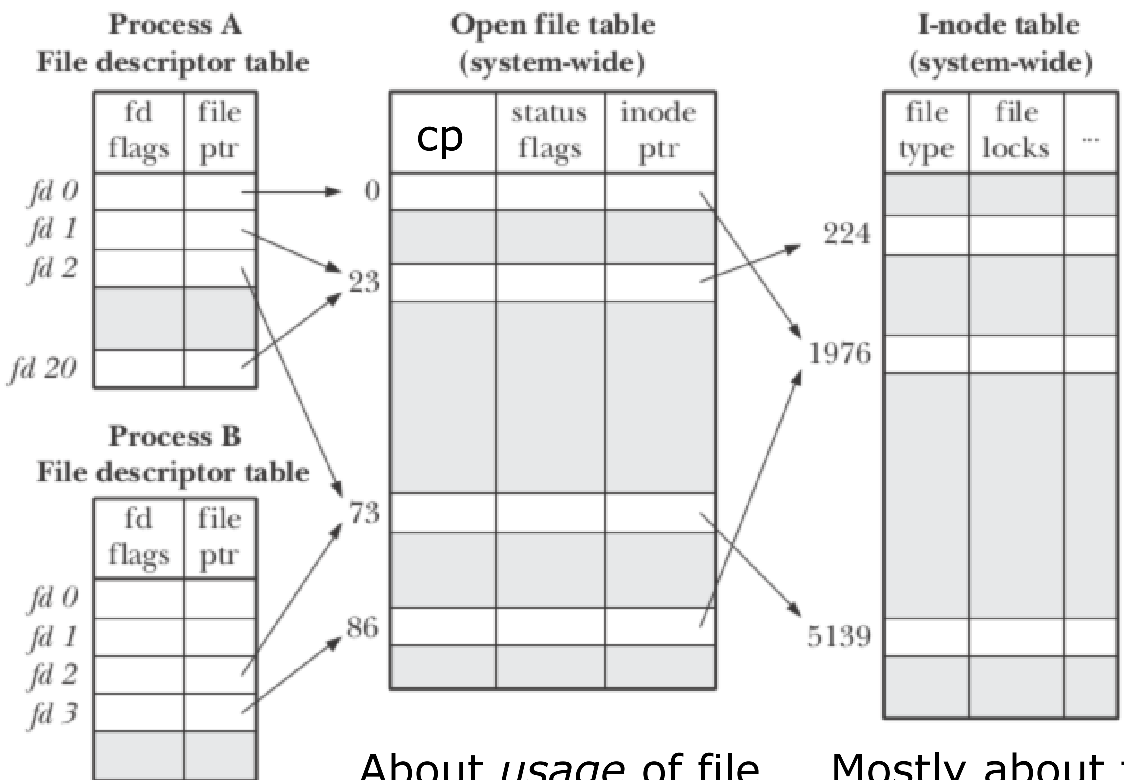
```
#include <fcntl.h>

int fd, buflen=3;
char buf[100];

fd = open("string.txt", O_RDWR); // open file "string.txt" for read and write
                                 // translate file name into file descriptor fd
                                 // start usage session, cp = 0

if (fd < 0) {      // open system call failed
      printf("Can't open file\n");
      exit(-1);
}

read(fd, buf, buflen);      // read abc into buf, cp = 3
read(fd, buf, buflen);      // read def into buf, cp = 6
   .
   .                        // repeated read/write in usage session
   .                        // could also use, say, lseek(fd, 10, SEEK_SET) to change cp to 10
   .

close(fd);                  // end usage session
```

Unix file system data structures

All these data structures are in kernel space:

NB: Process A did a **dup()** system call with argument 1, and the call returned 20 (i.e., fd 20 is now a *duplicate* of fd 1)

i-node (in right table) is Unix kernel's internal data structure for a file



About *usage* of file (e.g., cp – current byte offset for read/write)

Mostly about file itself (e.g., locations of its data blocks)

NB: This design is UNIX specific, similar to Fig. 11.3 in textbook, but not identical
Unless otherwise specified, we assume this design for this course

Mappings between table entries

- Multiple file descriptor table entries can point to same open file table entry (*many-to-one* mapping)
  - A process can have two or more file descriptors referencing the same **open file table** (i.e., middle table on Slide 10.11) entry (e.g., after **dup()** as illustrated on previous slide)
  - Different processes can also have their file descriptors point to same open file table entry
    - Child inherits parent's file descriptors after **fork()**
      - Child gets its *own* fd table, but this fd table initially has the same content as the parent's fd table
    - Unrelated processes can pass file descriptors to each other, e.g., using "Unix domain sockets"
  - If two file descriptors **fd1** and **fd2** reference same open file entry, they share *usage* (e.g., **cp**) of the file – read/write through **fd1** will advance **cp** seen by **fd2**

Mappings between table entries (cont'd)

- Multiple open file table entries can point to the same file (right table on Slide 10.11), i.e., also *many-to-one* mapping
  - *Different* usage instances of the same file (the different instances have *independent* **cp**)
  - e.g., a file opened multiple times by separate **open()** system calls (i.e., each open starts a *new* usage, instead of duplicating an existing usage as in **dup()**)
- Hence, in general, concurrent accesses to files are possible by different processes, shared files allow a form of IPC

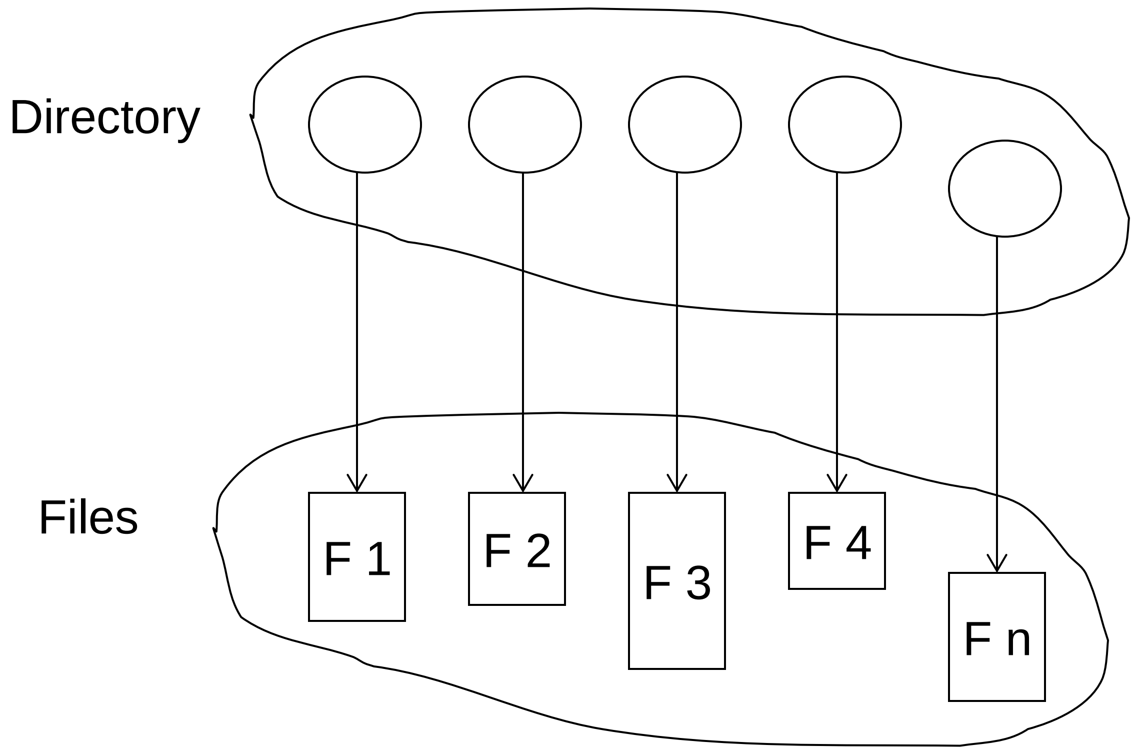
Activity 6.1

- Consider the snapshot of Unix open files shown on Slide 10.11
- Redraw the tables after the following sequence of actions:
  - *B* forks process *C*
  - *C* closes fd 2
  - *C* opens file 1976

  NB. **open()** system call allocates smallest fd that is not currently used

Directory Structure

- Meta-data that organizes files in a structured name space



Typically, both the directory structure and the files themselves reside on disk (and cached in memory).
Backups of these two structures can be kept on tapes.

Operations on directory (your Lab 4)

- Create a file or folder (subdirectory)
- Delete a file or folder
- List a directory
- Rename a file
- Search for a file
- Traverse the file system (beginning directory and all its subdirectories)
  - E.g., try **find(1)** command from your Ubuntu shell …
    - % find . // list all files starting from current directory
    - % find . -name '*.java' // find all files named '.java'
    - % find . -name '*.txt' -exec grep hello {} \; -print

      // find all .txt files that contain the word "hello"

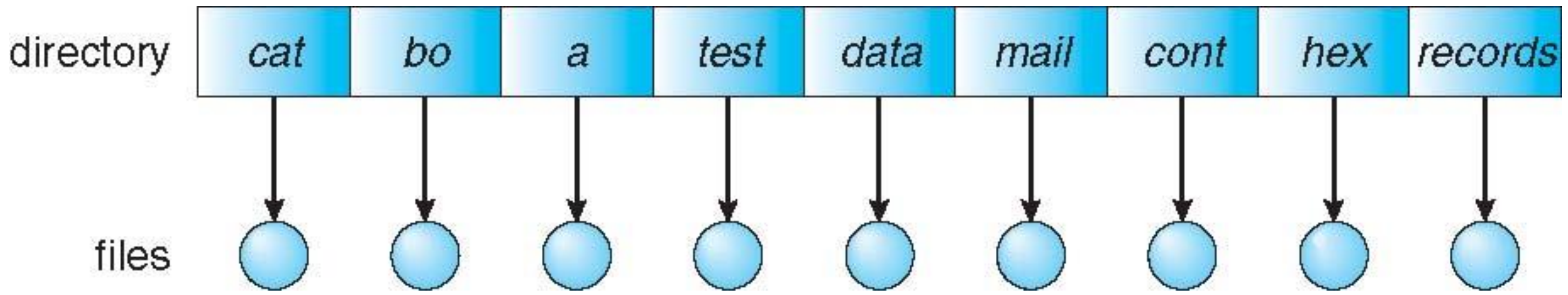
Purposes of directory structure

- Efficiency – locating a file or group of files quickly
- Naming – convenient & helpful to users
  - Users can pick same name for their (different) files without clashing
  - Same name for files of different types (different extensions)
  - Same file can have different names (multiple logical purposes; reference of same file/folder from different points in name space)
- Organization – logical grouping of files by various properties
  - Same user, project, purpose, type, …

Single-level directory

- A single directory for all the users

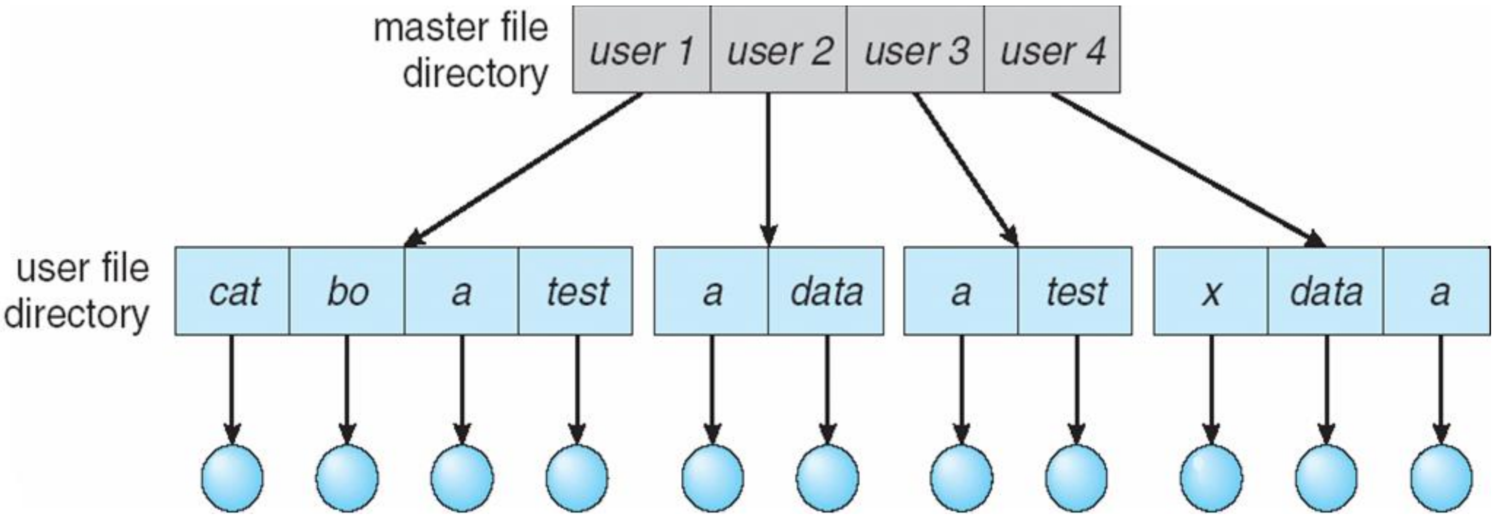
NB: circle nodes = files; square nodes = file names (in directory)



- Name clashes between users (users Tom and Amy can't pick same name for their files!)
- No logical grouping or organization

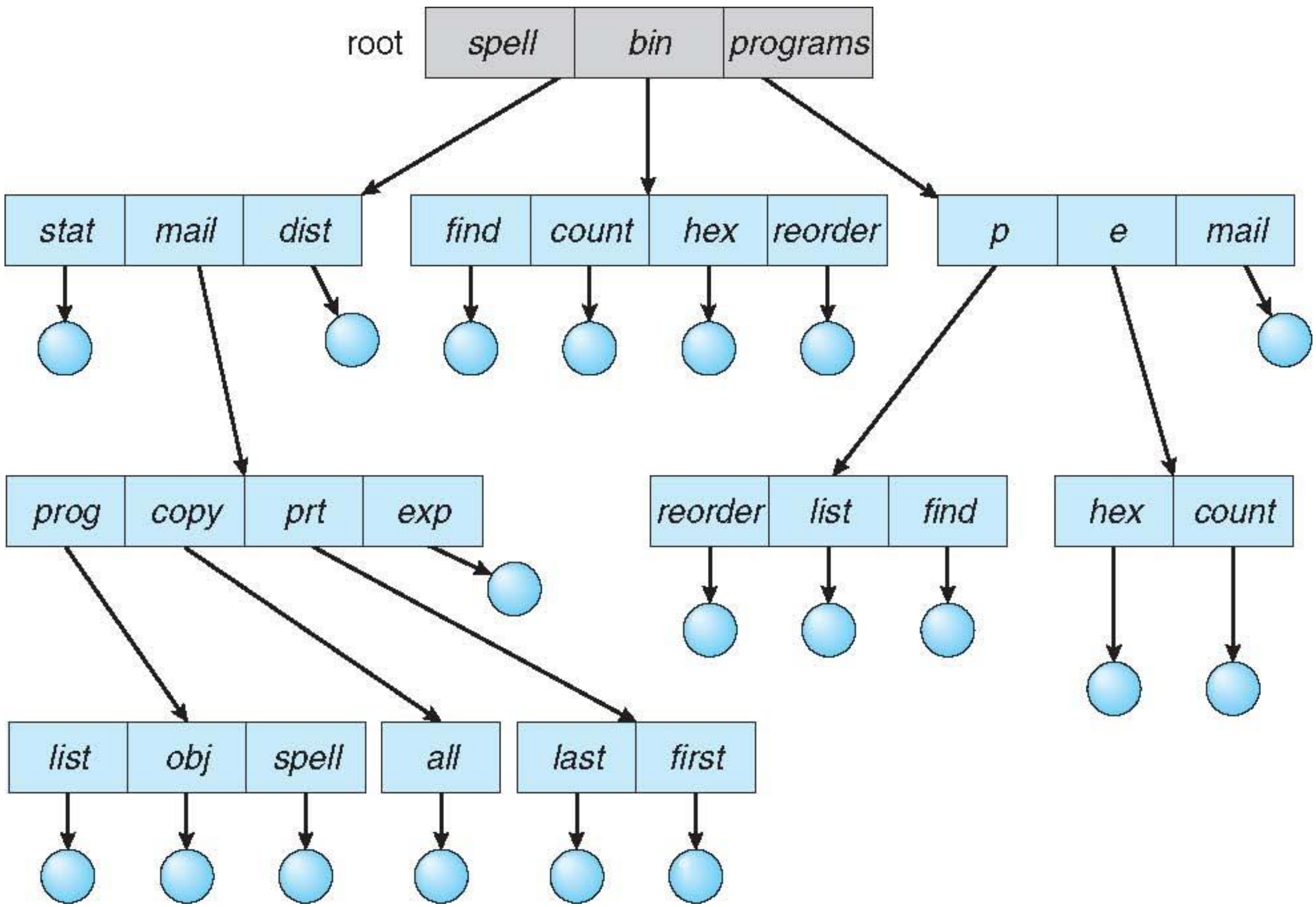
Two-level directory

- Separate directory for each user



- Notions of subdirectory and *path name* emerge, e.g., ***/user1/a***
- Each user has own separate name space (no clashes and more efficient search)
- Limited logical grouping
- Delete semantics – what happens when you delete non-empty directory?

Tree-structured directories



Tree-structured directories (cont'd)

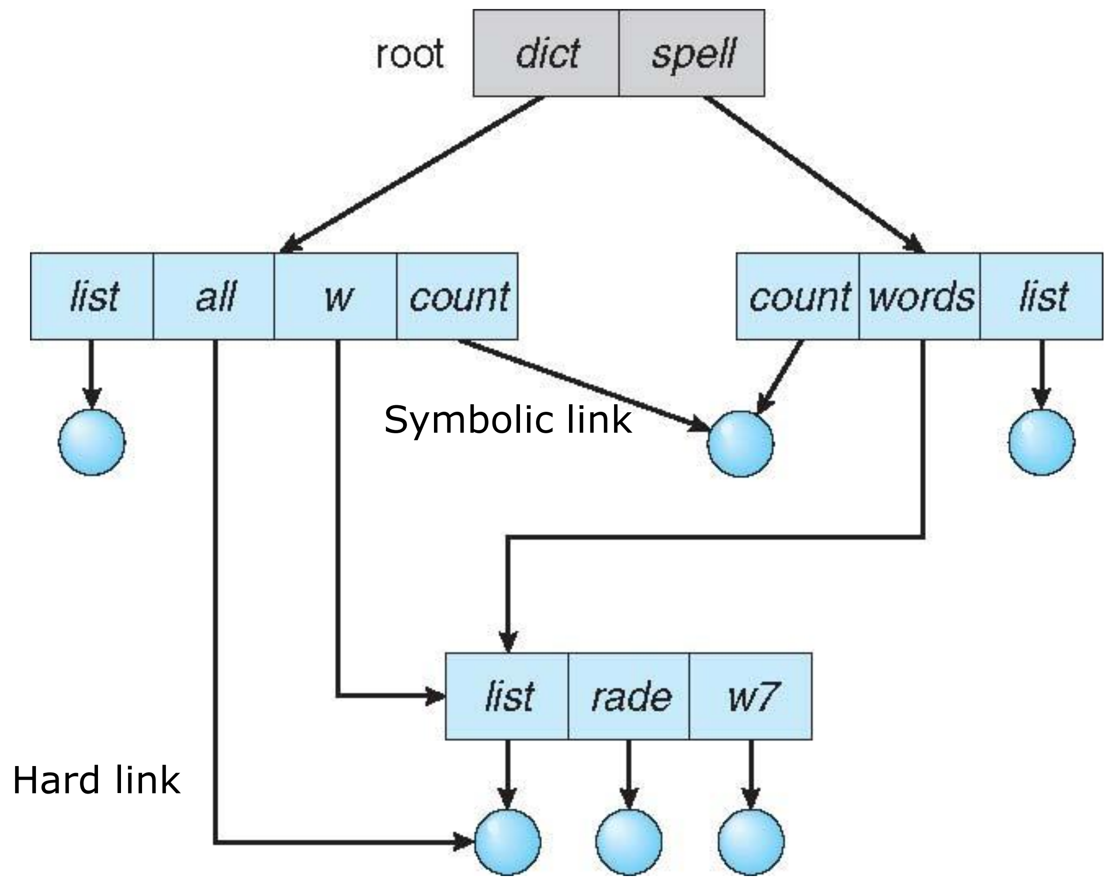
- Multiple levels of hierarchy allow more elaborate organization of files
  - But full path names can become long
    - e.g., /spell/mail/prt/first
  - Notion of *working directory* (wd) allows shorter *relative path names*
    - % cd /spell/mail/prt // wd is now /spell/mail/prt
    - % cat first // first is now relative to above wd
    - % cd ../copy // ../copy also relative to wd
  - Each process has its own current working directory
- So far, you can create a new file or subdirectory within a directory
  - But you can't point to an existing directory/file (i.e., you can't give an existing file another new name)

File links

- New type of file system objects, in addition to directories and files
  - Has path name in name space, just like files
  - But name *links* to another existing file system object
    - Hence, same object can now have multiple names (aliasing)
- In Unix, we have two flavors
  - **Symbolic links**, e.g. (last parameter is name of link): *same* file now has two names **/spell/count** and **/dict/count**
    - % ln -s /spell/count /dict/count
  - **Hard links**, e.g.,
    - % ln /dict/w/list /dict/all
    - **/dict/w/list** and **/dict/all** are again different names for the same file; in this case, they have no difference at all (can't tell which was created using **ln**, which one wasn't)

Acyclic graph directories

- Same file, multiple names (through symbolic/hard links)

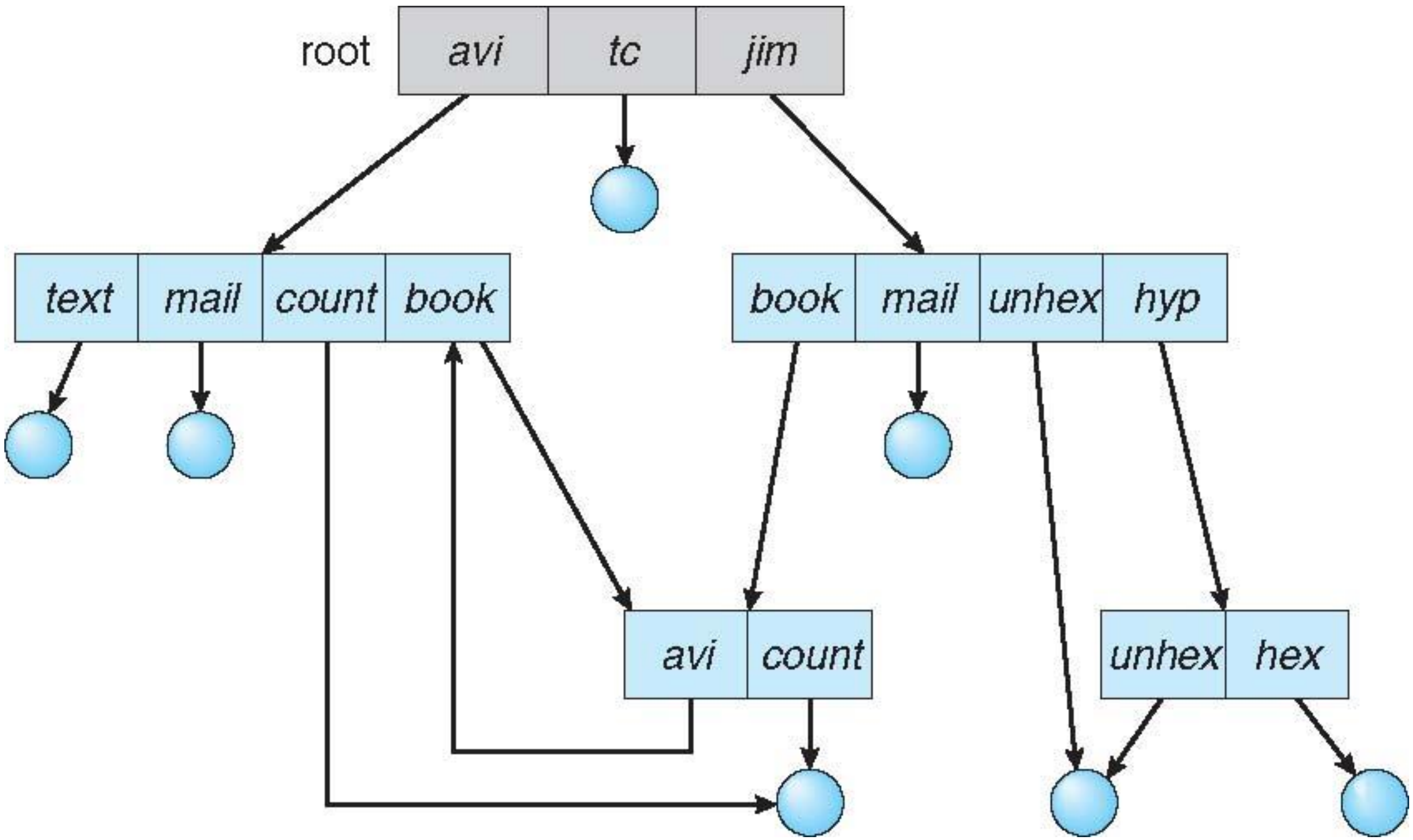


Acyclic graph directories (cont'd)

- What if we now delete /spell/count?
  - % rm /spell/count
  - Underlying file *is* removed
  - Symbolic link /dict/count remains, but becomes dangling pointer (name references non-existent file)
- What if we now delete /dict/w/list?
  - % rm /dict/w/list
  - Alternate name /dict/all keeps underlying file alive, i.e., file is *not* removed, exists under /dict/all only
  - Hard link increases reference count of file, file removed only if reference count becomes zero

General graph directory

What if you link to a *higher-level* subdirectory? What changes fundamentally?



General graph directory (cont'd)

- The directory structure becomes a general graph
  - i.e., *cycles* are possible
- Cycles can be tricky
  - Traversal by depth-first search, breadth-first search, etc, may not terminate
  - Reference count may not work – why?
    - Need garbage collection
  - Either check for and disallow cycles when creating a link, or deal with it during file system operations (e.g., traversals)

Activity 6.2: Linux file operations

- Get a Ubuntu Linux shell, and create a subdirectory called **working** under your home directory
- **cd** to **working**; create two text files **hello.txt** and **love.txt** using
  - *% echo ‘How are you’ > hello.txt*
  - *% echo ‘I love you’ > love.txt*
- Create links to **hello.txt** and **love.txt** using
  - *% ln -s hello.txt greet.txt*
  - *% ln love.txt like.txt*
- Remove **hello.txt** and **love.txt**, then **cat greet.txt** and **like.txt**. What happens?
- In **working**, create (by **mkdir**) 2 subdirectories **d1** and **d1/d2**, then **cd** to **d1/d2**
  - Create symbolic link **home** using *ln -s ../.. home***.** Does Ubuntu Linux allow cycles in your directory structure?
  - Now try *find . -name ‘*.txt’* as well as *find -L . ‘*.txt’.* What happens in each case? Can the **find** cope with the cyclic directory structure?
  - Try to make **home** a hard link instead. What happens? Does Linux allow hard links to create cycles?