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; file access methods; directory structures & symbolic/hard links; file protection

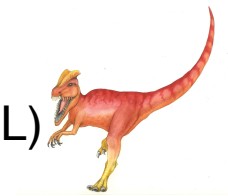
OS6: 29/2/2016
Textbook (SGG): Ch. 10.1, 10.2, 10.3.2-10.3.7, 10.6
(Assigned reading for this module will be included in the quiz on Feb 29, but not Feb 22.)

What is a file?

- Different kinds
  - Regular files (most familiar)
    - Store data – user data (your pics, music, emails, programs) or system data
  - Directories or folders (also familiar)
    - Organize files in a structured *name space* e.g., /Users/john/Desktop/work
  - *NB*. Also common for file system name space to name other objects – e.g. in Unix:
    - *procfs:* Let you peek inside the kernel, e.g., status of processes with ps(1)
    - Unix-domain sockets, named pipes, etc: for IPC
    - Block or character IO *devices* (try "ls -l /dev")
- Data usually stored in secondary storage (especially disks)
  - But not always, e.g., ramdisk (main memory) – new memory technologies (e.g., NAND flash) blurring 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 – uninterpreted sequence of words/bytes
- Simple record structure
  - Lines
  - Fixed length records
  - Variable length records
- Complex structures
  - Executable files (e.g., ELF format)
- Who interprets the format?
  - Operating system
    - 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
- Class/object has two key parts
  - State
    - Information about the files themselves – file data and meta-data (attributes)
    - Information about *usage* of the files
  - Interface (set of methods on the objects)

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

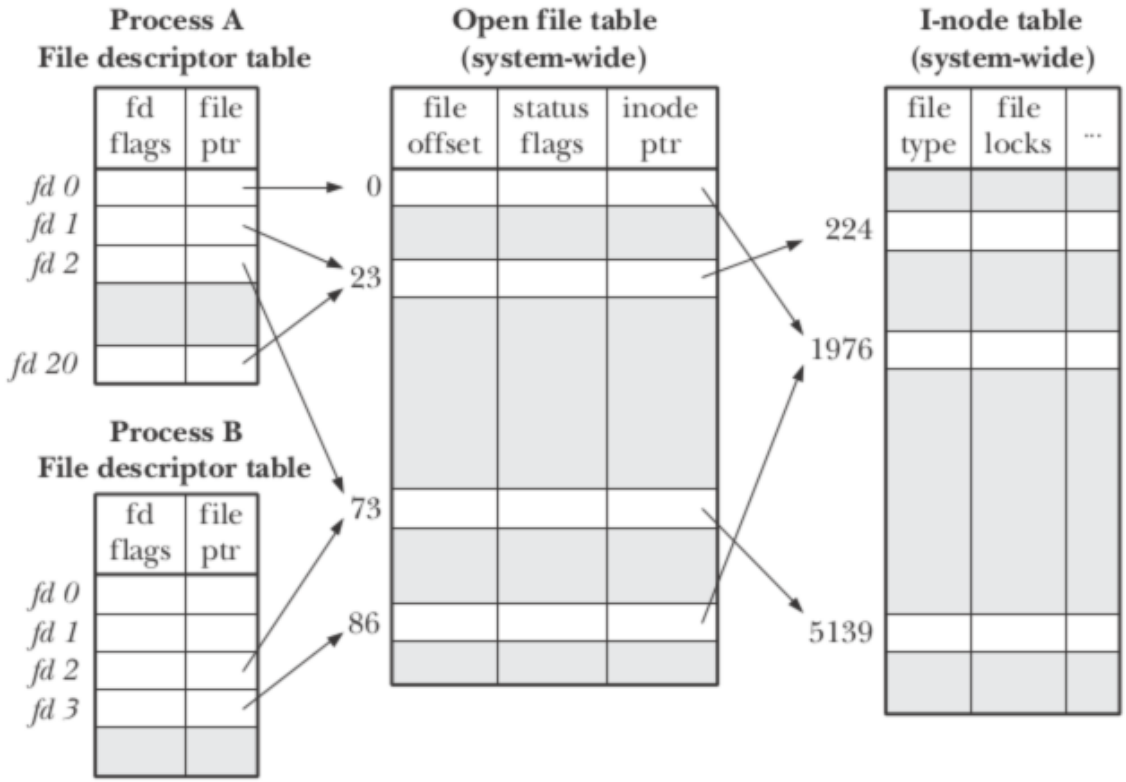
File interface (methods)

- Main operations on files
  - Create
  - Open
  - Read/write
  - Reposition within file (e.g., lseek() system call)
  - Memory map (map file into process's address space by mmap(2))
  - 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
  - Can be by explicit open() system call, but also other methods
- 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
  - 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) for a usage session
  - fd is (integer) index into *per-process* **file-descriptor table**
  - 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, e.g.,
  - Current file offset (*cp*): pointer to current read/write location
  - 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

Unix file system data structures



About *usage* of file

Mostly about file itself

NB. *inode* is Unix kernel's internal data structure for a file

NB: All these data structures are in kernel space

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 entry (e.g., dup() system call)
  - Different processes can also have their file descriptors point to same open file table entry
    - Child inherits parent's file descriptors after fork()
    - 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, read/write through *fd1* will affect *cp* seen by *fd2*

Mappings between table entries (cont'd)

- Multiple open file table entries can point to the same file (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
- Hence, in general, concurrent accesses to files are possible by different processes
  - Get a form of IPC
  - But be careful of race conditions
    - Shared read accesses are usually fine
    - Write access may need to be exclusive
    - OS may support shared and exclusive *file locks* that can be used where needed

Activity 6.1

- Consider the snapshot of Unix open files shown on Slide 10.13
- Redraw the tables after the following sequence of actions:
  - *B* forks process *C*
  - *C* closes fd 2
  - *C* opens file1976

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

File access methods

- Sequential access

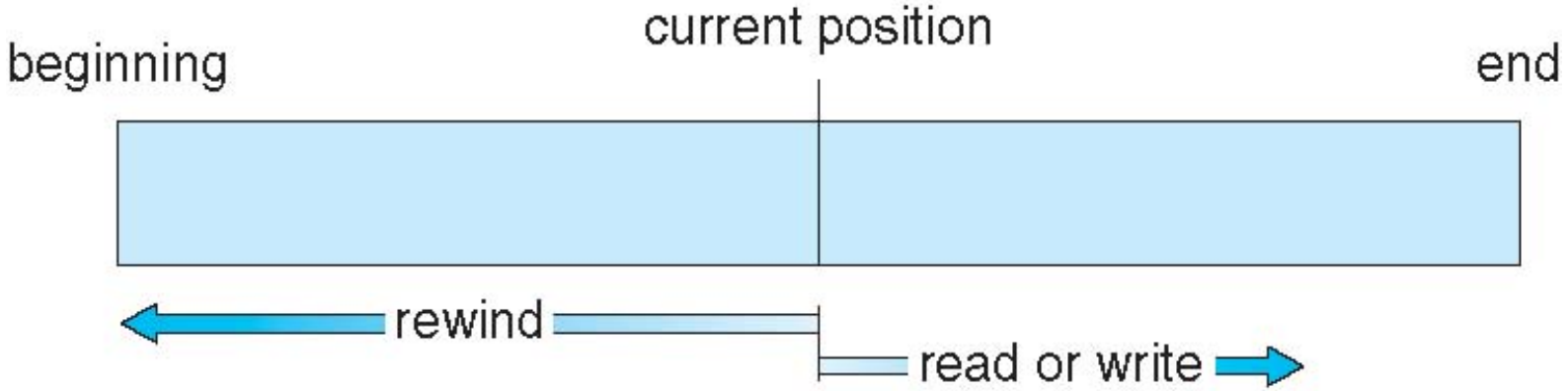
  read next
  write next
  reset

- Direct access (seek to specified position/offset *n* in file)

  read *n*
  write *n*
  position to *n*
  read next
  write next
  rewrite *n*

- Indexed access (allows search by one or more *keys*)

Sequential access of file

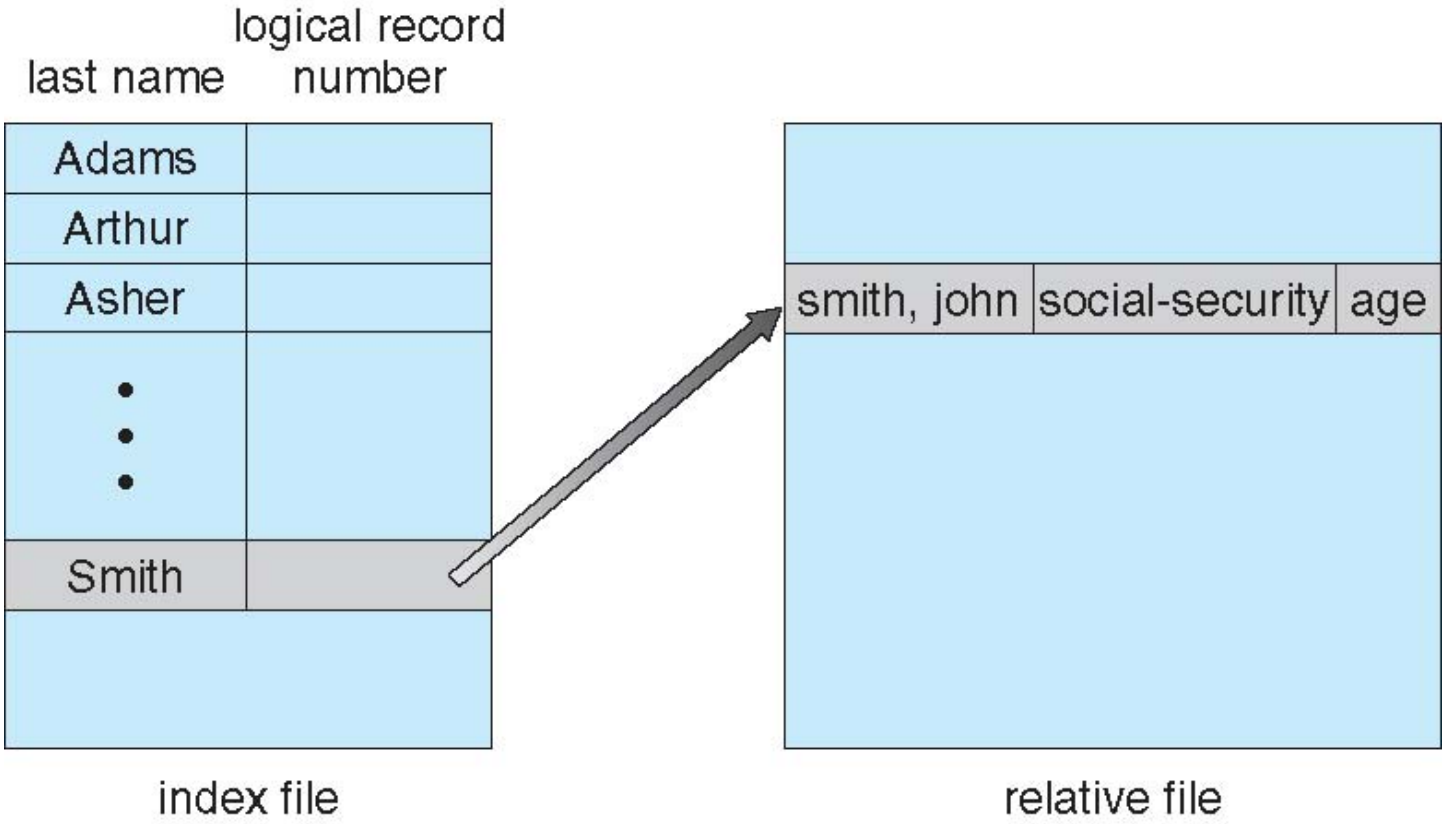


Sequential access by direct access

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

cp = current pointer (e.g., byte or record offset in file)

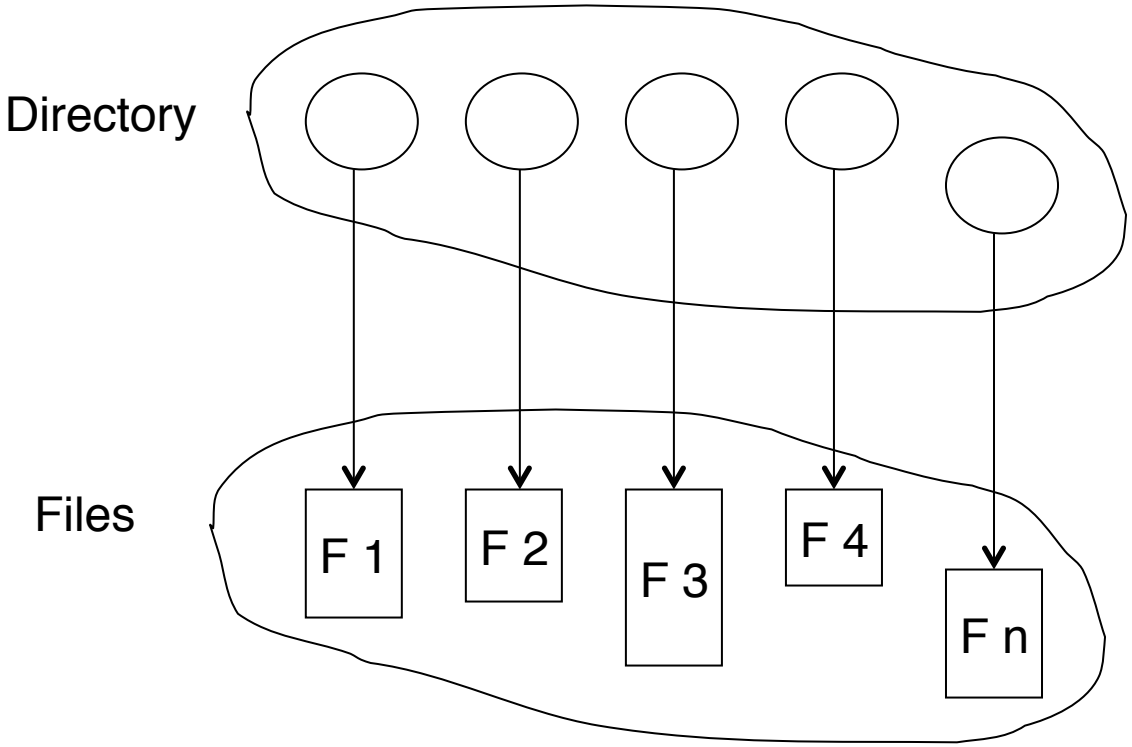
Indexed files



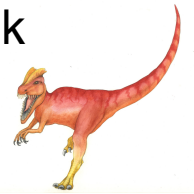
Supports efficient search by (sorted) search key (e.g., databases)
Same relative file can have multiple indices (several search keys)

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)

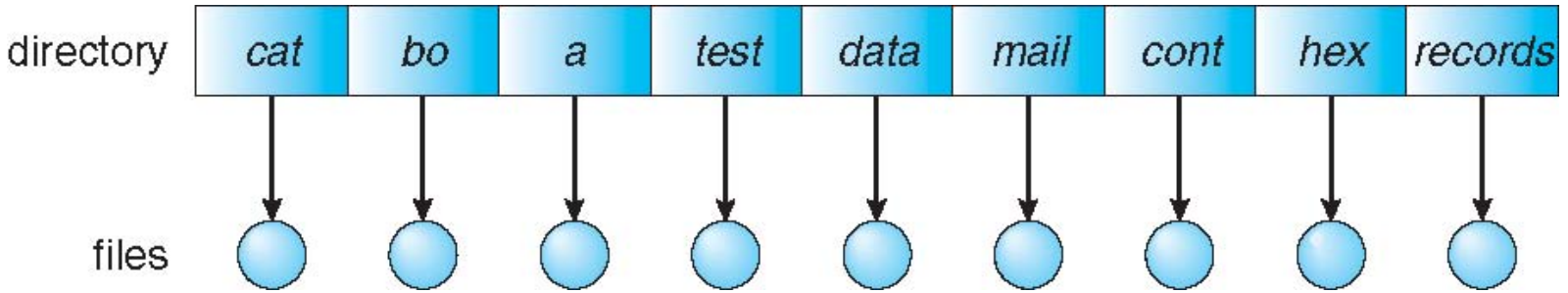
- Create a file or folder (subdirectory)
- Delete a file or folder
- List a directory
- Rename a file
- Search for a file
- Traverse the file system (beginning directory and all its subdirectories)
  - E.g., try Unix find(1) command from your shell …
    - % find .
    - % find . -name '*.java'
    -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

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

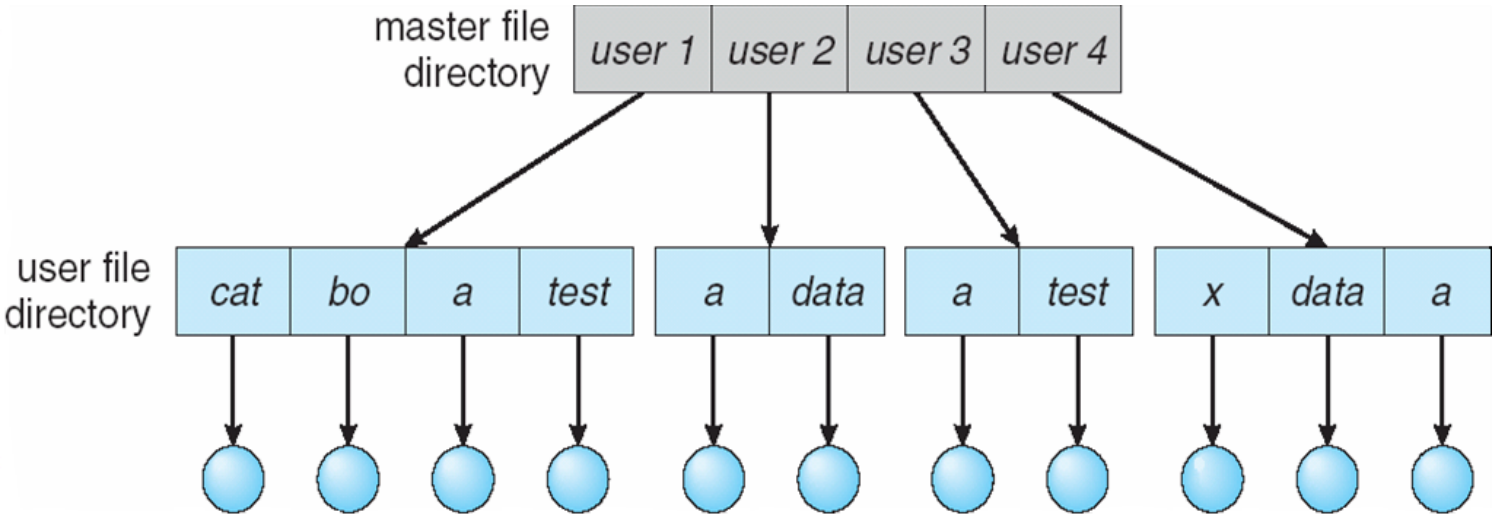
- A single directory for all the users



- Name clashes between users
- No logical grouping or organization

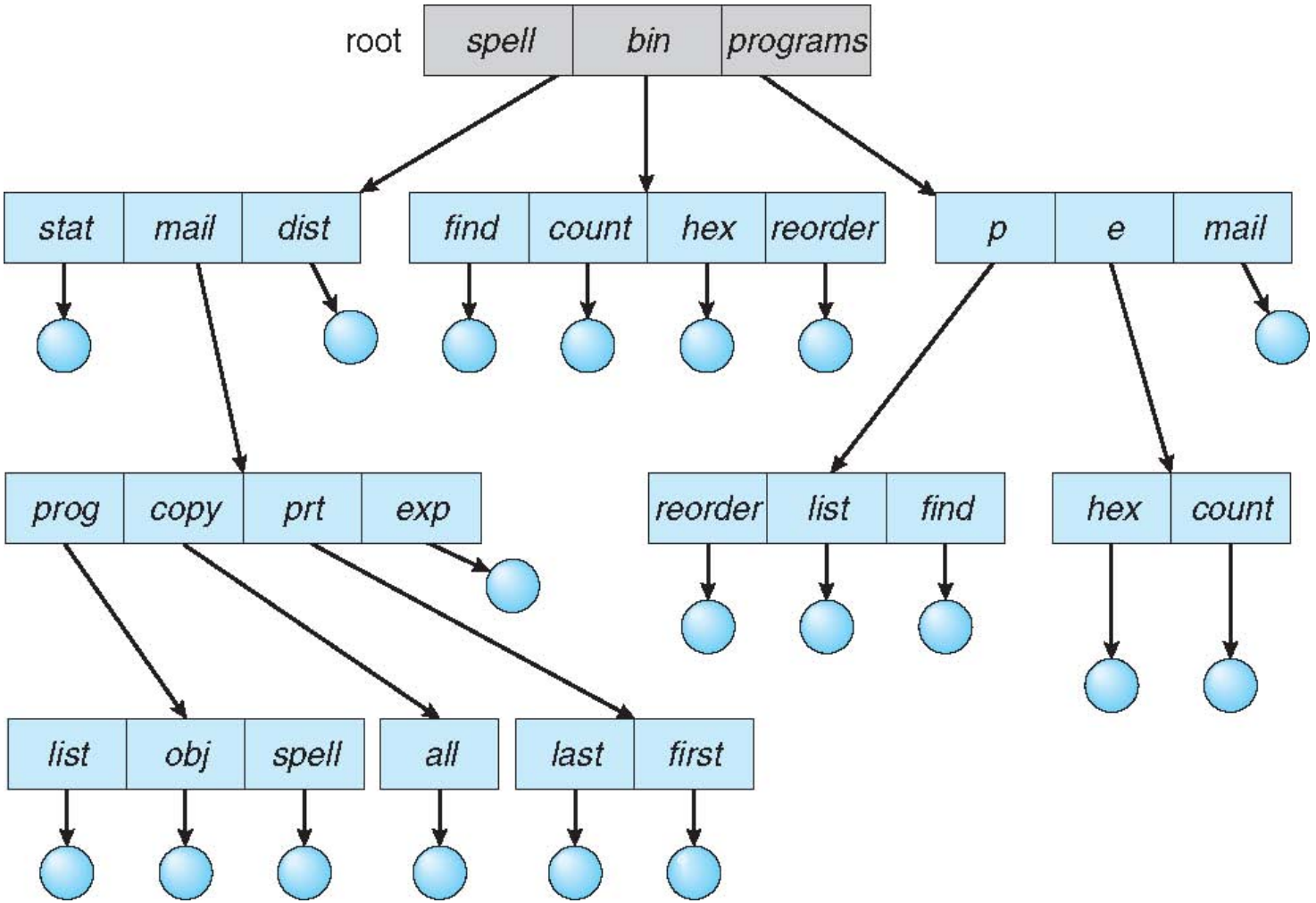
Two-level directory

- 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)

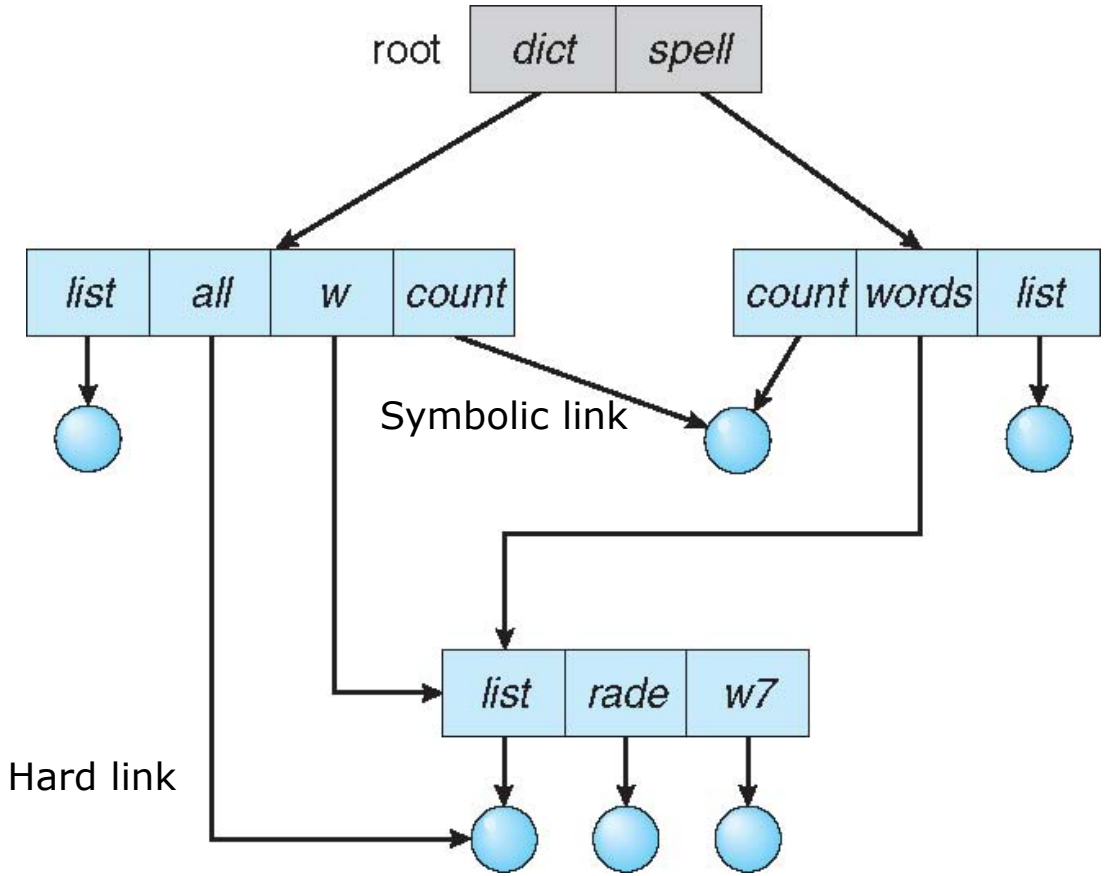
- Elaborate organization of files possible
  - But full path names can become long
    - e.g., /spell/mail/prt/first
  - Notion of *working directory* allows shorter *relative path names*
    - % cd /spell/mail/prt
    - % cat first
    - % cd ../copy
  - Each process has its current working directory
- You can create a file or subdirectory within a directory
  - But you can't point to an existing directory/file (i.e., give a file a new name without creating the file)

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),
    - % ln -s /spell/count /dict/count
  - Hard links, e.g.,
    - % ln /dict/w/list /dict/all
    - /dict/w/list and /dict/all 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)

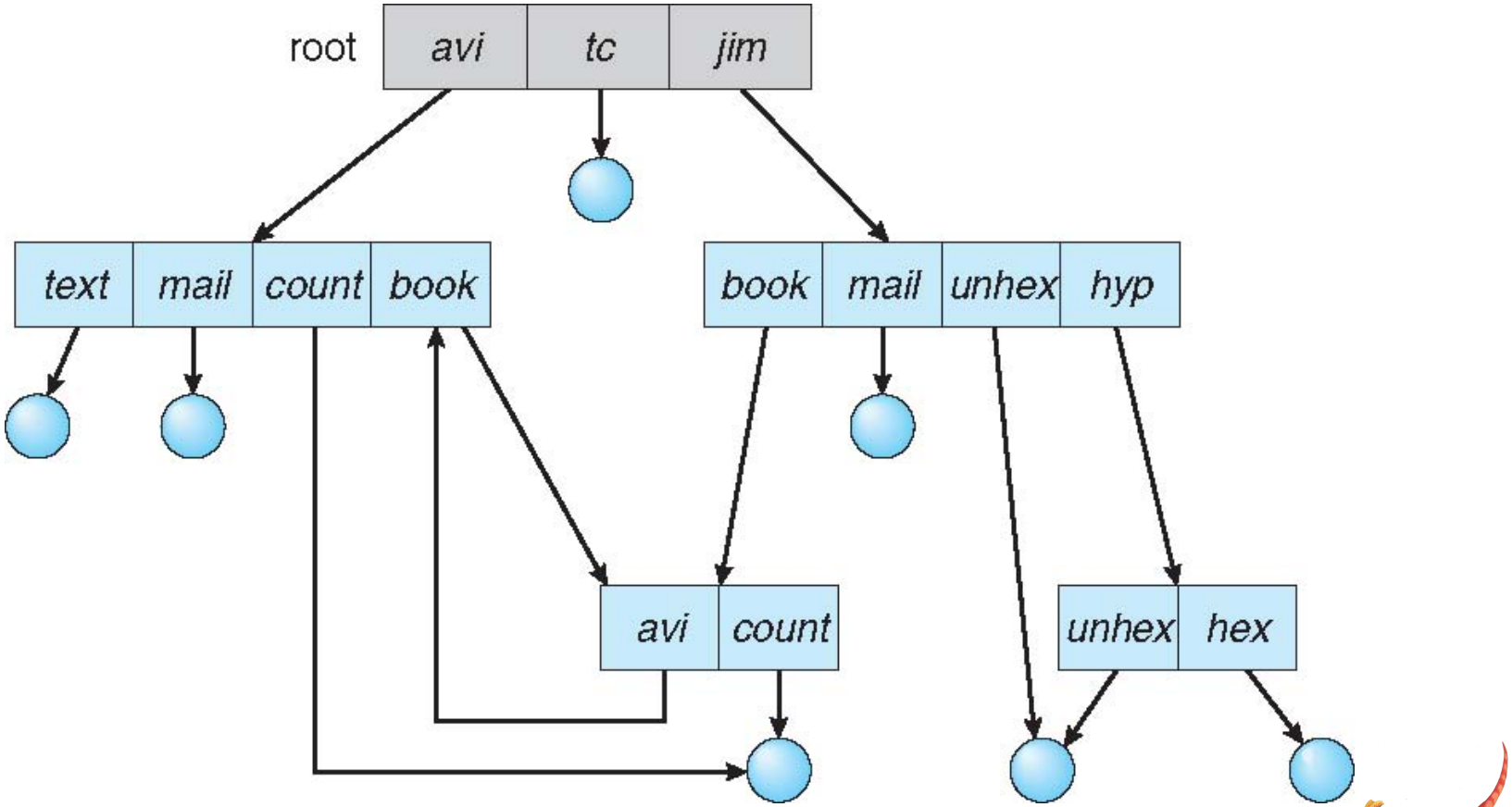


Acyclic graph directories (cont'd)

- 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 **working** subdirectory under your home directory
- **cd** to **working**; create two text files **hello.txt** and **love.txt**
  - *% 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?
- Under **working**, create two 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?

File protection

- File system name space is in general *global*
  - User *A* can access more than her own files – e.g., files owned by another user *B*, or files that belong to the OS
  - But subject to *protection* constraints (access rights granted usually by owners)
- Access control list (ACL) specifies what's allowed

Unix file access control

- Three access modes: read, write, execute
- Three classes of users

		RWX
owner access	7 ⇒	1 1 1
group access	6 ⇒	1 1 0
public access	1 ⇒	0 0 1

- System administrator creates group (identified by integer gid) and adds users to it, e.g.,
  - % chgrp gid game.exe
- Owner sets access rights for file, e.g.,
  - % chmod 761 game.exe
- What does ‘X’ access mean for regular file vs. directory?

Unix directory listing w/ protection info

```
-rw-r-----@ 1 david    staff  565181   Sep 21 21:57 AMEX.pdf
drwxrwxr-x 3 david     staff     102   Nov 11  2014 Adobe
drwxrwxr-x 4 david     staff     136   Nov 11  2014 Adobe Flash Builder 4.6
-rw-r--r--@ 1 david    staff   32993   Feb  7  2015 CISDeMContractPg22.pdf
-rw-r--r--@ 1 david    staff  913631   Nov 13  2014 huawei.pdf
drwxr-xr-x  3 david    staff     102   Nov 11  2014 Microsoft User Data
-rw-r--r--@ 1 david    staff   29407   Aug 22  2015 MyNotes.docx
-rw-r--r--@ 1 david    staff   21778   Jul  31  2015 MyString.docx
-rw-r-----@ 1 david    staff   16671   May  8  2015 Receipt2015.pdf
```