Today

- Last Project 0 questions
  - Segfaults & debugging
- Byte-level representation of values
- Bit manipulation
- System calls
- More library functions

Project 1 has been released (we will talk about it on Thursday)

Today

- Project 1
- Byte-level representation of values (int and float)
- System calls
- File system
- Bit manipulation

memset()

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

- Set the first n bytes in s to the value c
- Returns s

- Good for initializing buffers with a constant

memcpy()

Low-level byte copy

```
#include <string.h>
void *memcpy(void *dest, const void *src, size_t n);
```

- Copy n bytes from src to dest

scanf()

Parsing formatted input from STDIN

```
int i;
float f;
int ret = scanf("%d %f", &i, &f)
```

- Format string: same meaning as in printf()
- Fills in the values for i and f
- Returns the number of arguments that have been parsed. If this number does not match the number you expect, then something went wrong.

sscanf()

Parsing formatted input from a character buffer

```
int i;
float f;
char buffer[200];

// Buffer has been filled with a string
int ret = sscanf(buffer, “%d %f”, &i, &f)
```

File Systems

Data Storage Challenges

For any storage system, we have to answer questions such as:

- How will new data be stored? How do we select its location?
- When we want to retrieve data, how do we find this data and access it?

What matters:

- Efficiency in storage and access
- Integrity
- Volume of data
- Ease of access, even when faced with many different physical implementations

The Type of Application Matters

Different applications have different requirements for storage:

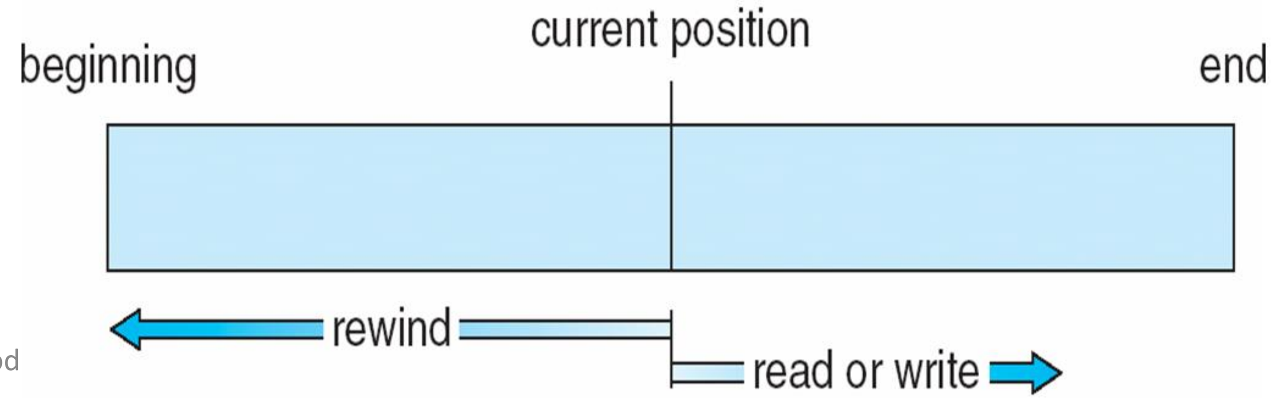
- Data collection: quickly storing data when it arrives in big bursts
- Databases: often highly-structured data
  - Rapid look-up by key (or multiple keys)
- Many other apps: semi-structured

File Concept

- Contiguous, logical address space
    - At the lowest level, each address just contains a byte of data
- At the more abstract side, files contain:
    - Data
        - numeric
        - character
        - binary
    - Program
- Contents defined by file's creator
    - Many types
        - Consider **text files, source files, executable files**

Low-Level Representation of a File (or a Stream)

- Sequence of bytes
- Current position tells us where in the file we are currently at. Formally called the ***file offset***
  - Sequential access: next read / write operation will access the file at this point and then advance the current position by one
  - We can also programmatically change the current location



File Attributes

Files have a set of attributes that describe the details of the file. These attributes are stored with the file.

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size

File Attributes

- **Protection**: controls who can do reading, writing, executing
- **Time, date, and user identification**: data for protection, security, and usage monitoring

- Information about files are kept in the directory structure, which is maintained on the disk
  - Many variations, including extended file attributes such as file checksum

File info Window on Mac OS X

Standard File Operations

File is an **abstract data type!**

- **Create**
- **Write:** at **write offset** location
- **Read:** at **read offset** location
- **Reposition within file: seek**
- **Delete**
- **Truncate**
- ***Open(F_i)*** – search the directory structure on disk for entry F_i, and move the content of entry to memory
- ***Close (F_i)*** – move the content of entry F_i in memory to directory structure on disk

Low-Level File/Stream Identification

File descriptor:

- A nonnegative integer that may refer to:
  - Regular files, pipes, FIFOs, sockets, terminals or devices
- Each process has its own assigned set of file descriptors
- Used by the system to refer to files that are open

Standard File Descriptors

- When a process starts executing, it is generally given three standard file descriptors that are already open
  - This includes programs that are started by your shell
- Standard In: input into the process. Bytes are received through functions such as getchar() or scanf()
- Standard Out: default output from the process. puts(), printf()
- Standard Error: separate output for error information only. fputs(), fprintf()

File descriptor	Purpose	POSIX name	*stdio* stream
0	standard input	STDIN_FILENO	*stdin*
1	standard output	STDOUT_FILENO	*stdout*
2	standard error	STDERR_FILENO	*stderr*

Key Low-Level I/O System Calls

fd = open(pathname, flags, mode)	• opens the file identified by pathname, returning a file descriptor.
numread = read(fd, buffer, count)	• reads at most count bytes from the open file referred to by fd and stores them in buffer.
numwritten = write(fd, buffer, count)	• writes up to count bytes from buffer to the open file referred to by fd.
status = close(fd)	• is called after all I/O has been completed, in order to release the file descriptor fd and its associated kernel resources.

Open

Opens the file identified by *pathname*, returning a file descriptor.

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags, ... /* mode_t mode */);
```

Returns file descriptor on success, or −1 on error

Open

Listing 4-2: Examples of the use of *open()*

```
/* Open existing file for reading */

fd = open("startup", O_RDONLY);
if (fd == -1)
    errExit("open");

/* Open new or existing file for reading and writing, truncating to zero
   bytes; file permissions read+write for owner, nothing for all others */

fd = open("myfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");

/* Open new or existing file for writing; writes should always
   append to end of file */

fd = open("w.log", O_WRONLY | O_CREAT | [redacted] O_APPEND,
                   S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");
```

Flag	Purpose
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing
O_CLOEXEC	Set the close-on-exec flag (since Linux 2.6.23)
O_CREAT	Create file if it doesn't already exist
O_DIRECT	File I/O bypasses buffer cache
O_DIRECTORY	Fail if *pathname* is not a directory
O_EXCL	With O_CREAT: create file exclusively
O_LARGEFILE	Used on 32-bit systems to open large files
O_NOATIME	Don't update file last access time on *read()* (since Linux 2.6.8)
O_NOCTTY	Don't let *pathname* become the controlling terminal
O_NOFOLLOW	Don't dereference symbolic links
O_TRUNC	Truncate existing file to zero length
O_APPEND	Writes are always appended to end of file
O_ASYNC	Generate a signal when I/O is possible
O_DSYNC	Provide synchronized I/O data integrity (since Linux 2.6.33)
O_NONBLOCK	Open in nonblocking mode
O_SYNC	Make file writes synchronous

File Permissions: Can Be Or'ed Together

	Read	Write	Execute
Owner/User	S_IRUSR	S_IWUSR	S_IXUSR
Group	S_IRGRP	S_IWGRP	S_IXGRP
Others	S_IROTH	S_IWOTH	S_IXOTH

A Note

- errExit() in the previous example is not a standard function
- Instead, use the following:

```
perror("some string to describe your context");
exit(-1);
```

- This will print out your message, a description of the error that occurred in the last system call and then terminate your program

Read

Reads at most *count* bytes from the open file referred to by *fd* and stores them in *buffer*.

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t count);
```

Returns number of bytes read, 0 on EOF, or –1 on error

Read

```
#define MAX_READ 20
char buffer[MAX_READ];

if (read(STDIN_FILENO, buffer, MAX_READ) == -1)
    errExit("read");
printf("The input data was: %s\n", buffer);
```

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t count);
```

Returns number of bytes read, 0 on EOF, or –1 on error

Read

```
char buffer[MAX_READ + 1];
ssize_t numRead;

numRead = read(STDIN_FILENO, buffer, MAX_READ);
if (numRead == -1)
    errExit("read");

buffer[numRead] = '\0';
printf("The input data was: %s\n", buffer);
```

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t count);
```

Returns number of bytes read, 0 on EOF, or −1 on error

Write

Writes up to *count* bytes from *buffer* to the open file referred to by *fd*.

```
#include <unistd.h>

ssize_t write(int fd, void *buffer, size_t count);
```

Returns number of bytes written, or –1 on error

Close

Called after all I/O has been completed, in order to release the file descriptor *fd* and its associated kernel resources.

```
#include <unistd.h>

int close(int fd);
```

Returns 0 on success, or −1 on error

File Offset

- Also called read/write offset or pointer
- The kernel records a file offset for each open file.
- The file offset is set to point to the start of the file (0) when the file is opened and is automatically adjusted by each subsequent call to read() or write()

File Access

- Sequential Access:
  - Start at beginning of file
  - Each read/write of a byte advances the file offset by one
- Direct Access (or Random Access):
  - Before a read/write operation, move the offset to the right point in the file

Seeking

Change the file offset for the specified file

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Returns new file offset if successful, or –1 on error

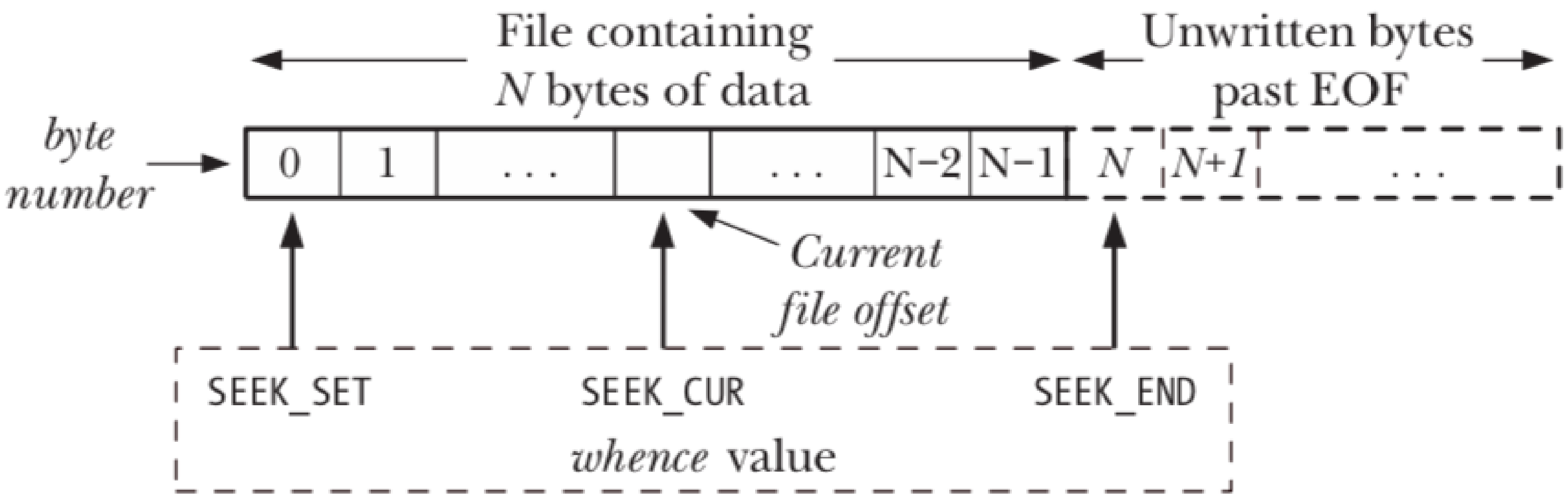


Figure 4-1: Interpreting the *whence* argument of *lseek()*

lseek() Examples

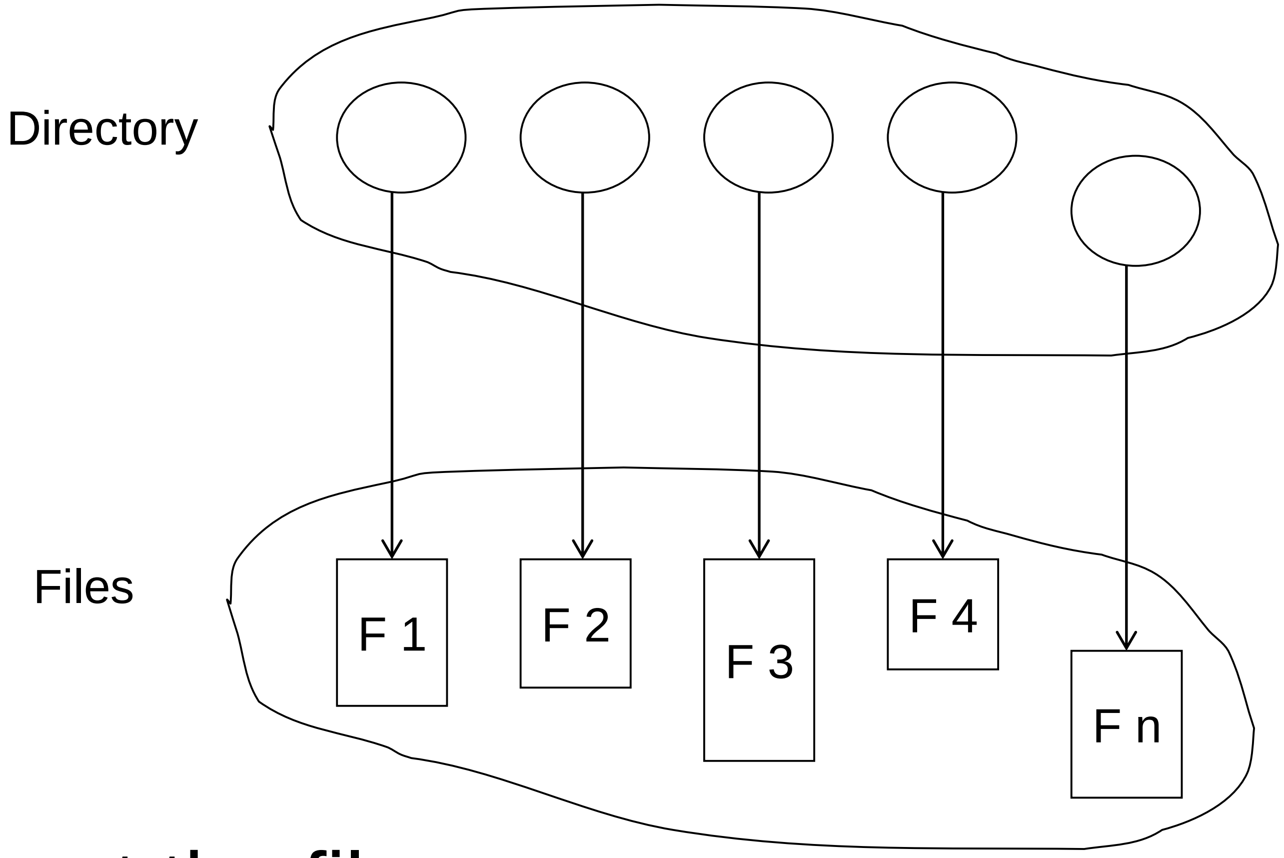
C

```c
lseek(fd, 0, SEEK_CUR); /* Returns current cursor loc of without change */
lseek(fd, 0, SEEK_SET); /* Start of file */
lseek(fd, 0, SEEK_END); /* Next byte after the end of the file */
lseek(fd, -1, SEEK_END); /* Last byte of file */
lseek(fd, -10, SEEK_CUR); /* Ten bytes prior to current location */
lseek(fd, 10000, SEEK_END); /* 10001 bytes past last byte of file */
```

Directory Structures

Fundamental challenge: how do we find the file that we are looking for?

Directory Structures



Directory:

- Container for a set of files
- Stores meta-information about the files

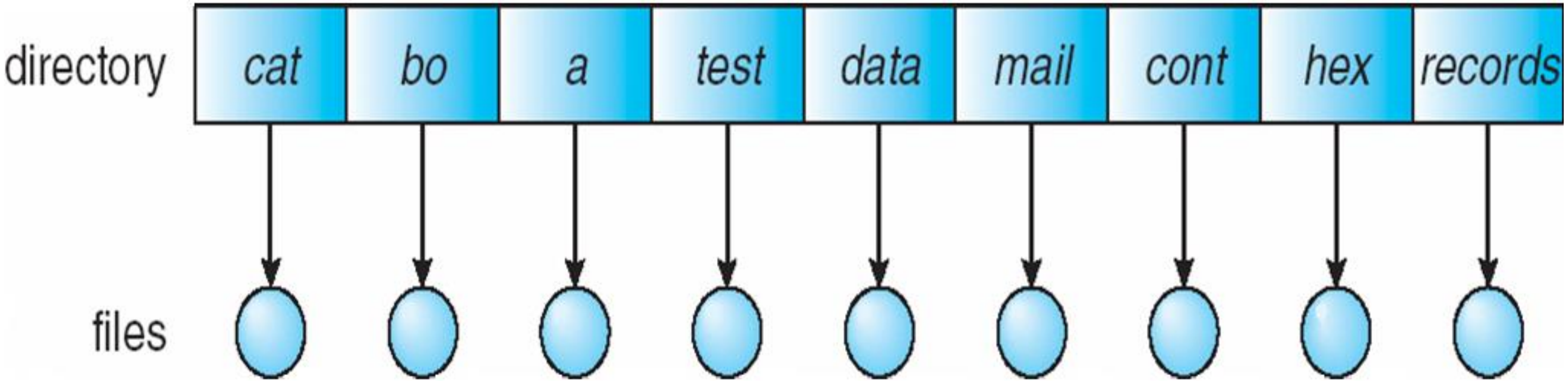
Directory Organization

The directory is organized logically to obtain:

- Efficiency: locating a file quickly
- Naming: convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- Grouping: logical grouping of files by properties, (e.g., all Java programs, all games, …)

Single-Level Directory

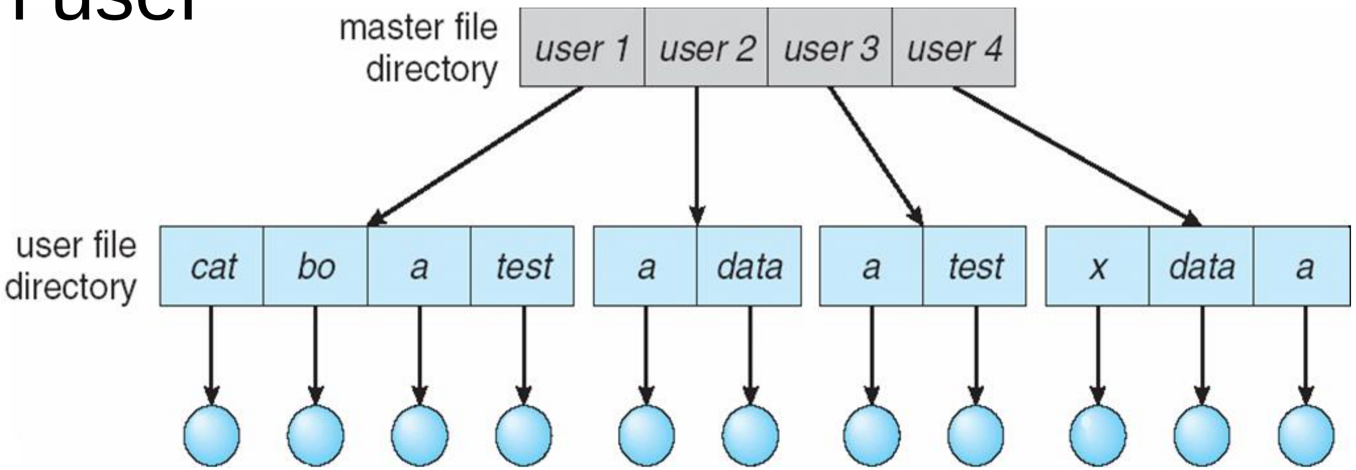
A single directory for all users



- Problems: does not support convenient naming and grouping

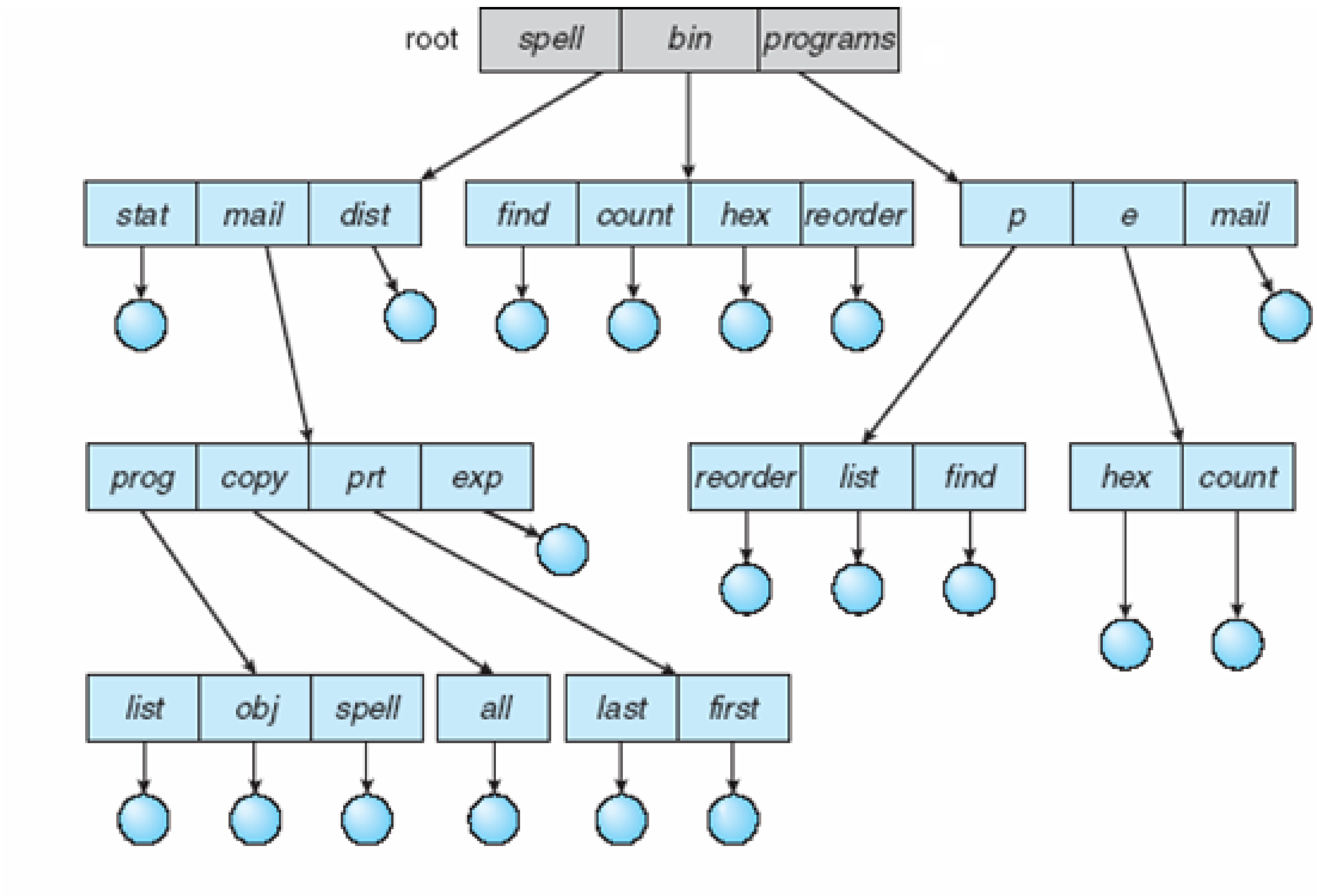
Two-Level Directory

Separate directory for each user



- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree-Structured Directories



Tree-Structured Directories

- Efficient searching
- Grouping capability
- Efficient use:
  - Each process has a notion of a current working directory

Tree-Structured Directories

- **Absolute** or **relative** path name
- Default behavior: creating a new file is done in current directory
- Delete a file

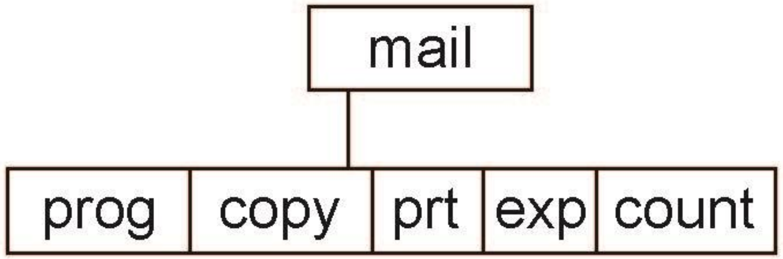
```
rm <file-name>
```

- Creating a new subdirectory is done in current directory

```
mkdir <dir-name>
```

Example: if in current directory **/mail**

```
mkdir count
```

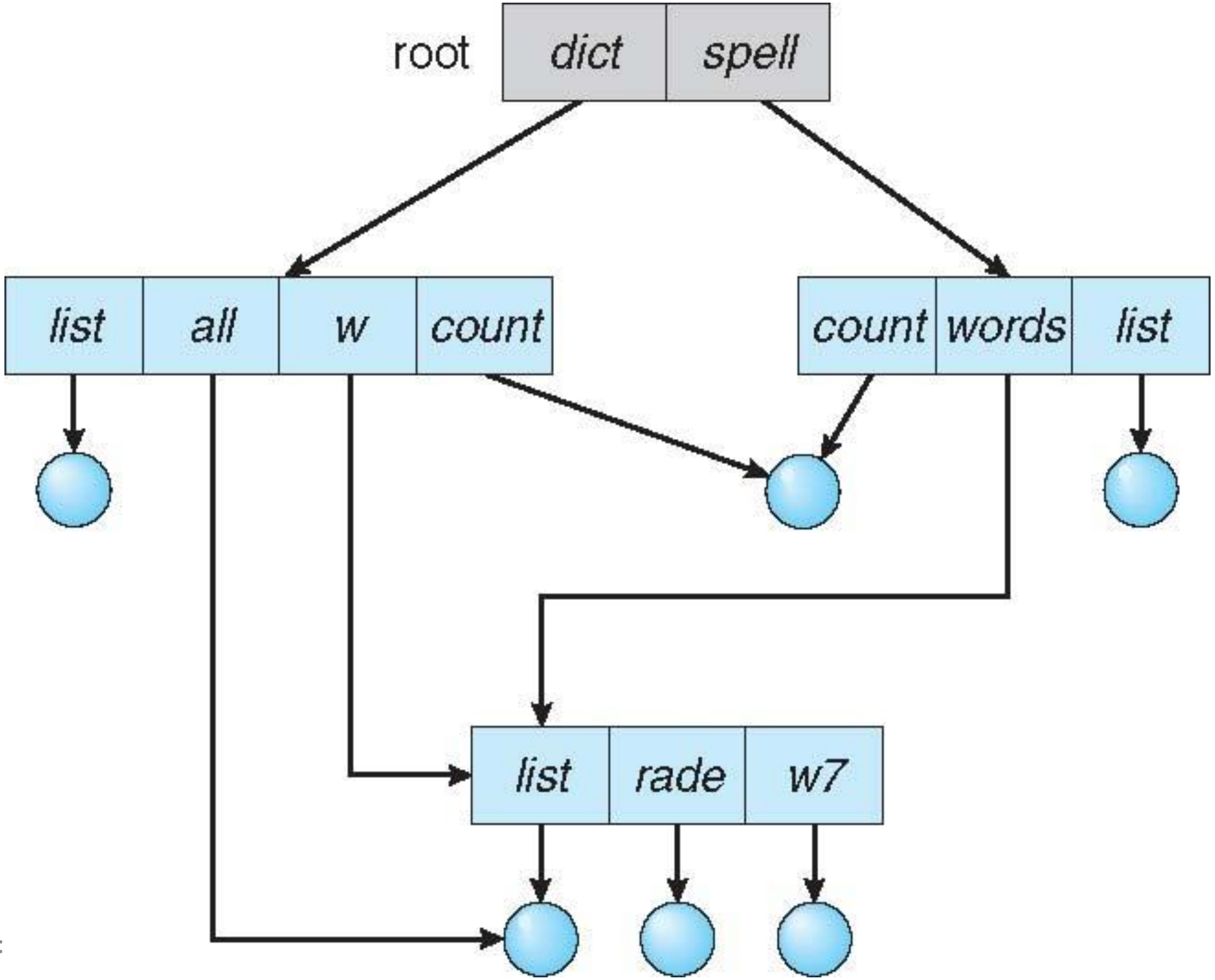


- Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”

Acyclic-Graph Directories

- Circles: files
- Rectangles: directories

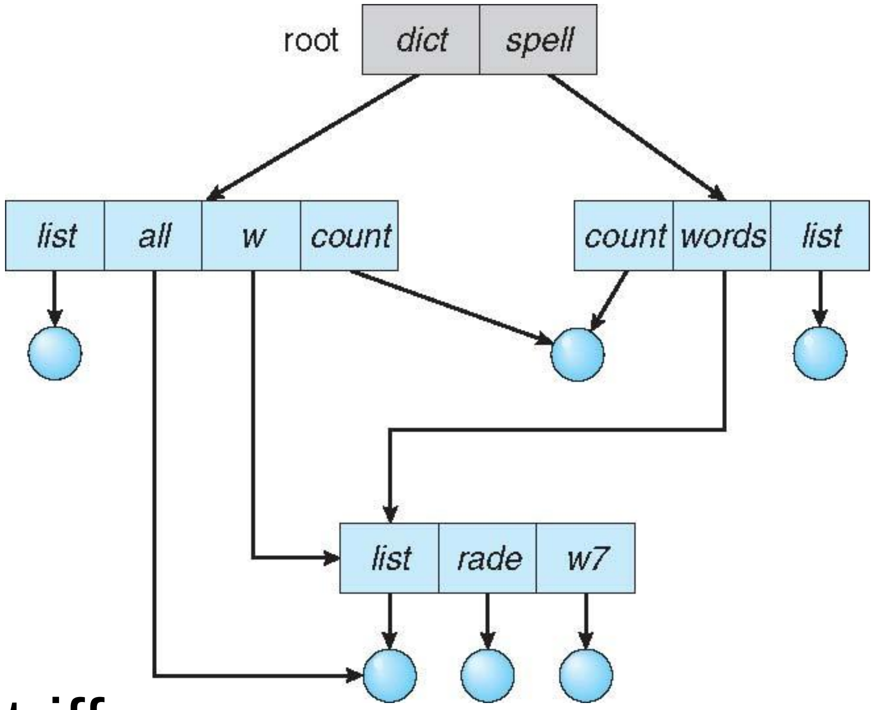
- The same file can be contained within multiple directories



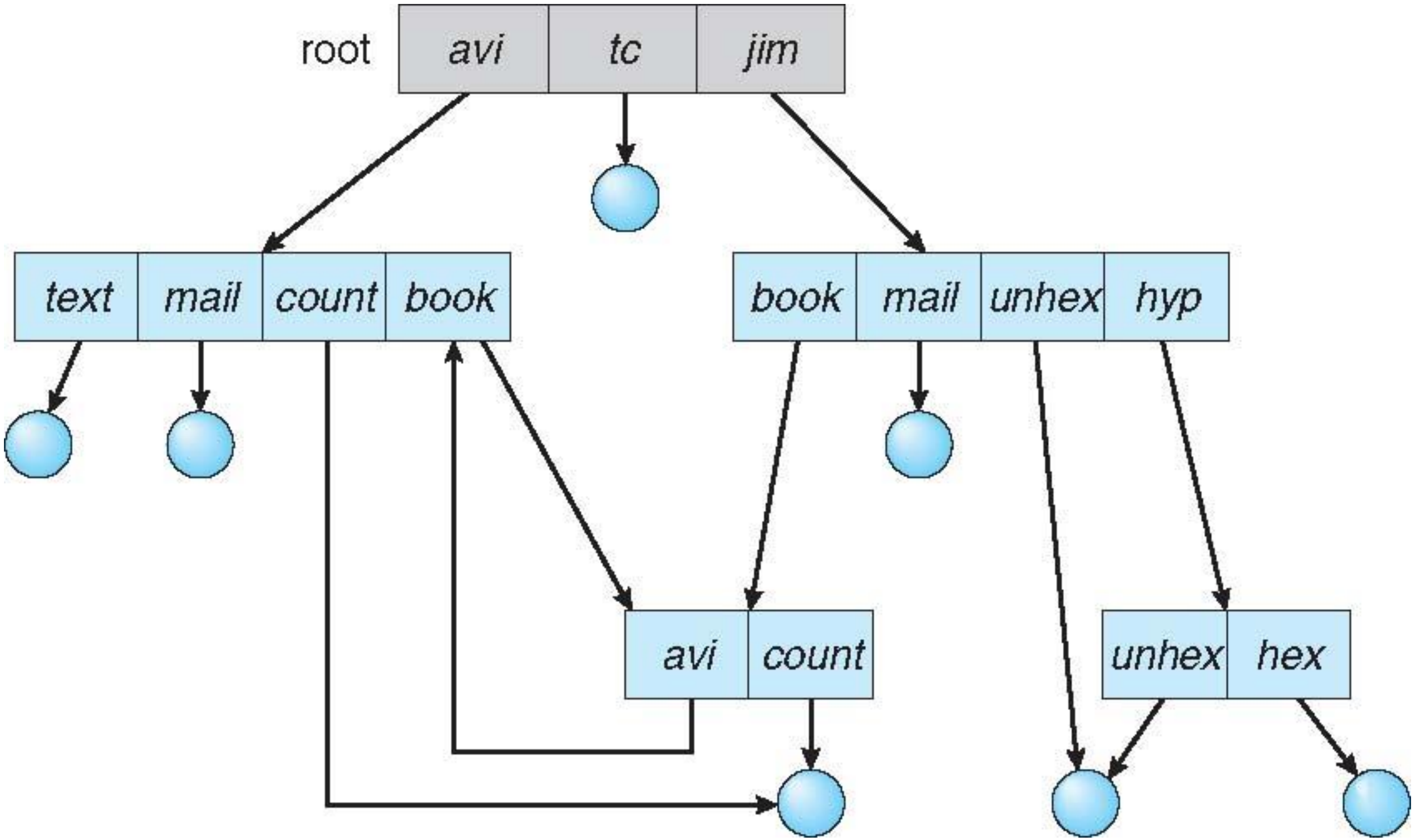
Acyclic-Graph Directories

- Two different names for the same element (aliasing)
- If ***dict*** deletes ***count*** $\Rightarrow$ dangling pointer

  Solutions:
  - Backpointers solution: identify all containing directories and delete all entries
  - Entry-hold-count solution: remove the element iff there are no other links to it



General Graph Directory



General Graph Directory

How do we guarantee no cycles?

- Allow only multiple links to files, not subdirectories
- Every time a new link is added use a cycle detection algorithm to determine whether it is OK

File/Directory Protection

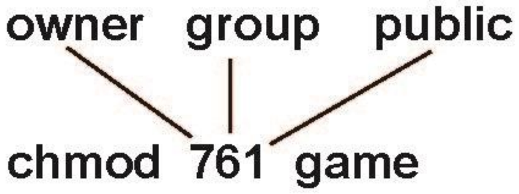
- File owner/creator should be able to control:
  - what can be done
  - by whom
- Common types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

Access Lists and Groups

- Common modes of access: read, write, execute
- Three classes of users on Unix / Linux

			RWX
a) **owner access**	7	$\Rightarrow$	1 1 1
			RWX
b) **group access**	6	$\Rightarrow$	1 1 0
			RWX
c) **public access**	1	$\Rightarrow$	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



A Sample UNIX Directory Listing

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

Another Look at Files

File descriptors + open/close/read/write: a low-level mechanism for representing and operating on a file (or stream)

- Every write is immediate: all written data are sent to the file
  - This can be problematic if we are calling write() for individual bytes
- Limited support for formatting of data (especially for translating raw data into strings of characters)

STDIO Library

The STDIO library adds another level of abstraction

- In-memory buffering for read/write operations
- API is more user-friendly
- Higher-level mechanisms for performing formatted I/O
    - printf(), fprintf(), sprint()
    - scanf(), fscanf(), sscanf()
    - fopen()
    - fclose()
    - fflush()

File Descriptors vs File Pointers

- File descriptor:
  - int type that references a table of open streams
  - Can reference files, pipes or sockets (more on the middle soon; latter is for inter-process communication)
  - Access through system calls: open(), read(), write(), close() …
- File pointer (FILE*)
  - FILE type defined in stdio.h (it is a struct)
  - Includes the file descriptor, but adds buffering and other features
  - Access through the stdio library: fopen(), fread(), fwrite(), fclose(), fprintf(), fscanf()
  - When working with files, this is the preferred interface

File Pointer Example

```
#include <stdio.h>

int main(int argc, char** argv)
{
  FILE* fp = fopen(argv[1], "w");
  if(fp == NULL){
    printf("Error opening file.\n");
  }else{
    fprintf(fp, "Foo bar: %s\n", argv[1]);
    fclose(fp);
  }
}
```

Another File Open Function …

```
FILE *freopen(const char *path, const char *mode, FILE *stream);
```

- Opens the specified file and associates it with the <stream> FILE
- If <stream> is already an open file, then it is closed first
- Returns <stream> if successful

Useful for substituting a file for the stdin stream

Flushing Streams

- Because FILE streams are buffered, a `fprintf()` does not necessarily affect the file immediately
- Instead, the bytes are dropped into a buffer; at some point the kernel will decide to move the bytes from the buffer to the file
- `fflush(fp)` will immediately force all bytes in the buffer to the file

Good Practices

- Generally, you should not mix use of file descriptors and file pointers (FILE*)
  - Since file pointers do buffering, things written to the corresponding file descriptor directly can “jump” ahead of things written to the file poiner
  - It can be stochastic as to which arrives first
- Instead, you should stick with only one for most of your work
  - fdopen() will wrap a FILE around a file descriptor

Note: stick with file descriptors for our projects

i.e., open/close/read/write/lseek