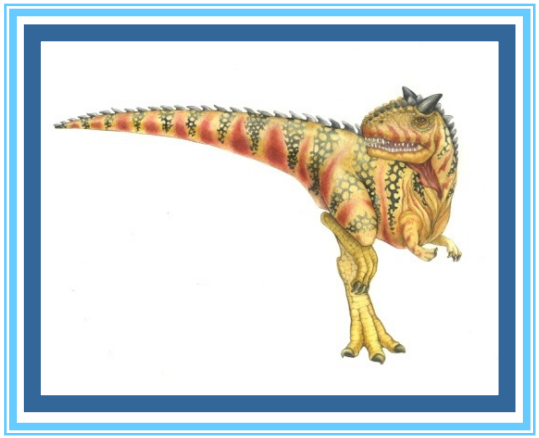
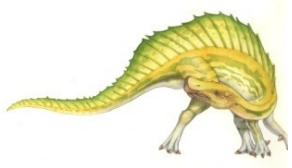
Chapter 12: File-System Interface



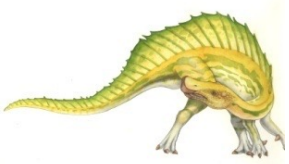
Chapter 12: File-System Interface

- File Concept
- File Operations
- File and Directory Structure
- File-System Partitions and Mounting
- File Sharing
- Protection and Access Control



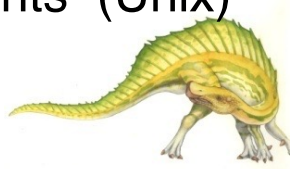
Objectives

- To explain the function of file systems
- To describe the interfaces to file systems
- To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures
- To explore file-system protection



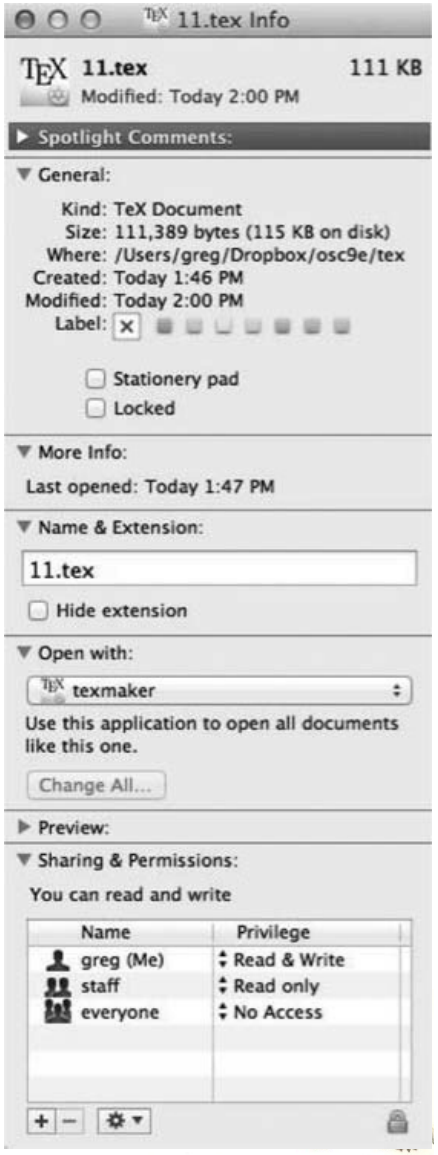
File Concept

- File systems (Contiguous logical address space)
  - Implement an abstraction (files) for secondary storage
  - Organize files logically (directories)
  - Permit sharing of data between processes, people, and machines
  - Protect data from unwanted access (security)
- A file is data with some attributes
  - Contents, size, owner, last read/write time, protection, etc.
- A file can also have a type
  - Understood by the file system
    - Block, character, device, portal, link, etc.
  - Understood by other parts of the OS or runtime libraries
    - Executable, dll, souce, object, text, etc.
  - A file's type can be encoded in its name (Windows) or contents (Unix)
    - .com, .exe, .bat, .dll, .jpg, etc.



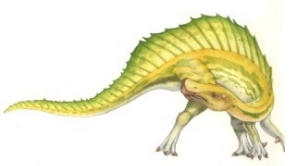
File Attributes

- **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
- **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
- Information kept in the directory structure



File Operations

- File is an **abstract data type**
- **Create**
- **Write –** at **write pointer** location
- **Read –** at **read pointer** 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



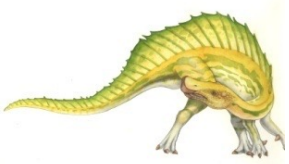
File Operations: Unix vs Windows

Unix

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

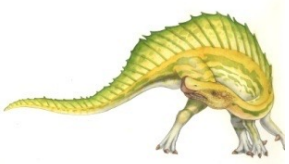
NT

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



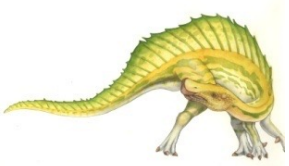
Open Files

- Several pieces of data are needed to manage open files:
  - **Open-file table**: tracks open files
  - File pointer: pointer to last read/write location, per process that has the file open
  - **File-open count**: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information



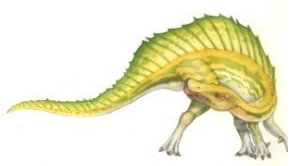
Open File Locking

- Provided by some operating systems and file systems
  - Similar to reader-writer locks
  - **Shared lock** similar to reader lock – several processes can acquire concurrently
  - **Exclusive lock** similar to writer lock
- Mediates access to a file
- Mandatory or advisory:
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do



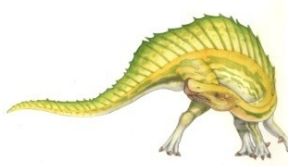
File Locking Example – Java API

```
import java.io.*;
import java.nio.channels.*;
public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;
    public static void main(String arsg[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;
        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
            // get the channel for the file
            FileChannel ch = raf.getChannel();
            // this locks the first half of the file - exclusive
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
            /** Now modify the data . . . */
            // release the lock
            exclusiveLock.release();
```



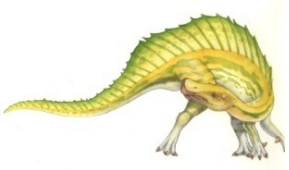
File Locking Example – Java API (Cont.)

```
                    // this locks the second half of the file - shared
                    sharedLock = ch.lock(raf.length()/2+1, raf.length(),
                                         SHARED);
                    /** Now read the data . . . */
                    // release the lock
                    sharedLock.release();
                } catch (java.io.IOException ioe) {
                    System.err.println(ioe);
                }finally {
                    if (exclusiveLock != null)
                    exclusiveLock.release();
                    if (sharedLock != null)
                    sharedLock.release();
                }
            }
        }
```



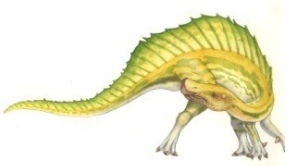
File Types – Name, Extension

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



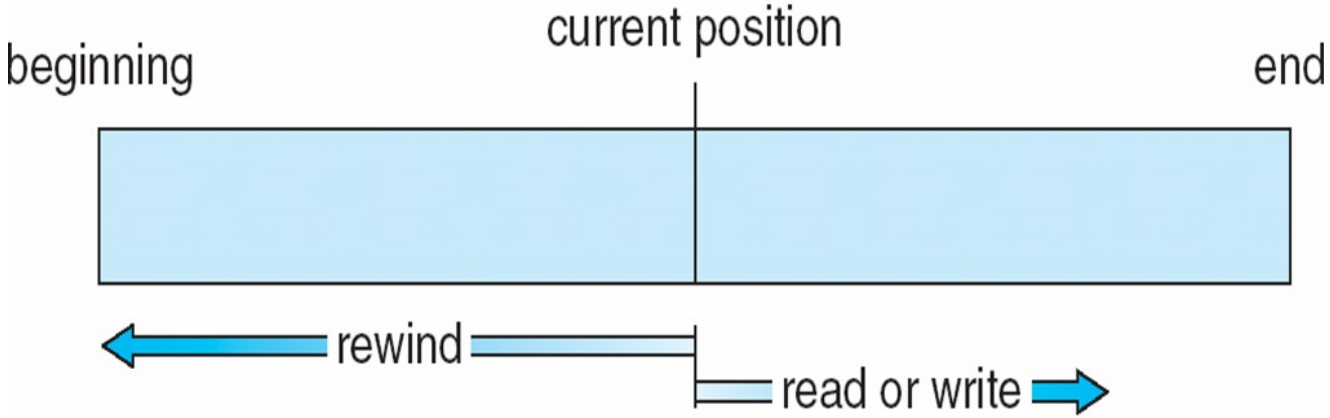
File Structure

- None - sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
  - Operating system
  - Program

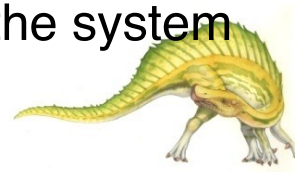


File Access Methods

- Some file systems provide different access methods that specify different ways for accessing data in a file
  - Sequential access – read bytes one at a time, in order

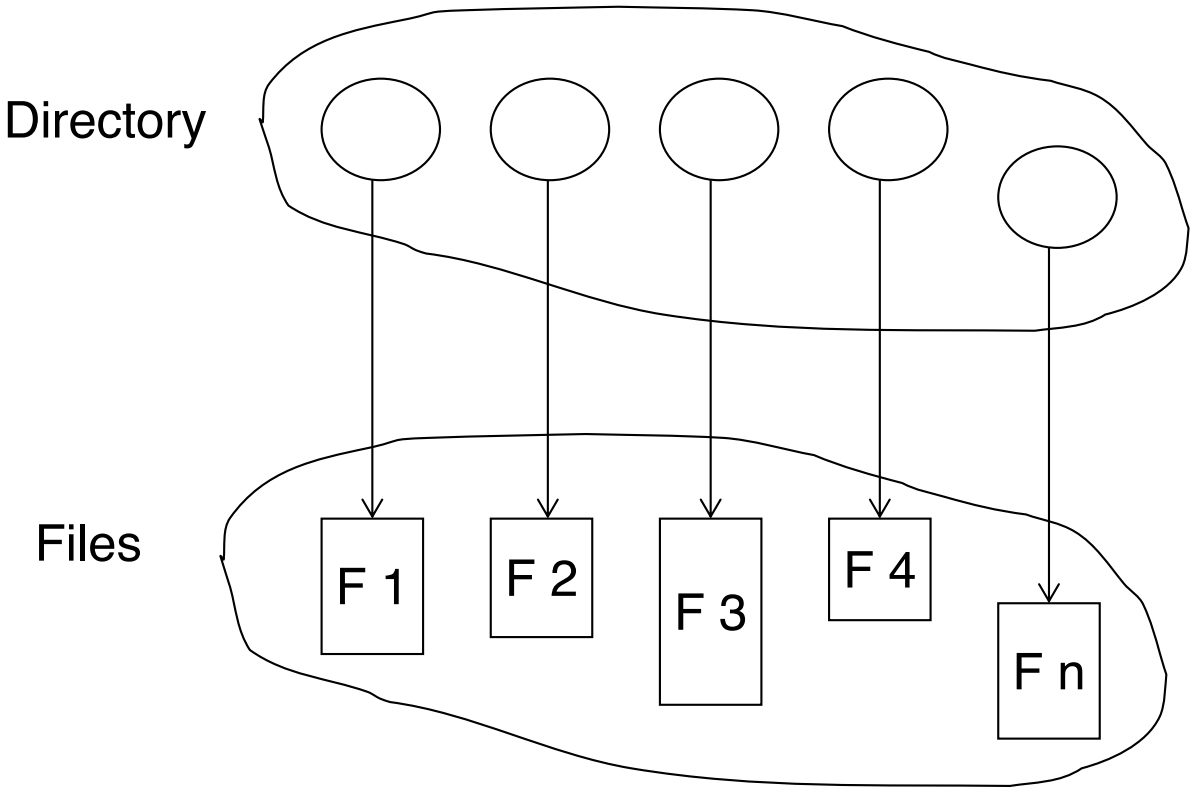


  - Direct access – random access given block/byte number
  - Record access – file is array of fixed- or variable-length records, read/written sequentially or randomly by record #
  - Indexed access – file system contains an index to a particular field of each record in a file, reads specify a value for that field and the system finds the record via the index (DBs)

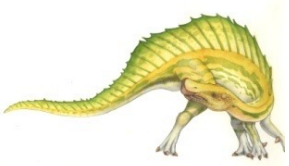


Directory Structure

- A collection of nodes containing information about all files

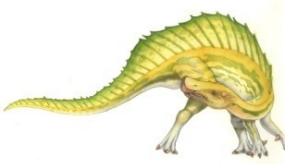


Both the directory structure and the files reside on disk



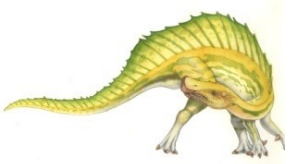
Directories

- Directories serve two purposes
  - For users, they provide a structured way to organize files
  - For the file system, they provide a convenient naming interface that allows the implementation to separate logical file organization from physical file placement on the disk
- Most file systems support multi-level directories
  - Naming hierarchies (/, /usr, /usr/local/, ...)
- Most file systems support the notion of a current directory
  - Relative names specified with respect to current directory
  - Absolute names start from the root of directory tree

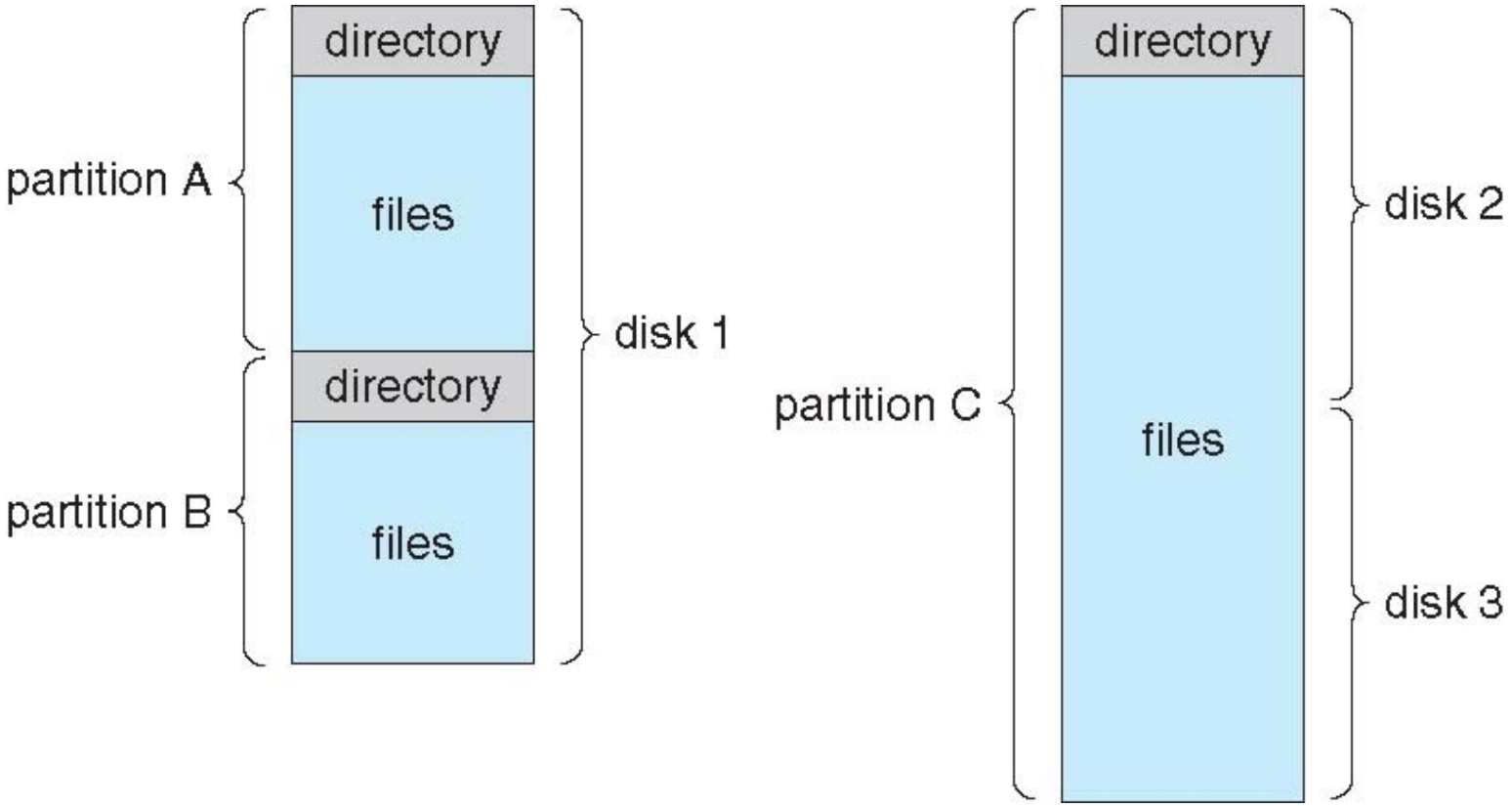


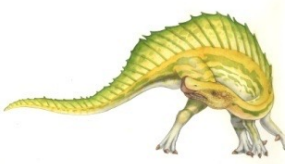
Directory Internals

- A directory is a list of entries
  - <name, location>
  - Name is just the name of the file or directory
  - Location depends upon how file is represented on disk
- List is usually unordered (effectively random)
  - Entries usually sorted by program that reads directory
- Directories typically stored in files
  - Only need to manage one kind of secondary storage unit



A Typical File-system Organization





Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

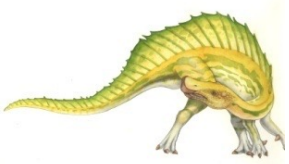
Basic Directory Operations

Unix

- Directories implemented in files
  - Use file ops to create dirs
- C runtime library provides a higher-level abstraction for reading directories
  - opendir(name)
  - readdir(DIR)
  - seekdir(DIR)
  - closedir(DIR)

NT

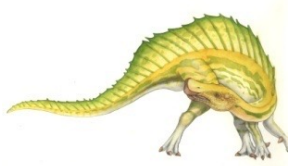
- Explicit dir operations
  - CreateDirectory(name)
  - RemoveDirectory(name)
- Very different method for reading directory entries
  - FindFirstFile(pattern)
  - FindNextFile()



Directory Organization

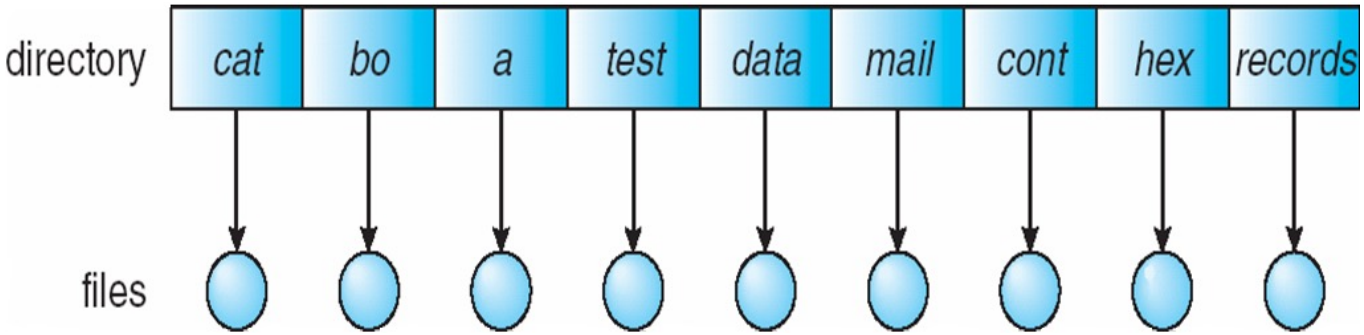
The directory is organized logically to obtain

- 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, …)

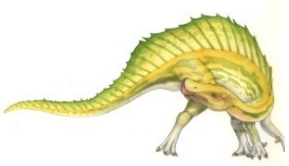


Single-Level Directory

- A single directory for all users

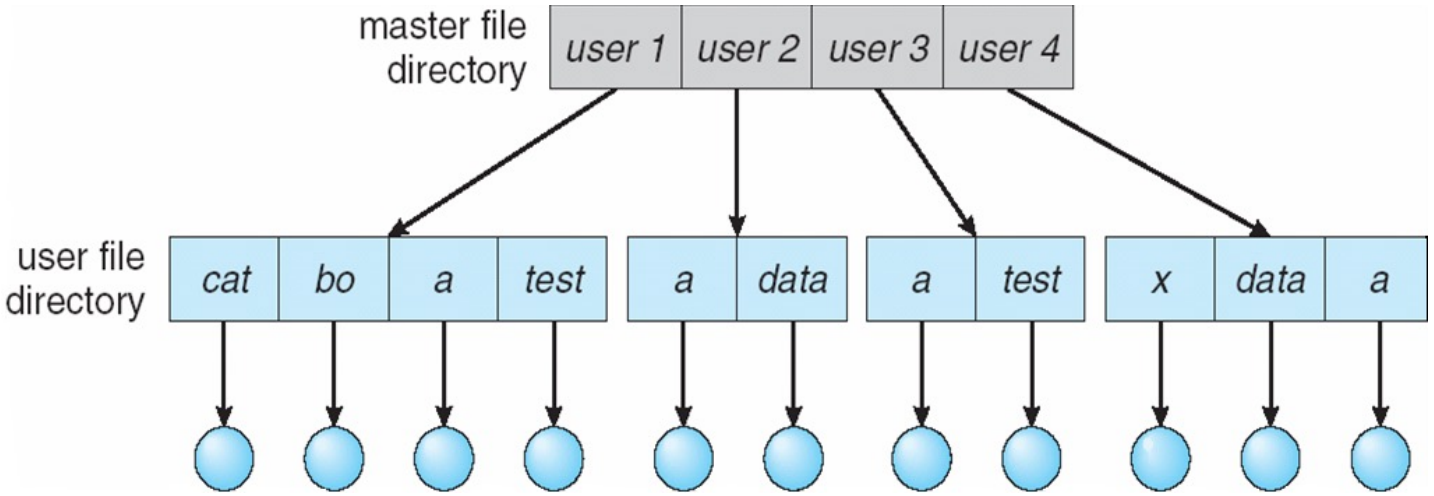


- Naming problem
- Grouping problem

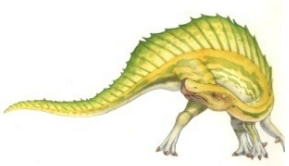


Two-Level Directory

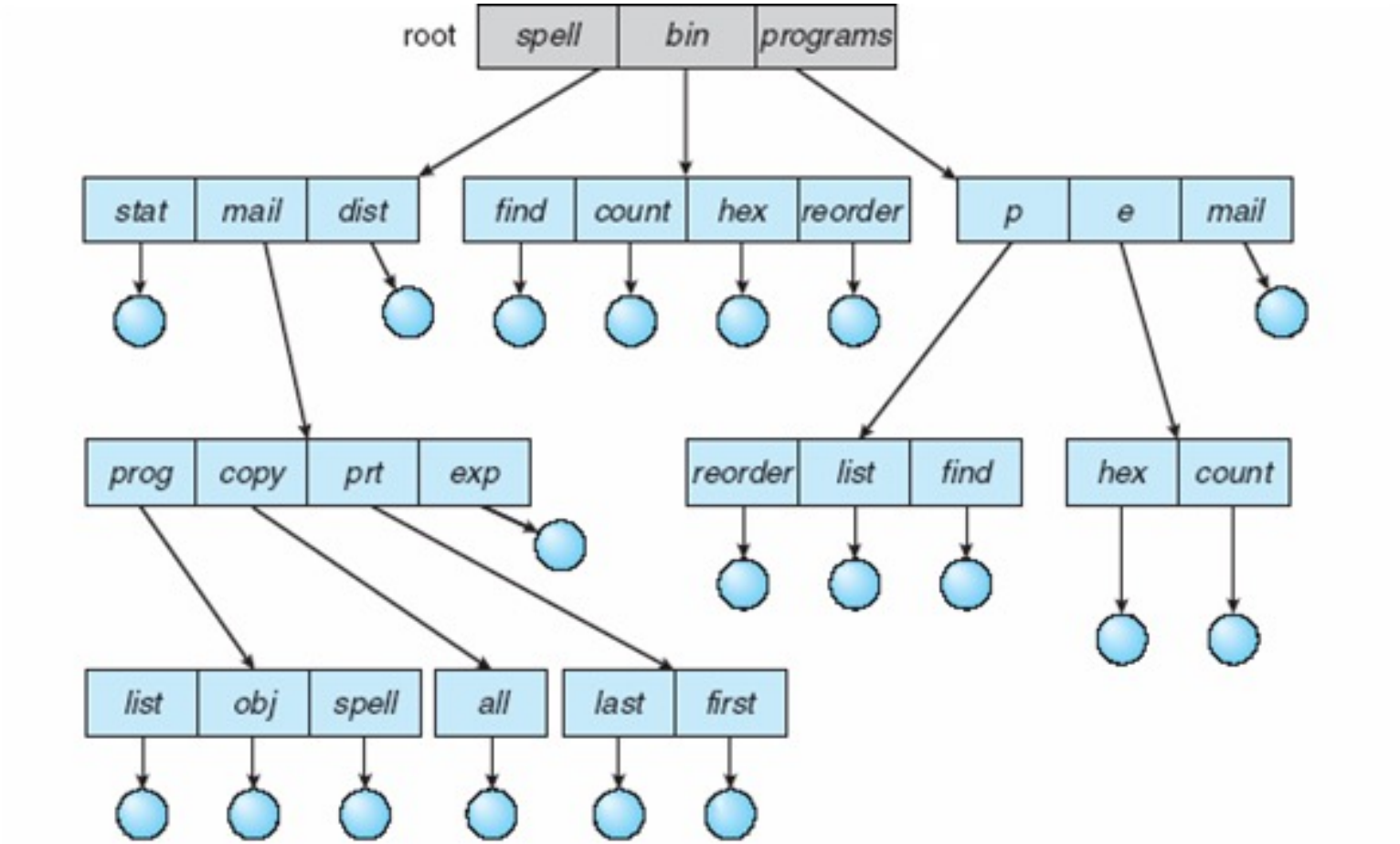
- Separate directory for each user

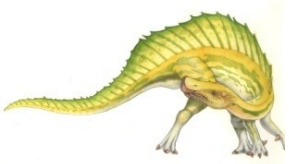


- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability



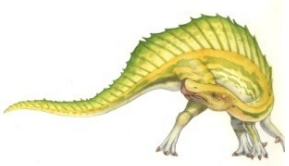
Tree-Structured Directories





Tree-Structured Directories (Cont.)

- Efficient searching (Systems spend a lot of time walking directory paths)
- Grouping Capability
- Let's say you want to open "/one/two/three". What does the file system do?
  - Open root directory "/" (well known, can always find)
  - Search for the entry "one", get location of "one" (in directory entry)
  - Open directory "one", search for "two", get location of "two"
  - Open directory "two", search for "three", get location of "three"
  - Open file "three"
- Current directory (working directory)
  - `cd /spell/mail/prog`
  - `type list`



Tree-Structured Directories (Cont)

- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file

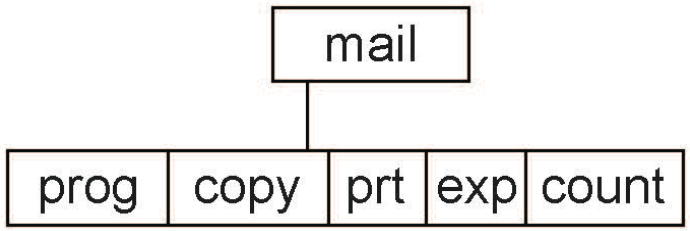
```
rm <file-name>
```

- Creating a new subdirectory is done in current directory

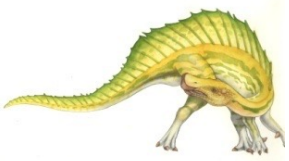
```
mkdir <dir-name>
```

Example: if in current directory `/mail`

```
mkdir count
```

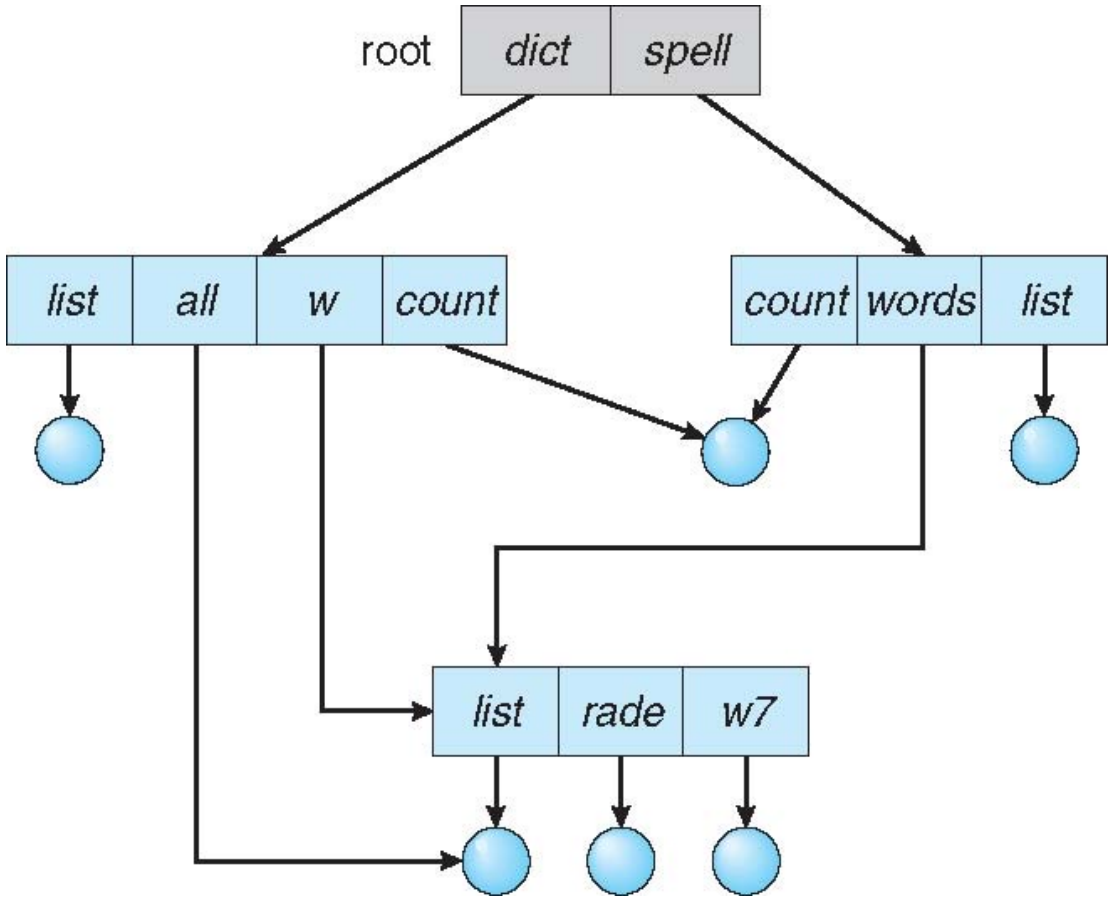


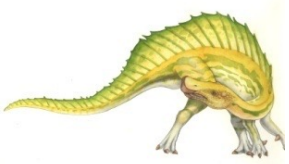
Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"



Acyclic-Graph Directories

- Have shared subdirectories and files

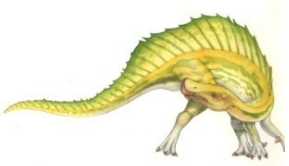




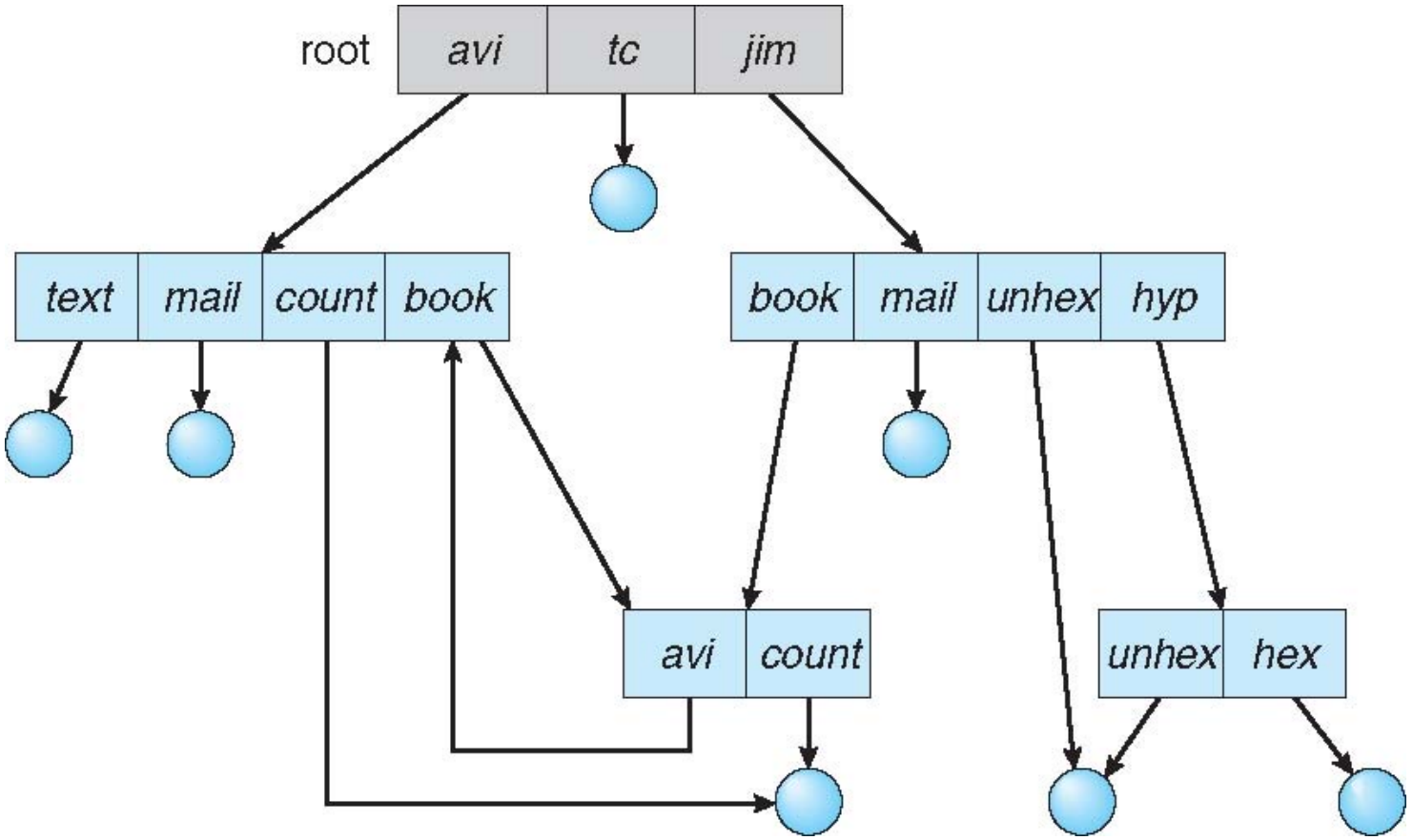
Acyclic-Graph Directories (Cont.)

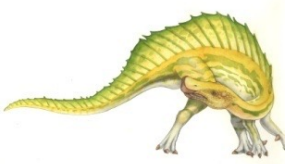
- Two different names (aliasing)
- If ***dict*** deletes ***list*** ⇒ dangling pointer

  Solutions:
  - Backpointers, so we can delete all pointers
    Variable size records a problem
  - Backpointers using a daisy chain organization
  - Entry-hold-count solution
- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file



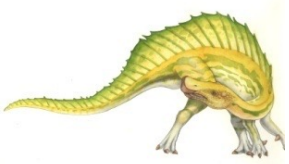
General Graph Directory





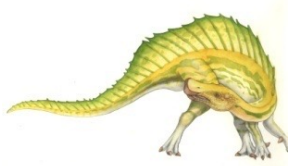
General Graph Directory (Cont.)

- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - **Garbage collection**
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK



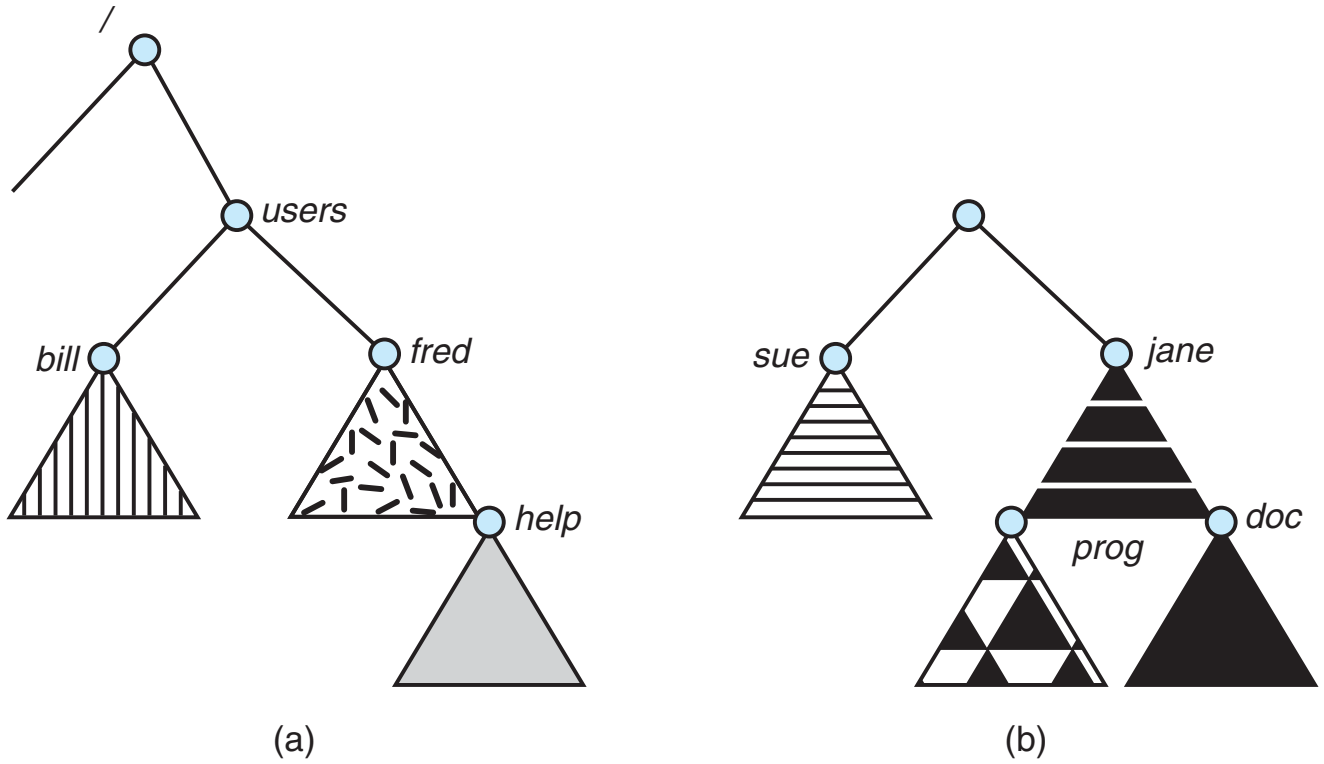
Partitions and Mounting

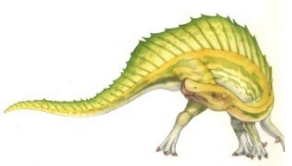
- Partition can be a volume containing a file system ("cooked") or **raw** – just a sequence of blocks with no file system
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting
- **Root partition** contains the OS, other partitions can hold other Oses, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
  - Is all metadata correct?
    - If not, fix it, try again
    - If yes, add to mount table, allow access



File System Mounting

- A file system must be **mounted** before it can be accessed
- A unmounted file system (i.e., Fig. 11-11(b)) is mounted at a **mount point**



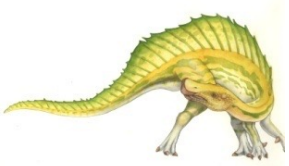


File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme

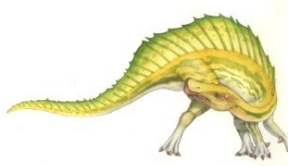
- Two key issues when sharing files
  - Semantics of concurrent access
    - What happens when one process reads while another writes?
    - What happens when two processes open a file for writing?

- If multi-user system
  - **User IDs** identify users, allowing permissions and protections to be per-user
    Group IDs allow users to be in groups, permitting group access rights
  - Owner of a file / directory
  - Group of a file / directory



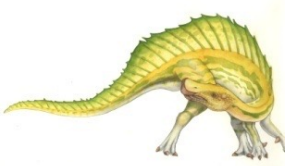
File Sharing – Remote File Systems

- Uses networking to allow file system access between systems
  - Manually via programs like FTP
  - Automatically, seamlessly using **distributed file systems**
  - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
  - Server can serve multiple clients
  - Client and user-on-client identification is insecure or complicated
  - **NFS** is standard UNIX client-server file sharing protocol
  - **CIFS** is standard Windows protocol
  - Standard operating system file calls are translated into remote calls
- Distributed Information Systems (**distributed naming services**) such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing



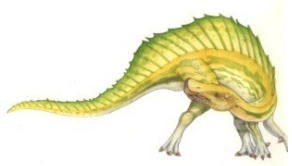
File Sharing – Failure Modes

- All file systems have failure modes
  - For example corruption of directory structures or other non-user data, called **metadata**
- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve **state information** about status of each remote request
- **Stateless** protocols such as NFS v3 include all information in each request, allowing easy recovery but less security



Protection

- File systems implement some kind of protection system
  - Who can access a file
  - How they can access it
- More generally... Objects are “what”, subjects are “who”, actions are “how”
- A protection system dictates whether a given action performed by a given subject on a given object should be allowed
  - You can read and/or write your files, but others cannot
  - You can read “/etc/motd”, but you cannot write it
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

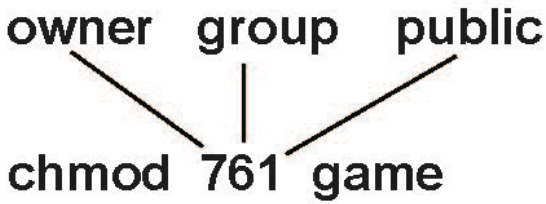


Access Lists and Groups

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

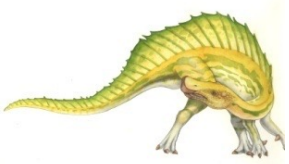
			RWX
a) **owner access**	7	⇒	1 1 1
b) **group access**	6	⇒	RWX 1 1 0
c) **public access**	1	⇒	RWX 0 0 1

- 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.

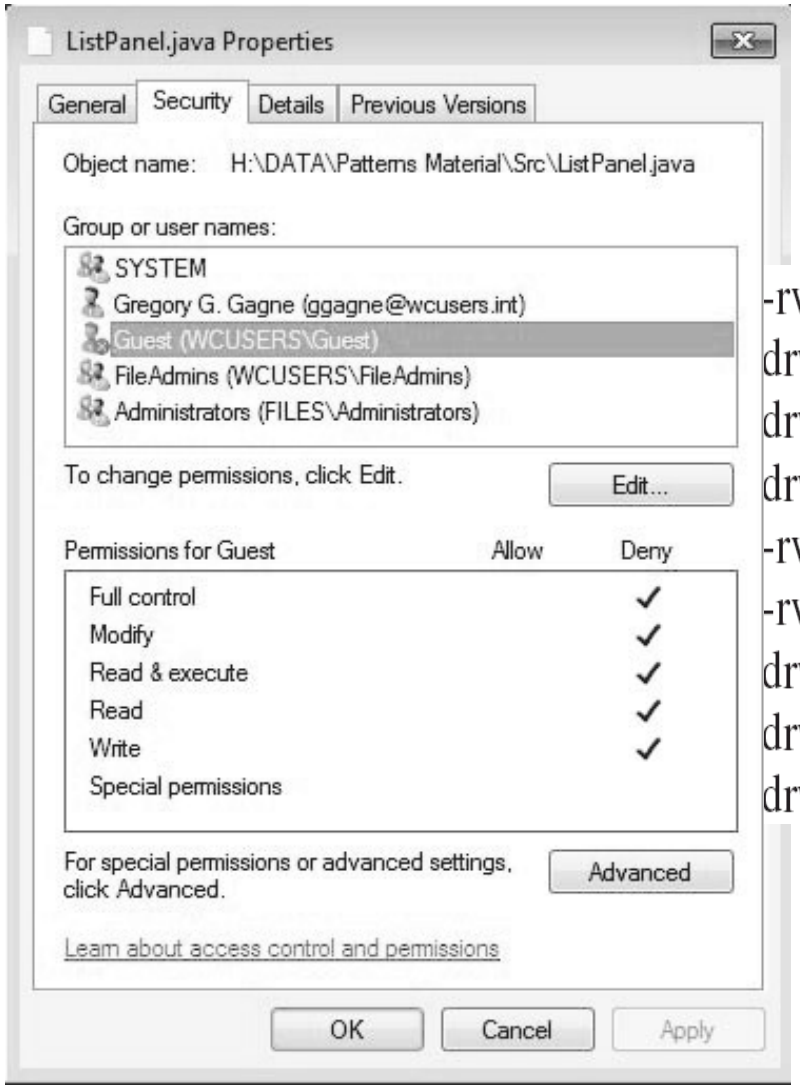


Attach a group to a file

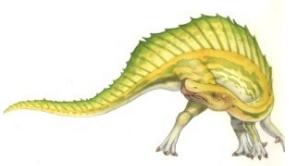
```
chgrp    G    game
```



Access-Control List Management

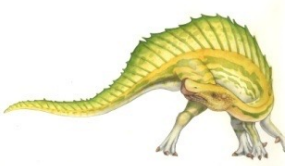



```
-rw-rw-r--   1 pbg   staff    31200  Sep 3 08:30   intro.ps
drwx------   5 pbg   staff      512  Jul 8 09.33   private/
drwxrwxr-x   2 pbg   staff      512  Jul 8 09:35   doc/
drwxrwx---   2 pbg   student    512  Aug 3 14:13   student-proj
-rw-r--r--   1 pbg   staff     9423  Feb 24 2003   program.c
-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/
```



Access Control Lists and Capabilities

- Access Control Lists (ACL)
  - For each object, maintain a list of subjects and their permitted actions
- Capabilities
  - For each subject, maintain a list of objects and their permitted actions
- Capabilities are easier to transfer
  - They are like keys, can handoff, does not depend on subject
- In practice, ACLs are easier to manage
  - Object-centric, easy to grant, revoke
  - To revoke capabilities, have to keep track of all subjects that have the capability – a challenging problem
- ACLs have a problem when objects are heavily shared
  - The ACLs become very large
  - Use groups (e.g., Unix)



End of Chapter 12