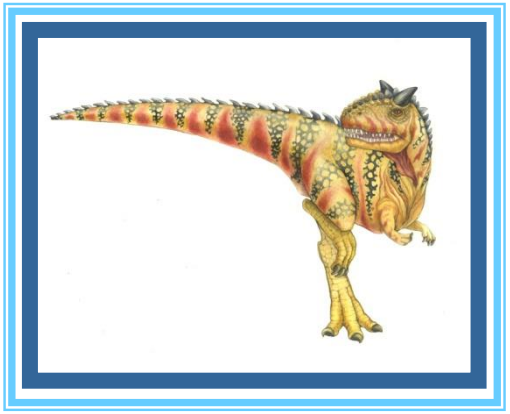
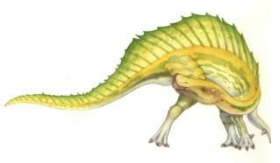
Chapter 11: File-System Interface



Long-term Information Storage

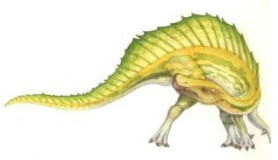
- Must store large amounts of data
- Information stored must survive the termination of the process using it
- Multiple processes must be able to access the information concurrently

- A file can be defined as a data structure which stores the sequence of records.
- Files are stored in a file system, which may exist on a disk or in the main memory.
- Files can be simple (plain text) or complex (specially-formatted).
- The collection of files is known as Directory.
- The collection of directories at the different levels, is known as File System.



File Concept

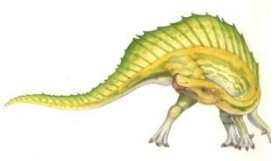
- The file system consists of two distinct parts:
  - a collection of files - storing related data
  - a directory structure - organizes and provides information about all the files in the system
- A file is a named collection of related information that is recorded on secondary storage.
- From a user's perspective, a file is the smallest allotment of logical secondary storage
  - Data cannot be written to secondary storage unless they are within a file.



File Concept

- Types:
  - Data
    - numeric
    - character
    - binary
  - Program
    - Source
    - Object
- Contents defined by file's creator
  - Many types
    - **text file, source file, executable file**

- A file has a certain defined structure, which depends on its type.
  - A **text file** is a sequence of characters organized into lines (and possibly pages).
  - A **source file** is a sequence of functions, each of which is further organized as declarations followed by executable statements.
  - An **executable file** is a series of code sections that the loader can bring into memory and execute.



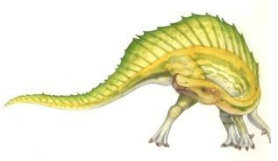
File Attributes

- **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

File info Window on Mac OS X

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, Append, Rename, Lock**
- **Get/Set Attributes**
- ***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



Open Files

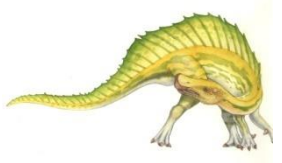
- Several pieces of data are needed to manage open files:
  - **Open-file table**: tracks open files
  - **File pointer**: pointer to last read/write location, per process that has the file open
  - **File-open count**: counter of number of times a file is open – to allow removal of data from open-file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

- Typically, the operating system uses two levels of internal tables:
  - a per-process table
  - a system-wide table.
- The per process table tracks all files that a process has open.
  - Stored in this table is information regarding the process's use of the file.
  - For instance, the current file pointer for each file is found here.
  - Access rights to the file and accounting information can also be included.

- Each entry in the per-process table in turn points to a system-wide open-file table.
  - The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size.
  - Once a file has been opened by one process, the system-wide table includes an entry for the file.
  - When another process executes an open() call, a new entry is simply added to the process's open-file table pointing to the appropriate entry in the system-wide table.
  - Typically, the open-file table also has an **open count** associated with each file to indicate how many processes have the file open.
  - Each close() decreases this open count, and when the open count reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.

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 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

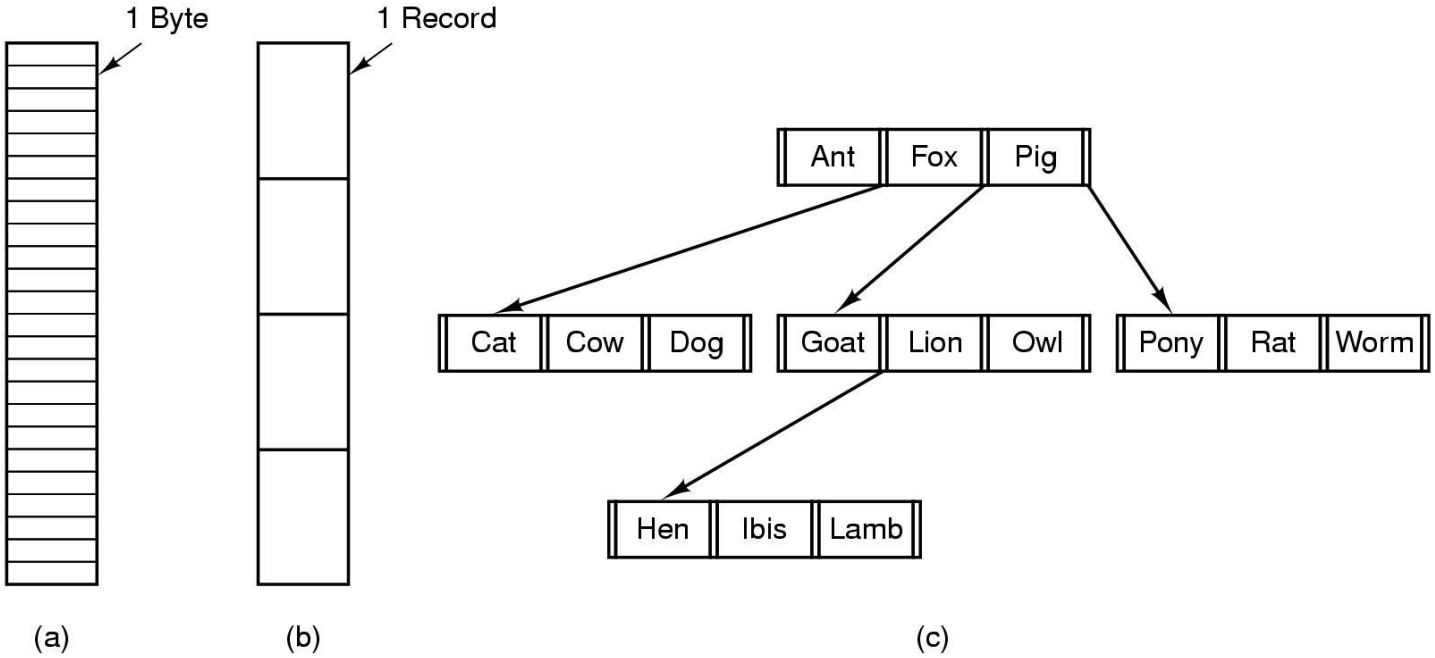
Internal File Structure

- Disk systems typically have a well-defined block size determined by the size of a sector.
- All disk I/O is performed in units of one block (physical record), and all blocks are the same size.
- It is unlikely that the physical record size will exactly match the length of the desired logical record.
- Logical records may even vary in length.
- Packing a number of logical records into physical blocks is a common solution to this problem.
- UNIX operating system defines all files to be simply streams of bytes.
  - Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical record size is 1 byte.

Internal File Structure

- The file system automatically packs and unpacks bytes into physical disk blocks—say, 512 bytes per block—as necessary.
- The logical record size, physical block size, and packing technique determine how many logical records are in each physical block.
- The packing can be done either by the user's application program or by the operating system.
- In either case, the file may be considered a sequence of blocks. All the basic I/O functions operate in terms of blocks.
- Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted.
  - All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

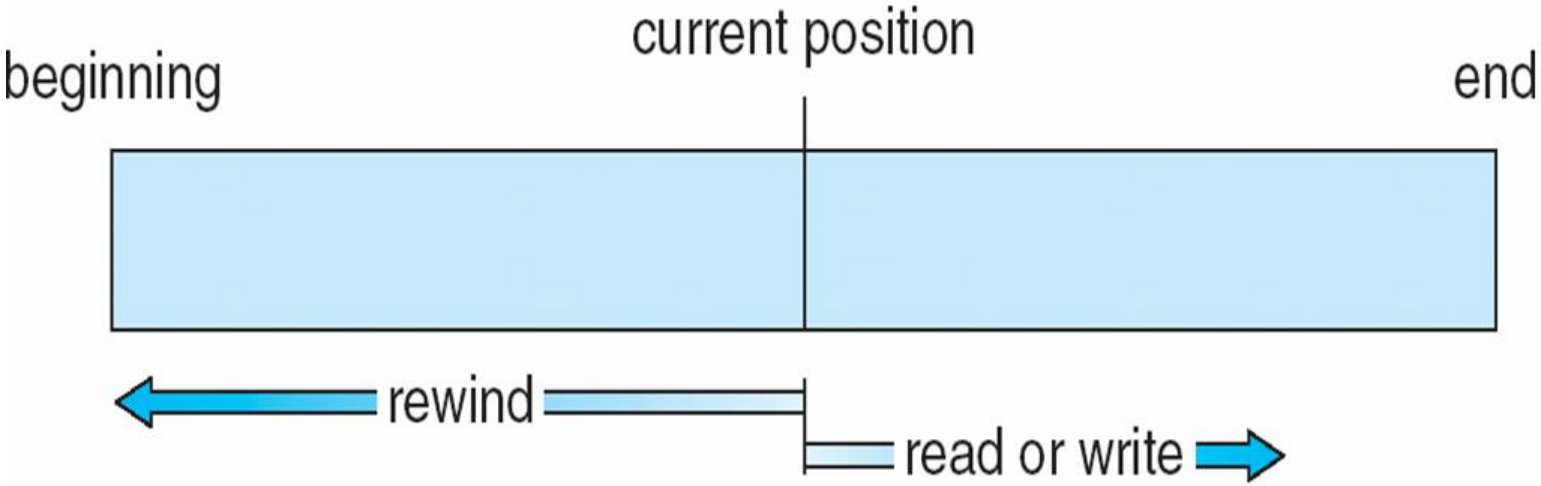
File Structure



- Three kinds of files
  - byte sequence
  - record sequence
  - tree

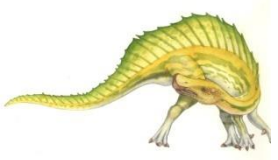
File Access

- **Sequential access**
  - read all bytes/records from the beginning
  - cannot jump around, could rewind or back up
  - convenient when medium was mag tape



Sequential-access File

- Read operation—read_next()—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
- Write operation—write_next()—appends to the end of the file and advances to the end of the newly written material (the new end of file).
- Reset to the beginning, and on some systems, a program may be able to skip forward or backward *n* records for some integer *n*—perhaps only for *n* = 1.
- no read after last write (rewrite)



Sequential-access File

- **Random access (direct access or relative access).**
  - bytes/records read in any order
  - essential for database systems
  - read can be …
    - move file marker (seek), then read or …
    - read and then move file marker

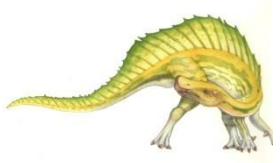
Direct Access

- File is fixed length logical records
- File operations must be modified to include the block number as a parameter.

```
read n
write n
position to n
     read_next
     write_next
rewrite n
```

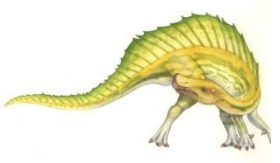
  n = relative block number

- Relative block numbers allow OS to decide where file should be placed



Simulation of Sequential Access on Direct-access File

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

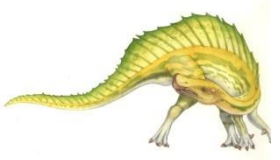


Other Access Methods

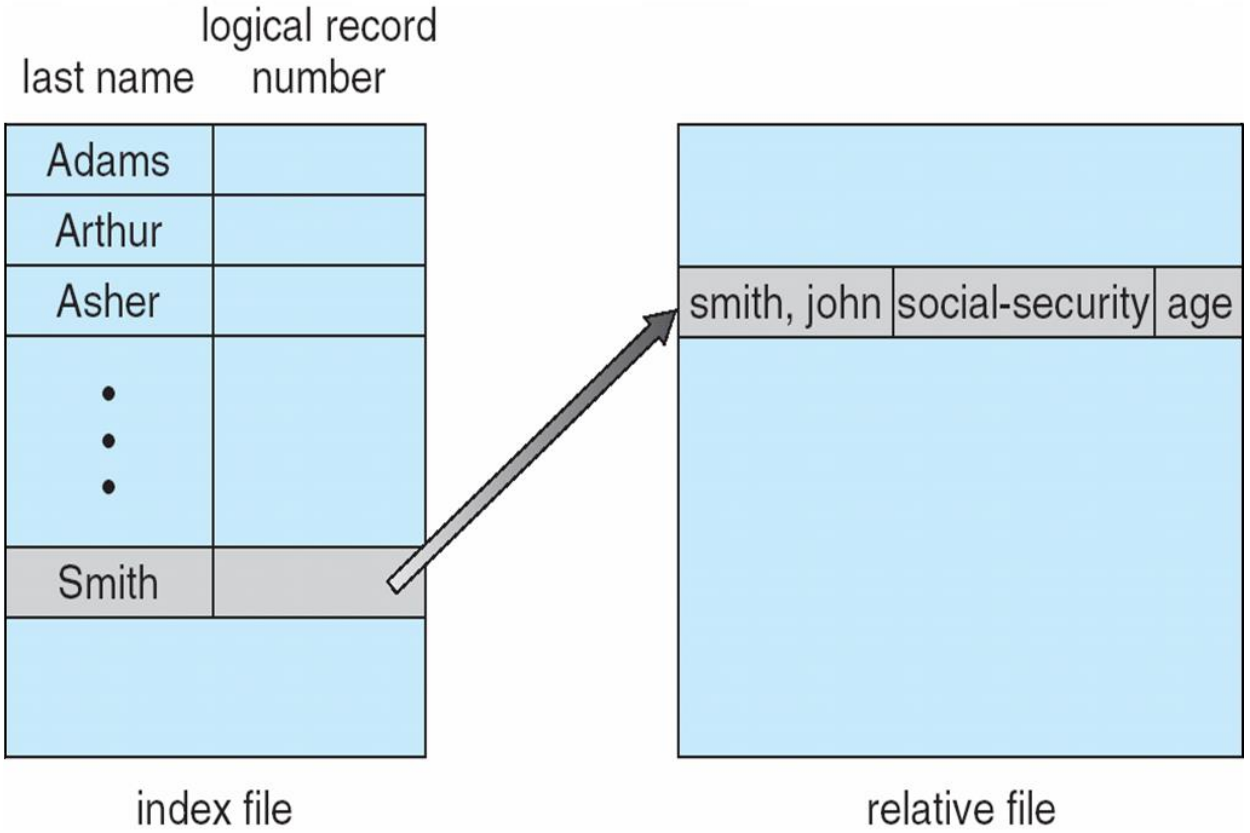
- Can be built on top of base methods
- General involve creation of an index for the file
- Keep index in memory for fast determination of location of data to be operated on
- If too large, index (in memory) of the index (on disk)

Other Access Methods

- IBM indexed sequential-access method (ISAM)
  - Small master index, points to disk blocks of secondary index
  - File kept sorted on a defined key
  - To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index.
  - This block is read in, and again a binary search is used to find the block containing the desired record.
  - Finally, this block is searched sequentially.
  - In this way, any record can be located from its key by at most two direct access reads.



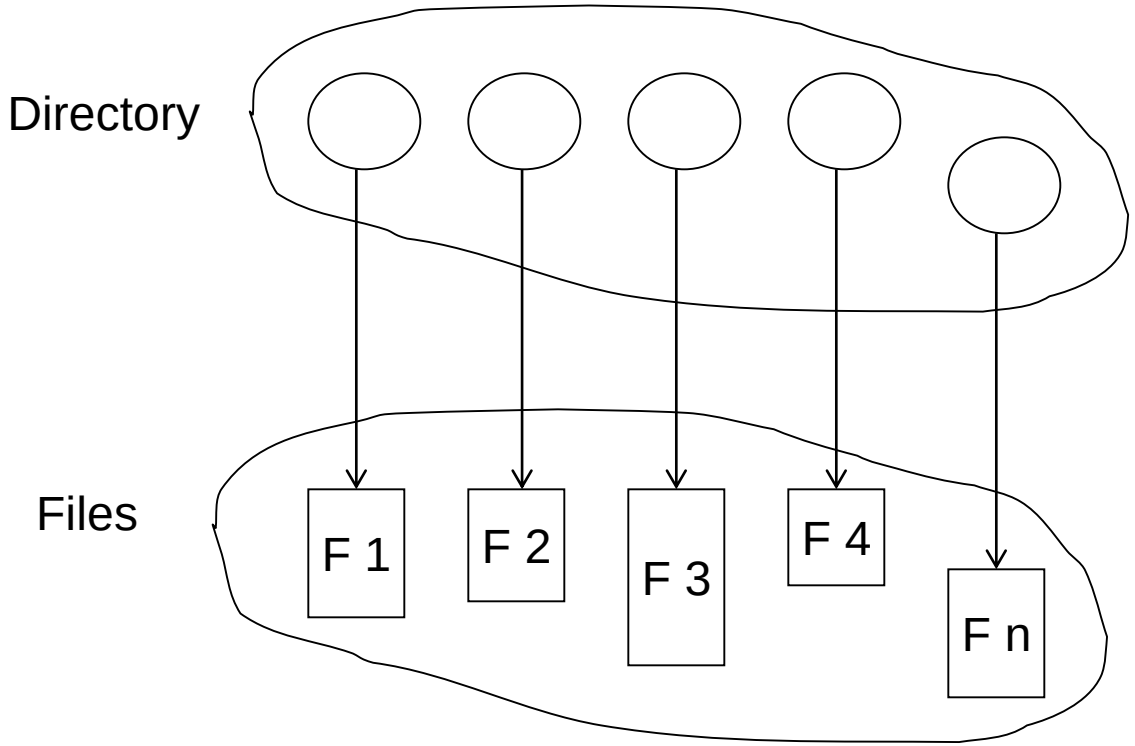
Example of Index and Relative Files



VMS operating system provides index and relative files

Directory Structure

- A collection of nodes containing information about all files

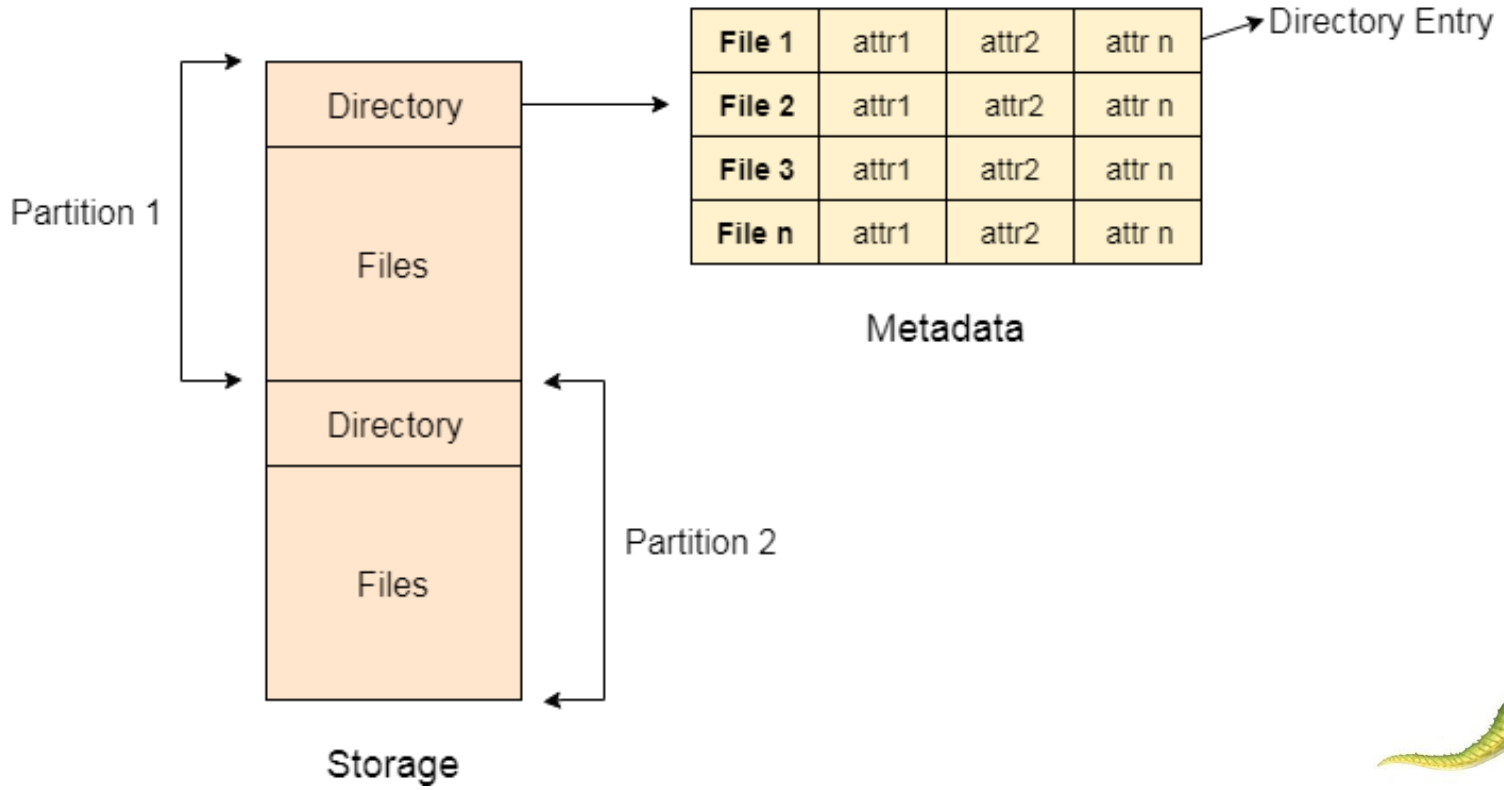


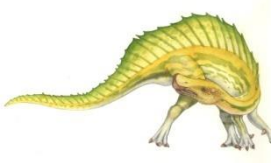
Both the directory structure and the files reside on disk

Disk Structure

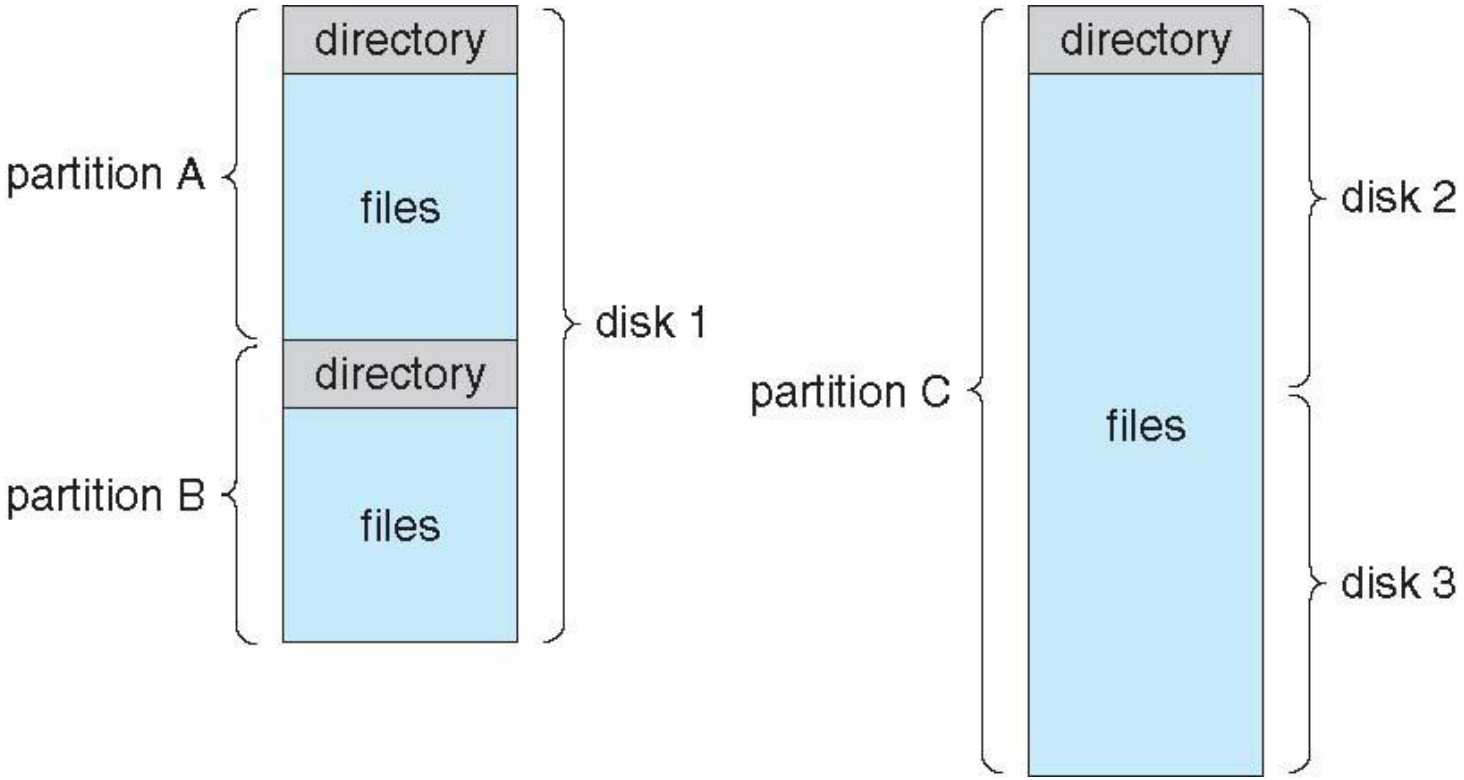
- Disk can be subdivided into **partitions**
- Disk or partition can be used **raw** – without a file system, or **formatted** with a file system
- Partitions also known as minidisks, slices
- Entity containing file system known as a **volume**
- Each volume containing file system also tracks that file system's info in **device directory** or **volume table of contents**
  - The device directory records information—such as name, location, size, and type—for all files on that volume.

- Each partition must have at least one directory in which, all the files of the partition can be listed.
- A directory entry is maintained for each file in the directory which stores all the information related to that file.

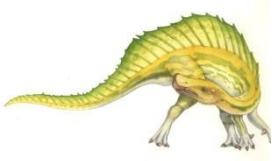




A Typical File-system Organization

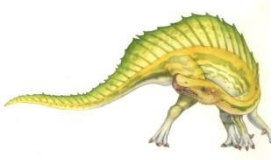


A directory can be viewed as a file which contains the Meta data of the bunch of files.



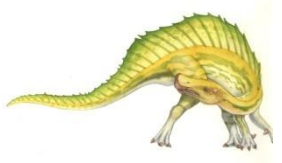
Types of File Systems

- Systems frequently have may file systems, some general- and some special- purpose
- Consider Solaris has
  - tmpfs – memory-based volatile FS for fast, temporary I/O
  - objfs – interface into kernel memory to get kernel symbols for debugging
  - ctfs – contract file system for managing daemons
  - lofs – loopback file system allows one FS to be accessed in place of another
  - procfs – kernel interface to process structures
  - ufs, zfs – general purpose file systems



Operations Performed on Directory

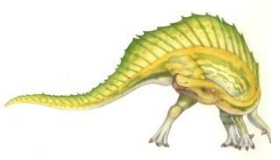
- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system



Directory Organization

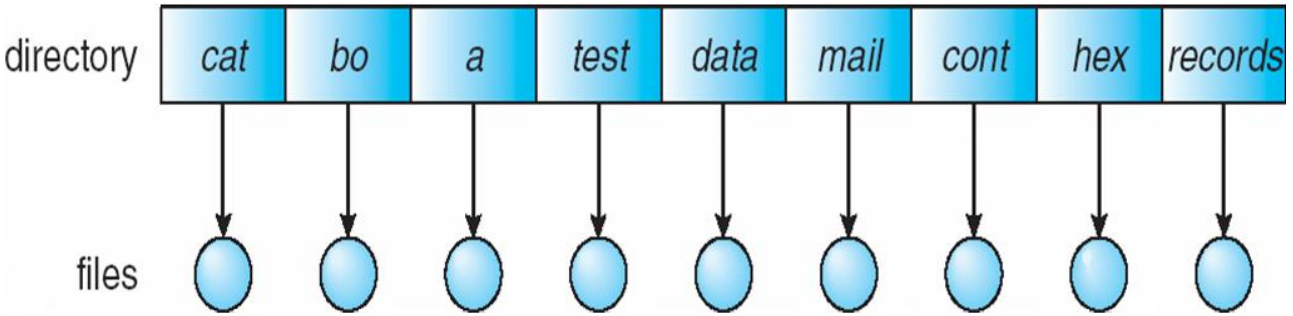
The directory is organized logically to obtain

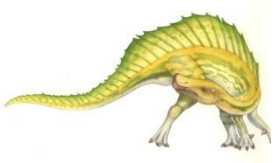
- Efficiency – locating a file quickly
- Naming – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)



Single-Level Directory

- The entire system will contain only one directory to mention all the files present in the file system
- The directory contains one entry per each file present on the file system.
- A single directory for all users

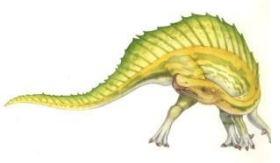




Single-Level Directory

Advantages

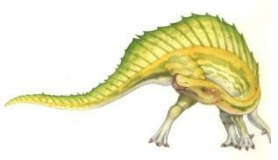
- Implementation is very simple.
- If the num of the files are very small then the searching becomes faster.
- File creation, searching, deletion is very simple since we have only one directory.



Single-Level Directory

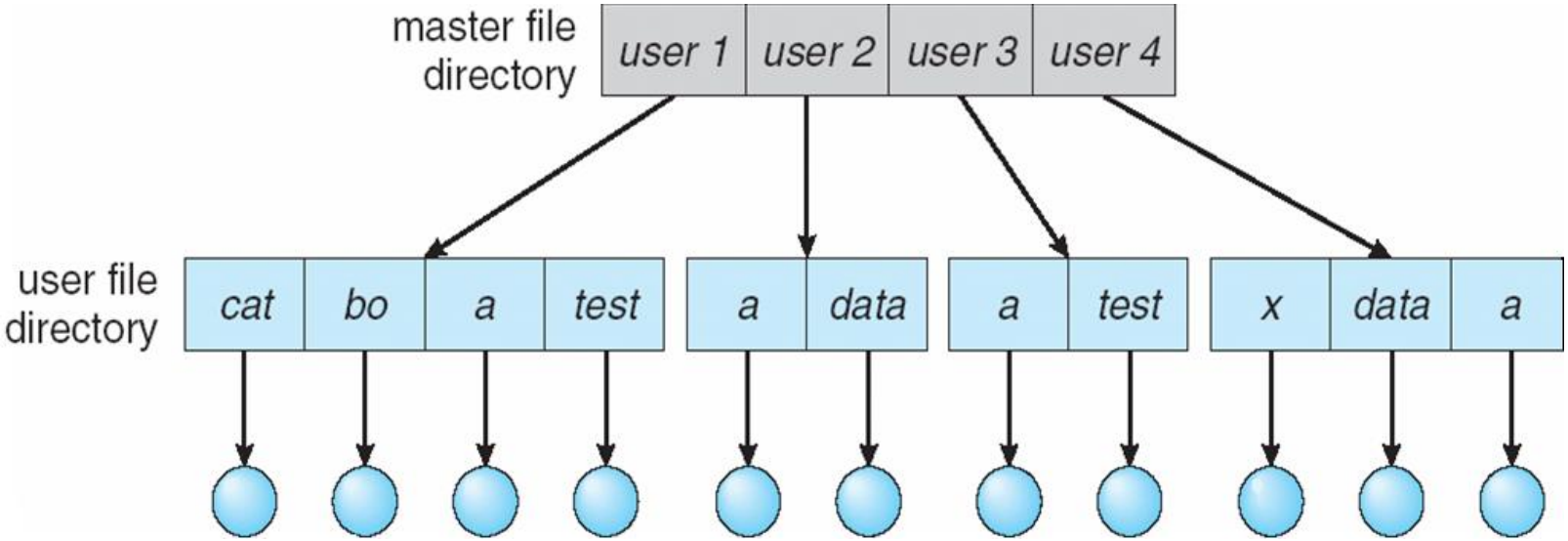
Disadvantages

- Naming problem : We cannot have two files with the same name.
- The directory may be very big therefore searching for a file may take so much time.
- Protection cannot be implemented for multiple users.
- Grouping problem : There are no ways to group same kind of files.
- Choosing the unique name for every file is a bit complex and limits the number of files in the system because most of the Operating System limits the number of characters used to construct the file name.

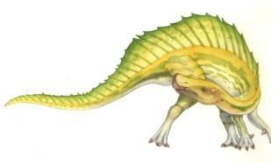


Two-Level Directory

- Separate directory for each user

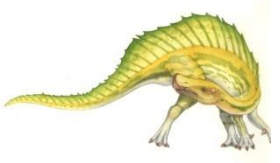


The system doesn't let a user to enter in the other user's directory without permission.

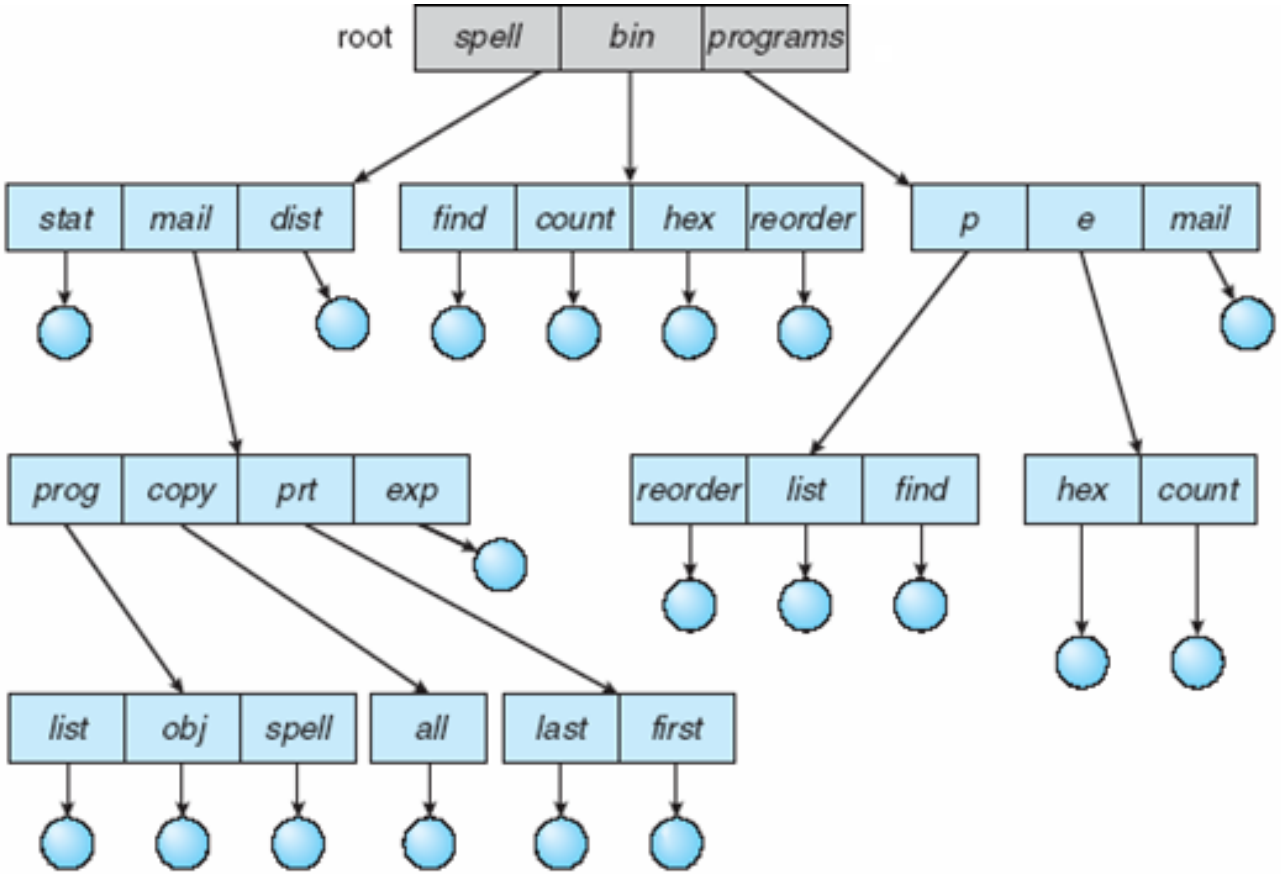


Characteristics of two level directory system

- Each files has a path name as ***/User-name/directory-name/***
- Different users can have the same file name.
- Searching becomes more efficient as only one user's list needs to be traversed.
- The same kind of files cannot be grouped into a single directory for a particular user.

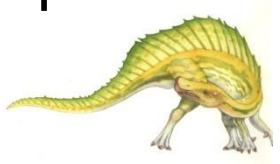


Tree-Structured Directories



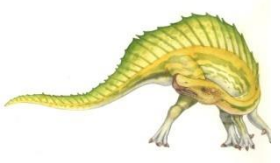
Tree structured directory

- In Tree structured directory system, any directory entry can either be a file or sub directory.
- Tree structured directory system overcomes the drawbacks of two level directory system.
  - The similar kind of files can now be grouped in one directory.
- Each user has its own directory and it cannot enter in the other user's directory.
- However, the user has the permission to read the root's data but he cannot write or modify this.
- Only administrator of the system has the complete access of root directory.



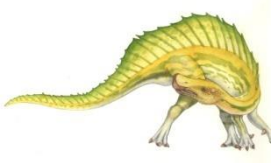
Tree-Structured Directories (Cont.)

- Searching is more efficient in this directory structure.
- The concept of current working directory is used.
- A file can be accessed by two types of path, either **relative or absolute.**
- Absolute path is the path of the file with respect to the root directory of the system while relative path is the path with respect to the current working directory of the system.
- In tree structured directory systems, the user is given the privilege to create the files as well as directories.



Permissions on the file and directory

- A tree structured directory system may consist of various levels therefore there is a set of permissions assigned to each file and directory.
- The permissions are **R W X** which are regarding reading, writing and the execution of the files or directory.
- The permissions are assigned to three types of users: owner, group and others.
- There is a identification bit which differentiate between directory and file. For a directory, it is **d** and for a file, it is dot **(.)**



Tree-Structured Directories (Cont)

- Creating a new file is done in current directory
- Delete a file

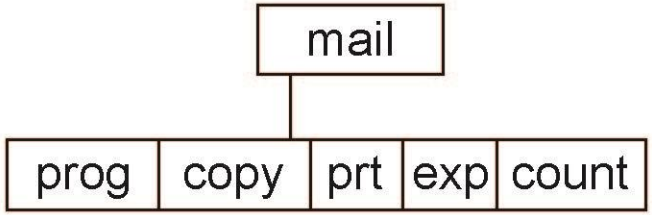
  `rm <file-name>`

- Creating a new subdirectory is done in current directory

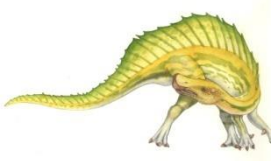
  `mkdir <dir-name>`

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

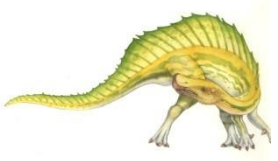
  `mkdir count`



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

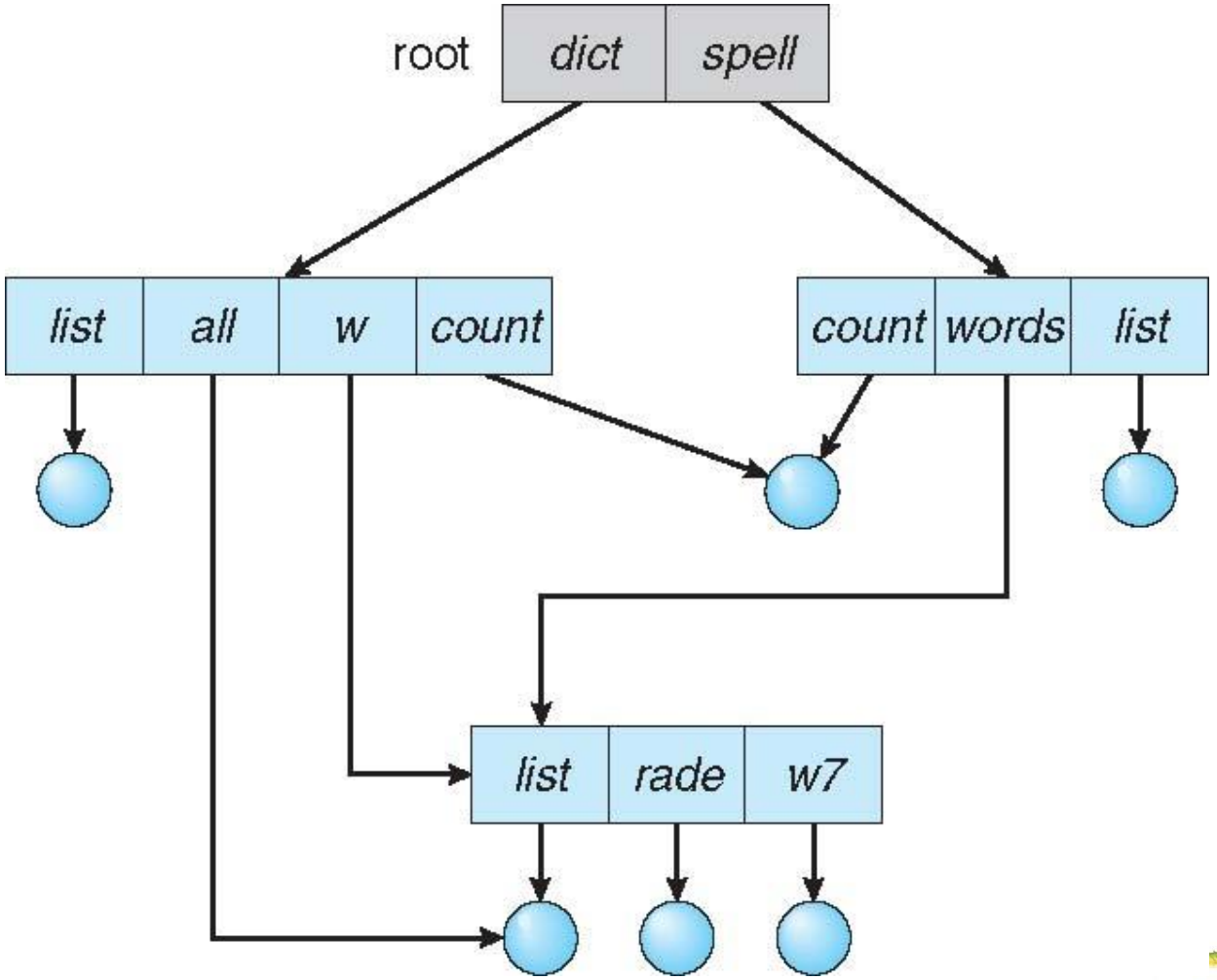


- The tree structured directory system doesn't allow the same file to exist in multiple directories therefore sharing is major concern in tree structured directory system.
- We can provide sharing by making the directory an acyclic graph.
- In this system, two or more directory entry can point to the same file or sub directory.
- That file or sub directory is shared between the two directory entries

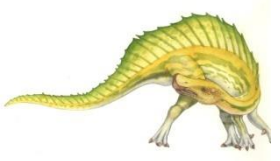


Acyclic-Graph Directories

- Have shared subdirectories and files



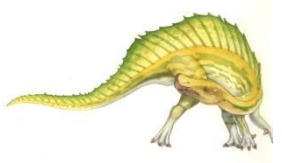
- These kinds of directory graphs can be made using links or aliases.
- We can have multiple paths for a same file.
- Links can either be symbolic (logical) or hard link (physical).
- If a file gets deleted in acyclic graph structured directory system, then
  - In the case of soft link, the file just gets deleted and we are left with a dangling pointer.
  - In the case of hard link, the actual file will be deleted only if all the references to it gets deleted.



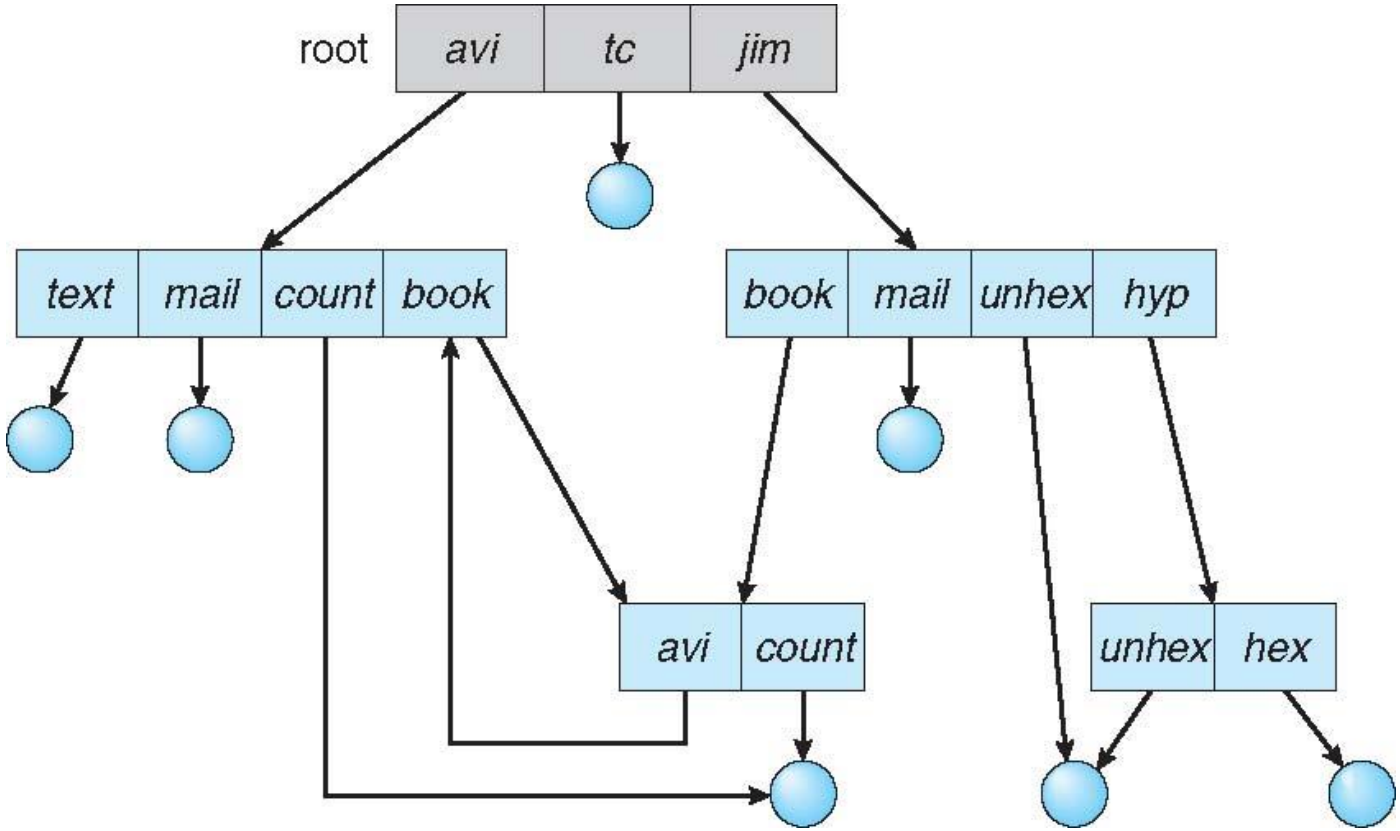
Acyclic-Graph Directories (Cont.)

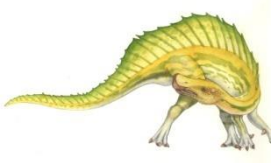
- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file
- Two different names (aliasing)
- If ***dict*** deletes ***list*** ⇒ dangling pointer

  Solutions:
  - Delete all the links (cost huge time)
  - Leave the link until an attempt is made to use them.
  - Preserve the file until all references to it is deleted.
    - Entry-hold-count solution



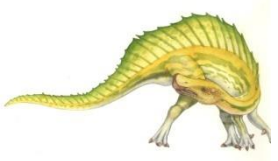
General Graph Directory





General graph directory

- Cycles are allowed within a directory structure where multiple directories can be derived from more than one parent directory.
- Issue is to calculate the total size or space that has been taken by the files and directories.
- **Advantages:**
  - It allows cycles.
  - It is more flexible than other directories structure.
- **Disadvantages:**
  - It is more costly than others.
  - It needs garbage collection.

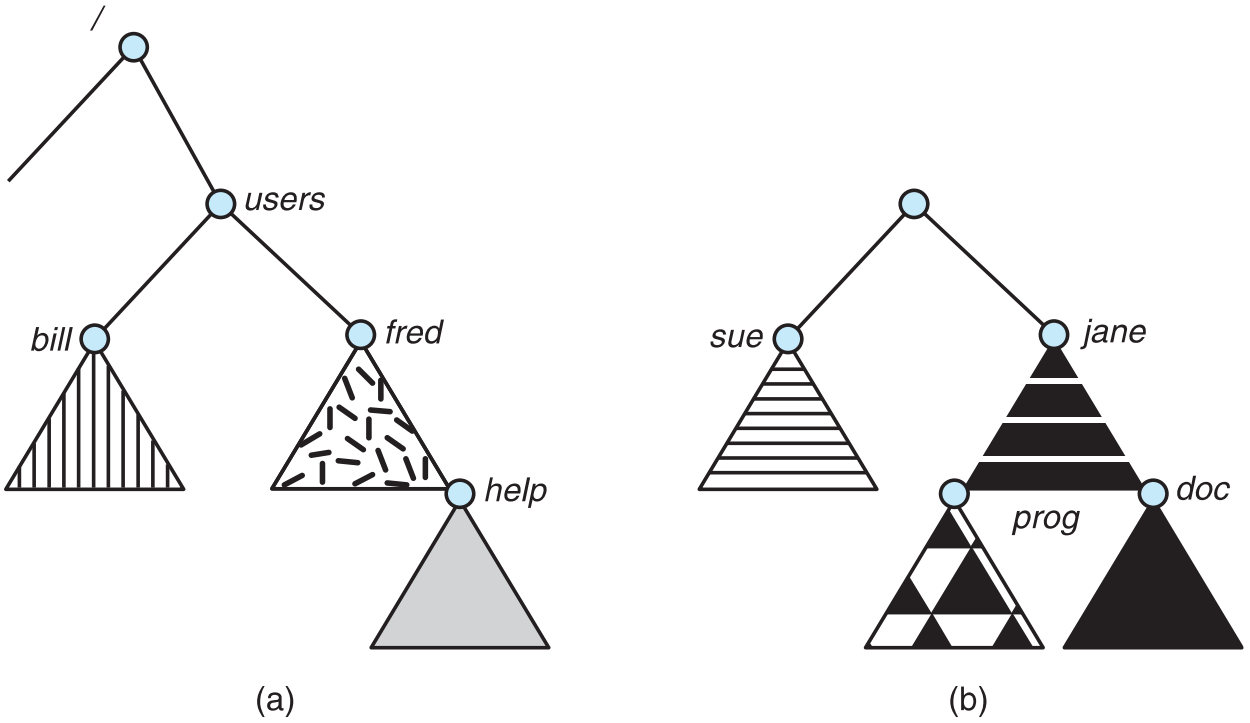


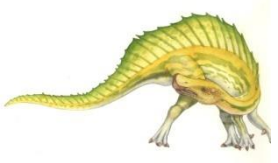
General Graph Directory (Cont.)

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

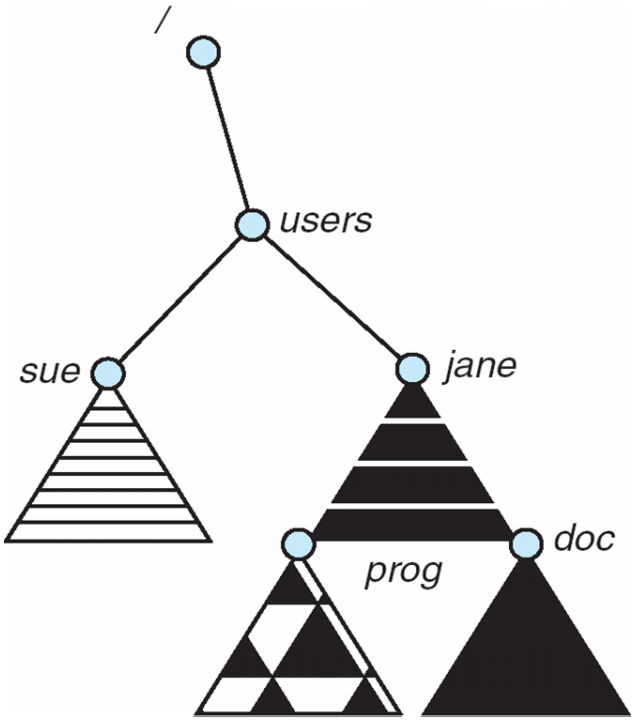
File System Mounting

- A file system must be **mounted** before it can be accessed
- A unmounted file system is mounted at a **mount point**



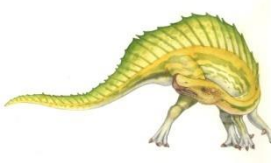


Mount Point



File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method
- 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

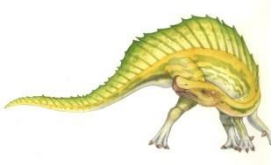


File Sharing – Remote File Systems

- 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

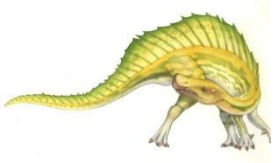
File Sharing – Failure Modes

- 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



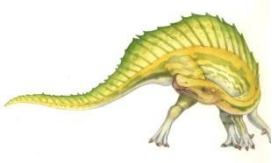
File Sharing – Consistency Semantics

- Specify how multiple users are to access a shared file simultaneously
  - Similar to Ch 5 process synchronization algorithms
    - Tend to be less complex due to disk I/O and network latency (for remote file systems
  - Andrew File System (AFS) implemented complex remote file sharing semantics
  - Unix file system (UFS) implements:
    - Writes to an open file visible immediately to other users of the same open file
    - Sharing file pointer to allow multiple users to read and write concurrently
  - AFS has session semantics
    - Writes only visible to sessions starting after the file is closed



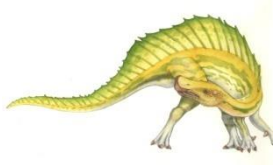
Protection

- When information is stored in a computer system, we want to keep it safe from physical damage (the issue of reliability) and improper access (the issue of protection).
- **Reliability** is generally provided by duplicate copies of files.
- Many computers have systems programs that automatically copy disk files to tape at regular intervals to maintain a copy should a file system be accidentally destroyed.
- File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes…..
- Files may be deleted accidentally.
- Bugs in the file-system software

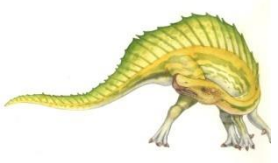


Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Protection mechanisms provide controlled access by limiting the types of file access that can be made
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

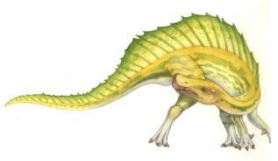


- Other operations, such as renaming, copying, and editing the file, may also be controlled.
- These higher-level functions may be implemented by a system program that makes lower-level system calls.
- Protection is provided at only the lower level.
- For instance, copying a file may be implemented simply by a sequence of read requests.
- In this case, a user with read access can also cause the file to be copied, printed, and so on.

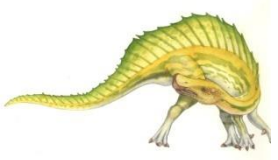


Access Control

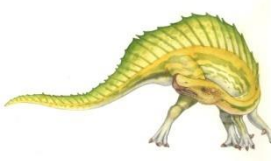
- The most general scheme to implement identity dependent access is to associate with each file and directory an **access-control list (ACL)** specifying user names and the types of access allowed for each user.
- When a user requests access to a particular file, the operating system checks the access list associated with that file.
- If that user is listed for the requested access, the access is allowed.
- Otherwise, a protection violation occurs, and the user job is denied access to the file.



- The main problem with access lists is their length.
  - If we want to allow everyone to read a file, we must list all users with read access.
- This technique has two undesirable consequences:
  - Constructing such a list - tedious and unrewarding task - if we do not know in advance the list of users in the system.
  - The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.



- These problems can be resolved by use of a condensed version of the access list.
- Three classifications of users in connection with each file:
  - **Owner**. The user who created the file is the owner.
  - **Group**. A set of users who are sharing the file and need similar access is a group, or work group.
  - **Universe**. All other users in the system constitute the universe.

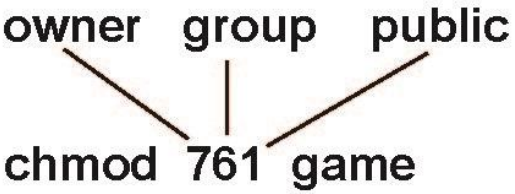


Access Lists and Groups

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

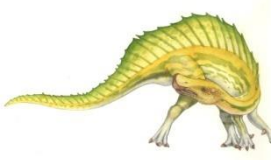
			RWX
a) **owner access**	7	⇒	1 1 1
b) **group access**	6	⇒	RWX 1 1 0
c) **public access**	1	⇒	RWX 0 0 1

- 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.

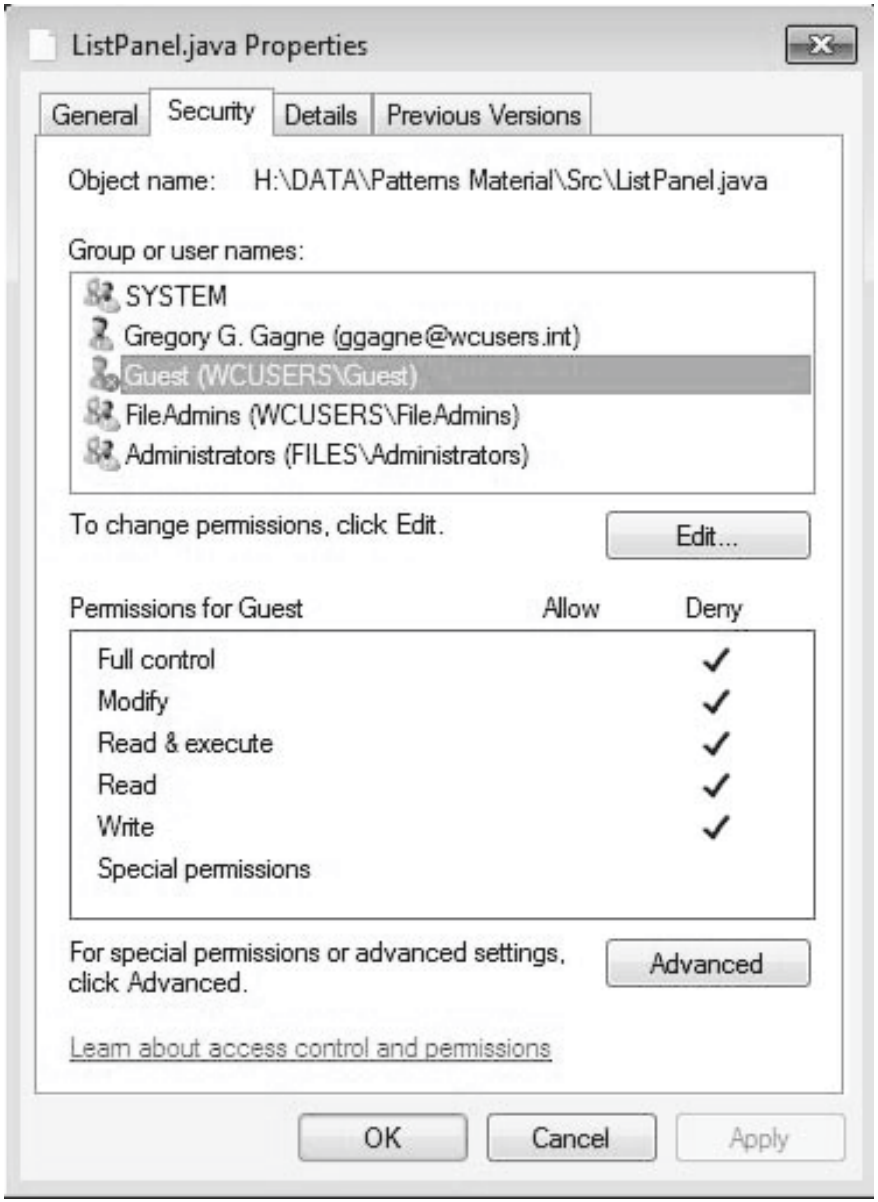


Attach a group to a file

```
chgrp    G    game
```

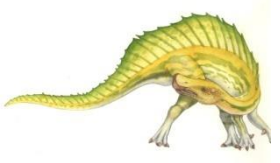


Windows 7 Access-Control List Management



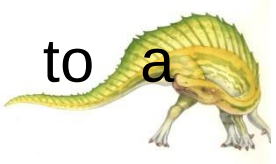
A Sample UNIX Directory Listing

-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/



Other Protection Approaches

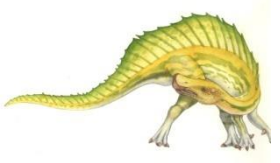
- Associate a password with each file.
  - Number of passwords that a user needs to remember may become large, making the scheme impractical.
  - If only one password is used for all the files, then once it is discovered, all files are accessible
  - Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to address this problem.
- Mechanism for directory protection.
  - Control the creation and deletion of files in a directory.
  - Listing the contents of a directory must be a protected operation.
- A given user may have different access rights to a particular file, depending on the path name used



Protection- GOALS

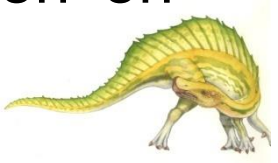
- In protection model, computer consists of a collection of objects
  - **hardware objects** (such as the CPU, memory segments, printers, disks, and tape drives)
  - **software objects** (such as files, programs, and semaphores).
- Each object has a unique name and can be accessed through a well-defined set of operations
- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so

- The operations that are possible may depend on the object.
  - CPU - can only execute.
  - Memory segments can be read and written,
  - CD-ROM or DVD-ROM can only be read.
  - Tape drives can be read, written, and rewound.
  - Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted.



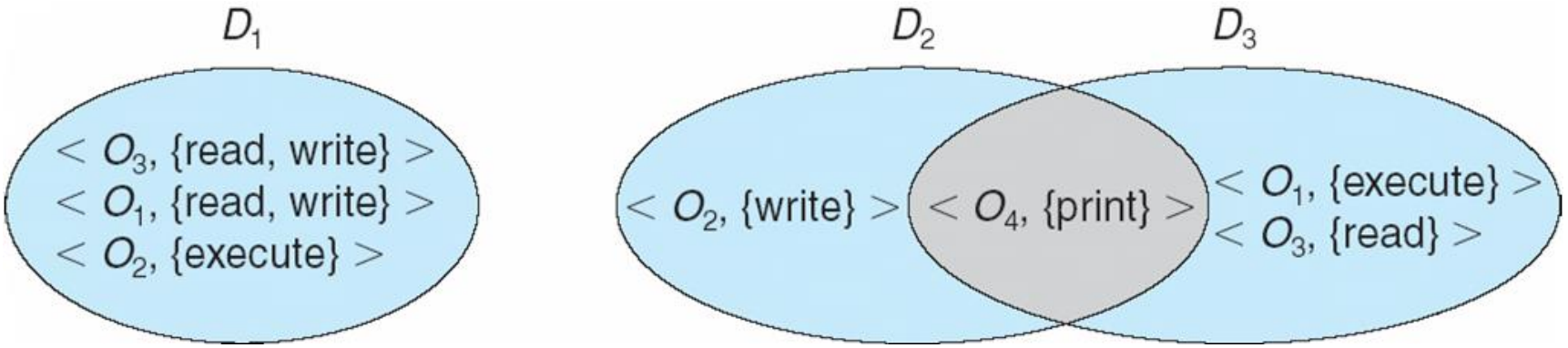
Domain structure

- A process operates within a **protection domain**, which specifies the resources that the process may access.
- Each domain defines a set of objects and the types of operations that may be invoked on each object.
- The ability to execute an operation on an object is an **access right**.
- A domain is a collection of access rights, each of which is an ordered pair <object-name, rights-set>.
  - For example, if domain *D* has the access right <file *F*, *{*read,write*}*>, then a process executing in domain *D* can both read and write file *F.*
  - It cannot, however, perform any other operation on that object.



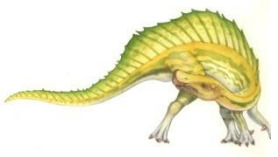
Domain Structure

- Access-right = <*object-name*, *rights-set*> where *rights-set* is a subset of all valid operations that can be performed on the object
- Domain = set of access-rights

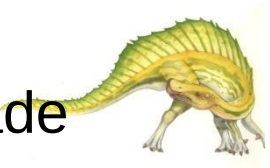


Principles of Protection

- Guiding principle – **principle of least privilege**
  - Programs, users and systems should be given just enough **privileges** to perform their tasks
  - Limits damage if entity has a bug, gets abused
  - Can be static (during life of system, during life of process)
  - Or dynamic (changed by process as needed) – **domain switching**, **privilege escalation**
  - "Need to know" a similar concept regarding access to data



- Each ***user*** may be a domain.
  - The set of objects that can be accessed depends on the identity of the user.
  - Domain switching occurs when the user is changed—generally when one user logs out and another user logs in.
- Each ***process*** may be a domain.
  - The set of objects that can be accessed depends on the identity of the process.
  - Domain switching occurs when one process sends a msg to another process and then waits for a response.
- Each ***procedure*** may be a domain.
  - The set of objects that can be accessed corresponds to the local variables defined within the procedure.
  - Domain switching occurs when a procedure call is made



Domain Implementation (UNIX)

- Domain = user-id
- Domain switch accomplished via file system
  - Each file has associated with it a domain bit (setuid bit)
  - When file is executed and setuid = on, then user-id is set to owner of the file being executed
  - When execution completes user-id is reset
  - When user A (userID = A) starts executing a file owned by B, whose associated domain bit is off, the userID of the process is set to A.
  - When the setuid bit is on, the userID is set to that of the owner of the file: B.
  - When the process exits, this temporary userID change ends

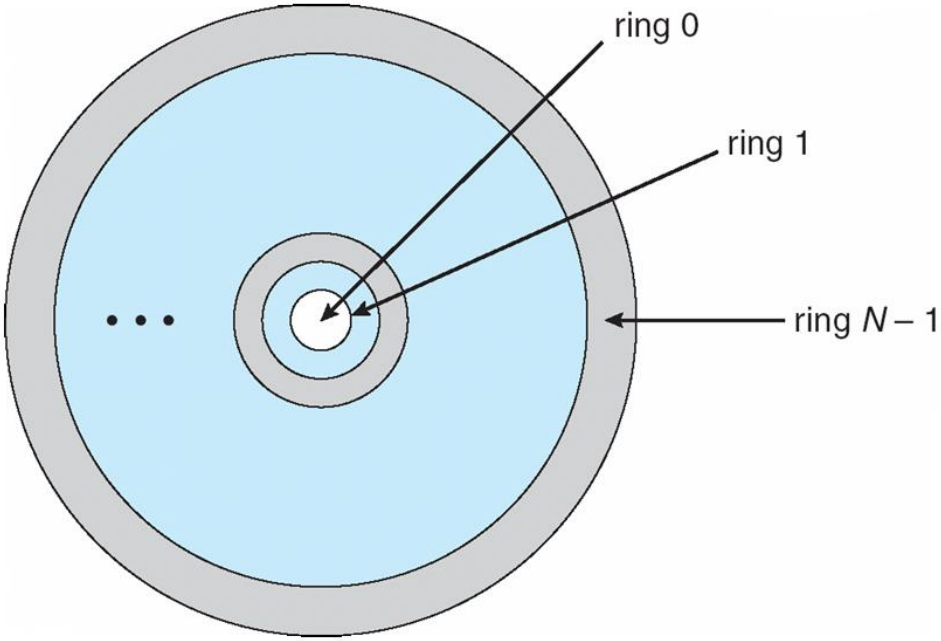
Domain Implementation (UNIX)

- Domain switch accomplished via passwords
  - `su` command temporarily switches to another user's domain when other domain's password provided
- Domain switching via commands
  - `sudo` command prefix executes specified command in another domain (if original domain has privilege or password given)

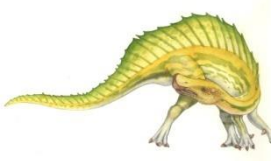
Domain Implementation (MULTICS)

- In the MULTICS system, the protection domains are organized hierarchically into a ring structure.
- Each ring corresponds to a single domain
- The rings are numbered from 0 to 7.
- Let D_i and D_j be any two domain rings
- If $j < i \Rightarrow D_i \subseteq D_j$

A process executing in domain $D0$ has the most privileges.



- MULTICS has a segmented address space; each segment is a file, and each segment is associated with one of the rings.
- A segment description includes an entry that identifies the ring number.
- In addition, it includes three access bits to control reading, writing, and execution.
- When a process is executing in ring i, it cannot access a segment associated with ring j ($j < i$).
- It can access a segment associated with ring k ($k \geq i$).



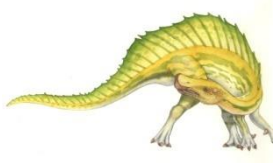
Multics Benefits and Limits

- Ring / hierarchical structure provided more than the basic kernel / user or root / normal user design
- Fairly complex -> more overhead
- But does not allow strict need-to-know
  - Object accessible in D_j but not in D_i, then ***j*** must be < ***i***
  - But then every segment accessible in D_i also accessible in D_j

Access Matrix

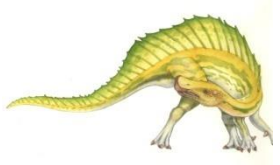
- View protection as a matrix (**access matrix**)
- Rows represent domains
- Columns represent objects
- **`Access(i, j)`** is the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	



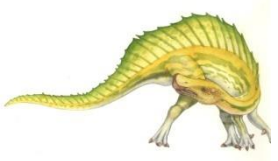
Use of Access Matrix

- If a process in Domain D_i tries to do "op" on object O_j, then "op" must be in the access matrix
- User who creates object can define access column for that object
- Can be expanded to dynamic protection
  - Operations to add, delete access rights
  - Special access rights:
    - *owner of* O_i
    - *copy op from* O_i *to* O_j *(denoted by "* ")*
    - *control –* D_i *can modify* D_j *access rights*
    - *transfer – switch from domain* D_i *to* D_j
  - *Copy* and *Owner* applicable to an object
  - *Control* applicable to domain object



Use of Access Matrix (Cont.)

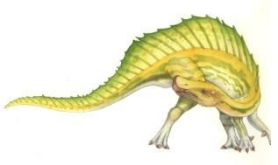
- **Access matrix** design separates mechanism from policy
  - Mechanism
    - Operating system provides access-matrix + rules
    - If ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced
  - Policy
    - User dictates policy
    - Who can access what object and in what mode
- But doesn't solve the general confinement problem



Access Matrix of Figure A with Domains as Objects

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

- The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (*) appended to the access right.
- The copy right allows the access right to be copied only within the column (that is, for the object) for which the right is defined



Access Matrix with *Copy* Rights

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

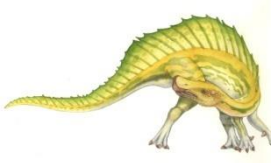
(a)

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

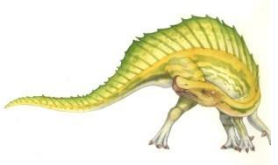
This scheme has two additional variants:

- A right is copied from access(i, j) to access(k, j); it is then removed from access(i, j).
  - This action is a transfer of a right, rather than a copy.
- Propagation of the copy right may be limited.
  - That is, when the right $R*$ is copied from access(i, j) to access(k, j), only the right R (not $R*$) is created.
  - A process executing in domain D_k cannot further copy the right R.



Access Matrix With *Owner* Rights

- Allow addition of new rights and removal of some rights.
- The owner right controls these operations
- If access(*i*, *j*) includes the owner right, then a process executing in domain D_i can add and remove any right in any entry in column *j*.



Access Matrix With *Owner* Rights

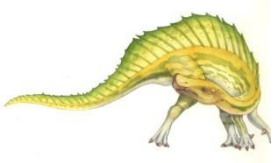
object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

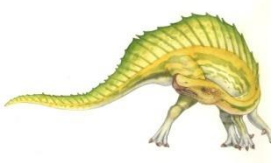
(b)

- The copy and owner rights allow a process to change the entries in a column.
- A mechanism is also needed to change the entries in a row.
- The control right is applicable only to domain objects.
- If access(*i*, *j*) includes the control right, then a process executing in domain D_i can remove any access right from row *j*.

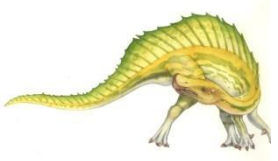


Modified Access Matrix of Figure B

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			



- The copy and owner rights provide us with a mechanism to limit the propagation of access rights.
- However, they do not give us the appropriate tools for preventing the propagation (or disclosure) of information.
- The problem of guaranteeing that no information initially held in an object can migrate outside of its execution environment is called the **confinement problem**.
- This problem is in general unsolvable



Implementation of Access Matrix

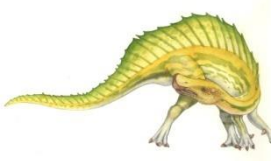
- Generally, a sparse matrix
- **Option 1 – Global table**
  - Store ordered triples **`<domain, object, rights-set>`** in table
  - A requested operation M on object O_j within domain D_i -> search table for $< D_i, O_j, R_k >$
    - with M $\in$ R_k
  - But table could be large -> won't fit in main memory
  - There is also no good way to specify groupings - If everyone has access to some resource, then it still needs a separate entry for every domain.

Implementation of Access Matrix (Cont.)

- **Option 2 – Access lists for objects**
- Each column = Access-control list for one object Defines who can perform what operation

  Domain 1 = Read, Write
  Domain 2 = Read
  Domain 3 = Read

- Each column of the table can be kept as a list of the access rights for that particular object, discarding blank entries.
  - Resulting per-object list consists of ordered pairs **<domain, rights-set>** defining all domains with non-empty set of access rights for the object



Implementation of Access Matrix (Cont.)

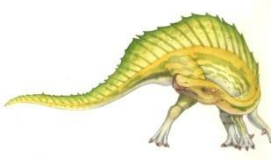
- When an operation M on an object O_j is attempted in domain D_i , search the access list for object O_j , looking for an entry $<D_i , R_k>$ with $M \in R_k$.
- If the entry is found, we allow the operation; if it is not, we check the default set.
- If M is in the default set, we allow the access. Otherwise, access is denied, and an exception condition occurs.
- For efficiency a separate list of default access rights can also be kept, and checked first.
  - Easily extended to contain default set -> If M ∈ default set, also allow access

Implementation of Access Matrix (Cont.)

- **Option 3 – Capability list for domains**
- Each row of the table can be kept as a list of the capabilities of that domain.
  - Instead of object-based, list is domain based
    - **Capability list** for domain is list of objects together with operations allows on them
    - Object represented by its name or address, called a **capability**
- To execute operation *M* on object *Oj* , the process executes the operation*M*, specifying the capability (or pointer) for object *Oj* as a parameter.

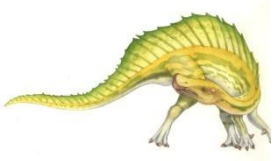
Implementation of Access Matrix (Cont.)

- Capability lists are associated with each domain, but not directly accessible by the domain or any user process.
  - Possession of capability means access is allowed
  - Rather, protected object, maintained by OS and accessed indirectly
  - Like a “secure pointer”
  - Capability-based protection relies on the fact that the capabilities are never allowed to migrate into any address space directly accessible by a user process (where they could be modified).
  - Idea can be extended up to applications



Implementation of Access Matrix (Cont.)

- **Option 4 – Lock-key**
  - Compromise between access lists and capability lists
  - Each object has list of unique bit patterns, called **locks**
  - Each domain as list of unique bit patterns called **keys**
  - Process in a domain can only access object if domain has key that matches one of the locks
  - Again, a process is not allowed to modify its own keys.

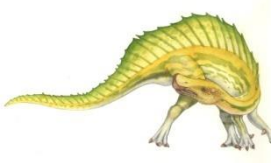


Comparison of Implementations

- Many trade-offs to consider
  - Global table is simple, but can be large
  - Access lists correspond to needs of users
    - Determining set of access rights for domain non-localized so difficult
    - Every access to an object must be checked
      - Many objects and access rights -> slow
  - Capability lists useful for localizing information for a given process
    - But revocation capabilities can be inefficient
  - Lock-key effective and flexible, keys can be passed freely from domain to domain, easy revocation

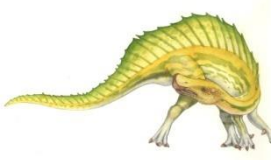
Comparison of Implementations (Cont.)

- Most systems use combination of access lists and capabilities
  - First access to an object -> access list searched
    - If allowed, capability created and attached to process
      - Additional accesses need not be checked
    - After last access, capability destroyed



Revocation of Access Rights

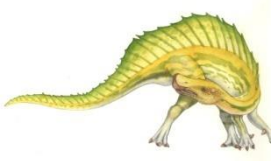
- The need to revoke access rights dynamically raises several questions:
  - Immediate versus delayed - If delayed, can we determine when the revocation will take place?
  - Selective versus general - Does revocation of an access right to an object affect *all* users who have that right, or only some users?
  - Partial versus total - Can a subset of rights for an object be revoked, or are all rights revoked at once?
  - Temporary versus permanent - If rights are revoked, is there a mechanism for processes to re-acquire some or all of the revoked rights?



- **Access List** – Delete access rights from access list
  - **Simple** – search access list and remove entry

Revocation of Access Rights (Cont.)

- With capabilities lists the problem is more complicated, because access rights are distributed throughout the system
- **Capability List** – Scheme required to locate capability in the system before capability can be revoked
  - **Reacquisition** - Capabilities are periodically revoked from each domain, which must then re-acquire them.
  - **Back-pointers** - A list of pointers is maintained from each object to each capability which is held for that object.
  - **Indirection** - Capabilities point to an entry in a global table rather than to the object. Access rights can be revoked by changing or invalidating the table entry, which may affect multiple processes, which must then re-acquire access rights to continue.



- **Keys** - A unique bit pattern is associated with each capability when created, which can be neither inspected nor modified by the process

- A master key is associated with each object.
  - key matches master key for access
- When a capability is created, its key is set to the object's master key.
- As long as the capability's key matches the object's key, then the capabilities remain valid.
- The object master key can be changed with the set-key command, thereby invalidating all current capabilities.