

15. File System Interface

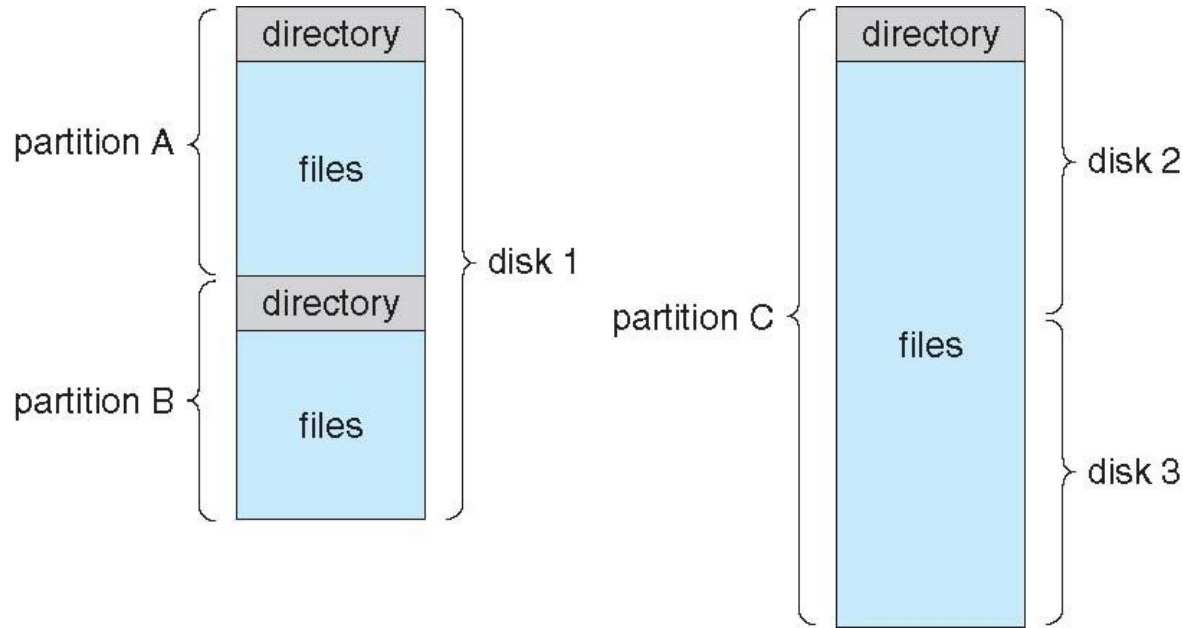
CS 4352 Operating Systems

File System

- A file system defines
 - How files are mapped onto secondary storage devices
 - How files are accessed
- A file system consists of two parts:
 - A directory structure
 - A collection of files

Typical File-system Organizations

- Disk can be subdivided into partitions
 - Partitions also known as minidisks, slices
- Disk or partition can be used raw – without a file system, or formatted with a file system



Types of File Systems

- We mostly talk of general-purpose file systems
- But systems frequently have many 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

Files

- File can be treated as an abstract data type that serves as the main information storage mechanism
- A file has several **file attributes**
 - Name
 - Type
 - Identifier within the file system
 - Location
 - Size
 - Access control information
- A file's type can be encoded in its name or contents
 - Windows encodes type in name
 - .exe, .bat, .dll, .jpg, etc.
 - Unix encodes type in contents
 - Magic numbers, initial characters (e.g., #! for shell scripts)

File Structures

- Who decides:
 - Operating system
 - Some OSes support a minimal number of file structures
 - Unix considers each regular file as a sequence of bytes
 - No interpretation of these bytes is made by the OS except for the executable files
 - All OSes must support at least one file structure -- the file structure of an executable file
 - Applications
- File types may be used to indicate file structures

File Operations

- File can be treated as an abstract data type
 - Create
 - Write – at write pointer location
 - Read – at read pointer location
 - Seek to reposition within file
 - 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

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 mediates access to a file
 - 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

Directory Structure

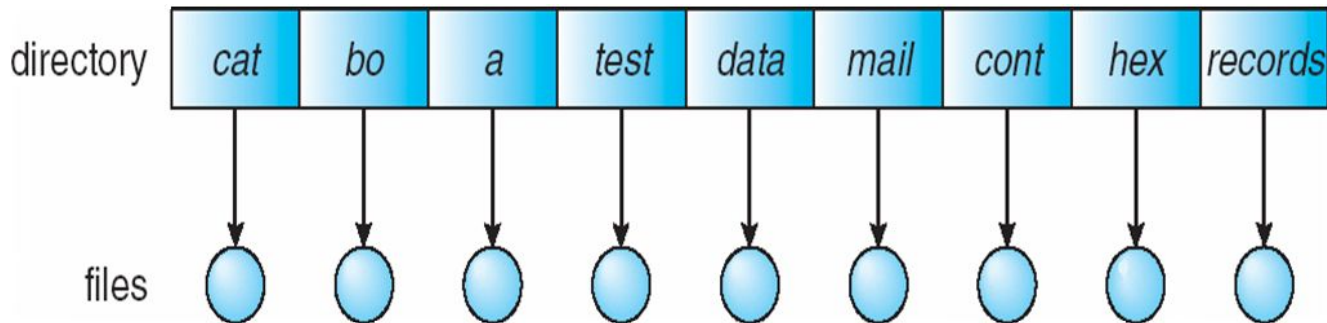
- A directory is a collection of entries containing information about the files under it
- Directory structure defines how directories are organized
 - The directories are organized logically to obtain
 - Efficiency – locating a file quickly
 - Naming – convenient to users
 - Two users can have same name for different files
 - The same file can have several different names
 - The sequence of directories searched when a file name is given is called the search path
- Both the directory structure and the files reside on disk

Operations Performed on Directory

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

Single-Level Directory

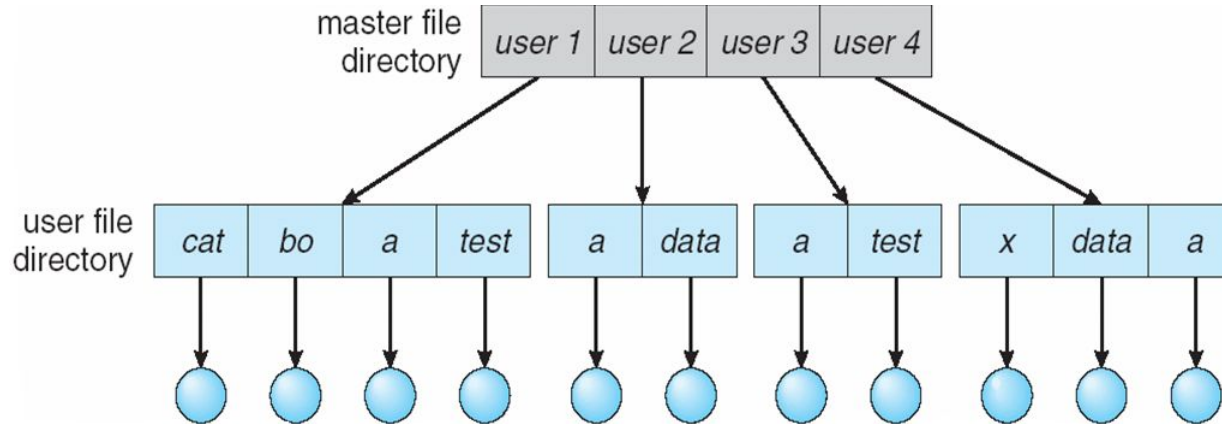
- A single directory for all users
 - The simplest directory structure



- Naming problem

Two-Level Directory

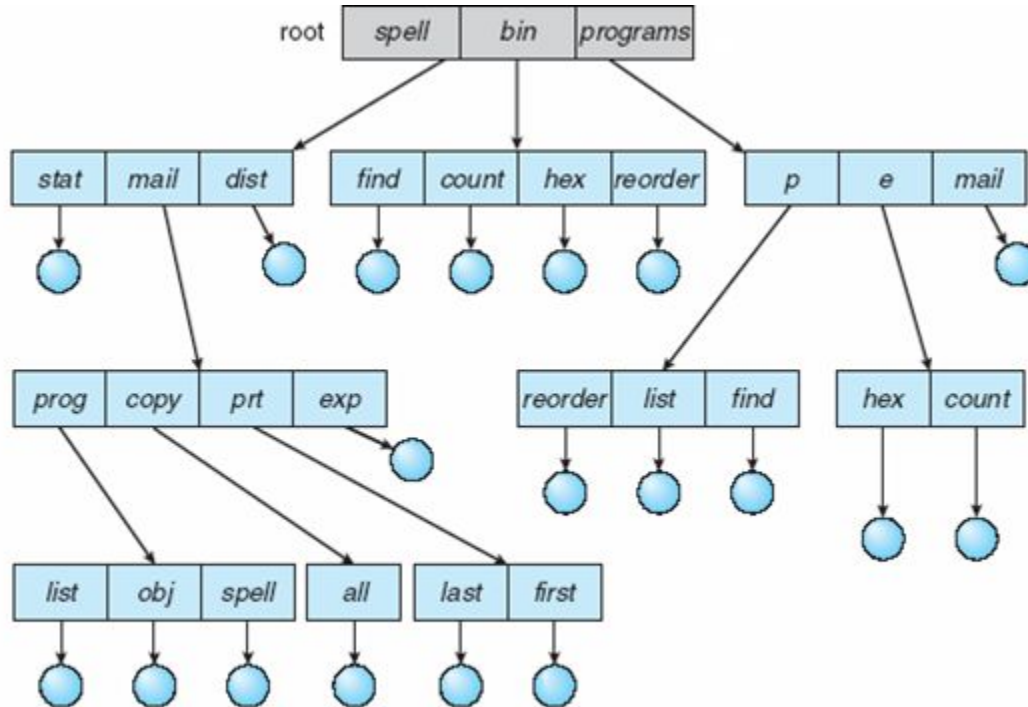
- Separate directory for each user
 - Can have the same file name for different user



Tree-Structured Directories

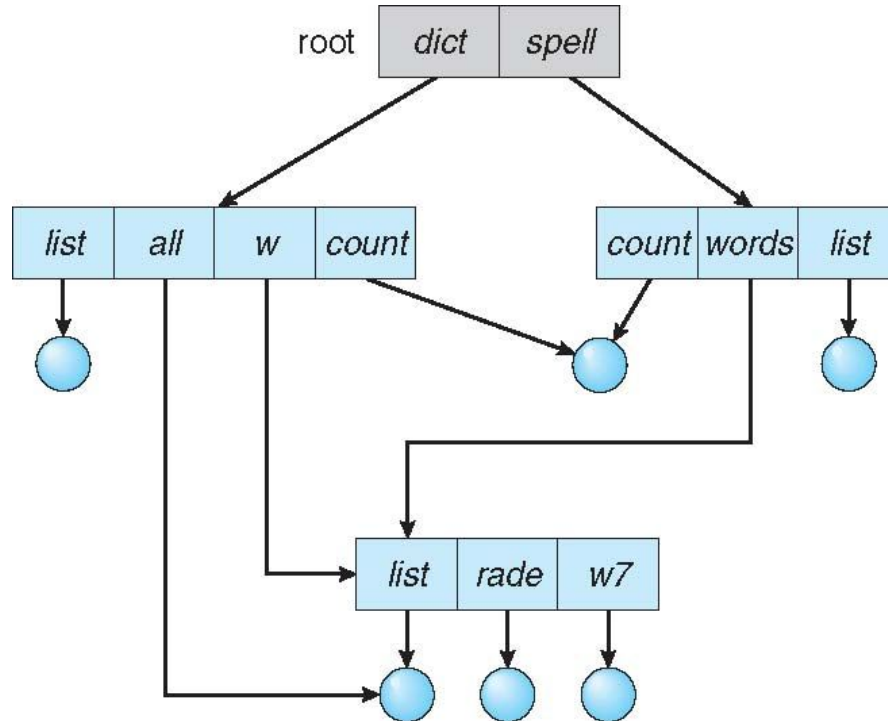
- There is a root directory
- A directory has a set of files & subdirectories
 - One bit in the directory entry defines if it is a file or a subdirectory
 - A directory is actually a file but treated in a special way
- A directory can be deleted when it is empty
- Every file has a unique path name
 - Pure tree-structured directory prohibits file & directory sharing
 - A more flexible directory structure is the acyclic-graph directory structure

Tree-Structured Directories (Cont.)



Acyclic-Graph Directories

- Have shared subdirectories and files



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 on a 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

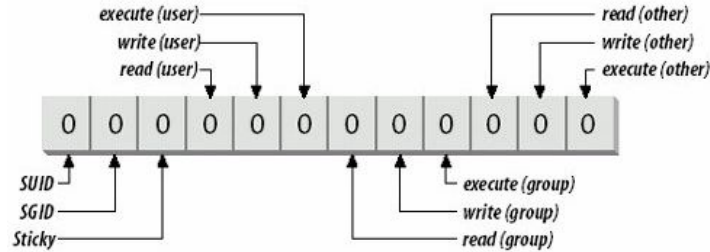
File Sharing – Remote File Systems

- Uses networking to allow file system access between systems
 - Manually via programs like FTP
 - Automatically, seamlessly using distributed file systems
- Client-server model allows clients to mount remote file systems from servers
 - Server can serve multiple clients
 - NFS is standard UNIX client-server file sharing protocol
 - CIFS is standard Windows protocol
 - Standard operating system file calls are translated into remote calls

Protection

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

Unix/Linux File Access Control



- When a file is created, it is owned by its creator
 - Owner can be changed using *chown* command
 - It also belongs to a specific group, which initially is either its creator's primary group, or the group of its parent directory if that directory has setgid permission set
 - Group can also be changed using *chown* command, or *chgrp* command
- There is a set of 12 protection bits for a file
 - Protection bits can be changed using *chmod* command
 - File access control can be realized by these 12 protection bits
 - 9 of the protection bits specify read, write, and execute permissions for the owner of the file, other members of the group to which this file belongs, and all other users
 - Three sets of bits (each set has 3 bits) to specify permissions
 - 3 remaining bits define special additional behavior

Permissions

```
drwxrwxr-x 2 zzk zzk 4096 Sep 7 12:13 directoryExample
-rwxrwxr-x 1 zzk zzk 37844928 Sep 7 12:17 executableExample
-rw-rw-r-- 1 zzk zzk 775 Sep 7 12:17 textFileExample
```

Binary Octal

---	no permission	000	0
--x	execute	001	1
-w-	write	010	2
-wx	write, execute	011	3
r--	read	100	4
r-x	read, execute	101	5
rw-	read, write	110	6
rwX	read, write, execute	111	7

Other class → Group name

Group class → Owner name

Owner class

Meanings for directories are different from regular files

- r: allows to list files within the directory
- w: allows to create, delete, rename files within the directory, and modify the directory's attributes
- x: allows to enter the directory and access files within the directory
 - ls /home/zzk
 - / x, home x, zzk r (what about --x)
 - cd /home/zzk
 - / x, home x, zzk x (what about r--)
 - cat /home/zzk/abc.txt
 - / x, home x, zzk x, abc.txt r

ACL Extension

- You may ask “Can I give my friend Kev a special treatment instead of the general group/other?”
 - Modern Unix-like OS supports access control lists
 - Any number of users and groups can be associated with a file, each with three bits for read, write, and execute access rights (using *setfacl* command by owner or root)
 - ACL is not a must, so a file may be protected solely by the traditional file access mechanism

```
zzk@isis:~/courses/cs4285$ setfacl -m u:kev:rwX executableExample
zzk@isis:~/courses/cs4285$ getfacl executableExample
# file: executableExample
# owner: zzk
# group: zzk
user::rwX
user:kev:rwX
group::rwX
mask::rwX
other::r-x
```

```
zzk@isis:~/courses/cs4285$ chmod g-w executableExample
zzk@isis:~/courses/cs4285$ getfacl executableExample
# group: zzk
user::rwX
user:kev:rwX
group::rwX
mask::r-x
other::r-x
#effective:r-x
#effective:r-x
```

+ denotes ACL extension is used

- Permissions for a named user or group in the ACL also depends on the mask
 - The owner class and other class entries in the 9-bit permission field have the same meaning as in the traditional file access mechanism
 - The group bits specify the maximum permissions that can be assigned to the named users or groups in the ACL (i.e., serve as a mask)

Different IDs of User and Process

- Each user is associated with

- uid: user ID (a unique positive integer)
- gid: user's primary group ID
- Also some other IDs of its supplementary groups

```
zzk@isis:~$ id zzk
uid=1000(zzk) gid=1000(zzk) groups=1000(zzk),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(docker)
```

- Each process is associated with

- ruid: real user ID, which is the uid of the user/process that created this process
- euid: effective user ID, which is the uid for evaluating privileges of the process
- rgid: real group ID, which is the gid of the user/process that created this process
- egid: effective group ID, which is the gid for evaluating privileges of the process
- Also some other IDs (e.g., saved uid, saved gid, supplementary group IDs)

- After a user login, the shell launched gets all the IDs from the user

- A spawned process inherits all the IDs from its parent

Effective IDs

- When a process needs to access a file, permissions are checked against the process's effective IDs
 - If euid is the file's uid, use the owner class permission bits
 - If egid or any supplementary gid's is the file's gid, use the group class permission bits
 - Otherwise, use the other class permission bits
- When a process creates a file, effective IDs are used
 - The file's uid is the process's euid
 - The file's gid is the process's egid or the directory's gid if the setgid of directory is set

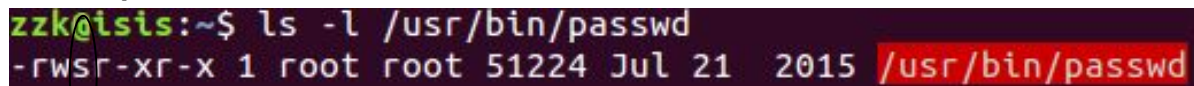
Setuid Programs

- Executables with “setuid bit” set

- When such programs run, the process’s euid will become the uid of the program owner
 - The process then has the privileges of the program owner
- Why do we need setuid programs? Take “sudo” for an example
 - If a user is in the /etc/sudoers file, he/she can use his/her own password to run such as privileged programs (e.g. for administrative tasks)
 - /etc/sudoers can only be read by root (ls -l gives “r-- r-- --- 1 root root”)
 - Since “sudo” is a setuid-root program, the process is enabled to read /etc/sudoers

- Common setuid program examples

- sudo: supervisor do
- su: switch user
- passwd: change password
- chsh: change login shell



```
zzk@isis:~$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 51224 Jul 21 2015 /usr/bin/passwd
```

“s” instead of “x”: setuid (and can be executed)

Setuid Programs (Cont.)

- We can set the setuid bit using chmod
 - chmod 4xxx executable (symbolically chmod u+s executable)
- Although a setuid program runs with the privilege of the program owner, the program logic should prevent the users from doing anything else other than the intended functionality
 - If there is some vulnerability that can be exploited to control the execution (e.g., buffer overflow), the program logic is meaningless
 - The program will do things it was never intended to do

Setgid

- Similar to setuid programs, when a setgid programs runs, the process's egid is the program's gid
- However, it only makes sense to use setuid permission for executables, but it makes sense to use setgid permission for both executables and directories
 - Any file created in a directory with setgid bit set will have the directory's gid
 - Any directory created in a directory with setgid bit set will have its setgid bit set

```
drwxrwsr-x 2 zzk zzk 4096 Oct  4 11:52 dir1
-rw-rwSr-- 1 zzk zzk    0 Oct  4 11:52 file1
```

s: setgid and can be executed
S: setgid and cannot be executed

Sticky Bit

- If a directory has its sticky bit set, the files in this directory can only be renamed or deleted by the file owner or the directory owner
 - Why do we need this sticky bit?
 - Think of some directory such as /tmp that has 777 permission (namely anyone can rename or delete files in the directory no matter whether he/she is the owner)
 - If you do not want others to rename or delete your files in such directories, these directories need to have sticky bit set
- When a directory has its sticky bit set, you will see t/T in the permission for others

t: sticky and can be executed

```
drwxrwxrwt 2 zzk zzk 4096 Oct  4 11:57 stickyDir1
drwxrwxrwt 2 zzk zzk 4096 Oct  4 11:57 stickyDir2
```

T: sticky and cannot be executed

Bonus Time

- What do Unix/Linux access-control permission bits control for directories?
 - A. If the execute bit of the owner class is set, it enables the owner to run executable files within the directory
 - B. If the write bit of the owner class is set, it enables the owner to modify the contents of files within the directory
 - C. If the read bit of the owner class is set, it enables the owner to list files within the directory
 - D. If the sticky bit is set, it prevents a non-privileged user from deleting other users' files within the directory (unless it is the directory owner)
 - E. It does not make sense to use setgid bit for directories

Next Lecture

We look into file system implementations