10.4 Disk Scheduling

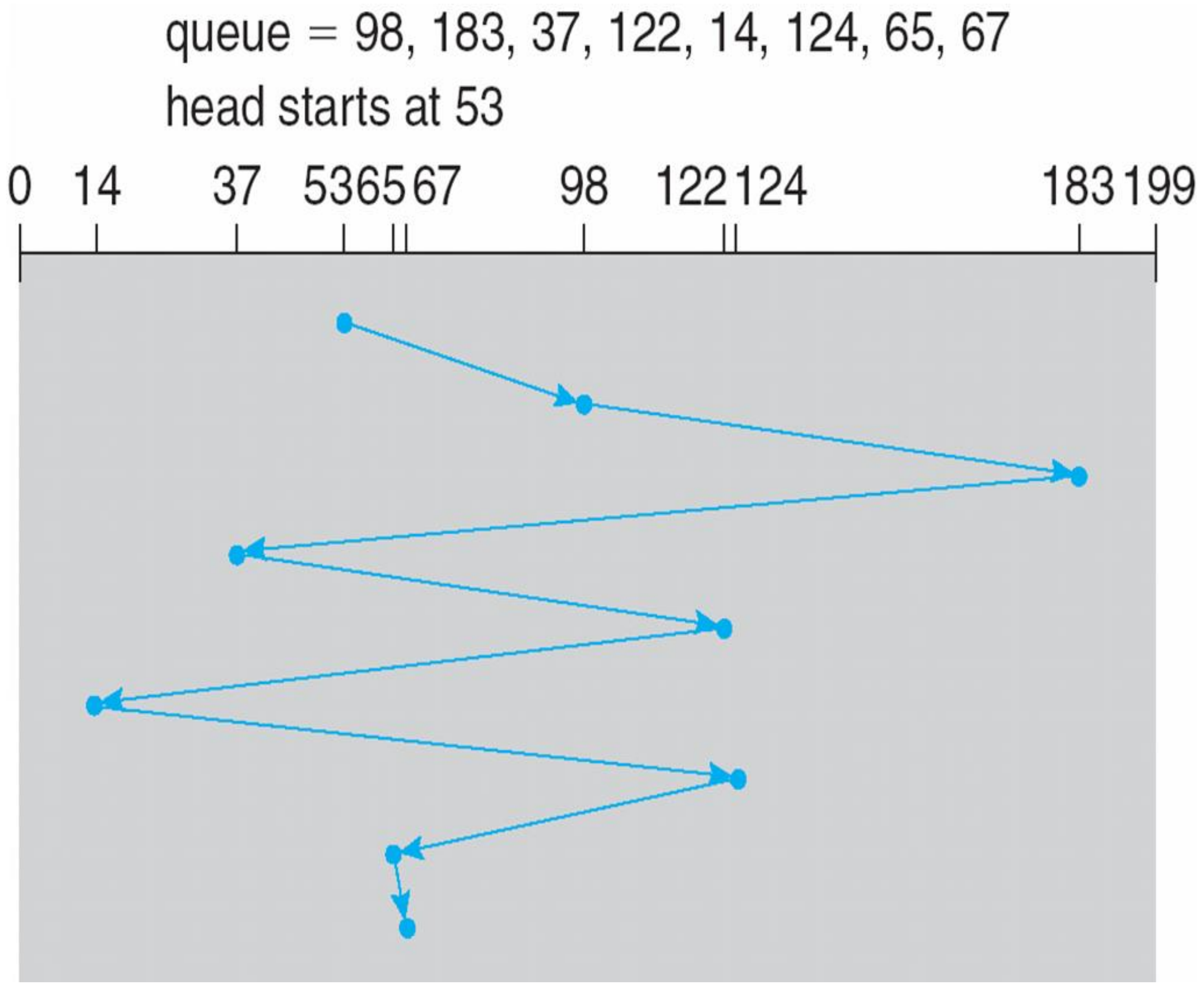
- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth
- Minimize seek time
- Seek time ≈ seek distance
- Disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer

Disk Scheduling - cont.

- There are many sources of disk I/O request
  - OS
  - System processes
  - Users processes
- To access a disk, an I/O request to the driver is made and it includes:
  - Input or output mode
  - Disk address
  - Memory address
  - Number of blocks to transfer
- OS maintains **queue of requests**, per disk or device
  - Note that drive controllers have limited buffers can manage a finite sized queue of I/O requests (of varying “depths”)
  - Idle disk can immediately work on I/O request
  - Busy disk means work must be queued
    - Optimization algorithms only make sense when a queue exists
    - The analysis is true for one or many platters

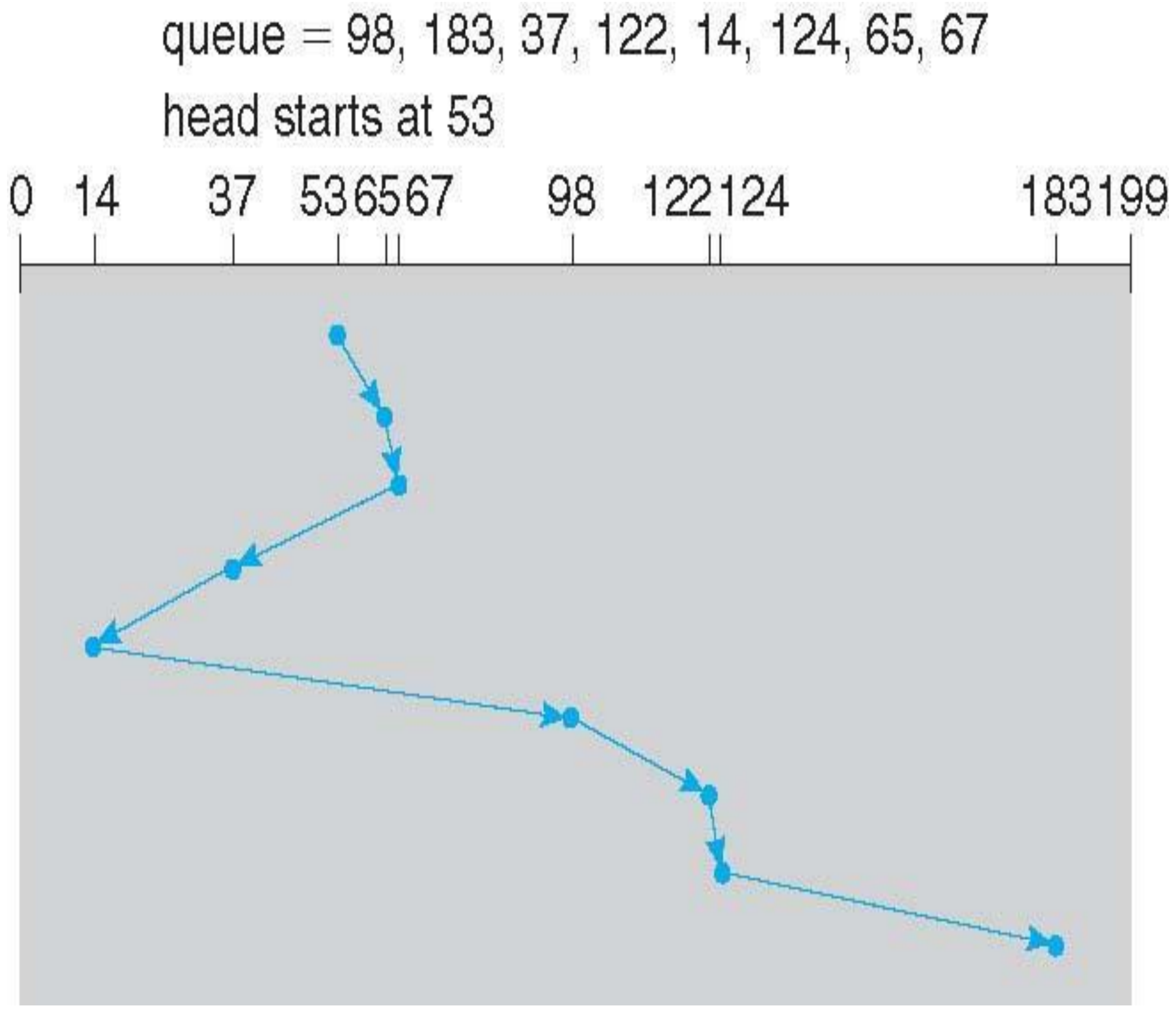
FCFS

- Requests are indicated by the requested **cylinder number**.
- As the name **First Come First Serve** implies, requests at the head of queue are served first.
- Illustration shows total head movement of 640 cylinders



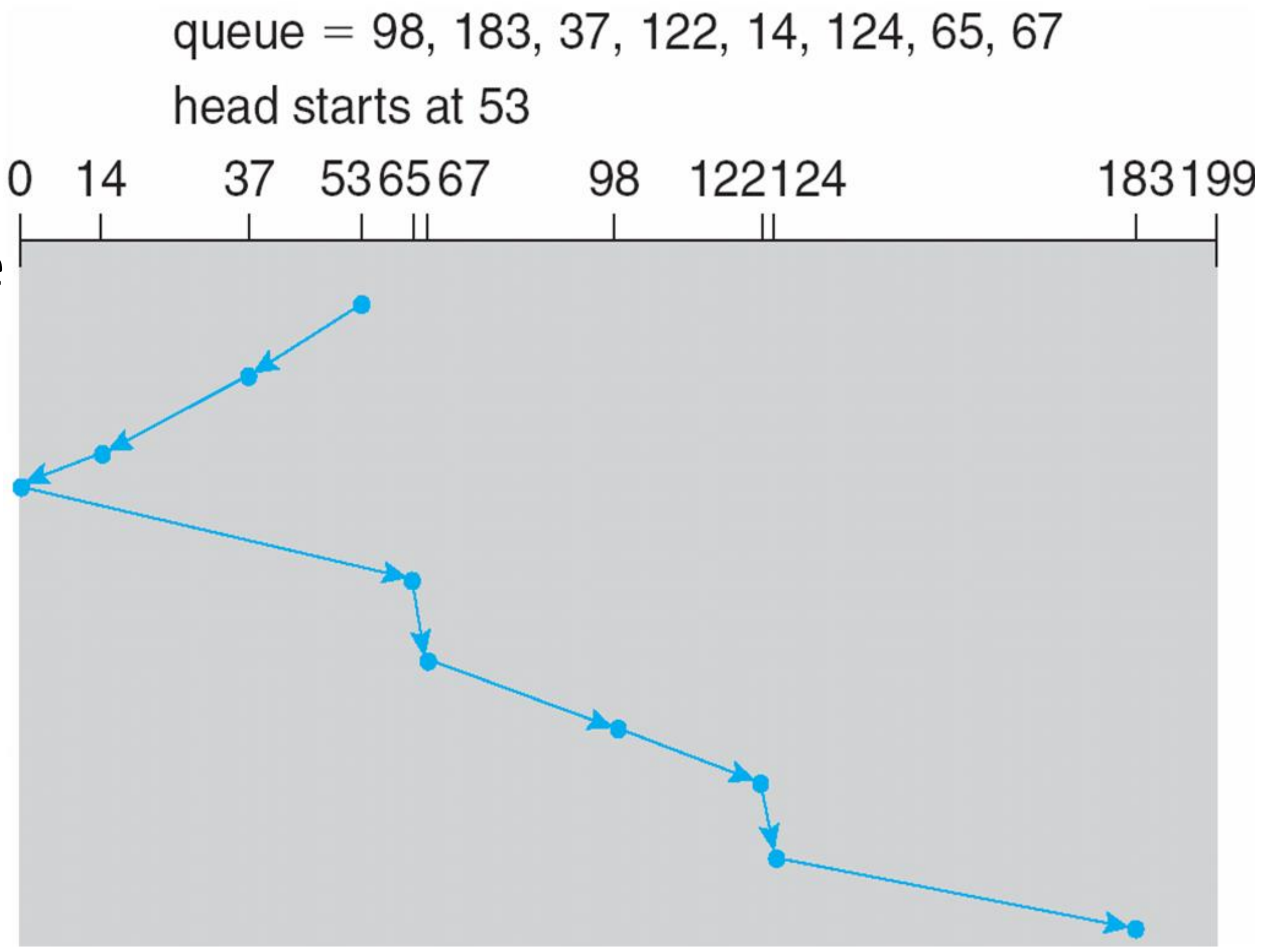
SSTF

- **Shortest Seek Time First** selects the request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF (Shortest Job First) scheduling; may cause starvation of some requests
- Illustration shows total head movement of 236 cylinders



SCAN

- The disk arm starts at one end of the disk and moves toward the other end (i.e. in one direction), servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- **SCAN algorithm** Sometimes called the **elevator algorithm**
- Illustration shows total head movement of 236 cylinders
- But note that if requests are uniformly dense, requests for cylinders at either ends of disk wait the longest

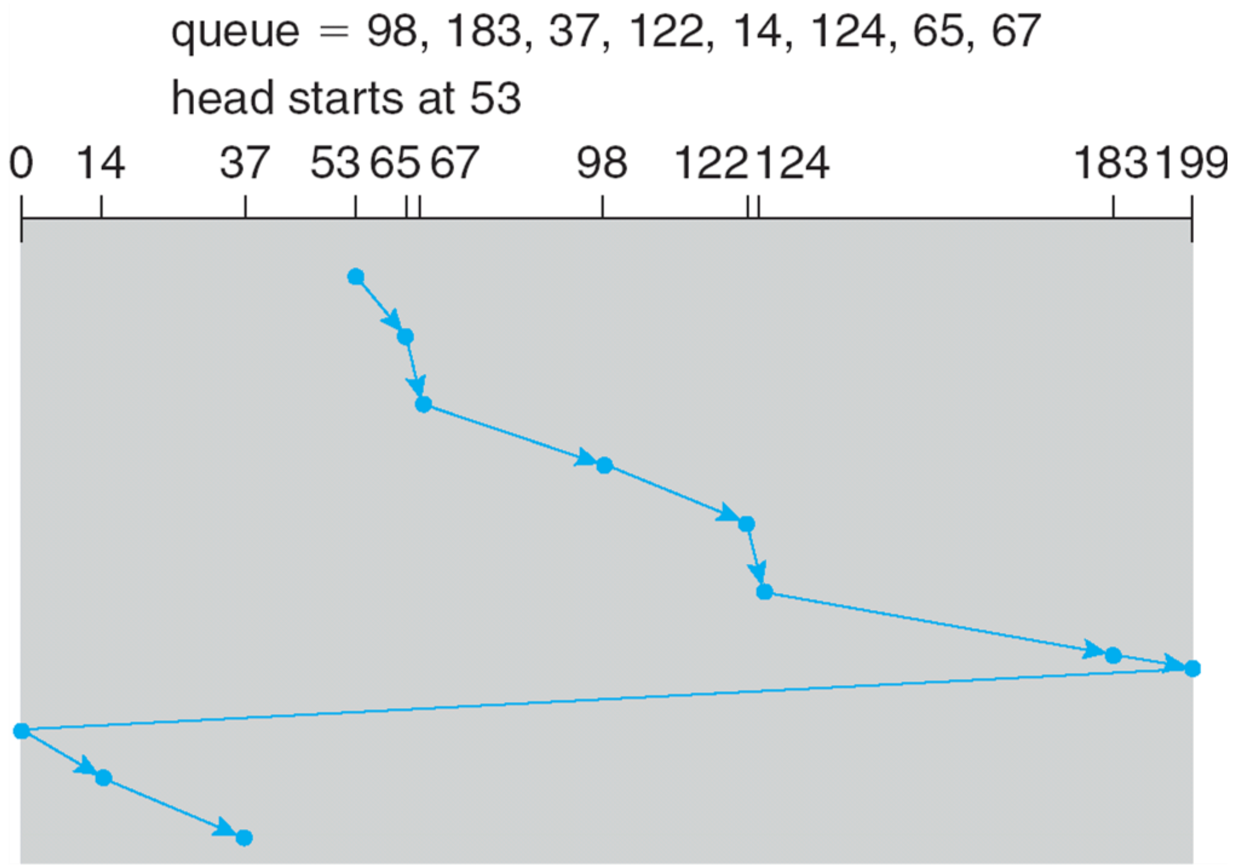


C-SCAN

- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one
- Total number of cylinders?

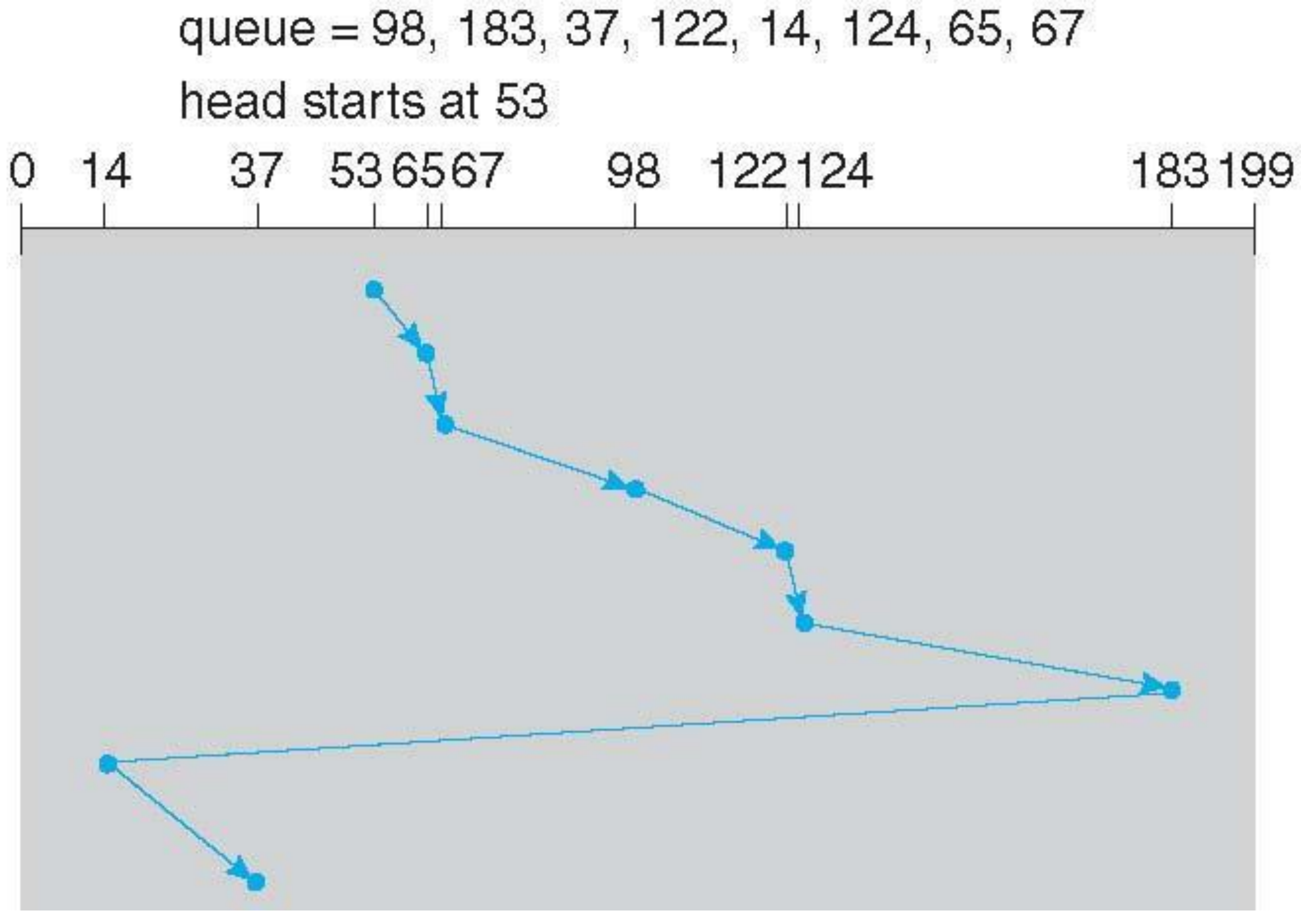
  (199-53) + (37-0) = 183

  - Note that the return-to-beginning trip relatively much quicker and is ignored.



C-LOOK

- LOOK a version of SCAN, C-LOOK version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk
- Total number of cylinders?
  - (183-53) + (37-14) = 153

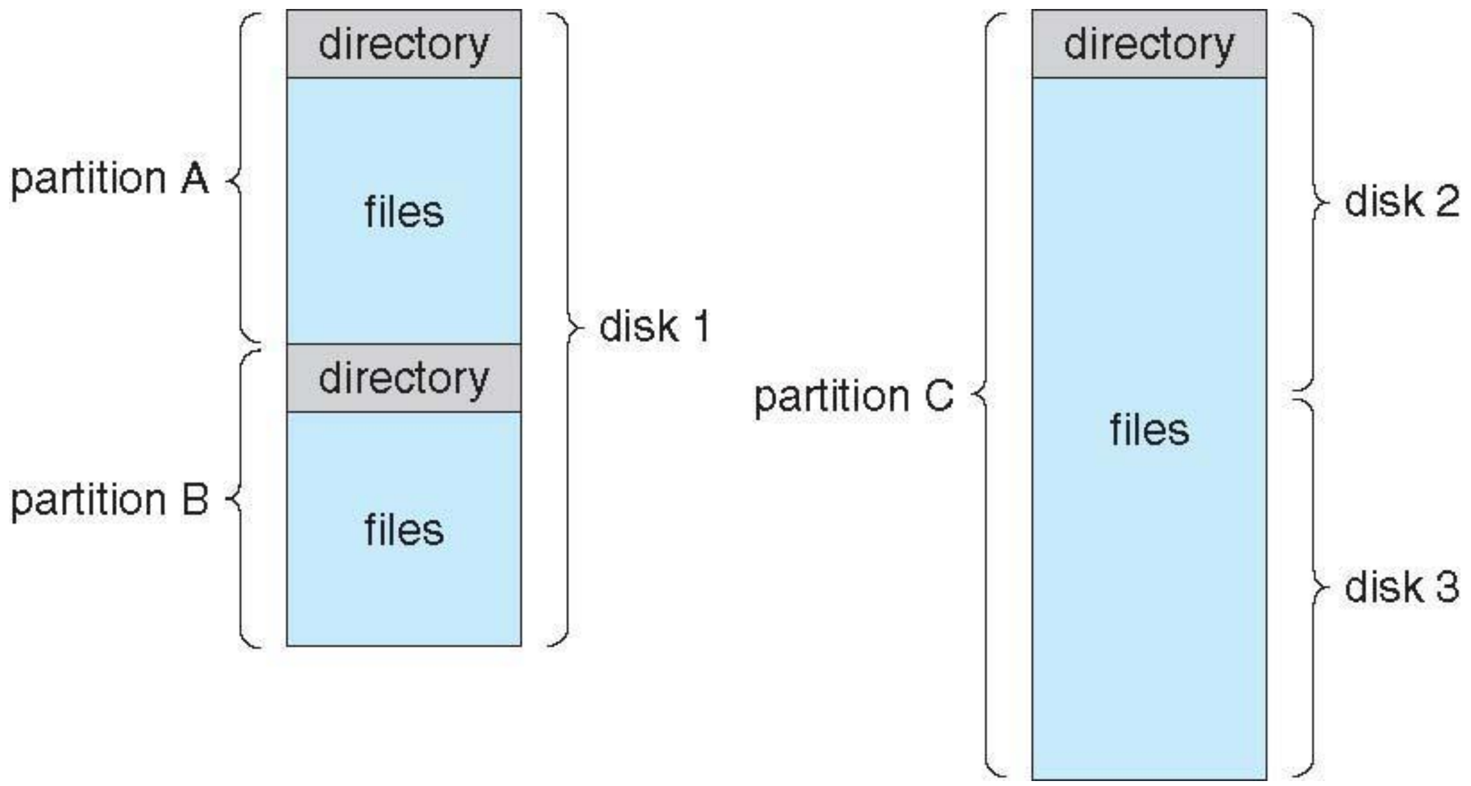


Disk Structure and file systems

- Disk can be subdivided into **partitions**
  - Partitions also known as minidisks, slices
- Disks or partitions can be **RAID** protected against failure
  - RAID, redundant array of independent disks improve reliability and performance.
- Disk or partition can be used:
  - **raw** – without a file system, or
  - **formatted** with a file system
- A partition containing (i.e. formatted with) a file system is known as a **volume**
- Each volume containing file system also tracks that file system’s info in **device directory** or **volume table of contents**
  - Root dentry in Linux

A Typical File-system Organization

- Linux allows volumes (not partitions) to span multiple disks using the **Logical volume management tool** (LVM).

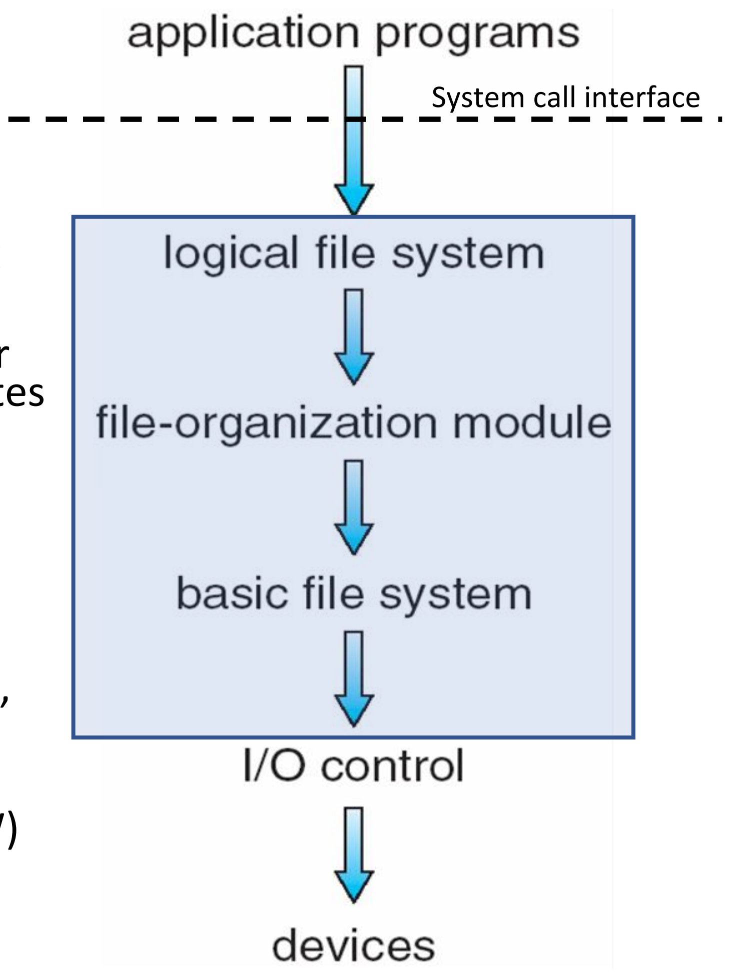


File-System Structure (from Ch. 12-1)

- **File system** resides on secondary storage (disks, flash memory, etc.)
  - Provides user interface to storage – defines:
    - A directory structure
    - File attributes (where data blocks reside, access rights, etc.)
    - Operations that may be performed on files.
  - Maps logical filesystem view to physical media (disk, flash, network, etc.)
- File
  - Logical storage unit
  - Physically, a collection of related information (data may be scattered across multiple non-contiguous blocks)
  - I/O transfers performed in **blocks.** A block is one or more **sectors.** A sector is 32 to 4096 bytes (usually 512 bytes)
  - **File control block** – object containing information about a file (**inode** in Linux)

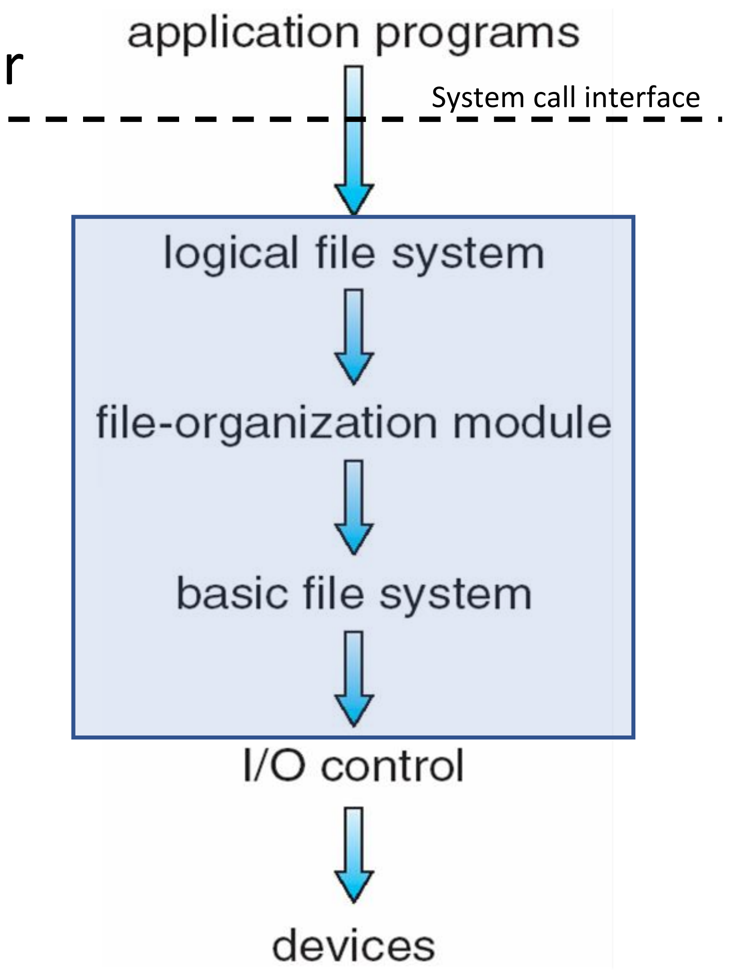
Layered File System

- The filesystem is implemented using a hierarchical (layered) design – every layer:
  - Uses services provided by lower layer
  - Provides services to the layer above it.
- The **I/O control layer** consists of **device drivers**, including their interrupt handlers.
  - Translates high level commands such as "read drive1, cylinder 72, track 2, sector 10, into memory location 1060" into lower memory-mapped register reads/writes into the device controller hardware.
  - Register writes/reads contain specific commands/status to the hardware controller.
- **Basic file system** translates commands such as "retrieve block 123" to device driver commands.
  - It may deal with different types of device drivers – e.g. disk drivers, flash drivers, etc.
  - **Abstracts the secondary storage as a linear set** of **physical blocks**.
  - Marks defective blocks and either avoids using them or tells the controller (H/W) to avoid using them.
  - Manages memory buffers and caches (allocation, freeing, replacement)
    - Buffers hold data in transit
    - Caches hold frequently used data



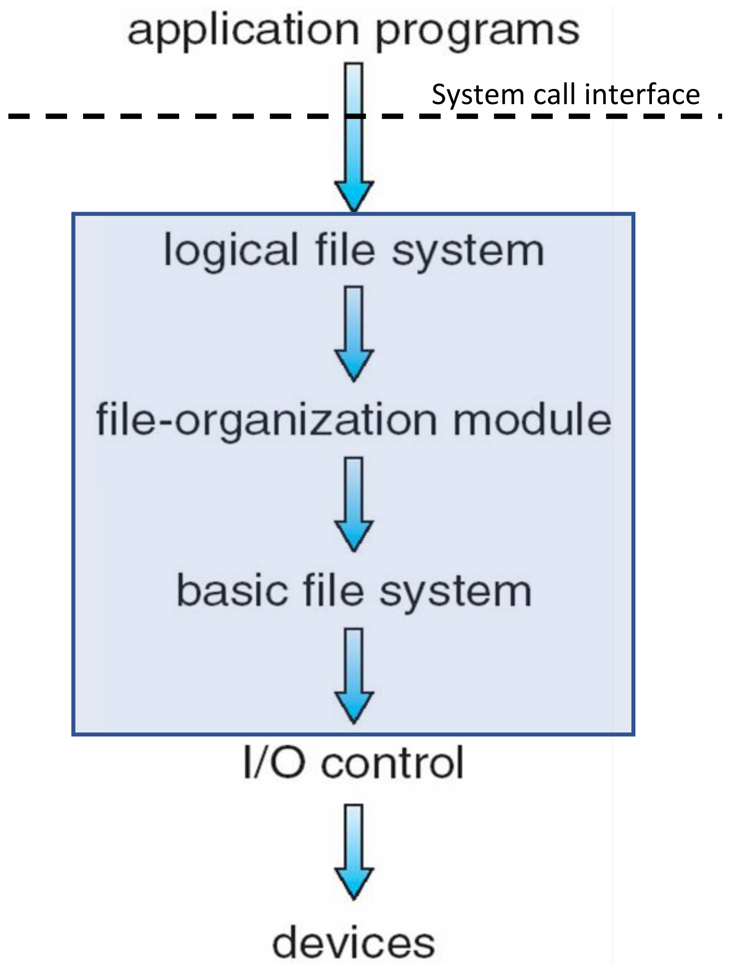
Layered File System – cont.

- **File organization module** understands files, **logical blocks** (or records), and physical blocks/records
  - **Abstracts a file as** a contiguous set of logical blocks
  - Translates a file's logical block # to physical block #. Physical blocks may not be allocated in a contiguous manner.
  - Manages free storage space, i.e. disk allocation
- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Directory management
  - Protection (inodes include info about ownerships and permissions)



Layered File System – cont.

- Layering reduces complexity and eases debugging but adds overhead and can decrease performance.
- An operating system may implement many file systems, sometimes many within an operating system, each with its own format.
  - CD-ROM (ISO 9660), Floppy, DVD, Blu-ray.
  - Windows has FAT, FAT32, NTFS
  - Unix has **UFS** (based on Berkeley's FFS), NFS
  - Linux has more than 40 types, with **extended file system** ext3, ext4 and NFS leading.
  - GoogleFS (for massive cloud storage arrays/networks)
  - New ones still arriving – ZFS, FUSE (user-mode filesystem)



Types of File Systems

- We mostly talk of general-purpose file systems
  - e.g. UFS or ext4 in Linux
- But systems may frequently have file systems, some general- and some special- purpose or virtual
  - e.g. proc and dev filesystems in Linux (mounted at /proc and /dev)

11.1 File Concept

- A unit of storage with some abstraction:
  - Separate blocks of bytes on different tracks/sectors $\xrightarrow{\text{ABSTRACTED AS}}$ contiguous chunk of bytes
  - Underlying storage may be:
    - A hard disk, magnetic tape, etc.
    - A network storage device
- A file may also be virtual
  - A Unix pipe
  - A virtual filesystem, that complies with the file system interface APIs, e.g. nodes in:
    - /proc
    - /sys
    - /dev
- File extensions/types refer to the format of data stored within files and is user (not OS) defined, e.g. text files, executable file, images, etc.
  - Thus not a topic of discussion in this course.

11.1.1 File Attributes

- **Name** – only information kept in human-readable form (an ascii string)
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to a device and the location of the file within the device
- **Size** – current file size (in bytes or blocks), and max allowed size.
- **User identification and Protection** – controls who can do reading, writing, executing
- **Time and date** – data for usage monitoring
- Many variations, including extended file attributes such as character encoding and file checksum
- Information about files are kept in the **directory structure**, on the same secondary storage
    - Typically a table of “file name - file ID” entries.
    - Used to access additional file information (may be in excess of 1 kB per file)

11.1.2 File Operations

Despite that most OSes are written in C, a file is implemented **as an object** (as in C++). Thus it has data and operations that manipulate that data:

- **Create** – in most systems, this takes place within the open() operation. It involves adding an entry in the directory structure and also allocating storage space for the file on secondary storage.
- ***Open(F_i)*** – create or open an existing file. Search the directory structure on disk for entry ***F_i***, and move the content of entry to memory.
- **Write –** at **write pointer** location (then increment the pointer)
- **Read –** at **read pointer** location (then increment the pointer)
- **Seek -** Reposition within file
  - Usually write pointer and read pointer are unified into a file position pointer that is manipulated by seek
- **Delete**
- **Truncate**
- ***Close (F_i)*** – move the content of entry ***F_i*** in memory to disk

File operations – cont.

- Other basic operations include:
  - **Renaming** a file,
  - Determining the **status** of a file, such as the file's length, or
  - **Setting file attributes** such as the file's owner.
- **Complex operations** may be constructed by combining the basic operations.
  - e.g. copying a file may involve creation of destination file, opening of source file and reading data from source file and writing it to destination file.
- Most commonly, when `open()` is invoked, the system adds an entry in the **open-file table** and returns an **index** to that entry.
  - The entry may contain a pointer to the file data structure (**file object**).
  - The index can be used thereafter for subsequent file operations – makes the file data easy to locate.

File operations – cont.

- The **system-wide open-file table**
  - Stores process independent information (or a pointer to that information) such as file location on disk, size, etc.
  - It also contains a **File-open count** (per entry) to keep track of number of times a file is open. This allows removing its entry from the open-file table when last process closes it.
- There exists a **per-process open-file table** storing process dependent information such as:
  - Access rights - e.g. if the file is open for read, then the per-process information indicates a read-only access right.
  - File position pointer.
  - It must also contain a pointer to the file’s entry in the system-wide open-file table.
- When another process opens the same file, it uses the existing system-wide open-file entry and the file-open count is incremented by 1.

Open File Locking

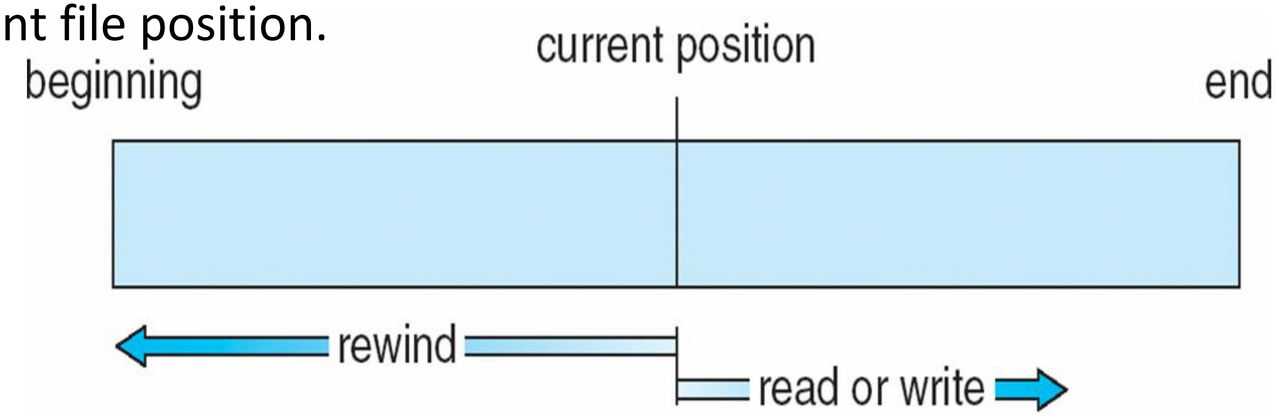
- Provided by some operating systems and file systems
  - Similar to reader-writer locks
    - **Shared lock** similar to reader lock – several processes can acquire concurrently for reading
    - **Exclusive lock** similar to writer lock
  - Some Oses may only support exclusive locks.
- Mandatory or advisory:
  - **Mandatory** (Windows) – access is denied depending on locks held and requested
  - **Advisory** (Unix)– system allows access to files, and the processes (not the system) must ensure they acquire the appropriate locks before operating on the file, e.g. if a file has an exclusive lock, the system allows any process to write to the file, but the process must prevent itself from doing so, till it acquires the lock and then write to the file.

```
int flock(int fd, int operation); // operation=LOCK_EX,
                                  //LOCK_SH or LOCK_UN
```

- Care must be taken to avoid deadlocks.

11.2 Access Methods

- Sequential Access
  - Modeled after tape storage
  - Reads and writes operate on current file position pointer, e.g. **read_next()**
  - Seek can move the file position pointer relative to current position (in some systems) or relative to the beginning of the file (in other systems).
    - In Linux and c standard library, an offset parameter is passed to **`seek()`** along with another parameter indicating whether to measure from start, end or current file position.

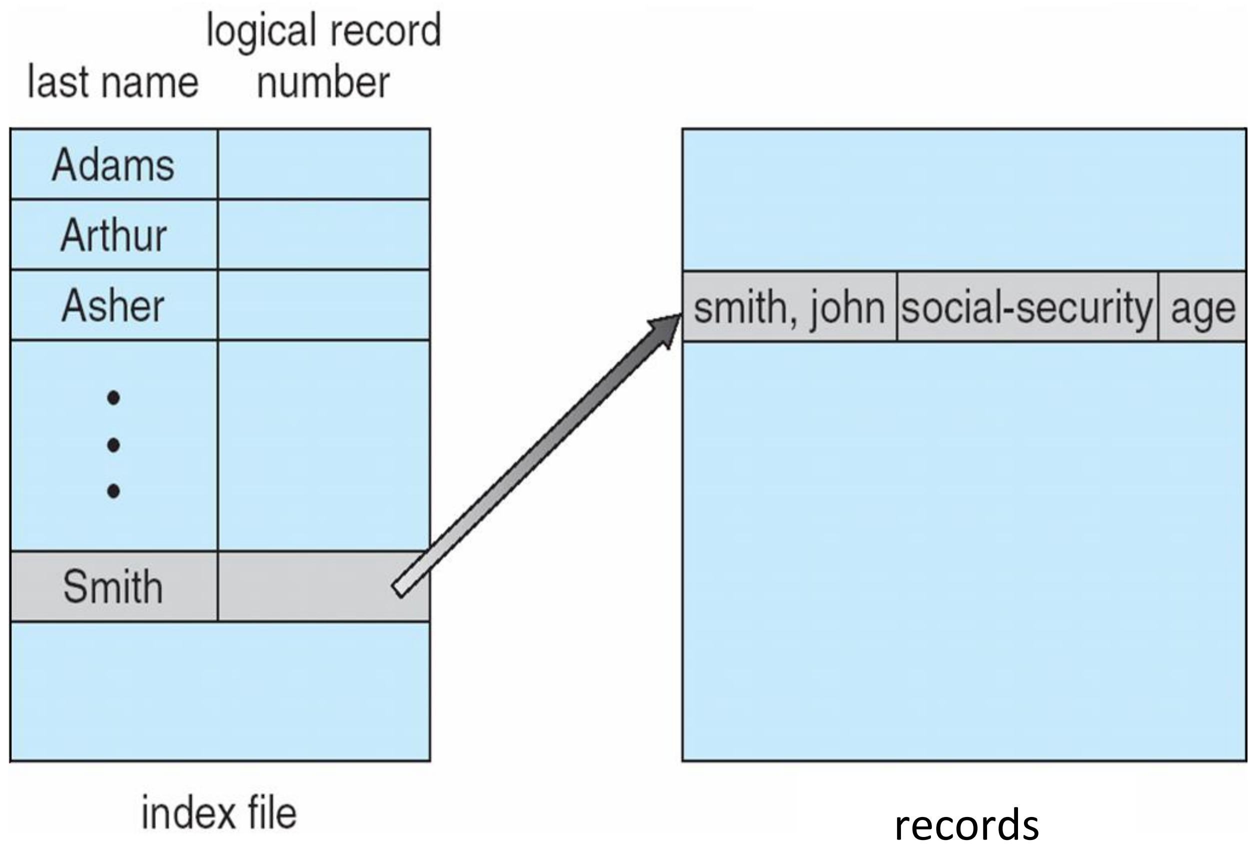


Access Methods – cont.

- Direct Access
  - Modeled after hard disk drivers
  - File operation allow access to any record (= block) within the file, e.g. **read(n)**.
  - Access uses **logical block numbers** (=relative = block number within the file, block 0 being the first) and the system will translate the logical block number to a physical block number on the storage disk/tape.
  - Sequential access may be implemented (or simulated) using direct access, and vise versa if seek is allowed.

Access Methods – cont.

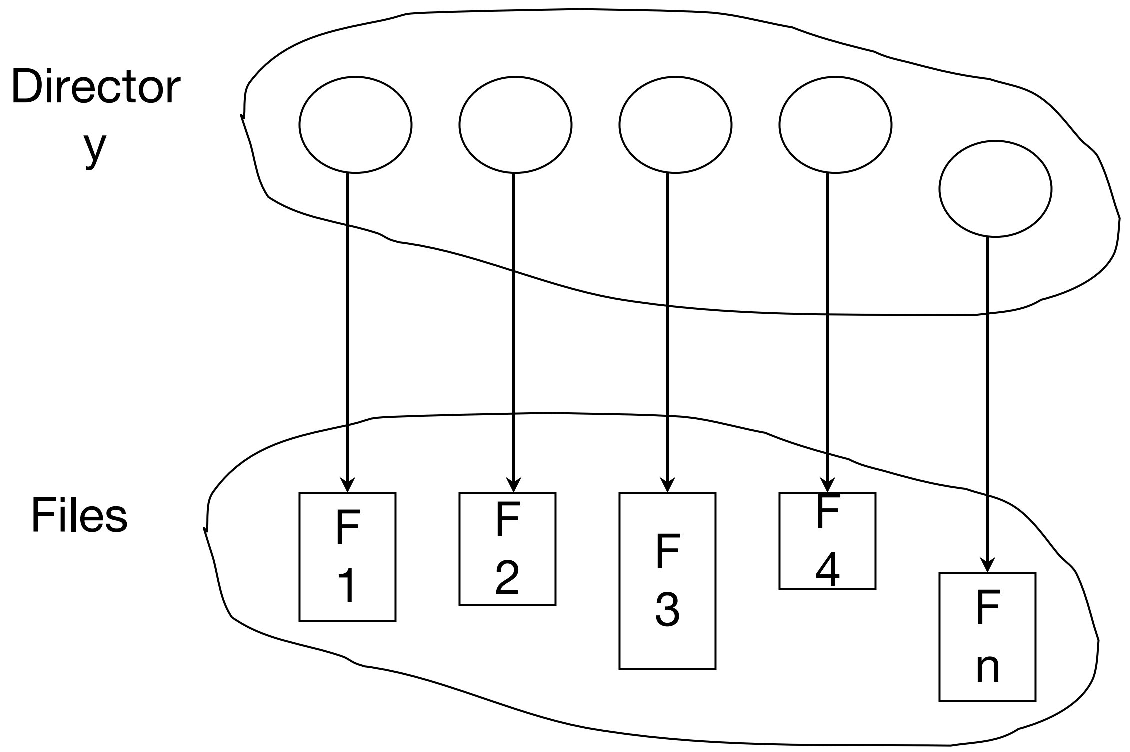
- Indexed files
  - Built on top of random access files.
  - A file is divided into records (instead of raw bytes).
  - Each entry in the index contains a key (e.g. last name) and the corresponding logical record number where the record is stored.
  - Entries are sorted by the key.
  - Finding an entry using its key (e.g. last name) thus involves a binary search within the index to obtain the logical record number, followed by a random access of the record.
  - e.g. IBM’s Indexed Sequential Access Method (ISAM)
  - e.g. Vendor management systems (VMS) relative and indexed files.



11.3 Directory Structure

Directory Structure:

- A collection of nodes containing information about all files
  - Both the directory structure and the files reside on disk



Operations Performed on Directory

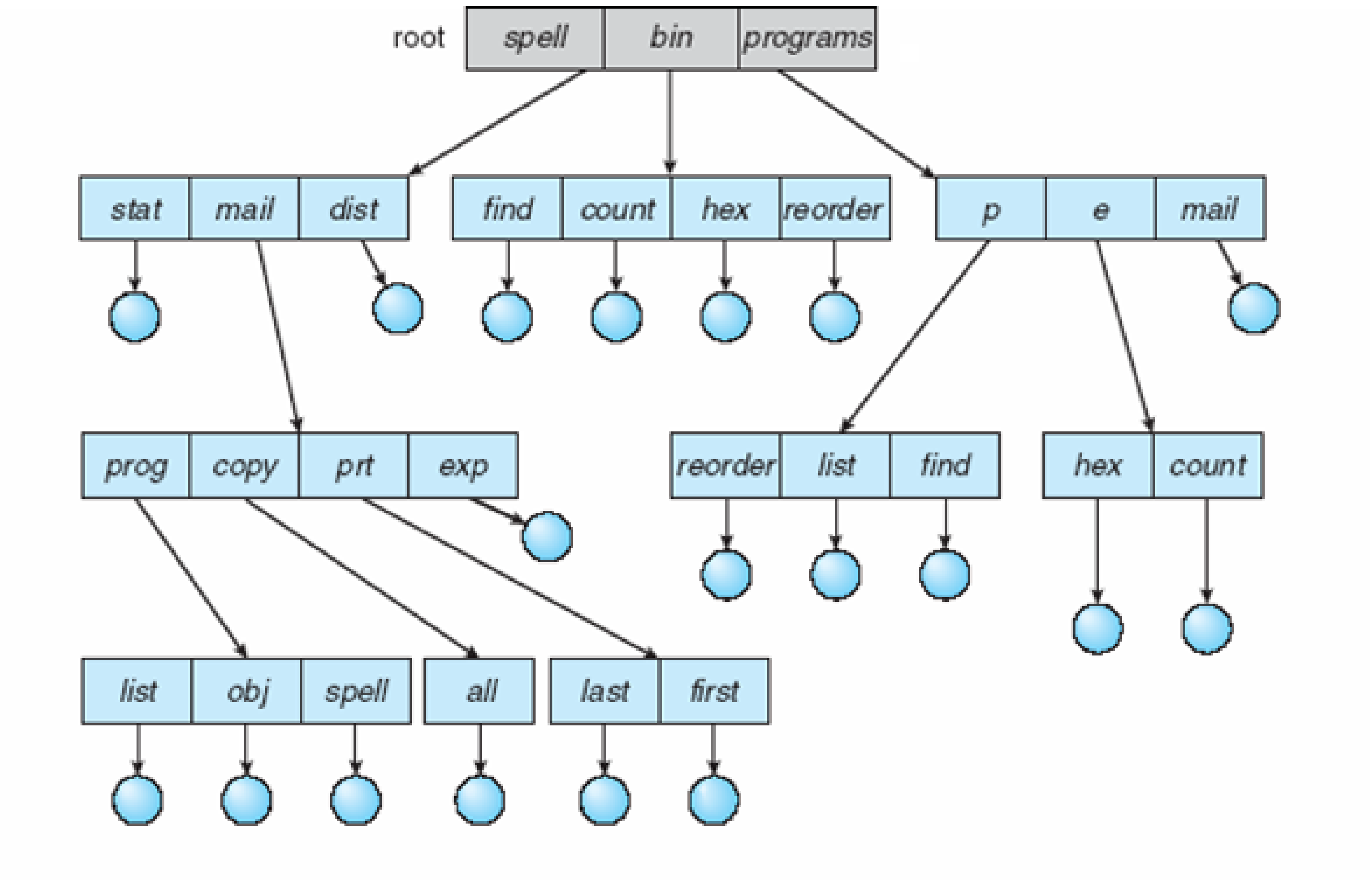
- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

Tree-Structured Directories

- Efficient searching
- Grouping Capability
- **Absolute** or **relative** path names may be used
- Current directory (working directory)

```
>cd /spell/mail/prog
>ls
```

(lists files in the current directory)



Tree-Structured Directories – cont.

- Delete a file

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

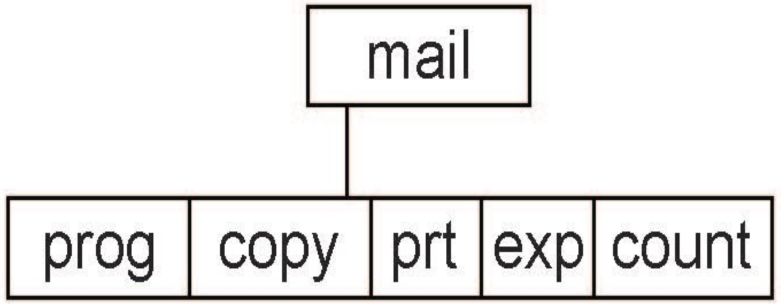
- Creating a new subdirectory is done in current directory or use absolute path

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

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

```
mkdir count
```

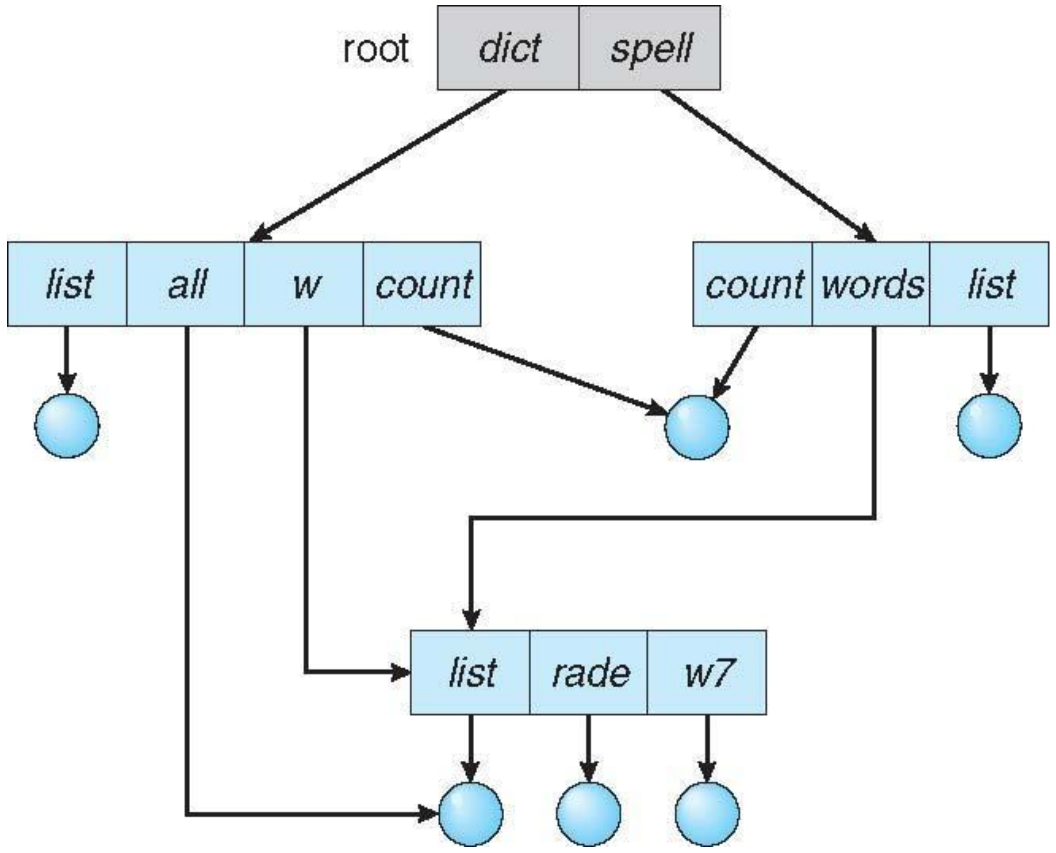
creates a directory in /mail/count



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

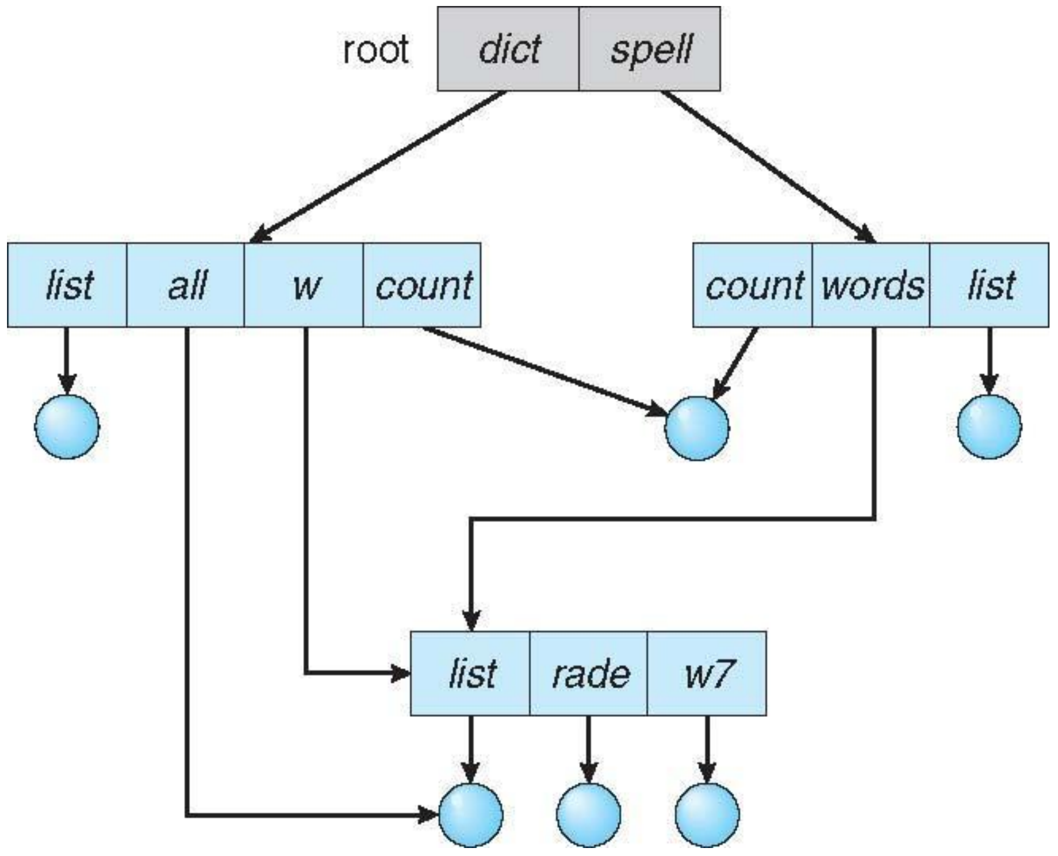
Acyclic-Graph Directories

- A file may be shared amongst two different directories has two absolute names (aliases). Same can be true for a directory.
- More flexibility but more complex
- Multiple ways to implement this feature: When creating a new file, a file descriptor object is created (aka **inode** in Unix) and also a pointer to that descriptor, known as a **hard link** is created sharing of a file amongst multiple directories may be implemented by:
  - Creating other **hard links** inside the other directories.
  - Creating **symbolic links** inside other directories. A symbolic link is another file (but with a special flag) that merely contains the absolute name of the file it is pointing to.
  - Fully duplicating the file content inside the other directories – reads may be performed from **any** of the locations, but writes must be done to **all** locations, to maintain consistency.



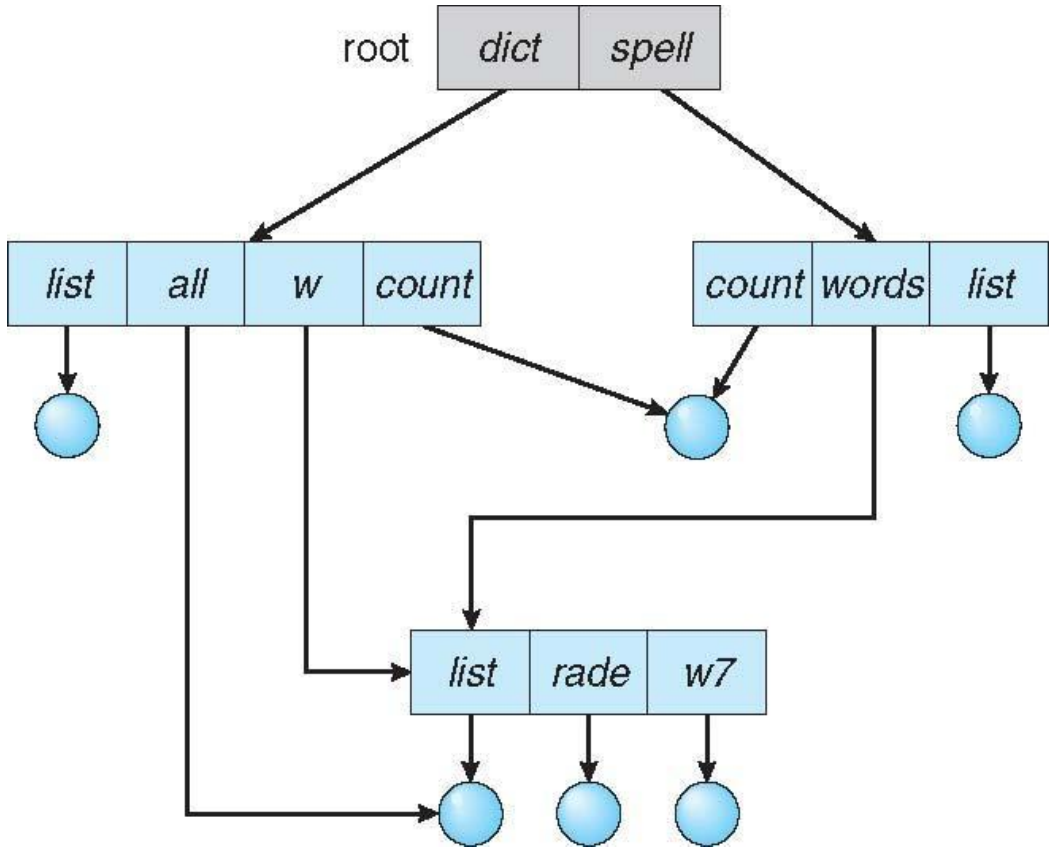
Acyclic-Graph Directories

- Challenges - If the filesystem is trying to **copy a directory**, it must be careful to **copy the file only once** (for the cases of hard or symbolic links).
- Another challenge is **how to delete a file** – one approach is to delete the file when the user deletes one of its hard links:
  - **For remaining symbolic links** no issue, the link is pointing to an invalid filename handled just like any other attempt to access an illegal filename. In Unix and Windows, symbolic links are left and not deleted when the file they point to is deleted.
  - **For remaining hard links** it is a dangling pointer to disk blocks that are now deleted. Those blocks may have been reassigned to another file after the deletion a bigger issue.



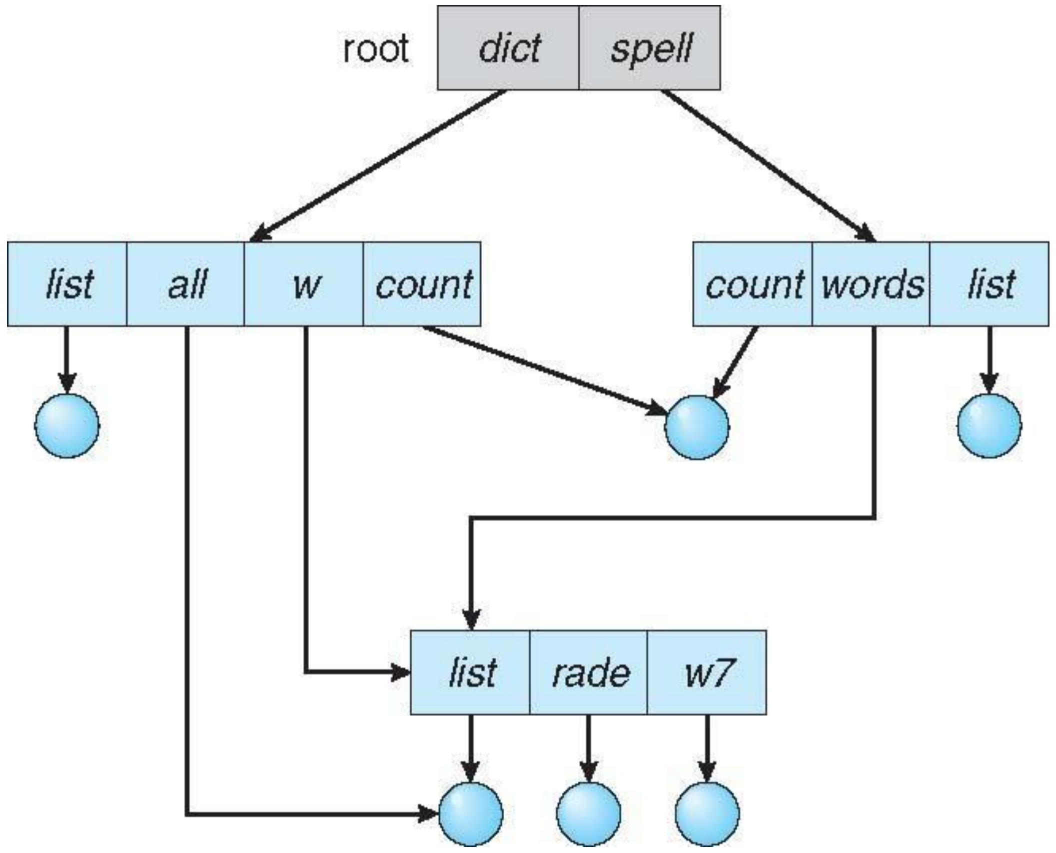
Acyclic-Graph Directories

- Another approach is to only delete the file when all its links are deleted:
  - Unix inodes keep a **reference count** of all the hard links and only deletes the inode when the count reaches zero.
  - The file descriptor may keep track of the links using a **backpointer** to each one of them Variable-sized file descriptors may sometimes be a problem
  - Backpointers using a daisy chain organization (like a doubly linked list)



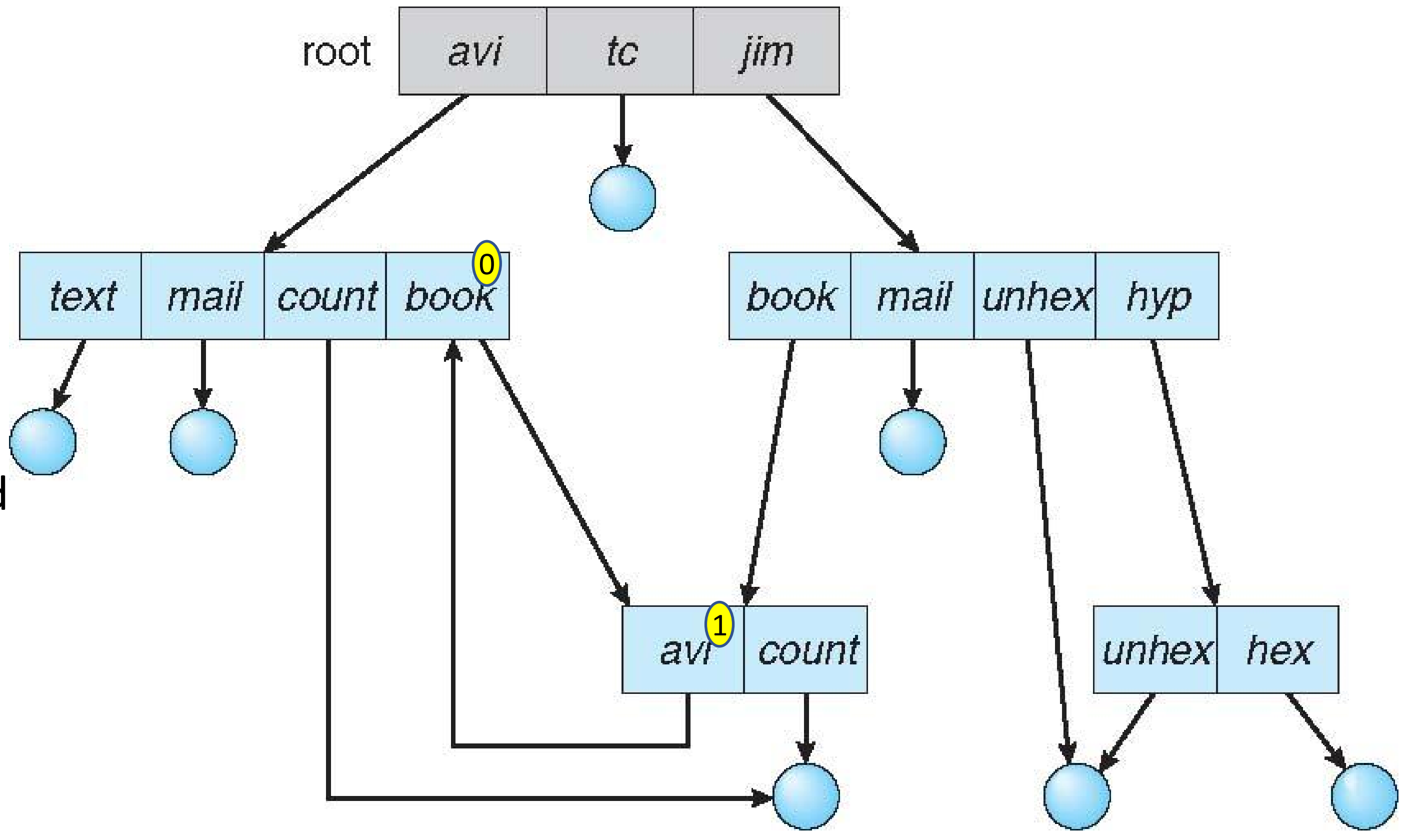
Acyclic-Graph Directories

- 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
- In Unix, hard links cannot point to a directory (avoids cycles), and it also cannot span file systems or partitions. Conversely, symbolic links can do both, including a network file.



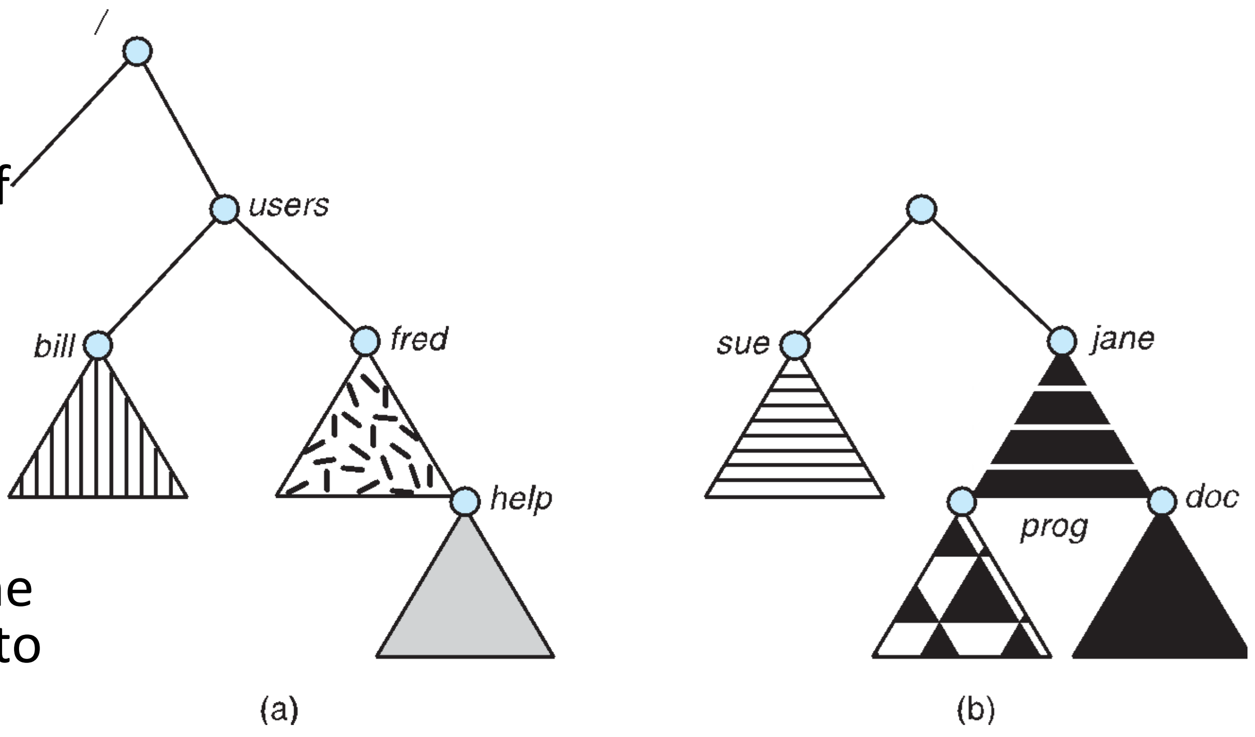
General Graph Directory

- **Garbage collection**
  - It is possible for a node to be unreachable without having a reference count of 0, e.g.:
    - Delete /avi/book.
    - Node 0 (yellow) remains since /jim/book/avi is still referencing it.
    - Then delete /jim/book/avi.
    - Now, we are left with nodes 0 and 1 (depicted by yellow circles) with a ref count of 1 but both unreachable.
  - Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space.
    - Very time consuming renders general graph directories unpopular



11.4 File System Mounting

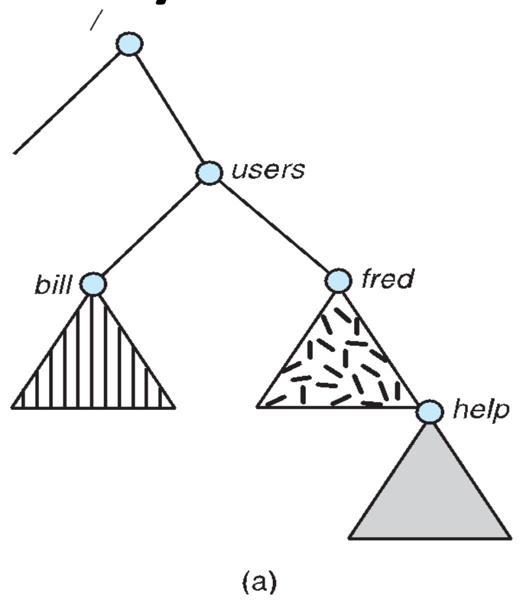
- A file system must be **mounted** before it can be accessed
  - If the directory structure is built out of multiple volumes, they must be mounted to make them available within the file-system name space.
- OS is given:
  - A device (by name)
  - A **mount point -** the location within the file structure where the file system is to be attached (usually an empty directory)
  - A filesystem type (although some systems can detect that on their own)
- OS asks the device driver to read the device directory and verifies it has the expected format.



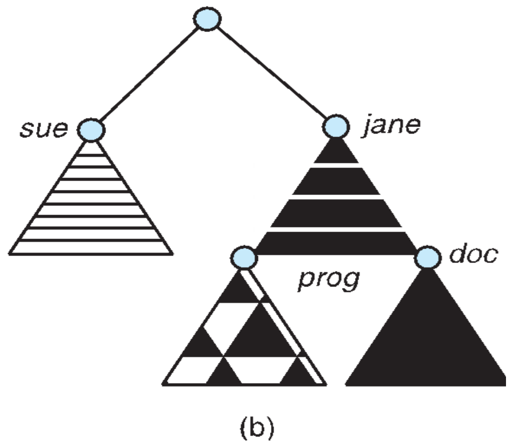
Existing file system

Unmounted volume (at /dev/dsk)

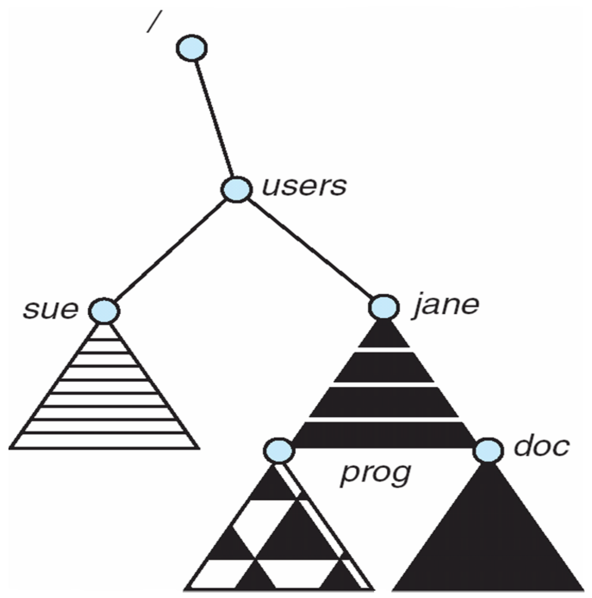
File System Mounting – Example



Existing file system



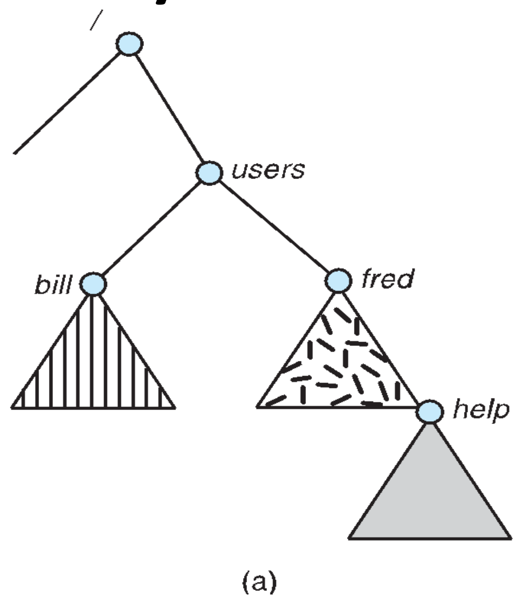
Unmounted volume (at /dev/dsk)



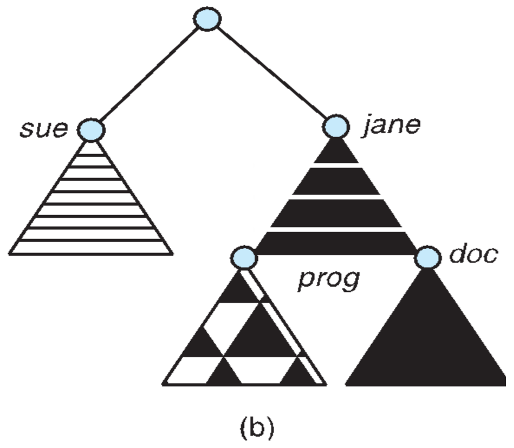
After the mount

- Systems may allow mounting over a non-empty directory:
  - If they do, they may obscure old directory, and bring back after unmounting.
- A volume may be allowed to mount multiple times (to multiple directories).

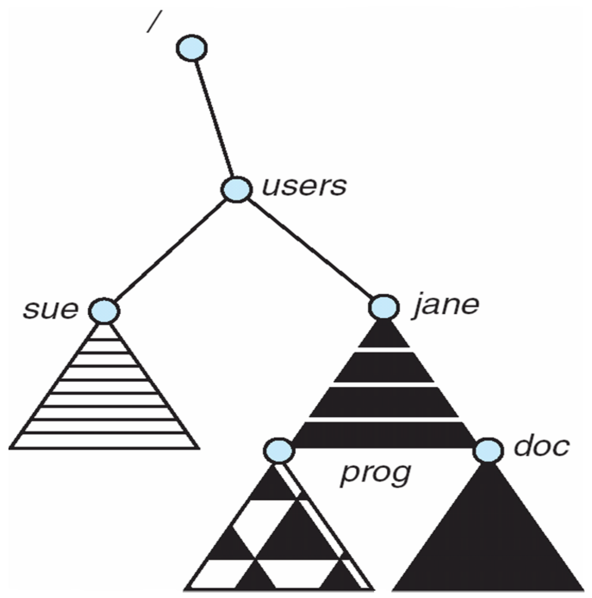
File System Mounting – Example



Existing file system



Unmounted volume (at /dev/dsk)

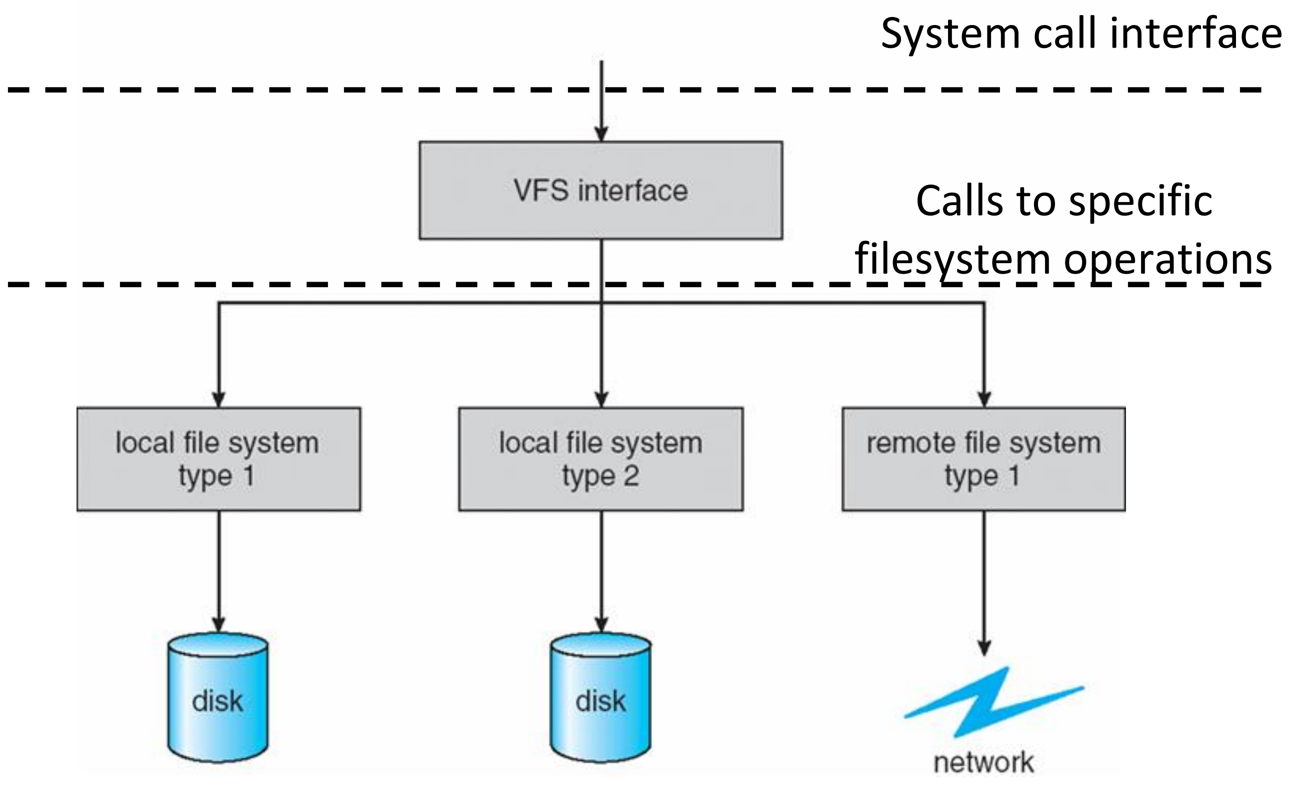


After the mount

- Automatic vs manual mounting:
  - Mac OS X automatically mounts a new disk device that it encounters into /volumes/filesystem_name)
  - In other Unix systems, the mount commands are explicit. A system configuration file contains a list of devices and mount points for automatic mounting at boot time, but other mounts may be executed manually.

Virtual File Systems

- **Virtual File Systems** (**VFS**) on Unix provide an object-oriented way of implementing file systems
  - Allows very dissimilar file-system types to be implemented within the same structure, including network file systems, such as NFS.
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
    - Implementation can be one of many file systems types, or network file system
    - Then dispatches operation to appropriate file system implementation routines



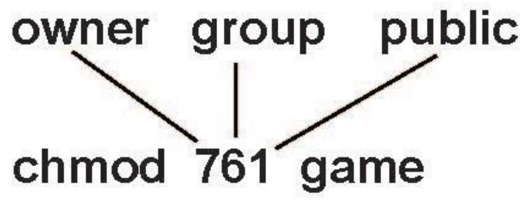
11.6 Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom
- 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

a) **owner access**	7	RWX ⇒	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 mygroup, and add some users to the group.
  - Groups may be added to the system

```
> addgroup mygroup
```

  - Users may be added to groups

```
> usermod –G mygroup omansour
```

Access Lists and Groups

- For a particular file, we can change its owner or change its group and then change access rights (for owner/group/public)
  - Owner of a file / directory, e.g.

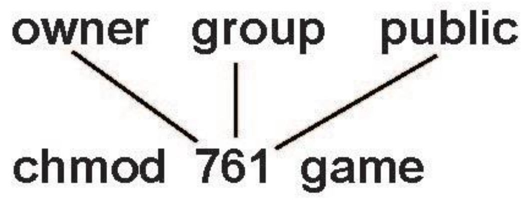
```
> chown omansour:mygroup game
```

  - Attach a group to a file/directory, e.g.

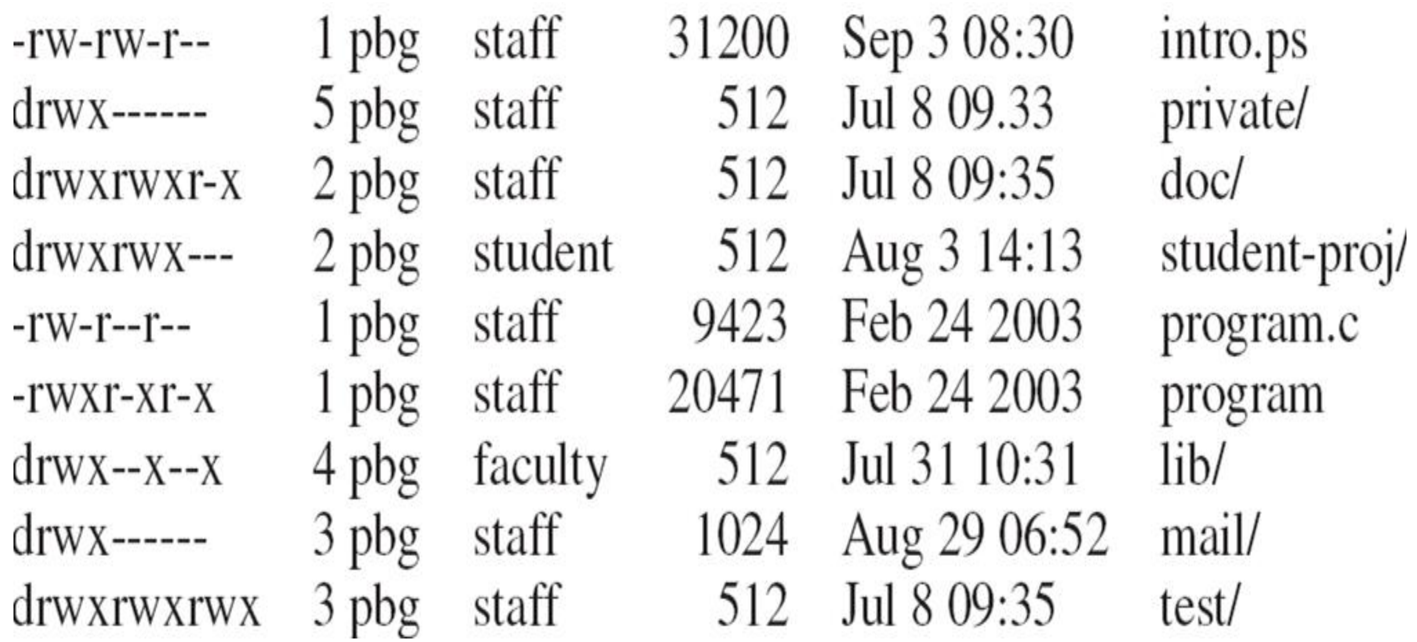
```
> chgrp group1 game
```

  - Change access rights, e.g.

```
> chmod 761 game
```



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/
```