CS343: Operating System

File System and Device Driver

Lect39 : 14th Nov 2023

Dr. A. Sahu

Dept of Comp. Sc. & Engg.

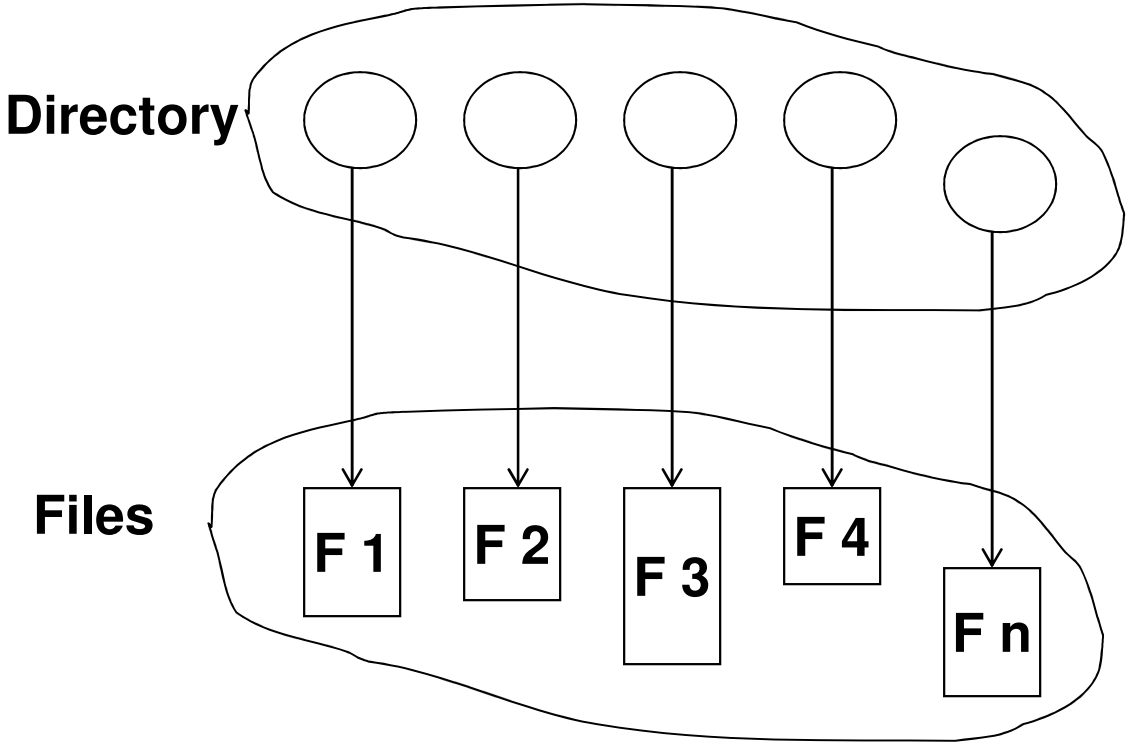
Indian Institute of Technology Guwahati

Outline

- FS Basic
- FS Implementation
- I/O subsystem
- Device Drivers

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

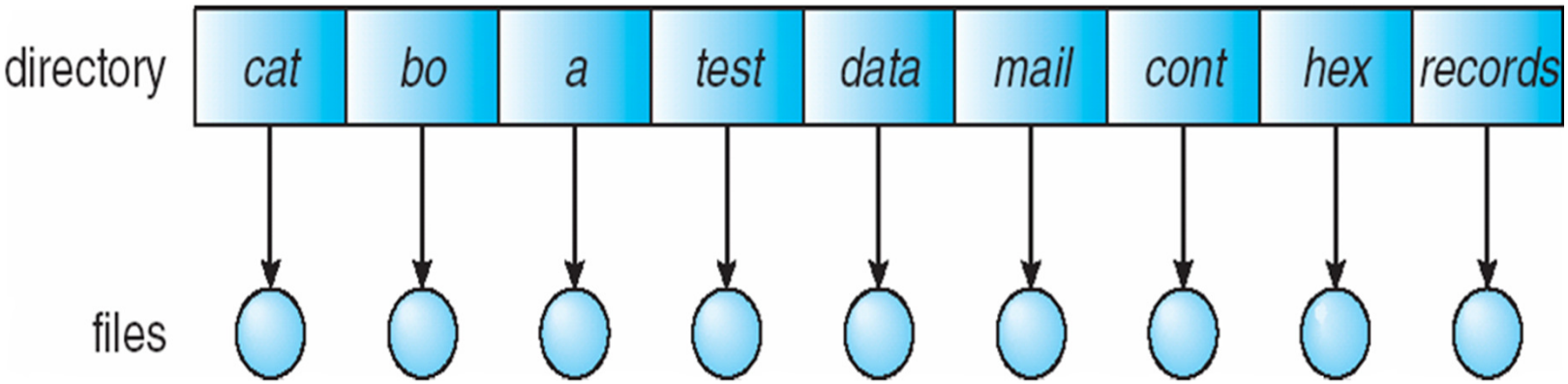
- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

Directory Organization

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

Single-Level Directory

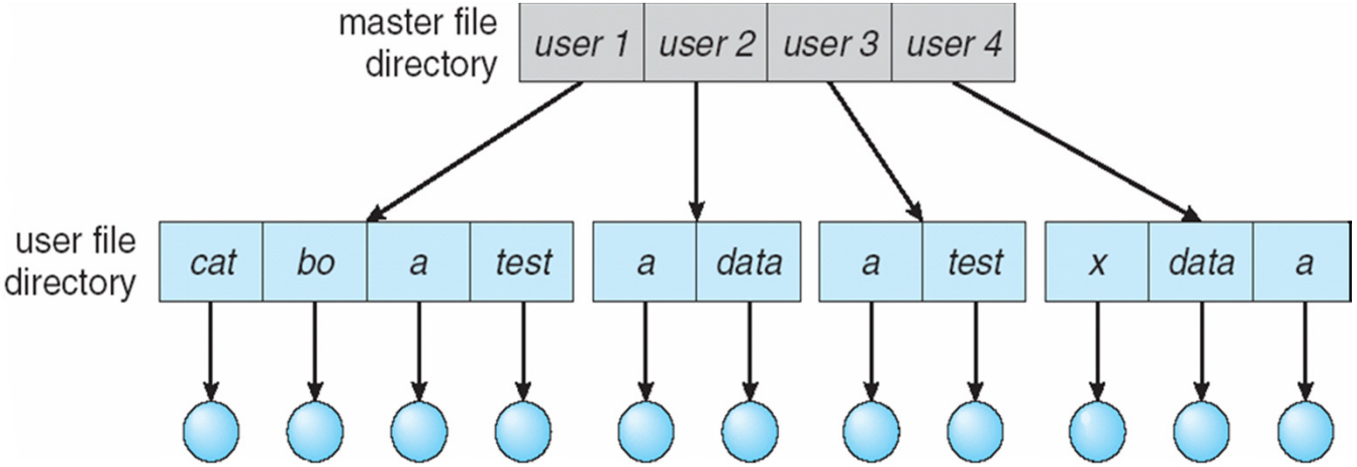
- A single directory for all users



- Naming problem
- Grouping problem

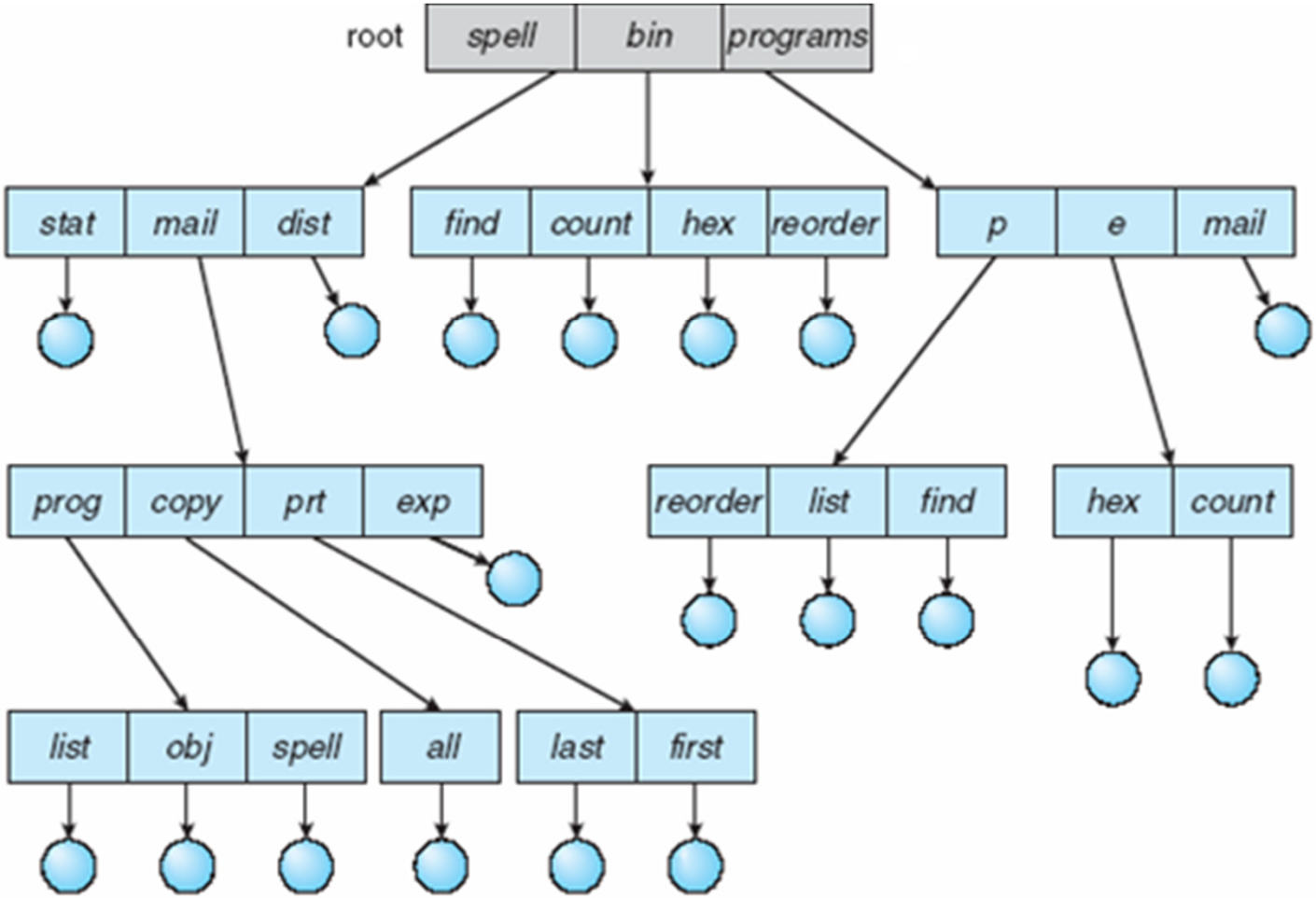
Two-Level Directory

- Separate directory for each user



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

Tree-Structured Directories



Tree-Structured Directories (Cont.)

- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - `cd /spell/mail/prog`
  - `type list`

Tree-Structured Directories (Cont)

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

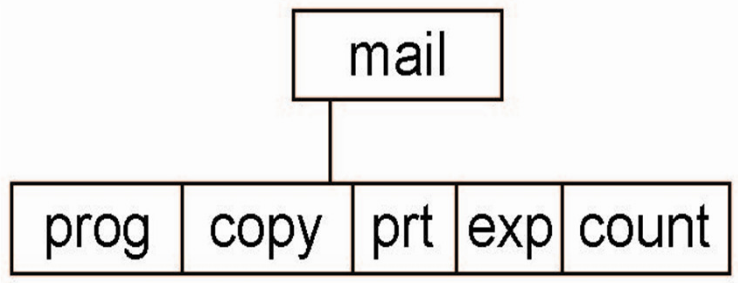
```
rm <file-name>
```

- Creating a new subdirectory is done in current directory

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

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

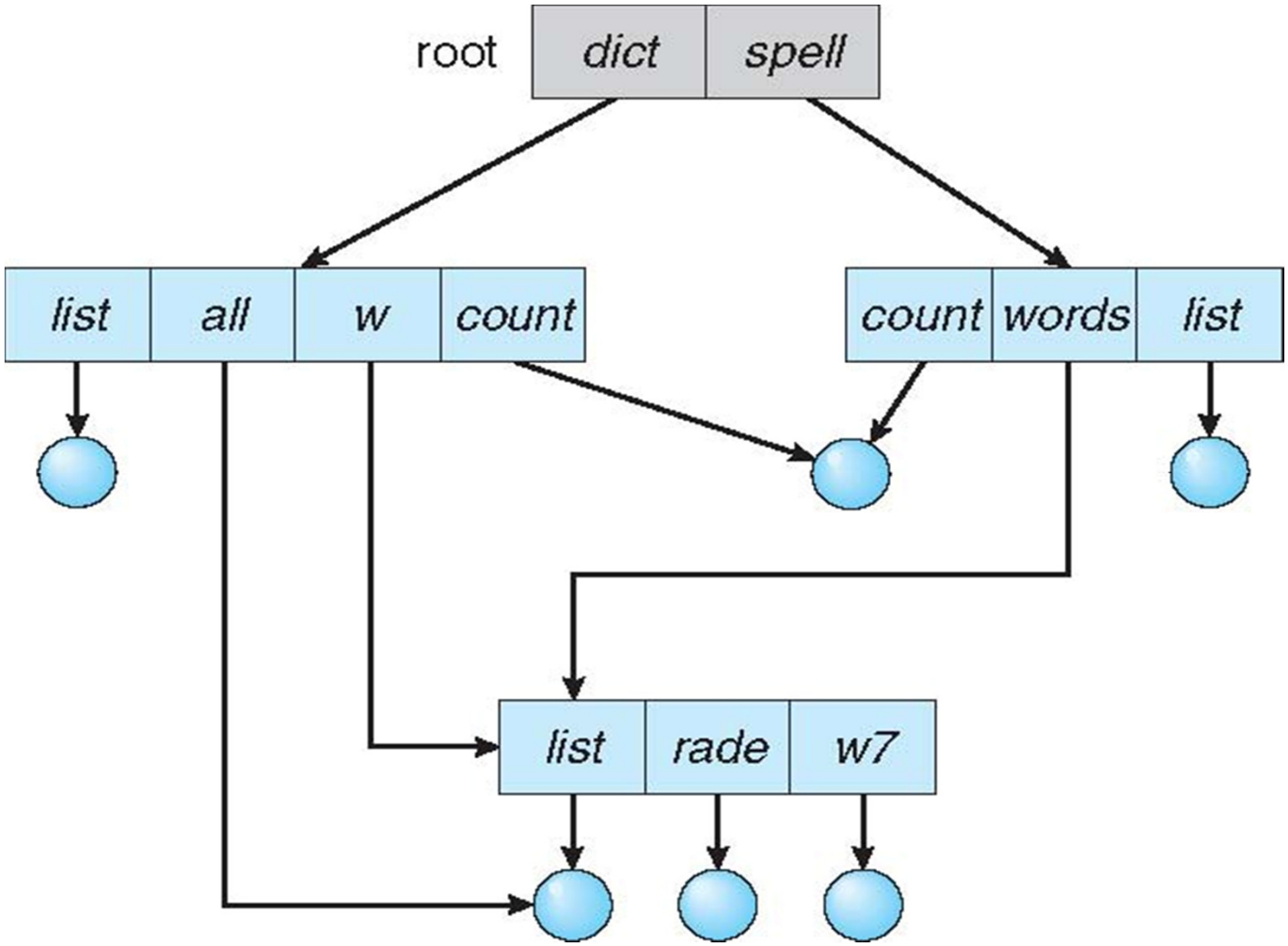
```
mkdir count
```



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

Acyclic-Graph Directories

- Have shared subdirectories and files

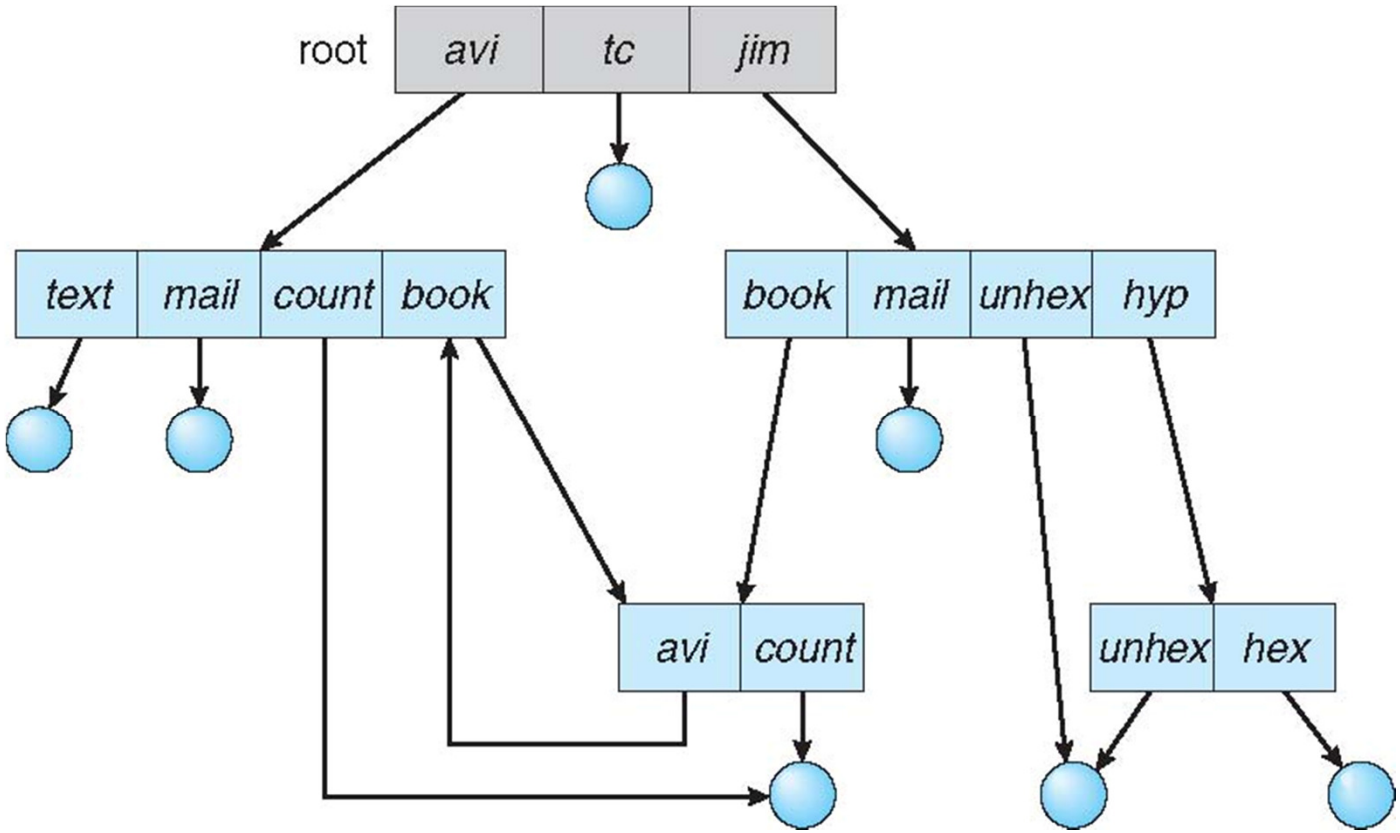


Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)
- If ***dict*** deletes ***list*** $\Rightarrow$ dangling pointer

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

General Graph Directory

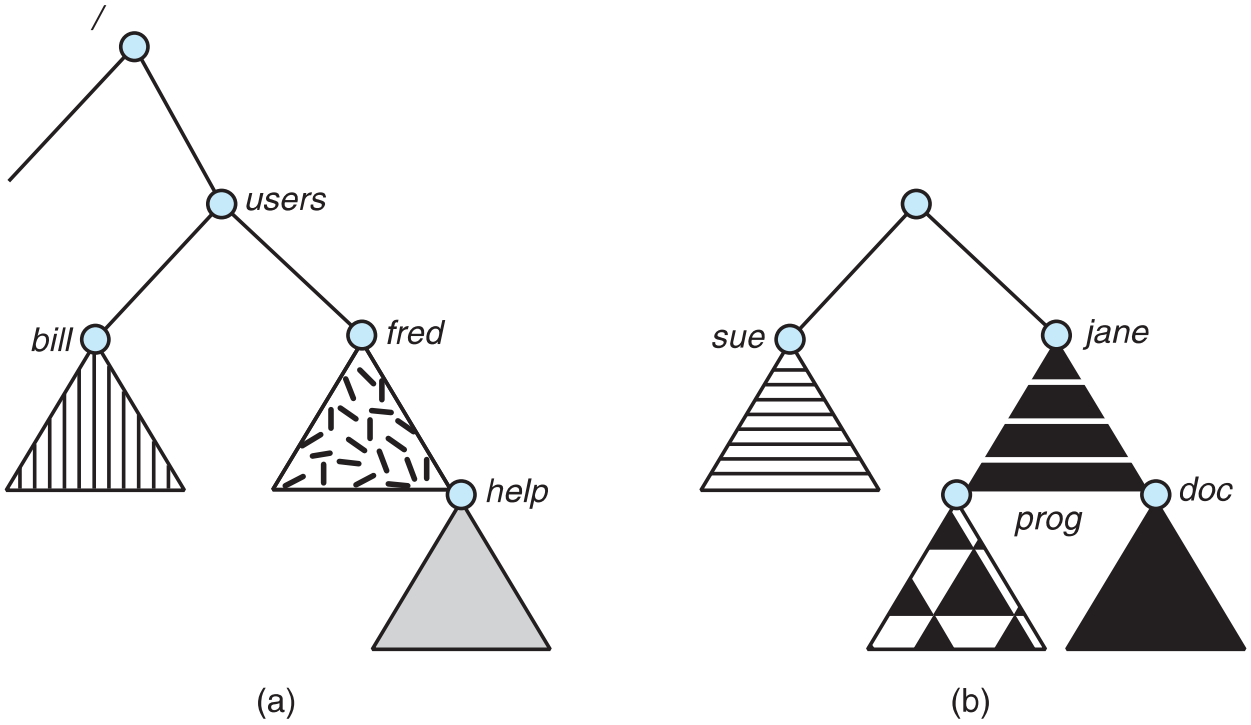


General Graph Directory (Cont.)

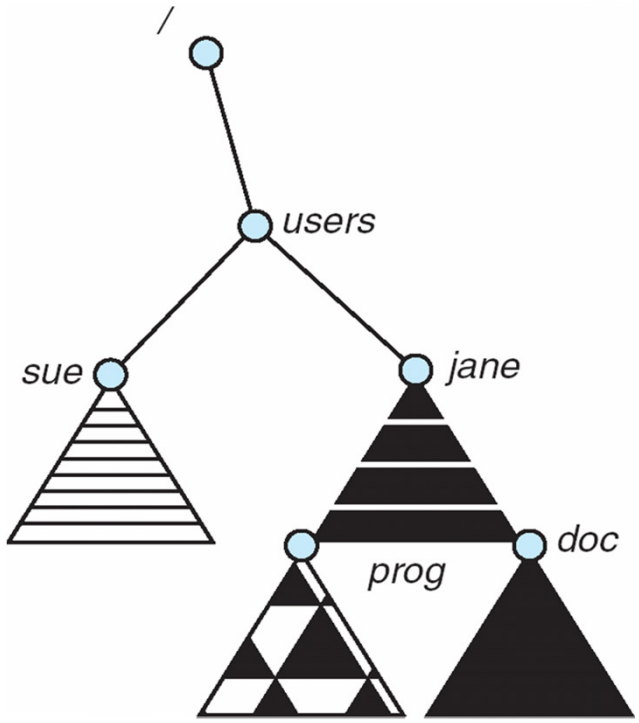
- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - **Garbage collection**
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

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



Basic of Unix File System (UFS)

- Same as BSD Fast FS (FFS)
- A UFS volume is composed of the following parts:
  - A few blocks at the beginning of the partition reserved for boot blocks (which must be initialized separately from the FS)
  - A superblock, containing a magic number identifying this as a UFS,
  - And some other vital numbers describing this filesystem's geometry and statistics and behavioral tuning parameters

Basic of Unix File System (UFS)

- A collection of cylinder groups. Each cylinder group has the following components:
  - A backup copy of the superblock
  - A cylinder group header, with statistics, free lists, etc., about this cylinder group, similar to those in the superblock
  - A number of inodes, each containing file attributes
  - A number of data blocks

Index Node (Or Inode)

- An inode, is a data structure used to represent a FS object
- Which can be one of various things including a file or a directory.
- Each inode stores the attributes and disk block location(s) of the FS object's data.
- FS object attributes may include
  - Manipulation metadata (e.g. change,[2] access, modify time),
  - As well as owner and permission data (e.g. group-id, user-id, permissions)

I-node

- A FS relies on data structures about the files, beside the file content.
  - The former is called metadata—data that describes data.
- Each file is associated with an inode, which is identified by an integer number
  - Often referred to as an i-number or inode number.
- Inodes store information about files and directories (folders),
  - Such as file ownership, access mode (read, write, execute permissions), and file type.

I-node

- On many types of file system implementations,
  - The maximum number of inodes is fixed at file system creation,
  - Limiting the maximum number of files the file system can hold.
- A typical allocation heuristic for inodes in a FS
  - One percent of total size.
- The inode number indexes a table of inodes in a known location on the device
- From the inode number, the FS driver portion of the kernel can access the contents of the inode
  - Including the location of the file allowing access to the file.

Inode Vs file descriptor

- inode numbers are like pointers within the file system, and are stored within the file system itself.
  - A directory entry is the combination of a name and an inode, and you can see this using ls –I .
- File descriptor numbers, on the other hand,
  - Are not stored anywhere within the file system, and are dynamically generated by the kernel when you call open()
  - They are pointers into the kernel's file descriptor table for a particular process.
- An inode number always refers to something on a device somewhere.
- A file descriptor may also refer to an anonymous pipe, a socket, or some other kind of resource

Inode Vs file descriptor

- An inode number unambiguously identifies a file or directory on a given device, but two files on different mounts may have the same inode.
- A file descriptor does not unambiguously identify anything by itself;
  - in combination with a process ID it unambiguously identifies some resource on the system, even if you don't know which device it's on.
- Every file or directory on a given device has a unique inode number.
  - If two files on the same device have the same inode number, then they are really the same file with two different names.
- On the other hand, a file or directory may be opened several times by the same process or by different processes, and thus have multiple different file descriptors.
- Additionally, files or directories that are not currently open by any process do not have any file descriptors referring to them.

Inode Vs file descriptor

- FD does not contain any metadata associated with the file itself, such as timestamps or Unix permission bits.
- An inode contains timestamps and Unix permission bits, but no file mode flags or offset.
- A valid file descriptor is associated with file mode flags and offset.
  - That is, it grants the possessing process the right to read or write the file (or both), based on how the file descriptor was obtained;
  - it also remembers some position within the file.

File-System Structure

- **File structure**
  - Logical storage unit
  - Collection of related information
- **File system** resides on (disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily

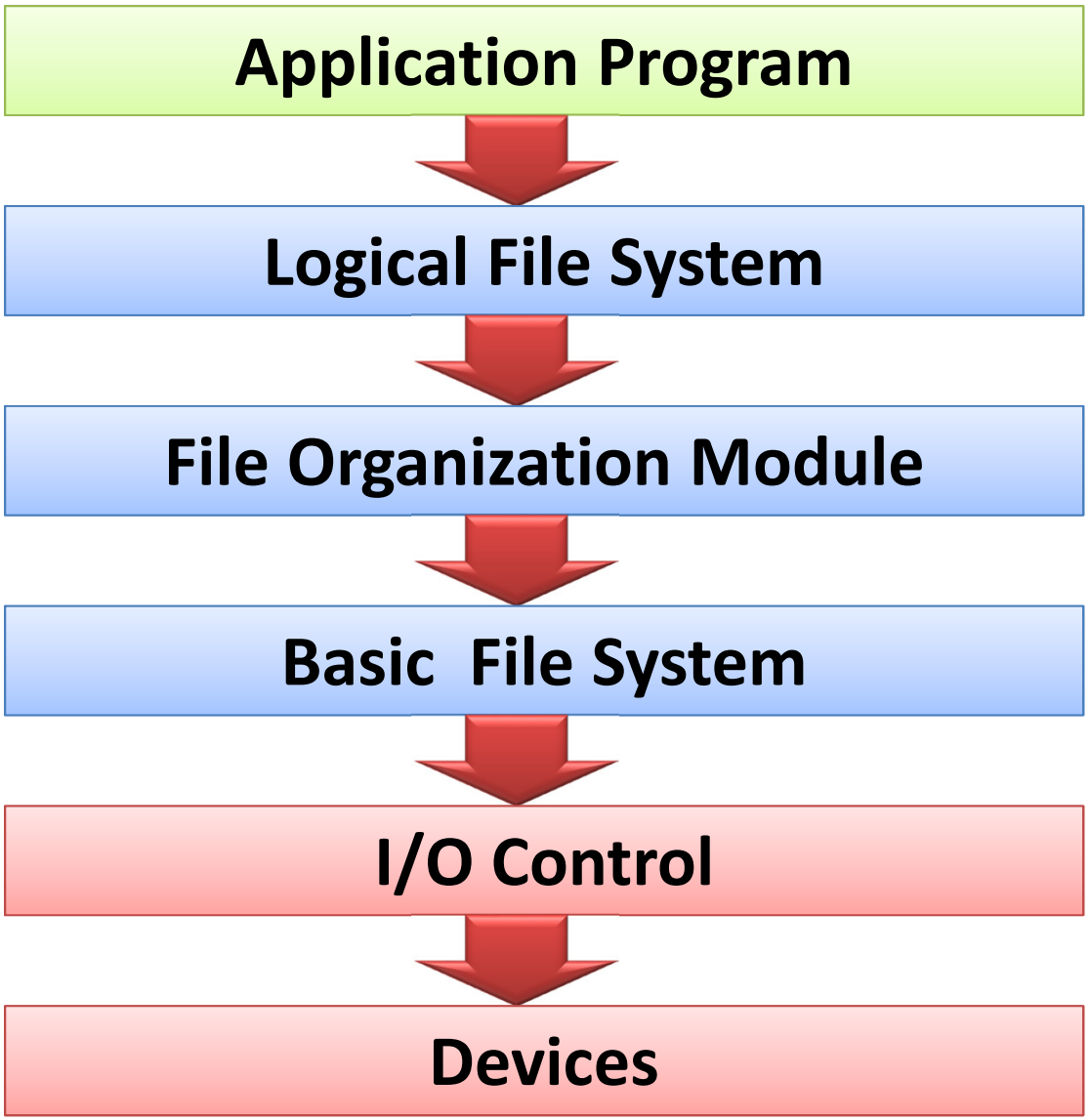
File-System Structure

- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File Control block** – storage structure consisting of information about a file

Inode

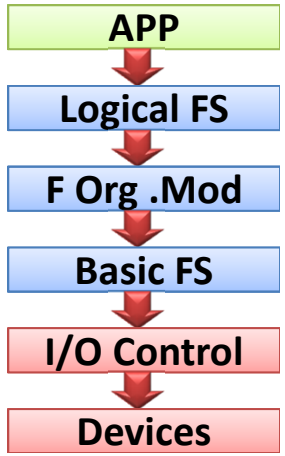
- **Device driver** controls the physical device
- File system organized into layers

Layered File System



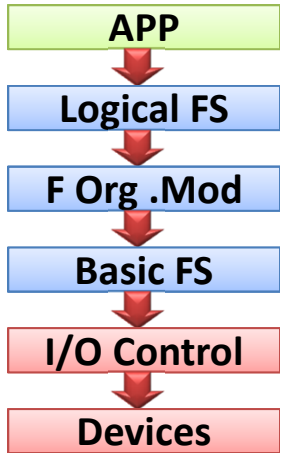
File System Layers

- **Device drivers**
  - Manage I/O devices at the I/O control layer
  - Given commands like
  - "read drive1, cylinder 72, track 2, sector 10, into memory location 1060"
  - Outputs low-level hardware specific commands to hardware controller



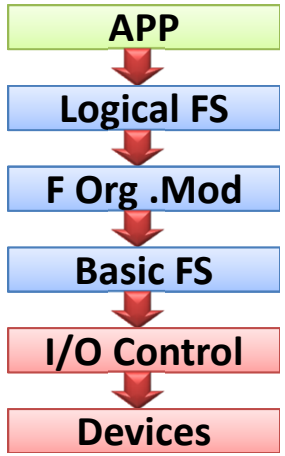
File System Layers

- **Basic file system**
  - Given command like “retrieve block 123” translates to device driver
  - Also manages memory buffers and caches Allocation, freeing, replacement
    - Buffers hold data in transit
    - Caches hold frequently used data



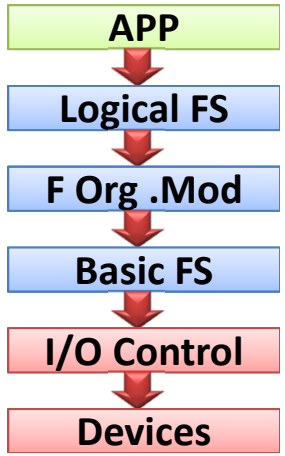
File System Layers

- **File organization module** understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation



File System Layers

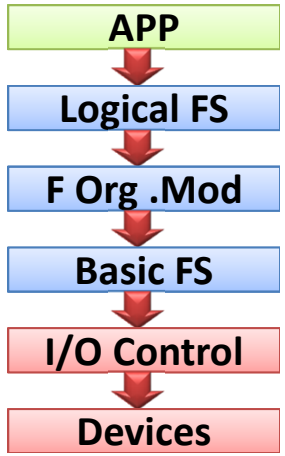
- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Directory management
  - Protection



File System Layers (Cont.)

- **Layering**
  - Useful for reducing complexity & redundancy,
  - But adds overhead : decrease performance
- **Translates File name into**
  - File number, file handle, location
  - By maintaining file control blocks (**inodes**)
  - Logical layers can be implemented

    by any coding method according to OS designer



File System Layers (Cont.)

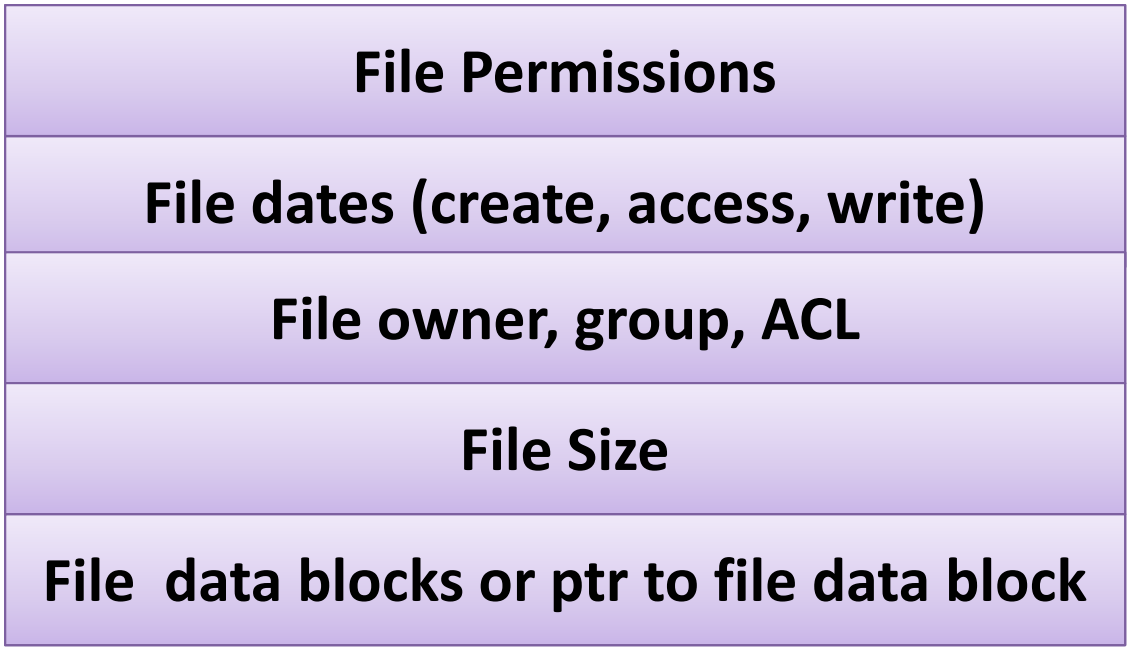
- Many file systems, sometimes many within an operating system, Each with its own format
  - CD-ROM is ISO 9660;
  - Unix has **UFS**, FFS;
  - Windows has FAT, FAT32, NTFS as well as Floppy, CD, DVD Blu-ray,
  - Linux has more than 40 types, with **extended file system** ext2 and ext3 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

File-System Implementation

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block (superblock, master file table)** contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table

File-System Implementation (Cont.)

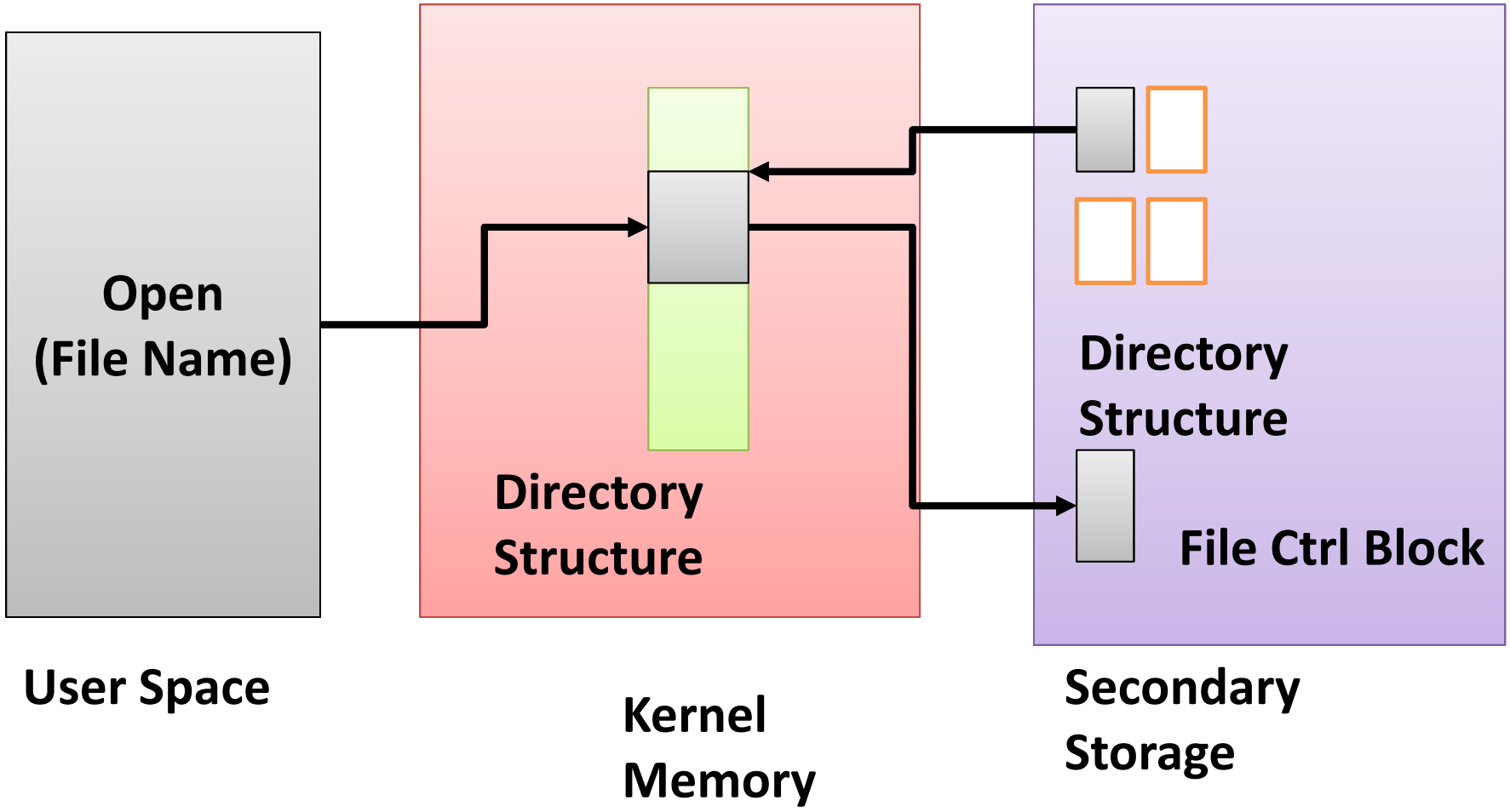
- Per-file **File Control Block (FCB)** contains many details about the file
  - inode number, permissions, size, dates
  - NFTS stores into in master file table using relational DB structures



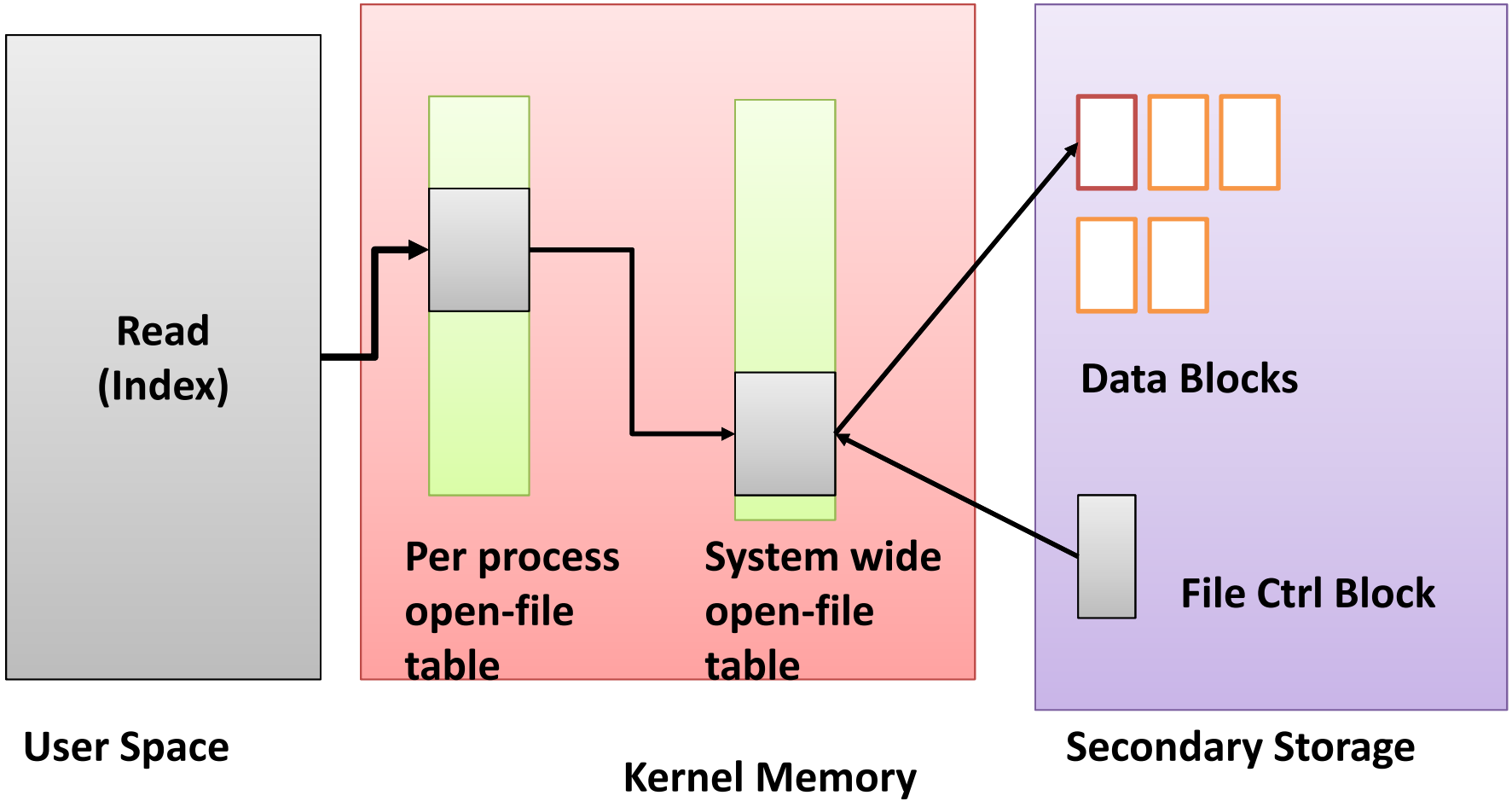
In-Memory File System Structures

- Mount table storing file system mounts, mount points, file system types
- Buffers hold data blocks from secondary storage
- Open returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address

In-Memory FS Structures : File Open()



In-Memory FS Structures : File Open()



Partitions and Mounting

- Partition can be a volume containing a file system (“cooked”) or **raw**
  - Just a sequence of blocks with no file system
- Boot block can point to
  - Boot volume or boot loader set of blocks
  - that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting

Partitions and Mounting

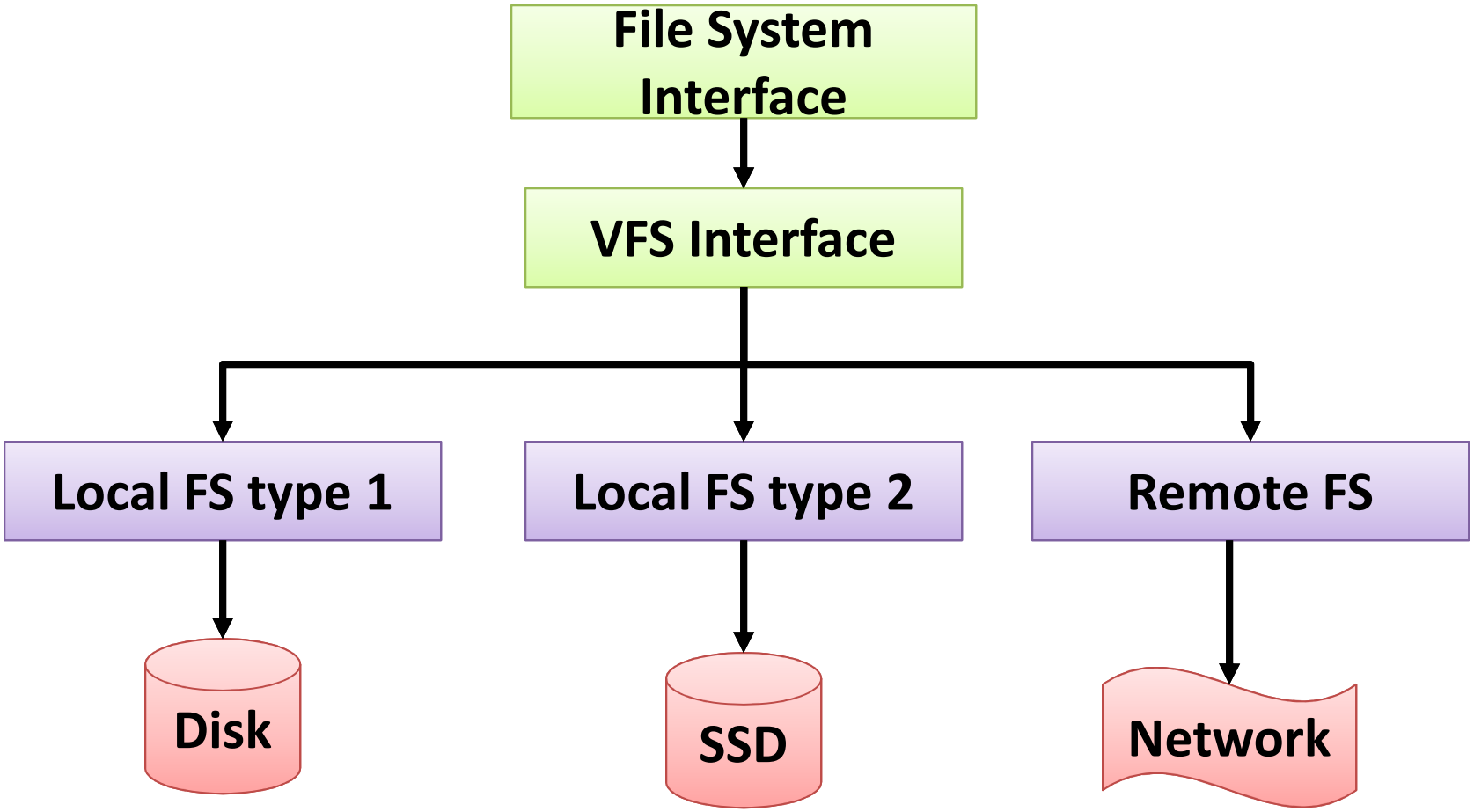
- **Root partition** contains the OS,
  - Other partitions can hold other OSes, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually
- At mount time, FS consistency checked
  - Is all metadata correct?
    - If not, fix it, try again
    - If yes, add to mount table, allow access

Virtual File Systems

- **VFS** on Unix provide an object-oriented way of implementing file systems
- 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
    - Implements **vnodes** which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementation routines

Virtual File Systems (Cont.)

- The API is to the VFS interface, rather than any specific type of file system



Virtual File System Implementation

- For example, Linux has four object types:
  - inode, file, superblock, dentry (directory entry)
- VFS defines set of operations on the objects that must be implemented
  - Every object has a pointer to a function table
    - Function table has addresses of routines to implement that function on that object
    - For example:

`int open(. . .)`—Open a file

`int close(. . .)`—Close an already-open file

`ssize t read(. . .)`—Read from a file

`ssize t write(. . .)`—Write to a file

`int mmap(. . .)`—Memory-map a file

Directory Implementation

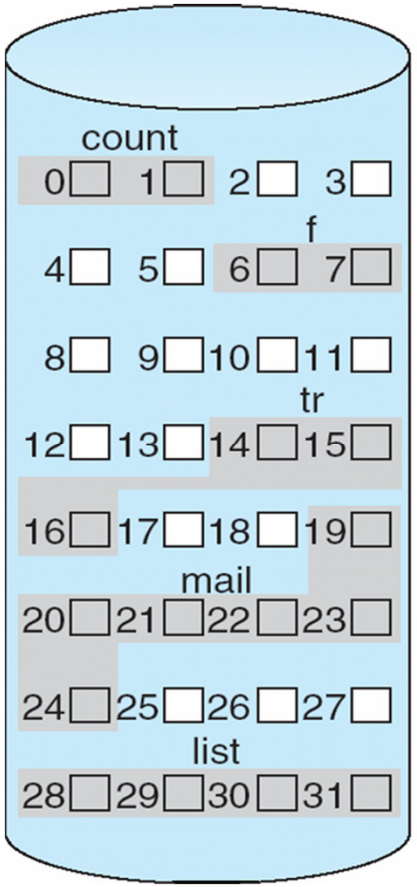
- **Linear list** of file names with pointer to the data blocks
  - Simple to program, Time-consuming to execute
  - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - **Collisions** – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

Allocation Methods - Contiguous

- An allocation method refers to how disk blocks are allocated for files
- **Contiguous allocation** – each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include finding space for file, knowing file size, external fragmentation, need for **compaction off-line** (**downtime**) or **on-line**

Contiguous Allocation

- Mapping from logical to physical



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Block to be accessed = Q + starting address
Displacement into block = R

Extent-Based Systems

- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents

Allocation Methods - Linked

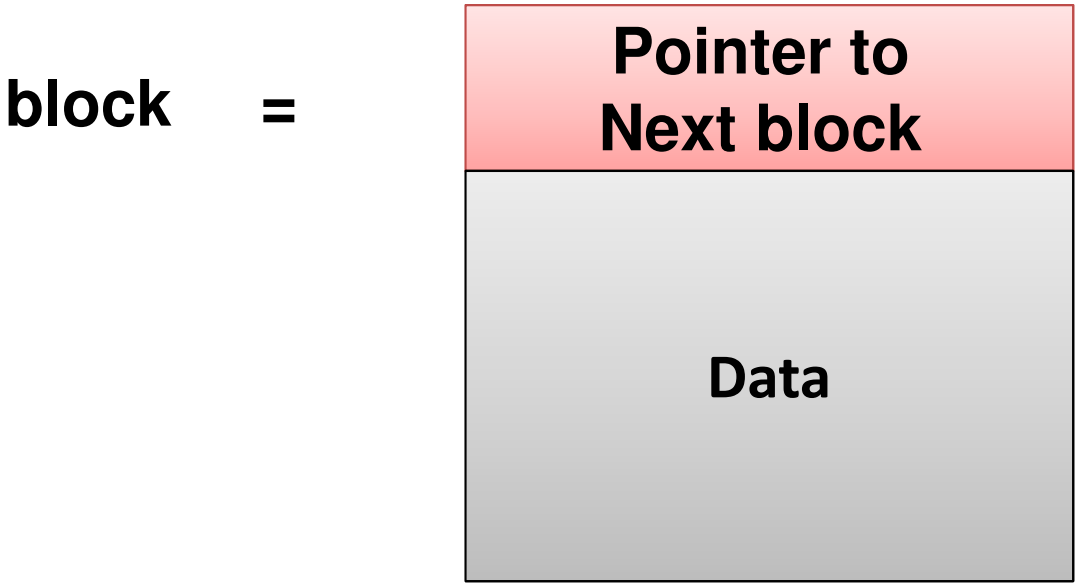
- Each file a linked list of blocks, File ends at nil pointer
- No external fragmentation
- Each block contains pointer to next block
- No compaction, external fragmentation
- Free space management system called when new block needed

Allocation Methods - Linked

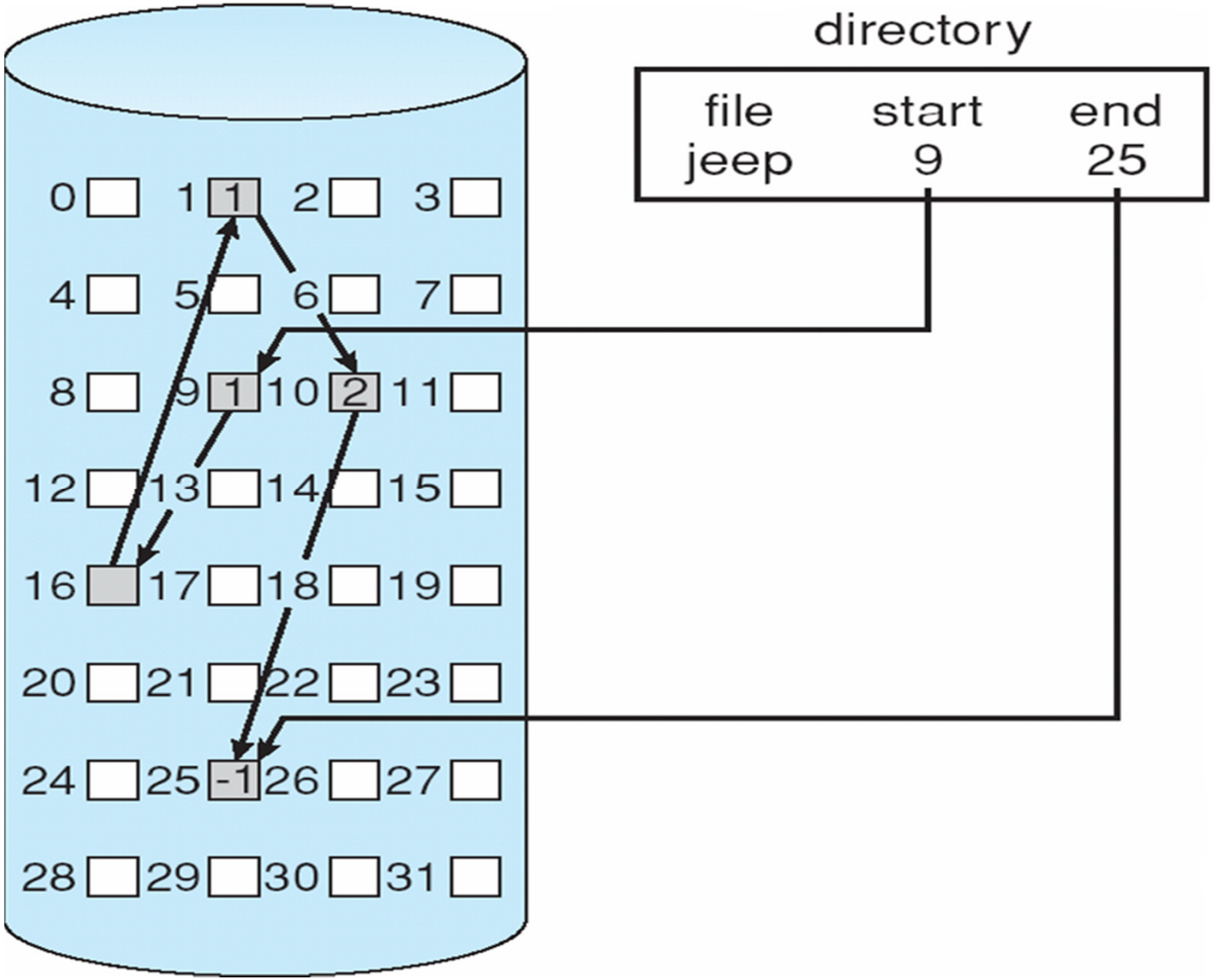
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks
- FAT (File Allocation Table) variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk

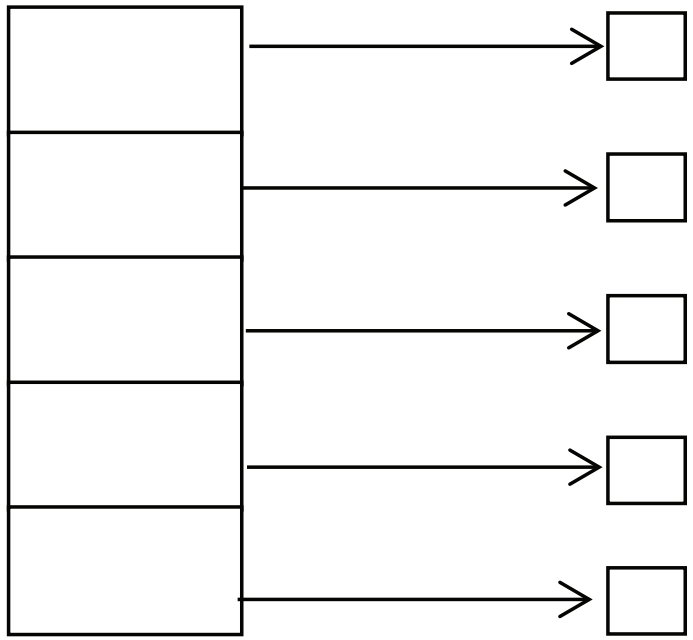


Linked Allocation

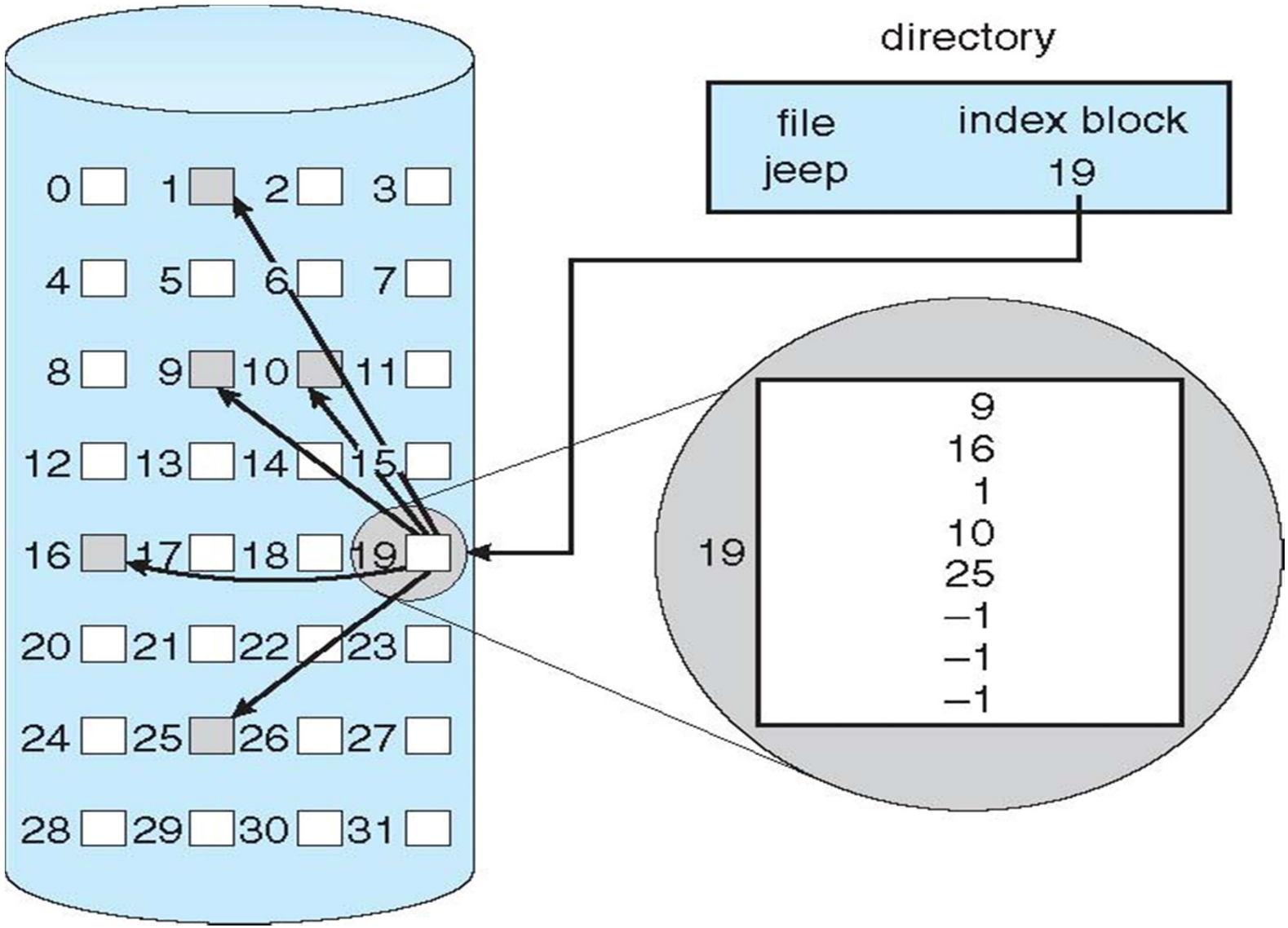


Allocation Methods - Indexed

- **Indexed allocation**
  - Each file has its own **index block**(s) of pointers to its data blocks
- Logical view



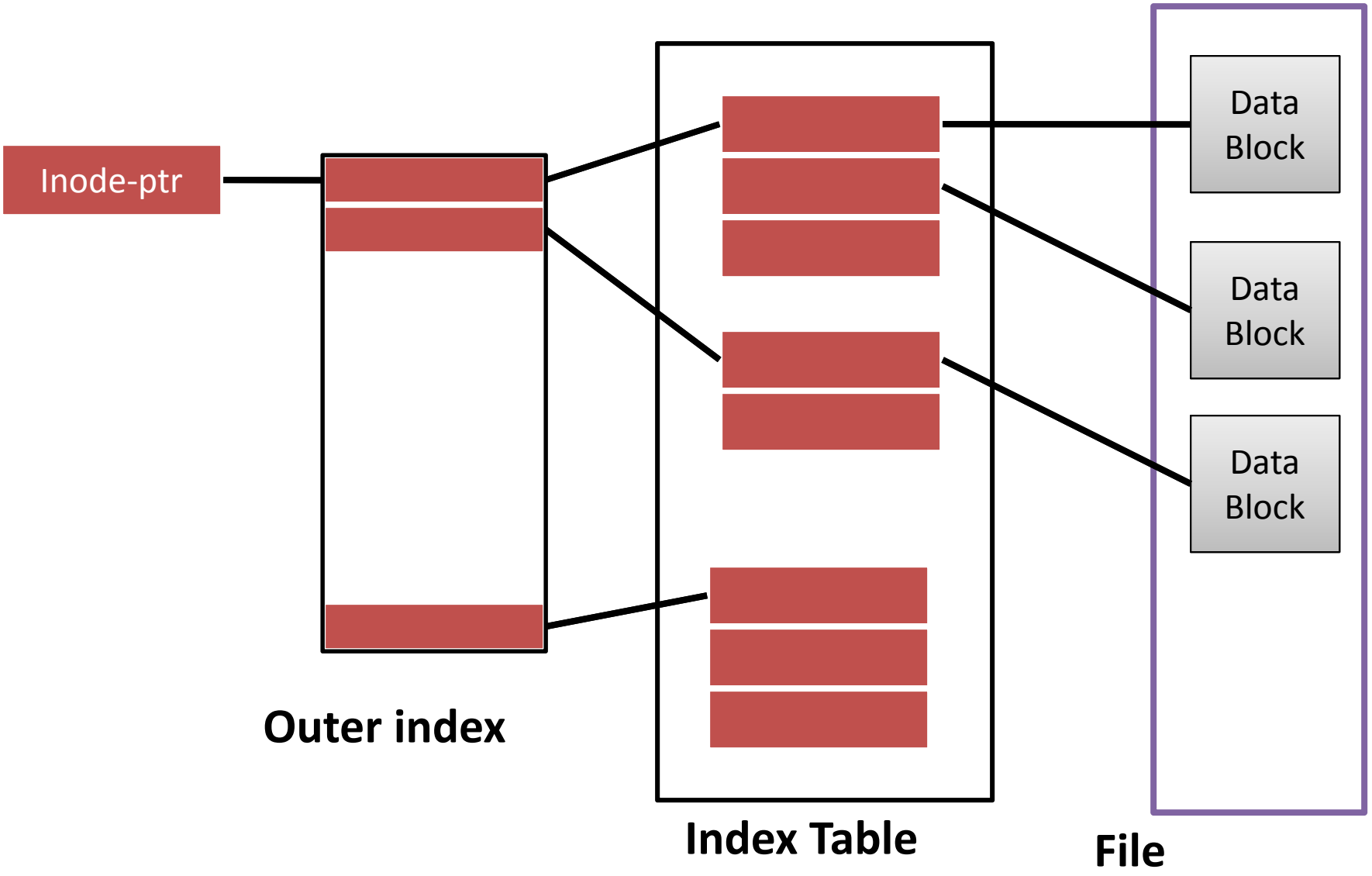
Example of Indexed Allocation



Indexed Allocation (Cont.)

- Need index table, Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes.
  - We need only 1 block for index table

Indexed Allocation – Mapping (Cont.)

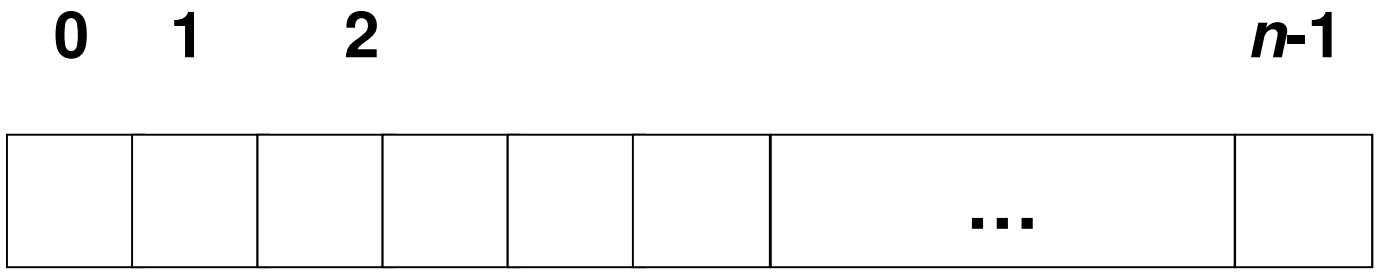


Performance

- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead

Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
  - (Using term "block" for simplicity)
- **Bit vector** or **bit map** (***n*** blocks)



$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Free-Space Management (Cont.)

- Bit map requires extra space
  - Example:

    block size = 4KB = 2^{12} bytes

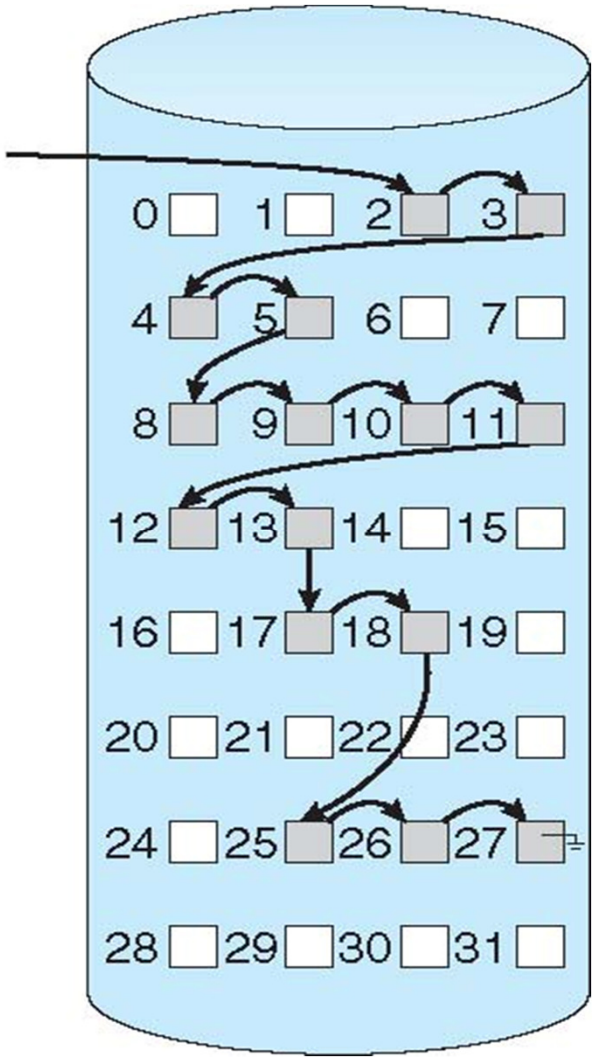
    disk size = 2^{40} bytes (1 terabyte)

    $n = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)

    if clusters of 4 blocks -> 8MB of memory
- Easy to get contiguous files

Linked Free Space List on Disk

- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
  - No need to traverse the entire list (if # free blocks recorded)



Free-Space Management (Cont.)

- Grouping
  - Modify linked list to store address of next *n-1* free blocks in first free block,
  - Plus a pointer to next block that contains free-block-pointers
- Counting
  - Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
    - Keep address of first free block and count of following free blocks
    - Free space list then has entries containing addresses and counts

Free-Space Management (Cont.)

- Space Maps
  - Used in **ZFS**
  - Consider meta-data I/O on very large file systems
    - Full data structures like bit maps couldn't fit in memory -> thousands of I/Os
  - Divides device space into **metaslab** units and manages metaslabs
    - Given volume can contain hundreds of metaslabs

Free-Space Management (Cont.)

- Space Maps
  - Each metaslab has associated space map
    - Uses counting algorithm
  - But records to log file rather than file system
    - Log of all block activity, in time order, in counting format
  - Metaslab activity -> load space map into memory in balanced-tree structure, indexed by offset
    - Replay log into that structure
    - Combine contiguous free blocks into single entry

Efficiency and Performance

- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures

Efficiency and Performance (Cont.)

- Performance
  - Keeping data and metadata close together
  - **Buffer cache** – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - No buffering / caching – writes must hit disk before acknowledgement
    - **Asynchronous** writes more common, buffer-able, faster
  - **Free-behind** and **read-ahead** – techniques to optimize sequential access
  - Reads frequently slower than writes

Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

Recovery

- **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to **back up** data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup

Log Structured File Systems

- **Log structured** (or **journaling**) file systems record each metadata update to the file system as a **transaction**
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated

Log Structured File Systems

- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata