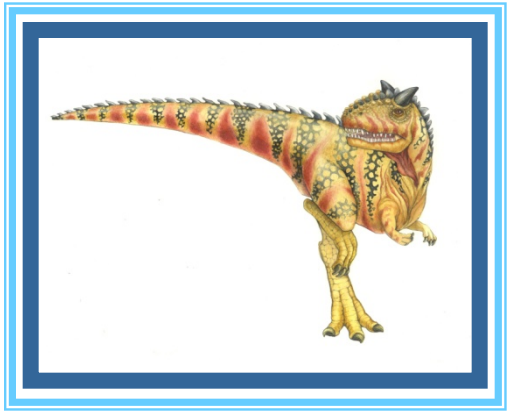
File system



Slides borrowed from Galvin, with many of our modifications

How should the OS efficiently manage a persistent device (your hdd / ssd)?

How should the OS efficiently manage a persistent device (your hdd / ssd)?

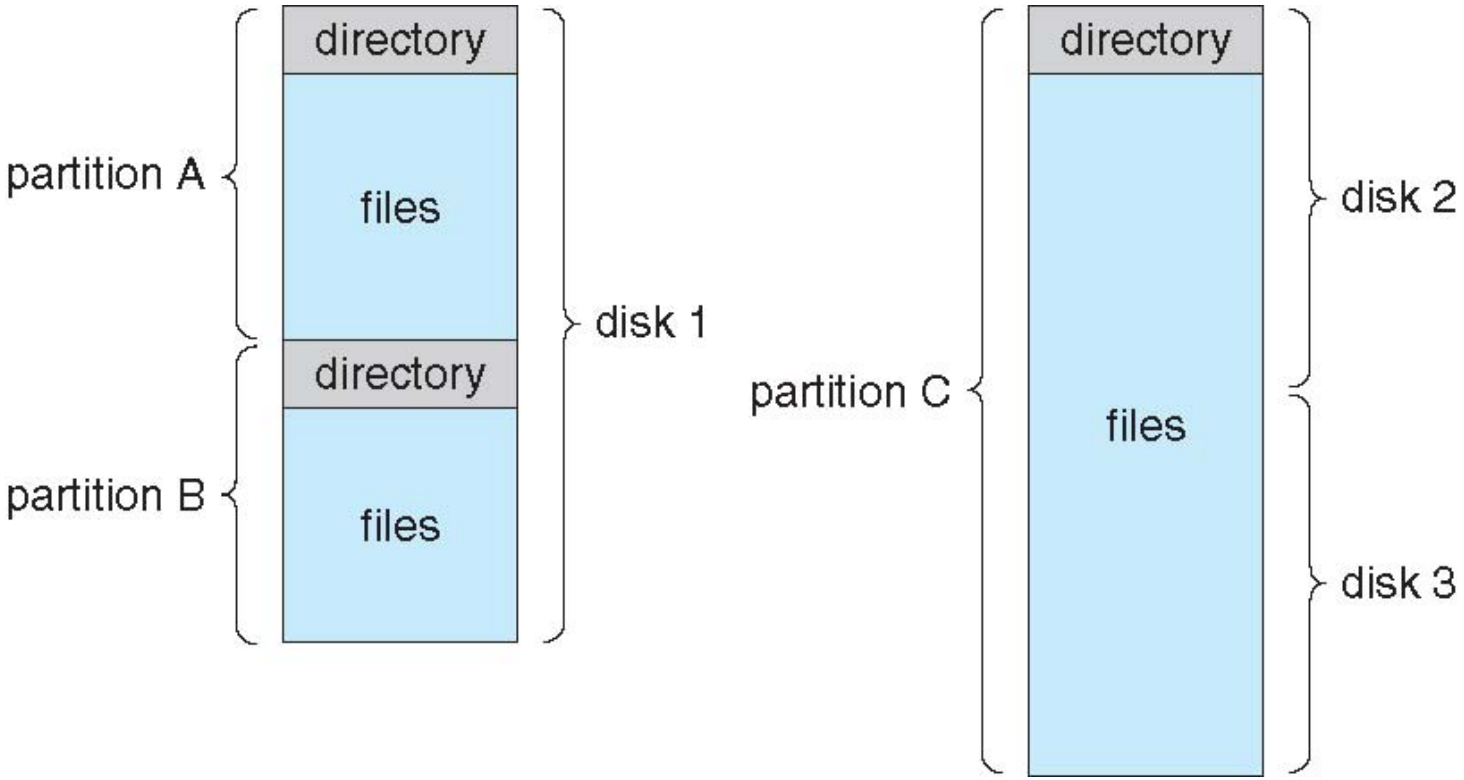
- What are data read/write abstractions?
- What are the APIs?
- How to implement file system related functions?

At the very beginning: You just have a disk

Disk Structure

- What is a disk: OS sees a set of blocks that can be accessed using memory addresses – logical view
- 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**

A Typical File-system Organization



Disk Structure (contd.)

- What is a disk: OS sees a set of blocks that can be accessed using memory addresses – logical view
- 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**

So what is a file system?

File system = Set of Files + Directory

Analogy: Book = Chapters + Index

What is a file?

File: R/W abstraction for storage

- Named collection of related information – smallest allotment of logical secondary usage
- Types:
  - Data: Numeric, character, binary
  - Program files which can be executed
- The meaning of the file content defined by file's creator
  - Many types
    - Consider **text file, source file, object file, executable file**
    - A file often has a **structure**, e.g., sequence of characters

File Structure

- None - sequence of words, bytes
- Simple record structure
  - Lines, Fixed length, Variable length
- Complex Structures
  - Special formatted document
  - Can simulate by inserting appropriate control characters
- Who decides the file structure:
  - Operating system
  - Sometimes programs define their special file structure

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

Attributes stored with files

- **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
- Sometimes extra information, like checksum to know file corruption.

Information about files are kept in the directory structure, which is maintained on the disk (persistent)

File info on Linux

```
  File: notes.txt
  Size: 39              Blocks: 2          IO Block: 1048576 regular file
Device: 5dh/93d Inode: 64392       Links: 1
Access: (0644/-rw-r--r--)  Uid: (21115/ mainack)   Gid: (21000/ns-science)
Access: 2020-04-26 20:31:20.812781321 +0200
Modify: 2020-04-26 20:31:20.814449217 +0200
Change: 2020-04-26 20:31:20.818990853 +0200
 Birth: -
```

- stat notes.txt
- file notes.txt
- ls –l notes.txt

Recap

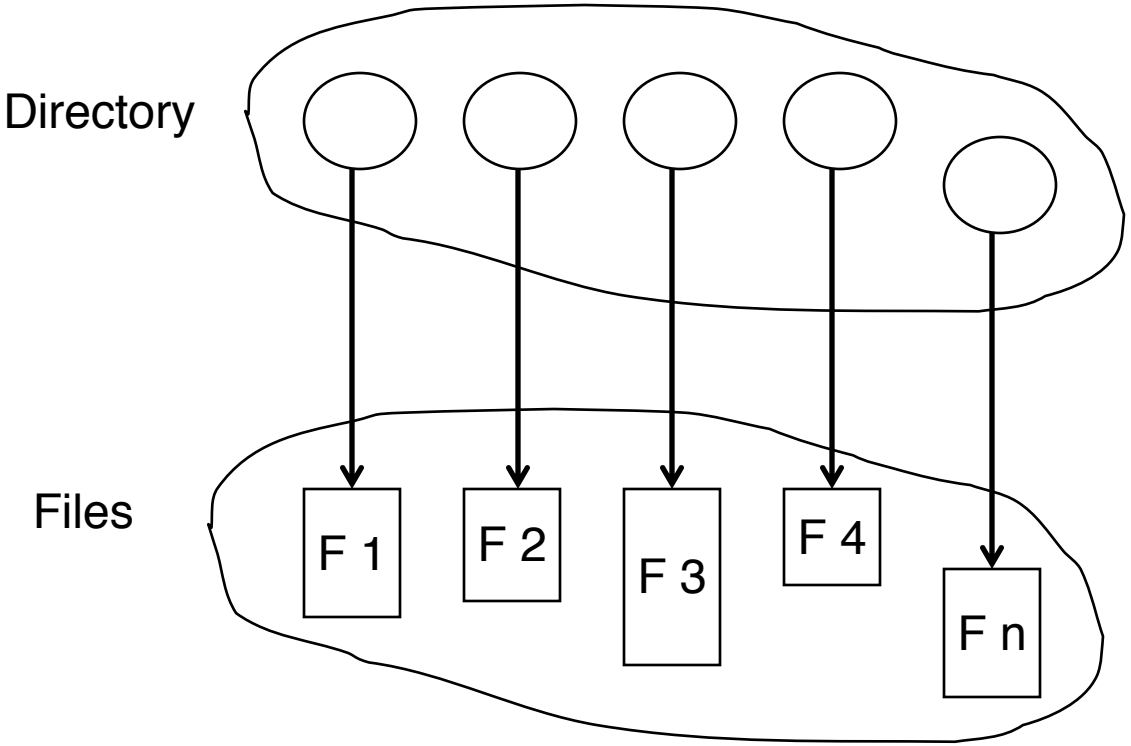
File system = Set of Files + Directory

Analogy: Book = Chapters + Index

What is a directory structure for Files?

Directory Structure

- A collection of nodes containing information about all files



Both the directory structure and the files reside on disk

Directory design: System model

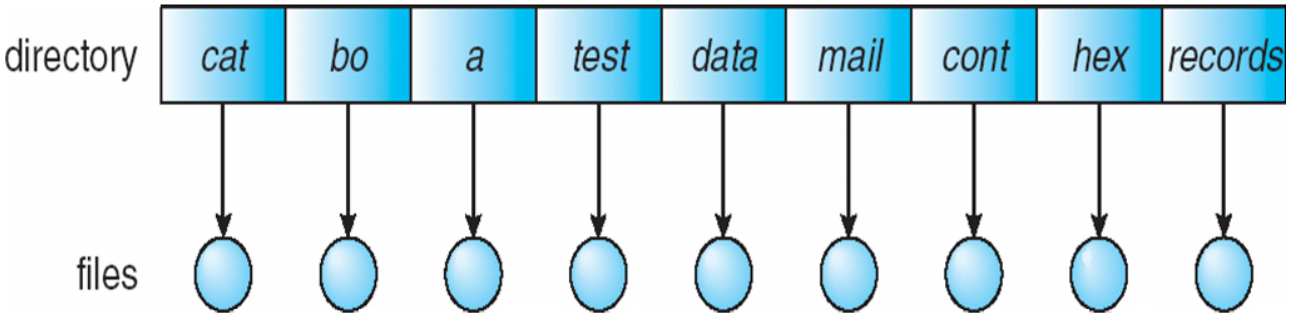
- System model
  - There are multiple users in a machine
  - Each user might need a bunch of files (e.g., in their home directory)
  - Users act independently

Desirable properties of directories

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

1. Single-Level Directory

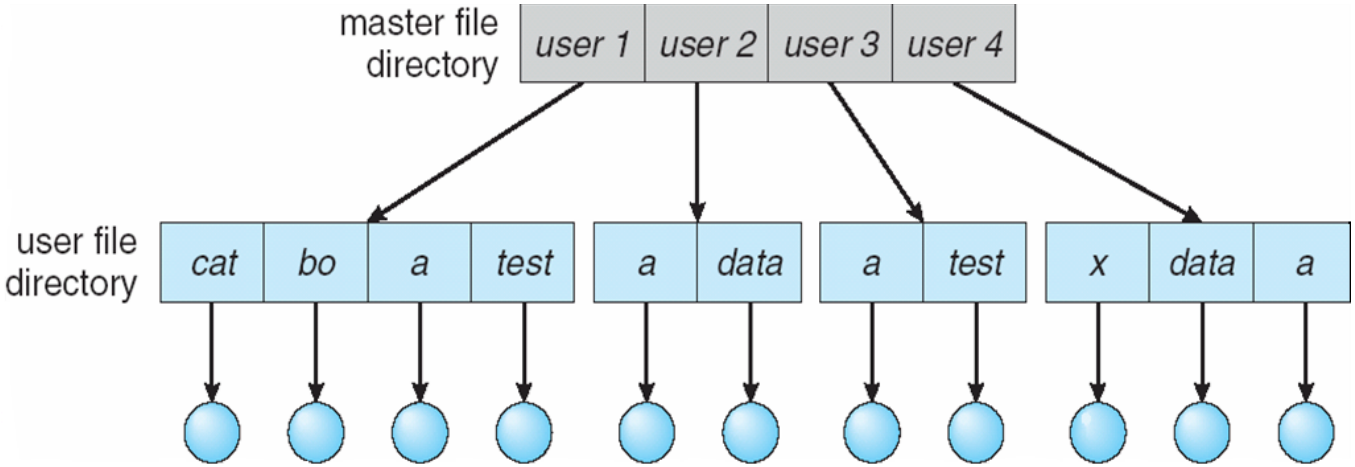
- A single directory for all users



- Problem: Searching might be linear
- Problem: Two different files can not have same name
- Problem: No grouping capability

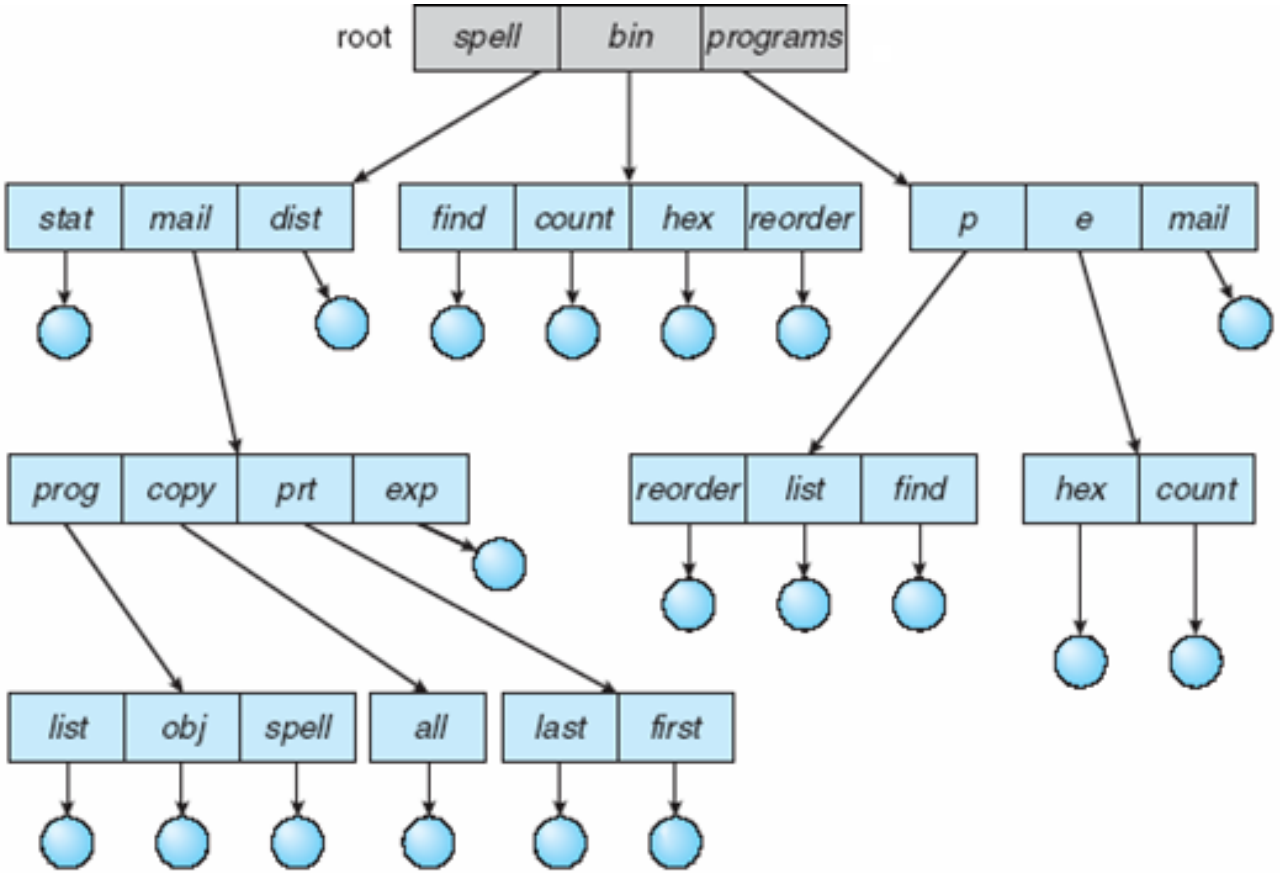
2. Two-Level Directory

- Separate directory for each user



- Good:
  - Efficient searching
  - Can have the same file name for different user (using path name)
- Problem:
  - No grouping capability

3. Tree-Structured Directories



root/spell/mail/prog/obj/filename

3. Tree-Structured Directories (Cont.)

- **Absolute** or **relative** path name
  - Creating a new file is done in current directory [command: mkdir]
- Good:
  - Efficient searching (tree traversal is better than linear search
  - Grouping Capability (using subdirectories)
- Current directory (working directory) [command: pwd]
  - `cd /spell/mail/prog` [change directory]

3. Tree-Structured Directories (Deletion)

- Delete a file

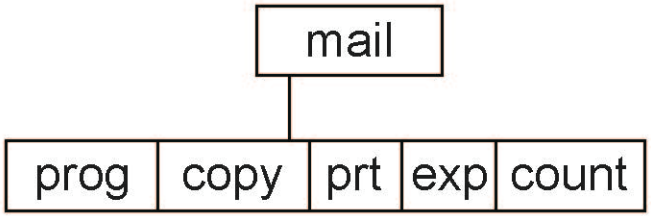
  `rm <file-name>`

- Creating a new subdirectory is done in current directory

  `mkdir <dir-name>`

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

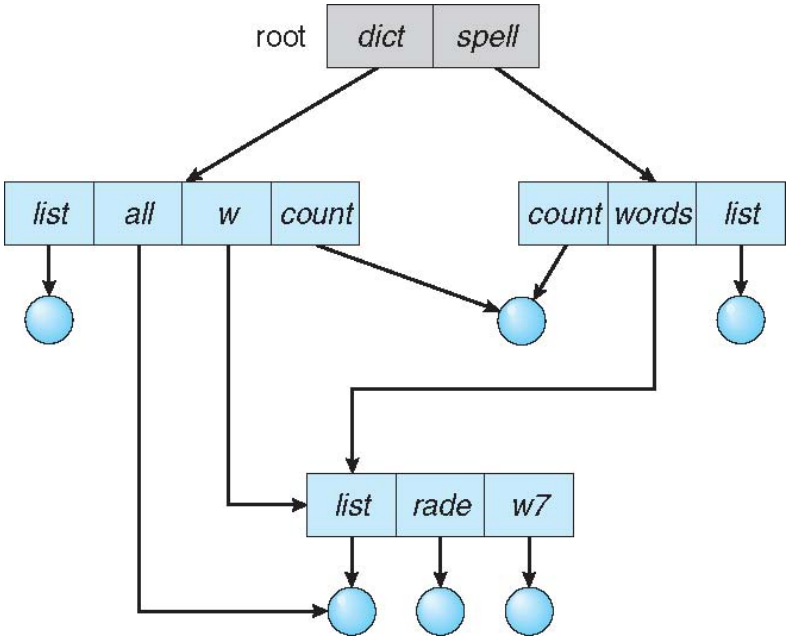
  `mkdir count`



- Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"
  - MS-DOS: If a directory contains file / subdir, cannot delete it
  - Unix: give a "recursive" option, delete all (**rm -r**)

4. Acyclic-Graph Directories: Enable file sharing

- Have shared subdirectories and files



- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file

Hard and soft linking

- In acyclic graph directories
  - Directories can share files using links between files
  - These links can be hard links or soft links
- **Hard links**
  - link to an actual data blocks in the disk
  - Command: ln <filename> <inkname>
  - cannot link directories
  - Hard links always refer to source files, even after its moved
- **Soft links (symbolic links)**
  - Symbolic link to the path where the file is stored
  - Command: ln -s <filename> <inkname>
  - Soft links can link directories
  - Destroyed when the file moved

Problem: How to ensure “acyclic”

- In if not acyclic what is the problem?
  - You have no idea how many nodes are in the tree using directory tree traversal
  - Say you have directory a and then create a link within a called c. c points to a
  - you go /a/c , /a/c/a, /a/c/a/c …
- **Solution**
  - 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

Problem: How to ensure “acyclic”

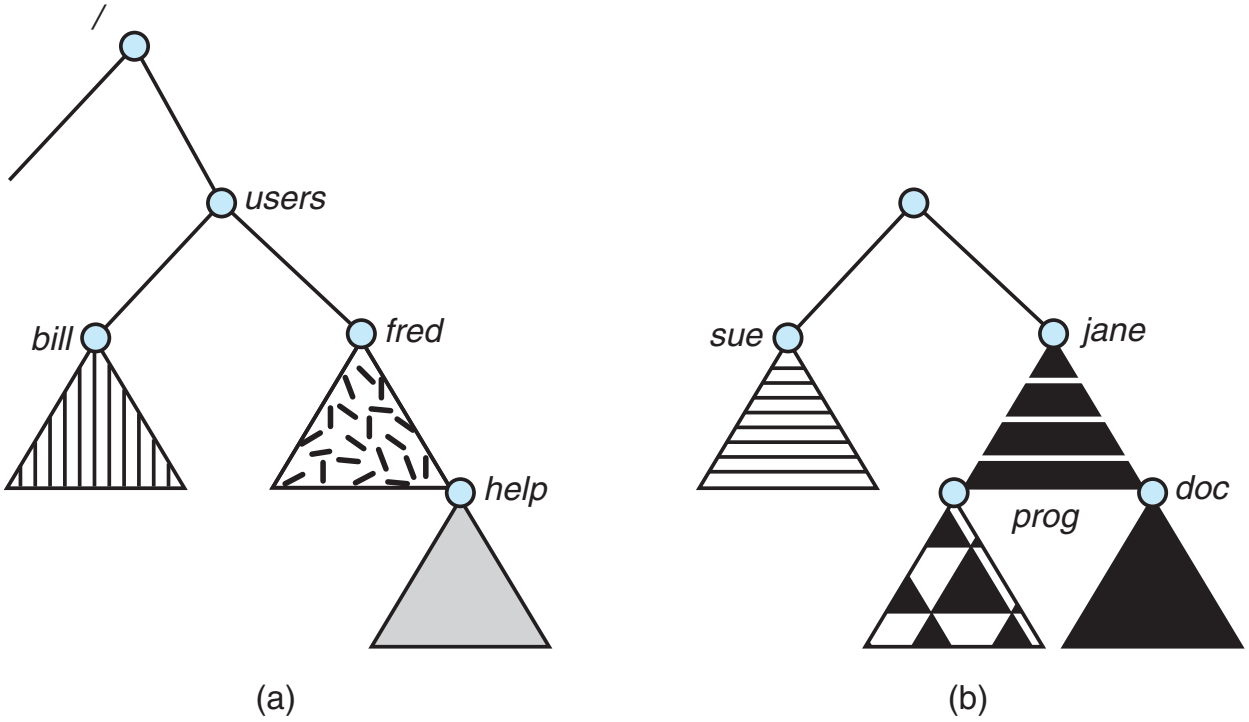
- In if not acyclic what is the problem?
  - You have no idea how many nodes are in the tree using directory tree traversal
  - Say you have directory a and then create a link within a called c. c points to a
  - you go /a/c , /a/c/a, /a/c/a/c …
- **Solution**
  - Allow only links to file not subdirectories
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK
- What is actually done?
  - Bypass links during directory traversal

File system = Set of Files + Directory

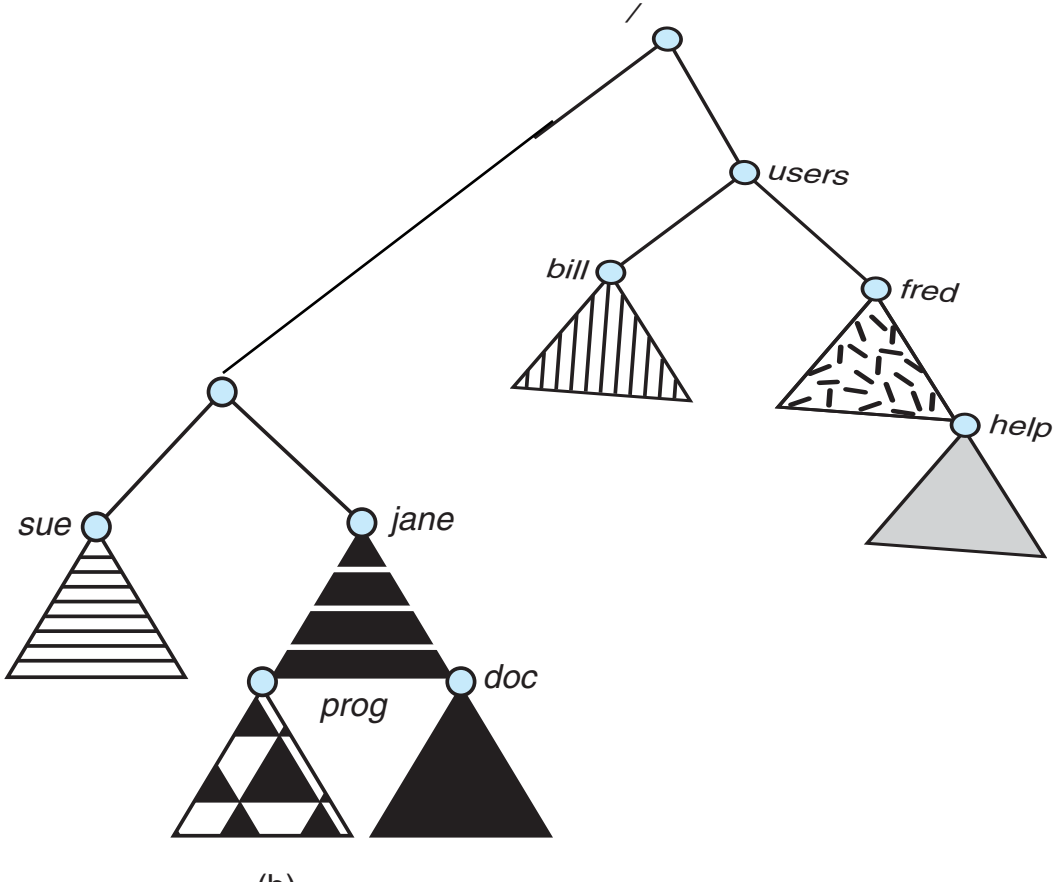
You can have a file system on a thumb drive : How can you tell that to an OS about the file system after plugging it in?

File System Mounting

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



Mount Point



How should the OS efficiently manage a persistent device (your hdd / ssd)?

Files, directory structures, mounting

- What are data read/write abstractions?
- What are the APIs?
- How to implement file system related functions?

How should the OS efficiently manage a persistent device (your hdd / ssd)?

Files, directory structures, mounting

- What are data read/write abstractions?
- What are the APIs?
- How to implement file system related functions?

Basic set of "FILE" Operations

- File is an **abstract data type**
- **Six basic operation**
  - **Create**
  - **Write –** at **write pointer** location
  - **Read –** at **read pointer** location
  - **Reposition within file - seek**
  - **Delete**
  - **Truncate (… > notes.txt** or **truncate –s <size> filename)** why is this operation needed?
- **Copying a file?**

```
#include<stdio.h>
void main() {
   FILE *fp;
   fp = fopen(”OS.txt", "w");
   …
   …
   fclose(fp);
}
```

Two ingredients for performing these operations: open, close

Open and Close Files

- ***Open(F_i)***
  - search the directory structure on disk for entry ***F_i*** *(file name)*
  - Checks privilege of the process to access file
  - Creates an entry in array of open files in the OS memory with reference to the file
  - The entry also contains the current r/w position in the file
  - Returns index of this file in the array – your file descriptor
- ***Close (F_i)***
  - That particular space for entry in your in-memory open file table can be reused

Open Files: Data structures needed

- ***Open(F_i)***
  - search the directory structure on disk for entry ***F_i*** *(file name)*
  - Checks privilege of the process to access file
  - Creates an entry in array of open files in the OS memory with reference to the file
  - The entry also contains the current r/w position in the file
  - Returns index of this file in the array – your file descriptor

Open Files: Data structures needed

- ***Open(F_i)***
  - search the directory structure on disk for entry F_i *(file name)*
  - Checks privilege of the process to access file

    Access rights: per-process access mode information
  - Creates an entry in array of open files in the OS memory with reference to the file

    Open-file table: tracks open files (per process or system?)
  - The entry also contains the current r/w position in the file

    File pointer: pointer to last read/write location, per process that has the file open
  - Returns index of this file in the array – your file descriptor

Close Files: Data structures needed

- ***Close (F_i)***
  - That particular space for entry in your in-memory open file table can be reused
  - **Requirement:** OS should not close a file (remove the entry from main memory) even if at least one process has it open

Close Files: Data structures needed

- ***Close (F_i)***
  - That particular space for entry in your in-memory open file table can be reused
  - **Requirement:** OS should not close a file (remove the entry from main memory) even if at least one process has it 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

So open file table entry will not be removed till this count is 0

Accessing Files after opening

- File pointer moves in the file to last r/w location. How?
- **Sequential Access**

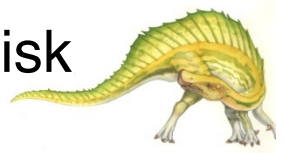
```
read next
write next
reset
```

- **Direct Access –** file is fixed length logical records (like an array)

```
read n
write n
rewrite n
```

n = relative block number (like an array index)

- Relative block numbers allow OS to control which disk block the user can start r/w from



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;

Basic set of "DIRECTORY" Operations

- Six basic operations
  - Search for a file (command: find)
  - Create a file (command: mkdir, touch)
  - Delete a file (command: rm –rf)
  - List a directory (command: ls)
  - Rename a file (command: rename)
  - Traverse the file system (command: cp -r)

How should the OS efficiently manage a persistent device (your hdd / ssd)?

Files, directory structures, mounting

- What are data read/write abstractions?
- What are the APIs?

Create, open, close, read write etc.

- How to implement file system related functions?

How should the OS efficiently manage a persistent device (your hdd / ssd)?

Files, directory structures, mounting

- What are data read/write abstractions?
- What are the APIs?

Create, open, close, read write etc.

- How to implement file system related functions?

How should the OS efficiently manage a persistent device (your hdd / ssd)?

Files, directory structures, mounting

- What are data read/write abstractions?
- What are the APIs?

Create, open, close, read write etc.

- How to implement file system related functions?
  - Implement open(), close() etc. API and directory
  - Allocate storage portion to files
  - Manage free space for efficient operation

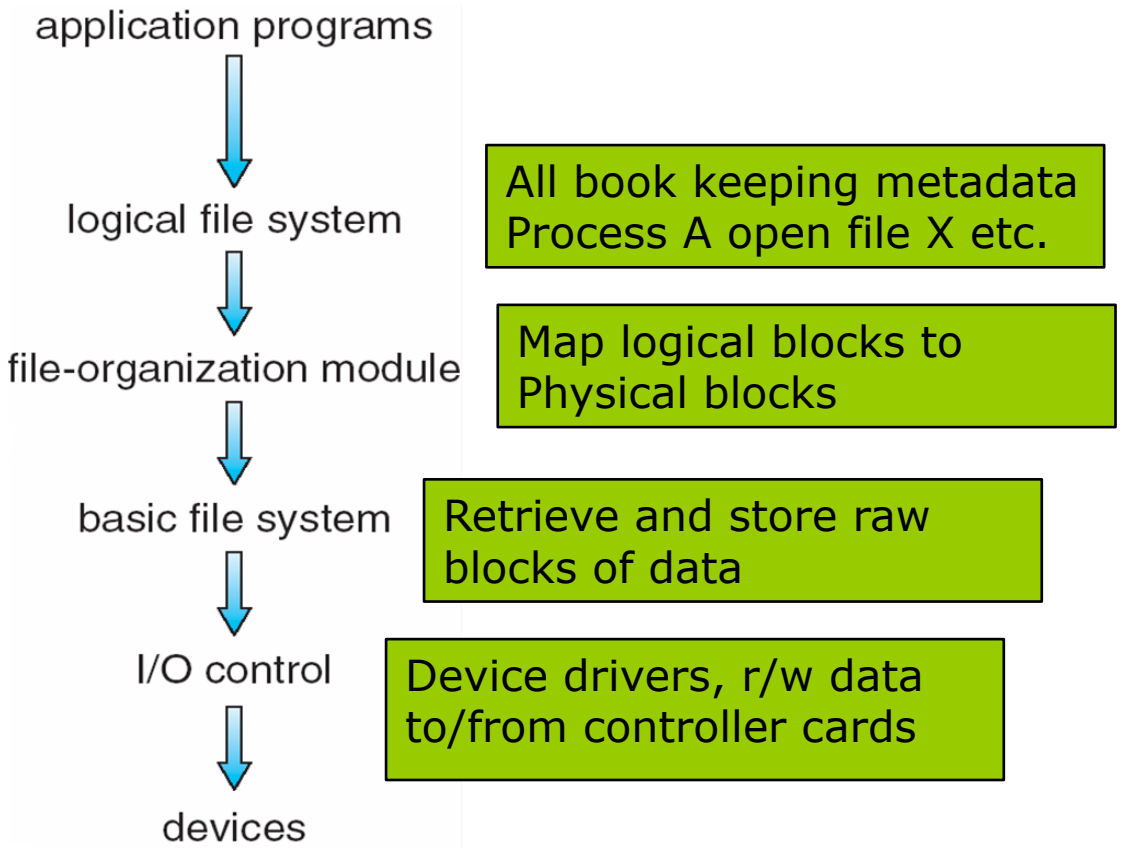
Background: Disk addressing

- We have hard drive/ssd/thumb drives in our computer and we want to use them via File systems
  - Disk exports its data as a logical array of blocks [0…N]
  - These blocks are the smallest unit of transfer
  - This is called **Logical Block addressing (LBA)**
  - We will call these blocks on secondary storage as “physical blocks”

Background: Logical structure of file system

- File structure
  - Logical storage unit
  - Collection of related information
- **File system** resides on secondary storage (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
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- File system organized into layers

Background: Layers of a File System



- **File control block** – storage structure consisting of information about a file
- **Device driver** controls the physical device

Implement API:
Data structures

On-Disk data structures

- **Boot control block**
  - One per partition
  - Information (address) of the boot loader (assembly code)
  - Boot loader loads OS code/data in the volume
  - Usually first block of the volume, may be empty
- **Volume control block (superblock, master file table)**
  - One per partition
  - Total # of blocks, block size, # of free blocks, block size, free block pointers, free File control block (FCB) or inode count, FCB/inode pointer array
- **Directory structure**
  - One per partition
  - Unix: File names and associated inode numbers,
  - NTFS: same thing, stored as master file table

On-Disk data structures (contd.)

- **File control block**
  - One per file
  - Unique identifier (Unix inode) number
  - permissions, size, dates
  - NFTS stores as part of in master file table using relational DB structures (one row, one file)

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

In-memory data structures

- **Mount Table**
  - Contains information about each mounted volume
- **Directory-Structure Cache**
  - Holds directory information about recently accessed directories
- **System-Wide Open-File Table**
  - Contains a copy of the FCB, i.e., inode, of each open file
- **Per-Process Open-File Table**
  - Contains a pointer to the appropriate entry in the System-Wide Open-File Table
- **I/O Memory Buffers**
  - Holds file-system blocks while they are being read from or written to disk

Implement API:

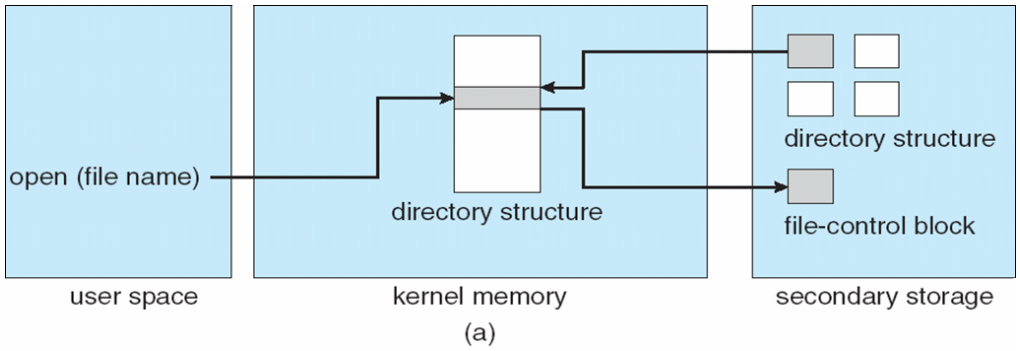
create, open, read, close

create() : Implementation

- Mechanism for create():
  - Application calls logical file system
  - Logical file system allocates a new FCB/inode
    - Allocated from free memory
    - selected from a pre-allocated set of free-FCB, i.e., free-inodes
  - OS reads corresponding (to the create call) directory in-memory
  - updates directory with the new file name and FCB
  - writes it back directory to the disk

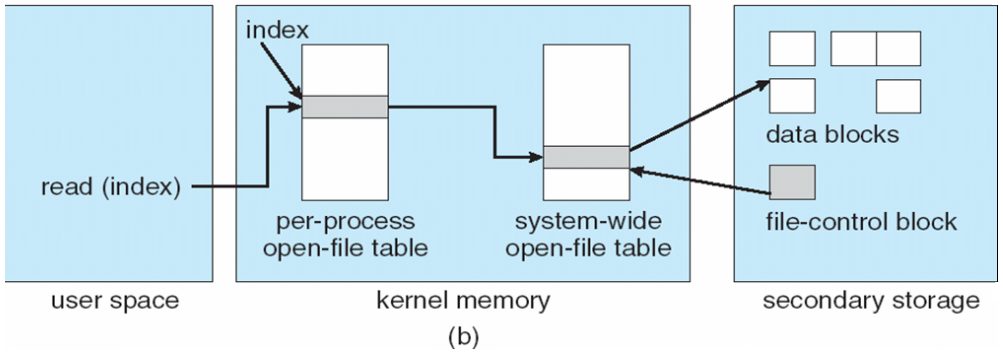
open() : Implementation

- Mechanism for open():
  - open() call passes a file name to the logical file system
  - Searches system-wide open-file table to check if the file is already in use by another process
    - Yes: a per-process open-file table entry created pointing to existing system-wide open-file table
    - No: directory structure is searched for the given file name (use directory caches). Once file is found, FCB is copied into a system-wide open-file table (with #process that called open()). Then same as the "Yes" case.
  - return a pointer to entry in per-process open-file table (the “file descriptor”)



read() : implementation

- Mechanism for read():
  - read() call passes the file descriptor (index in per-process open-file table)
  - The per-process open-file table entry at index points to the system-wide open-file table entry
  - The system-wide open-file table entry contain the FCB
  - Pass this information to the file-organization module and the layers below



(b)

close(): implementation

- Mechanism for close():
  - per-process open-file table entry removed
  - system-wide open-file table-entry's open count is decreased by 1
  - If the system-wide open-file table-entry's open count is 0
    - updated metadata is copied back to secondary storage directory
    - the system-wide open-file table entry is removed

Implementation note

- open(), close(), read(), write() etc. are nice understandable system calls
  - OS like Unix/Linux provide similar system calls also for other things like networking or basically any device your connect
  - system-wide open-file table-entries are not only FCB (i.e., inode) but also similar information for network connections and devices
  - Thus the adage: "Everything is a file" in Unix/Linux
  - More: https://en.wikipedia.org/wiki/Everything_is_a_file

- Why cache
  - Most OS keep *all* information about an open file in memory, except for the actual data blocks
  - in-memory buffer caches for both read() and write()
  - Average FS cache hit rate of Unix is 85%!

Implement Directory

Directory Implementation

- **Challenge:** Map the file name (i.e., full path to the file with directory names) to the physical data blocks on secondary storage

- **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 + Extra 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

How should the OS efficiently manage a persistent device (your hdd / ssd)?

Files, directory structures, mounting

- What are data read/write abstractions?
- What are the APIs?

Create, open, close, read write etc.

- How to implement file system related functions?
  - Implement open(), close() etc. API and directory
  - Allocate storage portion to files
  - Manage free space for efficient operation

File allocation methods

- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous allocation
    - Extent based allocation
  - Linked allocation
    - FAT tables
  - Indexed allocation
    - Multi-level indexed allocation

How to know good or bad?

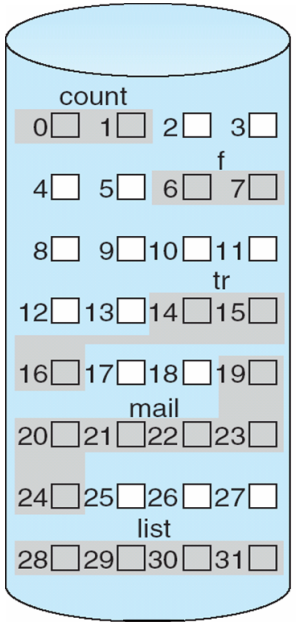
- Fragmentation
  - External: lots of holes
  - Internal: Whatever is allocated is not used
- Can you grow the file over time?
- Complexity of reading the files across blocks
- Easy implementation
- Storage overhead

1. Contiguous allocation

- Allocate files like continuous memory
  - Use base and limit values
  - User specifies length, file system allocates space all at once in continuous blocks
  - Metadata: start of the file and size

Contiguous Allocation: Schematic

Assign each file a set of contiguous blocks on the disk



directory

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

Contiguous Allocation: Good and Bad

- **Good**
  - Easy to implement
  - Low storage overhead
    - only storing starting location (block #) and length (number of blocks) are required
  - Fast sequential access: continuous blocks
  - Fast random access: accessing just an array index
- **Bad**
  - Wastes a lot of space
    - External fragmentation
    - Compaction (and related downtime)
  - Files cannot grow
    - Subsequent blocks may be occupied by other files

Contiguous Allocation: Mapping

- Mapping from logical address (LA) to physical
  - Logical address: relative to file, e.g., access LA'th byte in the file
  - Physical address: the actual physical block number on storage
  - Each physical block = 512 byte

$$Q = LA \,//\, 512$$

$$R = LA \bmod 512$$

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

1.1. Extent-based Allocation

- Extent-based file systems allocate disk blocks in extents
  - An extent is a contiguous fixed-size disk-block
  - Extents are allocated for file allocation
  - A file consists of one or more extents
- File = set of extents
  - Metadata: Small array of entries pointing to extent start address

Extent Allocation: Good and Bad

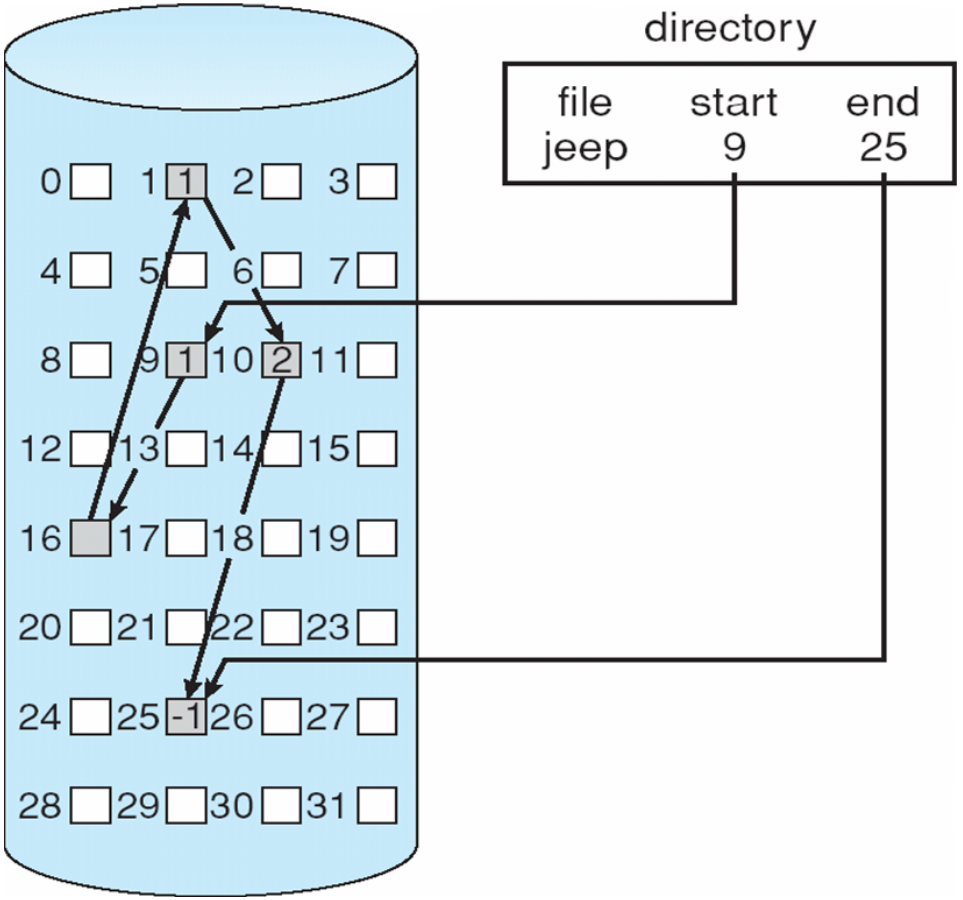
- **Good**
  - Easy to implement
  - Low storage overhead (a few entries to specify extents)
  - File can grow (unless run out of extents)
  - Fast sequential access
  - random access: Simple to calculate extent number and offset
- **Bad**
  - Internal fragmentation
  - External fragmentation can still happen when files grow and shrink dynamically

File allocation methods

- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous allocation
    - Extent based allocation
  - Linked allocation
    - FAT tables
  - Indexed allocation
    - Multi-level indexed allocation

2. Linked Allocation

- **Linked allocation** – each file a linked list of blocks
- Directory entry = address of first block
- File ends at nil pointer

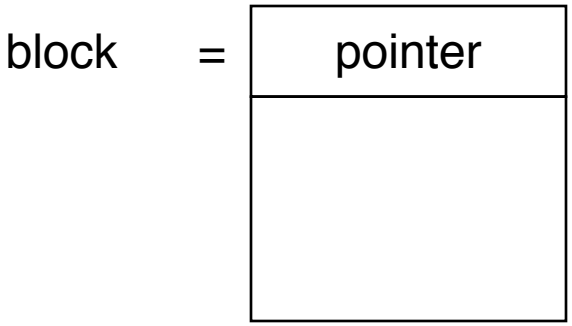


Linked Allocation: Good and Bad

- **Good**
  - No compaction, external fragmentation
  - Files can easily grow
  - Easy to implement but need to space space per block!
- **Bad**
  - Storage overhead for pointers
  - Linear search -- Locating a block can take many I/Os and disk seeks, slow sequential
  - Difficult to compute block number for random access
  - Reliability : Pointers can be broken
- Improve efficiency by clustering blocks into groups but increases internal fragmentation (wasted allocated space)

Linked Allocation : Mapping

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



- Pointer size = 4 bytes
  - Mapping from logical to physical

$$Q = LA \,//\, 508$$

$$R = LA \bmod 508$$

Block to be accessed is the Qth block in the linked chain of blocks representing the file.

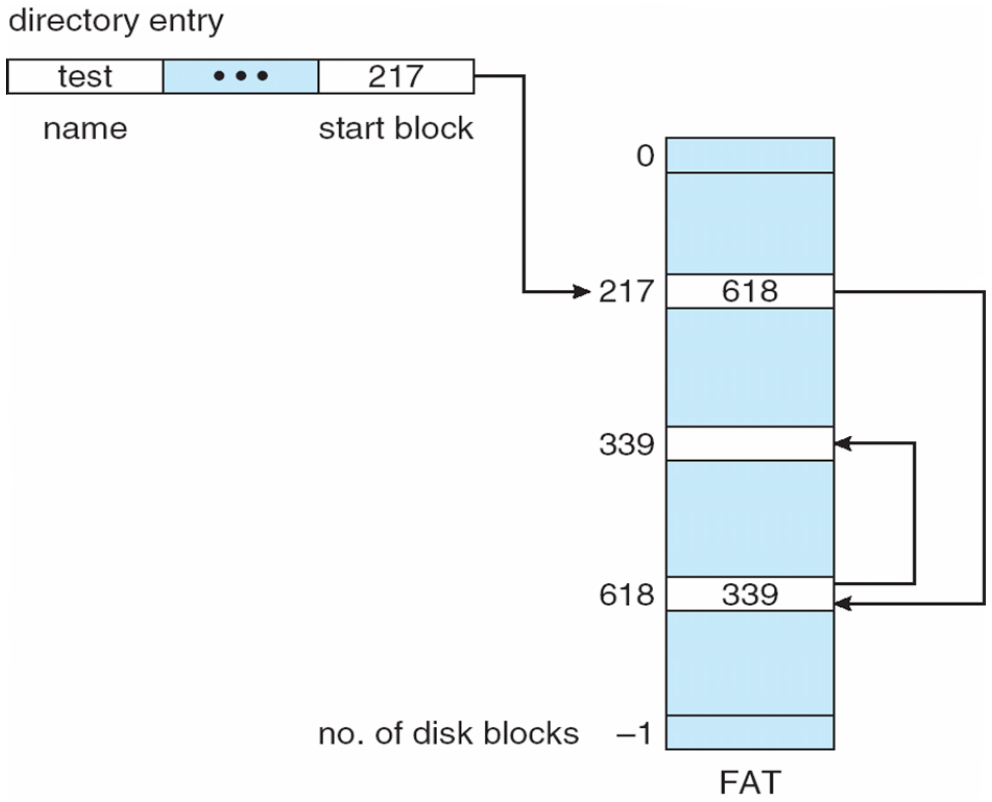
Displacement into block = R + 4

2.2. Special Linked allocation: FAT

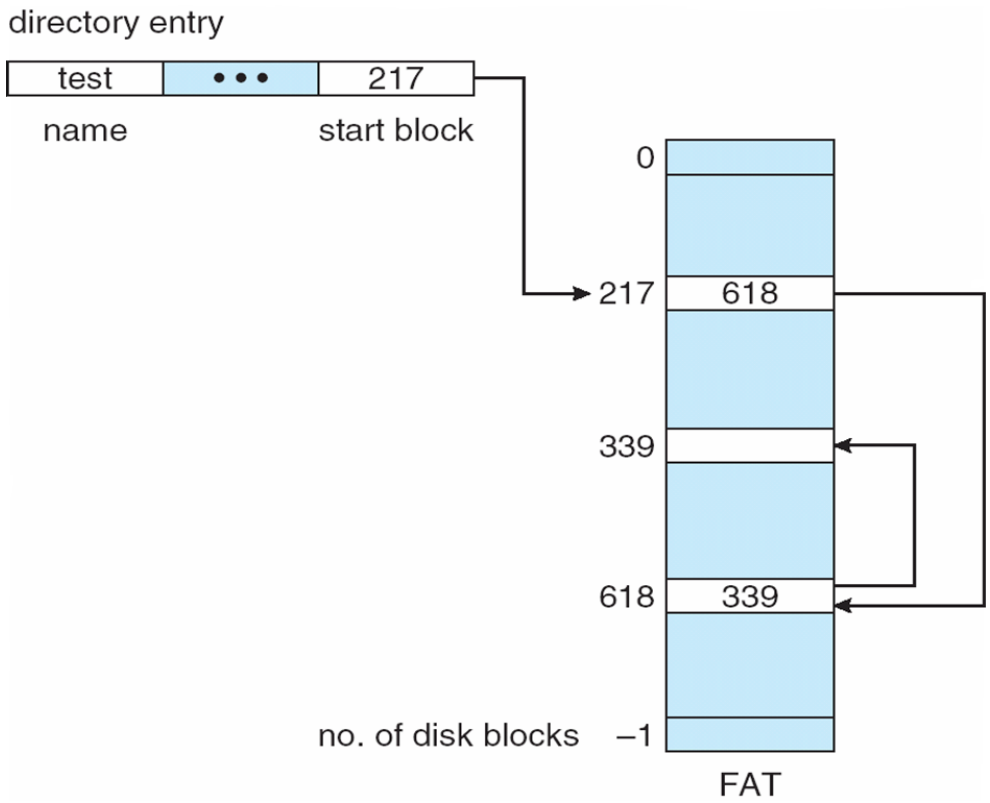
- FAT (File Allocation Table)
  - Idea: Store linked-list pointers outside block in **File Allocation Table**
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

- FAT (File Allocation Table) working
  - Directory contains first block number for a file
  - FAT entry indexed by the block number == next block
  - Go to next block and again consult FAT for next block
  - Last block == special EOF value in FAT entry

File-Allocation Table

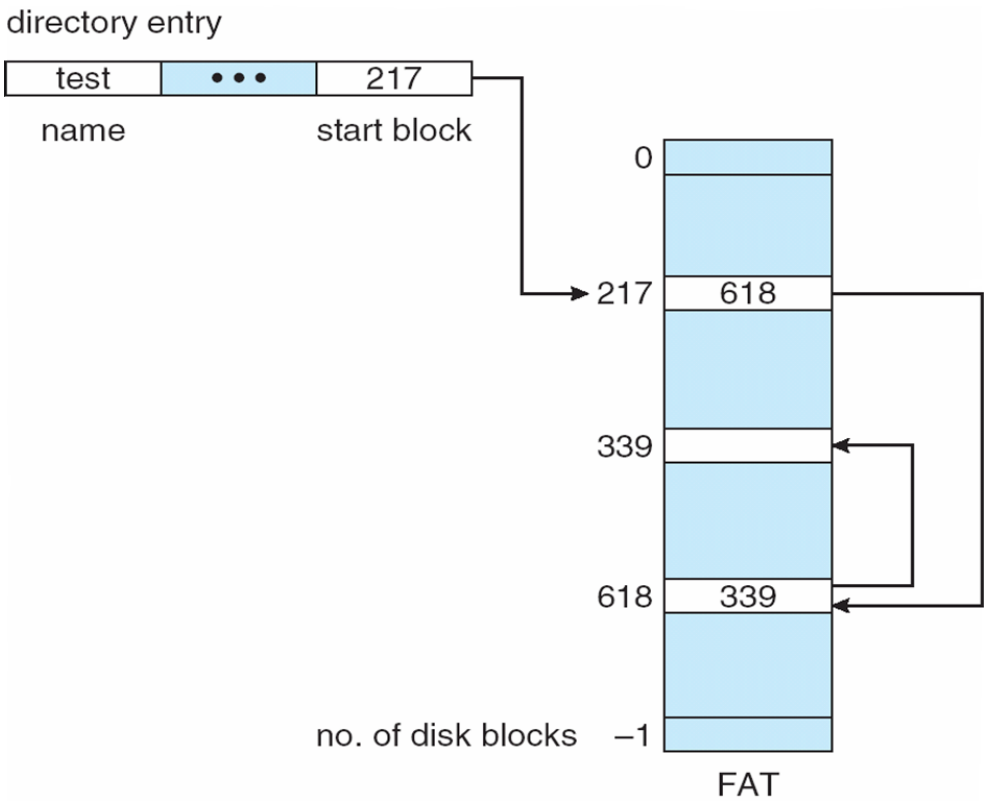


File-Allocation Table



Question: you have a 512 GB hard disk and each block size is 4 KB. If your File system contains FAT, what is the smallest amount of memory used for FAT?

File-Allocation Table



Question: you have a 512 GB hard disk and each block size is 4 KB. If your File system contains FAT, what is the smallest amount of memory used for FAT? = 27 bits * 2^27 blocks

FAT: Good and Bad

- **Good**
  - Fast random access, only search cached FAT in memory
- **Bad**
  - **Storage overhead for FAT**
  - Slow sequential access, repeated FAT lookup

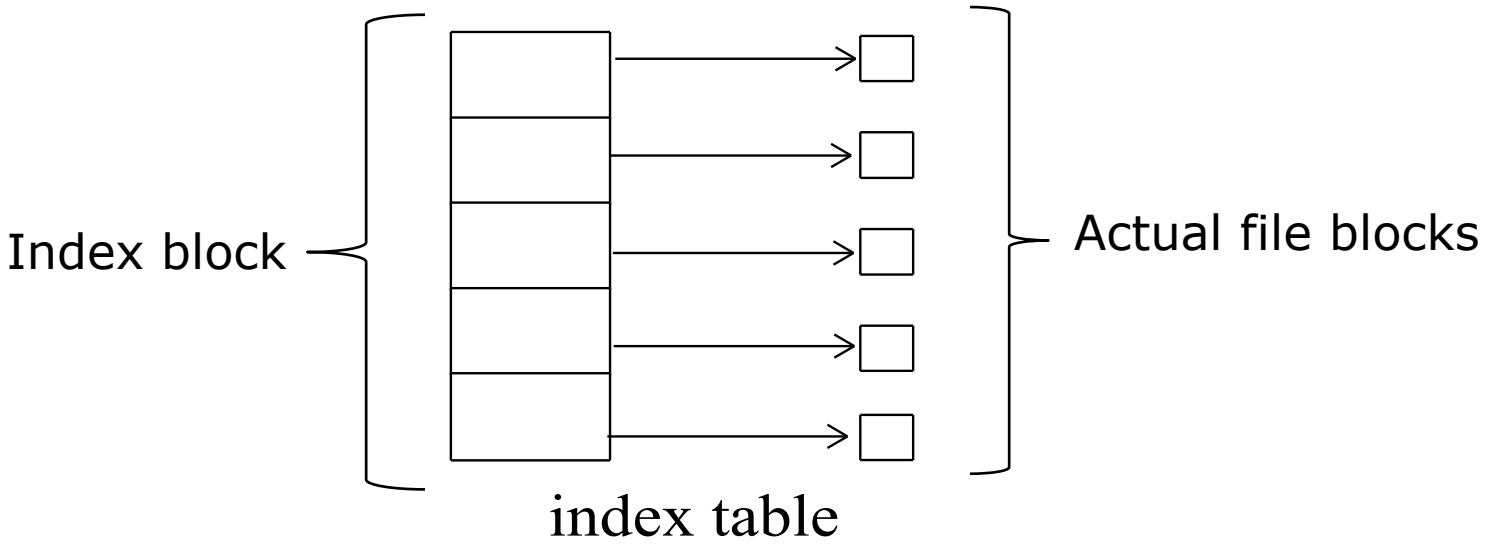
File allocation methods

- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous allocation
    - Extent based allocation
  - Linked allocation
    - FAT tables
  - **Indexed allocation**
    - **Multi-level indexed allocation**

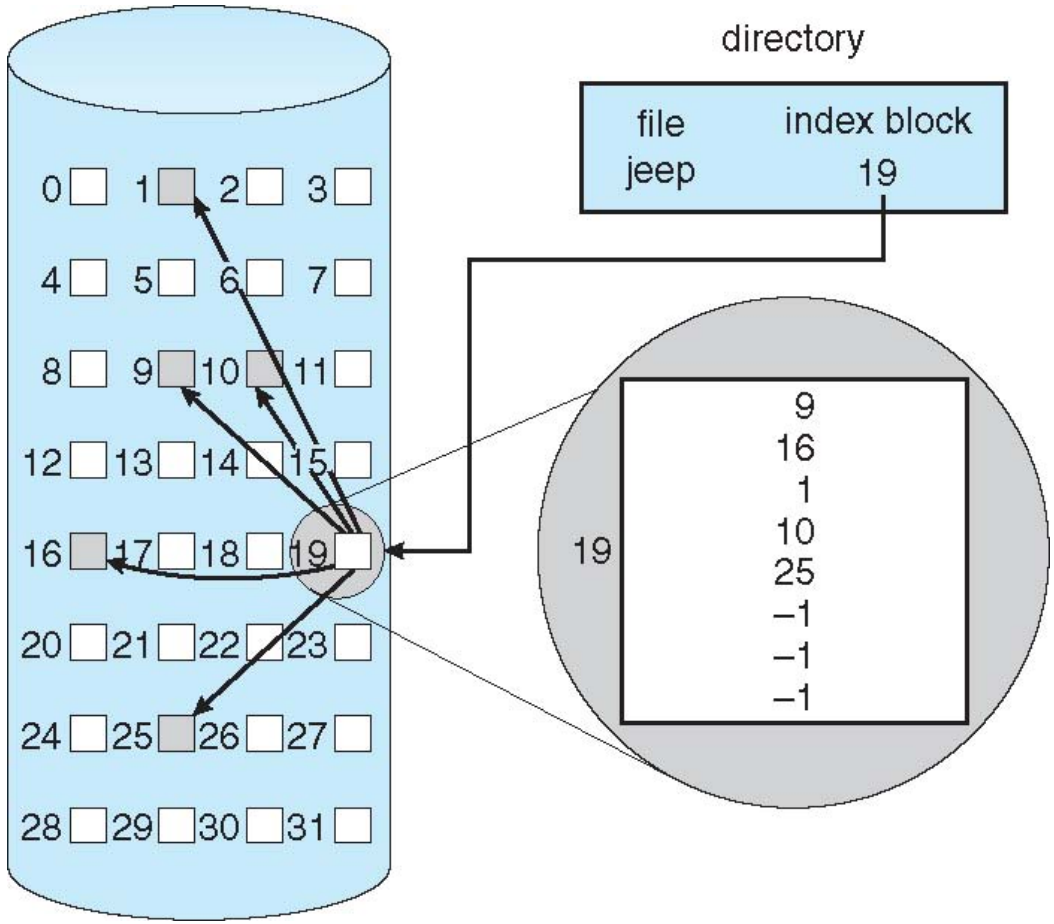
3. Indexed Allocation

- **Indexed allocation**
  - Each file has its own **index block**(s) of pointers to its data blocks
  - Directory entry points to index block
  - entry i in the index block points to the i-th physical block for the file

- Schematic view



Example of Indexed Allocation



Indexed Allocation (Cont.)

- Need index table
- Allow random access without external fragmentation
  - Dynamic increase is easy, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 64K bytes, block size of 512 bytes and entry size 32 bits
  - We need only 1 block for index table. Why?

$$Q = LA \,//\, 512$$

$$R = LA \bmod 512$$

Q = displacement into index table
R = displacement into block

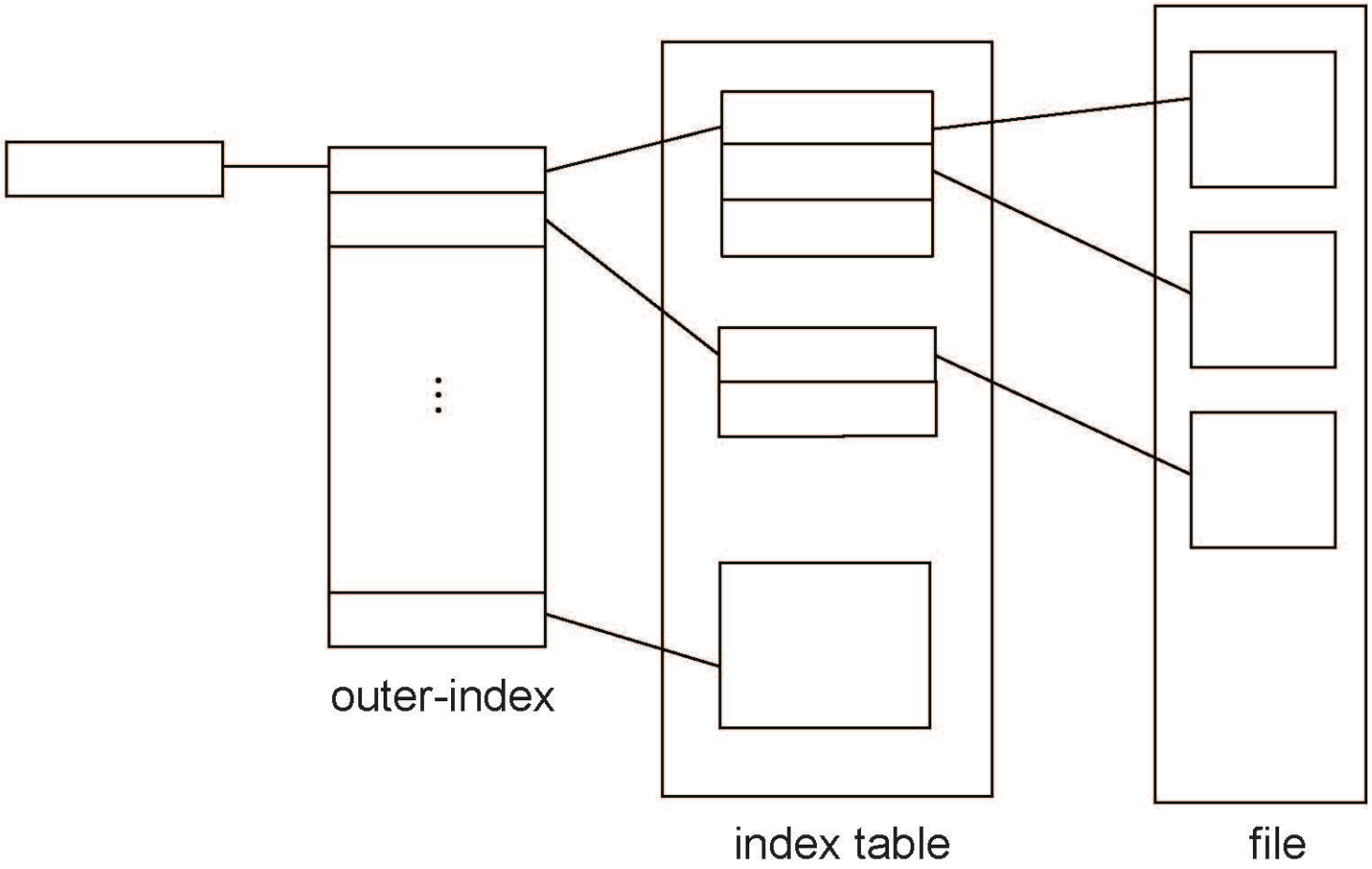
Indexed allocation: Good and Bad

- **Good**
  - Easy to implement
  - No external fragmentation
  - Files can be easily grown **(but has a maximum limit)**
  - Fast random access, use index block
- **Bad**
  - Storage overhead for index
  - Slow sequential access, repeated index table lookup

A slight improvement : Linked scheme

- Increase maximum size of file
- Create an index block as a disk block
- Then if necessary add a pointer to the end to another new index block
  - Linked scheme – Link blocks of index table (no limit on size)

3.2.Even better : multilevel index



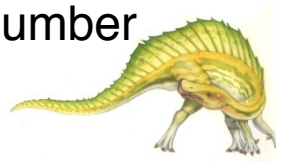
Multilevel index

- First-level index block is a disk block which holds address for second-level index block
- If level = 2, then second-level index points to data blocks

- What is the maximum size of files for two level index with 4kB sized blocks and 4 byte pointers?
  - Each 4KB index block, number of entries = (4 x 2^10 bytes) / 4 bytes = 2^10 pointers
  - First index block = 2^10 pointers to 2^10 index blocks
  - So, total #entries in all second level tables = 2^10 x 2^10
  - Each entry in each second level table points to a 4 KB block
  - So maximum file size = 2^10 x 2^10 x 4 KB = 4 GB

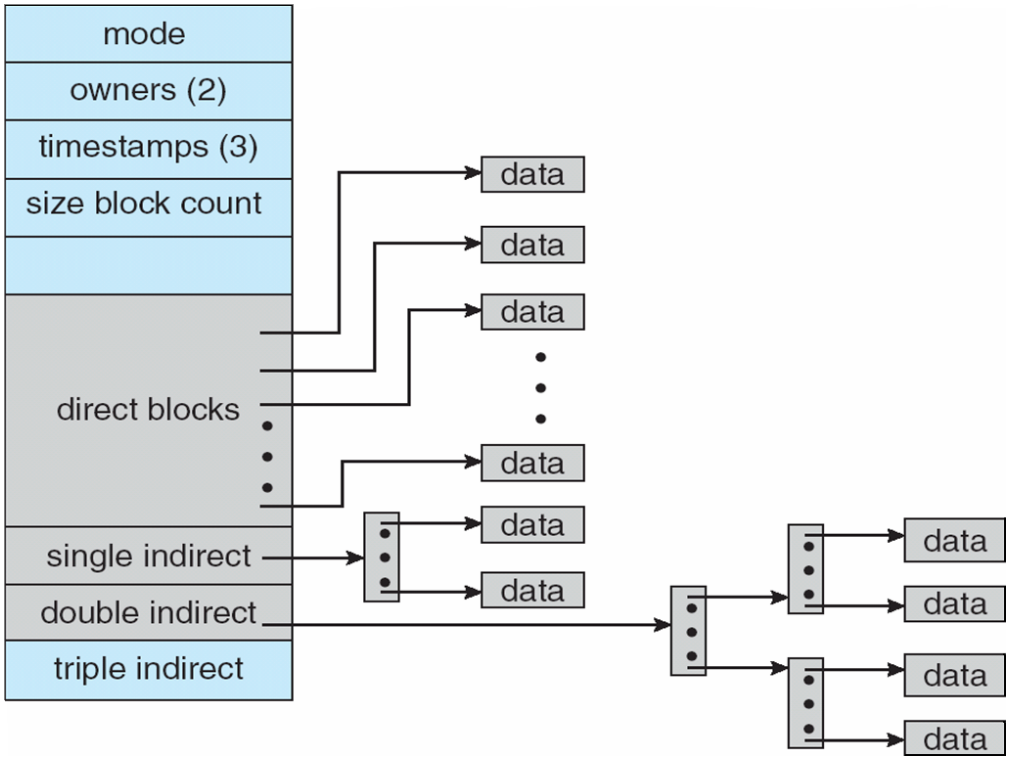
Multilevel index

- First-level index block is a disk block which holds address for second-level index block
- If level = 2, then second-level index points to data blocks
- What is the maximum size of files for two level index with 4kB sized blocks and 4 byte pointers?
  - Each 4KB index block, number of entries = (4 x 2^10 bytes) / 4 bytes = 2^10 pointers
  - First index block = 2^10 pointers to 2^10 index blocks
  - So, total #entries in all second level tables = 2^10 x 2^10
  - Each entry in each second level table points to a 4 KB block
  - So maximum file size = 2^10 x 2^10 x 4 KB = 4 GB
  - Exercise: What would be the max. file size for n level index with 4m byte block sizes and 4 byte pointer sizes? [Ans: 4 x m^(n+1)]
  - Exercise: Find the logical file specific byte to physical block mapping in this two–level scheme. Assume a block size of 512 bytes and pointer size of 4 bytes. [Hint: First find the block number in 2nd index table]



Indexed allocation: Combined Scheme:

Used in UNIX
4K bytes per block, 32-bit addresses



More index blocks than can be addressed with 32-bit file pointer

So which one to choose?

- 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
  - A hybrid is more preferable: contiguous for small files and indexed for large files

How should the OS efficiently manage a persistent device (your hdd / ssd)?

Files, directory structures, mounting

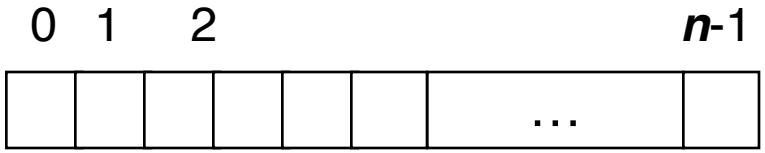
- What are data read/write abstractions?
- What are the APIs?

Create, open, close, read write etc.

- How to implement file system related functions?
  - Implement open(), close() etc. API and directory
  - Allocate storage portion to files
  - Manage free space for efficient operation

Free-Space Management: Bit map approach

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

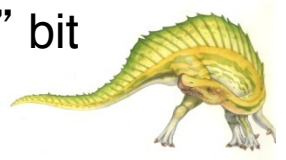


$$bit[i] = \begin{cases} 1 \Rightarrow block[i] \text{ free} \\ 0 \Rightarrow block[i] \text{ occupied} \end{cases}$$

Block number calculation

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

CPUs have instructions to return offset within word with first "1" bit



Free-Space Management (Cont.)

- Bit map requires extra space
  - Example:

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

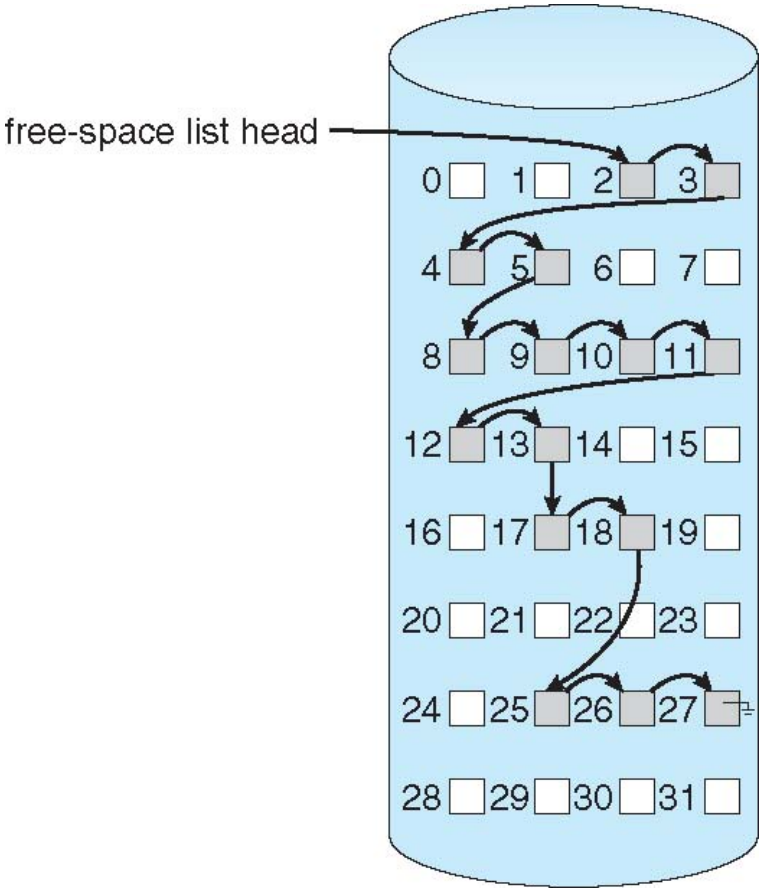
    disk size = 2^{40} bytes (1 terabyte)

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

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

Free-Space Management: Linked List approach

- Linked list (free list)
  - No waste of space
  - Expensive to traverse the list
    - Hard for contiguous assignment
  - But OS often needs one free block for a file
    - The first one is returned



Summary

- What are data read/write abstractions?
  - Files, directory structures, mounting
- What are the APIs?
  - Create, open, close, read write etc.
- How to implement file system related functions?
  - open(), close() etc. API and directory
    - In memory and on disk data structures, FCB/inode, open file tables, directory = Linear list + Hash table
  - Allocate storage portion to files
    - Contiguous (extent), linked (FAT), indexed (multi-level index)
  - Manage free space for efficient operation
    - bitmap, linked list

Extra concept: R/W optimization

- Performance improvement
  - **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
    - Remove a used block and read requested block + subsequent blocks
  - Reads frequently slower than writes.
    - Why? Read ahead

Extra Concept: Recovery of File systems

- Why do you need recovery?
  - Imagine your code has 4 system calls, first two creates a FCB and add it to directory, second two writes file data to disk blocks
  - Now what if system crash after first two?
  - The FCB is there, but no file data → inconsistent state
- How to deal with inconsistency?
  - scan all data on directory and compares with data in each block to determine consistency
  - *fsck* in Linux, *chkdsk* in Windows
  - Super slow
  - May require human intervention to resolve conflicts

Extra concept of File system security: Protection

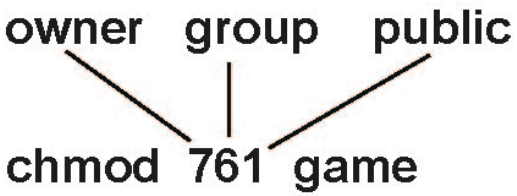
- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

Access Lists and Groups

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

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

- For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

```
chgrp      G      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/

- Question: What is the meaning of chmod 777, chmod 644, chmod 700
- Check : http://www.filepermissions.com/articles/what-are-file-permissions-in-linux-and-mac