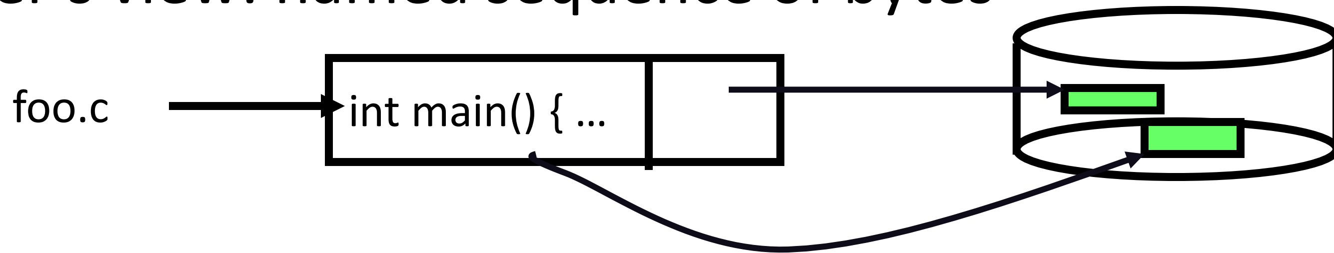
# Lecture 17: File Systems

# Building a File System

- File System: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.
- File System Components
  - Disk Management: collecting disk blocks into files
  - Naming: Interface to find files by name, not by blocks
  - Protection: Layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc
- User vs. System View of a File
  - User's view:
    - Durable Data Structures
  - System's view (system call interface):
    - Collection of Bytes (UNIX)
    - Doesn't matter to system what kind of data structures you want to store on disk!
  - System's view (inside OS):
    - Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
    - Block size $\geq$ sector size; in UNIX, block size is 4KB

# Files: named bytes on disk

- File abstraction:
  - user's view: named sequence of bytes




  - FS's view: collection of disk blocks
  - file system's job: translate name & offset to disk blocks

offset:int → disk addr:int

- File operations:
  - create a file, delete a file
  - read from file, write to file
- Want: operations to have as few disk accesses as possible & have minimal space overhead

# Files?

- Latex source, .o file, shell script, a.out, …
- UNIX: file = sequence of bytes
  - Shell scripts: first byte=#
  - Perl scripts: start with #!/usr/bin/perl, ….
- Mac: file has a type which associates it with the program that created it
- DOS/Windows: Use file extensions to identify file (ad-hoc)

# File attributes

- Name
- Type – in Unix, implicit
- Location – where file is stored on disk
- Size
- Protection
- Time, date, and user identification

- All filesystem information stored in non-volatile storage – important for crash recovery

# Lots of file formats, or few file formats?

- UNIX: one file format
- VMS: three file formats
- IBM: lots

# Translating from User to System View

- What happens if user says: give me bytes 2—12?
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block
- What about: write bytes 2—12?
  - Fetch block
  - Modify portion
  - Write out Block
- Everything inside File System is in whole size blocks
  - For example, `getc()`, `putc()` $\Rightarrow$ buffers something like 4096 bytes, even if interface is one byte at a time
- From now on, file is a collection of blocks

# What's so hard about grouping blocks???

- In some sense, the problems we will look at are no different than those in virtual memory
  - like page tables, file system meta data are simply data structures used to construct mappings.
  - Page table: map virtual page # to physical page #

  - file meta data: map byte offset to disk block address

  - directory: map name to disk block address

# Disk Management Policies

- Basic entities on a disk:
  - File: user-visible group of blocks arranged sequentially in logical space
  - Directory: user-visible index mapping names to files (next lecture)
- Access disk as linear array of sectors. Two Options:
  - Identify sectors as vectors [cylinder, surface, sector]. Sort in cylinder-major order. Not used much anymore.
  - Logical Block Addressing (LBA). Every sector has integer address from zero up to max number of sectors.
  - Controller translates from address $\Rightarrow$ physical position
    - First case: OS/BIOS must deal with bad sectors
    - Second case: hardware shields OS from structure of disk

# Disk Management Policies

- Need way to track free disk blocks
  - Link free blocks together $\Rightarrow$ too slow today
  - Use bitmap to represent free space on disk
- Need way to structure files: File Header
  - Track which blocks belong at which offsets within the logical file structure
  - Optimize placement of files' disk blocks to match access and usage patterns

# FS vs VM

- In some ways problem similar:
  - want location transparency, oblivious to size, & protection
- In some ways the problem is easier:
  - CPU time to do FS mappings not a big deal (= no TLB)
  - Page tables deal with sparse address spaces and random access, files are dense (0 .. filesize-1) & ~sequential
- In some way's problem is harder:
  - Each layer of translation = potential disk access
  - Space a huge premium! (But disk is huge?!?!) Reason? Cache space never enough, the amount of data you can get into one fetch never enough.
  - Range very extreme: Many <10k, some more than GB.
  - Implications?

# Some working intuitions

- FS performance dominated by # of disk accesses
  - Each access costs 10s of milliseconds
  - Touch the disk 50-100 extra times = 1 *second*
  - Can easily do 100s of millions of ALU ops in same time
- Access cost dominated by movement, not transfer
  - Can get 20x the data for only ~5% more overhead
  - 1 sector = 10ms + 8ms + 50us (512/10MB/s) = 18ms
  - 20 sectors = 10ms + 8ms + 1ms = 19ms
- Observations:
  - all blocks in file tend to be used together, sequentially
  - all files in a directory tend to be used together
  - all names in a directory tend to be used together
  - How to exploit?

# Common addressing patterns

- Sequential:
  - file data processed in sequential order
  - by far the most common mode
  - example: editor writes out new file, compiler reads in file, etc.
- Random access:
  - address any block in file directly without passing through predecessors
  - examples: data set for demand paging, databases
- Keyed access:
  - search for block with particular values
  - examples: associative data base, index
  - usually not provided by OS

# Problem: how to track file's data?

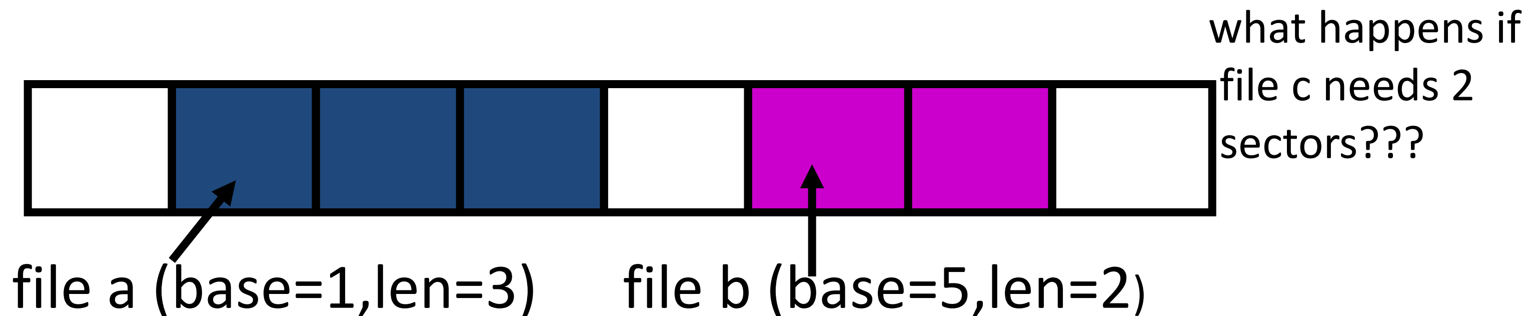
- Disk management:
  - Need to keep track of where file contents are on disk
  - Must be able to use this to map byte offset to disk block
  - The data structure used to track a file's sectors is called a file descriptor
  - file descriptors often stored on disk along with file
- Things to keep in mind while designing file structure:
  - Most files are small
  - Much of the disk is allocated to large files
  - Many of the I/O operations are made to large files
  - Want good sequential and good random access
  - What do I need?

# Designing a File System

- Sequential access mode only
  - Lay out file sequentially on disk
  - Problems?
- Direct access mode
  - Have a disk block table telling where each disk block is
- Indexed access mode
  - Build an index on the identifier (more sophisticated).
  - E.g., IBM ISAM (Indexed Sequential Access Mode): User selects key, and system builds a two-level index for the key

# Simple mechanism: contiguous allocation

- "Extent-based": allocate files like segmented memory
  - When creating a file, make the user specify pre-specify its length and allocate all space at once
  - File descriptor contents: location and size



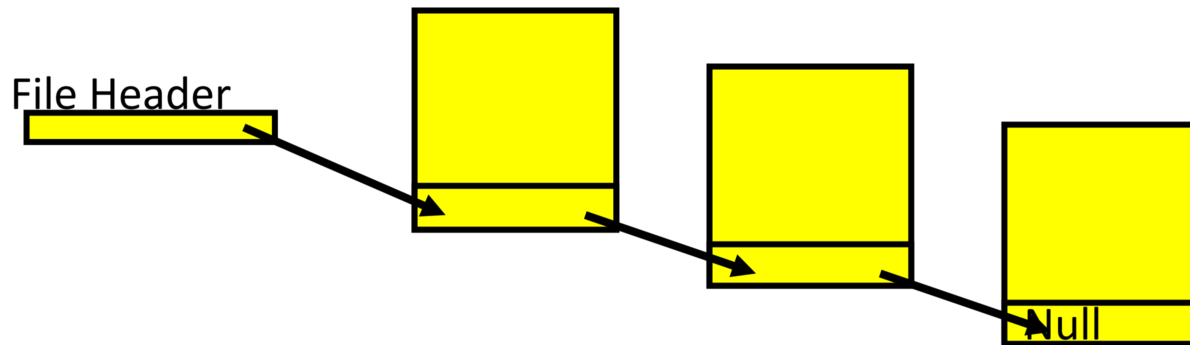

  - Example: IBM OS/360
  - Pro: simple, fast access, both sequential and random.
  - Cons? (What does VM scheme does this correspond to?)

# First Technique: Continuous Allocation

- Use continuous range of blocks in logical block space
  - Analogous to base+bounds in virtual memory
  - User says in advance how big file will be (disadvantage)
- Search bit-map for space using best fit/first fit
  - What if not enough contiguous space for new file?
- File Header Contains:
  - First block/LBA in file
  - File size (# of blocks)
- Pros: Fast Sequential Access, Easy Random access
- Cons: External Fragmentation/Hard to grow files
  - Free holes get smaller and smaller
  - Could compact space, but that would be *really* expensive

# Linked List Allocation

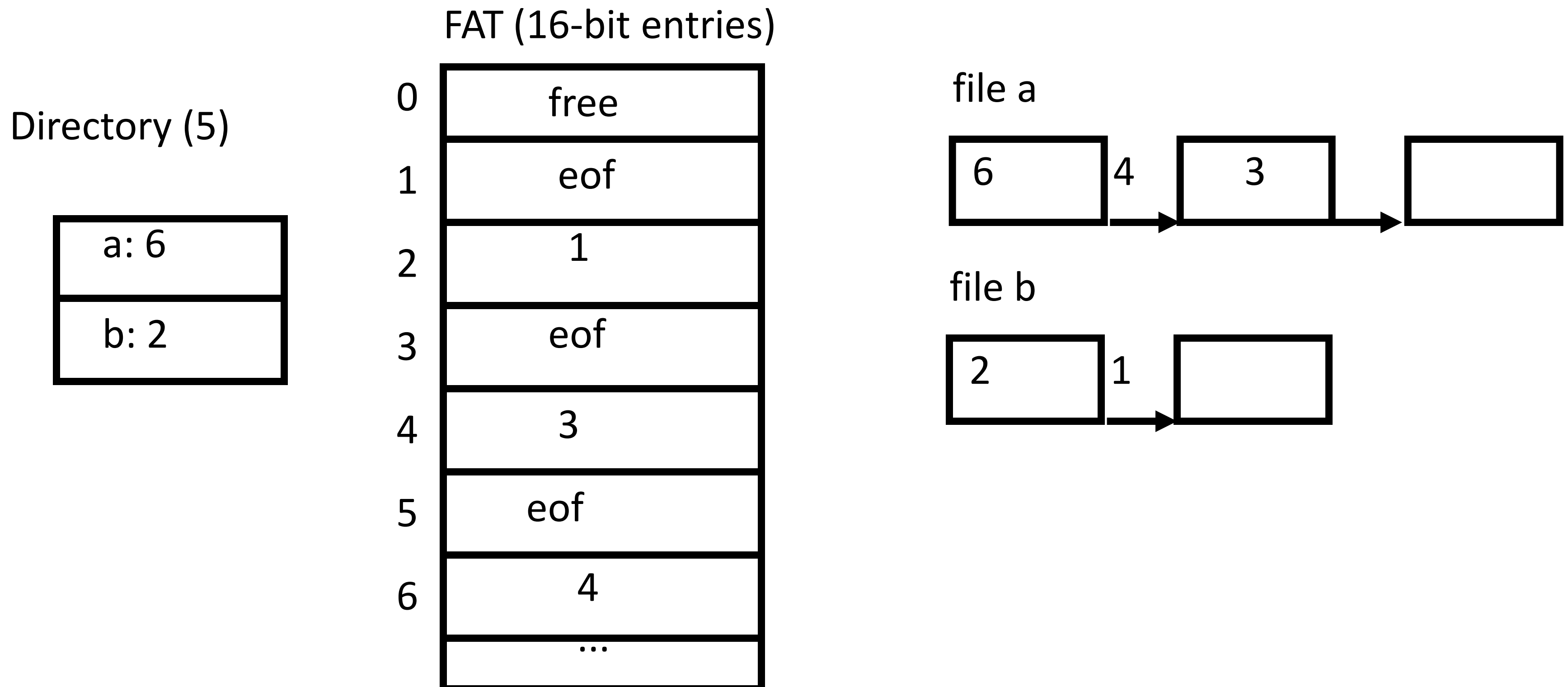
- Second Technique: Linked List Approach
  - Each block, pointer to next on disk




  - Pros: Can grow files dynamically, Free list same as file
  - Cons: Bad Sequential Access (seek between each block), Unreliable (lose block, lose rest of file)
  - Serious Con: Bad random access!!!!
  - Technique originally from Alto (First PC, built at Xerox)
    - No attempt to allocate contiguous blocks

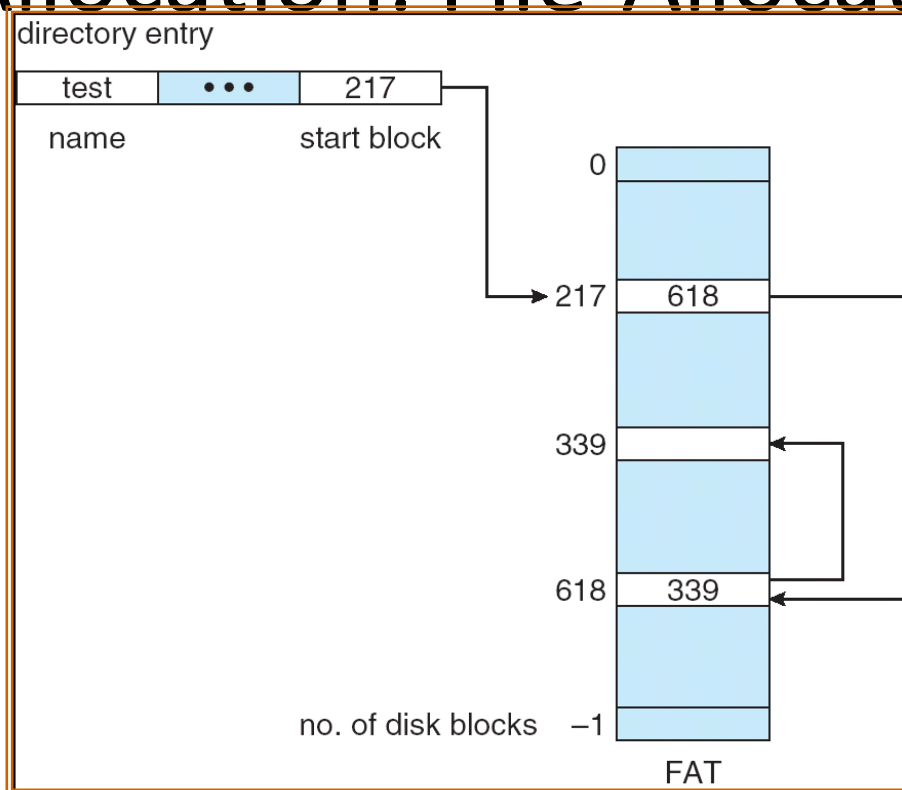
# Example: DOS FS (simplified)

- Uses linked files. Cute: links reside in fixed-sized "file allocation table" (FAT) rather than in the blocks.




  - Still do pointer chasing, but can cache entire FAT so can be cheap compared to disk access.

# Linked Allocation: File-Allocation Table




- MSDOS links pages together to create a file
  - Links not in pages, but in the File Allocation Table (FAT)
    - FAT contains an entry for each block on the disk
    - FAT Entries corresponding to blocks of file linked together
  - Access properties:
    - Sequential access expensive unless FAT cached in memory
    - Random access expensive always, but *really* expensive if FAT not cached in memory

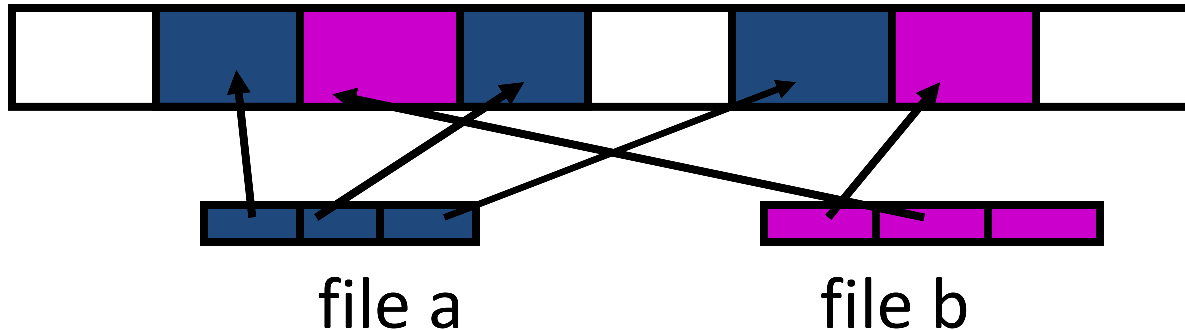
# FAT discussion

- Entry size = 16 bits.
  - What's the maximum size of the FAT?
  - Given a 512 byte block, what's the maximum size of FS?
  - Option: go to bigger blocks (called "Allocation Unit Size" at Format time). Pro? Con?

- Space overhead of FAT is trivial:
  - 2 bytes / 512 byte block = ~.4% (Compare to Unix)
- Reliability: how to protect against errors?
  - Create duplicate copies of FAT on disk.
  - State duplication a very common theme in reliability
- Bootstrapping: where is root directory?
  - Fixed location on disk / have a table in the bootsector (sector zero)

# Indexed files

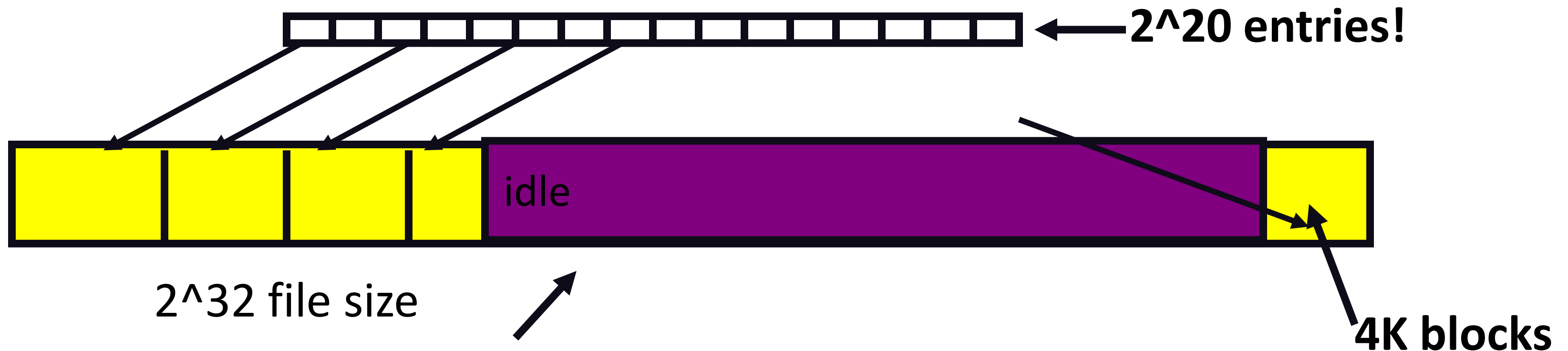
- Each file has an array holding all of it's block pointers
  - (purpose and issues = those of a page table)
  - max file size fixed by array's size (static or dynamic?)
  - create: allocate array to hold all file's blocks, but allocate on demand using free list



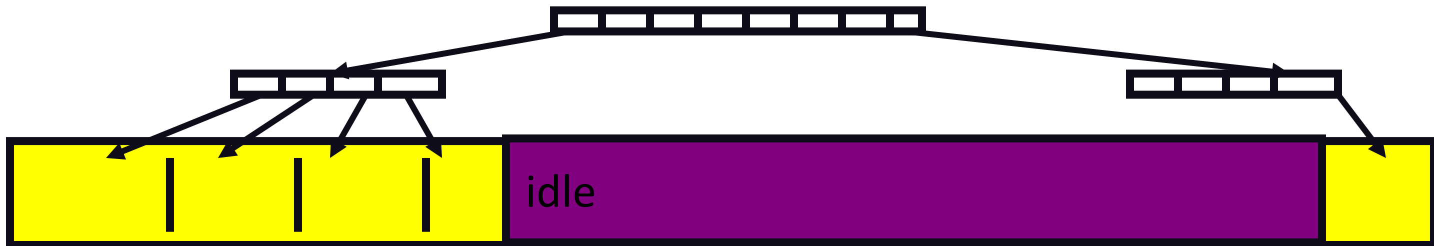

  - pro: both sequential and random access easy
  - con?

# Indexed files

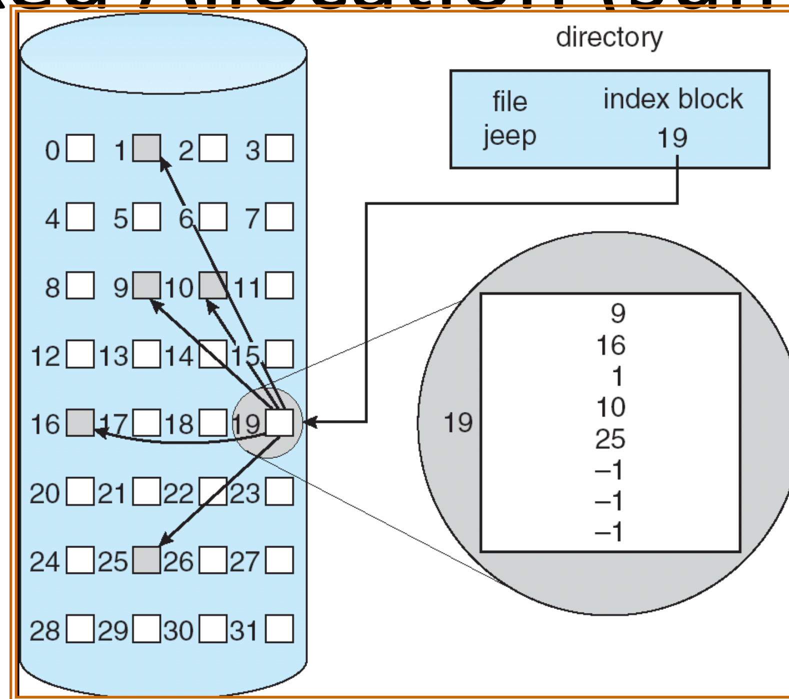
- Issues same as in page tables




  - Large possible file size = lots of unused entries
  - Large actual size? table needs large contiguous disk chunk
  - Solve identically: small regions with index array, this array with another array, …  Downside?

- 

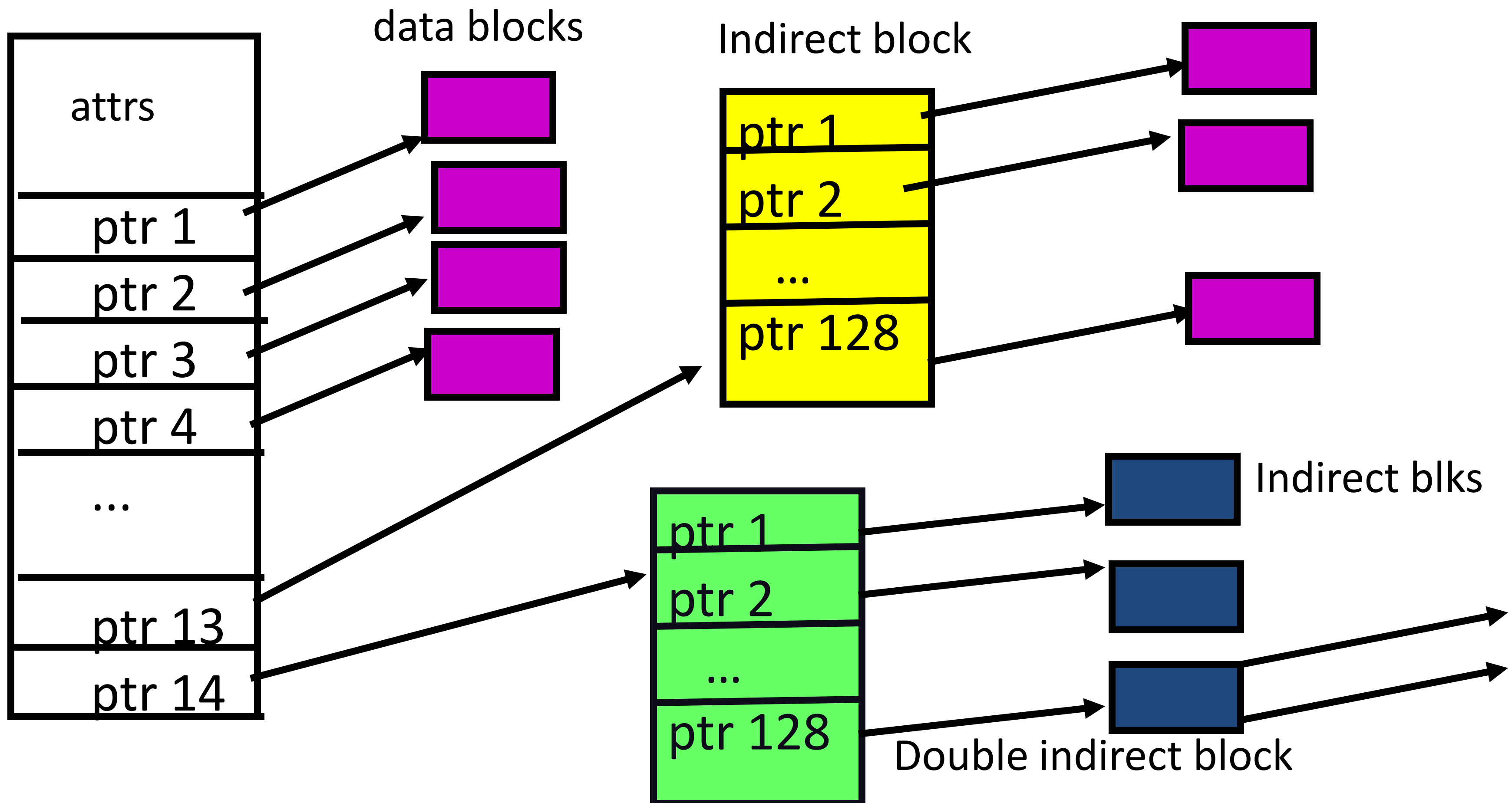
# Indexed Allocation (summary)




- Third Technique: Indexed Files (VMS)
  - System Allocates file header block to hold array of pointers big enough to point to all blocks
    - User pre-declares max file size;
  - Pros: Can easily grow up to space allocated for index
    Random access is fast
  - Cons: Clumsy to grow file bigger than table size
    Still lots of seeks: blocks may be spread over disk

# Multi-level indexed files: ~4.3 BSD

- File descriptor (inode) = 14 block pointers + attrs



# Unix discussion

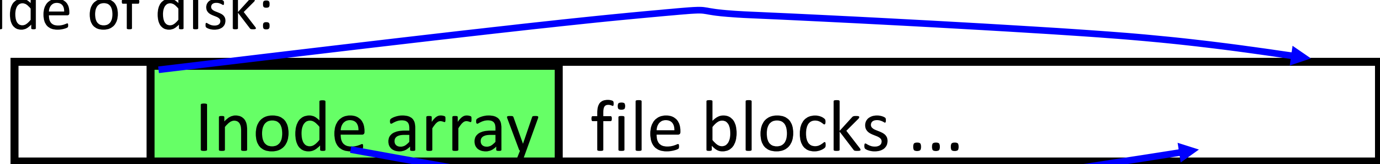
- Pro?
  - simple, easy to build, fast access to small files
  - Inode size?
  - Maximum file length fixed, but large. (With 4k blks?)

- Cons:
  - what's the worst case # of accesses?
  - What's some bad space overheads?

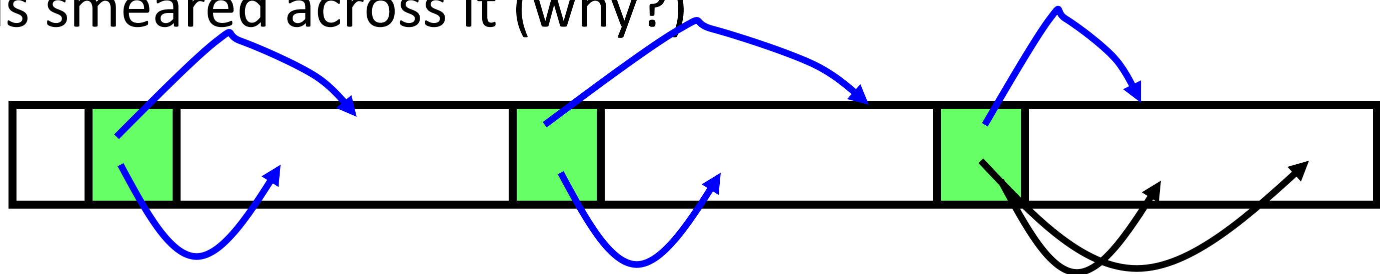
- An empirical problem:
  - because you allocate blocks by taking them off unordered freelist, meta data and data get strewn across disk

# More about inodes

- Inodes are stored in a fixed sized array
  - Size of array determined when disk is initialized and can't be changed. Array lives in known location on disk. Originally at one side of disk:



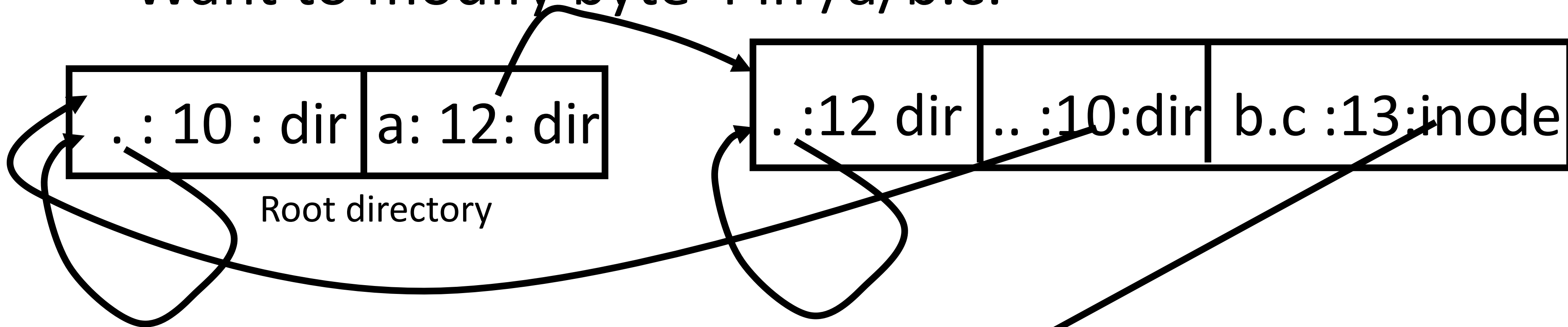

  - Now is smeared across it (why?)



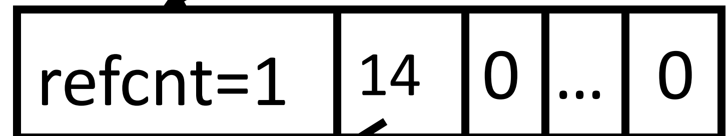
  - The index of an inode in the inode array called an i-number. Internally, the OS refers to files by inumber
  - When file is opened, the inode brought in memory, when closed, it is flushed back to disk.

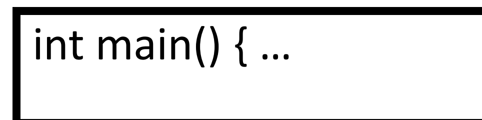
# Example: Unix file system

- Want to modify byte 4 in /a/b.c:




- readin root directory (blk 10)
- lookup a (blk 12); readin
- lookup inode for b.c (13); readin

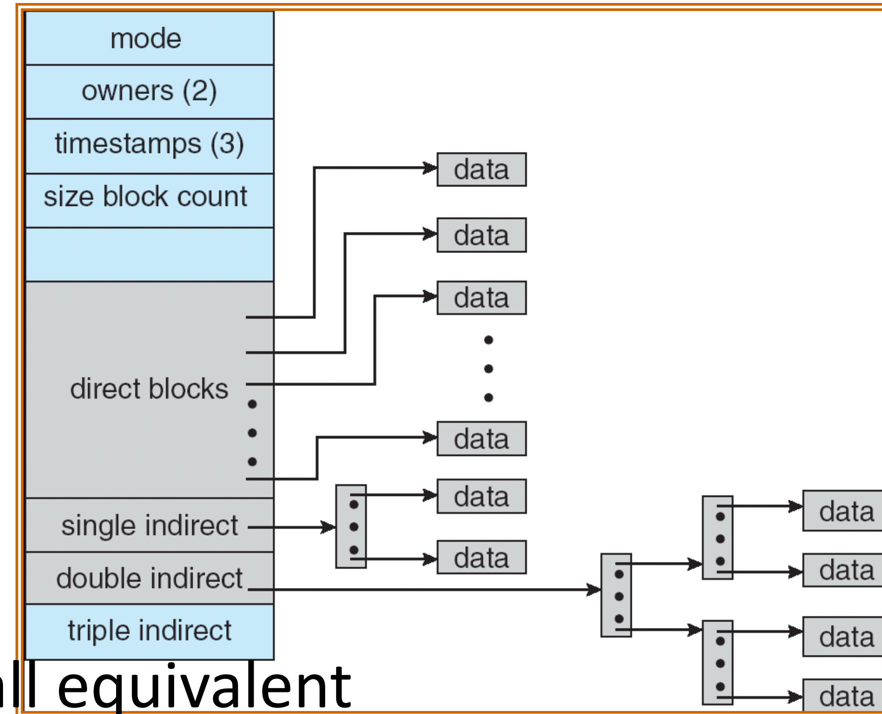



- use inode to find blk for byte 4 (blksize = 512, so offset = 0 gives blk 14); readin and modify

# Multilevel Indexed Files (UNIX 4.1)

- Multilevel Indexed Files: Like multilevel address translation (from UNIX 4.1 BSD)
  - Key idea: efficient for small files, but still allow big files

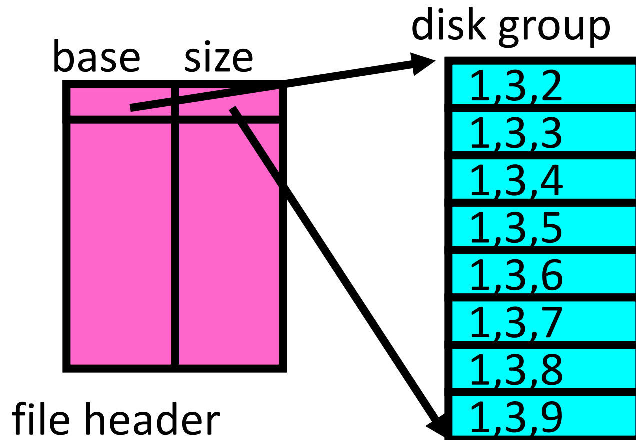



- File hdr contains 13 pointers
  - Fixed size table, pointers not all equivalent
  - This header is called an "inode" in UNIX
- File Header format:
  - First 10 pointers are to data blocks
  - Ptr 11 points to "indirect block" containing 256 block ptrs
  - Pointer 12 points to "doubly indirect block" containing 256 indirect block ptrs for total of 64K blocks
  - Pointer 13 points to a triply indirect block (16M blocks)

# Some important fields in a UNIX inode

- Mode
  - Protection info, file type (normal[-], directory[d], symbolic link[l]
- Owner
- Number of links
  - Number of directory entries that point to me
- Length (of file)
- Nblocks: number of blocks occupied on disk
- Array of 10 direct block pointers
- One indirect block pointer
- One doubly indirect block pointer
- One triply indirect block pointer

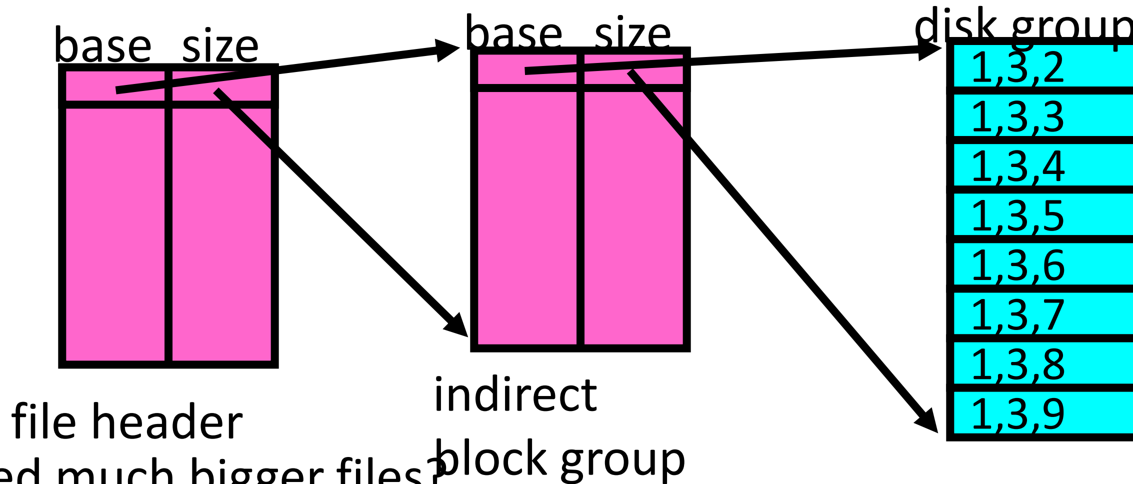
# File Allocation for Cray-1 DEMOS




Basic Segmentation Structure:
Each segment contiguous on disk

- DEMOS: File system structure similar to segmentation
  - Idea: reduce disk seeks by
    - using contiguous allocation in normal case
    - but allow flexibility to have non-contiguous allocation
  - Cray-1 had 12ns cycle time, so CPU:disk speed ratio about the same as today (a few million instructions per seek)
- Header: table of base & size (10 “block group” pointers)
  - Each block chunk is a contiguous group of disk blocks
  - Sequential reads within a block chunk can proceed at high speed – similar to continuous allocation
- How do you find an available block group?
  - Use freelist bitmap to find block of 0’s.

# Large File Version of DEMOS




- What if need much bigger files?
  - If need more than 10 groups, set flag in header: BIGFILE
    - Each table entry now points to an indirect block group
  - Suppose 1000 blocks in a block group ⇒ 80GB max file
    - Assuming 8KB blocks, 8byte entries⇒
      (10 ptrs×1024 groups/ptr×1000 blocks/group)*8K =80GB
- Discussion of DEMOS scheme
  - Pros: Fast sequential access, Free areas merge simply
    Easy to find free block groups (when disk not full)
  - Cons: Disk full ⇒ No long runs of blocks (fragmentation), so high overhead allocation/access

# How to keep DEMOS performing well?

- In many systems, disks are always full
  – How to fix? Announce that disk space is getting low, so please delete files? Have quotas.

- Solution:
  – Don’t let disks get completely full: reserve portion
    - Free count = # blocks free in bitmap
    - Scheme: Don’t allocate data if count < reserve
  – How much reserve do you need?
    - In practice, 10% seems like enough
  – Tradeoff: pay for more disk, get contiguous allocation
    - Since seeks so expensive for performance, this is a very good tradeoff

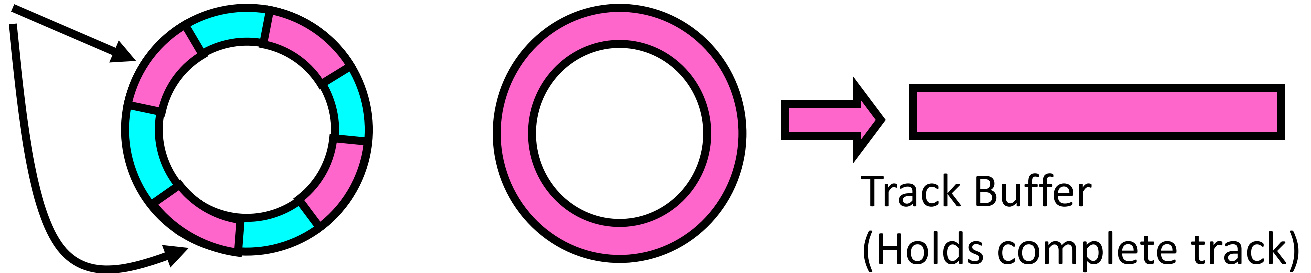
# UNIX BSD 4.2

- Same as BSD 4.1 (same file header and triply indirect blocks), except incorporated ideas from DEMOS:
  – Uses bitmap allocation in place of freelist
  – Attempt to allocate files contiguously
  – 10% reserved disk space
  – Skip-sector positioning (mentioned next slide)

- Problem: When create a file, don't know how big it will become (in UNIX, most writes are by appending)
  – How much contiguous space do you allocate for a file?
  – In Demos, power of 2 growth: once it grows past 1MB, allocate 2MB, etc
  – In BSD 4.2, just find some range of free blocks
    - Put each new file at the front of different range
    - To expand a file, you first try successive blocks in bitmap, then choose new range of blocks
  – Also in BSD 4.2: store files from same directory near each other

# Attack of the Rotational Delay

- Problem 2: Missing blocks due to rotational delay
  - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!




  - Solution1: Skip sector positioning ("interleaving")
    - Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
  - Solution2: Read ahead: read next block right after first, even if application hasn't asked for it yet.
    - This can be done either by OS (read ahead)
    - By disk itself (track buffers). Many disk controllers have internal RAM that allows them to read a complete track
  - Modern disk controllers: Track buffers, elevator algorithms, bad block filtering

# Protection

- Why have protection?
  - Because want to share but not share everything
  - Want protection on individual file and operation basis
- For convenience, create coarser grain concepts
  - All people in research group able to access files, others denied
  - Everybody should be able to read files in a directory
- Some Nouns
  - Operations: open, read, write, execute
  - Resources: files
  - Principals: users or processes
- Can describe desired protection using access matrix
  - Columns = principals
  - Rows = resources
  - Entry = operations allowed

# Two standard mechanisms for access control

- Access Lists
  - For each resource, give a list of principals allowed to access that resource and the access they are allowed
  - Access list = one row of access matrix
  - Instead of organizing on a principal-to-principal basis, can organize on a group basis
- Capabilities
  - For each resource and access operation, give out capabilities that give the holder the right to perform the operation on that resource
  - Capabilities must be unforgeable
  - Capability = column of access matrix
  - Can also organize capabilities on a group basis

# Directories

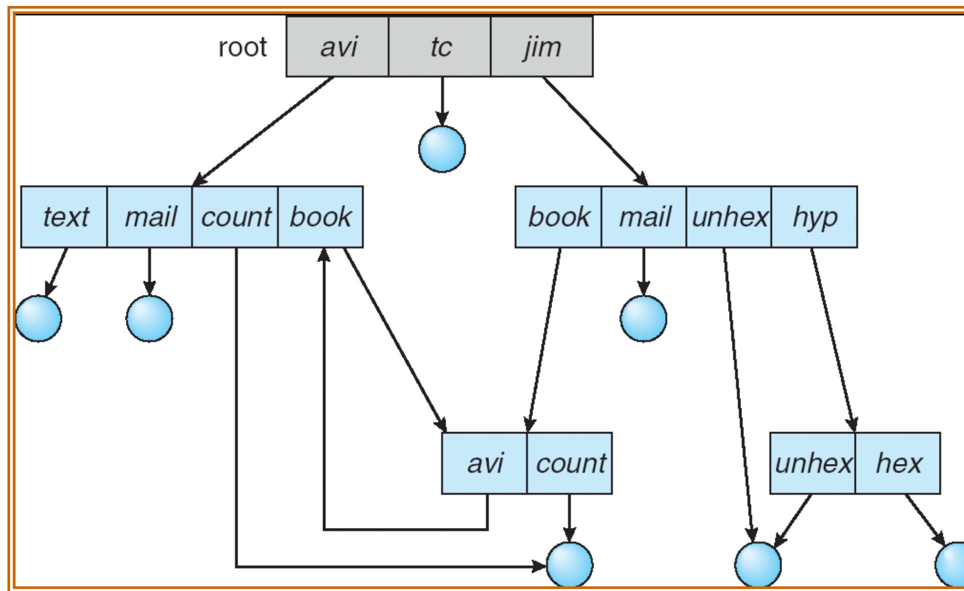
- Directory: a relation used for naming
  - Just a table of (file name, inumber) pairs
- How are directories constructed?
  - Directories often stored in files
    - Reuse of existing mechanism
    - Directory named by inode/inumber like other files
  - Needs to be quickly searchable
    - Options: Simple list or Hashtable
    - Can be cached into memory in easier form to search
- How are directories modified?
  - Originally, direct read/write of special file
  - System calls for manipulation: `mkdir`, `rmdir`
  - Ties to file creation/destruction
    - On creating a file by name, new inode grabbed and associated with new file in particular directory

# Directory Organization

- Directories organized into a hierarchical structure
  - Seems standard, but in early 70’s it wasn’t
  - Permits much easier organization of data structures

- Entries in directory can be either files or directories

- Files named by ordered set (e.g., /programs/p/list)




- Not really a hierarchy!
  - Many systems allow directory structure to be organized as an acyclic graph or even a (potentially) cyclic graph
  - Hard Links: different names for the same file
    - Multiple directory entries point at the same file
  - Soft Links: “shortcut” pointers to other files
    - Implemented by storing the logical name of actual file
- Name Resolution: The process of converting a logical name into a physical resource (like a file)
  - Traverse succession of directories until reach target file
  - Global file system: May be spread across the network

# How is a directory implemented?

- A file consisting of a list of (name, inode-number) pairs.
- UNIX:
  - Early UNIX:
    - Name : max 14 characters
    - Inode #: 2 bytes.
  - Current UNIX
    - Each directory entry also has the length of the name, so now name can be unbounded
- Ways to refer a file
  - Relative to current directory
  - Relative to root directory (where is root directory?)

# Directory Structure (Con't)

- How many disk accesses to resolve "/my/book/count"?
  - Read in file header for root (fixed spot on disk)
  - Read in first data block for root
    - Table of file name/index pairs.  Search linearly – ok since directories typically very small
  - Read in file header for "my"
  - Read in first data block for "my"; search for "book"
  - Read in file header for "book"
  - Read in first data block for "book"; search for "count"
  - Read in file header for "count"

- Current working directory: Per-address-space pointer to a directory (inode) used for resolving file names
  - Allows user to specify relative filename instead of absolute path (say CWD="/my/book" can resolve "count")