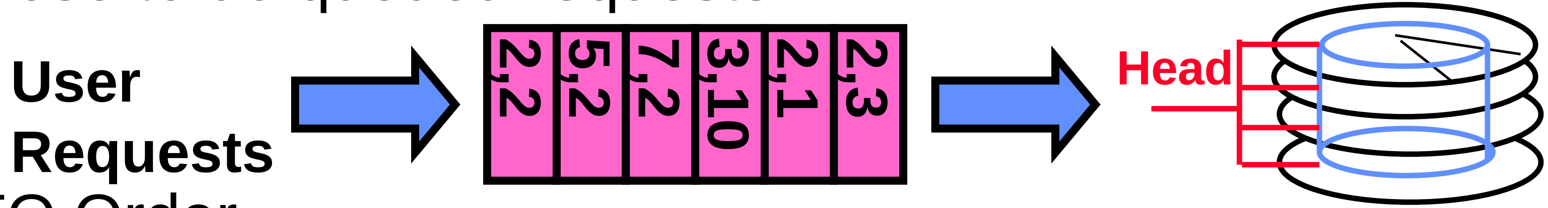
CSE150
Operating Systems
Lecture 20

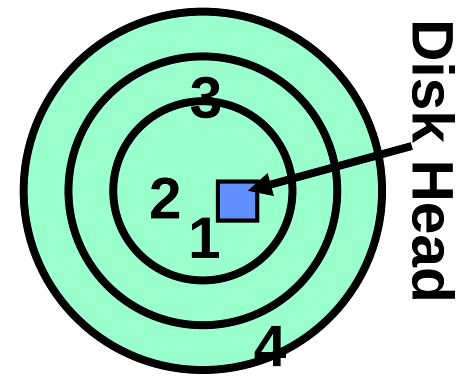
File Systems

Disk Scheduling (Review)

- Disk can do only one request at a time; What order do you choose to do queued requests?



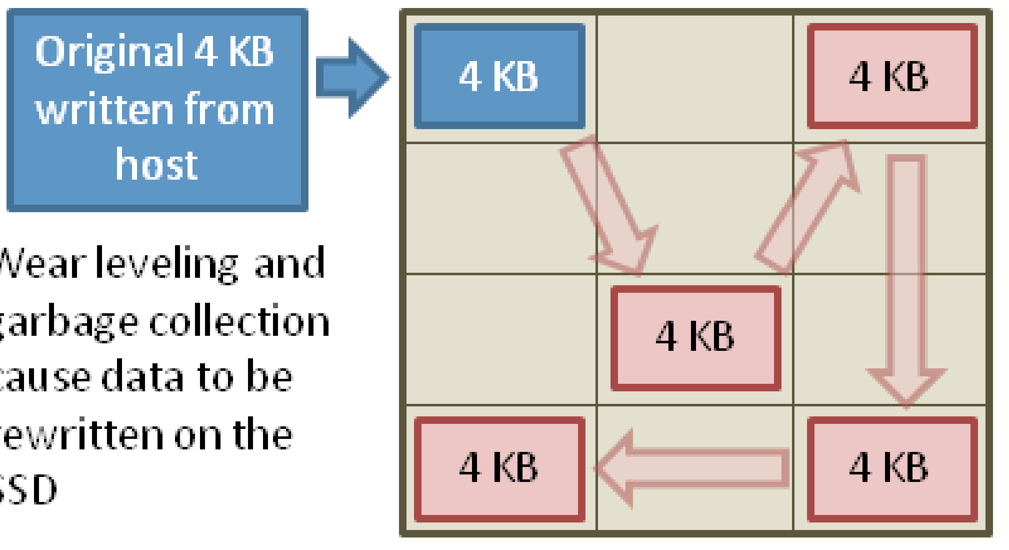
- FIFO Order
  – Fair among requesters, but order of arrival may be to random spots on the disk ⇒ Very long seeks
- SSTF: Shortest seek time first
  – Pick the request that's closest on the disk
  – Although called SSTF, today must include rotational delay in calculation, since rotation can be as long as seek
  – Con: SSTF good at reducing seeks, but may lead to starvation



- SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
  – No starvation, but retains flavor of SSTF
- C-SCAN: Circular-Scan: only goes in one direction
  – Skips any requests on the way back
  – Fairer than SCAN, not biased towards pages in middle

SSD Issues (Review)

- Writing data is complex! (~200μs – 1.7ms)
  - Can only write empty pages in a block
  - Erasing a block takes ~1.5ms
  - Controller maintains pool of empty blocks by coalescing used pages (read, erase, write).



Wear leveling and garbage collection cause data to be rewritten on the SSD

Storage Performance (Review)

- Hard (Magnetic) Disk Performance:
  - Latency = Queuing time + Controller + Seek + Rotational + Transfer
  - Rotational latency: on average ½ rotation
  - Depends on rotation speed and bit density

- SSD Performance:
  - Read: Queuing time + Controller + Transfer
  - Write: Queuing time + Controller (Find Free Block) + Transfer
  - Find Free Block time: depends on how full SSD is (available empty pages), write burst duration, …
  - Limited drive lifespan

Today

- File System Structures
- Naming and Directories
- File Caching, Durability, Authorization and Distributed 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: organizing 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.

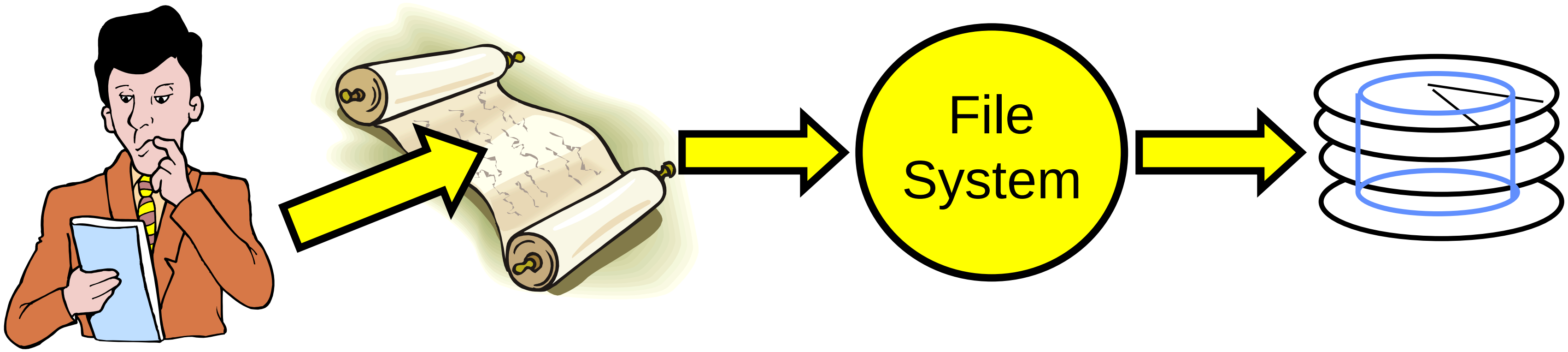
User vs. System View of a File

- 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 Linux, block size is 512 bytes to 32kB, and in Unix is 4kB

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()` ⇒ buffers something like 4096 bytes, even if interface is one byte at a time
- From now on, file is a collection of blocks

Disk Management Policies

- Basic entities on a disk:
  - File: user-visible group of blocks arranged sequentially in logical space
  - Directory: user-visible mapping of names to files (later)

- Access disk as linear array of sectors.
  - Logical Block Addressing (LBA): Every block has integer address from zero up to max number of sectors (LBA=0, LBA=1, …)
    - » Controller must deal with bad sectors (formerly OS/BIOS)
  - Controller translates from address $\Rightarrow$ physical position
    - » Hardware shields OS from structure of disk

Disk Management Policies (cont'd)

- 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

Designing the File System: Access Patterns

- Sequential Access: bytes read in order ("give me the next X bytes, then give me next, etc.")
  – Most of file accesses are of this flavor
- Random Access: read/write element out of middle of array ("give me bytes i—j")
  – Less frequent, but still important, e.g., memory page from swap file
  – Want this to be fast – don't want to have to read all bytes to get to the middle of the file
- Content-based Access: ("find me 100 bytes starting with ALICE")
  – Example: employee records – once you find the bytes, increase my salary by a factor of 2
  – Many systems don't provide this; instead, build DBs on top of disk access to index content (requires efficient random access)

Designing the File System: Usage Patterns

- Most files are small (for example, .login, .c, .java files)
  - A few files are big – executables, swap, .jar, core files, etc.; the .jar is as big as all of your .class files combined
  - However, most files are small – .class, .o, .c, .doc, .txt, etc
- Large files use up most of the disk space and bandwidth to/from disk
  - May seem contradictory, but a few enormous files are equivalent to an immense # of small files
- Although we will use these observations, beware!
  - Good idea to look at usage patterns: beat competitors by optimizing for frequent patterns
  - Except: changes in performance or cost can alter usage patterns.

File System Goals

- Maximize sequential performance
- Efficient random access to file
- Easy management of files (growth, truncation, etc)

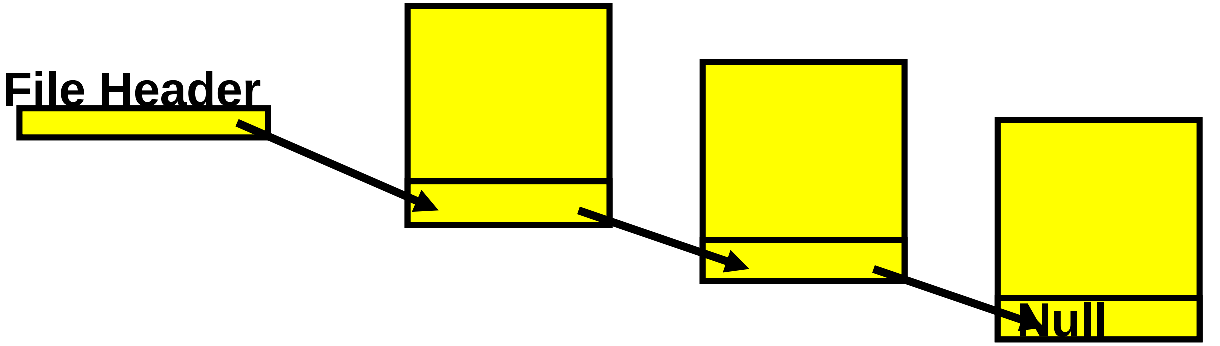
How to organize files on disk

- 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

- Continuous Allocation used by IBM 360
  – Result of allocation and management cost: People would create a big file, put their file in the middle

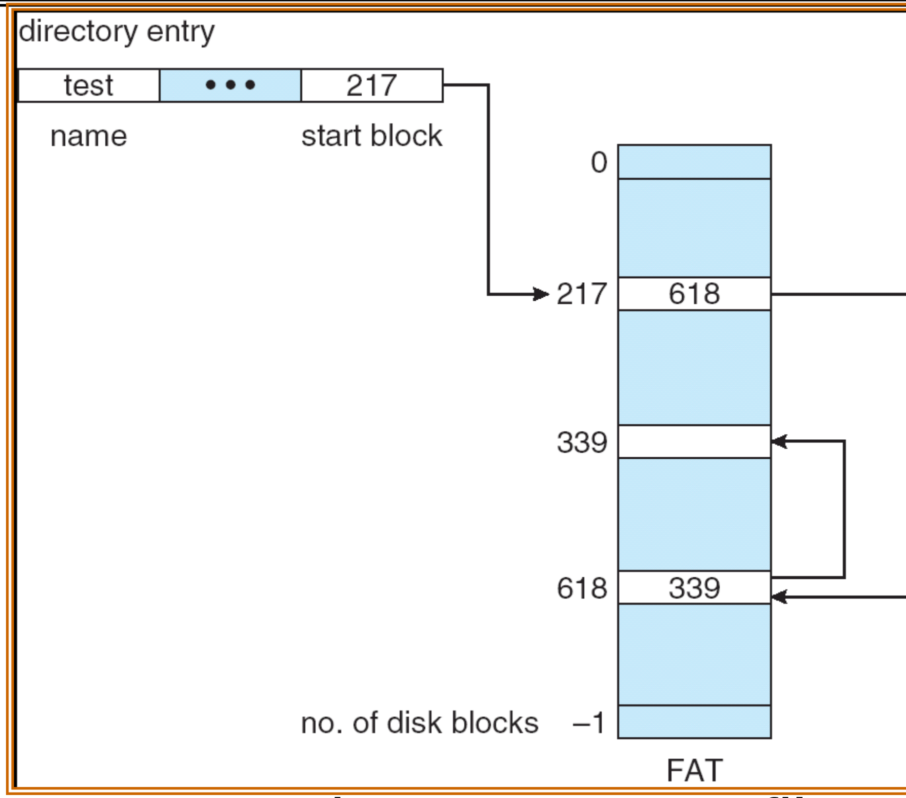
Linked List Allocation

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



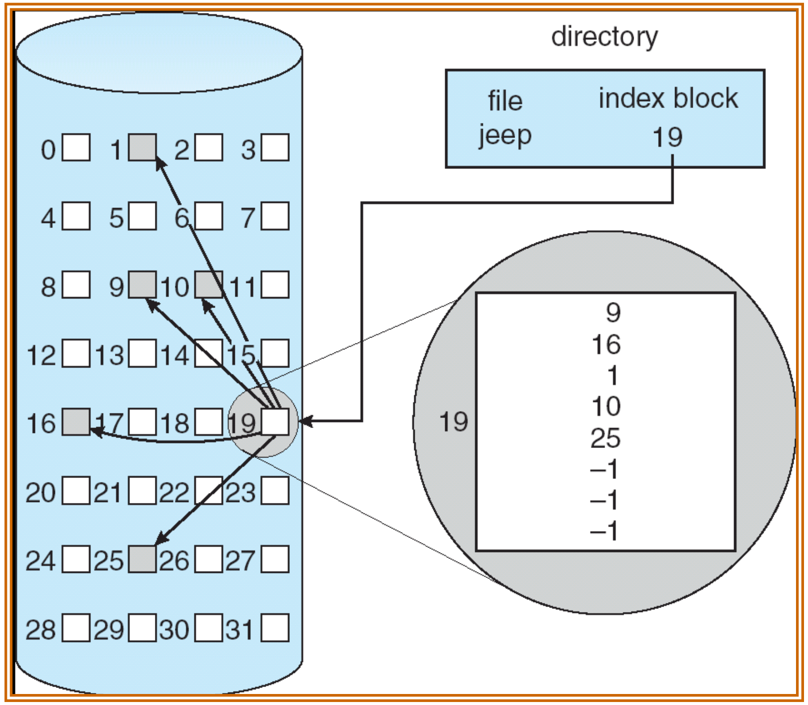
  – Pros: Can grow files dynamically, Free list same as file
  – Cons: Bad Sequential Access (seek between each block),
  Unreliable (lose block, lose rest of file)
  – Serious Con: Bad random access!!!!
  – Technique originally from Alto (First PC, built at Xerox)
    » No attempt to allocate contiguous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

Linked Allocation: File-Allocation Table (FAT)



- 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

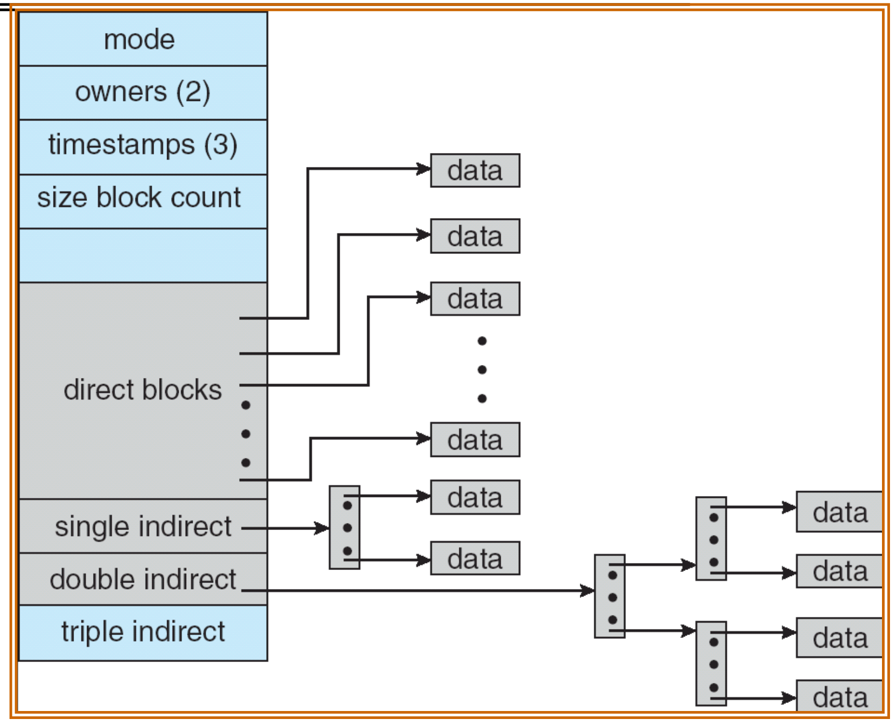
Indexed Allocation



- Third Technique: Indexed Files (Nachos, 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

Multilevel Indexed Files (UNIX 4.1)

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



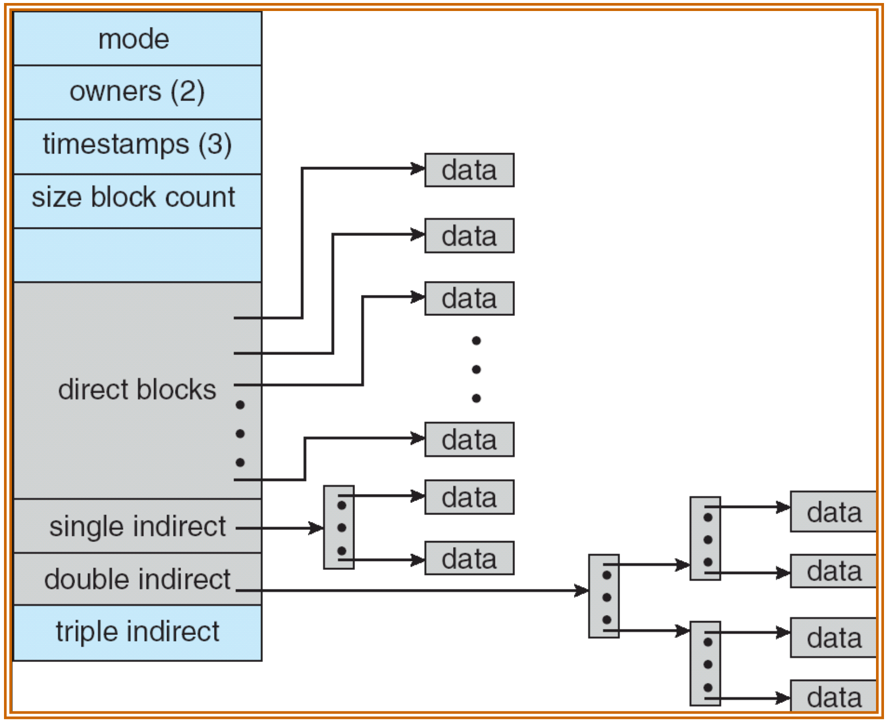
- File hdr contains 13 pointers
  – Fixed size table, pointers not all equivalent
  – This header is called an "inode" in UNIX
- File Header format:
  – First 10 pointers are to data blocks
  – Ptr 11 points to "(singly) indirect block" containing 256 block ptrs
  – Pointer 12 points to "doubly indirect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

Multilevel Indexed Files (UNIX 4.1): Discussion

- Basic technique places an upper limit on file size that is approximately 16Gbytes
  - Designers thought this was bigger than anything anyone would need. Much bigger than a disk at the time…
  - Fallacy: today, Facebook gets hundreds of TBs of logs every day!
- Pointers get filled in dynamically: need to allocate indirect block only when file grows > 10 blocks
  - On small files, no indirection needed

Example of Multilevel Indexed Files

- Sample file in multilevel indexed format:
  - How many accesses for block #5 (assume file header accessed on open)?
    - » One: One for data
  - How about block #23?
    - » Two: One for indirect block, one for data
  - Block #340?
    - » Three: double indirect block, indirect block, and data



- UNIX 4.1 Pros and cons
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks
    Very large files must read many indirect blocks (four I/O's per block!)

Summary

- File System:
  - Transforms blocks into Files and Directories
  - Optimize for access and usage patterns
  - Maximize sequential access, allow efficient random access
- File (and directory) defined by header, called “inode”
- Multilevel Indexed Scheme
  - Inode contains file info, direct pointers to blocks,
  - indirect blocks, doubly indirect, etc..