Operating Systems
(1DT020 & 1TT802)

Lecture 11

File system Interface (cont'd), Disk Management, File System Implementation

May 12, 2008

Léon Mugwaneza

http://www.it.uu.se/edu/course/homepage/os/vt08

Review : The file concept & File System

- ☛**File:** Collection of related information stored on a secondary storage
  - data files, program files (also, source, object, executable, ...).
- The structure of a file is determined by the user
  - sequence of bytes, lines, more complex (eg. object files, ..)
- File attributes: name, size, last update, owner, access rights, …
- File Operations: open, close, create, read, write, delete, ...

- ☛**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
- All information about a file contained in its file header
  - UNIX calls this an "inode", a global resource identified by index (inumber)
  - Once the header structure is loaded, all the other blocks of the file are locatable
- **Naming:** The process by which a system translates from user-visible names to system resources
  - User names files by textual names or icons, OS uses inumbers

Goals for Today

- File System Interface cont'd
- Disk management
- File System implementation

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne, others from Kubiatowicz - CS162 ©UCB Fall 2007 (University of California at Berkeley)

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

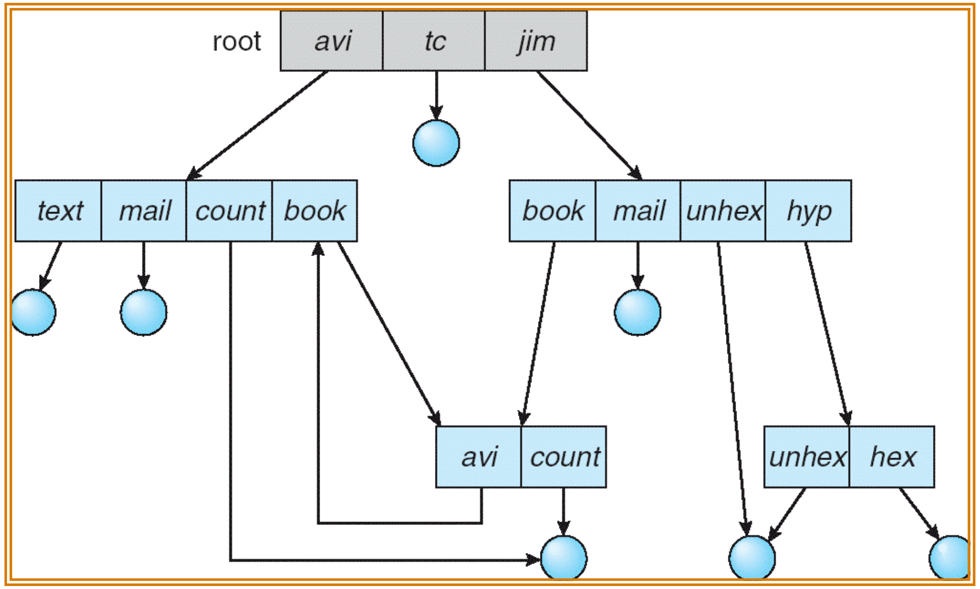
Directory Organization

- **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**

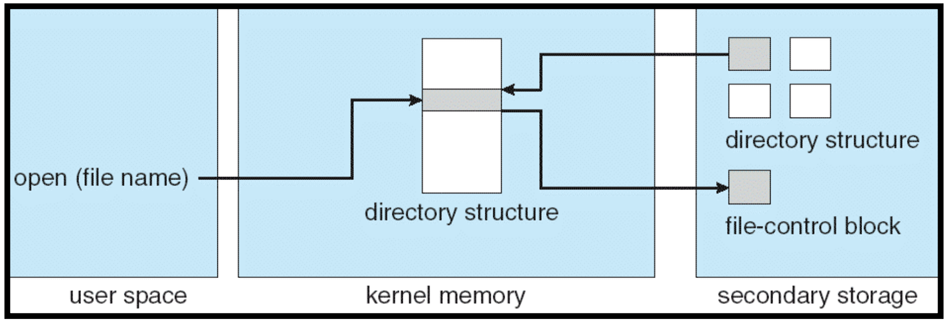
- **Files named by ordered set (e.g., /programs/p/list)**

Directory Structure

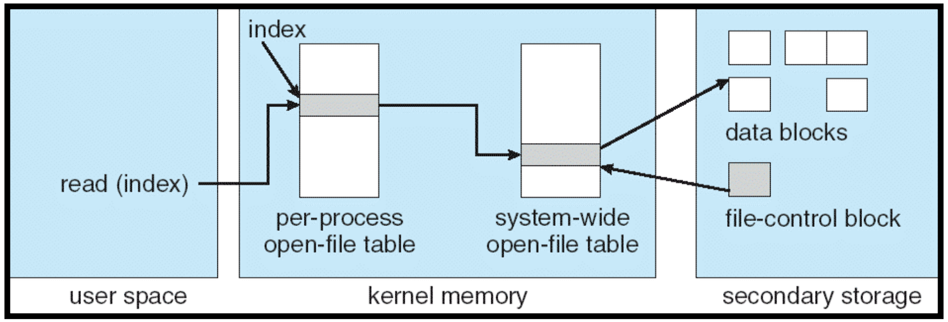


- Not really a hierarchy!
  - Many systems allow directory structure to be organized as an acyclic graph or even a (potentially) cyclic graph
  - Hard Links: different names for the same file
    - » Multiple directory entries point at the same file
  - Soft Links: "shortcut" pointers to other files
    - » 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

In-Memory File System Data Structures

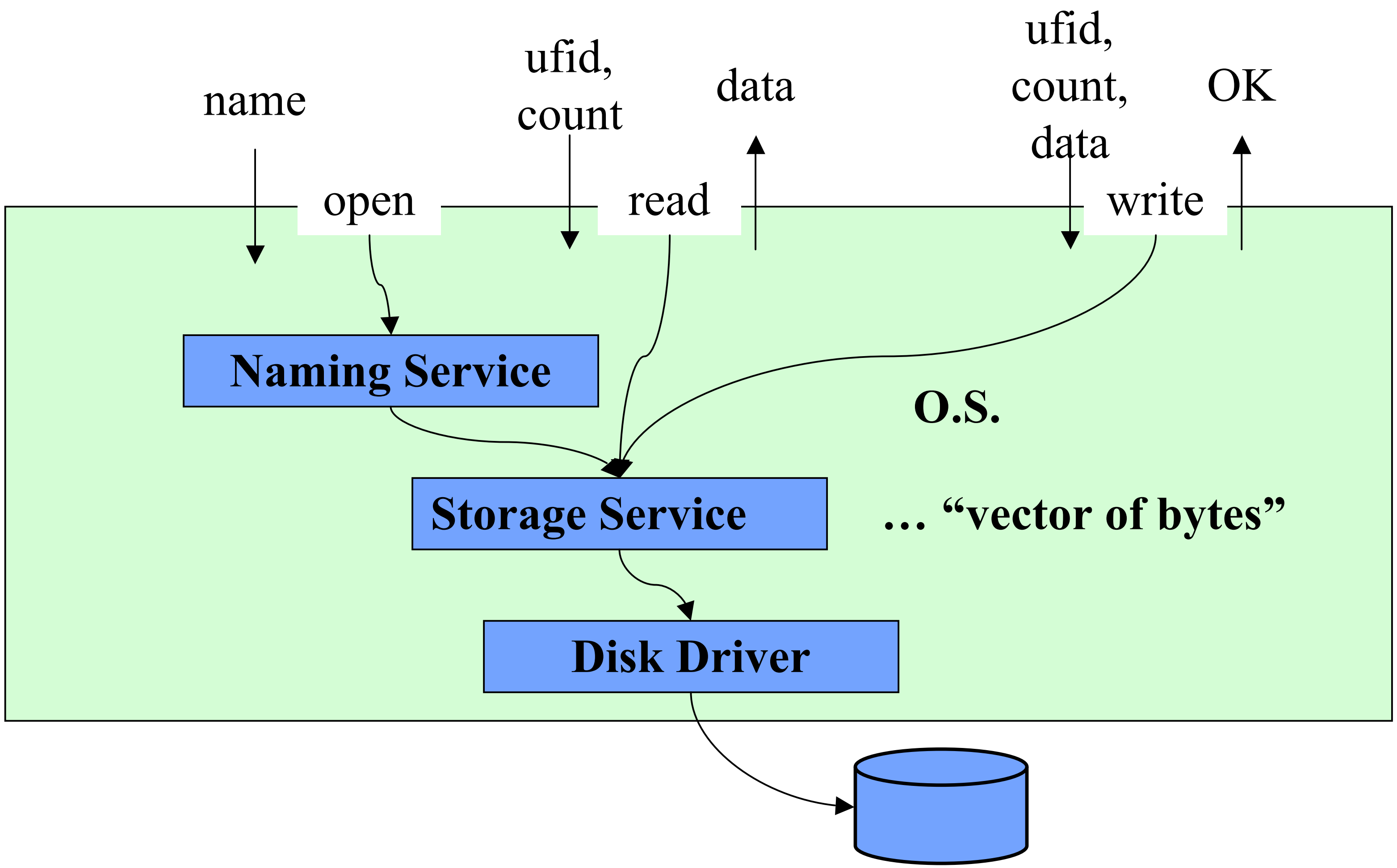


- **Open system call:**
  - **Resolves file name, finds file control block (inode)**
  - **Makes entries in per-process and system-wide tables**
  - **Returns index (called "file handle") in open-file table**



- **Read/write system calls:**
  - **Use file handle to locate inode**
  - **Perform appropriate reads or writes**

File System is Layered



Protection and Concurrency

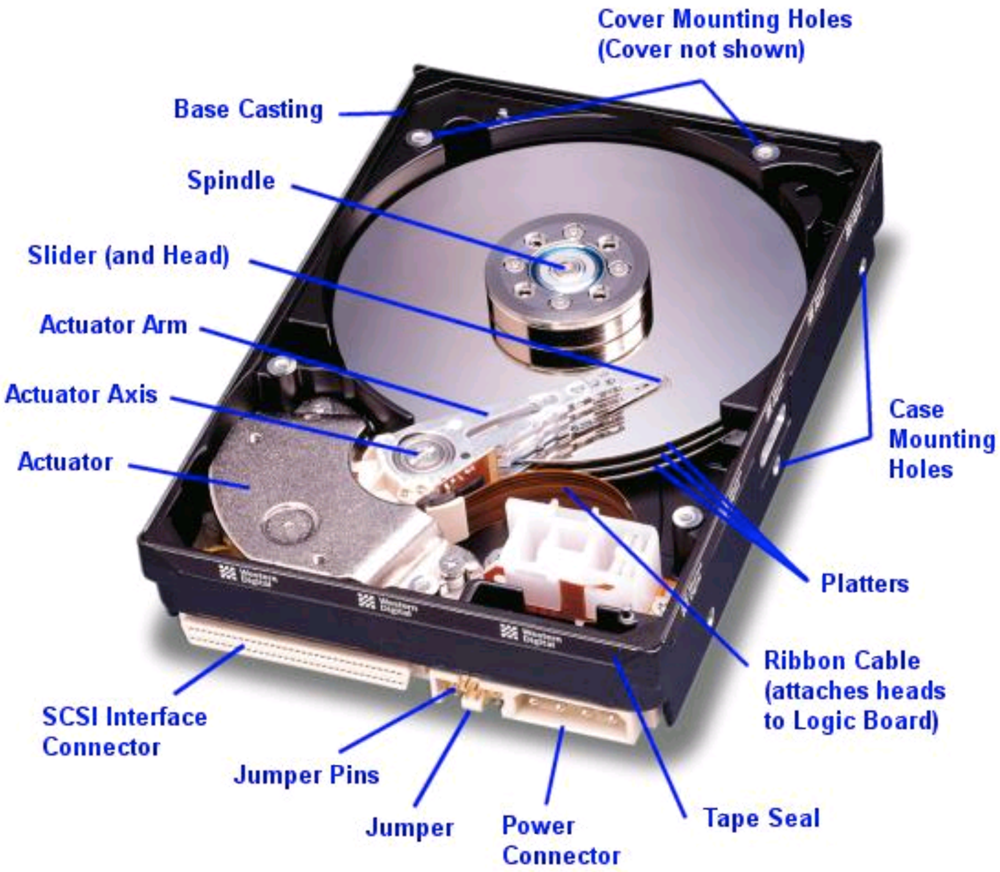
- **Any application can generate names independent of username**
  - **/etc/password**
  - **/lib/libc.a**
  - **/boot/vmlinuz-2.2.1**
- **Protection must be applied independently of naming**
  - **File owner should be able to control**
    - » **what can be done and by whom.**
  - **Types of access (eg, Unix: owner, group, public)**
- **Concurrency: how should multiple accesses be coordinated?**
  - **E.g., allow:**
    - » **either one writer**
    - » **or many readers**

Existence Control

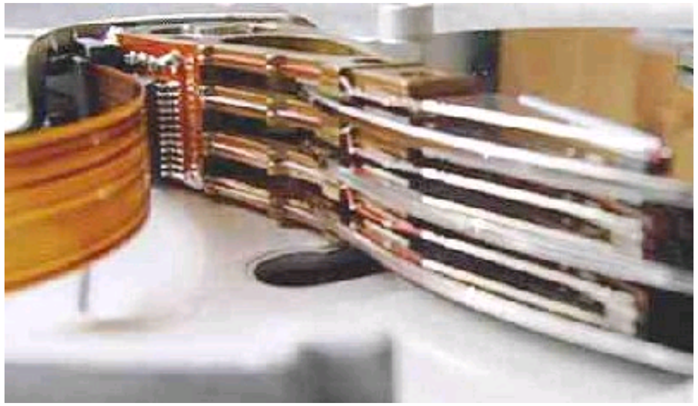
- File may have multiple names:
  - /etc/sendmail
  - /usr/bin/mailq
  - /root/bin/newaliases
- Any name may be deleted from directory

- When should file storage space be released?

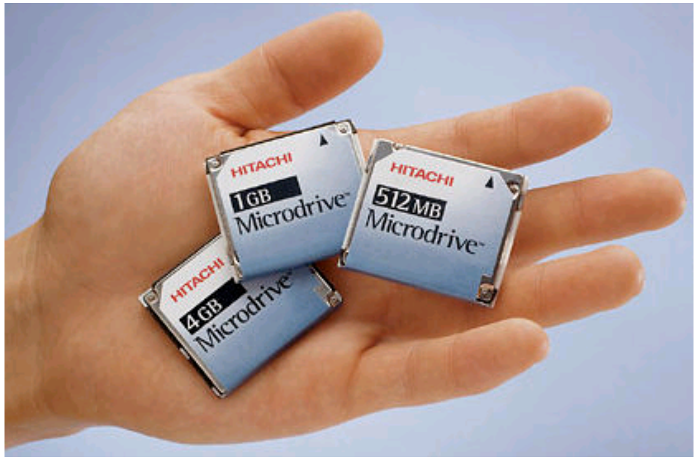
Hard Disk Drives



Western Digital Drive
http://www.storagereview.com/guide/

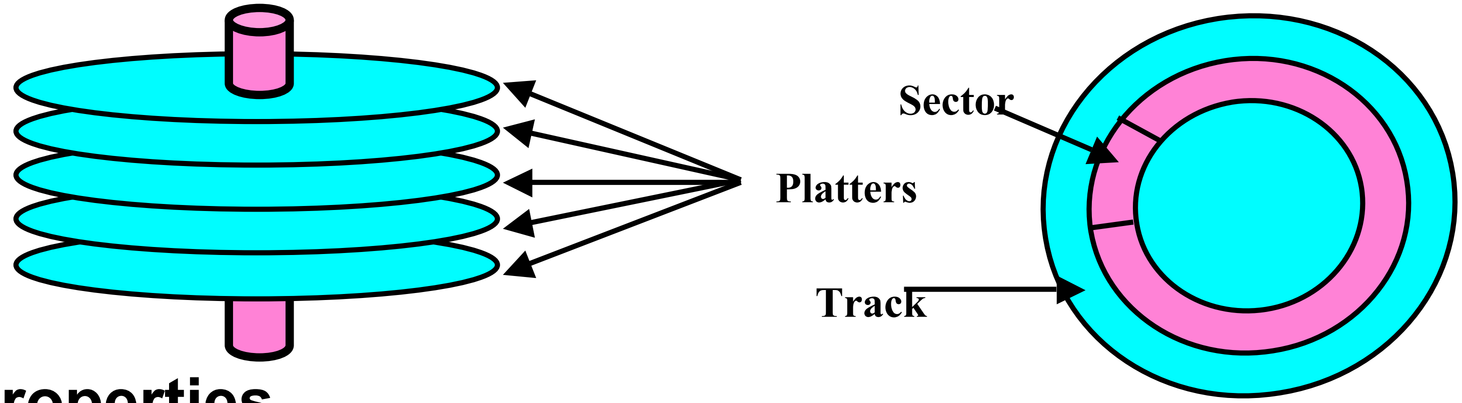


Read/Write Head
Side View



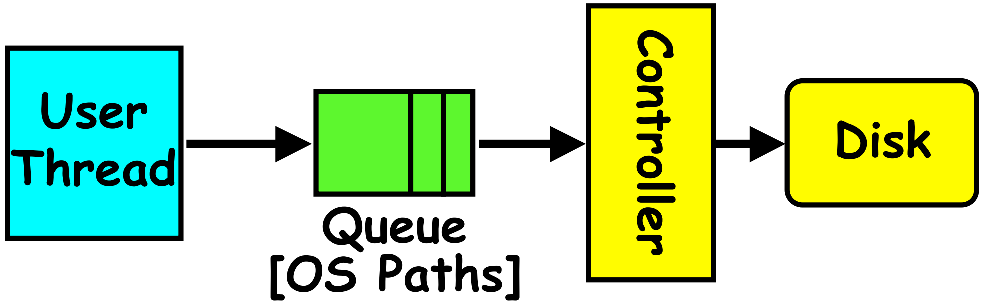
IBM/Hitachi Microdrive

Properties of a Hard Magnetic Disk

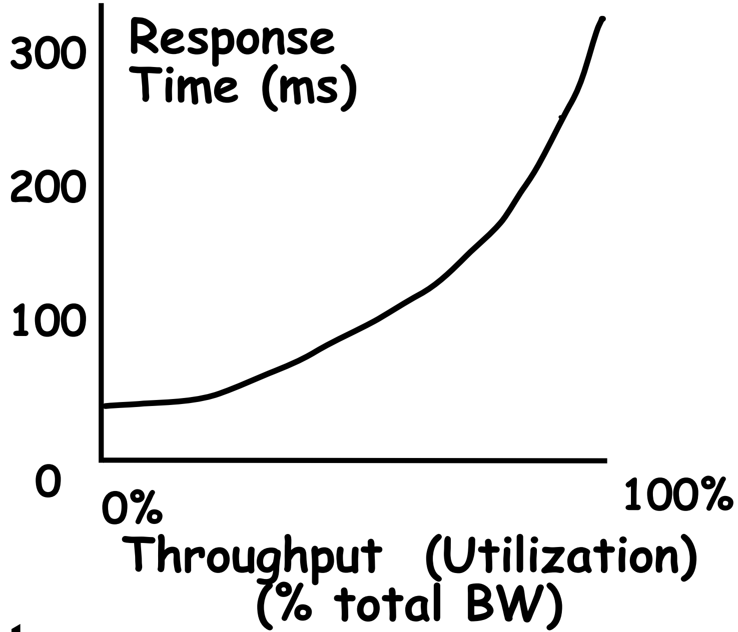


- **Properties**
  - Independently addressable element: sector
    - » OS always transfers groups of sectors together—"blocks"
  - A disk can access directly any given block of information it contains (random access). Can access any file either sequentially or randomly.
  - A disk can be rewritten in place: it is possible to read/modify/write a block from the disk
- **Typical numbers (depending on the disk size):**
  - 500 to more than 20,000 tracks per surface
  - 32 to 800 sectors per track
    - » A sector is the smallest unit that can be read or written
- **Zoned bit recording**
  - Constant bit density: more sectors on outer tracks
  - Speed varies with track location

Disk I/O Performance

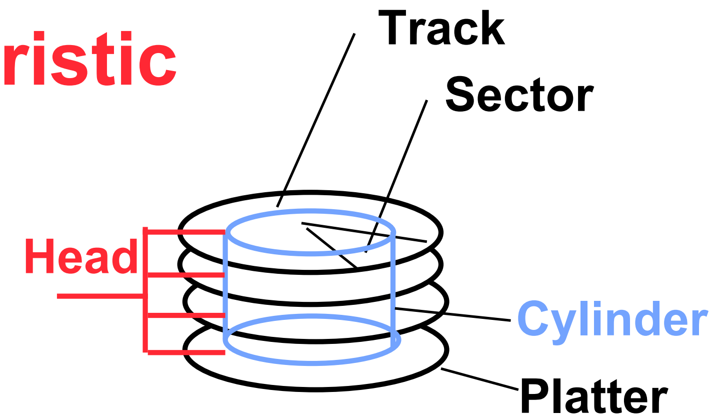


Response Time = Queue+Disk Service Time

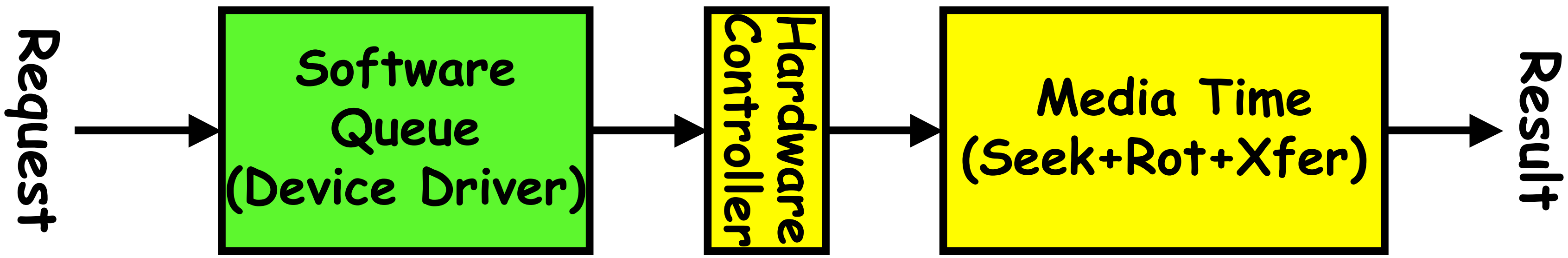


- **Performance of disk drive/file system**
  - Metrics: Response Time, Throughput
  - Contributing factors to latency:
    - » Software paths (can be loosely modeled by a queue)
    - » Hardware controller
    - » Physical disk media
- **Queuing behavior:**
  - Can lead to big increases of latency as utilization approaches 100%

Magnetic Disk Characteristic



- Cylinder: all the tracks under the head at a given point on all surfaces
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track (into proper cylinder)
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head
- Disk Latency = Queueing Time + Controller time +
- Seek Time + Rotation Time + Xfer Time



- Highest Bandwidth:
  - Transfer large group of blocks sequentially from one track

Typical Numbers of a Magnetic Disk

- **Average seek time as reported by the industry:**
  - Typically in the range of 8 ms to 12 ms
  - Due to locality of disk reference may only be 25% to 33% of the advertised number
- **Rotational Latency:**
  - *Most* disks rotate at 3,600 to 7200 RPM (Up to 15,000RPM or more)
  - Approximately 16 ms to 8 ms per revolution, respectively
  - An average latency to the desired information is halfway around the disk: 8 ms at 3600 RPM, 4 ms at 7200 RPM
- **Transfer Time is a function of:**
  - Transfer size (usually a sector): 512B – 1KB per sector
  - Rotation speed: 3600 RPM to 15000 RPM
  - Recording density: bits per inch on a track
  - Diameter: ranges from 1 in to 5.25 in
  - Typical values: 2 to 50 MB per second
- **Controller time depends on controller hardware**
- **Cost drops by factor of two per year (since 1991)**

Disk Performance Examples

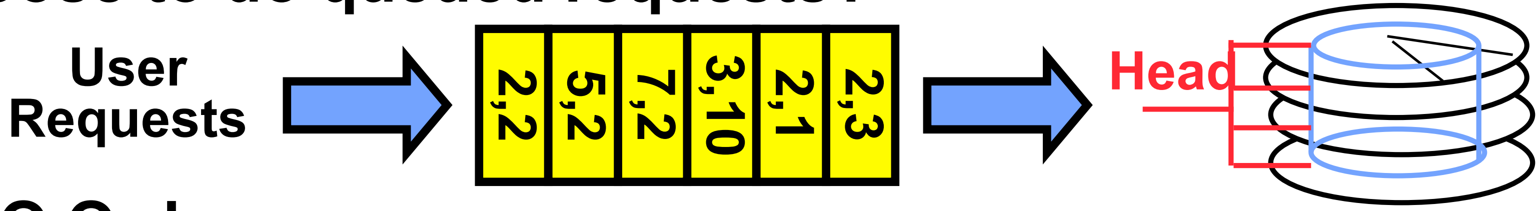
- **Assumptions:**
  - Ignoring queuing and controller times for now
  - Avg seek time of 5ms,
  - 7200RPM ⇒ Time for one rotation: ≈8 ms
  - Transfer rate of 4MByte/s, sector size of 1 KByte
- **Read sector from random place on disk:**
  - Seek (5ms) + Rot. Delay (4ms) + Transfer (0.25ms)
  - Approx 10ms to fetch/put data: 100 KByte/sec
- **Read sector from random place in same cylinder:**
  - Rot. Delay (4ms) + Transfer (0.25ms)
  - Approx 5ms to fetch/put data: 200 KByte/sec
- **Read next sector on same track:**
  - Transfer (0.25ms): 4 MByte/sec
- **Key to using disk effectively (esp. for filesystems) is to minimize seek and rotational delays**

Disk Tradeoffs

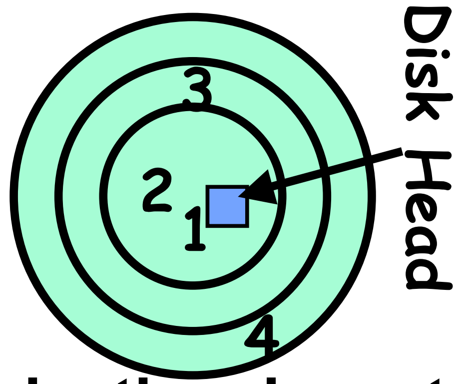
- How do manufacturers choose disk sector sizes?
  - Need 100-1000 bits between each sector to allow system to measure how fast disk is spinning and to tolerate small (thermal) changes in track length
- What if sector was 1 byte?
  - Space efficiency – only 1% of disk has useful space
  - Time efficiency – each seek takes 10 ms, transfer rate of 50 – 100 Bytes/sec
- What if sector was 1 KByte?
  - Space efficiency – only 90% of disk has useful space
  - Time efficiency – transfer rate of 100 KByte/sec
- What if sector was 1 MByte?
  - Space efficiency – almost all of disk has useful space
  - Time efficiency – transfer rate of 4 MByte/sec

Disk Scheduling

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

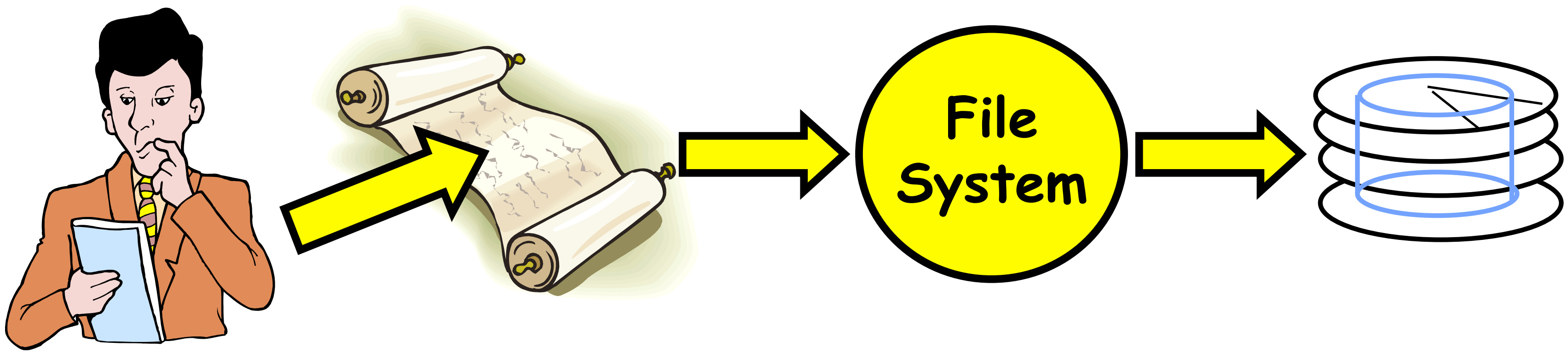


- **FIFO Order**
  - Fair among requesters, but order of arrival may be to random spots on the disk ⇒ Very long seeks
- **SSTF: Shortest seek time first**
  - Pick the request that's closest on the disk
  - Although called SSTF, today must include rotational delay in calculation, since rotation can be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

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**

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
- **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 ⇒ physical position
    - » First case: OS/BIOS must deal with bad sectors
    - » Second case: hardware shields OS from structure of disk
- **Need way to track free disk blocks**
  - Link free blocks together ⇒ too slow today
  - Use bitmap to represent free space on disk
- **File Header: a way to structure files**
  - 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

- **How do users access files?**
  - Need to know type of access patterns user is likely to throw at system
- **Sequential Access: bytes read in order ("give me the next X bytes, then give me next, etc")**
  - Almost all file access are of this flavor
- **Random Access: read/write element out of middle of array ("give me bytes i—j")**
  - Less frequent, but still important. For example, virtual memory backing file: page of memory stored in 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 Alpha")**
  - Example: employee records – once you find the bytes, increase my salary by a factor of 2
  - Many systems don't provide this; instead, databases are built on top of disk access to index content (requires efficient random access)

Designing the File System: Usage Patterns

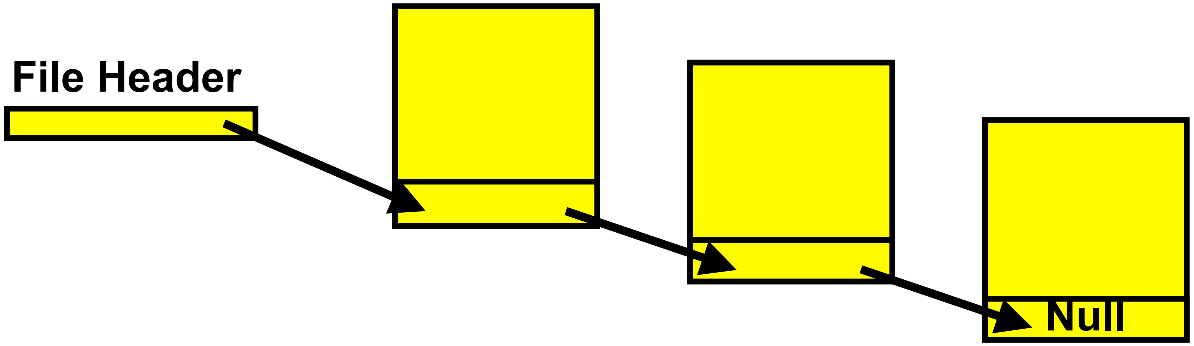
- **Most files are small (for example, .login, .c files)**
  - A few files are big – core files, etc.; executable are as big as all of linked object modules and statically linked library functions combined
  - However, most files are small – .java, .class's, .o's, .c's, etc.
- **Large files use up most 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 usage patterns:**
  - Good idea to look at usage patterns: beat competitors by optimizing for frequent patterns
  - Except: changes in performance or cost can alter usage patterns. Maybe UNIX has lots of small files because big files are really inefficient?
- **Digression, danger of predicting future:**
  - In 1950's, marketing study by IBM said total worldwide need for computers was 7!

How to organize files on disk

- **Goals:**
  - Maximize sequential performance
  - Easy random access to file
  - Easy management of file (growth, truncation, etc)
- **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

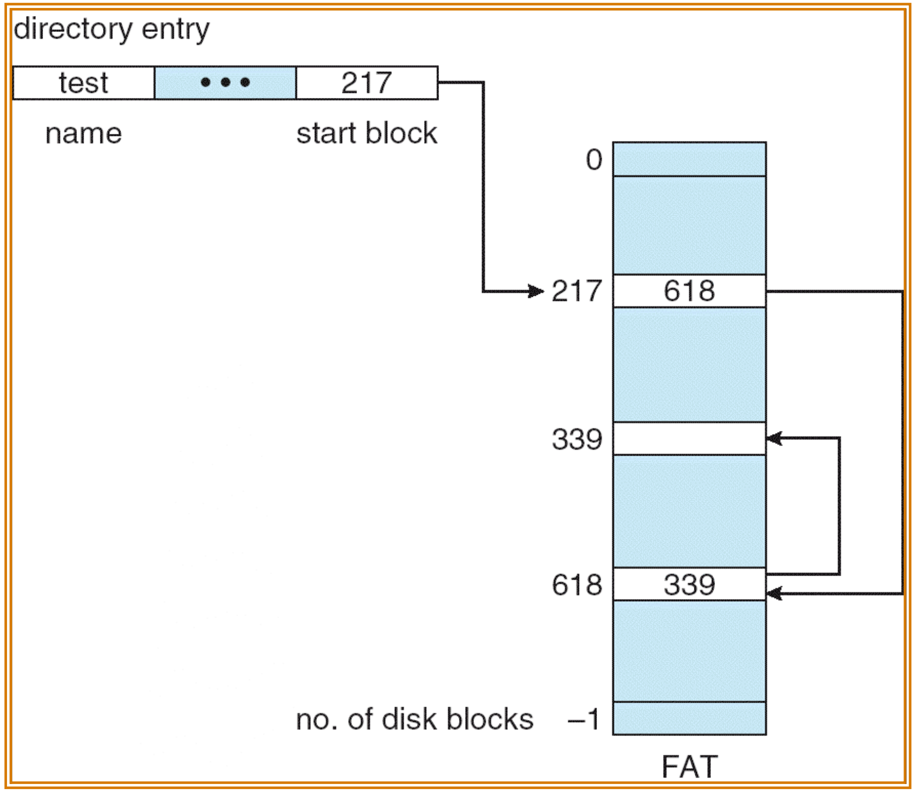
Linked List Allocation

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



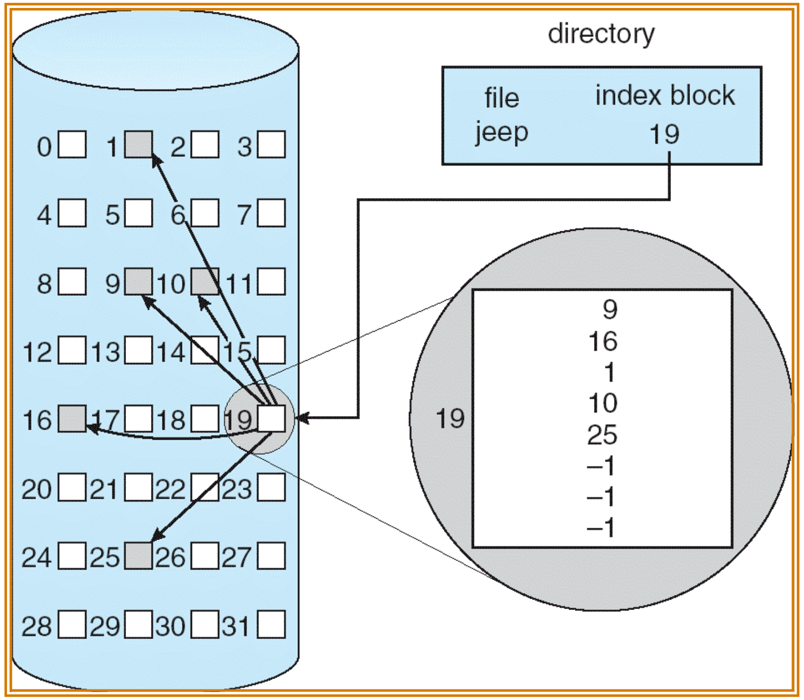
  - Pros: Can grow files dynamically, Free list same as file
  - Cons: Bad Sequential Access (seek between each block), Unreliable (lose block, lose rest of file)
  - Serious Con: Bad random access!!!!
  - Technique originally from Alto (First PC, built at Xerox)
    - » No attempt to allocate contiguous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

Linked Allocation: File-Allocation Table (FAT)



- **MSDOS links blocks together to create a file**
  - **Links not in blocks, 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**

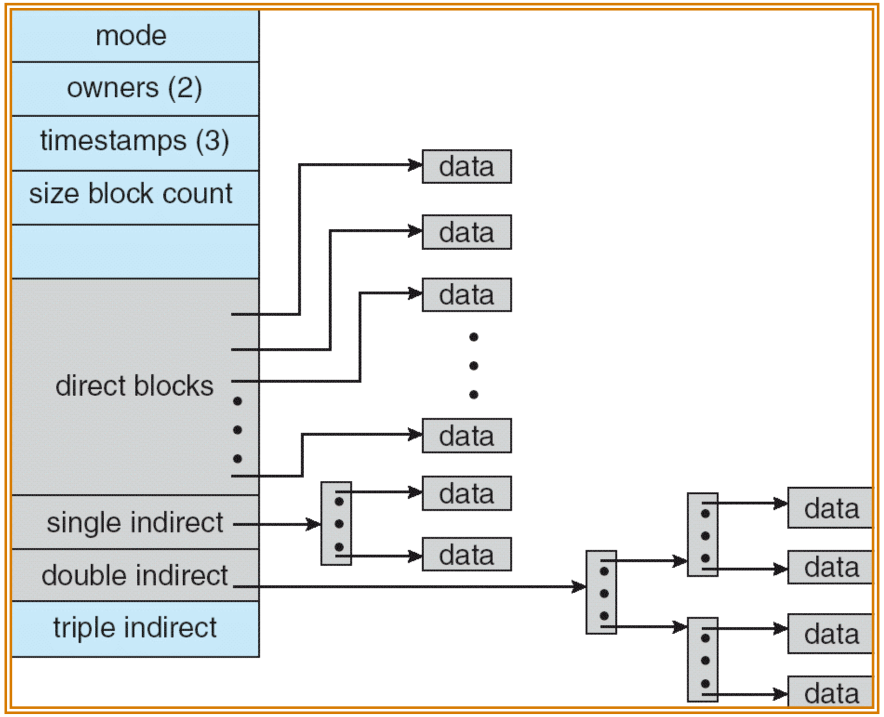
Indexed Allocation



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



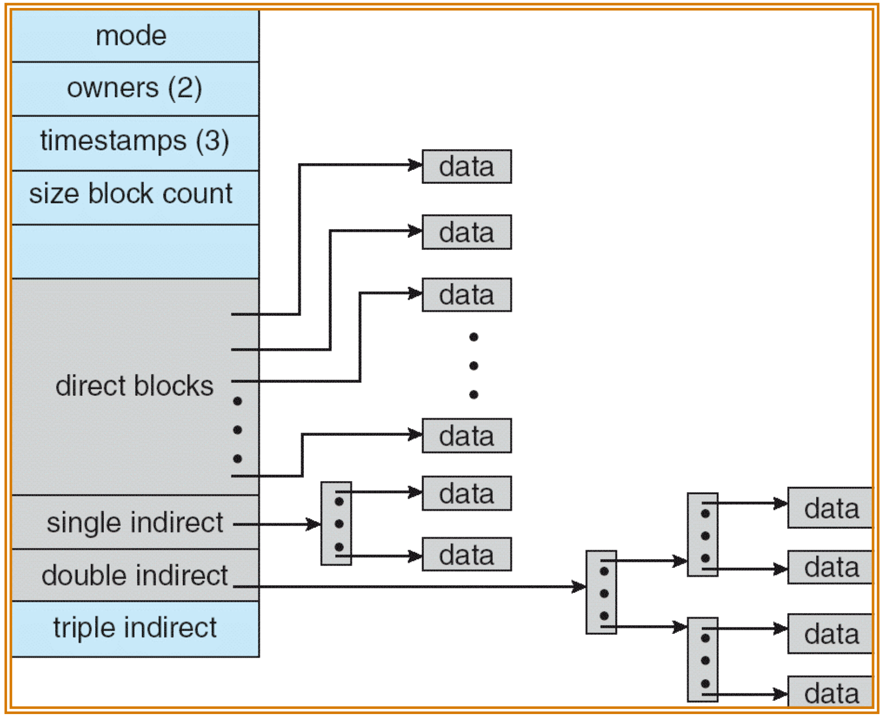
- **File hdr contains 13 pointers**
  – **Fixed size table, pointers not all equivalent**
  – **This header is called an "inode" in UNIX**
- **File Header format:**
  – **First 10 pointers are to data blocks**
  – **Ptr 11 points to "indirect block" containing 256 block ptrs**
  – **Pointer 12 points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

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, EOS producing 2TB of data per 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 #23? (assume file header accessed on open)?**
    - » **Two: One for indirect block, one for data**
  - **How about block #5?**
    - » **One: 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/Os per block!)

Summary

- naming service (how do users select files?)
  - Directories are used for naming
  - A file can have several names
- Protection, concurrency control, existence control
  - from unauthorised access: all users are not equal!)
  - File sharing control
  - When is the file storage space released?
- File (and directory) defined by header
  - Called “inode” with index called “inumber”
- Disk Performance:
  - Queuing time + Controller + Seek + Rotational + Transfer
  - Rotational latency: on average ½ rotation
  - Transfer time: spec of disk depends on rotation speed and bit storage density
- File System:
  - Transforms blocks into Files and Directories
  - Optimize for access and usage patterns
  - Maximize sequential access, allow efficient random access
- Multilevel Indexed Scheme
  - Inode contains file info, direct pointers to blocks,
  - indirect blocks, doubly indirect, etc..