

# CSC 369

Week 10:  
Disk I/O and File System Optimization  
Reading: Text, Ch. 5.4

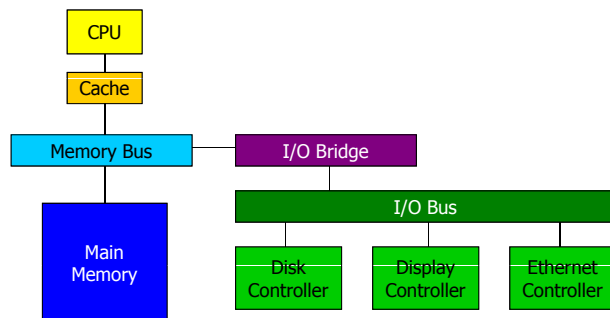


## Overview

- Last time:
  - File systems - Interface, structure, basic implementation
- Today:
  - Magnetic disks as secondary storage
  - Disk management
  - File system optimizations
- Assignment 2 Due Monday
- Assignment 3 Out Tuesday



## I/O Diagram



## Secondary Storage

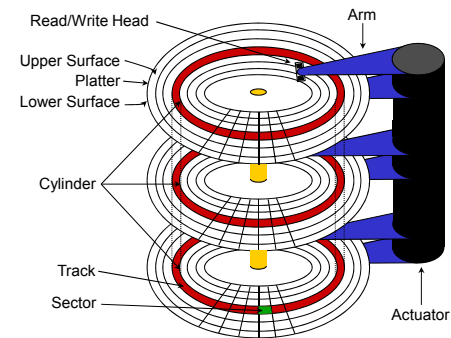
- Secondary storage is usually:
  - Anything outside of "primary memory"
  - Anything that does not permit direct instruction execution or data fetch via machine load/store instructions
- Characteristics
  - It's large - hundreds of megabytes, gigabytes, terabytes
  - It's cheap - 200 GB SATA disks costs <\$100 USD
  - It's persistent - data survives loss of power
  - It's slow - milliseconds to access (why is a millisecond slow?)



## Secondary Storage Devices

- Drums
  - Ancient history
- Magnetic disks
  - Fixed
  - Removable (floppy)
- Optical disks
  - Write-once, read-many (CD-R, DVD-R)
  - Write-many, ready-many (CD-RW)
- We're going to focus on the use of fixed (hard) magnetic disks for implementing secondary storage

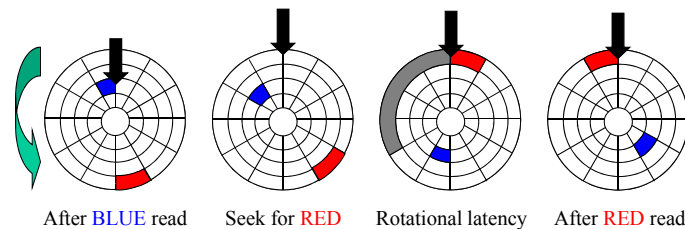
## Disk Components



## Disk Performance

- Disk request performance depends upon a number of steps
  - Seek - moving the disk arm to the correct cylinder
    - Depends on how fast disk arm can move (increasing very slowly)
  - Rotation - waiting for the sector to rotate under the head
    - Depends on rotation rate of disk (increasing, but slowly)
  - Transfer - transferring data from surface into disk controller electronics, sending it back to the host
    - Depends on density (increasing quickly)
- When the OS uses the disk, it tries to minimize the cost of all of these steps
  - Particularly seeks and rotation

## Traditional service time components



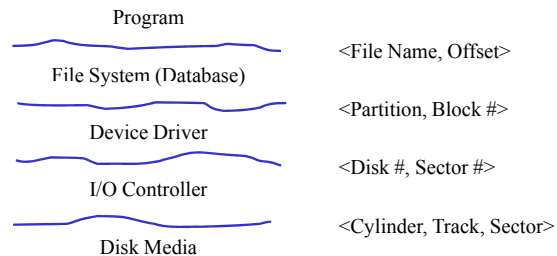
## Modern disk performance characteristics

- **Seek times:** 1-15ms, depending on distance
  - average 5-6ms
  - improving at 7-10% per year
- **Rotation speeds:**
  - 5600-7200 RPMs for cheap IDE disks
  - 10,000-15,000 RPMs for high-end SCSI disks
  - average latency of 3ms
  - improving at 7-10% per year
- **Data rates:** ~100 MB/s, depending on zone
  - average sector transfer time of ~5 us
  - improving at 40+% per year

## Disks and the OS

- Disks are messy physical devices:
  - Errors, bad blocks, missed seeks, etc.
- The job of the OS is to hide this mess from higher level software
  - Low-level device control (initiate a disk read, etc.)
  - Higher-level abstractions (files, databases, etc.)
- The OS may provide different levels of disk access to different clients
  - Physical disk (surface, cylinder, sector)
  - Logical disk (disk block #)
  - Logical file (file block, record, or byte #)

## Software Interface Layers

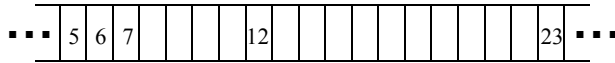


- Each layer abstracts details below it for layers above it
  - Naming and address mapping
  - Caching and request transformations

## Disk Interaction

- Specifying disk requests requires a lot of info:
  - Cylinder #, surface #, track #, sector #, transfer size...
- Older disks required the OS to specify all of this
  - The OS needed to know all disk parameters
- Modern disks are more complicated
  - Not all sectors are the same size, sectors are remapped, etc.
- Current disks provide a higher-level interface (SCSI)
  - The disk exports its data as a logical array of blocks [0...N]
    - Disk maps logical blocks to cylinder/surface/track/sector
  - Only need to specify the logical block # to read/write
  - But now the disk parameters are hidden from the OS

## The common storage device interface



OS's view of storage device

Storage exposed as linear array of blocks  
Common block size: 512 bytes  
Number of blocks: device capacity / block size

## Enhancing achieved disk performance

- Done best where disk management happens (eg, FS)
  - ... and optimization all starts with allocation algorithms
  - Note: there are, of course, exceptions to this rule
- High-level disk characteristics yield two goals:
  - Closeness
    - reduce seek times by putting related things close to one another
    - generally, benefits can be in the factor of 2 range
  - Amortization
    - amortize each positioning delay by grabbing lots of useful data
    - generally, benefits can reach into the factor of 10 range

## Disk Scheduling

- Because seeks are so expensive (milliseconds!), the OS tries to schedule disk requests that are queued waiting for the disk
  - FCFS (do nothing)
    - Reasonable when load is low
    - Long waiting times for long request queues
  - SSTF (shortest seek time first)
    - Minimize arm movement (seek time), maximize request rate
    - Favors middle blocks
  - SCAN (elevator)
    - Service requests in one direction until done, then reverse
  - C-SCAN
    - Like SCAN, but only go in one direction (typewriter)

## Disk Scheduling (2)

- LOOK / C-LOOK
  - Like SCAN/C-SCAN but only go as far as last request in each direction (not full width of the disk)
- In general, unless there are request queues, disk scheduling does not have much impact
  - Important for servers, less so for PCs
- Modern disks often do the disk scheduling themselves
  - Disks know their layout better than OS, can optimize better
  - Ignores, undoes any scheduling done by OS

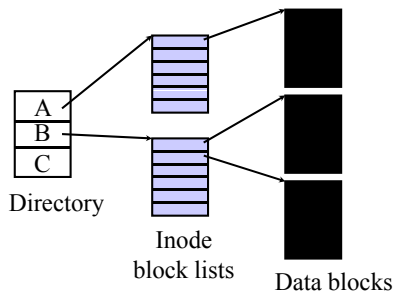
## Back to File Systems...

- File systems need to be aware of disk characteristics for performance
  - Eg) Size affects free space management
    - Bitvector - 1 bit per disk block, 0 if free, 1 if allocated
    - 1 GB disk, 512 byte block size
      - $2^{21}$  blocks,  $2^{18}$  bytes (=256 KB) for bitmap
    - 80 GB disk, 512 byte block size = 20MB for bitmap
  - Alternatives:
    - Linked list of free blocks - inefficient
    - Grouping - first free block contains addresses of N other free blocks
    - Counting - record first free, number of consecutive free

## Allocation strategies

- Disks perform best if seeks are reduced and large transfers are used
  - Scheduling requests is one way to achieve this
  - Allocating related data "close together" on the disk is even more important

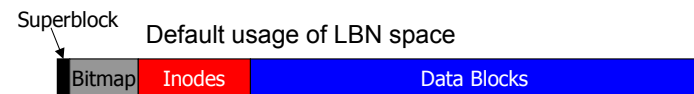
## Inodes: Indirection & Independence



- ➔ File size grows dynamically, allocations are independent
- ➔ **hard to achieve closeness and amortization**

## Original Unix File System

- Recall FS sees storage as linear array of blocks
  - Each block has a *logical block number (LBN)*



- Simple, straightforward implementation
  - Easy to implement and understand
  - But very poor utilization of disk bandwidth (lots of seeking)

## Data and Inode Placement

Original Unix FS had two placement problems:

1. Data blocks allocated randomly in aging file systems
  - Blocks for the same file allocated sequentially when FS is new
  - As FS "ages" and fills, need to allocate into blocks freed up when other files are deleted
  - Problem: Deleted files essentially randomly placed
  - So, blocks for new files become scattered across the disk
2. Inodes allocated far from blocks
  - All inodes at beginning of disk, far from data
  - Traversing file name paths, manipulating files, directories requires going back and forth from inodes to data blocks

Both of these problems generate many long seeks

## FFS

- BSD Unix folks did a redesign (early-mid 80s?) that they called the Fast File System (FFS)
  - Improved disk utilization, decreased response time
  - McKusick, Joy, Leffler, and Fabry, ACM TOCS, Aug. 1984
- Now the FS from which all other Unix FS's have been compared
- Good example of being device-aware for performance

## Cylinder Groups

- BSD FFS addressed placement problems using the notion of a **cylinder group** (aka *allocation groups* in lots of modern FS's)
  - Disk partitioned into groups of cylinders
  - Data blocks in same file allocated in same cylinder group
  - Files in same directory allocated in same cylinder group
  - Inodes for files allocated in same cylinder group as file data blocks



## Cylinder Groups (continued)

- Allocation in cylinder groups provides *closeness*
  - Reduces number of long seeks
- Free space requirement
  - To be able to allocate according to cylinder groups, the disk must have free space scattered across cylinders
  - 10% of the disk is reserved just for this purpose
    - Only used by root - why it is possible for "df" to report >100%
  - If preferred cylinder group is full, allocate from a "nearby" group

## More FFS solutions

- Small blocks (1K) in orig. Unix FS caused 2 problems:
  - Low bandwidth utilization
  - Small max file size (function of block size)
- Fix using a larger block (4K)
  - Very large files, only need two levels of indirection for  $2^{32}$
  - New Problem: internal fragmentation
  - Fix: Introduce "fragments" (1K pieces of a block)
- Problem: Media failures
  - Replicate master block (superblock)
- Problem: Device oblivious
  - Parameterize according to device characteristics

## FFS: Consistency Issues

- Inodes: fixed size structure stored in cylinder groups



- Metadata updates are synchronous operations:
  - Write newly allocated inode to disk before its name is entered in a directory.
  - Remove a directory name before the inode is deallocated
  - Write a deallocated inode to disk before its blocks are placed into the cylinder group free list.

## FFS Observation 1

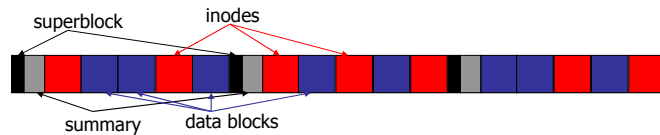
- If the server crashes in between any of these synchronous operations, then the file system is in an inconsistent state.
- Solution:  
Keep a log of updates to enable roll-back or roll-forward.

## FFS Observation 2

- The performance of FFS is optimized for disk block clustering, using properties of the disk to inform file system layout
- Observation: Memory is not large enough that the most of the reads that go to the disk are the first read of a file. Subsequent reads are satisfied in memory by file buffer cache.
- I.e., there is no performance problem with reads. But write calls could be made faster.
- Writes are not well-clustered, they include inodes and data blocks.

## Log Structured File System (LSF)

- Ousterhout 1989
- Write *all* file system data in a continuous log.
- Uses inodes and directories from FFS
- Needs an inode map to find the inodes
  - An inode number is no longer a simple index.
- Cleaner reclaims space from overwritten or deleted blocks.



CSC 369



## LFS Reads

- If the writes are easy, what happens to the reads?
- To read a file from disk:
  1. Read the superblock to find the index file
  2. Read the index file (linear search on block of inodes)
  3. Use the disk address in inode to read the block of index file containing the inode-map
  4. Get the file's inode
  5. Use the inode as usual to find the file's data blocks
- But remember, we expect reads to hit in memory most of the time.

CSC 369



## In practice?

- Cleaning turns out to be quite complicated
- Series of papers looking at performance.
  - Depending on when cleaning happens there may be significant performance loss.
- LFS inspired the logging features of journaling file systems in use today. E.g., Ext3.
- Logs:
  - Old and new versions of data are kept on disk until update is complete
  - For undo capabilities write old data to log
  - For redo capabilities write new data to log
  - A commit operation releases data from the log

CSC 369



## NTFS

- The New Technology File System (NTFS) from Microsoft replaced the old FAT file system.
- The designers had the following goals:
  1. Eliminate fixed-size short names
  2. Implement a more thorough permissions scheme
  3. Provide good performance
  4. Support large files
  5. Provide extra functionality:
    - Compression
    - Encryption
    - Types
- In other words, they wanted a file system flexible enough to support future needs.

CSC 369



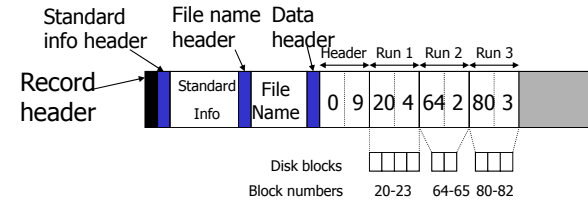
## NTFS

- Each volume (partition) is a linear sequence of blocks (usually 4 Kb).
- Each volume has a Master File Table (MFT).
  - Sequence of 1 KB records.
  - One or more record per file or directory
    - Similar to inodes, but more flexible
  - Each MFT record is a sequence of variable length (attribute, value) pairs.
  - Long attributes can be stored externally, and a pointer kept in the MFT record.
- NTFS tries to allocate files in runs of consecutive blocks.

CSC 369



## MFT Record

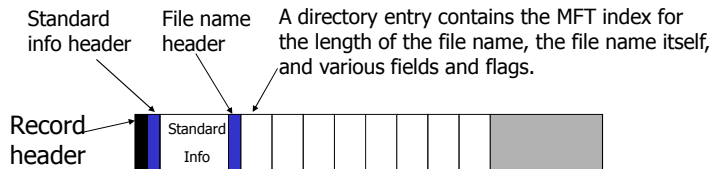


- An MFT record for a 3-run 9-block file.
- Each "data" attribute indicates the starting block and the number of blocks in a "run" (or extent)
- If all the records don't fit into one MFT record, extension records can be used to hold more.

CSC 369



## MFT Record for a Small Directory



- Directory entries are stored as a simple list
- Large directories use B+ trees instead.

CSC 369



## Summary

- I/O overview, Memory hierarchy
- Secondary storage
  - Large, persistent, but **slow**
- Disks
  - Physical structure, interface
  - Performance
  - Scheduling
- File System Optimizations
  - FFS, LFS, NTFS

CSC369H1F



3/21/2006

---

## Next time...

---

- Distributed File Systems
  - Read 8.3, 10.6.4