CSC 2233:

Topics in Computer System Performance and Reliability: **Storage Systems**!
How is functionality implemented?

- **Some in ASIC logic:**
  - Error detection and correction
  - Servo processing
  - Motor-seek control

- **Some in firmware running on control processor**
  - Request processing, queueing, scheduling
  - LBN to PBN mapping
    - Zones
    - Defects
    - Skew
Disk Scheduling

- Because seeks are so expensive, try to be smart about scheduling order of queued disk requests
  - 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/tracks
  - SCAN (elevator)
    - Service requests in one direction until done, then reverse
  - C-SCAN
    - Like SCAN, but only go in one direction (typewriter)
  - LOOK / C-LOOK
    - Like SCAN/C-SCAN but only go as far as last request in each direction (not full width of the disk)
Disk Scheduling (2)

- 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
OS sees storage as linear array of blocks

- Common disk block size: 512 bytes / 4KB
- Number of blocks: device capacity / block size
- Common OS-to-storage requests defined by few fields
  - R/W, block #, # of blocks, memory source/dest
OS sees storage as linear array of blocks

OS’s view of storage device

How does the OS implement the abstraction of files and directories on top of this logical array of disk blocks?
File System Implementation

- File systems define a block size (e.g., 4KB)
  - Disk space is allocated in granularity of blocks

Notice the terminology clash here: “block” is used for different things by the file system and the disk interface… and this kind of thing is common in storage systems!!

Superblock

Default usage of LBN space

Bitmap

Space to store files and directories
File System Implementation

- File systems define a block size (e.g., 4KB)
  - Disk space is allocated in granularity of blocks
- A “Master Block” determines location of root directory (aka superblock)
  - Always at a well-known disk location
  - Often replicated across disk for reliability
- A free map determines which blocks are free, allocated
  - Usually a bitmap, one bit per block on the disk
  - Also stored on disk, cached in memory for performance
- Remaining disk blocks used to store files (and dirs)
  - There are many ways to do this

Superblock

Bitmap

Default usage of LBN space

Space to store files and directories
Disk Layout Strategies

- Files span multiple blocks
- How do you allocate the blocks for a file?
  1. Contiguous allocation
Contiguous Allocation

Disk

<table>
<thead>
<tr>
<th>File Name</th>
<th>Start Blk</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>File A</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>File B</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>File C</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>File D</td>
<td>27</td>
<td>2</td>
</tr>
</tbody>
</table>

directory
Disk Layout Strategies

- Files span multiple disk blocks
- How do you find all of the blocks for a file?
  1. Contiguous allocation
     - Like memory
     - Fast, simplifies directory access
     - Inflexible, causes fragmentation, needs compaction
  2. Linked, or chained, structure
Linked Allocation

<table>
<thead>
<tr>
<th>File Name</th>
<th>Start Blk</th>
<th>Last Blk</th>
</tr>
</thead>
<tbody>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>File B</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
Files span multiple disk blocks
How do you find all of the blocks for a file?
1. Contiguous allocation
   - Like memory
   - Fast, simplifies directory access
   - Inflexible, causes fragmentation, needs compaction
2. Linked, or chained, structure
   - Each block points to the next, directory points to the first
   - Good for sequential access, bad for all others
3. Indexed structure (indirection, hierarchy)
   - An “index block” contains pointers to many other blocks
   - Handles random better, still good for sequential
   - May need multiple index blocks (linked together)
Indexed Allocation: Unix Inodes

- Unix inodes implement an indexed structure for files
- Each file is represented by an inode
- Each inode contains 15 block pointers
  - First 12 are direct block pointers (e.g., 4 KB data blocks)
  - Then single, double, and triple indirect
Unix Inodes, directories and Path Search

- Unix Inodes are **not** directories
- **They describe where on the disk the blocks for a file are placed**
  - Directories are files, so inodes also describe where the blocks for directories are placed on the disk
- **Directory entries map file names to inodes**
  - To open “/one”, use Master Block to find inode for “/” on disk and read inode into memory
  - inode allows us to find data block for directory “/”
  - Read “/”, look for entry for “one”
  - This entry locates the inode for “one”
  - Read the inode for “one” into memory
  - The inode says where first data block is on disk
  - Read that block into memory to access the data in the file
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
Cylinder Groups

- BSD Fast File System (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
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)
- Problem: Media failures
  - Replicate master block (superblock)
- Problem: Device oblivious
  - Parameterize according to device characteristics
Applications exhibit significant locality for reading and writing files

Idea: Cache file blocks in memory to capture locality
- This is called the file buffer cache
- Cache is system wide, used and shared by all processes
- Reading from the cache makes a disk perform like memory
- Even a 4 MB cache can be very effective

Issues
- The file buffer cache competes with VM (tradeoff here)
- Like VM, it has limited size
- Need replacement algorithms
  - E.g. LRU, LFU, various hybrids.
Read Ahead

◆ Many file systems implement “read ahead”
  ■ FS predicts that the process will request next block
  ■ FS goes ahead and requests it from the disk
  ■ This can happen while the process is computing on previous block
    ● Overlap I/O with execution
  ■ When the process requests block, it will be in cache
  ■ Compliments the on-disk cache, which also is doing read ahead

◆ For sequentially accessed files, can be a big win
  ■ Unless blocks for the file are scattered across the disk
  ■ File systems try to prevent that, though (during allocation)
Caching Writes

- On a write, some applications assume that data makes it through the buffer cache and onto the disk.
  - As a result, writes are often slow even with caching.
- Several ways to compensate for this:
  - "write-behind"
    - Maintain a queue of uncommitted blocks
    - Periodically flush the queue to disk
    - Unreliable
  - Battery backed-up RAM (NVRAM)
    - As with write-behind, but maintain queue in NVRAM
    - Expensive
  - Log-structured file system
    - Always write contiguously at end of previous write
    - Will learn more about LFS later in the class
Storage systems and persistence

◆ Threats to persistence?

◆ Device failures
  ■ Whole disk failure
  ■ Partial failure: Bad sectors on hard disk or solid state drive

◆ System crashes / power outages
  ■ File system inconsistencies

◆ Bugs in filesystems or firmware of devices
  ■ Silently corrupted data
  ■ File system inconsistencies

◆ How to protect against data loss?
### The top ten of replaced components

<table>
<thead>
<tr>
<th>Component</th>
<th>HPC1 %</th>
<th>COM1 %</th>
<th>COM2 %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard drive</td>
<td>30.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>28.5</td>
<td>Fan</td>
<td>8.0</td>
</tr>
<tr>
<td>Misc/Unk</td>
<td>14.4</td>
<td>CPU</td>
<td>2.0</td>
</tr>
<tr>
<td>CPU</td>
<td>12.4</td>
<td>SCSI Board</td>
<td>0.6</td>
</tr>
<tr>
<td>PCI motherboard</td>
<td>4.9</td>
<td>NIC Card</td>
<td>1.2</td>
</tr>
<tr>
<td>Controller</td>
<td>2.9</td>
<td>LV Power Board</td>
<td>0.6</td>
</tr>
<tr>
<td>QSW</td>
<td>1.7</td>
<td>CPU heatsink</td>
<td>0.6</td>
</tr>
<tr>
<td>Power supply</td>
<td>1.6</td>
<td>Memory</td>
<td>3.4</td>
</tr>
<tr>
<td>MLB</td>
<td>1.0</td>
<td>SCSI cable</td>
<td>2.2</td>
</tr>
<tr>
<td>SCSI BP</td>
<td>0.3</td>
<td>Fan</td>
<td>2.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU</td>
<td>2.2</td>
</tr>
</tbody>
</table>

- Disks are among the most frequently replaced hardware components.

Same ranking, even when normalized by population size.
Annual replacement rate (ARR) in the field

- Datasheet MTTFs are 1,000,000 to 1,500,000 hours. => Expected annual replacement rate (ARR): 0.58 - 0.88 %.

- Field replacement is a fairly different process from what one might predict based on datasheet MTTF.
Flash-based SSD replacements

- Percentage of drives replaced annually due to suspected hardware problems over the first 4 years in the field:

- ~1-2% of drives replaced annually, much lower than hard disks!
- 0.5-1.5% of drives developed bad chips per year
  - Would have been replaced without methods for tolerating chip failure
SSD partial failures

Non-transparent errors common:

- 26-60% of drives with uncorrectable errors
- 2-6 out of 1,000 drive days experience uncorrectable errors
- Much worse than for hard disk drives (3.5% experiencing sector errors)!
Failures as a function of age - model

Nominal lifetime – 5 years
HDD Replacements as a function of age

- HDD Wear-out seems to set in earlier than often assumed.
SSD wear-out (with program erase cycles)

- Big differences across models (despite same ECC)
- Linear rather than exponential increase
- No sudden increase after PE cycle limit
Disk failures: How to protect against them?

- Regular back-ups
  - Doesn’t solve the problem completely
- Different forms of redundancy
Disk mirroring

◆ Cons:
  ■ Space overhead

◆ Pros:
  ■ Two disks failing simultaneously not likely
  ■ Conceptually simple

◆ Depends:
  ■ Performance:
    ● Better reads (can read from two disks)
    ● Worse writes (wait for slowest disk to finish write)
Lower cost redundancy: Striping + Parity disks

- Striping (without redundancy):
  - Advantage? Parallel transfers.
  - Choice of stripe size?
    - Too large: No parallel transfers
    - Too small: Small request span stripe boundaries without benefits
Lower cost redundancy: Striping + Parity disks

- **Pros:**
  - Much lower space overhead
- **Cons:**
  - Parity disk an become bottleneck
Solution: Striping the parity

- Pros: Removes parity disk bottleneck
- Cons: Update still 2X cost of update in mirrored system
  - Need to read & update parity (1 whole rotation)

![Diagram showing parity striping process]
RAID (Redundant array of inexpensive disks)

- RAID-3/4: Bit and byte level striping, not commonly used
- RAID-6: schemes for tolerating double disk failure
Dealing with latent sector errors

- The disk stores with each sector some error correcting code (ECC)
- But sometimes too many bits are corrupted for correction with ECC => sector failure
- The disk remaps bad sector to another location on disk (using some built-in spare capacity)

- Problem: sector failures are latent
  - Become only visible when sector is being accessed
  - In the meantime another disk in same RAID might fail => data loss
  - => use a “scrubber” to proactively detect bad sectors
Parameterizing a scrubber

◆ Goal: find latent sector errors quickly
◆ Need to configure:
  ■ Scrub frequency
    ● Balancing impact on foreground workload versus speed of error detection
  ■ Scrub order (in which order are sectors scrubbed)
    ● Sequential is common
    ● Why something else than sequential?
      ◆ Making use of clusters of errors…
Scrubbing

- Standard sequential scrubbing
Scrubbing

- Standard sequential scrubbing

- Localized scrubbing
Scrubbing

- Standard sequential scrubbing

- Localized scrubbing

- Accelerated scrubbing
Scrubbing

- Standard sequential scrubbing

- Localized scrubbing

- Accelerated scrubbing

- Staggered scrubbing [Oprea et al. ‘10]
Scrubbing

- Standard sequential scrubbing
- Localized scrubbing
- Accelerated scrubbing
- Staggered scrubbing [Oprea et al. ‘10]
- Accelerated staggered scrubbing

How do those approaches perform in practice, i.e. on real-world data?
Scrubbing: Evaluation on NetApp data

- 10-35% improvement with staggered scrubs!
  - Without introducing any overheads or additional reads
  - Relatively insensitive to choice of parameters