RAID -- Two main ideas

- Parallel reading/writing (*striping*) (*for performance*)
  - Splitting data blocks across multiple disks and read/write them in parallel

- Mirroring (*for reliability*)
  - Have a “mirror” (shadow) disk that stores the same data
  - Every write performed on both disks
    - Can read from either disk
Raid Level 0: Stripe Only

- Level 0 is **non-redundant** disk array
- Files are striped across disks, no redundant info
- High read throughput
- Best write throughput among RAID levels (no redundant info to write)
- Any disk failure results in data loss
Raid Level 1: Mirroring

- Data is written to two places
  - On failure, just use surviving disk
- On read, choose fastest to read
  - Write performance is same as single drive, read performance is 2x better
- Expensive (but used by quite a few companies)
Parity

- What can you do to recover from losing a block in a stripe?
  - Let’s say, we have three disks, A, B, C, storing a stripe of data 011000
  - A:01, B:10, C:00

- Parity:
  - Calculate the XOR of data blocks in data disks
  - Parity(01,00,00) = XOR(01, 10, 00) = 11
  - Now we store parity (11) in a separate disk (D)
  - XOR(A, B, C, D) = 0
  - If we lose a disk (e.g., B)?
    » B = XOR(A, C, D, Parity) = XOR(01, 00, 11) = 10
Raid Level 4

- Block-level parity with striping
- Lower transfer rate for each block (by single disk)
- Higher overall rate (many small files, or a large file)
- Large writes → parity bits can be written in parallel
- Small writes → 2 reads + 2 writes!
- Heavy load on the parity disk
Raid Level 5

- Block Interleaved Distributed Parity
- Like parity scheme, but distribute the parity info over all disks (as well as data over all disks)
- Better (large) write performance
  - No single disk as performance bottleneck
File System Components

- **Naming/Access**
  - User gives file name, not track or sector number, to locate data

- **Disk management**
  - Arrange collection of disk blocks into files

- **Protection and permission**
  - Protect data from different users

- **Reliability/durability**
  - When system crashes, lose stuff in memory, but want files to be durable

User

File Naming

File access

Disk management

Disk drivers
File Systems

- File systems
  - Implement an abstraction (files) for secondary storage
  - Organize files logically (directories)
  - Permit sharing of data among processes, people, and machines
  - Protect data from unwanted access (protection)

- OSes abstract different file systems behind a common interface
  - Unix: virtual file system (VFS)

- Interface defines set of methods and data types
  - OS implemented to use any file system through this interface
  - Linux (ext3, ext4, xfs, btrfs), Windows (FAT16, FAT32, NTFS)
  - Another example of level-of-indirection in systems design
Abstraction: Files

• User view
  ♦ Named collection of bytes with some properties
    » Untyped or typed, with owner, size, last modified, permission, etc.
    » Examples: text, source, object, executables, application-specific
  ♦ Permanently and conveniently available

• Operating system view
  ♦ Map bytes as collection of blocks on physical non-volatile storage device
    » Magnetic disks, tapes, flash, NVM
    » Persistent across reboots and power failures

• File system performs the magic / translation
  ♦ Pack bytes into disk blocks on writing
  ♦ Unpack them again on reading
Why Files?

- Physical reality
  - Block oriented
  - Logical block address
  - No protection among users of the system
  - Data might be corrupted if machine crashes
    » Why can this happen?

- File system abstraction
  - Byte oriented
  - Named files
  - Users protected from each other
  - Robust to machine failures
# Basic File Operations

<table>
<thead>
<tr>
<th>Unix</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>creat(name)</code></td>
<td><code>CreateFile(name, CREATE)</code></td>
</tr>
<tr>
<td><code>open(name, how)</code></td>
<td><code>CreateFile(name, OPEN)</code></td>
</tr>
<tr>
<td><code>read(fd, buf, len)</code></td>
<td><code>ReadFile(handle, …)</code></td>
</tr>
<tr>
<td><code>write(fd, buf, len)</code></td>
<td><code>WriteFile(handle, …)</code></td>
</tr>
<tr>
<td><code>sync(fd)</code></td>
<td><code>FlushFileBuffers(handle, …)</code></td>
</tr>
<tr>
<td><code>seek(fd, pos)</code></td>
<td><code>SetFilePointer(handle, …)</code></td>
</tr>
<tr>
<td><code>close(fd)</code></td>
<td><code>CloseHandle(handle, …)</code></td>
</tr>
<tr>
<td><code>unlink(name)</code></td>
<td><code>DeleteFile(name)</code></td>
</tr>
<tr>
<td><code>rename(old,new)</code></td>
<td><code>MoveFile(name)</code></td>
</tr>
<tr>
<td></td>
<td><code>CopyFile(name)</code></td>
</tr>
</tbody>
</table>
Directories

• Directories serve two purposes
  ♦ For users, they provide a structured way to organize files
  ♦ For the file system, they provide a convenient naming interface that allows the implementation to separate logical file organization from physical file placement on the disk
  ♦ Also gives hints about accessing pattern: files under the same directory are more likely to be accessed together

• Most file systems support multi-level directories
  ♦ Naming hierarchies (/, /usr, /usr/local, …)

• Most OSes support the notion of a current directory
  ♦ Relative names specified with respect to current directory
  ♦ Absolute names start from the root of directory tree
  ♦ Maintained on a per-process basis
Tree-Structured (Hierarchical) Directories

- Directories can contain files and subdirectories
- Efficient searching, allows grouping
Directory Internals

• A directory is a list of entries
  ♦ <name, location>
  ♦ Name is just the name of the file or subdirectory
  ♦ Location depends upon how file is represented on disk

• List is usually unordered (effectively random)
  ♦ Entries usually sorted by program that reads directory
  ♦ Try “ls -U /bin”

• Directories stored as files in UNIX
  ♦ Only need to manage one kind of storage entity
  ♦ “Everything is a file”
  ♦ Use file ops to create/read dirs
  ♦ Some language libraries provide higher-level APIs
Path Name Translation (v1)

- Let’s say you want to open “/one/two/three”
- What does the file system do?
  - Open directory “/” (well known, can always find)
  - Search for the entry “one”, get location of “one” (in dir entry)
  - Open directory “one”, search for “two”, get location of “two”
  - Open directory “two”, search for “three”, get location of “three”
  - Open file “three”
- Systems spend a lot of time walking directory paths
  - Why open is separate from read/write
  - OS will cache prefix lookups for performance
    - /a/b, /a/bb, /a/bbb, etc., all share “/a” prefix
Protection

• File systems implement a protection system
  ♦ Who can access a file
  ♦ How they can access it

• More generally…
  ♦ Objects are “what”, subjects are “who”, actions are “how”

• A protection system dictates whether a given action performed by a given subject on a given object should be allowed
  ♦ You can read and/or write your files, but others cannot
  ♦ You can read “/etc/motd”, but you cannot write it
UNIX Access Rights

- Mode of access: read, write, execute
- Three classes of users
  
  a) **owner access**  
     - Reasoning:  
     - 7  \(\Rightarrow\) 1 1 1
     - RWX
  
  b) **group access**  
     - Reasoning:  
     - 6  \(\Rightarrow\) 1 1 0
     - RWX
  
  c) **public access**  
     - Reasoning:  
     - 1  \(\Rightarrow\) 0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.
- The user “root” is special on Unix  
  - It bypasses all file protection checks in the kernel
- Administrator is the equivalent on Windows
Roadmap

- Interface (API)
  - File operations (open, read, write, close)
  - Directories
- Performance
  - Disk allocation/layout
  - File system designs
  - Buffer cache
- Reliability
  - FS level
  - Disk level: RAID (already covered)
So What Makes File Systems Hard?

- Files grow and shrink
  - Little *a priori* knowledge
  - 6~8 orders of magnitude in file sizes

- Overcoming disk performance behavior
  - Highly nonuniform access
  - Desire for efficiency

- Coping with failure
File System Workloads Drive Designs

- Motivation: Workloads influence design of file system
- File characteristics (measurements of UNIX and NT)
  - Most files are small (about 8KB)
  - Most of the disk is allocated to large files
    - (90% of data is in 10% of files)
- Access patterns
  - Sequential: Data in file is read/written in order
    - Most common access pattern
  - Random: Access block without referencing predecessors
    - Difficult to optimize
  - Access files in same directory together
    - Spatial locality
  - Access metadata when access file
    - Need metadata to find data
Key Questions

• How do we keep track of blocks used by a file?
• Where do we store metadata information?
• How do we (really) do path name translation?
• How do we implement common file operations?
• How can we cache data to improve performance?
• How can we handle disk crashes properly?
Data Structures for A Typical File System

- Process control block
- Open file pointer array
- Open file table (system wide)
- File Metadata Cache
- File Data Cache

In memory

On disk

File system info
- File metadata
- Directories
- File data
File System Disk Layout

How do file systems use the disk to store files?
• File systems define a block size (e.g., 4KB)
  ♦ Disk space is allocated in granularity of blocks
• A “Master Block” (superblock) determines location of root directory
  ♦ 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
Data Allocation Problem

- Definition: allocation data blocks (on disk) when a file is created or grows, and free them when a file is removed or shrinks

- Does this sound familiar?
- Shall we approach it like segmentation or paging?

- Two tasks:
  - How to allocate blocks for a file?
  - How to keep track of blocks?
Disk Bandwidth: Sequential vs Random

• Disk is bandwidth-inefficient for page-sized transfers
  ♦ Sequential vs random accesses

• **Random accesses:**
  ♦ Need seeks, slow (one random disk access latency ~10ms)

• **Sequential accesses:**
  ♦ Stream data from disk (no seeks)

• Sequential access is ~10x or more bandwidth than random
  ♦ Still no where near the 10sGB/sec of memory
Hints

• OS allocates LBAs (logical block addresses) to metadata, file data, and directory data
  ♦ Workload items accessed together should be close in LBA space

• Implications
  ♦ Large files should be allocated sequentially
  ♦ Files in same directory should be allocated near each other
  ♦ Data should be allocated near its meta-data

• Metadata: Where is it stored on disk?
  ♦ Embedded within each directory entry
  ♦ In data structure separate from directory entry
    » Directory entry points to metadata
Allocation Strategies

• Progression of different approaches
  ♦ Contiguous
  ♦ Extent-based
  ♦ Linked
  ♦ File-allocation Tables
  ♦ Indexed
  ♦ Multi-level Indexed

• Questions
  ♦ Amount of fragmentation (internal and external)?
  ♦ Ability to grow file over time?
  ♦ Seek cost for sequential accesses?
  ♦ Speed to find data blocks for random accesses?
  ♦ Wasted space for pointers to data blocks?
Contiguous Allocation

- Allocate each file to contiguous blocks on disk
  - Metadata: Starting block and size of file
  - OS allocates by finding sufficient free space
    - Must predict future size of file; Should space be reserved?
  - Example: IBM OS/360

Advantages
- Little overhead for meta-data
- Excellent performance for sequential accesses
- Simple to calculate random addresses

Drawbacks
- Horrible external fragmentation (Requires periodic compaction)
- May not be able to grow file without moving it
Contiguous Allocation of Disk Space

Analogy in memory management?
**Linked Allocation**

- Allocate linked-list of fixed-sized blocks
  - Metadata: Location of first block of file
    - Each block also contains pointer to next block

Advantages
- No external fragmentation
- Files can be easily grown, with no limit

Disadvantages
- Cannot calculate an address w/o reading previous blocks
- Sequential bandwidth may not be good
  - Try to allocate blocks of file contiguously for best performance
- unreliable: losing a block means losing the rest

Trade-off: Block size (doesn’t need to be the same as sector size)
- Larger --> ??
- Smaller --> ??

FAT (File Allocation Table) (MS-DOS) uses a variation of linked allocation
Linked Allocation

directory

- file: jeep
- start: 9
- end: 25
Indexed Layout

- Indexed layouts use a special block (index block) to store pointers to the data blocks.

- Directory points to the index block.
- Still solves fragmentation problem (can fill in gaps).
- Can easily grow files.
- Also solves random access problem:
  - After reading the index block, know the locations of all blocks.
- For large files, need multiple index blocks.
Example of Indexed Allocation

Analogy in memory relocation?
Next time...

- Read Chapters 40, 41
Backup Slides
RAID 6

- Level 5 with an extra parity bit
- Can tolerate two failures
  - What are the odds of having two concurrent failures?
- No performance penalty on reads, slower on writes (compared to RAID5)