CIS3110 - Lecture 10 - Summary Notes

Introduction to Disk and File System Structure

(Topics: File-System Structure 14.1, 14.2, 14.3, File-System Internals 15.5)

- **Disk as a Data Structure**:
  - A disk is essentially an array of sectors
  - Despite being physically 3D (cylinders, heads, sectors), disks are logically mapped to a linear list
  - Logical ordering: cylinder 0, head 0, sector 0 → cylinder 0, head 0, sector 1, etc.
  - This allows consistent addressing as "Block 0, Block 1, Block 2, etc."
- **File System Organization Basics**:
  - All file systems depend on information stored at the beginning of the disk
  - This beginning information describes the entire disk structure
  - In FAT: boot block → indexing information → root directory → data blocks

FAT vs. Unix File Systems

(Topics: File-System Structure 14.1, 14.2, 14.3, 14.4, File-System Internals 15.5, 15.6)

- **FAT File System**:
  - Uses a linked list approach for tracking file blocks
  - Limitations: can only traverse forward, inefficient for random access
- **Unix File System**:
  - Uses an inode-based tree structure
  - Begins with a "super block" (similar to FAT's master boot record)
  - Super block contains critical information:
    - Number of cylinders, heads, sectors
    - Location of the inode for the root directory
- **Common to Unix-derived systems**:
  - Linux, Mac OS, FreeBSD, etc.
  - All use variations of the inode-based approach

Inode Structure

(Topics: File-System Structure 14.3, 14.4, File-System Internals 15.5, 15.6)

- **Inode Contents**:
  - File metadata (size, timestamps, permissions)
  - Direct pointers to first few data blocks
  - Pointers to indirect blocks for larger files
- **Multi-level Indexing**:
  - **Direct blocks:** First ~6 blocks referenced directly from inode
  - **Single indirect block**: Points to ~1024 data blocks (first level tree)
  - **Double indirect block**: Points to ~1024 single indirect blocks (~1 million blocks)
  - **Triple indirect block**: Points to ~1024 double indirect blocks (~1 billion blocks)
- **Addressing in the Tree**:
  - Block addresses are like memory addresses but for disk
  - They indicate location relative to beginning of disk
  - Example: Block 16,254,955,012 would be stored in the appropriate position in the tree
- **Efficiency Design**:
  - Direct blocks optimize for small files (most common case)
  - Most files are small (under 10KB), so they fit entirely in direct blocks
  - Larger files use progressively more complex structures
  - Design accommodates files from bytes to terabytes in size

File Allocation and Growth

(Topics: File-System Structure 14.5, 14.6, File-System Internals 15.5, 15.6)

- **On-demand Allocation**:
  - Blocks and indirect blocks only allocated when needed
  - Empty file (e.g., from `touch`) has inode but no data blocks
- **Growing a File**:
  - First bytes use direct blocks (efficient)
  - When direct blocks exhausted, allocate single indirect block + first data block it points to
  - Further growth may require double and triple indirect blocks
  - Allocations happen as needed:
    - Direct blocks → single block allocation
    - First block after direct → double allocation (indirect block + data block)

- First block after single indirect → triple allocation (double indirect, single indirect, data block)

File Holes

(Topics: File-System Structure 14.6, File-System Internals 15.6, 15.8)

- **Sparse Files**:
  - If you seek to position 1,000,000 and write one byte, only that byte is stored
  - Intervening space ("hole") consumes no disk space
  - Reading from holes returns zeros
  - Efficient for sparse data
- **Example Scenarios**:
  - **Option A**: Seek to 1,000,000, write null byte → stores only one block + minimal indexing
  - **Option B**: Write 1,000,001 null bytes sequentially → stores all bytes, using full disk space
  - To users, files appear identical when read
  - Tools like `tar` and `rsync` are aware of holes and preserve them

Directory Structure

(Topics: File-System Interface 13.1, 13.2, 13.3, 13.4, File-System Structure 14.1, 14.2, 14.3)

- **Directories as Files**:
  - A directory is a special type of file
  - Contains ordered pairs: (inode number, name)
  - Example: "to find file A, go to inode 3"
- **Directory File Type**:
  - In `ls -l` output, directories marked with 'd' in first column
  - Regular files marked with '-'
- **Directory Entries**:
  - Variable length names (typically up to 255 bytes)
  - Each entry has inode number and name
  - Unlike FAT/DOS which used fixed 8.3 naming format
- **Accessing Files via Directories**:
  - Find file name in directory → get inode number → access inode → access data blocks
  - Note: Name stored separately from file data/metadata (unlike FAT)

Hard Links and Link Count

(Topics: File-System Interface 13.4, File-System Internals 15.5, 15.8)

- **Link Count**:
  - Each inode has a link count tracking references to it
  - Initially 1 for a new file (one directory entry)
- **File Movement Operation**:
  1. Add entry with new name + same inode in destination directory
  2. Increment link count to 2
  3. Remove old directory entry
  4. Decrement link count back to 1
  - Efficient: no actual data blocks moved
  - Safe: power loss during move doesn't lose the file
- **Hard Links**:
  - Multiple directory entries pointing to same inode
  - All references access the same data (same inode)
  - Similar to multiple pointers to the same structure in memory
  - Limitation: can't hard link across different filesystems (inode numbers only unique within filesystem)
  - Cannot hard link directories (prevents loops in the filesystem)

File Opening and Deletion

(Topics: File-System Internals 15.5, 15.8)

- **Opening Files**:
  - When a file is opened, link count incremented
  - File descriptor references the inode
- **File Deletion Process**:
  - `rm` command removes directory entry and decrements link count
  - Actual data blocks only freed when link count reaches zero
  - If a file is open while deleted, it remains accessible to the program that has it open
- **Benefits**:
  - Programs can continue using files that have been "deleted"

- Critical for executable files and paging
- Allows updating system files without crashing running programs
- Explains why OS updates often require reboot (running kernel still using old files)

Hard Links vs. Soft Links

(Topics: File-System Interface 13.4, File-System Internals 15.8)

- **Soft Links (Symbolic Links)**:
  - Small file with type "soft link"
  - Contains path to another file
  - Similar to Windows shortcuts
  - Can point to files on different filesystems
  - Fragile: target can be deleted, leaving "dangling" link
- **Hard Links**:
  - Multiple directory entries referencing same inode
  - More robust: deleting one link doesn't affect others
  - Limited to same filesystem
  - Cannot link to directories

Root Directory and Filesystem Structure

(Topics: File-System Interface 13.2, File-System Structure 14.1, 14.2)

- **Root Directory**:
  - Has no name (slash is a delimiter, not a name)
  - Located via super block information
  - Entry point to entire filesystem hierarchy
- **Windows NT/NTFS Adoption**:
  - Microsoft moved from FAT/DOS model to inode-like structure
  - NTFS uses tree-based indexing similar to Unix
  - Addressed reliability issues of earlier Windows versions