

CIS3110 - Summary Notes (March 20, 2025)

Part 1: Assignment 4 - FAT File System (File-System Structure 14.1-14.6)

Overview of Assignment 4

- The assignment involves **reading blocks from a Microsoft FAT file system**
- Students need to load disk blocks into RAM to examine bytes and understand their structure
- Similar to assignment 1 but dealing with an established file system structure rather than creating one

FAT File System Structure

- Several types of blocks in the FAT file system:
 1. **Boot block** (one per file system)
 2. **FAT blocks** (number specified in boot block)
 3. **Root directory blocks** (number specified in boot block)
 4. **Data blocks** (remaining space)
- To read a file: get directory entry → find first block → chase blocks via FAT

FAT12 Format

- FAT12 means each FAT entry is 12 bits (reference to File-System Internals 15.5-15.6)
- Influences maximum addressable blocks (smaller systems use 12-bit entries, larger ones use 16 or 32)
- FAT12 is for small file systems, FAT16 for up to 16GB, FAT32 for larger systems
- Packed structure example: 2 12-bit entries packed into 3 bytes to save space

Working with Assignment Files

- Various utilities provided:
 - `mdir`: View contents of FAT image
 - `mcopy`: Copy files in/out of FAT image
 - `hexdump`: View file contents as hex and ASCII
 - Alternate dump utility to view contents as longs or shorts
- Sample files provided:
 - `blank.fdt`: Empty formatted file system
 - `bad.fdt`: Intentionally broken FAT file system for testing

- Other valid FAT12 images with files
- `write_data.c`: Utility to create test files with repeated characters

Part 2: File System Buffer Caches (I/O Systems 12.2-12.5)

Types of Buffer Caches

1. Physical block buffer cache

- Stores metadata (FAT table, root directory, inodes in Unix systems)
- File system specific - different structures for different file systems

2. Logical block buffer cache

- Stores actual file data blocks
- Same concept across different file systems
- Associates blocks with files and their offsets (logical block 0, 1, 2, etc.)
- Works for both local and network file systems

Reading Process

1. Calculate logical block number from file offset
2. Check if block is in cache
 - If found: copy data to user program (memory copy)
 - If not found:
 - Allocate space in buffer cache
 - Schedule block read via device driver
 - Look up metadata to find physical block
 - If block is in a "hole," fill with zeros
 - Otherwise issue read request to controller
 - Process blocks until I/O completes
 - Copy data to user program when ready

Writing Process

1. Calculate logical block number from offset
2. Check if block is in cache
 - If not found, allocate buffer
3. For holes or past end of file: fill block with zeros

4. For partial block updates of existing data: read block first
5. Copy user data into buffer at correct location
6. Mark buffer as modified
7. Schedule write to disk
8. Update metadata if needed

I/O Performance Optimizations

1. **Read-ahead:** Pre-populate buffer with blocks program is likely to need next
 - Effective when memory available and sequential access patterns
 - Benefits: faster apparent program execution
2. **Write-behind/Asynchronous writes:** Schedule writes but continue execution
 - Faster but less safe (data loss possible on crash)
3. **Delayed write:** Combine multiple writes to same logical block
 - Wait for block to fill or timer to expire
 - Benefits: fewer disk writes, especially for file modification times
 - Reduces wear on SSDs (limited write cycles)

Part 3: Operating System Security (Security 16.1-16.3, Protection 17.1-17.3)

Security Objectives

- Protect system from user damage
- Protect users from each other
- Protect from network-based threats
- Control access to resources and data

Threat Types

1. **Viruses**
 - Delivered as modifications to legitimate programs
 - Detected via "fingerprints" (byte patterns)
 - Prevention: virus scanners, education
2. **Trojans**
 - Custom-created malicious programs
 - Disguised as legitimate software

- Prevention: only use trusted sources, app stores

3. Worms

- Self-replicating software (no user intervention)
- Exploits buffer overflows/array bound writes
- Example: Morris Worm (1988)

Buffer Overflow Attacks

- How they work:
 1. Program accepts input into a fixed-size buffer
 2. Input exceeds buffer size
 3. Overwrites adjacent memory including return addresses
 4. Attacker can insert executable code and redirect execution
- Prevention:
 1. Bound checking on arrays
 2. Use safe functions (e.g., `fgets`) instead of `gets`)
 3. Allocate from heap instead of stack
 4. Memory protection techniques (e.g., stack randomization)

Security Measures

1. **Physical security:** Locked server rooms, access control
2. **User authentication:** Verify user identity
3. **User permissions:** Principle of least privilege
 - Run services as limited users (e.g., web servers as "www" user)
 - Separate administrator from regular accounts
4. **Resource quotas:** Prevent resource hogging
5. **Activity logging:** Track critical system changes

User Access Control

- Users identified by numeric user IDs
- Privileged users (root/administrator) vs. regular users
- Permission tracking for files, processes, system calls
- Authentication needed to prevent unauthorized access

Next Class

- Will continue with authentication and password encryption
- Upcoming weeks: complete security module followed by review sessions