File System Metadata Snapshotting Project

Executive Summary

This project implements a comprehensive file system metadata snapshotting solution for two major production file systems: **EXT4** (Linux) and **VFAT/FAT32** (Windows). The system creates lightweight snapshots of file system metadata structures that can be used for data recovery even when the original storage device or file system becomes corrupted.

The core innovation lies in extracting and preserving critical metadata structures in a compact, custom format that maintains file system hierarchy and block mapping information without storing actual file content, making recovery operations feasible even from partially damaged storage devices.

---

1. Project Overview and Motivation

1.1 Problem Statement

Modern file systems are vulnerable to various forms of corruption:

- Hardware failures (bad sectors, controller issues)
- Software bugs in drivers or file system code
- Power failures during critical operations
- Malware and ransomware attacks
- User errors

When corruption occurs, traditional backup solutions may be ineffective if:

- The corruption affects metadata structures needed for file access
- The file system cannot be mounted for normal recovery operations
- Only partial recovery tools are available

1.2 Solution Approach

This project addresses these challenges by:

1. **Proactive Metadata Extraction**: Capturing critical file system structures while the system is healthy
2. **Cross-Platform Implementation**: Supporting both EXT4 (Linux) and VFAT (Windows) file systems
3. **Lightweight Storage**: Storing only metadata without file content, making snapshots compact
4. **Independent Recovery**: Enabling data retrieval without mounting the original file system
5. **Hierarchical Preservation**: Maintaining complete directory structure and file organization

---

2. Architecture and Design

2.1 High-Level Architecture

```

```

```
┌─────────────────────────────────────────────────────────────┐
│                      File System Layer                      │
├──────────────────────┬──────────────────────────────────────┤
│     EXT4 Module      │              VFAT Module            │
├──────────────────────┼──────────────────────────────────────┤
│   • Superblock       │   • Boot Sector                     │
│   • Group Desc.      │   • FAT Tables                      │
│   • Inodes           │   • Directory Entries               │
│   • Extent Trees     │   • Cluster Chains                  │
│   • Dir Entries      │   • LFN Support                     │
└──────────────────────┴──────────────────────────────────────┘
                              │
                    ┌─────────────────┐
                    │  Snapshot Core  │
                    │   (Common)      │
                    └─────────────────┘
                              │
┌─────────────────────────────────────────────────────────────┐
│                     Recovery Interface                      │
├──────────────────────┬──────────────────────────────────────┤
│  Shell Interface     │          Filter System              │
│  • Navigation        │   • Time-based filters              │
│  • File listing      │   • Size filters                    │
│  • File extraction   │   • Permission filters              │
│  • Tree display      │   • Name pattern matching           │
└──────────────────────┴──────────────────────────────────────┘
```

2.2 Core Design Principles

1. **File System Agnostic Core**: Common interfaces for different file systems
2. **Metadata-Only Storage**: No file content duplication, only structural information
3. **Efficient Block Mapping**: Preserving file-to-block relationships for recovery
4. **Hierarchical Organization**: Maintaining parent-child directory relationships
5. **Incremental Processing**: Supporting large file systems through streaming algorithms

---

3. EXT4 Implementation

3.1 EXT4 File System Structure Analysis

The EXT4 implementation (`ext4/` directory) handles the complex metadata structures of the EXT4 file system:

3.1.1 Key Structures Captured

SuperBlock Structure (`ext4_utilities.h:10-32`):

- File system parameters (block size, inode count, feature flags)
- Critical for understanding file system layout
- Located at offset 1024 bytes from the beginning

Group Descriptors (`ext4_utilities.h:69-74`):

- Block and inode bitmap locations
- Inode table locations
- Support for 64-bit addressing when enabled

Inode Structure (`ext4_utilities.h:110-148`):

- File metadata (permissions, timestamps, size)
- Block mapping information (direct/indirect or extent trees)
- Extended attributes and feature flags

Extent Trees (`ext4_utilities.h:212-276`):

- Modern EXT4 block allocation method
- Hierarchical structure for efficient large file support
- Leaf and internal node handling

3.1.2 Snapshot Creation Process

The snapshot creation process (`snapshot.cpp` and `snapshot_beta.cpp`) follows these steps:

1. **Superblock Extraction** (`snapshot.cpp:392-395`):

   ```
   void* meta_data = malloc(1024);
   pread(super.fd,meta_data,1024,1024);
   pwrite(fd,meta_data,1024,parent_pos);
   ```

2. **Group Descriptor Table** (`snapshot.cpp:398-404`):

   ```
   temp = ceil((super.blocks_count*1.0)/super.blocks_per_group);
   temp *= super.desc_size;
   meta_data = malloc(temp);
   pread(super.fd,meta_data,temp,std::ceil(2048.0/block_size)*block_size)
   ;
   ```

3. **Directory Tree Traversal** (`snapshot.cpp:308-355`):
   - Recursive traversal starting from root inode (inode 2)
   - Extraction of directory entries with complete metadata
   - Support for both extent-based and traditional block addressing

4. **File Block Mapping** (`snapshot.cpp:25-162`):
   - **Extent-based files**: Flattening extent trees into linear extent lists
   - **Block-addressed files**: Converting indirect blocks into extent format
   - Preserving logical-to-physical block mappings

3.1.3 Advanced Features

Extent Tree Processing (`snapshot.cpp:25-56`):

- Handles hierarchical extent structures
- Sorts extents by logical block number for correct ordering
- Recursive processing of internal nodes

Block Address Conversion (`snapshot.cpp:58-162`):

- Converts traditional indirect block addressing to extent format
- Handles direct, single-indirect, double-indirect, and triple-indirect blocks
- Maintains compatibility with older EXT2/EXT3 structures

3.2 EXT4 Snapshot Format

The EXT4 snapshot file format:

```
[Device Path Length][Device Path][SuperBlock][Group Descriptors][Root
Entry][Directory Tree][File Mappings]
```

Entry Structure (`snap_utilities.h:256-265`):

```
struct Entry {
    Inode    inode;          // 156 bytes - Complete inode structure
    uint     inode_num,      // 4 bytes - Inode number
             name_len,       // 4 bytes - Filename length
             parent,         // 4 bytes - Parent directory location
             contents,       // 4 bytes - Contents location
             flag;           // 4 bytes - Last entry flag
    char*    name;           // Variable - Filename (4-byte aligned)
};
```

---

4. VFAT Implementation

4.1 VFAT/FAT32 File System Analysis

The VFAT implementation (`vfat/` directory) handles the FAT32 file system structure:

4.1.1 Boot Sector and FAT Analysis

Boot Sector Structure (`vfat/code/tree/boot_sector.c:19-38`):

- BPB (BIOS Parameter Block) information
- Cluster and sector size calculations
- FAT table locations and root directory cluster

Directory Entry Structures (`vfat/code/tree/dir_tree.c:68-88`):

```c
struct dir_table_entry {
    char        name[8], ext[3];    // 8.3 filename format
    u_int8_t    attr;               // File attributes
    u_int16_t   MSB, LSB;           // Cluster number (high/low)
    uint        size;               // File size
    // ... timestamp fields
};
```

Long Filename (LFN) Support (`vfat/code/tree/dir_tree.c:90-105`):

- Unicode filename support
- Multiple LFN entries for long names
- Checksum validation for LFN consistency

4.1.2 VFAT Snapshot Process

The VFAT snapshotting process involves several specialized tools:

1. **Directory Tree Analysis** (`vfat/code/tree/dir_tree.c`):
    - Traverses cluster chains using FAT table
    - Extracts directory entries with LFN reconstruction
    - Builds hierarchical directory structure
2. **Cluster Chain Mapping** (`vfat/code/dump/dump.c`):
    - Follows FAT entry chains to map file clusters
    - Creates portable mapping files for file reconstruction
    - Handles fragmented files across non-contiguous clusters
3. **Memory Mapping Creation** (`vfat/code/mappings/`):
    - Creates indexed access structures for fast file lookup
    - Unicode filename processing and storage
    - Efficient search capabilities for large directories

4.1.3 VFAT-Specific Challenges

Unicode Handling (`vfat/code/tree/dir_tree.c:107-132`):

```c
void name_resolve(uint file_entry, uint seq_num) {
    struct LFN long_name;
    for(uint i=1; i<=seq_num; i++) {
        pread(bs.fd,(void *)&long_name,32,file_entry-i*32);
        // Extract Unicode characters from LFN entries
        // Handle UTF-16 to ASCII conversion
    }
}
```

Cluster Chain Following (`vfat/code/tree/dir_tree.c:152-160`):

```
uint fat_entry(uint N) {
    uint MSB, LSB, start, entry;
    MSB = N>>16; LSB = N&0xffff;
    start = ((uint)bs.BPB_RsvdSecCnt)*512;
    pread(bs.fd,(void *)&entry,4,start+MSB*512+LSB*4);
    return entry;
}
```

---

5. Recovery and Restoration System

5.1 Shell Interface

The recovery system provides a Unix-like shell interface (`ext4/code/main/shell.cc`) for navigating and extracting data:

5.1.1 Navigation Commands

Directory Navigation (`shell.cc:158-181`):

- `cd` command for changing directories
- Path resolution with relative and absolute paths
- Support for `.` and `..` directory references

File Listing (`shell.cc:144-155`):

- `ls` command for directory contents
- Displays files and subdirectories from snapshot data
- Works without accessing original file system

Tree Display (`shell.cc:184-200`):

- Hierarchical directory tree visualization
- Recursive directory traversal
- Visual representation of file system structure

5.1.2 File Operations

File Extraction (`shell.cc:203-215`):

- Extracts files from snapshot metadata
- Reconstructs files using block mapping information
- Supports both individual files and directory trees

File Details (`shell.cc:218-241`):

- Displays comprehensive file metadata
- Shows permissions, timestamps, and file size
- Provides inode information for advanced users

5.2 Advanced Filtering System

The filtering system (`ext4/code/main/filter.cc`) provides sophisticated search capabilities:

5.2.1 Time-Based Filtering

Time Management (`filter.cc:14-120`):

```
class TimeManager {
    time_t time_resolve(std::string tme);    // Parse time strings
    bool check(time_t tme);                  // Validate against filters
    void set_time(std::string time_code);    // Set time constraints
    void set_day(std::string day_code);      // Set day constraints
};
```

Supported Time Filters:

- Creation time (crtime)
- Modification time (mtime)
- Access time (atime)
- Day-of-week filtering
- Range-based time queries

5.2.2 File Attribute Filtering

Permission Filtering (`filter.cc:146-220`):

- User, group, and other permission checking
- Executable, readable, writable flags
- Negation support (e.g., "not writable")

Size Filtering (`filter.cc:226-266`):

- Size range queries with unit support (B, KB, MB, GB)
- Exact size matching
- Greater than/less than comparisons

Name Pattern Matching (`filter.cc:271-297`):

- Prefix, suffix, and substring matching
- Regular expression-like capabilities
- Case-sensitive and case-insensitive options

5.2.3 File Type Filtering

Type Classification (`filter.cc:304-321`):

```
void Filter::detail(Inode inode) {
    if(0x1000<=inode.mode && inode.mode<0x2000)
        std::cout<<"FIFO";
    else if(0x4000<=inode.mode && inode.mode<0x6000)
        std::cout<<"Directory";
    else if(0x8000<=inode.mode && inode.mode<0xA000)
        std::cout<<"Regular File";
    // ... additional file types
}
```

---

6. Technical Implementation Details

6.1 Low-Level File System Access

6.1.1 Direct Block Device Access

Both implementations use direct block device access bypassing the kernel file system layer:

```
int fd = open("/dev/sda1", O_RDONLY);  // Direct device access
pread(fd, buffer, size, offset);       // Position-independent reading
```

Advantages:

- Works even when file system cannot be mounted
- Accesses raw metadata structures
- Bypasses file system caches and buffers
- Enables recovery from corrupted file systems

6.1.2 Endianness and Data Structure Handling

EXT4 Structure Reading (`ext4_utilities.h:35-61`):

```
void init_super(int fd, struct SuperBlock *super) {
    uint offset = 1024;  // EXT4 superblock location
    pread(fd,(void *)&super->inodes_count,4,offset);
    pread(fd,(void *)&super->blocks_count,4,offset+4);
    // ... read all superblock fields
}
```

VFAT Structure Handling (`vfat/code/tree/boot_sector.c:78-89`):

```
pread(fd,(void *)&BPB_BytsPerSec,2,11);    // Bytes per sector
pread(fd,(void *)&BPB_SecPerClus,1,13);    // Sectors per cluster
pread(fd,(void *)&BPB_RsvdSecCnt,2,14);    // Reserved sectors
```

6.2 Memory Management and Performance

6.2.1 Streaming Processing

The implementation uses streaming algorithms to handle large file systems:

Directory Processing (`snapshot.cpp:186-201`):

- Processes directories one block at a time
- Avoids loading entire directory structures into memory
- Supports file systems with millions of files

Extent Processing (`snapshot.cpp:28-56`):

- Sorts extents efficiently for optimal access patterns
- Uses vector containers for dynamic memory management
- Minimizes memory fragmentation during processing

6.2.2 Error Handling and Robustness

Graceful Degradation:

- Continues processing when encountering minor corruption
- Skips unreadable blocks while preserving accessible data
- Provides detailed logging for debugging issues

Validation Checks:

- Verifies magic numbers and checksums where available
- Validates pointer references before dereferencing
- Implements bounds checking for array access

---

7. Usage Examples and Workflows

7.1 Creating a Snapshot

7.1.1 EXT4 Snapshot Creation

```
# Compile the snapshot tool
cd ext4/code/main/
g++ -o snapshot snapshot.cpp

# Create snapshot (requires root access)
sudo ./snapshot
```

```
# Enter device path: /dev/sda1
# Enter snapshot file: /backup/ext4_snapshot.dat
```

7.1.2 VFAT Snapshot Creation

```
# Create VFAT directory tree analysis
cd vfat/code/tree/
gcc -o tree dir_tree.c
sudo ./tree > directory_structure.txt

# Create file mappings
cd ../dump/
gcc -o dump dump.c
sudo ./dump
```

7.2 Recovery Operations

7.2.1 Loading and Browsing a Snapshot

```
# Compile recovery shell
cd ext4/code/main/
g++ -o shell shell.cc filter.cc

# Load snapshot and start recovery shell
./shell
> load /backup/ext4_snapshot.dat
> ls /
> cd /home/user
> tree Documents/
```

7.2.2 Filtering and Searching

```
# Find all files modified in the last week
> filter mtime>2023-10-01

# Find large files (>100MB)
> filter size>100M

# Find executable files
> filter perm=x type=file

# Find files with specific names
> filter name_contains=config name_suffix=.conf
```

7.2.3 File Extraction

```
# Extract a specific file
> extract /home/user/important_document.pdf

# Extract entire directory tree
> extract_tree /home/user/Documents/

# Get detailed file information
> details /home/user/important_document.pdf
```

---

8. Advanced Features and Extensions

8.1 Implemented Features

8.1.1 EXT4-Specific Features

1. **Extent Tree Support**: Full support for EXT4's modern extent-based block allocation
2. **64-bit Block Addressing**: Support for large file systems (>2TB)
3. **Feature Flag Detection**: Automatic detection of EXT4 feature usage
4. **Group Descriptor Handling**: Support for both 32-bit and 64-bit group descriptors

8.1.2 VFAT-Specific Features

1. **Long Filename Support**: Complete Unicode LFN handling
2. **FAT32 Cluster Chains**: Efficient cluster chain following and mapping
3. **Directory Entry Parsing**: Support for deleted file recovery
4. **Endianness Handling**: Proper little-endian data interpretation

8.2 Future Enhancement Opportunities

8.2.1 Currently Missing Features

EXT4 Extensions (noted in source comments):

```
/*
    **  Remaining Work  **
    1)  Inline data
    2)  Symbolic links
    3)  Hash structures
    4)  Sparse representation
*/
```

Additional File System Support:

- NTFS support for Windows systems

- XFS support for high-performance Linux systems
- Btrfs support with snapshot-aware features
- exFAT support for cross-platform compatibility

8.2.2 Performance Optimizations

1. **Parallel Processing**: Multi-threaded snapshot creation for large file systems
2. **Incremental Snapshots**: Only capture changes since last snapshot
3. **Compression**: Compress metadata for reduced storage requirements
4. **Indexing**: Create search indexes for faster file lookups

8.2.3 Recovery Enhancements

1. **GUI Interface**: Graphical user interface for easier recovery operations
2. **Network Recovery**: Remote snapshot creation and recovery capabilities
3. **Automated Recovery**: Scripted recovery workflows for common scenarios
4. **Integration**: Integration with existing backup and recovery solutions

---

9. Testing and Validation

9.1 Test Coverage

The project includes several test files demonstrating functionality:

9.1.1 EXT4 Testing

Structure Validation (`ext4/code/tree/test.c`):

- Validates superblock parsing
- Tests inode location calculations
- Verifies group descriptor handling

Performance Testing (`ext4/code/main/time_test.cpp`):

- Measures snapshot creation time
- Tests memory usage patterns
- Validates large file system handling

9.1.2 VFAT Testing

Unicode Testing (`vfat/code/tree/unicode.c`):

- Tests long filename parsing
- Validates Unicode character handling
- Tests LFN checksum validation

Mapping Verification (`vfat/code/mappings/test.c`):

- Validates cluster chain following
- Tests file extraction accuracy

- Verifies directory structure preservation

9.2 Validation Methodology

1. **Round-Trip Testing**: Create snapshots and verify complete file recovery
2. **Corruption Simulation**: Test recovery from artificially corrupted file systems
3. **Cross-Platform Validation**: Test snapshots created on one system and recovered on another
4. **Scale Testing**: Validate performance with file systems of various sizes

---

10. Security and Reliability Considerations

10.1 Security Aspects

10.1.1 Access Control

- **Root Access Requirement**: Direct block device access requires administrative privileges
- **Read-Only Operations**: Snapshot creation uses read-only access to prevent accidental modification
- **Privilege Separation**: Recovery operations can run with reduced privileges

10.1.2 Data Protection

- **Metadata Only**: No sensitive file content stored in snapshots
- **Checksums**: Where available, validates data integrity using file system checksums
- **Secure Deletion**: Provides secure cleanup of temporary files during processing

10.2 Reliability Features

10.2.1 Error Handling

```
// Example error handling in snapshot creation
if(init_extent_manager(&manager,curr_loc,&super)) {
    // Handle extent tree processing
} else {
    // Fallback to block addressing
    pos = dump_file_blockaddress(pos);
}
```

10.2.2 Data Validation

- **Magic Number Checking**: Validates file system signatures before processing
- **Bounds Checking**: Prevents buffer overflows and invalid memory access
- **Consistency Checks**: Verifies internal data structure consistency

---

11. Performance Analysis

11.1 Snapshot Creation Performance

11.1.1 Time Complexity

- **EXT4**: O(n) where n is the number of files and directories
- **VFAT**: O(n*m) where n is files and m is average cluster chain length

11.1.2 Space Complexity

EXT4 Snapshot Size:

```
Size ≈ NumFiles × (156 + 8 + AvgNameLength) + NumExtents × 12 +
FixedOverhead
```

VFAT Snapshot Size:

```
Size ≈ NumFiles × (32 + AvgNameLength×2) + NumClusters × 4 + FixedOverhead
```

11.2 Recovery Performance

11.2.1 File Lookup Performance

- **Linear Search**: O(n) for unindexed searches
- **Tree Traversal**: O(log n) for hierarchical navigation
- **Filter Operations**: O(n×f) where f is filter complexity

11.2.2 File Extraction Performance

- **Sequential Access**: Optimal when extents are contiguous
- **Random Access**: Performance depends on storage device characteristics
- **Parallel Extraction**: Multiple files can be extracted concurrently

---

12. Conclusions and Future Directions

12.1 Project Achievements

This file system metadata snapshotting project successfully demonstrates:

1. **Cross-Platform Compatibility**: Working implementations for both Linux (EXT4) and Windows (VFAT) file systems
2. **Low-Level Expertise**: Deep understanding of file system internals and data structures
3. **Practical Recovery Solution**: Usable tools for data recovery scenarios
4. **Efficient Design**: Lightweight metadata-only approach minimizes storage requirements
5. **Extensible Architecture**: Clean separation between file system specific and common code

12.2 Technical Innovation

The project's key innovations include:

1. **Unified Extent Representation**: Converting different block addressing schemes to a common extent format
2. **Streaming Metadata Processing**: Handling large file systems without excessive memory usage
3. **Direct Block Device Access**: Bypassing file system layers for robust recovery capabilities
4. **Comprehensive Filtering**: Advanced search and filtering capabilities for efficient data location

12.3 Real-World Applications

This technology could be valuable in:

1. **Enterprise Backup Solutions**: As a complement to traditional backup systems
2. **Forensic Analysis**: For investigating damaged or corrupted storage devices
3. **Data Recovery Services**: Professional data recovery operations
4. **System Administration**: Proactive monitoring and recovery preparation
5. **Research and Development**: Academic study of file system structures and recovery techniques

12.4 Future Development Paths

12.4.1 Short-Term Improvements

1. **Complete Feature Implementation**: Add support for symbolic links, inline data, and hash directories
2. **Enhanced Error Handling**: More robust handling of edge cases and corruption scenarios
3. **Performance Optimization**: Multi-threading and improved algorithms for large-scale operations
4. **User Interface Improvements**: Better command-line interface and potential GUI development

12.4.2 Long-Term Expansion

1. **Additional File Systems**: Support for NTFS, XFS, Btrfs, and other modern file systems
2. **Cloud Integration**: Support for cloud storage metadata and distributed file systems
3. **Machine Learning Integration**: Intelligent corruption detection and recovery suggestions
4. **Enterprise Integration**: APIs and integration capabilities for existing backup and recovery solutions

---

Appendix A: File Organization

EXT4 Implementation Structure

```
ext4/
├── code/
│   ├── ext4_utilities.h        # Core EXT4 data structures
│   ├── snap_utilities.h        # Snapshot-specific utilities
│   ├── main/
│   │   ├── snapshot.cpp        # Main snapshot creation
│   │   ├── snapshot_beta.cpp   # Alternative implementation
│   │   ├── filter.cc/.hh       # Advanced filtering system
│   │   ├── shell.cc/.hh        # Recovery shell interface
│   │   └── test.cpp           # Testing and validation
│   └── tree/
│       ├── superblock.c        # Superblock analysis
│       ├── inode.c            # Inode processing
│       ├── group_des.c        # Group descriptor handling
│       ├── dir_entry.c        # Directory entry processing
│       └── root_dir.c         # Root directory handling
├── *.png                      # EXT4 structure diagrams
└── structure.png              # Overall EXT4 layout
```

VFAT Implementation Structure

```
vfat/
├── code/
│   ├── tree/
│   │   ├── boot_sector.c       # FAT32 boot sector analysis
│   │   ├── dir_tree.c          # Directory traversal
│   │   ├── unicode.c           # Long filename support
│   │   └── tree_with_unicode.c # Complete tree with LFN
│   ├── dump/
│   │   ├── dump.c              # File extraction utilities
│   │   ├── analysis.c          # File system analysis
│   │   └── search.c            # File search capabilities
│   └── mappings/
│       ├── parse.c             # Metadata parsing
│       ├── read_check.c        # Data validation
│       └── pr.c               # Print and display utilities
├── *.png                      # VFAT structure diagrams
├── *.pdf                      # FAT32 specification documents
└── Fat32_structure.svg        # FAT32 layout diagram
```

Appendix B: Compilation and Build Instructions

EXT4 Tools

```
# Navigate to EXT4 directory
cd ext4/code/main/

# Compile snapshot creator
g++ -std=c++11 -o snapshot snapshot.cpp

# Compile recovery shell
g++ -std=c++11 -o shell shell.cc filter.cc

# Compile test utilities
g++ -std=c++11 -o test test.cpp

# Tree analysis tools
cd ../tree/
gcc -o superblock superblock.c
gcc -o inode inode.c
gcc -o group_des group_des.c
```

VFAT Tools

```
# Navigate to VFAT directory
cd vfat/code/tree/

# Compile directory tree analyzer
gcc -o dir_tree dir_tree.c

# Compile boot sector analyzer
gcc -o boot_sector boot_sector.c

# Compile file dump utilities
cd ../dump/
gcc -o dump dump.c
gcc -o analysis analysis.c

# Compile mapping utilities
cd ../mappings/
gcc -o parse parse.c
gcc -o read_check read_check.c
```

System Requirements

- **Operating System**: Linux (for EXT4 tools), Windows/Linux (for VFAT tools)
- **Compiler**: GCC 4.8+ or equivalent with C++11 support
- **Privileges**: Root/Administrator access for direct block device access
- **Dependencies**: Standard C/C++ libraries, POSIX system calls

---

This report documents a sophisticated file system metadata snapshotting implementation that demonstrates deep understanding of file system internals and provides practical data recovery capabilities for both Linux and Windows environments.