Data Structures

User Indexing: AVL Tree

Structure Used: `AVLTree<UserInfo>` with username as key

Why AVL Tree?

- **O(log n) lookup complexity** for user authentication - critical for login operations
- **Self-balancing** ensures worst-case performance remains logarithmic
- **Ordered traversal** enables efficient user listing for admin operations

Alternative Considered: Hash table

Directory Tree Representation: Vector-based Parent-Child Structure

Structure Used:

cpp

```cpp
struct Tree {
    uint32_t prnt;              // Index into files[] array
    vector<uint32_t> childs;    // Child indices
};
vector<Tree> Root;              // Parallel to files[] vector
vector<FileEntry> files;        // Actual file/directory entries
```

Why This Approach?

- **Direct indexing** - O(1) access to any file entry via array index
- **Flexible hierarchy** - each Tree node maintains a dynamic list of children
- **Cache-friendly** - vectors provide contiguous memory allocation
- **Simple parent traversal** - single integer stores parent reference
- **Efficient for typical file systems** - most directories have few children

Free Space Tracking: Free List (Vector)

Structure Used: `vector<uint32_t> free_segments`

Why Free List?

- **Simple allocation** - pop from vector to allocate, push to deallocate
- **Minimal overhead** - only 4 bytes per free block reference
- **Fast allocation** - O(1) for sequential allocation patterns
- **Sufficient for block-level allocation** - filesystem uses fixed-size blocks

Implementation Details:

- Each entry stores the **byte offset** of a free block in the .omni file
- Blocks marked with `BlockHdr.isValid = true` when free
- Deleted file blocks returned to free_segments during deletion

Limitation Acknowledged: This approach leads to external fragmentation. A bitmap or buddy system would provide better fragmentation control but increases complexity.

File Path to Disk Location Mapping: Dual-Index System

Primary Structure: `AVLTree<FileMetadata> Mds` (path → metadata)

Why AVL Tree for Path Mapping?

- **O(log n) path resolution** for file operations
- **String key support** - full absolute paths as keys
- **Metadata caching** - stores computed block counts and actual sizes
- **Enables fast existence checks** without disk I/O

Secondary Structure: `findFileIndexByPath()` function

- Uses `Root[]` and `files[]` for hierarchical traversal
- Tokenizes path and walks directory tree
- Required for operations needing parent-child relationships

.omni File Structure

Overall Layout

[OMNIHeader: 504bytes]
[User Table: max_users × sizeof(UserInfo)]
[File Entry Table: max_files × sizeof(FileEntry)]

[Data Region: (BlockHdr + block_size) repeated]

Header (504 bytes)

- Magic number: "OMNIFS01"
- Format version, sizes, student ID, timestamp
- Critical: `user_table_offset`, `max_users`, `block_size`, `total_size`

User Table

- **Fixed-size allocation** - simplifies offset calculation
- Empty slots filled with zeroed UserInfo structs
- Admin user always at index 0 (after header)

File Entry Table

- **Fixed-size preallocated** for max_files entries
- Root directory ("/") always at index 0
- Empty entries marked with `type = 3` (unused sentinel value)
- Deleted entries marked with `type = 2` for reuse

Data Region

Each block consists of BlockHdr (12 bytes)

Design Choices:

- **Linked list chaining** for file blocks via `nxt` pointer
- **Separate size field** allows partial block usage

- **isValid flag** enables in-place free space tracking
- **Block offset calculation:** `data_region_start + block_index × (sizeof(BlockHdr) + block_size)`

Memory Management Strategies

What Lives in Memory

1. **Complete file entry table** (`vector<FileEntry> files`)
2. **Directory tree structure** (`vector<Tree> Root`)
3. **User AVL tree** (all UserInfo structures)
4. **File metadata AVL tree** (all FileMetadata with paths)
5. **Free block list** (`vector<uint32_t> free_segments`)

Rationale: For typical configurations (max_files: 10,000), entire metadata fits in ~5-10 MB RAM, enabling O(1) or O(log n) operations without disk seeks.

Lazy Disk I/O

- File **data blocks** never cached - read/written on demand
- Block headers read only when traversing file chains
- Metadata persisted only on shutdown in destructor

Deleted Entry Recycling

cpp

```
vector<uint32_t> deleted;  // Indices of deleted files
```

- Tracks reusable slots in files[] array
- `createFile()` checks deleted vector before appending

Memory Persistence Strategy

Destructor writes back:

1. Updated OMNIHeader
2. All users from AVL tree → user table
3. All files from vector → file entry table