ExFS2 File System Implementation Roadmap

Overview: ExFS2 is an **extensible file system** stored in regular files (segments) rather than a raw disk partition. It uses fixed-size 1 MB *segment files* to hold filesystem data: *inode segments* store directory inodes and metadata, and *data segments* store file contents in 4096-byte blocks. The file system supports a hierarchical directory structure and basic operations (via a command-line tool): adding files, reading/extracting files, removing files or directories, and listing directory contents. Below is a structured development plan, including code organization, data structures, implementation milestones, and design/testing considerations.

Codebase Structure

Organize the project into modular components, each with clear responsibilities. A possible directory and file layout is:

- **`src/`** – Source code directory (could also have subdirs like `include/` for headers, etc.)
  - **`main.c`** – Parses command-line arguments and dispatches to filesystem operations (`-a`, `-e`, `-r`, `-l`, `-D`).
  - **`fs.h`** / **`fs.c`** – Core filesystem logic (initialization, shutdown, high-level API for operations). Defines global structures (e.g. list of open segments) and functions to load/save segments and perform operations.
  - **`inode.h`** / **`inode.c`** – Inode structure definition and inode management functions. Handles reading/writing inodes in inode segments, inode allocation, and pointer management (direct/indirect blocks).
  - **`dir.h`** / **`dir.c`** – Directory handling: functions to create directories, lookup entries, and list directory contents. Manages directory entry structures (name→inode mappings) stored in data blocks.
  - **`data.h`** / **`data.c`** – Data block management: functions for allocating/freeing 4096-byte blocks in data segments, reading/writing file content blocks, and managing the free-block bitmap for each data segment.
  - **`segment.h`** / **`segment.c`** – Segment management: abstractions for opening segment files, tracking segment metadata in memory (e.g. file descriptors, segment IDs, free space). May handle creating new segment files when needed for expansion.
  - **`util.h`** / **`util.c`** – Utility functions (e.g. path parsing into components, bitmap operations for free lists, error handling, etc.).
  - **(Optional)** **`test/`** – Test code or scripts for unit and integration testing (not part of final program, but useful during development).
  - **`Makefile`** – Build script to compile the above into the `exfs2` binary.

This structure ensures separation of concerns: **`main.c`** only handles CLI and delegates to an *FS library* implemented in the other modules. Each module can be developed and tested in isolation (for example, test `inode.c` functions without the full CLI in place). Organizing by functionality (inodes, directories, data blocks, etc.) will make the code easier to maintain and extend.

Key Data Structures

Design the on-disk data structures and their in-memory representations. The **on-disk format** must follow the specification so the filesystem can be correctly interpreted across runs:

- **Inode Structure (`struct inode`):** Each file or directory has an inode that fits in one 4096-byte block. The inode should store:
  - **Metadata:** file type (directory or regular file), file size (in bytes), and possibly flags. We omit user IDs, permissions, and timestamps (not required for this project).
  - **Direct Pointers:** An array of direct block pointers (identifiers to data blocks holding file content) – include as many direct pointers as will fit after accounting for the metadata. For example, if using 32-bit block addresses, a 4096-byte inode could contain on the order of ~1000 direct pointers (exact count depends on metadata size).
  - **Indirect Pointers:** One single-indirect pointer, one double-indirect pointer, and one triple-indirect pointer. These pointers reference indirect blocks (which themselves reside in data segments) that contain additional block addresses. The triple-indirect pointer is optional per the spec, but including a field for it (even if you don't fully implement triple-level traversal) future-proofs the design.
  - *Pointer format:* A block pointer could be represented as a 32-bit value encoding the target data segment and block index. For instance, use high bits for the segment number and low bits for the block number within that segment. Alternatively, maintain a mapping structure to resolve a block pointer to the correct segment file and offset. Simpler is to assign each data block a global number sequentially or by combining segment ID and block index.
- **Directory Entry (`struct dir_entry`):** Directories are essentially files whose data blocks contain a list of entries (each entry mapping a name to an inode). Each directory entry can be a fixed-size record, for example:
  - `uint32_t inode_number;`
  - `char name[MAX_NAME_LEN];` (with `MAX_NAME_LEN` set to a fixed size, e.g. 256 bytes, or use a smaller size if needed to fit multiple entries in one 4096-byte block).

  Directory and file names should *only* be stored in these directory entry structures (not in inodes)file-bqzri7vsrdrbjz2yw892pj. Each directory's data block will contain an array of `dir_entry` records (unused entries can be marked by a special inode_number like 0 or an empty name). When a new file or subdirectory is created, a new entry is added to the parent directory's data. For simplicity, you can treat directory blocks as unordered lists of entries (linear search is fine for this scale). There's no need for `.` or `..` entries (unless desired) since we are not implementing a full POSIX filesystem environment.

- **Inode Segment Metadata:** Inode segments are the files that store multiple inodes. Each inode segment should include:
  - A **free inode bitmap** marking which inode slots in that segment are in use. This can be an array of bits (e.g. 256 bits for 256 possible inodes per 1 MB segment).

A simple approach is to allocate one 4096-byte block at the start of the segment as the bitmap (even though only a few bytes are needed, this simplifies alignment).
    - An **inode array** of structures, each occupying one 4096-byte block. The first inode of the very first segment is reserved for the root directory. In memory, you might represent an inode segment as a `struct InodeSegment` containing an array of inode structs and the bitmap (or pointers to them). When the program runs, it can load the segment file's contents into this structure or perform direct file I/O to read/write specific inode blocks as needed.
    - **Segment linking:** The filesystem may consist of multiple inode segment files if capacity grows. The spec suggests treating them like a linked list, using the file naming convention to chain segments. For example, name inode segment files sequentially (`inode_seg0.bin`, `inode_seg1.bin`, etc.) and in each segment store a reference or simply rely on the naming scheme to find the next segment. The code can maintain a list or array of open inode segments. If an operation requires an inode number beyond the first segment's range, load or create subsequent segment files accordingly.
- **Data Segment Metadata:** Data segments store file content blocks. Each 1 MB data segment includes:
    - A **free block bitmap** at the start, indicating which 4096-byte data blocks in that segment are free or allocated. Like in inode segments, this can be implemented as a bit array (with one bit per block). If a segment holds 256 blocks, the bitmap needs 256 bits (32 bytes); rounding this to a whole block (4096 bytes) simplifies handling.
    - **Data blocks:** The rest of the segment consists of 4096-byte blocks available for file data. For example, if one block is used for the bitmap, ~255 blocks remain for data in each segment. Blocks are referenced by the pointers in inodes. The code should map a block reference to a specific segment file and offset. A convenient scheme is to number data blocks globally in the order of segments (so block 0-254 in segment0, 255-509 in segment1, etc.), or use a pair (segment_index, block_index) internally.
    - **Segment expansion:** The code should handle multiple data segments. Keep track of all existing data segment files (e.g. `data_seg0.bin`, `data_seg1.bin`, ...). The filesystem starts with one data segment; when adding a file and no free block is available in existing segments, a new 1 MB data segment file is created and its free-block bitmap is initialized. The new segment is then added to the list of segments in the FS metadata.
- **Global Filesystem State:** It's useful to have an in-memory structure aggregating global state, for example:

```
c
Copy
struct ExFS2 {
    SegmentList inode_segments;
    SegmentList data_segments;
    // possibly other info, like open file descriptors or counters
};
```

This can hold arrays or linked lists of loaded segment info (each element containing the bitmaps and data or inode arrays, or file pointers). It can also have convenience lookups, e.g., a mapping from an inode number to its segment index and offset. This `ExFS2` context can be a singleton (global) or passed around to functions that need to access segments.

With these data structures defined, the on-disk layout is clear. For example, **segment files format:** An inode segment file begins with a bitmap and then a sequence of inode blocks; a data segment file begins with a bitmap and then a sequence of data blocks. Both types of segment can hold a fixed number of entries (inodes or blocks) based on the 1 MB size. We will use these structures to implement the required functionality.

Implementation Plan (Milestones)

Develop the file system in stages, verifying each milestone before proceeding. Below is a step-by-step plan:

1. **Filesystem Initialization (Format):** Start by implementing the ability to create and load the filesystem structure:
    - Write a function to **initialize a new FS** on disk. This will create the first inode segment file and first data segment file. Within the first inode segment, set up the free inode bitmap (mark inode 0 as used for root, others free) and create the root directory's inode at index 0. Initialize the root inode's fields (type=directory, size=0 initially) and allocate one data block for the root directory's contents (even if empty) so that it has a valid pointer to a directory block. Mark that block as used in the data segment's free block bitmap. Ensure the data segment's free-block bitmap is initialized with all other blocks free.
    - Implement **opening an existing FS**: since the FS persists in segment files, provide a way to load the state from existing segment files. This involves scanning for segment files (e.g., `inode_seg0.bin`, then `inode_seg1.bin`, etc., until none found, similarly for data segments) and loading their bitmaps and structures into memory. This will allow the tool to be run multiple times on the same FS image. For now, focus on the fresh format case, and verify that after initialization, the root directory inode exists and is correctly linked to an empty data block.
2. **Path Parsing and Navigation:** Implement utility functions to **parse paths and traverse directories**. This is crucial for almost all operations (add, extract, remove, list):
    - Create a function to split a path like "/dir1/dir2/file.txt" into components `["dir1","dir2","file.txt"]`. Also handle root (`"/"`) as a special case (which refers to the root inode).
    - Implement a **lookup function** that given a sequence of path components, starts from the root inode and navigates down the directory hierarchy to find the target. For each component, read the current directory's data block(s) and search for an entry matching the name. Use the entry's inode number to fetch the next inode (which may reside in an inode segment file). Continue until the final component is resolved or a component is not found.

- If any directory in the path is missing (and the operation is an addition that requires creating it), have a variant of the lookup that will create intermediate directories. For example, `lookup_path("/a/b/c", create_missing=true)` would create directories `/a` and `/a/b` if they don't exist, when adding a file in `/a/b/c` Directory creation involves allocating a new inode (from the free inode map) and a new data block for that directory's entries, then inserting an entry for the new directory into its parent directory.
- This path traversal logic should be carefully tested at this stage. For now, you can simulate some manual entries to test it. Ensure that lookup correctly distinguishes between file and directory inodes (e.g., if asked to navigate into a name that is a file, it should error). By implementing path traversal early, later operations (which all rely on it) will be easier to integrate.

3. **File/Directory Creation (Add operation):** Implement the `-a` (add) functionality to add a file (and any needed directories) into the FS:
   - **Add Directory:** Reuse the path traversal with create-if-missing for intermediate path components. If the final target is a directory (perhaps `-a` could also be used to create an empty directory, though the spec focuses on adding files), handle accordingly by creating an empty directory inode and block.
   - **Add File:** If the final component is a file to create, allocate a new inode for it. Find the first free inode slot by scanning inode segments in order (expand by creating a new inode segment if none free). Mark it used in the bitmap and initialize the inode (type=file, size=0 initially). Next, copy the external file's data into the filesystem:
     - Open the source file from the host OS (using the `-f /local/path` provided). Read it in 4096-byte chunks. For each chunk, allocate a free data block from the data segments (scan existing segments' free-block bitmaps for a 0 bit; if none free, create a new data segment file and update FS state). Mark the block as used and write the chunk to that block in the segment file.
     - As you allocate blocks, record their identifiers in the new file's inode. Use direct pointers for the first blocks. If the file exceeds the number of direct pointers, allocate an indirect block: obtain a free data block to serve as the single-indirect block, and start filling it with additional block addresses. If those overflow, allocate a double-indirect block, and so on for triple indirect if implemented. (Triple indirect might be optional; you can choose to support up to double indirect which should handle moderately large files.)
     - Update the file inode's size to reflect the number of bytes copied.
   - After writing file data, **update the directory entries**: Add an entry for the new file into its parent directory's data block. This involves writing the child's inode number and name into the parent directory's data. If the parent directory's current data block is full (no free entry slots), allocate a new data block for the directory (and link it via the directory inode's indirect pointers if needed, similar to file growth).
   - Verify the add operation by checking that the file's content blocks are correctly written and that you can find the file via the directory entry. At this stage, you can

test by adding a small text file and later implementing extract to verify the contents.

4. **Directory Listing (List operation):** Implement the `-l` functionality to list directory contents (or the whole filesystem tree):
    - For `exfs2 -l` with no path, the expected output is a recursive listing of the entire FS starting at root. Implement a function that given a directory inode, reads its directory entries (which may span multiple data blocks) and prints each entry. For each subdirectory, recursively list its contents, indenting the output for a tree view.
    - Use the path traversal function to find the target directory inode (for `-l /some/path` if a path is provided; if none, use root). Then read that directory's entries and output them. For subdirectories, you might print their name and a "/" to distinguish them, then recursively list their children.
    - Make sure to handle formatting nicely (e.g., indentation per depth as suggested in the spec). This is mostly a user-interface task; the core is reading directory blocks and interpreting entries.
    - Testing: After adding files and directories in previous steps, running the list command should show the correct hierarchy. For example, after `-a /a/b/c -f file.txt`, doing `-l` should show directory "a" containing subdir "b", containing subdir "c", containing file "file.txt". This confirms that directory creation and entry insertion from step 3 worked.
5. **File Reading (Extract operation):** Implement the `-e` option to read a file's content out of the FS (to stdout or a host file):
    - Use path traversal to locate the target file's inode (the path given after `-e`). Once you have the inode, open/create an output file or prepare to write to stdout as per the CLI usage.
    - Read the data by following the inode's pointers: for each direct pointer that is set, fetch the corresponding 4096-byte block from the appropriate data segment and write it out. For indirect pointers: if the single indirect pointer is set, read that indirect block (which contains a list of block numbers) and then read each referenced block. Similarly handle double indirect (two levels of indirection: one block points to a set of indirect blocks, each of which point to data blocks).
    - Write out bytes until the file size is exhausted (the last block may be only partially used, so use the size to know how many bytes to output from the final block).
    - This operation should essentially be the inverse of the add operation's data writing. After implementing this, test by extracting a file that was added and comparing the output with the original source file to ensure fidelity.
6. **File/Directory Removal (Remove operation):** Implement the `-r` command to remove a file or directory:
    - **Remove file:** Traverse to the file's parent directory and find the directory entry for the target file. Remove that entry (which could be done by marking it free or deleting it and compacting the list – simplicity: perhaps swap it with the last entry and shorten the list, or set a flag). Free the file's inode and all data blocks. This means:
        - Mark the inode as free in the inode bitmap (so it can be reused) and clear its structure (for safety).

        - Free all data blocks used by the file: for direct blocks, mark them free in the data segment bitmaps; for indirect blocks, you need to read those blocks and free the blocks they point to, then free the indirect block itself as well.
    - **Remove directory:** If the path refers to a directory, and the spec allows recursive deletion, then remove all entries inside it (which entails removing all files and subdirectories recursively). The spec specifies that if the target is a directory, you should remove that directory and *all files contained within it*. Implement this by first listing the directory's entries, and for each entry, call the remove procedure (recursive depth-first deletion). After clearing a directory's contents, free its inode and data blocks similarly to a file.
    - Handle edge cases: do not remove the root directory itself (the tool should probably prevent `-r /`). Also, as specified, when removing a single file, you should **not automatically remove parent directories** if they become empty. Only directories explicitly targeted by `-r` are removed. This means after removing a file, if its parent directory is empty, it remains (empty).
    - Test removal by adding then removing a file: ensure the file is gone from the directory listing and that the freed inode and blocks can be reused by a subsequent add. Also test removing a directory containing files (e.g., remove `/a/b` which has some files inside) – the operation should remove everything under `/a/b` and `/a/b` itself.
7. **Debugging and Verification (Debug Info, `-D`):** Implement the `-D` flag to output internal structure info for debugging. This can print details such as:
    - Number of inode segments and data segments currently in use.
    - For each segment, how many inodes/blocks are free vs used (you can dump the bitmaps or just counts).
    - A list of all inodes with their type, size, and block pointers (useful to verify indirect pointers, etc.).
    - Any other consistency checks (e.g., verify that all allocated blocks are referenced by some inode). This debug output will help during testing to confirm that, for example, after adding files, the expected number of blocks were allocated and pointers set, or after removals the space was freed. It's primarily a development tool.
8. **Edge Cases & Polishing:** Once core functionality is working, handle any remaining issues:
    - Support extremely large files (if implementing triple indirect, ensure that works; if not, perhaps explicitly document the maximum file size supported by direct+double indirect).
    - Ensure that if the FS grows to need a new inode or data segment, the linking is handled (the code to create new segments was added in steps 3 and 5; test these by creating many files to exhaust a segment).
    - Optimize if necessary (not critical for a prototype, but you might consider caching recently used segments or using efficient lookup for free bits, etc.).
    - Write additional tests for concurrent scenarios (though the tool likely runs as single process, we don't consider multi-process access).

Each milestone should be tested thoroughly before moving to the next. This iterative approach helps isolate bugs. For example, get the path traversal and basic file create working before tackling indirect pointers; test block allocation and freeing logic in small cases before using it broadly.

Modular Design Considerations

Following the code structure and plan above, ensure the design is modular:

- **Separation of Concerns:** Keep filesystem logic (inodes, blocks, directories) separate from the CLI parsing. The `main.c` should ideally just interpret arguments and call higher-level functions like `fs_add(path, src_path)`, `fs_list(path)`, `fs_remove(path)`, etc. This separation allows you to test those functions directly without command-line arguments.
- **Encapsulation:** Each module (inode, directory, data) should expose a clear interface. For example, `inode.c` can expose `alloc_inode()` and `read_inode(inode_num, struct inode *out)` and `write_inode(inode_num, const struct inode *in)` functions, without exposing how segments are managed internally. This way, if the inode storage mechanism changes, other parts of the code (like directory logic) don't break.
- **Reuse via Libraries:** Use standard C library functions and data structures when applicable. For example, `<string.h>` for manipulating names, `<stdint.h>` for fixed-size types, and standard file I/O (`fopen`, `fread`, `fwrite`, etc.) for reading/writing segment files. You might also use memory-mapped files (`mmap`) to treat a segment file as an array of structures in memory for convenience, though it's optional.
- **Naming Conventions:** Use consistent naming for segment files and perhaps store the base name or directory for them. For instance, always look for files named `exfs2_dir_seg0.dat`, `exfs2_dir_seg1.dat`, ... for inode segments, and `exfs2_data_seg0.dat`, ... for data segments. This predictable naming (or a config in the code) avoids needing a special superblock to find segments and follows the spec suggestion of using the file name to link segmentsfile-bqzri7vsrdrbjz2yw892pj.
- **Indirect Block Management:** Treat indirect blocks similarly to how files are treated – they are blocks that contain an array of block numbers. You can reuse block read/write functions to handle indirect blocks as well. Consider writing small helper functions, e.g., `add_block_to_inode(inode, block_num)` which knows how to add a block either to direct list or into an indirect block (allocating the indirect block if needed), etc. This encapsulates the pointer logic in one place.
- **Memory Management:** Clearly define which data structures live in memory versus on disk. For example, the `struct inode` in your code can mirror the on-disk format. You might read an inode block from disk into a `struct inode` in memory, modify it, then write it back. Keep track of allocated buffers and free them appropriately. Use dynamic allocation (malloc) for variable-sized things like reading a whole directory's entries if needed, and free after use to avoid leaks.

Modularity will make the system easier to debug and extend. For instance, if you wanted to add an optional caching layer or change block size, a well-structured codebase localizes such changes. It also enables focused testing of each component.

Testing Strategy

Given the complexity of a file system, a rigorous testing approach is needed:

- **Unit Testing:** Test individual modules and functions in isolation whenever possible.
  - Write a small test (either as part of the program under a debug flag, or separate test code) for the bitmap operations: allocate a sequence of inodes/blocks and then free some, ensuring the next allocation reuses freed spots. This verifies your free list (bitmap) management.
  - Test the path parser with various inputs (`"/"`, `"/foo"`, `"/foo/bar/baz"` including edge cases like trailing slashes or multiple slashes) to ensure it splits correctly.
  - Test the directory entry lookup function on an in-memory fake directory block: create a list of entries in a buffer and try finding names, to ensure string matching and handling of not found cases works.
  - If you have a function for adding a block to an inode, test it with scenarios like: add direct blocks until direct capacity exhausted, then ensure it properly allocates an indirect block, etc.
- **Integration Testing:** After implementing each major feature (add, list, extract, remove), test them in combination:
  - **Basic Scenario:** Create a new FS, add a small file to root, list the root to see the file, extract the file and compare content, then remove the file and list again to confirm it's gone.
  - **Nested Directories:** Add a file to a nested path (as in the example `-a /a/b/c -f file`), then list the whole FS to see the directory structure. Extract the file to verify content. Remove the `/a/b/c/file` and ensure that the file is removed but directories `/a/b/c` remain (now empty). Then remove the `/a/b/c` directory with `-r` and check that it deletes `c` and not `a/b` since they might be empty but not explicitly removed (depending on spec interpretation).
  - **Large File:** Add a file larger than a few blocks to force the use of indirect pointers. For example, a file of size > 50KB (which would exceed, say, 10 direct blocks). After adding, use the debug (`-D`) to inspect that the file's inode has a single-indirect block allocated. Extract the file and verify the full content is correct. If feasible, test a file large enough to use a double-indirect block as well.
  - **Many Files:** Write a loop to add, say, 300 small files (to exceed 255 inodes in one segment). Ensure that after 255 files, a new inode segment is created. Use `-D` or internal checks to verify that two inode segment files exist and that the 256th file went into the new segment. Similarly, add many blocks (via either one huge file or many files) to trigger creation of a new data segment. Verify new segment files are present and used.
  - **Consistency Checks:** After a sequence of operations, use the debug output to cross-verify consistency: number of allocated inodes matches number of files/dirs created minus removed, free block count looks correct given how many bytes of file data were added, etc. You could also write a consistency-check function that scans all inodes and builds a set of used blocks, then compares with the bitmaps.

    - **Error Handling:** Test attempts to add a file to a path where an intermediate component is a file (should error), or extracting a non-existent file (should report not found), removing a non-existent path, etc., to ensure graceful handling.
- **Memory and Resource Tests:** Run the program under tools like `valgrind` to catch any memory leaks or invalid memory accesses. Also ensure that files (segments and any local files) are properly closed after operations to avoid descriptor leaks.

During development, the **`-D` debug output is invaluable** for step-by-step verification. For example, after adding a complex directory tree, printing the entire inode table and directory entries can confirm that all links are in place as expected. Use it frequently in tests.

By following this roadmap and testing plan, you will incrementally build up a working ExFS2 file system. Each component can be verified on its own, and by the end, the integration of all features will result in a robust implementation compliant with the provided specification. Good modular design and thorough testing will ensure that the ExFS2 tool reliably supports adding, listing, reading, and removing files and directories as intended. Enjoy the process of bringing this file system to life!