FS.c uses bitmap-based allocation for inodes and data blocks. Rather than store the bitmap information on disk, these allocation bits are recreated at mount time by scanning all extant inodes and marking the data blocks they point to. This allows the filesystem to have an accurate view of its used and free space with no additional metadata that must be kept consistent.

This approach simplifies the on-disk layout of the filesystem while still maintaining efficient and reliable space allocation. By dynamically rebuilding the bitmaps, FS.c avoids redundancy and reduces the complexity of the disk management, making the implementation more understandable and easier to maintain.

Mounting and Initialization

During the mounting phase, FS.c loads the filesystem disk image into memory and validates its structure before allowing any file operations. As part of this process, internal data structures such as the inode table, data blocks, and the in-memory open file table are initialized. The allocation bitmaps for inodes and data blocks are then reconstructed by scanning existing inodes and their referenced blocks, ensuring that the in-memory view of the filesystem accurately reflects the on-disk state prior to use.

File Access and Operations

The basic file operations inherited from the file system include opening of files, reading of file contents, listing of directory entries, and displaying of file data. The opened files are tracked with an in-memory open file table that keeps track of each file by its associated inode number and by the current read offset.

All file access goes via inode-based reads. These translate a logical file offset into the appropriate physical data block; this permits the filesystem to abstract away low-level disk access, while still supporting sequential file access semantics.

Additional Functionality

As an extension, FS.c also contains functionality that randomly selects files from the root directory and prints their contents after applying a simple text transformation to them. This feature demonstrates how to do directory traversal, random selection of files, and file reading, and it also illustrates how file data can be processed in memory without actually affecting the file system state.

Error Handling

FS.c follows basic error validation principles by reporting an error in case of attempts to access operations that would act on non-existent files or using illegal file descriptors. This is a good practice to avoid undefined behavior and ensure safe usage of the filesystem. However, mechanisms such as exception handling, file writing support, permission

enforcement, or concurrency control are deliberately not implemented. The simplicity in design keeps the implementation focused on core filesystem concepts rather than full production-level robustness. Conclusion SimpleFS is a very good example to illustrate several of the basic concepts of any filesystem: inode management, block allocation, directory traversal, and path resolution. Although it is limited in functionality, FS.c provides a very sound basis for understanding how filesystems really work internally, hence a very good starting point for experimenting and extending with additional features.