**1. Introduction**

The BFS simulates a simple file system environment by managing files on a raw disk using specified block operations. This system introduces concepts like disk blocks, inodes, file descriptors, and more.

**2. System Architecture**

- 2.1. Block I/O Level (bio)

This layer interacts with the raw BFS disk represented by the BFSDISK file, which consists of 100 blocks of 512 bytes each. The `bioRead` and `bioWrite` functions allow reading and writing entire blocks of data.

- \*\*Functions\*\*:

- `i32 bioRead(i32 dbn, void\* buf)`

- `i32 bioWrite(i32 dbn, void\* buf)`

- 2.2. Filesystem Level (fs)

Provides a user-level interface to interact with files via operations like opening, reading, writing, and closing. Files can be scattered across disk blocks and do not need to be contiguous.

- \*\*Functions\*\*:

- `i32 fsFormat()`

- `i32 fsMount()`

- `i32 fsOpen(str fname)`

- `i32 fsRead(i32 fd, i32 numb, void\* buf)`

- `u32 fsWrite(i32 fd, i32 numb, void\* buf)`

- `i32 fsSeek(i32 fd, i32 offset, i32 whence)`

- `i32 fsClose(i32 fd)`

- `i32 fsSize(i32 fd)`

- `i32 fsTell(i32 fd)`

- 2.3. BFS Internal Level (bfs)

The bfs layer manages internal structures like the SuperBlock, Directory, and Inodes used to hold metadata and manage file blocks efficiently.

**3. Project Environment**

Retrieved project files from Canvas and ensured I have the following files: `alias.h`, `bfs.h`, `bfs.c`, `bio.h`, `bio.c`, `errors.h`, `errors.c`, `fs.h`, `fs.c`, `p5test.h`, `p5test.c`, and `main.c`.

**4. Functions and Interfaces**

- 4.1. bio Interface

The `bio` interface facilitates interaction at the block level.

- Used `bioRead` to load data from disk into memory.

- Used `bioWrite` to save data from memory to disk.

- 4.2. fs Interface

Acts as the user-facing API for handling files without direct concern for block management.

- 4.3. bfs Interface

Provides internal support functions crucial for the fs functionalities. These are not exposed directly to the user.

**fsRead - Implementation Explanation**

The `fsRead` function in the BFS file system reads a specified number of bytes (`numb`) from a file associated with the given file descriptor (`fd`) into a user-provided buffer (`buf`). It returns the actual number of bytes read, which might be less than `numb` if the end of the file (EOF) is reached. The function is designed to abort execution on failure, such as invalid file descriptors or block access errors.

Step-by-Step Explanation

1. \*\*Validate File Descriptor and Get Inode Number\*\*

- The function retrieves the inode number associated with the file descriptor (`fd`) using `bfsFdToInum(fd)`.

- If the inode number is invalid (less than zero), it aborts the operation with an error (`FATAL(EBADINUM)`).

2. \*\*Validate Open File Table Entry\*\*

- It checks for a valid open file table entry by calling `bfsFindOFTE(inum)`.

- The function aborts with an error (`FATAL(EOFTFULL)`) if the table is full or if the entry is invalid.

3. \*\*Get Cursor Position and File Size\*\*

- The current cursor position within the file is obtained using `bfsTell(fd)`.

- The size of the file is retrieved with `bfsGetSize(inum)`.

- If the cursor is at or beyond the file size, the function returns 0, indicating EOF has been reached.

4. \*\*Calculate Bytes to Read\*\*

- Determines the number of bytes to read, which is the lesser of the remaining file size from the cursor or `numb`.

5. \*\*Read Data Block-by-Block\*\*

- The function reads data in blocks, adhering to block boundaries:

- \*\*Determine File Block Number (FBN)\*\*: Calculates the current file block number based on the cursor and block size.

- \*\*Determine Offset and Bytes to Read\*\*: Indicates where to start within the block and the number of bytes to read up to the end of the block or until `remainingBytes` are exhausted.

- \*\*Get Disk Block Number (DBN)\*\*: Retrieves the disk block number using `bfsFbnToDbn(inum, fbn)`.

- Abort with error (`FATAL(EBADREAD)`) if the disk block number is invalid.

- \*\*Read Block from Disk\*\*: Reads the complete block using `bioRead(dbn, block)`.

- \*\*Copy Data to Buffer\*\*: Copies the relevant data segment from the block into the user buffer (`buf`).

- Updates cursors and byte counters accordingly.

6. \*\*Update Cursor and Return Result\*\*

- After successfully reading, it updates the cursor position using `bfsSetCursor(inum, cursor)`.

- Returns the number of bytes actually read.

Error Handling

The function uses `FATAL` for error handling, which immediately aborts the program if:

- The inode number is invalid.

- The open file table entry is invalid.

- A read operation fails due to an invalid block.

**fsWrite - Implementation Explanation**

The `fsWrite` function in BFS writes a specified number of bytes (`numb`) from a user-provided buffer (`buf`) into a file currently open on a given file descriptor (`fd`). The writing starts at the current file offset (cursor position). On success, the function returns the number of bytes written, while on failure, it aborts execution.

Step-by-Step Explanation

1. \*\*Validate File Descriptor and Get Inode Number\*\*

- The function starts by converting the file descriptor (`fd`) to an inode number using `bfsFdToInum(fd)`.

- If the inode number is invalid (less than zero), the function aborts with an error (`FATAL(EBADINUM)`).

2. \*\*Validate Open File Table Entry\*\*

- It verifies that a valid Open File Table entry exists using `bfsFindOFTE(inum)`.

- If the table is full or the entry is invalid, the function aborts (`FATAL(EOFTFULL)`).

3. \*\*Get Current Cursor Position\*\*

- The function retrieves the current cursor position using `bfsTell(fd)`, indicating where writing begins in the file.

4. \*\*Write Data Block-by-Block\*\*

- The function iteratively writes data in blocks:

- \*\*Determine File Block Number (FBN)\*\*: Calculates the file block number based on the current cursor.

- \*\*Determine Offset and Bytes to Write\*\*: Identifies the offset within the block and calculates how much to write within this block.

- \*\*Check and Allocate Block\*\*: If the block doesn't exist (invalid DBN), it attempts to allocate a new block via `bfsAllocBlock(inum, fbn)`. If allocation fails due to no free space, it aborts (`FATAL(EDISKFULL)`).

- \*\*Prepare and Manipulate Block Buffer\*\*:

- Initializes a block buffer and reads existing block data using `bioRead(dbn, block)`.

- Copies the relevant bytes from the buffer (`buf`) to the block buffer at the correct offset.

- \*\*Write to Disk\*\*: Outputs the modified block back to disk with `bioWrite(dbn, block)`.

- Updates the bytes written and advances the cursor accordingly.

5. \*\*Update Cursor and File Size\*\*

- After completing the write, the function updates the cursor position using `bfsSetCursor(inum, cursor)`.

- If the write operation extends the file size, it updates the file's size in the inode using `bfsSetSize(inum, cursor)`.

6. \*\*Return Result\*\*

- Finally, the function returns the total number of bytes written, signaling successful completion of the write operation.

Error Handling

The `fsWrite` function uses `FATAL` for immediate error response on detection of:

- Invalid inode numbers.

- Inadequate space in the open file table.

- Disk space exhaustion during block allocation.