File System Project

Summer 2023

Team: Vancouver
Github Name: Vijayt2001

Vijayraj Tolnoorkar, 922110069
Phillip Ma, 920556972
Janvi Patel, 917944864
Leo Wu, 917291133

Description:

The format of our file system is that we have either created or modified three main program files. The first modified program file is the fsInit.c file. The purpose of the fsInit.c file is to initialize and handle various components of the file system. Such components include the VCB (Virtual Control Block), Free Space Map, and information about the root directory. The VCB contains information about the signature of the file system and the block number where the free space starts. The Free Space Map points to the free and allocated blocks in the file system. Information about the root directory includes the filename, date, id, size, and location within the file system. Once these structures are defined, the file system would be initialized. It reads the VCB from the disk and checks whether it has already been initialized. If the file system is already initialized, it would exit the function. If it is not initialized, the function would calculate the number of blocks needed for the free space map and allocate memory. Then, the free space map and VCB are written to disk and the allocated memory is freed at the end.

The program file that needs to be created is the mfs.c file. The purpose of the mfs.c file is to implement the functions defined in the mfs.h file. The functions enable file and directory operations, such as creating a new directory with the fs_mkdir function, removing a directory with the fs_rmdir function, and determining whether a file exists in the file system with the fs_isFile function.

The last program file that needs to be modified is the b_io.c file. The purpose of the b_io.c file is to implement the file system with essential functions. There are several functions within the b_io.c file that enables interaction with the file system, which includes opening, reading, writing, seeking, and closing files. In the beginning, a structure for the FCB (File Control Block) is defined, which contains the following information about a file: metadata, a buffer, file offset, index, and buffer length. The file control blocks are initialized. A file with a specific filename is opened. Once the file is opened, a file offset is used to set the read and write position. Then, data is written from the buffer to the file and data from the file is read. These functions would return the number of blocks written and the number of bytes read. After these operations, the file is closed; and resources and the FCB are cleared.

Issues we had:

1. Sparse documentation on implementation was provided for many of the functions, so our first issue was how to create these functions based on each other's own idea of how to implement them. Communication was crucial, as VJ uploaded templates of the work he did, and the rest of the teammates worked around his idea of how to implement it. Everyone contributed, making the file directory functions off of VJ’s direction, #define and array sizes. Parsing the file leads to memory leaks and buffer overflows. This function was also written without a fully realized root directory implementation but the

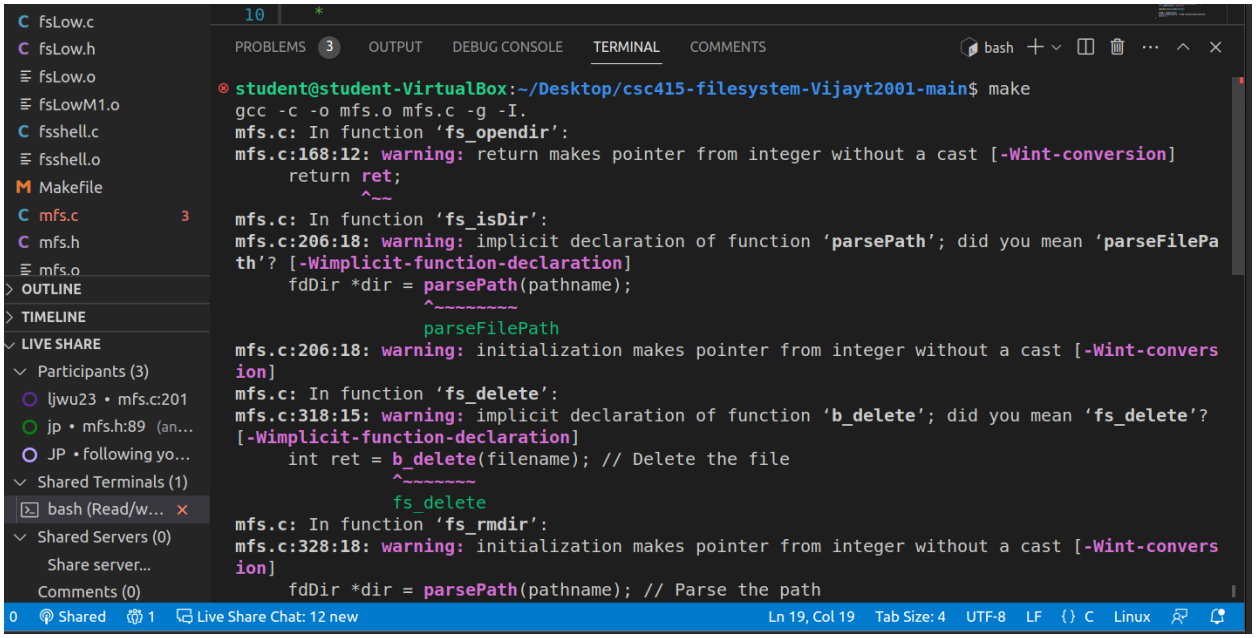
code was supported by logic and the documentation provided. Buffer overflows were solved by error-checking the length of pathname before calling strtok(), and all functions handled memory allocation properly by freeing variables like pathnameCopy within the parsing function and the arrays used.

2. We have issues starting with implementing VCB and freeSpace. Our group ran into warnings in the 'configureFileSystem' function because we were supplying an int ** to 'cleanUpMemory' rather than the intended uint64_t ** (aka long unsigned int **). This kind of mismatch has the potential to cause problems.

   We could've resolvced this by making sure that the third argument we send to 'cleanUpMemory' is of the expected kind. It appears that freespace should be of type uint64_t * rather than int *. We need to go over our code again and change the type of freespace properly.
   'b_delete' is an undeclared function in 'fs_delete':

```
student@student-VirtualBox:~/Desktop/Vancouver-main$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -o fsshell fsshell.o fsInit.o fsLow.o -g -I. -lm -l readline -l pthread
fsshell.o: In function `displayFiles':
/home/student/Desktop/Vancouver-main/fsshell.c:115: undefined reference to `fs_readdir'
/home/student/Desktop/Vancouver-main/fsshell.c:123: undefined reference to `fs_stat'
/home/student/Desktop/Vancouver-main/fsshell.c:124: undefined reference to `fs_isDir'
/home/student/Desktop/Vancouver-main/fsshell.c:131: undefined reference to `fs_readdir'
/home/student/Desktop/Vancouver-main/fsshell.c:133: undefined reference to `fs_closedir'
fsshell.o: In function `cmd_ls':
/home/student/Desktop/Vancouver-main/fsshell.c:215: undefined reference to `fs_isDir'
/home/student/Desktop/Vancouver-main/fsshell.c:218: undefined reference to `fs_opendir'
/home/student/Desktop/Vancouver-main/fsshell.c:223: undefined reference to `fs_isFile'
```

```
student@student-VirtualBox:~/Desktop/Team Vancouver$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > md aaaa
Makefile:66: recipe for target 'run' failed
make: *** [run] Segmentation fault (core dumped)
student@student-VirtualBox:~/Desktop/Team Vancouver$
```



How our driver program works:

The purpose of the driver program provided is to implement a shell, in order to perform various commands to test a file system. There are several parts within the structure of the driver program. The first part allows user input of commands. Then, the dispatch table is used to handle user commands by corresponding them with the appropriate action. The dispatch table contains the following command entries: ls, which lists the file in a directory; cp, which copies a file from a source to a destination; mv, which moves a file from a source to a destination; md, which makes a new directory; rm, which removes a file or directory; touch, which creates a file; cat, which displaces the file to the console; cp2l, which copies a file from the test file system to the linux file system; cp2fs, which copies a file from the Linux file system to the test file system; cd, which changes the directory; pwd, which prints the working directory; history, which prints out the history; and help, which prints out help. If the user types any of these commands, it would be compared to the entries in the dispatch table and be pointed to their respective command function. The command functions dictate the behavior of their respective commands. They would carry out the operation and output a result informing the user what action has been done. Before these actions can be tested, the values of the commands defined in the macros located on top of the driver program code must be changed from 0 to 1.

- ## Essential structures for our File System:

Summary of the structures: We have made a lot of modifications to our structures compared to our milestone 1 structures.

VCB:

```
typedef struct {
    uint64_t blockSize;
    uint64_t numberOfBlocks;
    //uint64_t vcbBlockCount;
    uint64_t rootDirectory;
    uint64_t firstFreeBlockIndex;
    uint64_t magicNumber;
    uint64_t freespaceBlockCount;
} vcb;
```

Explanation:

Freespace:

```
typedef struct {
   // int *bitmap;  // An array of integers acting as a bitmap
    uint64_t size; // The size of the bitmap array
} Freespace;
```

Root-directory:

```
typedef struct DirEntry {
  char name[256];
  uint64_t location;
  uint64_t size;
  uint64_t type;
} DirEntry;
```

- ## Important Functions:

1. **parseFilePath():** The purpose of the parseFilePath() function is to parse a file path and break it down into its component parts. The function takes one argument: the file path to be parsed. The function then returns an array of strings, where each string represents a component of the file path.The function first checks if the file path is absolute or relative. If the file path is absolute, the function starts parsing the file path from the root directory. If the file path is relative, the function starts parsing the file path from the current working directory. The function then iterates through the file path, one character at a

time. If the character is a /, the function adds the current component of the file path to the array of strings. If the character is not a /, the function appends the character to the current component of the file path. The function continues iterating through the file path until the end of the file path is reached. The function then returns the array of strings.

2. **fs_mkdir():** The purpose of the fs_mkdir() function is to create a new directory in the file system. The function takes two arguments: the path to the directory to be created, and the mode of the directory. The mode of the directory determines the permissions that users have to access the directory. The function first parses the path to the directory to be created. This involves breaking the path down into its component parts, such as the parent directory and the name of the new directory. The function then checks if the parent directory exists. If the parent directory does not exist, the function returns an error.

3. **fs_rmdir():** The purpose of the fs_rmdir() function is to remove a directory in the file system.

4. **fs_opendir():** The purpose of the fs_opendir() function is to open a directory in the file system. The function takes one argument: the path to the directory to be opened. The function then returns a pointer to the inode for the directory.The function first calls the b_open() function to open the directory. If the b_open() function fails, the fs_opendir() function returns NULL. If the b_open() function succeeds, the fs_opendir() function calls the getInode() function to get the inode for the directory. The getInode() function returns a pointer to the inode for the directory.

5. **fdDir:** The purpose of the fdDir function is to maintain information on a file or directory entry.

6. **closeDir():** The purpose of the fs_closedir() function is to close a directory that has been opened in the file system. It checks whether the directory pointer to a directory structure is valid and frees memory from the directory pointer.

7. **fs_getcwd():** The purpose of the fs_getcwd() function is to get the current working directory. It returns a pointer to ‘buf’ array which would have the current working directory. A buf(buffer) and len(length) are passed to the method. If the current working directory’s length is less than or equal to the specified length then it copies the path to the buffer and returns it or else it would show an error message and return NULL if the current directory is too long.

8. **fs_setcwd():** The purpose of the fs_setcwd() function is to update the current working directory of the file system to another directory specified. The function takes in one

variable, a buffer. The buffer is checked to ensure that it is not NULL. Then, a pointer is obtained to the new directory specified by a path in the buffer. The function updates the pointer to point to the new working directory and deallocates the previous working directory.

9. **fs_isFile():** The purpose of the fs_isFile() function is to check whether an entry in a path to the file system contains a valid file. The result is that either the file exists or does not exist.

10. **fs_delete():** The purpose of the fs_delete() function is to delete a file from the file system. The function takes one argument: the path to the file to be deleted. The function then returns 0 if the file was deleted successfully, and -1 if an error occurred. The function first calls the getInode() function to get the inode for the file to be deleted. The getInode() function returns a pointer to the inode for the file. The function then calls the removeFromParent() function to remove the file from its parent directory. The removeFromParent() function returns 0 if the file was removed successfully, and -1 if an error occurred.

Screenshots and Explanations:

```
student@student-VirtualBox:~/Desktop/csc415-filesystem-Vijayt2001-main$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
fsInit.c: In function ‘configureFileSystem’:
fsInit.c:71:62: warning: passing argument 3 of ‘cleanUpMemory’ from incompatible pointer type
[-Wincompatible-pointer-types]
         if (!readBuffer) return cleanUpMemory(NULL, &ourVCB, &freespace, NULL), -1;
                                                              ^
fsInit.c:30:6: note: expected ‘uint64_t ** {aka long unsigned int **}’ but argument is of typ
 ‘int **’
 void cleanUpMemory(char **buffer, vcb **vcbVar, uint64_t **freeSpace, fdDir **directory);
      ^~~~~~~~~~~~~
fsInit.c:76:64: warning: passing argument 3 of ‘cleanUpMemory’ from incompatible pointer type
[-Wincompatible-pointer-types]
         if (!fsCWD) return cleanUpMemory(&readBuffer, &ourVCB, &freespace, NULL), -1;
                                                                ^
fsInit.c:30:6: note: expected ‘uint64_t ** {aka long unsigned int **}’ but argument is of typ
 ‘int **’
 void cleanUpMemory(char **buffer, vcb **vcbVar, uint64_t **freeSpace, fdDir **directory);
      ^~~~~~~~~~~~~
gcc -c -o mfs.o mfs.c -g -I.
mfs.c: In function ‘fs_delete’:
mfs.c:325:15: warning: implicit declaration of function ‘b_delete’; did you mean ‘fs_delete’?
[-Wimplicit-function-declaration]
     int ret = b_delete(filename); // Delete the file
               ^~~~~~~~
               fs_delete
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o mfs.o b_io.o fsLow.o -g -I. -lm -l readline -l pthread
fsshell.o: In function `main':
/home/student/Desktop/csc415-filesystem-Vijayt2001-main/fsshell.c:785: undefined reference to
`initFileSystem'
```