File System Project

Group Name: BeerMan

Members in Group:
Andy Almeida - 922170012
Masen Beacham - 918724721
Christian Mcglothen - 918406078
Kendrick Rivas - 922898506

Github:

https://github.com/CSC415-2023-Summer/csc415-filesystem-pie240

Game Plan/Understanding of Project:

What is a file system?

A file system is a crucial component of any operating system that manages how data is stored, organized, and retrieved on storage devices like hard drives, solid-state drives, and more. It provides a hierarchical structure for organizing files and directories (also known as folders) in a way that allows easy access, storage, and management of data.

Let's explore how a file system works, step by step:

1. **Storage Media**: File systems are designed to work on specific types of storage media, such as hard disk drives (HDDs), solid-state drives (SSDs), USB drives, or even virtual disk images. Each storage medium has its characteristics, such as capacity, speed, and access times.
2. **Partitioning:** The storage media is divided into partitions, which are separate logical divisions of the disk. Each partition can have its file system. Partitioning allows you to isolate data, have different file systems, or use different operating systems on the same physical disk.
3. **Formatting:** Before using a partition, it needs to be formatted with a specific file system. Formatting creates the necessary data structures on the partition to enable file storage and retrieval. Common file system formats include FAT32, NTFS (Windows), ext4 (Linux), APFS (Apple), and many others.
4. **Data Organization:** The file system organizes data into two main components: files and directories.
    - **Files:** A file is a unit of data that can be a document, an image, a program, or any other type of data. Files are stored on the storage media as a sequence of binary data.
    - **Directories:** A directory (folder) is a container for files and other directories. It provides a hierarchical structure that allows files to be organized in a tree-like manner.
5. **File System Metadata:** The file system maintains metadata for each file and directory. Metadata includes information like file/directory names, creation date, modification date, file size, permissions, and pointers to the actual data on the storage media.
6. **File Allocation Methods:** File systems use different methods to allocate space for files on the storage media. Common methods include:
    - **Contiguous Allocation:** Files are stored as contiguous blocks of data on the disk. This method can lead to fragmentation, where free space becomes scattered.
    - **Linked Allocation:** Files are stored as a linked list of blocks. Each block points to the next block in the file. This method reduces fragmentation but can lead to overhead due to maintaining pointers.
    - **Indexed Allocation:** A separate index block contains pointers to blocks of data for each file. This method reduces fragmentation and overhead but can lead to wasted space for small files.

7. **Access Control and Permissions:** File systems implement access control mechanisms to ensure data security and privacy. They define permissions for files and directories, specifying who can read, write, execute, or modify them.
8. **File System Operations:** File systems provide a set of operations to interact with files and directories, such as creating, opening, reading, writing, deleting, renaming, and moving files. These operations are performed through system calls provided by the operating system.

Overall, the file system acts as an intermediary between the user/application and the physical storage media, abstracting the complexities of low-level data storage and providing a user-friendly interface for managing data on a computer system.

What are Extents?

An extent is a consecutive range of blocks or sectors on a storage device that are allocated to store a file's data. Extents are used as a more efficient way to manage file storage compared to traditional block-based allocation methods, which can lead to fragmentation and decreased performance.

Fragmentation is where parts of a file are scattered across non-contiguous blocks on the storage medium.Fragmentation can lead to slower file access times and reduced overall performance.

Extents were introduced to address this issue and improve file system performance. Instead of allocating individual blocks or using linked lists for each file, extents group multiple contiguous blocks together to store a file's data. When a file is created or extended, the file system searches for free contiguous blocks on the disk and allocates them as a single extent to the file. This approach reduces fragmentation and enhances file access times, as the data of a file is stored in larger continuous chunks.

Description of the Directory system:

Our directory system starts at the beginning of the storage space with our Volume Control Block. Our VCB contains the location for the root block. With this location, we can begin to allocate our Directory Entries. The directory entries contain the data on the information that relates to the blocks containing the files/directories saved in our file system. The directory entries contain data such as the file name, the file type, file size, metadata (creation, last modified, last opened), and very importantly, its location block in the file system. With the use of Directory Entries, we can keep track of files and their information/location. Directory Entries on directories would simply point us to a block that contains another list of Directory Entries. This is the basis for our File System.

A description of the VCB structure:

The volume control block serves as a main point of information regarding the state of the volume that we are allocating for space. To complete this, the VCB contains fields that help the

system figure out its most important data. In our VCB, we have the signature which is a field that will help us ensure that our volume control block is correctly allocated as expected, followed by [root] which holds the number indicating where the directory starts, [totalBlocks] which indicated the total number of blocks in the system, [block_size] which indicates the size of a block in bytes, and [free_space_start_block] which indicates the block at which the free space starts. All fields except for the signature holds its value as an int, whereas the signature is in the form of an unsigned long.

A table of who worked on which components:

Component	Contributors

char * fs_getcwd(char *pathname, size_t size);
 	
int fs_rmdir(const char * pathname);
 	
char * fs_setcwd(char *pathname, size_t size);
 	
int fs_isFile(char * filename);
 	
fdDir * fs_opendir(const char * pathname);
 	
struct fs_dirItemInfo * fs_readdir(fdDir * dirp);
 	
fs_isFile(filename);
	Christian

fs_readdir(fdDir) fs_isDir(char * pathname); fs_closedir(fdDir)	**Masen**
parsePath(Directory Entry, PathInfo) method	**Andy**
Directory Entry	Andy
VCB	Andy
Init.h	Christian, Andy, Masen, Kendrick (Given By Bierman, Group added dates to structs)

Init.c - initFileSystem	Andy
Init.c - initRootDirectory	Andy
FreeSpace.c	Christian
b_io.c	Masen, Christian(only help with b_open)
PDF Milestone 1 Write-Up/ Final Write-Up	Andy, Christian

Description of FreeSpace.c:

Our freespace system has a functionality of allocating and releasing free space. It uses a simple bitmap representation to track the availability of blocks and provides functions to allocate and release contiguous blocks as needed.

void init_bitmap(Bitmap* bitmap)

Goal: This function initializes a given Bitmap structure by setting all its bits to 0.

Description: The function iterates through the data array of the Bitmap structure and sets each element to 0. The Bitmap structure represents a bitmap, where each bit represents the usage status of a block or element.

void set_bit(Bitmap* bitmap, int bit_index)

Goal: This function sets a specific bit at the given index to 1 in the provided Bitmap structure.

Description: The function calculates the element_index and the offset of the target bit based on the bit_index. It then uses bitwise OR (|) to set the bit at the calculated offset in the element_index to 1.

void clear_bit(Bitmap* bitmap, int bit_index)

Goal: This function clears a specific bit at the given index (sets it to 0) in the provided Bitmap structure.

Description: Similar to set_bit, this function calculates the element_index and the offset of the target bit based on the bit_index. It then uses bitwise AND with the negation (~) of a bit mask with a 1 at the calculated offset to set the bit at the element_index to 0.

bool test_bit(Bitmap* bitmap, int bit_index)

Goal: This function checks if a specific bit at the given index is set to 1 in the provided Bitmap structure.

Description: Like the previous two functions, this one calculates the element_index and the offset of the target bit based on the bit_index. It then uses bitwise AND (&) to check if the bit at the calculated offset in the element_index is 1. If it is, the function returns true; otherwise, it returns false.

int InitFreeSpace()

Goal: This function initializes the free space management system using a bitmap to track block usage.

Description: It first calls init_bitmap() to initialize the FreeSpaceMap (bitmap) with all bits set to 0. Then, it sets the first 6 bits (corresponding to blocks 0 to 5) to 1, indicating that they are used (VCB and FreeSpaceMap blocks). After this setup, it writes the FreeSpaceMap to disk and returns the index of the first free block after the VCB and FreeSpaceMap.

int SearchForSpace()

Goal: This function searches for the first unused (0) bit in the FreeSpaceMap, indicating an available block.

Description: It reads the FreeSpaceMap from disk, and then it iterates through the bitmap from index 6 onwards to find the first bit set to 0. If it finds one, it returns its index, which represents the index of the first available block. If no unused block is found, it returns -1 to indicate that storage is full.

int GetFreeSpace(int NumberOfBlocks)

Goal: This function searches for a contiguous block of a specified number of unused blocks in the FreeSpaceMap.

Description: It reads the FreeSpaceMap from disk and then iterates through the bitmap from index 6 onwards. It looks for a sequence of NumberOfBlocks consecutive bits set to 0 (unused blocks). If it finds such a sequence, it returns the starting index of that sequence, representing the index of the first block in the contiguous free space. If no contiguous block of the required size is found, it returns -1 to indicate that there is not enough free space.

int ReleaseSpace(int Location, int NumberOfBlocks)

Goal: This function releases a specified number of blocks, starting from the given Location, by setting their bits to 0 in the FreeSpaceMap.

Description: It reads the FreeSpaceMap from disk and then iterates through the bitmap from the specified Location to Location + NumberOfBlocks - 1. For each block, it uses clear_bit() to set its corresponding bit to 0, indicating that the block is now available. Finally, it writes the updated FreeSpaceMap back to disk and returns 0 to indicate successful release of space.

Description of fsInit.c:

initFileSystem(uint64_t numberOfBlocks, uint64_t blockSize):

Goal: Initialize the file system.

Description: This function is responsible for initializing the file system. It first checks if the file system has been previously initialized by reading the VCB (Volume Control Block) from the disk. If the signature in the VCB matches the predefined signature (indicating that the file system has already been initialized), it loads the root directory and sets the current working directory to the root. If the signature doesn't match, it initializes the VCB with the provided number of blocks and block size, sets up the free space bitmap, initializes the root directory, and writes the VCB to the disk.

InitRootDirectory():

Goal: Initialize the root directory.

Description: This function is responsible for initializing the root directory. It allocates memory for the root directory, sets its initial values, including "." and ".." entries, and writes it to the disk. It returns the block number of the first block used by the root directory.

exitFileSystem():

Goal: Clean up and exit the file system.

Description: This function is called when the file system is exiting. It can be used to clean up any resources or perform other necessary operations before the program terminates.

Description of mfs.c:

fs_mkdir(const char *pathname, mode_t mode):

Goal: Create a new directory at the specified pathname.

Description: This function creates a new directory at the provided pathname. It first parses the path to ensure it is valid. Then it finds free space to store the new directory's information, updates the parent directory entry with the new directory information, initializes the new directory with appropriate values, and writes it to the disk.

fs_rmdir(const char *pathname):

Goal: Remove an existing directory from the file system.

Description: This function removes the directory specified by the pathname. It parses the path to find the target directory and then deallocates its space on the disk. It also updates the parent directory entry to remove the reference to the deleted directory.

fs_opendir(const char *pathname):

Goal: Open a directory for reading its contents.

Description: This function opens the directory specified by the pathname and returns a pointer to an internal data structure (fdDir) used to iterate through the directory entries.

fs_readdir(fdDir *dirp):

Goal: Read the next directory entry from an opened directory.

Description: This function reads the next directory entry from the directory pointed to by the provided fdDir pointer. It returns information about the current directory entry, such as its name and type (file or directory). It also updates the fdDir data structure to point to the next entry.

fs_closedir(fdDir *dirp):

Goal: Close a directory after reading its contents.

Description: This function closes the directory pointed to by the provided fdDir pointer. It releases any dynamically allocated memory associated with the directory reading process.

fs_getcwd(char *pathname, size_t size):

Goal: Get the current working directory.

Description: This function retrieves the current working directory and stores it in the provided buffer pathname. The size parameter specifies the size of the buffer to prevent buffer overflow.

fs_setcwd(char *pathname):

Goal: Set the current working directory.

Description: This function sets the current working directory to the directory specified by the provided pathname. It updates the global variable cwd (current working directory) to reflect the new working directory.

fs_isFile(char *filename):

Goal: Check if a given entry is a file.

Description: This function checks if the entry specified by the given filename exists and is a file (not a directory). It returns 1 if it's a file and 0 otherwise.

fs_isDir(char *pathname):

Goal: Check if a given entry is a directory.

Description: This function checks if the entry specified by the given pathname exists and is a directory. It returns 1 if it's a directory and 0 otherwise.

fs_delete(char *filename):

Goal: Delete a file from the file system.

Description: This function deletes the file specified by the given filename from the file system. It deallocates the space used by the file and updates the directory entry for the parent directory.

fs_stat(const char *path, struct fs_stat *buf):

Goal: Get file status information.

Description: This function retrieves the status information of the file specified by the given path and stores it in the struct fs_stat pointed to by buf. This information includes details like file size, type, and timestamps.

FindEntryInDir(DirectoryEntry *parent, char *token):

Goal: Find the index of a given entry in a directory.

Description: This function searches for a given entry (token) in the specified parent directory. It returns the index of the entry if found, or -1 if not found.

loadDir(DirectoryEntry parent):

Goal: Load a directory from disk.

Description: This function loads a directory from disk, given its parent directory entry. It allocates memory for the directory, reads its contents from disk, and returns a pointer to the loaded directory.

isDir(DirectoryEntry directoryEntry):

Goal: Check if a directory entry represents a directory.

Description: This function checks if the provided directory entry represents a directory. It returns true if it's a directory and false otherwise.

isUsed(DirectoryEntry de):

Goal: Check if a directory entry is in use.

Description: This function checks if the provided directory entry is in use, i.e., it represents a valid file or directory. It returns true if the entry is in use and false if it's empty (not in use).

Description of b_io.c:

b_fcb struct:

Goal: This struct represents the file control block (FCB) for a buffered file.

Description: The b_fcb struct contains various attributes related to an open file, including the file's directory entry (fi), current location within the file (currLocation), remaining blocks to read/write (blockRemainder), remaining bytes to read/write in the current buffer (sizeRemainder), and other buffer-related information (buf, index, and buflen).

fcbArray array:

Goal: This array holds the file control blocks for various open files.

Description: The fcbArray array is used to manage the FCBs for different open files. Each element of the array (fcbArray[i]) corresponds to an open file, and it contains the relevant information about that file, as defined in the b_fcb struct. If an element of the array has a NULL buffer (buf), it indicates that the corresponding FCB is not in use and can be allocated for a new file.

startup variable:

Goal: This variable indicates whether the system has been initialized.

Description: The startup variable is used to track whether the system has been initialized using the b_init() function. It is set to 0 initially and becomes 1 after the system initialization.

b_init() function:

Goal: Initialize the file system.

Description: This function initializes the file system by setting all elements of the fcbArray to have a NULL buffer, indicating that all FCBs are available for use. It also sets the startup variable to 1, indicating that the system has been initialized.

b_getFCB() function:

Goal: Get a free FCB element from the fcbArray.

Description: This function iterates through the fcbArray and returns the index of the first FCB element with a NULL buffer, indicating that it is available for use. If all elements are in use, it returns -1.

b_open(char * filename, int flags) function:

Goal: Open a buffered file and obtain a file descriptor for it.

Description: This function opens a file with the specified filename and the given flags, which determine the access mode (read-only, write-only, or read-write) for the file. It uses the open() system call to obtain the file descriptor. The function also allocates memory for the file buffer in the corresponding FCB (fcbArray[fd].buf) and initializes other relevant FCB attributes. It returns a file descriptor (a value from 0 to MAXFCBS-1) on success or -1 on failure.

b_seek(b_io_fd fd, off_t offset, int whence) function:

Goal: Move the file pointer to a specified position within the file.

Description: This function performs a seek operation on the file with the given file descriptor fd. It uses the lseek() system call to move the file pointer to the desired offset position relative to the position specified by whence. The function returns 0 on success or -1 on failure.

b_write(b_io_fd fd, char * buffer, int count) function:

Goal: Write data to a buffered file.

Description: This function writes count bytes from the buffer to the file with the given file descriptor fd. It uses the write() system call to perform the write operation. The function returns the number of bytes written on success or -1 on failure.

b_read(b_io_fd fd, char * buffer, int count) function:

Goal: Read data from a buffered file.

Description: This function reads up to count bytes from the file with the given file descriptor fd and stores them in the buffer. It uses the read() system call to perform the read operation. The function returns the number of bytes read on success or -1 on failure.

b_close(b_io_fd fd) function:

Goal: Close a buffered file and release associated resources.

Description: This function closes the file with the given file descriptor fd and frees any memory allocated for the file buffer in the corresponding FCB. It also marks the FCB as available for reuse by setting its buffer to NULL. The function returns 0 on success or -1 on failure.

Description of fsshell.c

Goal: The goal of the code is to implement a simple command-line interface for interacting with a custom file system. The code defines various commands, such as ls, cp, mv, md, rm, touch, cat, cp2l, cp2fs, cd, pwd, history, and help, that allow users to perform operations on the custom file system.

Description:

The code includes several header files, standard libraries, and custom file system-related files.

It defines various macros for enabling or disabling specific commands (e.g., CMDLS_ON, CMDCP_ON, etc.). By setting these macros to 1, the corresponding command is enabled; otherwise, it is disabled.

The code defines a structure dispatch_t to hold information about each command, including its name, function pointer, and description.

Each command is implemented as a separate function (e.g., cmd_ls, cmd_cp, etc.), and each function corresponds to a specific operation on the custom file system.

The displayFiles function is used to list files in a directory, and it is called by the cmd_ls function.

The main function sets up the custom file system, initializes it, and then enters a loop to process user commands.

Within the loop, the user input is read using readline, and the command is processed through the processcommand() function.

The processcommand function tokenizes the user input and matches it with the available commands, executing the corresponding command if it exists.

The main function also includes an option to run low-level tests (runFSLowTest) if provided as a command-line argument.

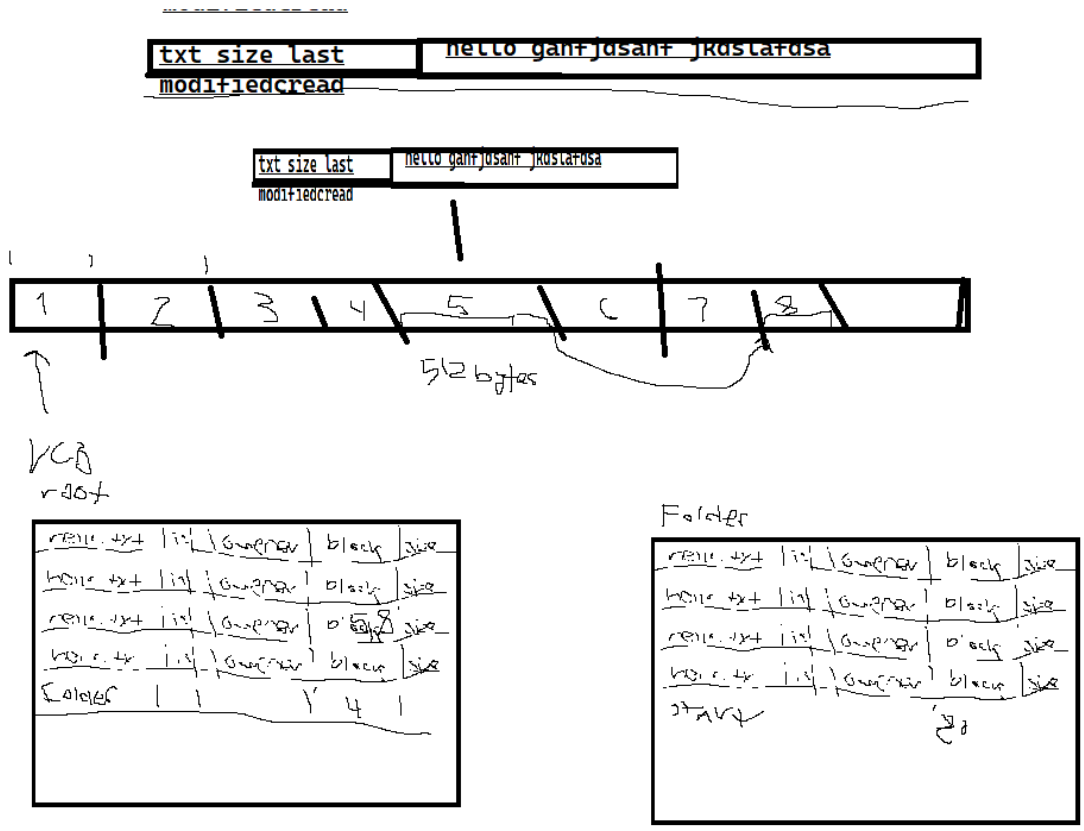
How did your team work together, how often you met, how did you meet, how did you divide up the tasks:

Our team is very active on this project. We have a solid team of dedicated workers in our group. We have had everyone meet one time in person, not counting the times before or in class. We have also all met in discord two times. Through these meetings we tend to work together on one machine, discussing and collaborating ideas, having one person write those thoughts down in code. The remaining work for the milestone ended up getting filled up by the members who took it upon themselves to complete it. Our group is focused, and the work that is required is being completed by whoever notices it needs to be done. This is how this group has successfully divided up the tasks.

A discussion of what issues you faced and how your team resolved them:

The first issue we have faced as a group was building the necessary structures such as the Directory Entry and the Volume Control Block. The directory entry was pretty straightforward and relatively easy since we were able to get feedback on what we had and modify it accordingly. The Volume Control Block was a bit harder since we had very little knowledge of how a VCB comes into play in the entire File System. Two of our team members went into Office Hours in hopes of clearing the air up. Along with notes taken from class, we were able to eventually come up with a structure that we believe makes sense for our project.

The second issue we ran into started after the structures were built. As a group, we had very little understanding of what a File System even looked like. We had some conflicting ideas of how to approach the whole project. We had some saying it would be best to learn as we go and try to start coding. Others proposed that it would be a good idea to spend some time reading and learning about what a File System is. Ultimately we agreed that we should allocate some time to read and study up on what a File System is. This helped our group get some sense of direction and understanding of the task at hand. This helped us move forward with the implementation of our file system. Attached below is a small drawing one of our group members made to help the other members of the team understand block and memory allocation. The rectangle above was meant to signify the bytes of data, with the header at the start. The rectangle with numbered blocks was meant to signify the memory space that contains the blocks of data. The two squares below were created to signify the volume control blocks that would contain directory entries.



The third issue we ran into came after we started building the initFileStructure function in the fsInit.c file. We were able to gain some ground within the function, but we hit a wall very quickly when we realized that we had to allocate and initialize the Volume Control Block. Most of the fields were fairly straightforward, but when we had to start allocating the root directory along with the “.” and “..” entries, we had some troubles. Eventually our questions were answered by the next upcoming class we went to. Our professor ended up clearing the air on how the “.” and “..” entries played a role. This led us to being able to build the initRootDirectory function for our fsInit.c file, making it successfully compile and run how it should.

The fourth issue we came across was involving extents. As a group we didn't have any prior knowledge on what it was or how to possibly implement it. During class on Monday, the professor went over examples to help get a better understanding of how to use it in this project and it helped out a lot. Instead of storing information about every single block, like we had before, we only needed to store the start block and the number of blocks that made up that extent, simplifying our space management.

What works, What does not work(Testing Project):

Commands: md hello, touch hello.txt, ls, pwd

Output:

```
Prompt > md hello
==> Make Directory <==
NewDirectoryFirstBlock:39
Free Directory Entry Found [2] -> Set Data in Parent Directory
Initializing New Directory
Direcotory Successfully Created
Prompt > touch hello.txt
Created new file: hello.txt
Prompt > ls
Opened directory: .

hello
hello.txt
No more directory entries to read.
Prompt > pwd
.
Prompt >
```

Commands: cd Hello, touch test.txt, ls, pwd

Output:

```
Prompt > cd hello
helloPrompt > touch test.txt
Created new file: test.txt
Prompt > ls
Opened directory: hello

hello
test.txt
No more directory entries to read.
munmap_chunk(): invalid pointer
Makefile:67: recipe for target 'run' failed
make: *** [run] Aborted (core dumped)
```

Issues: We receive an error when doing pwd after running an ls, if we drop the ls from the commands listed above or run pwd first, there is no error.

```
Prompt > md Hello
==> Make Directory <==
NewDirectoryFirstBlock:39
Free Directory Entry Found [2] -> Set Data in Parent Directory
Initializing New Directory
Direcotory Successfully Created
Prompt > cd Hello
HelloPrompt > touch test.txt
Created new file: test.txt
Prompt > pwd
Hello
Prompt >
```

Commands: md Hello, rm Hello

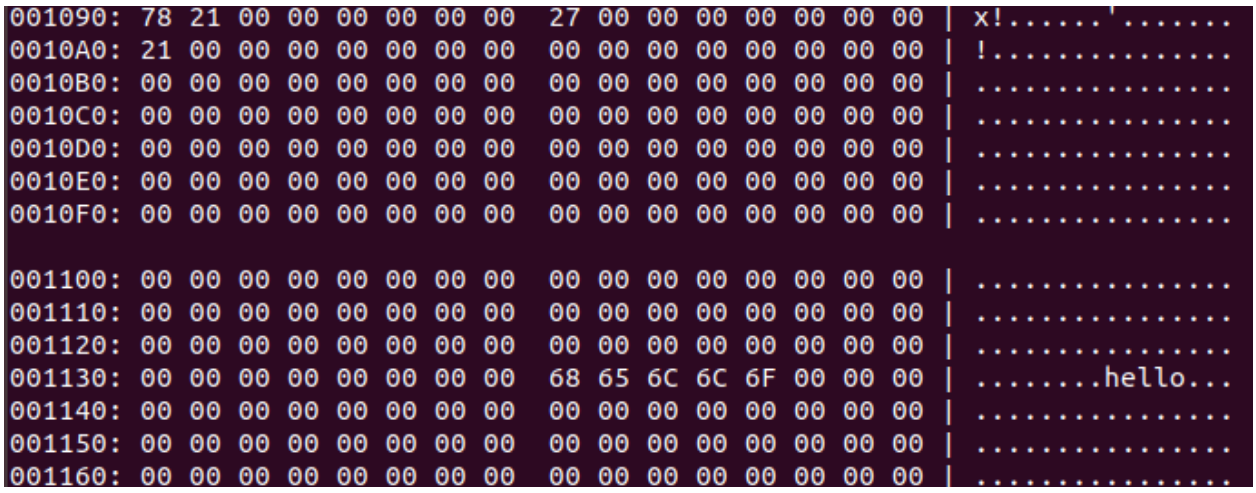
Hex Dump Output (md Hello) & (Bitmap):

```
001090: 78 21 00 00 00 00 00 00  27 00 00 00 00 00 00 00 | x!......'.......
0010A0: 21 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | !...............
0010B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

001100: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001110: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001120: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001130: 00 00 00 00 00 00 00 00  68 65 6C 6C 6F 00 00 00 | ........hello...
001140: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001150: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001160: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Bitmap:

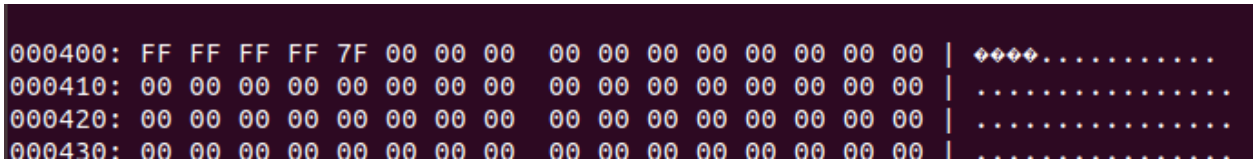
```
000400: FF FF FF FF FF FF FF FF  FF 00 00 00 00 00 00 00 | ◆◆◆◆◆◆◆◆◆.......
000410: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Hex Dump Output (md Hello) & (Bitmap):



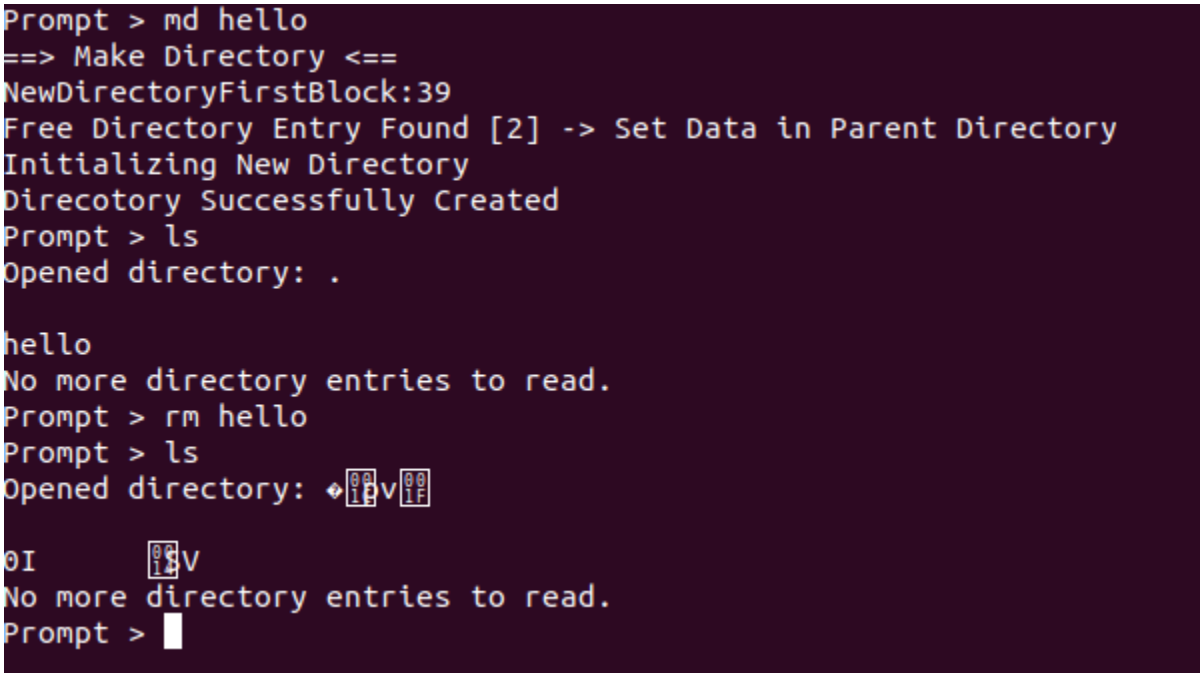
```
001090: 78 21 00 00 00 00 00 00  27 00 00 00 00 00 00 00 | x!......'.......
0010A0: 21 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | !...............
0010B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

001100: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001110: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001120: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001130: 00 00 00 00 00 00 00 00  68 65 6C 6C 6F 00 00 00 | ........hello...
001140: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001150: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001160: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Bitmap(when removing hello):



```
000400: FF FF FF FF 7F 00 00 00  00 00 00 00 00 00 00 00 | ................
000410: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000420: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000430: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Issue: The value of the bitmap changes correctly and when you make a new directory hello is overwritten by a new directory entry (which we want) but the value does not change in the Hex Dump (see above). When running md hello, rm hello, followed by “ls” you can see that the name of hello is replaced with “ “ (which we want) but again the value does not change in the hex dump until you create a new directory. A screenshot of this is provided below:



```
Prompt > md hello
==> Make Directory <==
NewDirectoryFirstBlock:39
Free Directory Entry Found [2] -> Set Data in Parent Directory
Initializing New Directory
Direcotory Successfully Created
Prompt > ls
Opened directory: .

hello
No more directory entries to read.
Prompt > rm hello
Prompt > ls
Opened directory: ◆ipv
0I      V
No more directory entries to read.
Prompt >
```

Screenshot of compilation: