Team Name: Team Poke
Class: CSC 415 -⅔ Spring 2022
Team Members:
Kilian Kistenbroker — ID: 920723372 — Section 3
Emily Huang — ID: 920499746 — Section 3
Sean Locklar — ID: 920506337 — Section 2
Shauhin Pourshayegan — ID: 920447681 — Section 3

Github Name: KilianKistenbroker
https://github.com/CSC415-2022-Spring/csc415-filesystem-KilianKistenbroker

Purpose

This project aims to bring together what we've learned in class about file systems and apply that knowledge through implementation of a basic file system.

A file system is a program which resides on secondary storage which can manipulate the file structure to allow a user access to secondary storage. Its primary goal is to link the logical file system, which the user has access to, to a physical location on the disc.

Team Poke's file system is designed to hold its own directories as well as any file. These directories and files can be manipulated in terms of their location within the logical level of the file system. Directories can contain subdirectories and the file system can add new directories to itself. Files, as well as directores, can also be removed, which deletes them from the file system. Files can be copied over to and from the linux file system.

Overview

The purpose of the volume control block is to act as an in-memory structure for manipulating the VCB in storage. The VCB structure that our file system uses contains 5 data fields, all ints which make the size of the VCB structure only 24 bytes. The VCB will contain information about the filesystem's individual block size, total blocks in filesystem, starting position of the FreeSpace bitmap, starting position and size of the root directory, and the signature (magic number) of the VCB structure. The purpose of this block is to contain this information so the file system can be loaded into memory from the in-storage structure.

```
typedef struct vcb
{
    int magic_num;
    int block_size;
    int total_blocks;
    int free_block_start;
    int root_start;
    int root_size;
} vcb;
```

The file control block is an in-memory structure that is used when a file is currently open or otherwise in use. The purpose of an FCB is to contain metadata about a file so it can be manipulated by the file system. The fcb structure contains a character pointer buffer, a file descriptor to hold the file's position in the FCB array, the length of the buffer, the position in the buffer, the position in the destination folder, the position in the source folder, as well as a directory entry structure for the parent directory it is in. The buffer is a character pointer and the directory entry is a unique structure, but everything else is an int.

```
typedef struct b_fcb
{
    // holds the open file buffer.
    char * buf;

    // holds position in fcbArray.
    int file_descriptor;

    // holds the size of the buffer.
    int len;

    // holds an index to the current byte in the buffer.
    int pos;

    // holds a position in the 'dest' folder we are copying to.
    int pos_in_dest_parent;

    // holds a position in the 'src' folder we are copying from.
    int pos_in_src_parent;

    // holds the parent directory of the file.
    dir_entr * parent_dir;

} b_fcb;
```

The FreeSpace structure consists of a bitmap mapping each block within the volume to a bit in the map marking it either free or used. Upon initializing the bitmap, and after space is allocated and the first 6 bits are set to 1 to mark the blocks containing the VCB and

FreeSpace map as used, the map is then written to disk. Aside from initialization we have two other functions which contribute to the FreeSpace structure including update_free_block_start, which finds the next free block in the map, and allocate_space, which iterates through the bitmap to check if a block is free or used and marks the amount of blocks to be allocated as used. The updated bitmap needs to be written to disk and the FreeSpace start block must also be updated and written to disk.

The directory entry is an in-memory structure that contains information about the directory being referenced. The purpose is similar to that of the VCB in-memory structure, to provide an in-memory structure so that what is saved on disk can be manipulated. The structure contains 20 available filenames which act as the references to the files the directory is linked to, a start position, a size and 3 separate char arrays to hold creation, modification and access time for each file. They also contain an int to indicate whether this is a file or directory and another int called mode to designate read/write/execute permissions.

```
typedef struct dir_entr
{
    char filename[20];
    int starting_block;
    int size;
    int is_file;
    int mode;

    char create_time[20];
    char access_time[20];
    char modify_time[20];
} dir_entr;
```

Issues and Solutions

What are the limitations of our file system?

- External fragmentation would occur when there is enough space but cannot allocate because we cannot find contiguous space in the freespace bitmap.
- Internal fragmentation due to the way we allocate free space. This will happen when we remove a file or directory or overwrite a file or directory, since we don't have a method to fill up any freed space.

Issues you had

- What was difficult to accomplish for each passing milestone?

Milestone one presented a challenge because we initially had set up our free space management system in a way such that we were using up more space than necessary to represent one bit.

For milestone two we repeatedly had to tweak our path parsing so that it would not conflict with the different commands which our shell needed to accept and respond to.

In milestone three we had an issue where a file would be opened multiple times when it was not necessary causing segmentation faults from setting the buffer size equal to the files size instead of accounting for the block size.

After milestone three we were free to continue our work, though we still faced many challenges. Implementing the directory functions that would perform the actions of changing what is both on the logical file system and on the disk, functions like fs_mkdir, fs_rmdir, and fs_delete, was difficult although we overcame it by looking to the default way these functions are implemented on the c manual pages.

- What features were difficult or confusing to implement?

One of the harder features that was implemented was the acceptance of either relative or absolute paths when calling commands such as mv and cp.

- How did we overcome these issues? As a group? Individually?

We used different strategies to solve different problems based on what was most comfortable for the people in the group. For some issues we each attempted to come up with our own individual solutions and then decided on the most optimal solution out of our combined ideas and for other problems we worked together at length in trial and error to find a viable solution as a group.

- What is missing or could’ve been a good addition to the file system?

Compaction would solve our problems with both external and internal fragmentation.

Driver Program Explanation

The driver program starts in fsshell.c. Starting with main, the program checks if there are the correct number of arguments and then assigns the correct values to the variables, filename, volume size, block size otherwise variations of errors are returned. The assigned variables are used in the startPartitionSystem function which allows us to treat the array of blocks like a logical block array.

Then initFileSystem is called. The initFileSystem function begins by mallocing memory for the VCB block and reading the first block in the LBA to bring it in for use. Then we initialize the data we currently have for it including the magic number, total number of blocks, and the size of each block.

After that are two functions related to the freespace, init_bitmap() and update_free_block_start(). The init_bitmap() initializes the freespace and marks 6 spots as used. 1 for the VCB, 5 for the initialized freespace. Update_free_block start() simply sets the pointer to the next free space after additions to the bitmap.

Then is the initialization of the root directory. The root directory is created by the create_dir function from the file with the directory functions.

After initialization, back in the fsshell in its while loop, is where the user is prompted for commands until they type in exit. Also in the while loop is the processCommands function, which will process the user's commands and match it with the ones in the dispatch table of commands to call the specific call function that the user wants to use. The commands use functions implemented in dir_func.c and bio.c.

ls command

```
Prompt > ls -a -l

D        3072   2022-04-27 21:04:56   .
D        3072   2022-04-27 21:04:56   ..
D        3072   2022-04-26 22:39:33   home
D        3072   2022-04-26 22:39:36   student
D        3072   2022-04-27 01:52:49   new_folder
D        3072   2022-04-27 16:33:05   pictures
D        3072   2022-04-27 16:57:19   documents
```

The ls command lists all the directory and files in a directory. A user can list all children of a directory with ls -a and list the long format with ls -l. It uses the opendir, readdir, and closedir functions from our dirfunc.c. The opendir func loads the given path/directory. Readdir then returns an item from the directory each time it's called until it reaches the end. If it is called again it returns NULL. Closedir cleans up the pointers used by open and readdir by freeing and NULLing them. Then the display function from fsshell.c is used to display the directories and files in the given directory path. If an empty pathname is given by the user, then it will be assumed that the user is attempting to display the contents of the current working directory. A user can also parse a relative or absolute pathname, and display the contents of a directory in '-a', '-l', or '-a -l' format. For example, proper usage would be something like 'ls /home/Desktop/folder -a -l' to display the contents of the folder in '-a -l' format. The contents to be displayed with the '-l' format will display the file type, byte size, last accessed time, and name of the contents in the displaying directory.

cp command

```
Prompt > cp /newImage.jpg /pictures/newImage.jpg
openned 'newImage.jpg' in parent directory: '.'.
finished reading file.
finished writing file.
```

The cp command copies a file within a directory of our file system to a different directory in the file system. It can use both relative and absolute paths. It uses the functions b_open, b_write, b_read, and b_close from the bio.c file which contains our file functions.The b_open function open the given file after it's path has been parsed and return a file descriptor that'll be used for b_read and b_write. B_read only allows the user to read the file, while b_write only allows the user to write to the file. The file being read or written to is determined by the file descriptor passed into b_read and b_write by b_open. A buffer and count is also passed into the read and write. For b_read, the buffer is empty so that the file contents of the file can be written into it and the count (capped at 200 bytes at time) determines how much is read into the buffer. B_write then uses the buffer filled by b_read to then write it to the file descriptor's buffer. In the fsshell.c after the file is opened with b_open, b_read is used to read the contents of the src file, while b_write writes the contents to the new destination file. Finally b_close cleans up and frees and nulls any pointers used in b_open, b_write, and b_read.

mv command

```
Prompt > mv /BigFile.txt /documents
move directory completed.
```

The move command moves a directory or file to another location in the directory tree by specifying a source pathname, followed by a destination pathname. For example, 'mv /src /dest' would be a correctly formatted input. The move command also accepts absolute paths, allowing users to move a file or directory, regardless of the user's current working directory. The destination must be a directory for the moving object to be stored. If the head path of the destination is unknown, it is assumed the user may want to change the file or directory name of the moving object. In this case the move command will update the name to reflect the given head path. For example: 'mv /src/my_file /dest/new_name' would change the filename of 'my_file' to 'new_name', if 'new_name' doesn't exist in the destination folder.

```
Prompt > mv /ImageFile.jpg /newImage.jpg
move directory completed.
```

If the name does exist, it will be treated as a destination folder. If the head path of both the source pathname and the destination pathname are identical in name, then the user will be prompted with the option to overwrite the identically named file or directory in the destination folder. However, overwriting can only be done when a moving object and the identically named object in the destination folder are of the same file type. For example, a directory cannot overwrite a file and vice versa. If a user wishes to overwrite an entire directory that is not empty, they can do that at the cost of losing everything in the overwritten folder. In other words, two folders cannot merge, instead the overwritten one is replaced with the incoming folder. A couple other restrictions include that a user cannot move either the current working directory, the root directory, nor can a user move a directory into itself or one of its children.

md command

```
Prompt > md /documents
Prompt > md /pictures
```

The md command will create and add a directory to the directory tree and to the logical block address. For this process to work, a user must pass in a valid pathname that can either be absolute or relative, where only the head path is unknown to the parent folder. For example: ‘md /home/desktop/new_folder’ would be a valid md command if all paths leading up to ‘new_folder’ were valid, alongside the ‘new_folder’ directory not being contained (unknown) inside the it’s parent ‘desktop’ directory. At this point, the head path will be treated as the new directory to create, and the most recent parent will be where it is placed in the directory tree. After successful allocation of contiguous space on disk for the created directory, the new directory will be written to disk.

rm command

```
Prompt > rm /home/Desktop
-- updated disk --
```

The rm command will remove a directory or file from disk and the directory tree. For this to happen for a file is pretty simple. After the user inputs a command ‘rm /folder/file.txt’ a function called isfile() will check to see if the head path ‘file.txt’ is indeed a file. Our helper functions parse_pathname() and validate_path() make this easy by returning the enum ‘FOUND_FILE’ and setting up a temporary pointer to the metadata this file is stored in, if the head of the path is indeed a file. Once the file is found, it will have its occupied space in the freespace bitmap marked to zero, implying that it is free to overwrite. Afterwards, the parent directory will have the filename where the file is stored set to an empty string, which will

either be filled by the rightmost child, or marked as free to be overwritten if the file was the only child in its parent. Removing a directory takes a similar approach, with the added restriction that the directory must be empty before deletion. After a user passes the command ‘rm /folder/dir’, the helper functions for parsing a path will return the enum ‘VALID’ if the head path is a directory, which will also set up a temporary pointer to the directory at the head path for the filesystem to manipulate. At this point, the directory will be checked if it is empty, and follow the same steps for deletion as files, so long as the directory is empty.

cp2l command

```
Prompt > cp2l /newImage.jpg /home/student/Desktop/ImageFile.jpg
openned 'newImage.jpg' in parent directory: '.'.
finished reading file.
```

The purpose of the cp2l command is to copy a file from the file system to the linux subsystem using a combination of open, read, write, and close functions from both our file system and the linux file system.. It can use both absolute and relative path for the source destination, but for the destination, it would have to be an absolute path. The command is similar to the cp command in that it uses the same functions: b_open, b_read, and b_close, but it doesn’t use b_write. This is because we’re writing to the linux file system so it uses a write command we did not write. In the fsshell.c, you’ll see that our source file is opened and read by our b_open and b_read functions. Then the linux write uses the file descriptor, buffer, and count returned from our b_open and b_read functions to write to the linux file system. Finally b_close and linux’s close cleans up the pointers used by our functions.

cp2fs command

```
Prompt > cp2fs /home/student/Desktop/TestBigFile.txt /BigFile.txt
finished writing file.
```

The purpose of the cp2fs command is to copy a file from the linux subsystem to the file system using a combination of open, read, write and close from both our file system and the linux file system. Since we’re copying a file from the linux system, the path/source would have to be an absolute path, but the destination within our own filesystem can be a relative or absolute path. The command is the opposite of the cp2l command. Where cp2l would use our file system’s b_open and b_read, the cp2fs would use the linux open and read functions because it’s working with a file in the linux file system. The linux functions open and read values are then used in our b_write to write to a destination on our filesystem. Then b_close and linux’s close are used to free and clean up any pointers used by the open and read functions.

cd command

```
Prompt > cd /home/Desktop/folder
```

The purpose of the cd command is to change the working directory. It uses the set_cwd from the dir_func.c file to load the desired path. First it checks if the path is equal to ".." which indicates that the user wishes to go back to the parent directory. If so, then the pointer(curr_dir) that is pointing to the parent directory is loaded. If the path is not "..", then the path is parsed in our parsed_pathname function to first check if it's valid, if so then the pointer(temp_dir) pointing to the parsed path is loaded/set as our current directory.

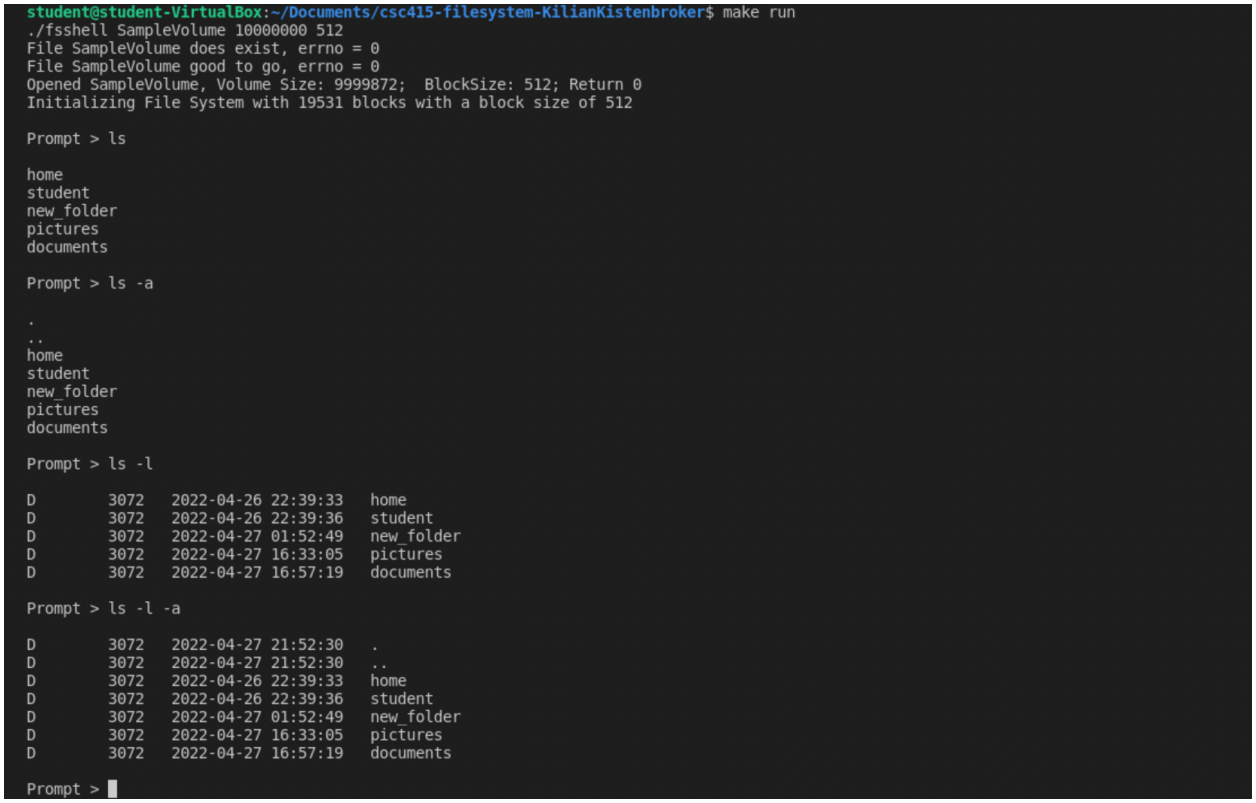
pwd command

```
Prompt > pwd
/./home/Desktop/folder
```

The purpose of the pwd command is to print the entire path of the current working directory. This is accomplished by the fs_getcwd() function, which will iterate up the directory tree to fetch all the parent directory names for displayal. The function will traverse up the tree after reading the current working directory into memory, and storing its name, before reading its parent into memory to get the name of the parent. This cycle continues until the root is read into memory. Temporary strings are brought into memory to store and concatenate all the gathered names, until they are ready to be stored into a buffer that will be returned to the shell for displayal.

Screenshots showing each of the commands listed in the readme:

ls command:



cp command:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-KilianKistenbroker$ 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 > ls

home
student
new_folder
pictures
documents

Prompt > cd /pictures
Prompt > ls

pictures
newImage.jpg

Prompt > cp /newImage.jpg /./home/newImage.jpg
openned 'newImage.jpg' in parent directory: 'pictures'.
finished reading file.
finished writing file.

Prompt > ls

pictures
newImage.jpg

Prompt > cd ..
Prompt > cd /home
Prompt > ls

home
newImage.jpg

Prompt >
```

mv command:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-KilianKistenbroker$ 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 > ls

home
student
new_folder
pictures
documents

Prompt > cd /home
Prompt > ls

home

Prompt > cd ..
Prompt > cd /documents
Prompt > ls

documents

Prompt > cd ..
Prompt > mv /documents /home/documents
move directory completed.

Prompt > ls

home
student
new_folder
pictures

Prompt > cd /home
Prompt > ls

home
documents

Prompt > cd /documents
Prompt > pwd
/./home/documents

Prompt >
```

md command:

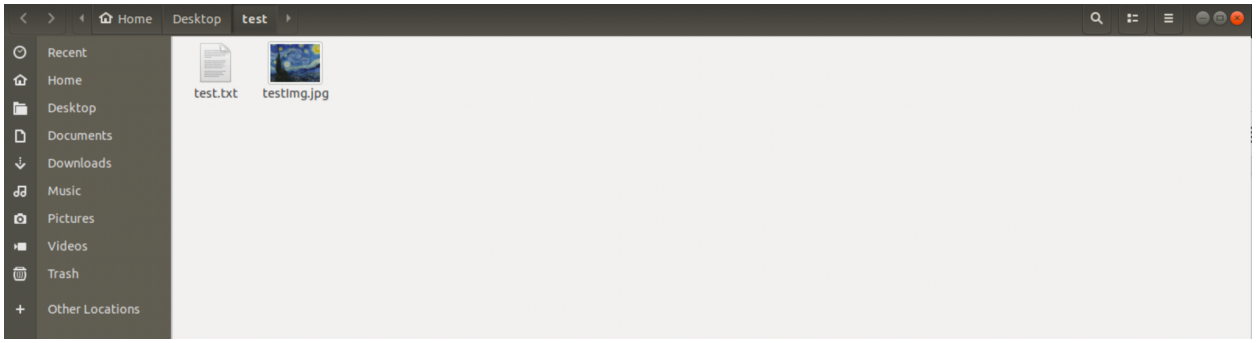
```
student@student-VirtualBox:~/Documents/csc415-filesystem-KilianKistenbroker$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > ls

home
school

Prompt > md /newDir
-----------CREATED NEW DIRECTORY-----------
Size of dir: 3072
dir[0].starting_block : 24
dir[1].starting_block : 6
dir[0].filename : newDir
dir[1].filename : .
dir[0].mode : 511
dir[1].mode : 700

Prompt > ls

home
school
newDir

Prompt > cd /newDir
Prompt > pwd
/./newDir

Prompt > cd ..
Prompt > rm /newDir
-- updated disk --

Prompt > ls

home
school

Prompt >
```

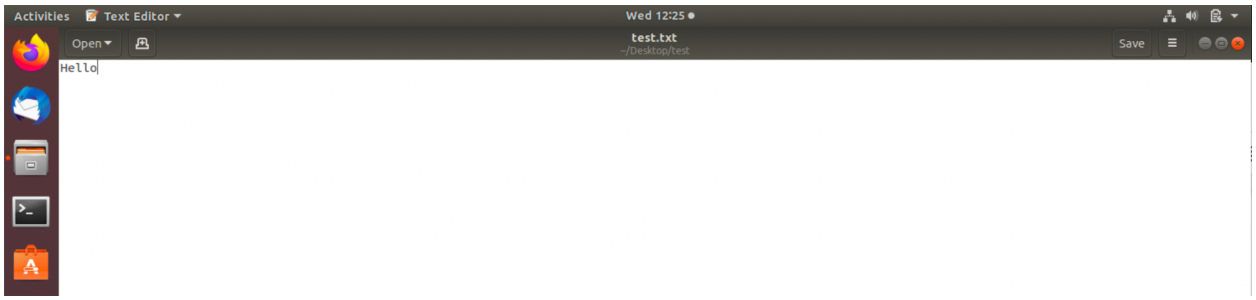
rm command:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-KilianKistenbroker$ 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 > ls

home
school

Prompt > md /newDir
-----------CREATED NEW DIRECTORY-----------
Size of dir: 3072
dir[0].starting_block : 24
dir[1].starting_block : 6
dir[0].filename : newDir
dir[1].filename : .
dir[0].mode : 511
dir[1].mode : 700

Prompt > ls

home
school
newDir

Prompt > cd /newDir
Prompt > pwd
/./newDir

Prompt > cd ..
Prompt > rm /newDir
-- updated disk --

Prompt > ls

home
school

Prompt >
```

cp2fs command:

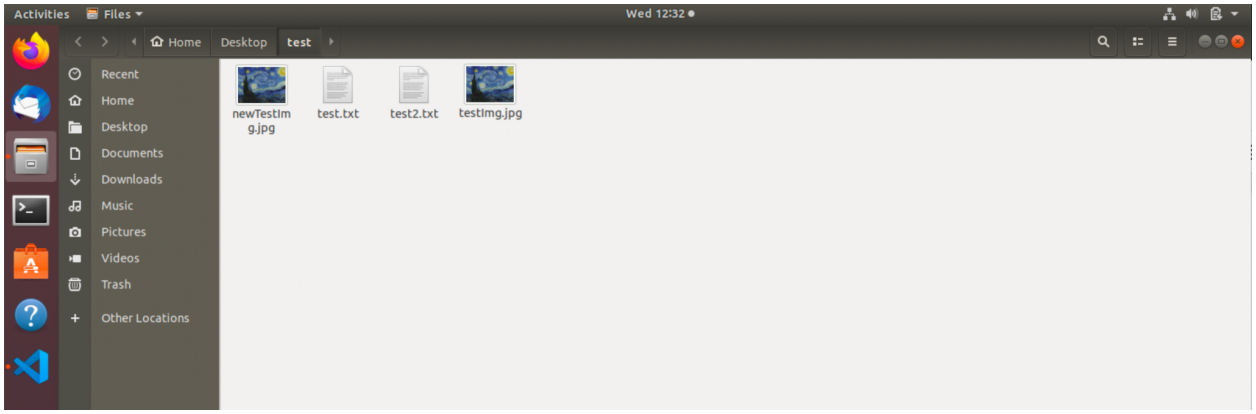
```
student@student-VirtualBox:~/Documents/csc415-filesystem-KilianKistenbroker$ make run
gcc -o fsshell fsshell.o fsInit.o fsLow.o -g -I. -lm -l readline -l pthread
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > ls

home
school

Prompt > cd /home
Prompt > cp2fs /home/student/Desktop/test/test.txt /test.txt
finished writing file.

Prompt > ls

home
test.txt

Prompt > cp2fs /home/student/Desktop/test/testImg.jpg /newTestImg.jpg
finished writing file.

Prompt > ls

home
test.txt
newTestImg.jpg

Prompt > cp2l /test.txt /home/student/Desktop/test/test2.txt
openned test.txt in parent directory: home with fd 0.
finished reading file.

Prompt > cp2l /newTestImg.jpg /home/student/Desktop/test/newTestImg.jpg
openned newTestImg.jpg in parent directory: home with fd 0.
finished reading file.
```

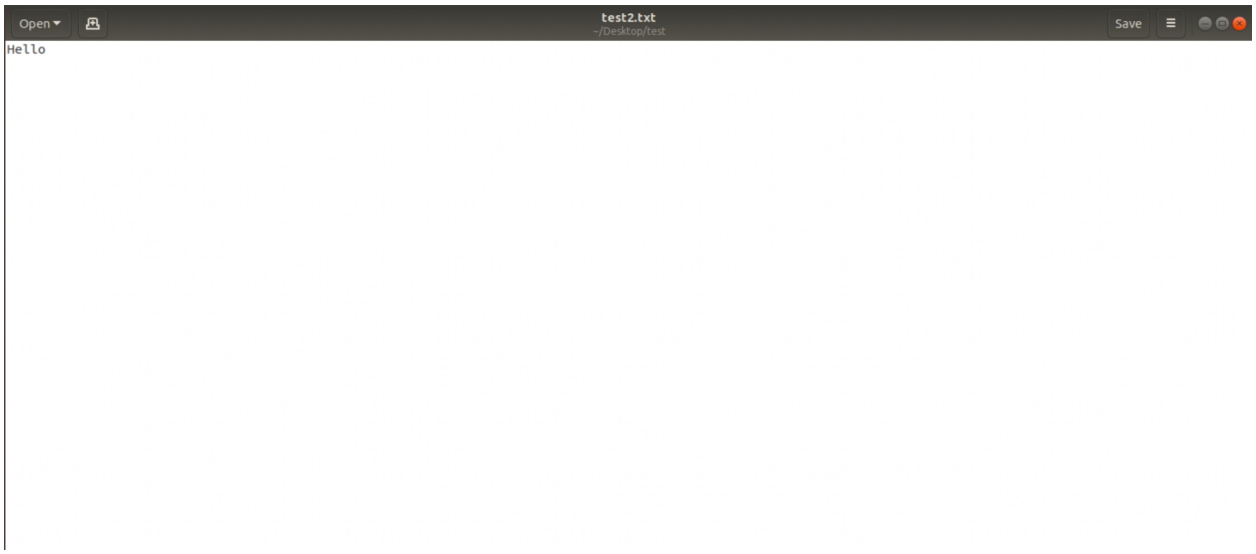


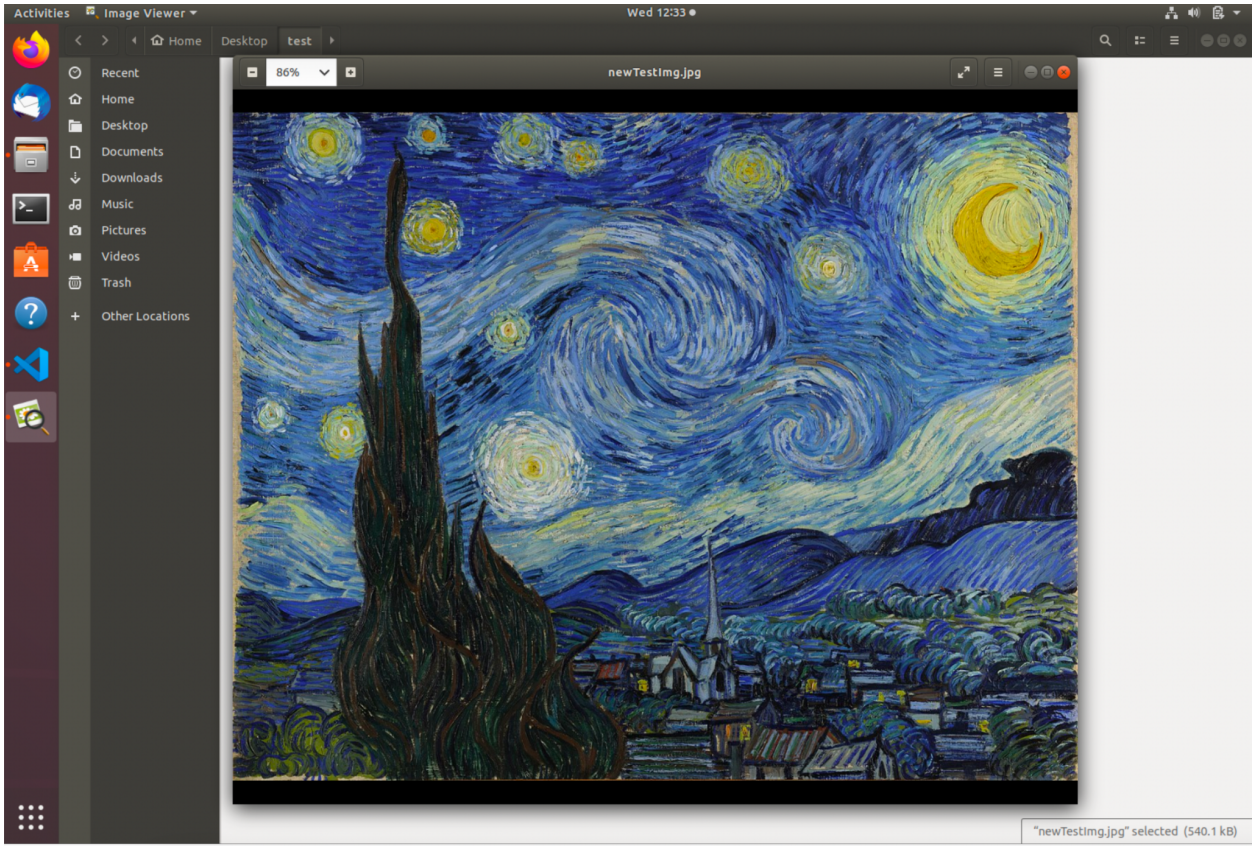


cp2l command:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-KilianKistenbroker$ make run
gcc -o fsshell fsshell.o fsInit.o fsLow.o -g -I. -lm -l readline -l pthread
./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 > ls

home
school

Prompt > cd /home
Prompt > cp2fs /home/student/Desktop/test/test.txt /test.txt
finished writing file.

Prompt > ls

home
test.txt

Prompt > cp2fs /home/student/Desktop/test/testImg.jpg /newTestImg.jpg
finished writing file.

Prompt > ls

home
test.txt
newTestImg.jpg

Prompt > cp2l /test.txt /home/student/Desktop/test/test2.txt
openned test.txt in parent directory: home with fd 0.
finished reading file.

Prompt > cp2l /newTestImg.jpg /home/student/Desktop/test/newTestImg.jpg
openned newTestImg.jpg in parent directory: home with fd 0.
finished reading file.
```


Activities
Files
Wed 12:32
Home
Desktop
test
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
Other Locations
newTestImg.jpg
test.txt
test2.txt
testImg.jpg


Open
test2.txt
~/Desktop/test
Save
Hello



cd command:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-KilianKistenbroker$ 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 > pwd
/.

Prompt > cd /home
Prompt > pwd
/./home

Prompt > cd ..
Prompt > cd /school
Prompt > pwd
/./school

Prompt >
```

pwd command:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-KilianKistenbroker$ 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 > pwd
/.

Prompt > cd /home
Prompt > pwd
/./home

Prompt > cd ..
Prompt > cd /school
Prompt > pwd
/./school

Prompt >
```