File System Project

GitHub Link: https://github.com/CSC415-Fall2021/csc415-filesystem-anthonyzhang1
GitHub Name: anthonyzhang1

Group Name: Michael
Group Members:
Edel Jhon Cenario (ID: 921121224)
Michael Wang (ID: 921460979)
Michael Widergren (ID: 921363622)
Anthony Zhang (ID: 921544101)

Description of Our File System

Volume Control Block:
We read and write to the volume control block by means of a globally scoped pointer to a VCB struct. It stores information we need like the size of a block in bytes, the size of a directory in blocks, the block the root directory starts on, etc.

Bitmap / Free Space Management:
We use a bitmap to keep track of free space. In it, a 0 represents a free block, while a 1 represents a used block. We read and write to the bitmap by means of a globally-scoped uint32_t pointer. The way we search the bitmap for free blocks is unique, in that when we search for free blocks, we do not always start the search at block 0. Rather, the search starts from where the search left off last time. For example, if we searched from block 0 to block 20, the next time we search we start from block 20, rather than block 0. When we hit the end of the bitmap, we reset back to block 0 so that we search the entire bitmap, rather than only the tail-end of it. We use a counter to ensure we do not search more than numberOfBlocks blocks, which prevents us from entering an infinite loop if we are unable to find enough contiguous free blocks.

Upon running the program again, this starting block counter is reset to 0, which lets us fill in any gaps left by deleted files that the starting block counter had already moved past the last time the program ran. We implemented our bitmap search function this way because we thought it was wasteful to start the search at block 0 every time. For instance, if the first 10’000 blocks were used, it would be inefficient to start searching from 0 each time we wanted to find space for a file. In our version, the search would start at block 10’000, letting us find free blocks immediately. Again, upon program restart or upon reaching the end of the bitmap, we reset the counter back to 0, letting us fill in any gaps left by deleted files.

Not simply starting our search at block 0 each time complicated things a lot more. It is a lot harder to explain the idea in words than it was to code it. Hopefully the reason behind why we did our bitmap search this way makes sense.

Also, we have a function called getContiguousFreeBlocks(), which finds however many contiguous free blocks the caller needs. It returns the block the contiguous free blocks start at. We already cited the source in the block comments, but this page was very helpful in helping us learn how to manipulate the bits in our bitmap: stackoverflow.com/questions/2525310/how-to-define-and-work-with-an-array-of-bits-in-c.

Root Directory:
We initialize the root directory when we initialize the volume. We prevent the user from moving, renaming, overwriting, or deleting the root directory, amongst other things. Unlike the VCB and bitmap, the root directory is not global nor is it kept in memory for the entirety of the program. We load it into memory only when we need it.

Directories / Directory Entries:
Our directory entries store the name of the entry, the block the entry starts on, the size of the entry in bytes, its type (directory, file, or a free entry), the time it was created, the time it was last modified, and the time it was last opened. Our directory entries were somewhat large, taking up 108 bytes each. Our directories can hold 52 directory entries, meaning our directories are 11 blocks large, assuming that there are 512 bytes per block.

Current Working Directory:
We track our current working directory by storing its starting block, rather than storing a pointer to a directory entry. When we need to access the current working directory, we can just read a directory's worth of blocks from disk into a dir_entry pointer, since we know the starting block of the current working directory.

Pathing / Path Parsing:
Our path parser works by returning the start block of the parent directory of the basename directory entry. e.g. given the path x/y/z, we return the start block of y, since z is the basename directory entry. The parser can handle both relative and absolute paths. Admittedly, we could have simplified things by just returning the parent directory instead of the parent's start block. This did lead to us repeating code in our program. We were already struggling to meet milestone deadlines; refactoring the path parser was relatively low priority.

We also have a function getBasename() to get the string "z" in the path x/y/z. This allows us to search y for a directory entry named z, which returns the index of z in y. This is useful because y is z's link to the root directory. Also, y holds all of z's metadata, like its name and start block. We can easily delete z by just removing all of z's metadata in y and freeing z's used blocks in the disk.

Files:
When we copy files to our file system, we try to find at least 5 contiguous free blocks to write our file to. This number can be changed freely by changing MIN_FREE_CONT_BLOCKS at the top of b_io.c. We decided 5 was a good number because it was large enough for small text files. If MIN_FREE_CONT_BLOCKS was too small, we might have our file writes repeatedly interrupted as we use up the contiguous free space for the file. For example, if MIN_FREE_CONT_BLOCKS was 1, and there was a gap of 2 free blocks in the middle of some used blocks, we would write our file to the first of the 2 free blocks. If you wanted to write a file that took up 3 blocks, you would only write 2 blocks worth of content before being the write is interrupted. Note that even if the write is interrupted, the file will still be created. It will just be missing part of its contents. The half-written file can still be deleted and freed like any other file; the blocks are not lost.

On the other hand, we did not want to make MIN_FREE_CONT_BLOCKS too large, since we would end up skipping over small segments of free blocks in our disk that might have fit our file. If we knew the size of the file beforehand, this whole issue of contiguous free blocks would have been avoided.

In addition: if we made a new file, the next time we search for free blocks, we start at the new file's start block. This is because files can use up less than 5 blocks. If a file was 1 block large, there would be 4 free blocks left over. We did not want to leave a gap between used blocks unnecessarily. Starting the search at the new file's start block (block 0 in this case) allows us to account for those 4 free blocks left over the next time we search. If block 5 was free, we could fit in another file starting at block 1, leaving no wasted space.

0-byte files do not take up a block on disk. They only exist as a directory entry. We only write a file to disk when the file actually has content in it, i.e. the file is greater than 0 bytes.

b_open searches for the file we are trying to open. We then set the file offset and set the fcb's data accordingly. We are able to check for the flags set, like O_CREAT, O_TRUNC, and O_APPEND. O_CREAT makes a new file, O_TRUNC essentially deletes the old file then makes a new one with the same name, and O_APPEND makes the file pointer point to the end of the file. For O_APPEND, we LBAread the contents of the file's last block into the buffer. We can only write whole blocks to the disk, so if we just leave garbage in the beginning of the buffer, we will end up writing garbage to the file's last block. We also check for O_WRONLY, O_RDONLY, and O_RDWR to determine how we will be using the file.

b_seek adjusts the file offset, only if the new offset is valid. We LBAread the block the file pointer is pointing to for the same reason as above. We do not want to read/write garbage into the caller's buffer or disk.

Our b_write and b_read functions use Robert Bierman's method of splitting count into three parts from Assignment 2b. We have credited Professor Bierman in the block comments for b_write and b_read.

Both b_write and b_read should be compatible with b_seek. Instead of reading/writing from position 0 and moving linearly from there, we are able to handle reading/writing at a specific offset. We did not have any shell commands to test b_seek, but at least b_write and b_read seem to work when the offset is 0.

b_write checks whether the block we are writing to is actually safe to write to, i.e. the block is not used by another file and is not out of the disk. We wrote directly from the caller's buffer to disk one block at a time because we could not be sure that the next block was safe for us to write to. In b_read we are able to copy as many blocks as we want, though. There are no concerns about free or used blocks in the caller's buffer.

After b_write finishes, the fcb buffer can still have content that needs to be written to disk. b_close handles the leftover bytes in the fcb buffer. If we are writing to a new block, we just LBAwrite the buffer to disk. However, if we are overwriting a block in our file, we should LBAread the block first into a temp buffer, then copy only the parts that we changed into the temp buffer. We then write the contents of this temp buffer to disk. We cannot just write the fcb buffer to disk as is because the fcb buffer would contain garbage from the last time the fcb buffer was filled. After writing the fcb buffer's contents to disk, we update the file's metadata in the parent directory and mark the blocks the file used in our bitmap.

Issues We Had

The greatest issue we had was working together as a team. Each member's contribution to the group project was not equal for a number of reasons. One reason why contribution was so imbalanced was that some of us did not have a clear idea on what to do. This problem only became worse the more progress we made on the project because those who were lost fell further and further behind. Admittedly, the code was hard to follow at times; refactoring was not a luxury we could afford given that we were persistently behind schedule.
Another reason was that we did not work on the project consistently because we had other commitments; we tended to work in short bursts usually before a milestone deadline. Such was the case for Milestone 1. This meant we would go from 0 percent done to 100 percent done with a milestone in a matter of days. However, this was not good in the long run. The group members who did not actively contribute during this sprint would have to take the time to understand what was worked on and how it works, which slowed down our progress in the next milestone.
We believe the next reason why group work failed was that we did not set deadlines for each of our assigned parts. Some parts of the file project had to be done before the others could be completed, but without deadlines, we did not know when a critical component was going to be finished. With the milestone deadline approaching and progress not being made, we had to reconsider how we divided up our work.

This was mentioned earlier, but an issue we had was time management. We could not devote enough time to the project because some of us had work and other assignments with deadlines sooner than the one for the file system project. There was a lot to take in when we first started the project, and the milestones only grew more complex. This led to us making some impromptu decisions that hurt us later on, such as with the path parser returning the parent directory's starting block rather than the parent directory itself, causing some repeated code. We could have also reduced the bloat of our code by moving repeated code into functions.

It was annoying that we had to remember to free all of the mallocs whenever a potential error caused us to return from a function prematurely. All of the free(ptr) and ptr = NULL lines added a lot of lines to our code because we do a lot of error checking.
We remedied this inconvenience by using a goto statement that frees all of the mallocs in the function upon error. We believe this is less error prone than having a hundred frees scattered everywhere because all the frees happen in one place: after the error label. Furthermore, it reduced the number of lines we used.

The move command stalled our progress because we had to deal with the various ways of breaking the file system or losing a directory and its blocks permanently. To name a few, moving a directory into itself would cause the directory to disappear, but it would still take up space on disk. Renaming the root directory caused the root directory to become an entry within itself somehow. Moving the current working directory or any of its ancestors would lead to a segfault.
We wrote many additional error checks to prevent the user from doing anything that would damage the integrity of the file system, which is explained below in the move section. We believe move is safe to use now.

How the Driver Program Works

List (`ls`):
`ls [--all/-a] [--long/-l] [pathname]`

`ls` without any flags lists all non-hidden directory entries within the directory at `pathname`.
The list's order is based on the directory entry's index within the directory at `pathname`. Starting from entry index 0 and ending at the maximum number of entries in a directory, `ls` prints the entry at index `i` on a new line. It does not print anything if the entry is free though, so there will not be any gaps in our list.

If `pathname` is not provided, `ls` lists all non-hidden directory entries within the current working directory.
You can use `ls` on a file, but that only lists the name of the file, even with the `-a` and `-l` flags enabled.

You can add the `-a` flag to list all directory entries, including the hidden ones. Without the `-a` flag, `ls` does not list entries that start with the '.' character.

You can add the `-l` flag to show the details of each directory entry. Specifically, it shows the entry's type (directory or file), the size of the entry in bytes, the **creation date** of the entry, and the name of the entry.

The `-a` and `-l` flags can be combined to list the details of every directory entry, including hidden ones.

Copy (`cp`):
`cp src_file [dest_file]`

`cp` copies `src_file` to `dest_file` within the file system. If `dest_file` already exists, `dest_file` will be overwritten with `src_file`.
If `dest_file` is not provided, `dest_file` will be the same as `src_file`. In other words, `src_file` will overwrite itself, which seems to serve no purpose.

If `src_file` is invalid, nothing is copied.
`dest_file`'s name cannot exceed 64 characters, including the null terminator.

Move (`mv`):
`mv src_file target_file`

`mv` moves and/or renames `src_file` to `target_file`. `src_file` and `target_file` can be a file or a directory. If `target_file` is a file, `target_file` will be overwritten with `src_file`. If `target_file` is a directory, `src_file` will be moved into `target_file`.
If there is already an entry in `target_file` whose name matches `src_file`'s name, that entry will be overwritten by `src_file`.

Some important caveats to protect the integrity of the file system:

You cannot move or rename the root directory.
You cannot move a directory into itself or a subdirectory of itself.
`src_file` and `target_file` cannot be the same file.
You cannot overwrite `target_file` if `src_file`'s type is not the same as `target_file`'s, i.e. you cannot overwrite a directory with a file or vice versa.
You cannot move the current working directory nor an ancestor of the current working directory.
You cannot overwrite a non-empty directory.
You cannot overwrite the root directory nor the current working directory.

Make Directory (`md`):
`md pathname`

`md` makes a new directory at `pathname`.
`pathname` must not exist already, and the parent of `pathname` must not be full. In addition, the new directory's name cannot exceed 64 characters, including the null terminator.

Remove (`rm`):
`rm path`

`rm` deletes the file/directory located at `path`. `rm` marks the entry's blocks as free and removes the entry's directory entry from the parent of `path`. The contents of the entry on disk are not wiped or overwritten, the blocks are merely marked as free.

Some caveats:
You can only delete empty directories with `rm`. We do not support recursively deleting a directory's children.
You cannot delete the root directory nor the current working directory.

Copy to Linux (`cp2l`):
`cp2l src_file [Linux_dest_file]`

`cp2l` copies the file `src_file` from our file system to Linux's file system. The destination of the copy is `Linux_dest_file`. If `Linux_dest_file` is not provided, `Linux_dest_file` will be the same as `src_file`.

If `src_file` does not exist or is not a file, a 0-byte file shall be created in Linux's file system at `Linux_dest_file`.

Copy to File System (cp2fs):
`cp2fs Linux_src_file [dest_file]`

`cp2fs` copies the file `Linux_src_file` from Linux's file system to our file system. The destination of the copy is `dest_file`. If `dest_file` is not provided, `dest_file` will be the same as `Linux_src_file`. `dest_file`'s name cannot exceed 64 characters, including the null terminator.

If `Linux_src_file` does not exist or is not a file, nothing shall be copied. No file will be created in our file system.

Note:
`cp2fs` is considerably slower than `cp2l`. Copying a 3.3 million byte (3.3 MB) file from Linux to our file system took 11 seconds. Copying that same 3.3 MB file back to Linux took a fraction of a second.
Copying a 22'300 byte (22.3 kB) file was nearly instantaneous for both `cp2fs` and `cp2l`, though.

Change Directory (cd):
`cd path`

`cd` changes the current working directory to `path`. `path` must be a directory.

Print Working Directory (pwd):
pwd

pwd prints the absolute path of the current working directory. The root directory is denoted with the forward slash character /.

History (history):
`history`

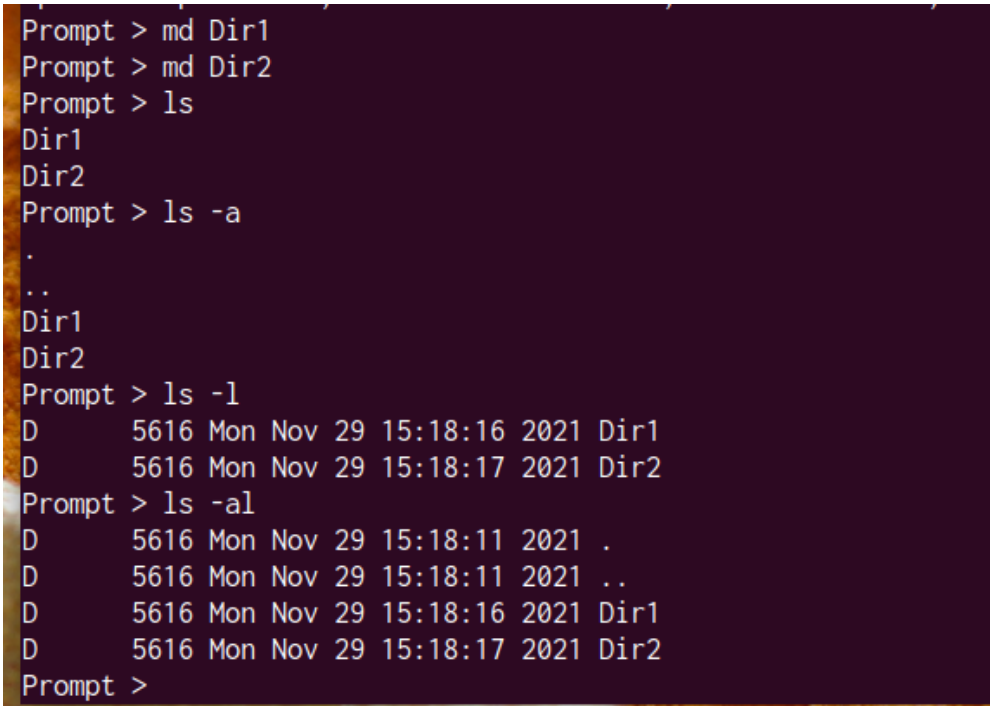
`history` prints the previous commands that were entered into the shell.

Help (help):
`help`

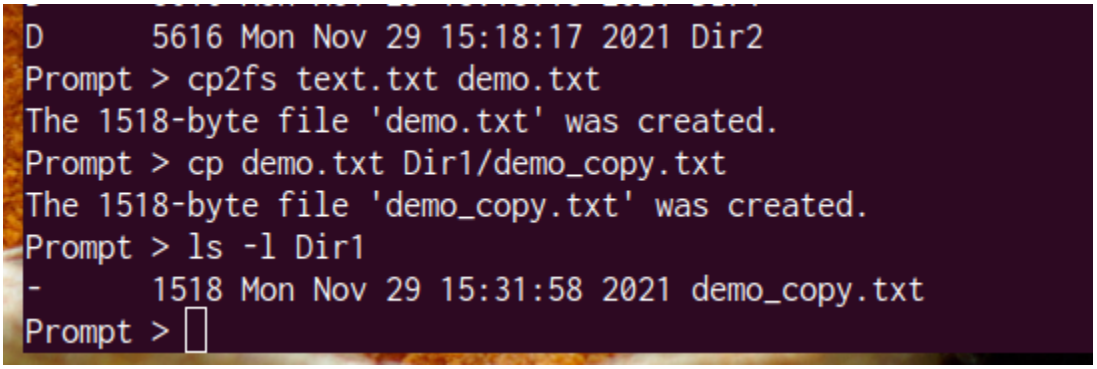
`help` displays the commands the shell supports, along with a brief explanation of each command.

Screenshots of Commands

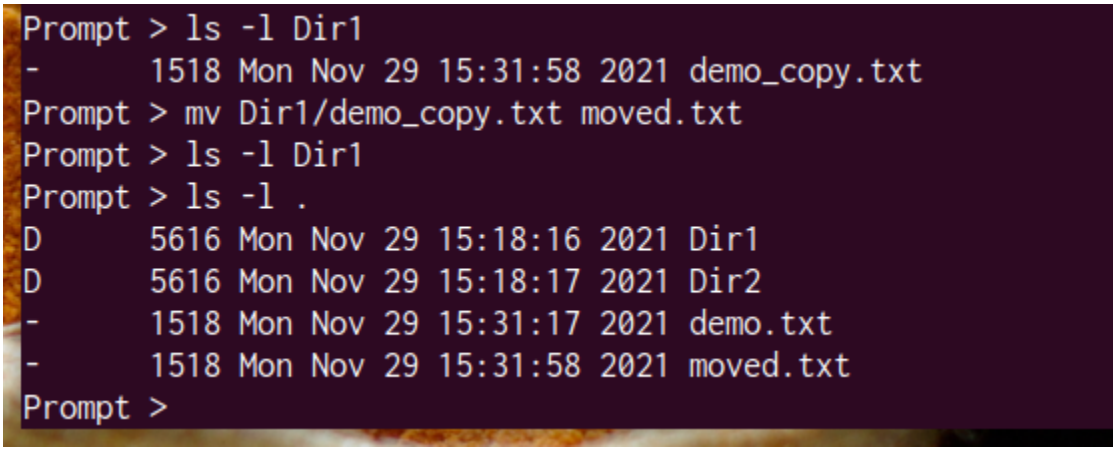
List (ls):
Shows all the flag combinations.



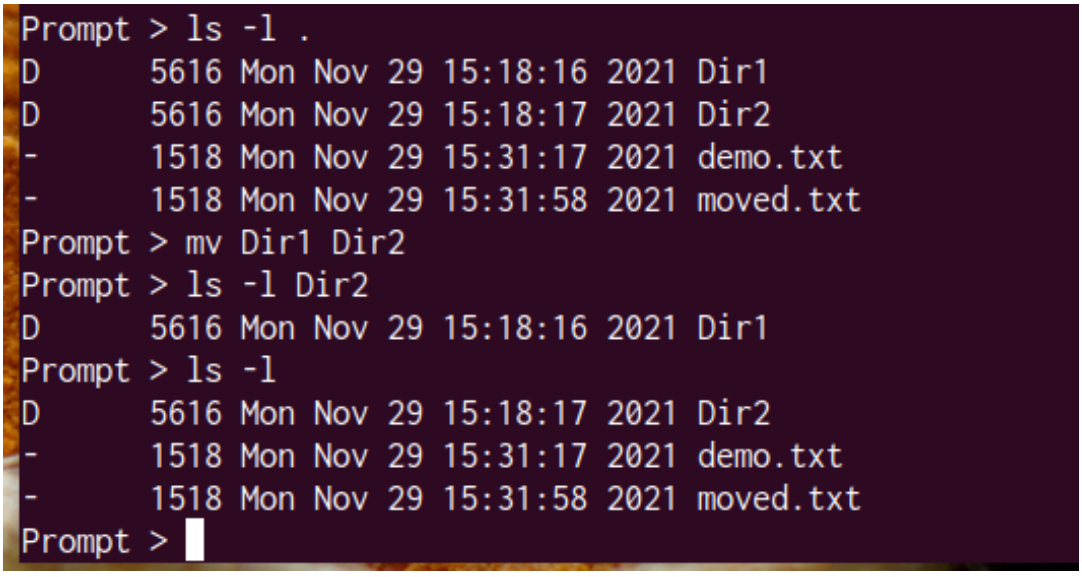
Copy (cp):
Had to copy a sample file (text.txt) from Linux in order to demonstrate copy.



Move (`mv`):
This shows us moving the file demo_copy.txt from the directory Dir1 to the current working directory. We show that Dir1 is empty after the move, while the current working directory contains moved.txt, since we also renamed the file to moved.txt.



This shows us moving the directory Dir1 from the current working directory into the directory Dir2. We show that Dir2 contains Dir1 after the move, while the current working directory no longer contains Dir1.



Make Directory (`md`):
We see that NewDir now occupies the directory entry that Dir1 used to occupy. When we moved Dir1, we marked its entry as free, so NewDir took that free entry when it was moved into the root directory/current working directory. We also create a new directory called NewestDir, which is shown in the list at the bottom.

```
Prompt > md /NewDir
Prompt > md NewestDir
Prompt > ls -l
D       5616 Mon Nov 29 15:55:18 2021 NewDir
D       5616 Mon Nov 29 15:18:17 2021 Dir2
-       1518 Mon Nov 29 15:31:17 2021 demo.txt
-       1518 Mon Nov 29 15:31:58 2021 moved.txt
D       5616 Mon Nov 29 15:55:29 2021 NewestDir
Prompt >
```

Remove (`rm`):
This shows us trying to remove the directory Dir2. We failed because we cannot delete non-empty directories. So we deleted Dir2's contents, then we deleted Dir2 successfully.

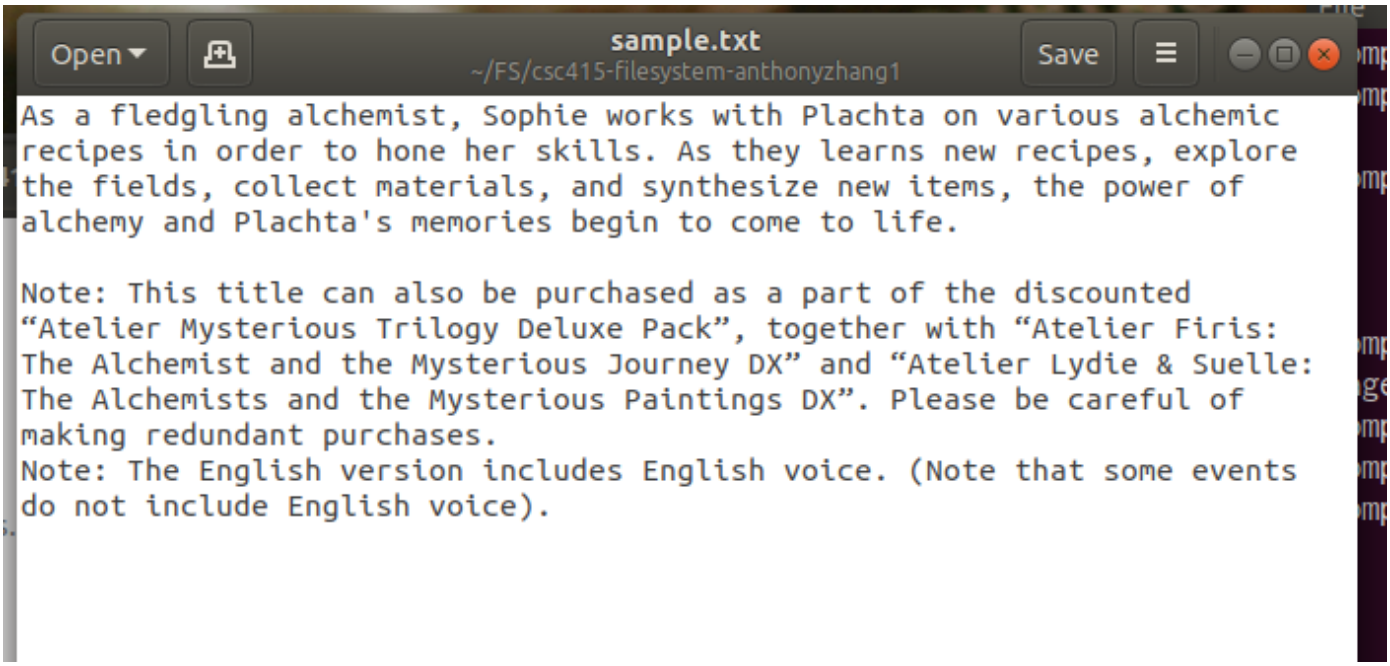
```
Prompt > ls -l
D       5616 Mon Nov 29 15:55:18 2021 NewDir
D       5616 Mon Nov 29 15:18:17 2021 Dir2
-       1518 Mon Nov 29 15:31:17 2021 demo.txt
-       1518 Mon Nov 29 15:31:58 2021 moved.txt
D       5616 Mon Nov 29 15:55:29 2021 NewestDir
Prompt > rm Dir2
You can only remove empty directories. 'Dir2' is not empty. Remove directory failed.
Prompt > ls Dir2
Dir1
Prompt > rm Dir2/Dir1
Prompt > ls Dir2
Prompt > rm Dir2
Prompt > ls -l
D       5616 Mon Nov 29 15:55:18 2021 NewDir
-       1518 Mon Nov 29 15:31:17 2021 demo.txt
-       1518 Mon Nov 29 15:31:58 2021 moved.txt
D       5616 Mon Nov 29 15:55:29 2021 NewestDir
Prompt >
```

This shows us removing 2 files: demo.txt and moved.txt.

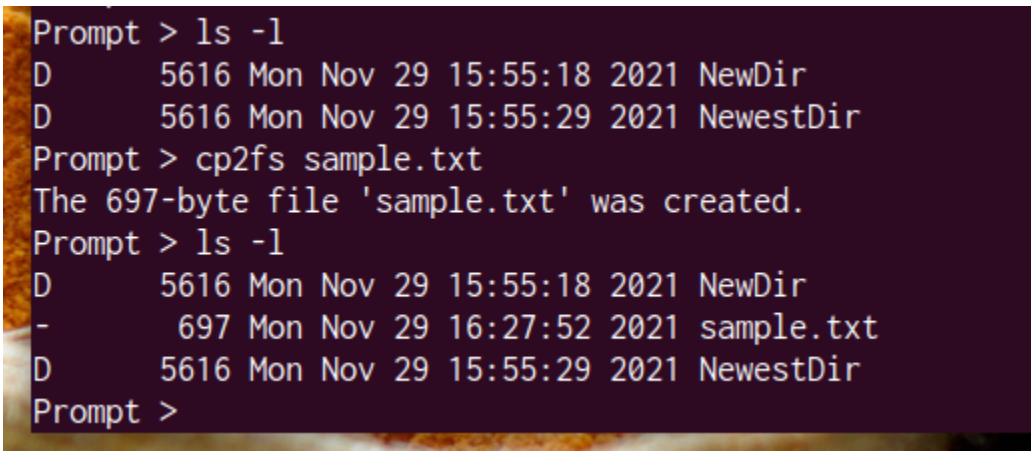
```
Prompt > ls -l
D      5616 Mon Nov 29 15:55:18 2021 NewDir
-      1518 Mon Nov 29 15:31:17 2021 demo.txt
-      1518 Mon Nov 29 15:31:58 2021 moved.txt
D      5616 Mon Nov 29 15:55:29 2021 NewestDir
Prompt > rm demo.txt
Prompt > rm moved.txt
Prompt > ls -l
D      5616 Mon Nov 29 15:55:18 2021 NewDir
D      5616 Mon Nov 29 15:55:29 2021 NewestDir
Prompt >
```

Copy to File System (cp2fs):
We demonstrate `cp2fs` before `cp2l` because we need a file to copy to Linux first.

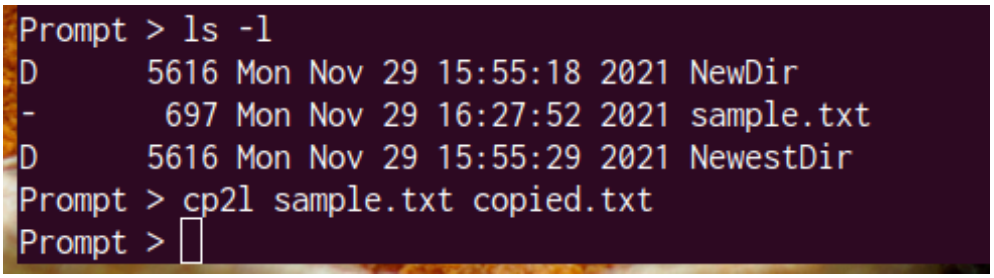
We copy the file sample.txt from Linux to our file system. sample.txt is 697 bytes and here are its contents:



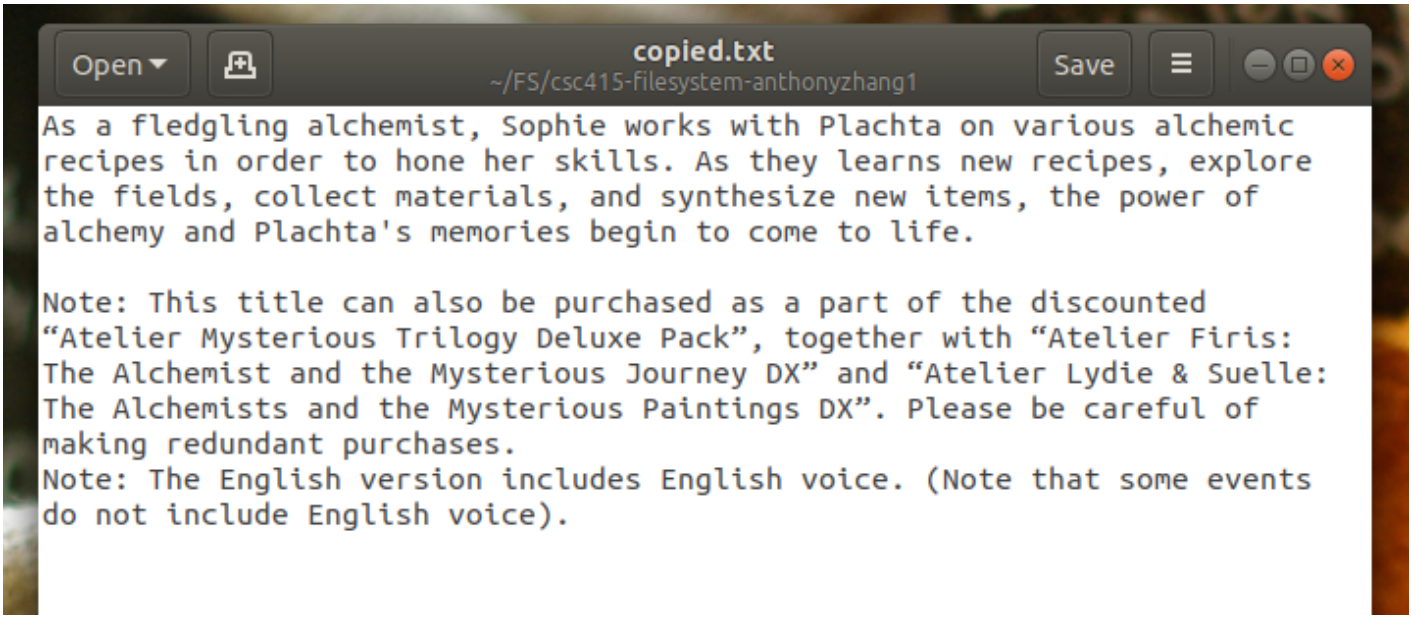
We successfully copy all 697 bytes of sample.txt into our file system.



Copy to Linux (cp2l):
We copy sample.txt from our file system to Linux's file system, where the copied file will be called copied.txt.

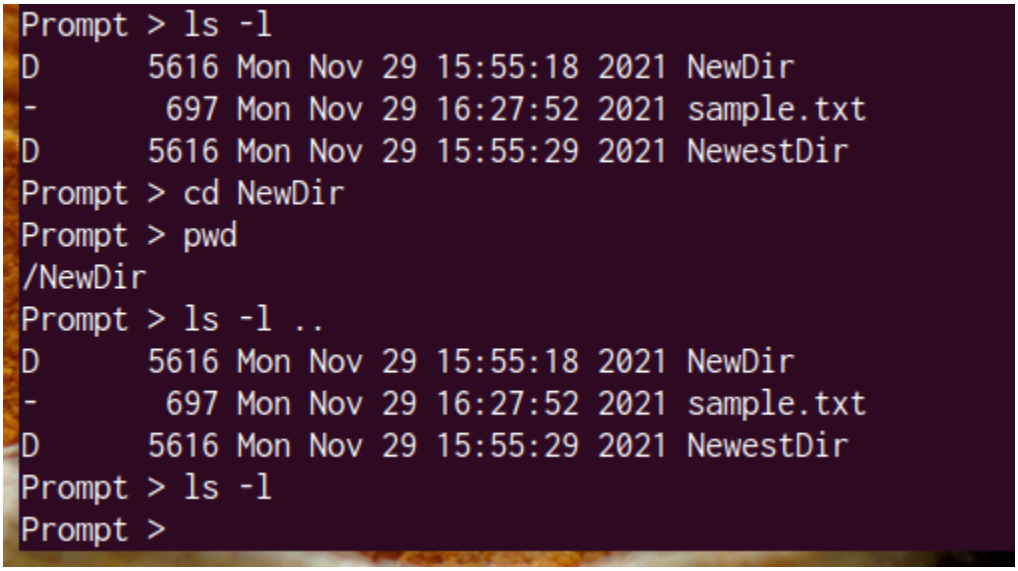


Here is copied.txt's contents. Linux states that copied.txt is 697 bytes, which is the same as sample.txt.

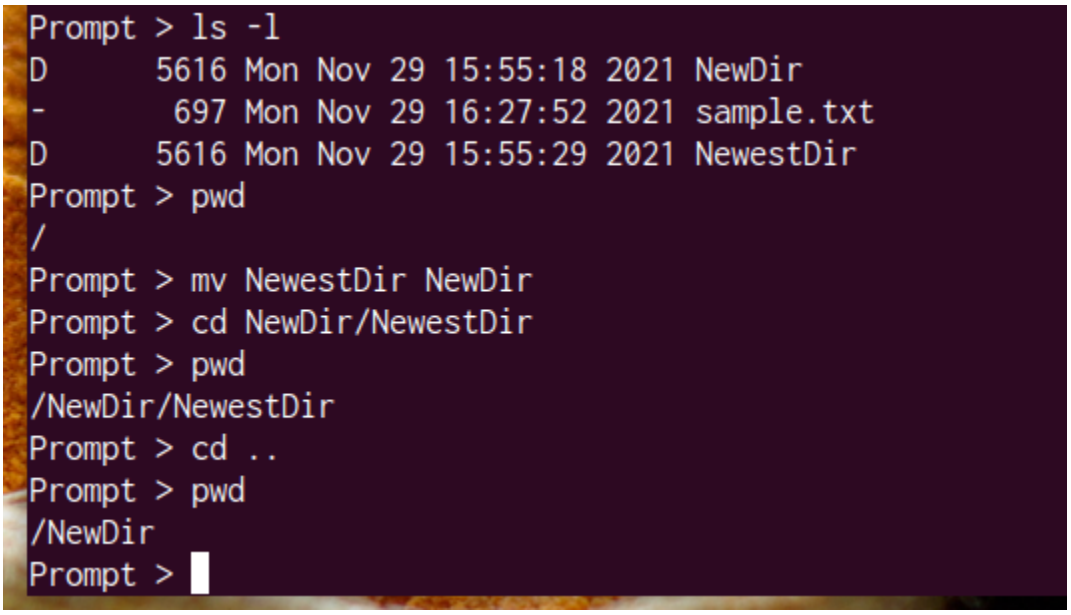


Both `cp2fs` and `cp2l` were successful. The entire text file was copied between the two file systems without issue. The contents of the file appear to be unaltered as well.

Change Directory (cd):
We successfully change our current working directory from the root directory to NewDir.



Print Working Directory (pwd):
Here we show a few uses of pwd. It correctly prints the absolute path of the current working directory.



History (`history`):
We did not alter `history` in our file system, so it should work as it did on Day 1.

Help (help):
We did not alter `help` in our file system, so it should look as it did on Day 1.