File System Documentation

CSC 415-03 Operating Systems

Team: dev/null

GitHub: kpcrocks

Tommy Truong	913660519
Yueling Liu	922272361
Steve Betts	921898143
Nicholas Hamada	918602131

Table of Contents

Github Repository

[kpcrocks]

Project Introduction and Description

The aim of this project is to write a basic file system. We need to format the volume, create and maintain a free space management system, initialize a root directory, and maintain directory information. We need the functions to create, read, write, and delete files as well as display information.

The overall design of the file system begins with the Volume Control Block. Inside the first 512 byte block of storage (index 0) is the legend of the entire volume. Here we store various volume variables such as the location of freespace, location of root as well as the unique signature. Following the VCB is the freespacemap that we allocated 5 blocks. This map is a collection of bits stored inside of bytes, stored inside of the 5 blocks.

A constraint of the file system is that directories are always allocated contiguously and are capped at a maximum of 50 directory entries each. Files, on the other hand, are allocated using indexed allocation. Each file has one or more "index blocks" that each contain up to 127 "pointers" to individual file chunks that may be allocated throughout the volume. The 128th "pointer" points to the next index block, if it exists.

Milestone 1

Descriptions of VCB and FreeSpace

We Initialized a VCB data structure which contains five variables:

```
typedef struct VCB {
    // Dictate the total number of blocks in the volume
    int numBlocks;

    // Dictate how many bytes correspond to a single block
    int blockSize;

    // "Pointer" to first block of freespace bitmap
    int locOfFreespace;

    // "Pointer" to the first block of root directory
    int locOfRoot;

    // unique magic number to identify if the volume belongs to us
    long signature;
} VCB;

//Ensuring the VCB is accessible from anywhere
extern VCB* vcb;
```

- *int numBlocks* — *dictates the total number of blocks in the volume*
- *int blockSize* — *dictates how many bytes correspond to a single block*
- *int locOfFreespace* — *“pointer” to the first block of the free space bitmap*
- *int locOfRoot* — *“pointer” to the first block of the root directory*
- *long signature* — *a unique number to identify if the volume belongs to us*

The VCB structure is the first block of the volume. It is used as an information hub containing all things pertinent to the volume. Initially, the signature is compared against the signature of the VCB to determine the volume’s owner. The remainder of the variables are to determine the total number of blocks in the volume, the size of each block, the start of the free space, and the start of the root directory.

FreeSpace

For our free space structure, we used a bitmap. In our bitmap, we use "1" to indicate a used block and "0" to indicate a free block. We had functions for setting bits to one [void setBitOne()], setting bits to zero [void setBitZero()], and getting bits [bool getBit()]. We also wrote a function for allocating a contiguous set of free blocks, to be used for the root directory.

Description of directory entry

The directory entry system organizes all the files, logically. Each directory entry contains a filename along with a structure of information describing the attributes of the file. For our directory entry structure. We have six properties:

- *char name[256]* — *the unique name of the entry, and is used for lookup*
- *long size* — *the size of the file so we know how far to read up to*
- *int numOfDE* — *the total number of directory entries we want for a directory*
- *int bytesNeeded* — *the bytes we need multiply the size of directory entry*
- *long location* — *the location we want to keep*

- *unsigned char fileType — the file type for directories or files*

For our root directory, we created it by initializing an array of 50 directory entry structs. We then followed the procedure of setting the first two DEs to "." and ".." which both point to the location of the root itself, before writing the array to its designated location in storage.

Description of functions

In milestone 1, we mainly focus on initializing our file system, formatting the volume, initializing the free space, and implementing the functions below:

- **int initFileSystem(unit64_t numberOfBlocks, unit64_t blockSize)**
  - This function is where we started initializing our file system. It takes the number of blocks, and the block size from the client, which with 19531 blocks, and each block size is 512 bytes.
  - initFreespace(): This function is where we use bitmap for our free space management. Based on the values passed in with the number of blocks and the size of blocks, we calculated we will have 5 blocks, and because block 0 is the VCB, we wanted to set the first 6 bits as used and set the remainder as free.
  - The setBitone function is to mark the first 6 bits as used.

- **int getFreespaceSize(int numberOfBlocks, int blockSize)**
  - This function is calls in the initFileSystem function. It takes two parameters: number of blocks and block size. We calculated the bytes and blocks needed in our file system by using the following formulas:

$$int\ bytesNeeded\ =\ (numberOfBlocks\ +\ 7)\ /\ 8;$$
$$int\ blocksNeeded\ =\ (bytesNeeded\ +\ (blockSize\ -\ 1))\ /\ blockSize;$$

- **int initRootDE(int blockSize, int FSSize)**
  - The purpose of this function is to initialize the root directory. We initialized 50 directory entries, calculated bytes and blocks needed, as well as allocated a directory entry pointer. We looped through these directory entries, and initialized each directory entry structure to be free. To achieve this, we simply set the names to “”.
  - We set the first and the second directory entry name to be “.” and “..” as well as initializing the size, location, file type and number of directory entries for each directory entry.

```
strcpy(directoryEntries[0].name, ".");

    directoryEntries[0].size = MAXDE * sizeof(DirectoryEntry);

    directoryEntries[0].location = locOfRoot;

    directoryEntries[0].fileType = FT_DIRECTORY;

    directoryEntries[0].numOfDE = MAXDE;

```

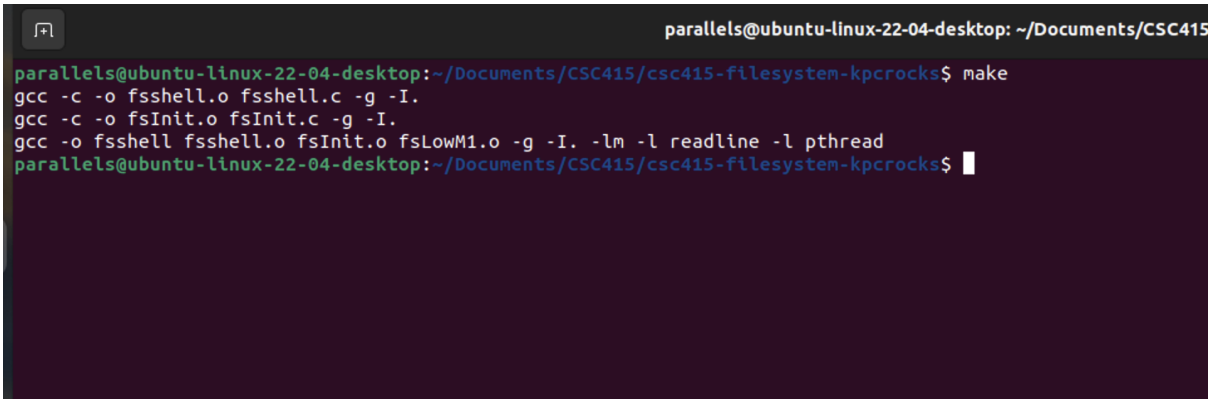
```
    // set the dot dot
    strcpy(directoryEntries[1].name, "..");
    directoryEntries[1].size = MAXDE * sizeof(DirectoryEntry);
    directoryEntries[1].location = locOfRoot;
    directoryEntries[1].fileType = FT_DIRECTORY;
    directoryEntries[1].numOfDE = MAXDE;
```

- **allocContBlocks and allocSingleBlock**
  - These two functions are defined in freespace.c and are used to allocate either a contiguous set of blocks or just a single block.
  - allocContBlocks is used primarily by the functions in milestone 2 to allocate a set of contiguous blocks to represent a directory and store directory entries.
  - allocSingleBlock is used by the functions in milestone 3 to allocate space for a file index block, or to allocate space for file chunks one at a time, whose locations would then be stored in an index block.

Issues and Resolutions

For milestone 1, one issue that we faced was understanding how to use bitmaps. To wrap our heads around bitmaps, we researched more about them from articles and videos online and discussed what we learned during our meetings. When debugging our code, we used printf statements to determine if the outputs were correct. When the outputs were incorrect, we discussed on Zoom how to modify the code so that we got the right outputs. For example, when we created a double pointer for one of the bitmap functions, the code did not compile correctly. Once we realized that we should have just used one pointer instead, we were able to successfully compile the code.

Screenshot(s) of compilation



Milestone 2

Description of functions

In milestone 2, we focused on implementing the key directory functions, such as making, removing, and opening a directory; checking whether a passed in path name is a valid file; etc.

fs_getcwd and fs_setcwd

The get current working directory function takes in pathname and size and returns the globalPath. In the process, it copies the current global path into the passed in pathname. This is an important function for parsedPath as it returns the global path that is needed to concatenate with the input path. This along with setcwd changes the relative path to an absolute path.

The set current working directory function changes the current working directory or in our program, the globalpath. The function first checks if the path passed in is a file or directory. Then the function checks for if the path is just root (“/”). If that is the case, then we would just concatenate the path and set that as the global path. Otherwise, we would concatenate “/” and the path before setting that as the final path.

fs_isFile and fs_isDir

The fs_isFile function checks to confirm if the passed in argument is a file or directory. This function first calls the parsedPath function to parse the passed in parameter. The parsedPath returns a struct of type fdPathResult. This structure contains the directory (n-1) and the index of the passed in parameter. With that information, we first LBAread in the location of directory (n-1). Then, we iterate through the directory entries at directory(n-1) to find the file with the same name and compare the fileType with FT_REGFILE.

Similar to fs_isFile, fs_isDir runs parsedPath, LBAreads in the directory (n-1), iterates through the directory entries, and compares the final argument with FT_DIRECTORY.

markChunksFree

This function serves as a free space setter and is called exclusively by fs_delete. The function first LBAread up the freeSpaceMap, and we run setBitZero at the passed-in parameter indexBlockLoc, which is the location of the file's first index block. This marks the first index block as free. Next, we calloc some space to load in the values within the first index block. From there, we iterate through and for every valid "pointer" indicating an allocated file chunk, we set its corresponding freespace bit as free. If the end of the index block is reached, the next index block in the chain is loaded to continue freeing file chunks. Finally, we write back to freeSpaceMap.

fs_mkdir

fs_mkdir is a function to make a directory. It returns a negative 1 when it fails making a directory. This function takes two parameters, a pathname and mode_t mode. We begin with parsing the passed-in pathname. If the file already exists, it returns a negative 1. If the file didn't exist, it iterates through the directory entries and starts from index 2 since the first and the second directory entry will be (".") and ("..."). When it finds the first available directory entry slot, it calls getFreeSpaceSize function to prepare freespace, and then begins preparing the new directory's directory entry by strcpy, the passed-in pathname, as well as setting the max number of directories, file type, and location. Then, it prepares the new directory itself by setting its first two directory entries to (".") and ("..."). Finally, both the new directory and its directory entry are written back to disk.

fs_delete and fs_rmdir

fs_delete is a function to delete a file. First, it runs parsedPath, and we calloc some space to load up the directory entries. Then, we LBAread the path into our calloc-ed parentDir. We run markChunksFree to ensure all of the file's index blocks and file chunk locations are marked free and not in use in the freespace bitmap. From here, we set the file's directory entry name as (""), indicating that the file is not in use. Finally, we write parentDir to the location.

fs_rmdir is a function to remove a directory. First, it runs parsedPath, and we need to access the directory that we want to remove by reading in its parent directory. Then we LBAread the directory blocks into parent directory, and we read in the directory we want to remove. It iterates through the directory, checking each directory entry except the (".") and ("..."), then we mark the blocks as free, set the directory that needs to be removed to be free, set its name to be (""), location, file type, and number of directories to be zero. Finally, we write freespace and parent directory back to disk and free the memory.

fs_readdir

fs_readdir takes a pointer and returns a pointer to fd_diriteminfo struct. The fd_diriteminfo struct contains d_name, fileType, and d_reclen. It returns null when error occurs. We first iterate through directory entries from the directory entry position of that file to the max directory entries we have. We check whether the directory is used; when this directory is not used, we copy the buffer name from our directory entry to the struct, as well as file type, then return the pointer that points to the struct.

fs_opendir and fs_closedir

fs_opendir allocates space and creates a folder descriptor. First, we run parsedPath and LBAread the path into a tempBuffer. Then, we check if the folder at the index is a directory. From here, we malloc some space for the file descriptor and assign the starting location. We also malloc some space for the diriteminfo.

fs_closedir function frees the malloc-ed memory in fs_opendir.

struct fdPathResult parsedPath

The purpose of parsedPath is to iterate through the layers of directory entries to confirm if the passed in path is valid. To achieve this, first we have to confirm whether the input path is absolute or relative. This is determined by the first character in the input path. If the very first character is a “/”, it is an absolute path. If the first character is not “/”, then the path is relative to the current working directory.

To begin, we will start with absolute paths. After a check for relative or absolute, we tokenize the inserted path. With the tokens inserted into an array, we can now iterate through the directory entries to confirm the path. The initial for loop we create is to loop through the values inside of the tokenArray. The next loop is a bit more complicated. The while loop starts at 0 and iterates through the directory entries until it hits the total maximum count of directory entries. As it loops, it compares the token with the directory entry’s name. Once the first token is found, we LBAread in that directory entry’s location, the while loop breaks, and then the next token is compared with the new set of directory entries. This loop continues for however many tokens are available.

Ultimately, parsedPath returns a struct with three values:

```
int dirPtr          // pointer to directory
int index           // index of file or directory
char lastArg[20]    // the name of the final file or directory
```

In the process of looping, if the iterator is equal to n - 1, this means that we are currently in the second to last argument. This would mean that we are in the folder which contains the final argument. At this point, we want to store the index of the final argument in the n-1 folder. Also, if we reach the end of the directory entries, and the file does not exist, we still want to return the directory pointer but return the index as -1. The -1 allows us to know that the path was not valid. Otherwise, if after all the loops are run and the if statements are handled, we return the result with the directory pointer, index and lastArg.

With the absolute path handled, we now work on the relative path. For relative paths, we grab the global path and concatenate it with a “/” as well as the path that was passed in. We then run parsedPath on the new path. Since the new path is an absolute path, the function will process the path and return the validity of the path. At the end, just like in absolute path, we return dirPtr, index, and lastArg.

fs_pathReconstruction

Path reconstruction came about as a handler for a specific edge case. If the user passes in /banana/apple/.././ parsedPath would handle the path but the global path would still remain as /banana/apple/.././ when it should be /banana. To solve this, we first grab the global path after it has been processed by parsedPath and tokenize it to see if there are (“..”) or (“.”) to set off a flag. From here, we set a loop to iterate through the global path and search for (“.”) and (“..”). If the token is neither (“.”) or (“..”) then the token is added to a new array called finalPathArray and count is incremented. Count manages the position in the finalPathArray. If the next token is (“.”) then nothing happens. Finally, if the token is (“..”) then

the value at finalPathArray[count] is set as (“”) and the count is decremented. This solves our problem as (“..”) would delete the previous value and the final path would only contain the final absolute path.

In the end, we concatenate the “/” and copy the path to the global path.

fs_stat

This simple function first uses parsedPath() on a supplied path parameter to load up a sought directory entry, then uses information from the DE to fill in fields such as st_size and st_blocks within a supplied fs_stat structure.

Issues and Resolutions

One issue that we had was understanding the readdir() function. Initially, we did not understand the directory pointer (dirP). Tommy thought that the dirP was a pointer to a struct fs_disiteminfo, but this was convoluted and did not make sense. To resolve this issue, we watched a Zoom lecture from another section, which helped to clarify things. We saw that dirP was a pointer to dir, which made more sense as we needed to load the directory entries into memory and iterate through them.

Another issue we faced had to do with the fs_isDir() function not working. When we initially tried to run the program, we got a free() invalid pointer. During one of our meetings, we spent most of the time trying to debug this. It took a long time to debug since we could not figure out what was causing the error. To solve this issue, we commented out all of the code and uncommented sections of code to pinpoint where the error was coming from. After doing so, we finally figured out that the error was coming from LBAread(); MAXDE had to be changed to blocksNeededForDir(MAXDE). Once we fixed this, we no longer had errors with fs_isDir().

```
// LBAread (tempBuffer, MAXDE, dirPtr);
LBAread(tempBuffer, blocksNeededForDir(MAXDE), tempPath.dirPtr);
```

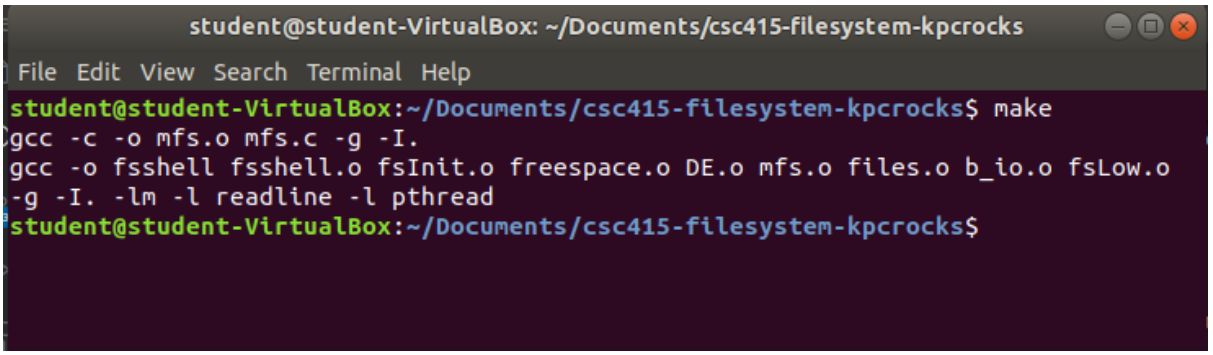
We also had an issue with setcwd that allowed us to change directories into a file:

```
Prompt > pwd
/banana
Prompt > ls

jeep
Prompt > cd jeep
Prompt > ls
return NULL from opendir
Prompt >
```

To fix this problem, we added our isFile to return -1 so we would not be able to change directories into a file.

```
student@student-VirtualBox:~/Documents/csc415-filesystem-kpcrocks$ make
make: 'fsshell' is up to date.
student@student-VirtualBox:~/Documents/csc415-filesystem-kpcrocks$ 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

banana
Prompt > pwd
/
Prompt > cd banana
Prompt > ls

jeep
Prompt > cd jeep
Could not change path to jeep
Prompt >
```

Screenshot(s) of compilation



Milestone 3

Description of functions

In milestone 3, we focused on implementing interfaces that will be interacting with the driver. We implemented b_read, b_open, b_write, b_seek, and b_close functions.

Files are allocated using indexed allocation where "pointers" to individual file chunks are contained within index blocks. In addition, multiple index blocks for a large file are linked together like a linked list and many of the helper routines mentioned below will involve creating or traversing the index blocks of files.

The structures used in milestone 3 include **b_fcb**, the **file control block**, which contains a local buffer, a **chunkNumber** representing the current working n-th chunk of a file, and a **chunkOffset** serving as byte marker inside the current chunk, among a few other tracking variables. It also stores a pointer to a **fileInfo struct**, which serves as a miniature copy of the directory entry for a file.

createIndexBlock

This function is defined in files.c and takes no parameters. Its purpose is to construct an integer array with enough space for 128 integer file locations, ask the freespace for a single LBA block to store it, then write this array to the disk to serve as a file's index block. The function returns the LBA location of the newly written index block.

makeNewFile

This function is defined in files.c and takes a pathname as a parameter. It is designed to be called inside b_open() in the O_CREAT case and is responsible for creating the directory entry for the new file using largely the same logic as fs_mkdir. However, it calls createIndexBlock to obtain the location of a new index block to assign to the DE's location variable.

getFileInfo

Defined in files.c and designed to be called by b_open, getFileInfo takes a file path and calls parsedPath to obtain the directory entry for a file. It then copies the file size and index block location into a new fileInfo struct object. The validated file path is then copied to the fileInfo struct to serve as the file name.

b_open

b_ open function takes in two parameters: a filename and an integer flag. We check the mode of this flag, set it to be the corresponding mode, and set the file information, including the localBuff, index, chunkNumber, and the currentIndexBlockLoc. In this function,

there is a case where if the flag is 0_CREAT, which means we must create a file, the makeNewFile function handles this case and makes a file.

The makeNewFile function takes in the filename; it parses this filename by calling the parsedPath function. To make a new file, we need to load a directory for this file. We allocated the size of VCB->blockSize to the number of blocks needed for the directory, and we read it to the memory. The makeNewFile function loops through all the directory entries, when it finds the first available directory entry slot it starts making a file in that slot.

The createIndexBlock function is called inside the makeNewFile function. This function writes a new index block to disk and returns its disk location.

b_read

b_read takes a file descriptor, a caller-supplied buffer, and a count of bytes to read and will attempt to fill that buffer with the requested amount of data from the file. The function first calculates the bytes remaining in the file based on the starting position (as set in the file control block). If the requested count is greater than the bytes remaining, the remaining bytes is set as the new count instead. If no bytes remain, the function is aborted.

Otherwise, if the count is large, the function will transfer entire block-size chunks directly to the caller's buffer until the count is less than block/chunk. When the count is less than chunk size, data from the file will first be transferred into a local buffer before being copied to the caller's buffer. All the while, tracking variables in the FCB such as chunkNumber and chunkOffset are being updated for the benefit of the next b_read() call.

getBlockN

The getBlockN function was defined in files.c and is one of the critical helper functions utilized in both b_write and b_read. It takes an integer and a pointer that points to fileInfo struct and returns the LBA location of the n-th chunk of a file. We used the following formulas to find out which index block the n-th chunk is pointed to by and the index number within the block:

$$int\ blockNumber\ =\ n\ /\ ((vcb\ ->\ blockSize\ -\ sizeof(int))/sizeof(int));$$
$$int\ indexInBlock\ =\ n\ \%\ ((vcb\ ->\ blockSize\ -\ sizeof(int))/sizeof(int));$$

Starting from the first index block at *fileInfo fi->location, a while loop is then used to iterate through the collection of index blocks storing the file chunk locations. This is similar to traversing a linked list. The while loop stops at blockNumber (terminating early if -1 is encountered where the "pointer" to the next index block should be) and returns the file chunk location value stored at indexInBlock.

getBlockN serves to abstract the indexed file allocation and allows b_write and b_read to operate similarly to if the file chunks were allocated contiguously.

getIndexBlockLoc

Somewhat similar to getBlockN, although given an integer representing the n-th chunk of a file, it instead returns the location of the index block that stores the location of the n-th chunk. For example, assuming one calls the function passing the integer 150 representing the 150th chunk of a file, and assuming block size 512 bytes, the function should return the LBA location of the file's second index block, since each index block stores the locations of 127 chunks. This function is called to update a "currentIndexBlockLoc" variable inside the b_fcb struct whenever b_fcb.chunkNumber is incremented or reset.

b_write

b_write writes up to a number of bytes to the file from the buffer; both the number of bytes and buffer are passed in as arguments. It returns the number of bytes written if on success. In this function, we first check a base condition on whether the passed-in fd is a valid file descriptor. We also check if the chunk for this file exists by calling getBlockN function.

The major logic of the b_write function was dealing with the passed-in count. The base case is for when the count is smaller than the chunk size, it will just read 1 block and the fileChunk to the fcbArray[fd].buffer.

The first edge case is when chunkOffset starts off greater than 0 and the passed in count is greater than the available space left in the starting chunk. In this case, we write to the remaining space of the starting chunk and then increment the chunkNumber.

Another case we needed to handle is when the count is bigger than the chunk size, so we copy 512 bytes to the local buffer then write localBuff to the fileChunk, then we grab the next fileChunk and repeat for every 512 chunks in count. Afterwards, the base case would handle the remaining bytes.

Finally, an important portion of this function deals with the situation that the next fileChunk does not exist for new bytes to be written in. This could happen when writing to a brand new 0 byte file, when write() fills up an entire chunk, or when b_seek() sets the chunkNumber + chunkOffset to EOF. In these situations, we would need to call initializeWritableChunks() to add more blocks.

initializeWritableChunks

This is another utility function in files.c that takes an index block location and an amount of bytes needing to be written to a file as parameters. Based on the amount of bytes, the function calculates the number of file chunks to be initialized using a formula that increases the amount of chunks by doubling amounts until it is more than enough to cover the count.

Then, the function traverses the given index block, asking the freespace to allocate single blocks for file chunks and assigning the locations to the index block until all the

chunks are written or the end of the index block is reached. In the latter case, the function will also call createIndexBlock, link the new block to the current block, and then continue initializing/assigning file chunks in the next block.

This function is designed to be called by b_write to ensure that file chunks existed for the function to be written to.

b_seek

The b_seek function sets the file offset for the open file descriptor. It takes in three parameters: the file descriptor, offset, and an integer whence. We initialize a currentOffset variable; this currentOffset is the chunk number * 512 + the chunk offset of the file descriptor.

b_seek also checks whence. When the whence is SEEK_SET, which means the passed in offset will be set at the location offset from the beginning of the file, it returns the offset. When the whence is SEEK_CUR, the offset will be set at the location offset from the previous offset of the file, it returns the currentOffset + the passed in offset. When the whence is SEEK_END, it will be set from where the end of file size plus the passed in offset.

b_close

b_close sets the size of file so future calls would be able to utilize that information. The function also frees the file information of fcbArray[fd] and releases the buffer for the fcbArray[fd].

Issues and Resolutions

Near the end of the project, Steve discovered that when we created a file, its index block was created at location 0 and written over the VCB. This indicated that something was wrong with the freespace functions like allocSingleBlock or the freespace itself. One issue with the freespace is that even though it was declared a global extern variable, the bitmap would not stay in memory on its own, and thus the freespace would occasionally "reset" to all zeroes. This was fixed by making sure we LBAread to the global freespace variable every time it was to be invoked. We had issues with deleting files. When we set the bits to 1s, it worked fine, but when we set the bits to 0s, it did not work correctly. The output seemed to set the entire 8 bits to zero instead of a single bit. We fixed the setBitZero function which was previously

freeSpaceMap[i >> 3] &= (1 << (i & 0x7))

to

freeSpaceMap[i >> 3] &= ~(1 << (i & 0x7)) (adding the bitwise not operator)

which led to the correct output.

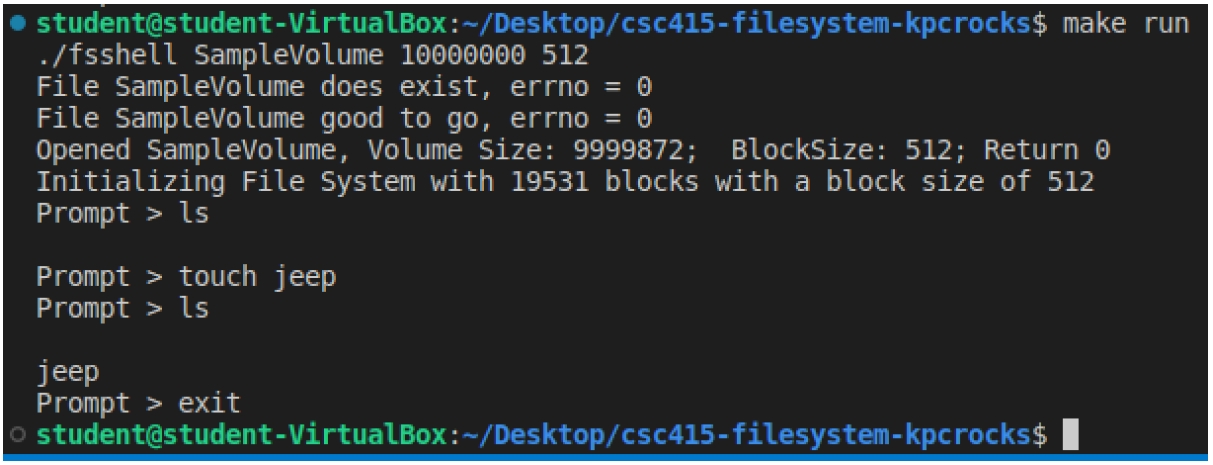
```
after  Freespace: 11111111111111111111111111111110000000011111111111111111111110
bit to set 0: 38
before Freespace: 11111111111111111111111111111110000000011111111111111111111110
after  Freespace: 11111111111111111111111111111110000000011111111111111111111110
bit to set 0: 39
before Freespace: 11111111111111111111111111111110000000011111111111111111111110
after  Freespace: 11111111111111111111111111111110000000011111111111111111111110
bit to set 0: 40
before Freespace: 11111111111111111111111111111110000000011111111111111111111110
after  Freespace: 11111111111111111111111111111110000000000000000011111111111110
bit to set 0: 41
before Freespace: 11111111111111111111111111111110000000000000000011111111111110
after  Freespace: 11111111111111111111111111111110000000000000000011111111111110
bit to set 0: 42
before Freespace: 11111111111111111111111111111110000000000000000011111111111110
after  Freespace: 11111111111111111111111111111110000000000000000011111111111110
bit to set 0: 43
before Freespace: 11111111111111111111111111111110000000000000000011111111111110
after  Freespace: 11111111111111111111111111111110000000000000000011111111111110
bit to set 0: 44
before Freespace: 11111111111111111111111111111110000000000000000011111111111110
after  Freespace: 11111111111111111111111111111110000000000000000011111111111110
bit to set 0: 45
before Freespace: 11111111111111111111111111111110000000000000000011111111111110
after  Freespace: 11111111111111111111111111111110000000000000000011111111111110
bit to set 0: 46
before Freespace: 11111111111111111111111111111110000000000000000011111111111110
after  Freespace: 11111111111111111111111111111110000000000000000011111111111110
bit to set 0: 47
before Freespace: 11111111111111111111111111111110000000000000000011111111111110
after  Freespace: 11111111111111111111111111111110000000000000000011111111111110
bit to set 0: 48
before Freespace: 11111111111111111111111111111110000000000000000011111111111110
after  Freespace: 11111111111111111111111111111110000000000000000000000011111110
bit to set 0: 49
before Freespace: 11111111111111111111111111111110000000000000000000000011111110
after  Freespace: 11111111111111111111111111111110000000000000000000000011111110
bit to set 0: 50
before Freespace: 11111111111111111111111111111110000000000000000000000011111110
after  Freespace: 11111111111111111111111111111110000000000000000000000011111110
```

Screenshot(s) of compilation

```
student@student-VirtualBox:~/Desktop/csc415-filesystem-kpcrocks$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o freespace.o freespace.c -g -I.
gcc -c -o DE.o DE.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o files.o files.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o freespace.o DE.o mfs.o files.o b_io.o fsLow.o -g -I. -lm -l readline -l pthread
student@student-VirtualBox:~/Desktop/csc415-filesystem-kpcrocks$
```

Screenshots of Output

ls - List the file in a directory

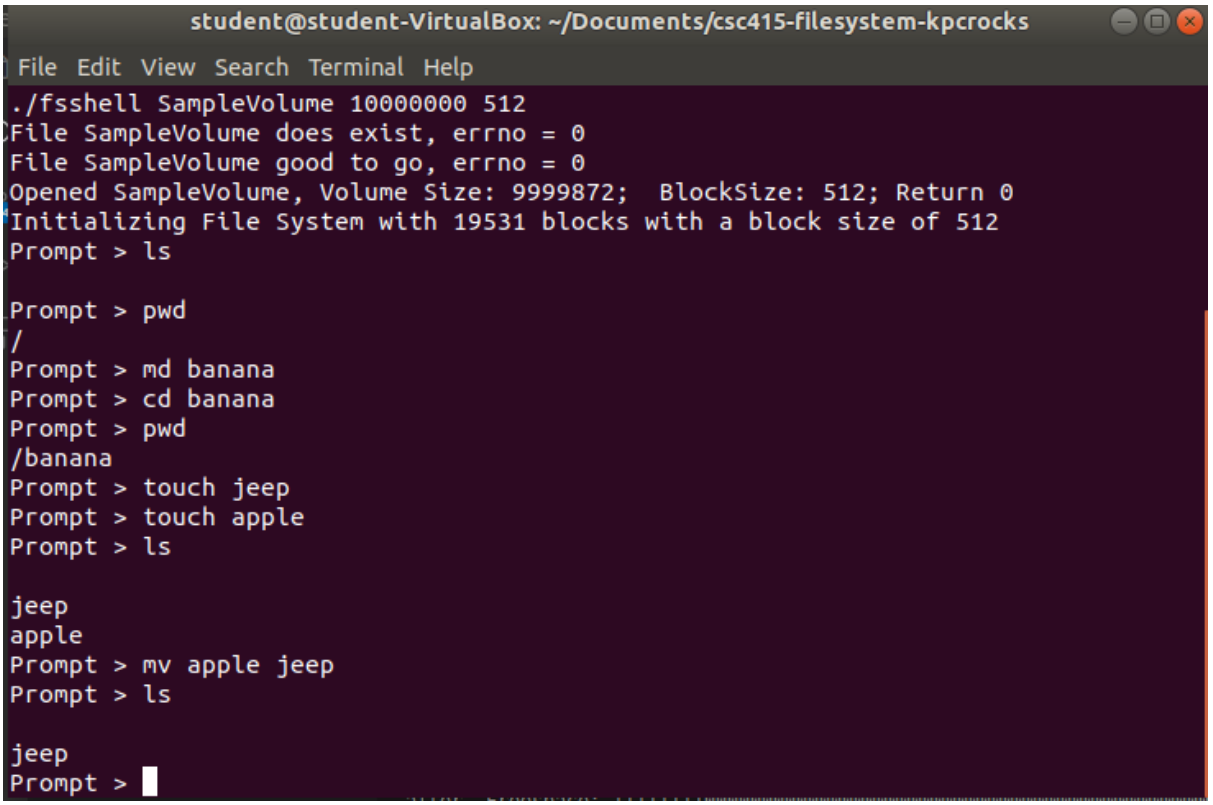


cp - Copy a file - source [dest]

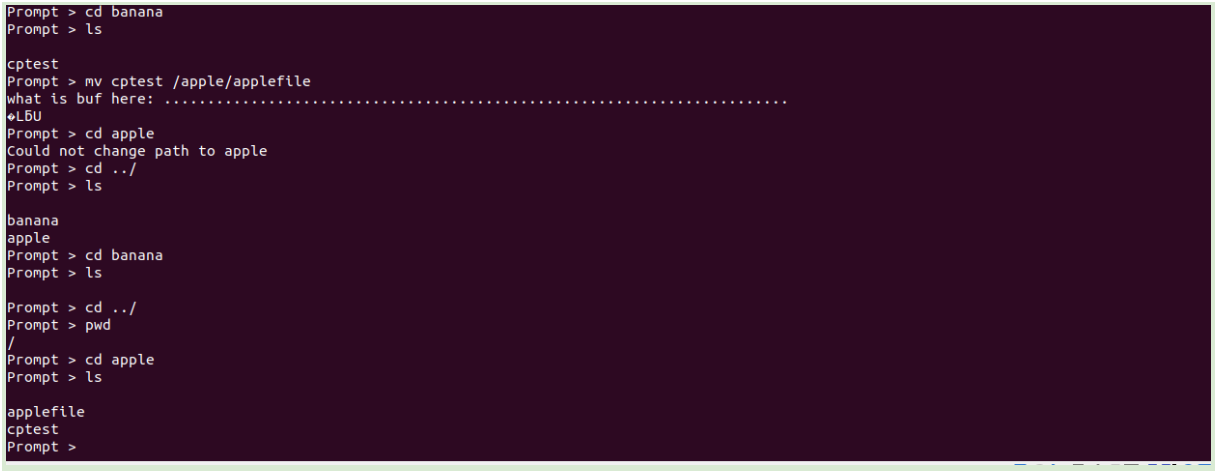
```
File  Edit  View  Search  Terminal  Help
student@student-VirtualBox:~/Documents/csc415-filesystem-kpcrocks$ make
make: 'fsshell' is up to date.
student@student-VirtualBox:~/Documents/csc415-filesystem-kpcrocks$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does not exist, errno = 2
File SampleVolume not good to go, errno = 2
Block size is : 512
Created a volume with 9999872 bytes, broken into 19531 blocks of 512 bytes.
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > ls

Prompt > pwd
/
Prompt > ls

Prompt > md banana
Prompt > cd banana
Prompt > touch cptest
Prompt > ls

cptest
Prompt > cd ../
Prompt > pwd
/
Prompt > md apple
Prompt > ls

banana
apple
```

```
Firefox Web Browser
student@student-VirtualBox: ~/Documents/csc415-filesystem-kpcrocks
File  Edit  View  Search  Terminal  Help
/
Prompt > cd banana
Prompt > ls

cptest
Prompt > cp2fs /home/student/temp cptest
Prompt > cat cptest
.....................................................................
Prompt > cd ../
Prompt > pwd
/
Prompt > ls

banana
apple
Prompt > cd banana
Prompt > ls

cptest
Prompt > cp cptest /apple/applefile
Prompt > cd apple
Could not change path to apple
Prompt > cd ../
Prompt > pwd
/
Prompt > cd apple
Prompt > ls

applefile
Prompt > cat applefile
.....................................................................
Prompt >
```

mv - Move a source file to a destination



Move a file in a folder to another folder



md - Make a new directory

```
parallels@ubuntu-linux-22-04-desktop:~/Documents/CSC415/csc415-filesystem-kpcrocks$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does not exist, errno = 2
File SampleVolume not good to go, errno = 2
Block size is : 512
Created a volume with 9999872 bytes, broken into 19531 blocks of 512 bytes.
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > ls

Prompt > pwd
/
Prompt > md banana
Prompt > ls

banana
Prompt > cd banana
Prompt > md apple
Prompt > ls

apple
Prompt > cd apple
Prompt > ls
```

rm - Remove a directory

```
student@student-VirtualBox: ~/Documents/csc415-filesystem-kpcrocks
File  Edit  View  Search  Terminal  Help
student@student-VirtualBox:~/Documents/csc415-filesystem-kpcrocks$ make
make: 'fsshell' is up to date.
student@student-VirtualBox:~/Documents/csc415-filesystem-kpcrocks$ 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 banana
Prompt > ls

banana
Prompt > rm banana
Prompt > ls

Prompt >
```

touch - Create a file

```
student@student-VirtualBox:~/Desktop/csc415-filesystem-kpcrocks$ 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 > touch jeep
Prompt > ls

jeep
Prompt > exit
student@student-VirtualBox:~/Desktop/csc415-filesystem-kpcrocks$
```

jeep's directory entry in root

```
001040: 6A 65 65 70 00 00 00 00  00 00 00 00 00 00 00 00 | jeep............
001050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001080: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010A0: 00 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  00 00 00 00 00 00 00 00 | ................
001140: A7 04 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | §...............
001150: 23 00 00 00 00 00 00 00  08 00 00 00 00 00 00 00 | #...............
```

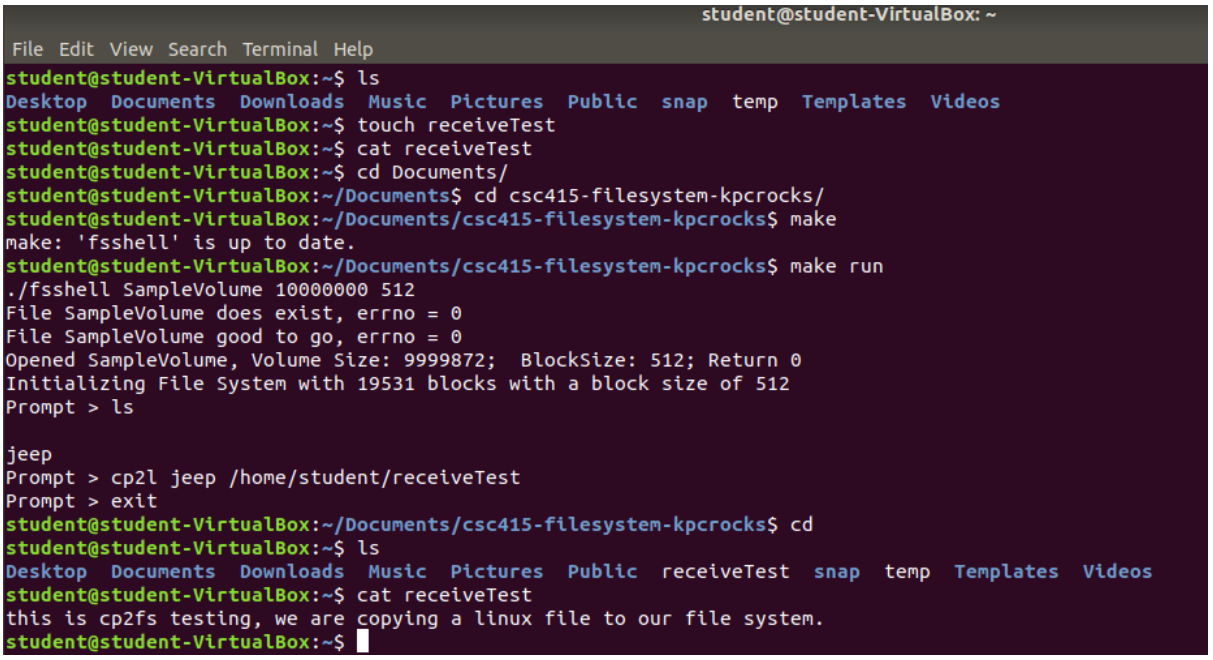
jeep's index block, containing only -1s

```
004800: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004810: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004820: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004830: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004840: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004850: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004860: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004870: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004880: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004890: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0048A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0048B0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0048C0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0048D0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0048E0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0048F0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ

004900: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004910: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004920: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004930: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004940: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004950: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004960: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004970: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004980: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004990: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0049A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0049B0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0049C0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0049D0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0049E0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0049F0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

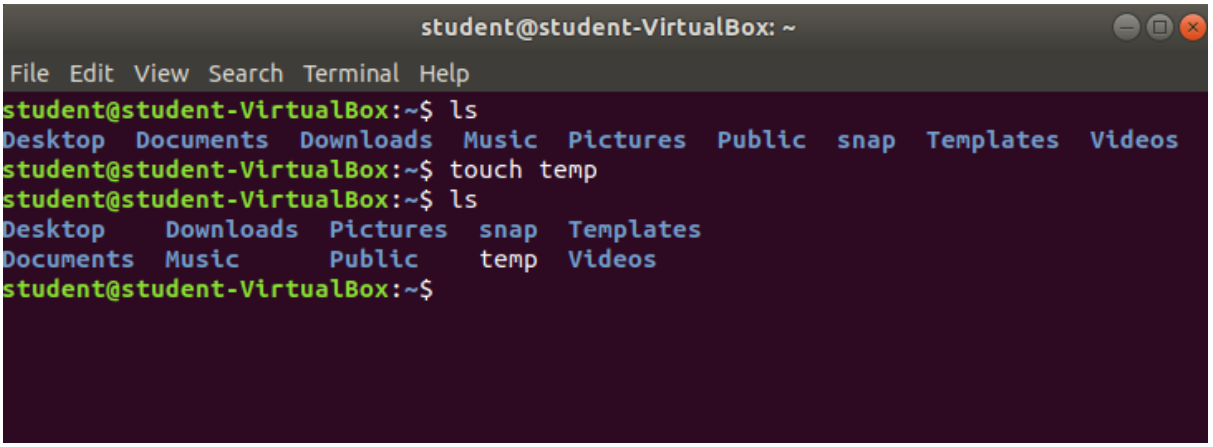
cat - Display the contents of a file

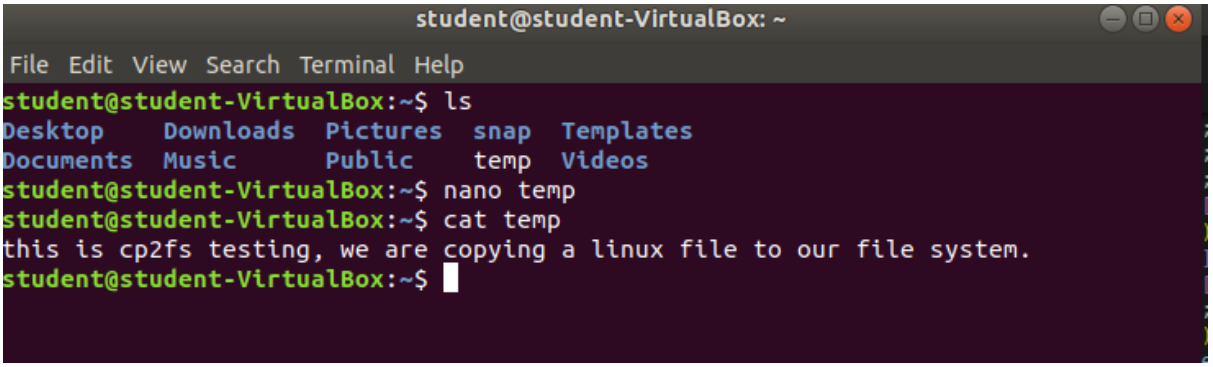
```
student@student-VirtualBox: ~/Documents/csc415-filesystem-kpcrocks
File  Edit  View  Search  Terminal  Help
student@student-VirtualBox:~/Documents/csc415-filesystem-kpcrocks$ make
make: 'fsshell' is up to date.
student@student-VirtualBox:~/Documents/csc415-filesystem-kpcrocks$ make run
./fsshell 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

Prompt > touch jeep
Prompt > ls

jeep
Prompt > cp2fs /home/student/temp jeep
Prompt > cat jeep
this is cp2fs testing, we are copying a linux file to our file system.
Prompt >
```

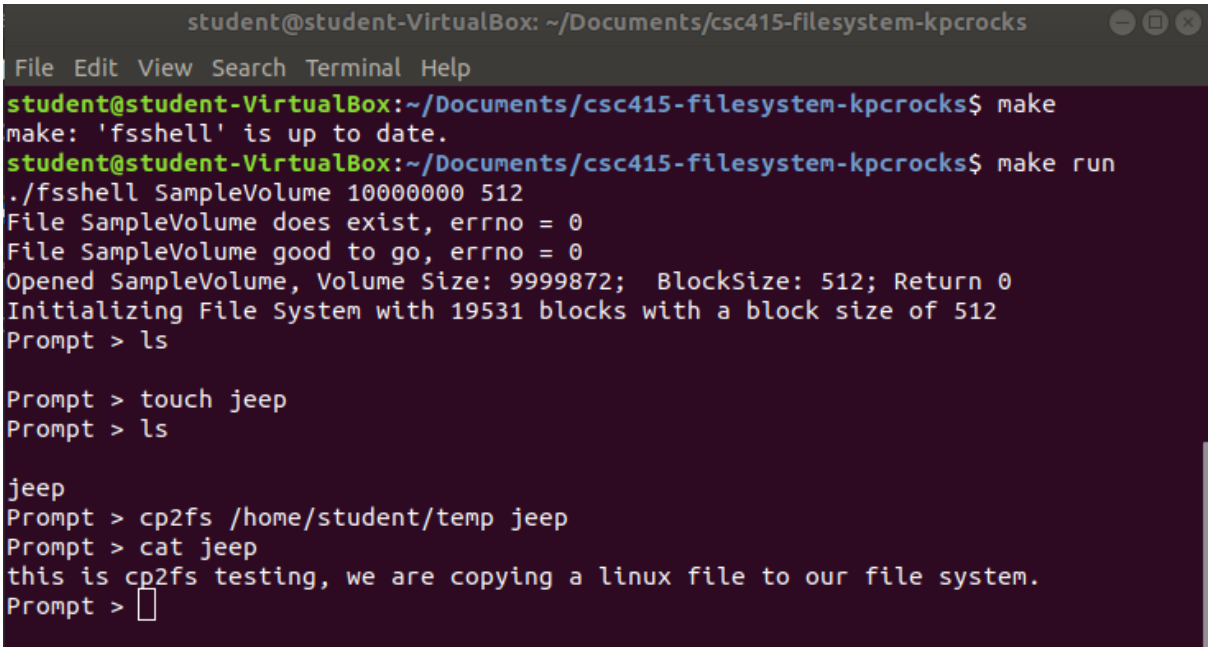
cp2l - Copy a file from test file system to the linux file system



cp2fs - Copy a linux file to file system





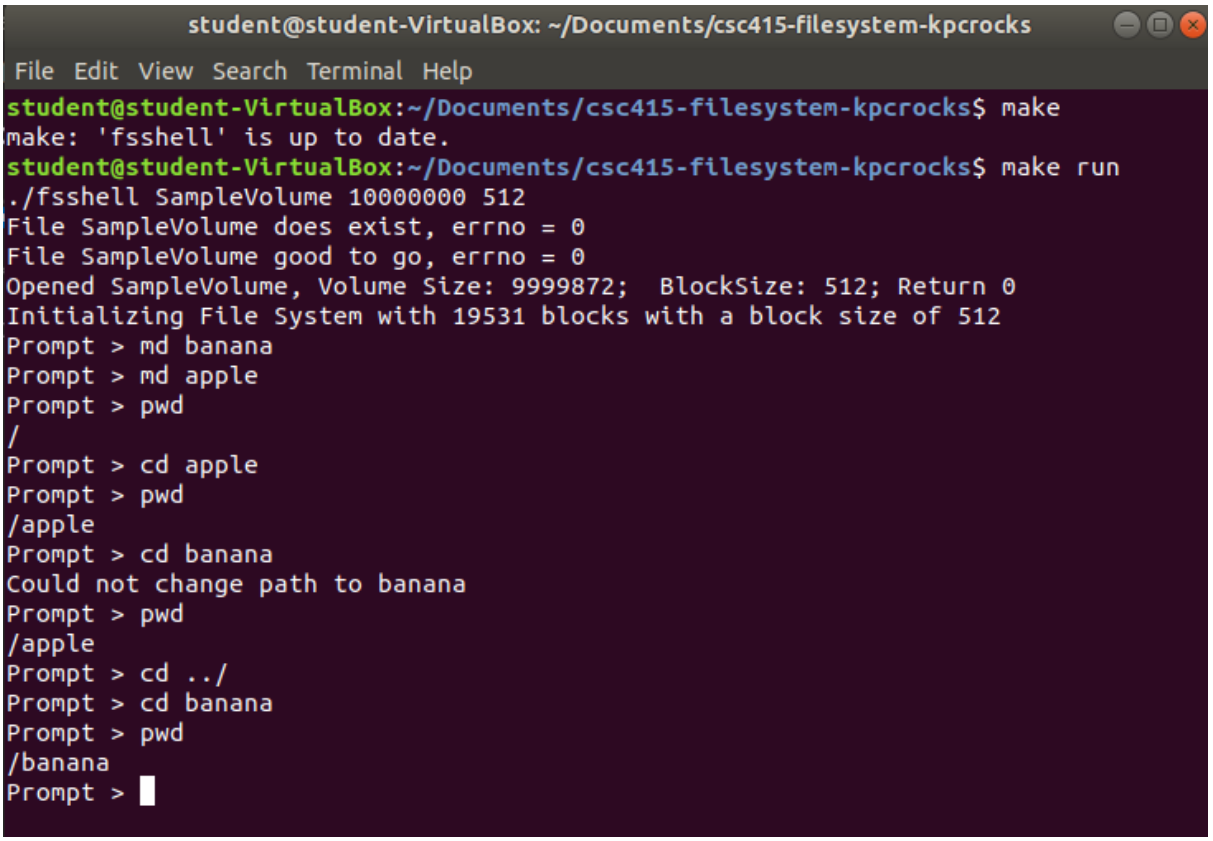


```
004A00: 74 68 69 73 20 69 73 20  63 70 32 66 73 20 74 65 | this is cp2fs te
004A10: 73 74 69 6E 67 2C 20 77  65 20 61 72 65 20 63 6F | sting, we are co
004A20: 70 79 69 6E 67 20 61 20  6C 69 6E 75 78 20 66 69 | pying a linux fi
004A30: 6C 65 20 74 6F 20 6F 75  72 20 66 69 6C 65 20 73 | le to our file s
004A40: 79 73 74 65 6D 2E 0A 00  00 00 00 00 00 00 00 00 | ystem..........
004A50: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004A60: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004A70: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004A80: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004A90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004AA0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004AB0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004AC0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004AD0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004AE0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004AF0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

004B00: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004B10: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004B20: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004B30: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004B40: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004B50: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004B60: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004B70: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004B80: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004B90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004BA0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004BB0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

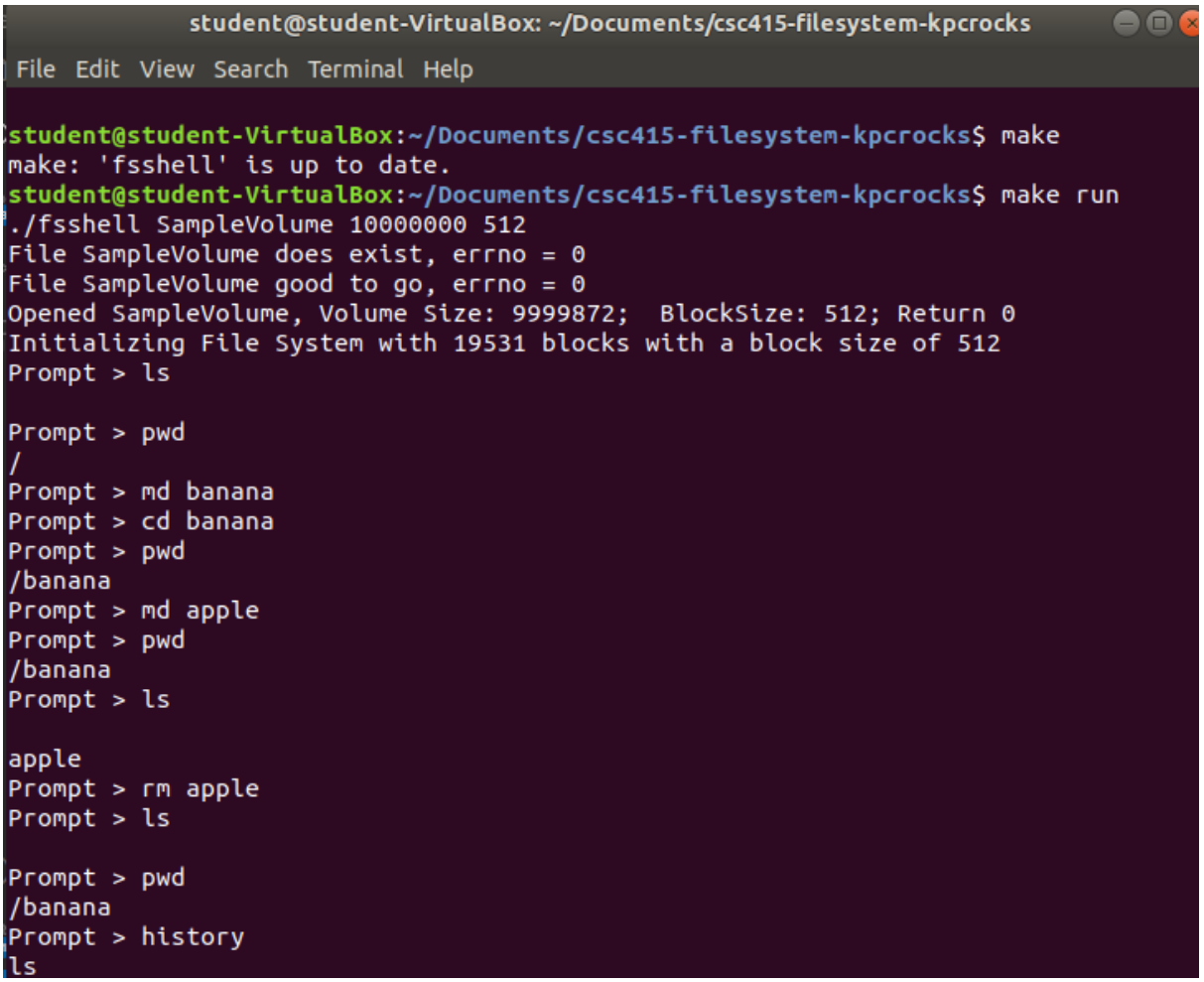
cd - Change a directory

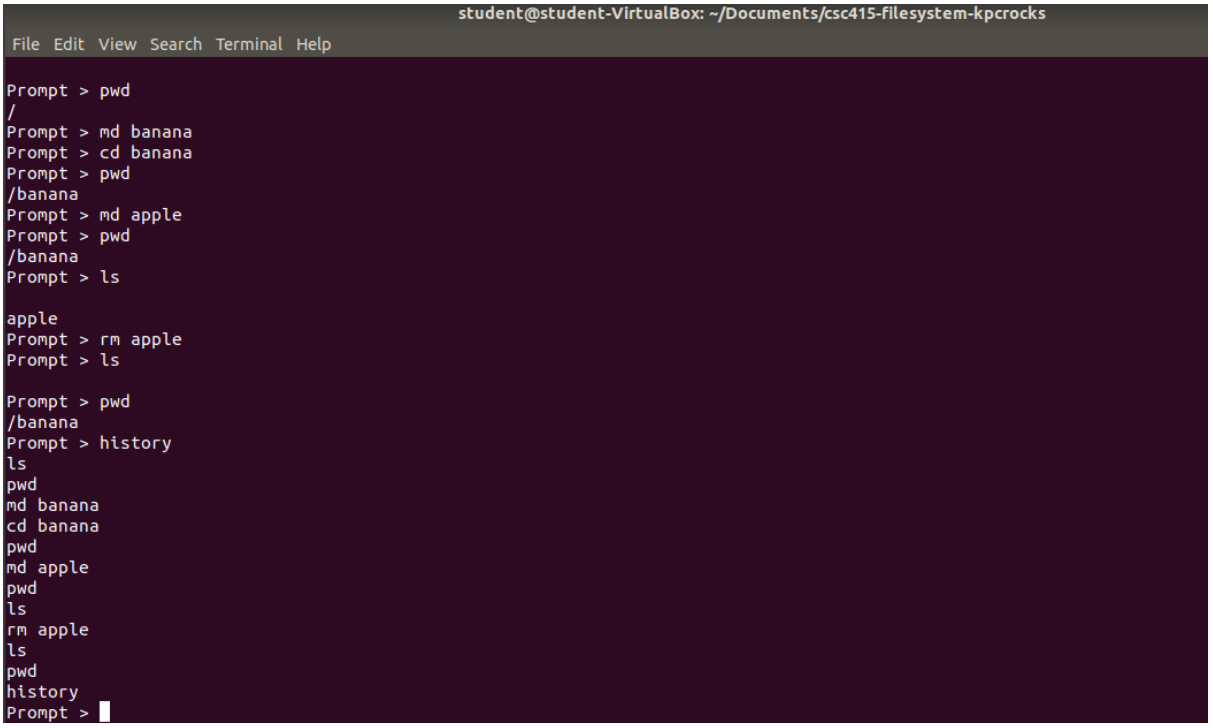
```
student@student-VirtualBox: ~/Documents/csc415-filesystem-kpcrocks
File  Edit  View  Search  Terminal  Help
student@student-VirtualBox:~/Documents/csc415-filesystem-kpcrocks$ make
make: 'fsshell' is up to date.
student@student-VirtualBox:~/Documents/csc415-filesystem-kpcrocks$ make run
./fsshell 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

Prompt > md banana
Prompt > md apple
Prompt > cd apple
Prompt > pwd
/apple
Prompt > cd ../
Prompt > pwd
/
Prompt > cd ../
Prompt > pwd
/
Prompt >
```

pwd - Print the working directory



history - Print out the history





help - Print out help

```
student@student-VirtualBox:~/Desktop/csc415-filesystem-kpcrocks$ 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 > help
ls      Lists the file in a directory
cp      Copies a file - source [dest]
mv      Moves a file - source dest
md      Make a new directory
rm      Removes a file or directory
touch   Touches/Creates a file
cat     Limited version of cat that displace the file to the console
cp2l    Copies a file from the test file system to the linux file system
cp2fs   Copies a file from the Linux file system to the test file system
cd      Changes directory
pwd     Prints the working directory
history Prints out the history
help    Prints out help
Prompt > exit
student@student-VirtualBox:~/Desktop/csc415-filesystem-kpcrocks$
```