Daniel Simpson

Jay Jaber

Travis Nguyen

Malcolm Angelo de Villar

CSC415-03

12/4/2020

Our file system is a fairly simple file system that uses a simple bitmap for free space management, and uses contiguous allocation. One shortcoming of our file system is that our directories don't utilize extents, only files. The directories can only hold a limited amount of directory entries. One unique aspect of our file system is that the directory entry arrays in the directories are just arrays of integers corresponding to a logical block array index where a directory entry is located. This allows the directories to only take up one block, with the tradeoff of requiring an extra read to retrieve the directory entry. Looking back, this was maybe not the best way to do it, as the best performance practice is to avoid disk reads and writes as much as possible. However, it made sense to us at the time of implementation as it allowed us to always know that a directory and directory entry each take up one block of data, instead of either having a very large initial block allocation for a directory(in a tiny volume) or needing to keep track of the block size of the directory.

We had multiple issues in the process of implementing our file system. One of the initial issues we had was just figuring out where to start. The whole file system project seemed overwhelming at first, especially when we saw how many files were included as starter code. It took us a while to recognize that we could ignore many of those files at first, and that we should start with the initialization of the file system. After completing that initial step of initialization, all the other tasks seemed to naturally present themselves as we went along. Of course there were many bumps in the road along the way, but functionality progressed at a somewhat consistent rate overall. However I do wish we would have done earlier testing with the shell or at least looking at how the lower level functions were used in it, because many of the shell functions provided guidance on the intended functionality/implementations of the functions that they called in the mfs and b_io implementation files. One very good example of why this would have been a good idea, was how I struggled with the logic of b_read and b_write, and specifically how they handle writing or reading less than a block's worth of data. If I had looked at the cp2l and cp2fs functions in the fsshell, I think I would have been able to understand how to handle that behavior much quicker, as the cp2l implementation showed that I just needed to return the amount of bytes read from b_read and the shell would handle the rest. I was also overthinking the b_write function by trying to figure out how to write less than a block's worth of data, when I should have realized that was not the intention just by how the file system can only work in blocks and not bytes. Another issue we had was figuring out how to implement pwd in a good way. In its current state, it basically takes a path, iterates up the path and concatenates the names of each parent directory to a string, returning it at the end. We unfortunately didn't have

time to fix this convoluted way of returning the current path and it will return a path that is incorrect at times, especially when cd dot dot is used a lot, although it won't crash and will reset back to normal sometimes as well. This kind of is related to the most obvious issue of our file system, which is it's inefficiency. We followed the philosophy of just getting all the functions working, and optimizing later if we had time. Unfortunately, we never really had the time to delve into optimization, and there is a lot of unnecessary reads/writes to and from disk. As we know from the textbook, we ideally want to limit those reads and writes as much as possible, as those are the most expensive operations that a computer can execute. We also were not able to add as much error checking as we would have liked, and although in our testing we rarely are able to break things too much, there are likely some cases in which errors do not get caught. We did not get a chance to delve deep into tracking file stats, the only extra info recorded is the size in bytes. Finally, because b_seek wasn't used for the rest of the file system, we left it for last and didn't have time to implement it to a working state.

Another major problem we had was with b_read and I unfortunately only found about it on the last day. Whenever I would try to copy a jpg file from our file system to linux, it would only copy about half the jpg. It turns out I had written b_read to only handle text files with some code that would not work with other types of files. Luckily I was able to fix it on the night of submission , and I confirmed it working with large image files up to 4 mb. Although I did not get a chance to test on anything bigger than that or other types of files.

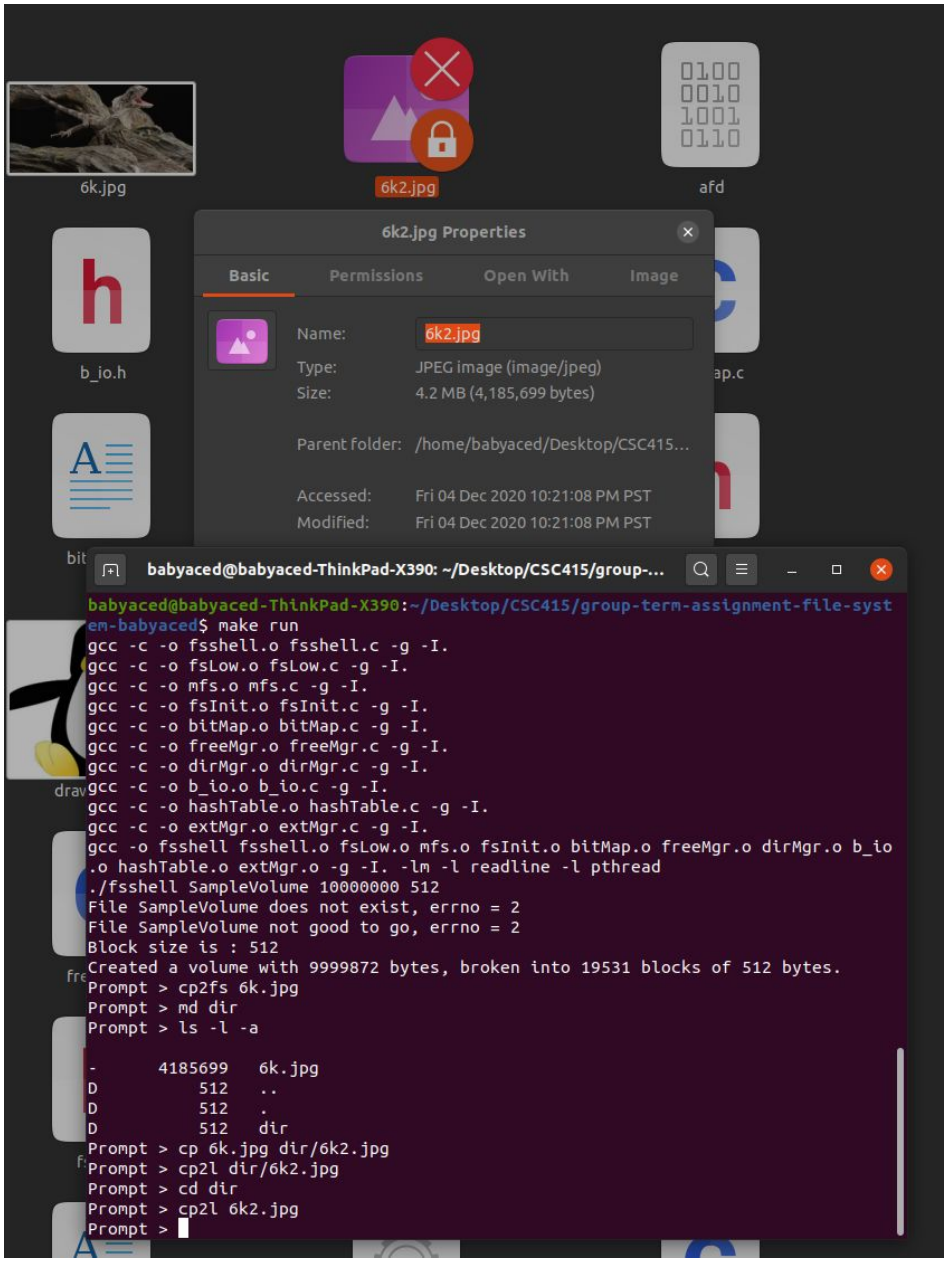
The driver program is very similar to the one provided with a few small changes. The first change is obviously the addition of code to initialize the file system. First startPartitionSystem is called to either create a volume file, or fetch the one that is already created, and get it ready for writes and reads. Then we allocate space for a temporary volume control block and read into it from logical block 0. If the magic number is populated with the hardcoded number that the magic number variable in the volume control block structure is always initialized with, then we know the volume already exists and we call our initGlobals() function. The initGlobals function initializes our global volume control block variable and free space bitmap, that are used by nearly all the functions in the file system, by reading from the disk. If the magic number does not match our hardcoded 64 bit number, then we call the formatVolume function that takes in the desired volume size and block size. It uses the volume size and block size to initialize variables in the volume control block and allocate enough bits in the free space bitmap. After the shell is exited, the closePartitionSystem function is called, which closes the volume file loaded in startPartitionSystem. Finally we call our freeGlobals function that frees the memory associated with the global variables initialized in initGlobals or formatPartitionSystem.

One change we unfortunately had to make with the cmd_ls function, when calling it on the current directory, is setting the path returned by our unorthodox fs_getcwd() as well as the cwd variable to empty strings after calling fs_opendir on the path returned by it. Otherwise we would get a path name with remnants of the previous call to ls, which would cause the fs_opendir function to be unable to find the correct directory.

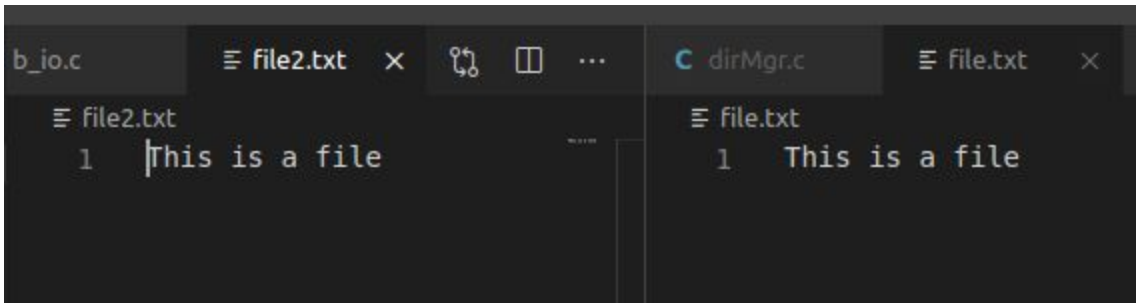
When it came to filing allocation methods, we ended up switching from contiguous to an extended file system design. After we designed the read method in b_io.c, I realized we needed a way to append blocks to a file efficiently. Simply allocating a blob of so many blocks, would risk internal fragmentation and files being unable to grow. Moving files around for more space and compacting files didn't seem efficient, so we decided to restart. We then moved on to adding a new block to a file each time we need one. This helped reduce the fragmentation but became inefficient for large files. Instead, we double the size of each new extent, which helps to reduce the amount of metadata needed to be stored on disk. To incorporate extents into our file system, I made a function that translates a logical address into an LBA index. This is the only function needed to append to a file since it either returns the appropriate block from a file's current extent or adds a new extent to fulfill the requested logical address.
For keeping track of the extents, we store the LBA address and count of each extent in a file's directory entry. We used four primary extent and 25 secondary extents. The reason for these sizes is to keep directory entries stored on disk in whole blocks to reduce fragmentation.

Command Screenshots Starting on Next Page:

Cp2l and cp in one screenshot:



```
babyaced@babyaced-ThinkPad-X390:~/Desktop/CSC415/group-term-assignment-file-system-babyaced$ 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.
Prompt > ls

Prompt > md dir1
Prompt > ls

dir1
Prompt > pwd
/
Prompt > cd dir1
Prompt > pwd
/dir1
Prompt > ls

Prompt > cp2fs file.txt
Prompt > ls -l -a

D          512   ..
D          512   .
-           14   file.txt
Prompt > mv file.txt file2.txt
Prompt > ls -l -a

D          512   ..
D          512   .
-           14   file2.txt
Prompt > mv file2.txt ..
Prompt > cd ..
Prompt > ls

file2.txt
dir1
Prompt > rm dir1
Prompt > ls

file2.txt
Prompt > cp2l file2.txt
Prompt > exit
```



ls Lists the file in a directory

```
gcc -o fsshell fsshell.o fsLow.o mfs.o fsInit.o bitMap.o freeMgr.o dirMgr.o b_io.o hashTable.o extMgr.o -g -I. -lm -l readline -l pthread
./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.
Prompt > ls

Prompt > md lsTestDir
Prompt > cp2fs test.txt
Prompt > ls

lsTestDir
test.txt
Prompt > ls -l

D          512   lsTestDir
-           26   test.txt
Prompt > ls -l -a

D          512   ..
D          512   .
D          512   lsTestDir
-           26   test.txt
Prompt > ls lsTestDir

Prompt >
```

mv Moves a file - source dest (works even when moving a file from a directory to another)

```
student@student-VirtualBox:~/Desktop/CSC415/fs/group-term-assignment-file-system-babyaced$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Prompt > md test1
Prompt > md test2
Prompt > cp2fs test.txt
Prompt > ls

test1
test2
test.txt
Prompt > mv test.txt test1
Prompt > ls

test1
test2
Prompt > cd test1
Prompt > ls

test.txt
Prompt > mv test.txt ..
Prompt > ls

Prompt > pwd
/test1
Prompt > cd ..
Prompt > ls

test1
test2
test.txt
Prompt >
```

md Make a new directory

```
student@student-VirtualBox:~/Desktop/CSC415/fs/group-term-assignment-file-system-babyaced$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Prompt > ls -l -a

D          512   ..
D          512   .
Prompt > md mdTestDir
Prompt > ls

mdTestDir
Prompt > ls -l -a

D          512   ..
D          512   .
D          512   mdTestDir
Prompt >
```

rm Removes a file or directory

```
student@student-VirtualBox:~/Desktop/CSC415/fs/group-term-assignment-file-system-babyaced$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Prompt > ls

test1
test.txt
Prompt > rm test.txt

fs_delete
de delete SUCCESS
Prompt > ls

test1
Prompt > cd test1
Prompt > ls

testDir
test.txt
Prompt > rm testDir
Prompt > rm test.txt

fs_delete
de delete SUCCESS
Prompt > ls

Prompt > pwd
/test1
Prompt > cd ..
Prompt > ls

test1
Prompt > rm test1
Prompt > ls

Prompt > pwd
/
Prompt >
```

cp2fs Copies a file from the Linux file system to the test file system

```
student@student-VirtualBox:~/Desktop/CSC415/fs/group-term-assignment-file-system-babyaced$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Prompt > ls

mdTestDir
Prompt > cp2fs test.txt
Prompt > ls

mdTestDir
test.txt
Prompt > md cp2fsTestDir
Prompt > ls

mdTestDir
cp2fsTestDir
test.txt
Prompt > cd cp2fsTestDir
Prompt > ls

Prompt > cp2fs test.txt
Prompt > ls

test.txt
Prompt > pwd
/cp2fsTestDir
Prompt > cd ..
Prompt > ls

mdTestDir
cp2fsTestDir
test.txt
Prompt >
```

history Prints out the history

```
Prompt > cd test.txt
Could not change path to test.txt
Prompt > cd cp2fsTestDir
Prompt > ls

test.txt
Prompt > rm test.txt

fs_delete
de delete SUCCESS
Prompt > cd ..
Prompt > ls

mdTestDir
cp2fsTestDir
test.txt
Prompt > cd mdTestDir
Prompt > cp2fs test.txt
Prompt > ls -l -a

D          512   ..
D          512   .
D          512   test1
D          512   test2
D          512   test3
-           26   test.txt
Prompt > history
ls
cd test.txt
cd cp2fsTestDir
ls
rm test.txt
cd ..
ls
cd mdTestDir
cp2fs test.txt
ls -l -a
history
Prompt >
```

Commands cd and pwd, also showed where the test.txt is coming from:

```
student@student-VirtualBox:~/Desktop/CSC415/fs/group-term-assignment-file-system-babyaced$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Prompt > ls

mdTestDir
cp2fsTestDir
test.txt
Prompt > cd mdTestDir
Prompt > ls

test1
test2
test3
Prompt > cd test2
Prompt > ls

test4
Prompt > cd test4
Prompt > ls

Prompt > pwd
/mdTestDir/test2/test4
Prompt > cd ..
Prompt > pwd
/mdTestDir/test2
Prompt > cd ..
Prompt > pwd
/mdTestDir
Prompt > cd ..
Prompt > pwd
/
Prompt >
```

```
student@student-VirtualBox:~/Desktop/CSC415/fs/group-term-assignment-file-system-babyaced$ ls
afd     b_io.o    bitMap.o  dirMgr.o  extMgr.o   freeMgr.o  fsInit.o     fsLowDriver.c  fsshell    hashTable.c  Hexdump   mfs.h      SampleVolume
b_io.c  bitMap.c  dirMgr.c  extMgr.c  freeMgr.c  fsInit.c   fsLow.c      fsLow.h        fsshell.c  hashTable.h  Makefile  mfs.o      test.txt
b_io.h  bitMap.h  dirMgr.h  extMgr.h  freeMgr.h  fsInit.h   fsLowDriver  fsLow.o        fsshell.o  hashTable.o  mfs.c     README.md
student@student-VirtualBox:~/Desktop/CSC415/fs/group-term-assignment-file-system-babyaced$
```

help Prints out help

```
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
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 >
```