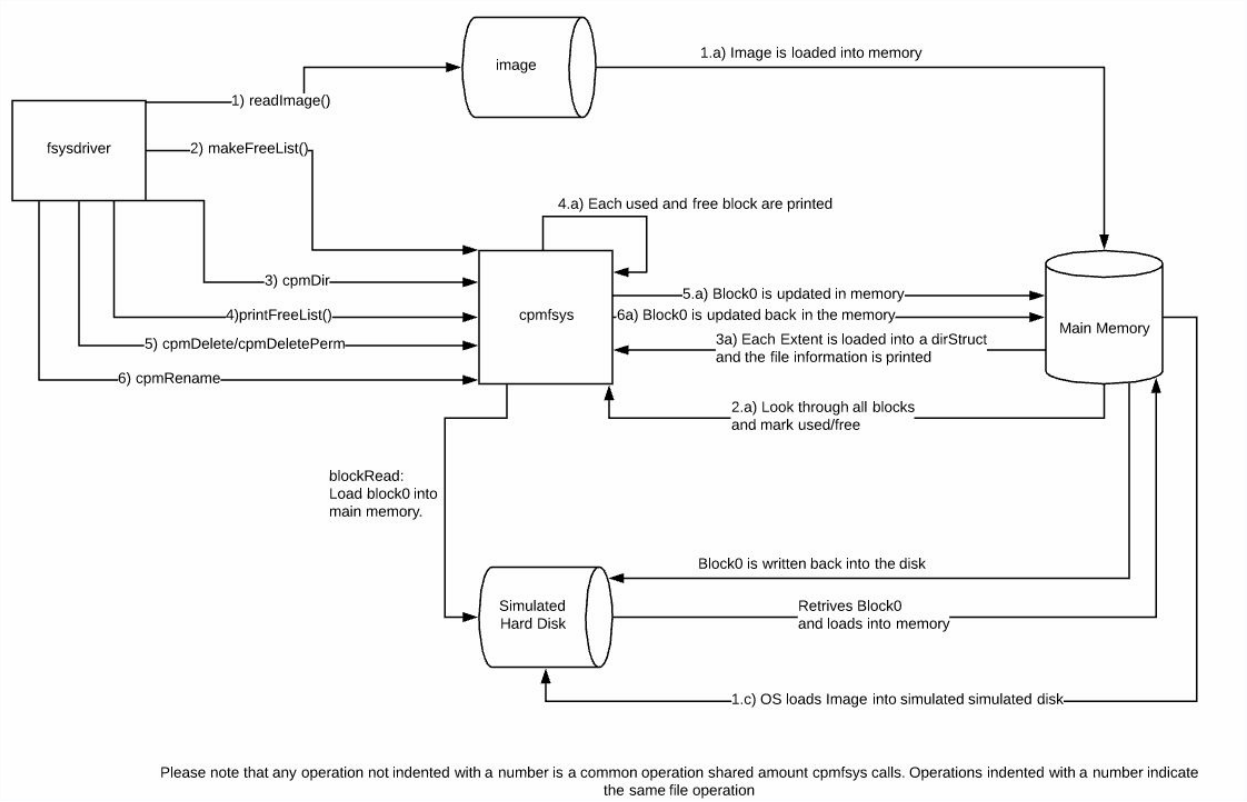
William Christopher White
Comp7500

Design, Implementation, and DFD Diagram:

DTD:



The above Data Flow Diagram gives the outline of the flow of the program. The driver program is fsysdriver. This is simply an example usage of cpmfsys. The example flow first loads an image into a simulated disk. Then it runs makeFreeList to determine which blocks in the disk are used and which are not. Then cpmDir goes through each extent and prints the information of all used extents. Next, printFreeList prints the information gathered about the blocks. Then, cpmDelete is performed to do a soft delete of a file before the newly updated extent information is performed. Finally two renames are performed. After the rewrite completes, the newly updated extents are printed. Examples from Dr. Qin's lectures were used for mkDirStruct, makeFreeList, findExtentByName, and all function prototypes. Below, I will outline each function and its design.

DirStructType *mkDirStruct(int index,uint8_t *e):

This method takes in an extent index and the memory address to block 0. It then goes through the extent and extracts out the status, name, extension type, block information(XL, BC, XH, RC), and loads each of 16 blocks into a DirStruct. The resulting DirStruct is returned. For name and extension, whenever a ‘ ‘ (or space) is found, it is padded with a null terminator. A space indicates the end of file or extension name.

void writeDirStruct(DirStructType *d, uint8_t index, uint8_t *e)

This method can be seen as the inverse of the previous function. It takes in the directory extent to be written, destination extent index, and memory address for block0. It copies each piece of the DirStruct into the memory address pointing to block0. Once this is completed, the block is written back into the simulated disk. After the disk is written, the free list is updated by calling makeFreeList.

void makeFreeList()

A global variable called free_list is used to store each block's usage. At the beginning of each call, this list is set to all free. Then the super block is set to used, as this always contains the extent information. This method loads in block0 into main memory. It then loops through each possible extent and loads that extent’s DirStruct into memory. If the directory is marked as used, then each block is checked. If a block is not zero, then it is marked as used.

void printFreeList()

This method simply loops through the free_list global variable to list the current block usage.

bool checkLegalName(char *name)

This method determines if either a filename or it’s extension is valid. It first checks to see if the first character of the filename is alphanumeric and then checks to see if the filename has any spaces. If either of the conditions fail, then this returns false. Otherwise it returns true. Things such as file name length, extension length, and if filename is blank are checked in other functions.

int splitCheckName(char *name, char *fname, char *extName)

This method takes in a file name and pointers to the split up name and extension of the file. The name is then split by delimiter “.” into filename and file extension pointer. The name and extension is padded with null terminators. Then the length of both the name and the extension are validated. If all of this passes, then the validity of the name is checked. If at any point this fails, -1 is returned indicating an error. 0 is returned for success

int findExtentWithName(char *name, uint8_t *block0)

This function takes in the memory address to block0 and a name of a file to locate. It validates that the filename is valid using previously described methods. And then loops through each extent looking for a match. If a match is found, then the index to the matching extent is returned. A -1 is returned if the filename is invalid or not found.

void cpmDir()

This function's main purpose is to list the directory information of each valid extent. It displays the filename, extension name, and size in bytes. In order to calculate file size, the number of full blocks are counted. Once the number of full blocks(NB) are calculated, it is multiplied by block size. From here, the last block's size is determined. Since the last block is only partially filled, we look at RC (Number of full sectors) and BC(Number of bytes in the last sector). RC is multiplied by the size of a sector and BC added to this number. This number is then added to the already calculated block size. The calculation looks like this:

Length = (NB - 1) * BLOCK_SIZE + RC * 128 + BC

Like previous functions, this method reads in the super block and loops through each extent loading that extent into a DirStruct. If the DirStruct is in use, then the number of used blocks are calculated and the above calculation is performed.

int cpmDelete(char *name)

This function attempts to perform a soft delete of a file name. The super block is loaded into memory. Then findExtentWithName is used to determine if the file exists. If the file exists, then the index of the extent the file is stored in is loaded. Otherwise, -1 is returned indicating either a bad name or a file not found. If the file is found, then the extent index is used to load the DirStruct in. From here, the status of the extent is set to not used, but no data is actually over written. In this way, the filesystem sees the data as deleted, but the data still physically exists on the disk. This means that the data can still be recovered. The new DirStruct information is written into the memory address pointing to block0. Then block0 is written back into the disk. Finally, the free_list is updated with the new block information.

Please note that by design, a soft delete is supposed to leave the data behind. This reduces wear on the disk and lets the data be recovered. The only way that the file system performs an overwrite of this data is when a write operation is received and the filesystem sees that the extent is marked as unused. At this point, the filesystem will overwrite the extent with the new data.

int cpmDeletePerm(char *name)

This function does the exact same thing as cpmDelete, except instead of a soft delete, it performs a hard delete. Not only is the status set to unused, but each element of the Struct is overwritten by a zero. This makes the data physically deleted meaning data recovery becomes much harder. The caveat to this method is that there are extra write operations required, causing potential performance loss. Once the DirStruct has been zero wiped, the memory address to block0 is updated, and then written back to the disk. The free_list is then updated.

int cpmRename(char *oldName, char * newName)

This function is similar to the delete operation. The difference is once a filename is found, the DirStruct filename is overwritten with the new name. Both the new and old names are validated for correctness. If the old name is not found, then the operation returns a -1. Once the DirStruct has its new filename, the memory address pointing to the superblock is updated, and the disk is updated with the new name. The operation for searching the extent and loading the extent into the DirStruct is the same as the delete operation.

<u>Issues with Implementation.</u>

The main problem with my implementation is with how I update the memory address pointing to block0. In the writeDirStruct method, I go through each element of the struct and overwrite it even if it does not need to. While this does not go against the requirements, it does go against what I wanted to do with my delete functions and rename function. One of the goals of my implementation was to minimize writes to only what was needed. The goal was two fold:

1. Minimize the write ware of the disk
2. Minimize the performance loss by doing too many costly writes to the disk.

As mentioned above, I separated the deletes into a soft delete and a hard delete. Whenever a soft delete occurs, the only part of the extent that should be written into main memory is the status. This leaves the rest of the data alone. At this point only the status should be updated in the disk. What actually ends up happening is that each element of the extent gets rewritten into memory. And as a result each element of the extent gets written onto the disk. The issue with this is listed above. To fix this, two steps should be taken:

1. Only the status should be updated in main memory
2. From here only the difference in old block0 and new block0 should be written into this disk
    a. In this example only the status should have been written into disk

The above example was only partially implemented due to complexity. In the future, I plan to update this project with a mechanism that meets the above requirements. It is important to mention that while minimizing writes to memory is a desirable feature, the main goal was to minimize writes to disk, as disk writes are considerably more costly than main memory writes. It is also important to notice that the rename function suffers from this same drawback. If a file is renamed and only renamed, then the only write to disk should be the new file name. Nothing else from the structure should be written to disk. While the above design has so far focused on minimizing disk writes, the other benefit to a soft delete is that files are recoverable. If a file is soft deleted, it is still physically on the disk. This allows the file to be recovered. This also introduces a security issue where if a bad actor gets access to the disk either through direct OS access or through a disk dump, they can recover supposedly deleted files.

This security issue can be fixed by doing a hard delete. In my code, a hard delete was achieved by zero wiping the entire disk. In this case, the above issue of doing too many writes is mute. While doing zero wipe deletes does perform ware on the disk, this is by design. Realistically, a user should only perform zero wipe deletes (Permenant deletes) on files they know are confidential. All non confidential files should be handled by a soft delete(Assuming the soft delete is implemented correctly) to reduce wear on the disk. There are two additional issues needed to be addressed with the permanent delete. First off, a user may not understand the power of the permanent delete. To mitigate this, the OS should require root privlegdes to perform such a delete. The second issue is that a zero wipe delete may not be enough to keep a bad actor from recovering the data. There are forensic tools out there that allow for rebuilding of a drive that has been fully zero-wiped. To get around this, a zero wipe may be performed x amount of times, or each char used to overwrite data could be a randomly generated alphanumeric character. Randomly generated alphanumeric chars can only be run with multiple passes.

Sample Usage

Below a session of the tool in action. This is based on the sample output that was released in the project description. The tool can be compiled with the included makefile. And can be executed by running ./cpmRun

```
DIRECTORY LISTING
mytestf1.txt 15874
holefile.txt 1152
shortf.ps 1032
mytestf. 1026
FREE BLOCK LIST: (* means in-use
```

```
00: * . * . * . * . * . * . * . * .
10: * . . . . . . . . . . . . . . .
20: * . . . * . . . . . . . . . . .
30: * * . . . . . . . . . . . . . .
40: . . . . . . . . . . . . . . . .
50: . . . . . . . . . . . . . . . .
60: . . . . . . . . . . . . . . . .
70: . . . . . . . . . . . . . . . .
80: . . . . . . . . . . . . . . . .
90: . . . . . . . . . . . . . . . .
a0: * . . . * . . . . . . . . . . .
b0: . . . . . . . . . . . . . . . .
c0: . . . . . . . . . . . . . . . .
d0: . . . . . . . . . . . . . . . .
e0: . . . . . . . . . . . . . . . .
f0: . * . * . * . * . * . * . * . *
DIRECTORY LISTING
mytestf1.txt 15874
holefile.txt 1152
mytestf. 1026
cpmRename return code = 0,
DIRECTORY LISTING
mytest2.tx 15874
holefile.txt 1152
mytestv2.x 1026
FREE BLOCK LIST: (* means in-use
00: * . * . * . * . * . * . * . * .
10: * . . . . . . . . . . . . . . .
20: * . . . * . . . . . . . . . . .
30: . . . . . . . . . . . . . . . .
40: . . . . . . . . . . . . . . . .
50: . . . . . . . . . . . . . . . .
60: . . . . . . . . . . . . . . . .
70: . . . . . . . . . . . . . . . .
80: . . . . . . . . . . . . . . . .
90: . . . . . . . . . . . . . . . .
a0: * . . . * . . . . . . . . . . .
b0: . . . . . . . . . . . . . . . .
c0: . . . . . . . . . . . . . . . .
d0: . . . . . . . . . . . . . . . .
e0: . . . . . . . . . . . . . . . .
f0: . * . * . * . * . * . * . * . *
```