A Rudimentary File System in User Space

2d Lt Kevin M. Trigg

ENG, AFIT

CSCE 489: Operating Systems

Lt Col Mark Duncan

September 3, 2024

Project Goals

Project three focused on open ended exploration of a desired topic covered in CSCE 489. To this end, I pursued the development of a file system that could take the place of the system calls made in project 1, Simple Linux Shell. The file system needed to have an analog to disk memory, a way to make new files, update (append) to those files, list the contents of the file, and list the files in the disk. The scope of the project was limited to writing less than a dozen .txt documents in a flat file structure to pass the standards of project 1's needs.

Model development

To begin making a file system, a definition is required for what it needs to perform. "A filesystem is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk" (Wirzenius et al., *Linux System Administrators Guide*). With this and the required operations from project 1 a system began to take shape. Original planning called for the system to be developed using a file system in User Space (FUSE) through the libFUSE repository (https://github.com/libfuse/libfuse). It became apparent that this method would become a significant source of scope creep unless headed off quickly. The FUSE research did however direct me to the idea of simulating the disk space as an array of data in user space to act as the disk management.

The size of the file system needed to be substantial enough for diverse testing and a practical enough example. To define the size three key aspects needed to be decided. The number of writeable data blocks and the of those blocks, and the minimum writeable size within those blocks. Since the system would be designed with text documents in mind, the minimum writeable size would be best set to the size of a single character from that file. Character data types are 1 byte each so the system could be designed for byte level writing. Since bytes are the lowest level that the files will be written in, making the blocks each 256 bytes long makes referencing the position within the block easier while holding a significant amount of characters per block. Finally the number of blocks would be defined by the final size of the desired system. For this reason, 256 total blocks was decided as the total number of blocks as it would allow for easy reference to specific block locations with a single byte of information.

Next the structure of an individual file would need to be defined. Each file needed to have metadata about the file and the data it stores. A name, filetype, and file size would serve as the attributes of the file. The file attributes would be a key component of the FCB, with the files name being the largest space in it. Each file would need to have a predefined maximum to conserve space, 32 bytes per file for the name allows for more than enough expression within the name for this scale and doesn't occupy nearly an entire block of data as a typical file systems 255 limit would. The type of a file only needs a single byte which grants more than enough options in the case of future expansion for different handlings of other file types.

The data storage would need to be broken down based on the size of the file system and desired number of files and accessing method of the file system. With blocks set at 256 bytes and 256 total blocks, limiting file size would be helpful for a defined file control block (FCB) size. To allow each file a decent amount of space, while still allowing for a dozen or so files in the system, each file may have up to eight blocks of data each. To limit wasted space across the disk, indexed allocation would be implemented for the data storage of whole blocks. Each file would have an array of disk addresses that correspond to the data blocks that they occupy, and time a new block needs to be assigned, the next block in the files would be assigned to the next free block from the disk. Inside the blocks, sequential access allows for efficient use of the space while limiting the amount of metadata of the file. To make efficient use of these blocks, the file needs to know where to write when a block is only half full from previous writes. For this, an index within a block is necessary. Since the block size is 256, a byte of information is able to store the next available byte within the last block.

Using the number of occupied blocks and the block indexing, the size of the file can be attributed. In this case, the size of the file can be stored with two bytes as the maximum size of a file is 2kB. Overall, each file needs less than 64 bytes,43 bytes specifically, to store it's own data in the FCB. Withholding 64 bytes per file is useful rather than the exact size of a files attributes because it allows for easy expansion of the system while keeping the size a power of two which eases the math involved in accessing data.

The last defined storage piece of the file system would be the method to quickly determine if a block is already occupied or not. For this I implemented a bitmap which corresponds to each block of data. Thanks to the 256 block number sizing, only 32 bytes of data need to be reserved for this map, the single place where sub byte addressing is used. This with the number of files allowed mean that the first four blocks can be reserved for the FCB and bitmap while still leaving plenty of space for the data.

Implementation

To construct the file system, I wrote the program in C++ to allow for object oriented design of the disk space. I created a Memory.cpp file which created an array of 65536 chars, 64kB. This would represent the data on a byte level. At the head of the file I created constant variables for the potentially changeable sizes of the file as seen in the following block of code:

```
const int numberOfBlocks=256;   //how many blocks are in the memory
const int fileNameLength =32;   //how long a files name can be
const int maxFileCapacity=16;   //max number of files in the memory
const int bitMapOffset=32;      //number of byte offset to make space in mem for the bit map of the occupied blocks
const int fileHeaderLength=64; //size of each file header at start of mem behind the bitmap
const int blockSize=256;        //size of the blocks in bytes
const int blocksInAFile=8;      //max number of blocks occupied by a file
const int blockIDLength=1;      //related to number of blocks, which specific block is indexed
roundup(log(blockSize)/8)
const int fileTypeSize=1;       //size of the definition of the file type in the header
const int fileSizeLength=2;     //# of bytes to represent file size in bytes 2^11 =2048 extra bit used for spacing ease
char *mem;                      //array of chars used as memory
```

This allowed me to make the code more modular for future implementations that might expand upon some aspect of the files or general size of the disk space. In addition, it allowed me to use comprehensible words to represent what point in the disk I intended to reference. For instance, the following line of code would be incredibly difficult to debug for improper references if I had to use the offset values directly rather than the variable for each piece of the file I needed to move over. In this snippet I need to get the index of the next writeable location within the files last block.

```
blockOffset= mem[bitMapOffset + (i*fileHeaderLength) + fileNameLength + fileTypeSize +
blocksInAFile*blockIDLength+fileSizeLength];
//load offset from  the file in question past the map,correct file, filename,type specifier,and data blocks to get
nextByte
```

The constructor and destructor were simple as the only character array needs to be set up and deallocated after operation. Within the constructor, the bitmap needs to be set properly otherwise the data could overwrite the FCB. The first 4 bits in the bitmap were thus instantiated to be 1 regardless of content they held, preventing them from being overwritten in intended data manipulations.

When creating new files I used the following basic pseudocode, this allowed me to outline my method and refer back to something when designing the code. The first character null check is critical since my traversal across all files checks if the first byte in the name is null, if it it then it must be an empty slot. The most significant issue I ran into when making this method was my original implementation would write the file to all empty file slots rather than finding the first then ending operations.

```
Create file( filename){
        Input sanitation checks
                Is the name a proper length?
                Is the first character value null?
        Storage location checks
                Is there an open file location in FCB?
                Was the file already created?
        Write the name and file type in first open slot in FCB //other blocks start null anyways
}
```

After creating the files, there needed to be a way for their data to be written. To this end, I started forming the write() method from the following pseudocode. As the broad strokes, it served me well. When implementing the pseudocode, I quickly realized that encapsulating finding the location to write and finding new blocks for that location were best left in their own methods. The code to find the write location quickly became much bulkier than the code to actually write the data. The indexing, and finding the specific block needed were significantly more delicate than just writing to the data blocks.

An extremely significant issue I ran into at this point was the size of addressing the location was different than I had started to implement. In the beginning I made a foolish error where I set on using eight bytes for each block location in the FCB. This meant that the space each file took in the FCB would be significantly grown, sending the write location to the write() method would require returning an array for the location and the offset within that block tacked on. The source for the error seemed to be that I used eight bytes rather than the eight bits needed to store this location. It should have been apparent to me since the blocks are the same size as the number of blocks and I had properly understood that the block offset was only one byte. Upon my realization, while frustrating in the moment, I was able to simplify the method to locate the write location as I only needed to return the block location and the offset within that block, saving me many for loops to parse byte level addressing in the disk space.

```
write( filename, data){
    Get write location
        What block to write to?
            Get the last non null block used from the files FCB
        What offset within that block?
            Get the value from the FCB
        Error check on if those values are possible
            Last block used and need a new block?
            Sanity check, impossible location?
        What if a new block is needed?
            Go through bitmap for first open block
            Update occupied blocks list in both bitmap and file with new block
    Write data to block
        Write byte by byte
        Update block offset to maintain correct position
            If the offset exceeds the size of a block loop to 0
}
```

The list() method was a great reprieve after my troubles with the write() method. For this, iterating through each block of the file and printing the result was simple. Since the expectation is for text documents, the output even formats itself. The check for null location is critical or every empty block would print out the bitmap and the contents of several of the files in the FCB.

```
list(filename){
    Find the correct file in the FCB
        Loop through each block in that file
            If the block location is null, break
            Print out the entire block
}
```

The simplest method of them all was dir(), this method only needed to print the names of the files stored in the disk space. Since I implemented a flat file structure, there weren't any directories to proceed through. Originally, directories were planned to be implemented in the file system however, due to scope containment and time constraints they were not included.

```
dir(){
    loop through each file in the FCB
        loop through the name of each file
            print the contents of the name
        space between files
}
```

Future development

With the scope set at the beginning of the project several avenues of expansion are readily available. Directories are a significant expansion point as they could be implemented fairly quickly with adjustments to the FCB with a set space for directories and using some of the empty space within each file in the FCB to assign ownership to a specific directory. The most significant expansion would be the implementation of a proper FUSE system that could be mounted to a Linux kernel. I decided against using this for the project due to the amount of additional research it would require to implement and my early research confusing me more than educating myself.

Further ideas would be expanding the possible methods to include overwriting data, deleting files, reading and storing different file types differently to accommodate their data pattern, and scaling the file system to the size of the container rather than having it set at 256 blocks by 256 characters in each block.

Summary

This project focused on developing a file system in user space, aiming for key functionality of creating files, writing to those files, listing them out to the user, and printing the directory of the file system. The project went well successfully representing a file systems job in the operating system. The planning and research stage of the project was reasonable well performed with several issues throughout the course of implementation due to human error and evaluating ideas for implementation in the research of the model. The FUSE model was put aside along with several features to contain the scope of the project which allowed the file system to come to fruition. Room for future expansion and development are open with several ideas being ready to implement next.

Works Cited

Wirzenius, L., Oja, J., Stafford, S., & Weeks, A. (2001, December 3). *Filesystems* . Linux System Administrators Guide. https://tldp.org/LDP/sag/html/filesystems.html

Libfuse. (n.d.). *Libfuse/libfuse: The reference implementation of the linux fuse (filesystem in userspace) interface*. GitHub. https://github.com/libfuse/libfuse