Joseph Jaeschke (jjj93), Adeeb Kebir (ask171), Crystal Calandra (crystaca)
Tested on the cray1 iLab machine
Asst3: A Simple File System
CS 416

Introduction

This project entailed the creation of a file-system in user space. To accomplish this, a preexisting framework was used called FUSE. Using FUSE, it was possible to create a file-system by filling in a few functions. This will document the implementations and decisions made during the project.

The first decision that needed to be made was the layout of the file-system. This started with the creation of a structure similar to a Unix inode. To accomplish this, a struct was defined to hold information about a file. This struct holds the inode number, the permissions of the file, how many links reference the file, the size of the file, the access, modify, and change times of the file, the disk blocks that make up that file and the name of the file. All of this information was sufficient to identify a file and to retrieve the information of that file. In addition, more global information of the file-system was needed. This metadata is stored in a superblock which contains the amount of files currently in the system and a char list to keep track of used and unused inodes. However this is not all the metadata of the file-system as the used and unused disk blocks for data needed to be tracked. This was a lot of data however and required several blocks to store it for and, for this reason, it was not stored within the superblock. This will be discussed in a later section. Since this information was not stored in the superblock, the superblock remained quite small, well within the limit of a single disk block. Similarly, the inodes were kept under the size of a disk block which allowed for one inode to be stored in one disk block. This way of storing the superblock and inodes were a bit wasteful with space, but it made their access much easier and greatly simplified the file-system.

Inode indirection

While only direct-mapped files were required, this file-system supports single and double indirection as the improvement. Every inode has 32 direct mapped disk blocks that it can use for the file's data. In addition there are 64 indirect blocks that an inode can have. Each of these indirect blocks can map to 128 disk blocks for a total of 8192 disk blocks. Finally each inode has a double indirect block. This double indirect block maps to an indirect block which maps to 128 indirect blocks which map to 128 disk blocks. This totals 16384 disk blocks. The total number of disks blocks one inode can map is 24608 blocks which is a total of just over 12MB for a single file. Since no maximum file size was defined, this is the limit I went with.

File-system metrics

Now the basic structure of the file-system is laid out, a more detailed description of the system can be given. The only requirements for our file-system was that it needed to support at least 128 files and take up no more space than 16MB. With this in mind I decided to plan a file-system that supports 128 files. For this reason the maximum amount of files supported at any time is 128. Since the maximum file size for a single file is about 12MB, there had to be a limit on the amount of indirect blocks (single and double) available to prevent overflowing the 16MB. To introduce this limit, there are only enough indirect and direct blocks available for one inode to fill up. This means there is only 192 indirect blocks and 1 double indirect block. There are 192 indirect since an inode has 64 and the

double indirect has 128. Hence the limit is the sum of those numbers. Since there was not an infinite amount of indirect blocks metadata was needed. Similar to the inode char list in the superblock, a char list was used to whether an indirect block was in use or not. This additional metadata was stored alongside the superblock and inodes in a separate disk block.

Now that this limit of indirect blocks is defined, the maximum size of the file-system can be deduced. Assuming all 128 files exist and all 192 indirect blocks are used, there is a total of 28672 disk blocks in use. This translates to exactly 14MB. Note that there are an additional 2MB that was not in the calculation for the maximum file size. This is because there is no hard limit on direct mapped blocks. Every file created is guaranteed 32 of those. This bring the 12MB total to 14MB Not included in this calculation is the metadata used which would bring the total to about 14.2MB. The assignment description was not very specific on the total size of the file-system when it is filled, only that it be under 16MB. The total for this file-system is under that.

Now that it was clear how many addressable disk blocks could be used for data, there needed to be a way to track them. To do this, the same strategy was used to track the inode and the indirect blocks. However this became quite large. Since one byte was used to represent one disk block, that needed 28672 bytes. This is a total of 56 disk blocks to store whether a data disk block is used or not. Ultimately this is what the file-system uses, but another option was possible. Rather than keeping byte lists, the state of a disk block could be stored in a single bit. This would reduce the memory needed by a factor of 8. However this requires a more sophisticated implementation and introduces the need for bitwise operations and bit shifting. Seeing this a big potential for bugs, the char option was used, especially considering there was some room left from the 16MB limit.

In addition, the permissions of the file-system were not specified in the assignment description. Since the implementation of chmod was not necessary, every file created had all permissions. This means that the owner, the group of the owner, and others could read, write, and execute the files. Since there was no way to alter the permissions of a created file, it was best to allow anyone to do anything to a file. While this is not good practice, it allows for simple usage of the system and nothing specified that this should not be done in the assignment description.

FUSE

In writing this file-system, many aspects of FUSE were not realized until further in. For example, when initially writing this file-system, I was not sure if the sfs program stays running as long as the file-system was mounted. For this reason, every time a change is made to metadata, the block that metadata belongs to is rewritten. This is opposed to keeping a global structure of inodes and metadata which could be accessed faster. In most cases this is not too big of a deal as most of the data did not have to be written until the end of the function. As a result of this the sfs_destory function was left empty since the state of the file-system was already sotred. Although due to the many disk block writes this is not the most efficient implementation. However, in the event of a sudden power loss, the state of the file-system will be preserved contingent on the power loss not occurring during the middle of an operation.

Testing

One of the problems with testing the file-system was the repeated operations needed between the tests. Every time this file-system was to be tested, the directory had to be unmounted, the log had to be cleared, the flat-file the file-system was in had to be cleared, and the program needed to be run.

Due to this repetition, the making of a bash script was very helpful. Rather than doing four things, one script could be run to automate this. In addition to the automation of setting the file-system up, scripts could be used to test it. Since many tests were run repeatedly until the functionality was correct, these sequences of commands could be put into a script which could be run much faster and much easier than if they were inputted manually.

Most of the testes run with the file-system were simple bash commands. This really spanned the creation of a file, the deletion of a file, writing to a file, and reading from a file. Since directory support was not required, moving a file to another directory was not implemented. By looking at the assignment description, it was possible to determine the sequence of commands that made up each of these functions in bash. This made getting a command to work fairly straightforward. Test cases were then made to make sure the command worked as expected. Finally tests were made to test the functionality of different commands sequentially. When in doubt, it was possible to read from the flat-file that the file-system was stored in. This made it possible to see if an inode was created, or if a write happened for example and to determine that a read or directory list did not change the state of the file-system.