CS170: Spring 2016

Project 3

Project 3: Upgrading the Minix File System

Summary

Due June 3, 2016, 11:59:59PM

Implement and extend the Minix file system to add support for immediate files

Then add a system call to return detailed information on files

To be completed in teams of two

This project is all Minix, and requires significant changes to the Minix kernel. So please start as soon as possible. Do not wait. If you have questions about what I expect from this assignment, please speak up early, do not wait until a few days before the deadline. There are also two separate parts to this project. You can parallelize them to some extent, but they are very much related. You will need to work closely with your partner to maximize your productivity for both parts.

WARNING

For this project, you are modifying the file system. If you mess up, your file system can become damaged beyond repair. To safeguard yourself against data loss, I strongly suggest that you regularly back up your file system. This is as easy as making a copy of the Minix image file (the one that Qemu uses, for example). Obviously Qemu must be powered off when performing the copy, or you'll end up corrupting your copy! Also, try to proceed one step at a time, and make sure that everything works before moving on. Some debugging output at each stage is helpful to monitor your ongoing progress.

Part I: Implement Support for Immediate Files in MINIX

The first part of your project is to implement support for immediate files for Minix. Immediate files are basically storage for small files where the data is stored directly in the inode, without the need to traverse pointers to external blocks. Let's first review how file systems work in Minix:

Each regular Minix file is represented by an inode that stores metadata about a file (such as file size or the user id of file owner), as well as a number of disk blocks that store the file's content. To find the disk blocks storing the file content, an inode also stores pointers (in the inode's i_zone array). These pointers can either point directly to a disk block storing data, or to a block that stores a list of additional pointers to data blocks (indirect blocks). But for really small files, say 1 or 2 byte files, then a complete disk block still needs to be allocated. In Minix, these blocks are generally called Zones, but it's the same thing. Because each block or zone has a minimum size of 4KB, this clearly wastes a lot of space.

To make storage more efficient for small files, and to reduce internal fragmentation, we can use immediate files. An immediate file is a file whose data is not stored in a data block, but directly inside the inode itself. An inode in Minix is 64 bytes long, and 40 bytes are used to hold pointers to data blocks. For immediate files, you can clearly use those 40 bytes to store the file contents instead of pointers. Your job, should you choose to accept it (and you'd better), is to implement support for this file type in Minix. Since Minix uses the VFS and MFS services to do file system operations, most of your modifications will likely reside in `/usr/src/services/(vfs|mfs)`.

Your support for immediate files should store at least 32 bytes of data. Whenever a new file is created (with a length of 0), it should be created as an immediate file. Whenever the length of the file exceeds 32 bytes, it must be transformed into a regular file. Clearly, this requires that the current contents of the file stored in the inode to be moved to a newly allocated data block on disk. The space you were using now needs to be vacated so that you can store pointers to data blocks instead. Note that to simplify things a bit, we are not requiring you to do the reverse. In other words, if a file gets cut down from some size > 32B, to something smaller than 32B, you do NOT have to change the file into an immediate file. In other words, once a regular file, always a regular file. This also implies that you do not have to change existing files in the current file system. However, if you create a new file (which is an immediate file), and you add bytes to it without pushing it over 32B, then the data should be stored in the inode and no new data blocks should be allocated. Of course your change need to be permanent and stay that way across reboots of the OS.

Implementing this involves several key steps. First, you need to define an additional flag (e.g. for an i_mode member variable of an inode) that marks your file and inode as immediate. Then you need to think about the different file operations that are affected by immediate files, and modify them to support these files. If you think a bit, you should come up with a number of operations, including (but not limited to) creating, deleting, reading and writing a file. Specifically, here are some requirements:

- Whenever you create a new file (or implicitly creating a non-existent file by opening it), this new file should be marked as immediate.
- Whenever you delete or unlink an immediate file, you have to make sure that the code does not try to free data blocks for them, since there are none. You have to make sure that you do NOT follow the block pointers in the inode, since those are not used and generally invalid. For the pointers, check out member i_zone in inode.h
- Similarly, when you read from an immediate file, do not follow the links, just retrieve the data from the inode itself.
- For each write operation, you need to check to determine whether the data after your write will exceed the 32 maximum bytes in your immediate file. If not, then no problem, just add the bytes to the inode storage. If so, then the immediate file must be changed into a regular file. This means you need to allocate blocks to store the data that is currently in the inode. Then you need to move the data over to the new block and make sure you fix the inode so that the pointers are valid (make sure all block pointers are correctly set, and the others are set to NO_ZONE).
- Check out files from the current file system to see how current operations are performed, and where the code needs to be changed. Make sure to check to see which functions are already implemented, and reuse existing functions whenever possible.

Remember to test your functionality incrementally. It will make life much more manageable than if you test everything in one run, when your output is probably affected by multiple hard to track bugs all interacting to corrupt your VFS/MFS. ***Part II: Tracking File Usage***

Implementing immediate files is useful, but not quite enough for a full project. So we're going to go a step further and extend more functionality into the file system. In particular, we're going to add a system call that allows the user to monitor usage of files in your file system. Let's call this system call list resources, or lsr, with the interface:

```
int lsr (char *path);
```

This call takes one argument, a string that holds the name of a file whose resources should be listed. The string could be a local file in the current directory, or a full or partial path to a file, i.e. lsr("/etc/motd") or lsr("../bin/myexec"). When called on a particular file, lsr's job is to print out:

- The list of all process IDs of processes that have this file open
- The list of blocks on disk that store the contents of the file. If the file is an immediate file, or if the file is empty, then print out a message to say that.
- If the file referred to in the path does not exist, then an error message should be returned to the caller.

By now, you should be pretty familiar with implementing system calls. The system call sends a message to the file system (VFS) service. Make sure you have a wrapper function in the POSIX library that can be called by user programs.

The corresponding syscall handler in the VFS service should forward the request to the MFS service which should first resolve the path name to the correct inode, and return an error immediately if the file does not exist. Otherwise, you need to check which processes have this file open. This involves looking at the filp table to see which entries point to your inode. Then look at the per-process file descriptor tables (struct fproc) to see which processes have pointers to the filp entries you're interested in. For each process that has a file descriptor pointing to a filp entry that points to your inode, that process has your file open. Print out the PID for all such processes directly inside the VFS service; you don't have to return it to the caller of lsr.

The other component is to find all the disk blocks that correspond to this file. Start at the i_zone array that stores pointers to blocks. Of course you will need to handle large files that use indirect pointers, and make sure you traverse those pointers correctly. Again, no need to return the data to the caller, just print out the numbers of all blocks in the file, separated by spaces (in sequential order as if you were reading the file from beginning to end). You'll want to save time by reusing functions already inside Minix, e.g. functions to find inodes given a path name, or pack a string into a message and unwrap it inside the VFS or MFS service (trace the open call for an example).

Submission Process

Your submission must be submitted prior to the deadline in order to be graded. Do a man turnin to find more info about the turnin program. To submit:

- Ensure your current working directory is a directory containing:
  - a file named patch that contains your changes to the Minix source to support both immediate files and the lsr system call . The patch must apply cleanly to a fresh source code tree and be run on your dev machine via:

    ```
    diff -ruNp minix_src_clean/ proj3/ > patch
    ```

an optional README file explaining what you've done

Execute the turnin program:

```
turnin proj3@cs170 patch [README]
```

You can execute turnin as many times as required. The most recent submission prior to the deadline will be used for grading. You do not need to inform the TAs if you intend on using your two day extension for this project. Any submissions past the deadline will be assumed to be using your 2-day extension. If you have already used your extension, then your latest version before the original deadline will be graded.