Programming Project #4

Extending a filesystem

Introduction

In this assignment you will learn about the internals of UNIX filesystems and how to create custom interfaces to the OS. Your task is to extend the MINIX filesystem to include a special metadata area for each file. This extra space can be used to store notes about the file that are separate from the normal file contents. You'll also need to design tools to allow users convenient access to these areas.

Your tasks:

- Extend the standard MINIX filesystem implementation to optionally keep track of one extra metadata region for each file that can hold up to 1024 bytes.
- Design an interface (syscall) for user space processes to read and write that metadata, and add the metadata region to files that don't have one.
- Add system calls to MINIX to implement your interface.
- Implement the correct actions in response to the messages in the filesystem server.
- Design and implement library functions to make calling your syscalls easier.
- Write some test programs to verify that the metadata is being stored properly.
- Write command-line utility programs using your library functions so users can create and access metadata.
- Test your implementation by writing some demonstration shell scripts.

Details

Having extra space in a file that is not part of its normal contents can be convenient. Earlier versions of Mac OS filesystems allowed every file to have a resource fork [1] which could be used to store graphical data, executable code, and file metadata. Your version will allow a small amount (1024 bytes) of unstructured data to be stored alongside normal files.

The goal of the assignment is to get you familiar with filesystems, system calls, inodes, and MINIX hacking. As with earlier projects, you should not need to write excessively many lines of code, but it is critical to understand where you need to modify MINIX. Your final code will be in several parts:

- user space test programs
- library functions wrapping your syscalls
- implementation of your syscalls in the filesystem (VFS, MFS)
- other small changes to MINIX headers, servers, etc. as needed

VFS

One complication is that the filesystem in MINIX is split into two parts: the virtual file system server (VFS) and the MINIX file system server (MFS). Read `/usr/src/servers/vfs/README` for an explanation of what VFS does. Basically it accepts filesystem syscall requests and routes them to the appropriate actual filesystem server. There are multiple filesystems; for example MFS deals with disk filesystems and PROCFS deals with the special `/proc` location.

You are adding a syscall that userspace processes can use to talk to a filesystem so you will need to route your new syscall through VFS. The design and implementation of VFS is documented in Balázs Geröfi's Master's Thesis [2].

Your system call must be able to:

- Create a new metadata region if one is not present
- Write data to a file's metadata region
- Read data from a file's metadata region

One way to satisfy these requirements it to add a system call to VFS that is similar to `read()` and `write()`; in other words a “system call with file descriptor argument”. The VFS server can map the file descriptor to a vnode. A vnode is a reference to an inode on an actual FS. Once VFS checks the arguments and finds the inode it passes a message to the actual FS (MFS for disk files). You will need to add a new message type for reading and writing the metadata region. Existing messages from VFS to filesystems are in `/usr/include/minix/vfsif.h`.

Your system call will probably have the same arguments as `read()` and `write()`; a file descriptor, pointer to memory, and count of bytes to read or write. You should be able to reuse message formats and other declarations for read and write.

Allocating an Extra Block

To keep track of the metadata you'll need to allocate an unused data block. You'll also need to keep track of which block is used. You need to put the block number in the inode somewhere.

One subtle point is that you need to maintain backwards compatibility with the existing filesystem, otherwise you will quickly discover your system is unusable! You will need to figure out a way to reuse part of the inode data elements as a block or zone number for the extra region that you are adding. Since there are already files potentially using all the elements, you will need to figure out a protocol to avoid misinterpreting the element that you are reusing.

For example, if you reuse the ctime field in the inode structure, you may decide that -1 indicates there is no extra data, otherwise it is a zone number for the data. After you change MFS, recompile, and reboot, there will be files that have arbitrary ctime values that your code might happily think are valid zone numbers.

Instead of looking for -1 in ctime, you might instead decide to reinterpret the sticky bit as “metadata present”. Adding metadata could then set the sticky bit. Reading metadata would check for the sticky bit; if it was not set then you know there is no metadata (even though ctime has some number in it). This way you still have a slight problem with files in MINIX that already have the sticky bit set, but by default no files in MINIX have the sticky bit set.

Note that ctime may not be the best place to store the extra metadata location. The example is just illustrative. A better location might be the last zone pointer, zone 9 in the inode, which is normally a triple indirect pointer. Few files are large enough to need a triple indirect pointer, so this field can be reused as a direct pointer to your extra metadata.

A report about modifying the MINIX inode structure [5] may be helpful.

Command Line Tools

In order to make your change useful you need tools for users that let them read and write the extra metadata. Here is one idea for how the interface could work:

`metacat FILE`	print metadata of FILE to stdout
`metatag FILE TEXT`	write metadata of FILE, replacing any existing contents

```
metatag README.txt “This file is great”
metacat README.txt >> notes.txt
```

Plan

Here is a suggested plan for systematically making progress.

- Add a new system call to MINIX that prints "hello" when called, handled by any server you want.
- Add a new system call handled by VFS that prints "vfs hello".
- Add a new message from VFS to MFS that duplicates the `read()` functionality but also prints out a debugging message.
- Add a message from VFS to MFS that duplicates `write()` (`metawrite()`).
- Update the functionality of `metawrite()` to allocate a block and store its location somewhere in the inode of the file if there is not already a block allocated (and not do a write to the normal file contents).
- Test your implementation, make sure it works as you expect.
- Add code to actually write the metadata to the block.
- Implement `metaread()` so that it actually reads the metadata.
- Test read and write in combination, debug any errors.
- Extend your testing programs into useful command-line tools.
- Write some shell scripts to test and demonstrate that your solution is working properly.

Another idea is to work to get one int stored as metadata per file, and build up all the syscalls and user space tools to set and retrieve that integer. After backing up your files, transition from reading and writing one int to reading and writing up to 1024 characters and using the int to refer to an allocated zone or block.

Testing

Your testing should be able to:

- Demonstrate compatibility with the existing MINIX filesystem (either by showing existing files not being corrupted for a change in MFS, or by showing your alternate filesystem mounted alonside MFS).
- Demonstrate adding a note “This file is awesome!” to a README.txt file, and later reading back the note.
- Demonstrate that changing the regular file contents does not change the extra metadata.
- Demonstrate that changing the metadata does not change the regular file contents.
- Demonstrate that copying a file with metadata copies the metadata.
- Demonstrate that changing the metadata on the original file does not modify the metadata of the copied file.
- Demonstrate that creating 1000 files, adding metadata to them, then deleting them does not decrease the free space on the filesystem.

Deliverables

Turn in a `.tgz` file for your project directory into eCommons. It should include:

- Build and run instructions in `README.txt`
- Your design document, named `design.pdf`
- Source files (no generated files!)
- Any extra build files (e.g. `Makefile`)
- Any test files

Your project must be submitted in this format or you will be asked to fix your submission and resubmit (using grace days or getting a penalty). If we try to follow the instructions in your `README.txt` file and it doesn't work, you will be asked to resubmit (using grace days or getting a penalty). We will try your submissions in a fresh MINIX 3.1.8 (or MINIX 3.2.1 installation if you specify that in your instructions). Don't forget to include any files needed. Also, do not include files that could be generated automatically from source files (each such file included will be penalized).

For building, we would prefer to type one command to build everything. This might involve copying files from your directory into `/usr/src` and running commands such as `make install` from the `/usr/src/tools/` directory.

References

[1]: http://en.wikipedia.org/wiki/Resource_fork
[2]: http://www.minix3.org/doc/gerofi_thesis.pdf
[3]: http://cise.ufl.edu/class/cop4600sp14/Minix-Syscall_Tutorialv2.pdf
[4]: http://www.cis.syr.edu/~wedu/seed/Labs/Documentation/Minix3/How_to_add_system_call.pdf
[5]: http://www.cis.syr.edu/~wedu/seed/Labs/Documentation/Minix3/Inode.pdf (See section 5)

Coding Style

All code submitted must follow these guidelines. Deviations may be penalized. Unclear or sloppy code will always be penalized. Good coding style makes your code more readable to others and is critical for keeping complex software maintainable.

Comments

- Every file must contain a comment on top that indicates whether you created it, or it was part of MINIX and you modified it. If you created the file, put CREATED in the command and a date. If you modified an existing MINIX file, put CHANGED in the comment with a date.
- Every function must have a comment immediately before it that describes what the function does. It should describe the parameters, return value, intended action, and any assumptions the function makes.
- Code should be well commented. Trickier parts of code need more comments, simple parts need fewer comments. You should end up with about as many lines of comments as you have lines of code.
- If you changed an existing MINIX file, put comments that indicate where the change starts and ends. Put “CHANGE START” and “CHANGE END” in these comments.

Indentation

- Only use spaces, no tabs.
- Indent a fixed number of spaces at every level (e.g. 2 or 4).
- Wrap long lines manually in a visually reasonable way.

Spacing

- Every significant comment should be preceded by a blank line. Do not put a blank line after comments before the code that is being described.
- Put a blank line between chunks of code that do different things.
- Put a blank line between variable declarations and other code.
- Put a blank line after every right brace unless the next line is also a right brace.

Braces

- For function declarations, put the opening brace in column 0.
- For other structures, put the left brace on the same line as the keyword (if, while, etc.). Put the right brace to align with the opening keyword (if, while, etc.)

Identifiers

- Functions and variable names should be all lowercase, with underscores between words.
- Names should be descriptive without being too long (this is an art).
- Single letter names may only be used as trivial loop counters.

Global Variables

- Global variables may not be used unless they are the correct solution to a particular implementation issue.
- Any global variables must be explained in the design document.