Project 4 Design Document
File System Extension

Desmond Vehar - dvehar
Dave Lazell - dlazell
Neil Killeen - nkilleen
Melanie Dickinson - mldickin

May 29, 2014

changes since implementation:

- for test5, instead of using a compiled c program, we created the 1000 files in a shell script
- for fs_metawrite(), we do not write the changed inode to disk ourselves as it is normally handled on file close
- added test results

problems we had with implementation were:

- In our metatag command, we opened the file with “wb” instead of “r+”. This caused the normal data in the file (zone 0) to be deleted.
- Initially we tried to run the commands from our mounted directory, which was running hgfs instead of mfs.
- Trying to write the inode back to disk in fs_metawrite with put_inode caused a kernel panic. This is because doing so releases the inode, and we still need it. We simply removed that call, as it is handled when the file is closed.

Introduction

In this project, we modify the Minix 3.2.1 file system to allow for files to have an extra 1024 byte hidden area for metadata, separate from the normal data area for the file. This metadata region is made accessible to user programs via the library functions metaread and metawrite. These functions interface with the actual Minix File System (MFS) through the Virtual File System (VFS) abstraction. The library functions invoke the VFS via a syscall. The VFS then forwards the request message, as a global variable, to the MFS, which performs the actual handling of the metadata region. The data read is transferred to/from the user program, and a message is populated and sent back to the VFS. The VFS then notifies the user program that the data is ready/sent. In addition to this call chain through the system to access the metadata region, we also require shell commands metacat and metaread. These commands let us access the metadata from the command line and allow us to write test shell scripts to verify proper operation of our implementation.

Our project is split into four main parts:

1. Designing an interface (library functions) for metaread() and metawrite() for use by user programs. For this we use the same interface as read() and write() because it was pre-existing and users are already familiar with this interface.
2. Modifying Minix to route our library functions through the system: from library function to syscall to VFS and via messages to MFS. Upon looking at the pre-existing code, the functionality for read() and write() are almost exactly what we need for metaread() and metawrite(), with the exception of the lowest level in MFS. Our code will therefore duplicate what we can of this path, all the way down until we diverge at fs_metaread() and fs_metawrite(). By making new functions that duplicate this path, we avoid having to change the existing functions, which also avoids having the system crash if we don't change them correctly. See Figure 1 on the last page for a chart of our function call path.
3. Implementing the actual reading and writing of the metadata in MFS (fs_metaread() and fs_metawrite()).
4. Writing commands metacat and metatag to allow access to the metadata region from the shell, and testing the implementation via shell scripts.

Interface / Library Functions

Our interface for metaread() and metawrite() requires the same parameters as read() and write() do: a file descriptor, a buffer, and the number of bytes to read or write. Our functions behave the same as read() and write() except for the following cases: 1) For both functions, the max size is 1024, otherwise an error will be returned. 2) The position is always considered 0, so each call will read/write starting at 0. 3) For metaread(), if there is no metadata, we return 0 bytes read. 4) For metawrite, we zero the 1024 bytes before writing the buffer. We decided to reuse the code for read() and write() because they required minimal changes. The library functions then do a syscall to the VFS, and are our top level modifications to the Minix system. These functions are added to the read.c and write.c in the sys-minix directory, so that building the libraries will build our code in with read() and write(), and we will not have to change any makefiles.

VFS functions

Once the syscall is completed we are inside the VFS. Again, we duplicate the functions do_read_write and read_write as we require most of the same functionality. However, the metadata region is only used by normal files and not pipes or special files, so we return with an error if we have these cases. We also error if the size requested is greater than 1024 bytes. Also, since we decided that all reads and writes to the metadata region will begin at position 0, we remove the references to position in the functions. VFS maps the file descriptor to a vnode, which contains a reference to an actual inode on the filesystem. VFS then creates a request message in req_metareadwrite(), containing the inode, which is forwarded to the MFS. Our req_metareadwrite duplicates the functionality of req_readwrite, with the exception of having 0 for

the position. We add these functions the the read.c and write.c files in the vfs directory, so that compiling the kernel will also include our files without the need to change any makefiles.

MFS functions

The main.c function in MFS dispatches the received message to our functions fs_metaread() and fs_metawrite(). fs_metaread() reads the data from the metadata region if available, otherwise returns 0 bytes read. fs_metawrite() writes data to this region, creating it if the region does not already exist. These functions are added to the read.c file in the mfs directory, so that compiling the kernel will also include our files without the need to change any makefiles.

fs_metaread()

This function receives the inode of the file from the fs_m_in message via VFS. Once we have the inode we do the following:
1) check to see if zone 9 is unused. If it is, we return 0 bytes read (in the fs_m_out message).
2) if zone 9 is in use, then it is our metadata, so we:
3) obtain the first block in the zone
4) copy the first 1024 bytes (or less if requested) to the buffer.
5) return the number of bytes read (in the fs_m_out message).

fs_metawrite()

This function receives the inode of the file from the fs_m_in message via VFS. Once we have the inode we do the following:
1) check to see if zone 9 is unused. If it is we allocate a new zone via alloc_zone
2) obtain the first block of the zone
3) zero out the block
4) write our data from the buffer to the block
5) mark the dirty bits of the block and inode
6) write the block to disk

Additional changes to Minix headers

The system calls are added to the Minix system by inclusion in the list of syscall constants in callnr.h, and prototype definitions in unistd.h. Each of the server source directories contain two files, table.c and proto.h, which contain definitions for the function pointer tables and function prototypes, respectively. Corresponding entries are added to these files for the VFS and MFS servers. Two new requests that VFS may send to MFS are also declared in vfsif.h.

Allocating the extra zone

To keep track of the extra metadata region, we need to allocate a new zone and also keep track of that zone. To keep track of the new zone we allocate, we use the 9th zone in the inode structs izone array. In the Minix filesystem version 2, of the 10 zones in the array, the first 7 are direct pointers to zones, zone 7 is a single indirect pointer, zone 8 is a doubly indirect pointer, and zone

9 is unused. Since zone 9 is unused, it is a perfect choice to safely store the zone used for our metadata.

Commands and Testing

Commands

To facilitate testing, we create the commands metacat and metatag. Metacat will read the metadata of the specified file and output it to stdout, if the metadata exists. Metatag will take a file and a string, and write the string to the metadata of the file, creating the metadata region if necessary. Both commands directly call our library routines metaread() and metawrite() and therefore are restricted to 1024 bytes and starting at position 0.

Testing Plan

The goals for our testing are to:

- Demonstrate compatibility with the existing MINIX filesystem by showing our alternate filesystem mounted alongside the existing filesystem.
- 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.

Scripts we used for testing the above goals:

test1 - This script writes metadata to itself, and then does an “ls”, showing that the file system can handle (display) both files with metadata, and files without metadata.

test2 - This script adds the note “This file is awesome!” to the README.txt file, reads it back, and compares it to the written metadata to see if it is the same.

test3 - This script creates a test file “test.txt”, with the contents “this is a test”, and the metadata “and this is metadata”. It then changes the contents to “test data”. and reads the metadata, making sure it has not changed. It then changed the metadata to “test metadata” and reads the normal contents, making sure it has not changed.

test4 - This script copies “test.txt” to “test2.txt” and verifies that the metadata copied correctly. It then changes the metadata of the original file, reads the metadata of the copied file, and verifies that it did not change.

test5 - This script runs du to display the space used on the disk. It then creates 1000 files, adds metadata to each of them, and then deletes the files. It then runs du again, so any difference will be immediately noticeable.

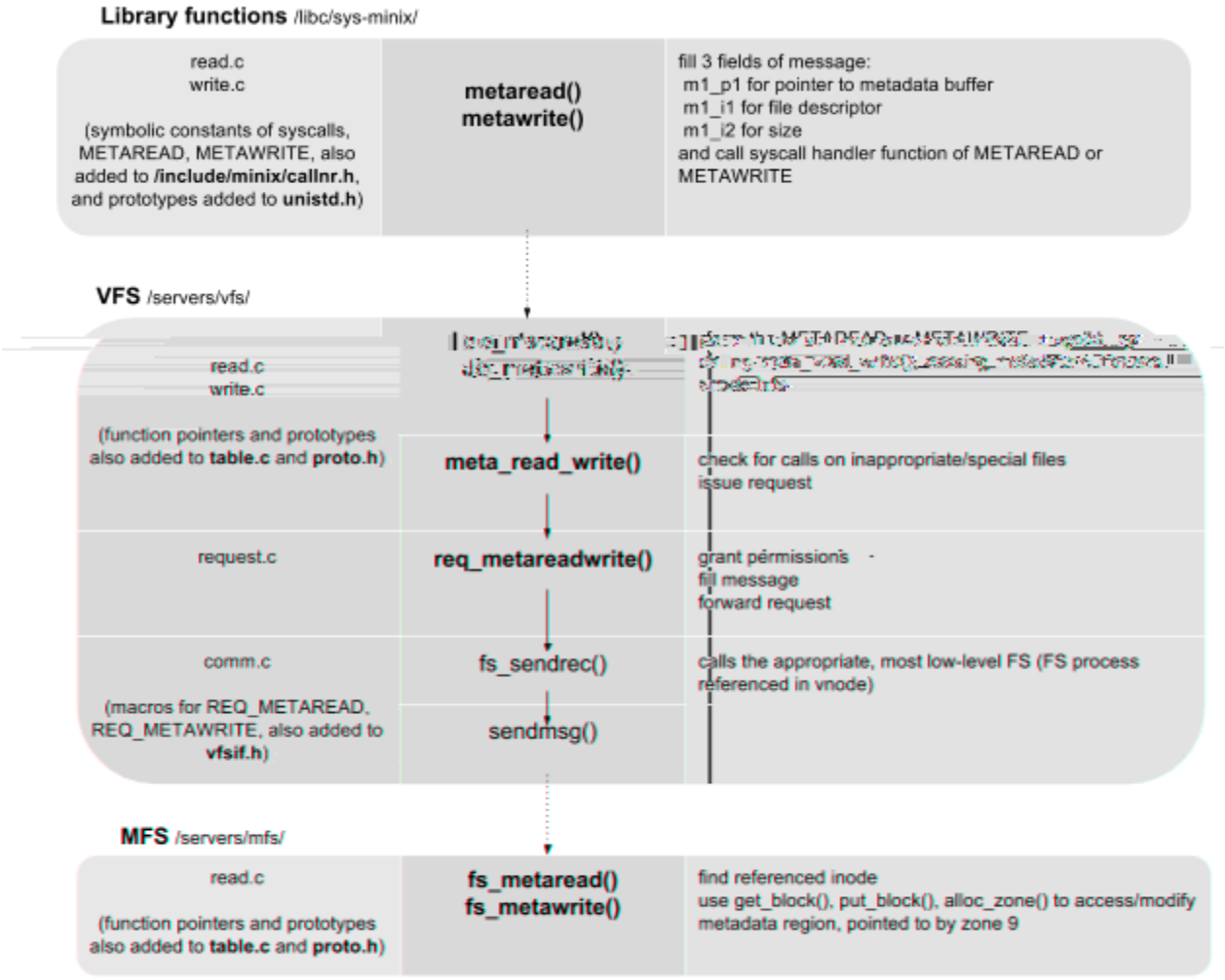
Testing Results

All of our tests passed, with the exception of test4. The ability to copy the metadata from file to file was neither designed or implemented, so therefore this feature was not available, and the test likewise failed.

Figure1:

Diagram showing added functions and changes to Minix, organized by file locations and control flow of the project code: from user invocation of library function, with a particular string buffer of metadata and a file, to VFS system calls, to MFS functions manipulating metadata region of an inode. Once the MFS processes a request, by reading from or writing to the user-specified buffer and the block pointed to by the inode, reply messages are then passed back up through VFS to indicate completion and return to user. The returning message procedures are unchanged for this project.

Solid arrows, A→B, mean “function A calls function B”. Dotted arrows refer to indirect calling, via function dispatch. Bolded functions are new additions to Minix. Unbolded functions are existing/unmodified.


Library functions /libc/sys-minix/
read.c
write.c
(symbolic constants of syscalls, METAREAD, METAWRITE, also added to /include/minix/callnr.h, and prototypes added to unistd.h)
metaread()
metawrite()
fill 3 fields of message:
m1_p1 for pointer to metadata buffer
m1_i1 for file descriptor
m1_i2 for size
and call syscall handler function of METAREAD or METAWRITE
VFS /servers/vfs/
read.c
write.c
(function pointers and prototypes also added to table.c and proto.h)
meta_read_write()
check for calls on inappropriate/special files
issue request
request.c
req_metareadwrite()
grant permissions
fill message
forward request
comm.c
(macros for REQ_METAREAD, REQ_METAWRITE, also added to vfsif.h)
fs_sendrec()
sendmsg()
calls the appropriate, most low-level FS (FS process referenced in vnode)
MFS /servers/mfs/
read.c
(function pointers and prototypes also added to table.c and proto.h)
fs_metaread()
fs_metawrite()
find referenced inode
use get_block(), put_block(), alloc_zone() to access/modify metadata region, pointed to by zone 9