Assignment 4
CMPS 111

Forrest Kerslager, Nick Noto, David Taylor, Kevin Yeap, Connie Yu

June 10, 2014

Purpose

The purpose of this assignment is to extend the MINIX filesystem to include a special metadata area for each file. This extra space can be used to store notes and information about the file that are separate from the normal file contents.

Changes

The design of the program has had changes since our first design document was submitted. Here is a summary of these changes:

- The source file for our metaread/metawrite user library functions are located in /usr/src/lib/libc/posix, not /minix.
- Methods discussed in the implementation of library and system calls are now more detailed and corrected according to our version of Minix.
- Details on file copying
- Details on file deletion added
- Implementation details
- Testing details

1 Design

1.1 Metadata

Metadata regions are attached to files that are accessed and edited using the new metaread() and metawrite() functions. This region will be a 1024 byte block that we must point to using a file's i-node structure. In order to maintain backwards compatibility we must repurpose unused fields in the i-node structure to signify the presence and location of a metadata region. In the i-node structure we will reuse two preexisting fields to keep track of a file's metadata. The sticky bit is set to 0 by default for all files. We will reuse the sticky bit by setting it to 1 if metadata is available for a file. Since the sticky bit is only used for directories this should not present any conflicts with the preexisting file system. If the sticky bit is 1 we will access the block number of the metadata in the i-node field i_zone[9]. Zone pointer 9 holds a triple indirect pointer which is rarely if ever used. Using i_zone[9] will only cause conflicts for the very large files that require it. We can check if i_zone[9] is unallocated and therefore available by comparing it with the macro NO_ZONE. A file must also be prevented from growing large enough to use i_zone[9].

1.2 System Call

New system calls metaread(int fd, void *buf, size_t count) and metawrite(int fd, void *buf, size_t count) will be created that take a file descriptor, pointer to a buffer, and the number of bytes to read or write. These calls resemble read() and write() except their purpose is to read or write to a file's metadata region. If a metadata block is not available or empty on a read nothing will be returned. If a metadata block is not available on a write it will create one and link the file to it. If a file is large enough to require use of i_zone[9] then metadata can not be used. The calls return the number of bytes read/written just as read()/write() work.

1.3 User Library

A user library allows a user program to invoke a system call. First a prototype of our function must be placed in a preexisting header, we chose /usr/src/include/stdio.h. If the declaration was put in a new header file then the makefile would have to be edited to include it. A source file must then be created in /usr/src/lib/libc/posix/ and added to Makefile.inc in the same directory. The source file must be the name of the function prepended

by an underscore. The defined function will first create a message that includes the parameters of the call including file descriptors, buffer addresses, and the number of bytes to read/write. It will then pass execution to the the vfs system call using _syscall(int who, int syscallnr, register message *msgptr). This call tells a specific server which handler to use and passes it the call details in the message. Finally an unused call number must be assigned in /usr/src/include/minix/callnr.h and /usr/src/include/minix/callnr.h.

1.4 VFS

In order to create a new system call handled by VFS we must modify a few files in /usr/src/servers/vfs. Any new system calls must be added to an unused entry in table.c. A prototype for the system call must then be added to proto.h. The functions themselves will be defined in their appropriate .c files. For the purpose of defining the new metaread() and metawrite() functions we will add definitions to read.c and write.c.

VFS does not directly handle the new system calls. VFS will use message passing to communicate with a FS, a specific instance of MFS, that can handle the system call. When it receives one of the new system calls it will use the file descriptor to look up a v-node using get_filp(). Using the v-node it can find the reference to the i-node in the correct FS. The function cpf_grant_magic() must also be called to allow the servers to write to userspace. A message will be built using information from the vnode, grant id, number of bytes to transact, and a request code. The request code is defined in /usr/include/minix/vfsif.h and /usr/src/include/minix/vfsif.h. In these files a new request must be defined and NREQS must be incremented. Adding this request number requires us to add dummy spaces in other unrelated systems request handler tables. The file table.c must have a no_sys entry for the same request number in /usr/src/servers/hgfs and /usr/src/lib/libvtreefs. If these additions aren't made the compilation will fail with an "array size is negative" error.

Now that the message is fully built it can be sent to the correct filesystem using the call fs_sendrec(fs_e, &m).

1.5 Message Passing

Messages are used to allow communication between VFS and MFS. When VFS begins a metaread() or metawrite() it will send a message to the instance of MFS that handles the i-node it is interested in. A field to pass an i-node

reference already exists. However, as discussed above a message type must be added to /usr/include/minix/vfsif.h that signals a read or write to the metadata. The message type will act as an index for MFS to run our handler for metadata.

1.6 MFS

When MFS receives a message from VFS it will use the m_type message field to index into its handler table and call the metaread or metawrite handler. As discussed in the Metadata section these handlers will use the repurposed fields in the i-node to identify if metadata exists and where the address of the metadata. If the region does not exist or is empty on a read it will return nothing. If the region does not exist on a read or write it will create one and link the i-node to it. Any newly allocated blocks for metadata will be intially zeroed. If a read signal is received MFS will copy the specified number of bytes of the metadata block into the buffer. If a write signal is received MFS will copy the specified number of bytes of the buffer into the metadata region. When MFS finishes its request VFS revokes the grant it previously requested.

1.7 File Deletion

When a file with metadata is deleted the block allocated for metadata should also be freed. A file is removed using the unlink() library call which eventually leads to the unlink() handler in MFS. The function that eventually takes part in returning allocated zones to the pool of free memory is free_inode() which calls free_bit(). free_bit() only requires a pointer to the relevant superblock structure, a flag to use the IMAP or ZMAP, and the inode or zone number. An easy way to free metadata would be to call free_bit() from put_inode() where the inode data is easily accessible. If the sticky bit is set and zone 9 is not NO_ZONE then the contents of zone 9 can be passed to free_bit().

1.8 File Copying

When a file is copied it should allocate the new file a metadata block and copy the metadata to the new block. Copying a file should not link both files to the same block. The code for copying files is located in /usr/src/cp/cp.c. The function of interest is copy1() which calls copy(). The process of copying metadata could be repeated on a lower level here, but it is easier to library calls to metaread() and metawrite(). If a file's sticky bit is set and zone 9 is not NO_ZONE then we can fill a buffer with the source files metadata

by calling metaread(). After the copy function has created the new file we simply call metawrite() and pass it the buffer containing the source file's metadata.

1.9 Errors

In general many of the same errors handled by read()/write() are also handled by metaread()/metawrite(). This is because the code is largely based on read()/write() as an example. However there are errors unique to the meta functions.

- Passing a negative number of bytes into a meta call returns EINVAL.
- A failure to find a vnode based on the file descriptor with get_filp() returns an error code just as read()/write().
- If a request for a grant using cpf_grant_magic fails panic() is called with an error message.
- A failure to find an inode in MFS using find_inode() returns EINVAL.
- If the sticky bit is set and zone 9 is empty then this will be considered a directory. An error will be printed and 0 will be returned to the user.
- If the sticky bit is cleared and zone 9 contains a pointer this will be considered a valid file that requires use of zone 9. An error will be printed and 0 will be returned to the user.

2 Implementation

2.1 /usr/include/minix/callnr.h /usr/src/include/minix/callnr.h

Macros for metawrite and metaread are added to these files in the unused slots 69 and 70. These are necessary to create the metaread() and metawrite() interface.

2.2 /usr/include/minix/vfsif.h /usr/src/include/minix/vfsif.h

Message request types are added as macros to these files for metaread and metawrite. These allow us to send unique requests to MFS using the m_type

field to specify calling metaread() or metawrite(). NREQS must be incremented to account for these two new message types.

2.3 /usr/src/include/stdio.h

A prototype for the interface version of metaread(int _fd, void *_buf, size_t _n) and metawrite(int _fd, void *_buf, size_t _n) was added to this file. A user program should include this header to use the interface.

2.4 /usr/src/lib/libc/posix/_metaread.c /usr/src/lib/libc/posix/_metawrite.c

These two files provide a clean interface for a user to call the do_metaread and do_metawrite handlers in VFS. We pack a message with the interface's parameters and make a call to VFS using _syscall(). This function passes a call number and message to the process pointed at by VFS_PROC_NR. The call numbers for metaread and metawrite are 69 and 70, respectively. These files must be added to Makefile.inc in the same directory.

2.5 /usr/src/lib/vbtreefs/table.c

A no_sys marker must be place in this table in place of the metaread and metawrite handlers. Without it compilation fails.

2.6 /usr/src/servers/hgfs/table.c

A no_sys marker must be place in this table in place of the metaread and metawrite handlers. Without it compilation fails.

2.7 /usr/src/servers/vfs/table.c /usr/src/servers/vfs/proto.c

Call numbers for do_metaread and do_metawrite are placed into unused slots of this table. These numbers are used by _syscall() to call these vfs handlers from our interface. The prototypes for do_metaread(), meta_read_write(), do_metawrite(), and req_meta() are placed in proto.h.

2.8 /usr/src/servers/vfs/write.c

2.8.1 do_metawrite()

We implement most of our code within meta_read_write(). With do_metawrite(), we pass in the WRITING flag into meta_read_write() so we can perform the metawrite system call. This function returns meta_read_write(rw_flag) with WRITING as the flag.

2.9 /usr/src/servers/vfs/read.c

2.9.1 do_metaread()

We implement most of our code within meta_read_write(). With do_metaread(), we pass in the READING flag into meta_read_write() so we can perform the metaread system call. This function returns meta_read_write(rw_flag) with READING as the flag.

2.9.2 meta_read_write(rw_flag)

This function's only parameter flag whether to do a read or write using the READING or WRITING macro. The majority of its actual parameters are stored in the global message pointer m_in. This function first does some error checking before building the message to send to MFS. The field m_in.nbytes is checked to make sure a negative number of bytes was not requested for copying. Next a vnode is retrieved by passing the file descriptor m_in.fd to get_filp(). If get_filp() encounters any errors it returns. If the requested number of bytes is zero then zero is immediately returned to the user. req_meta() is then called to build the request message to MFS.

2.10 /usr/src/servers/vfs/request.c

2.10.1 req_meta(fs_e, inode_nr, rw_flag, user_e, user_addr, num_of_bytes, cum_iop)

This function sends the actual request to MFS. First it acquires a grant with cpf_grant_magic() which allows us to copy back and forth between user and kernel space. A message is built whose field m_type is set equal to the request type defined in vfsif.h, REQ_METAREAD or REQ_METAWRITE. We then send the request in the form of the message using fs_sendrec() and finally revoke the grant using cpf_revoke().

2.11 /usr/src/servers/mfs/table.c /usr/src/servers/mfs/proto.c

Call numbers for fs_metaread and fs_metawrite are placed into unused slots of this table. These numbers are matched to the value of a message request type defined in vfsif.h to call fs_metaread and fs_metawrite. The prototypes for fs_metaread and fs_metawrite are placed in proto.h.

2.12 /usr/src/servers/mfs/read.c

2.12.1 fs_metaread()

We implement most of our code within rw_block(), but this functions purpose is retrieve the inode and setup the call to rw_block(). First we locate the inode that is referred to by the user using find_inode(). If the inode is NULL, we return(EINVAL). Along with this check, we exit the program if the specified region was a special block file. Our new function rw_block() is this call along with various parameters including a READING flag. After it returns we mark the inode dirty if any IO was performed.

2.12.2 fs_metawrite()

Performs a similar function to fs_metaread() but for writing instead. Preferably this code should be in its own function and have each handler calling the new function with a READING or WRITING parameter.

2.12.3 rw_block(rip, b, off, chunk, left, rw_flag, gid, buf_off, block_size, completed)

This function contains instructions to read or write a block, depending on the rw_flag passed in. We check two constraints to see if reading/writing is possible. If the user is trying to read/write from a directory or large file, then we print a warning and return zero bytes written. If block_t b is not present, we allocate a new zone with alloc_zone() and zero its first block with zero_block(). The sticky bit in i_mode is toggled and i_zone[9] is equated to the new block number.

At this point a block buffer is fetched by passing get_block() a block number. We then call sys_safecopyto() to either write to a user buffer or the block buffer. If writing, we mark the block dirty. We then return the block using put_block(). The function finally returns the cumulative number of bytes processed which eventually reaches its way back to the user's library call.

3 Testing

- File: no testing file

  Purpose: 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).

  Results: Normal use of minix including compiling the kernel, rebooting, creating, editing, and deleting non-meta files shows no sign of errors due to the changes.

- File: test2.c

  Purpose: Demonstrate adding a note "This file is awesome!" to a README.txt file, and later reading back the note.

  Results: Correctly writes and reads metadata. Output should include the string "This file is awesome!" and does.

- File: test3.c

  Purpose: Demonstrate that changing the regular file contents does not change the extra metadata.

  Results: Correctly outputs the proper metadata after regular file contents were changed. Output should include the string "Don't Change" and does.

- File: test4.c

  Purpose: Demonstrate that changing the metadata does not change the regular file contents.

  Results: Correctly outputs the original regular file contents after metadata was changed. Output should include the string "Regular file contents" and does.

- File: test5.c

  Purpose: Demonstrate that copying a file with metadata copies the metadata.

Results: The copied file should correctly contain the same metadata, but this feature was only discussed in design. The feature was not actually implemented.

- File: test6.c

  Purpose: Demonstrate that changing the metadata on the original file does not modify the metadata of the copied file.

  Results: The metadata of the copied file should not change when changing the original file's metadata, but this feature was only discussed in design. The feature was not actually implemented.

- File: test7.c

  Purpose: Demonstrate that creating 1000 files, adding metadata to them, then deleting them does not decrease the free space on the filesystem.

  Results: Calling df before and after test7.c executed should show that the free space did not decrease, but this feature was only discussed in design. The feature was not actually implemented.

For testing purposes the sticky bit can be toggled on a file using the command

```
chmod +t filename
```