Design Doc 4

John Carlyle, Andrew Edwards,
Morgan McDermott, and Kaya Zekioglu

June 8, 2014

1 Changes Since Last Submission

We combined our two system calls into a single `.c` file as they were highly dependent upon the same subroutines. These calls each then in turn use code from the `request.c` file in the vfs server to pass on the meta read or meta write request to mfs. The request is sent to mfs via minix's message passing system. Before the request leaves vfs it is packed up into a message. If it is a meta read request then the address of an empty buffer is in the message to receive a response, if it is a meta write request then the address of a character buffer is in the message for mfs to write to the metadata region of the specified file.

In mfs the message is received by `main.c` which, based upon the message type decides which of the functions in its function table to have handle the request message. We created a new message type by reusing the constants `REQ_READ` and `REQ_WRITE` in the header file for `vfsif`, when these request types are seen in MFS it checks to see if our flag (which is the unused `REQ_SEEK_POS_HI` integer in the read/write request message) is set for "meta". In our case the handler function (defined in `mfs/meta.c`) takes over at this point to preform the read/write task. Similar in function to the existing `mfs/read.c`, our `fs_metareadwrite` function finds the desired inode and allocates a new zone for its metadata (if not already done). Once the zone has been located we store or retrieve the relevant metadata and pack the results up into the response message. This message, received by either of the `do_meta` functions, contains the status code (did the read or write succeed?) and the relevant buffer (assuming a read occurred).

2 General Description

This project consists of a user library and a pair of system calls for reading and writing metadata to and from a specified region in the Minix file system. Each file can have up to one kilobyte of associated metadata. Users can read and write this metadata using the system calls `metar` and `metaw` respectively. These system calls will be added to `/usr/src/servers/vfs`.

The Minix file system uses inodes to represent files and directories. Each inode contains metadata records, group and user IDs, timestamps for last access/data modification/inode modification, and a list of zones. According to a report by Karthick Jayaraman, the zone pointer 9 is safe to use for storing metadata. However we found this not to be the case in our own testing. Instead we decided to use zone 6. The rational for this is two-fold:

1. When we used zone[9] to store metadata `fsck` complained loudly at each reboot.
2. We evaluated the filesystem for maximum filesize and decided that zone[6] should be safe to use because no individual inode was large enough to need it.

Furthermore, we can check if metadata exists by checking if `i_zone[6] == NO_ZONE` (`NO_ZONE` is just a macro for 0). To actually allocate the data, we will use `alloc_zone` and `get_block`. Because of the modularity of the Minix filesystem, our system calls will have to send a message from VFS to MFS for reading and writing metadata. These will be introduced to `/usr/include/minix/vfsif.h`.

3 System Calls

Our system calls will be defined in the VFS directory in two files: `metaw.c` and `metar.c`. The function prototypes will be added to `proto.h`, and we will register the two functions to `table.c` under unused entries.

3.1 `metaw`

The function `metaw` takes 3 arguments: a file descriptor, a pointer to a character buffer, and a number of bytes to write. This input file descriptor will be mapped to the corresponding vnode. From here, we will get a handle on the the data necessary to make a request for a grant. The grant will allow us to then make a request to MFS where we will finally have access to an inode. With access to an inode we now have a handle on the i_zone array. We can then ascertain if a block already exists for metadata, if it does not we create one and zero it (by way of getting a block number and then calling `get_block` on it). Now that we have a block pointer we can get a pointer to a `struct buf*` within the block pointed to by our block pointer. Here we can directly insert the passed message using `sys_safecopyfrom`. The results of this copy are then stored in the response message which is sent back to VFS. In VFS the response is opened and the resulting buffer, integer pointer, and status code are passed down to the original user program which called our system call `metaw`.

3.2 `metar`

The function `metar` is very similar to `metaw`, except it only takes two arguments: a file descriptor and a number of bytes to read. All steps above are again taken to get direct access to the block number in `i_zone[9]`. If the block doesn't exist we report this error and read nothing. If the block exists we get it and then extract the `struct buf*` which contains the metadata to be read. We use `sys_safecopyto` and put the results into the buffer that was passed in the original message (up to `nbytes` are read). The response message is then filled in and sent back to VFS. In VFS the response is opened and the resulting buffer, integer pointer, and status code are passed down to the original user program which called our system call `metar`.

3.3 Changes to MFS

3.3.1 `main.c`

The main loop of MFS was modified to catch our meta flag before it indexes into its call vector. This is done because changing the length of the call vector had global consequences (across all servers).

3.3.2 `meta.c` (and `proto.h`)

We define new functions (and prototypes for them) in MFS. These functions emulate the functionality written into `mfs/read.c`. After some error checking on the incoming message we grab a pointer to the metadata struct buf (if it doesn't exist yet, make it) and preform the read/write using the `sys_safecopy` suite.

4 Testing

In the `tst` directory you will find two files named `metacat.c` and `metatag.c`, run `make` to compile them into executables (after installing the filesystem changes and rebooting of course). Usage is as specified in project instructions. To test our changes we preform the following operations:

- write n bytes to metadata then read $\leq n$ bytes
- write n bytes to metadata then read $> n$ bytes
- attempt to read bytes before they have been written
- attempt to write too many bytes (more than 1kB)
- attempt to write to an invalid file decriptor

Then to test compatabillity with the system/environment we preform the following tests:

- `cat` the contents of a file before and after adding metadata to ensure it doesn't corrupt contents
- `metacat` the metadata on a file after editing the file to ensure it hasn't been corrupted

- copy a file with metadata and then `metacat` it to show persistence
- as above, but after copying we change the metadata and check that it has not changed the original file's metadata
- record free space, create 1000 files, add metadata to each, delete all 1000 files, check that free space is same

4.1 Copying

This section is included in testing because that is the only place in the project instructions where copying is mentioned.

To effectively copy metadata it is necessary to, for each file to be copied or moved, ascertain if that file contains metadata. The process of doing this takes us all the way into the metadata location so we simply grab it if it exists. One way to implement this would be to edit `/usr/src/commands/cp/cp.c` to preform this task for us. This could be accomplished by calling our `metar` on the source file and then `metaw` on on the destfile (with the same buffer to thread the metadata through).

Alternatively if we didn't want to edit `cp` itself we could create a wrapper for `cp` in a shell script. We would call the normal `cp` function to make the destfile and then grab its fd. Once we had that we could call our system calls to update the metadata afterwards. This approach is less elegant but interferes less with existing use of `cp` (that way other uses of `cp` on non-metadata carrying files are not affected by the overhead of checking for metadata).

4.2 Test Results

The result of this test suite is summarized below.

test	**result**
normal `metatag`/`metacat`	no error, data returned as expected
try to read too many bytes	error reported from syscall `metar`
try to read non-existent metadata	get empty buffer back with non-zero status code
try to write too many bytes	first 1024 get written, non-zero status code returned
try to write to a non-existent file	error message and non-zero status code
check that contents of file are not corrupted after `metatag`	same contents, no corruption
check that metadata hasn't been corrupted by changing file contents	same metadata, no corruption
copy a file with metadata and then `metacat` it to show persistence	doesn't persist
as above but we confirm that two copies of a file have independent metadata	n/a
freespace check on 1000 metatagged files	rm does not remove metadata

5 Shortcomings

In this section we would like to discuss where we fell short. First off, we were unable to use zone 9 of the inode. Our system calls worked fine, but whenever we rebooted, the filesystem check (fsck.c) complained that we were using zone 9 as a direct pointer, when it expected an indirect one. We scoured the Minix source code to find where to change the interpretation of zone 9, but we could not find anything. Another problem with our design is that calling metacat on a file with no metadata gives an MFS error. We tracked this error to its source, which is within mfs/main.c. Our idea was to have a logical within mfs/meta.c that would return a special "NO_METADATA" code if a user attempted to metacat a file with no metadata. This worked, except that after the call to metar completed, the main loop in mfs/main.c crashed with an error. For some reason it was getting a negative request number. We know that negative request numbers are inteneded for replies, so we are not sure how this occured. We also did not modify any of the Minix system commands, so cp and rm do not behave as desired.