Cameron Gravel
Anthony D'Ambrosio
Rachelle Tanase
Jordan Hyman

Design

Posix Library Functions

In Posix, we created two functions `metacat()` and `metatag()` that sys call do_metacat and do_metatag.

- `metacat()` reads the metadata stored in the specified file's inode.
- `metatag()` writes the specified message to the specified file's inode.

VFS Handler

We created two VFS sys call handlers to handle `metacat()` and `metatag()`, called `do_metaread()` and `do_metawrite()`, respectively.

- Request The request function, `req_metarw()`, opens a grant to communicate with MFS. It then packs the relevant information into a message and sends the message to the MFS.
- Message Our message takes a similar form to the existing message for `read()` and `write()`. It holds 4 fields: the request type, the inode number of the file, the grand ID, and the number of bytes to transfer.

MFS Handler

We decided to use the triple indirection zone 9 to store our metadata because it rarely used for most files. Our MFS handler allocates a block in zone 9 in the inode if it is not already allocated, then reads or writes to the block as requested.

Implementation

Posix Library Functions and System Calls

We changed two entries of the VFS call table: `do_metaread()` in entry 69 and `do_meatwrite()` in entry 70. These tell the system (ie sys calls) which functions will handle the system calls sent from `metacat()` and `metatag()`. The `metacat()` and `metatag()` library functions are in `_metarw.c` in `lib/libc/posix`. They both take a file descriptor, number of bytes, and buffer pointer as arguments. They pack the arguments into a message, and then call `_syscall()` with their respective call table entries.

VFS Handler

In `servers/vfs/read.c`, `do_metaread()` and `do_metawrite()` are our system call handlers. They each call `meta_read_write()` and pass in an argument that represents their intention to read or write. `meta_read_write()` checks validity of the arguments for our request function, `req_metarw()`, and then calls that request.

Request Handler

In `servers/vfs/request.c`, `req_metarw()` first calls `cpf_grant_magic()` to set up a grant to transfer data between the user space and MFS. It then populates a message with the inode number of the file, the grand ID, and the number of bytes. Finally it uses `fs_sendrec()` to send the request to the MFS.

MFS Handler

We first needed to add two calls to the MFS call table. We added `fs_metarw()` to entries 33 and 34. We also had to add a 34^{th} and 35^{th} entry to each other filesystem server (ext2, hgfs, pfs). The new entries in the extra filesystems are left as `no_sys`. In `fs_metarw()` we first find the inode of the desired file. Then it determines the block size of the inode and store the values sent in with the request message. Finally, it calls `meta_rw_chunk()`, which handles the actual reading and writing. `meta_rw_chunk()` first allocates zone 9 of the inode if it has not been allocated yet, otherwise it just accesses `i_zone[9]` scaled by `s_log_zone_size` (found in the super block). Next, it uses `sys_safecopyfrom()` to write from the grant to the inode block, or `sys_safecopyto()` to read from the inode block to the grant. If we wrote metadata, we first zero the block using `zero_block()` so that any old data will not be re-used, and then we mark the block dirty before calling `put_block()` and returning.

Copy

In `/usr/src/commands/cp`, `cp.c` holds the minix implementation for copying a file. In `copy()`, after the file contents have been copied, we call `metacat()` on the original file to get the metadata associated with it. We then `metatag()` that data onto the new copy.

Testing

We created a test script, `mk.tests`, to test all requirements. Running `./mk.tests` will cause all tests to run. First we add metadata to a file and `metacat()` it to the screen. Then we copy that file to a new file and affirm that the metadata remains the same in both files. Next we change the metadata of the original file and affirm that the new file's metadata remains unchanged. Next we change the content of a file and affirm the metadata associated with it does not change. Then we check that changing the metadata does not change the file contents. Finally we create 1000 files with metadata and remove them, and affirm that we have no memory leaks.

Results

We were not able to implement removing of metadata, resulting in a loss of 4kB for when a file with metadata is removed. All other tests run as expected.