CS 111 - Project 4

Design Document

5.28.14

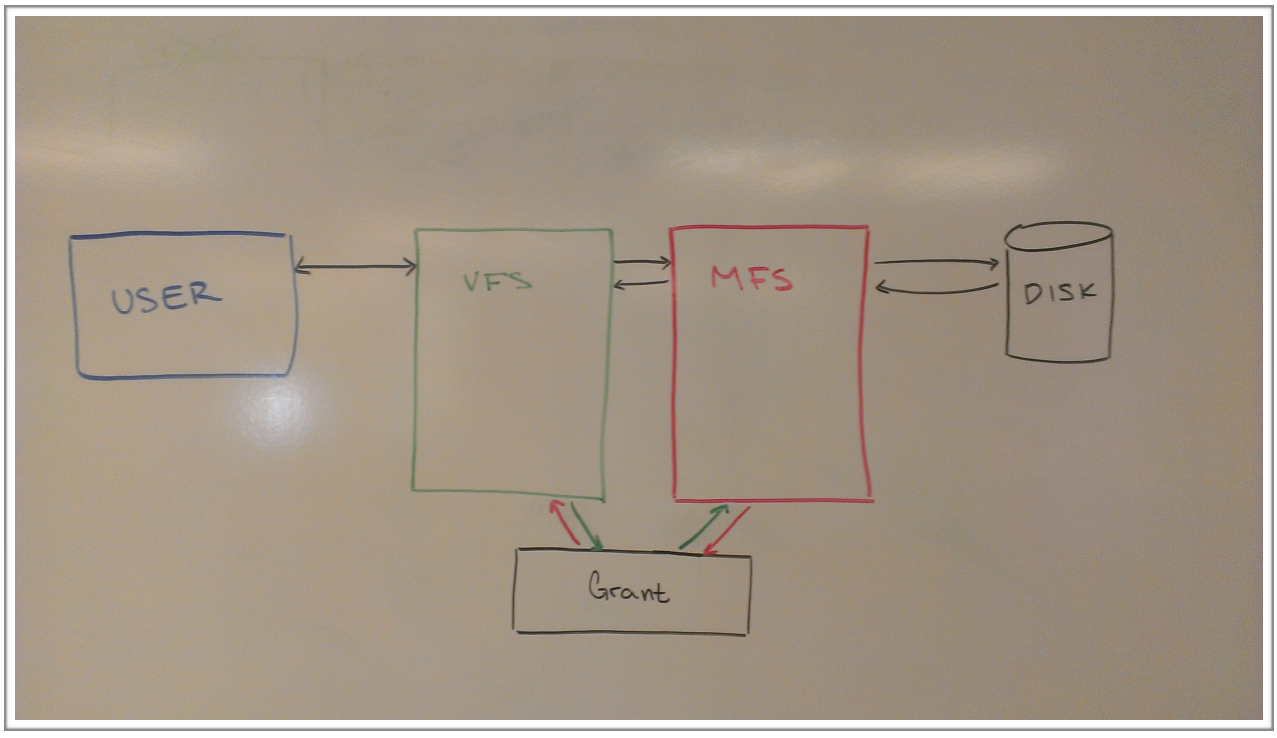
Greg Arnheiter
Ian Hamilton
Justin Kwok
Benjamin Lieu
Yugraj Singh

Purpose - This project overviews the design and implementation of extending the MINIX file system to create a custom interface for the Operating System. For this project, an extra metadata area will be allocated to a file to store special info that is separated from the file's normal attributes, and provides convenient user access to these regions and data.

DESIGN - A high-level description can be broken down into the following subparts:

User Interface	Implements wrappers for sys calls that access data
VFS	Retrieves data necessary for FS access and determines procedure to be done in MFS.
MFS	Carries out chosen procedure and returns / writes requested data to shared grant area.

First, system calls were made to pass messages between VFS and MFS. The messages must be interpreted by MFS to access the correct data fields, and provide the correct operation to that data field depending on the user message. These system calls duplicated functionality of existing file system calls for read/write, but were modified to manipulate the fields specified (meta region or regular region). A user interface was then created to wrap the existing system calls with the modified versions that maintain functionality of the existing filesystem while expanding to include meta region operations.



Project Design Modules

IMPLEMENTATION

Part 1 - User Interface

This project details the design and implementation of an extra metadata field to be associated with the existing VFS/MFS in MINIX 3.1.8. An interface is needed so that applications in user space can read and write to the extra region. To do this, a user library is created to provide functions for using sys calls and making structure access easier. First `metatag` and `metacat` programs were implemented to open a given file and return the file descriptor, the size of the block, and the buffer for data to be read or written to. These values will be passed into the respective library functions; `Metatag` calls `metawrite()` and `metacat` calls `metaread()`. If an error occurs while opening a file, then the program exits and prints an error message to `stdout` saying where the error occurred. The program will also exit from in `metatag()` if the user's input message is larger than 1024 bytes.

The functions `metawrite() and metaread()` were created, to act as wrapper functions for the system calls already in the MINIX file system. They save a place on the message for the FD, size, and a pointer to the buffer used for the metadata. We used the numbers 65 and 66 to represent indices for read and write in the sys call look-up table.

	Operation
metacat	Program that provides user access to data region by creating file descriptor and passing it with additional arguments such as size and message buffer to _metaread() and additionally, prints buffer to stdout. Located in `metacat.c`
metatag	Program that provides user access to data region by creating file descriptor and passing it with additional arguments such as size and message buffer (with user data) to _metawrite(). Located in `metatag.c`
metaread()	Wrapper function for sys call meta_read() used in VFS. Allocates metadata region, initializes fields that're passed to the function: FD, size, message buffer. Located in `_metaread.c`
metawrite()	Wrapper function for sys call meta_write() used in VFS. Allocates metadata region, initializes fields that're passed to the function: FD, size, message buffer. Located in `_metawrite.c`

Part 2 - VFS

Within VFS, system calls `meta_read()` and `meta_write()` were created to mimic the functionality of `do_read()` and `do_write()` which are existing system calls that pass the correct parameters to meta_readwrite() that carriers out the intended action given by one of the flags `READING` or `WRITING`. Both `meta_readwrite()` and `read_write()` call `req_readwrite()`, except `meta_readwrite()` sets an extra (added to the original) parameter `meta = 1 when calling req_readwrite()`, indicating that the sys call is

going to be used on the extra metadata region rather than the original file data region. The message is then passed to MFS using the `fs_sendrec()` function.

Function	Operation
meta_read()	Returns result from meta_readwrite(READING). Located in read.c
meta_write()	Returns result from meta_readwrite(WRITING). Located in write.c
meta_readwrite()	Implements read_write() with an added parameter 'meta' that indicates 0: normal read/write and 1: meta read/write. Located in read.c
req_readwrite()	Checks meta flag for correct procedure (regular or meta data region). Meta = 1 utilizes and unused portion of message (REQ_SEEK_POS_HI) to indicate to MFS that it's the metadata region being manipulated and not original FS data. Located in request.c

Part 3 - MFS and Allocating Extra Memory

When `req_readwrite()` is called by `meta_read/write()` it passes the built message to the MFS via `fs_sendrec()` which calls the function `fs_readwrite()`. In this function, we check the message flag (`REQ_SEEK_POS_HI`) that was passed from VFS to MFS to determine whether the metadata region is being accessed. If not, `fs_readwrite()` performs normal action, otherwise the region is allocated (if unallocated) and accessed.

After extra memory for metadata has been allocated, a pointer to the entire region of data is stored in zone 9 of the i-node of the file to which the metadata belongs. If metadata memory has not been allocated for a given file, zone 9 will contain a pointer to the first block of memory containing all zeros. Whenever the metadata field is accessed, the pointer to that region will be scaled by `log(zone_size)` to provide the first block of unused memory in the region.
Once we have determined meta or regular, we must also determine if the operation is a read or write, and then to call the appropriate `safecopy_to/from()` function to safely read or write to the grant.

In the existing MINIX file system, `i-node` zone 9 is used to hold triple-indirect pointers for files of unusually large size. By using this existing field for metadata region pointers, it is no longer possible to have arbitrarily large files.

Part 4 - Testing

Test	Descriptor
MINIX Compatibility	Compare FS's before / after to ensure corruption is avoided.
"This File is Awesome!"	Write this message to metadata and retrieve later.
Changing Regular File Contents	Changing file doesn't change metadata.
Changing Metadata	Changing metadata doesn't change file.
Copied File Copies Metadata	Copying file also copies the metadata. Uses temp buffer to copy metadata and compare against new copy for error.
Change Metadata on Original File	Changing data on copied file doesn't change original
Remove Orphaned Metadata	Adding then removing many files doesn't decrease free space. [This test was NOT implemented successfully in project]

The above tests are designed to demonstrate the backwards compatibility of the new functions, by manipulating various elements in the File System and (hopefully) observing the appropriate changes without unintended effects propagating to other elements or data fields not intended in the original call.

Part 5 - Miscellaneous and Additional Info

In order to get metadata to copy from file to file (when copying files) we use metaread() to copy the source file's metadata to a buffer and then metawrite() to copy that buffer to the destination file. This happens within cp.c. Another modification made was to exec.c because the file contained instances of req_readwrite(), a function that we modified the prototype for.