# Project 4: Extending a Filesystem

Design Document

rev2 6/8/2014

Matt Caruano, Taylor Caswell, Eric Evans, Mesuilame Mataitoga, Radhika Mitra

## Purpose

The purpose of this project is to extend the MINIX file system to allocate a special metadata area for each file. This extra space can be used to store notes about the file that are separate from the normal file contents. Tools must also be designed and implemented to allow users convenient access to this metadata.

## Design Plans

### Overall implementation

At a high level, we will expand upon the existing MINIX file system by adding the ability to store metadata with every file. This will be done using file inodes, and additionally will only use existing inode components, thus maintaining backwards compatibility. Users will be able to read and write file metadata through the use of two provided C programs *metacat* and *metatag*.

### Metadata internals

- Our system will allow users to store up to 1024 bytes of metadata for each file. What the user wishes to store here is at his/her own discretion.
- We will need to be able to keep track of a binary value to indicate if a given file has metadata or not. This will need to be kept track of it in the inode, together with the block number where the metadata will be stored.
    - We will keep track of the address of the metadata information block in a rarely-used “Zone 9” pointer. This pointer is only used for excessively large files, and should not interfere for the purposes of this project
    - A value of 0 in the Zone 9 pointer of the inode will be used to indicate that a file has no metadata.
- We will also be sure that the appropriate free() call gets made to the metadata of a file that has metadata at the time it is deleted from the filesystem.

### Metadata interaction

- We implemented user space functions metadata_read() and metadata_write() so users can read and write file metadata. In order to accomplish this, we implemented functions do_mread() and do_mwrite() syscalls in VFS, which will in turn be called by our user space functions: metadata_read() and metadata_write().
- The parameters for the user space functions are as follows:
    - metadata_read(int *file_descriptor*, char * *buffer*, unsigned n_*bytes*)
    - metadata_write(int *file_descriptor*, char * *buffer*, unsigned n_*bytes*)

- do_mwrite() will start the chain of calls in VFS that will ultimately send a message to MFS to allocate a block to write metadata to the file.
    - do_mread(), just like do_mwrite() will start the chain of calls in VFS which will ultimately send a message to MFS to read metadata from a file.
    - The hierarchy of calls in the VFS are in the following order:
        1. do_mread() /do_mwrite()
        2. mread_write()
        3. req_mreadwrite () → calls → sendrec () to pass the message to MFS
- In order to get access to the MFS, we reused the same request that was used in the read and write calls (message parameters: REQ_READ, REQ_WRITE)
- We used an unused field in the message struct, REQ_REN_LEN_OLD in order to differentiate within the MFS between a regular read/write call and metaread/metawrite calls.
- Inside MFS read.c, the function fs_readwrite() we first got hold of the files inode struct from its file descriptor using flag within the message struct: REQ_INODE_NR. In order to allocate (if not already allocated) the ZONE 9 block of the block zone pointers on the inode struct we used it to store user metadata.

**Testing**

- Testing will be done incrementally throughout the project to ensure that functionality is working for all parts of the project up to the part currently being implemented.
    - A simple function to be called by the VFS that prints debug statements after handling our own syscall. This will ensure we are correctly making a syscall being caught by the VFS.
    - A couple of message sent from the VFS to the MFS. These will each handle a function that re-implements either the read() or write() calls, along with a debug() function. This will ensure that we are correctly message passing.
    - To the redundant write() call, include a call to allocate the 1024 bytes block for the meta if it is not already allocated, and do something with it to ensure that the block is properly allocated, such as copying characters there and then printing them out within a separate call. This will ensure that the allocated block for the metadata is properly being allocated.
    - Within the redundant read() call, add a code to read the data in the block, and print it out. This will ensure that the block that will contain the metadata is properly storing its data, and that it can be access by our own read() call.
- After the program is finished, we will make test scripts that demonstrate the required abilities of our program as layout out in the project 4 specifications.