UNIVERSITY OF CALIFORNIA, SANTA CRUZ
CMPS 111: OPERATING SYSTEMS
PROFESSOR NATHAN WHITEHEAD
SPRING 2014

Programming Project 4: Extending a Filesystem

Team Awesometastic
Kevin Andres, Jeffrey Deng, Benjamin Gordon, Xiaoli Tang, Wai Wong

Monday 9th June, 2014

1 Changes to Original Design

Since the first design, we've focused our attention on getting VFS to pass messages to MFS successfully. System calls to VFS should have a file descriptor argument, which can be checked by VFS so it can find an inode and then pass a message to another filesystem server. We have thought of methods that make it possible for the library function *metaread()* can use a system call to dispatch to VFS *do_metaread* to call *req_metaread* which packages a message to be sent to MFS *fs_metaread*.

2 Purpose

In this project, our goal is to extend the MINIX filesystem to include a *metadata* area for each file. Such an area can be used to store other information about a file that is separate from the normal file contents (and can hold up to 1024 bytes). We are to design system calls for user space processes which allows for both the reading and writing of metadata and to add a metadata to files that do not already have one. We will also create tools to allow users to access the metadata areas conviniently.

3 Design

In order to effectively approach creating a new metadata region and to allow system calls for the user to read and write to files' metadata region, we must utilize the *virtual file system server* (VFS). This server accepts filesystem syscall requests and passes them to an actual filesystem server. In this project, we will use the MINIX filesystem server (MFS) to handle the disk filesystems.

3.1 System call

In order for userspace processes to communicate with a filesystem, we must be able to create a syscall that can be routed through VFS. This system call should be able to:

- Create a new metadata region if one is not present
- Write data to a file's metadata region.
- Read data from a file's metadata region.

We can add a system call to VFS similar to *read()* and *write()* and will have arguments including:

- a file descriptor
  - VFS can map this to a *vnode*, which is a reference to an *inode* in MFS.
- a pointer to memory
- the number of bytes to be read or written.

We will need to add a message type for reading and writing the metadata region, and a message can be passed to MFS once VFS checks the arguments.

3.2 Managing Metadata

We will need to keep track of metadata. Metadata points to a space in memory and a file may or may not have it. There are four cases of interacting with metadata:

- reading metadata from a file that has it.
- writing metadata to a file that has it.
- reading metadata from a file that does *not* have metadata.
- writing metadata to a file that does *not* have metadata.

3.2.1 Inode structure

We will allocate an unused data block in the inode:

- If a block is not already allocated, use get_block() to safely do so.

If it is already allocated, proceed to write the data normally.

We can use the last zone pointer in the inode (zone 9), which is a triple indirection pointer, as the *sticky bit*. Since it is unlikely that the file system will use it, we will utilize this as the sticky bit for storing metadata. As this field is initialized to zero, adding metadata sets the sticky bit, and if metadata is read, the sticky bit can be checked:

- If the sticky bit is set, metadata already exists and represents the metadata itself.
- If the sticky bit is not set, there is no metadata and is kept initialized to zero.

3.2.2 Vnode

Vnode of a file does not contain the pointer of the inode associated with the file. However, we can obtain access to the inode via the message protocol. By including a new message type, we can invoke system calls in MFS which returns the inode.

3.3 Interface

We will create command line tools for the read and write of metadata by simply invoking our metaread() and metawrite() system calls. To simplify the design, we will first run the commands as a program. If time permitted, we will implement the kernel calls by modifying system.c and system.h to include the custom commands.

4 Implementation

The coding for this program should pretty much only be implemented in the following areas:

- User space test programs
- Library functions wrapping the syscalls
- The implementation of the syscalls in the filesystem (VFS, MFS)
- Changes to MINIX (including header files and servers)

We will deal with the following files:

- table.c (from /usr/src/servers/vfs/table.c)
- proto.h (from /usr/src/servers/vfs/proto.h)
- proto.h (from /usr/src/servers/mfs/proto.h)

- vfsif.h (from /usr/src/include/minix.vfsif.h)
- callnr.h (from /usr/src/include/minix/callnr.h)
- exec.c (from from /usr/src/servers/vfs/exec.c)
- write.c (from /usr/src/servers/vfs/write.c)
- read.c (from /usr/src/servers/vfs/read.c)
- read.c (from /usr/src/servers/mfs/read.c)

The two files *write.c* and *read.c* in this VFS server folder will include our implementation of our system calls.

4.1 table.c

In this file, there are some *unused* entries for system calls. We will use a couple of these entries for the system calls we will create for this project:

- In entry 105, we will have *do_metaread*
- In entry 106, we will have *do_metawrite*

4.2 proto.h (VFS)

We will of course add the prototypes of these system calls at the end of this file as well:

- _PROTOTYPE(int do_metaread, (void));
- _PROTOTYPE(int do_metawrite, (void));

4.3 proto.h (MFS)

We will add the prototype for MFS as well:

- fs _meta_readwrite()

4.4 vfsif.h

We will define macros:

- #define REQ_META_READWRITE (VFS_BASE + 33)
- #define NREQS 34
- #define REQ_META m9_s4
- #define REQ_BUFF m9_s2

4.5 callnr.h

We will define macros for these system calls as well:

- #define METAREAD 105
- #define METAWRITE 106

4.6 exec.c

Here, we redefine *req_readwrite* with 0,0 to fill up the two new parameters.

4.7 write.c

At the VFS level, when it comes to writing metadata, we will deal with creating the system call *do_metawrite*, as well as the function *read_write()* to actually call the *WRITE* system call.

- From the userspace processes called *metawrite*, *do_metawrite* receives the message (whatever nees to be written to the file).
- The function *do_metawrite* will call the function *read_write()*.
  - We add 0 as a parameter for read_write() for metadata.
  - *do_metawrite* will handle the message and interpret it so that it can be passed to *read_write*
- In *read_write* we can check whether a file has metadata or not.
  - This of course utilizes the *sticky bit* inside the inode structure (as discussed in the design section).
- *read_write* will pass the message into zone 9 of an inode, which points to the allocation of the metadata that we want to store.
- From here we can deal with the MFS-level process *write_map()*
  - This will actually allow writing to metadata for this file system.

4.8 read.c (VFS)

We will of course take a similar approach in terms of reading metadata.

- A program can call *metaread* which calls for metadata to be read from a file.
- We need to check if metadata even exists for the file at this point
  - We check for the sticky bit, and if it is set, then it does indeed exist.
- The system call *do_metaread* is called, which in turn calls *read_write* which calls the *READ* system call.
- The MFS-level process *read_map()* allows metadata from a file to be read.

4.9 read.c (MFS)

We focused on two important functions in this file:

- fs_meta_readwrite()
- rw_metachunk()

4.9.1 fs_meta_readwrite()

We used find_inode() to return an inode for use. From inode access, we set variables *scale*, *bp*, size of each block in the zone. We set *scale* to the size of the zone. We set variable *block size* equal to inode's block size. We set *f_size* to inode's *i_size*.

We also had to set all the variables based on the *fs_m_in*, which is the global variable of message in: *rw_flag*, *gid*, *position*, and *nrbytes*.

We want to check if inode's izone 9 is empty (which is indicated by NO_ZONE).

- If it is empty:
  - and we are *writing*, we will use alloc_zone() to define the memory space and then set struct bp which contains the block and the relevant information.
  - and we are *reading*, we will print an error message and 0 block.
- If it is *not* empty:
  - We will loop through the chunks of data (while *nrbytes* > *0*) and use metaread or metawrite to process each chunk of data and adjust counters and pointers for next chunk of data.

Arguments and return value:

- Arguments: This function takes no arguments
- Return Value: Returns OK

4.9.2 rw_metachunk()

This function sets *scale* to the size of the zone. We will set bp and check if it is null (block is not there).

- If we are reading:
  - we will perform perform *sys_safecopyto()* to retrieve data from userspace and store in bp->b_data.
- If we are writing:
  - we will perform *sys_safecopyfrom()* to pass data from MFS to user space.

4.10 Commands

We created two new folders for the commands:

- metacat
- metatag

Metacat functions similarly to *cat* while *metatag* essentially relies on *metawrite*. *Metacat* allows the user to call from the command line and read metadata, while *metatag* is called from the command line to write to it.

5 Testing

In order to show that the requirements for this project our met we must be able to:

- 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).
- demonstrate adding a note "This file is awesome!" to a README.txt file, and later reading back the note.
- demonstrate that changing the regular file contents does not change the extra metadata.
- demonstrate that changing the metadata does not change the regular file contents.
- demonstrate that copying a file with metadata copies the metadata.
- demonstrate that changing the metadata on the original file does not modify the metadata of the copied file.
- demonstrate that creating 1000 files, adding metadata to them, then deleting them does not decrease the free space on the filesystem.