CSCI 401 Operating Systems
Instructor: Gedare Bloom

Final Project
Wednesday November 16, 4:00 P.M.
Wednesday December 7, 2:00 P.M.

Requirements

If any of these requirements create a problem for you or are unclear, come talk to me.

- You should do this assignment in a small group of 2 or 3. If your group is not working out, you can request to be removed from the group and be responsible for completing on your own; this request must be made no later than 11:59 P.M. on December 1.
- You should not discuss the problem, code, or solutions with other students who are not in your group. You may not use code downloaded from the Internet without specific permission from me.
- If you discover any ambiguities in this assignment ask on Piazza and I'll clarify.
- Always watch your email and the course site for updates on the assignments.
- Use the same Linux kernel version we have been using, 3.16.0

Goals

The goal of this assignment is to gain a better understanding of Linux filesystems and performance.

Pre-Requisite Task

You need to create a "Team" on Bitbucket with all of your team members, and add Professor Bloom (user: gedare) to your team. Then create a private Git repository from your team's page and add Issue Tracking and a Wiki. Make sure all of your team members can Write to the repository (under Access Management of the Repo Settings), and you might want to prevent deletion and history re-writes of your master branch (under Branch Management). You should keep your design documents and team notes in the Wiki, and use the Issues Tracking to identify features to implement and bugs to fix. Assign issues to the responsible developer based on your team roles. All code should go into this repository. You also may like to include your written reports and presentation slides in the repository.

Assignment

Your group can choose one of the following project ideas, or create your own. If you create your own you need to discuss it with me before starting on it.

All projects should be viewed as multi-step projects, and you should do them that way. So at each stage you should have working copies of the first few steps or simplified solutions, and can demo them and submit them – even while you may be working on

more advanced features. Each project suggests several 'feature' steps, although you can make your own decisions about exactly where you want to break it up. But you definitely want to have multiple, incremental, solutions.

Because these projects are multi-step, I don't expect every group to complete all of the possible steps – some will and do extra features, others may run into some technical problems and only get some features completed, others may do really nice versions of the core features that justifies not getting to all of the steps. The number of steps completed will also depend on the size of the group.

In all cases, you must submit an initial design, a final report, presentation, and source code as specified in the Submission section.
The documentation at the http://lwn.net/Kernel/ and http://lwn.net/Kernel/Index/ website should be very useful and the always useful LXR source index at http://lxr.linux.no/source/

Project: Structured File Filesystem

File systems do not have to be generic storage systems that store and view data as binary blobs. A file system can be specialized to know about the structure and types of data that is stored in it. This can allow the file system to present a useful view of structured data and let users run standard tools like grep, ls, echo, and editors to view and modify files.

In this project you will select a file format that has a well defined structure. For example:

- unix password or shadow files (or other software configuration files such as for webservers, email servers, etc)
- xml (maybe a specific form of xml that is restricted in some ways).
- xhtml (as it is much more structured then html)
- latex (restricted somehow)
- …

You will need to create a new filesystem, for which the backing store (where the data is stored) is not a raw disk partition or block device, but rather a file of the selected format.

Your file system should:

1. be able to be mounted and unmounted, where the file that should be the backing store is specified at mount time

2. create a directory hierarchy based on the file format structure of directories representing the different elements of the file. For example, in the password case, you might make a directory for each user (i.e. each line of the password file) and then subdirectories for each field in the user record. In the latex case, you might make a directory for each top-level section command, and subdirectories for paragraphs.

3. Each piece of data in the file should be visible either as contents of a file in the file-system, or as metadata attached to a file or directory. For example if an XML

entity has an attribute of creation time that could be shown as the creation time on the file storing that entity. The names of the files in the directory tree should be appropriate to the specific format of the file.

4. The contents of any files can be read through standard system calls such as read, open, readdir. The first version of your file system should be 'read-only' meaning that the backing store file cannot be modified by writing or editing the files in your file system. You can return an ENOTSUPP error for any 'write' calls or operations that are requested by a userspace program. It is as if you mounted your filesystem readonly.

However, once that is working you should then make a 'version 2' of your file system that adds support for writes. This can be divided into two separate feature sets:

1. Only files support write operations. So the content of a record that is viewable as a file can be edited, but the directory structure and metadata can not be edited. So I can change the value of a particular users' password or change which shell they selected, but I can not create new users (as that would require making a new directory).

2. Directory and metadata operations are write-enabled. Now the directories can be deleted, added, renamed and metadata can be changed.

You can use the libfs file system library or the FUSE "user-space" filesystem library to help implement your file system.

Testing

To test your filesystem you should create several different data files that are valid examples of your structured file format. You can use existing files but make sure the data in them is public since others will see it.

You should then write several shell scripts that test your file system by using standard unix commandline programs such as 'ls', grep', and cat for the read-only portion, and 'echo' and some editor to test the modifiecation of your file. You can also run some more complex tests by hand using standard editors such as emacs or vi, for these results capture a screenshot or a text log of the shell to include in your report to show the tests you did. Your tests should verify that all of the required features of the file system work and that in common error cases, reasonable error return values are generated.

You then should create a small c program that will access your file system using the standard C file IO libc functions and will time how long it takes to do certain operations when the input size varies. For example, you could time how long it takes to open each directory in your filesystem (for a specific backing file) or how long it takes to read all the data in the filesytem into a memory buffer (for a small and a large file) to check how scalable your filesystem is to large documents

Project: Filesystem Attributes

One of the newer features of some file systems is file attributes. Read the explanation of how Mac OS X is using them at http://arstechnica.com/reviews/os/macosx-10.4.ars/7.

Think about how you would support file attributes in each of the three software levels:

1. Application: So a specific application stores attributes (like Word).

2. Desktop: A desktop environment like Gnome/KDE/Win32 stores attributes that are visible only through the desktop (not commandline).

3. Library: In a common library (like the C library) that many applications and other libraries use.

4. Kernel: Inside the OS and file system.

User Defined Attributes

You will create a mechanism to allow users to attach specific attributes to files and directories. This will consist of several new system calls, modification to some existing file system code, and the creation of new user-space programs to display the attributes.

An attribute is attached to a specific file or directory and consists of a AttrName and a AttrValue.

All attributes will be stored in a special "attribute" directory that is created for each file or directory that has attributes attached. This attribute directory will have a name created by pre-pending the file name with a dot "." and appending the file name with the string "_attr". So a file testfile will have an attributed directory called .testfile_attr. Creating attributes to files that already start with a "." is not allowed. All other files or directories can have attributes attached to them.

An attribute will be stored in the attribute directory as a file whose name is the AttrName and whose contents is the AttrValue. So an attributed named "Creator" whose value was "Gedare Bloom" attached to file "testfile" would be stored in .testfile_attr/Creator with the Creator file contents being "Gedare Bloom".

The file permissions for the attribute directory and the attribute files should match the permissions of the file they are attached to.

Specification

This specifies exactly what features you need to add and how they should work.

System Calls

For all system calls, appropriate error values should be returned for situations like 'removing an attribute that doesn't exist', 'adding an attribute with an invalid name', lack of resources, etc.

```
long cs401_set_attribute(char *filename, char *attrname,
                         char *attrvalue, int size)
```
This call sets the attribute named “attrname” to the value in “attrvalue” which has length “size” to the file named “filename”. The filename must be an absolute path to the file. The strings should be null terminated, and reasonable limits should be enforced on all strings. The return value should be 0 for success, or a negative error value.

```
long cs401_get_attribute(char *filename, char *attrname,
                         char *buf, int bufsize)
```
This call gets the value of the attribute named “attrname” attached to the file named “filename”. The value is stored in the buf pointer which must have at least “bufsize” bytes of storage. The filename must be an absolute path to the file. The strings should be null terminated, and reasonable limits should be enforced on all strings. The return value should be the number of bytes returned for success, or a negative error value.

```
long cs401_get_attribute_names(char *filename, char *buf,
                               int bufsize)
```
This call gets all of the names of attributes that are set for the file “filename”. The list of attributes is returned as a “:” (colon) separated list in the “buf” string. The buf pointer must have at least “bufsize” bytes of storage. The filename must be an absolute path to the file. The strings should be null terminated, and reasonable limits should be enforced on all strings. The return value should be the number of bytes returned for success, or a negative error value. This system call can be used with the cs401_get_attribute() call to list ‘all’ of the attributes attached to a specific file.

```
long cs401_remove_attribute(char *filename, char *attrname)
```
This call removes the specified attribute from the file. It should remove the file that was created in the attribute directory. If this was the last user attribute for this file, the attribute directory should also be removed. Note, this will require some locks as some other process may be adding a new attribute at the same time, and should not fail because the directory was removed from under it. The return value should be 0 for success, or a negative error value. This system call can be used with the cs401_get_attribute_names() call to remove ‘all’ of the attributes attached to a specific file.

User space test programs

You should create two user-space programs that cat set attribute values and retrieve them. These programs must make use of the system calls specified above and can not directly access the attribute directory.

setattr The setattr program should take as input a single name=value pair and a list of files to apply the attributes to. For example (using shell wildcards for *, your program does not need to parse for wildcards)

```
> setattr "Professor=Dr. Jekyll" *.c myfile.txt
> setattr Type=worddoc *.doc
```

In the first case all .c files and the myfile.txt file should have the attribute of Professor added with value “Dr. Jekyll”. In the second case the Type attribute should be set to “worddoc” for all files in the directory with file extension .doc.

listattr This program should take as input an attribute name and a list of files, and should output the values and names of the attributes requested for those files. For example:

```
> listattr Owner *.c yourfile.asd
file.c Owner=Gedare Bloom
second.c Owner=Fred Flintstone
yourfile.asd Owner=Jones
> listattr Type *
file.c Type=Cfile
second.c Type=Cfile
myfile.txt Type=Text
test.doc Type=worddoc
```

The first should list the Owner name/value pair for all c files and the file “yourfile.asd”. The second should list the “Type” attribute for all files in the local directory. If a file does not have the attribute requested it should not be listed.

A special attribute name is “LISTALL” which will cause the listaddr program to list all of the attributes attached to the specified files.

Kernel Support for Attributes

These attributes you have now created should be better integrated with the regular kernel file functions. For example if you remove a file, all of the attributes should be removed with it. If a file is renamed, the attributes should move with it.

Look at the kernel implementation of ‘unlink’ and ‘rename’ and add support in them to use your system calls to correctly remove the attributes for unlink, or to move the attributes to the new file name for rename.

You will need to verify that your changes work on the ext2 and ext3 file systems, but you do not have to have them work with any other specific file systems. However, many others will probably work if you implement your changes in the generic unlink and rename code. For example look at the system calls sys_rename and sys_unlink and see how they work. The lxr source browser will be very useful for this stage of the project to figure out how these work and where to place your changes.

Testing attributes

You should now create a small microbenchmark program to evaluate the performance overhead of your attribute feature. This micro benchmark should create a number of attributes attached to some files and evaluate how much time it takes to add/remove/list attributes compared with just creating/removing/stating the file itself. Note that caching effects might need to be mitigated or adjusted in your analysis.

You should also test the impact of your ‘unlink’ and ‘rename’ changes by timing how long they take on files that have attributes and files that do not. Also you should compare their cost with a kernel that does not have any of your changes in it so you can see what the performance cost of simply having support for your attributes adds even if you don’t use them.

Project: Device Driver

You will implement a pseudo-device driver for mailbox objects. A mailbox is like a persistent pipe. It has a read end and a write end. Processes can open either end, then read or write to it, then they close it when they are done. The mailbox and its contents do not go away when one or both ends of it are closed.

When used, programs will be able to open, read, write and close files in /dev that correspond to the mailbox. When processes write into the box, the data they write in will be available for any process to read. Each process that reads will get the data that was written by previous processes in order.

The semantics of the operations should be that a process must open the mailbox before reading or writing. Only one process can have each end of the mailbox open at once, so one can be reading and one can be writing, but two cannot be writing at the same time. Mailboxes are persistent until the kernel is rebooted. So data written into a mailbox will remain until it is read out or the system restarts. If nothing is in the mailbox, a reader will block. If something is written into a mailbox and a reader is blocked, the reader will be unblocked. If the writer closes the mailbox while a reader is waiting, the reader continues to wait. Mailboxes have a maximum size, if a writer tries to write and insufficient space is available, the writer will block until a reader removes data from the mailbox. All reads and writes will return or write the full amount of data requested, if they cannot they will block. So partial writes and reads can not occur.

You will need to create a kernel module that implements your device driver. The module should be able to be loaded multiple times, each time creating a different device as the mailbox, so several distinct mailboxes can exist. The module should register with the VFS to handle the open/read/write/close operations. If your module is unloaded then the contents of the mailbox may be lost. Your module should not be able to be unloaded until all user processes have closed the mailbox. You probably will want to use some kernel locks and wait queues in your driver.

You can add additional features to your module, such as allowing a query for how many bytes of data are available, supporting partial writes and reads, allowing multiple writers or readers to open the mailbox, but having each read or write be atomic.

You should break this project into several steps, starting with creating a basic module with appropriate data structures that can load and unload, then you can go to a module that only allows reading and initialize your module with fixed string of bytes in the init function (so there is something to read), and then adding write functionality. Some locking and error handling can then deal with the remaining cases.

Write one or more userspace testing programs that will open/close/read/write to your driver all at the same time, or in series. These tests should verify the correctness of your driver. They should also do some benchmarking of your driver to determine the overhead of doing a write, how does it scale with large writes and reads (many KB to MB), what is the latency to 'ping-pong' a write to a reader and get a return write on a separate mailbox.

Project: Freezing the Filesystem

You will implement kernel changes to ensure that modifications to the filesystem are not persistent between reboots. Similar to a snapshot system, except that it ensures that the machine always boots into a known configuration, rather than allowing for the return to an arbitrary point in the state of the system.

At any time, the system can either be "thawed" or "frozen". If a machine is thawed, then any changes made to the filesystem will remain. If the machine is put into a frozen state, then all changes to the filesystem will be reset when that part of your program is run.

In this project, you most likely will not have time to implement a full freezing of the filesystem. Instead, you will need to identify a set of changes that you can log and roll-back.

Kernel Support

You will need to hook in to system calls that modify the filesystem in order to log changes that are made. The strace command will be very helpful in identifying the appropriate system calls. Syscalls receive file pointers that need to be translated to paths via path_lookup and dpath to determine precisely which file is being processed.

The log entries will need to be pushed to a userspace application that can write the log to persistent storage. Additional system calls may be helpful to tell the kernel how to find the userspace application.

Userspace Helper

You will need to maintain a buffer between a userspace application and the kernel modifications. Your userspace application will commit log entries to a file. When activated, your userspace application (same or different) will parse the log file and roll back any modifications to the filesystem.

Proceed through this project in phases:

1. Logging from Kernel space to user space
   First establish hooks in system calls that access/modify the filesystem, and set up a logging infrastructure that you can test. Figure out which system calls you will monitor, ultimately freezing (by logging) all changes would be ideal, but you will need to limit the scope of your project.

2. Tracking changes

Add the capability of your logging to detect changes in the filesystem. You may find it is easiest to identify a file as "dirty" if a write is performed. Tracking a minimal set of changes may prove difficult. Also, be careful that actions performed by your userspace application are not themselves logged, as this could cause an infinite loop in the kernel.

3. Backing up files and directories
   Using the tracking capability, you will store clean copies of files and directory structures (with the userspace app) so that you can undo the changes.

4. Automatic restore on command
   Extend your userspace app to roll back changes that were logged.

Testing and Performance

For determining correctness, generate some simple test cases and observe that your logs reflect your test cases. Your test cases will probably consist of scripts that make deterministic accesses/modifications to the filesystem.

Gather timing statistics for operations that you modified for your kernel (e.g. create, unlink) and compare with an unmodified kernel. Note that the effects of the page cache can easily destroy timing analysis, so be sure to mitigate its effects.

Initial Design

You should find a group and select a project during this week/weekend. Each group then should submit a 1-2 page plan stating who is in the group, what project you will be doing (whether one I suggest, or a different one), a brief project plan with timeline listing each piece of the project and when you plan to be done with it, and any choices your group has made so far about variations in the project or specific features you are going to emphasize. All group members must arrange to join the same FinalProject_Groups on Blackboard. This is due by Wednesday Nov 16th at 4 P.M. on BlackBoard.

Presentation

You should prepare a short 10 minute presentation on your project including what you did, how it worked, any design changes you made, your benchmarking and testing results, and any conclusions you have. Some slides with sample output from your program and graphs of any data should be included. The presentation will be submitted with your final package as well as separately, so that all presentations can be downloaded and prepared for smooth transitions. Do not plan to use your own laptop for the presentation, and do not plan to give a demo.

Submission Instructions

1. Your submission must contain no object code or binaries. Only submit source code, makefiles, data files, and your writeup.

2. Verify your code works using test programs.

3. For kernel modifications, generate patch files that include **ONLY** your source code changes. Read your patch file to verify that it contains the changes you want. Do not submit a copy of the Linux kernel!

4. When you are done, create a README that describes your submission, any notes about problems you had, any special features you implemented, and the results of running your test program with an explanation of how the test output verifies the correctness of your code.

5. Prepare a tar ball (zipped folder is fine) by creating a directory that has all group members last names in the name and containing:
    - each kernel feature as a separate patch file
        - include any incremental steps that were complete
        - subfolders can be used for phases, but place final product in top directory
    - .c/.h source files for kernel modules
    - .c/.h source files for user programs
    - .c/.h source files for testing and benchmarking programs
    - Makefile (preferably with multiple targets, and make clean)
    - Report including write-up of all the material from your presentation with additional detail and results.
    - Presentation Slides

    and tar and compress the directory using

    ```
    tar -zcvf LastName1LastName2LastName3.tgz
    LastName1LastName2LastName3/
    ```

6. Your file must be complete and uploaded to Blackboard by the deadline. Only one group member needs to upload the final submission. Upload your presentation in both the tgz file as well as e-mail it separately to the instructor. This will make the presentations go more smoothly. Do not expect to give a demo or rely on your own laptop for the presentation, as it may not work perfectly.