Advanced Operating Systems

Computer Science Department
Indiana University
Bloomington, IN 47405

Main Menu

- Home
- Contact
- About
- Schedule
- Assignments
- Resources
- Fun

The Tao

Thus spake the master programmer:

"Without the wind, the grass does not move. Without software, hardware is useless."

— Geoffrey James, *The Tao of Programming*

Programming Assignment 4: File System

Assigned	Nov 7, 2012
Design	Nov 14, 2012
Due	Dec 3, 2012
Note	This problem set is to be done in groups of two.

Introduction

The OS/161 file system, `emufs`, is just a layer on top of the Unix file system. In this assignment you will augment `sfs`, the native OS/161 file system, in three ways:

- Provide support for concurrent access.
- Add a hierarchical directory structure.
- Add a buffer cache to improve performance.

As usual, this assignment is broken into two parts. Be aware of the due dates for each part! The design document includes some code-reading questions, the results of a test run, and your design for the modifications outlined above. The implementation consists of everything you need to do to make your design run!

Code Reading (10 points)

The OS/161 `sfs` file system we provide is very simple. The implementation resides in the `fs/sfs` directory. The `fs/vfs` directory provides the infrastructure to support multiple file systems.

`kern/include`: You should examine the files `fs.h`, `vfs.h`, `vnode.h`, and `sfs.h`.

Question 1. What is the difference between `VOP_` routines and `FSOP_` routines?

`kern/fs/vfs`: The file `device.c` implements raw device support.

Question 2. What vnode operations are permitted on devices?

devnull.c implements the OS/161 equivalent of /dev/null, called "null:". vfscwd.c implements current working directory support.

Question 3. Why is VOP_INCREF called in vfs_getcurdir()?

vfslist.c implements operations on the entire set of file systems.

Question 4. How do items get added to the vfslist?

vfslookup.c contains name translation operations. vfspath.c supports operations on path names and implements the vfs operations. vnode.c has initialization and reference count management.

kern/fs/sfs/sfs_fs.c has file system routines for sfs.

Question 5. There is no buffer cache currently in sfs. However, the bitmaps and superblock are not written to disk on every modification. How is this possible?

Question 6. What do the statements in Question 5 mean about the integrity of your file system after a crash?

Question 7. Can you unmount a file system on which you have open files?

Question 8. List 3 reasons why a mount might fail.

sfs_io.c has block I/O routines, and sfs_vnode.c has file routines.

Question 9. Why is a routine like sfs_partialio() necessary? Why is this currently a performance problem? What part of this assignment will make it less of one?

sbin/mksfs implements the mksfs utility which creates an sfs file system on a device. disk.h/disk.c defines what the disk looks like.

Question 10. What is the inode number of the root?

Question 11. How do files get removed from the system?

Setting up

Propagate any changes you've made to your previous config files into the ASST4 config file. Then config and build a kernel. Tag your current repository asst4-begin.

Initial Testing

Once you have everything built, format the disk and run the file system performance test from the kernel menu by specifying fs1.

System Calls

You will find a utility in sbin called dumpsfs. Having a tool that can dump an entire file system is an invaluable debugging aid. As you modify your file

system, be sure to keep this utility up to date, so that it can continue to be useful to you. First, you will need to add support for the system calls listed below.

- `open`, `read`, `write`, `lseek`, `close`, `dup2`, `chdir`, `getcwd`
- `mkdir`, `rmdir`, `getdirentry`
- `fstat`
- `remove`, `rename`
- `sync`, `fsync`

The general requirements for error codes are the same as in Assignment 2; for details, consult the OS/161 man pages. Specific requirements:

- If a file or directory is expected to exist by the semantics of a call, and it does not, the resulting error code should be ENOENT.
- If a file is encountered where a directory was expected, the resulting error code should be ENOTDIR.
- If a directory is encountered where a file was expected, the resulting error code should be EISDIR.
- If an operation cannot be completed because the disk is full, the resulting error code should be ENOSPC.
- If removal of a non-empty directory is attempted, the resulting error code should be ENOTEMPTY.
- As in assignment 2, when an invalid file handle is used, the resulting error code should be EBADF.
- If an attempt is made to operate in a prohibited manner upon "." or "..", the resulting error code should be EINVAL.
- If an attempt is made to rename a directory into one of its subdirectories, the resulting error code should be EINVAL.

`open`, `read`, `write`, `lseek`, `close`, `dup2`, `chdir`, `getcwd`

For any given process, the first file descriptors (0, 1, and 2) are considered to be standard input (stdin), standard output (stdout), and standard error (stderr). These file descriptors should start out attached to the console device ("con:"), but your implementation must allow programs to use `dup2()` to change them to point elsewhere.

A large part of this assignment is designing and implementing a system to track this state. Some of this information (such as the current working directory) is specific only to the process, but others (such as file offset) is specific to the process and file descriptor. Don't rush this design. Think carefully about the state you need to maintain, how to organize it, and when and how it has to change.

Note that there is a system call `__getcwd()` and then a library routine `getcwd()`. Once you've written the system call, the library routine should function correctly.

Synchronization

SFS does not currently protect itself from concurrent access by multiple threads. If you think back to Assignment 1, we asked you to solve the cat and mouse problem. If your code was not properly synchronized, mice would get eaten, which leads to an ugly scene. A similar thing can happen here. For example,

because there is no synchronization on the free block bitmap, two threads creating files could both decide to use the same free sector. Your mission is to bullet-proof the filesystem so that two threads can use the filesystem harmoniously. You must allow multiple threads to have the same file open. When this is the case, your filesystem needs to implement the following (UNIX-like) filesystem semantics.

- **Atomic Writes:** file writes are atomic. If one thread has a file open for writing, and a second has the same file open for reading, the second thread will always be presented with a consistent view of the file with respect to each `write()` system call. On each read, returned data will be considered the state of the file before a write system call started, or after the write system call ended, but not what it looked like during the `write()`.

  Example: Say thread A performs a syscall to write 512 bytes to file F starting at location 0. At the same time, thread B performs a syscall to read 256 bytes of file F starting at location 128. Thread B is guaranteed that it will get 256 bytes of the file either before thread A's write takes place, or after thread A's write took place. It will never get back some bytes from F from before thread A's write started and some from after thread A's write ends.

- **Removal:** if a thread has a file open, a remove of that file eliminates the name in the directory immediately, but postpones actual removal of the file's contents until the file is closed by all threads that have it open (however, since the name is removed from the directory, no subsequent opens of the file will succeed). In addition, you should guarantee that the removal of a directory that is the current working directory (CWD) of some process does not cause a process (or our operating system) to crash. You may choose to disallow the removal or you may remove it and handle the process whose CWD was removed very carefully. In UNIX, the removal succeeds, however, subsequent accesses to "." or ".." can fail for the process that had the directory as its CWD.

  Example: Say Thread C has file F open (it doesn't matter whether it's for reading or writing). Thread D performs an remove syscall on file F. Thread C (and file F) are not affected by the remove. When thread C closes file F, it is then deleted.

- **Atomic Create:** If two threads attempt to create a file of the same name at the same time, specifying `O_CREAT | O_EXCL`, one should succeed, and the other should fail.

- **Synchronous Directory Operations:** If your system crashes after any of the following operations return, the disk should be left in a state reflecting that the call completed: `create`, `remove`, `mkdir`, `rmdir`.

Be careful to return appropriate error codes from calls to file-related methods in the file system! The syscalls you implemented in Assignment 2 rely on these values to operate correctly!

Synchronization Design

In your design document, write a one to two page description of what you will change. Discuss how you will provide synchronization for the file system. List which pieces need to be protected, how, and which synchronization primitives you will use.

Synchronization Implementation

Synchronize SFS. Ensure that the semantics described above are supported.

Be sure to test your code. Include test scripts, programs, and/or output with your submission. The `f_test` and associated programs in `testbin` are good places to start.

Hierarchical Directories

At this point, the file system should be working well; however, it would be much nicer if it handled hierarchical directories (i.e., pathnames). Not only is this possible, but fairly straightforward: currently, each entry in a directory is a regular file. By careful modification SFS can be extended to store both regular files and directories.

We have provided you with programs that implement the `mkdir`, `rmdir`, and `ls` commands. They are found in `bin`.

If you've answered the questions above, you'll notice that our pathnames are a superset of typical UNIX pathnames. As in UNIX, we use "/" as a pathname component separator.

Your code must do the following:

- If there is a file or directory in the top level directory named `foo`, accept an open request for `foo` (the leading "/" will be removed by the vfs layer).
- If there is a directory in the top level directory named `skippy` and a file in that directory named `crunchy` accept an open, create, or remove request for `skippy/crunchy` (see comment above about leading "/" characters).
- If there is no directory in the top level directory named `smuckers` it should disallow the creation, open, or remove of a file named `smuckers/grape`.
- Allow removal of an empty directory; disallow removal of a non-empty directory.
- Accept a directory name that ends in a / (though directory names that do not end in / are OK as well.)

Your code must **not** assume that the user wants all the missing directories created automatically when presented with a pathname that doesn't exist, on a create. For example, if there is a directory named `/bim/ska/la` and you mistakenly try to create a file named `/bum/ska/la/bim`, I don't want SFS to create the directories `/bum`, `/bum/ska`, and `/bum/ska/la` so that it can create `bim`. It should return an error.

Hierarchical Directory Design

Explain how you will implement hierarchical directories. Discuss any new data structures and synchronization you will need. Identify the parts of the system that will need to change. Where do you expect to have the greatest difficulty?

Explicitly discuss how you will implement cross-directory rename. This is very tricky as you will need to synchronize across multiple directories. While better than race conditions, deadlocks will make your system unusable. If the rename fails in any way, you must ensure that the file system is left in a consistent and

correct state. Think very carefully about this!

When deleting a directory, make sure that it contains no in-use entries.

Hierarchical Directory Implementation

Implement hierarchical directories. The `mkdir` and `rmdir` programs should work with them, as well as `cat`, `cp`, and `ls`. After you've created a file in the root directory named `test`, make sure that all of the following commands work when typed at your shell:

```
% /bin/ls /
% /bin/mkdir /foo
% /bin/cp /test /foo/test
% /bin/ls /foo
% /bin/mkdir /foo/bar
% /bin/cp /test /foo/bar/test
% /bin/ls /foo
% /bin/ls /foo/bar
% /bin/cat /foo/bar/test
```

Buffer Cache

You will find that the performance of SFS is not great. In particular, there is no buffer cache. You should add a buffer cache to OS/161. You will need to decide what data structures you need to implement this and what interface routines you will use. This is probably the most critical piece of your design. Figure out exactly where the buffer cache will interface with the rest of the system, and how you will maintain the integrity of your file system in the face of caching.

We recommend that you add the buffer cache last. Gather extensive performance measurements before you add the cache and after. You can use those measurements to demonstrate the effectiveness of your cache.

Buffer Cache Design

Include a description of your interface and data structures, your replacement algorithms, and any extra precautions you take to ensure the integrity of the file system. Also, discuss any additional support necessary to make sure that buffered data eventually gets written to disk.

Tools

It is required that the following user-level binaries (whose source we give you in bin) are functional when you submit! You shouldn't have to modify the source code for these tools; if, for some reason, you may think you do (because of a different implementation of the OS/161 filesystem than that assumed in these files), contact the instructor first and discuss it!

- ls.c
- rm.c
- cp.c

- cat.c
- mkdir.c
- rmdir.c

What to Submit

- Your original design document, as well as a list of any changes made to your design during implementation.
- A script output (`demo.script`) from running your shell creating, removing, and listing several directories; copying files of various sizes, and other tests that demonstrate that your file system is fully functional.
- The script output from running the synchronization stress test.

Please submit by the following:

- Create a directory in the root of your repository called `asst4`. Place all of your written work (design document, scripts, answers to written work) in this directory.
- Once everything is done and committed, tag your repository as `asst4-end`.

Please submit your design document by just putting it in the `asst4` directory and tagging your repository as asst4-design.

If you have comments or suggestions, email *p536@cs.indiana.edu*

Author: Andrew Lumsdaine
E-Mail: p536@cs.indiana.edu

Created: Sat Aug 25, 2001
Modified: Mon 29-Oct-2012