Asst 4 : File System

Assignment 4: File System

Due: Design document due 11:59 pm on Friday, April 17, 2009
Final submission due 11:59 pm on Friday, May 1, 2009

Objectives

After completing this assignment, you should:

- Be familiar with the `sfs` implementation and the OS/161 VFS layer.
- Understand the difference between file system dependent and file system independent layers in an operating system.
- Be able to implement fine-grained synchronization in an existing body of code.
- Be able to propose, implement, and evaluate a file system performance improvement.
- Be able to familiarize yourself with a large body of code and then extend its functionality, without rewriting the entire system from scratch.

Introduction

The OS/161 file system you have been using, `emufs`, is just a thin layer on top of the underlying Unix file system. In this assignment you will start using the native OS/161 file system, `sfs`, and work on it and the VFS (virtual file system) interface that allows both of these to exist at once.

The assignment has four parts:

- Add fine-grained locking to both SFS and the VFS-level code.
- Add support for hierarchical directory structure.
- Add a few more file-related system calls to allow using subdirectories.
- Do performance optimization and analysis.

As usual, this assignment also has a design phase. Be aware of the due dates for both the design and implementation! At the design due date, please submit your design for the modifications outlined above; also include your answers to the code-reading questions below and your initial peformance test results. Your final submission should include your code (everything you need to do to make your design run) and your performance writeup.

Code Reading (10 points)

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

Before you answer the code reading questions, be sure to apply the patch with the buffer cache code that we are providing for you this year.

In `kern/include`: Examine the files `fs.h`, `vfs.h`, `vnode.h`, `buf.h`, `sfs.h`, and `kern/sfs.h`.

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

`kern/vfs`: The file `buf.c` contains the buffer cache code.

Question 2. Why can't you call reserve_buffers() twice in the same file system operation? What could happen if this were allowed?

The file `device.c` implements raw device support.

Question 3. What vnode operations are permitted on devices?

`devnull.c` provides the OS/161 equivalent of `/dev/null`, called "`null:`".
`vfscwd.c` contains the code that handles current working directories.

Question 4. Why is `VOP_INCREF` called in `vfs_getcurdir()`?

`vfslist.c` implements the operations that apply to the entire set of file systems. It contains the One Big Lock currently used for all file system access.

Question 5. How do items get added to the VFS devices list?

Question 6. What happens if you try to acquire the VFS big lock when you already hold it?

`vfslookup.c` contains name translation operations.
`vfspath.c` supports operations on path names and implements the VFS-level operations.
`vnode.c` has filesystem-independent code for vnodes; mostly initialization and reference count management.

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

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

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

`sfs_io.c` has block I/O routines.
`sfs_vnode.c` has file-level routines.

Question 9. Why is a routine like `sfs_partialio()` necessary?

`user/sbin/dumpsfs` is a utility that dumps out the state of an SFS filesystem. This can be an extremely useful debugging aid.

`user/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 directory?

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. Tag your current repository `asst4-begin`.

Initial Benchmarking

Note: we want to adjust the System/161 disk timing model to make it more realistic. To make sure your starting numbers remain comparable to your ending numbers, please wait to do this until we install the new System/161 and give the all-clear. (We should be done by Tuesday night or Wednesday afternoon.)

Because performance testing forms an important part of this assignment, you should establish a baseline right away. Config and build a kernel, and collect initial performance numbers for the following tests:

- `fs1` from the kernel menu
- `fs3` from the kernel menu
- `testbin/psort`

Reformat the disk for each test run, to make sure you have a clean slate. Practice good benchmarking and statistical hygiene.

Place your initial performance numbers in a file called `initial.txt` and include this with your design document.

Requirements: Synchronization

Currently, the entire file handling subsystem of OS/161 (both SFS and the VFS-level code) uses one big lock. This lock is called `vfs_biglock` and is found in `vfs/vfslist.c`.

Using one big lock (and, especially, holding it across slow operations like disk I/Os) is not generally considered adequate. Your mission is to implement fine-grained locking for both SFS itself and the VFS support code. You should remove `vfs_biglock` and replace it with a locking system that allows concurrent operation.

Your implementation must be correct (and not, for example, allow two different files to both allocate and use the same disk block), and must guarantee (at least) the following conventional Unix file semantics:

- **Concurrent Opens:**

  A file can be freely opened any number of times, by any number of processes, whether for read or write.

- **Atomicity:**

  All file-related system calls must be atomic: they should either happen or not happen, and no other process should be able to see the filesystem existing in an intermediate state.

  In particular, `read` and `write` calls are atomic with respect to one another. Suppose process P calls `write` to write 512 bytes to file F starting at offset 0, and process Q calls `read` to read 256 bytes of file F starting at offset 128. Process P may see either the contents of the file from before A's write, or the contents after, but it must not see some of each.

- **Removal:**

If a thread has a file open, removing that file eliminates the name in the directory immediately. However, the contents of the file itself must not be removed until the file is closed.

Removing a directory that is open (or is some process's current directory, or is in use in some other fashion) must not cause a system (or process) crash. In this case it is acceptable to deny the removal attempt (Windows semantics?) or allow the `rmdir` to succeed, and remove the directory's name from the filesystem but keep a skeleton directory around (Unix semantics). In the latter case, attempts to access the "." or ".." names in the skeleton directory should fail.

Note that attempts to remove or rename the reserved "." and ".." *names* in a directory are specifically prohibited.

- **Atomic Exclusive 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.

- **Name Uniqueness:**

  The system should never allow two (or more) files (or other objects) with the same name to appear in the same directory at once.

Be careful to return appropriate error codes from the vnode and file-system operations in SFS! The syscalls you implemented in Assignment 2 rely on these values to operate correctly.

Note that we do *not* require that your file system be able to recover from a system crash. However, your file system must always, upon a clean unmount, leave the disk in a correct and consistent state.

Requirements: Hierarchical Directories

As provided, SFS provides only a top-level directory and has no support for subdirectories or directory operations. Each entry in a directory must be a regular file, not another directory. You must add support for hierarchical directories.

To implement hierarchial directories, you will need to extend SFS so that directories can contain other directories. You will also need to implement the vnode-level operations that pertain to hierarchical directories, including but not limited to `mkdir` and `rmdir`, and extend the `rename` operation to allow cross-directory rename.

You will also need to extend SFS and/or the VFS code to perform hierarchical name lookup. In OS/161, as you may have noticed, pathnames can have "device:path" syntax. Code already in VFS takes care of device names, and keeping track of what the overall root directory ("/") is; you must handle the path part, using the slash character (like Unix) as the pathname component separator. Decide whether (and how much) to handle this in FS-independent code vs. in SFS-specific code.

Each directory must always "." and ".." entries that point to self and the parent directory, respectively.

The path "skippy/crunchy" names an object "crunchy" in a top-level directory named "skippy". If the filesystem is called "test", is physically located on "lhd1" (the second disk device), and has been set as "/" (with the "bootfs" menu command), the following paths all name the object "crunchy":

- `/skippy/crunchy`
- `lhd1:skippy/crunchy`
- `test:skippy/crunchy`

- `skippy/crunchy` (after `cd test:`)
- `crunchy` (after `cd test:skippy`)
- `./crunchy` (after `cd test:skippy`)
- `../skippy/crunchy` (after `cd test:jif`)

All the system calls that accept pathnames must accept all forms of pathnames. (That is, they should all share the same name translation code.)

Directories should be created only explicitly (via `mkdir`) and never implicitly as the result of a pathname lookup. That is, if no top level directory `smuckers` exists, attempts to access `smuckers/grape` should fail and not create `smuckers`.

Other specific requirements:

- Only empty directories (directories containing only "." and "..") may be removed.
- It must be possible to rename whole directory trees at once, and this must be done in such a way as to always maintain the consistency of the directory hierarchy. (The tree must not, for example, be allowed to become a cyclic graph or a forest.)
- Attempts to remove or rename the special-purpose "." and ".." directory entries must be rejected.
- Path names that end in a trailing slash must be accepted, but may only be used to refer to directories and not regular files.

Requirements: New System Calls

You must add support for these system calls:

- `mkdir`, `rmdir`, `getdirentry`
- `fstat`
- `remove`, `rename`
- `sync`, `fsync`

These are all, at the system call level, straightforward translations to VFS or vnode operations. (You will also, for most of them, need to implement or update the filesystem-level support as discussed below.)

The detailed specifications for these calls can be found in the OS/161 man pages.

Requirements: Performance Hacks

As part of the benchmarking and performance analysis component (see below for further discussion) you must choose and implement two file system performance hacks.

These come in three basic flavors:

1. Change the on-disk layout to make accesses faster.
2. Change the internal mechanisms to use the existing layout more effectively.

3. Adjust the knobs, parameters, and magic constants in the code to find the optimum settings.

These involve different kinds of work; the first two involve a lot more coding (of different kinds) and the third involves little or no coding but a lot more benchmarking.

Choose from the following lists, picking two from different categories; or, invent your own. If you have some idea of your own or a favorite feature from some real file system, feel free to tackle it, but please clear it with us first; this is to make sure you don't get in way over your head.

On-disk layout:

- **Cylinder groups/zones.** Implement a cylinder group or zone scheme for SFS to reduce seek overheads. Divide the free map across the zones. Allocate blocks (and inode blocks) by zone.
- **Embedded data.** For sufficiently small files, store the data directly in the inode block instead of allocating a separate block for it. And, for files with link count 1, store a copy of the inode in the directory containing the file to further reduce seek overheads. (This is really two hacks in one, but both are fairly small.)
- **Larger disk blocks.** SFS just uses 512-byte disk blocks; switch to some other size (e.g. 4K).

Internal operation:

- **Sequential allocator.** Write a smarter block allocation scheme for SFS that tries to allocate sequential ranges of blocks even when many files are being extended at once.
- **Delayed block allocation.** Avoid allocating physical blocks for files until they're just about to be written to disk. This in many cases allows allocating a sequential run of exactly the right length.
- **Name cache.** Implement caching for directory name lookup. Note that caching negative results (names that do not exist) can be equally or more important than caching positive results.
- **Vnode cache.** Instead of dropping every vnode immediately when it's no longer referenced, keep vnodes around for a while in case the same file is reopened.
- **Read-ahead.** Implement read-ahead (cache prefetching) for sequential accesses. Avoid doing read-ahead on random accesses.
- **Small file prediction.** Ordinary read-ahead works best for medium to large files. Small files are frequently too small to benefit much. However, often small files are used in a fixed order or in easily recognizable patterns. Implement code to predict which small files will be used next and prefetch them.

Tuning:

- **Tune the buffer cache.** The buffer cache code has a lot of magic constants, most of which have default values that were picked out of a hat. Examples include the number of buffers written each time the sync daemon wakes up, when and how often the sync daemon should be awakened, the maximum fraction of memory to use for buffers, and so forth. Experiment with different values for these settings under a range of workloads and choose values that give good average performance.

For each of your choices, you must select and/or implement benchmarks or workloads that you will use to measure the performance. Pick workloads that can reasonably be expected to show improvement given what you're doing. Be sure to think about this in advance, and discuss the benchmarks you intend to implement in your design document. There's nothing more frustrating than finding out that your performance hacking didn't do any good, and nothing more aggravating than realizing it was because you were testing it on the wrong stuff.

Your performance writeup must evaluate the benefit of each of the optimizations you've done separately, and also the benefit of both together.

Note: if you pick one of the hacks that involves changing or extending the on-disk format, be sure to update the SFS tools (mksfs, dumpsfs, etc.) to match. (Otherwise you will find it hard to develop, and more importantly, we will find it hard to grade...)

Design Document

Your design document should explain how you intend to implement the hierarchical directory system, the fine-grained synchronization, and the performance hacks you plan to do.

Discuss any new data structures you will need. Describe your locking model, and the ordering of your locks, in detail. List what pieces of state exist, and which need to be protected, when, and how. 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. 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!

As discussed above, explain what benchmarks you intend to implement to support your performance optimization, and why you expect them to be relevant to the optimization you intend to do.

Implementation

First, implement the system call layer functions for the new system calls. Have them call the corresponding VFS or vnode operations. By this point in the semester this should take very little time or effort.

Then implement hierarchical directories and locking. Decide whether to do one first and then the other, or both at once, or what. It is vitally important to think through both first before coding either. Many of the trickiest aspects of the locking are connected to handling directories, and vice versa.

Hold off on the performance hacks until after you have this much done, because (as explained below) you will want to run more benchmarks at this point.

Several user-level utilities, mostly in `/sbin` inspect or update SFS filesystems. As you update the SFS code, be sure to keep these utilities up to date; this makes sure they stay useful to you. And to us, when we're grading.

Conversely, you shouldn't have to modify the source code for the basic tools in`/bin`. If, for some reason, you think you do (perhaps because of a different implementation of the OS/161 filesystem than that assumed in these files), contact us first and discuss the issue.

Be sure to test your code. Include test scripts, programs, and/or output with your submission. Various tests are provided in `testbin` to help you. Furthermore, the `mkdir`, `rmdir`, `cat`, `cp`, `mv`, `rm`, `pwd`, and `ls` programs should work with your hierarchical directory implementation. All these programs must be functional when you submit.

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
$ /bin/mv /foo/bar /bar
$ /bin/mv /bar/test /foo/test
$ /bin/cat /foo/test
```

Performance Optimization and Analysis

Just the fine-grained locking changes should increase the throughput of many of the tests. Once you have implemented the locking, and before doing any further performance hacking, collect new performance numbers for the tests you did before starting (`fs1`, `fs3`, and `psort`). Discuss the changes in your performance writeup, and draw conclusions about the use of one big lock at the file system level.

Now, implement the workloads and benchmarks that you will be using to measure your performance optimizations. Collect baseline numbers for these.

Now, implement your performance hacks in SFS. Benchmark the performance improvement they give you, and demonstrate that they provide a significant benefit. (Or that they don't.) Measure the effect of each improvement separately, and then also the effect of both together. Use the workloads and benchmarks you've chosen. Reformat the disk for each test run, to make sure you have a clean slate. Practice good benchmarking and statistical hygiene.

Finally, write up your performance measurements, both for the optimizations you've done and also for the effect of fine-grained locking on the base tests.

Think of your performance writeup as a lab report; it should show your results, discuss what they mean, include graphs where appropriate, point out interesting features in the data, and so forth. Draw whatever conclusions (if any) are appropriate.

Do *keep* all your raw data, but please don't *include* it in the writeup; your writeup should present your *results*, based on good measurement and statistical practices.

The writeup will be a significant portion of your grade for the assignment, so don't treat it as an afterthought.

What to Submit

- Your original design document, as well as a list of any changes made to your design during implementation.
- An exported tar file of your source tree (don't forget to commit the latest version!)
- 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.
- Your performance writeup.