 Prof. Peter Desnoyers

For details on the fs7600 file system format and the FUSE library please see the accompanying document.

Materials

You will be provided with the following files in your team repository:

- Makefile
- homework.c – skeleton code
- misc.c – additional support code
- image.c, blkdev.h – the disk image device and blkdev header file (see docs)
- mktest.c – creates a simple file system image
- test.sh – file system testing framework
- trace1.sh – simple workload generator
- read-img.c – utility for parsing disk image files and displaying them.

Other information:

- homework-3-docs.pdf – documentation on the file system structure, FUSE, debugging hints, etc.
- git-vm-instructions.pdf – instructions on Git and the provided virtual machine image

Additional information may be found in the source file comments.

Usage and other hints:

- Makefile – type 'make' to compile everything
- To create a sample image file (containing 1 file) named "foo.img", or re-write an existing damaged one:
  `./mktest foo.img`
- To run the file system:
  - create a directory: `mkdir dir`
  - start the FUSE file sytem: `./homework -d -s -image foo.img dir`
  the '-d' (debug) option causes it to run in the foreground, and the '-s' specifies single-threaded. These are also the options you would use running under the debugger.
- After running the file system, you have to umount it with the command 'fusermount -u dir'. This is the case even if your program crashed. If you don't do that, attempts to access the mount directory or to re-run your program will fail with "transport endpoint is not connected".
- When testing write functionality, create a clean disk image each time you re-run the program, since it's likely that your (still buggy) code will have corrupted the disk image.

Additional scripts for testing and measurement:

- read-test.sh – read-only tests
- write-test.sh – write tests
- compilebench – benchmark for performance testing

Additional documentation on the test scripts will be posted to Piazza.

Step 1 – file system functionality

Implement the methods defined in homework.c – when all of them are working, you will have a fully functional file system. Some guidelines and hints:

- at startup (i.e. in the init function) read in the superblock to get the file system parameters
- [suggestion] at startup, create copies of the inodes, inode map, and block map in memory. When you update them, be sure to flush the corresponding page back to disk.
- [suggestion] Implement order - getattr first. To test it:
  ```
  ls -d dir
  ls dir/file.A
  ```
  where 'dir' is the mount point – this will generate calls to getattr("/") and getattr("/file.A"). (this is the only file on the image created by mktest.c
- Then readdir:
  ```
  ls dir
  ```
  this will generate a call to readdir("/").
- fs_read, using 'cat dir/file.A' to test.
- Chmod and utime, since they just modify the inode. Invoke them on file.A with 'chmod' and 'touch' commands, then verify with 'ls -l'
- Mkdir and mknod. You can test mknod (without write) with the command 'touch dir/file', where 'file' doesn't exist – it will create a zero-length file.
- Write, then truncate, unlink and rmdir, and rename.

Measurement

Run the compilebench[1] script, which simulates a compile-like workload. (you'll need a disk image of >=50MB to do this) Report (a) the total number of blocks read and written, as well as (b) the total number of disk operations after merging all consecutive read operations and all consecutive write operations. For reproducible results you will probably want to run this on a newly created disk image, and you will find it useful to create a small script or program to merge reads and writes.

Part 2 – path translation caching

Here you will take advantage of the open and opendir calls to cache information about files. In particular, open passes a pointer to a fuse_file_info structure which is passed to all subsequent calls to read and write, and you can save information in the 'fh' field in this structure:

```
int (*open) (char *path, struct fuse_file_info *info);
struct fuse_file_info {
        …
    uint64_t fh;
};
```

If you cache all the inodes in memory, it should be sufficient to put the inode number in the 'fh' field, allowing you to find the corresponding inode without needing to do any directory lookup.

1 Assuming you are in the hw1 directory, and you've mounted your file system in hw1/mnt, you can do this with the command:

```
compilebench/compilebench -s ./compilebench -D ./mnt -i 1 -r 1
```

You can make this code conditional on '(homework_part > 1)', allowing you to re-run your part1 tests after you have implemented this.

Measurement

Again run compilebench, and report (a) total blocks read and written, and (b) disk operations (after merging consecutive reads and consecutive writes).

Part 3 – directory entry cache

Implement a 50-entry cache of mappings from [inode#,name] to inode#. Make sure that rmdir, unlink, and rename flush the corresponding entries from the cache. (on rename you can change the entry or flush it, as you wish) Replacement should be LRU.

Note that there is no requirement for efficient code - we're measuring efficiency of disk access, not CPU usage. So if you e.g. use linear search to find the LRU element that's fine.

Measurement

Same as for parts 1 and 2.

Part 4 – write-back cache

implement a layered block device with 2 LRU pools - one for up to 10 dirty pages, and the other for up to 30 clean pages. Dirty pages are flushed on eviction; clean pages are moved to the dirty pool (evicting a page if full) if they are written to.

Measurement

Same as previously.

Final deliverable:

1. Your code and any test scripts, pushed to the repository.
2. A written document describing the results of the experiments and your conclusions.