CS 241 Honors
Filesystems

Aneesh Durg, Jonathan Wexler, Robert Andrews

University of Illinois Urbana-Champaign

April 18, 2017

ext2

- In CS 241, we cover the basics of the ext2 filesystem, which was introduced in **January 1993**.
- Upper limits on file size and filesystem size have increased over time
- New functionality has been developed to improve performance and reliability of filesystems
- We'll go over some of these new features that found their way into ext3 (2001) or ext4 (2008)

Journaling

- Suppose we want to delete some file `foo.txt` from an ext2 filesystem. Assuming there are no hard links to `foo.txt`, three things need to happen:
  1. Remove the `foo.txt` entry from its directory
  2. Release the inode corresponding to `foo.txt` to the pool of free inodes.
  3. Release the data blocks corresponding to `foo.txt` to the pool of free data blocks.
- What if the system crashes during any of these three operations? We might corrupt `foo.txt` or leak disk space
- We can work around this problem by introducing a journal into our filesystem

Journaling

- Before we go through the process of removing `foo.txt`, write down "remove `foo.txt`" in the journal
- Once we've finished removing `foo.txt`, mark that journal entry as completed.
- If the system crashes, we only have a problem if the crash occurs during the journal write or during the removal of `foo.txt`
  - If the system crashes during the actual removal, we can go back to the journal and replay any action not marked as completed.
  - If the crash occurs while writing to the journal, we don't do anything
- In either case, we recover from the crash without having a corrupted filesystem

Journaling

- Journaling is supported in ext3 and ext4. A journal consists of a journal superblock and a collection of transactions, each of which has three parts:
  - Descriptor block – information about the transaction
  - One or more data or revocation blocks – the content of the transaction
  - Commit block – signals the end of a transaction
- There are three journaling modes supported by ext3 and ext4:
  - *Writeback*: Only metadata is journaled, file contents may or may not be written before transactions are committed
  - *Ordered*: Only metadata is journaled, but the transaction is guaranteed to be committed only after the file contents are written
  - *Journal*: Metadata and file contents are journaled

Disk Fragmentation

- Accessing data on disk is **slow**
- In a perfect world, all our data is contiguous on disk; we don't want to spend time waiting for the drive to seek to another sector
- The ext4 filesystem has a number of features that help reduce fragmentation, including:
  - Extents
  - Allocate-on-flush
  - Multiblock allocator

Extents

- In ext2, data exists on disk at the granularity of blocks
- Moderate-sized files require a single-indirect block, and large files may require a double-indirect or even triple-indirect block
- As the size of the file grows, we spend more and more time chasing pointers to data blocks
- What if only had to follow one pointer to find the entire file?
- This leads to the idea of an *extent*: an extent describes a contiguous region of data blocks that are allocated for the file

Multiblock Allocator

- In ext2 and ext3, data blocks are allocated one at a time
- This poses an issue when allocating data blocks for a large file: how do we avoid fragmentation?
- Example: consider the effect of making 256 calls to `malloc(4)` as opposed to one call to `malloc(1024)`
- ext4 handles this by providing a multiblock allocator that, as the name suggests, can allocate multiple blocks in a single call

Allocate-on-Flush

- Suppose we are continuously appending to a large file in small increments equal in size to one block
- If a block is allocated every time a write finishes, we spend a lot of time allocating disk space and may end up with a highly-fragmented file
- ext4 uses an allocate-on-flush strategy, where the filesystem tries to cache as many writes as possible before allocating space on the disk
- Used in tandem with extents and the multiblock allocator, this leads to less fragmentation on disk and better overall performance of the filesystem in most cases

Review of Mounting

- An individual filesystem forms a tree
- What if we want to access two filesystems at once? (Example: we want to access two partitions of a disk simultaneously)
- We can attach one tree as a subtree of the other
- This is called mounting, and the point at which we mount the subtree is, not surprisingly, called the mountpoint
- When your computer boots, your root partition is mounted to /
- You can think of mounting as assigning a name to the top-level directory of the filesystem you are mounting
- *Everything is a file*. CD's and hard disks are files; you interact with them by mounting their filesystem

Virtual Filesystems

- So far, we have used filesystems as a way to interact with storage devices
- Not all filesystems correspond to physical media
- These *virtual filesystems* can provide useful layers of abstraction between the user and the data they interact with

sshFS

- Replaces `ssh` with a filesystem interface
- Remote files and local files appear exactly the same to the user
- Very useful if you do a lot of combined remote and local work (consider setting this up to use with your CS 241 VM)
- Easy to set up: just run

  ```
  sshfs hostname: mountpoint/
  ```

- More information at https://github.com/libfuse/sshfs

GmailFS

- Idea: read and write emails via a filesystem interface
- Neat idea, but suffers from security flaws and lack of maintenance
- Google will block any attempts you make via GmailFS (Robert tried)
- This would make for a really neat side-project (and Robert would use it)
- You can find the remnants of this project at `https://sr71.net/projects/gmailfs/`

WikipediaFS

- Allows the user to read and edit Wikipedia pages as if they were local files
- Particularly useful if you want to use `vim` or `emacs` to edit Wikipedia instead of the web-based interface
- The project is currently unmaintained, but there is interest in continuing development. More info at `http://wikipediafs.sourceforge.net/`

EncFS

- Provides a layer of abstraction that automatically encrypts files for the user
- As the end user, you only ever see the unencrypted files
- Since individual files are encrypted, EncFS allows for resizable encrypted directories
- You could combine EncFS with sshFS to set up an encrypted directory on a remote machine with almost no hassle
- Reading material: `https://vgough.github.io/encfs/`

GitFS

- Provides a filesystem-like interface to a git repository
- Automatically converts changes into commits
- Can push local changes to upstream repository automatically
- You can find the project at `https://github.com/PressLabs/gitfs`

GVFS

- Like GitFS, but it requires the git server to have some additional endpoints.
- Is intended to be used for developing large projects where git checkout or even git status could take hours!
- Works by only checking out files when they are accessed, and keeping track of which files have been modified.
- Developed by Microsoft to aid development of Windows. (Unfortunately) Written in C#.
- You can find the project at `https://github.com/Microsoft/gvfs/`

FUSE FS (Filesystem in USErspace)

- Examples of virtual filesystems written with FUSE:
  - sshFS
  - GmailFS
  - WikipediaFS
  - EncFS
  - GitFS
- Typically, filesystem code is written at the kernel level
- Non-trivial filesystem code requires both work and permission to run
- FUSE FS attempts to remove some of the work and almost all of the permissions needed to write your own virtual filesystem

How does FUSE work?

- As the developer, you write callback functions to replace usual system calls like `write`, `read`, `open`, etc.
- Your callbacks may use the vanilla system calls, but they don't have to
- If your filesystem represents actual files on a disk, you'll probably want the usual system calls
- If your filesystem doesn't actually operate on files, you may not need whatever the system provides

How does FUSE work?

Let's try opening a file with FUSE:

```
static int my_open(const char *path,
                   struct fuse_file_info *fi)
{
  int res;

  res = open(path, fi->flags);
  if (res == -1)
    return -errno;

  close(res);
  return 0;
}
```

A lot is happening here; let's look at this line by line.

How does FUSE work?

```
static int my_open(const char *path,
                   struct fuse_file_info *fi)
```

Here we define a function, `my_open`, which takes as arguments the path[1] to the file you want to open, and a pointer to a `fuse_file_info` struct. Just know for now that the `flags` field in this struct is used exactly for open flags.

```
    int res;

    res = open(path, fi->flags);
```

We open the file using the system call and the flags FUSE provided to us. Nothing special here.

[1] Paths behave oddly in FUSE, more on this later

How does FUSE work?

```
    if (res == -1)
      return -errno;

    close(res);
    return 0;
}
```

By convention, negative return values indicate failure and non-negative return values indicate success. If there was a problem opening the file, we return `-errno` and FUSE will propogate `errno` back to the calling program. If we successfully opened the file, we return 0 even though `open` returns a file descriptor. FUSE magically takes care of returning the file descriptor for us.

Using your callbacks

Once you've defined all your callback functions, you need to tell FUSE how to find them. FUSE puts no constraints on what you call your functions; `close` can be `open` and `open` can be `close`.

This is one of the easiest things to do in FUSE. You just set up a struct and populate its fields:

```
static struct fuse_operations my_oper = {
    .init    = my_init,
    .open    = my_open,
    .close   = my_close,
    .read    = my_read,
    .write   = my_write,
    etc...
}
```

Starting your filesystem

```
static struct fuse_operations my_oper = {
    .init    = my_init,
    .open    = my_open,
    .close   = my_close,
    .read    = my_read,
    .write   = my_write,
    etc...
}
```

Once you've told FUSE where to find all your functions, you still need an entry point for your code.

This is as simple as it gets[2]:

```
int main(int argc, char *argv[])
{
    umask(0);
    return fuse_main(argc, argv, &my_oper, NULL);
}
```

[2] Modulo the `umask` bit. Again, more on that later.

Starting your filesystem

What if you want to keep some sort of global information about your filesystem?

FUSE provides a way to maintain this private data! Just place all your data into some struct `foo` and then pass the address of `foo` as the fourth argument to `fuse_main`.

```
int main(int argc, char *argv[])
{
  umask(0);
  return fuse_main(argc, argv, &my_oper, &foo);
}
```

You can access your private data from your callbacks by using the call

```
(struct foo_struct *) fuse_get_context()->private_data;
```

Note that you need to properly type cast the data in the `private_data` field.

Mounting and unmounting your filesystem

Let's say you've compiled your filesystem to `my_fs`. You can mount your filesystem by running

```
./my_fs [optional args] mountdir
```

This will mount the root of your filesystem at `mountdir`. You may provide whatever arguments your filesystem needs, but FUSE expects the final command-line argument to be the mount location.

You can unmount your filesystem `my_fs` by running

```
fusermount -u mountdir
```

Quirks of FUSE

FUSE has some nice "features".

- Paths are always given relative to the root of your filesystem. If I am in the top-level directory of my filesystem and call `ls`, your system will see this as `ls /`. To get around this, you must handle relative and absolute paths yourself in your code.[3]
- Permissions are broken. All your code will run with the permissions of whoever ran

  `./my_fs`

  This opens up the potential for users to access parts of the main filesystem that they should not be allowed to access. This is why we set `umask(0)`; without doing so, users may not be able to edit files they create on the filesystem.

[3] You may not need to worry about this depending on what you're writing.

Quirks of FUSE

- FUSE runs multi-threaded by default. While this is faster, reasoning about race conditions in filesystem code can be very difficult, and can make otherwise perfectly reasonable code break. Run your code with the `-s` option to make it run in single-threaded mode.

/dev/null

`/dev/null` is a filesystem that throws away anything that is written to it. Let's look at an implementation of `/dev/null` using FUSE.

/dev/null – Open

```
static int null_open(const char *path,
                     struct fuse_file_info *fi)
{
  (void) fi;

  if(strcmp(path, "/") != 0)
    return -ENOENT;

  return 0;
}
```

/dev/null – Read

```
static int null_read(const char *path, char *buf,
    size_t size, off_t offset, struct fuse_file_info *fi)
{
  (void) buf;
  (void) offset;
  (void) fi;

  if(strcmp(path, "/") != 0)
    return -ENOENT;

  if (offset >= (1ULL << 32))
    return 0;

        memset(buf, 0, size);
        return size;
      }
```

/dev/null – Write

```
static int null_write(const char *path, const char *buf,
    size_t size, off_t offset, struct fuse_file_info *fi)
{
  (void) buf;
  (void) offset;
  (void) fi;

  if(strcmp(path, "/") != 0)
    return -ENOENT;

  return size;
}
```

/dev/null – Truncate

```c
static int null_truncate(const char *path, off_t size)
{
  (void) size;

  if(strcmp(path, "/") != 0)
    return -ENOENT;

  return 0;
}
```

/dev/null – Getattr

```
static int null_getattr(const char *path,
    struct stat *stbuf)
{
  if(strcmp(path, "/") != 0)
    return -ENOENT;

  stbuf->st_mode = S_IFREG | 0644;
  stbuf->st_nlink = 1;
  stbuf->st_uid = getuid();
  stbuf->st_gid = getgid();
  stbuf->st_size = (1ULL << 32); /* 4GB */
  stbuf->st_blocks = 0;
  stbuf->st_atime = stbuf->st_mtime = stbuf->st_ctime =
      time(NULL);

  return 0;
}
```

FUSE References

The example code in these slides was taken from `github.com/libfuse/libfuse/tree/master/example`. Looking through the code here is a great way to learn more about writing code using FUSE.

- `fusexmp.c` contains a vanilla filesystem. All the FUSE calls fall through to the usual system calls.
- `null.c` is an implementation of `/dev/null`.
- `hello.c` is a filesystem that is hard-coded to have a single file, `hello`, which contains the text `Hello World!`

FUSE Project Ideas

- Implement a journaling filesystem
- Build a filesystem interface for the Chara queue
- Build a filesystem interface for Piazza
- Build a filesystem interface for Reddit
- Build a filesystem that handles UIUC registration. (Please)
- Make printers appear as directories that print whatever is copied into them
- Allow users on a shared machine to exchange cryptocurrencies with a filesystem interface