CS 137: File Systems

General Filesystem Design

Promises Made by Disks (etc.)

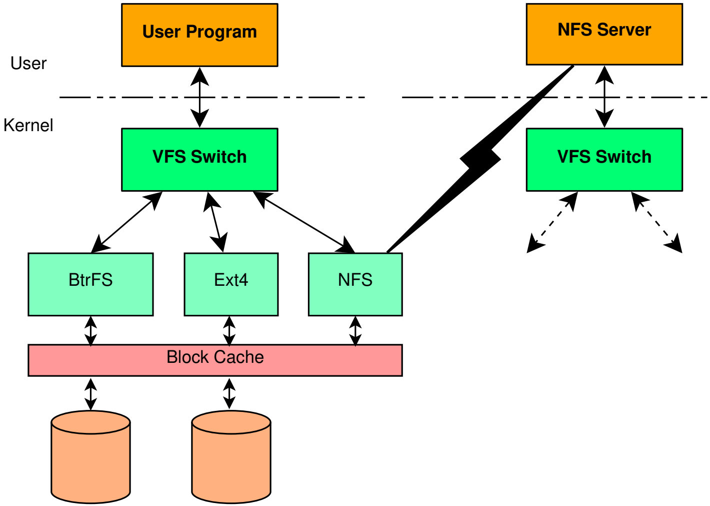
1. I am a linear array of fixed-size blocks[1]
2. You can access any block fairly quickly, regardless of previous accesses[2]
3. You can read or write any block independently of any other
4. Block writes are atomic: all or nothing
5. If you give me bits, I will keep them and give them back later

[1] MRAM and PCRAM promise byte-size blocks—which turns out to cause problems!
[2] But disks and SSDs strongly prefer sequential access

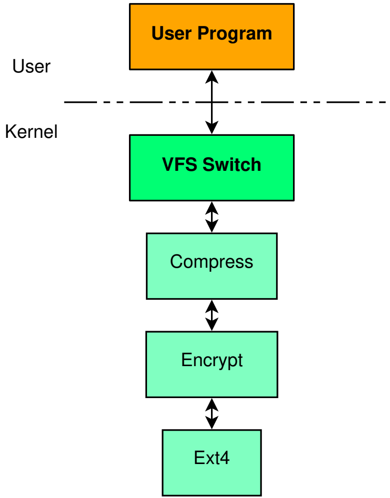
Promises Made by (Most) Filesystems

1. I am a structured collection of data
2. My indexing is more complex and powerful than just numbers
3. I keep track of data as aggregates (e.g., files)
4. Aggregates can be of (somewhat) arbitrary size, and usually you can extend an aggregate
5. You can read and write at the block or even byte level
6. You can find the data you gave me
7. I will give you back the bits you wrote

Linux Virtual File System Layer



VFS Stacking



VFS Interface Functions

The list is long and the interface is complex. Here are a few sample functions:

- `lookup` Find directory entry
- `getattr` Return file's attributes; roughly Unix **`stat`**
- `mkdir` Create a directory
- `create` Create a file (empty)
- `rename` Works on files and directories; normally atomic
- `open` Open a file (possibly creating it)
- `read` Read bytes

VFS Interface Functions

The list is long and the interface is complex. Here are a few sample functions:

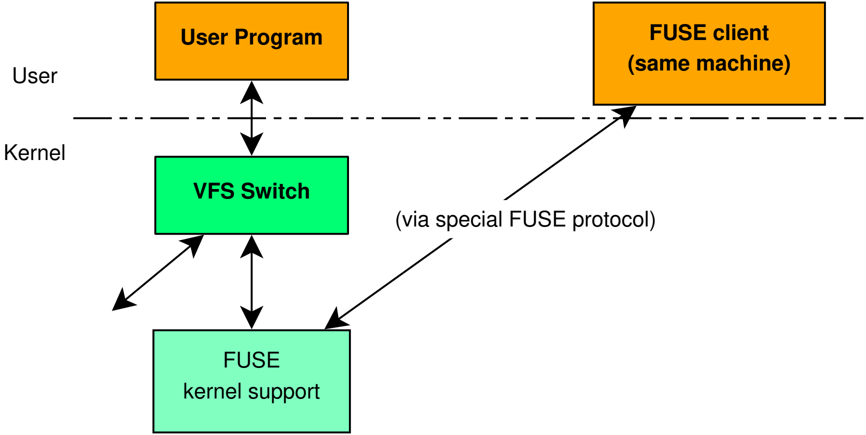
`lookup`	Find directory entry
`getattr`	Return file's attributes; roughly Unix **`stat`**
`mkdir`	Create a directory
`create`	Create a file (empty)
`rename`	Works on files and directories; normally atomic
`open`	Open a file (possibly creating it)
`read`	Read bytes

Important: A particular file system doesn't have to implement all operations!

- E.g., `write` makes no sense on a CD-ROM.

FUSE Structure

FUSE (Filesystem in USEr space) works sort of like NFS:



FUSE Clients

- Must implement a minimum subset of VFS protocol
- What happens internally is hugely flexible
  - Serve requests from internal memory
  - Serve them programmatically (e.g, reads return $\sqrt{\text{writes}}$)
  - Feed them on to some other filesystem, local or remote
  - Implement own filesystem on local device or *inside a local file*
- Samples widely available:

hellofs	Programmatic “hello, world”
sshfs	Remote access via ssh
Yacufs	Makes ogg look like mp3, etc.
Wikipediafs	Wikipedia is a filesystem!
rsbep	ECC for your files
unpackfs	Look inside tar, zip, or gzip archives

The FUSE Interface (1)

FUSE is somewhat like VFS, but can be stateless. Full list of operations (2 slides):

*`getattr`	Get file attributes or properties
*`readdir`	Read directory entries
*`open`	Open file
*`read`	Read bytes
`write`	Write bytes
`mkdir`	Make directory
`rmdir`	Remove directory
`mknod`	Make device node or FIFO
`readlink`	Read symlink destination
`symlink`	Create symbolic link
`link`	Create hard link
`unlink`	Remove link or file
`rename`	Rename directory or file

The FUSE Interface (2)

FUSE operation list, continued:

`truncate`	Delete tail of file (or extend file)
`access`	Check access permissions
`chmod`	Change permissions
`chown`	Change ownership
`utimens`	Update access and modify times
`statfs`	Get filesystem statistics
`release`	Done with file (kind of like close)
`fsync`	Flush file data to stable storage
`getxattr`	Get extended attributes
`setxattr`	Set extended attributes
`listxattr`	List extended attributes
`removexattr`	Remove extended attributes

A Minimal FUSE Filesystem

The “hello, world” example filesystem:

- **`getattr`** If path is “/” or “/hello”, return canned result; else fail
- **`readdir`** Return three canned results: “.”, “..”, “hello”
- **`open`** Fail unless path is “/hello” and open is for read
- **`read`** If path is “/hello” and read is within string, return bytes requested. Otherwise fail.

96 lines of well-formatted (but uncommented) code!

A Minimal FUSE Filesystem

The “hello, world” example filesystem:

- **`getattr`** If path is “/” or “/hello”, return canned result; else fail
- **`readdir`** Return three canned results: “.”, “..”, “hello”
- **`open`** Fail unless path is “/hello” and open is for read
- **`read`** If path is “/hello” and read is within string, return bytes requested. Otherwise fail.

96 lines of well-formatted (but uncommented) code!

Oh, and you can do it in Python or Perl. . .

What Is FUSE Good For?

- Quick filesystem development
- Filesystems that need user-level services
- Extending existing filesystems
- Trying out radical ideas (e.g., SQL interface to filesystem)

What is FUSE Bad At?

- Performance can be worse than in-kernel systems
  - But with work, can come very close
  - Direct access to devices can mitigate
  - Some situations can even outperform kernel
  - Also useful if you need early performance measurements on your cool new idea
- Porting FUSE code to kernel is possible but nontrivial
  - Kernel interface to almost everything (network, disks, memory allocation, etc.) is very different

What a Filesystem Must Provide

- Unix has had big effect on filesystem design
- To succeed today, must support the POSIX interface:
  - Named files (buckets of bytes)
  - Hierarchical directory trees
  - Long file names
  - Ownership and permissions
- Many ways to accomplish this goal
- Today we'll look at single-disk filesystems

Disk Partitioning

- For various bad historical reasons, disks are logically divided into *partitions*
- Table inside cylinder 0 tells OS where boundaries are
- OS makes it look like multiple disks to higher levels
- Early computers had no BIOS, so booting just read block 0 (“boot block”) of disk 0
  - Block 0 had enough code to find rest of kernel & read it in
  - Even today, block 0 is reserved for boot block (Master Boot Record)
  - Original scheme had (small) partition table inside MBR
- Contents of individual partition are up to filesystem

Mounting

A partition contains a filesystem. How to access it?

Windows approach: partition gets special name, file syntax allows specifying partition

- E.g., `C:` (hard drive), `E:` (CD-ROM)

Unix approach: partition is *mounted* over some directory

- "Root" partition is mounted on `/`
- User files might live in `/home`
- OS hides boundaries so can't tell if `/home/geoff/` is on root or separate partition
- Can nest arbitrarily

Basic Filesystem Structure

Any (single-disk) filesystem can be divided into five parts:

1. “Superblock” at well-known location
2. “Free list(s)” to track unallocated space & data structures
3. Directories (folders) that tell where to find other files and directories
4. “Root directory” findable from superblock
5. Metadata for each file or directory:
    - Name
    - How to find contents
    - Possibly other useful information

The Superblock

- Must be findable when FS is first accessed ("mounted")
- Only practical approach: have well-known location (e.g., block 2)
- Everything is up to designer, but usually has:
  - "Magic number" for identification
  - Checksum for validity
  - Size of FS (redundant with partition size, but convenient)
  - Location of root directory
  - Location of metadata (or first metadata)
  - Parameters of disk and of FS structure (e.g., blocks per cylinder, how things are spread across disk)
  - Location of free list(s)
  - Bookkeeping data (e.g., date last mounted or validity-checked)

The Free List

- Usually one of simplest data structures in filesystem
- Popular approaches:
  - Special file holding all free blocks
  - Linked list of blocks
  - "Chunky" list of blocks
  - Bitmap
  - List of extents (contiguous groups of blocks identified by start & length)
  - B-tree or fancier structure

Directories

- Requirement: associate name with *something* usable for locating file's attributes & data
- Simplest approach: array of structures, each of which has name, attributes, pointer(s) to where data is stored
  - Makes directories big ⇒ skipping unwanted entries is expensive
  - Puts "how to find" information far from file & makes it expensive to access
  - Can't support hard links & certain other nice features
- Better: associative map of pairs (name, id-number) where *id-number* tells where to find rest of metadata about file
- From Unix, traditionally referred to as *i-node number*
- Inode (from "index node") can be array or complex structure
- Every directory must also have "." and ".." (or equivalent)

The Root Directory

- This part is easy: on any sensible FS it's identical to any other except for being easily findable
- ".." must be special, since you can't go up from root
  - Exception: if filesystem is mounted under a subdirectory, going up makes sense
  - Unix special-cases that one internally; FS never sees

Metadata About Files and Directories

- Most is just a struct of useful information
  - Under Unix, almost precisely what `stat`(2) returns
  - Type, permissions, owner, group, size in bytes, three timestamps
- Fun part is “how to find the data itself”
  - Desirable properties of a good scheme:
    - Cheap for small files, which are common
    - Supports *very* large files
    - Efficient random access to small and large files
    - Lets OS know when blocks are contiguous (i.e., cheap to read sequentially)
    - Easy to return blocks to free list
  - Can’t be array of block numbers, since inode usually fixed-size
  - Various schemes; for example, could give root of B-tree, or first of linked list of block numbers
  - Can be useful to use extents & try to have sequences of blocks
  - Can use hybrid scheme where first few blocks listed in inode, remainder found elsewhere

Final Thoughts

- Optimal design depends on workload
  - Read vs. write frequency
  - Sequential vs. random access
    - Both within file and across files
    - E.g. are same files repeatedly accessed?
  - File-size distribution
  - Long- vs. short-lived files
  - Proportion of files to directories
  - Directory size
  - Spinning disk vs. SSD
  - ...

Final Thoughts

- Optimal design depends on workload
  - Read vs. write frequency
  - Sequential vs. random access
    - Both within file and across files
    - E.g. are same files repeatedly accessed?
  - File-size distribution
  - Long- vs. short-lived files
  - Proportion of files to directories
  - Directory size
  - Spinning disk vs. SSD
  - . . .
- There is no perfect filesystem!