COMSW 4118-23-1 11/30/2017

HW6 - write an FS that logs GPS info. Modify ext4 to include that location info. Since ext4 is used by Linux, then existing FSs expect ext4 formats on the files and metadata, this has to be backwards compatible. The location info must be persistently stored to disk as metadata, not part of file data. Metadata is typically stored in the inode. Modify the inode, so it stores this location information. Suggested to create a new, GPS aware file inode, that is only instantiated when you use a GPS aware filesystem. Take the existing inode and add GPS info to it (at the end, to be backwards compatible). To distinguish b/n GPS aware and unaware FSs, add some info to the FS metadata in the superblock.

Ext2 FS = derived from BSD FFS (adding groups to the original SSFS)

↓

Ext3 FS = added journaling, log keeps all changes in time order. If system crashes, the journal gives you indication of what was saved last.

- You always write to journal & write to normal FS (inodes, data blocks)
- You use the log to determine if FS is OK:
  - Look at the last few entries of journal & check if FS reflects them
  - If change not reflected, you know when the crash happened and you can choose to discard the entries that are not reflected in the FS.
  - Also, you can correct operations that consisted of several steps but were not executed in full (ex. write to inode & data block, but only one reflected in FS out of two entries in the Log, one for inode write and one for block write).
  - NOTE: the journal entry can be large, fully duplicating the data written, but reducing disk space by a factor of 2
  - BUT: You can only journal the metadata, to only guarantee FS consistency=small journal. (inode writes, updates to blocks in use)

↓

Ext4 FS = added extra optimizations for large files

- What if we introduce a **new FS** w/ only the journal and NOT the FS per se - note that we journal the data in this case, otherwise it will be useless?
- This should reduce seek times, since writes can be sequential and disk head does not move all over the place.
- Now reads are complicated... Where is the inode of a file? You need an **inode map** that gets updated (in memory) with the latest inode data written to each inode, periodically writing the map to disk (to the journal).

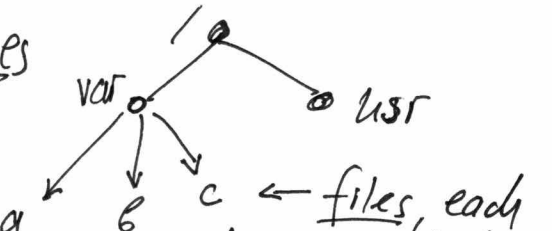
COMSW 4118-23-2 11/30/2017

- The journal would just keep growing, but:
  - If we overwrite blocks, we can get rid of older versions in the log:
    Cleaning = go through journal, find old entries, and compress the log. (delete them, and...)
    (Cleaning adds overhead)
- This scheme is known as **Log Structured FS (LFS)**

- **SSDs** = no moving parts (more reliable) and random access (seeking not a problem).
  - Now, the optimization criteria changes away from optimizing seek time.
  - BUT: you can't just write to an SSD - you have to erase first. (erase is expensive) So, you want to erase big areas
  - ALSO: Wear out, you can only write so many times to a location...
    - Use wear leveling to even out where you write to the SSD (ideally all over)
  - What is the "right" FS for an SSD?
    - S5FS - superblock will wear out, if you only write to one file, it will also wear out
    - LFS - good for wear leveling, also supports erasing big areas when compressing the log. NOTE: Funny thing is LFS was originally designed to minimize seek time
    - some other FS (Ext4) with the disk controller implementing wear leveling and erasing big blocks (transparently)

Verdict is still out.

- **Putting everything together in an FS:** directories



files, each has an inode that identifies where its data (blocks) is.

  - How do we find the inode for a file?
    The same way we find the file, through the directory structure
    - Root location is well known, the underlying directories are just listings of the files (or inodes) within them.
    - Thus, you can search from the root and find the inode for a file.

- **Linux implementation: /fs** ← 2 things here:
  a) actual implementation of FS structure
  b) implementations of individual FSs: ./ext2, ./ext3, ./ext4, ./fat, ./logfs, ./nfs, ./ramfs
  - Linux makes some assumptions about what basic structures you need for an FS...
  - **VFS** = Virtual FS interface to/from the OS
    VFS is part of the OS, but everything else just talks to the VFS.
    Provides a generic, implementation independent view of an FS.

COMSW4118-23-3 11/30/2017

Ex. VFS abstracts open(), read(), write(), etc.
Those are translated to FS-dependent implementation (via function pointers, like in the scheduler code)

▶ FS operations (ex. open(), close()) typically have both independent and dependent parts on the particular FS used.
↑ pretty fixed (implemented in VFS as template)
↑ depends on complexity of the underlying FS.

- On-disk FSs (associated w/ storing data on disk) are usually very FS dependent (specific stuff happens)
- In-memory FSs tend to be simpler/easier (ex. /proc/fs is in memory)

▶ Abstractions in a UNIX FS (that any supported FS has to implement)

- inode - metadata associated w/ a file. more general in VFS than in ext3, etc. In VFS, the inode is generic metadata of a file (even if you use FAT or other filesystem that does not use inodes per se).
- File - the data itself

NOTE: inode has — on-disk inode (stored on disk) } not necessarily the same
— in memory inode ← metadata for file, stored in memory (generally, a superset of on-disk inode)

- superblock - metadata for FS
- dentry - directory entry, an in-memory type of structure.
  - Pathnames are strings: /tmp/foo/bar/file ← typically...
  - those are parsed (slashes are separators) and we need to them identify the inodes for each component - i.e. find tmp's inode, then in it find foo's inode, etc.
  - The system caches those lookups in dentries for performance, those are maintained in-memory.

NOTE: the file descriptor is not part of the file system:
Process has an fd table 

to keep track of the files that are open. It is process-specific and unrelated to the FS.

▶ /include/linux/fs.h has many of the key structures (inodes, superblocks, the file, file operations (struct file_operations) and inode specific operations (struct inode_operations), superblock operations (struct super_operations) ← those are templates for each operation. Look first at those for HWs.

COMSW4118-23-4 11/30/2017

/fs/ext4/ext4.h - has the ext4 FS-specific operations

▶ If you want to look at an "easy" FS to understand:

/fs/debugfs - only has mode.c and file.c ~~has~~ simple implementations of a debugging file system, uses ↖ simple file operation implementations

/fs/libfs - a library for file system writers, with simple operations that can be linked to implement default behavior (ex. simple_dir_mode operations, simple_dir_operations, etc.)

▶ ext4:

/fs/ext4/file.c - note that file operations are more involved

▶ Operations on an FS:

1) mount - whenever the OS hits a mount point, it identifies the type of FS and communicates w/ VFS, informing it what implementation to use (ex. ext4 if the mount point is ext4)

NOTE: mount points are embedded in the overall directory structure

.....