Christoph Hellwig

Data integrity is hard

- Writes in Posix are not durable by default:
  - Required a f(data)sync to be persistent
  - Or the O_SYNC / O_DSYNC options
  - Or and even bigger hammer on macOS
- But even the order of writes is undefined:
  - For example the OS first writes bytes 1024 to 4095 first, then 0 to 1023 for a 4kiB file

Solution 1: The fsync and rename dance

- The only portable way to write or update a file atomically is to write into a new file, fsync it and then rename it into place
  - Due to rename() semantics that can also be used to replace and existing file

```
fd = open(tmpname, O_CREAT | O_WRONLY, 0600);
write(fd, data, datalen);
…
fdatasync(fd);

rename(tmpname, realname);
dirfd = open(dirname(realname), O_RDONLY | O_DIRECTORY);
fsync(dirfd);
```

Issues with fsync and rename

- High overhead:
  - Needs a complete rewrite of the file every time
- Can’t easily be used in combination with mmap()

Solution 2: a journal/log in the application

- Instead of directly overwriting files keep a separate log with intended updates, and only update the main data area after the log commit
- Still needs checksums and sequence numbers to deal with torn writes in the log
- Often used by databases

- Management of the data area is non-trivial
  - Only worth if for complex applications like databases
  - Writes a lot of data twice
  - Duplicates a lot of file system functionality

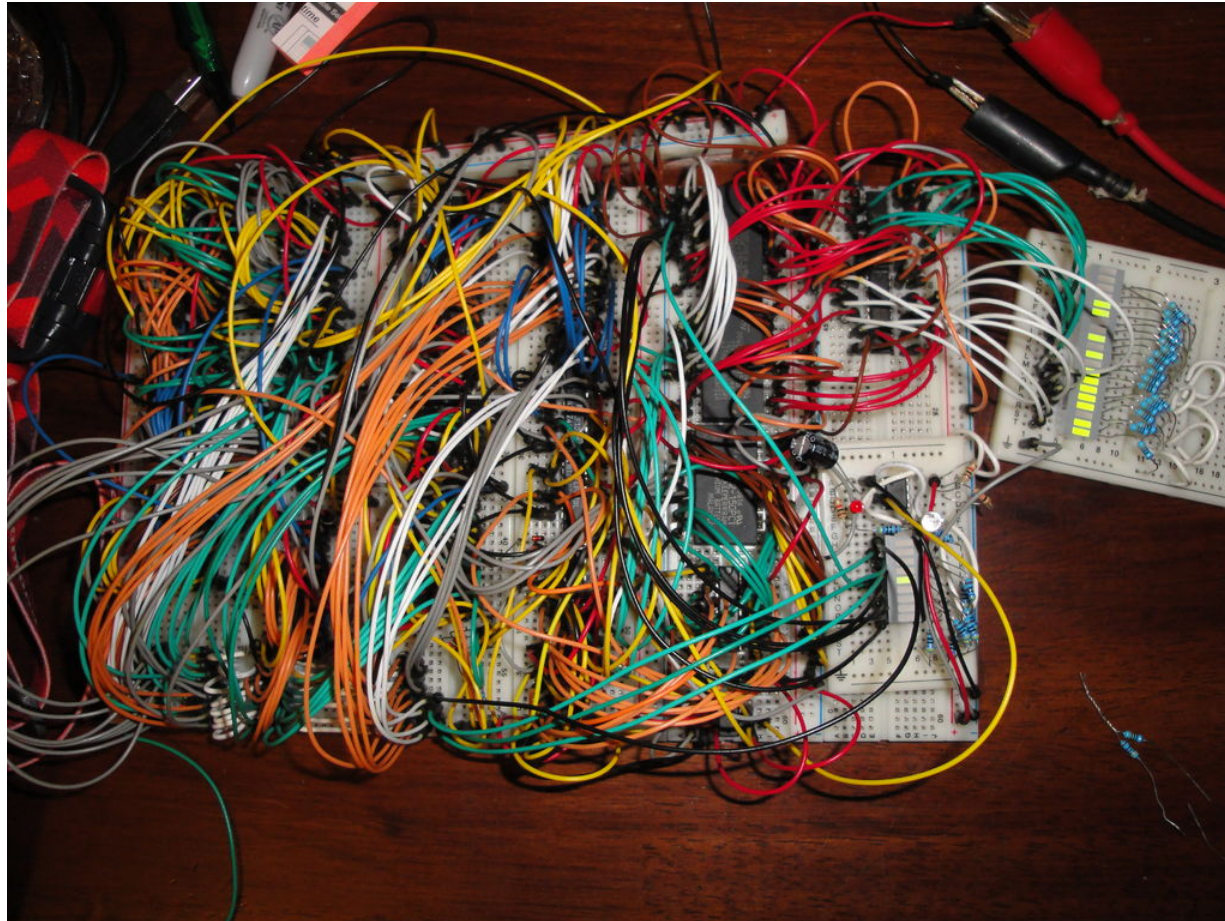
File system help for safely updating files

- The O_ATOMIC flag was first proposed in 2015 by Hewlett-Packard:
  - If a file is opened with the *O_ATOMIC* flag, existing file data on disk will not be updated until fdatasync is called on the file, or msync on a range of a mapped file

```
fd = open(realname, O_ATOMIC | O_WRONLY);
pwrite(fd, data, datalen, someoffset);
…
fdatasync(fd);
```

Everyone loves magic pixie dust..

But how will it work?



Reflinks

- Various Linux file systems (btrfs, xfs, ocfs2) allow to clone files for Copy on Write operations
  - the block map in multiple files reference the same blocks
  - Once a rage gets written to, the blocks are unshared, and new blocks are allocated

Inode 19	Offset 0 Len 256 Disk block 142	Offset 256 Len 20 Disk block 1948	Offset 276 Len 10 Disk block 9562
Inode 31	Offset 0 Len 256 Disk block 142	Offset 256 Len 20 Disk block 1948	Offset 276 Len 10 Disk block 9562

Reflinks in XFS

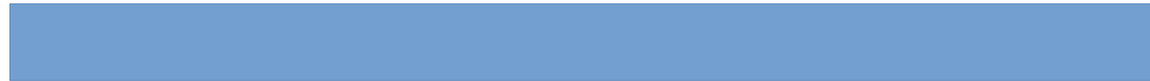
- In XFS blocks allocated for in-progress COW operations are kept in the “COW fork”, which includes an alternative block map
- Once a write I/O to shared block range has completed the block mappings are moved from the COW fork to the normal data fork

O_ATOMIC in XFS

- The O_ATOMIC support makes use of the reflink infrastructure and the COW fork, and thus is very simple (~ 100 lines of code):
  - For a file opened using O_ATOMIC all writes are treated like those needing an out of place write and a new block allocation, and thus are tracked in the COW fork.
  - On I/O completion the newly allocated blocks are not moved to the data fork
  - Only an explicit fdatasync moves the new blocks to the data fork

O_ATOMIC in XFS

Before writing data

Data Fork: 

Cow Fork:

After writing data, before fdatasync:

Data Fork:

Cow Fork:

After fdatasync:

Data Fork:

Cow Fork:

O_ATOMIC performance

- With O_ATOMIC each overwrite becomes similar to an allocating (append or hole fill) write.
  - Depending on the media and workload this can be a 100% or more degradation
  - But compared to rewriting the whole file or logging it still is a lot faster
- And there is another trick waiting to be implemented..

O_ATOMIC for block devices

- NVMe devices support the concept of larger than sector size atomic writes:
  - An “*Atomic Write Unit Power Fail*” value is exposed that tells how blocks will always be updated atomically if written together
- As there is no fdatasync equivalent our model won’t fully work for block devices
- But there still is O_DSYNC

O_ATOMIC for NVMe

- Thus we can claim support for O_ATOMIC when only used together with O_DSYNC for NVMe, including a limitation on the write size
  - For example databases can for example write larger commit blocks atomically
  - Or we can use the block device support internally in the file system to avoid new block allocations for some O_ATOMIC writes.

O_ATOMIC status

- Patches first posted in February 2017
- Still haven't resubmitted them because I'm too busy, but I'll get to it.
  - The biggest stumbling block is automated power fail testing
  - But we now have dm-log-writes in the kernel, and test cases using it in xfstests

Limitations and future work

- The maximum size of all atomic writes until a fdatasync is limited to 2GiB due to transaction subsystems details
- Both reflinks and O_ATOMIC are not supported for the DAX mode that provide direct access to persistent memory
- The XFS support should make use of block device capabilities in NVMe transparently to the application

Questions?

Links

- Ensuring data reaches disk:

  https://lwn.net/Articles/457667/

- Failure-Atomic Updates of Application Data in a Linux File System:

  https://www.usenix.org/system/files/conference/fast15/fast15-paper-verma.pdf

- NVMe 1.3a specification:

  http://nvmexpress.org/wp-content/uploads/NVM-Express-1_3a-20171024_ratified.pdf

- XFS patches:

  https://www.spinics.net/lists/linux-xfs/msg04536.html

  http://git.infradead.org/users/hch/vfs.git/shortlog/refs/heads/O_ATOMIC