High-performance File Systems

Required reading: soft updates.

Overview

A key problem in designing file systems is how to obtain performance on file system operations while providing consistency. With consistency, we mean, that file system invariants are maintained is on disk. These invariants include that if a file is created, it appears in its directory, etc. If the file system data structures are consistent, then it is possible to rebuild the file system to a correct state after a failure.

To ensure consistency of on-disk file system data structures, modifications to the file system must respect certain rules:

- Never point to a structure before it is initialized. An inode must be initialized before a directory entry references it. An block must be initialized before an inode references it.
- Never reuse a structure before nullifying all pointers to it. An inode pointer to a disk block must be reset before the file system can reallocate the disk block.
- Never reset the last point to a live structure before a new pointer is set. When renaming a file, the file system should not remove the old name for an inode until after the new name has been written.

The paper calls these dependencies update dependencies.

xv6 ensures these rules by writing every block synchronously, and by ordering the writes appropriately. With synchronous, we mean that a process waits until the current disk write has been completed before continuing with execution.

- What happens if power fails after 4776 in mknod1? Did we lose the inode for ever? No, we have a separate program (called fsck), which can rebuild the disk structures correctly and can mark the inode on the free list.
- Does the order of writes in mknod1 matter? Say, what if we wrote directory entry first and then wrote the allocated inode to disk? This violates the update rules and it is not a good plan. If a failure happens after the directory write, then on recovery we have an directory pointing to an unallocated inode, which now may be allocated by another process for another file!
- Can we turn the writes (i.e., the ones invoked by iupdate and wdir) into delayed writes without creating problems? No, because the cause might write them back to the disk in an incorrect order. It has no information to decide in what order to write them.

xv6 is a nice example of the tension between consistency and performance. To get consistency, xv6 uses synchronous writes, but these writes are slow, because they perform at the rate of a seek instead of the rate of the maximum data transfer rate. The bandwidth to a disk is reasonable high for large transfer (around 50Mbyte/s), but latency is low, because of the cost of moving the disk arm(s) (the seek latency is about 10msec).

This tension is an implementation-dependent one. The Unix API doesn't require that writes are synchronous. Updates don't have to appear on disk until a sync, fsync, or open with O_SYNC. Thus, in principle, the UNIX API allows delayed writes, which are good for performance:

- Batch many writes together in a big one, written at the disk data rate.
- Absorp writes to the same block.
- Schedule writes to avoid seeks.

Thus the question: how to delay writes and achieve consistency? The paper provides an answer.

This paper

The paper surveys some of the existing techniques and introduces a new to achieve the goal of performance and consistency.

Techniques possible:

- Equip system with NVRAM, and put buffer cache in NVRAM.
- Logging. Often used in UNIX file systems for metadata updates. LFS is an extreme version of this strategy.
- Flusher-enforced ordering. All writes are delayed. This flusher is aware of dependencies between blocks, but doesn't work because circular dependencies need to be broken by writing blocks out.

Soft updates is the solution explored in this paper. It doesn't require NVRAM, and performs as well as the naive strategy of keep all dirty block in main memory. Compared to logging, it is unclear if soft updates is better. The default BSD file systems uses soft updates, but most Linux file systems use logging.

Soft updates is a sophisticated variant of flusher-enforced ordering. Instead of maintaining dependencies on the block-level, it maintains dependencies on file structure level (per inode, per directory, etc.), reducing circular dependencies. Furthermore, it breaks any remaining circular dependencies by undo changes before writing the block and then redoing them to the block after writing.

Pseudocode for create:

```
create (f) {
   allocate inode in block i  (assuming inode is available)
   add i to directory data block d  (assuming d has space)
   mark d has dependent on i, and create undo/redo record
   update directory inode in block di
   mark di has dependent on d
}
```

Pseudocode for the flusher:

```
flushblock (b)
{
  lock b;
  for all dependencies that b is relying on
    "remove" that dependency by undoing the change to b
    mark the dependency as "unrolled"
  write b
}

write_completed (b) {
  remove dependencies that depend on b
```

```
  reapply "unrolled" dependencies that b depended on
  unlock b
}
```

Apply flush algorithm to example:

- A list of two dependencies: directory->inode, inode->directory.
- Lets say syncer picks directory first
- Undo directory->inode changes (i.e., unroll)
- Write directory block
- Remove met dependencies (i.e., remove inode->directory dependency)
- Perform redo operation (i.e., redo)
- Select inode block and write it
- Remove met dependencies (i.e., remove directory->inode dependency)
- Select directory block (it is dirty again!)
- Write it.

An file operation that is important for file-system consistency is rename. Rename conceptually works as follows:

```
rename (from, to)
   unlink (to);
   link (from, to);
   unlink (from);
```

Rename it often used by programs to make a new version of a file the current version. Committing to a new version must happen atomically. Unfortunately, without a transaction-like support atomicity is impossible to guarantee, so a typical file systems provides weaker semantics for rename: if to already exists, an instance of to will always exist, even if the system should crash in the middle of the operation. Does the above implementation of rename guarantee this semantics? (Answer: no).

If rename is implemented as unlink, link, unlink, then it is difficult to guarantee even the weak semantics. Modern UNIXes provide rename as a file system call:

```
   update dir block for to point to from's inode // write block
   update dir block for from to free entry // write block
```

fsck may need to correct refcounts in the inode if the file system fails during rename. for example, a crash after the first write followed by fsck should set refcount to 2, since both from and to are pointing at the inode.

This semantics is sufficient, however, for an application to ensure atomicity. Before the call, there is a from and perhaps a to. If the call is successful, following the call there is only a to. If there is a crash, there may be both a from and a to, in which case the caller knows the previous attempt failed, and must retry. The subtlety is that if you now follow the two links, the "to" name may link to either the old file or the new file. If it links to the new file, that means that there was a crash and you just detected that the rename operation was composite. On the other hand, the retry procedure can be the same for either case (do the rename again), so it isn't necessary to discover how it failed. The function follows the golden rule of recoverability, and it is idempotent, so it lays all the needed groundwork for use as part of a true atomic action.

With soft updates renames becomes:

```
rename (from, to) {
   i = namei(from);
   add "to" directory data block td a reference to inode i
   mark td dependent on block i
   update directory inode "to" tdi
   mark tdi as dependent on td
   remove "from" directory data block fd a reference to inode i
   mark fd as dependent on tdi
   update directory inode in block fdi
   mark fdi as dependent on fd
}
```

No synchronous writes!

What needs to be done on recovery? (Inspect every statement in rename and see what inconsistencies could exist on the disk; e.g., refcnt inode could be too high.) None of these inconsitencies require fixing before the file system can operate; they can be fixed by a background file system repairer.

Paper discussion

Do soft updates perform any useless writes? (A useless write is a write that will be immediately overwritten.) (Answer: yes.) Fix syncer to becareful with what block to start. Fix cache replacement to selecting LRU block with no pendending dependencies.

Can a log-structured file system implement rename better? (Answer: yes, since it can get the refcnts right).

Discuss all graphs.