Crash Recovery

Ordering Disk Writes

Order disk writes to avoid dangling pointers on crash. Allow disk block leaks. On reboot after crash, optionally perform a global filesystem check (fsck) to identify inconsistencies (like space leaks).

- e.g., create/append: initialize child block/file before adding pointer inside parent inode/dir
- e.g., unlink/truncate: remove parent inode/dir's pointer before deallocating child block/file

The fsck program can assume that inconsistencies will be limited to certain types (e.g., space leaks are possible but dangling pointers are not). If it finds an inconsistency (e.g., space leak), it will fix it. But, can we always avoid filesystem inconsistencies? Consider the following example:

```
$ mv a/foo b/foo
```

In this case, the pointer to `foo`'s inode needs to be removed from directory `a`, and added to directory `b`, involving two disk block writes. If we remove the pointer from directory `a` first, we risk losing the file `foo` on a crash. On the other hand, if we add the pointer to directory `b` first, we risk an inconsistency, where the same file `foo` appears in two directories (this may not be allowed depending on the filesystem semantics).

In this case, on a reboot after crash, the `fsck` program will notice the inconsistency, but will not know how to resolve it, i.e., should it remove the pointer from directory `a` or from directory `b`? At these points, the `fsck` program can ask the user to resolve this inconsistency by suggesting a choice.

Let's see how long does `fsck` take. A random read on disk takes about 10 milliseconds. Descending the directory hierarchy might involve a random read per inode. So maybe (n-inodes / 100) seconds? This can be made faster if all inodes (and dir blocks) are read sequentially (in one go), and then descend hierarchy in memory.

fsck takes around 10 minutes per 70GB disk w/ 2 million inodes. Clearly this performance is possible only by reading many inodes sequentially. The time taken by fsck is roughly linear in the size of the disk. As disks grew in size, the cost of running fsck started becoming impractical. We will next study a more modern method of ensuring crash recovery, called logging.

But before that, ordering of disk writes, as we have discussed it so far, would involve synchronous writes to disk. A write-back cache could potentially reorder the disk writes (i.e., writes to buffers in the cache may be in a different order from the actual writes to the disk). This may not be acceptable from a crash-recovery standpoint. On the other hand, a write-through buffer cache is unacceptable from a performance standpoint. (Recall that creating a file and writing a few bytes takes 8 writes, probably 80 ms, so can create only about a dozen small files per second. think about un-tar or rm *)

One solution is to maintain a dependency graph in memory for the buffer cache. The dependency graph indicates an ordering dependency between flushes of different blocks to disk. For example, for the command, `mv a/foo b/foo`, a dependency edge should be drawn from directory `b`'s block to directory `a`'s block, indicating that `b` should be flushed before `a`.

However, consider the following example, where the dependency graph can contain a cycle:

```
$ mv a/foo b/foo
$ mv b/bar a/bar
```

The first command draws a dependency edge from `b` to `a`, while the second command draws a dependency edge from `a` to `b`. At this point, no matter which block is flushed first, there is a chance that one file may get lost (either foo or bar). The solution to this problem is to identify potential cycles in the dependency graph (at the time of making in-memory updates), and avoid them by flushing existing dirty buffers to disk. In this example, before the second command is executed, the OS should flush the blocks for directories `a` and `b` (thus eliminating the dependency edge between them) so that the addition of a new dependency edge does not cause a cycle.

Logging

Goals:

- Atomic system calls w.r.t. crashes
- Fast recovery (no hour-long fsck)
- Speed of write-back cache for normal operations

The basic idea behind logging:

- You want atomicity: all of a system call's writes, or none. Let's call an atomic operation a "transaction".
- Record all writes the sys call *will* do in the log
- then record "done"
- then do the writes
- on crash+recovery:
    - if "done" in log, replay all writes in log
    - if no "done", ignore log

this is a WRITE-AHEAD LOG

Simple logging (as in xv6):
[diagram: buffer cache, FS tree on disk, log on disk]

- FS has a log on disk
- syscall:

```
begin_trans()
  bp = bread()
  bp->data[] = ...
  log_write(bp)
  more writes ...
commit_trans()
```

- begin_trans:
  - need to indicate which group of writes must be atomic!
  - lock -- xv6 allows only one transaction at a time
- log_write:
  - record sector #
  - append buffer content to log
  - leave modified block in buffer cache (but do not write)
- commit_trans():
  - record "done" and sector #s in log
  - do the writes
  - erase "done" from log
- recovery:
  if log says "done":
  - copy blocks from log to real locations on disk

What's wrong with xv6's logging?

- only one transaction at a time
  - two system calls might be modifying different parts of the FS
- log traffic will be huge: every operation is many records
- logs whole blocks even if only a few bytes written
- eager write to log -- slow. As discussed, the log blocks and the commit record needs to be written at `commit_trans()` time. This allows only 4-5 disk writes to get batched at a time. It would be much better if hundreds of disk writes get batched together.
- eager write to real location -- slow. This can be easily fixed by asynchronously applying the logged writes to the FS tree.
- Every block written twice

Fixing xv6 Logging (as done in Linux ext3):

- Basic idea: Batch many (atomic) disk operations into a single transaction. Each operation can involve multiple disk writes. Replace `begin_trans()` and `commit_trans()` with `begin_atomic_op()` and `commit_atomic_op()`. The commit of an atomic operation does not cause a commit of the whole transaction.
- Have one transaction *open* at any time.
- Add all new disk operations to the currently open transaction.
- *Close* the transaction at periodic intervals (e.g., 30 seconds).

Closing a transaction:

- Open a new transaction: all future disk operations that start will get added to the new transaction.
- Mark this transaction as closed. No new disk operations will be admitted to it.
- Wait for ongoing disk operations to complete, i.e., wait for them to call `commit_atomic_op()`.
- Write descriptor and data blocks to the log
- Write the commit record
- Allow the blocks in log to be applied to the FS tree (but not forced). Typically, done asynchronously with many in-flight I/Os to take best advantage of the disk scheduler.

In doing this, we alleviate many of the problems with xv6's logging:

- Many disk operations can simultaneously execute
- Writing the log will involve one (or two) large sequential writes, every few seconds. So log traffic is significantly reduced.
- Even though whole blocks are logged, multiple writes to the same block are absorbed in the buffer cache. For example, only the final value of the block (at the end of the transaction) needs to be logged to the disk.
- The write to the log is still eager, but only at a coarse periodic interval (e.g., few seconds). Also, the write can be completed in one sequential write benefitting from batching many different disk operations.

- The logged blocks are applied to the FS tree asynchronously and lazily, so the disk can efficient schedule these writes for maximum disk throughput.
- Even though every block is still written twice, we no longer pay the price of a disk seek and rotational latency for each write. These writes are fast as they are either a part of a large sequential write (in case of log) or can be efficiently scheduled with many other in-flight I/Os (in case of asynchronous writes to the FS tree).