Logging

ext3 structures:

- in-memory write-back block cache
- in-memory list of blocks to be logged, per-transaction
- on-disk FS
- on-disk circular log file

what's in the ext3 log?

- superblock: starting offset and starting seq #
- descriptor blocks: magic, seq, block #s
- data blocks (as described by descriptor)
- commit blocks: magic, seq

sys call:

- h = start()
- get(h, block #)
    - warn logging system we'll modify cached block.
      added to list of blocks to be logged
    - prevent writing block to disk until after xaction commits
- modify the blocks in the cache
- stop(h)
- guarantee: all or none
- stop() does *not* cause a commit
- notice that it's pretty easy to add log calls to existing code

is log correct if concurrent syscalls?

- e.g. create of "a" and "b" in same directory
- inode lock prevents race when updating directory
- other stuff can be truly concurrent (touches different blocks in cache)
- transaction combines updates of both system calls

what if a crash?

- crash may interrupt writing last xaction to log on disk
- so disk may have a bunch of full xactions, then maybe one partial
- may also have written some of block cache to disk
    - but only for fully committed xactions, not partial last one

how does recovery work?

1. find the start and end of the log
    - log "superblock" at start of log file
    - log superblock has start offset and seq# of first transaction. found start.
    - scan until bad record or not the expected seq #
    - go back to last commit record. found end.
2. Replay all blocks through last complete xaction, in log order

what if block after last valid log block looks like a log descriptor?

- perhaps left over from previous use of log? It will have the wrong sequence number
- perhaps some file data happens to look like a descriptor? It will not have the magic number.

when can ext3 free a transaction's log space?

- after cached blocks have been written to FS on disk
- free == advance log superblock's start pointer/seq

what if not enough free space in log for a syscall?

- suppose we start adding syscall's blocks to T2
- half way through, realize T2 won't fit on disk
- we cannot commit T2, since syscall not done
- but can free T1 to free up log space.

- Log space should be bigger than expected size of a single transaction

ext3 not as immediately durable as xv6

- creat() returns -> maybe data is not on disk! crash will undo it.
- need fsync(fd) to force commit of current transaction, and wait
- would ext3 have good performance if commit after every sys call?
    - would log many more blocks, no absorption
    - 10 ms per syscall, rather than 0 ms

what if syscall B reads uncommited result of syscall A?

```
A: echo hi > x &
B: ls > y &
```

could B commit before A, so that crash would reveal anomaly?

- case 1: both in same xaction -- ok, both or neither
- case 2: A in T1, B in T2 -- ok, A will commit first (assuming T1 < T2)
- case 3: B in T1, A in T2
    - could B see A's modification?
        - This is possible if B called `start()` before A, but called `stop()` after A. In this case, B could see A's modification, and yet be in an earlier transaction.
    - Solution: ext3 must wait for all ops in prev xaction to finish before letting any in next start, so that ops in old xaction don't read modifications of next xaction. Notice that ops in new xaction can start even if the old xaction has not yet committed.

T2 starts while T1 is committing to log on disk

- what if syscall in T2 wants to write block in prev xaction?
- can't be allowed to write buffer that T1 is writing to disk
    - then new syscall's write would be part of T1
    - crash after T1 commit, before T2, would expose update
- T2 gets a separate copy of the block to modify
    - T1 holds onto old copy to write to log

performance?
create 100 small files in a directory

- would take xv6 over 10 seconds (many disk writes per syscall)
- repeated mods to same direntry, inode, bitmap blocks in cache
    - write absorbtion...
- then one commit of a few metadata blocks plus 100 file blocks
- how long to do a commit?
    - seq write of 100*4096 at 50 MB/sec: 10 ms
    - wait for disk to say writes are on disk
    - then write the commit record
    - that wastes one revolution, another 10 ms
    - modern disk interfaces can avoid wasted revolution