```
6.S081/6.828 2019 Lecture 13: Crash Recovery, Logging

Plan
  problem: crash recovery
    crash leads to inconsistent on-disk file system
  solution:
    logging

This is the last xv6 lecture
  next week we'll switch to papers

# Why crash recovery

What is crash recovery?
  you're writing the file system
  then the power fails
  you reboot
  is your file system still useable?

the problem:
  crash during multi-step operation
  may leave FS invariants violated
  after reboot:
    bad: crash again due to corrupt FS
    worse: no crash, but reads/writes incorrect data

examples:
  create:

    $ echo hi > x
    // create trace from last lecture:
    bwrite: block 33 by ialloc   // allocate inode in inode block 33
    bwrite: block 33 by iupdate  // update inode (e.g., set nlink)
    bwrite: block 46 by writei   // write directory entry, adding "x" by dirlink()
    bwrite: block 32 by iupdate  // update directory inode, because inode may have changed

    crash between iupdate and writei
      allocate file inode
      crash: inode not free but not used -- not so bad

    what if the file system first wrote 46 and 32 and then 33
      if crash between 32 and 33, then dirent points to free inode -- disaster!
      crash again, or worse if inode is allocated for something else

  write:
    inode addrs[] and len
    indirect block
    block content
    block free bitmap
    crash: inode refers to free block -- disaster!
    crash: block not free but not used -- not so bad

  unlink:
    block free bitmaps
    free inode
    erase dirent

what can we hope for?
  after rebooting and running recovery code
  1. FS internal invariants maintained
     e.g., no block is both in free list and in a file
  2. all but last few operations preserved on disk
     e.g., data I wrote yesterday are preserved
     but perhaps not data I was writing at time of crash
```

```
      so user might have to check last few operations
   3. no order anomalies
      echo 99 > result ; echo done > status

 correctness and performance often conflict
   disk writes are slow!
   safety => write to disk ASAP
   speed => don't write the disk (batch, write-back cache, sort by track, &c)

 crash recovery is a recurring problem
   arises in all storage systems, e.g. databases
   a lot of work has gone into solutions over the years
   many clever performance/correctness tradeoffs

 # Logging solution

 most popular solution: logging (== journaling)
   goal: atomic system calls w.r.t. crashes
   goal: fast recovery (no hour-long fsck)

 will introduce logging in two steps
   first xv6's log, which only provides safety and fast recovery
   then Linux EXT3, which is also fast in normal operation

 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 on disk (log)
   then record "done" on disk (commit)
   then do the FS disk writes  (install)
   on crash+recovery:
     if "done" in log, replay all writes in log
     if no "done", ignore log
   this is a WRITE-AHEAD LOG

 write-ahead log rule
   install *none* of a transaction's writes to disk
   until *all* writes are in the log on disk,
   and the logged writes are marked committed.

 why the rule?
   once we've installed one write to the on-disk FS,
   we have to do *all* of the transaction's other
   writes -- so the transaction is atomic. we have
   to be prepared for a crash after the first installation
   write, so the other writes must be still available
   after the crash -- in the log.

 logging is magic
   crash recovery of complex mutable data structures is generally hard
   logging can often be layered on existing storage systems
   and it's compatible with high performance (topic for next week)

 # Overview of xv6 logging

 xv6 log representation
   [diagram: buffer cache, in-memory log block # array,
             FS tree on disk, log header and blocks on disk]
   on write add blockno to in-memory array
   keep the data itself in buffer cache (pinned)
   on commit:
     write buffers to the log on disk
     WAIT for disk to complete the writes ("synchronous")
     write the log header sector to disk
       block numbers
```

```
      non-zero "n"
    after commit:
      install (write) the blocks in the log to their home location in FS
      unpin blocks
      write zero to "n" in the log header on disk

  the "n" value in the log header on disk indicates commit
    non-zero == committed, log content valid and is a complete transaction
    zero == not committed, may not be complete, recovery should ignore log
    write of non-zero "n" is the "commit point"

  xv6 disk layout with block numbers
     2: log head
     3: logged blocks
    32: inodes
    45: bitmap
    46: content blocks

  Let's look at an example.
    I've modified bwrite() to print low-level disk writes,
    i.e. the disk writes that occur during transaction commit.

    $ echo a > x

    // create
    bwrite 3    // inode, 33
    bwrite 4    // directory content, 46
    bwrite 5    // directory iode, 32
    bwrite 2    // commit (block #s and n)
    bwrite 33   // install inode for x
    bwrite 46   // install directory content
    bwrite 32   // install dir inode
    bwrite 2    // mark log "empty"
    // write
    bwrite 3
    bwrite 4
    bwrite 5
    bwrite 2
    bwrite 45   // bitmap
    bwrite 595  // a  (note: bzero was absorbed)
    bwrite 33   // inode (file size)
    bwrite 2
    // write
    bwrite 3
    bwrite 4
    bwrite 2
    bwrite 595  // \n
    bwrite 33   // inode
    bwrite 2

  let's look at the 2nd transaction, a write()
    first file.c:syswrite
      compute how many blocks we can write before log is full
      write that many blocks in a transaction

    combined with fs.c:writei
      begin_op()
        bmap() -- can write bitmap, indirect block
          log_write to bzero new block
        bread()
        modify bp->data
        log_write()
          absorbs bzero
        iupdate() -- writes inode
      end_op()
```

```
  begin_op() in log.c:
    need to indicate which group of writes must be atomic!
    need to check if log is being committed
    need to check if our writes will fit in remainder of log
  log_write():
    add sector # to in-memory array
    bpin() will pin block in buffer cache, so that bio.c won't evict it
  end_op():
    if no outstanding operations, commit
  commit():
    copy updated blocks from cache to log on disk
    record sector #s and "done" in on-disk log header
    install writes -- copy from on-disk log to on-disk FS
      bunpin() will unpin from cache --- now it can be evicted
    erase "done" from log

What would have happened if we crashed during a transaction?
  memory is lost, leaving only the disk as of the crash
  kernel calls recover_from_log() during boot, before using FS
    if log header block says "done":
      copy blocks from log to real locations on disk
  what is in the on-disk log?
    crash before commit
    crash during commit -- commit point?
    crash during install_trans()
    crash just after reboot, while in recover_from_log()
  note: it is OK to replay the log more than once!
    as long no other activity intervenes

note xv6 assumes the disk is fail-stop
  it either does the write correctly, or does not do the write
    i.e. perhaps it can't do the last write due to power failure
  thus:
    no partial writes (each sector write is atomic)
    no wild writes
    no decay of sectors (no read errors)
    no read of the wrong sector

# Challenges

challenge: prevent write-back from cache
  a system call can safely update a *cached* block,
    but the block cannot be written to the FS
    until the transaction commits
  tricky because e.g. cache may run out of space,
    and be tempted to evict some entries in order
    to read and cache other data.
  consider create example:
    write dirty inode to log
    write dir block to log
    evict dirty inode
    commit
  xv6 solution:
    ensure buffer cache is big enough
    pin dirty blocks in buffer cache
    after commit, unpin block

challenge: system's call data must fit in log
  xv6 solution:
  - compute an upper bound of number of blocks each system call writes
    set log size >= upper bound
  - break up some system calls into several transactions
    for example, large write()s
    thus: large write()s are not atomic
```

```
      but a cransh will leave a correct prefix of the write

challenge: allowing concurrent system calls
  must allow writes from several calls to be in log
  on commit must write them all
  BUT cannot write data from calls still in a transaction

xv6 solution
  allow no new system calls to start if their data might not fit in log
    must wait for current calls to complete and commit
  when number of in-progress calls falls to zero
    commit
    free up log space
    wake up waiting calls

challenge: a block may be written multiple times in a transaction
  writes affect only the cached block in memory
  so a cached block may reflect multiple uncommitted transactions
  but install only happens when there are no in-progress transactions
    so installed blocks reflect only committed transactions
  good for performance: "write absorbtion"

# Summary

what is good about xv6's log design?
  correctness due to write-ahead log
  good disk throughput: log naturally batches writes
    but data disk blocks are written twice
  concurrency

what's wrong with xv6's logging?
  not very efficient:
    every block is written twice (log and install)
    logs whole blocks even if only a few bytes modified
    writes each log block synchronously
      could write them as a batch and only write head synchronously
    log writes and install writes are eager
      both could be lazy, for more write absorbtion
      but must still write the log first
  trouble with operations that don't fit in the log
    unlink might dirty many blocks while truncating file
```