6.828 2011 Lecture 10: Crash Recovery, Logging

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

the main problem:
 crash during multi-step operation
 leaves FS invariants violated
 can lead to ugly FS corruption

examples:
 create:
  new dirent
  allocate file inode
  crash: dirent points to free inode -- disaster!
  crash: inode not free but not used -- not so bad
 write:
  block content
  inode addrs[] and len
  indirect block
  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
   user might have to check last few operations
 3. no order anomalies
   echo 99 > result ; echo done > status

simplifying assumption: disk is fail-stop
 disk executes the writes FS sends it, and does nothing else
  perhaps doesn't perform the very last write
 thus:
  no wild writes
  no decay of sectors

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

we'll discuss two approaches:

synchronous meta-data update + fsck
logging (xv6 and linux ext3)

synchronous meta-data update
an old approach to crash recovery
simple, slow, incomplete

most problem cases look like dangling references
inode -> free block
dirent -> free inode

idea: always initialize *on disk* before creating reference
implement by doing the initialization write,
waiting for it to complete,
and only then doing the referencing write
"synchronous writes"

example: file creation
what's the right order of synchronous writes?
1. mark inode as allocated
2. create directory entry

example: file deletion
1. erase directory entry
2. erase inode addrs[], mark as free
3. mark blocks free

example: rename() (not in xv6)
between directories, i.e. mv d1/x d2/y
1. create new dirent
2. erase old dirent
or the other way around?
probably safest to create then erase!

what will be true after crash+reboot?
all completed sys calls guaranteed visible on disk
reachable part of FS will be mostly correct
except interrupted rename leaves file in both directories!
blocks and inodes may be unreferenced but not marked free

so: sync meta-data update system needs to check at reboot
to free unreferenced inodes and blocks
descend dir tree from root, remembering all i-numbers and block #s seen
mark everthing else free
probably have to punt on interrupted rename()

many kinds of UNIX used sync writes until 10 years ago

problems with synchronous meta-data update
very slow during normal operation
very slow during recovery

how long would fsck take?
a read from a random place on disk takes about 10 milliseconds

descending the directory hierarchy might involve a random read per inode
so maybe (n-inodes / 100) seconds?
faster if you read all inodes (and dir blocks) sequentially,
then descend hierarchy in memory
my server: fsck takes 10 minutes per 70GB disk w/ 2 million inodes
clearly reading many inodes sequentially, not seeking
still a long time, probably linear in disk size

ordinary performance of sync meta-data update?
creating a file and writing a few bytes takes 8 writes, probably 80 ms
(ialloc, init inode, write dirent, alloc data block, add to inode,
write data, set length in inode, one other mystery write to data)
so can create only about a dozen small files per second!
think about un-tar or rm *

how to get better performance?
RAM is cheap
disk sequential throughput is high, 50 MB/sec
(maybe someday solid state disks will change the landscape)
we'll talk about big memory, then sequential disk throughput

why not use a big write-back disk cache?
no sync meta-data update
operations *only* modify in-memory disk cache (no disk write)
so creat(), unlink(), write() &c return almost immediately
bufs written to disk later
if cache is full, write LRU dirty block
write all dirty blocks every 30 seconds, to limit loss if crash
this is how old Linux EXT2 file system worked

would write-back cache improve performance? why, exactly?
after all, you have to write the disk in the end anyway

what can go wrong w/ write-back cache?
example: unlink() followed by create()
an existing file x with some content, all safely on disk
one user runs unlink(x)
1. delete x's dir entry **
2. put blocks in free bitmap
3. mark x's inode free
another user then runs create(y)
4. allocate a free inode
5. initialize the inode to be in-use and zero-length
6. create y's directory entry **
again, all writes initially just to disk buffer cache
suppose only ** writes forced to disk, then crash
what is the problem?
can fsck detect and fix this?

how can we get both speed and safety?
write only to cache
somehow remember relationships among writes
e.g. don't send #1 to disk w/o #2 and #3

most popular solution: logging (== journaling)
goal: atomic system calls w.r.t. crashes
goal: fast recovery (no hour-long fsck)
goal: speed of write-back cache for normal operations

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

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

xv6's simple logging
[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

let's look at the code:
sys_unlink, sheet 54
begin_trans before ilock to avoid deadlock
then error checks, which need the inode lock
on err, commit empty transaction
writei of dirent
iupdate and iunlockput of file
thus freeing of blocks, erasing of addrs[], freeing inode
commit_trans

  begin_trans, sheet 41
    why only one transaction at a time?
  log_write
  commit_trans
    write_head
    install_trans
  recover_from_log

let's look at today's homework
the log header is at 1014
$ rm README
bwrite sector 1015 -- 29, writei
bwrite sector 1016 -- 2, iupdate
bwrite sector 1017 -- 28, bfree
bwrite sector 1017 -- 28, bfree
bwrite sector 1017 -- 28, bfree
bwrite sector 1017 -- 28, bfree
bwrite sector 1016 -- 2, iupdate
bwrite sector 1016 -- 2, iupdate
bwrite sector 1014 -- log header <-- commit point
bwrite sector 29   -- dir content
bwrite sector 2   -- root and file inodes
bwrite sector 28   -- free bitmap
bwrite sector 1014 -- erase transaction

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
  eager write to real location -- slow
  every block written twice
  trouble with operations that don't fit in the log
    unlink might dirty many blocks while truncating file