

## 6.828 2011 Lecture 14: Linux ext3 crash recovery =

Plan logging for crash recovery xv6: slow and immediately durable ext3: fast but not immediately durable trade-off: performance vs. safety

topic crash recovery crash may interrupt a multi-disk-write operation leave file system in an unuseable state most common solution: logging last lecture: xv6 log -- simple but slow today: Linux ext3 log -- fast theme: speed vs safety speed: don't write the disk safety: write the disk ASAP

example problem: appending to a file two writes: mark block non-free in bitmap add block # to inode addrs[] array we want atomicity: both or neither so we cannot do them one at a time

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

xv6 --

review of xv6 logging [diagram: buffer cache, in-memory log, FS tree on disk, log on disk]

in-memory log: blocks that must be appended to log, in order log "header" block and data blocks each system call is a transaction begin\_op, end\_op syscall writes in buffer cache and appends to in-memory log some opportunity for write absorption at end\_op, each written block appended to on-disk log but NOT yet written to "home" location "write-ahead log" preserve old copy until sure we can commit on commit: write "done" and block #s to header block then write modified blocks to home locations then erase "done" from header blocks recovery: if log says "done": copy blocks from log to real locations on disk

homework Q: what does "cat a" produce after panic? cannot open a Q: how about "ls" panic unknown inode type Problem: dirent is on disk and has an inode# that inode hasn't been written to disk to the right place in fact, on-disk it is marked as free Q: what does "cat a" produce after recovery? empty file. recovery wrote inode to the right place it is now allocated dirent is valid create and write are separate transactions create made it but write didn't modification to avoid reading log in install\_trans() why is buffer 3 still in buffer cache?

what's wrong with xv6's logging? it is slow! all file system operation results in commit if no concurrent file operations synchronous write to on-disk log each write takes one disk rotation time commit takes a another a file create/delete involves around 10 writes thus 100 ms per create/delete -- very slow! tiny update -> whole block write creating a file only dirties a few dozen bytes but produces many kilobytes of log writes synchronous writes to home locations after commit i.e. write-through, not write-back makes poor use of in-memory disk cache

Ext3 --

how can we get both performance and safety? we'd like system calls to proceed at in-memory speeds using write-back disk cache i.e. have typical system call complete w/o actual disk writes

Linux's ext3 design case study of the details required to add logging to a file system Stephen Tweedie 2000 talk transcript "EXT3, Journaling Filesystem" ext3 adds a log to ext2, a previous xv6-like log-less file system has many modes, I'll start with "journaled data" log contains both metadata and file content blocks

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 |super: offset+seq #|...|Descriptor 4+magic|...metadata blocks...|Commit 4+magic| |Descriptor 5+magic|...

how does ext3 get good performance despite logging entire blocks? batches many syscalls per commit defers copying cache block to log until it commits log to disk hopes multiple syscalls modified same block thus many syscalls, but only one copy of block in log much more "write absorption" than xv6

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

ext3 transaction [circle set of cache blocks in this xaction] while "open", adds new syscall handles, and remembers their block #s only one open transaction at a time ext3 commits current transaction every few seconds (or fsync())

committing a transaction to disk open a new transaction, for subsequent syscalls mark transaction as done wait for in-progress syscalls to stop() (maybe it starts writing blocks, then waits, then writes again if needed) write descriptor to log on disk w/ list of block #s write each block from cache to log on disk wait for all log writes to finish append the commit record now cached blocks allowed to go to homes on disk (but not forced)

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 syscall B reads uncommitted 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 must commit first case 3: B in T1, A in T2 could B see A's modification? 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

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 are there now *two* versions of the block in the buffer cache? no, only the new one is in the buffer cache, the old one isn't does old copy need to be written to FS on disk? no: T2 will write it

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 absorption... 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

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 done by e2fsck, a utility program 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 scan until bad record or not the expected seq # go back to last commit record crash during commit -> last transaction ignored during recovery 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? (seq...) perhaps some file data happens to look like a descriptor? (magic #...)

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 block in T1 has been dirtied in cache by T2? can't write that block to FS on disk note ext3 only does copy-on-write while T1 is committing after T1 commit, T2 dirties only block copy in cache so can't free T1 until T2 commits, so block is in log T2's logged block contains T1's changes

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 can we free T1 to free up log space? maybe not, due to previous issue, T2 maybe dirtied a block in T1 deadlock!

solution: reservations syscall pre-declares how many block of log space it might need block the syscall from starting until enough free space may need to commit open transaction, then free older transaction OK since reservations mean all started sys calls can complete + commit

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 (Rethink the Sync addresses this problem)

no checksum in ext3 commit record disks usually have write caches and re-order writes, for performance sometimes hard to turn off (the disk lies) people often leave re-ordering enabled for speed, out of ignorance bad news if disk writes commit block before preceding stuff in principle cannot happen because write barrier before writing commit block in practice, inexpensive disks lie it would be good to detect possible incorrect log if not, then recovery replays "descriptors" with random block #s! and writes them with random content! ext4 has journal checksumming additional advantage: don't need write barrier before writing commit record

ordered vs journaled journaling file content is slow, every data block written twice perhaps not needed to keep FS internally consistent can we just lazily write file content blocks? no: if metadata updated first, crash may leave file pointing to blocks with someone else's data ext3 ordered mode: write content block to disk before committing inode w/ new block # thus won't see stale data if there's a crash most people use ext3 ordered mode

correctness challenges w/ ordered mode: A. rmdir, re-use block for file, ordered write of file, crash before rmdir or write committed now scribbled over the directory block fix: defer free of block until freeing operation forced to log on disk B. rmdir, commit, re-use block in file, ordered file write, commit, crash, replay rmdir file is left w/ directory content e.g. . and .. fix: revoke records, prevent log replay of a given block

final tidbit open a file, then unlink it unlink commits file is open, so unlink removes dir ent but doesn't free blocks crash nothing interesting in log to replay inode and blocks not on free list, also not reachably by any name will never be freed! oops solution: add inode to linked list starting from FS superblock commit that along with remove of dir ent recovery looks at that list, completes deletions

does ext3 fix the xv6 log performance problems? synchronous write to on-disk log -- yes, but 5-second window tiny update -> whole block write -- yes (indirectly) synchronous writes to home locations after commit -- yes

ext3 very successful but: no checksum -- ext4 but: not efficient for applications that use fsync()

Edit By [MaHua](#)