```
6.S081 2020 Lecture 16: Linux ext3 crash recovery

Topic: high-performance file system logging
  via Linux ext3 case study

review: why logging?
  problem example: appending to a file
  FS writes multiple blocks on the disk, e.g.
    mark block non-free in bitmap
    add block # to inode addrs[] array
  it's dangerous for the FS to just do the disk writes
    a crash+reboot after one write would leave FS incorrect
      very damaging if block is both free and in use by a file
    such a file system lacks "crash recoverability"
  logging causes groups of writes to be atomic: all or none
    with respect to crashes
    logging is one way to get crash recoverability

review of xv6 logging
  [ on-disk FS, in-memory cache, on-disk log ]
  each system call is a transaction: begin, writes, end
    initially writes affect only cached blocks in memory
  at end of system call:
    1. write all modified blocks to log on disk
    2. write block #s and "done" to log on disk -- the commit point
    3. install modified blocks in home locations in FS on disk
    4. erase "done" from log on disk
       so logged blocks from next xaction don't appear committed
  if crash (e.g. computer loses power, but disk OK):
    reboot, then run log recovery code
    if commit "done" flag is set on disk
      replay all the logged writes to home locations
      clear the "done" flag
  write-ahead rule:
    don't start home FS writes until all writes committed in log on disk
    otherwise, if crash, recovery can neither un-do nor complete the xaction
  freeing rule:
    don't erase/overwrite log until all home FS writes are done

what's wrong with xv6's logging?
  it is slow!
  each syscall has to wait for all disk I/O-- "synchronous update"
  new syscalls have to wait for commit to finish
    prevents concurrent execution of multiple processes
  every block written twice to disk, once to log, once to FS
  so:
    creating an empty file takes 6 synchronous disk writes
    60 ms on a hard drive
    thus only 10 or 20 creates per second; that's not much!

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"
    http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html
  ext3 adds a log to ext2, a previous 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
    per-transaction info:
      sequence #
      set of block #s to be logged
```

```
      set of outstanding "handles" -- one per syscall
  on disk:
    FS
    circular log

what's in the ext3 log (== journal)?
  log superblock: log offset and seq # of earliest valid transaction
    (this is not the FS superblock; it's a block at start of log file)
  descriptor blocks: magic, seq, block #s
  data blocks (as described by descriptor)
  commit blocks: magic, seq

the log can hold multiple transactions
  |super: offset+seq #| ... |Descriptor 4|...blocks...|Commit 4| |Descriptor 5|

how does ext3 get good performance?
  asynchronous disk update
  batching
  concurrency

asynchronous disk updates
  system calls only modify blocks in cache
    they do not update the disk (and thus don't wait for disk writes)
  why does async update help performance?
    1. system calls return quickly
    2. I/O concurrency; app can do other things while disk writes
    3. allows batching

asynchronous disk updates change crash semantics
  a crash may "forget" the last few seconds of completed system calls
  this is a significant different from xv6 and many non-logging file systems
  careful applications need to be aware (e.g. DB or mail server or editor)
  fsync(fd) forces updates to log (and commit) before returning
    databases, text editors, &c use fsync()

batching
  ext3 commits every few seconds
    so each transaction typically includes many syscalls
  why does batching help performance?
    1. amortize fixed transaction costs over many system calls
       fixed costs: descriptor and commit blocks, seek/rotate, syscall barrier
    2. "write absorbtion"
       many syscalls in the batch may modify the same block (bitmap, dirent)
       thus one disk write for many syscalls' updates
    3. disk scheduling: sort writes for fewer seeks, sequential runs

concurrency, in two forms
  1. multiple system calls may be adding to the current transaction
  2. multiple transactions may be at various stages:
     * one "open" transaction that's accepting new syscalls
     * committing transactions doing their log writes
     * committed transactions writing to home locations in FS on disk
     * old transactions being freed
  why does concurrency help performance?
    many processes can be using the FS at the same time
    new system calls can proceed while old transaction(s) write the disk
    disk is most efficient when you hand it lots of work, give it choice

ext3 sys call code:
  sys_unlink() {
    h = start()
    get(h, block #) ...
    modify the block(s) in the cache
    stop(h)
  }
```

```
  start():
    tells the logging system about a new system call
      can't commit until all start()ed system calls have called stop()
    start() can block this sys call if needed
  get():
    tells logging system we'll modify cached block
      added to list of blocks to be logged
  stop():
    stop() does *not* cause a commit
    transaction can commit iff all included syscalls have called stop()

committing a transaction to disk
  1. block new syscalls
  2. wait for in-progress syscalls to stop()
  3. open a new transaction, unblock new syscalls
  4. write descriptor to on-disk log w/ list of block #s
  5. write modified blocks from cache to on-disk log
  6. wait for all log disk writes to finish
  7. write the commit record to on-disk log
  8. wait for the commit disk write to finish
     this is the commit point
  9. now modified blocks allowed to go to homes on disk (but not forced)

when can ext3 re-use transaction T2's log space?
  log is circular: SB T4 T1 T2 T3
  T2's log space can be freed+reused if:
    all transactions prior to T2 have been freed in the log, and
    T2's blocks have all been written to home locations in FS on disk
  freeing writes on-disk log superblock with
    offset/seq of resulting oldest transaction in log

what if a crash?
  crash causes RAM (and thus cached disk blocks) to be lost
    but disk is assumed to be OK
  crash may interrupt writing last xaction to on-disk log
  so on-disk log may have a bunch of complete xactions, then one partial
  may also have written some of block cache to disk
    but only for fully committed xactions, not partial last one

how ext3 recovery works
  0. reboot with intact disk
  1. look in log "superblock" for offset and seq# of oldest transaction
  2. find the end of the log
     scan until bad magic (missing commit) or unexpected seq # (old entry)
     go back to last valid commit block, to ignore final partial transaction
       in case the crash occuring midway through writing the log
  3. replay all blocks through last valid commit, in log order

what if block after last valid commit block looks like a log descriptor?
  i.e. looks like the start of a new transaction?
  perhaps a descriptor block left over from previous use of log?
    no: seq # will be too low
  perhaps some file data happens to look like a descriptor?
    cannot happen
    ext3 replaces magic number in data blocks in journal with zero
    and sets a flag for that block in the descriptor

what if another crash during log replay?
  after reboot, recovery will replay exactly the same log writes

that was the straightforward part of ext3.
  there are also a bunch of tricky details!

why does ext3 delay start of T2's syscalls until all of T1's syscalls complete?
  i.e. why this:
```

```
    T1: |-syscalls-|
    T2:             |-syscalls-|
  this barrier sacrifices some performance
  example problem that this barrier prevents:
    T1: |-create(x)-|
    T2:              |-unlink(y)-|
                             X crash
  if we allow unlink to start before create finishes
    unlink(y) free y's i-node -- i.e. mark it free in block cache
    create(x) may allocate that same i-node -- i.e. read unlink's write
  if T1 commits, but crash before T2 commits, then
    x will exist but use the same i-node as y!
  we cannot allow older transactions to incorporate updates from newer!
  (note ext3's copy-on-write doesn't fix this: it gives T1
   a copy of all blocks as of the end of T1, but the unlink
   executed before T1 ended.)
  (note we can't give T2 its own copies of the blocks, separate from
   T1's copies, since generally we do want later system calls to
   see the effects of earlier system calls.)

what if a T1 syscall writes a block,
    T1 is committing,
    and syscall in T2 writes same block?
  e.g.
    create d/f in T1
    create d/g in T2
  both of them write d's directory content block
  so
    T1: |--d/f--|-logWrites-|
    T2:         |--d/g--      crash
  if crash after T1 finishes committing,
    but before T2 commits,
    directory entry d/g will exist,
    but point to an i-node that was never initialized (since T2 didn't commit)
  ext3's solution:
    give T1 a private copy of the block cache as it existed when T1 closed
    T1 commits from this snapshot of the cache
    it's efficient using copy-on-write
    the copies allow syscalls in T2 to proceed while T1 is committing

what if there's not enough free log space for a transaction's blocks?
  free oldest transaction(s)
  i.e. install its writes in the on-disk FS

what if so many syscalls have started that the entire log isn't
  big enough to hold their writes, even if all older transactions
  have been freed?
  (unlikely, since each syscall generates few writes compared to log size)
  syscall passes max number of blocks needed to start()
  start() waits if total for current transaction is too high
    and works on freeing old transactions

another (complex) reason for reservations:
  T1 writes block 17
  T1 commits
  T1 finishes writing its log and commit record
     but has not yet written block 17 to home location
  T2 writes block 17 in the cache
     ext3 does *not* make a copy of block 17
     ext3 only copies if needed to write T1's log
  a T2 syscall is executing, does a write for which no log space
  can ext3 install T1's blocks to home locations on disk,
    and free T1's log space
  no: the cache no longer holds T1's version of block 17
    (ext3 could read the block from the log, but it doesn't)
```

```
  reservations detect the potential log space exhaustion,
    prevent that T2 syscall from starting

so far we've been talking about "journaled data" mode
  in which file content blocks are written to log
  as well as meta-data (i-nodes, directory content, free bitmaps)
  so file content is included in atomicity guarantee
    e.g. when appending data to file, new data in newly allocated block,
         updated block bitmap, updated i-node, they are all logged; all or none

logging file content is slow, every data block written twice
  data is usually much bigger than meta-data
  can we entirely omit file content from the log?
  if we did, when would we write file content to the FS?

can we write file content blocks at any time at all?
  no: if metadata committed first, crash may leave i-node pointing
      to blocks from someone else's deleted file, with previous
      file's content!

ext3 "ordered data" mode:
  (don't write file content to the log)
  write file content blocks to disk *before* commiting log
    i.e. update of inode w/ new size and block #
  if no crash, there's no problem -- readers will see the written data
  if crash after data write, but before commit:
    block on disk has new data
    but not visible, since i-node size and block list not updated
    no metadata inconsistencies
      neither i-node nor free bitmap were updated, so blocks still free
  most people use ext3 ordered data mode

correctness challenges w/ ordered data mode:
  A. rmdir, re-use directory content block for write() to some file,
       crash before rmdir or write committed
     after recovery, as if rmdir never happened,
       but directory content block has been overwritten!
     fix: no re-use of freed block until freeing syscall committed
  B. rmdir, commit, re-use block in file, ordered file write, commit,
       crash+recover, replay rmdir
     but no replay of file content write!
     file is left w/ directory content e.g. . and ..
     fix: put "revoke" records into log, prevent log replay of a given block
  note: both problems due to changing the type of a block (content vs meta-data)
    so another solution might be to never change a block's type

Summary of ext3 rules
  The basic write-ahead logging rule:
    Don't write meta-data block to on-disk FS until committed in on-disk log
  Wait for all syscalls in T1 to finish before starting T2
  Don't overwrite a block in buffer cache before it is in the log
  Don't free on-disk log transaction until all blocks have been written to FS
  Ordered mode:
    Write datablock to FS before commit
    Don't reuse free block until freeing syscall is committed
    Don't replay revoked syscalls

another corner case: open fd and unlink
  open a file, then unlink it
  file is open, so unlink removes dir entry but doesn't free inode or blocks
  unlink commits (just removal of dir entry)
  crash
  log doesn't contain writes that free the i-node or blocks
  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

  checksums
    recall: transaction's log blocks must be on disk before writing commit block
      ext3 waits for disk to say "done" before starting commit block write
    risk: 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 the rest of the transaction
    solution: commit block contains checksum of all data blocks
      on recovery: compute checksum of datablocks
          if matches checksum in commit block: install transaction
      if no match: don't install transaction
    ext4 has log checksumming, but ext3 does not

  does ext3 fix the xv6 log performance problems?
    synchronous disk updates -- fixed with async
    most sys calls generate lots of disk writes -- fixed with async batching
    little concurrency -- fixed with batching, and concurrent commit
    every block written twice -- partially fixed with ordered data mode
    ext3 very successful!

  References:
  https://www.usenix.org/system/files/login/articles/04_tso_018-021_final.pdf
```