11/27/23

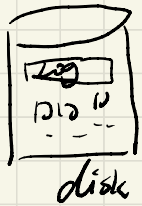
Review

(100) (101) (102)

-> each fs op may cause multiple block updates: inode, data block, data bitmap
-> journaling:
  -> txn abstraction tx_begin, updated 100, updated 101, updated 102, tx_commit

-> logging data & metadata
  -> double the writes: once to the log, once to actual locs
-> logging metadata
  -> only log metadata (inode, bitmaps, directory data)
  -> persist the data blocks first

(ext4 ordered mode)

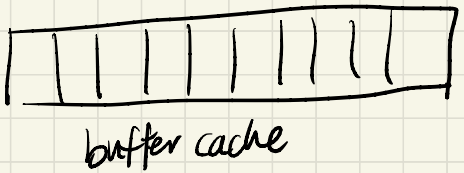


Request reordering problem

tx_begin, updates, tx_commit

-> one txn may need to write a number of blocks
-> concurrent requests can be reordered by disk controller
    -> concurrent = request sent without waiting for a previous request's completion
    -> serial = request sent after a previous request completes
-> write to tx_begin and tx_commit may complete before updates are fully written
-> how do we deal with this?
    -> detect it:
        compute and write checksum of the full txn as part of tx_commit
        on recovery, if txn doesn't have matching checksum, it's not valid, shouldn't be applied
    -> avoid it:
        restrict order of writes! send barrier command before the tx_commit write
        ensures that all previous requests are done before writing tx_commit

Interaction with Buffer/Block Cache
-> cache disk blocks in memory
-> typically a write back cache
-> disk block is cached upon first access
on cache hit, no disk I/O needed
-> when cache is full, run eviction
-> log block are also cached



Interaction with fsync
-> simple semantic: each fs op update persist immediately on disk
-> very slow filesys! disk I/O is slow
-> fsync: lets processes request persistence explicitly
-> lets up other ops update only cached blocks in memory
-> persistence done periodically and through fsync calls
-> on fsync, the journal needs to be persisted!
-> is a per file/directory API
-> calling fsync on a new file doesn't necessarily persist changes to the parent dir
-> to truly persist the file, need to fsync parent dir and fsync new file
-> fsync involves disk I/O and is often slow
-> users want to both reduce the amount of fsync calls and still fsync enough to keep application state consistent

Transaction size

-> so far, we assume one txn per operation

-> accumulates txns in the log, persist log upon fsync

-> what does the log look like if we keep appending 1 byte to a file

logs every version of the inode despite we only care about the latest version at the time of persistence (fsync)

-> group multiple operations into a single txn

-> avoid logging intermediate versions of metadata

-> consolidate updates to shared metadata(e.g. block bitmap) into one

-> ext4 has a single active global txn at a time

-> txn commits on fsync

-> downside: performance interference! unrelated ops are grouped into the same txn, caller of fsync needs to persist all changed data blocks first before persisting the log

```
// Buffer cache.
//
// The buffer cache is a linked list of buf structures holding
// cached copies of disk block contents.  Caching disk blocks
// in memory reduces the number of disk reads and also provides
// a synchronization point for disk blocks used by multiple processes.
//
// Interface:
// * To get a buffer for a particular disk block, call bread.
// * After changing buffer data, call bwrite to write it to disk.
// * When done with the buffer, call brelse.
// * Do not use the buffer after calling brelse.
// * Only one process at a time can use a buffer,
//     so do not keep them longer than necessary.
//
// The implementation uses two state flags internally:
// * B_VALID: the buffer data has been read from the disk.
// * B_DIRTY: the buffer data has been modified
//     and needs to be written to disk.
```

bio.c

```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;
```

```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
```

fs.c

```c
// Read data from inode.
// Returns number of bytes read.
// Caller must hold ip->lock.
int readi(struct inode *ip, char *dst, uint off, uint n) {
  uint tot, m;
  struct buf *bp;

  if (!holdingsleep(&ip->lock))
    panic("not holding lock");

  if (ip->type == T_DEV) {
    if (ip->devid < 0 || ip->devid >= NDEV || !devsw[ip->devid].read)
      return -1;
    return devsw[ip->devid].read(ip, dst, n);
  }

  if (off > ip->size || off + n < off)
    return -1;
  if (off + n > ip->size)
    n = ip->size - off;

  for (tot = 0; tot < n; tot += m, off += m, dst += m) {
    bp = bread(ip->dev, ip->data.startblkno + off / BSIZE);
    m = min(n - tot, BSIZE - off % BSIZE);
    memmove(dst, bp->data + off % BSIZE, m);
    brelse(bp);
  }
  return n;
}
```

asking buffer cache to bring the block into memory

contiguous data layout, find data block for a given offset into the file

```c
// Return a locked buf with the contents of
struct buf *bread(uint dev, uint blockno) {
  num_disk_reads += 1;
  struct buf *b;

  b = bget(dev, blockno);
  if (!(b->flags & B_VALID)) {
    iderw(b);
  }
  return b;
}
```

bio.c

```c
// Look through buffer cache for block on device dev.
// If not found, allocate a buffer.
// In either case, return locked buffer.
static struct buf *bget(uint dev, uint blockno) {
  struct buf *b;

  acquire(&bcache.lock);

  // Is the block already cached?
  for (b = bcache.head.next; b != &bcache.head; b = b->next) {
    if (b->dev == dev && b->blockno == blockno) {
      b->refcnt++;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // Not cached; recycle some unused buffer and clean buffer
  // "clean" because B_DIRTY and not locked means log.c
  // hasn't yet committed the changes to the buffer.
  for (b = bcache.head.prev; b != &bcache.head; b = b->prev) {
    if (b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
      b->dev = dev;
      b->blockno = blockno;
      b->flags = 0;
      b->refcnt = 1;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }
  panic("bget: no buffers");
}
```