CS 134
Operating Systems

March 25, 2019

Crash Recovery & Logging

Final project

- Choose project 6 (JOS networking) or JOS-related final project of your choice
- Some project ideas are in the Lab 7 writeup
  - Piazza Discussion Due, March 28, 2019
    - Find partners (team of up to 3), share ideas
  - Proposals Due, April 4, 2019
    - Will say yes or no (level of difficulty, relevance to OS)
  - Code repository (including brief writeup). Due, May 2, 2019
  - In-person Check-off, May 3 or 6, 2019

Crash recovery

- Problem: crash can lead to inconsistent file system
- Solution 1: file system check on boot
- Solution 2: logging

What is crash recovery?

- You're writing to the file system
- Then, the power fails
- You reboot
- Is your file system still usable?

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
  - new dirent
  - allocate file inode
  - crash: dirent points to free inode—disaster
    - crash again, or worse if inode is allocated for something else
  - crash: inode not free but not used—not so bad

Examples

- write
  - inode addr[] 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

Examples

- unlink
  - block free bitmaps
  - free inode
  - erase dirent
  - crash: inode refers to free block—disaster
  - crash: dirent refers to free inode—disaster

What can we hope for?

- After rebooting and running recovery code:
  1. FS internal invariants maintained
     - For example, no block is in both the free list and in a file
  2. All but the last few operations are preserved on disk
     - For example, data I wrote yesterday is preserved, but not necessarily data I was writing at the time of the crash
     - User might have to check the 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 to disk
  - Batch
  - Write-back cache
  - Sort by track
  - etc.

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

- Most popular solution
- *aka* journaling
- Goal: atomic system calls w.r.t. crashes
- Goal: fast recovery (no hour-long `fsck`)

We'll look at logging in two steps

1. In xv6, which only provides safety and fast recovery
2. Then, in Linux's EXT3, which is also fast in normal operation

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 a system call *will* do in the log on a disk (log)
- Then, record "done" in the log (commit)
- Then, do the FS disk writes (install)
- On crash+recovery:
  - If "done" is in the log, replay all the writes in the log.
  - Else, ignore log
- This is a *write-ahead log*

Write-ahead log rule

- Write *none* of a transaction's writes to the FS
  - Until *all* writes are in the log
  - And, the logged writes are *committed*

Why the rule?

- Once we’ve installed one write to the on-disk FS
  - We have to do *all* the other writes in the transaction (so the transaction is atomic)
  - To be prepared for a crash after the first installation write
    - The other writes must be available for replay
      - In the log

Logging is magic

- Crash recovery of complex mutable data structures is generally hard
- Logging can often be layered on top of existing storage systems
- And, it’s compatible with high performance

Challenge: prevent writeback from cache

- A system call can safely update a cached block
  - But, the block cannot be written to the FS until the transaction completes
- Tricky, because, for example, cache may run out of space and may be tempted to evict some entries in order to read and cache other data

Challenge: prevent writeback from cache

- `create` example
  - Write dirty inode to log
  - Write dir block to log
  - Evict dirty inode
  - Commit
- Solution:
  - Ensure buffer cache is big enough
  - Pin dirty blocks in the buffer cache
  - Afer commit, unpin blocks

xv6 log representation

- 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 to the disk
    - block numbers
    - non-zero “n”
  - After commit:
    - Install (write) the blocks in the log to their home location in the FS
    - Write zero to “n” in the log header

The “n” value in the log header on disk indicates commit

- zero == not committed—may not be complete: recovery should ignore log
- non-zero == committed—log content is valid and is a complete transaction

- The write of the non-zero “n” is the commit point

Challenge: system-call's writes must fit in log

- Compute an upper bound on the number of blocks each system call writes
  - set log size ≥ upper bound
- Break up some system calls into several transactions
  - Large `write()`s
  - Thus, large `write()`s are not atomic
    - But, a crash will leave a valid prefix of the large write

Challenge: allowing concurrent system calls

- Must allow writes from several system calls to be in the 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 into the 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 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 absorption*

xv6 disk layout with block numbers

Block num	Usage
0	unused (usually boot block)
1	super block
2	log for transactions
32	array of inodes, packed into blocks
58	Block in-use bitmap (0=free, 1=used)
59	file/dir content blocks
…	…

An example: `echo a > x`

Create x

Block num written	Explanation
3	inode: 35
4	directory content: 63
2	commit (block #s and n)
35	install inode
63	Install directory content
2	mark log “empty

Write ‘a’

Block num written	Explanation
3	bitmap: 58
4	file content: 533
5	inode: 35
2	commit (block #s and n)
58	bitmap
533	“a”
35	inode (file size)
2	mark log “empty

Write ‘\n’

Block num written	Explanation
3	file content: 533
4	inode: 35
2	commit (block #s and n)
533	“a\n”
35	inode (file size)
2	mark log “empty

Deep dive into second transaction

```
filewrite(struct file *f, char *addr, int n)
{
  …
  if(f->type == FD_INODE){
    // write a few blocks at a time to avoid exceeding
    // the maximum log transaction size, including
    // i-node, indirect block, allocation blocks,
    // and 2 blocks of slop for non-aligned writes.
    // this really belongs lower down, since writei()
    // might be writing a device like the console.
    int max = ((MAXOPBLOCKS-1-1-2) / 2) * 512;
    int i = 0;
    while(i < n){
      int n1 = n - i;
      if(n1 > max)
        n1 = max;

      begin_op();
      ilock(f->ip);
      if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
        f->off += r;
      iunlock(f->ip);
      end_op();
   …
}
```

Write ‘a’

Block num written	Explanation
3	bitmap: 58
4	file content: 533
5	inode: 35
2	commit (block #s and n)
58	bitmap
533	“a”
35	inode (file size)
2	mark log “empty

Deep dive into second transaction

Can write bitmap, indirect block

```
writei(struct inode *ip, char *src, uint off, uint n)
{
  …
  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(bp->data + off%BSIZE, src, m);
    log_write(bp);
    brelse(bp);
  }

  if(n > 0 && off > ip->size){
    ip->size = off;
    iupdate(ip);
  }
  return n;
}
```

Can write bitmap, indirect block

Deep dive into second transaction

- Need to indicate which groups 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

```
void begin_op(void)
{
  acquire(&log.lock);
  while(1){
    if(log.committing){
      sleep(&log, &log.lock);
    } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
      // this op might exhaust log space; wait for commit.
      sleep(&log, &log.lock);
    } else {
      log.outstanding += 1;
      release(&log.lock);
      break;
    }
  }
}
```

Deep dive into second transaction

```
void log_write(struct buf *b)
{
  int i;

  if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
    panic("too big a transaction");
  if (log.outstanding < 1)
    panic("log_write outside of trans");

  acquire(&log.lock);
  for (i = 0; i < log.lh.n; i++) {
    if (log.lh.block[i] == b->blockno)   // log absorbtion
      break;
  }
  log.lh.block[i] = b->blockno;
  if (i == log.lh.n)
    log.lh.n++;
  b->flags |= B_DIRTY; // prevent eviction
  release(&log.lock);
}
```

Deep dive into second transaction

- If no outstanding transactions, commit

```
void end_op(void)
{
  acquire(&log.lock);
  log.outstanding -= 1;
  if(log.outstanding == 0){
    do_commit = 1;
    log.committing = 1;
  } else {
    // begin_op() may be waiting for log space,
    // and decrementing log.outstanding has decreased
    // the amount of reserved space.
    wakeup(&log);
  }
  release(&log.lock);
  if(do_commit){
    …
    commit();
    acquire(&log.lock);
    log.committing = 0;
    wakeup(&log);
    release(&log.lock);
  }
}
```

Deep dive into second transaction

- Copy updated blocks from cache to disk log
- Record sector #s and “done” to disk
- Install writes—copy from on-disk log to on-disk FS
  - ide.c will clear B_DIRTY for block written—now it can be evicted
- Erase “done” from log

```
static void
commit()
{
  if (log.lh.n > 0) {
    write_log();     // Write modified blocks from cache to log
    write_head();    // Write header to disk -- the real commit
    install_trans(); // Now install writes to home locations
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
  }
}
```

What would happen if we crash during a transaction?

- Memory is lost—only disk at time of crash
- Kernel calls `recover_from_log()` during boot (before using FS)
  - If log headers say “done”:
    - copy blocks from log to real location 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()`
- Replaying the log is *idempotent*
  - as long as no other FS activity intervenes

xv6 assumes disk is fail-safe

- Atomic: Either the write occurs correctly, or the write doesn’t occur
  - No partial writes
- No wild writes
- No decay of sectors (no read errors)
- No read of the wrong sector

What is good about xv6's log design?

- Correctness: due to write-ahead log
- Good disk throughput: log naturally batches writes
  - But, disk blocks are written twice
- Concurrency

What is bad about xv6's log design?

- Not very efficient
  - Every block is written twice
  - Logs whole blocks even if only a few bytes are modified
  - Writes each log block synchronously
    - Could write them as a batch and only write head synchronously
  - Trouble with operations that don't fit in the log
    - unlink might dirty many blocks while truncating file