11/21 Crash Consistency

Persistent

-> data bitmap, inode bitmap, inodes & data blocks

-> kernel / sw needs things in memory to operate on.

-> path traversal

① read inode

② use inode content to find data block loc.

③ read data block.

reading data block or inodes are all reading disk blocks

What's the mechanism for reading / writing disk blocks?

disk inode

power of 2, ≤ 512 bytes

```c
// On-disk inode structure
struct dinode {
  short type;           // File type
  short devid;          // Device number (T_DEV only)
  uint size;            // Size of file (bytes)
  struct extent data;   // Data blocks of file on disk
  char pad[46];         // So disk inodes fit contiguosly in a block
};
```

cached inode

```c
// in-memory copy of an inode
struct inode {
  uint dev;   // Device number
  uint inum;  // Inode number
  int ref;    // Reference count
  int valid;  // Flag for if node is valid
  struct sleeplock lock;

  short type; // copy of disk inode
  short devid;
  uint size;
  struct extent data;
};
```

inode cache

```c
struct {
  struct spinlock lock;
  struct inode inode[NINODE];
  struct inode inodefile;
} icache;
```

```c
// Reads the dinode with the passed inum from the inode file.
// Threadsafe, will acquire sleeplock on inodefile inode if not held.
static void read_dinode(uint inum, struct dinode *dip) {
  int holding_inodefile_lock = holdingsleep(&icache.inodefile.lock);
  if (!holding_inodefile_lock)
    locki(&icache.inodefile);

  readi(&icache.inodefile, (char *)dip, INODEOFF(inum), sizeof(*dip));

  if (!holding_inodefile_lock)
    unlocki(&icache.inodefile);

}
```

```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;
}
```

read blocks using buffer cache ←

```c
// Return a locked buf with the contents of the indicated block.
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;
}
```

actual disk read (blocking)

1 request at a time

```c
// Sync buf with disk.
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
// Else if B_VALID is not set, read buf from disk, set B_VALID.
void iderw(struct buf *b) {
  struct buf **pp;

  if (!holdingsleep(&b->lock))
    panic("iderw: buf not locked");
  if ((b->flags & (B_VALID | B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if (b->dev != 0 && !havedisk1)
    panic("iderw: ide disk 1 not present");

  acquire(&idelock); // DOC:acquire-lock

  // Append b to idequeue.
  b->qnext = 0;
  for (pp = &idequeue; *pp; pp = &(*pp)->qnext) // DOC:insert-queue
    ;
  *pp = b;

  // Start disk if necessary.
  if (idequeue == b)
    idestart(b);

  // Wait for request to finish.
  while ((b->flags & (B_VALID | B_DIRTY)) != B_VALID) {
    sleep(b, &idelock);
  }

  release(&idelock);
}
```

```c
// Write b's contents to disk.  Must be locked.
void bwrite(struct buf *b) {
  if (crashn_enable) {
    crashn--;
    if (crashn < 0)
      reboot();
  }
  if (!holdingsleep(&b->lock))
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```

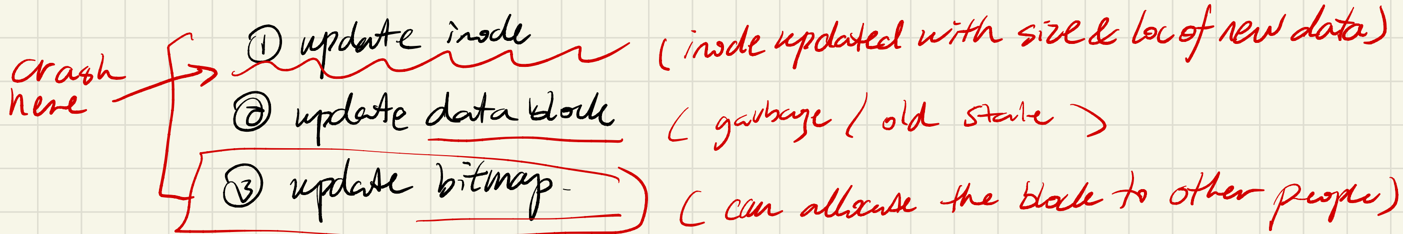
synchronous write back

Buffer Cache operation

```c
// Release a locked buffer.
// Move to the head of the MRU list.
void brelse(struct buf *b) {
  if (!holdingsleep(&b->lock))
    panic("brelse");

  releasesleep(&b->lock);

  acquire(&bcache.lock);
  b->refcnt--;
  if (b->refcnt == 0) {
    // no one is waiting for it.
    b->next->prev = b->prev;
    b->prev->next = b->next;
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }

  release(&bcache.lock);
}
```

FS Data Structures

- → bitmaps (inode, data), inode array, data blocks
- → changes when doing an overwrite: data block, inode (depends on metadata)
- → changes when doing an append
  1. inode: size, loc of new data
  2. data block
  3. data bitmap
- → computer may crash at any point, only promise is that write to a single sector either happen or not happen



Crash Consistency!

Solution 1 : The Filesys Checker (fsck)

- → scan through all metadata, check & resolve inconsistency
- → find out data blocks usage via inodes
- → check bad values

Solution 2 : Journaling / Write ahead Logging

- → group updates as a unit (transaction)
- → API: begin, op, op, op... end/commit
- → reserve log space on disk, write transaction to log, persist & then apply the actual disk updates.

begin
blk A
blk B
end

☆ Recall that I/O requests might be reordered.

- → what if the end message is written first?
- → sol 1: write end after all other log blocks are written
- → sol 2: checksum the full txn, can detect if only part of the txn is written