# Lecture 16: FileSystems II

Professor G. Sandoval

Some slides derived from : Tanenbaum/Bo, and Brendan Dolan-Gavitt
Thanks !!

# Today

- File System Implementation (Cont)
  - Log Structured Systems
  - Journaling
- XV6 File System Layers
  - Buffer Cache Layer
  - Logging Layer
  - Inode Layer
  - Directory Layer
  - PathName Layer
  - File Descriptor

# Write Performance Issues

- Most disks now have fast caches that make reads fast
- Writing is still slow – and worse because typical workload consists of many small writes
    - To write a file, you need to write to the directory entry, i-node, and finally file data
- Overhead dominates: the actual write may take only 50 microseconds, but requires:
    - 10ms to seek (move the read arm into the right position)
    - 4ms for disk platters to rotate into the right position

# Log-Structured Filesystem

- To solve this, Rosenblum and Ousterhout (1992) created the *log-structured filesystem*

- Basic idea - structure the disk as **one big log**

- Periodically data buffered in memory collected into a single segment and written to the disk.

# Log-Structured Filesystem

- Buffer writes in memory, then periodically flush them out in one contiguous segment at the end of the log

- At the start of the segment, place information about where to find i-nodes and file data within the current log segment

- Finally, maintain a map that says which log segment the i-nodes for files/directories can be found

# Log-Structured Filesystem

- If we overwrite data in a file, or if we delete it, we will waste space (and eventually run out of space)

- To solve this, a *cleaner thread* constantly runs, removing unused entries from the back of the log and placing old but still-in-use entries at the front

# Log-Structured Filesystem

- Structuring as a log also provides *crash recovery*
- If the OS crashes or we lose power in a traditional filesystem, we may leave things in an inconsistent state
    - For example: wrote file data, but didn't update the directory entry
- LFS solves this by keeping a *checkpoint*, which tracks what the most recent consistent filesystem state is
- After a crash, we can either just revert to the last checkpoint, or revert and then replay as many log entries as possible (while keeping the filesystem consistent

# Today

- File System Implementation (Cont)
  - Log Structured Systems
  - Journaling
- XV6 File System Layers
  - Buffer Cache Layer
  - Logging Layer
  - Inode Layer
  - Directory Layer
  - PathName Layer
  - File Descriptor

# Journaling Filesystems

- LFS is a major change to how filesystems work and hasn't been widely adopted

- However, one of its key ideas – using a log to provide recovery in the case of a crash – has been incorporated into modern filesystems

- These are called *journaling filesystems*

    - NTFS, HFS+, and ext3 all support journaling

    - ReiserFS was the first Linux FS to support journaling

# Crashes and Inconsistency

- Crashes can lead to inconsistent filesystem states
- Consider operations to delete a file
  - Remove the file from the directory entry
  - Mark the i-node as free
  - Mark the file data blocks as free
- What happens if we do some of these and not the others because of a system crash?

# Journaling

- Instead of just doing these three writes operations, first write a log entry to the *journal* saying what operations you're about to do
- Now if we crash, we can look at the journal and re-run the operations listed
- Once we're done, we can delete the log entry
- At worst, when the system crashes, we might end up repeating some operations

# Idempotence

- Because the operations listed in the journal log entry could be carried out more than once, they must be *idempotent* (doing it twice is the same as doing it once)
- For example, marking a block as free in a bitmap is idempotent
- Adding a block to a list of free blocks is *not* idempotent
- But we can make it idempotent by adding a check to make sure it's not already in the list

# In the real World

- Windows NT (NTFS) has been doing journaling since 1993 and it’s structure is rarely corrupted by system crashes.
- Linux:
    - First File System to do it was ReiserFS but not very popular
    - Ext3 is more popular

# Today

- File System Implementation (Cont)
    - Log Structured Systems
    - Journaling
- XV6 File System Layers
    - Buffer Cache Layer
    - Logging Layer
    - Inode Layer
    - Directory Layer
    - PathName Layer
    - File Descriptor

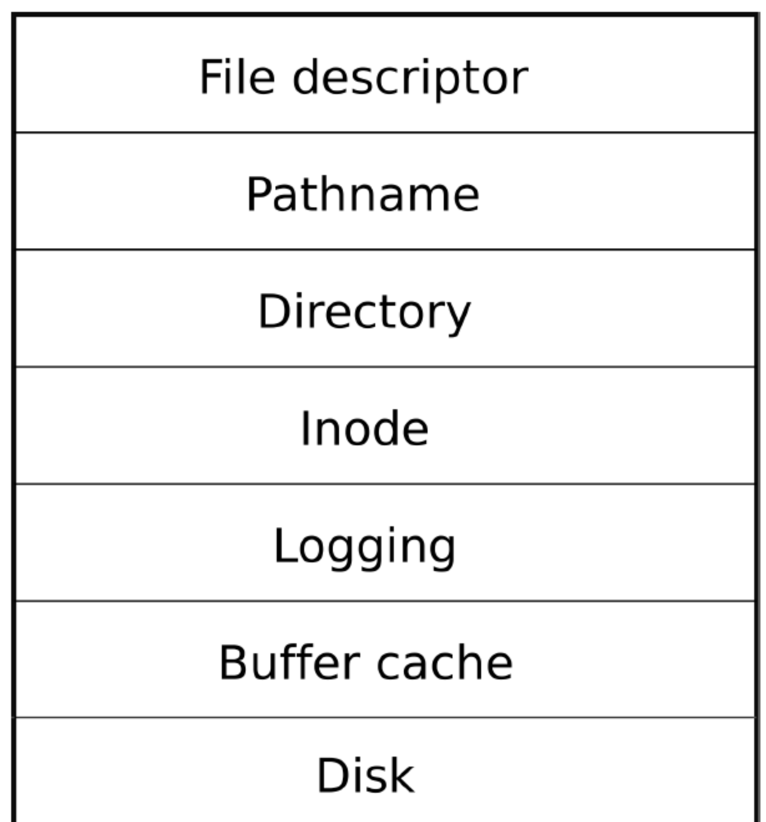
# xv6 FS Design

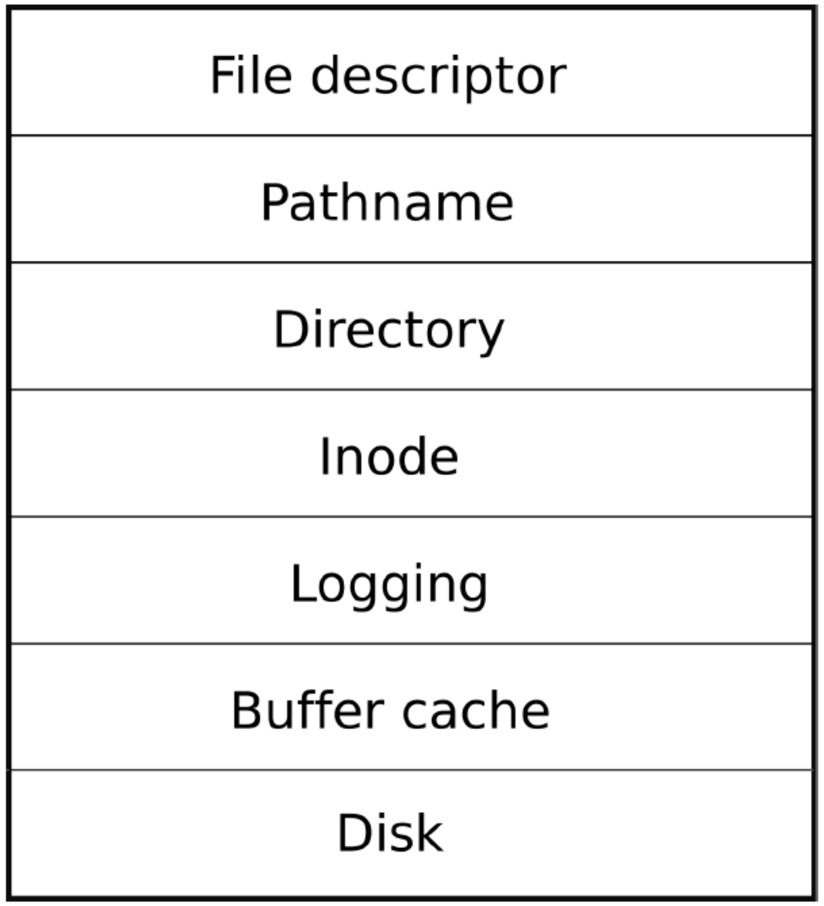
- On-disk data structures to represent tree of files & directories
- Crash recovery (never end up with inconsistent fs)
- Concurrent access by multiple files
- Provide caching, since disks are slow

# xv6 FS

- The filesystem is one of the most complex parts of xv6
- Organized into seven layers:

**We've seen these already**

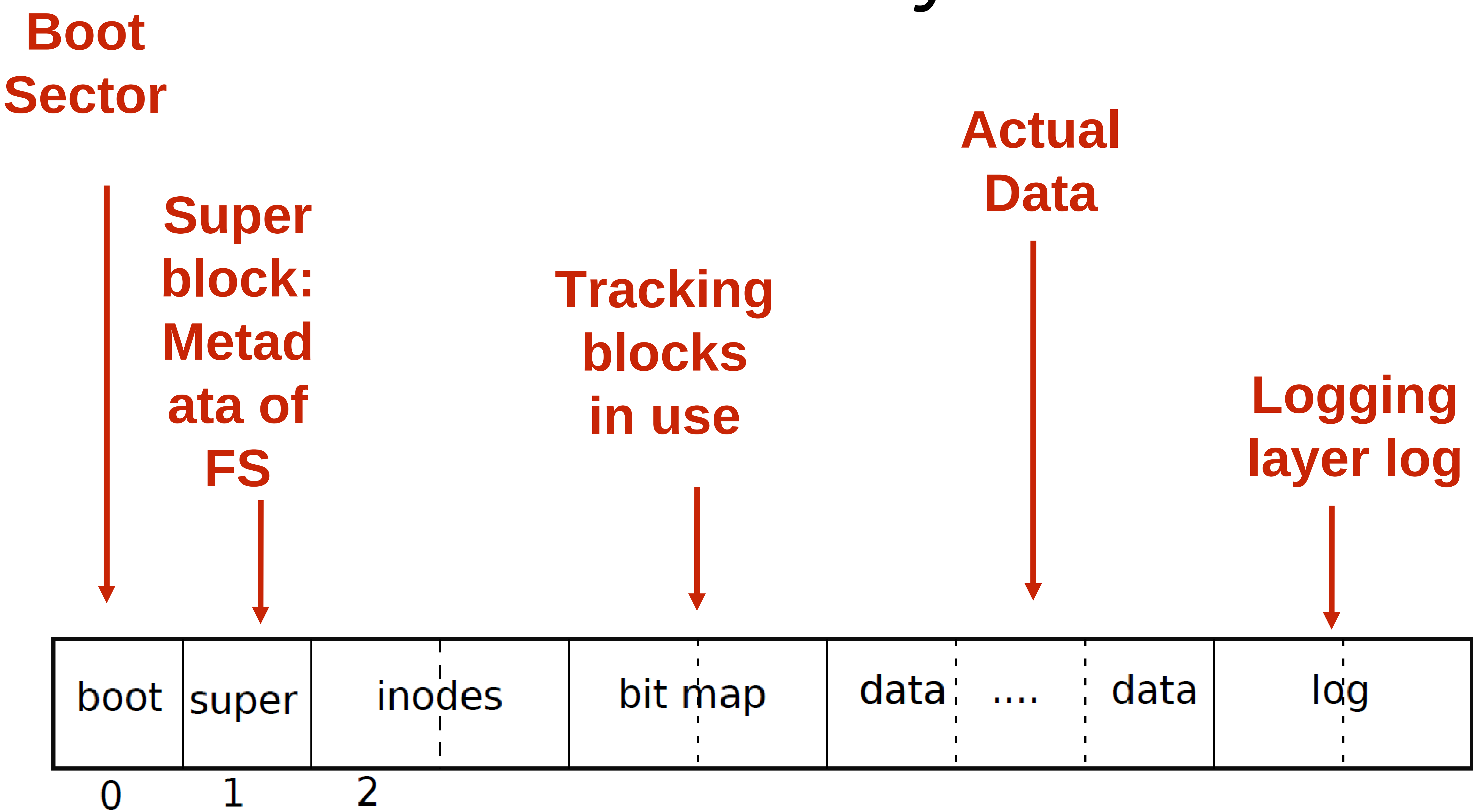





# FS Layers

- **Disk layer** reads and writes blocks on the IDE drive
- **Buffer cache** layer caches blocks and synchronizes access to them
- **Logging layer** wraps multiple operations in a single atomic *transaction*, providing crash recovery
- **i-node layer** represents files as we saw last time
- The **directory layer** contains a special type of i-node that gives a list of names and i-node pointers
- **The pathname layer** resolves paths to i-nodes
- **The file descriptor layer** provides an abstraction for accessing several kinds of object (pipes, files, devices) as files

# Xv6 disk layout



# Today

- File System Implementation (Cont)
    - Log Structured Systems
    - Journaling
- XV6 File System Layers
    - Buffer Cache Layer
    - Logging Layer
    - Inode Layer
    - Directory Layer
    - PathName Layer
    - File Descriptor

# The Buffer Cache

- Two jobs:
  - Synchronize access to blocks so that two processes don't try to access the same data simultaneously
  - Cache commonly used blocks so that we don't have to read from the disk all the time
- Implemented in bio.c

# High-Level Interface

- **`bread`**():
    - Obtains a buffer containing a block that can be read or modified in memory.
- **`bwrite`**():
    - Writes a modified buffer to disk.
- **`brelse`**():
    - Called when done with a buffer to clear the B_BUSY flag

# Buffer Cache Structure

```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
#define B_BUSY  0x1  // buffer is locked by some process
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk

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

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

**Linked list used for IDE queue**

# Buffer Read/Write

```
// Return a B_BUSY buf with the contents of the indicated block.
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;

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

// Write b's contents to disk.  Must be B_BUSY.
void
bwrite(struct buf *b)
{
  if((b->flags & B_BUSY) == 0)
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```

# Buffer Cache Synchronization

- Calling `bread()` attempts to get the block either from cache or by reading the disk
- It acquires a lock on the buffer cache, then tries to find a cached copy
- If the cached copy is found and not in use, it sets a flag `(B_BUSY)` on it, releases the lock, and returns
- If it is in use, it calls `sleep()` to wait until the block is free

# Buffer Cache Synchronization (1/3)

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

  acquire(&bcache.lock);

 loop:
  // Is the block already cached?
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      if(!(b->flags & B_BUSY)){
        b->flags |= B_BUSY;
        release(&bcache.lock);
        return b;
      }
      sleep(b, &bcache.lock);
      goto loop;
    }
  }
```

**Access to cache protected by lock**

**Sleep if block is found but busy**

# Buffer Cache Synchronization (2/3)

```
  // Not cached; recycle some non-busy and clean buffer.
  // "clean" because B_DIRTY and !B_BUSY 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->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
      b->dev = dev;
      b->blockno = blockno;
      b->flags = B_BUSY;
      release(&bcache.lock);
      return b;
    }
  }
  panic("bget: no buffers");
}
```

**Note: this is only safe because we already know there is no buf with this blockno and still have lock ☺**

**If not found in cache, just pick any non-busy buffer**

# Buffer Cache Synchronization (3/3)

```
  // Not cached; recycle some non-busy and clean buffer.
  // "clean" because B_DIRTY and !B_BUSY 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->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
      b->dev = dev;
      b->blockno = blockno;
      b->flags = B_BUSY;
      release(&bcache.lock);
      return b;
    }
  }
  panic("bget: no buffers");
}
```

**Note: clears B_VALID and B_DIRTY**

**If not found in cache, just pick any non-busy buffer**

# Buffer Caching

- To actually cache frequently used blocks, xv6 maintains an LRU cache
- Implemented as a doubly linked list (**prev** and **next** pointers in the **struct buf**)
- After releasing a block, we move it to the head of the list so it can be found quickly

# LRU

```
// Look through buffer cache for block on device dev.
// If not found, allocate a buffer.
// In either case, return B_BUSY buffer.
static struct buf*
bget(uint dev, uint blockno)
{
[...]
  // Is the block already cached?
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
[...]
  }
  // Not cached; recycle some non-busy and clean buffer.
  // "clean" because B_DIRTY and !B_BUSY means log.c
  // hasn't yet committed the changes to the buffer.
  for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
[...]
  }
```

**Search for cached block starts at the head**

**Search for free block starts at the tail**

# Releasing a Buffer

```c
// Release a B_BUSY buffer.
// Move to the head of the MRU list.
void
brelse(struct buf *b)
{
  if((b->flags & B_BUSY) == 0)
    panic("brelse");

  acquire(&bcache.lock);

  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;

  b->flags &= ~B_BUSY;
  wakeup(b);

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

**Clear B_BUSY flag**

**Wake up anyone waiting on the buffer**

# Today

- File System Implementation (Cont)
  - Log Structured Systems
  - Journaling
- XV6 File System Layers
  - Buffer Cache Layer
  - Logging Layer
  - Inode Layer
  - Directory Layer
  - PathName Layer
  - File Descriptor

# Logging Layer

- Operations on the filesystem often require multiple writes to disk
- Crashes may leave things in an inconsistent state, e.g. during deletion:
  - May end up with directory entry pointing to a freed block (bad!)
  - May end up with a block that is not marked free but not referenced (harmless but wasteful)
- Logging layer allows higher layers to group multiple writes into a single *transaction* that will be committed atomically

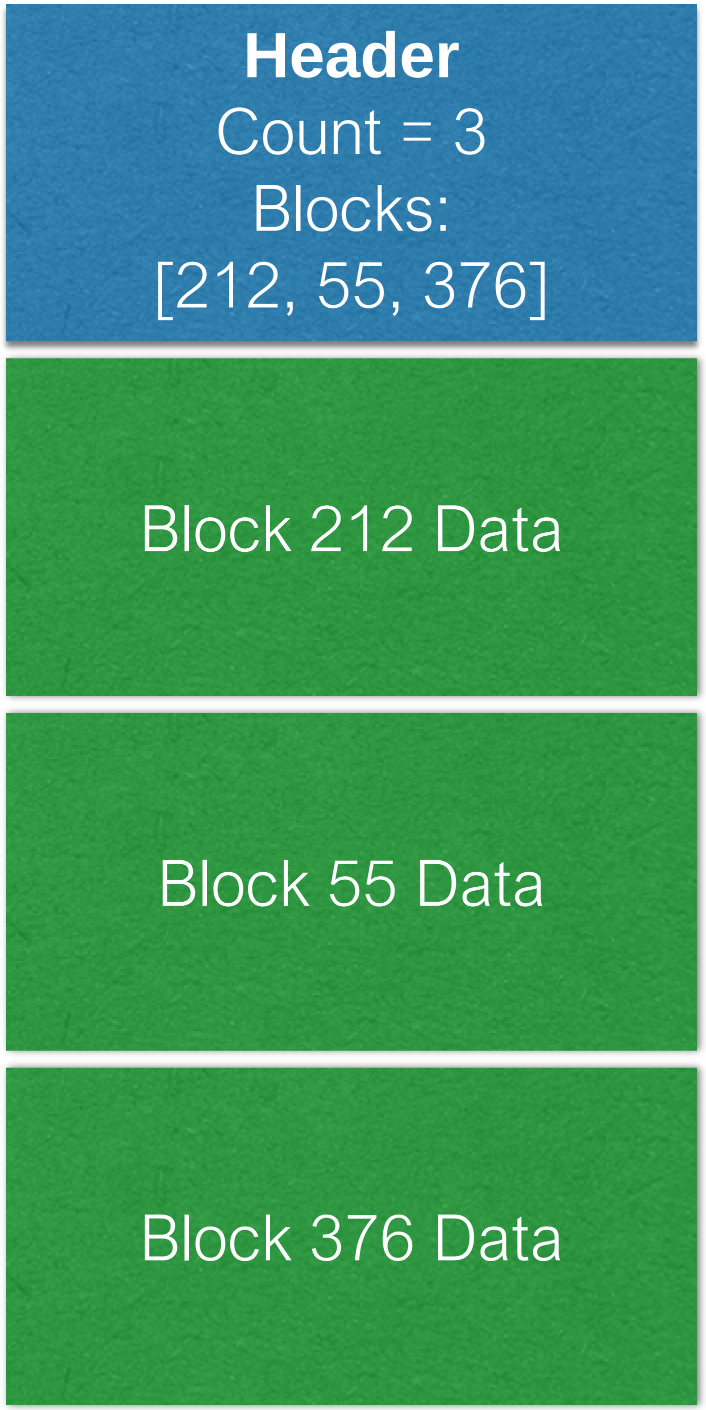
# Logging

- Instead of performing modifications to the disk directly, system calls in xv6 write out a *log* of pending operations to the disk
    - Each log entry consists of the data we intend to copy and the destination
- Once all modifications have been written to the log, write a special *commit* record
- Now, if there is a crash and the log is complete (has a commit record) we can replay it to recover

# Log Layout

- Log is fixed size and at fixed location on the disk
- Starts with a header, which gives a count of the number of block writes pending as well as an array listing what the block numbers are
- Then, LOGSIZE blocks with the pending write data

# Log Layout


**Header**
Count = 3
Blocks:
[212, 55, 376]
Block 212 Data
Block 55 Data
Block 376 Data

# Committing

- Once a transaction is complete, xv6 writes the updated log header to disk
- When it has finished writing out all the logged blocks to disk, it sets the count in the log header to 0
- Now two possibilities when we crash:
  - Partway through creating a transaction: count is zero, transaction is ignored
  - Partway through writing: count is non-zero, so we redo the copy

# Group Commit

- Multiple system calls can write to the log at once, allowing some concurrency
- So one log transaction may include data from multiple system calls
- xv6 makes sure not to actually commit until the last concurrent call finishes

# Logging Code

- Interface exposed
  - **begin_op** – start a transaction (may sleep)
  - **write_log** – write a disk block (replaces bwrite)
  - **end_op** – ends a transaction
- Logging layer uses bread/bwrite/brelse (buffer cache layer)
- Logging code implemented in **log.c**

# Using Logging

```
struct buf *bp;

begin_op();
[...]

bp = bread(...);
// modify bp->data


e
brelse(bp);
[...]

end_op();
```

# Logging Structures

```
// Contents of the header block, used for both the on-disk header block
// and to keep track in memory of logged block# before commit.
struct logheader {
  int n;
  int block[LOGSIZE];
};

struct log {
  struct spinlock lock;
  int start;
  int size;
  int outstanding; // how many FS sys calls are executing.
  int committing;  // in commit(), please wait.
  int dev;
  struct logheader lh;
};
struct log log;
```

# Beginning a Transaction

```
// called at the start of each FS system call.
// #define MAXOPBLOCKS  10  // max # of blocks any FS op writes
// #define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
// #define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
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;
    }
  }
}
```

# Writing to the Log

```
// Caller has modified b->data and is done with the buffer.
// Record the block number and pin in the cache with B_DIRTY.
// commit()/write_log() will do the disk write.
//
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);
}
```

Records block num

Mark buffer as dirty

# Ending a Transaction

```
// called at the end of each FS system call.
// commits if this was the last outstanding operation.
void
end_op(void)
{
  int do_commit = 0;

  acquire(&log.lock);
  log.outstanding -= 1;
  if(log.committing)
    panic("log.committing");
  if(log.outstanding == 0){
    do_commit = 1;
    log.committing = 1;
  } else {
    // begin_op() may be waiting for log space.
    wakeup(&log);
  }
  release(&log.lock);

  if(do_commit){
    // call commit w/o holding locks, since not allowed
    // to sleep with locks.
    commit();
    acquire(&log.lock);
    log.committing = 0;
    wakeup(&log);
    release(&log.lock);
  }
}
```

# Commit

```
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(); // copy from log area to diskhome locations
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
  }
}
```

# write_log

```
// Copy modified blocks from cache to log.
static void
write_log(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *to = bread(log.dev, log.start+tail+1); // log block
    struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
    memmove(to->data, from->data, BSIZE);
    bwrite(to);  // write the log
    brelse(from);
    brelse(to);
  }
}
```

# write_head

```
// Write in-memory log header to disk.
// This is the true point at which the
// current transaction commits.
static void
write_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *hb = (struct logheader *) (buf->data);
  int i;
  hb->n = log.lh.n;
  for (i = 0; i < log.lh.n; i++) {
    hb->block[i] = log.lh.block[i];
  }
  bwrite(buf);
  brelse(buf);
}
```

# Init Log

```
void
initlog(int dev)
{
  if (sizeof(struct logheader) >= BSIZE)
    panic("initlog: too big logheader");

  struct superblock sb;
  initlock(&log.lock, "log");
  readsb(dev, &sb);
  log.start = sb.logstart;
  log.size = sb.nlog;
  log.dev = dev;
  recover_from_log();
}
```

# Crash Recovery

```
// Called from init_log which is called during boot
// Before the first process runs

static void
recover_from_log(void)
{
  read_head();
  install_trans(); // if committed, copy from log to disk
  log.lh.n = 0;
  write_head(); // clear the log
}
```

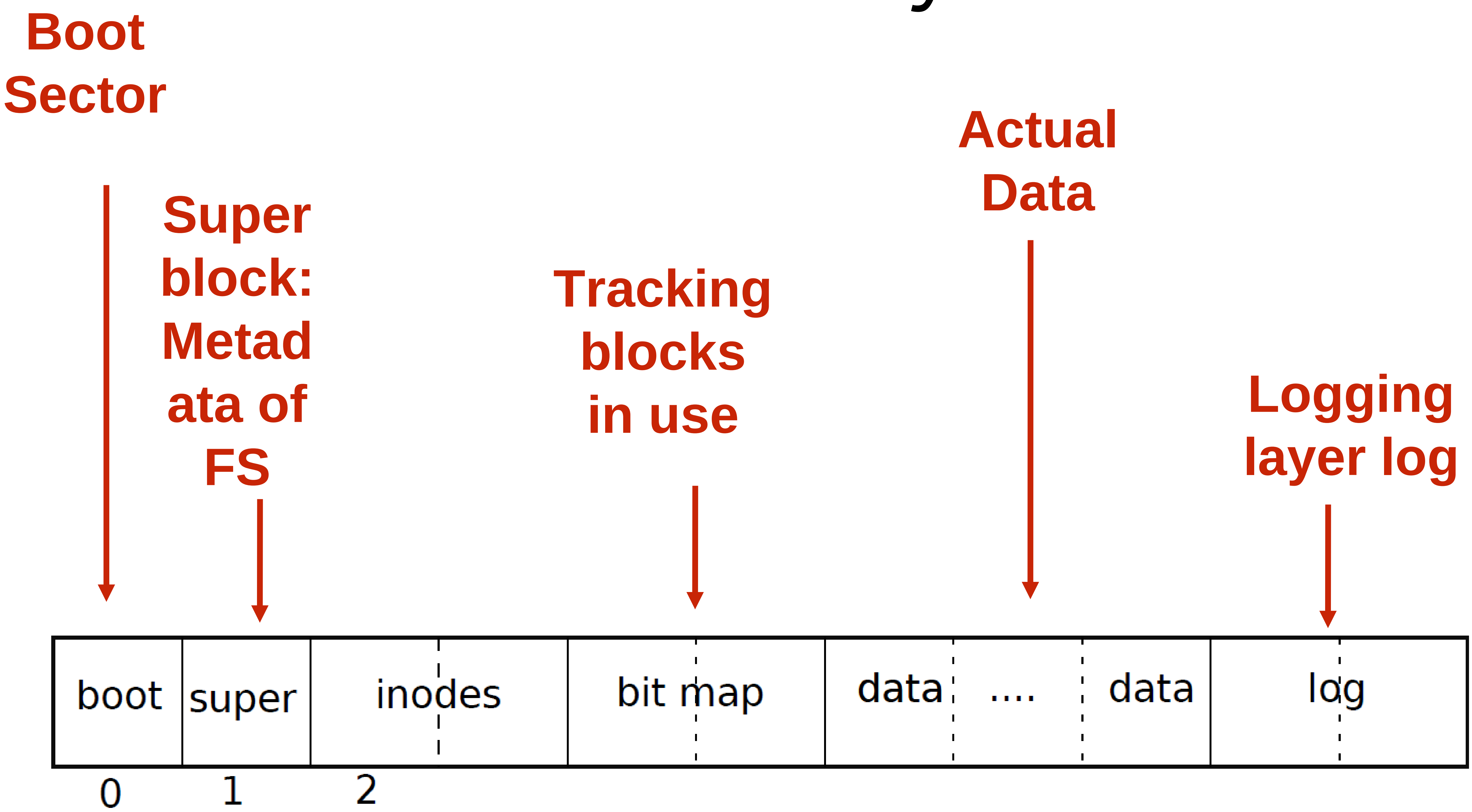
# install_trans

```
// Copy committed blocks from log to their home location
// Until you call install_trans nothing changes on the OS
static void
install_trans(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
    struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
    memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
    bwrite(dbuf);  // write dst to disk
    brelse(lbuf);
    brelse(dbuf);
  }
}
```

# Today

- File System Implementation (Cont)
  - Log Structured Systems
  - Journaling
- XV6 File System Layers
  - Buffer Cache Layer
  - Logging Layer
  - Inode Layer
  - Directory Layer
  - PathName Layer
  - File Descriptor

# Xv6 disk layout



# i-node Layer

- i-nodes store pointers to file blocks
- Every i-node is the same size and they are stored in a single area on disk, so it's easy to look up an i-node by number: inode_start + inode_num

# i-node Layer

- i-nodes in xv6 have both an on-disk (`struct dinode`) and in-memory representation (`struct inode`)
- The type field distinguishes between files, directories and special files (devices)
- xv6 maintains a cache of in-memory i-nodes in order to help synchronize access to i-nodes by multiple processes

# i-node Structures

```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  int flags;          // I_BUSY, I_VALID

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

```
// On-disk inode structure
struct dinode {
  short type;           // File type (file, dir, or device. 0 means it's free)
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

# i-node Interface

- **`iget`** – gets an i-node from the cache, does not guarantee that access is exclusive (many processes can access at once). May not have useful content in order to ensure it holds something call ilock!

- **`iput`** – releases the i-node, decrementing its reference count

- **`ilock`** – actually reads in the i-node data from disk (if not already present) and gets exclusive access

- **`iunlock`** – releases the lock on the i-node

# i-node Creation

- To create an i-node, we use `ialloc`
- Scans the on-disk i-nodes looking for one that's free
- **Remember:** The fact that the buffer layer guarantees exclusive access to a **block** means we don't have to worry about another process claiming the same free i-node

# i-node Creation Code

```
// Allocate a new inode with the given type on device dev.
// A free inode has a type of zero.
struct inode*
ialloc(uint dev, short type)
{
  int inum;
  struct buf *bp;
  struct dinode *dip;

  for(inum = 1; inum < sb.ninodes; inum++){
    bp = bread(dev, IBLOCK(inum, sb));
    dip = (struct dinode*)bp->data + inum % IPB;
    if(dip->type == 0){  // a free inode
      memset(dip, 0, sizeof(*dip));
      dip->type = type; // set it to our type: file, dir/dev
      log_write(bp);   // mark it allocated on the disk
      brelse(bp);
      return iget(dev, inum);
    }
    brelse(bp);
  }
  panic("ialloc: no inodes");
}
```

# iget

```
// Find the inode with number inum on device dev
// and return the in-memory copy. Does not lock
// the inode and does not read it from disk.
static struct inode*
iget(uint dev, uint inum)
{
  struct inode *ip, *empty;

  acquire(&icache.lock);

  // Is the inode already cached?
  empty = 0;
  for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
    if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
      ip->ref++;
      release(&icache.lock);
      return ip;
    }
    if(empty == 0 && ip->ref == 0)    // Remember empty slot.
      empty = ip;
  }
[ ... code to initialize the inode ...]
  release(&icache.lock);

  return ip;
}
```

# iput

```
// Drops the ref count to an in-memory inode.
// If that was the last reference, the inode cache entry can
// be recycled.
// If that was the last reference and the inode has no links
// to it, free the inode (and its content) on disk.
// All calls to iput() must be inside a transaction in
// case it has to free the inode.
void
iput(struct inode *ip)
{
  acquire(&icache.lock);
  if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
    // inode has no links and no other references: truncate and free.
    if(ip->flags & I_BUSY)
      panic("iput busy");
    ip->flags |= I_BUSY;
    release(&icache.lock);
    itrunc(ip);
    ip->type = 0;
    iupdate(ip);
    acquire(&icache.lock);
    ip->flags = 0;
    wakeup(ip);
  }
  ip->ref--;
  release(&icache.lock);
}
```

# ilock

**Note: copy inode from disk into memory**

```c
// Lock the given inode.
// Reads the inode from disk if necessary.
void
ilock(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

  if(ip == 0 || ip->ref < 1)
    panic("ilock");

  acquire(&icache.lock);
  while(ip->flags & I_BUSY)
    sleep(ip, &icache.lock);
  ip->flags |= I_BUSY;
  release(&icache.lock);

  if(!(ip->flags & I_VALID)){
    bp = bread(ip->dev, IBLOCK(ip->inum, sb));
    dip = (struct dinode*)bp->data + ip->inum%IPB;
    ip->type = dip->type;
    ip->major = dip->major;
    ip->minor = dip->minor;
    ip->nlink = dip->nlink;
    ip->size = dip->size;
    memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
    brelse(bp);
    ip->flags |= I_VALID;
    if(ip->type == 0)
      panic("ilock: no type");
  }
}
```

# iupdate

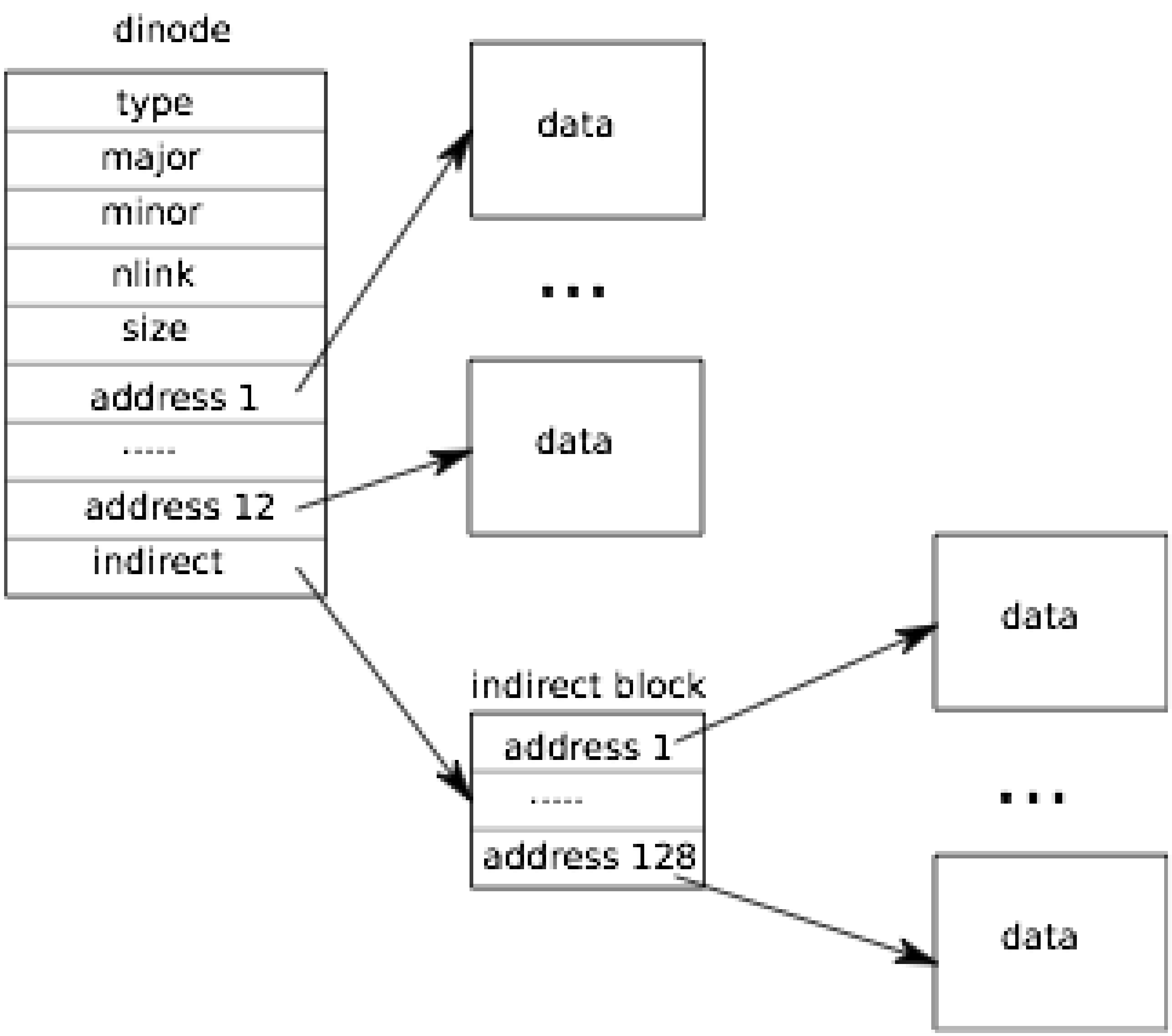
**Note: write a modified inode back to disk**

```
// Copy a modified in-memory inode to disk.
void
iupdate(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

  bp = bread(ip->dev, IBLOCK(ip->inum, sb));
  dip = (struct dinode*)bp->data + ip->inum%IPB;
  dip->type = ip->type;
  dip->major = ip->major;
  dip->minor = ip->minor;
  dip->nlink = ip->nlink;
  dip->size = ip->size;
  dip->created = ip->created;
  memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
  log_write(bp);
  brelse(bp);
}
```

# iunlock

```c
// Unlock the given inode.
void
iunlock(struct inode *ip)
{
  if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
    panic("iunlock");

  acquire(&icache.lock);
  ip->flags &= ~I_BUSY;
  wakeup(ip);
  release(&icache.lock);
}
```

# i-node Content



# i-node Content

- Note that there's only *one* level of indirection here

- So there is a maximum file size:
  (number of direct blocks + number of indirect blocks)
  x blocksize
  = (12 + 128) x 512 bytes = 70 KB

# Today

- File System Implementation (Cont)
  - Log Structured Systems
  - Journaling
- XV6 File System Layers
  - Buffer Cache Layer
  - Logging Layer
  - Inode Layer
  - Directory Layer
  - PathName Layer
  - File Descriptor

# Directory Layer

- Implemented internally much like a file
- Its inode has type T_DIR
- It's data is a sequence of directory entries
- Each entry is `struct dirent`

# xv6 Directories

- Directories are just another kind of file, whose content is a list of **struct dirents**

File & directory names are limited to 14 characters

```
// Directory is a file containing a
// sequence of dirent structures.
#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

# Directory Lookup

```
// Look for a directory entry in a directory.
// If found, set *poff to byte offset of entry.
struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)
{
  uint off, inum;
  struct dirent de;

  if(dp->type != T_DIR)
    panic("dirlookup not DIR");

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink read");
    if(de.inum == 0)
      continue;
    if(namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
      inum = de.inum;
      return iget(dp->dev, inum);
    }
  }

  return 0;
}
```