xv6 File Systems — Low-level I/O & Buffer Cache

Computer Science 432 — Lecture 17 — Duane Bailey

April 20, 2022

Announcements

- No small group meetings; office hours W1-4, R1-4, F9-10:30. Meetings *will* happen next week (and, maybe, the next)
- Lab 6 is due Friday.

<Finishing: direct examination of a file system>

File Systems

- Next few lectures will be spent discussing File Systems/Chapter 8.
  - Today: review disk structure, disk I/O & buffer caching.
  - Monday (4/26): Consistency, using logging and locking.
  - Next Wednesday (4/28): Other file system approaches.

File System Abstractions

- Today we're looking at the 2 lowest levels

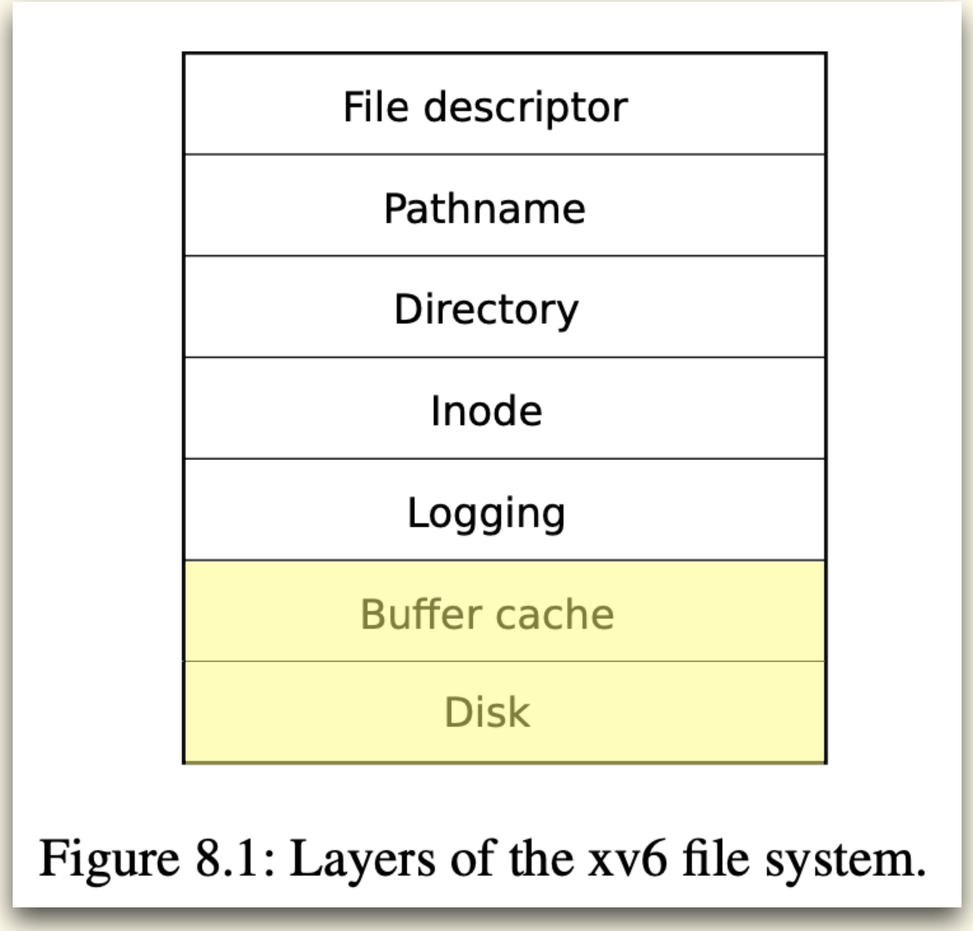

Figure 8.1: Layers of the xv6 file system.

File System Layer Concepts

- File system layers, lowest to highest:
  - Disk system—a “virtio” disk device interface provided by `qemu`.
  - Buffer cache—a cache of blocks/buffers that avoids constant disk I/O

The virtio Disk Subsystem (qemu)

- The virtio devices are a open, standards-based mechanism for implementing *paravirtualized* devices. (author: Rusty Russell)
  - An approach to providing classes of virtual devices that look like physical devices to virtual environments and hypervisors.
  - virtio-blk: takes the contents of the a file and presents it internally as a disk device.
  - qemu implementation is particularly high performance (100Mb/s)
  - xv6 driver is found in kernel/virtio_disk.c and virtio.h

The virtio Disk Interface

- The virtio block device is controlled by several memory-mapped regs.
  - Register descriptions are found in virtio.h
  - Based at location 0x10001000.
  - For example, first register, offset 0 contains magic 0x74726976 (why?)
- Communication with drive controlled by a *ring* of "available" request and "used" response descriptors.
- Drive raises an interrupt whenever a request has been responded to.
- Interrupts routed through trap system, much like console UART.
- Drive performs *direct memory access* (DMA) to read/write data.

The Block Reader/Writer Method

- One method allows user to read (or write) a block from (or to) the disk
  - `virtio_disk_rw(buf, wr)`: performs read (write) of buf if `wr`=0 (1)
  - Hides complexity:
    - Converts from a disk *block (1024b)* address to a *sector (512b)* address
    - Adding request to the disk queue
    - Waiting for the interrupt/sleeplock associated with the request
    - Transferring of information from the response to `buf`

The `buf` Structure

- Metadata:
  - `valid`: is data valid, here?
  - `disk`: this a target of disk I/O?
  - `dev`: the disk
  - `blockno`: the address of block
  - `refno`: # kernel refs to buf
  - `prev/next`: LRU ordering
- Payload:
  - `data`: the buffer data

```
struct buf {
  int valid;   // has data been read from disk?
  int disk;    // does disk "own" buf?
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *prev; // LRU cache list
  struct buf *next;
  uchar data[BSIZE];
};
```

The Buffer Cache

- The kernel uses a *buffer cache* to minimize disk I/O requests
- Unit of disk transfer is a *buffer* represented by the `buf` structure
- Cache ($) features we hope to achieve:
  - Requesting a buffer from the cache
    - Checks for presence of buffer in the cache
    - If it is missing, it is read from the disk (virtio_disk_rw), added to cache
    - Returns buffer (now) resident in cache
  - Writing a buffer to the cache
    - Writes *through* the cache
  - Cache replacement policy—least recently used—implemented with circ. queue

Cache Initialization

- `NBUF` buffers are kept in a doubly linked list:
  - The cache maintains a dummy node, head (seems unnecessary)
  - `head.next` is just-used buffer
  - `buf.next` was used less recently than `buf`
  - Unused buffers appear after used buffers

- `binit()` sets up the buffer cache; all buffers unreferenced

Typical Buffer Access

- Typically, to process a disk block:
  - `bread(dev, b#)` reads to `bp`
  - Process buffer `bp`
  - `brelse(bp)` de-refs the `bp`
- Buffer cache keeps track of referenced buffers, releases others
- Reference counting helps $ decisions

```
struct superblock sb;

// Read the super block.
static void
readsb(int dev, struct superblock *sb)
{
  struct buf *bp;

  bp = bread(dev, 1);
  memmove(sb, bp->data, sizeof(*sb));
  brelse(bp);
}
```

Buffer access details

- `bget()` looks for a buffer in the buffer cache; allocates one if not found
  - Increases the reference count

- `bread()` reads a block from disk into a buffer:
  - Calls `bget` to find existing or allocate new `buf`
  - If buffer is not `valid`, data is read from the disk

- `brelse()` indicates you're not using—releasing—the buffer
  - Decreases the reference count
  - If reference count is nonzero, moves buffer to head of LRU queue

Writing a buffer

- From inode and up, we call log_write to write a buffer
  - (As we'll see on Monday) adds the write to buffers in transaction
- `bwrite()` is only called by the logging layer:
  - Writing only happens as transaction commits
- `bwrite()` is responsible for writing a buffer to the disk
  - Simply: calls `virtio_disk_rw(buffer, 1)`

Recall: dinode

- Structure of file on disk
  - `dinode` holds metadata
  - first 12 `addresses` point to disk blocks
  - last `address` points to a disk block of pointers to even more disk blocks
- Within `f/s` `inode` holds `dinodes` along with even more metadata

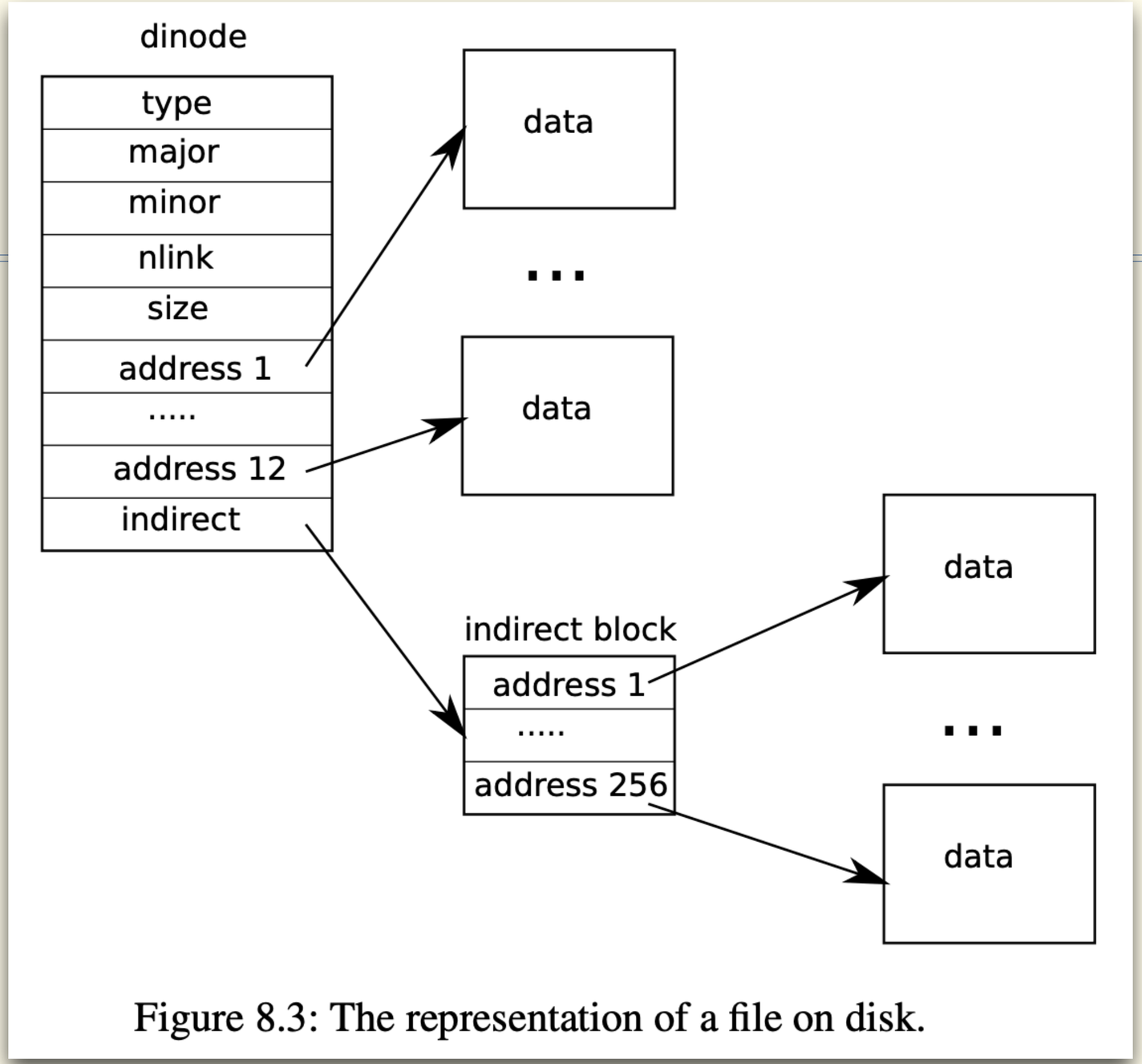

Figure 8.3: The representation of a file on disk.

inode e.g. — bmap

- Reads the block `bn` from file
  - `inode` describes file
  - `balloc(dev)` finds a free data block on disk, marks it allocated, zeros it
- Part of a larger transaction
  - again, `log_write`: writes only when transaction committed
- `bmap()` is subject of next lab…

```
// Return the disk block address of the nth block in inode ip.
// If there is no such block, bmap allocates one.
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;

  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }

  panic("bmap: out of range");
}
```