xv6 File Systems — Overview

Computer Science 432 — Lecture 16 — Duane Bailey

April 18, 2022

Announcements

- Masks are back…Zoom links for lectures in calendar.
  Other times: `https://tinyurl.com/dab-office`.
- Meeting during lab times, in lab. Optional, if you need help.
- No small group meetings; office hours T1-3, W1-4, R1-4, F9-10:30.
- Lab 6 is due Friday. Questions?

File Systems

- Next few lectures will be spent discussing File Systems/Chapter 8.
  - Today: overview and file system structure.
  - Wednesday: disk I/O & buffer caching.
  - Next Monday: Consistency, using logging and locking.

File System Abstractions

- File systems in xv6 are nesting abstractions:
  - Highest level: the general file descriptor
  - Middle levels: file system structure
  - Lowest levels: disk I/O & buffering
- Layers build increasingly complex abstractions from bottom to top

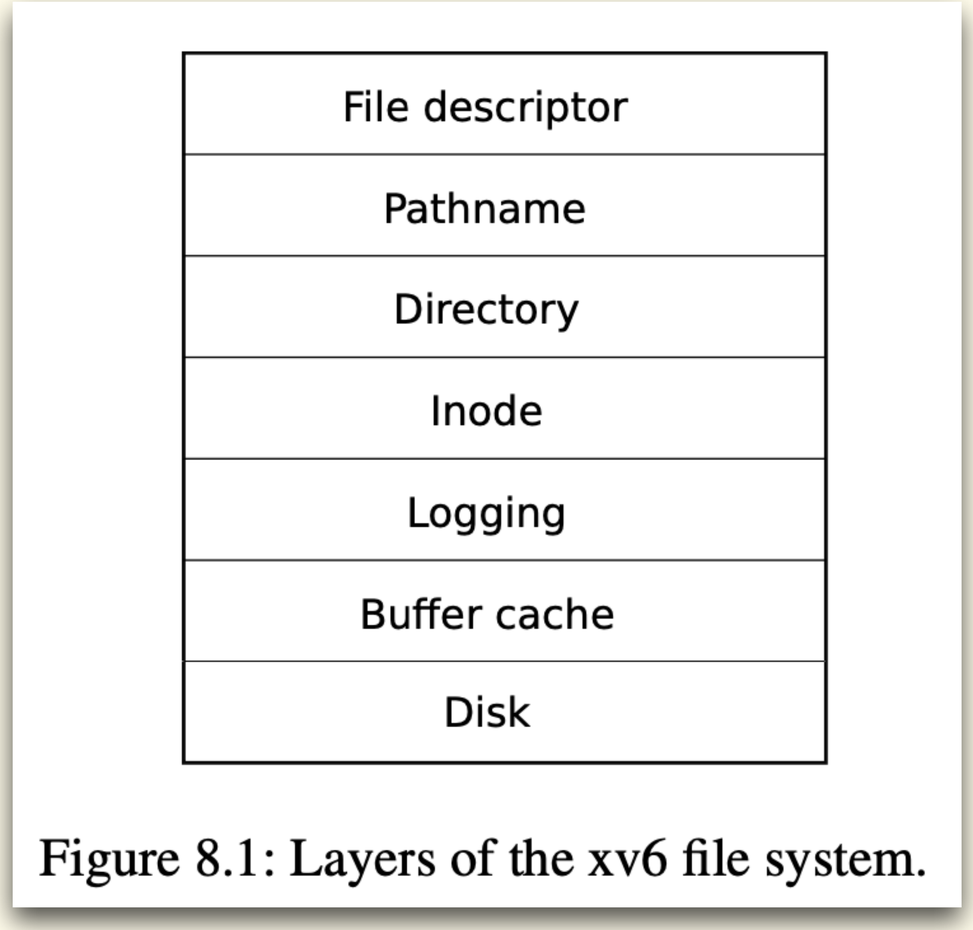


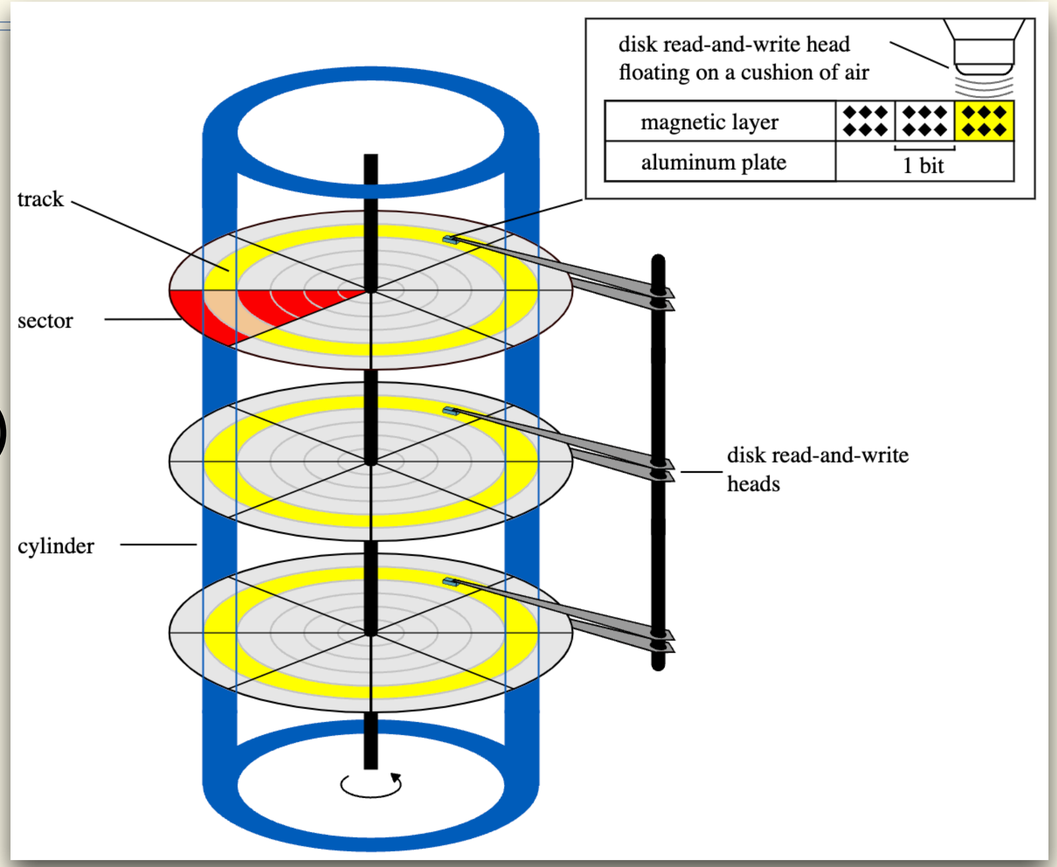
Figure 8.1: Layers of the xv6 file system.

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
  - Logging—a layer that supports *transactions* that help survive crashes
  - `inode`—the basic file structure component shared by all Unix systems
  - Directory—a simpl(istic) directory structure we've already seen
  - Pathname—a recursive structure for traversing the directory tree
  - File descriptor—a general interface to files, directories, and devices.

The Disk as a Block-Addressable Memory

- Unix was developed for disk drives:
  - A hard disk consists of several *platters*
  - Each platter has concentric *tracks*. (Parallel tracks on platters form *cylinders*.)
  - Each track is divided into *sectors*.
  - We can enumerate the sectors of a disk with sequential platter::track::sector addresses



wikipedia

- To Unix: disk is a random access array of data

The Unix File System Structure

- The `xv6` assumption about disk layout is:
  - Every block has 1024 bytes; block address is 2-bytes (64K blocks)
  - Block 0: the *boot block* contains boot code
  - Block 1: the *superblock* (SB). Metadata about parameterization of disk
  - Then: a series of blocks holding the *disk log*. Size determined by SB
  - Then: a series of blocks holding *inodes*. The metadata describing files.
  - Then: a series of blocks holding a *bitmap* identifying used blocks.
  - Then: all the blocks that hold data within files.

The Boot Block

- No matter what the file system structure, block 0 is the boot code.
  - Early stages of the boot process are not very able.
  - The approach to loading boot code must be very simple.

- It is fairly easy to load block 0 off a disk drive:
  - Approximately the same approach is used to read block 0 from a floppy disk, or a CD, or a hard drive, or an SSD.
  - Interfaces for complex/high performance disks facilitate booting

- Boot code then focuses on reading a larger boot file from drive

The Superblock

- Depending on the *geometry* of the disk, there are many parameters that determine the structure of a Unix file system.
- The *superblock* holds these metadata.
  - Superblock is typically block 1.
  - With information from superblock, you should be able to *compute* locations of other important structures, easily:
    - Transaction log—location and extent
    - Inodes—location and number; individual inode in known location
    - Bitmap—location and extent; block *n* bit at a computable location
    - Data blocks—location of data block *n* is easily computed.

The Superblock

```
// mkfs computes the super block and builds an initial file system. The
// super block describes the disk layout:
struct superblock {
  uint magic;        // Must be FSMAGIC
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks
  uint logstart;     // Block number of first log block
  uint inodestart;   // Block number of first inode block
  uint bmapstart;    // Block number of first free map block
};

#define FSMAGIC 0x10203040
```

(Look at fs.img using “od -Xa -Ax fs.img”)

The (disk) inode—Anonymous File Descriptor

- The `dinode` is *the* basic structure describing a file:
  - `Type` (plain, dir, device); major/minor
  - `nlink`—reference count for file
  - `size`—extent of file, in bytes (note: no `EOF`!)
  - `addrs` array:
    - direct: address of each successive logical block
    - indirect: address of one *block of addresses*

```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)

// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file sys
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

The (disk) inode

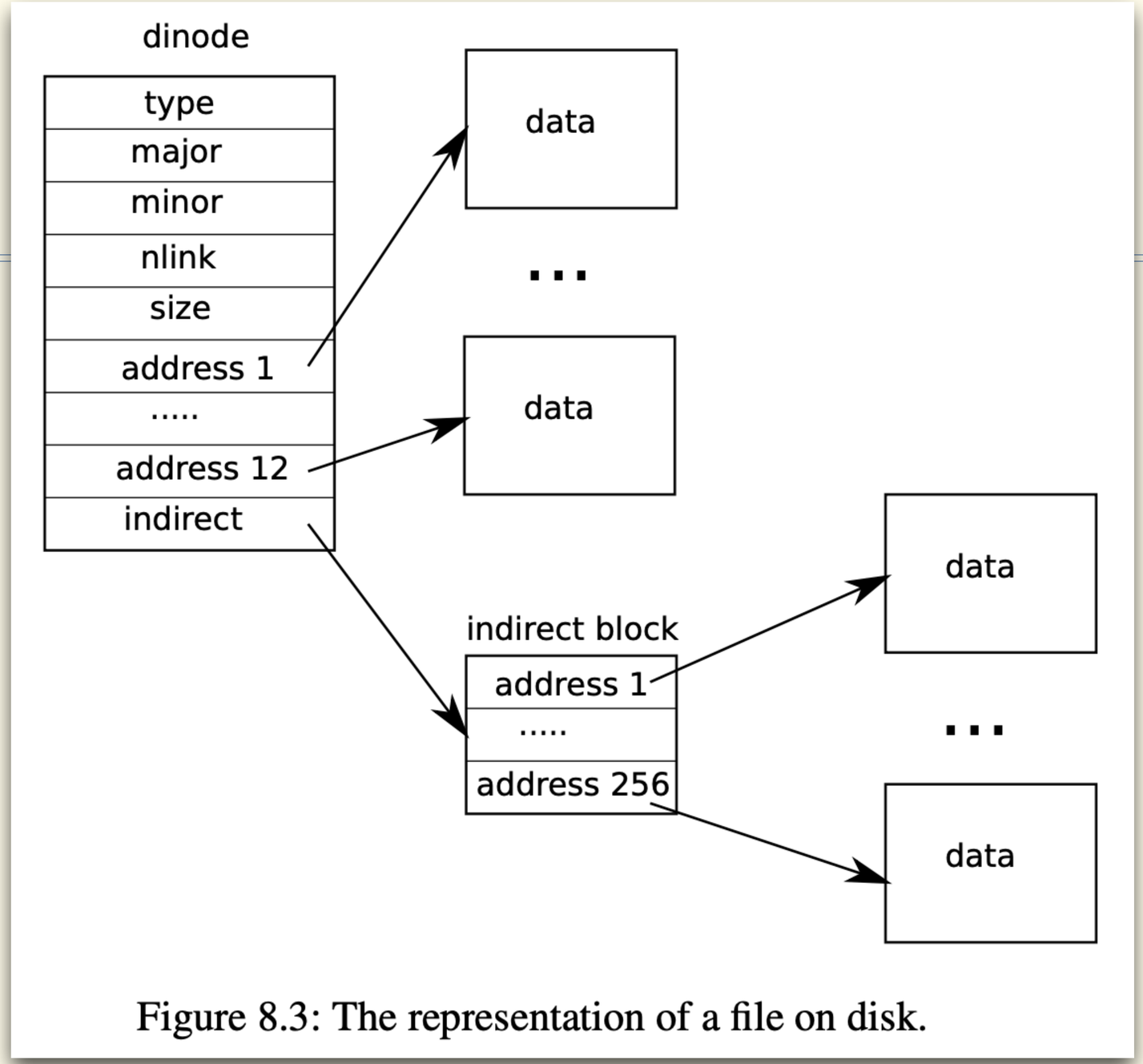


Figure 8.3: The representation of a file on disk.

The (disk) inode—Anonymous File Descriptor

- Notice that the `inode` describes a *nameless* file:
  - Directory entries attach name(s) to `inodes`
  - Multiple references to a single file are called *hard links*
  - File remains in system as long as there is a single link
- The ratio of direct::indirect links is a tradeoff:
  - One indirect block: simple to code and compute
  - Limits maximum size of files
  - Could easily have multiple indirect blocks, or tree-ish structure.
- All used *data blocks* on disk are pointed to by *some* `inode`. This can be checked!
- `Inodes` are numbered starting at 1. *Why?*
- Data blocks are addressed using *disk* block addresses (no one accesses boot)

The Directory Structure

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

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

- Directories are simply files that contain lists of fixed sized dirents
  - We've seen this in our implementation of `find`.
  - The `dirent` contains only an `inode` number and a name
    - All other metadata is stored in (shared) inode
    - Zero inode means: unused `dirent`
    - Name is zero-terminated if it doesn't use all `DIRSIZ` bytes
  - Requires a sequential read of the directory to find a file.
    This could be (and often is) improved with a tree-based layout of `dirents`

- Inode 1 is always the root directory (/) for the file system.

The Bitmap

- The *bitmap* is used to keep track of the availability of all disk blocks.
  - The bits of the bitmap are interpreted as a long bitfield
  - Bitmap is lengthened to take up a whole number of blocks
  - A bit value of 1 indicates allocation, 0 otherwise.

  - Notice: it is impossible to reconstruct this bitmap by looking at data
    Any check of this structure must scan inodes.

mkfs: Make file system

- In xv6, “make qemu” causes a build or *make* of the file system:
  - You can now read the source code for mkfs/mkfs.c; an Ubuntu app.
  - Notice that there are decisions made here; tradeoffs are weighed:
    - How many inodes do we need?
      - Increasing inodes eats into data storage
      - Useful to think about average number of blocks in a file.
    - How much log space should be reserved?
      - We’ll think about this in a week as we consider transactions.

<Direct examination of a file system>