Week 12 - Friday

CS222

Last time

- What did we talk about last time?
- Low level file I/O

Questions?

Project 5

Quotes

Programs must be written for people to read and only incidentally for machines to execute.

Harold Abelson and Gerald Jay Sussman

Authors of *The Structure and Interpretation of Computer Programs*

Example

- Use low level I/O to write a hex dump program
- Print out the bytes in a program, 16 at a time, in hex, along with the current offset in the file, also in hex
- Sample output:

```
0x000000 : 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0x000010 : 02 00 03 00 01 00 00 00 c0 83 04 08 34 00 00 00
0x000020 : e8 23 00 00 00 00 00 00 34 00 20 00 06 00 28 00
0x000030 : 1d 00 1a 00 06 00 00 00 34 00 00 00 34 80 04 08
```

File descriptors revisited

- A file descriptor is not necessarily unique
  - Not even in the same process
- It's possible to duplicate file descriptors
  - Thus, the output to one file descriptor also goes to the other
  - Input is similar

Duplicating descriptors on the command line

- **`stderr`** usually prints to the screen, even if **`stdout`** is being redirected to a file

```
./program > output.txt
```

- What if you want **`stderr`** to get printed to that file as well?

```
./program > output.txt 2>&1
```

- You can also redirect only **`stderr`** to a file

```
./program 2> errors.log
```

dup() and dup2()

- If you want a new file descriptor number that refers to an open file descriptor, you can use the **`dup()`** function

```
int fd = dup(1); //makes a copy of stdout
```

- It's often more useful to change an existing file descriptor to refer to another stream, which you can do with **`dup2()`**

```
dup2(1, 2);
//makes 2 (stderr) a copy of 1 (stdout)
```

- Now all writes to **`stderr`** will go to **`stdout`**

I/O buffering in files

- Reading from and writing to files on a hard drive is expensive
- These operations are buffered so that one big read or write happens instead of lots of little ones
  - If another program is reading from a file you've written to, it reads from the buffer, not the old file
- Even so, it is more efficient for your code to write larger amounts of data in one pass
  - Each system call has overhead

Buffering in stdio

- To avoid having too many system calls, **`stdio`** uses this second kind of buffering
  - This is an advantage of **`stdio`** functions rather than using low-level **`read()`** and **`write()`** directly
- The default buffer size is 8192 bytes
- The **`setvbuf()`**, **`setbuf()`**, and **`setbuffer()`** functions let you specify your own buffer

Flushing a buffer

- Stdio output buffers are generally flushed (sent to the system) when they hit a newline (**`'\n'`**) or get full
  - When debugging code that can crash, make sure you put a newline in your **`printf()`**, otherwise you might not see the output before the crash
- There is an **`fflush()`** function that can flush **`stdio`** buffers

```
fflush(stdout); //flushes stdout
//could be any FILE*
fflush(NULL); //flushes all buffers
```

Disks and partitions

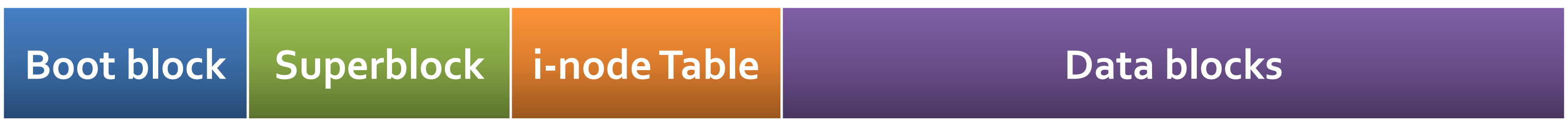
- Until SSDs completely take over, many physical hard drives are electronically controlled spinning platters with magnetic coatings
  - Disks have circular **tracks** divided into **sectors** which contain **blocks**
  - A block is the smallest amount of information a disk can read or write at a time
- Physical disks are partitioned into logical disks
- Each partition is treated like a separate device in Linux
  - And a separate drive (`C:`, `D:`, `E:`, etc.) in Windows
  - Each partition can have its own file system

Popular file systems

- Linux supports a lot of file systems
  - ext2, the traditional Linux file system
  - Unix ones like the Minix, System V, and BSD file systems
  - Microsoft's FAT, FAT32, and NTFS file systems
  - The ISO 9660 CD-ROM file system
  - Apple's HFS
  - Network file systems, including Sun's widely used NFS
  - A range of journaling file systems, including ext3, ext4, Reiserfs, JFS, XFS, and Btrfs
  - And more!

Partition layout

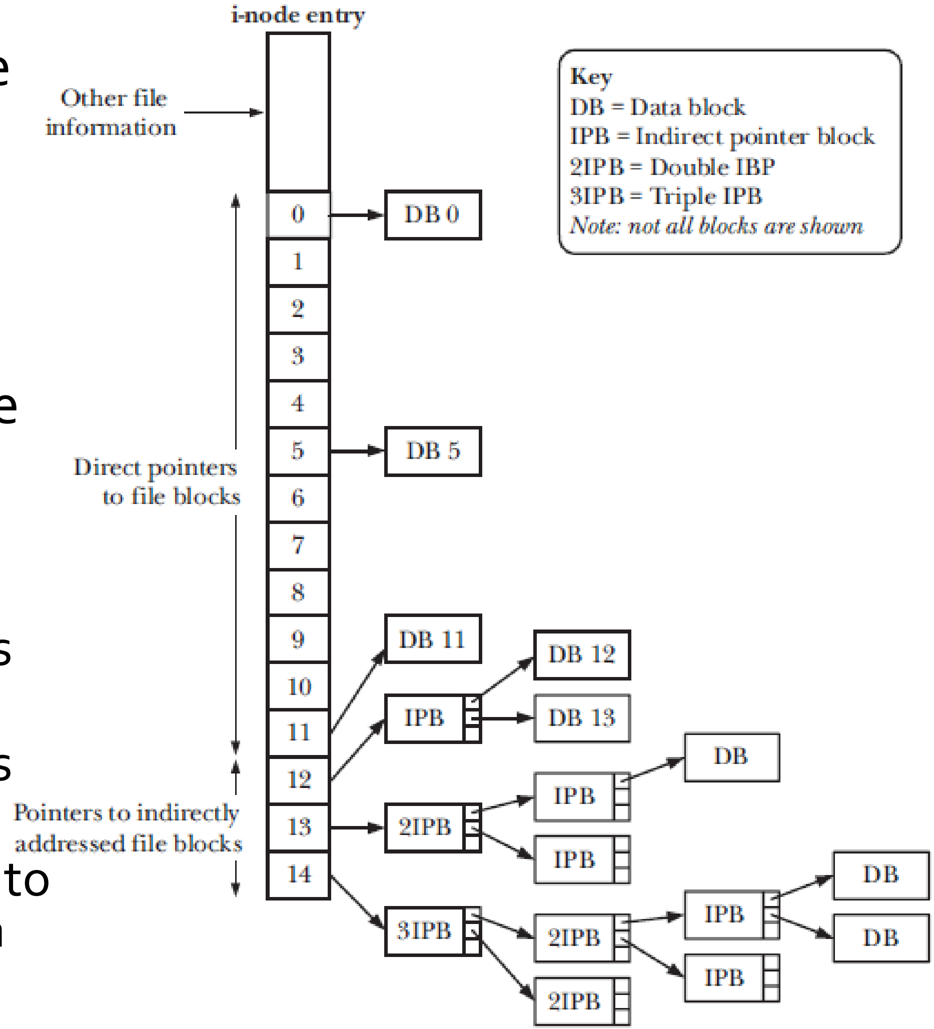
- Virtually all file systems have each partition laid out something like this



- The boot block is the first block and has information needed to boot the OS
- The superblock has information about the size of the i-node table and logical blocks
- The i-node table has entries for every file in the system
- Data blocks are the actual data in the files and take up almost all the space

i-nodes

- Every file has an i-node in the i-node table
- Each i-node has information about the file like type (directory or not), owner, group, permissions, and size
- More importantly, each i-node has pointers to the data blocks of the file on disk
- In ext2, i-nodes have 15 pointers
  - The first 12 point to blocks of data
  - The next points to a block of pointers to blocks of data
  - The next points to a block of pointers to pointers to blocks of data
  - The last points to a block of pointers to pointers to pointers to blocks of data



Journaling file systems

- If a regular file system (like ext2) crashes, it might be in an inconsistent state
- It has to look through all its i-nodes to try to repair inconsistent data
- A **journaling file system** (like ext3, ext4, and Reiserfs) keeps metadata about the operations it's trying to perform
- These operations are called **transactions**
- After a crash, the file system only needs to repair those transactions that weren't completed

File attributes

- Files have many attributes, most of which are stored in their i-node
- These attributes include:
  - Device (disk) the file is on
  - i-node number
  - File type and permissions
  - Owner and group
  - Size
  - Times of last access, modification, and change
- There are functions that will let us retrieve this information in a C program
  - **`stat()`**, **`lstat()`**, and **`fstat()`**

stat structure

- Attributes can be stored in a **stat** structure

```
struct stat {
    dev_t st_dev; /* IDs of device on which file resides */
    ino_t st_ino; /* I-node number of file */
    mode_t st_mode; /* File type and permissions */
    nlink_t st_nlink; /* Number of (hard) links to file */
    uid_t st_uid; /* User ID of file owner */
    gid_t st_gid; /* Group ID of file owner */
    dev_t st_rdev; /* IDs for device special files */
    off_t st_size; /* Total file size (bytes) */
    blksize_t st_blksize; /* Optimal block size for I/O (bytes)*/
    blkcnt_t st_blocks; /* Number of (512B) blocks allocated */
    time_t st_atime; /* Time of last file access */
    time_t st_mtime; /* Time of last file modification */
    time_t st_ctime; /* Time of last status change */
};
```

`utime()` and `utimes()`

- I really shouldn't tell you about these
- But there are functions that can change the timestamp of a file
  - **`utime()`** lets you change the access and modification times for a file, with units in seconds
  - **`utimes()`** lets you do the same thing, but with accuracy in microseconds

Lab 12

Upcoming

Next time…

- Networking overview
- Sockets

Reminders

- Finish Project 5
  - Due tonight by midnight
- Read LPI Chapters 56, 57, 58, and 59