Lab 12: File System 2

Sejong Yoon, Ph.D.

References:

- Silberschatz, et al. *Operating System Concepts* (9e), 2013
- Materials from OS courses offered at TCNJ (Dr. Jikai Li), Princeton, Rutgers, Columbia (Dr. Junfeng Yang), Stanford, MIT, UWisc, VT

Agenda

- File System Example: xv6

Indirect link and max file size

- For ease of debugging, we change Makefile so that CPUS option to 1 and add -snapshot switch at the end of QEMUOPTS as explained in above link.

- Let's think about the size of the file system that we are trying to make. We want to implement doubly-indirect system. How many blocks at maximum will one inode (i.e., one file) can access? That should be

  12 (direct) + 128 (single-indirect) + 128 * 128 (double-indirect) = 16,524

- That's the number of blocks that an inode file can access. This is equivalent to

  16,524 * 512 = 8,460,288 bytes = 8,262 KB = about 8.07 MB

- So, the maximum file size in this system will be about 8.07 MB.

big.c and big1.c

- Reference: http://eng.utah.edu/~cs5460/OS-s14/bigxv6.html

- The demo programs provided in the above reference link
  - big.c will demo the maximum file size that current file system can create.
  - big1.c will demo larger files (16 MB, 32 MB, …), writing 1 MB at a time instead of 512 bytes (= 1 block).

- Feel free to use these codes for verifying your implementation's correctness for Project 4

- For submission, you need to provide a program like either one, that shows
  - (required) Your xv6 can create a file larger than 71,680 bytes, say, 72,000 bytes
  - (optional) It can create an 8 MB file

fs.h and fs.c

- you need to read (and probably modify) following files, among others:
  - fs.h: Contains the inode definition, file system size, etc.
  - fs.c: Implementation of most of the file system features

- On the other hand, you may want to make change to mkfs.c file. This is **NOT** a user program in xv6. It is a Linux program to initialize xv6 disk image and file system. Consider this task as installing a new hard disk and formatting it. By default, it has following definition near the top:

```
int nblocks = 985;
int nlog = LOGSIZE;
int ninodes = 200;
int size = 1024;
```

fs.h and fs.c (cont.)

- The last number denotes the total size of the disk, i.e., total number of sectors. Thus, fs.img, our hard disk size should be 1024 * 512 = 524,288 bytes. You can check this number using ls.
- If size = 5,860,844 sectors as in the reference link above, that means the disk size will be about 2.8 GB.
- Let's try to create a 20 MB disk. To do this, change size to be 40,960. If you run [make clean] and [make], you will see an error:

```
./mkfs fs.img README _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests _wc _zombie
used 39 (bit 11 ninode 26) free 39 log 10 total 1034
mkfs: mkfs.c:93: main: Assertion `nblocks + usedblocks + nlog == size' failed.
Makefile:169: recipe for target 'fs.img' failed
make: *** [fs.img] Aborted (core dumped)
make: *** Deleting file 'fs.img'
rm wc.o grep.o mkdir.o rm.o ln.o stressfs.o kill.o echo.o init.o usertests.o zombie.o cat.o sh.o ls.o
```

- Why?

fs.h and fs.c (cont.)

- As the assertion message shows, it should be (**number of usable blocks**) + (**number of blocks used by file system**) + (**number of blocks used by log/journal**). The last part is defined in param.h as

```
#define LOGSIZE 10
```

- To understand the middle part, you need to read mkfs.c. You can find following lines:

```
bitblocks = size/(512*8) + 1;
usedblocks = ninodes / IPB + 3 + bitblocks;
freeblock = usedblocks;

printf("used %d (bit %d ninode %zu) free %u log %u total %d\n", usedblocks,
        bitblocks, ninodes/IPB + 1, freeblocks, nlog, nblocks+usedblocks+nlog);

assert(nblocks + usedblocks + nlog == size);
```

fs.h and fs.c (cont.)

```
bitblocks = size/(512*8) + 1;
usedblocks = ninodes / IPB + 3 + bitblocks;
freeblock = usedblocks;

printf("used %d (bit %d ninode %zu) free %u log %u total %d\n", usedblocks,
        bitblocks, ninodes/IPB + 1, freeblocks, nlog, nblocks+usedblocks+nlog);

assert(nblocks + usedblocks + nlog == size);
```

fs.h

```
// Inodes per block.
#define IPB           (BSIZE / sizeof(struct dinode))
```

512 B

Number of i-nodes contained in each block

In the original xv6, this is 8 since
sizeof (dinode) =
2 + 2 + 2 + 2 + 4 + 4 * 13 = 64
and
512 / 64 = 8

fs.h and fs.c (cont.)

- Comments at top of fs.h:

```
// Block 0 is unused
// Block 1 is super block
// Blocks 2 through sb.ninodes/IPB  hold inodes
// Then free bitmap blocks holding sb.size bits
// Then sb.nblocks data blocks
// Then sb.nlog log blocks
```

- Here, important part is free bitmap blocks. They contain information whether each block in disk is in use of not, by using bitmap calculation. Each byte in the bitmap block can contain information for 8 blocks (8 bits = 1 byte), and each block is 512 bytes.
- Thus, given 40.960 total number of sectors (= potential blocks), we need 10 blocks to contain availability information. 1 additional block is for numerical remainders after division.

fs.h and fs.c (cont.)

- Comments at top of fs.h:

```
// Block 0 is unused
// Block 1 is super block
// Blocks 2 through sb.ninodes/IPB  hold inodes
// Then free bitmap blocks holding sb.size bits
// Then sb.nblocks data blocks
// Then sb.nlog log blocks
```

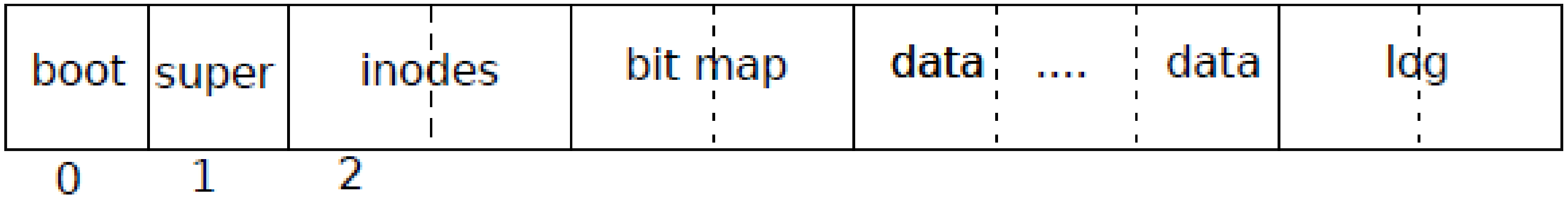
- We need the additional bitblock to accommodate the remaining blocks. Total number of usedblocks in line 87 of mkfs.c, means the total number of blocks used by super block, inodes, and the bitmap.
- **3 additional blocks come from the fact that, as described in fs.h, we have 2 blocks (1 boot block + 1 super block) at the beginning, and another extra block for remainder of ninodes / IPB calculation.**

TCNJ THE COLLEGE OF NEW JERSEY

fs.h and fs.c (cont.)

- That explains every single portion of our disk structure. Since we did not change the maximum number of inodes, we need

1	boot block
1	super block
26	inode blocks (200 / 8 + 1)
11	bitblocks (40,960 / (512 * 8) + 1)
10	log/journal blocks



fs.h and fs.c (cont.)

- Therefore, we need total of 49 blocks used for the file system in 20 MB size of disk. Remainder can be allocated for data, i.e., data blocks, as defined as nblocks in line 14 of mkfs.c.
- In default xv6, since the disk size was 1024 blocks, only 1 bit block suffice the need. Thus, in that case, 39 blocks were needed for file system, thus 985 blocks can be used for data blocks. Therefore, in Project 4, nblocks should be 40,960 – 49 = 40,911 blocks.

```
14  int nblocks = 40911;
15  int nlog = LOGSIZE;
16  int ninodes = 200;
17  int size = 40960;
```

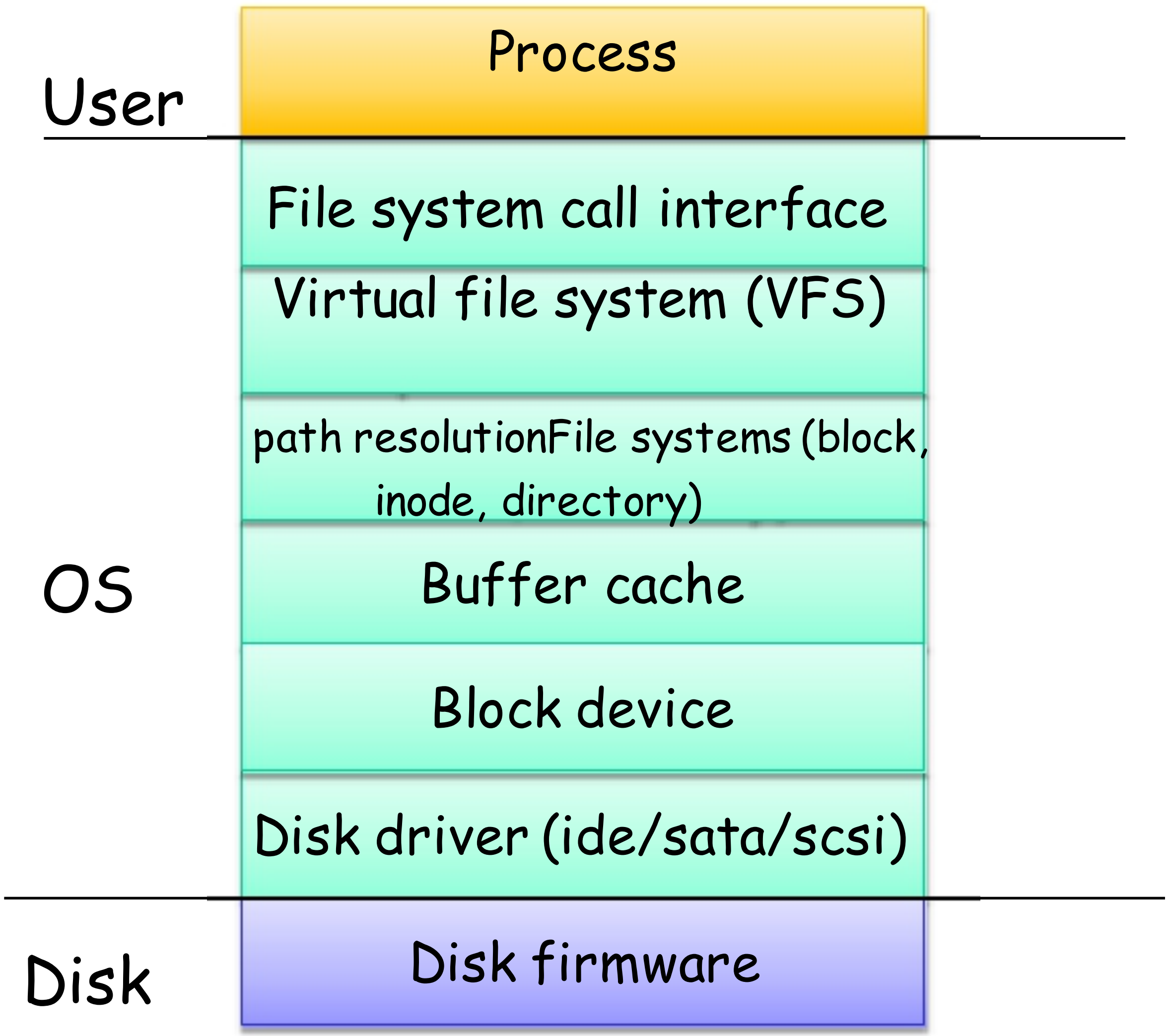
- Try to make changes accordingly in mkfs.c and recompile xv6

fs.h and fs.c (cont.)

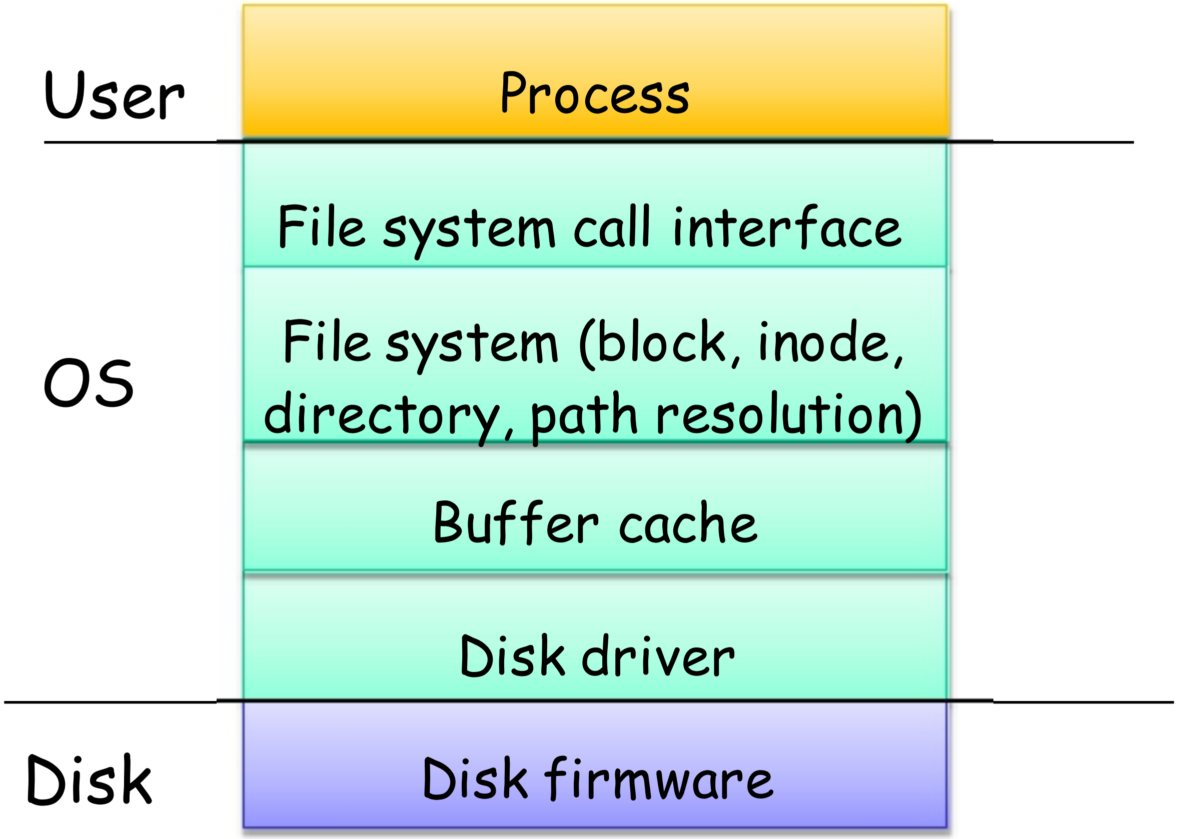
- fs.img file was created with correct size of 20,971,520 bytes = 20 MB.

```
oscreader@ubuntu:~/work/xv6$ ls -l fs*
-rw-rw-r-- 1 oscreader oscreader    14877 Nov 28 02:19 fs.c
-rw-rw-r-- 1 oscreader oscreader      113 Nov 28 03:21 fs.d
-rw-rw-r-- 1 oscreader oscreader     1516 Nov 27 20:03 fs.h
-rw-rw-r-- 1 oscreader oscreader 20971520 Nov 28 04:17 fs.img
-rw-rw-r-- 1 oscreader oscreader    17412 Nov 28 03:21 fs.o
oscreader@ubuntu:~/work/xv6$ ls -hl fs*
-rw-rw-r-- 1 oscreader oscreader  15K Nov 28 02:19 fs.c
-rw-rw-r-- 1 oscreader oscreader  113 Nov 28 03:21 fs.d
-rw-rw-r-- 1 oscreader oscreader 1.5K Nov 27 20:03 fs.h
-rw-rw-r-- 1 oscreader oscreader  20M Nov 28 04:17 fs.img
-rw-rw-r-- 1 oscreader oscreader  18K Nov 28 03:21 fs.o
oscreader@ubuntu:~/work/xv6$
```

Layered approach to storage systems



xv6 storage layers



xv6 disk driver

- ide.c
- iderw(struct buf *b): read or write disk sector
- idestart(struct buf *b): start request for b
- ideintr(): ide interrupt handler
- ideinit(): ide initializer

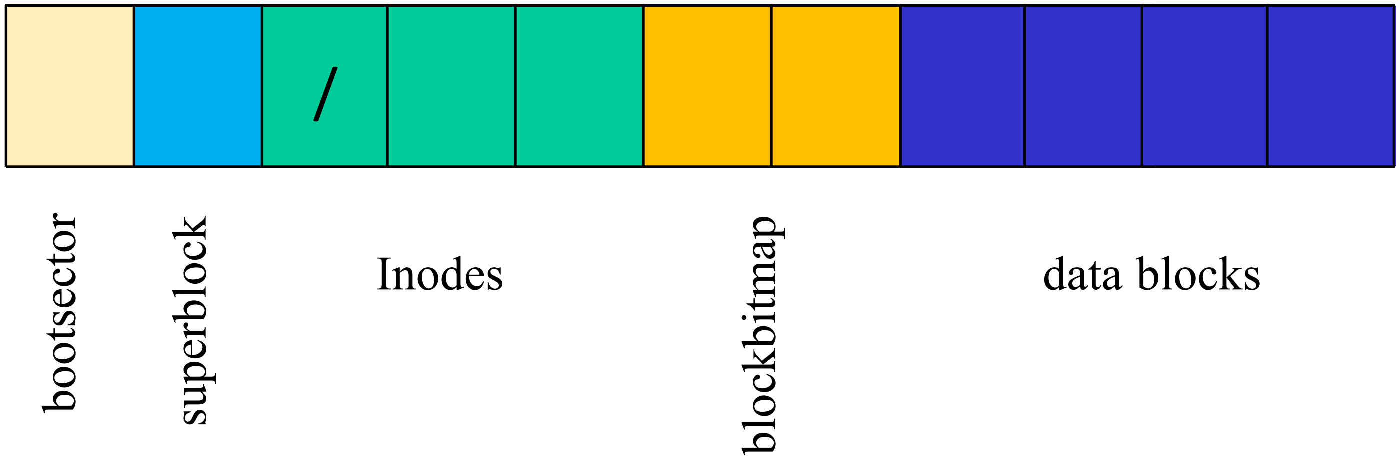
xv6 buffer cache

- bio.c
- struct buf
  - flags: B_BUSY, B_VALID, B_DIRTY
- struct bcache
  - head: LRU list of cached blocks
- bread(): read disk sector and return buffer
- bwrite(): write buffer to disk sector
- bget(): look up buffer cache for sector and set busy flag
- brelse(): clear busy flag and move buffer to head
- binit(): initialize buffer cache

xv6 buffer cache locking

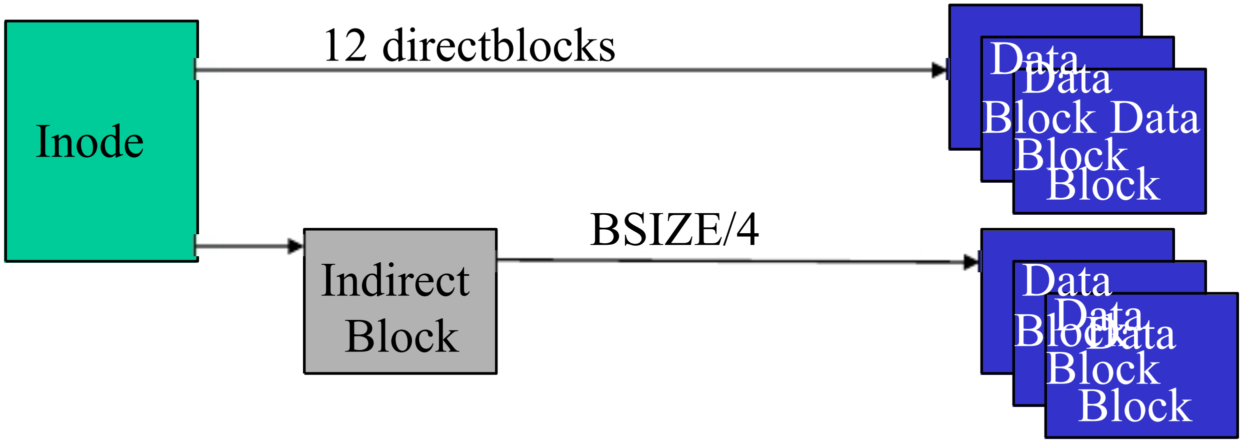
- bcache.lock: lock for entire buffer cache
- b->flags & B_BUSY: busy bit for each buffer
- Ensures that only one process can be touching a struct buf at any time

xv6 file system layout



- fs.h, fs.c, mkfs.c
- struct superblock

xv6 file and directory layout



- NDIRECT = 12
- NINDIRECT = BSIZE/4 = 128
- struct dinode in fs.h, struct inode in file.h
- struct dirent in fs.h

xv6 block operations

- readsb(): read on-disk super block into in-mem super block

- bzero(): zero a block
- balloc(): allocate a block, set bitmap
- bfree(): free a block, clear bitmap

xv6 inode operations

- bmap(): map data block number to disk block number
- itrunc(): resize inode data
- ialloc(): allocate a new inode
- iupdate(): update information in inode

xv6 inode synchronization operations

- iget(): find in-memory inode from inode cache and bump reference count
- idup(): bump reference count
- iput(): decrement reference count and truncate inode if necessary
- ilock(): lock inode for read and write by setting I_BUSY flag
- iunlock(): unlock inode by clearing I_BUSY flag; must call iunlock() before iput()

xv6 file system calls

- file.c, sysfile.c

- Examples file system calls
  - sys_open()
  - sys_mkdir()

- Path resolution
  - namei()
  - nameiparent()

Lab 12 assignment

- Read Chapter 5 and write a report explaining, in your own words,
  - How xv6's buffer cache works
  - How xv6's journaling (log) works
  - How what is inode and how it is implemented in xv6
- Add the following line at the beginning of the log_write() function in log.c:

  cprintf("log_write %d\n", b->sector);

  - Try the commands '**$ echo > a**', '**$ echo x > a**', and '**$ rm a**'. Take a snapshot of each command result. Explain, in detail, what is happening, **line by line**, of the outcome. (It is very important that you understand why each line is printed!)
- Make your report file as lab12.pdf containing answers to above
- Submit your pdf format report via Canvas

TCNJ THE COLLEGE OF NEW JERSEY