

Lab 4

More File Stuff



Think back to lab 1...

- Files were read-only

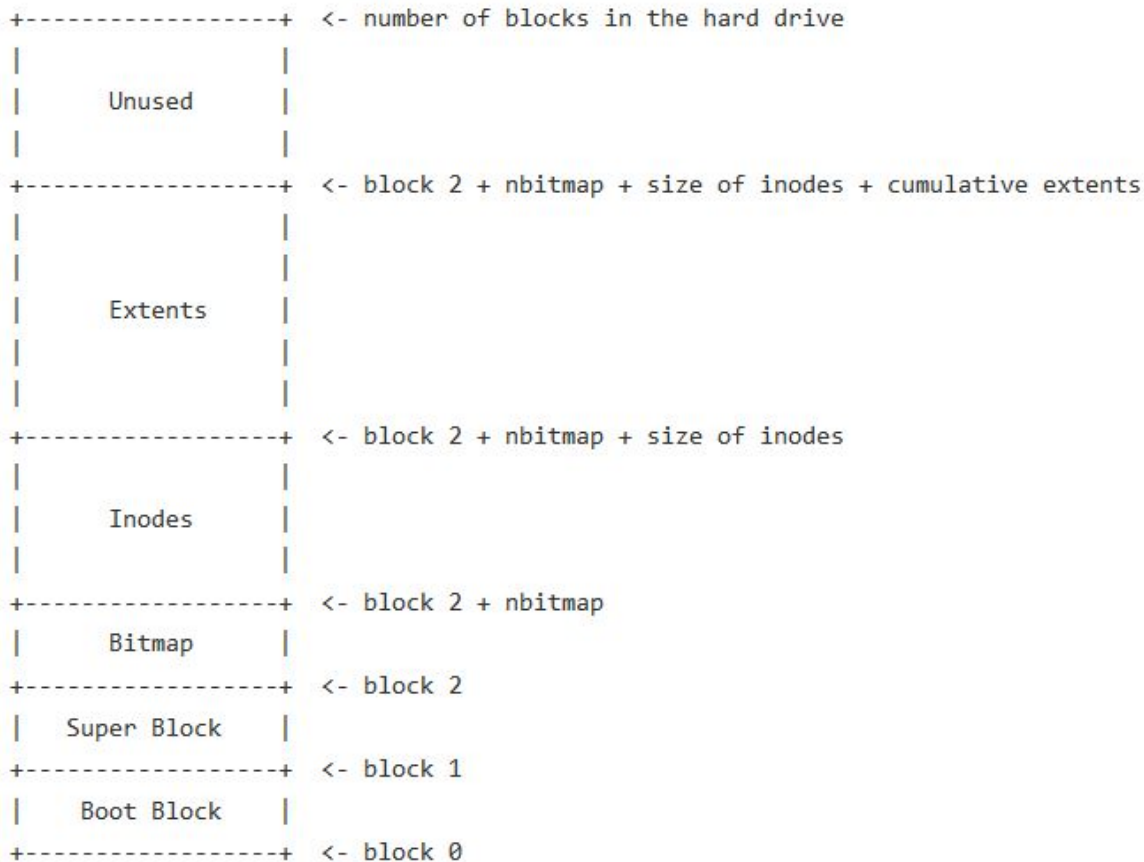
BUT NOW...

Lab 4: Two parts

- 1) Implement file-write
 - a) Actually mess around with the file system!
- 2) Make the filesystem crash-safe
 - a) Probably implement some form of logging

Prologue: Struct Tour

Disk Layout



Boot Block:

Initialization data for bootloader

Super Block:

Describe how disk is formatted
(layout type, region size, etc)

Bitmap:

Track which disk blocks are used

Inodes:

Keep inode for each file
(file metadata)

Extents:

Where all the actual file data is
stored

struct superblock - inc/fs.h

```
16 // mkfs computes the super block and builds an initial file system. The
17 // super block describes the disk layout:
18 struct superblock {
19     uint size;        // Size of file system image (blocks)
20     uint nblocks;    // Number of data blocks
21     uint bmapstart;  // Block number of first free map block
22     uint inodestart; // Block number of the start of inode file
23 };
```

Parameters for xk's file system superblock

-> Much simpler than what a "real" filesystem like FFS or NTFS would require

struct dinode - inc/fs.h

```
25 // On-disk inode structure
26 struct dinode {
27     short type;           // File type
28     short devid;        // Device number (T_DEV only)
29     uint size;          // Size of file (bytes)
30     struct extent data; // Data blocks of file on disk
31     char pad[46];       // So disk inodes fit contiguosly in a block
32 };
```

You used “inodes” in lab 1 -- it’s the data about files.

On disk, we represent inodes as “dinodes” (disk inode), which include things like padding and omit runtime data like locks that the in-memory-inode has

- dinodes are read from disk
- inodes are loaded from dinodes

struct extent - inc/extent.h

```
3 // represents a contiguous block on disk of data
4 struct extent {
5     uint startblkno; // start block number
6     uint nblocks;    // n blocks following the start block
7 };
```


We need a way to keep track of where on disk files are stored.

For xk, we consider each file a contiguous region of blocks

Note, this is unlike FFS with indirect pages and references to individual blocks

Padding

```
25 // On-disk inode structure
26 struct dinode {
27     short type;           // File type
28     short devid;        // Device number (T DEV only)
29     uint size;          // represents a contiguous block on disk of data
30     struct extent data;
31     char pad[46];
32 };
```



We use padding to ensure that there is always a whole number of dinodes on a page.
Size = 2 + 2 + 4 + (4 + 4) + 46 = 62 + 2 = 64 (size of all fields plus struct padding - remember 351)

Implementing the inode region is a lot easier if there's never going to be a dinode split across pages
 $\text{BLOCK_SIZE \% sizeof(dinode)}$ must be 0

inodes and dinodes

```
// in-memory copy of an inode
struct inode {
    uint dev; // Device number
    uint inum; // Inode number
    int ref; // Reference count
    int valid; // Flag for if node is valid
    struct sleeplock lock;

    short type; // copy of disk inode
    short devid;
    uint size;
    struct extent data;
};
```

```
25 // On-disk inode structure
26 struct dinode {
27     short type; // File
28     short devid; // Device
29     uint size; // Size
30     struct extent data; // Data
31     char pad[46]; // So di
32 };
```

- If you update dinode you'll want to update inode too
- If you update dinode you'll need to adjust padding

Part A: File Write

Write

- Modify **writei** in kernel/fs.c so that inodes can be used to write back to disk
- Use **bread, bwrite, brelse**
 - Note that you can't read/write with the disk in quantities smaller than a page
- Look at **readi** as your example

- Also modify **file_open** to allow writing (and patch lab1 tests if you want to)

Append

- If you write at the end of a file, its size should grow.
- Somehow you'll need extra space to write into
 - Option 1 (fancy): Update inode to have multiple extents (out of space? Add a new extent)
 - Option 2 (easy): Just allocate 20 blocks per file when they're created
 - This will require updating the "mkfs.c" file which builds the filesystem to allocate extra space for files which already exist.

Create

Be able to create a new file when **O_CREATE** is passed to **file_open**

Multiple parts!

- 1) Create a new inode on disk
- 2) Update root directory to reference this new inode (nested dirs not required)
- 3) Find extents for inode & update bitmap
- 4) Add the new inode to the inode region

Note: you'll probably need to update `mkfs.c` to allocate extra space for the inode region.

File *deletion* is not required!

Lab4test_a

should now pass

Lab4test_b

should also pass if your
file concurrency is good

Part B: Crash Safety

Suppose we try to append...

Simple example: say we have “file.txt” which is 256 bytes long.
We try to append 50 bytes to this file.

We need two block writes

- 1) The inode region, updating the size of this file to 306 bytes
- 2) The actual file data on disk (the new 50 bytes)

But this entire operation is not atomic

- Invoke “file_write”
- Compute new file size
- Update size on disk (inode region)
- Update file contents in memory
- ~~Write the new file contents to disk~~ **CRASH**

Reboot the system... We think “file.txt” is 306 bytes long, but the last 50 bytes are garbage, not what we tried to write!

The goal: make multi-block operations atomic

How?

Journaling.

The big idea: write changed blocks into a **log** rather than the final slot on disk. Once all blocks are written to the log, copy them into the actual destination.

- If the system crashes before all blocks written, trash the log - fs consistent!
- If the system crashes after all blocks in log, redo the copying - fs consistent!

The protocol, in more detail

For any operation which must write multiple disk blocks atomically...

- 1) Clear out any data currently in the log
- 2) Write new blocks into the log, rather than target place. Track what target is.
- 3) Once all blocks are in the log, mark the log as “committed”
- 4) Copy files from the log to where they should be

On system boot, check the log. If not committed, do nothing. If so, redo the copy (copy is idempotent)

Step 1: “log_begin()”

Make sure the log is cleared



The Log



The Disk
(Main Storage)

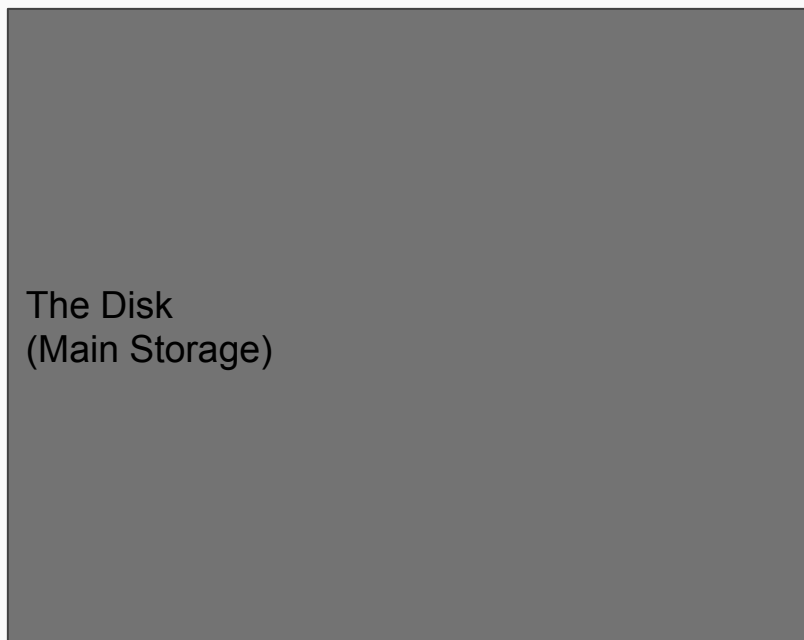
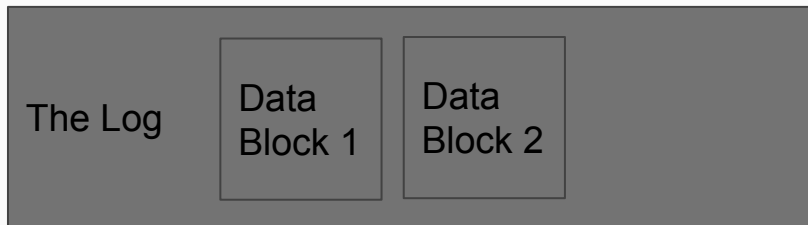
Step 2: “bwrite(data block 1)”

Write into the log, rather than the place in the inode/extents region we want it to go



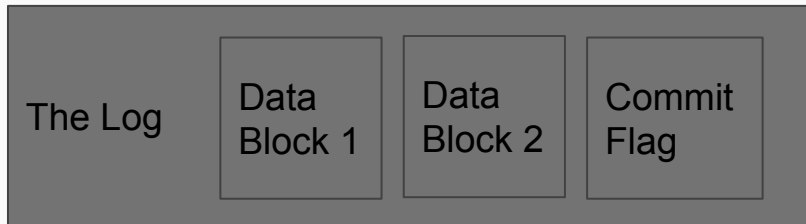
Step 3: “bwrite(data block 2)”

Write into the log, rather than the place in the inode/extents region we want it to go




Step 4: “log_commit()” [1]

Mark the log as “committed”



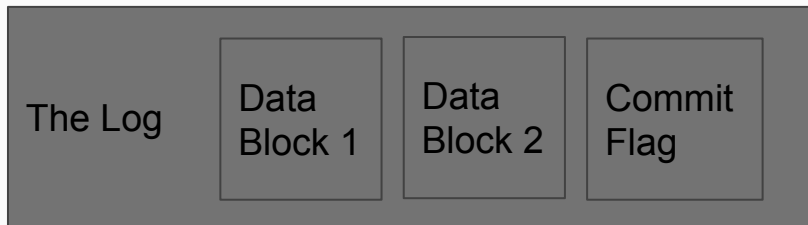
The Disk
(Main Storage)



A large, empty gray rectangular area representing the disk or main storage.

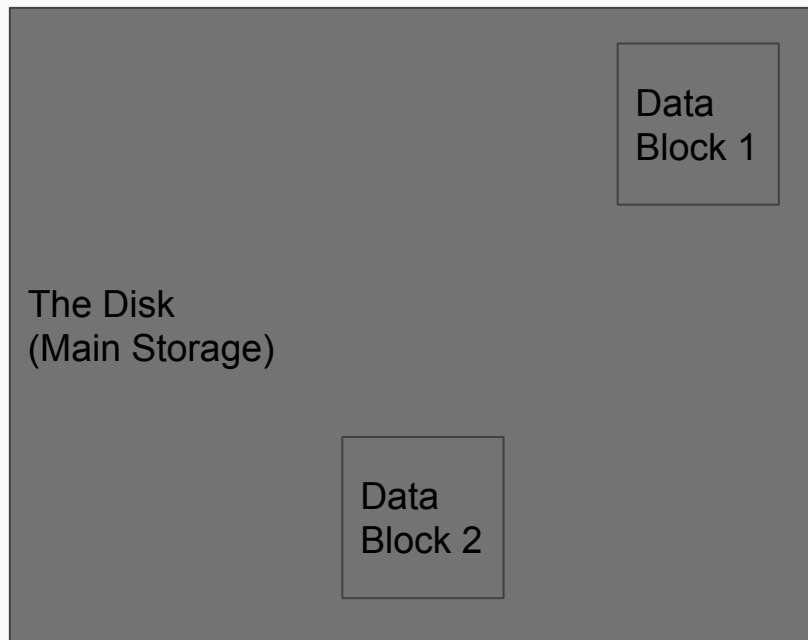
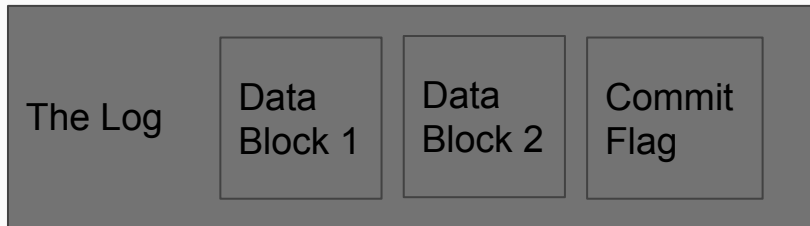
Step 5: “log_commit()” [2]

Copy the first block from log onto disk



Step 6: “log_commit()” [3]

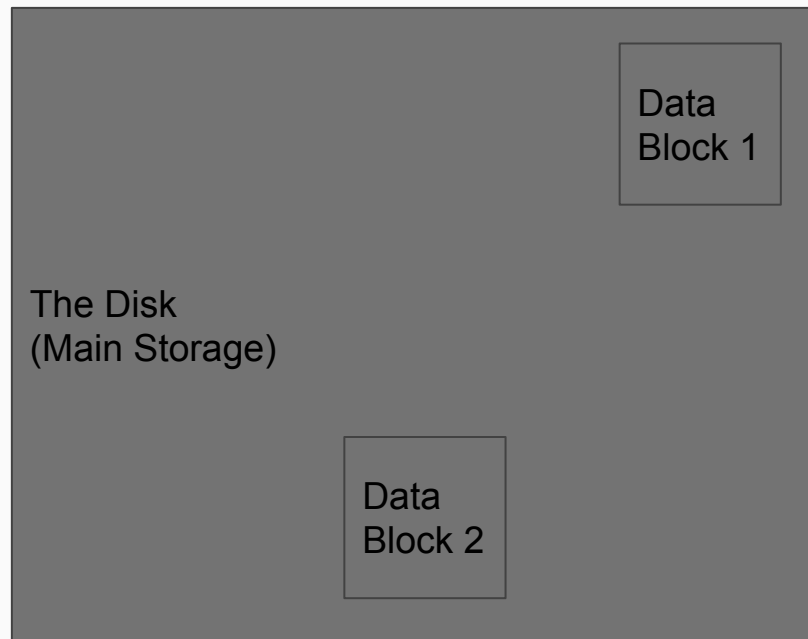
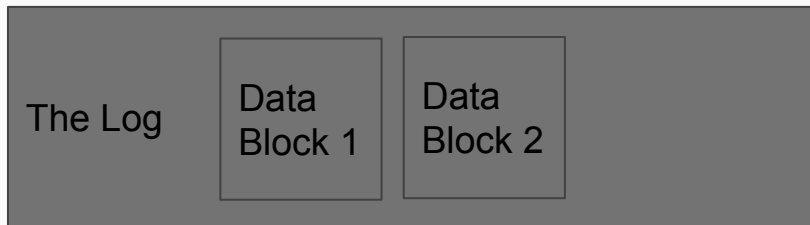
Copy the second block from log onto disk



Done!

We have both data blocks 1 and 2 on disk - everything was successful.

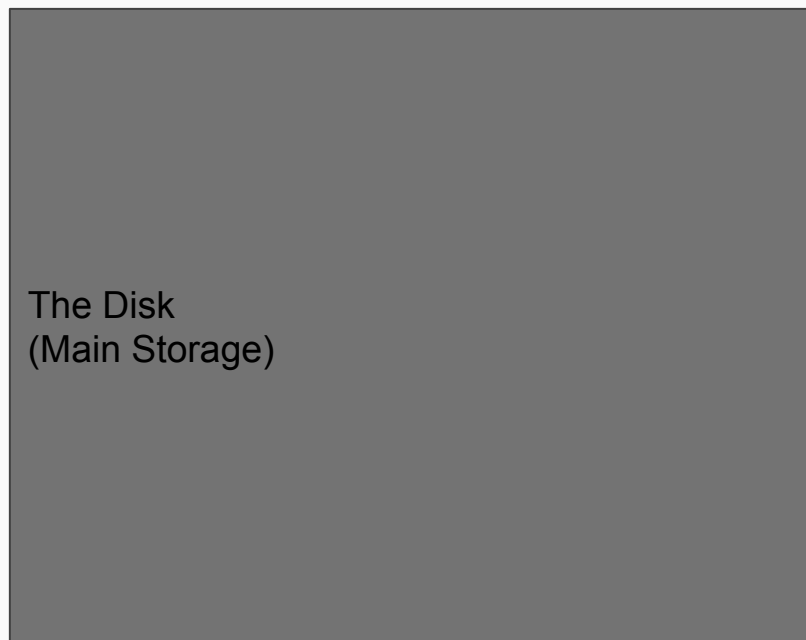
For efficiency, we can zero out the commit flag so the system doesn't try to redo this



Example: ~~Step 3: "bwrite(data block 2)~~ CRASH

On reboot...

There's no commit in the log, so we should *not* copy anything to the disk



Example: ~~Step 6: “log_commit()” [3]~~ CRASH

On reboot, we see that there *is* a commit flag

We can then copy block 1 and 2 to disk --
even though DB1 *was* already copied over,
overwriting it with the same data is fine

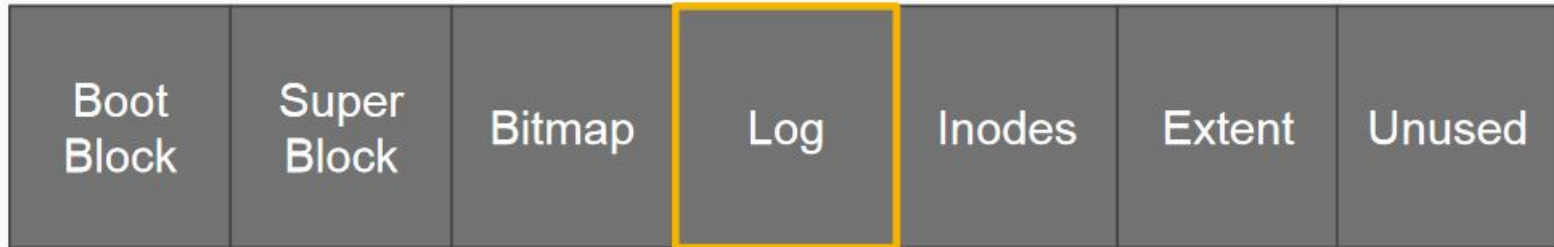


Where to Log?

It's just blocks on disk, so you can put it anywhere you want (within reason)

After-bitmap, before-inodes is a pretty good place

You'll need to update the superblock struct and mkfs.c



Context

KERNEL LAND

<i>Userland</i>	KERNEL LAND			
System Calls	File API	Inode API	Block API	IDE API
write() open()	filewrite() fileappend() filecreate()	writei() readi()	bread() bwrite() brelse()	iderw()

Questions?