Lab 4 Intro

Quick notes

- Please pull the github repo before working on lab4 - there were some updates last night after the ed announcement
- Lab3 due tomorrow
- Lab4 design doc due next wednesday

Think Back To Lab 1...

- Files were read-only
  - open denies O_WRONLY and O_RDWR flag for files

But For Lab4

Lab 4: Two parts

1) Make the filesystem writable
   a) remove file write restriction (no need to check T_DEV, will fail lab1test and that's fine)
   b) support file overwrite (write to existing blocks, implement writei)
   c) change the inode layout to support file extension
   d) support file creation and deletion (allocate/free inode, adjust data in root directory)
2) Make the filesystem crash-safe
   a) implement some form of logging

Prologue: Tour of the xk Storage Layer

Major Layers

- File System, Files, and Directories (fs.h/fs.c, file.h/file.c, extent.h)
  - inodes: struct dinode (disk inode, persistent), struct inode (in memory copy of the inode)
  - inode file: special file where the file data is a list of disk inodes
  - extents: how inode tracks its data location
  - bitmap: used to keep track of free and used blocks on disk
- Block Cache/Buffer Cache (bio.c)
  - brings block/sector into memory and manages them (evict, writeback)
  - struct buf: metadata for managing buffer, buf->data = sector data
  - APIs: bread (brings in the sector into memory, locks the buffer, no one else can access the block), bwrite (marks the buffer dirty), brelse (releases the buffer)
- IDE Connector (ide.c)
  - block interface, no need to modify it, can read if curious

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

FS: Superblock (inc/fs.h)

```
12  // Disk layout:
13  // [ boot block | super block | free bit map |
14  //                                          inode file | data blocks]
15  //
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  };
```

xk's file system superblock, track metadata for the file system
much simpler than what a "real" filesystem like FFS or NTFS would need

FS: 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  };
```

Disk inode

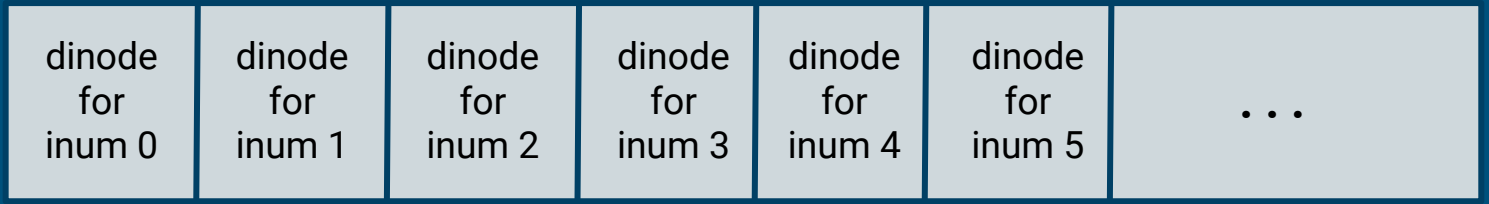
- metadata for files/directories
- persistent, lives on disk, cannot have any pointer fields (why?)
- defines data layout, currently only supports 1 extent
  - should be modified to support multiple extents (can cap at 30)
  - sizeof(struct dinode) must be a power of 2 (currently 64), why?

FS: dinode (inc/fs.h)

- dinodes are stored in the inodefile
  - reuse dinodes if existing ones are free
  - can create more disk inode by appending to the inodefile
- How do you know a dinode is free?
  - Up to you to decide:
    - reuse old fields: type = -1 or size = -1
    - create a new flag in dinode: probably need to pad the struct anyway
  - make sure you mark it as unused when deleting the file in the same way

FS: inodefile

- Special file for storing dinodes
  - data is an array of dinodes
- Starts at sb.inodestart
- Where is the dinode for inodefile?
  - it's the first dinode in inodefile's data, inum = 0



inodefile data

FS: inode

```
// 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;
```

cached from dinode

In memory inode

- a cache copy of the disk inode & in memory bookkeeping
  - lock, refcount, valid (matters while running but not persistent)
- if you change your dinode, make sure to change inode as well
  - locki will synchronize the inode with dinode when inode->valid == 0

FS: inode

- Allocated when we need information from a disk inode
  - read disk inode read through read_dinode and cache info into the inode struct
  - in memory inodes live in icache.inode
- Changes to an inode are purely in memory
  - changes will not reflect on the cached dinode unless you issue a write to the dinode with updated information

FS: extent

```
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  };
```

- A way to track where data is 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
- Every extent tracks a contiguous chunk of sectors
  - startblkno: sector number of the beginning sector
  - nblocks: number of sectors
  - tracks sectors [startblkno, startblkno + nblocks)
- Multiple extents track multiple chunks of sectors

FS: bitmap (kernel/fs.c)

Bitmap sectors live on disk and track the usage information of all disk sectors

- sectors start at sb.bmapstart, all the way up to (not including) sb.inodestart
- If bit in sector is 0, the corresponding block is free
- If bit in sector is 1, the corresponding block is in use

- Existing xk API helps manage the bitmap for you

FS: bitmap (kernel/fs.c)

- balloc()
  - Allocates consecutive blocks for a given device
  - Panics when not enough consecutive blocks available
  - Does not guarantee that block contents have been zeroed
- bfree()
  - Frees consecutive blocks for a device
  - Will not free contiguous blocks belonging to different bitmap sectors
- bmark()
  - Used to mark bits of bitmap to represent free and used blocks

WARNING: these functions do not change the bitmap on disk. You will need to update them to do so.

FS: directories

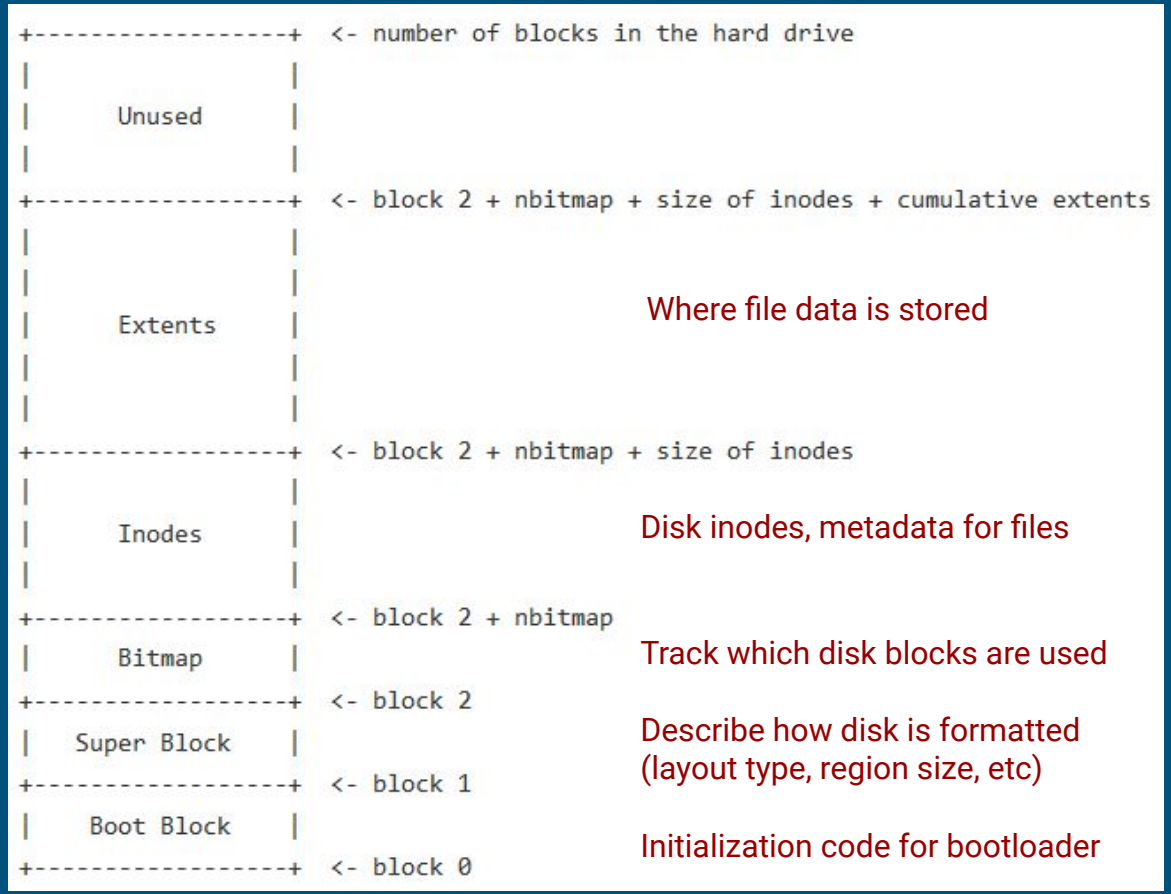
- Directories are like ordinary files (they have an inode associated with)
- Data is an array of directory entries (dirents)
- Dirent has two fields, name and inum

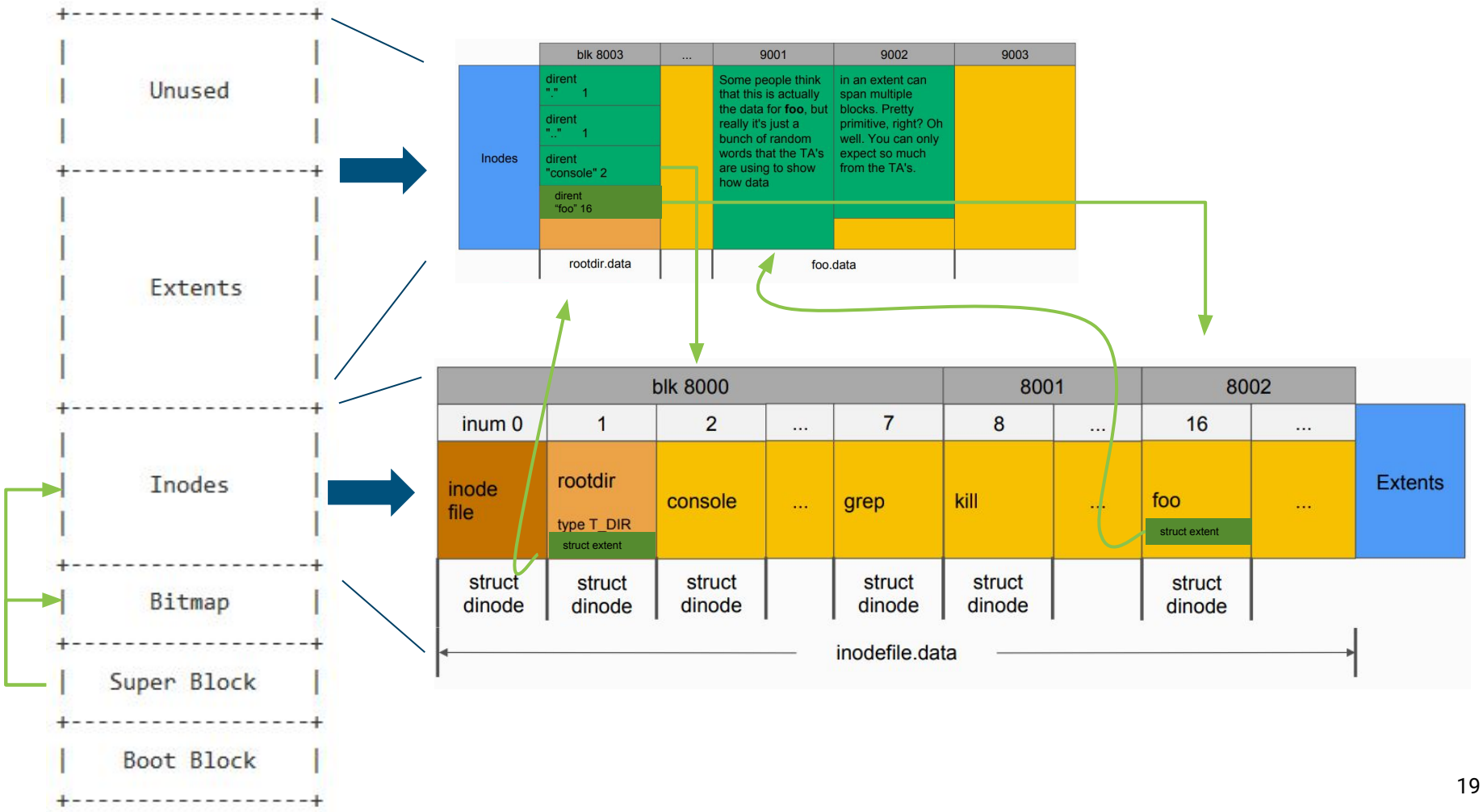
```
46  struct dirent {
47    ushort inum;
48    char name[DIRSIZ];
49  };
```

Initial Disk Layout

How things are stored on disk

mkfs.c (not an xk program) writes the initial disk image following this layout




Unused
Extents
Inodes
Bitmap
Super Block
Boot Block
blk 8003
...
9001
9002
9003
Inodes
dirent "." 1
dirent ".." 1
dirent "console" 2
dirent "foo" 16
Some people think that this is actually the data for foo, but really it's just a bunch of random words that the TA's are using to show how data
in an extent can span multiple blocks. Pretty primitive, right? Oh well. You can only expect so much from the TA's.
rootdir.data
foo.data
blk 8000
8001
8002
inum 0
1
2
...
7
8
...
16
...
inode file
rootdir
type T_DIR
struct extent
console
...
grep
kill
...
foo
struct extent
...
Extents
struct dinode
struct dinode
struct dinode
struct dinode
struct dinode
struct dinode
inodefile.data

Part A: Writable FileSys

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 block**
- 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
  - Update the dinode with new size
- Somehow you'll need extra space to write into
  - You can use the bitmap to find free blocks
  - Update dinode to allow for multiple extents (can cap at 30)

Create

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

Multiple parts!

1. Create a new dinode by appending to inodefile
2. Update root directory to reference this new dinode (nested dirs not required)
   - directory is just a special file that contains a list of `struct dirent`
3. If needed, update bitmap to reflect the newly allocated dinode

Delete

- unlink(char* path) system call
  - If path exists and no open references to the file, delete from the file system*
    - Effectively undoing steps from file creation
  - Otherwise, error

- Supporting file deletion -> inodefile can be fragmented
  - You will need to ensure file creation can fill holes in the inodefile
- Update parent directory's dirents to reflect the deletion

*unlink in Linux will delete the name from the file system, but keep the file object in memory until all references close - not necessary for our purposes

Lab4test_a should now pass

Lab4test_b
should also pass if your file concurrency is good

Part C: Crash Safety

Suppose we try to append...

Simple example: say we have “file.txt” which is 512 bytes long.
We try to append 50 bytes to this file.

We need multiple block writes
1) The dinode of the file, updating the size of this file to 562 bytes
2) The added file data on disk (a new sector for the new 50 bytes)
2) The bitmap sector to reflect the newly allocated sector

But this entire operation is not atomic

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

When we reboot the system… We think “file.txt” is 562 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 data 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)

Any questions?

Additional Slides For Logging

Step 1: “log_begin()”

Make sure the log is cleared

The Log

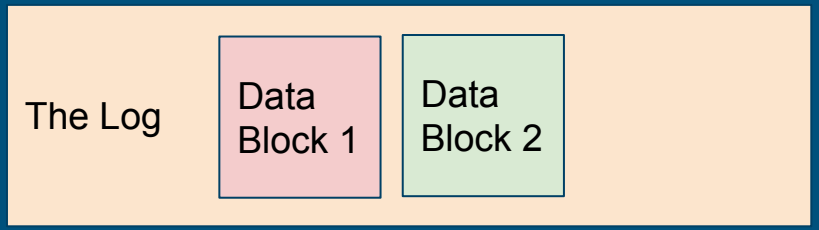
The Disk
(Main Storage)

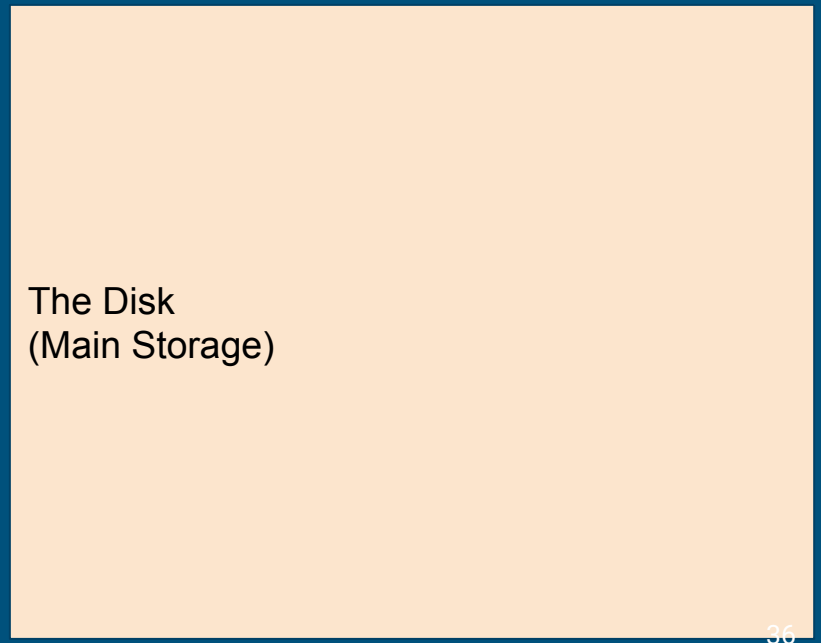
Step 2: “bwrite(data block 1)”

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

Step 3: “bwrite(data block 2)”

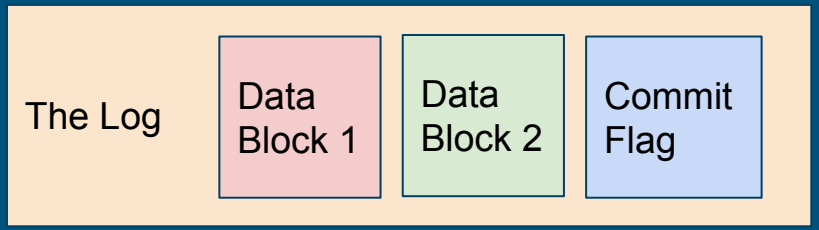
Write into the log, rather than the place in the inode/extents region we want it to go





Step 4: “log_commit()” [1]

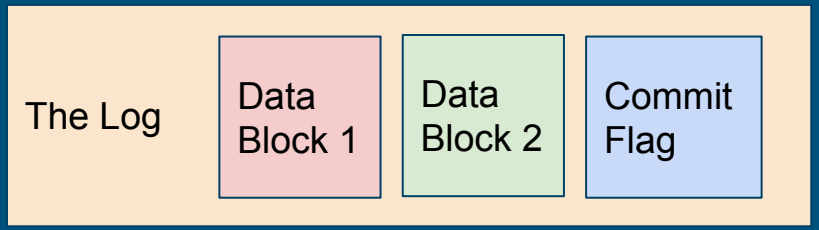
Mark the log as “committed”

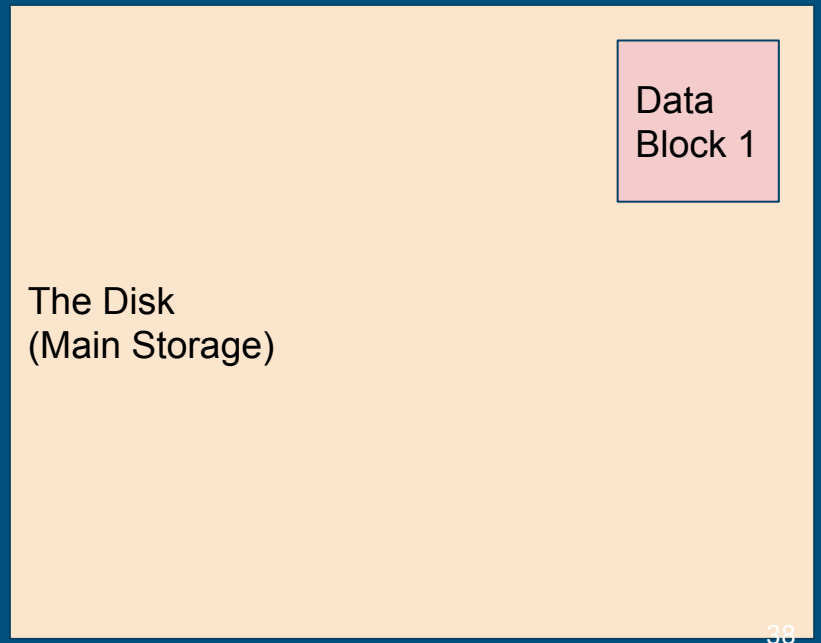


The Disk
(Main Storage)

Step 5: “log_commit()” [2]

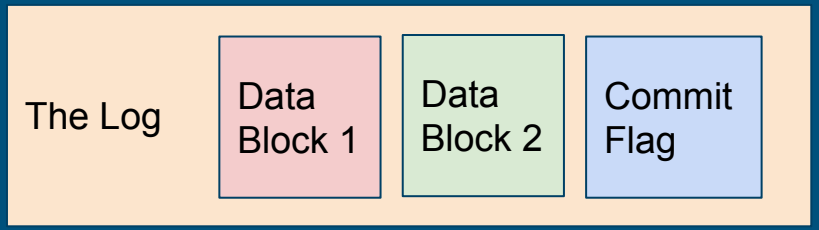
Copy the first block from log onto disk

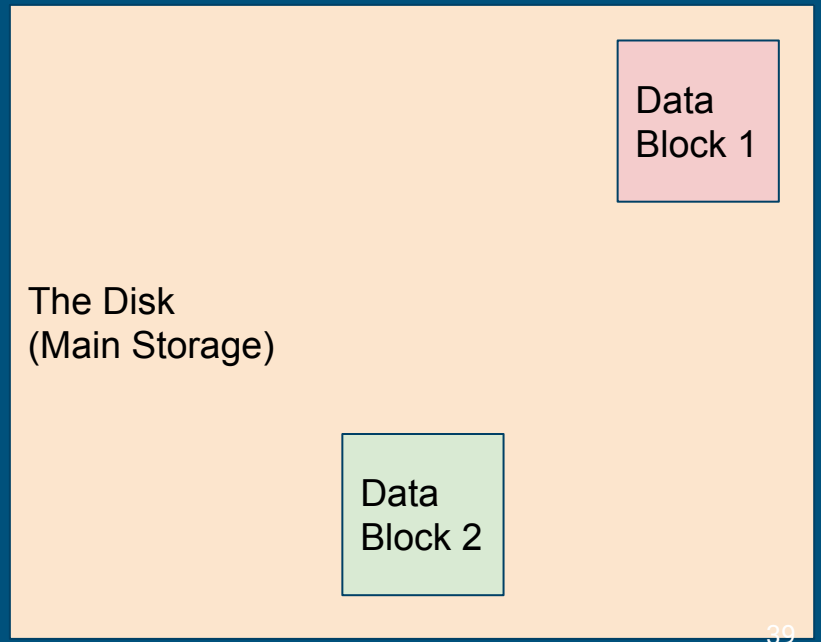




Step 6: “log_commit()” [3]

Copy the second block from log onto disk

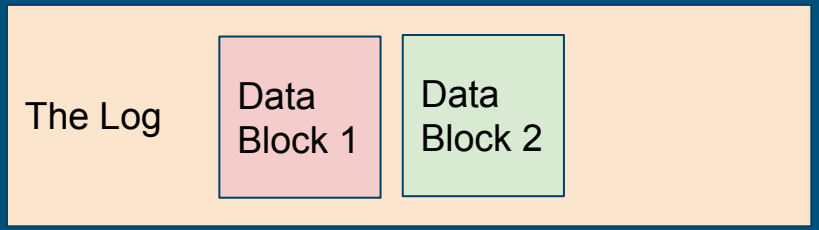


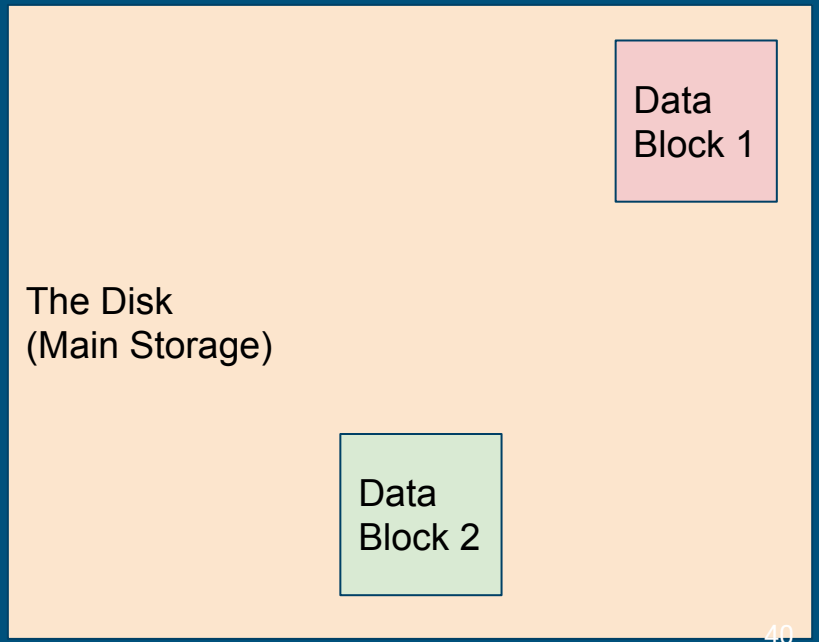


Done!

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

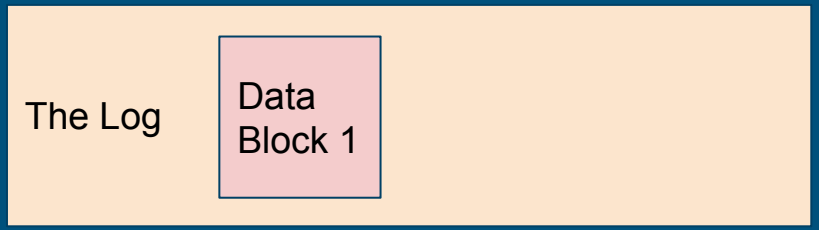
For efficiency, we can zero out the commit flag so the system doesn't try to redo this

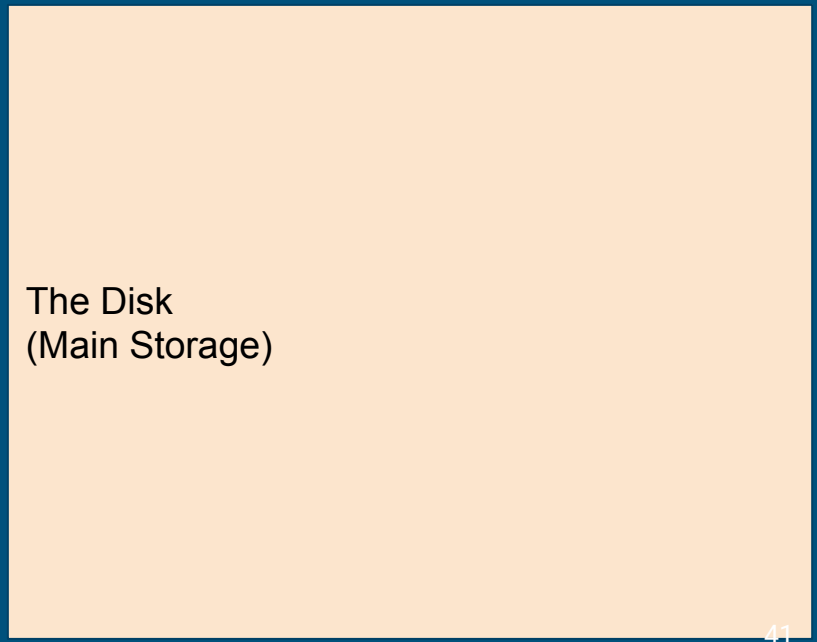




Example: ~~Step 3: “bwrite(data block 2)~~ CRASH

On reboot…
There’s no commit in the log, so we should *not* copy anything to the disk

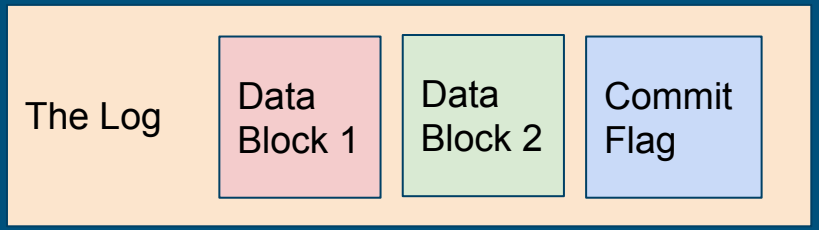


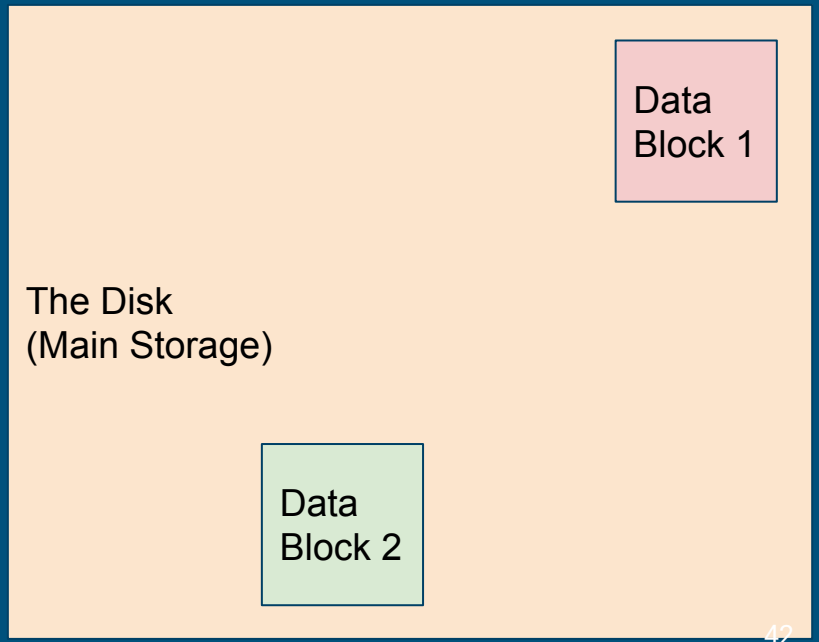


Example: ~~Step 6: “log_commit()” [3]~~ CRASH

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



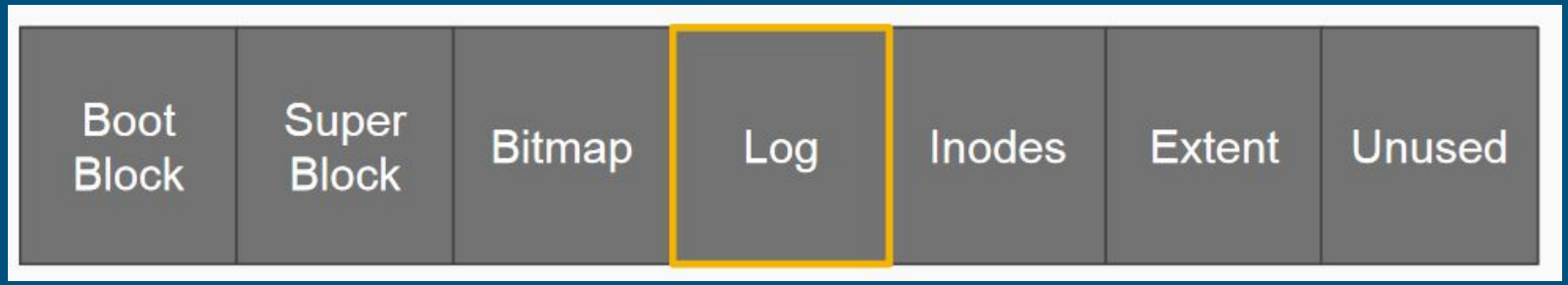


Where to Log?

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 (mkfs.c initializes the disk during the compiling process)



Log API

- The spec recommends designing an API for yourself for log operations:
  - **log_begin_tx()**: (optional) begin the process of a transaction
  - **log_write()**: wrapper function around normal block writes
  - **log_commit_tx()**: complete a transaction and write out the commit block
  - **log_recover()**: log playback when the system reboots and needs to check the log for disk consistency
    - Where/when should this be called? (Hint: inspect **kernel/fs.c**)