# 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: Disk Struct Tour

# Disk Layout

```
+------------------+  <- number of blocks in the hard drive
|                  |
|      Unused      |
|                  |
+------------------+  <- block 2 + nbitmap + size of inodes + cumulative extents
|                  |
|                  |
|      Extents     |
|                  |
|                  |
+------------------+  <- block 2 + nbitmap + size of inodes
|                  |
|      Inodes      |
|                  |
+------------------+  <- block 2 + nbitmap
|      Bitmap      |
+------------------+  <- block 2
|    Super Block   |
+------------------+  <- block 1
|    Boot Block    |
+------------------+  <- block 0
```

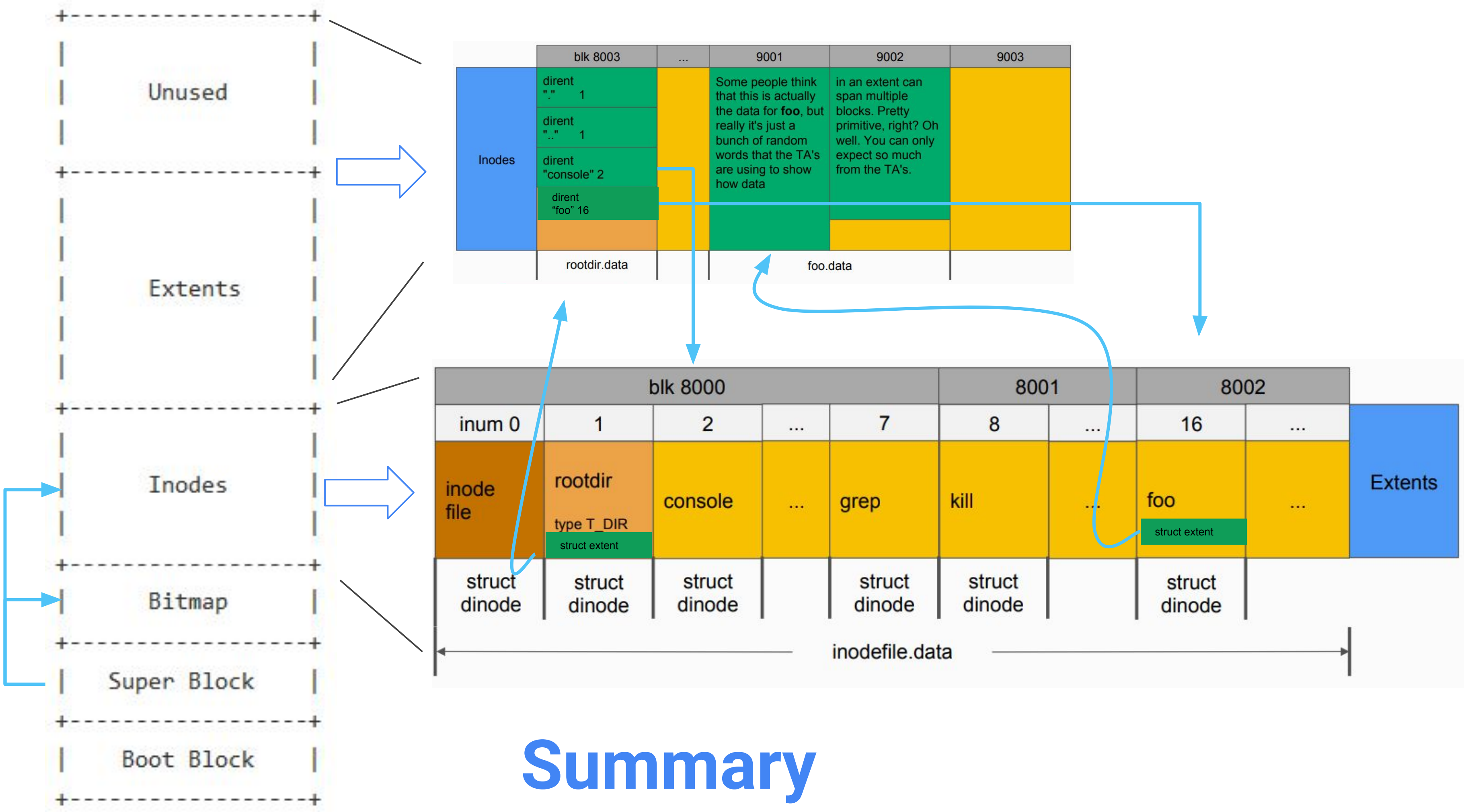
Boot Block:
  Initialization code for bootloader

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



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


# Summary

# struct superblock - inc/fs.h

```
// Disk layout:
// [ boot block | super block | free bit map |
//                                              inode file | data blocks]
//
// mkfs computes the super block and builds an initial file system. The
// super block describes the disk layout:
struct superblock {
 uint size;         // Size of file system image (blocks)
 uint nblocks;      // Number of data blocks
 uint bmapstart;    // Block number of first free map block
 uint inodestart;   // Block number of the start of inode file
};
```

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

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

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

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

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

```c
// On-disk inode structure
struct dinode {
  short type;           // File type
  short devid;          // Device number (T_DEV only)
  uint size;
  struct extent data;
  char pad[46];
};
```

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

Padding ensures there is always a whole number of dinodes on a block
(i.e. BLOCK_SIZE % sizeof(dinode) == 0, so no dinode is split between blocks)

Size = 2 + 2 + 4 + (4 + 4) + 46 = 62 **+ 2** = 64 (size of all fields plus struct padding - remember 351)

# inodes and dinodes

```c
// 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;
};
```

```c
// On-disk inode structure
struct dinode {
  short type;           // File
  short devid;          // Devic
  uint size;            // Size
  struct extent data;   // Data
  char pad[46];         // So di
};
```

- If you update dinode you'll want to update inode too
  - locki will synchronize the inode with dinode when inode->valid == 0
- If you add/delete fields in struct dinode, you'll need to adjust padding so that the whole size is a power of 2

# 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 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.
- Somehow you'll need extra space to write into
  - Option 1 (fancy): Update dinode 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)
    - The directory is just a special file that contains a list of `struct dirent`, pointing to files in the directory
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.

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

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

# 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

*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 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**

When we 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 Disk
(Main Storage)

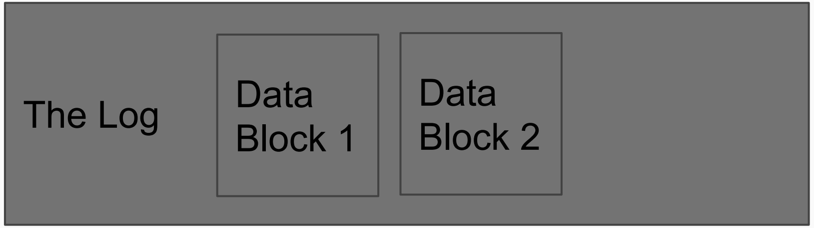
The Log

# Step 2: “bwrite(data block 1)”

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

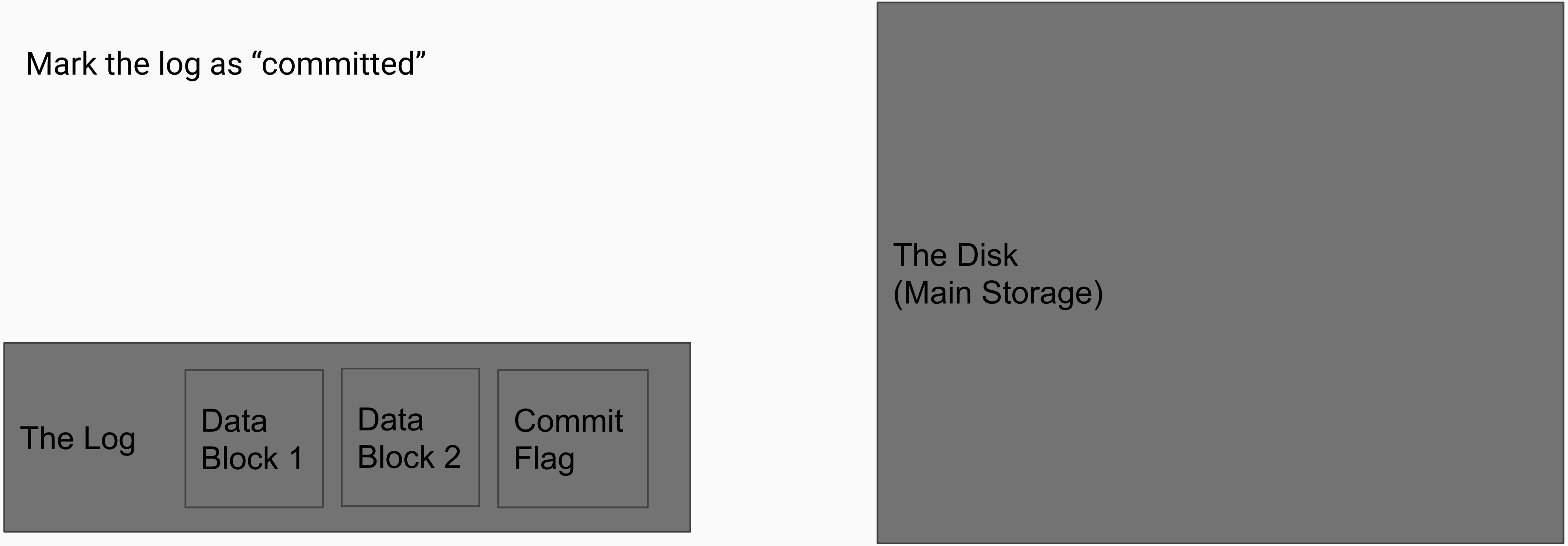

# Step 3: “bwrite(data block 2)”

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



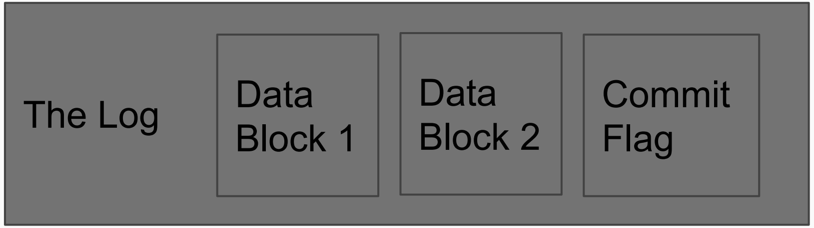

# Step 4: “log_commit()”  [1]

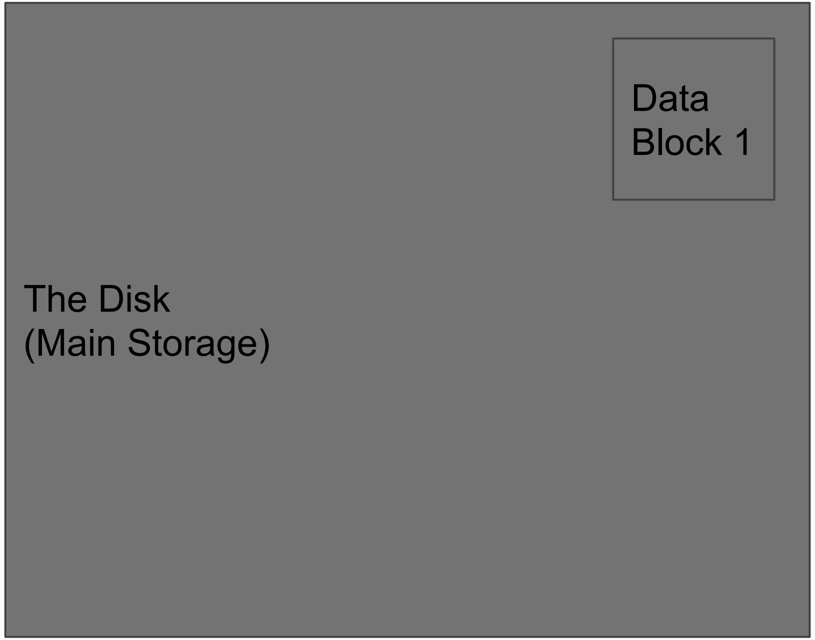
Mark the log as “committed”



# Step 5: “log_commit()” [2]

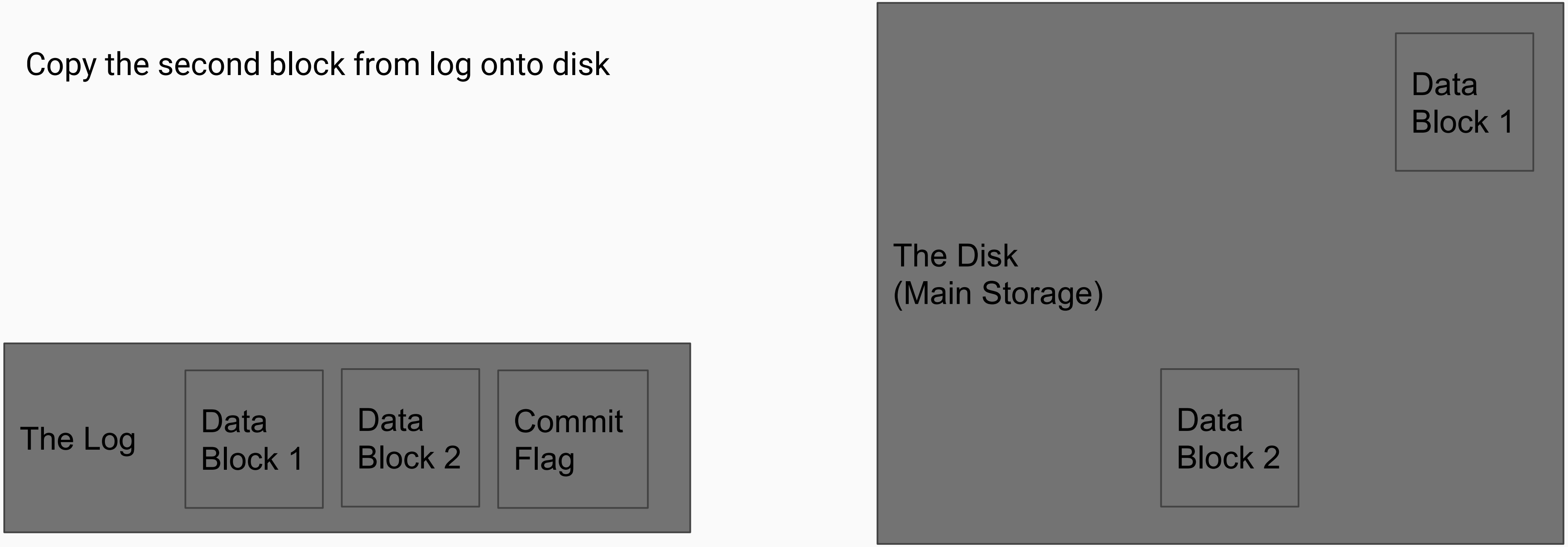
Copy the first block from log onto disk



# Step 6: “log_commit()” [3]

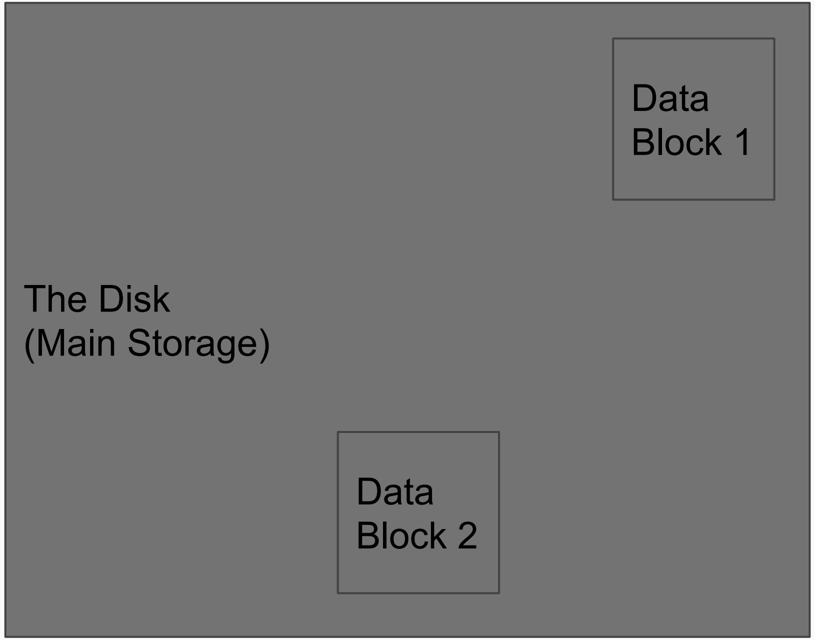
Copy the second block from log onto disk

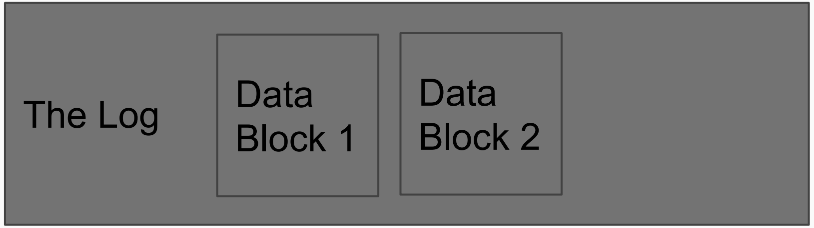


# Done!

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



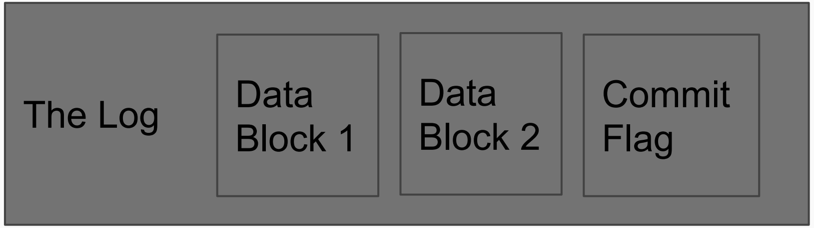
# Example: ~~Step 3: “bwrite(data block 2)~~ CRASH

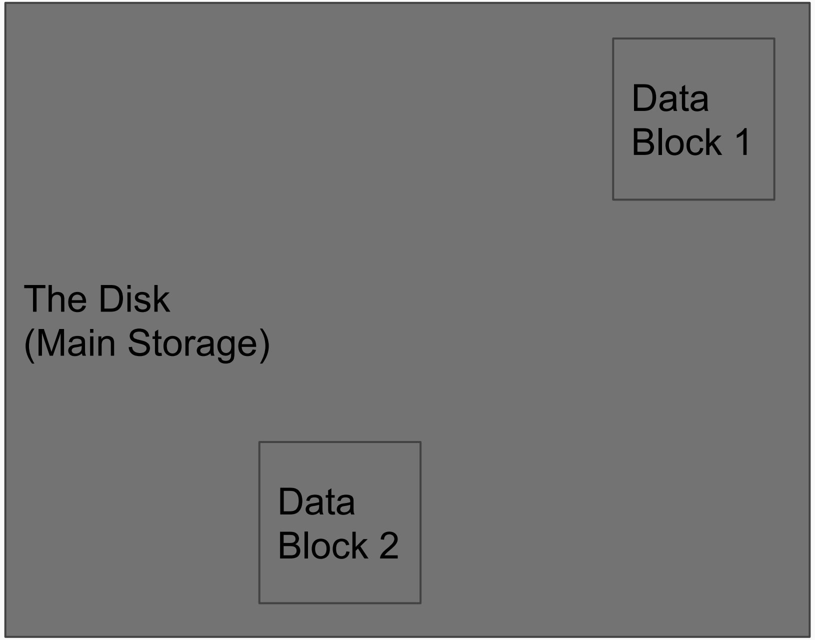
On reboot…
There’s no commit in the log, so we should *not* copy anything to the disk


# 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 (lab 1: File API. lab 4: Inode API)

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

# Questions?