Lab 4 Details

Part A: Writable FileSys

Dinode Data Layout

- need to change the data layout in struct dinode to support multiple extents
- once it's changed, need to update mkfs.c to reflect the new data layout
  - mkfs.c writes the initial file system image with user executables (labxtest, ls, exec, echo, cat..)
- after mkfs.c is updated to write the new dinode layout, update xk's file system operations to work with the new layout
  - readi & writei (base inode operations used by other fs functions)
  - would be good to have a function that takes an offset and finds the block #
  - can use in both readi and writei to find which block to read/write to

mkfs.c Explained

- it's NOT an xk program, it's compiled and runs on linux
- writes the initial file system image upon `make`
  - that's how we get content in our read only file system!
- understand the xk file system format

```
#include <inc/fs.h>
#include <inc/stat.h>
#include <inc/param.h>
```

includes from mkfs.c

macros defined by param.h

```
#define LOGSIZE (MAXOPBLOCKS * 3) // max data blocks in on-disk log
#define NBUF (MAXOPBLOCKS * 3)    // size of disk block cache
#define FSSIZE 50000              // size of file system in blocks
```

mkfs.c Explained

```
95    fsfd = open(argv[1], O_RDWR|O_CREAT|O_TRUNC, 0666);
```

creates the fs image

```
101   // 1 fs block = 1 disk sector
102   nmeta = 2 + nbitmap;
103   nblocks = FSSIZE - nmeta;
104
105   sb.size = xint(FSSIZE);
106   sb.nblocks = xint(nblocks);
107   sb.bmapstart = xint(2);
108   sb.inodestart = xint(2+nbitmap);
```

computes superblock info & writes it to the first sector of the fs image via `wsect(1,…)`

```
118   memmove(buf, &sb, sizeof(sb));
119   wsect(1, buf);
```

nmeta
# of reserved sectors
(boot, sb, & bitmap)

```
// Disk layout:
// [ boot block | sb block | free bit map | inode file start | data blocks ]
```

+ rest of the disk blocks

nbitmap
# of sectors taken up by bitmap

nblocks
number of non reserved blocks in the entire disk

mkfs.c Explained

```
124   inodefileino = ialloc(T_FILE);
125   assert(inodefileino == INODEFILEINO);
126
127   // setup inode file data area
128   rinode(inodefileino, &din);
129   din.data.startblkno = sb.inodestart;
130   inodefileblkn = inum_count/IPB;
131   if (inodefileblkn == 0 || (inum_count * sizeof(struct dinode) % BSIZE))
132     inodefileblkn++;
133   din.data.nblocks = xint(inodefileblkn);
134   din.size = xint(inum_count * sizeof(struct dinode));
135   winode(inodefileino, &din);
```

set up inodefile, inum_count = number of inodes needed for the initial fs image (user program binaries, inodefile itself, root dir)

ialloc allocates an empty inode, writes it to the fs image, returns the inode number
read in the inode with `rinode`, update/write the inode with `winode`

mkfs.c Explained

```
283  uint
284  ialloc(ushort type)
285  {
286    uint inum = freeinode++;
287    struct dinode din;
288
289    bzero(&din, sizeof(din));
290    din.type = xshort(type);
291    din.size = xint(0);
292    winode(inum, &din);
293    return inum;
294  }
```

by default, all fields of an inode is set to 0s, except for the type of the file

when you change the data layout, you may want to adjust ialloc to set default values (if non zero) for your new fields

mkfs.c Explained

Then mfks.c sets up inodes and data for root dir and program binaries

When you update the data layout of the disk inode, you should search for any reference to `data.startblkno` and `data.nblocks`,
and change it to work with your new data layout

If you do an array of extent, you can update these to refer to the first entry of your extent array

icache

```
struct {
 struct spinlock lock;
 struct inode inode[NINODE];
 struct inode inodefile;
} icache;
```

- reading from disk is slow, so we have an inode cache
- icache.inodefile: cache copy of inodefile's inode
- icache.inode: in-memory cache of opened inodes
  - iget gets a cached inode, allocates an entry in the cache if not found in the cache, updates inode's in memory refcount, since iget returns a handle to inode
  - irelease decrements refcount for an inode, refcount = 0 => evicted from the inode cache
  - merely a cache, does not contain all inodes from disk
- inode vs dinode
  - in memory vs on disk
  - need to keep them consistent (synchronized)
    - read_dinode (provided, used in locki) move data from disk to memory
    - write_dinode move data from memory to disk (not provided)

read_dinode

```
// Reads the dinode with the passed inum from the inode file.
// Threadsafe, will acquire sleeplock on inodefile inode if not held.
static void read_dinode(uint inum, struct dinode *dip) {
  int holding_inodefile_lock = holdingsleep(&icache.inodefile.lock);
  if (!holding_inodefile_lock)
    locki(&icache.inodefile);

  readi(&icache.inodefile, (char *)dip, INODEOFF(inum), sizeof(*dip));

  if (!holding_inodefile_lock)
    unlocki(&icache.inodefile);

}
```

```
// offset of inode in inodefile
#define INODEOFF(inum) ((inum) * sizeof(struct dinode))
```

- What does the function do?
  - Reads in struct dinode at index `inum` from inodefile
- Having a similar write_dinode() can be helpful (not provided in starter code)
  - When should we write dinode?

Helpful Functions

- mkfs.c:

```
nblocks = dinode.size/BSIZE + (dinode.size % BSIZE == 0 ? 0 : 1);
```

  ^ existing code to compute how many blocks are needed given a file size
  (feel free to do your own math, but just know that this math is correct)

- fs.c:
  - *read_dinode*: returns the dinode for a given inumber
  - *iget*: returns the in memory inode for a given inumber, inode may not have cached information from dinode
  - *locki*: locks the inode and guarantees that inode has info from dinode
  - *dirlookup*: finds the offset of a directory entry with matching name
    - skips over dirent with inum of 0

Tips

- write & append:
  - append = writing past end of file
  - if you are just overwriting an existing block of data, do you need to update its dinode?
  - what if you are appending more data to the file, do you need to update its dinode?
- balloc, bfree, bmark:
  - balloc and bfree only updates cached bitmap sectors in memory
  - this is done through setting the bp->flag dirty in bmark
  - if you want to **write bitmap sector back to disk**, you need to call bwrite yourself on the bp (handle to the changed bitmap sector)

Part C: Crash Safety

Journaling

For any operation which must write multiple disk blocks atomically…

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

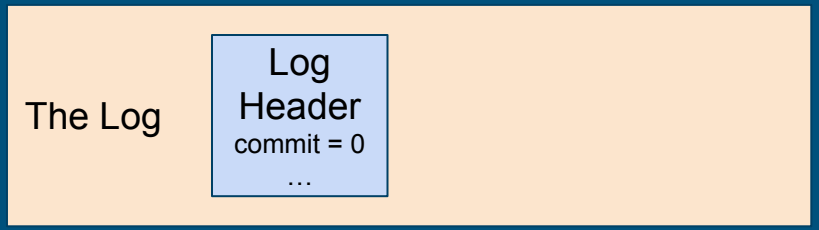
On system boot, check the log. If not committed, do nothing. If so, redo the copy (copy is idempotent)

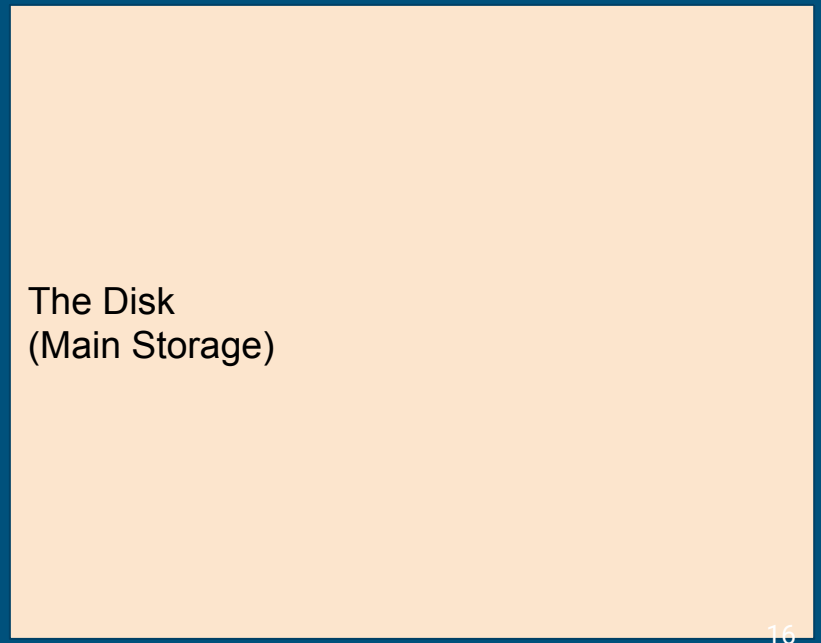
Log Header Format

- Log header = metadata for the log
  - a structure that lives on disk
  - should not exceed a sector
- Designed by you! Should at least track:
  - transaction status (committed or not)
  - usage status of log region
  - where to apply logged blocks

Step 1: “log_begin()”

Make sure the log is cleared

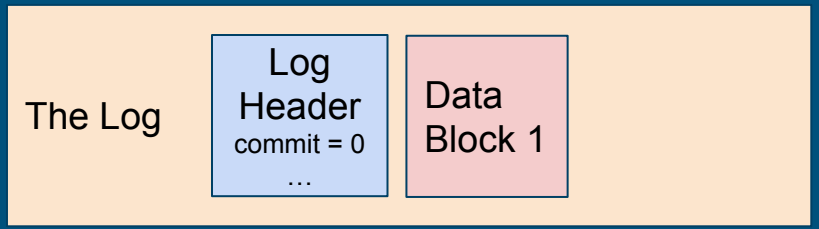




Step 2: “bwrite(data block 1)”

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

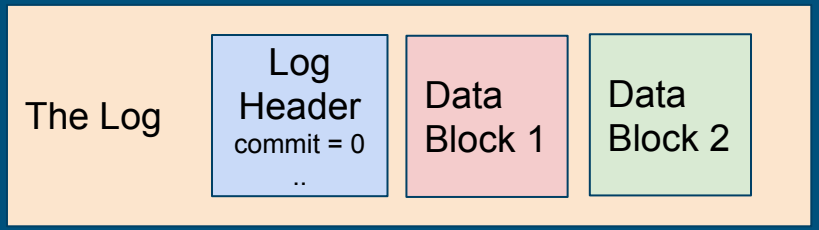
Also need to track the actual location of the data block so you know where to write logged blocks to on recovery!



The Disk
(Main Storage)

Step 3: “bwrite(data block 2)”

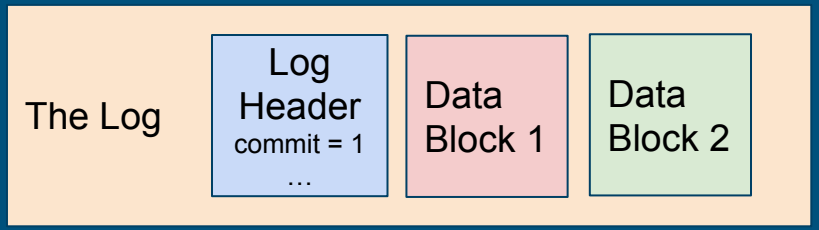
Write into the log, rather than the place in the inode/extents region we want it to go



The Disk
(Main Storage)

Step 4: “log_commit()” [1]

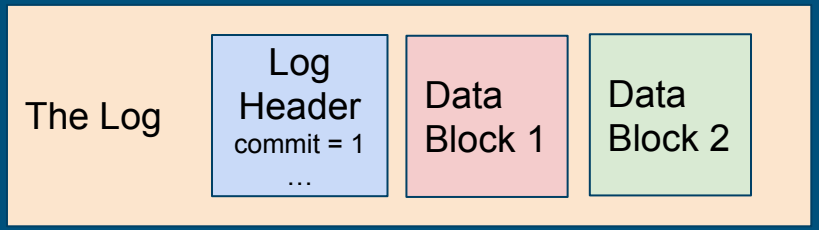
Mark the log as “committed”

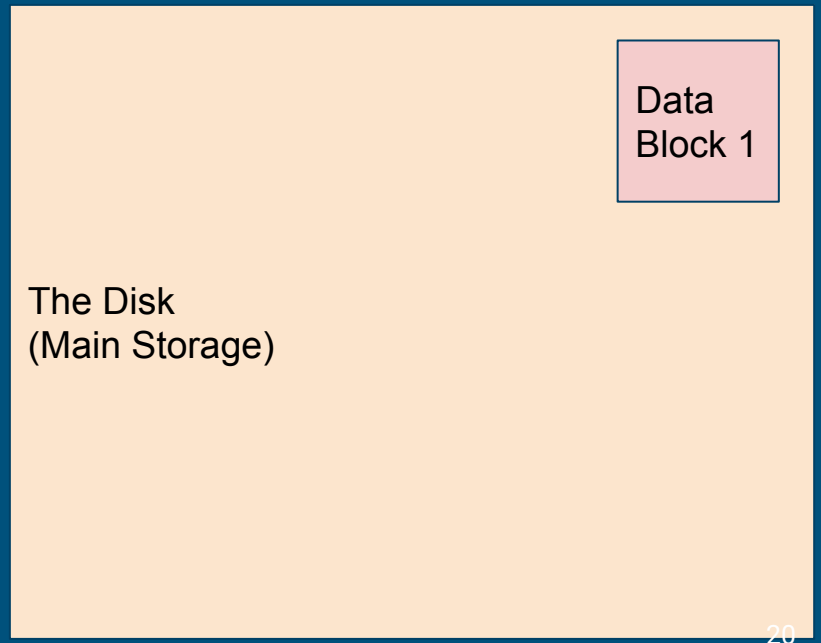


The Disk
(Main Storage)

Step 5: “log_commit()” [2]

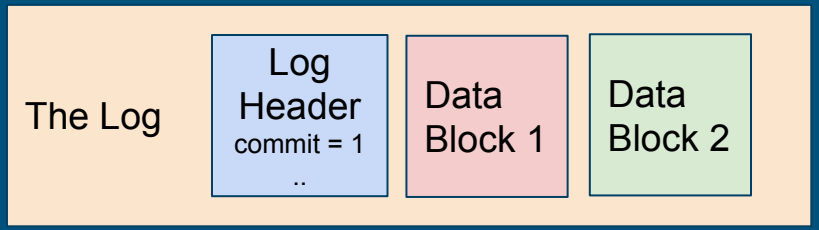
Copy the first block from log onto disk

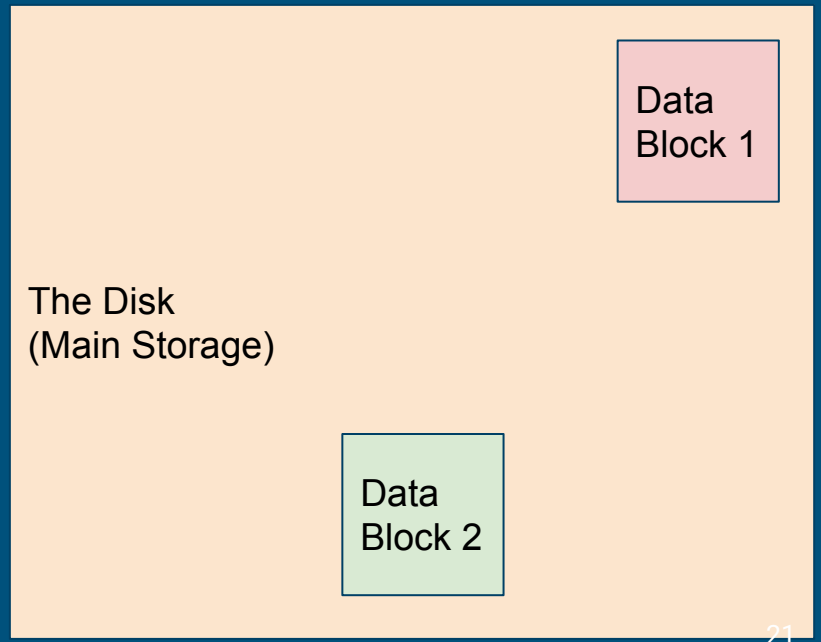




Step 6: “log_commit()” [3]

Copy the second block from log onto disk

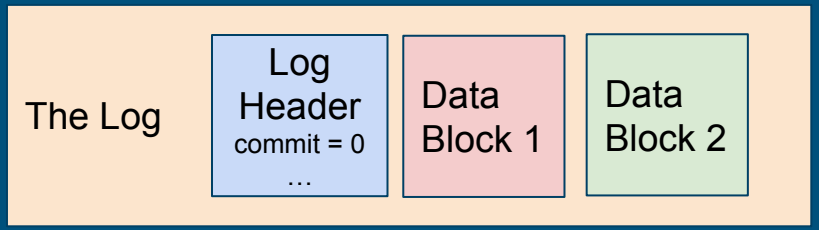


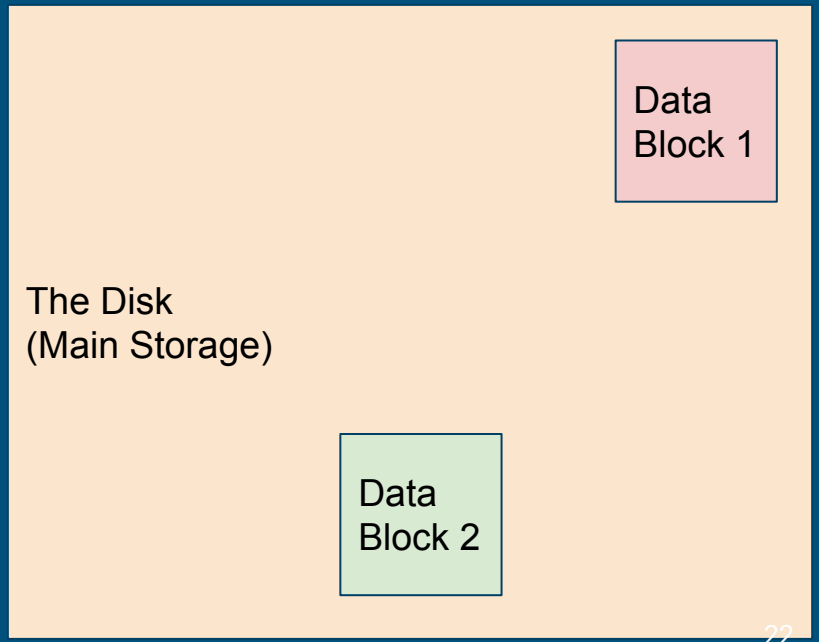


Done!

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

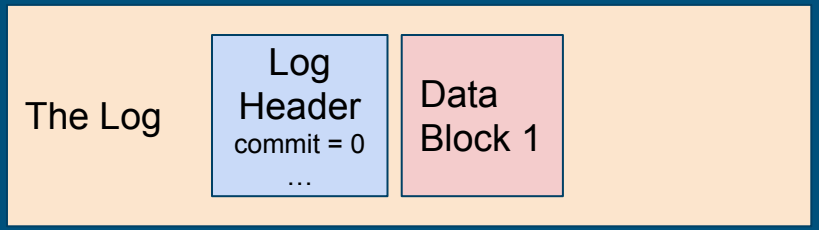
For efficiency, we can zero out the commit flag so the system doesn't try to redo this

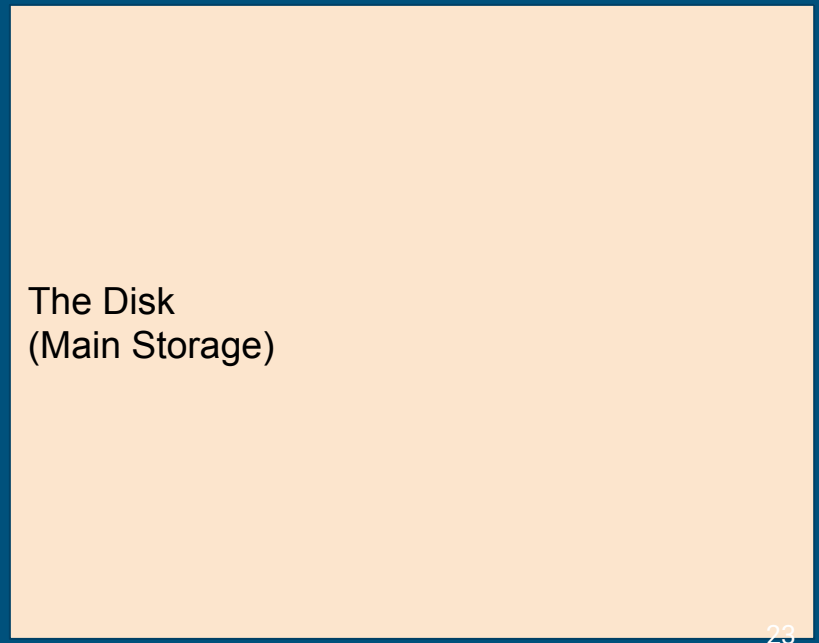




Example: before commit—CRASH

On reboot (start up)…
There's no commit in the log, so we should *not* copy anything to the disk

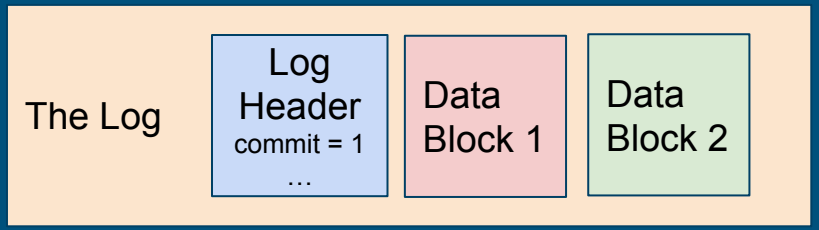


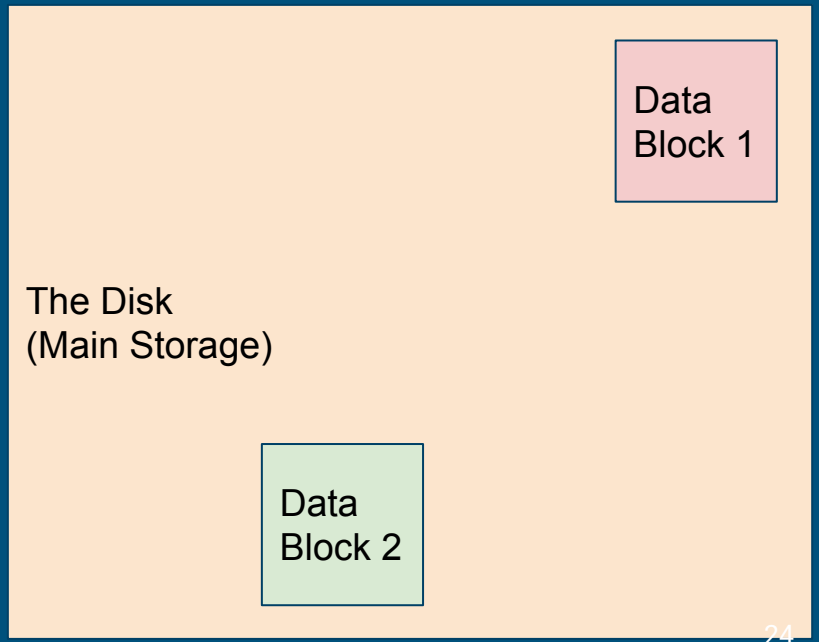


Example: after commit, before clear—CRASH

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



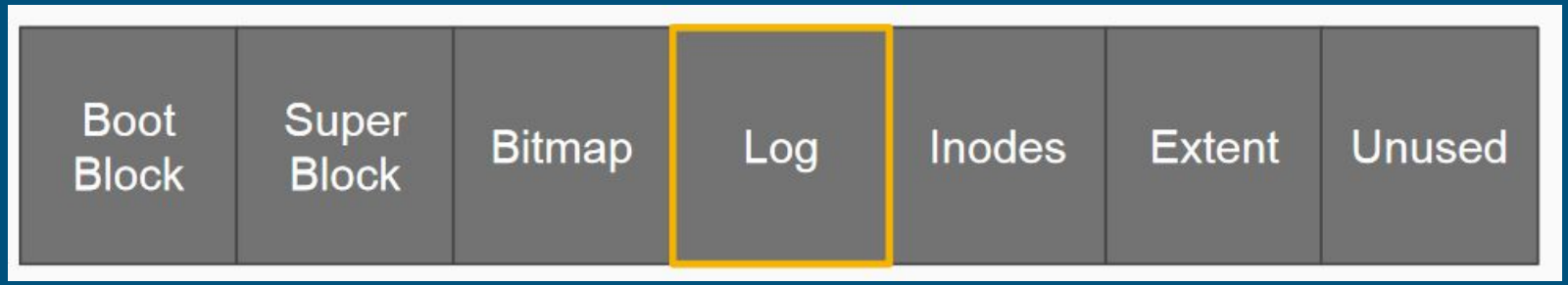


Where to Log?

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_apply()**: 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**)

What should log_write() do differently?

- log_write() intended to be a wrapper function for bwrite() operations
- Instead of writing the block to its location on disk, we want to:
  - Write the block information to our log region
  - Update the log header with the location of the block

What happens after log_write()?

- Once all block writes in transaction have called log_write(), log_commit_tx() will be called
- Commit
  - Flush commit block to disk
  - Copy blocks from previous log_writes to their actual location on disk
  - Reset commit flag