Lab 4 Intro

File System

Administrivia

- Lab 3 due tomorrow 2/28 (grace period applies)
- Lab 4 is out!
  - Lab 4 Design Doc **due Thursday (3/06)!** (No late days)
  - Lab 4 Code + Questions due 3/19 (No grace period)

Think Back to Lab 1...

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

Remember when we promised to eventually allow writing to the file system?

For Lab 4...

Your job is to:

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

Let’s go ahead and get started!

Part A: Writable FileSys

Quick Reminder!

Make sure to first modify file_open to allow opening files in write mode (and patch lab1 tests if you want to)

Note that Lab 1 tests pertaining to the read_only restrictions will no longer pass

The Existing Xk File System: Dinode & Extent

```
// On-disk inode structure
struct dinode {
  short type;          // File type (device, directory, regular fil
  short devid;         // Device number (T_DEV only)
  uint size;           // Size of file (bytes)
  struct extent data; // Data blocks of file on disk
  char pad[48];        // So disk inodes fit contiguosly in a block
};
```

```
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  };
```

- dinode: metadata for file & directories
- extent: tracks a contiguous region of data
- each file's data is tracked by a single extent

The Existing Xk File System: Dinode & Extent

```
// On-disk inode structure
struct dinode {
  short type;          // File type (device, directory, regular fil
  short devid;         // Device number (T_DEV only)
  uint size;           // Size of file (bytes)
  struct extent data; // Data blocks of file on disk
  char pad[48];        // So disk inodes fit contiguosly in a block
};
```

```
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  };
```

- disk inodes are stored on disk (persistent)
- can't store locks/memory pointers! Why?

The Existing Xk File System: Dinode & Extent

```
// On-disk inode structure
struct dinode {
  short type;           // File type (device, directory, regular fil
  short devid;          // Device number (T_DEV only)
  uint size;            // Size of file (bytes)
  struct extent data;   // Data blocks of file on disk
  char pad[48];         // So disk inodes fit contiguosly in a block
};
```

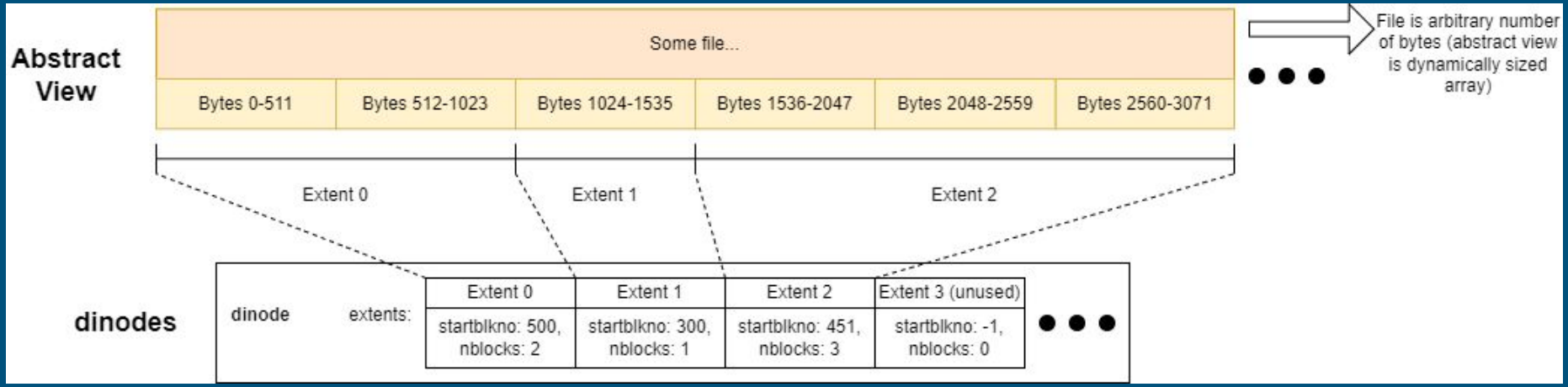
```
3  // represents a contiguous block on disk of data
4  struct extent {
5    uint startblkno; // start block number
6    uint nblocks;    // n blocks following the start block
7  };
```

- single extent => unable to track more data regions
- padding exists to ensure sizeof(dinode) is a power of 2

File Growth

- dinode currently tracks just 1 extent but you need to support multiple extents
  - initially only the first extent is filled with actual data, the rest are empty
  - upon each file extension
    - allocate data blocks and track it in the next free extent
    - should support at least 30 file extensions
  - each dinode must fit within a single sector
    - size should be a power of 2
    - padding the struct should help
- You must update other metadata within the dinode (size)

Example of Dinode With Multiple Extents



File Growth

- Ok that makes sense… but
  - what are the side effects of changing the dinode definition?
  - upon file growth, how do we update the disk inode?
  - upon file growth, how do we allocate new blocks?
  - how do we write new data blocks to disk?

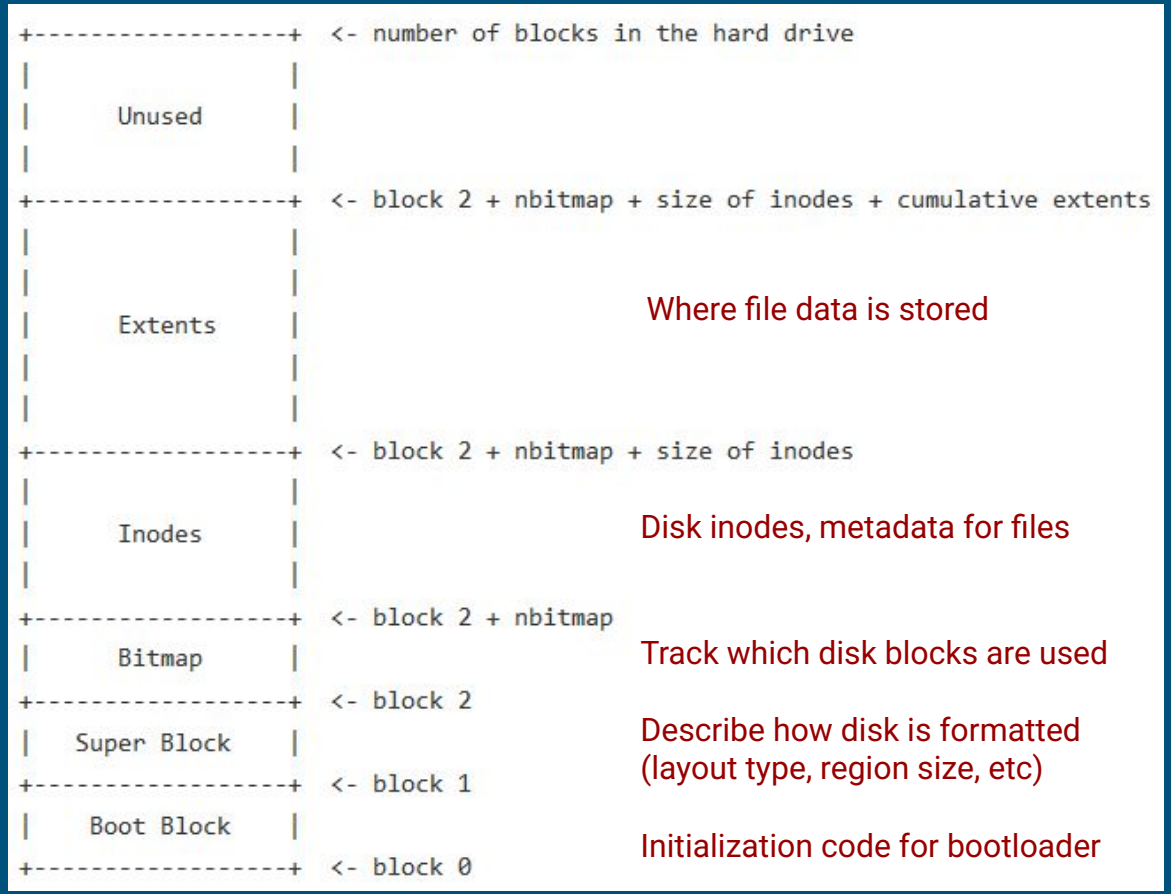
Change Dinode ➡ Change the FS Image

- why are there files in the file system if it's not writable in the first place?
  - because the initial file system image is written with a POSIX program
    - mkfs.c! runs during `make`
  - mkfs.c understands the xk file system format
    - sets up superblock, bitmap, inode table
    - writes user programs and files into the image so we can run programs in xk!
- when dinode is updated, file system format changes, so mkfs needs to be updated as well!

Initial Disk Layout

How things are currently stored on disk!

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



mkfs.c: Setting up Dinodes with Multiple Extents

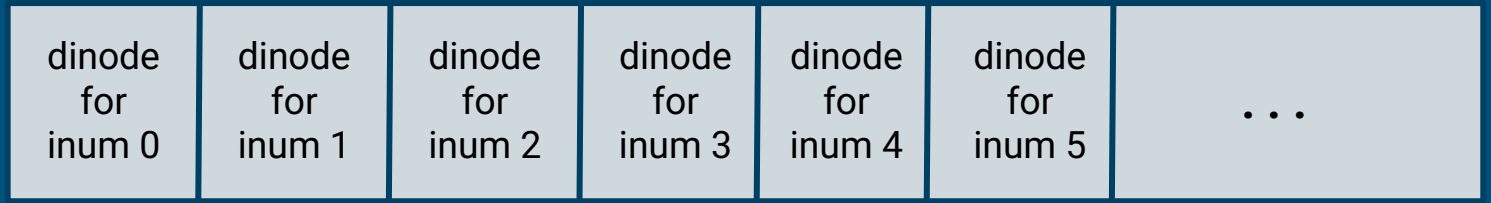
```
struct dinode din;
```

```
// Properly setup the inodetable's dinode (primarily t
rinode(inodetableino, &din);
din.data.startblkno = sb.inodestart;
din.data.nblocks = xint(inodetableblkn);
din.size = xint(inum_count * sizeof(struct dinode));
winode(inodetableino, &din);
```

- writes inodetable's dinode by populating it and invoking `winode`
- For an array of extents, you should update the first extent (data[0])
- make sure to update all dinodes written by mkfs.c!

FS: inodetable

- Special file for storing on metadata for file/directory
  - data is an array of on disk inodes (dinode)
  - data block starts at sb.inodestart (a block number)
- Where is the metadata for inodetable?
  - it's the first dinode in inodetable data, inum = 0
  - inodetable is special in that its metadata is stored within its data
  - how do we find metadata (first data block)? superblock tells us where the data starts!



Inode Cache

- Recall that for performance reasons, filesys always keeps a cache of inodes
  - check out the inode cache section of the lab 4 spec!
  - not all disk inodes are cached! only those that are open

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

- The inodes you've been working with are from the inode cache!
  - `iopen` allocates an entry from the inode cache and populates it with dinode content
  - `irelease` decrements an inode's ref count and evicts it from the cache when ref == 0

Change Dinode ➡ Change Cached Inodes

```
// 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;

  // copy of disk inode (see fs.h for details)
  short type;
  short devid;
  uint size;
  struct extent data;
};
```

cached from dinode

a cached (in-memory) copy of the disk inode

also stores in-memory only information (lock, ref count, valid)

should update "data" to be consistent with dinode!

Synchronize Cached Inodes With Disk Inodes

fs.c functions operate on cached inodes, but changes must be written to disk to persist!

- update inode with on disk inode content
  - `iget`: allocates an entry from icache, valid is 0
  - `locki`: if valid == 0, read disk inode read_dinode, and populate the inode fields

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

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

  if (!holding_inodetable_lock)
    unlocki(&icache.inodetable);
```

Synchronize Cached Inodes With Disk Inodes

fs.c functions operate on cached inodes, but changes must be written to disk to persist!

- inode changes are not automatically updated to dinode
  - consider writing your own *`write_dinode`* function to write out a disk inode

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

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

  if (!holding_inodetable_lock)
    unlocki(&icache.inodetable);
```

dinodes live in the inode table on disk

update a dinode = writing to the inode table at an offset

Change Dinode ➡ Change Reads

- existing read code (`readi`) assumes there's a single extent

```
for (tot = 0; tot < n; tot += m, off += m, dst += m) {
  bp = bread(ip->dev, ip->data.startblkno + off / BSIZE);
  m = min(n - tot, BSIZE - off % BSIZE);
  memmove(dst, bp->data + off % BSIZE, m);
  brelse(bp);
}
```

- make sure to update this to work with new dinode & inode format

Block Allocation

- done through bitmap! (kernel/fs.c)
  - bitmap sectors live on disk and track the usage information of all disk sectors
  - xk provides functions to manage the bitmap for you!
- `balloc()`
  - allocates consecutive blocks for a given device
  - panics when not enough consecutive blocks available
  - no guarantees on the content of returned blocks
- `bfree()`
  - frees consecutive blocks for a device

Block Allocation

WARNING: `balloc` and `bfree` only update the cached bitmap sectors but do not change the bitmap on disk.

To persist your bitmap changes, you need to update them to write the changed bitmap sector to disk.

Writing Data Blocks to Disk

- Just as we have a cache for inodes, we also have a cache for disk blocks!
- FS interact with disk blocks via the block cache (kernel/bio.c)
  - brings block/sector into memory and manages them (evict, writeback)
  - struct buf:
    - metadata for managing buffer
    - buf->data = sector data
      - update buf->data to new data content!
  - block cache operations:
    - bread: reads the sector into memory , locks the cached block
    - bwrite: marks the block dirty and issues a write
    - brelse: releases the lock on the cached buffer

Phew, that was a lot of content!
Now you are ready to implement writei!

writei

Currently, writei returns an error when writing to non-device files… Let's fix that!

```
int writei(struct inode *ip, char *src, uint off, uint n) {
  if (!holdingsleep(&ip->lock))
    panic("not holding lock");

  if (ip->type == T_DEV) {
    if (ip->devid < 0 || ip->devid >= NDEV || !devsw[ip->devid].write)
      return -1;
    return devsw[ip->devid].write(ip, src, n);
  }
  // read-only fs, writing to inode is an error
  return -1;
}
```

writei

Overview of writei arguments:

- writei(inode, src, offset, n)
  - inode: file to write
  - src: content to write
  - offset: offset into the file to start writing
  - n: bytes to write

writei

You need to allow for overwrites and appends in Lab 4!

Writing to a file can extend file, two cases:

- Overwrite: no change to metadata, change data in existing data blocks
  - E.g.: file of length 100 bytes, write 20 bytes at offset 0 is just an overwrite
- append: metadata changes, add new data blocks!
  - file of length 100 bytes, write 20 bytes at offset 90 appends 10 bytes to the file
  - may cause additional blocks to be allocated => populate new extent

More on Append

- Append always changes the file size
  - But does it always require new blocks?
  - Not always! If new data can fit within the current block no need to allocate more blocks.
    - How many blocks are allocated for a file with length 100?
    - Do you need to allocate a new block for 10 more bytes?
- Must update the in memory inode and on disk inode upon an append!
  - How do you do this? (hint: review slide 20!)

Tips for `writei`

- Can simplify `writei` logic by separating block allocation from data write
  - First append file space if necessary:
    - compute total blocks needed to perform the write
    - new blocks needed = new total blocks - old total blocks
    - allocate new blocks, populate extent
  - *Then* write data assuming extents are already allocated!

- You'll frequently want to get block number containing a file offset ⇒ We recommend writing helper function for this:
  - can be used by both `readi` and `writei`
  - makes debugging easier
- But feel free to do whatever you want! It's your design!

Tips for `writei`

- mkfs.c computes blocks needed given a file size by

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

(feel free to do your own math, but just know that this math is correct)

- Make sure to update inode and dinode upon an append!
- Your `writei` should persist changes immediately (no fsync in xk)
- Hint: `readi` is a helpful example for how to interact with block cache

File Creation

Moving on to File Creation

You should be able to create a new file when O_CREATE is passed to file_open

- Only create if file does not already exist
- What to do upon a create case:
  - Allocate a new disk inode
  - Update parent directory (root) with a new directory entry

Allocate a New Dinode

How do I allocate and populate a dinode?

- Find a free dinode in the inodetable
  - How to tell if a dinode is free?
  - Reuse old fields, set type or size = -1, or add a new field
    - If you change the default value of dinode field that needs to be updated in mkfs.c as well
- If no free dinodes, create a new dinode by appending to the inodetable
  - What function should you use to append to the inodetable?
- Write to the inodetable
  - Implement write_dinode (see read_dinode)

Add a New Directory Entry

- Update parent directory (root) with new directory entry
  - New dirent: new file name, dinode number
  - Use `writei` to write the direntry
    - if root dir has any invalid dirent, can reuse that entry for your new dirent
    - what's an invalid dirent? take a look at dirlookup…
- All files will be created under root dir, no nested directories for this lab

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

Quick Note on O_CREATE

When file_open gets called by the tests with the O_CREATE flag, it gets called with O_CREATE | ANOTHER_FLAG.

How do we access both flags?

Quick Note on O_CREATE

When file_open gets called by the tests with the O_CREATE flag, it gets called with O_CREATE | ANOTHER_FLAG.

How do we access both flags?

Use the & operator to extract each flag!

And Now Time for Delete

This is pretty much the reversed logic of create!

- `unlink`(path)
  - if path exists and no open references to the file, delete from the file system*
    - how do you count open reference?
      - hint: all open inodes are in the inode cache!
    - effectively undoing steps from file creation
      - frees the dinode so future creates can reuse the dinode
      - update parent dir's data to invalidate the dir entry
  - otherwise, error

*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

That's it for Part A!

Part B: Concurrency

Notes

- concurrent_* should protect the corresponding inode functions
- what happens if multiple processes try to create the same file?
  - only 1 process should succeed in creating the file
  - all other processes should simply open the created file
  - how can you determine if a file exists?
    - be careful of time of check to time of use problem!

Part C: Crash Safety

Guaranteeing Atomic Operations

Now that our file system is writable, we need to ensure that it is crash consistent. But what do we mean by crash consistent? Let’s take a look at an example:

Say we have “file.txt” which is 512 bytes long. We try to append 50 bytes to this file.

Is this operation atomic?

Guaranteeing Atomic Operations

No! We need to multiple **block writes** to accomplish this simple append:

1. Invoke file_write
2. Compute new file size
3. **Update size on disk (dinode)**
4. **Update file contents in memory**
5. **Write the new file contents to disk**

That’s three block writes!

Guaranteeing Atomic Operations

What happens if we crash before we write the new file contents to disk?

1. Invoke file_write
2. Compute new file size
3. **Update size on disk (dinode)**
4. **Update file contents in memory**
5. ~~**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!

So how do we solve this problem?

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 (apply the log!)
4) Clear the commit flag

On file system initialization(iinit) check the log for recovery

If not committed, do nothing

If so, apply the log (this is idempotent!)

Designing Your Log

- Specify a 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

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`: apply the actual content of the log
    - use at commit time and during recovery time

More on `log_write`

- `log_write` is 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

More on `log_commit_tx`

- Should first write the log header to disk to indicate that txn is committed
- Then apply the log content (`log_apply`)
  - Copy blocks from previous log_writes to their actual location on disk
- Reset commit flag when done

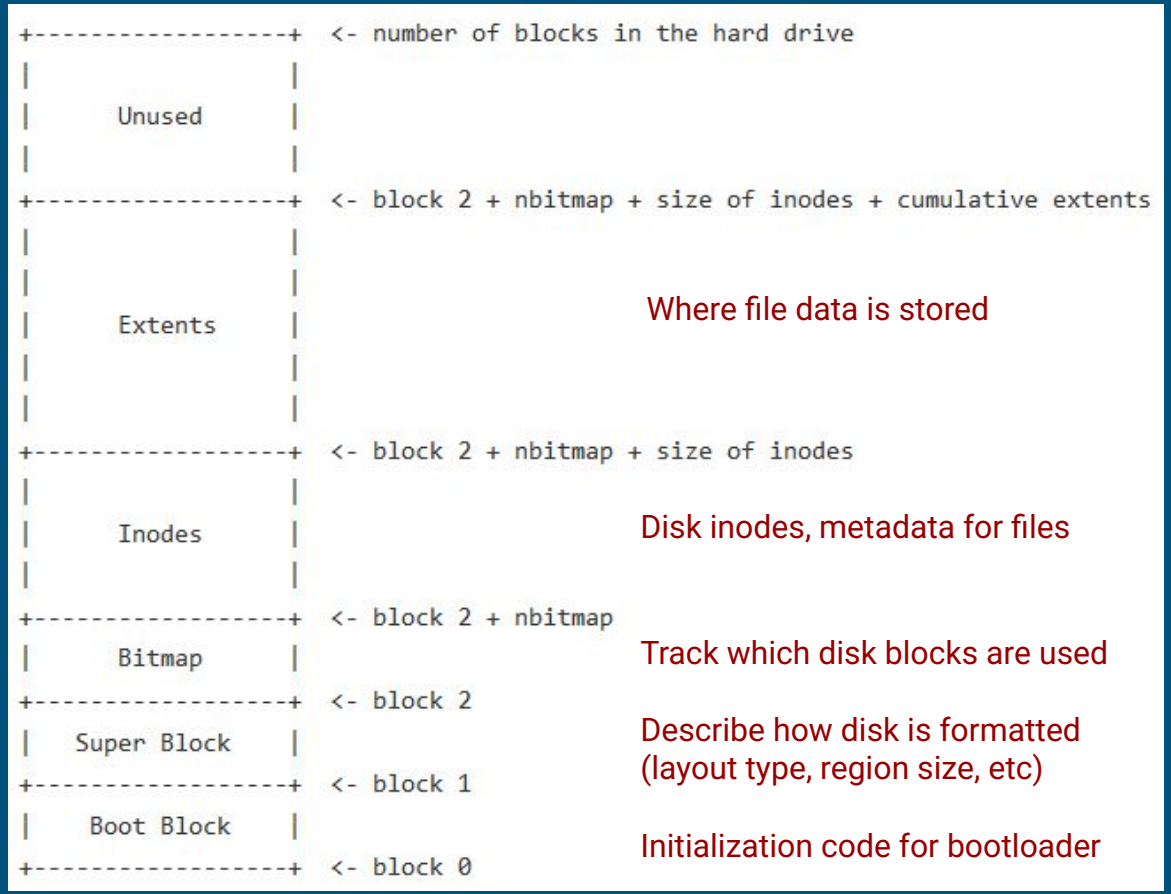
Supplemental Info

Prologue: Tour of the xk Storage Layer

Initial Disk Layout

How things are currently stored on disk!

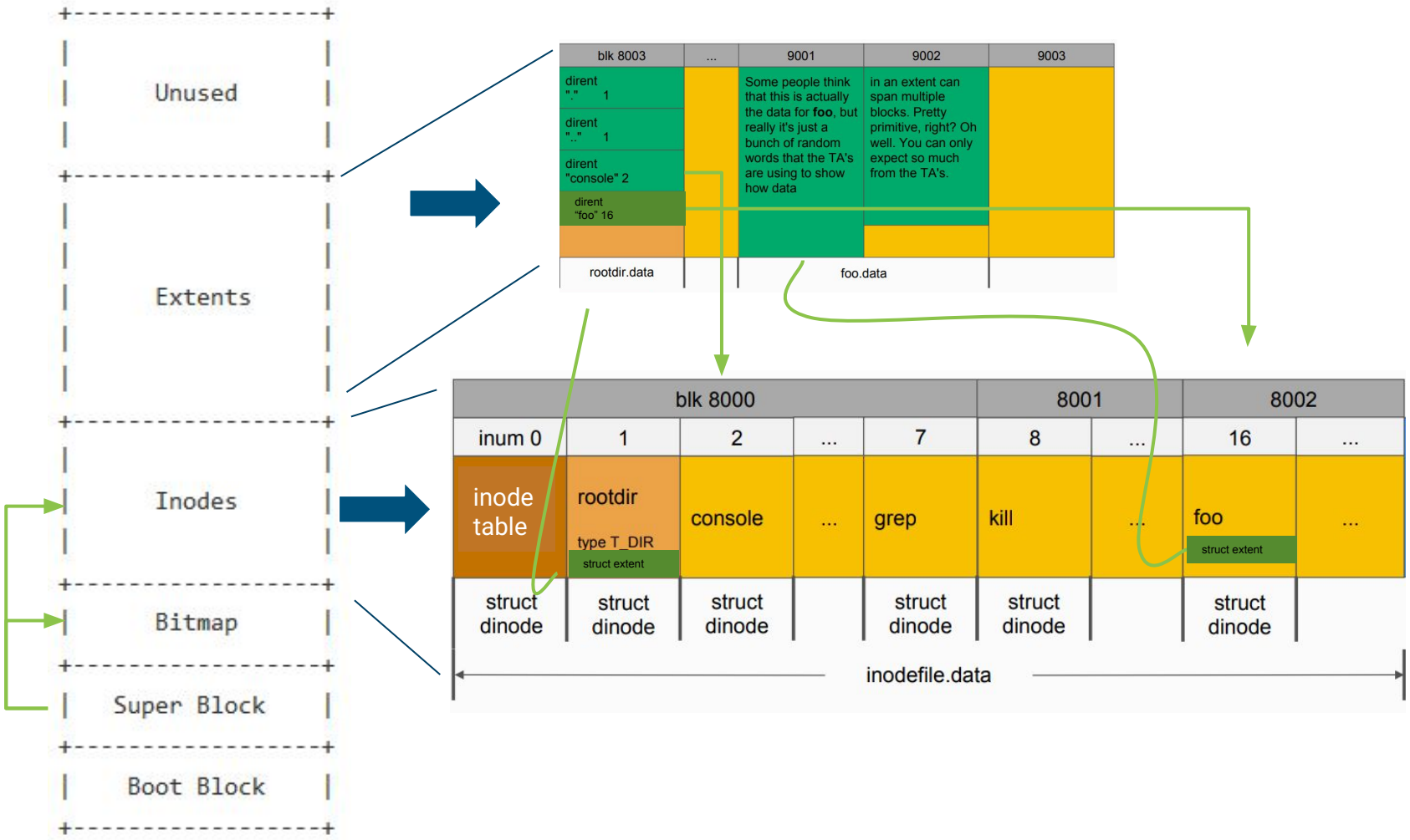
mkfs.c (a POSIX program, not an xk program!)
writes the initial disk image
following this layout



FS: Superblock

```
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  };
```

track metadata for the entire file system, persistent structure
- tracks location for bitmap
- tracks location for metadata table (inode array / inodetable)


Unused
Extents
Inodes
Bitmap
Super Block
Boot Block
blk 8003
...
9001
9002
9003
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 table
rootdir
type T_DIR
struct extent
console
...
grep
kill
...
foo
struct extent
...
struct dinode
struct dinode
struct dinode
struct dinode
struct dinode
struct dinode
inodefile.data

In Memory Data

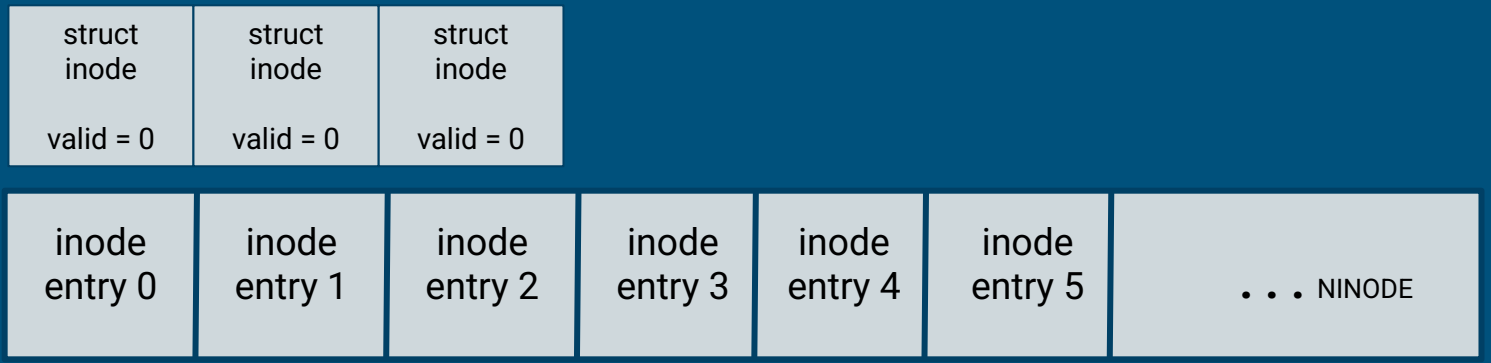
FS: icache

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

For ease & speed of access, we keep a cache of on disk structures in memory. This includes a lock protecting accesses to the cache, an inode cache for on disk inodes, and the cached inode for the inodetable itself.

icache.inodes array

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



initially, no dinodes are cached, all entries' ref and valid field == 0