Lab 4 Details

Administrivia

- Problem set 7 due this Friday (5/24)
- Memorial day next Monday (5/27), no class!

More mkfs Details

mkfs.c Explained

Just as a reminder, mkfs writes the initial file system image upon `make`

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

How do we set up the inodes?

- `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

```c
uint
ialloc(ushort type)
{
  uint inum = freeinode++;
  struct dinode din;

  bzero(&din, sizeof(din));
  din.type = xshort(type);
  din.size = xint(0);
  winode(inum, &din);
  return inum;
}
```

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

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 extents, you can update these to refer to the first entry of your extent array!

mkfs.c Explained

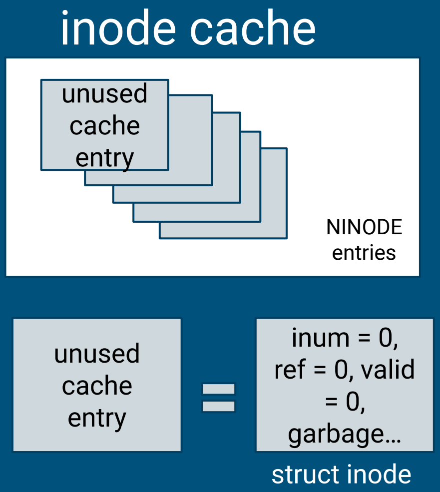
FAQ: Can you use ialloc to create a new on disk inode in xk?

Answer: No! ialloc is a mkfs function. Since mkfs is not build as a part of your kernel, you cannot call ialloc in your kernel code. The same applies to all mkfs functions!

Any Questions on mkfs?

Part A: Clarification & Tips

In-memory inode

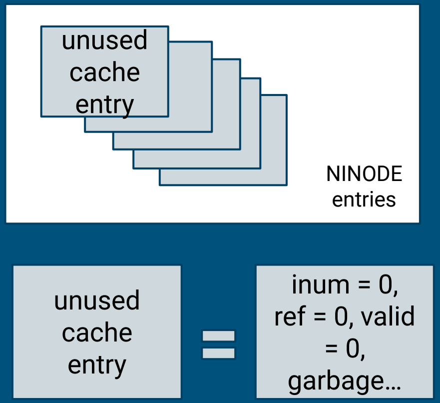


initially, all entries of icache.inodes are unused

*diagram skipped cached root dir inode for simplicity

In-memory inode

inode cache



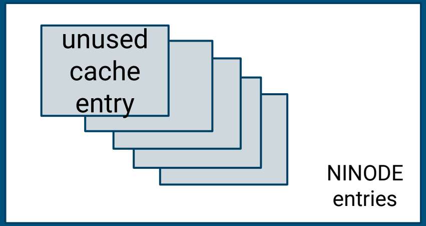
struct inode

user program:
open(file, O_RDWR)

*diagram skipped cached root dir inode for simplicity

In-memory inode

inode cache



unused cache entry = inum = 0, ref = 0, valid = 0, garbage…

struct inode

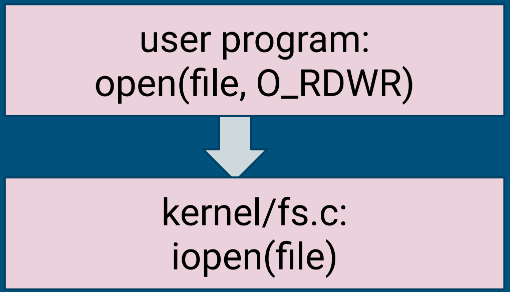


*diagram skipped cached root dir inode for simplicity

In-memory inode

inode cache

unused
cache
entry

NINODE
entries

unused
cache
entry

=

inum = 0,
ref = 0, valid
= 0,
garbage…

struct inode

user program:
open(file, O_RDWR)

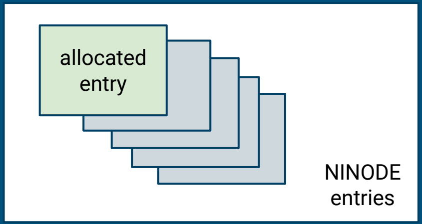
kernel/fs.c:
iopen(file)

kernel/fs.c:
namei(file)
*path traversal, finds inum 10 for file

*diagram skipped cached root dir inode for simplicity

In-memory inode

inode cache



allocated entry = inum = 10, ref = 1, valid = 0, garbage…

struct inode

*in this example, inode 10 is opened for the first time, if inode 10 is already cached, iget simply increments existing entry's ref count and returns a pointer to it

user program:
open(file, O_RDWR)

kernel/fs.c:
iopen(file)

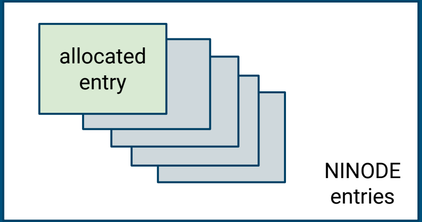
kernel/fs.c:
namei(file)
*path traversal, finds inum 10 for file

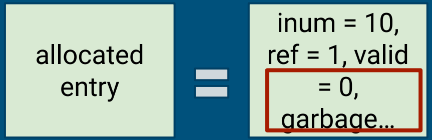
kernel/fs.c:
iget(10)
*returns a pointer to the cached inode

*diagram skipped cached root dir inode for simplicity

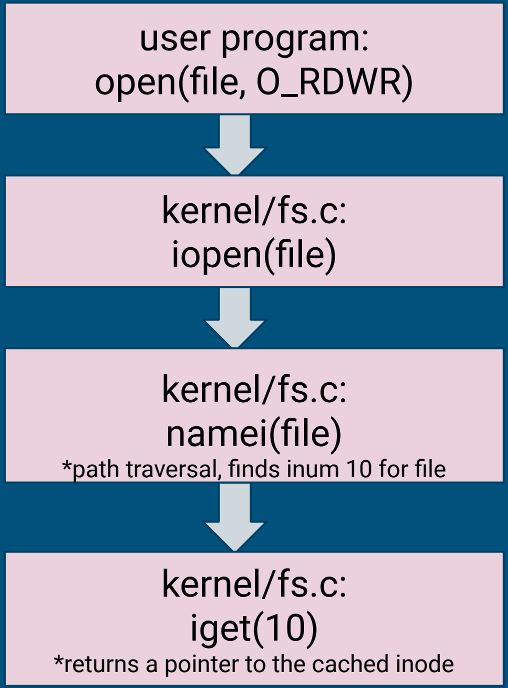
In-memory inode

inode cache





struct inode

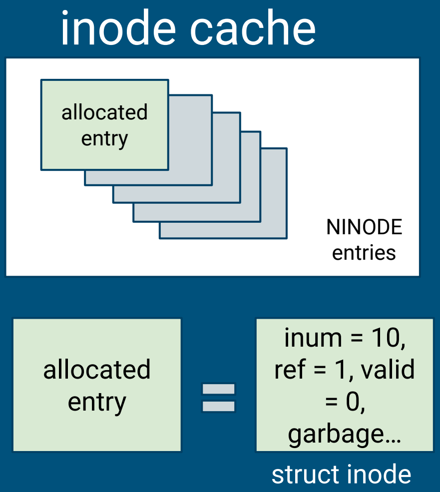


*in this example, inode 10 is opened for the first time, if inode 10 is already cached, iget simply increments existing entry's ref count and returns a pointer to it

freshly allocated (valid ==0) inode returned by `iget` does not contain accurate disk inode data yet (hence garbage)!

*diagram skipped cached root dir inode for simplicity

In-memory inode



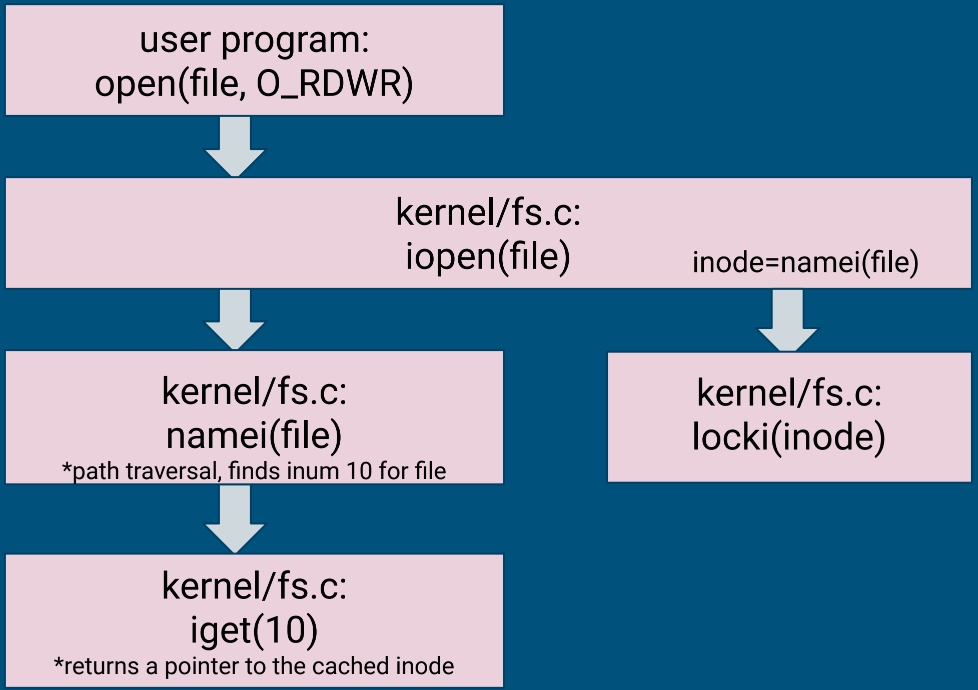
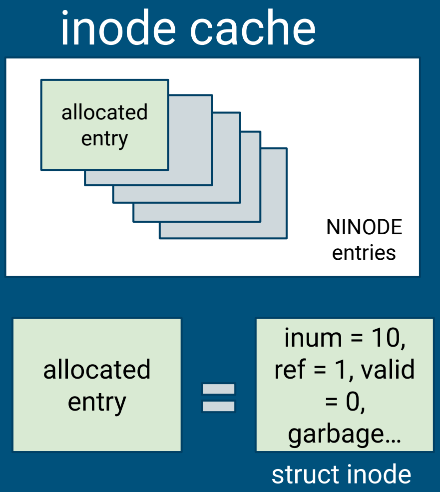


*diagram skipped cached root dir inode for simplicity

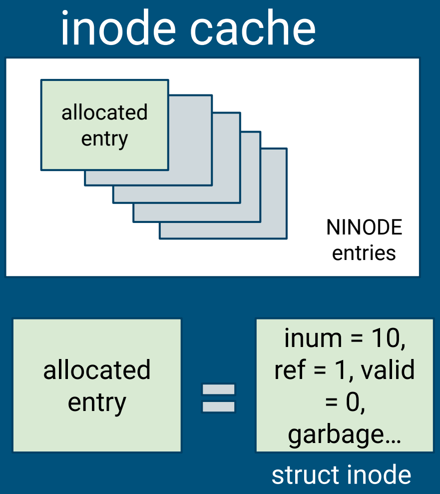
In-memory inode



kernel/fs.c:
locki(inode)

*diagram skipped cached root dir inode for simplicity

In-memory inode



kernel/fs.c:
locki(inode)

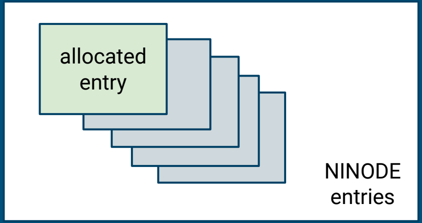
if inode is not valid
read in disk inode

kernel/fs.c:
read_dinode(10, …)

*diagram skipped cached root dir inode for simplicity

In-memory inode

inode cache



allocated entry = inum = 10, ref = 1, valid = 0, garbage…

struct inode

kernel/fs.c:
locki(inode)

if inode is not valid
read in disk inode

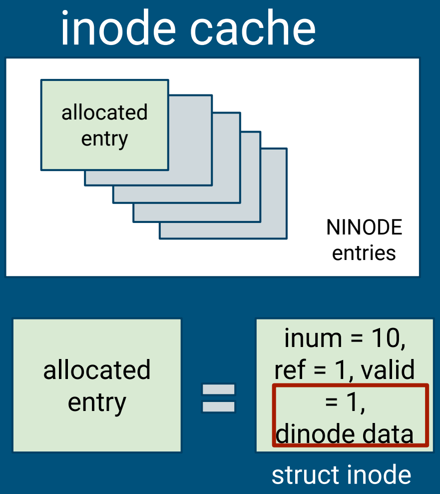
kernel/fs.c:
read_dinode(10, …)

kernel/fs.c:
readi(inodetable, …, INODEOFF(10))

*uses bread to request the disk block containing inode 10

*diagram skipped cached root dir inode for simplicity

In-memory inode



kernel/fs.c:
locki(inode)

↓ if inode is not valid read in disk inode

kernel/fs.c:
read_dinode(10, …)

↓

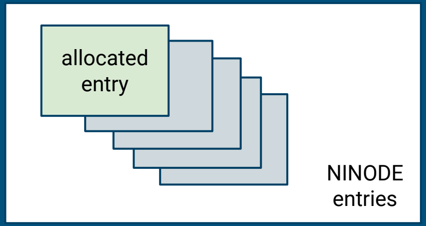
kernel/fs.c:
readi(inodetable, …, INODEOFF(10))
*uses bread to request the disk block containing inode 10

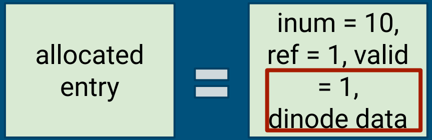
use the result of read_dinode to populate the in memory inode!

*diagram skipped cached root dir inode for simplicity

In-memory inode

inode cache





struct inode

And now this inode is ready to be used for fs operations!

kernel/fs.c:
locki(inode)

if inode is not valid
read in disk inode

kernel/fs.c:
read_dinode(10, …)

kernel/fs.c:
readi(inodetable, …,
INODEOFF(10))
*uses bread to request the disk block containing inode 10

use the result of read_dinode to populate the in memory inode!

*diagram skipped cached root dir inode for simplicity

More on read_dinode

```c
// Reads the dinode with the passed inum from the inode table.
// 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);
```

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

- Do you need to call read_dinode anywhere yourself?
  - No, read_dinode is called in locki to cache the on disk inode into in memory inode
  - you only need to modify the in memory inode and write back changes via write_dinode

More on write_dinode

- Should look just like read_dinode but calls writei instead of readi
- When should write_dinode be called?
  - in writei, when the inode changes!
- But wait, write_dinode calls writei, wouldn't there be an infinite recursion?
  - writei(file inode) => write_dinode(file inum) => writei(inodetable, INODEOFF(file inum))
  - does the last writei make changes to the inodetable's inode?
    - no! it's an overwrite! the writei to inodetable will not trigger more write_dinode!

More on write_dinode

- write_dinode(inum, struct dinode *)
  - mhmm… what should I pass as arguments to write_dinode?
    - get inum from the cached inode
    - set up a temporary struct dinode and populate it with data from cached inode!

Bitmap API

You interact with the block bitmap via fs.c: `balloc` & `bfree`

- `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 the changed bitmap sector back to disk*, you must call bwrite yourself!
  - hint: you can update balloc and bfree

Block Cache API

You can read and write disk blocks via the Block Cache in bio.c !

- brings blocks/sectors into memory and manages them (evict, writeback)
- struct buf
  - metadata for managing buffer
  - buf->data = block data
  - buf->blockno = the cached block #, very helpful for debugging!
- APIs
  - bread: brings the block into memory, locks (exclusive access) the cached block
  - bwrite: marks the block dirty and issues a write
  - brelse: releases the lock on the cached block

Part B: Concurrent FS Ops

Concurrent Create

- When there are multiple create calls to a single file, only 1 process can actually create the file!
    - How can we achieve this?
        - only one process can look up whether the file exists and create it at a time!
        - time of check to time of use! the entire read modify needs to be atomic
            - hint: is there any lock on the path traversal that can be used to prevent concurrent lookups?

Concurrent Delete

- When a file being deleted has open references, it cannot be deleted!
  - how to check if a file has open references?
    - all opened files have their inodes in the inode cache!
    - that's what each file info struct track to request fs operations!
- When there are multiple delete calls to a single file, only 1 process can successfully delete the file!
  - should only allow one process to check whether the file has open reference and perform the delete atomically (similar to create)!

Part C: Crash Safety

Journaling: Quick Recap

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

This is designed by you!

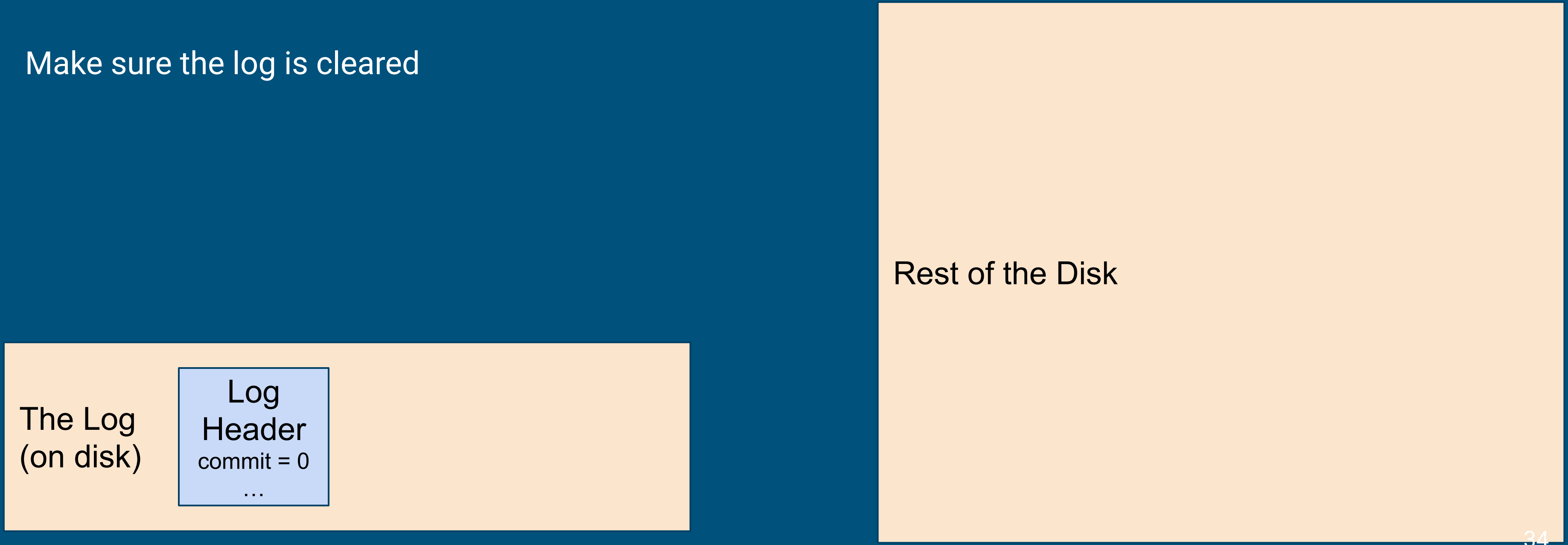
Should at least track:

- transaction status (committed or not)
- where to apply logged blocks

How journaling works without crashes

Step 1: “log_begin()”

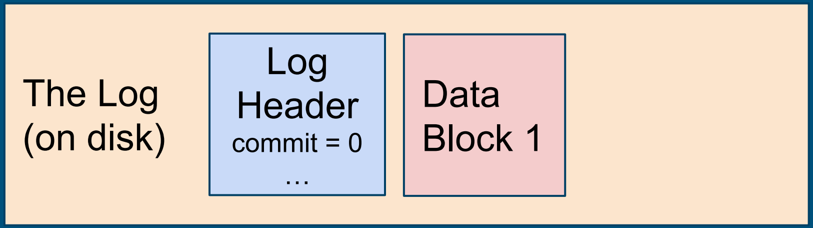
Make sure the log is cleared

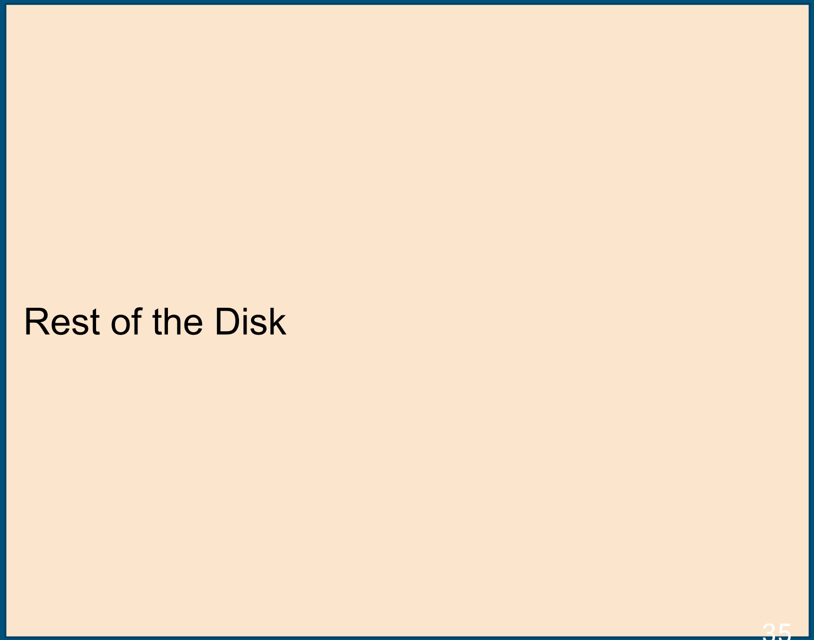


Step 2: “log_write(data block 1)”

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!





Step 3: “log_write(data block 2)”

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

The Log (on disk)

Log Header
commit = 0
..

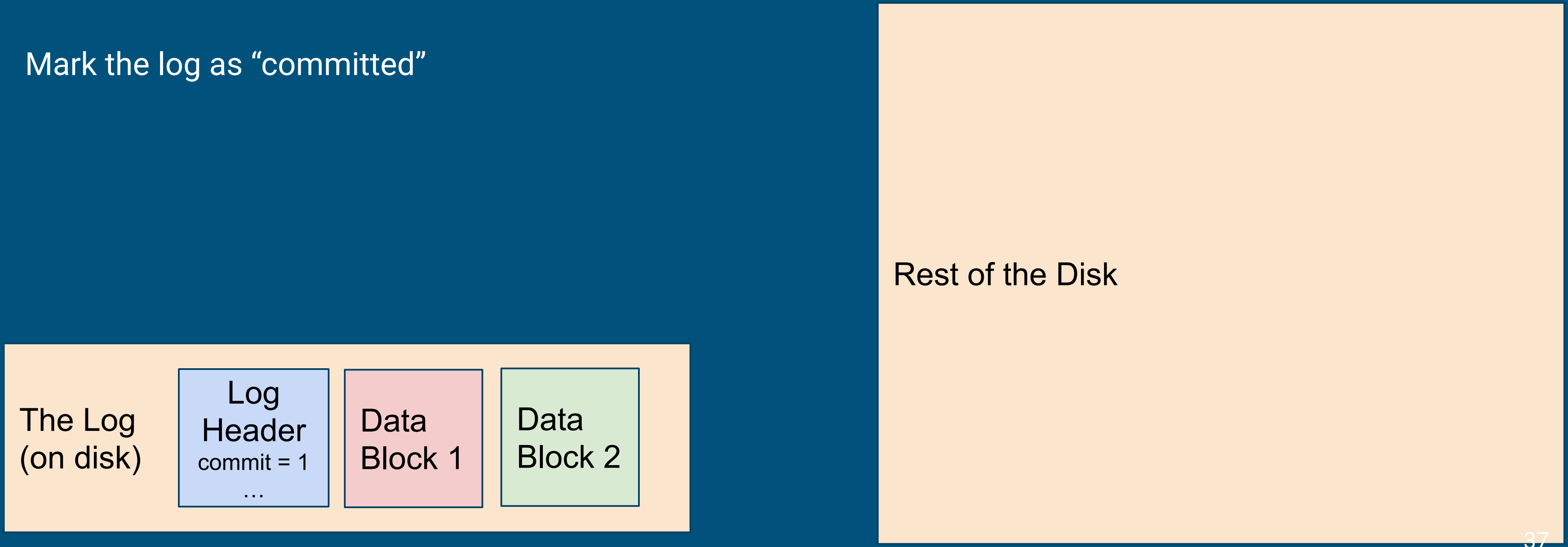
Data Block 1

Data Block 2

Rest of the Disk

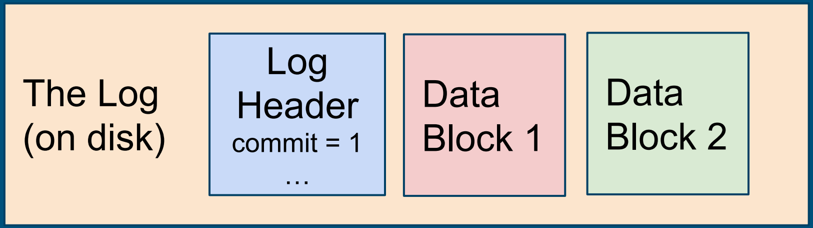
Step 4: “log_commit()” [1]

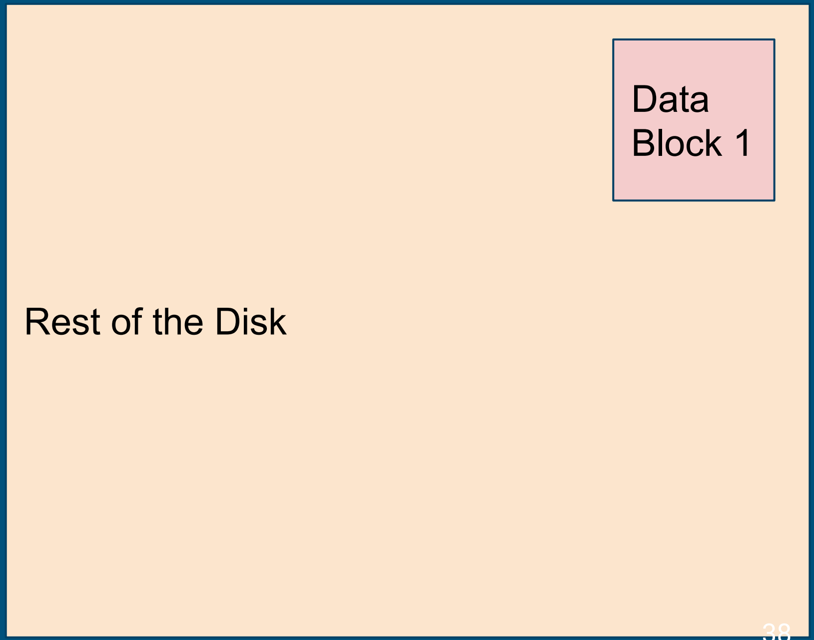
Mark the log as “committed”



Step 5: “log_commit()” [2]

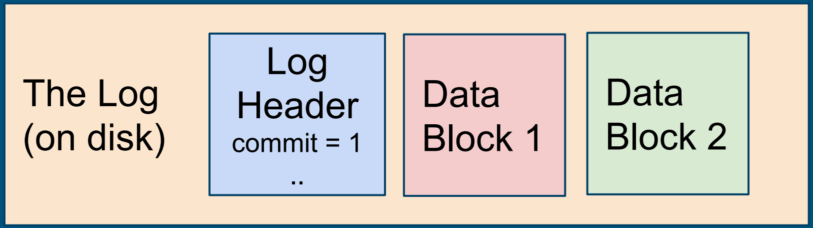
Copy the first block from log onto disk

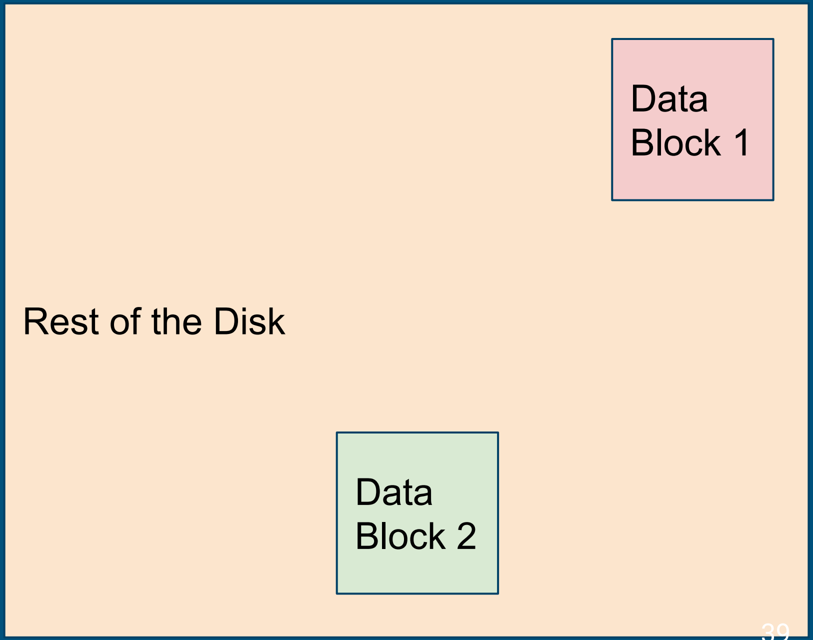




Step 6: “log_commit()” [3]

Copy the second block from log onto disk

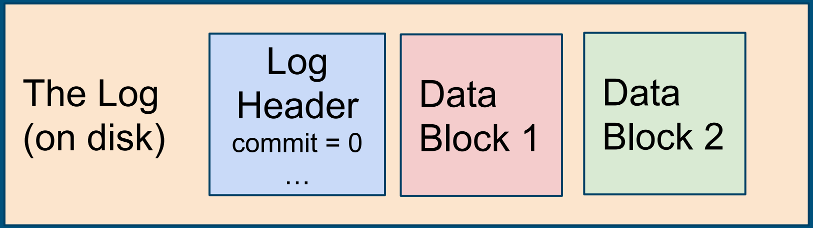


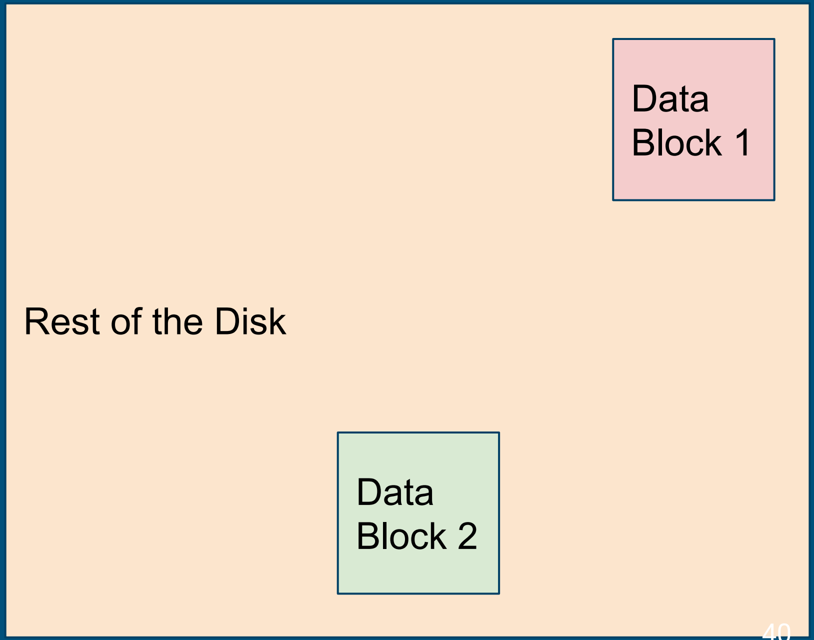


Done!

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





But what if we crash?

Example: **before** commit—CRASH

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

Rest of the Disk

The Log (on disk)

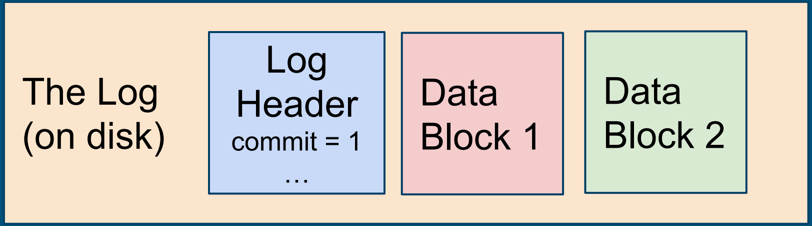
Log Header
commit = 0
…

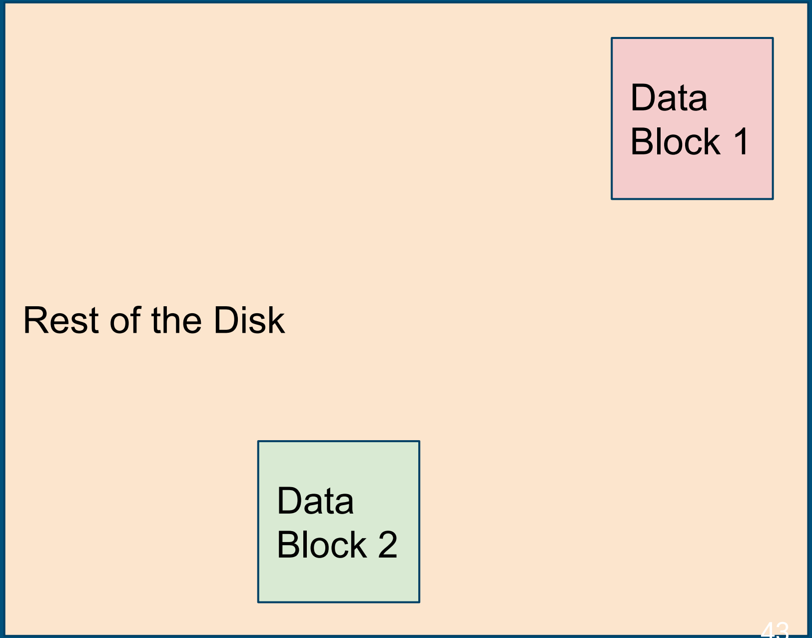
Data Block 1

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

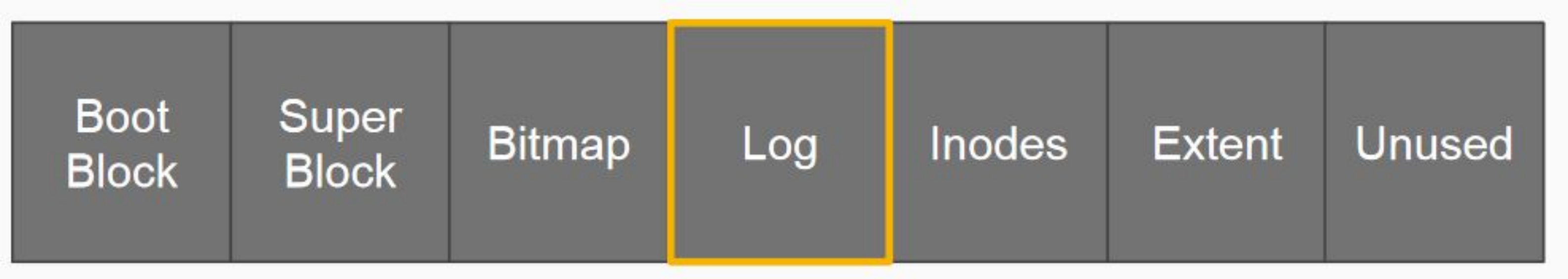




Where Do I put the Log?

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



Reflect the log on disk

In order to reflect log region in the initial disk image, what do you need to update?

- Mkfs.c
- Superblock struct
    - to track the location of the log region

What should log_write() do differently?

- log_write() instead of bwrite()
  - Just replace the bwrite calls with log_write!
- 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

How do we synchronize log access?

We recommend tracking a single transaction in the log

- How do we ensure that log access remains synchronized?

Questions?