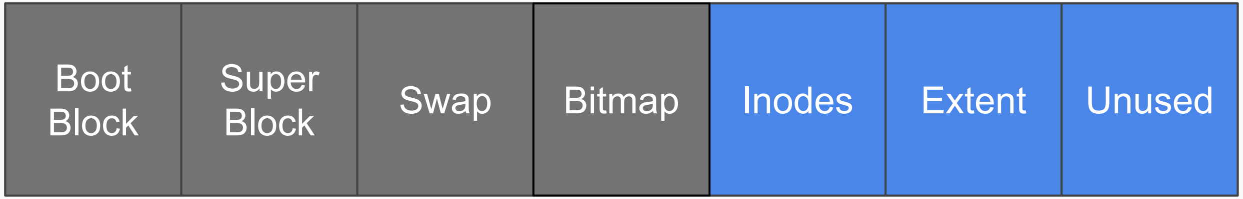
Section 8: Lab 4 Details

CSE 451 19AU

Part A: File Operations

Details of Inode/Extent Regions

Let’s take a quick look at the layout of how the inodes are laid out on disk.



Boot Block

Super Block

Swap

Bitmap

Inodes

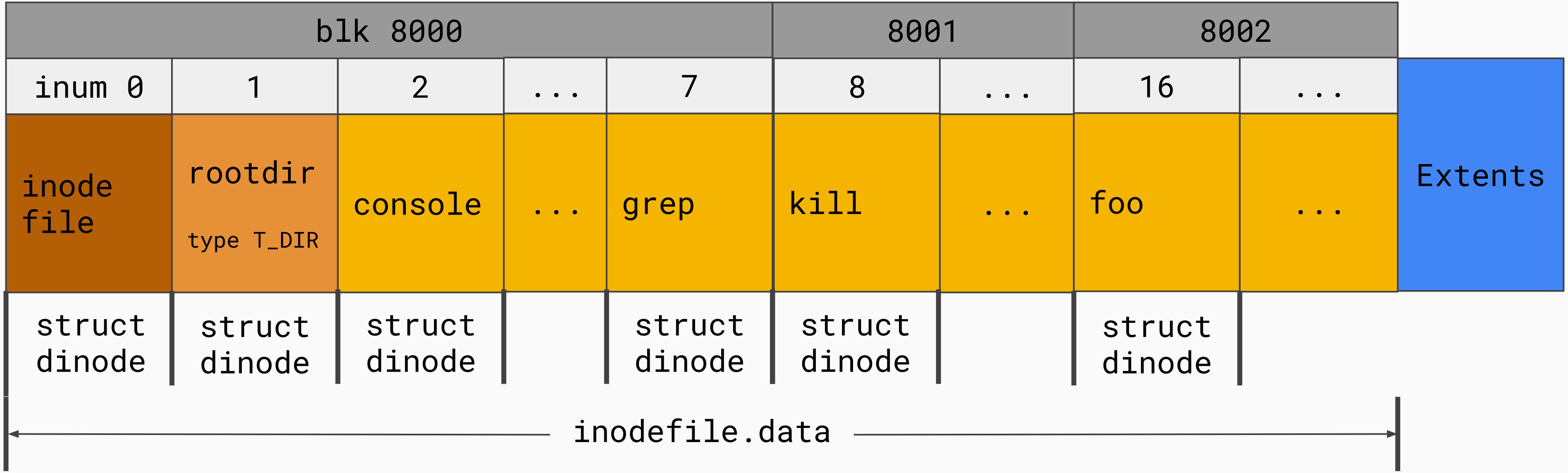
Extent

Unused

Inodefile

- The inodefile is the “inodes” section on disk, which stores the table of inodes (`struct dinode`)
  - Reading from and writing to file is just like reading/writing a normal file
- The 0th inode is the inodefile itself: the data field in the 0th inode corresponds to the inodes region
- The 1st inode is the root directory: the data block for this inode is an array of directory entries

Inodefile



blk 8000					8001		8002		
inum 0	1	2	...	7	8	...	16	...	
inode file	rootdir type T_DIR	console	...	grep	kill	...	foo	...	Extents
struct dinode	struct dinode	struct dinode		struct dinode	struct dinode		struct dinode		

inodefile.data

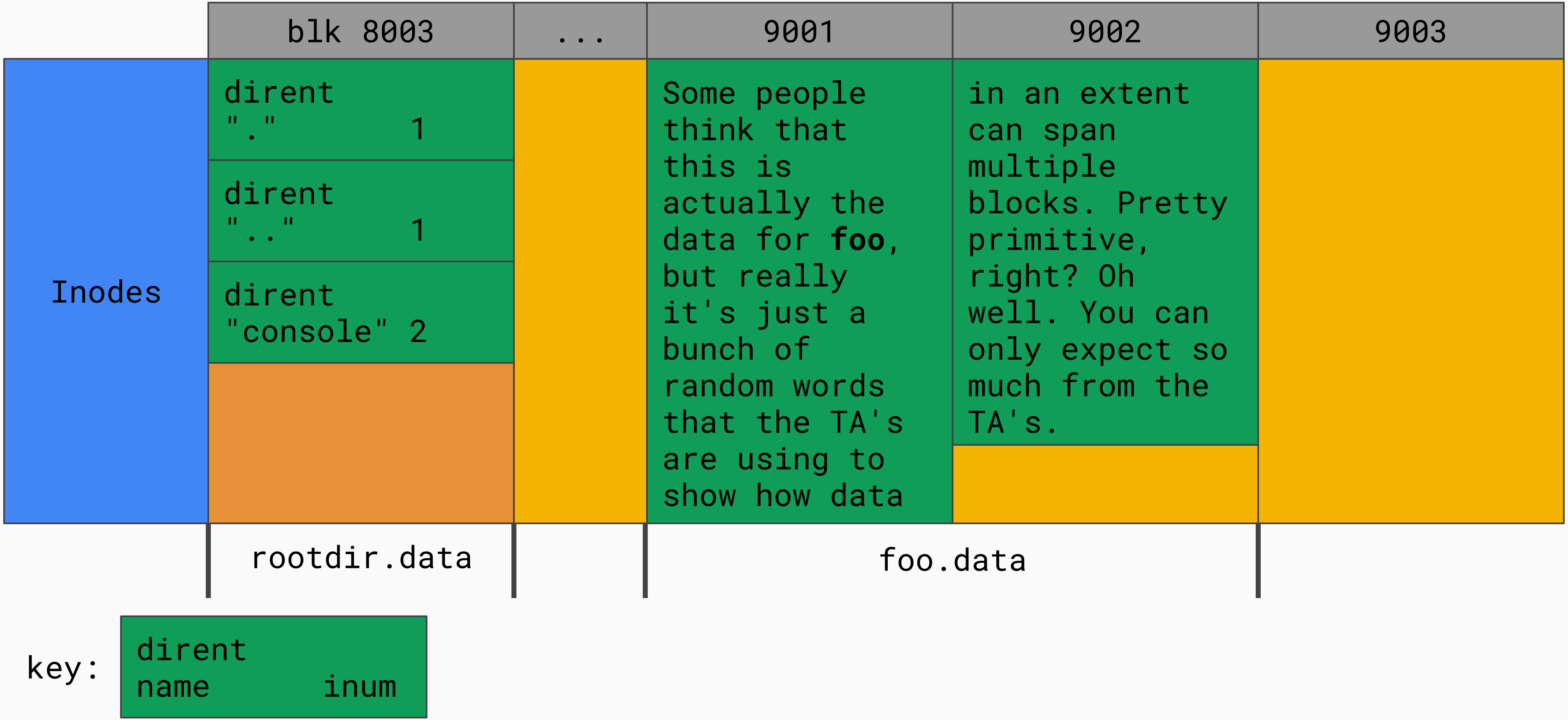
Extents

- The extents region is where the actual data (that is not the initial inodefile) for all the files in the filesystem lives.
- An “extent” is a sequence of contiguous blocks of memory, and a single inode can consist of multiple extents.
- Your job in part A is to work out a clean design of working with these extents.

Growable Extents

- There are multiple ways of modifying the dinode to allow for growable space. Here are just a few:
    - repurpose the padding to convert the single `extent data` to an `extent data[N]` array.
    - when allocating a new extent, allocate more blocks than strictly necessary by the write request
    - allocate K direct extent blocks and L indirect extent blocks, which in turn point to another array of direct extent blocks
        - While this is the system used by the Unix Filesystem, it's nastily complex to get right; so save this for a "fun" spring break project rather than lab 5 :)
- Important to keep in mind that sizeof(dinode) should divide into 512. Why?

Extents



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.

The 5-Layer Filesystem API

Userland	KERNEL LAND			
System Calls	File API	Inode API	Block API	IDE API

write()
open()
	filewrite()
	fileappend()
	filecreate()
		writei()
		readi()
			bread()
			bwrite()
			brelse()
				iderw()

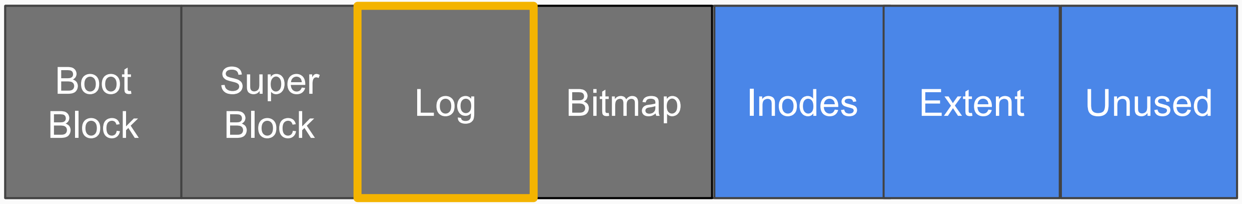
Part B: Crash Safety

Write-Ahead Log

- Our goal with the write-ahead log is to save a single disk **transaction/operation** (which can consist of a logical sequence of writes to the disk) so that we can recover from crashes.
- The “write-ahead” aspect entails writing the updated blocks to the log before writing to the actual blocks themselves.
- The design for the log should ensure that the disk is not corrupted by partial operations.

Disk Layout

- Add the log region after the superblock region
- The log will at most contain one transaction, since we clear the log out after every successful transaction.



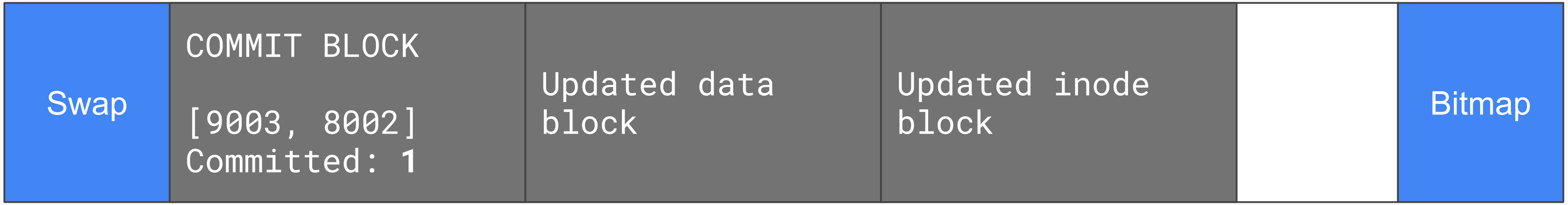
Boot Block | Super Block | Log | Bitmap | Inodes | Extent | Unused

Logging Example (from last week)

- Suppose the user is writing to an empty file `cat.txt`. This "file write" **operation** involves at least two **block writes**, which we tie together as a single transaction:
    - Update extent block with new data
    - Update inode block with new size
- Now we want to ensure crash safety by making sure that these writes happen **atomically**, i.e. either all writes occur or neither do.

Recommended Log Structure

- Commit block:
  - Array of block #'s to write to
  - Flag of whether the transaction is completed or not
- Sequence of data blocks to write



Swap

COMMIT BLOCK

[9003, 8002]
Committed: 1

Updated data block

Updated inode block

Bitmap

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`**)

Order of Writes

Since this is a **write-ahead log**, we update our log before actually writing to the disk.

- write updated data block to log block #1
- write updated inode block to log block #2
- write commit block to log block #0
- flush updated data block to extent block 9003
- flush updated inode block to inode block 8002
- *zero out log* → Do you really have to zero out the whole log?

Order of Writes

Since this is a **write-ahead log**, we update our log before actually writing to the disk.

- write updated data block to log block #1
- write updated inode block to log block #2
- write commit block to log block #0
- flush updated data block to extent block 9003
- flush updated inode block to inode block 8002
- *zero out log* →

Nope! Just zeroing out the commit block is all you need, and is faster than clearing the whole log!

What should `log_write()` do differently?

- The `log_write()` function is intended to be a wrapper function around all normal `bwrite` operations.
- Instead of directly writing the block to disk, we want to:
  - write the block information to our log data structures
  - keep the block in memory until the transaction has been successfully committed
- In order to write to a block but *keep the changes in memory*, look into setting the `B_DIRTY` bit for that block when calling `bwrite` -- this will ensure that the changes are not immediately flushed to disk

What should `log_write()` do differently?

- Now, once all the block writes for a single transaction have called `log_write()`, `log_commit_tx()` will be called
- Here, once the commit block is successfully flushed to disk, we want to flush all of our dirty blocks from the previous `log_write()`s to their actual location on disk
- So we can unset the `B_DIRTY` bit, which will ensure that the block cache flushes them to disk once the refcount on the block in memory goes to 0.

What happens if we crash?

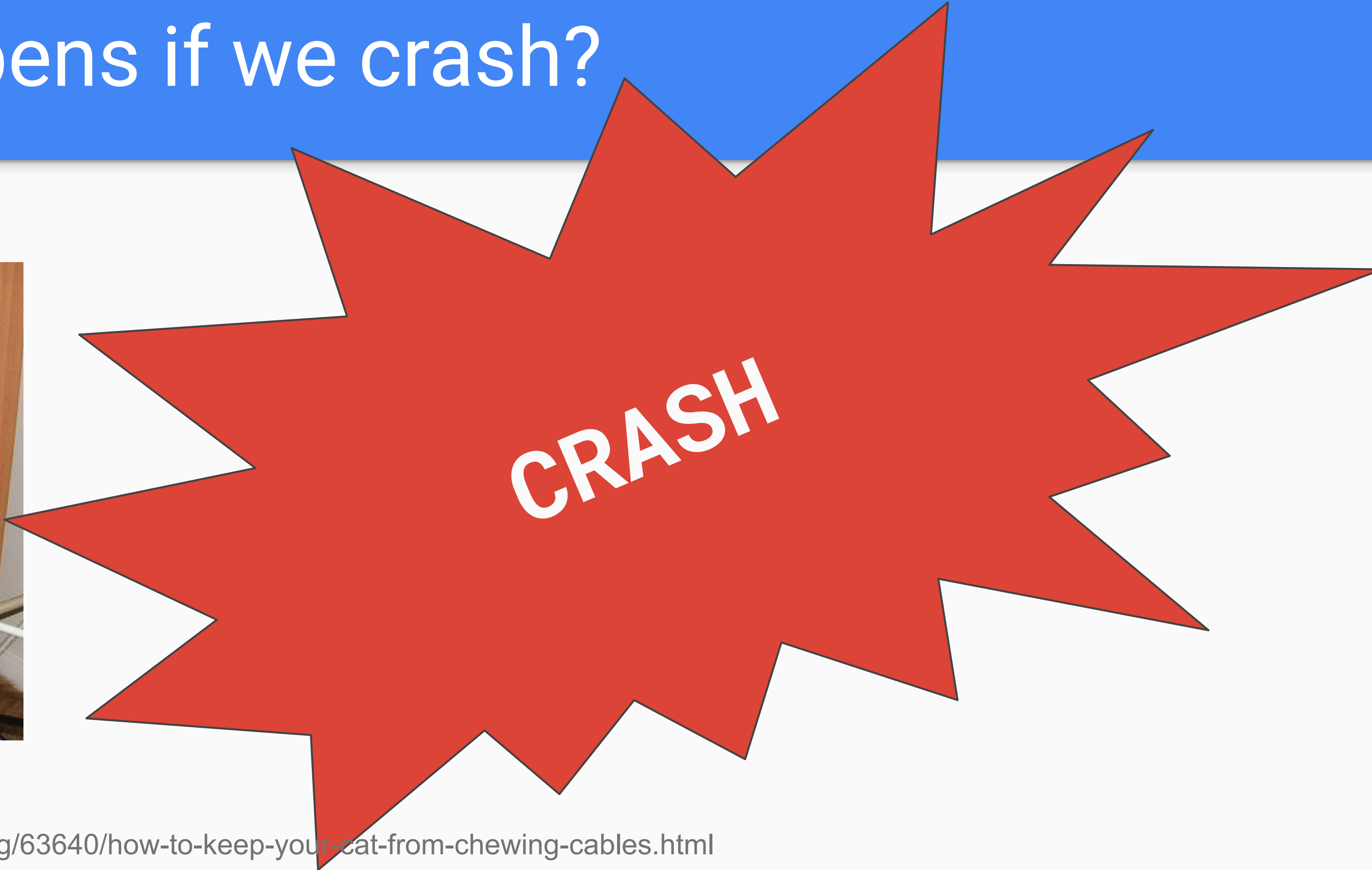
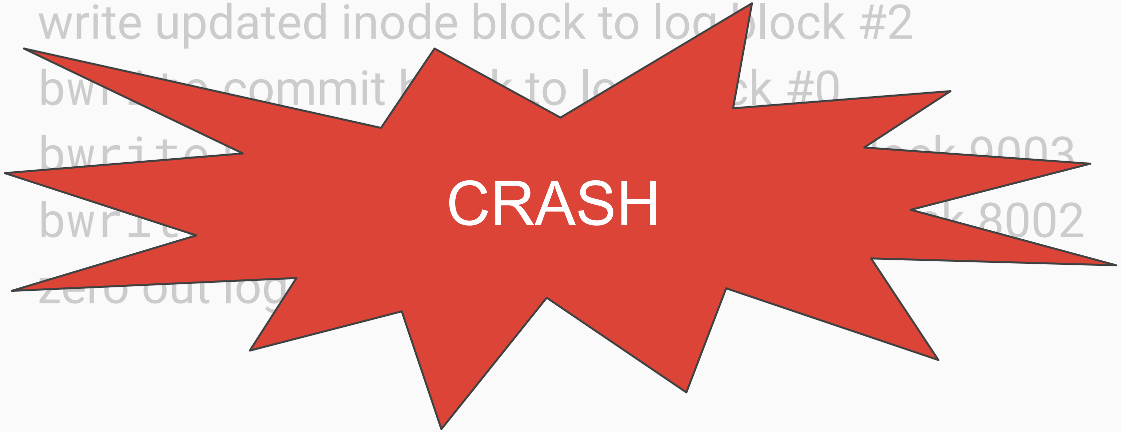


Image: https://www.petbucket.com/blog/63640/how-to-keep-your-cat-from-chewing-cables.html

Crash Scenario #1: Codename 0xDEADBEEF

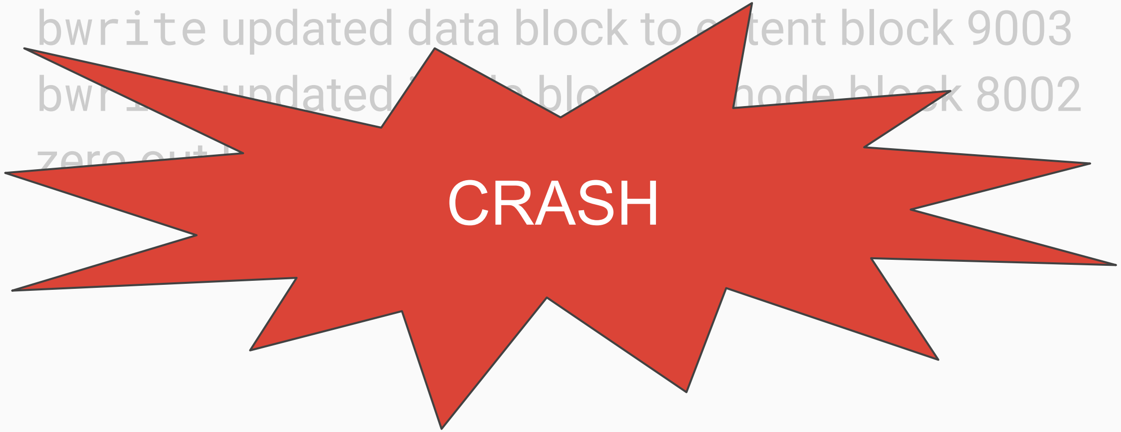
- write updated data block to log block #1
- write updated inode block to log block #2
- bw ite commit k to lo ck #0
- bwrite ock 9003
- bwri k 8002
- zero out log



On recovery, we inspect the log and find that the commit block is empty so the committed flag in log block 0 is set to 0. Thus we know that the operation did not complete and we discard the log.

Crash Scenario #2: Codename 0xCAFEBABE

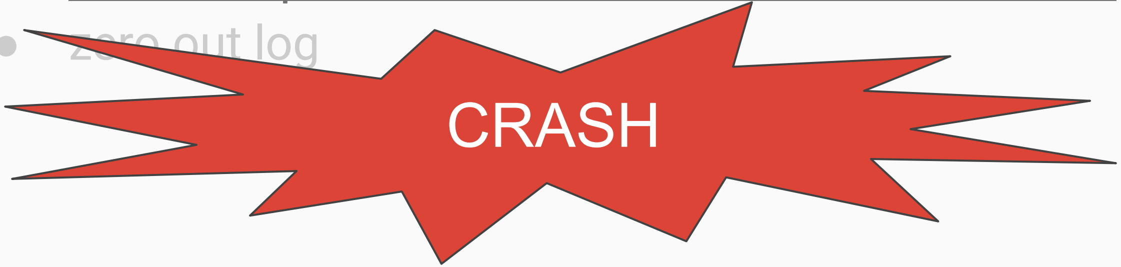
- bwrite updated data block to log block #1
- bwrite updated inode block to log block #2
- bwrite commit block to log block #0
- bwrite updated data block to c[illegible]tent block 9003
- bwr[illegible] updated [illegible] blo[illegible] node block 8002
- zero out [illegible]



On recovery, we inspect the log and find that the committed flag in log block 0 is set to 1. Thus we have the entire operation in the log, so we can replay the operations, updating the data and inode blocks, and then clear the log.

Crash Scenario #3: Codename `0xF007BA11`

- `bwrite` updated data block to log block #1
- `bwrite` updated inode block to log block #2
- `bwrite` commit block to log block #0
- `bwrite` updated data block to extent block 9003
- `bwrite` updated inode block to inode block 8002
- zero out log



On recovery, we inspect the log and find that the committed flag in log block 0 is set to 1. Thus we have the entire operation in the log, so we can replay the operations and clear it. Note that because our writes are **idempotent**, we can rewrite the blocks even if they were already written.

What’s wrong with attu? 😧

What to do if attu is being slow…

Might have noticed that attu, specifically the file system, has been slow at certain times during the past few days. Here are some alternatives to working on attu:

- A Linux VM
- Lab Machine (working with a cloned repo in /tmp directory)
- Working on a mac*

*Not recommended due to differences between MacOS and linux. We will be grading on attu, so beware!

Linux VM

You can find information about using a VM here:

https://www.cs.washington.edu/lab/software/linuxhomevm

You’ll want to download and use the 18WI VM.

Can be a bit of a pain to set-up, but once it’s up it’s very nice to have! 😍

Lab Machine /tmp directory

Accessing a lab machine:

- Walk into the lab and find a free linux machine
- SSH into a lab machine (<hostname>.cs.washington.edu)
  - Need to be on CSE wifi
  - Hostname is posted on the physical machine, go into the labs and write down some names
- SSH into attu and then SSH into a lab machine (when you're off campus)
  - (attu's main overhead lately has been the network file system so this should still be pretty fast)

Lab Machine /tmp directory

Lab machine /tmp directory is stored locally on the machine and is not part of the attu file system. This means it will be fast even when attu is slow.

Clone your repo in the /tmp directory and work on the lab.

Be sure to **commit and push often**, the /tmp directory is short for **temporary!**

MUST PUSH when finished working, or else changes could be lost.