File Systems

Part 2

Charts: Augmented from MIT's Adam Belay
https://pdos.csail.mit.edu/6.828/2023/lec/l-fs2.pdf

Crash Recovery

- **Issue**: Crashes leave disk in inconsistent state
- **Solution**: Logging

Crash Scenario

- Imagine you are using the filesystem
- Power is suddenly lost
- The system reboots
- Is the filesystem still usable?
- Is your data still there?

Crash Scenario is a Difficult Problem

- Filesystems perform multi-step operations
  - reserve an inode, then reserve a bit in the bitmap, then update a directory, then fill in an inode, then write data, etc.
  - A crash could leave invariants violated
- After rebooting:
  - Bad outcome: Crash again due to corrupt FS
  - Worse outcome: Silently read/write invalid data

create /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data
		read				
					read	
			read			
						read
	read write					
						write
				read write		
			write			

```
struct dinode { // disk inode
  short type;   // File type: dir, file
  short major;  //
  short minor;  //
  short nlink;  // # links to inode
  uint size;    // file sz (bytes)
  uint addrs[NDIRECT+1]; // Data blocks
}
```

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

open /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
		read					
					read		
			read				
						read	
				read			

```
struct dinode { // disk inode
  short type;   // File type: dir, file
  short major;  //
  short minor;  //
  short nlink;  // # links to inode
  uint size;    // file sz (bytes)
  uint addrs[NDIRECT+1]; // Data blocks
}
```

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

write to /foo/bar (assume file exists and has been opened)

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
read							
write							
							write
				write			

```
struct dinode { // disk inode
  short type;   // File type: dir, file
  short major;  //
  short minor;  //
  short nlink;  // # links to inode
  uint size;    // file sz (bytes)
  uint addrs[NDIRECT+1]; // Data blocks
}
```

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

read /foo/bar – assume opened

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
							read
				write			

```
struct dinode { // disk inode
  short type;   // File type: dir, file
  short major;  //
  short minor;  //
  short nlink;  // # links to inode
  uint size;    // file sz (bytes)
  uint addrs[NDIRECT+1]; // Data blocks
}
```

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

close /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

```
struct dinode { // disk inode
  short type;    // File type: dir, file
  short major;   //
  short minor;   //
  short nlink;   // # links to inode
  uint size;     // file sz (bytes)
  uint addrs[NDIRECT+1]; // Data blocks
}
```

nothing to do on disk!

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

Xv6 Various disk writes for file creation/population

- $ echo hi > x
- bwrite: block 33 by ialloc // allocate inode (block 33)
- bwrite: block 33 by iupdate // update inode (e.g., set nlink)
- bwrite: block 46 by writei // write directory entry
- bwrite: block 32 by iupdate // update directory inode with new len

```
struct dirent {
  ushort inum;
  char
name[14];
};
```

```
struct dinode {
  short type;
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[12+1];
}
```

create /x

data bitmap	inode alloc	root inode	x inode	root data
		read		
				read
	read			
			write	
			write	
				write
				write

46: Data Blocks
45: Free Blk BM
32: Inodes
3: Log Blocks
2: Log Head
1: Super Block

Various disk writes for file creation/population

- $ echo hi > x
- bwrite: block 33 by ialloc // allocate inode (block 33)
- bwrite: block 33 by iupdate // update inode (e.g., set nlink)

Crash →

- bwrite: block 46 by writei // write directory entry
- bwrite: block 32 by iupdate // update directory inode with new len

What Happens

- Not much bad happens
- Inode allocated and wasted, never usable in future

What about this order?

- $ echo hi > x
- bwrite: block 46 by writei // write directory entry
- bwrite: block 32 by iupdate // update directory inode with new len

Crash →

- bwrite: block 33 by ialloc // allocate inode (block 33)
- bwrite: block 33 by iupdate // update inode (e.g., set nlink)

What Happens

- Disaster!
- Inode could be reallocated again
- Directory points to uninitialized inode

What Order Could Really Happen

- Kernel (and maybe the disk too) reorders writes to minimize seeks
- In general, any order is possible

Writing Files Has Multiple Disk Writes

1. inode addrs[] and len
2. indirect block
3. block content
4. block free bitmap

Crash Scenario 1: inode refers to free block -- disaster!
Crash Scenario 2: block not free but not used -- not so bad

Unlink has Multiple Disk Writes

1. block free bitmaps
2. free inode
3. erase dirent

File System Recovery Goals

After reboot, run recovery code

1. Internal FS invariants must hold
   - e.g., no block is both free and used by an inode
2. All but the last few operations stored on disk
   - Data I wrote yesterday should be there!
   - But perhaps data at the time of crash will be lost
3. No reordering of data writes
   - echo 99 > result ; echo done > status

Correctness and performance

- Often at odds with one another!
- Disk writes are very slow
- Safety: Write data right away
- Speed: Wait and batch together writes

Crash recovery

- Arises in all storage systems, e.g., fs, databases
- Many clever solutions
- Performance/correctness tradeoffs

Logging (or Journaling)

- Goal: Atomic system calls w.r.t. Crashes
- Goal: Fast recovery (no hour-long fsck)
- xv6: minimal design for safety
- ext3: adds more speed

Logging basics

- Atomicity: All of system call's writes applied or none are
- Each atomic op is called a transaction
- Three phase operation:
- 1. Log phase: Record all the writes the system call will perform on disk
- 2. Commit phase: Record done on disk
- 3. Install phase: Do the actual disk writes

Crash recovery w/ logging

- Crash recovery of complex mutable data structures is hard
- But logging makes it easy, can retrofit on top of existing FS designs
- If ”done” found in log, replay all writes
- If “done” not found, ignore entries in log
- Called write-ahead logging

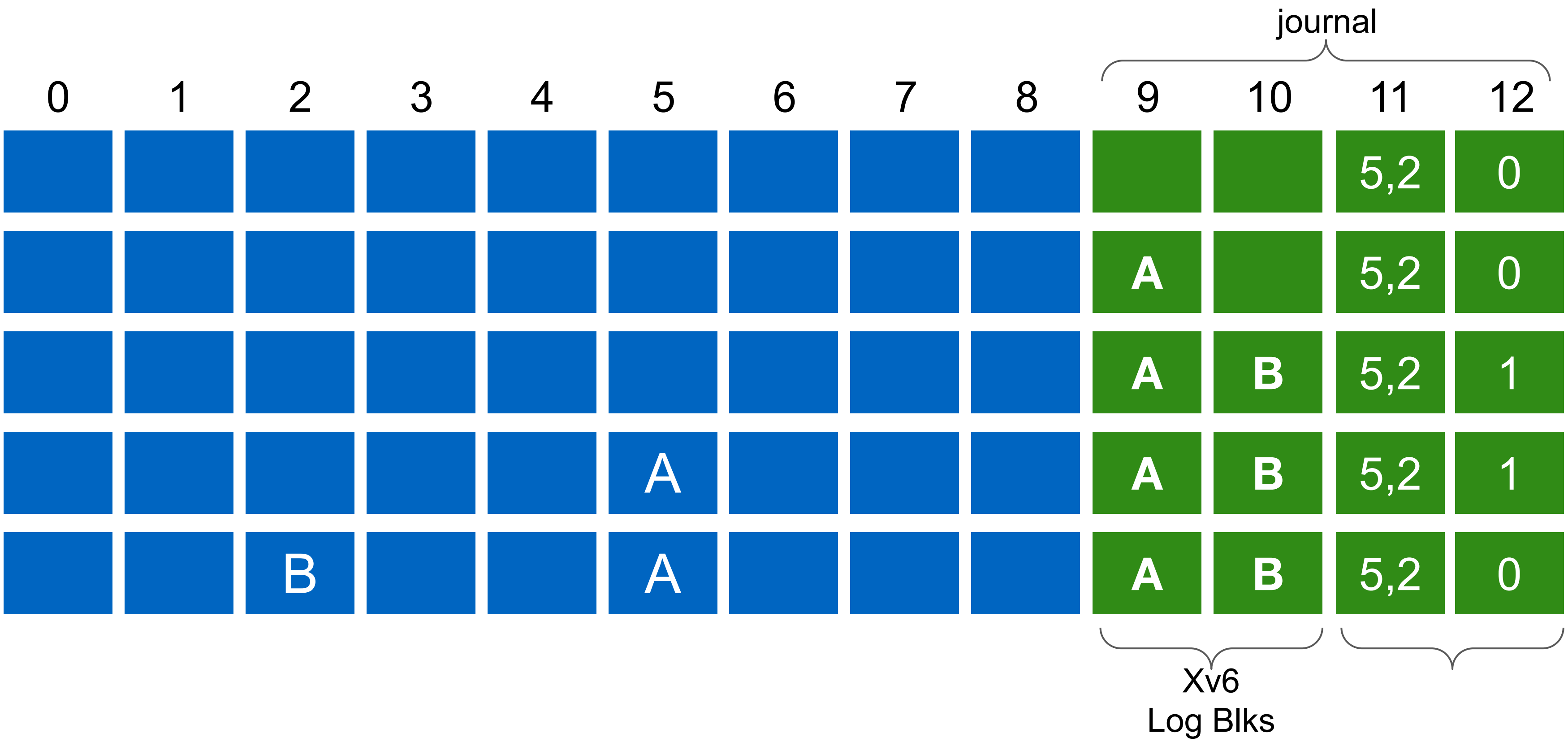
Rules

- install *none* of a transaction's writes to disk
- until *all* writes are in the log on disk, and the logged writes are marked committed

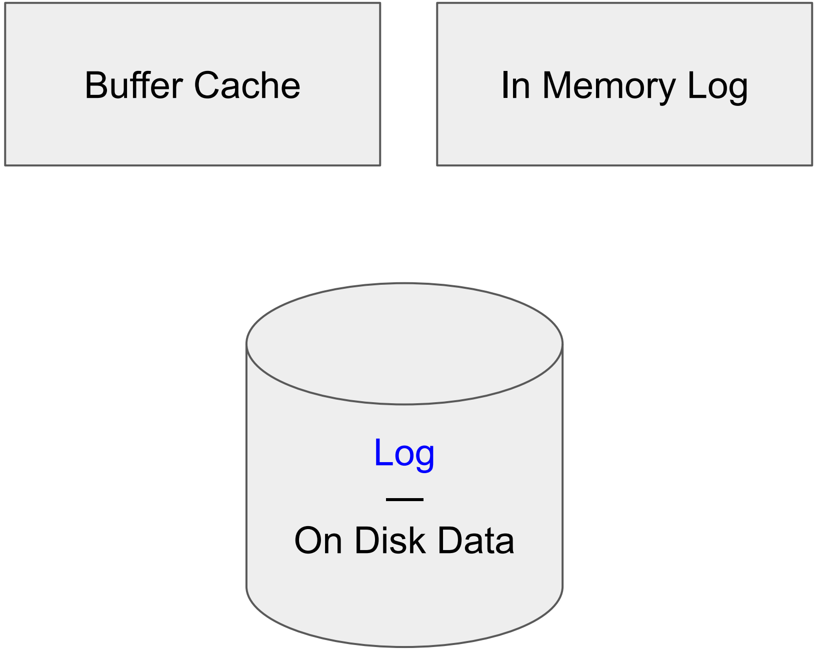
Once we’ve installed a transaction on disk...

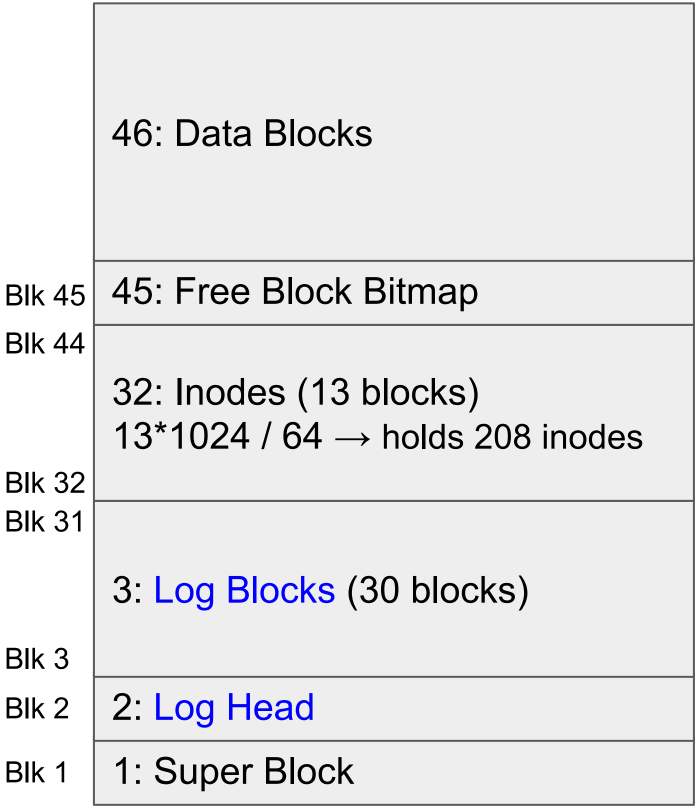
- We have to do all its writes
- This ensures the transaction is atomic
- Log allows us to detect if all steps in the transaction are there
- If not, we can safely abort the transaction

transaction: write A to block 5; write B to block 2



Xv6 Logging


Buffer Cache
In Memory Log
Log
—
On Disk Data


46: Data Blocks
Blk 45
45: Free Block Bitmap
Blk 44
32: Inodes (13 blocks)
13*1024 / 64 → holds 208 inodes
Blk 32
Blk 31
3: Log Blocks (30 blocks)
Blk 3
Blk 2
2: Log Head
Blk 1
1: Super Block

Xv6 Logging Steps

- On write:
  - Add blockno to in-memory array
  - Keep the data itself in the buffer cache (pinned)
- On commit:
  - Write buffered log to disk
  - Wait for disk to complete writing (synchronous)
  - Write the log header sector to disk
- After commit:
  - Install (write) the blocks in the log to their location in FS
  - Unpin the blocks in the buffer cache
  - Write zero to the log header sector on disk
- NOTE - X6 assumes
  - Either an entire sector is written or it is not (no partial writes)
  - No decay of sectors (no read errors)
  - No read of the wrong sector (seek errors)

Log Header

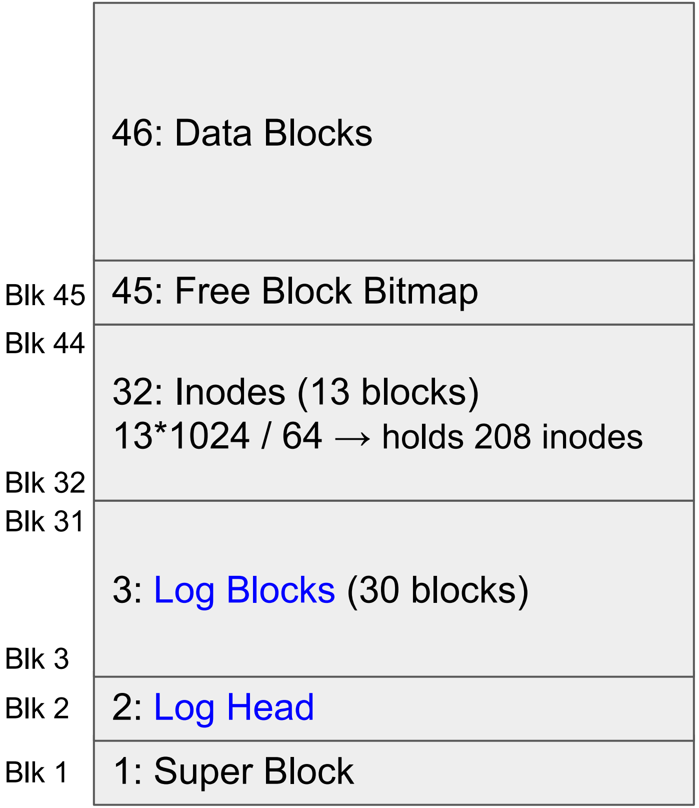
- An “n” value on disk indicates the commit point
- Nonzero: Indicates a valid transaction is committed on disk
- Zero: Not committed, may not be complete
- Records block #s that were updated
- And the number of blocks in log

$ echo “hi” > x

- Create inode
- Write ‘hi’ to file x
- Write '\n’ to file x

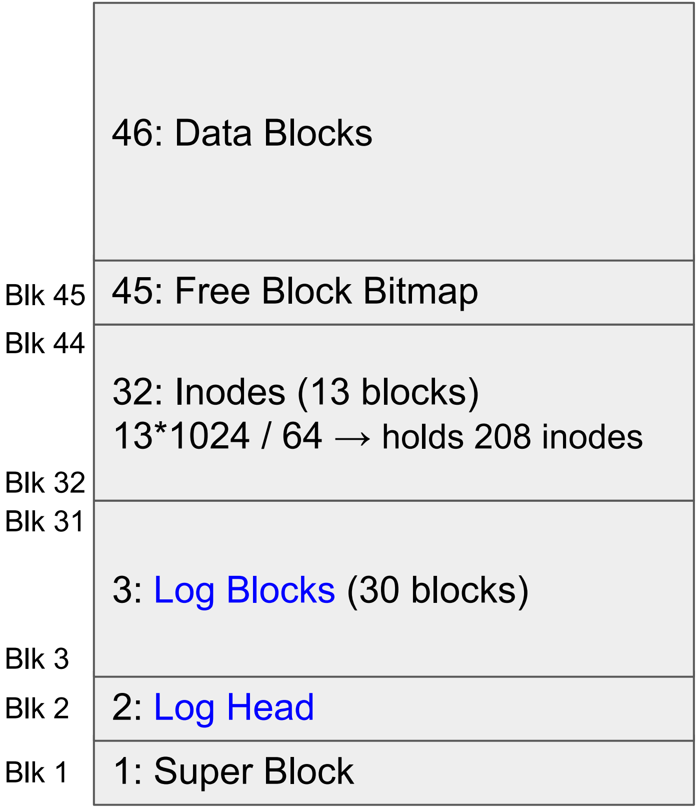
Create File X

- bwrite 3 // inode, 33
- bwrite 4 // directory content, 46
- bwrite 5 // directory inode, 32
- bwrite 2 // commit (block #s and n)
- bwrite 33 // install inode for x
- bwrite 46 // install directory content
- bwrite 32 // install dir inode
- bwrite 2 // mark log "empty”



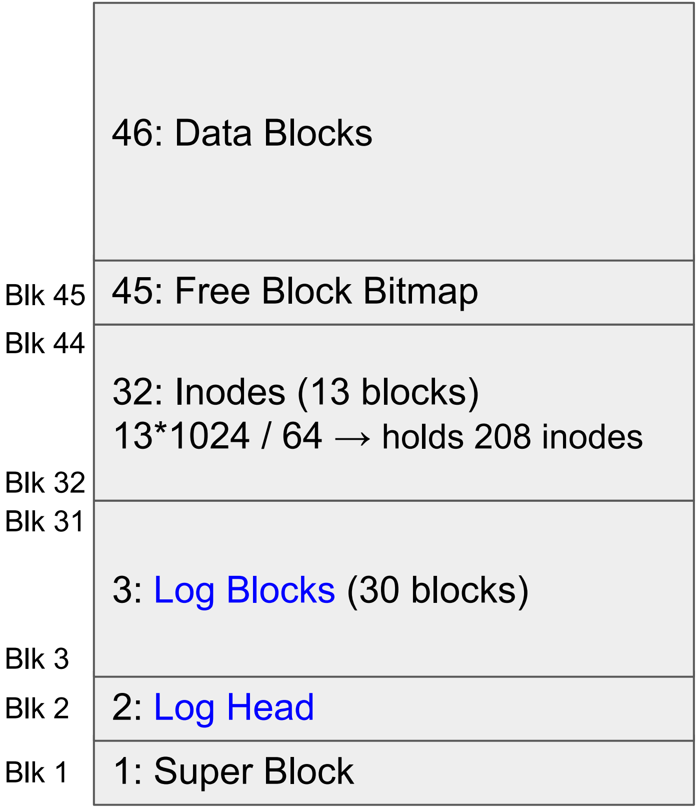
Write “hi” to x

- bwrite 3 // bitmap update (45)
- bwrite 4 // actual data (746)
- bwrite 5 // inode update (33)
- bwrite 2 // commit (block #s and n)
- bwrite 45 // bitmap
- bwrite 746 // a (note: bzero was absorbed)
- bwrite 33 // inode (file size)
- bwrite 2 // mark log "empty”



Write “\n” to x

- bwrite 3 // actual data (746)
- bwrite 4 // inode update (33)
- bwrite 2 // commit (block #s and n)
- bwrite 746 // \n
- bwrite 33 // inode (file size)
- bwrite 2 // mark log "empty”



Logging Challenges

- Challenge: Prevent write-back
  - Buffer cache holds in-memory copies of disk blocks
  - Can’t let the buffer cache write back until logged
  - Tricky because cache could run out of memory
- xv6’s solution:
  - Ensure buffer cache is big enough
  - Pin dirty blocks in buffer cache (can’t cycle out)
  - After commit, unpin blocks
- Challenge: Data must fit in the Log
- xv6 solution:
  - Compute upper bound on number of blocks each system call could write
  - Set the log size to be greater than this upper bound
  - Break up some system calls into several transactions
    - E.g., really large write()’s
    - Large writes not atomic, but crash will leave correct prefix

Summary

- Logging makes file system transactions atomic
  - Either they complete fully or not at all
- Write-ahead logging is the key solution in xv6
  - Log written in batches, good for performance
  - But now each disk write happens twice!
  - And have to wait (synchronous) for disk writes
  - Trouble with operations that don't fit in log
- Overall, performance is quite a bit worse
  - Next lecture: How can we make this fast?