



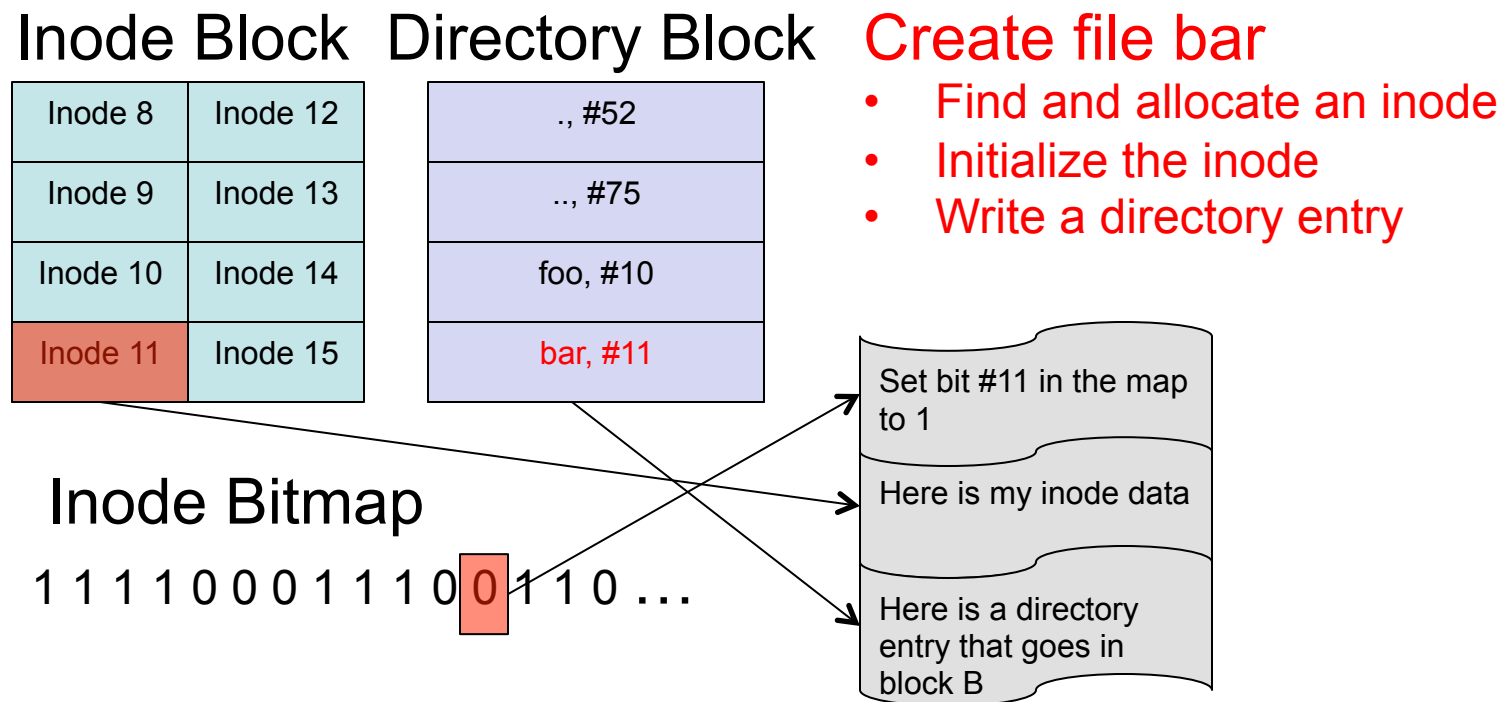
# Journaling File Systems

- Learning Objectives
  - Explain log journaling/logging can make a file system recoverable.
  - Discuss tradeoffs between a journaling file system and FFS with soft updates.
  - Identify shortcomings in journaling file systems
- Topics
  - Those “notes” we attached to buffers in soft updates
  - Turning notes into a log
  - Different approaches to logging



# Log Records

- Log records are “notes” that the running file system leaves for the recovery process.





# What do we do after a crash?

- Read the notes:
  - Use the notes to figure out what you need to do to make the file system consistent after the crash.
  - Might need to undo some operations; why/when?
  - Might need to redo some other operations; why/when?
- Using database parlance, we call these “notes” a **log**.
  - The act of reading the log and deciding what to do is called **recovery**.
  - Backing out an operation is called **undoing** it.
  - Re-applying the operation is called **redoing** it.



# Is this a good idea?

- Why is this logging and recovery better than simply writing the data synchronously?
  - Synchronous writes can appear anywhere on disk.
  - The log is typically a contiguous region of disk, so writing to it is usually quite efficient.
  - You can buffer data in memory longer.



# Journaling File Systems: A Rich History

- The Database Cache (1984)
  - Write data to a sequential log.
  - Lazily transfer data from the sequential log to actual file system.
- The Cedar File System (1987)
  - Log meta-data updates.
  - Log everything twice to recover from any single block failure.
  - Data is not logged.
- IBM's JFS (1990)
  - Log meta-data to a 32 MB log.
  - Also uses B+ tree for directories and extent descriptors.
- Veritas VxFS (1990 or 1991)
  - Maintain and intent log of all file system data structure changes.
  - Data is not logged.
- Many journaling file systems today; some log meta-data only; some log data too; some make data logging optional
  - Ext3
  - Reiser
  - NTFS
  - ZFS
  - BTRFS
  - ...



# Journaling/Logging

- Motivation
  - Sequential writes are fast; random writes are slow.
  - Meta-data operations typically require multiple writes to different data structures (e.g., inodes and directories).
  - **Logging/Journaling converts random I/Os to sequential ones.**
- How a journaling file system works
  - Before writing data structure changes to disk, write a log record that describes the change.
  - Log records typically contain either or both:
    - REDO Information: describes how to apply the update in case the data does not make it to disk.
    - UNDO: Information: describes how to back out the update in case the data gets to disk but shouldn't (because some other update never made it to the disk or the log).
  - Make sure that log records get to disk before the data that the log record describes (write-ahead logging or WAL).
  - After a system crash, replay the log and redo/undo operations as appropriate to restore file system to a consistent state.

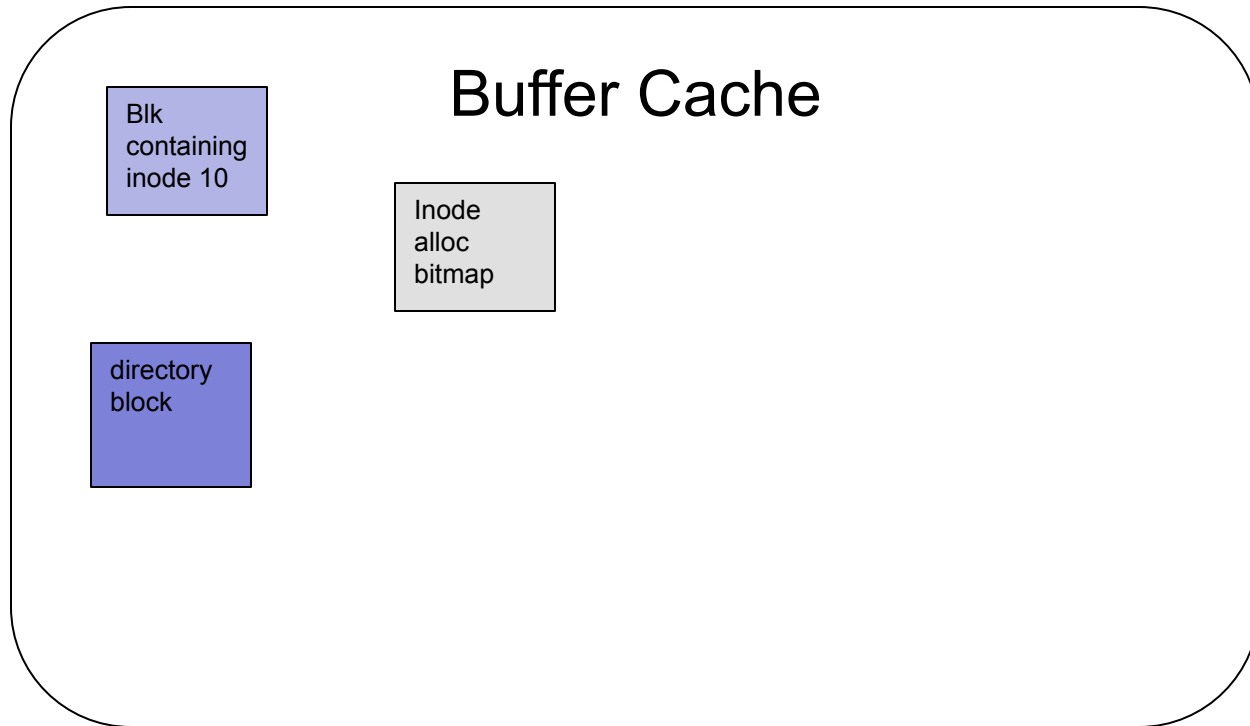






# Journaling Example (1)

Log

Create file A

Allocate and initialize inode 10
Add dir entry A,10



-  FS Bitmaps
-  Inodes
-  Directory Blocks
-  Data Blocks

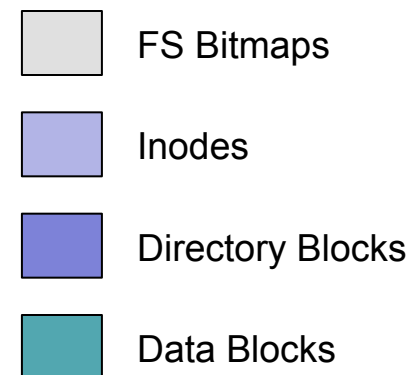
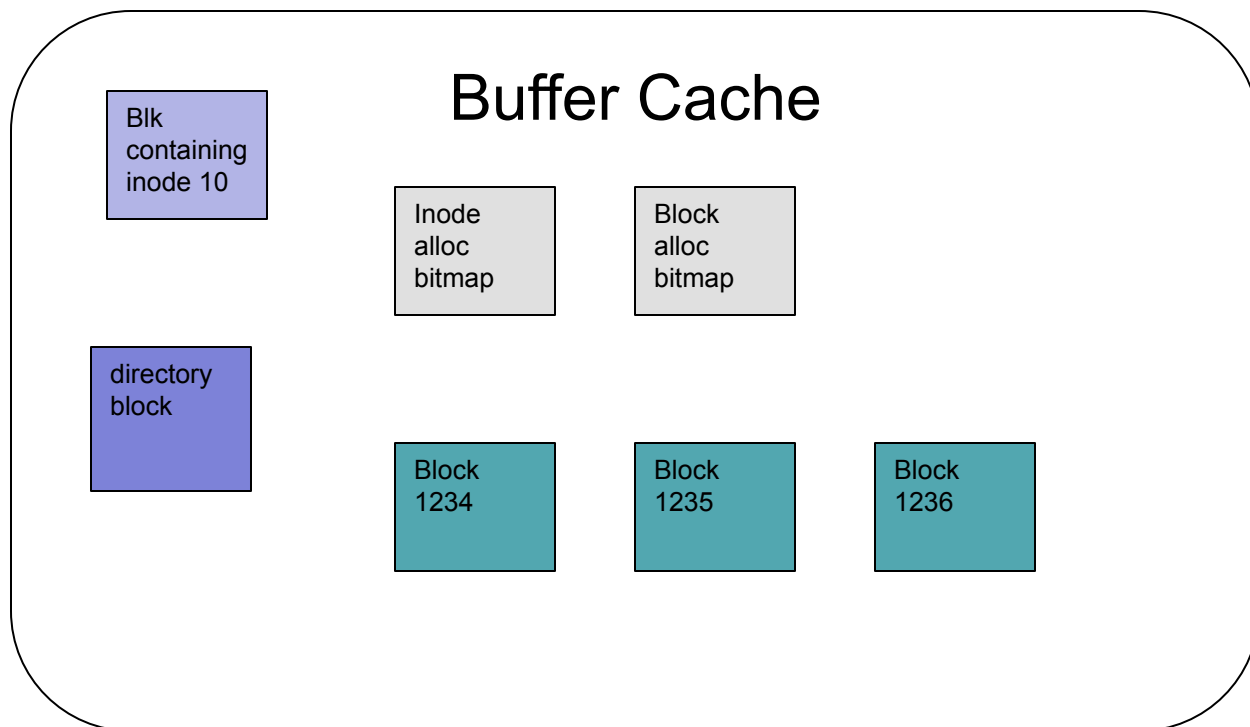


# Journaling Example (2)

## Log

Create file A  
Write blocks 0-2 to file A

Allocate and initialize inode 10
Add dir entry A, 10
Alloc block 1234 to inode 10; lbn 0
Alloc block 1235 to inode 10; lbn 1
Alloc block 1236 to inode 10; lbn 2



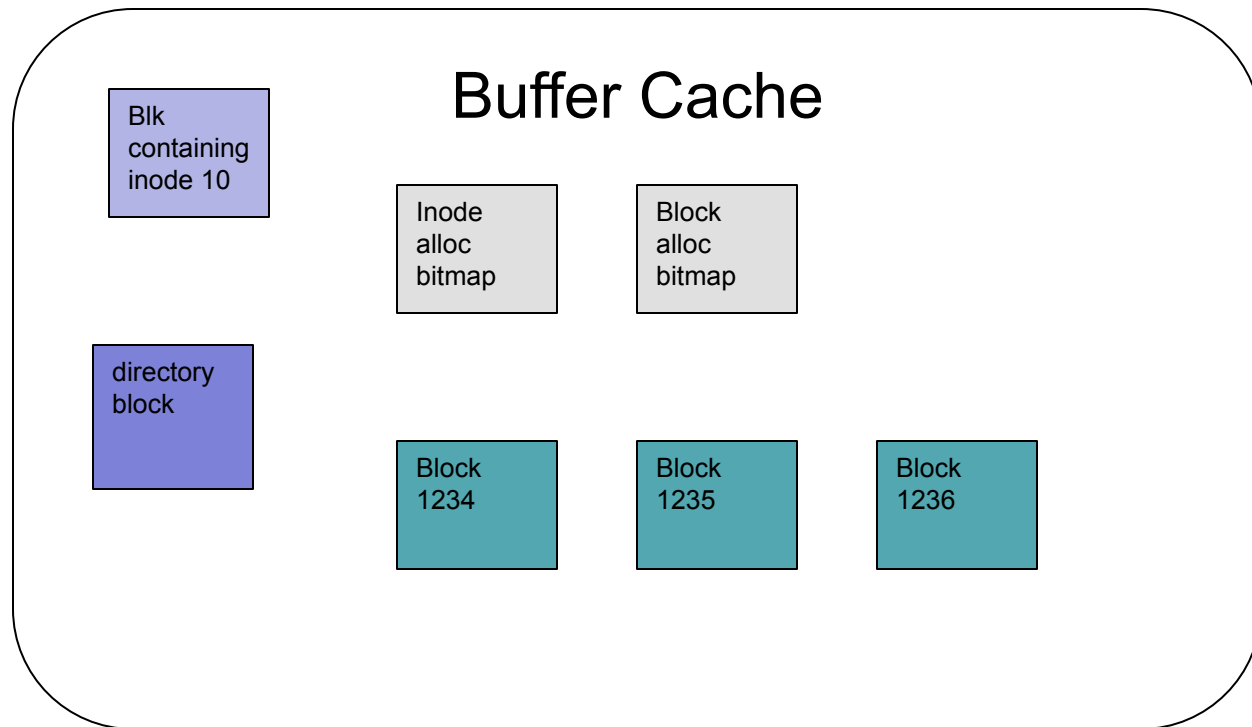




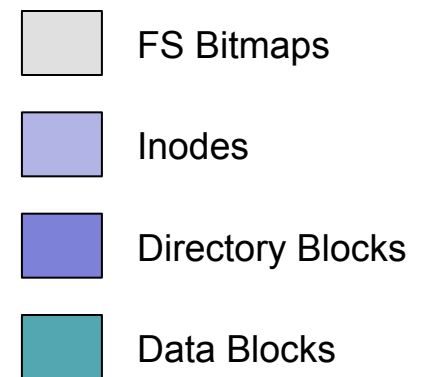
# Journaling Example (3)

Log

Create file a  
Write blocks 0-2 to file A  
**Delete file A**



Remove directory entry A,10
Free block 1234
Free block 1235
Free block 1236
Clear inode 10
Deallocate inode 10

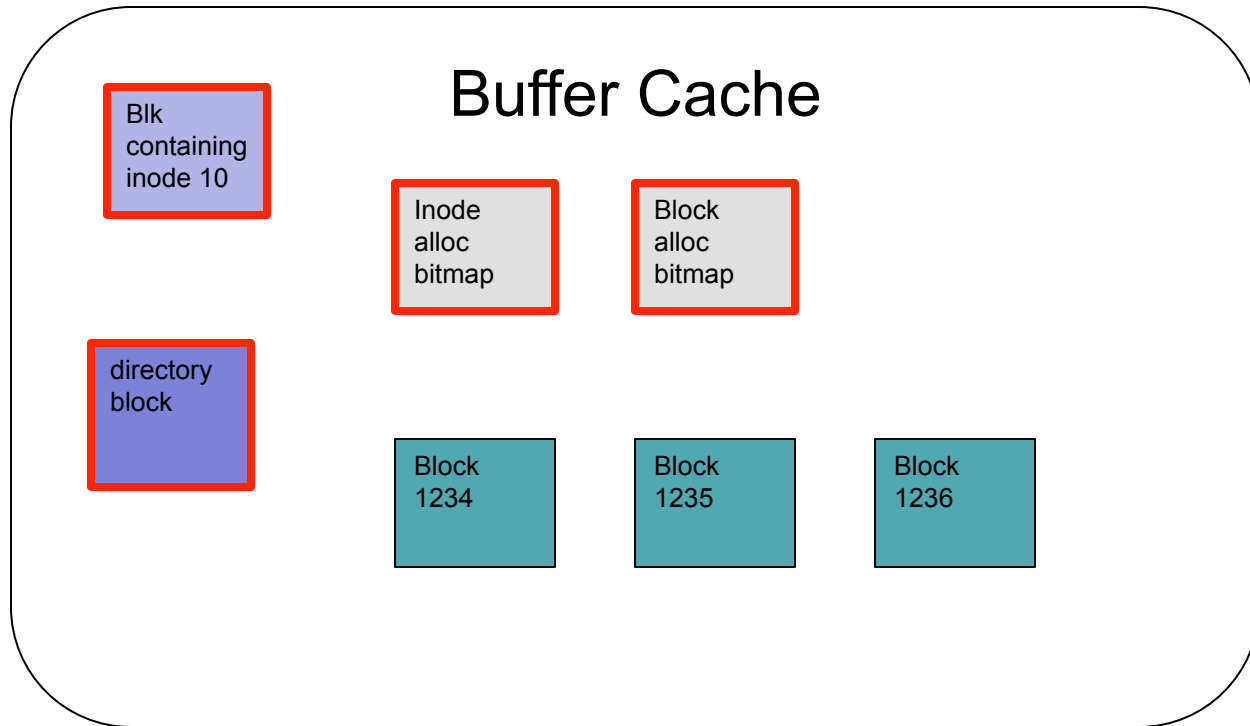




# Journaling Example (4)

Log

Create file a  
Write blocks 0-2 to file A  
Delete file A



- Remove directory entry A, 10
- Free block 1234
- Free block 1235
- Free block 1236
- Clear inode 10
- Deallocate inode 10

- FS Bitmaps
- Inodes
- Directory Blocks
- Data Blocks



# Operations versus Transactions

- Notice that a single file system API call consists of multiple operations:
  - API call: create file A
  - Operations:
    - Allocate inode
    - Initialize inode
    - Create directory entry
- There are a number of different ways that one could record this information in a log record...



# Approaches to Logging: Granularity

- Logical logging
  - Write a single high level record that describes an entire API call: allocate inode 10, initialize it, and create a new directory entry in directory D that names inode 10 “a”
- Physical logging
  - Write a log record per physical block modified:
    - (inode allocation) Here is the old version of page 100; here is the new version (they differ by one bit)
    - (initialize inode) Here is the old version of a block of inodes; here is the new version.
    - (directory entry) Here is a the old version of a block of a directory; here is the new version.
- Operation logging
  - Write a log describing a modification to a data structure:
    - “Allocate inode 10” or “change bit 10 on page P from 0 to 1”
    - “Write the following values into inode 10” or “change bytes 0-31 from these old values to these new values”
    - Add the directory entry <a, 10> into directory D or “change bytes N-M from these old values to these new values”



# “Big Records” vs “Small Records”

- The fundamental difference between high level logical logging and low level operation logging is whether each API call translates into one or more records.
- One record:
  - You can consider the operation done once you have successfully logged that record.
- Multiple records:
  - You may generate records, but not get all the way to the end of the sequence of records and then experience a failure: in that case, even though you have some log records, you can't finish the complete whatever you were doing, because you don't have all the information.
  - A partial sequence must be treated as a failure during recovery.



# Transactions

- We call the sequence of operations comprising a single logical operation (API call) a transaction.
- Transactions come from the database world where you want to apply multiple transformations to a data collection, but have all the operations appear **atomically** (either all appear or non appear).
- In the database world, we use the ACID acronym to describe transactions:
  - A: Atomic: either all the operations appear or none do
  - C: Consistent: each transaction takes the data from one consistent state to another.
  - I: Isolation: each transactions behaves as if it's the only one running on the system.
  - D: Durable: once the system responds to a **commit** request, the transaction is complete and must be complete regardless of any failures that might happen.



# File System Transactions

- File systems typically abide by the ACI properties, but may not guarantee durability.
  - Example: we might want the write system call to follow the ACI properties, but we may not require a disk write and might tolerate some writes not being persistent after a failure.
- Transactions:
  - Are an elegant way to deal with error handling (just abort the transaction).
  - Let you recover after a failure: roll forward committed transactions; roll back uncommitted ones.



# Transaction APIs

- Begin transaction
- Commit transaction: make all the changes appear
- Abort transaction: make all the changes disappear (as if the transaction never happened).
- Prepare (used for distributed transactions, but we don't need it here).
- For recovery purposes, any transaction that does not commit is aborted.





# What to log: Undo/Redo

- Undo information lets you back out things during recovery.
  - Under what conditions might you want to back out things?
  - Assuming that we only use the log to recover after a failure, under what conditions would you never need UNDO records?
- Redo information lets you roll a transaction forward.
  - Under what conditions might you need to do this?
  - Under what conditions would you never need REDO records?



# What do log: Undo/Redo

- Undo information lets you back out things during recovery.
  - Under what conditions might you want to back out things?
    - An API call did not finish, but has started modifying some data.
  - Assuming that we only use the log to recover after a failure, under what conditions would you never need UNDO records?
    - If any data in an active transactions is never allowed to go to disk.
- Redo information lets you roll a transaction forward.
  - Under what conditions might you need to do this?
    - A transaction completed, but not all the data got written to disk.
  - Under what conditions would you never need REDO records?
    - Part of the commit required writing all the changes to disk.



# Implementation Details

- We provide:
  - A log container into which you can write records and from which you can read and iterate over records during recovery. You will want to become familiar with the APIs to this code.
- We require:
  - Operation logging
  - Log only metadata: your goal is to make sure that the meta data of the file system is consistent after recovery; you need not guarantee that all the data is intact.
- We recommend:
  - UNDO/REDO logging



# Journaling Pros/Cons

- Advantages
  - Write log records sequentially
  - Data (random) writes can be buffered much longer.  
Implications:
    - You may get multiple writes into the buffer cache before you have to write to disk.
    - You can accumulate a large number of blocks to write to disk and then schedule them cleverly, so they are faster than the same number of random writes..
  - Facilitate fast recovery: just replay the log (no costly FSCK).
- Disadvantages
  - If you do not log data, then you can end up with intact file system structures, but incorrect or missing data.
  - You write everything twice.



# Looking Forward

- Annoying tidbits
  - You end up writing all your meta-data twice: once to the log and once to the file system.
  - If you want the data recoverable, then you have to log that too and that means you're writing the data twice.

*Isn't there a better way???*
- Two approaches we'll study:
  - Could we enforce ordering constraints without A) journaling everything or B) issuing a lot of synchronous writes?
  - Is there a way to make the log records themselves be the actual data, so you don't have to write everything twice (once to the log and once to the file system data).