COS 318: Operating Systems

File Performance and Reliability

Vivek Pai
Computer Science Department
Princeton University

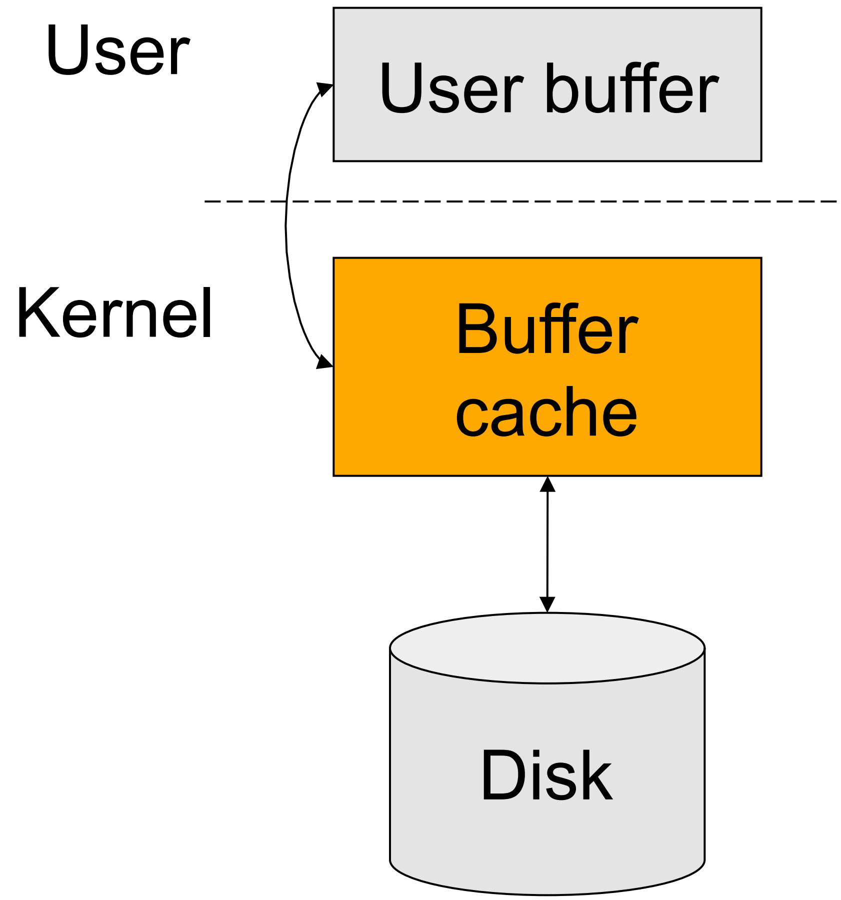
http://www.cs.princeton.edu/courses/archive/fall11/cos318/

Topics

- File buffer cache
- Disk failure and file recovery tools
- Consistent updates
- Transactions and logging

File Buffer Cache for Performance

- Cache files in main memory
  - Check the buffer cache first
  - Hit will read from or write to the buffer cache
  - Miss will read from the disk to the buffer cache
- Usual questions
  - What to cache?
  - How to size?
  - What to prefetch?
  - How and what to replace?
  - Which write policies?

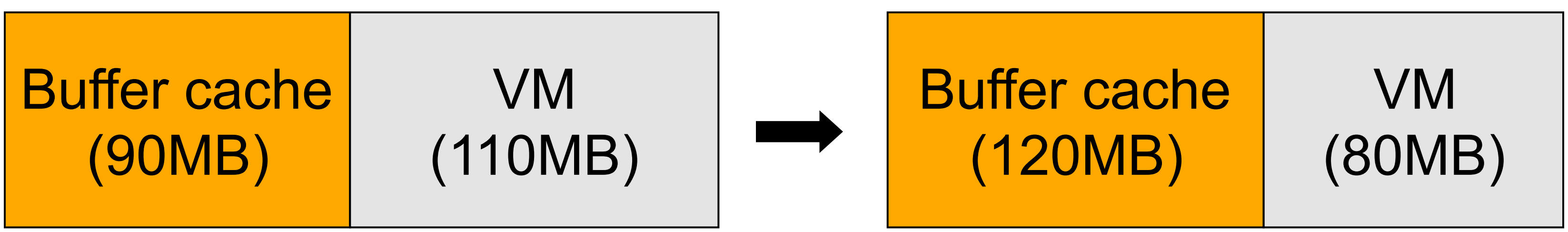


What to Cache?

- Things to consider
  - i-nodes and indirect blocks of directories
  - Directory files
  - I-nodes and indirect blocks of files
  - Files
- What is a good strategy?
  - Cache i-nodes and indirect blocks if they are in use?
  - Cache only the i-nodes and indirect blocks of the current directory?
  - Cache an entire file vs. referenced blocks of files

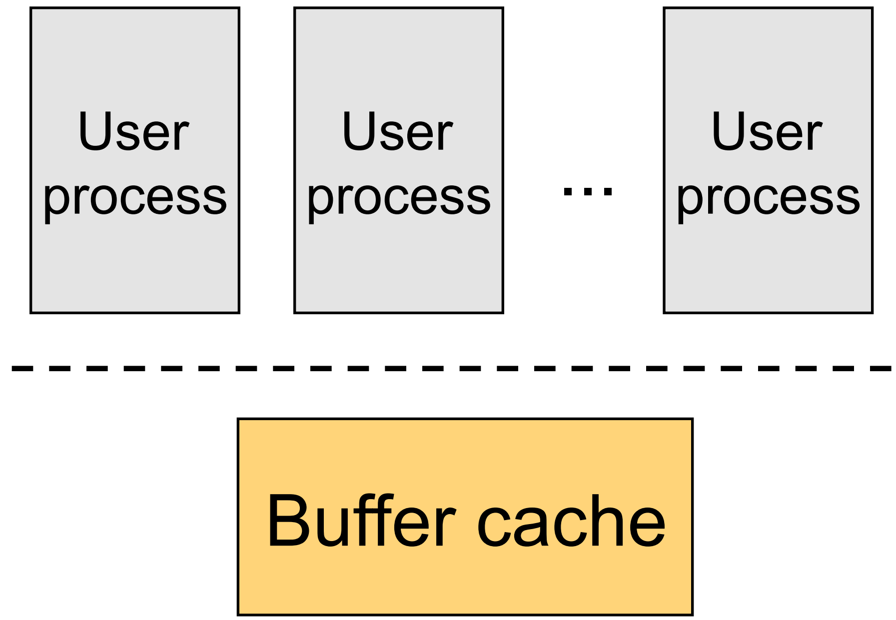
How to Size?

- An important issue is how to partition memory between the buffer cache and VM cache
- Early systems use fixed-size buffer cache
  - It does not adapt to workloads
- Later systems use variable size cache
  - But, large files are common, how do we make adjustment?
- Solution
  - Basically, we solve the problem using the working set idea, remember?



Challenges: Multiple User Processes

- Kernel
  - All processes share the same buffer cache
  - Global LRU may not be fair
- Solution
  - Working set idea again
- Questions
  - Can each process use a different replacement strategy?
  - Can we move the buffer cache to the user level?
  - What about duplications?
  - Do we need to pin user buffers?

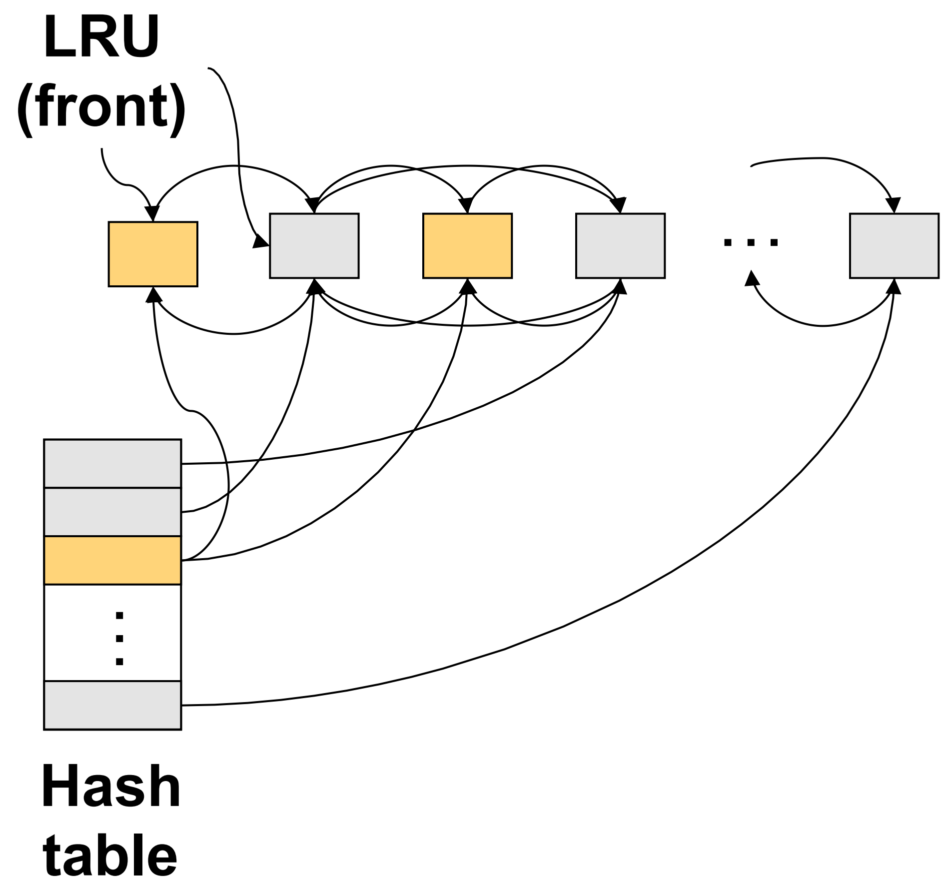


What to Prefetch?

- Optimal
  - The blocks are fetched in just enough time to use them
  - But, life is hard
- The good news is that files have locality
  - Temporal locality
  - Spatial locality
- Common strategies
  - Prefetch next k blocks together (typically > 64KB)
  - Some discard unreferenced blocks
  - Cluster blocks (to the same cylinder group and neighborhood) make prefetching efficient, directory and i-nodes if possible

How and What to Replace?

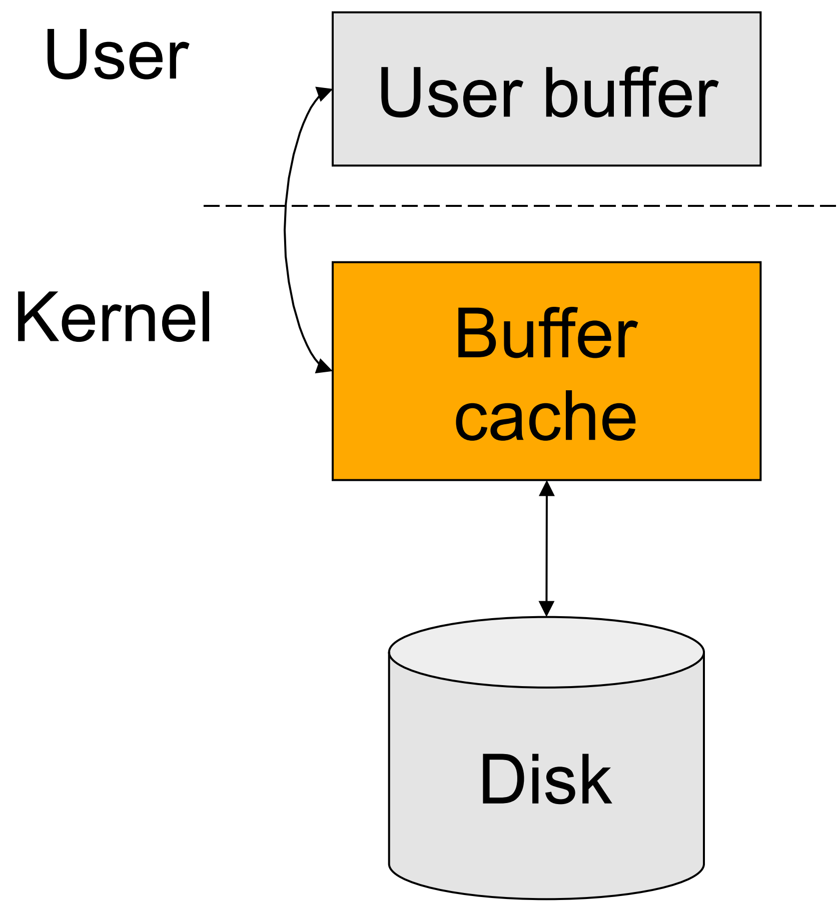
- Page replacement theory
  - Use past to predict future
  - LRU is good
- Buffer cache with LRU replacement mechanism
  - If b is in buffer cache, move it to front and return b
  - Otherwise, replace the tail block, get b from disk, insert b to the front
  - Use double linked list with a hash table
- Questions
  - Why a hash table?
  - What if file >> the cache?



Which Write Policies?

- Write through
  - Whenever modify cached block, write block to disk
  - Cache is always consistent
  - Simple, but causes more I/Os

- Write back
  - When modifying a block, mark it as dirty & write to disk later
  - Fast writes, absorbs writes, and enables batching
  - So, what’s the problem?

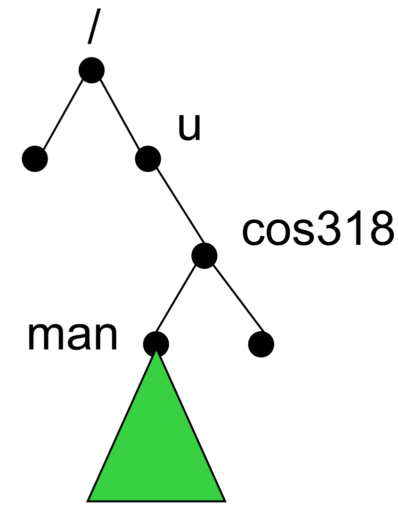


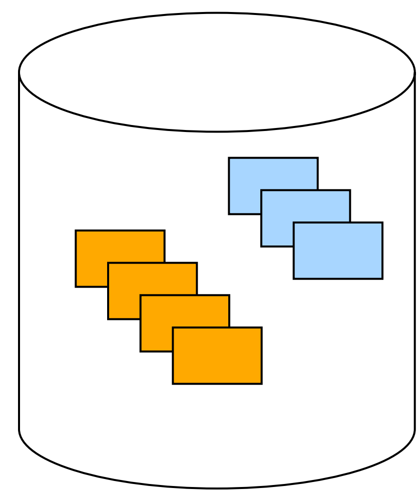
Write Back Complications

- Fundamental tension
  - On crash, all modified data in cache is lost.
  - The longer you postpone write backs, the faster you are and the worse the damage is
- When to write back
  - When a block is evicted
  - When a file is closed
  - On an explicit flush
  - When a time interval elapses (30 seconds in Unix)
- Issues
  - These write back options have no guarantees
  - A solution is consistent updates (later)

File Recovery Tools

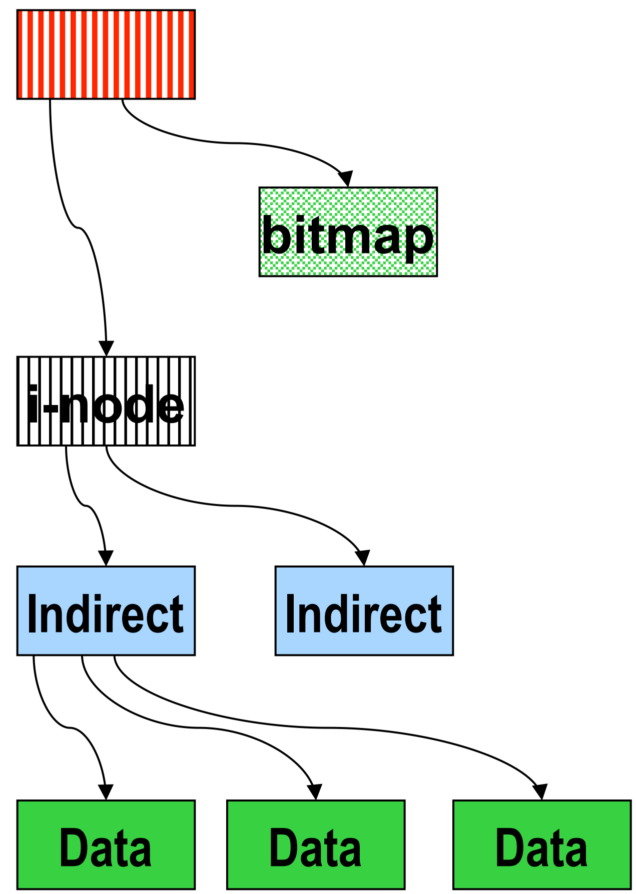
- Physical backup (dump) and recovery
  - Dump disk blocks by blocks to a backup system
  - Backup only changed blocks since the last backup as an incremental
  - Recovery tool is made accordingly
- Logical backup (dump) and recovery
  - Traverse the logical structure from the root
  - Selectively dump what you want to backup
  - Verify logical structures as you backup
  - Recovery tool selectively move files back
- Consistency check (e.g. fsck)
  - Start from the root i-node
  - Traverse the whole tree and mark reachable files
  - Verify the logical structure
  - Figure out what blocks are free





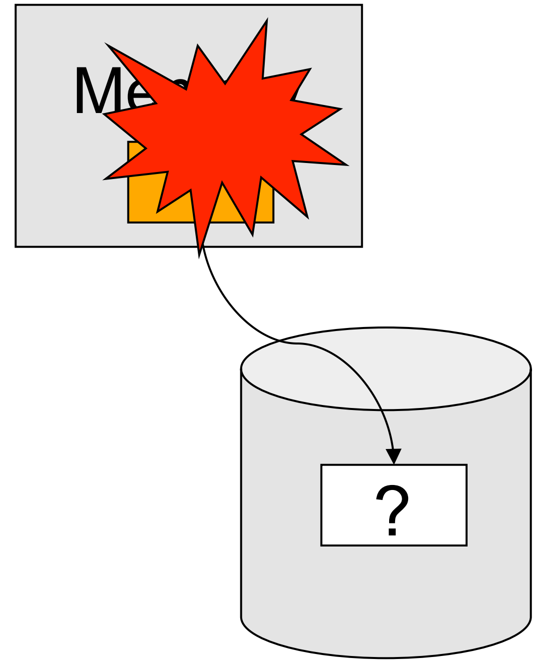
Recovery from Disk Block Failures

- Boot block
  - Create a utility to replace the boot block
  - Use flash memory to duplicate the boot block and kernel
- Super block
  - If there is a duplicate, remake the file system
  - Otherwise, what would you do?
- Free block data structure
  - Search all reachable files from the root
  - Unreachable blocks are free
- i-node blocks
  - How to recover?
- Indirect or data blocks
  - How to recover?



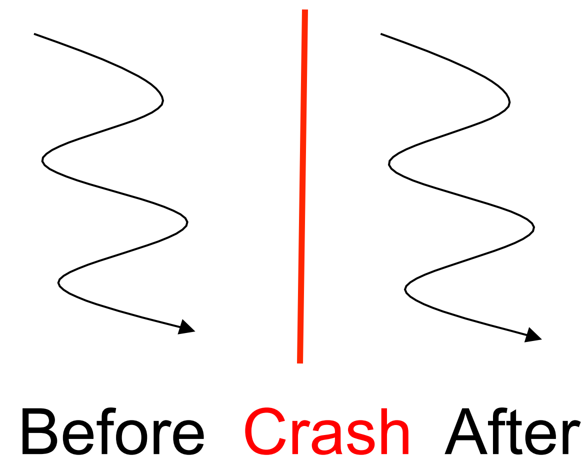
Persistency and Crashes

- File system promise: Persistency
  - File system will hold a file until its owner explicitly deletes it
  - Backups can recover your file even beyond the deletion point
- Why is this hard?
  - A crash will destroy memory content
  - Cache more ⇒ better performance
  - Cache more ⇒ lose more on a crash
  - A file operation often requires modifying multiple blocks, but the system can only atomically modify one at a time
  - Systems can crash anytime

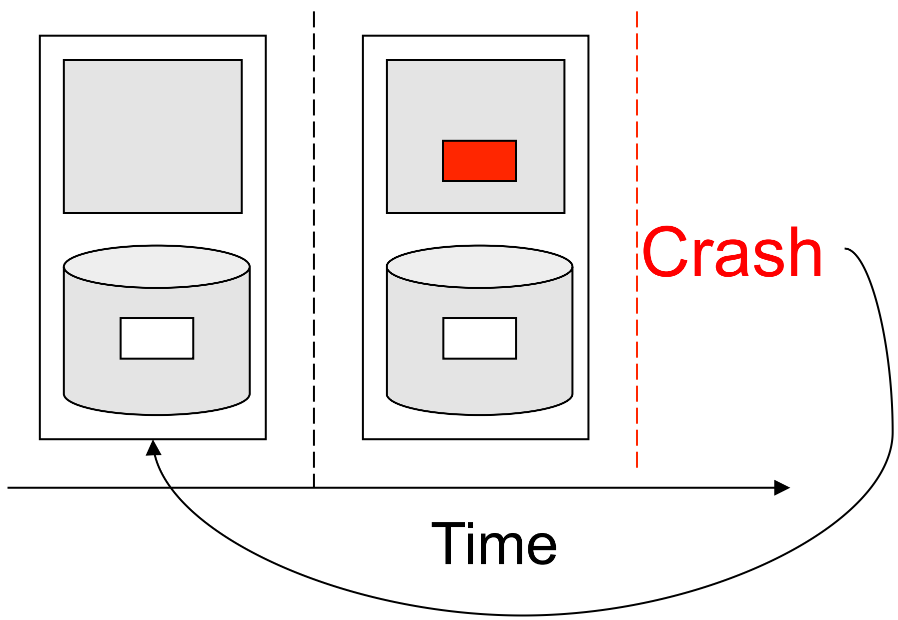


What Is A Crash?

- Crash is like a context switch
  - Think about a file system as a thread before the context switch and another after the context switch
  - Two threads read or write same shared state?



- Crash is like time travel
  - Current volatile state lost; suddenly go back to old state
  - Example: move a file
    - Place it in a directory
    - Delete it from old
    - Crash happens and both directories have problems



Approaches

- Throw everything away and start over
  - Done for most things (e.g., make again)
  - Not what you want to happen to your email
- Reconstruction
  - Figure out where you are and make the file system consistent and go from there
  - Try to fix things after a crash ("fsck")
- **Make consistent updates**
  - Either new data or old data, but not garbage data
- **Make multiple updates appear atomic**
  - Build arbitrary sized atomic units from smaller atomic ones
  - Similar to how we built critical sections from locks, and locks from atomic instructions

Write Metadata First

- Modify /u/cos318/foo
  - Traverse to /u/cos318/

Crash → **Consistent**

  - Allocate data block

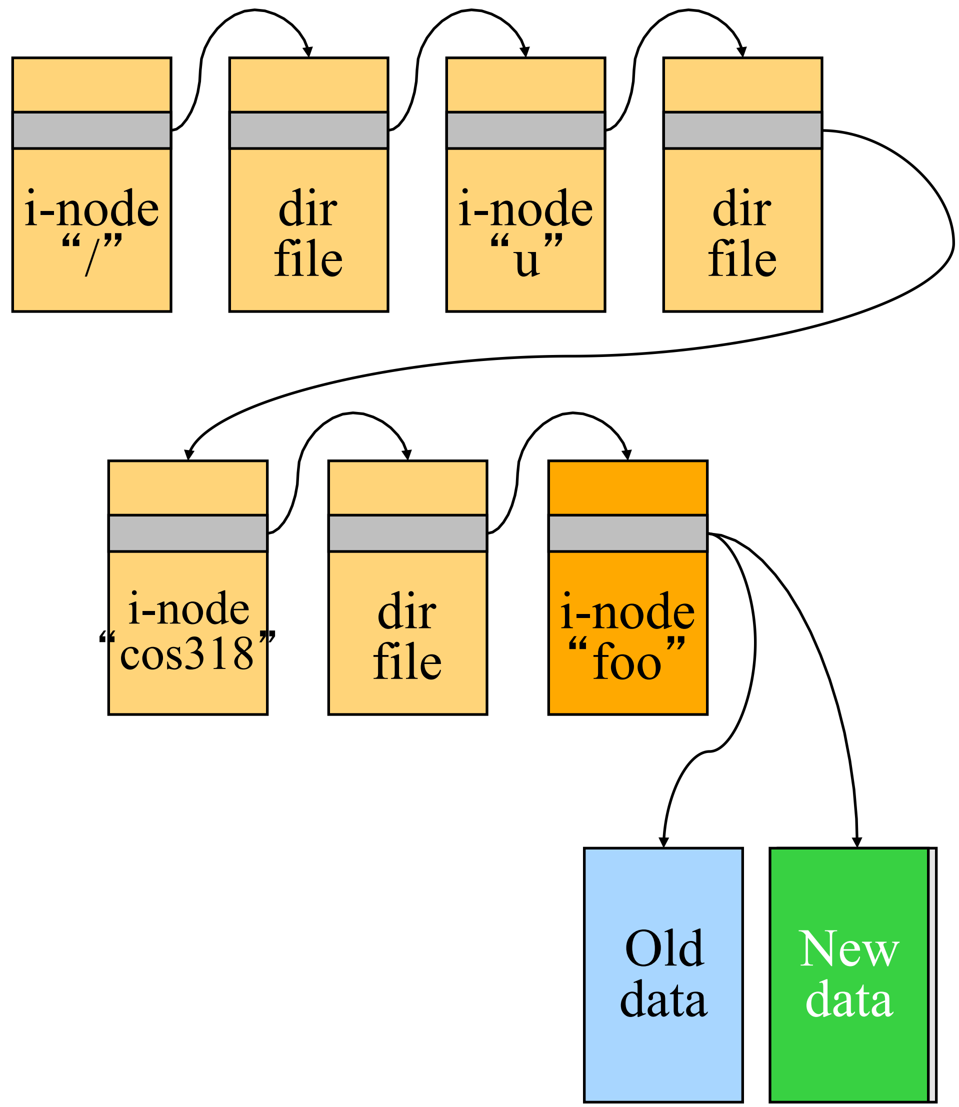
Crash → **Consistent**

  - Write pointer into i-node

Crash → **Inconsistent**

  - Write new data to foo

Crash → **Consistent**



Writing metadata first can cause inconsistency

Write Data First

- Modify /u/cos318/foo
  - Traverse to /u/cos318/

Crash → **Consistent**

  - Allocate data block

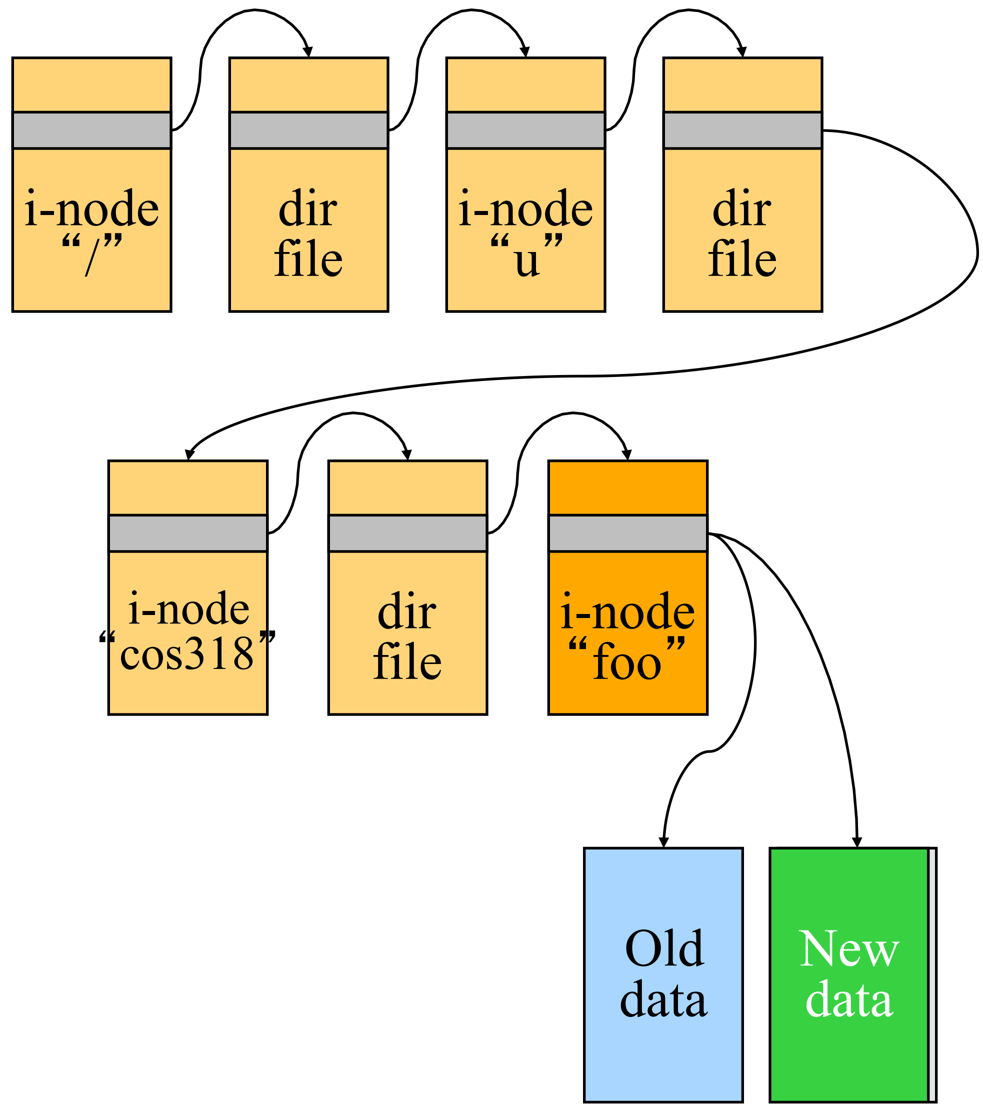
Crash → **Consistent**

  - Write new data to foo

Crash → **Consistent**

  - Write pointer into i-node

Crash → **Consistent**



Consistent Updates: Bottom-Up Order

- The general approach is to use a “bottom up” order
  - File data blocks, file i-node, directory file, directory i-node, …
- What about file buffer cache
  - Write back all data blocks
  - Update file i-node and write it to disk
  - Update directory file and write it to disk
  - Update directory i-node and write it to disk (if necessary)
  - Continue until no directory update exists
- Does this solve the write back problem?
  - Updates are consistent but leave garbage blocks around
  - May need to run fsck to clean up once a while
  - Ideal approach: consistent update without leaving garbage

Transaction Properties

- Group multiple operations together so that they have "ACID" property:
  - Atomicity
    - It either happens or doesn't (no partial operations)
  - Consistency
    - A transaction is a correct transformation of the state
  - Isolation (serializability)
    - Transactions appear to happen one after the other
  - Durability (persistency)
    - Once it happens, stays happened
- Question
  - Do critical sections have ACID property?

Transactions

- Bundle many operations into a transaction
  - One of the first transaction systems is Sabre American Airline reservation system, made by IBM
- Primitives
  - BeginTransaction
    - Mark the beginning of the transaction
  - Commit (End transaction)
    - When transaction is done
  - Rollback (Abort transaction)
    - Undo all the actions since "Begin transaction."
- Rules
  - Transactions can run concurrently
  - Rollback can execute anytime
  - Sophisticated transaction systems allow nested transactions

Implementation

- BeginTransaction
  - Start using a "write-ahead" log on disk
  - Log all updates
- Commit
  - Write "commit" at the end of the log
  - Then "write-behind" to disk by writing updates to disk
  - Clear the log
- Rollback
  - Clear the log
- Crash recovery
  - If there is no "commit" in the log, do nothing
  - If there is "commit," replay the log and clear the log

- Assumptions
  - Writing to disk is correct (recall the error detection and correction)
  - Disk is in a good state before we start

Use Transactions in File Systems

- Make a file operation a transaction
  - Create a file
  - Move a file
  - Write a chunk of data
  - …
  - Would this eliminate any need to run fsck after a crash?
- Make arbitrary number of file operations a transaction
  - Just keep logging but make sure that things are idempotent: making a very long transaction
  - Recovery by replaying the log and correct the file system
  - This is called logging file system or journaling file system
  - Almost all new file systems are journaling (Windows NTFS, Veritas file system, file systems on Linux)

Issue with Logging: Performance

- For every disk write, we now have two disk writes (on different parts of the disk)?
  - It is not so bad because once written to the log, it is safe to do real writes later
- Performance tricks
  - Changes made in memory and then logged to disk
  - Log writes are sequential (synchronous writes can be fast if on a separate disk)
  - Merge multiple writes to the log with one write
  - Use NVRAM (Non-Volatile RAM) to keep the log

Log Management

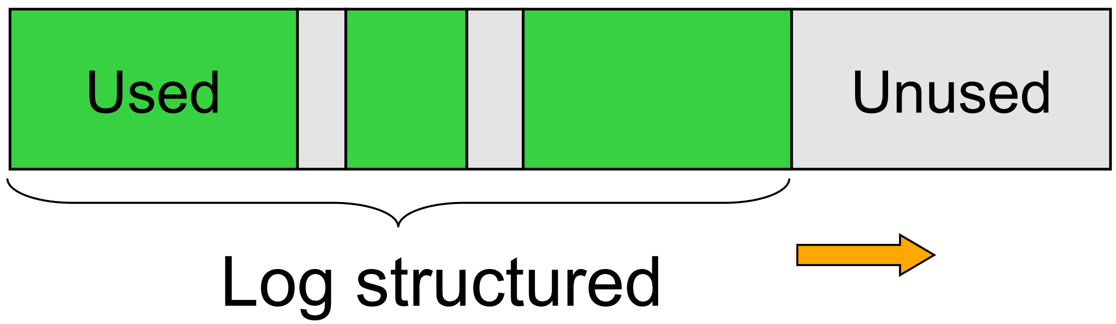
- How big is the log? Same size as the file system?
- Observation
  - Log what's needed for crash recovery
- Management method
  - Checkpoint operation: flush the buffer cache to disk
  - After a checkpoint, we can truncate log and start again
  - Log needs to be big enough to hold changes in memory
- Some logging file systems log only metadata (file descriptors and directories) and not file data to keep log size down
  - Would this be a problem?

What to Log?

- Physical blocks (directory blocks and inode blocks)
  - Easy to implement but takes more space
  - Which block image?
    - Before operation: Easy to go backward during recovery
    - After operation: Easy to go forward during recovery.
    - Both: Can go either way.
- Logical operations
  - Example: Add name “foo” to directory #41
  - More compact
  - But more work at recovery time

Log-structured File System (LFS)

- Structure the entire file system as a log with segments
- A segment has i-nodes, indirect blocks, and data blocks
- All writes are sequential (no seeks)
- There will be holes when deleting files
- Questions
  - What about read performance?
  - How would you clean (garbage collection)?



Summary

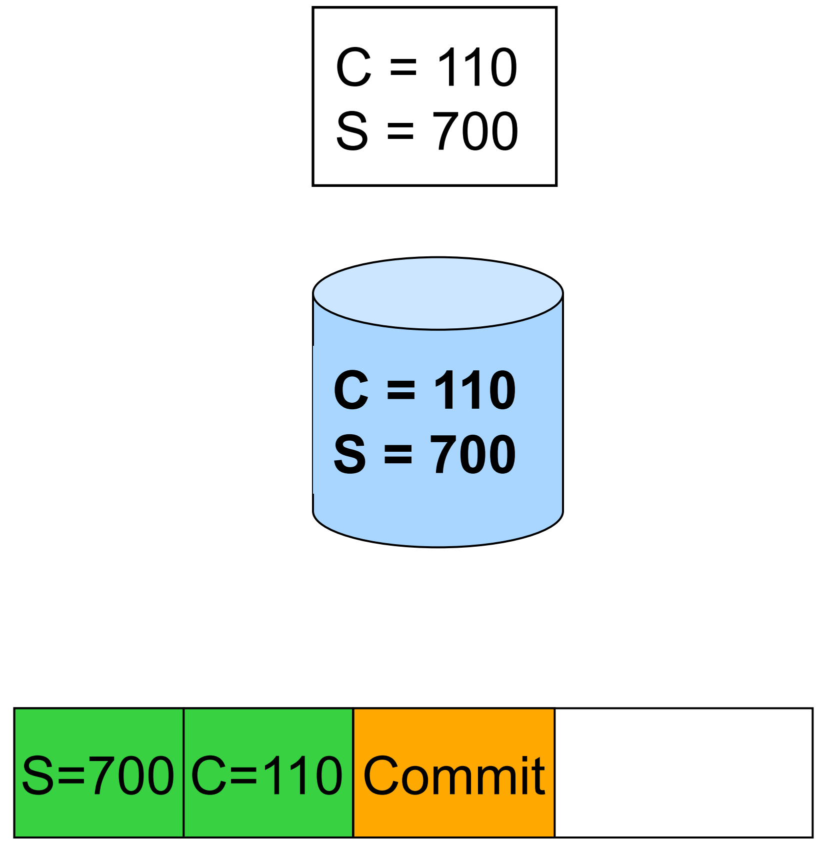
- File buffer cache
  - True LRU is possible
  - Simple write back is vulnerable to crashes
- Disk block failures and file system recovery tools
  - Individual recovery tools
  - Top down traversal tools
- Consistent updates
  - Transactions and ACID properties
  - Logging or Journaling file systems

An Example: Atomic Money Transfer

- Move $100 from account S to C (1 thread):

```
BeginTransaction
    S = S - $100;
    C = C + $100;
Commit
```

- Steps:
  1: Write new value of S to log
  2: Write new value of C to log
  3: Write commit
  4: Write S to disk
  5: Write C to disk
  6: Clear the log
- Possible crashes
  - After 1
  - After 2
  - After 3 before 4 and 5
- Questions
  - Can we swap 3 with 4?
  - Can we swap 4 and 5?



Revisit The Implementation

- BeginTransaction
  - Start using a “write-ahead” log on disk
  - Log all updates
- Commit
  - Write “commit” at the end of the log
  - Then “write-behind” to disk by writing updates to disk
  - Clear the log
- Rollback
  - Clear the log
- Crash recovery
  - If there is no “commit” in the log, do nothing
  - If there is “commit,” replay the log and clear the log

- Questions
  - What is “commit?”
  - What if there is a crash during the recovery?

Two Threads Run Transactions

- Apply to the mid-term AtomicTransfer program

```
1: BeginTransaction
2: if ( a1->id < a2->id ) {
         Acquire( a1->lock ); Acquire( a2->lock );
   } else {
         Acquire( a2->lock ); Acquire( a1->lock );
   }
3: if ((a1->balance - $100 ) < 0) {
         Release( a2->lock ); Release( a1->lock );
         goto 7;
    }
4: a1->balance -= $100;
5: a2->balance += $100;
6: Release( a2->lock ); Release( a1->lock );
7: Commit
```

- What happens if
  - Thread A performs 1-6; context switch
  - Thread B performs 1-7; crash!

Two-Phase Locking for Transactions

- First phase
  - Acquire all locks
- Second phase
  - Commit operation release all locks (no individual release operations)
  - Rollback operation always undo the changes first and then release all locks