COS 318: Operating Systems

File System Reliability

Kai Li
Computer Science Department
Princeton University

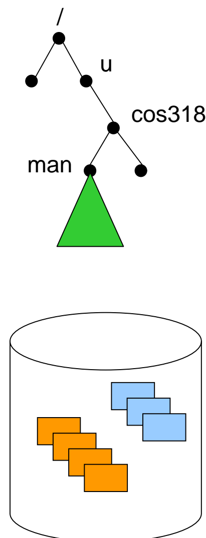
(http://www.cs.princeton.edu/courses/cos318/)

Topics

- Disk block failures and file system recovery tools
- Crashes and recovery

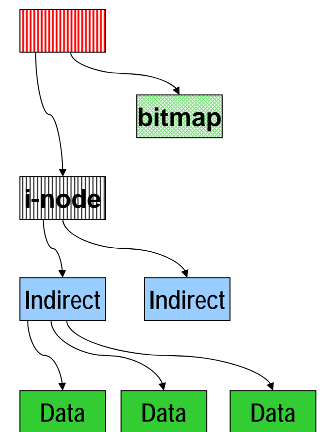
File Recovery Tools

- Physical dump
  - Dump disk blocks by blocks to a backup system
  - Backup only changed blocks since the last backup as an incremental
  - Recovery tool will recover blocks of a backup
- Logical dump
  - 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 from the backup
- Consistency check
  - Start from the root i-node
  - Traverse the entire tree
  - Verify the logical structure
  - Figure out what blocks are free



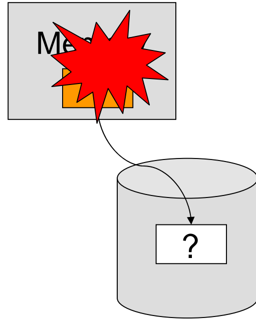
Recovery from Disk Block Failures

- Boot block
  - Create a utility to replace the boot block
  - Use a 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?



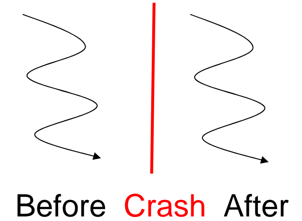
Persistency and Crashes

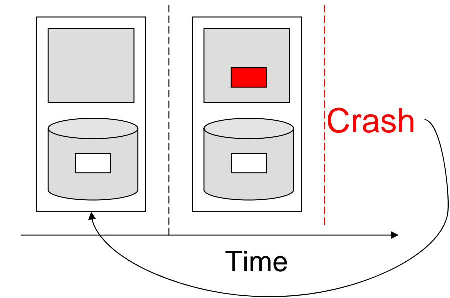
- File system promise: Persistency
  - File system will hold a file until its owner explicitly deletes it
  - Backups can recover your file even beyond the deletion point
- Why is this hard? Crashes
  - A crash will destroy memory content
  - Cache more ⇒ lose more on a crash
  - Cache more ⇒ better performance
  - 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



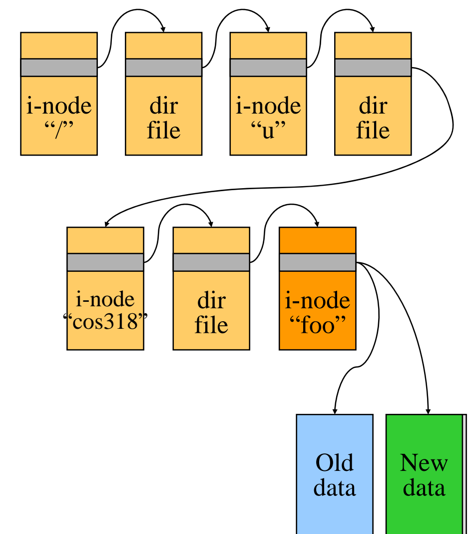


Approaches

- 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 what state you are in, make the file system consistent and go from there
  - Try to fix things after 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

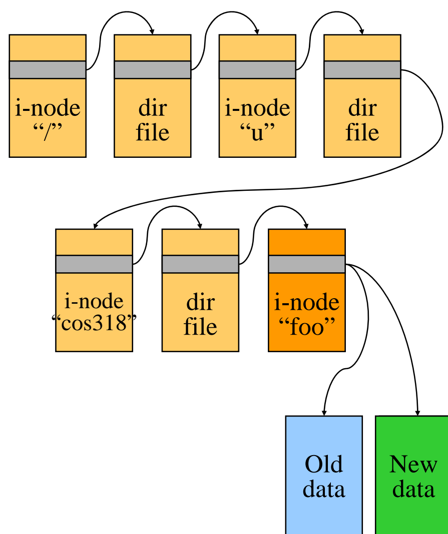
File Consistency Issue

- It often takes multiple I/O operations per file write
  - System does one I/O at a time
  - Crash can happen anytime
- Example
  - Modify /u/cos318/foo
- Write metadata first
  - A crash
  - Allocate data block
  - A crash
  - Write pointer into i-node
  - A crash
  - Write new data to foo
  - A crash



Consequences of Writing Data First

- Example
  - Modify /u/cos318/foo
- Write metadata first
  - A crash
  - Allocate data block
  - A crash
  - Write new data to foo
  - A crash
  - Write pointer to i-node
  - A crash



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
- Is this enough to deal with non-disk crash failures?

Making Multiple Updates "Atomic"

- Bundle multiple operations together such that they execute indivisibly
  - Just like how we made critical sections out of locks and locks out of atomic instructions
  - We are now making atomic disk updates against crashes
- The basic method is transaction
  - Implement transactions with state duplication
  - One of the first transaction systems is Sabre American Airline reservation system, made by IBM

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?

Transaction 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."

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

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?

C = 110
S = 700

C = 110
S = 700

S=700	C=110	Commit	

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( a1->lock ); Acquire( a2->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

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 to keep the log

Log Management

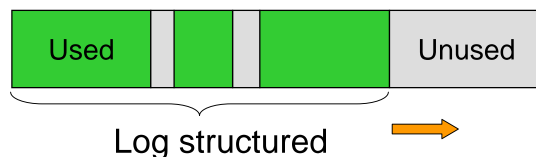
- How big is the log? Same size as the file system?
- Observation
  - Log what's needed for crash recovery
- Management method
  - Checkpoint operation: flush the buffer cache to disk
  - After a checkpoint, we can truncate log and start again
  - Log needs to be big enough to hold changes in memory
- Most 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

- Disk block failures and file system recovery tools
  - Individual recovery tools
  - Top down traversal tools
- File systems should be designed to avoid using recovery tools
  - Transactions and ASID properties
  - Logging or Journaling file system
- Limitations of file system designs
  - Disasters, human faults and virus attacks?
  - Backups for active failover and recovery of any data