



COS 318: Operating Systems

File Performance and Reliability



Topics

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



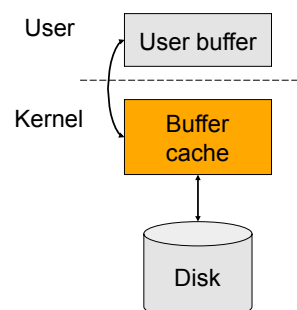
2


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?

User

Kernel






3

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



4

How to Size?

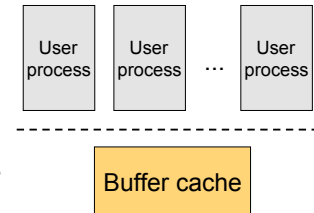
- ◆ An important issue is how to partition memory between the buffer cache and VM cache
- ◆ Early systems use fixed-size buffer cache
 - It does not adapt to workloads
- ◆ Later systems use variable size cache
 - But, large files are common, how do we make adjustment?
- ◆ Solution
 - Basically, we solve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



5

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?



6

What to Prefetch?

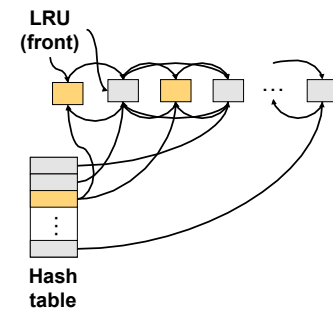
- ◆ Optimal
 - The blocks are fetched in just enough time to use them
 - But, too hard to do
- ◆ The good news is that files also have locality
 - Temporal locality
 - Spatial locality
- ◆ Common strategies
 - Prefetch next k blocks together (typically $> 64\text{KB}$)
 - Some discard unreferenced blocks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



7

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



8

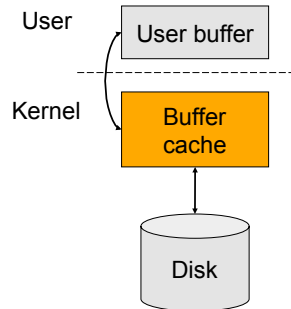
Which Write Policies?

Write through

- Whenever modify cached block, write block to disk
- Cache is always consistent
- Simple, but cause 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?



9

Write Back Complications

Fundamental tension

- On crash, all modified data in cache is lost.
- The longer you postpone write backs, the faster you are but the worst the damage is on a crash

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)



10

File Recovery Tools

Physical backup (dump) and recovery

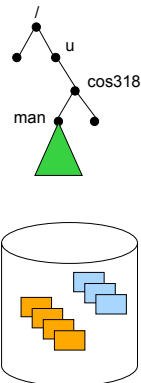
- Dump disk blocks by blocks to a backup system
- Backup only changed blocks since the last backup as an incremental
- Recovery tool built accordingly

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



11

What fsck does

Get default list of filesystems to check from */etc/fstab*

Inconsistencies checked:

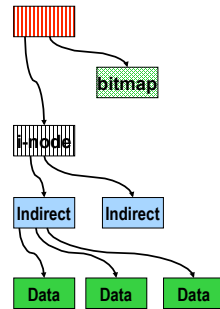
- Blocks claimed by more than one inode or the free map
- Blocks claimed by an inode outside range of the filesystem
- Incorrect link counts
- Size checks (directory size etc)
- Bad inode format
- Blocks not accounted for anywhere
- Directory checks:
 - File pointing to unallocated inode; Inode number out of range; . or .. not first two entries of a directory or have wrong inode number
- Super Block checks
 - More blocks for inodes than are in the filesystem; Bad free block map format; Total free block and/or free inode count incorrect
- Put orphaned files and directories in lost+found directory



12

Recovery from Disk Block Failures

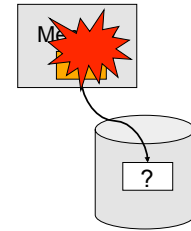
- ◆ Boot block
 - Create a utility to replace the boot block
 - Use a flash memory to duplicate the boot block and kernel
- ◆ Super block
 - If there is a duplicate, remake file system
- ◆ Free block data structure
 - Search all reachable files from the root
 - Unreachable blocks are free
- ◆ i-node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



13

Persistence and Crashes

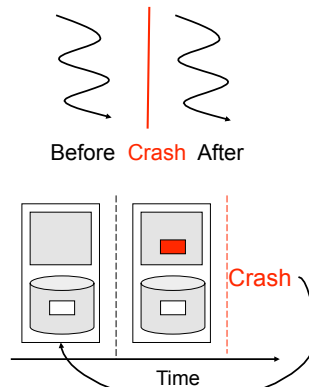
- ◆ File system promise: Persistence
 - File system will hold a file until its owner explicitly deletes it
- ◆ Why is this hard?
 - A crash will destroy memory content
 - Cache more \Rightarrow better performance
 - Cache more \Rightarrow lose more on a crash
 - A file operation often requires modifying multiple blocks, but the system can only atomically modify one at a time
 - Systems can crash anytime



14

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



15

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 updates consistent
 - 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

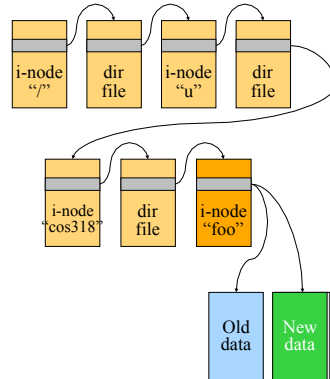


16

Consistent Updates: Problem

♦ 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**



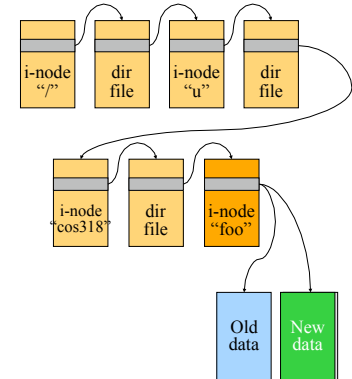
Writing metadata first can cause inconsistency

17

Consistent Updates: Data Before Metadata

♦ 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**



18

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



19

Transactions

- ♦ E.g. move money from Savings to Checking in bank

BEGIN_TRANSACTION

Success = Withdraw (\$300, Savings);

Success = Success AND Deposit (\$300, Checking);

IF (!Success) THEN ABORT_TRANSACTION ELSE

END_TRANSACTION



20

Transaction Properties

- ◆ Group multiple operations together so that they have “ACID” property:
 - Atomicity
 - Consistency
 - Isolation (Serializability)
 - Durability (Persistence)
 - Once it happens, stays happened
- ◆ Question
 - Do critical sections have ACID property?



21

Transaction Properties

Atomicity

- ◆ Either entire transaction happens, or none of it does
- ◆ If transaction happens, it appears to have happened as a single atomic action (across all resources it touches)

Consistency

- ◆ A transaction results in a valid transformation of the system state; i.e. the results of the transaction do not violate the rules or invariants of the system
- ◆ If they held before, they hold after
- ◆ E.g. total money in bank is not changed by internal transaction
- ◆ Invariants need not be maintained within transaction, only before and after



22

Transaction Properties

Isolation (aka Serializability)

- ◆ If two or more transactions are running concurrently, then to each of them and to all external processes, the final result looks as if all transactions ran sequentially in some (system-dependent) order.
 - i.e. they appear to have run serially, not concurrently
- ◆ From outside the transaction, cannot see intermediate results

Durability (aka Persistence)

- ◆ Once a transaction has successfully committed, its results are permanent, regardless of what failures happen after that
- ◆ No later failure can undo the results or cause them to be lost



23

Transactions

- ◆ Bundle many operations into a transaction
 - One of the first transaction systems was the Sabre American Airline reservation system, developed 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



24

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



25

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



26

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?



27

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!**



28

Two-Phase Locking for Transactions

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



29

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)



30

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



31

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



32

What to Log?

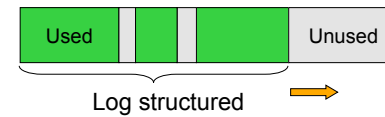
- ◆ Physical blocks (directory blocks and inode blocks)
 - Easy to implement but takes more space
 - Which block image?
 - Before operation: Easy to go backward during recovery
 - After operation: Easy to go forward during recovery.
 - Both: Can go either way.
- ◆ Logical operations
 - Example: Add name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



33

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)?



34

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



35