


# COS 318: Operating Systems

## File Caching and Reliability

Jaswinder Pal Singh and a Fabulous Course Staff  
 Computer Science Department  
 Princeton University


<http://www.cs.princeton.edu/courses/cos318/>



1

## Topics

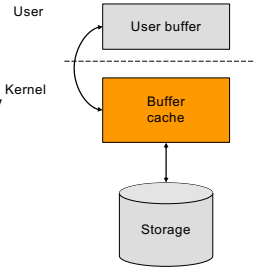

- ◆ Motivating the Problem: File buffer cache
- ◆ Possible Solutions



2

## File Buffer Cache


- ◆ A large cache in kernel
- ◆ Read: check if the block is in
  - Yes: Copy block to user buffer
  - No: Read from storage to buffer cache and copy to user buffer
- ◆ Write: check if the block is in
  - Yes: Update it with user buffer
  - No: Copy block to buffer cache (may replace a block). Write the block.
- ◆ Usual questions
  - What to cache?
  - How to size the cache?
  - What to prefetch?
  - How and what to replace?
  - Which write policies?

3

## What to Cache?

- ◆ For different kinds of blocks
  - i-nodes
  - Indirect blocks
  - Directories
  - Data blocks
- ◆ Issues
  - Are all blocks equal?



4

### Buffer Cache Size

- ◆ Competition
  - Competes with VM and the rest of the system for memory
- ◆ Two approaches
  - Fixed size
  - Variable size
- ◆ How to adjust buffer cache size?
  - Users make decisions
  - Working set idea with dynamic adjustments within thresholds

5

5

### Why in the Kernel?

- ◆ DMA
  - DMA works with "pinned" physical memory
- ◆ Multiple user processes
  - Share the buffer cache
- ◆ Typical replacement strategy
  - Global LRU
  - Working set for each process

6

6

### What to Prefetch?

- ◆ Optimal
  - Prefetch in just enough time to use them
- ◆ Good news: file accesses have locality
  - Temporal locality
  - Spatial locality
- ◆ Common strategies
  - Prefetch next k blocks together
  - Discard unreferenced blocks
  - Layout consecutive blocks to the same cylinder group
  - Fetch directory and i-nodes together
- ◆ Advanced strategy
  - Prefetch all small files of a directory
  - Prefetch beginning portions of large files

7

7

### Write Policies

- ◆ Write through
  - Write to storage immediately
  - Cache is consistent
  - Simple, but cause more I/Os
- ◆ Write back
  - Update a block in buffer cache and mark it as dirty
  - write to storage later
  - Fast writes, absorbs writes, and enables batching
  - So, what's the problem?

9

9

## Write Back Complications

- ◆ Tension
  - On crash, all modified data in cache is lost.
  - Postpone writes  $\Rightarrow$  better performance but more damage
- ◆ 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 options have no guarantees about written data being lost



10

10

## File System Reliability

- What if disk loses power or machine crashes?
  - Some operations in progress may complete
  - Some operations in progress may be lost
  - Overwrite of a block may be only partially complete
- File system wants durability (as a minimum)
  - Data previously stored can be retrieved (maybe after some recovery step), regardless of failure



11

11

## Multiple Updates

- If multiple updates needed to perform some operations, a crash can occur between them
  - Moving a file between directories:
    - Delete file from old directory
    - Add file to new directory
  - Create new file
    - Allocate space on disk for header, data
    - Write new header to disk
    - Add the new file to directory
- What if there is a crash in the middle?
- Problems even with write-through cache



12

12

## Storage Reliability Problem

- Single logical file operation can involve updates to multiple physical disk blocks
  - inode, indirect block, data block, bitmap, ...
- At a physical level, operations complete one at a time
  - But we want higher level concurrent operations for performance
- How do we guarantee consistency regardless of when crash occurs?




13

13

### Approaches

- ◆ Throw everything away and start over
  - Done for most things (e.g., make again)
  - What about your email?
- ◆ Check, and recover what you can when stuff gets corrupted: Reconstruction
  - ◆ Try to fix things after a crash (e.g. "fsck")
  - ◆ Figure out where you are, make file system consistent
- ◆ Try not to let stuff get corrupted
  - ◆ Careful ordering to make consistent updates
  - ◆ Copy on Write
  - ◆ Logging and transactions

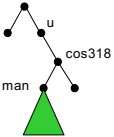



14

14

### Reconstruction: File Recovery Tools

- ◆ Physical backup (dump) and recovery
  - Dump disk block by block 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
  - Unreachable blocks are free
  - Lots of other consistency checks on superblocks, inodes, data blocks etc.

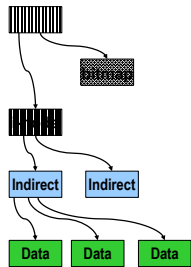




16

16

### 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
- ◆ Free block data structure
  - Search all reachable blocks from the root
  - Unreachable blocks are free





17

17

### Approaches

- ◆ Throw everything away and start over
  - Done for most things (e.g., make again)
  - What about your email?
- ◆ Check, and recover what you can when stuff gets corrupted: Reconstruction
  - ◆ Try to fix things after a crash (e.g. "fsck")
  - ◆ Figure out where you are, make file system consistent
- ◆ Try to not let stuff get corrupted:
  1. Careful ordering to make consistent updates
  2. Copy on Write
  3. Logging and transactions



20

20

### Careful Ordering: 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

21

### 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

22

### 1. 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
- ◆ 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

23

### Careful Ordering in General

- Sequence operations in a specific order
  - Careful design to allow sequence to be interrupted safely
- Post-crash recovery
  - Read data structures to see if there were any operations in progress
  - Clean up/finish as needed
- Approach taken in FAT, FFS (fsck), and many app-level recovery schemes (e.g., Word)

24

### Careful ordering

- Pros
  - Works with minimal support in the disk drive
  - Works for most multi-step operations
- Cons
  - Can require time-consuming recovery after a failure
  - Difficult to reduce every operation to a safely interruptible sequence of writes
  - Difficult to achieve consistency when multiple operations occur concurrently
  - Garbage left around that needs to be collected



30

30

### 2: Copy-on-Write

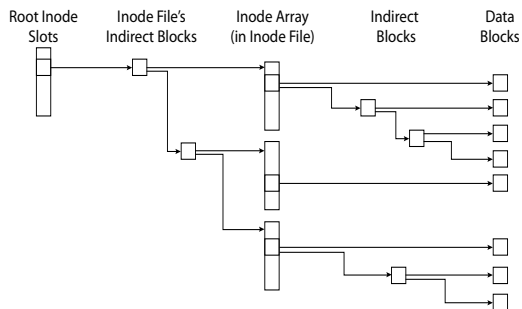
- Never update in place
  - To update file system, write a new version of the blocks/data structures containing the update
  - Reuse existing unchanged disk blocks
- Seems expensive. But:
  - Updates can be batched
  - Almost all disk writes can occur in parallel
- Approach taken in network file server appliances (WAFL, ZFS)



31

31

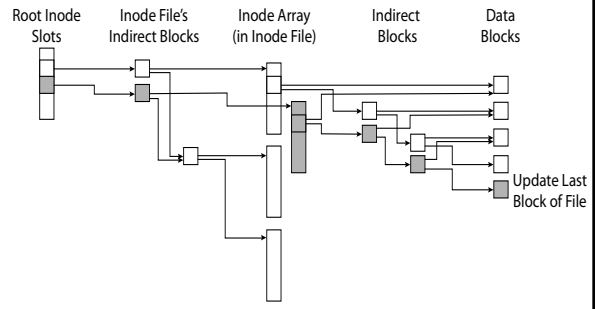
### Copy on Write



32

32

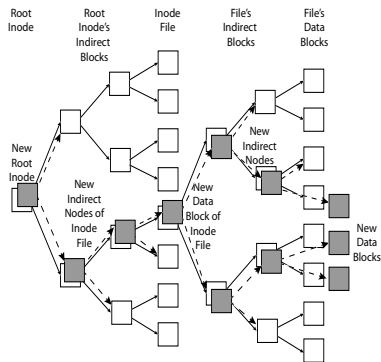
### Copy on Write



33

33

## Copy on write batch update



34

34

## Copy-on-Write Garbage Collection

- For write efficiency, want contiguous sequences of free blocks
    - Spread across all block groups
    - Updates leave dead blocks scattered
  - For read efficiency, want data read together to be in the same block group
    - Write anywhere leaves related data scattered
- => Background coalescing of live/dead blocks



35

35

## Copy-on-Write

- Pros
  - Consistent behavior regardless of failures
  - Fast recovery
  - High throughput (best if updates are batched)
- Cons
  - Potential for high latency
  - Small changes require many writes
  - Garbage collection essential for performance
    - Updates leave dead blocks scattered, but want contiguous free blocks and grouped related data



36

36

## 3: Logging and Transactions

- Instead of modifying data structures on disk directly, write changes to a journal/log
  - Intention list: set of changes we intend to make
  - Log/Journal is **append-only**
- Once changes are on log, safe to apply changes to data structures on disk
  - If there is a crash, recovery can read log to see what changes were intended
- Once changes are copied, safe to remove log



37

37

## Transactions

- ◆ Group multiple operations to have "ACID" property
  - Atomicity
    - Any observed result is as if the atomic set all happened or none happened (no partial operations)
  - Consistency
    - Yields a correct transformation of the state
  - Isolation (Serializability)
    - Transactions appear to happen one after the other, not interleaved
  - Durability (Persistency)
    - Once it happens (is committed), stays happened
- ◆ Question
  - Do critical sections have ACID property?



38

38

## Transactions

- ◆ Bundle operations into a transaction
- ◆ Basic idea: Do operations 'tentatively'. If get to commit, great. Otherwise, roll back operations as if transaction never happened
- ◆ 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



39

39

## Transaction Implementation

- Example: money transfer from account x to account y:
 

```

Begin transaction
  S = S - $100
  C = C + $100
Commit
      
```
- Keep "redo" log on disk of all changes in transaction.
  - A log is like a journal, never erased, record of everything you've done
  - Once both changes are on log, transaction is committed.
  - Then can "write behind" changes to disk --- if crash after commit, replay log to make sure updates get to main disk



40

40

## 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 a "commit," replay the log and clear the log
- ◆ Assumptions
  - Writing to disk is correct (recall error detection and correction)
  - Disk is in a good state before we start



41

41



### An Example: Atomic Money Transfer

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

```

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

C = 110  
S = 700

C = 110  
S = 700

S=700

C=110

Commit

- Steps:
  - Write new value of S to log
  - Write new value of C to log
  - Write commit
  - Write S to disk
  - Write C to disk
  - Clear the log and reclaim space
- Possible crashes
  - After 1
  - After 2
  - After 3 before 4 and 5

42

42

### Transaction implementation (cont'd)

S=700 C=110 commit

- Write new value of S to log
- Write new value of C to log
- Write commit
- Write S to disk
- Write C to disk
- Reclaim space on log

- What if we crash after 1?
  - No commit, nothing on disk, so just ignore changes
- What if we crash after 2? Ditto
- What if we crash after 3 before 4 or 5?
  - Commit written to log, so replay those changes back to disk
- What if we crash while we are writing "commit"?
  - As with concurrency, we need some primitive atomic operation or else can't build anything. (e.g., writing a single sector on disk is atomic)

43

43

### Revisit The Implementation

- BeginTransaction
  - Start using a "write-ahead" log on disk
  - Log all updates
- Commit
  - Write "commit" at the end of the log
  - Single disk write to make transaction durable
  - 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
- Question: What if there is a crash during the recovery?

49

49

### Performance

- Log written sequentially
  - Often kept in flash storage
- Asynchronous write back
  - Any order as long as all changes are logged before commit, and all write backs occur after commit
- Can process multiple transactions
  - Transaction ID in each log entry
  - Transaction completed iff its commit record is in log


50

50

### Transaction Isolation (Serializability)

<p>Process A</p> <p>move file from dir x to y mv x/file y/</p>	<p>Process B</p> <p>grep across x and y grep x/* y/* &gt; log</p>
--	---

What if grep starts after changes are logged, but before commit?




51

51

### Transaction isolation

<p>Process A</p> <p>Lock x, y move file from x to y mv x/file y/ Commit and release x,y</p>	<p>Process B</p> <p>Lock x, y, log grep across x and y grep x/* y/* &gt; log Commit and release x, y, log</p>
---	---

Grep occurs either before or after move




52

52

### Two-Phase Locking for Transactions

- ◆ First phase
  - Acquire all locks (avoids deadlock concerns)
- ◆ Second phase
  - All unlocks happen at commit operation (no individual release operations)
  - Rollback operation: always undo the changes first and then release all locks

Thread B can't see any of A's changes until A commits and releases locks. This provides serializability.




53

53

### Serializability

- With two phase locking and redo logging, transactions appear to occur in a sequential order (serializability)
  - Either: grep then move or move then grep
- Other implementations can also provide serializability
  - Optimistic concurrency control: abort any transaction that would conflict with serializability



54

54

## Use Transactions in File Systems

- ◆ Make a file operation a transaction
  - Create a file
  - Move a file
  - Write a chunk of data
  - ...
- ◆ Make arbitrary number of file operations a transaction
  - Make sure logging is idempotent
  - Recovery by replaying the log
  - Called "logging file system" or "journaling file system"



56

56

## Performance Issue with Logging

- ◆ For every disk write, we now have two disk writes
  - They are on different parts of the disk!
- ◆ Performance tricks
  - Changes made in memory and then logged to disk
  - Merge multiple writes to the log with one write
  - Use NVRAM (Non-Volatile RAM) to keep the log



57

57

## Log Management

- ◆ How big is the log?
- ◆ Observation
  - Log what's needed for crash recovery
- ◆ 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
- ◆ Question
  - If you only log metadata (file descriptors and directories) and not data blocks, are there any problems?



58

58

## 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



59

59