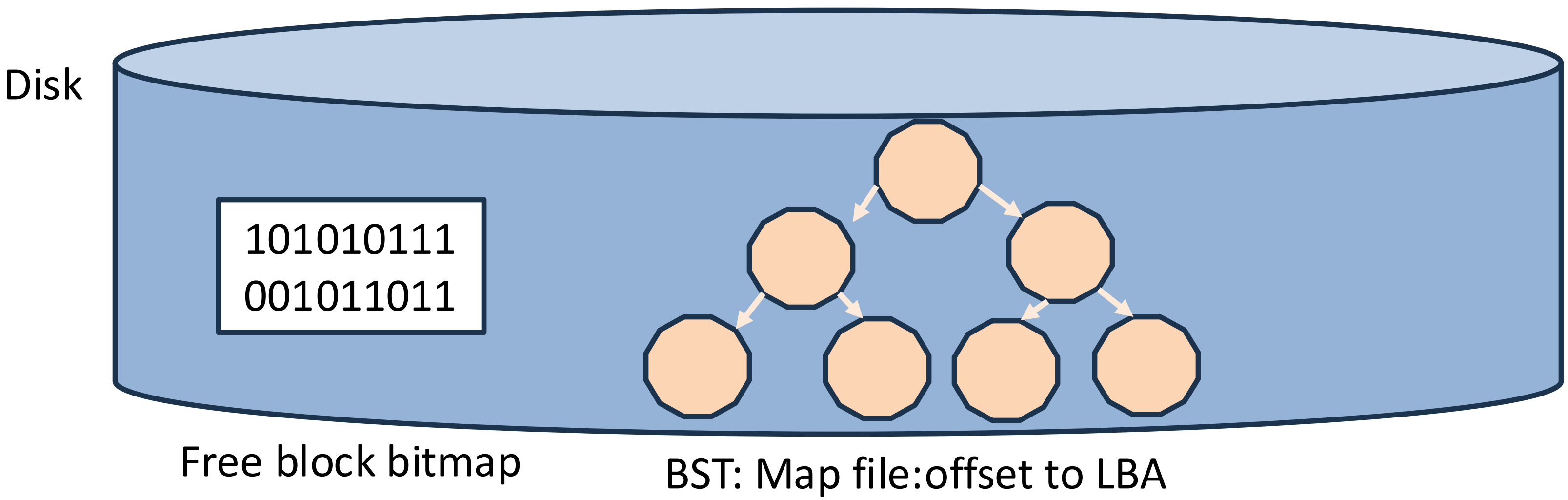
File Systems: Crash Consistency

Don Porter

Portions courtesy Emmett Witchel

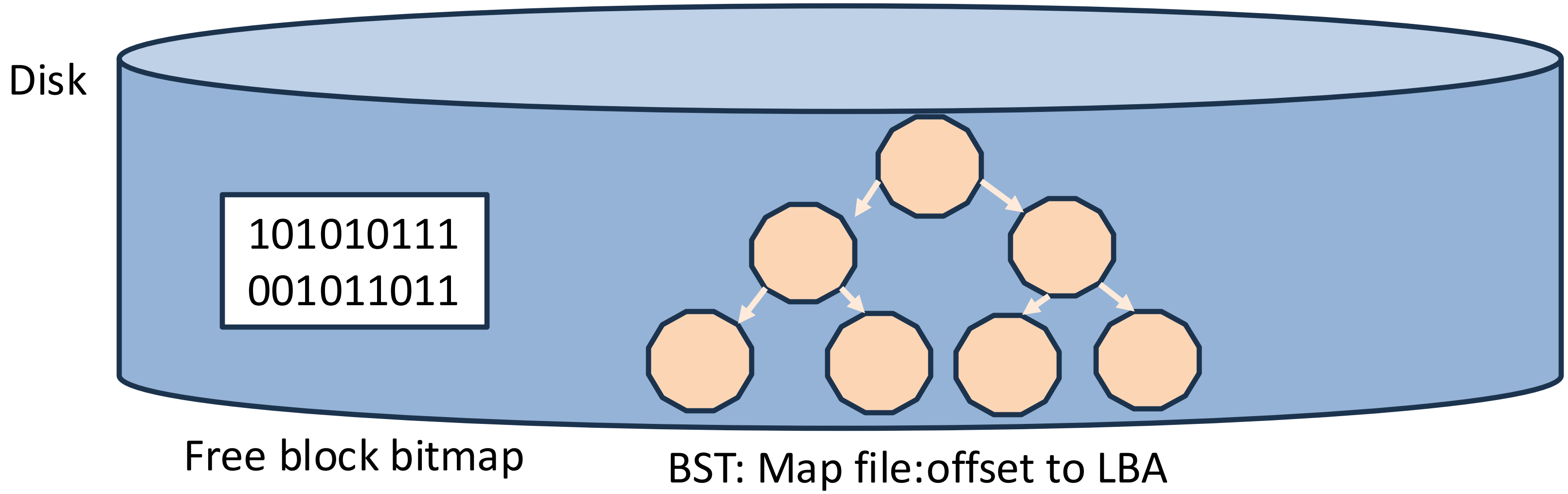
Context (1)

- File systems store metadata on disk
  - Simple example:
    - Bitmap for free space,
    - Binary Search Tree to map <file:offset> to LBA
- File system has invariants:
  - BST: Sorting invariant
  - Every LBA in the BST should be marked ‘0’ in bitmap



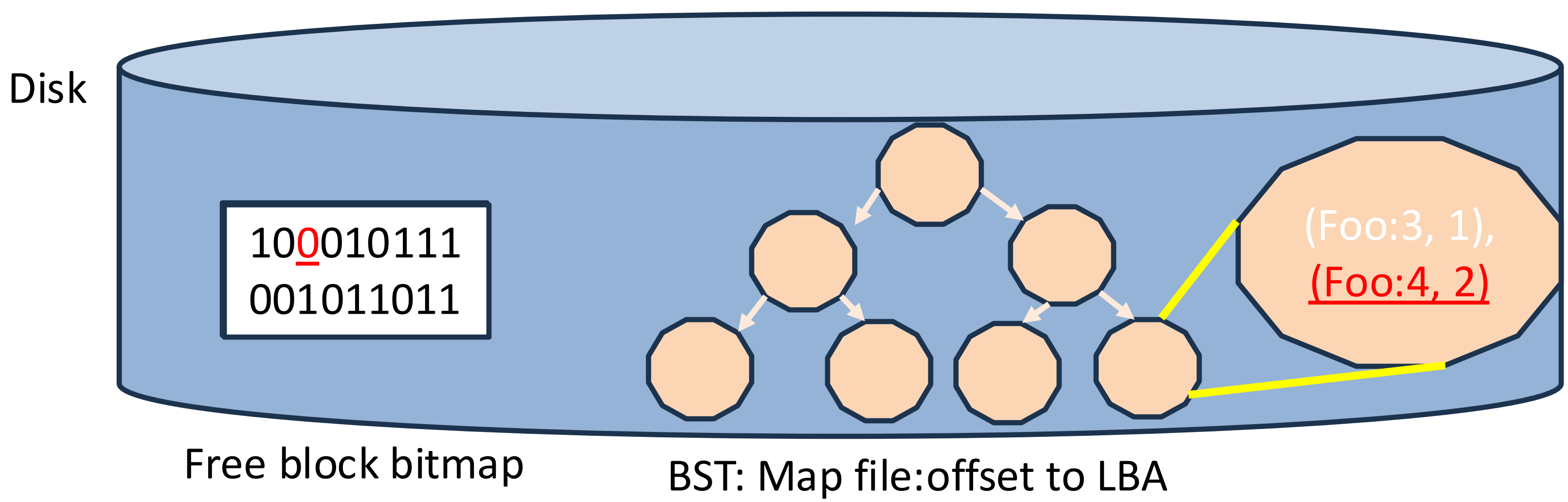
Context (2)

- Recall: Disk writes are atomic (at sector granularity)
- Recall: FS invariants can span multiple sectors
  - E.g., An LBA in the BST must be marked zero in bitmap
- Problem: System can crash between any 2 disk writes
  - After reboot, FS invariants can be violated…



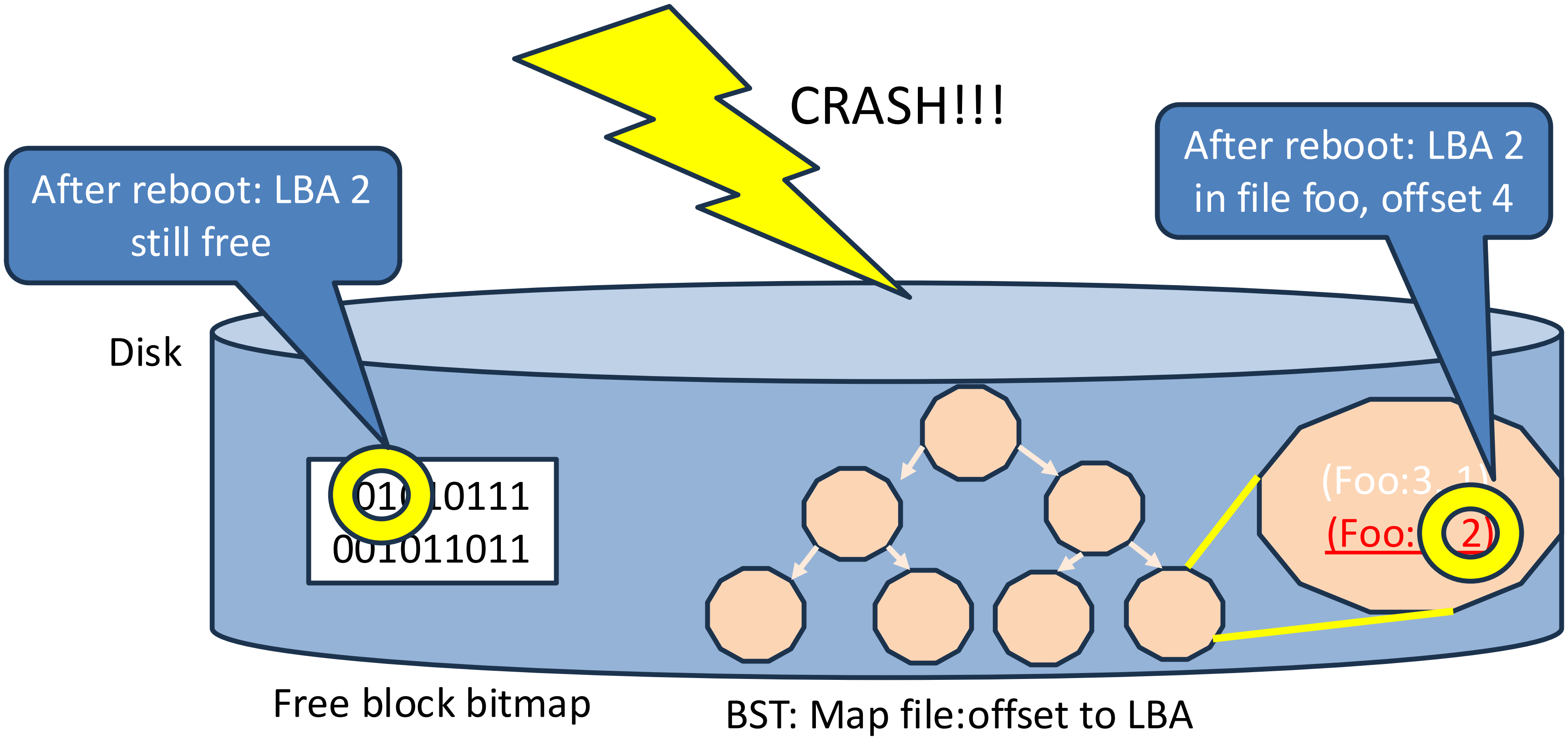
Example: Add block to a file

1. Add entry to BST, mapping Foo:4 to block 2
2. Mark block 2 in use (zero) in bitmap



Example: Add block to a file

1. Add entry to BST, mapping Foo:4 to block 2
2. Mark block 2 in use (zero) in bitmap



Crash Inconsistency

- After a crash, a file system invariant is violated
  - Prev. example: Used block in file marked free
- Worse than just losing the last operation:
  - Can corrupt entire file system
  - Prev. example: LBA 2 can be allocated to a *second* file
    - Writes to one file clobber data in another
      - Long after the crash and reboot!
- Key issue: Metadata updates that span 2+ LBAs
  - Can only write to one LBA atomically

Crash Consistency Strategies

- If updates that span 2+ LBAs cause crash inconsistencies, the solution is…
- …to boil them down (logically) to a single-LBA write

- Three main strategies:
  - Brute-force checks after reboot
  - Copy-on-write
  - Logging/journaling

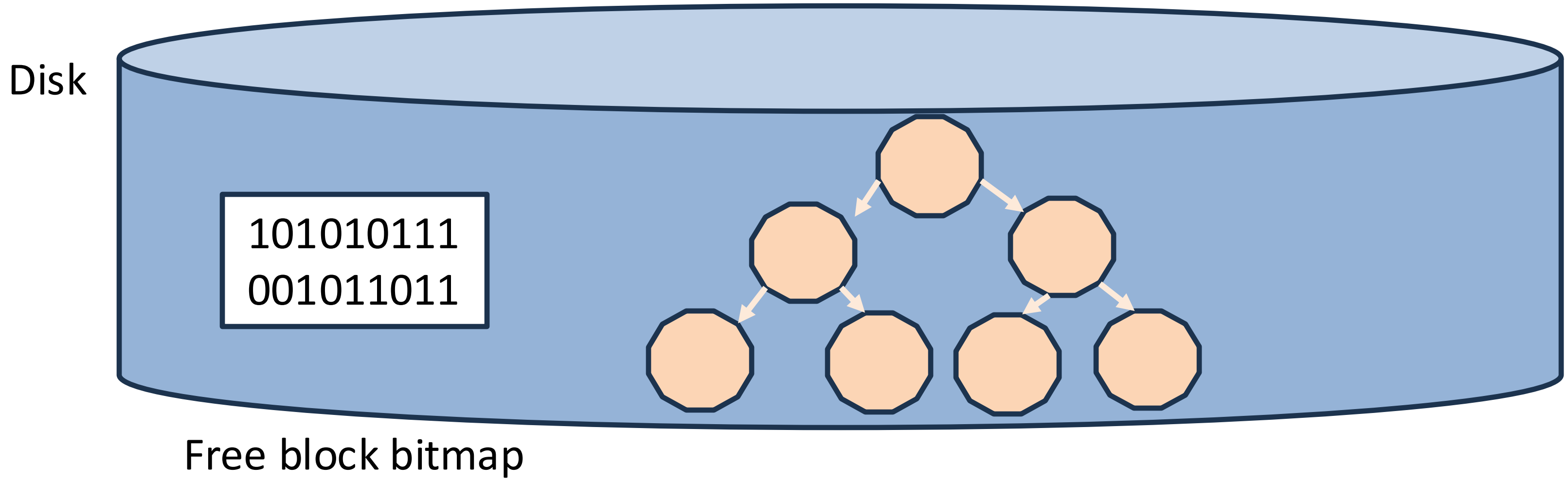
A note on data loss

- If a system can crash, it can lose in-progress writes
  - Like death and taxes, cannot be avoided
- File systems also hold “dirty” data in RAM as an optimization
  - This increases the risk of lost writes
- Strategy: Most kernels bound how long something can stay dirty in RAM – typically 5—30 seconds

- In crash consistency, the goal is not to lose *other* data
  - E.g., not corrupting unrelated data written weeks ago
  - Focus on metadata and data structures, rather than file contents

Strategy 1: Brute-force checks

- Idea: After a reboot, just check every invariant
- Example:
  - Rebuild a free block bitmap from walking BST
  - Compare to what is on disk
    - In use, but unreachable LBAs may have lost data



fsck

- Unix tool for brute-force checking a file system

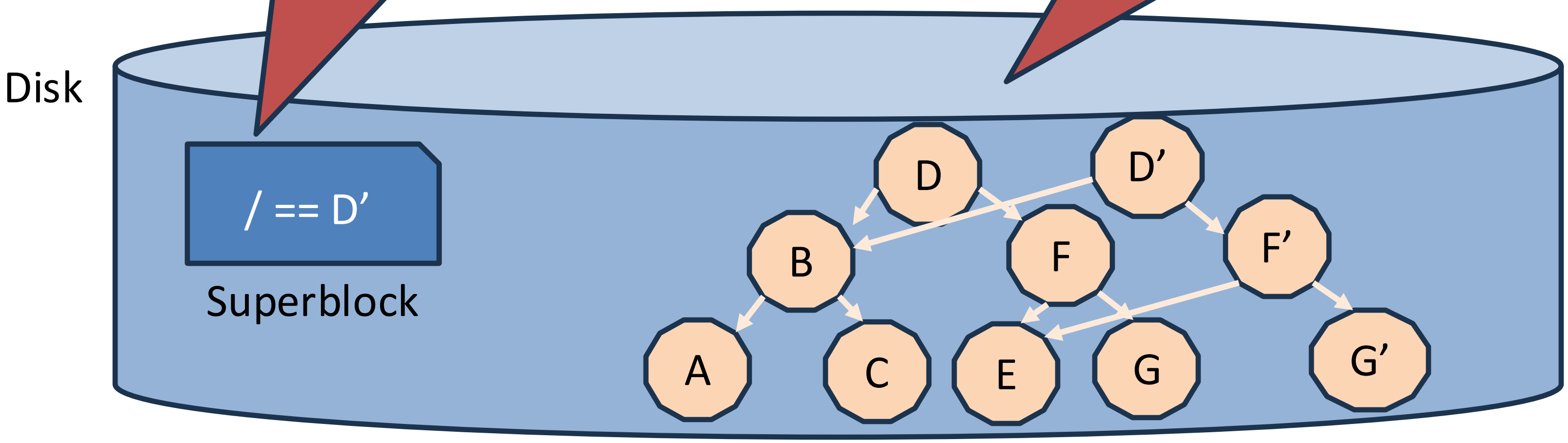
- Downsides:
  - Really, really slow (hours on a modern hard disk)
  - May still be unable to recover lost/corrupted data
    - E.g., What if a block is marked in use in bitmap, but not in tree? What file to put “orphaned” block back into?
  - Requires developers to specify all invariants…

Strategy 2: Copy-on-write (CoW)

- Idea: "Publish" a complex update with a single pointer write
  - Typic … e
- Examp … ify G to G
  - Rew … , publish by

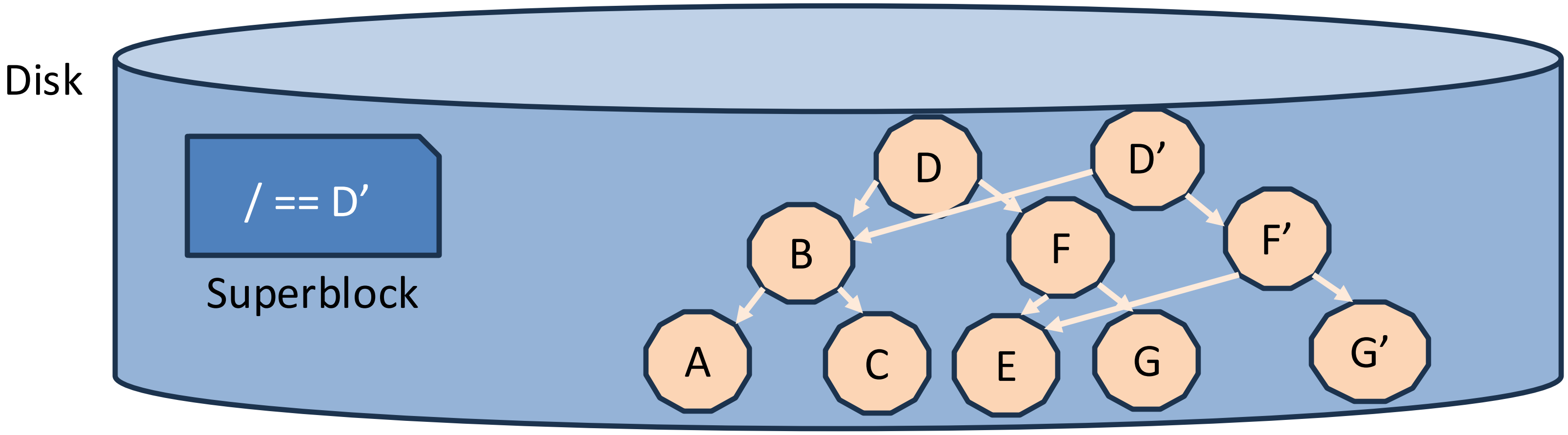
Publish D' with single, atomic LBA write to superblock

Crash up to this point: only lose recent updates (D'); BST (D) still consistent



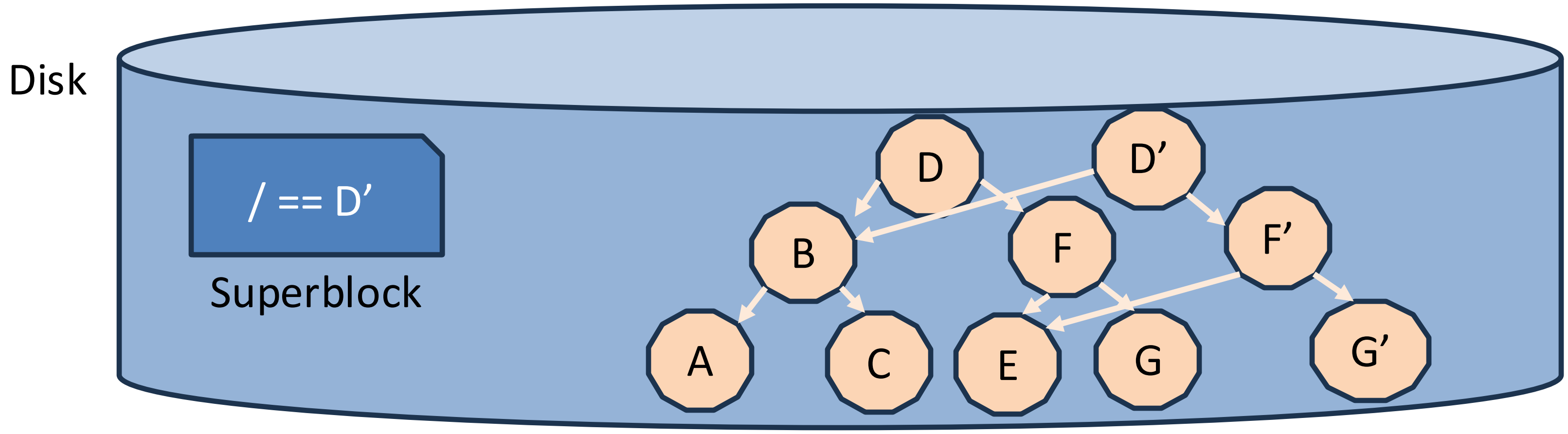
CoW Caveats

- Still need fsck to clean up unpublished copies after a crash
- Also need to garbage collect old versions of data structures
  - E.g., Once D is no longer the root, reclaim D, F, G



CoW Caveats, continued

- Can still lose data between updates to root node
  - Sometimes a new, consistent root is called a **checkpoint**
- How to bound data loss vulnerability (e.g., to 5 s)?
  - Ensure a checkpoint every 5s, ensure all data dirtied more than 5s ago is in the checkpoint



Checkpoints can be expensive

- Unfortunately, it is possible for a checkpoint to get too large to write every 5s
  - Degenerate case: random writes over large file system
    - May dirty and rewrite entire tree
- Motivates our third strategy: logging/journaling

Strategy 3: Logging/journaling

- Idea: reserve a region of disk to act as a circular, ordered log
  - Between checkpoints, record all modifications in the log
  - Next checkpoint logically contains same exact operations in the log; after checkpoint finishes, reset log
- After a crash: replay log against stable checkpoint
- How does a log ensure atomicity/crash consistency?
  - Log for change that spans 2+ LBAs in one, atomic LBA write
  - Log entries written in order
    - After a crash, always a consistent "prefix" of operations in log
- Window for data loss now == the interval between log writes

Logging without CoW

- When used with CoW data structures, log is used to replay recent operations (**redo log**)
- A file system can, instead, update data structures in place
  - Logging still helps!
  - But may need to be more detailed: How to either finish the operation, or how to undo it
    - E.g., Add a new block to a file
      - A crash after writing the allocation bitmap not sufficient to know which block was allocated, in order to finish updating the file mapping
- Lots of edge cases with update-in-place!
  - E.g., unlink (foo); create(foo);

Faster fsck with a journal

- The oldest Unix file systems were update-in-place
  - E.g, ext2
- Ext3 introduced a journal to accelerate reboot/fsck time
  - Just walk the journal instead of a brute-force fsck --- much faster!
  - Does assume data structures are consistent
    - Alas, studies indicate this can be untrue in practice
    - Modern Linux systems still do a brute-force fsck at least once a year on ext3/ext4, just to be safe

Limiting the size of the journal

- Journals and logs have finite space
  - Usually a region of disk, treated as a circular buffer
  - For update-in-place, also record when “in flight” operations complete
- Periodically **checkpoint** the log to skip past completed operations
  - Update log’s “tail pointer” in superblock
    - Indicates where to start reading the log after a reboot
- Allows FS to treat log space as a circular buffer
  - If head of log catches up to the tail, checkpoint and advance tail pointer

Recap: 3 crash consistency strategies

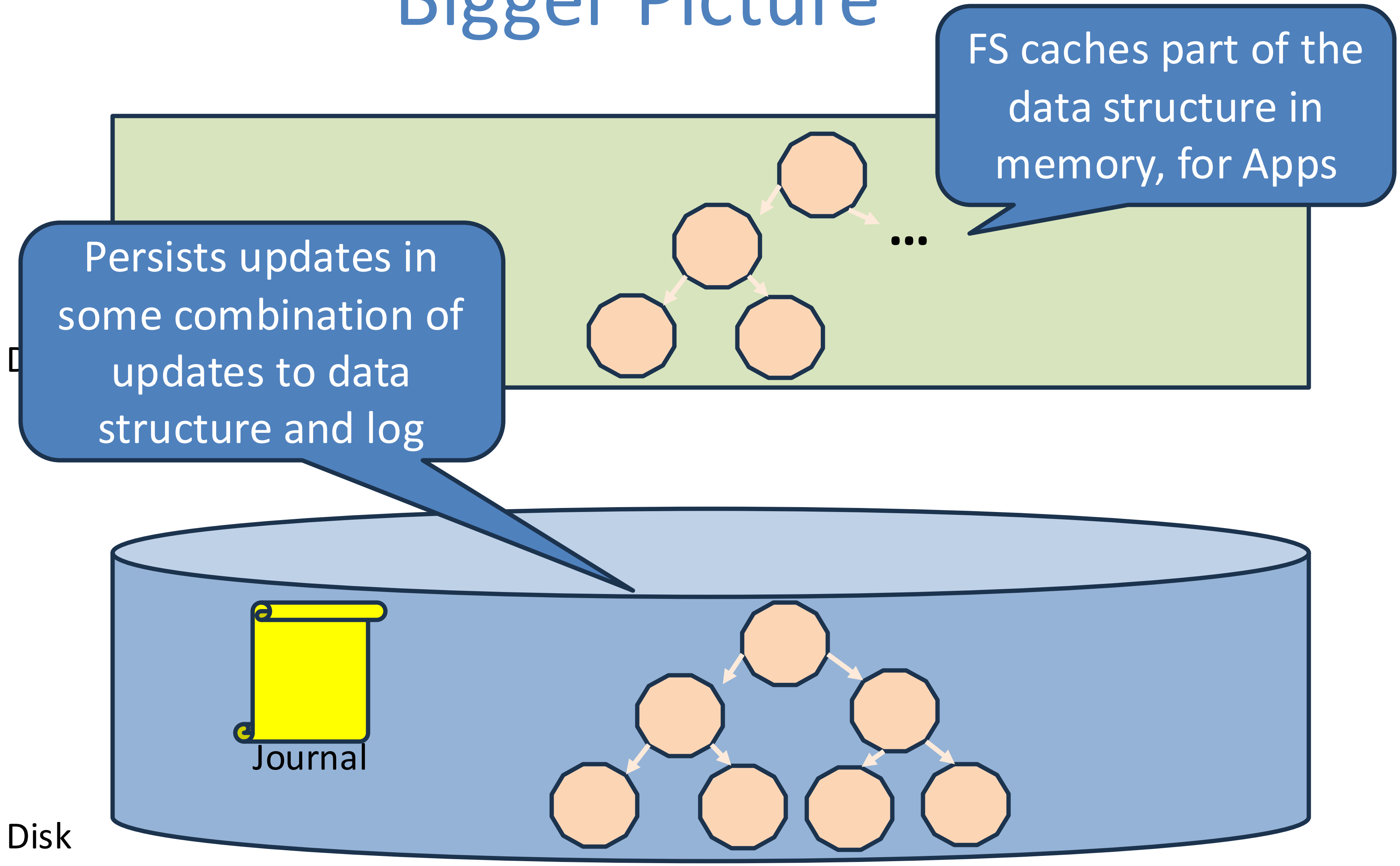
1. fsck: expensive, brute force invariant checks after reboot
2. Copy-on-write: Publish new version of the data structure with one LBA write
   - Data structure always consistent on disk
   - At cost of rewriting unchanged nodes and garbage collection, and possibly longer window to lose recent writes
3. Logging/Journaling: Atomically write log of operations (how to finish or undo them), to recover consistency after reboot

Can use a combination of all 3 strategies

Which is a metadata consistency problem?

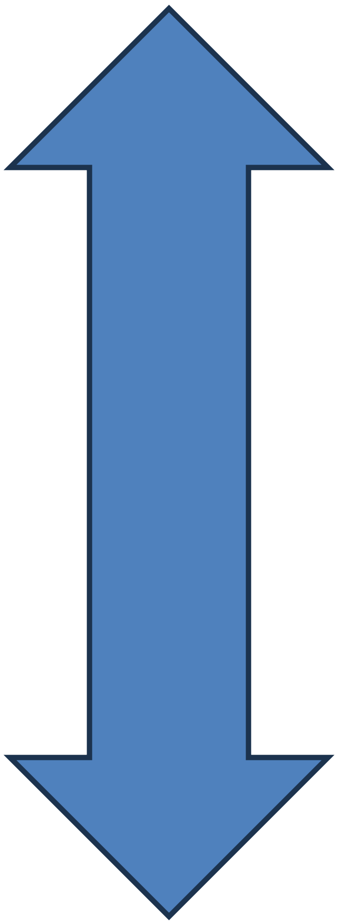
- A. Null double indirect pointer
- B. File created before a crash is missing
- C. Free block bitmap contains a file data block that is pointed to by an inode
- D. Directory contains corrupt file name

Bigger Picture



A logging continuum

No Log on Disk



Only Log on Disk

- No log: All writes sync (slow!); fsck required
- Data structures mostly consistent, short log (faster boot, some sync writes)
- Log-structured FS (e.g., F2FS): Very infrequent data structure checkpoints to bound GC, only sync writes are to flush log blocks
- Nothing but a log: Fast runtime, no sync writes, but really slow boot, difficult GC

What about applications?

- For performance reasons, file systems often provide crash consistency of metadata only
  - I.e., a crash doesn't corrupt the whole FS
- Crash consistency of the file contents left as an exercise to application developer (i.e., you!)

- Alas, data consistency semantics not standard across file systems ☹

Motivating example

- Suppose I have a web application that stores xml for clients
- Requirements:
  - I can fail to save an xml file, but if I tell the client I have saved it, I must return the exact file contents later
  - The client must be able to update the file with new versions. Old versions need not be retained.
  - A file can be larger than 1 FS block
- Crash consistency concerns?

File must survive a crash before sending ack to client

When updating, can't mix blocks of two versions

Two key tools for developers

1. sync(), fsync(), fdatasync()
2. rename()

sync() and friends

- sync(): Write *all* dirty data and metadata *for all files and file systems systems* to disk.
- fsync(fd): Write the inode and data blocks (if dirty) to disk for file handle fd
- fdatasync(fd): Write any dirty data blocks for fd to disk, but let the inode stay dirty in memory if possible
  - If the file size or block mapping changes, inode will be written
  - But may delay things like updating last modification time

Where should we put fsync?

```
int fd = open("foo.xml",
O_CREAT|O_WRONLY, 0700);
write(fd, buffer, length);
close(fd);
dirfd = open(".",   O_DIRECTORY|
                    O_RDONLY);
fsync(dirfd);
```

fsync(fd); // Ensures foo.xml data blocks written to disk

Ensures directory updates written to disk

Directory contents also "data blocks"

What about updates?

- If an xml file is larger than one block, no way to make a multi-block write() atomic
  - Can end up with half of two xml files
- How to work around this?

- Create (and fsync) a new, temporary file
- Then rename() the temp file over the old version
  - Leverages atomicity of rename() call
- And fsync() the parent directory!

Common FS Consistency Properties

- Metadata-only Journaling: Only ensure crash consistency of changes to metadata
- Ordered mode: Metadata-only mode, with a twist:
  - Data blocks always written to disk before inode goes into journal
- Full data mode: Crash consistency of data and metadata

Conclusion

- Understand key issue of crash consistency: invariants that span multiple LBAs
- Three key techniques for crash consistency in FS:
  - Fsck, copy-on-write, journaling
- Logging creates opportunity to trade reboot time for fewer sync writes
- Two key tools for crash consistency in application:
  - Sync and rename