File Systems:

Consistency Issues

File Systems: Consistency Issues

- File systems maintain many data structures
 - > Free list/bit vector > Directories
 - File headers and inode structures
 - Data blocks
- All data structures are cached for better performance
- > Works great for read operations
- > ... but what about writes?
- If modified data is in cache, and the system crashes → all modified data can be lost
- If data is written in wrong order, data structure invariants might be violated (this is very bad, as data or file system might not be consistent)
- Solutions:
 - $\diamond\,$ Write-through caches: Write changes synchronously \rightarrow consistency at the expense of poor performance
 - $\diamond\,$ Write-back caches: Delayed writes \rightarrow higher performance but the risk of losing data

What about Multiple Updates?

Several file system operations update multiple data structures

Examples:

- > Move a file between directories
- * Delete file from old directory * Add file to new directory
- Create a new file
 - Allocate space on disk for file header and data
- Write new header to disk
- Add new file to a directory
- What if the system crashes in the middle? > Even with write-through, we have a problem!!
- The consistency problem: The state of memory+disk might not be the same as just disk. Worse, just disk (without memory) might be inconsistent.

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

Consistency: Unix Approach

- Meta-data consistency
- > Synchronous write-through for meta-data
- > Multiple updates are performed in a specific order
- > When crash occurs:
- * Run "fsck" to scan entire disk for consistency
- * Check for "in progress" operations and fix up problems
- allocated but not reflected in the bit map \rightarrow update bit map
- Issues:
 - Poor performance (due to synchronous writes)

* Slow recovery from crashes

Consistency: Unix Approach (Cont'd.)

- Data consistency
 - > Asynchronous write-back for user data Write-back forced after fixed time intervals (e.g., 30 sec.) * Can lose data written within time interval
 - > Maintain new version of data in temporary files; replace older
- version only when user commits
- What if we want multiple file operations to occur as a unit?
- ➤ Example: Transfer money from one account to another → need to update two account files as a unit
- > Solution: Transactions

Transactions

- Group actions together such that they are
 > Atomic: either happens or does not
 - > Consistent: maintain system invariants
 - Isolated (or serializable): transactions appear to happen one after another. Don't see another tx in progress.
 - Durable: once completed, effects are persistent
 - > Durable: once completed, encets are persistent
- Critical sections are atomic, consistent and isolated, but not durable

Two more concepts:

- Commit: when transaction is completed
- > Rollback: recover from an uncommitted transaction



Transactions in File Systems

- Write-ahead logging → journaling file system
 Write all file system changes (e.g., update directory, allocate blocks, etc.) in a transaction log
 "Create file", "Delete file", "Move file" --- are transactions
- · Eliminates the need to "fsck" after a crash
- In the event of a crash
- Read log
- If log is not committed, ignore the log
 If log is committed, apply all changes to disk
- Advantages:
- Reliability
- Group commit for write-back, also written as log
- Disadvantage:
- > All data is written twice!! (often, only log meta-data)

Where on the disk would you put the journal for a journaling file system?

- 1. Anywhere
- 2. Outer rim
- 3. Inner rim
- 4. Middle
- 5. Wherever the inodes are

Transactions in File Systems: A more complete way

- Log-structured file systems
 Write data only once by having the log be the only copy of data and meta-data on disk
- Challenge:
 - > How do we find data and meta-data in log?
 - Data blocks → no problem due to index blocks
 Meta-data blocks → need to maintain an index of meta-data blocks also! This should fit in memory.
- Benefits:
 - All writes are sequential; improvement in write performance is important (why?)
- Disadvantage:
- Requires garbage collection from logs (segment cleaning)

File System: Putting it All Together

- Kernel data structures: file open table
 - $\succ\,$ Open("path") \rightarrow put a pointer to the file in FD table; return index
 - ➢ Close(fd) → drop the entry from the FD table
 - \succ Read(fd, buffer, length) and Write(fd, buffer, length) \rightarrow refer to the open files using the file descriptor
- What do you need to support read/write?
 - > Inode number (i.e., a pointer to the file header)
 - Per-open-file data (e.g., file position, ...)

Diagd

