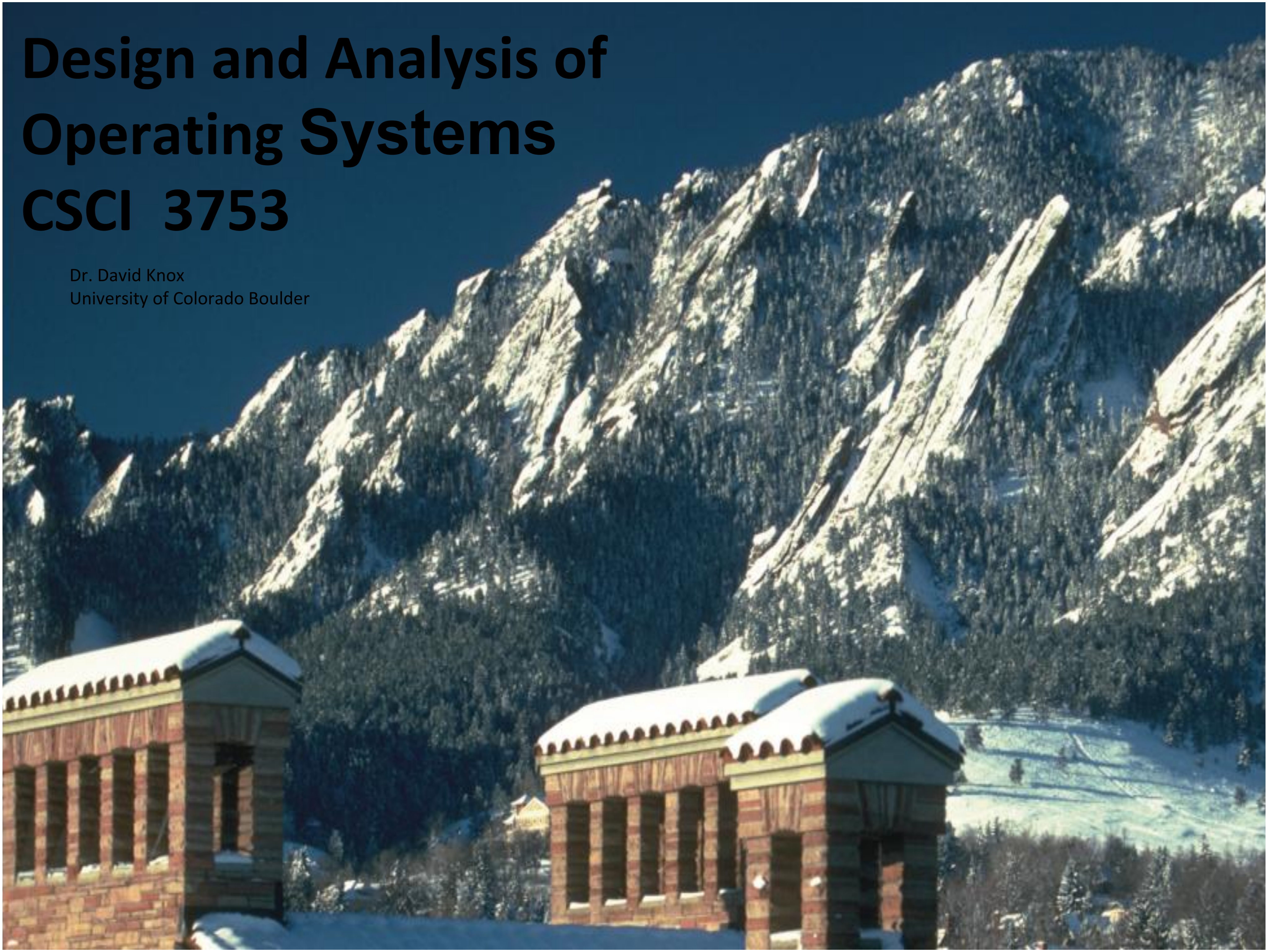
Design and Analysis of Operating Systems CSCI 3753

Dr. David Knox
University of Colorado Boulder



Design and Analysis of Operating Systems CSCI 3753

File System Performance, Reliability, and Fault Recovery

Dr. David Knox
University of Colorado Boulder

File System Performance, Reliability, and Fault Recovery

File System Performance

- **Approaches to improve performance in a file system:**
  - In memory:
    - file header: caching FCB information about open files in memory improves performance (faster access)
    - directory:
      - caching directory entries in memory improves access speed.
      - And hash the directory tree to quickly find an entry and see if it's in memory.

File System Performance

- On disk:
  - file data: indexed allocation is generally faster than traversing linked list allocation
  - free block list: counting, grouped, linked list allows fast allocation of large # of files

File System Performance

- **Other potential optimizations:**
  - the disk controller can also have its own cache that stores file data/FCBs/etc. for fast access
  - Cache file data in memory
  - Smarter layout on disk: keep an inode/FCB near file data to reduce disk seeks, and/or file data blocks near each other
  - *read ahead*:
    - if the OS knows this is sequential access, then read the requested page and several subsequent pages into main memory cache in anticipation of future reads

File System Performance

- **Some other potential optimizations:**
  - *asynchronous writes*: delay writing of file data until sometime later.
  - Advantages:
    - removes disk I/O wait time from the critical path of execution,
      – e.g. a write(X) to a file can return quickly rather than waiting for completion of disk I/O
      – allowing the program to move forward in its execution
    - This allows a disk to schedule writes efficiently
      – grouping nearby writes together
    - May avoid a disk write if the data has been changed again soon
    - Note: in certain cases, you may prefer to enforce synchronous writes
      – e.g. when modifying file metadata in the FCB on an open() call

Memory-Mapped Files

- **Map some parts of a file on disk to pages of virtual memory**
  - Use mmap() call
  - First read/write to file data not in memory will on-demand page in the desired blocks of file into main memory pages
  - Subsequent reads/writes to that file data are served quickly by memory
  - Later, flush changes in file to disk
  - Uses virtual memory system
    - fast compared to read()/write() system calls to change file data contents

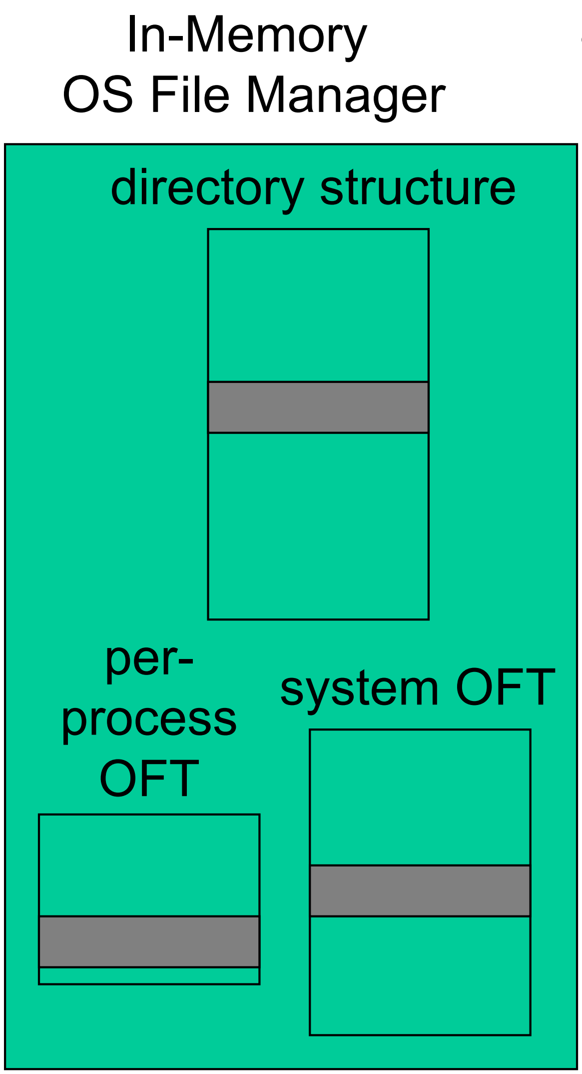
File System Reliability and Fault Recovery

File System Reliability & Fault Recovery

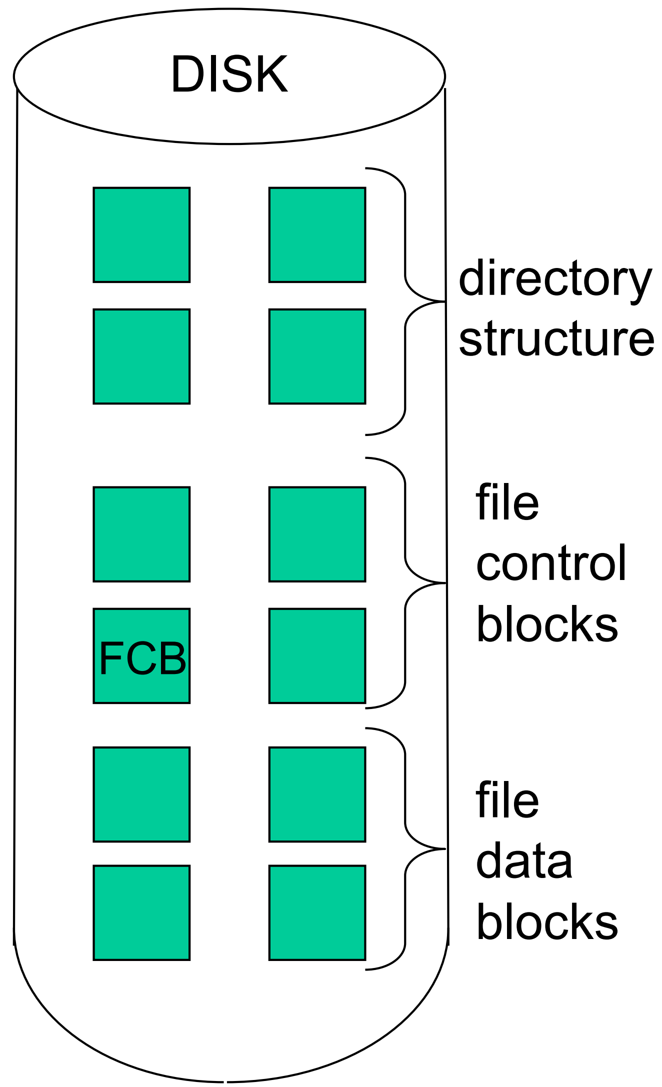
- **In general, OS should gracefully recover from hardware or software failure**
  - The file system needs to be engineered to ensure reliability/fault recovery

- **Problem: File system is quite fragile to system crashes**
  - There is a portion of the file system that is cached in memory
  - This portion may be inconsistent with the complete file system stored on disk

File System Fragility



- All in-memory file system data are lost on a power loss:
  - Directories, file metadata, and file data may all be cached in memory
  - They may all be modified
  - These modifications are lost if they weren't saved to disk



File System Reliability & Fault Recovery

- Problem #1: asynchronous writes produce inconsistency between in-memory and on-disk file system
  - **Promised writes of filed data that were delayed by asynchrony may be lost**
  - **Asynchronous writes of directory metadata can create inconsistency between the file system on disk and the writes cached in RAM**
    - Directory information in RAM can be more up to date than disk
    - cached writes may be lost if there is a system failure
      - such as a power loss
      - in this case, the promised writes will not be executed

File System Reliability & Fault Recovery

- **To address asynchronous write inconsistency,**
  - UNIX caches directory entries for reads
  - But UNIX does not cache any data write that changes metadata or free space allocation

    These changes to critical metadata are written synchronously (immediately) to disk, before the data blocks are written

- **Problem #2: Even if all writes are synchronous, there is still a consistency problem:**
  - any of the individual synchronous/asynchronous writes to disk can fail halfway through the operation, leaving a half-written directory entry, FCB, or file data block.

File System Reliability & Fault Recovery

- **Problem #3: Complex operations can create inconsistency while waiting for them to complete**
  - e.g. a file create() involves many operations, and may be interrupted at any time in mid-execution
  - file create() updates the directory, FCB, file data blocks, and free space management
  - if there is a failure after creating the FCB, then the file system is in an inconsistent state because the file data has not yet been saved on disk,
    - i.e. the directory says there is a file and points to the FCB, but the FCB is incomplete because its index block hasn't been fully allocated

Reliability/Fault Recovery Solutions

- **Approach: file systems can run a consistency checker like fsck in UNIX or chkdsk in MSDOS**
  - in linked allocation, would check each linked list and all FCB's to see if they are consistent with the directory structure.
    - similar checks for indexed allocation
  - Check each allocated file data block to see that its checksum is valid
- **Disadvantages:**
  - This is heavyweight, and takes a long time to check the entire file system.
  - This can detect an error, but doesn't ensure recovery or correction

Reliability/Fault Recovery Solutions

- **Approach: *log-based recovery* is a solution that helps you *recover* from file system failures:**
  - OS maintains a log or journal on disk of each operation on the file system
  - called log-based or journaling file systems

- **The log on disk is consulted after a failure to reconstruct the file system**
  - In a journaling file system, the log is seen as a separate entity from the file data.
  - In a log-structured system, the log *is* the file system, and there are no separate structures for storing file data and metadata – it's all in the log.

Log-Based Recovery

Log-Based Recovery

- **Each operation on the file system is written as a record to the log on disk *before* the operation is actually performed on data on disk**
  - this is called *write-ahead logging*
- **The file system has a sequence of records of operations in the log about what was intended in case of a crash**
  - The log contains a sequence of statements like “I’m about to write this directory entry/file header/file data block”,
  - and “I just finished writing this directory/FH/data”.

Log-Based Recovery

- **Operations are grouped in sets called *transactions***
  - e.g. a file create() has many steps.
    - You either want the entire file created, or not at all if it fails at any step along the way
  - Need the set of steps as a single logical unit
    - It is performed in its entirety or not at all
- **Transactions are viewed as atomic**
  - either succeed in their entirety or not at all

Log-Based Recovery

- **A transaction T_i looks like the following:**
  - begins with < T_i starts>
  - followed by a sequence of records like write(X), read(Y), ... needed to complete the transaction, e.g. a file create()
  - ends with < T_i commits>

- **Write each of these operations to the log**

Log-Based Recovery

← older LOG newer →

. . .	T1 <start>	T1 write(X, old, new)	T1 write(Y, old, new)	T1 <commit>	T2 <start>	T2 write(Z, old, new)	T3 <start>	T3 write(Q, old, new)	T2 <commit>	. . .

- **Each log record of an operation within a transaction consists of:**
  - transaction name T_i
  - data item name, e.g. X
  - old value
  - new value

- **Both the old and new values must be saved in order for the system to recover from crashes in mid-transaction**

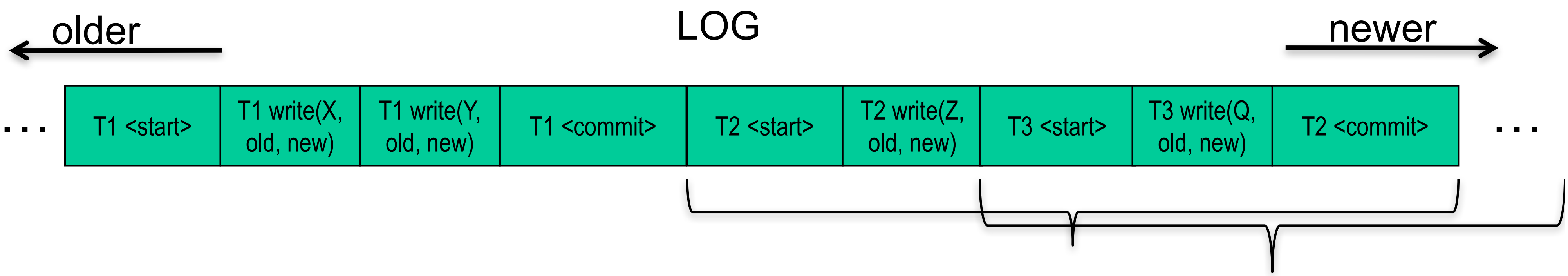
Log-Based Recovery

older ← LOG → newer

. . .	T1 <start>	T1 write(X, old, new)	T1 write(Y, old, new)	T1 <commit>	T2 <start>	T2 write(Z, old, new)	T3 <start>	T3 write(Q, old, new)	T2 <commit>	. . .

- **A transaction is not considered complete until it is committed in the log**
  - once the <commit> appears in the log, then even if the system crashes after this point, there is enough information in the log to fully execute the transaction upon recovery
  - therefore, once the <commit> appears in the log, it is OK to return from the system call that called file create() or file write()

Log-Based Recovery



- **Operations in different transactions can overlap in the log**

- **For asynchronous writes, the actual write(X) to disk may occur much later than the entry written to the log**

Log-Based Recovery

older LOG newer

T1 <start>	T1 write(X, old, new)	T1 write(Y, old, new)	T1 <commit>	<checkpoint>	T2 <start>	T2 write(X, old, new)	T2 <commit>	T3 <start>	T3 write(Z, old, new)

Completed transactions are all full transactions prior to a *checkpoint*. All such transactions have been both:

1. committed to the log *and*
2. have been executed by the file system

Committed transactions are those that

1. have been committed to log but
2. have not yet been executed by the file system (since this is a write-ahead log),

- They are candidates for *redo*() after a crash

Partial transactions are candidates for *undo*() after a crash

Log-Based Recovery

older LOG newer

T1 <start>	T1 write(X, old, new)	T1 write(Y, old, new)	T1 <commit>	**<checkpoint>**	T2 <start>	T2 write(X, old, new)	T2 <commit>	T3 <start>	T3 write(Z, old, new)

completed transactions — *committed* transactions — Partial transactions

- The checkpoint indicates that all full transactions (those with a <start> and <commit>) prior to the checkpoint (to the left) have been written to *both* the disk *and* log

- Committed transactions are full transactions in the log that are to the right of the most recent checkpoint, and thus have not been written to disk yet

Log-Based Recovery

older LOG newer

T1 <start>	T1 write(X, old, new)	T1 write(Y, old, new)	T1 <commit>	**<checkpoint>**	T2 <start>	T2 write(X, old, new)	T2 <commit>	T3 <start>	T3 write(Z, old, new)

completed transactions *committed* transactions Partial transactions

- In normal operation, the file system will
  - periodically *replay* committed transactions in the log onto disk,
  - Then add a new checkpoint to the log,
  - thus all committed transactions to the left of the newly added checkpoint are converted into completed transactions
  - completed transactions can be removed from the log, or just written over if it's a circular log

Log-Based Recovery

- **On a failure, the OS looks for the latest checkpoint in the log, and redo()'s committed transactions and undo()'s partial transactions from that point on**
  - redo() transaction T_i if the log contains both < T_i starts> and < T_i commits> and these transactions appear after a checkpoint
  - undo() transaction T_k if the log contains < T_k starts> but not < T_k commits>
    - this is called an aborted transaction
    - during recovery, such a transaction is rolled back to its former state

Log-Based Recovery

- **Note on an undo(),**
  - the log records contain both the old and new data,
  - so there is enough information in the log to restore or roll back a file or object to its old state if the transaction was not completed

- **OS can cleanly recover the file system**
  - from a checkpoint, reapply changes

- **There is a Write-Twice Penalty with Log-Based**
  - Use log-structured file systems to avoid this penalty
    - The journal is the file system
    - There are no separate data structures
    - The journal contains enough information to be entire file system
  - Flash file systems like JFFS, JFFS2, YAFFS use log-structured file systems

Journaling File Systems

- **Some file systems like NTFS only write changes to the metadata of a file system to the log**
  - file headers and directory entries only
  - NOT any changes to file data
  - The journal is separate from the main file system
- **Copy-on-write file systems (such as ZFS and Btrfs)**
  - avoid in-place changes to file data by writing out the data in newly allocated blocks
  - followed by updated metadata that would point to the new data and disown the old
  - followed by metadata pointing to updated metadata repeatedly to the root of the file system hierarchy
  - has the same correctness-preserving properties as a journal, without the write-twice overhead.
  - add metadata (checksums) to insure data integrity

Journaling File Systems

- **Linux's ext4fs can be parameterized to operate in 3 modes:**
  1. *Journal mode*: both metadata and file data are logged. This is the safest mode, but there is the latency cost of two disk writes for every write.
  2. *Ordered mode:* only metadata is logged, not file data, and it's guaranteed that file contents are written to disk before associated metadata is marked as committed in the journal.
     - This way, you don't have say a file header pointing to a new block of data, yet that data has not yet been written to disk. This is the default on many Linux distributions.
  3. *Writeback mode:* only metadata is logged, not file data, and no guarantee file data written before metadata, so files can become corrupted.
     - This is riskiest mode/least reliable but fastest.

Department of Computer Science
UNIVERSITY OF COLORADO **BOULDER**

Design and Analysis of Operating Systems CSCI 3753

Dr. David Knox

University of Colorado Boulder