6.824 2006 Lecture 7: Logging

```
What's the overall topic?
  Atomic updates of complex data w.r.t. failures.
  Today just a single system, we'll be seeing distributed versions
later.

Why aren't synchronous meta-data updates enough?
  (from last lecture on file system crash recovery)
  They're slow
  Recovery may require scanning the whole disk
  Some operations don't have an obvious single committing write

Example: FFS rename
  editor could use re-name from temp file for careful update
  echo a > d1/f1
  echo b > d2/f2
  mv d2/f2 d1/f1
  need to update two directories, stored in two blocks on disk.
  remove then add? add then remove?
    probably want add then remove
  what if a crash?
  what does fsck do?
    it knows something is wrong, since link count is 1, but two links.
    can't roll back -- which one to delete?
    has to just increase the link count.
    this is *not* a legal result of rename!
    but at least we haven't lost the file.
  so FFS is slow *and* it doesn't get semantics right.

You can push tree update one step farther.
  Prepare a new copy of the entire affected sub-tree.
  Replace old subtree in one final write.
  Very expensive if done in the obvious way.
  But you can share structure between old and new tree.
  Only need new storage between change points and sub-tree root.
  (NetApp WAFL does this and more.)
  This approach only works for tree data structures.
    and doesn't support concurrent operations very well

What are the reasons to use logging?
  atomic commit of compound operations. w.r.t. crashes.
  fast recovery (unlike fsck).
  well-defined post-recovery state: serial prefix of operations.
    as if synchronous and crash had occured a bit earlier
  can be applied to almost any existing data structure
    e.g. database tables, free lists
  representation is compact on a disk, so very fast to append
  useful to coordinate updates to distributed data structures
    let's all do this operation
    oops, someone didn't say "yes"
    how to back out or complete?

Transactions
  The main point of a log is to make complex operations atomic.
    I.e. operations that involve many individual writes.
    You want all writes or none, even if a crash in the middle.
```

```
  A "transaction" is a multi-write operation that should be atomic.
  The logging system needs to know which sets of writes form a
transaction.
  re-organize code to mark start/end of group of atomic operations
  create()
    begin_transaction
      update free list
      update i-node
      update directory entry
    end_transaction
  app sends writes to the logging system
  there may be multiple concurrent transactions
    e.g. if two processes are making system calls

Terminology
  in-memory data cache
  on-disk data
  in-memory log
  on-disk log
  dirty vs clean
  sync write vs async

naive re-do log
  keep a "log" of updates
    B TID   [begin]
    W TID B# new-data   [write]
    E TID   [end == commit]
  Example:
    B T1
    W T1 B1 25
    E T1
    B T2
    W T2 B1 30
    B T3
    W T3 B2 99
    W T3 B3 50
    E T3
  for now, log lives on its own infinite disk
  note we include record from uncommitted xactions in the log
  records from concurrent xactions may be inter-mingled
  we can write dirty in-memory data blocks to disk any time we want
  recovery
    1. discard all on-disk data
    2. scan whole log and remember all Committed TIDs
    3. scan whole log, ignore non-committed TIDs, replay the writes
  why can't we use any of on-disk data's contents during recovery?
    don't know if a block is from an uncommitted xaction
    i.e. was written to disk before commit
  the *real* data is in the log!
    the on-disk data structure is just a cache for speed
    since it's hard to *find* things in a log
  so what have we achieved?
    atomic update of complex data structures: gets rename() right
    recoverable
    operations are fast
  problems:
    we have to store the whole log forever
```

```
    recovery has to replay from the beginning of time

re-do with checkpoint
  most logs work like this, e.g. FSD
  allows much faster recovery: can use on-disk data
  write-ahead rule
    delay flushing dirty blocks from in-memory data cache
    until corresponding commit record is on disk
  so keep updates of uncommitted xactions in in-memory data cache (not
disk)
  so no un-committed data on disk.
    but disk may be missing some committed data
    recovery needs to replay committed data from the log
  how can we avoid re-playing the whole log on recovery?
    recovery needs to know a point in log at which it can start
    a "checkpoint", pointer into log, stored on disk
    how to ensure recovery can ignore everything before the checkpoint?
  checkpoint rule:
    all data writes before check point must be stable on disk
    checkpoint may not advance beyond first uncommitted Begin
  in background, flush a bunch of early writes, update checkpoint ptr
  three log regions:
    data guaranteed on disk
    (checkpoint)
    data might be on disk
    (log write point)
    data cannot be on disk
    (end of in-memory log)
  on recovery, re-play commited updates from checkpoint onward
  it's ok if we flush but crash before updating checkpoint pointer
    we will re-write exactly the same data during recovery
  can free log space before checkpoint!

problem:
  uncommitted transactions use space in in-memory data cache
  a problem for long-running transactions
  (not a problem for file systems)

un-do/re-do with checkpoint
  suppose we want to write uncommitted data to disk?
    need to be able to un-do them in recovery
    so include old value in each log record
    W TID B# old-data new-data
  now we can write data from in-memory data cache to disk
    after log entry is on disk
    no need to wait for the End to be on disk
    so we can free in-memory data cache blocks of uncommitted
transactions
  recovery:
    for each block mentioned in the log
    find the last xaction that wrote that block
    if committed: re-do
    if not committed: un-do
  two pointers stored on disk: checkpoint and tail
  checkpoint:
    all in-memory data cache entries flushed up to this point
    no need to re-do before this point
```

```
    but may need to un-do before this point
  tail:
    start of first uncommitted transaction
    no need to un-do before this point
    so can free before this point
  it's ok if we crash just before updating the tail pointer itself
    we would have advanced it over committed transaction(s)
    so we will re-do them, no problem
  what if there's an un-do record for block never written to disk?
    it's ok: un-do will re-write same value that's already there
  what if
    B T1
    W T1 B1 old=10 new=20
    B T2
    W T2 B1 old=20 new=30
    crash
    The right answer is B1 = 10, since neither committed
    But it looks like we'll un-do to 20
    What went wrong? How to fix it?

careful disk writing
  log usually stored in a dedicated known area of the disk
    so it's easy to find after a reboot
  where's the start?
    checkpoint, a pointer in a known disk sector
  where's the end?
    hard if crash interrupted log append
    append records in order
    include unique ascending sequence # in each record
    also a checksum for multi-sector records (maybe in End?)
    recovery must search forward for highest sequential #
  i'm assuming disk sector writes are atomic, and "work correctly"
    see FSD paper for better handling of disk failures

why is logging fast?
  group commit -- batched log writes.
    could delay flushing log -- may lose committed transactions
    but at least you have a prefix.
  single seek to implement a transaction.
    maybe less if no intervening disk activity, or group commit
  write-behind of data allows batched / scheduled.
    one data block may reflect many transactions.
    i.e. create many files in a directory.
    don't have to be so careful since the log is the real information
```