

6.033: Fault Tolerance: Logging
Lecture 16
Katrina LaCurts, lacurts@mit.edu

0. Introduction

- Currently: building reliable systems out of unreliable components. We're working on implementing transactions which provide
 - atomicity
 - isolation
- So far: have a poorly-performing version of atomicity via shadow copies.
- Today: Logging, which will give us reasonable performance for atomicity. Logging also works when we have multiple concurrent transactions, even though for today we're not thinking about concurrency.

1. Motivating example

```
- begin      // T1
  A = 100
  B = 50
  commit     // At commit: A=100; B=50
```

```
begin      // T2
A = A - 20
B = B + 20
commit     // At commit: A=80; B=70
```

```
begin      // T3
A = A + 30
--CRASH--
```

- Problem: A = 110, but T3 didn't commit. We need to revert.

2. Basic idea

- Keep a log of all changes and whether a transaction commits
 - every transaction gets a unique ID
 - UPDATE records include old and new values of a variable
 - COMMIT records specify that transaction committed
- (See slides for the log for this example)
- Nice: updates are small appends

3. How to use a log for transactions

- On begin: allocate new transaction ID (TID)
- On write: append entry to log
- On read: scan log to find last committed value
- On commit: write commit record
 - This is the commit point
 - Atomic because we can assume it's a single-sector write
 - Another way to do it would be to put checksums on each record

- and ignore partially-written records
- On recover: nothing
- (see slide for code)

4. Performance of log

- Writes: good. sequential = fast.
- Reads: terrible. Must scan entire log.
- Recovery: instantaneous

5. Cell Storage

- Improve read performance with cell storage.
 - (For us) stored on disk, i.e., non-volatile storage
 - Updates go to log and cell storage
 - Read from cell storage
- "Log" = write to log. "Install" = write to cell storage
- How to recover
 - Scan the log backwards, determine what actions were uncommitted and undo them
 - (see slide for code)
 - What if we crash during recovery? No worries; recover() is idempotent. Can do it repeatedly.
- How to write
 - Log before install, not the other way; otherwise, can't recover from a crash in between the two writes.
 - This is write-ahead logging

6. Performance of log + cell storage

- Writes: Okay, but now we write to disk twice instead of once
- Reads: fast
- Recovery: Bad. Have to scan the entire log.

7. Improving performance

- Improve writes: use a (volatile) cache
 - Reads go to cache first, writes go to cache and are eventually flushed to cell storage
 - Problem: After crash, there may be updates that didn't make it to cell storage (were in cache but not flushed)
 - Also could be updates in cell storage that need to be undone, but we had that problem before
 - Solution: We need a redo phase in addition to an undo phase in our recovery (see slide for code)
- Improving recovery
 - Problem: recovery takes longer and longer as the log grows
 - Solution: truncate the log
 - How?
 - Assuming no pending actions
 - Flush all cached updates to cell storage
 - Write a CHECKPOINT record
 - Truncate the log prior to the CHECKPOINT record
 - Usually amounts to deleting a file

- With pending actions, delete before the checkpoint and earliest undecided record.

8. What about un-undo-able actions?

- What if our transaction dispenses money from an ATM but then ends up not committing?
- Typically: wait for software that controls the action to commit and then take the action, but have a special way to detect whether the action has/will happened

9. Summary

- Logging is a general technique for achieving atomicity
 - Writes are fast, reads can be fast with cell storage
 - Need to log before installing (write-ahead), and need a recovery process
- Tomorrow is recitation: logging for file systems
- Now: we're good with atomicity
 - In fact, logging will work fine with concurrent transactions; the problem will be figuring out which steps we can actually run in parallel
- Wednesday: isolation
- Next week: distributed transactions