Banking problem

- **Bank with two accounts**
- **One operation: transfer $1 from A to B**
- **Components:**
  - client programs (concurrent)
  - System R
  - disk cache (for performance)
  - Disks

First implementation: mmap a file

- **What goes wrong?**
- **Atomicity:**
- **Consistency:**

First implementation: mmap a file

- **What goes wrong?**
- **Atomicity:**
  - What if server crashes between A -= $1 and B += $1?
  - What if client crashes/cancels between two operations?
- **Consistency: read-modify-writes interfere**

```
R = B.balance;
                          R = B.balance;
                          R = R + $1;
                          B.balance = R;
R = R + $1;
B.balance = R;
```

Next implementation: Shadow files

- **Never update disk pages in place**
  - Save copies of modified disk blocks
  - Update pointers to blocks
  - Atomically switch directory ptr from shadow PT to current
- **Problems:**

Next implementation: Shadow files

- **Never update disk pages in place**
  - Save copies of modified disk blocks
  - Update pointers to blocks
  - Atomically switch directory ptr from shadow PT to current
- **Problems:**
  - Doesn't offer consistency with multiple clients
  - Bad for both sequential and random access
    - Sequential blocks not near each other
    - Index too large to keep in core
  - Requires twice the storage to rewrite whole file!

Transactions

- **Series of modifications to state**

```
BEGIN
actions
END
```

- **Has the following properties:**
  - A. atomic – either completes entirely of completely undone
  - C. consistent – results as if all transactions were serialized
  - D. durable – once complete, crash won't erase effects
- **What are dangers to consistency?**

Consistency failures

- **Lost Update. READ–WRITE–WRITE**
  - Second write overwrites the first (banking example)
- **Dirty read. WRITE–READ–WRITE**
  - Read returns uncommitted value
- **Unrepeatable read. READ–WRITE–READ**
  - Two reads of same data return different values
- **Can view transaction dependencies graphically**
  - If one transcation depends on another, draw arrow
  - Any cycle represents a dependency failure

Locks

- **Restrict concurrent access per DB record**
- **Acquired as transaction reads/writes data**
  - Shared for reading
  - Can upgrade to exclusive for writing
  - Possibility of deadlock
- **Released at end of transaction**
- **Assure no one reads uncommitted values!**

When do actions not complete?

When do actions not complete?

- **Abort**
  - Cancelled by user/client (3%)
  - Cancelled by system ($\ll$ 1%)
    (deadlock, resource shortage, system shutdown, etc.)
- **Restart**
  - After crash and reboot (OS or hardware failure)
- **Media failure**
  - Disk dies
- **Unrecoverable error**
  - Operator error (disk fails & didn't do backups)
  - Software bug in RSS

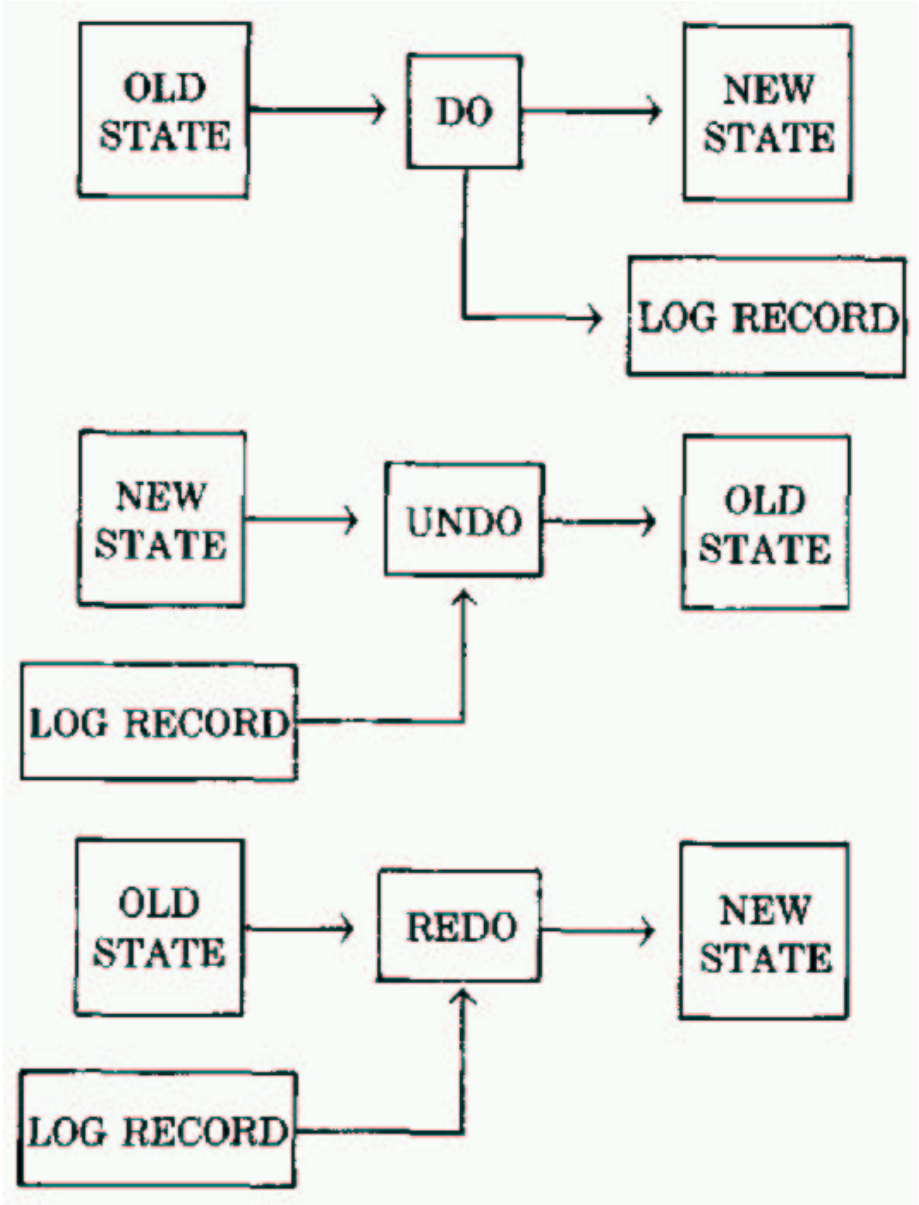
Save points

- **Allow transaction to rewind to previous state**
  - Example: Multihop airline reservation
  - Don't flush entire trip if one seat/flight unavailable
- **Programming interface**
  - SAVE directive creates save point N
  - UNDO N returns to save point N
- **Implementation:**
  - Record state of all locks and cursors in log
  - Release any locks acquired since save
  - No locks to reacquire–locks held until end of transaction

Nested transactions

- **Not available in system R**
  - More general than save points
- **Can begin one transaction within another**
  - Abort backs out all changes–like save points
  - Commit will release locks
  - Nested commits not globally visible until outermost commit
- **Very good for concurrency**
  - Perform various components of a transaction in parallel
    E.g., $A -= \$1$ and $B += \$1$
- **Harder to implement efficiently**

Logging


OLD STATE
DO
NEW STATE
LOG RECORD
NEW STATE
UNDO
OLD STATE
LOG RECORD
OLD STATE
REDO
NEW STATE
LOG RECORD

Logging

- **Keep log of all actions**
  - Optionally duplex log
- **Each action has 3 operations:**
  - DO – performs & logs action
  - UNDO – reverts action
  - REDO – repeats action
- **Merge all actions in one log**
  - Reduces seek time
  - Reduces fragmentation
  - Requires linked list of actions within one transaction

What's in a log record?

- **Modification being performed**
  - File name
  - Record ID
  - Old value
  - New value
- **Log system info**
  - Log record length
  - Transaction ID
  - Action ID
  - Timestamp
  - Pointer to previous record in this transaction

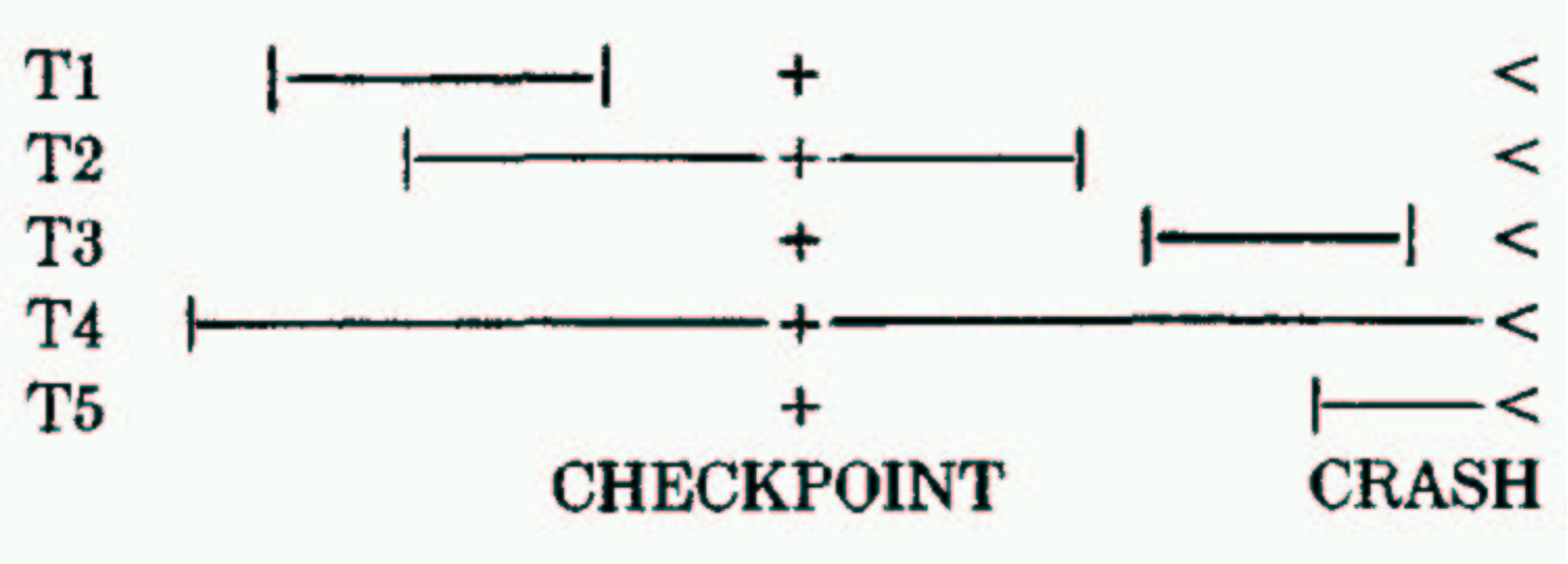
Commit processing

- **Write commit record to the log**
- **Never update shadow file before log**
- **Transaction considered committed after log written**
  - Don't need to update shadow on each commit
- **After a crash, don't know which actions are in file**
  - What point in log does shadow state correspond to?
  - Does state even correspond to one point in log?
- **Plus, don't want to keep log around forever...**

Checkpoints

- **Wait until no RSS actions in progress (why?)**
- **Write checkpoint record to log**
  - List of all transactions in progress
  - Pointer to most recent log entry of each transaction
- **Update shadow state of all files with current state**
- **Update shadow state to point to checkpoint record**
- **Crash betw. writing to file & checkpoint to log?**
  - Shadow directory contains pointer to checkpoint record

System restart

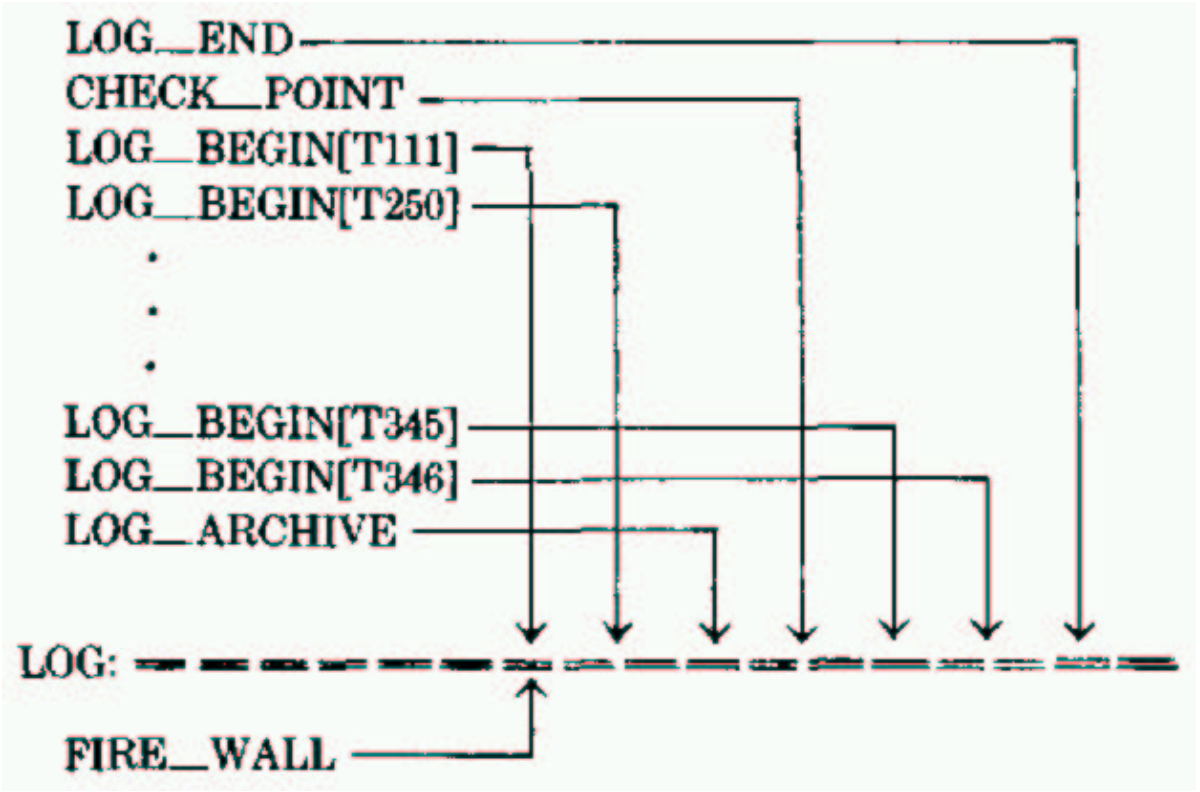


- Scan log forwards, backwards
- Categorize & process winners & losers
- Write new checkpoint to log

Locking and UNDO

- **Problem: What if deadlock while executing UNDO**
  - Cannot abort UNDO – transaction already aborting
- **Good news: Record locks held to end of transaction**
  - Will still be held at time of abort
- **Bad news: Page-level locks (false sharing)**
  - UNDO transactions marked as "Golden"
  - Only one "golden" transaction executes at a time
  - Always abort the non-golden xaction after deadlock
- **More bad news: What about during restart**
  - No locking during restart
  - All transactions recovered sequentially

Cleaning the log



- **Must keep log to earliest of:**
  - Last checkpoint
  - Oldest record of a pending transaction (can always abort really old)
  - Last archive (for media failures)

Recovery from media failure

- **Same as system restart**
  - Use older backup
  - Use more of log
- **No need to implement extra code for this**

Other transaction techniques

- Group commit
- Write-ahead logging
- Keep undo log separate, garbage collect it sooner
- Message logging! (Big omission in system R)
- Keep locks after commit (or downgrade to shared)
- Two-phase commit
- How to use disks to get better throughput?
- How might you implement truly atomic rename?