Crash recovery

All-or-nothing atomicity & logging

Institute of Parallel and Distributed Systems
Rong Chen

*Some slides adapted from Jinyang Li

What we've learnt so far…

- Consistency in the face of ≥2 copies of data and concurrent accesses
  - Sequential consistency
    - everyone sees same read and write order
  - Release consistency
    - everyone sees write in unlock order
  - Eventual consistency
    - eventual convergence of state
    - causality preserving (optionally)
- This class: ensure data consistency across node failure (crashes/reboots)

Crash at the “wrong time” is problematic

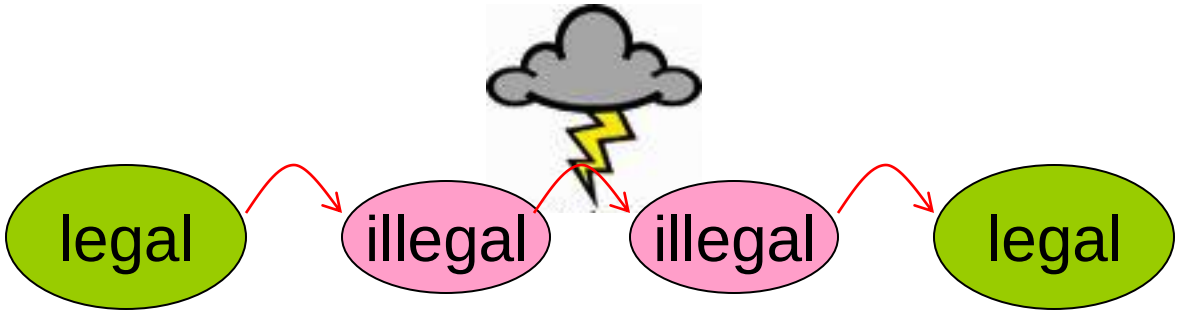
- Examples:
  - Failure during middle of online purchase
  - Failure during “mv /home/jinyang /home/jy”
- What guarantees do applications need?

All-or-nothing atomicity

- All-or-nothing
  - A set of operations either all finish or none at all
  - No intermediate state exist upon recovery
- All-or-nothing is one of the guarantees offered by database transactions

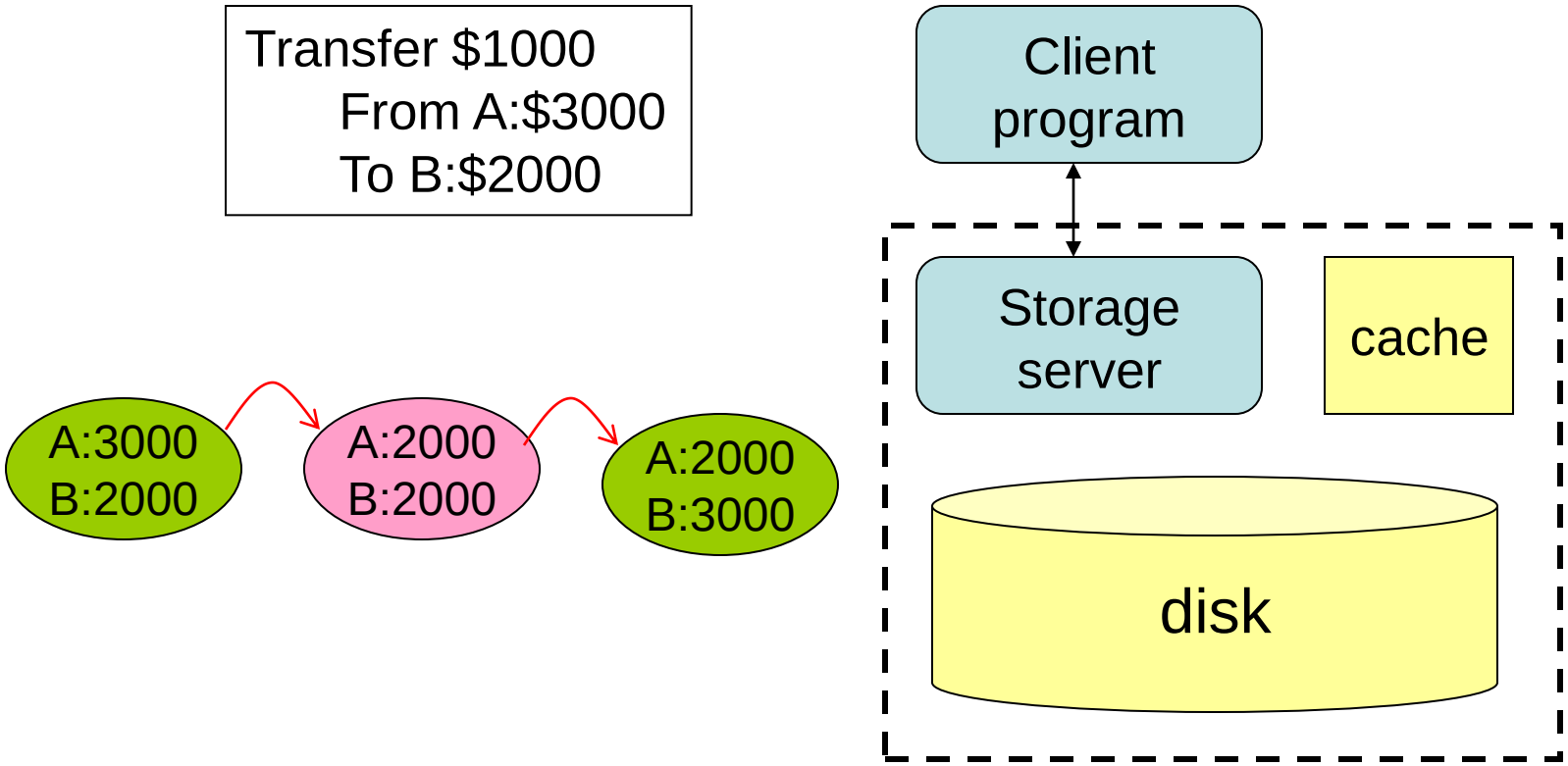
Challenges of implementing all-or-nothing

- Crash may occur at any time

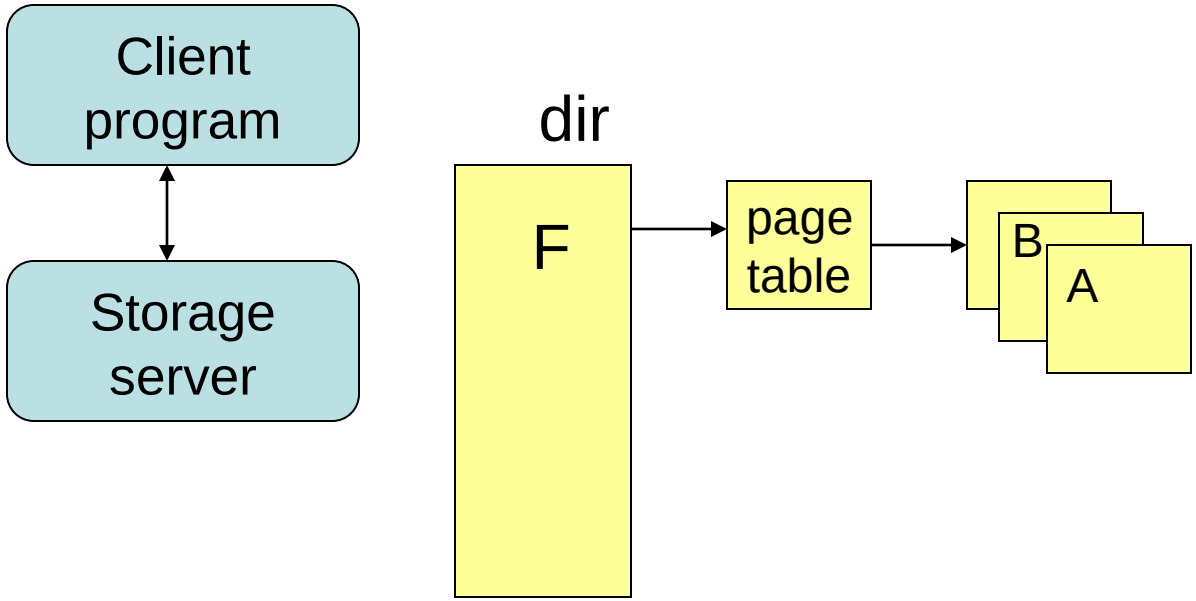


- Good normal case performance is desired
  - Systems usually cache state

An Example

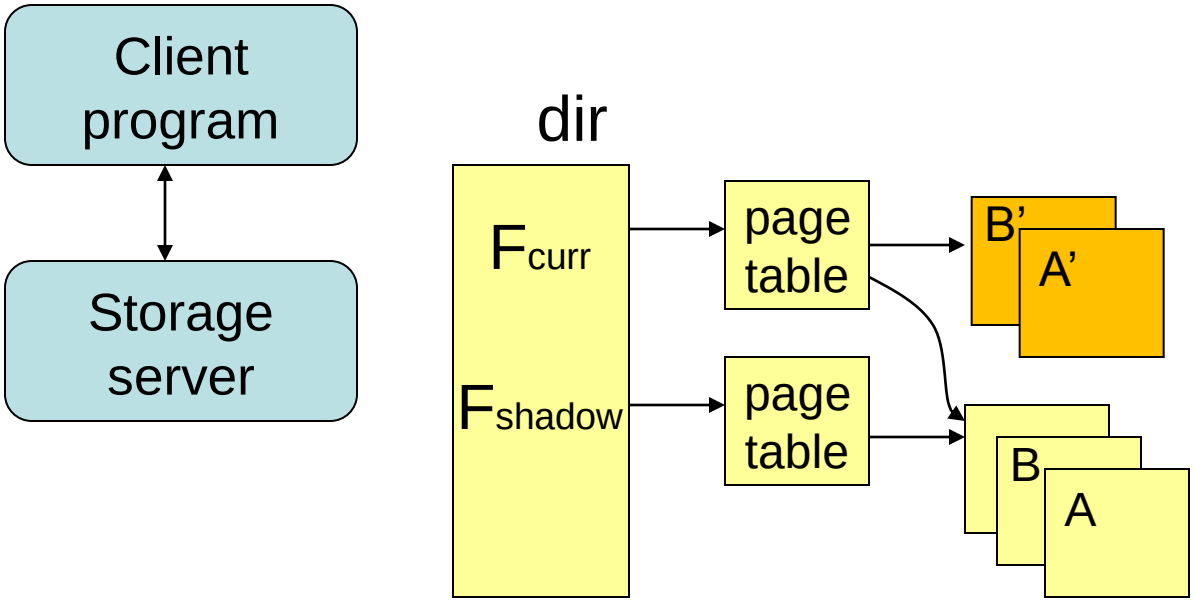

Transfer $1000
From A:$3000
To B:$2000
A:3000
B:2000
A:2000
B:2000
A:2000
B:3000
Client program
Storage server
cache
disk

1st try at all-or-nothing



- Map all file pages in memory
- Modify A = A-1000
- Modify B = B+1000
- Write A to disk
- Write B to disk

2nd try at all-or-nothing



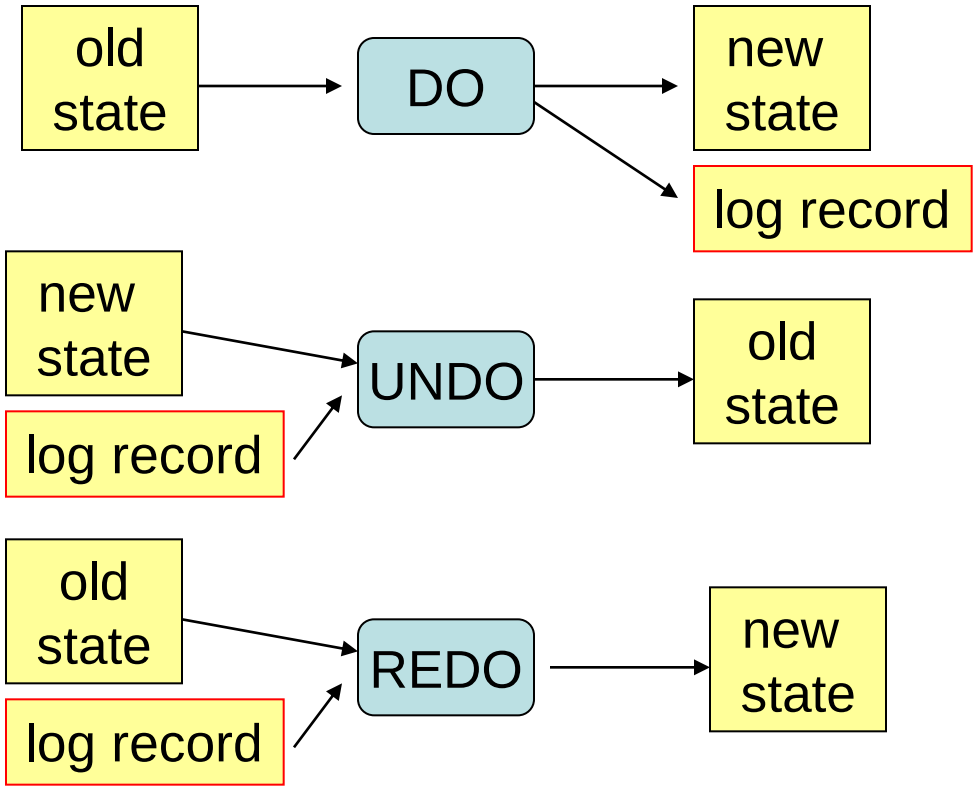
- Read A from F_{curr}, read B from F_{curr}
- A=A-1000; B = B+1000;
- Write A to F_{curr}
- Write B to F_{curr}
- Replace F_{shadow} with F_{curr}

Problems with the 2nd try

- Multiple transactions might share the same file:
  - Two concurrent transactions:
    - T1: transfer 1000 from A to B
    - T2: transfer 10 from C to D
    - A&C are on the same page
  - Committing T1 would (falsely) write intermediate state of T2 to disk

3rd try is a charm

- Keep a log of all update actions
- Each action has 3 required operations



SysR: logging

- Merge all transactions into one log
  - Append-only
  - Reduce random access
  - Require linked list of actions within one transaction
- Each log record consists of
  - Transaction ID
  - Action ID
  - Pointer to previous record in this transaction
  - Action (file name, record ID, old & new value)
  - ….

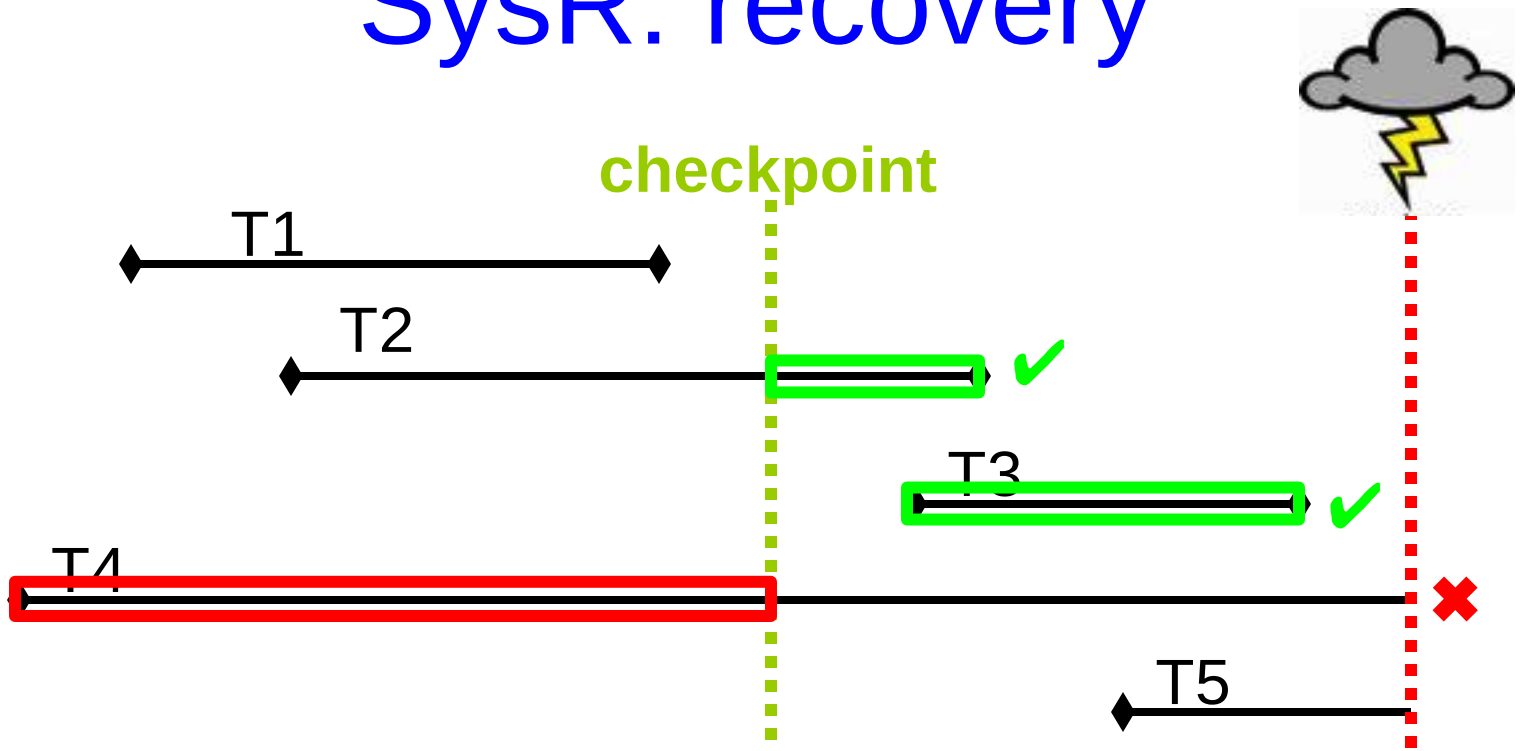
SysR: logging

- How to commit a transaction?
- SysR logging rules:
  1. Write log record to disk before modifying persistent state (e.g. replace Fshadow with Fcur) → Write Ahead Log (WAL) protocol
  2. At commit point, append a commit record and force all transaction's log records to disk (write commit record to disk at last)
- How to recover from a crash? (no checkpoint)

SysR: checkpoints

- Checkpoints make recovery fast
  - No need to start from a blank state
- How to checkpoint?
  1. Wait till no ~~transactions~~ actions are in progress (why?)
  2. Write a checkpoint record to log
     - Contains a list of all transactions in progress and pointers to their most recent log records
  3. Save all files
  4. Atomically save checkpoint by updating directory root to point to latest checkpoint record

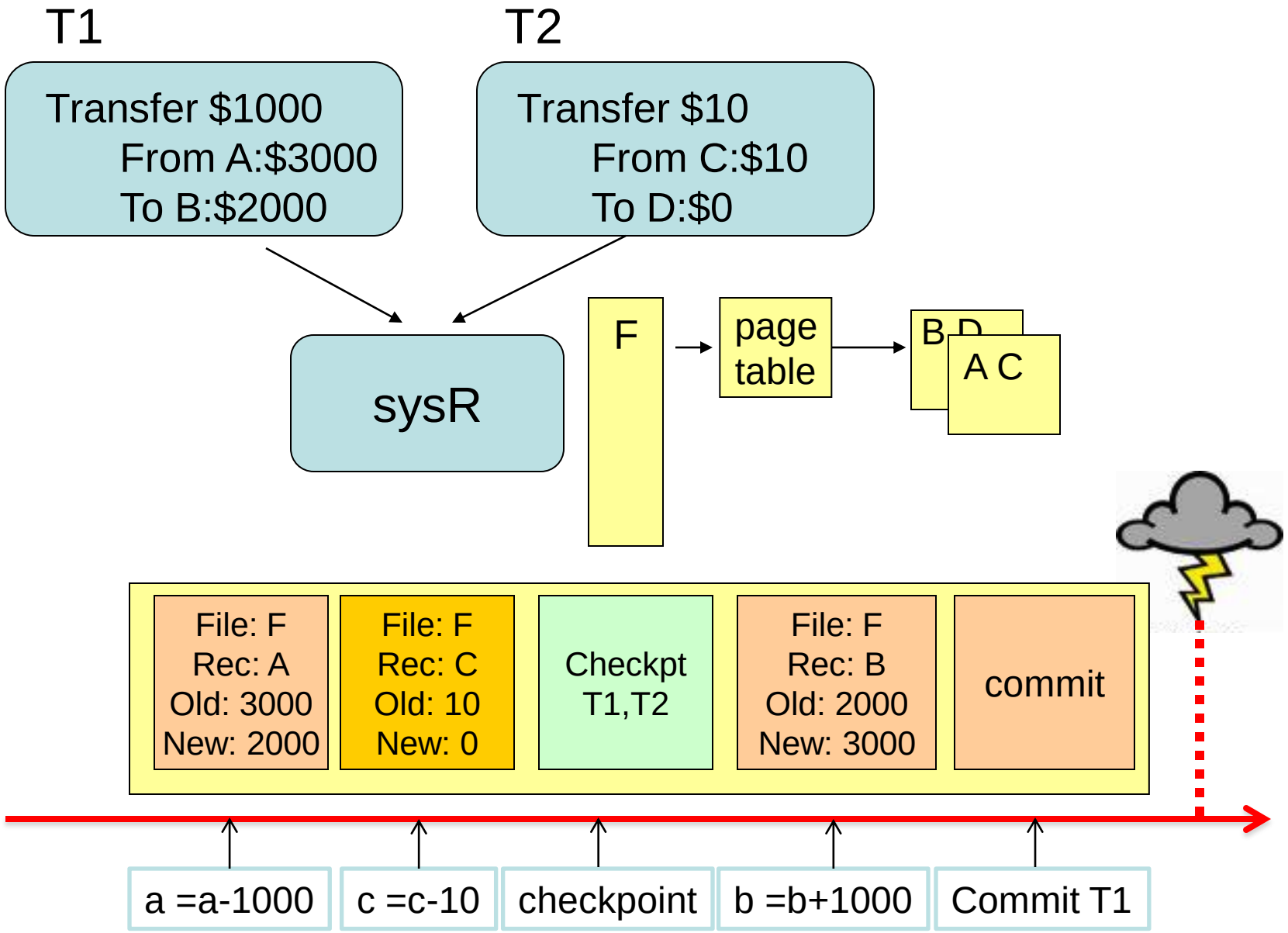
SysR: recovery



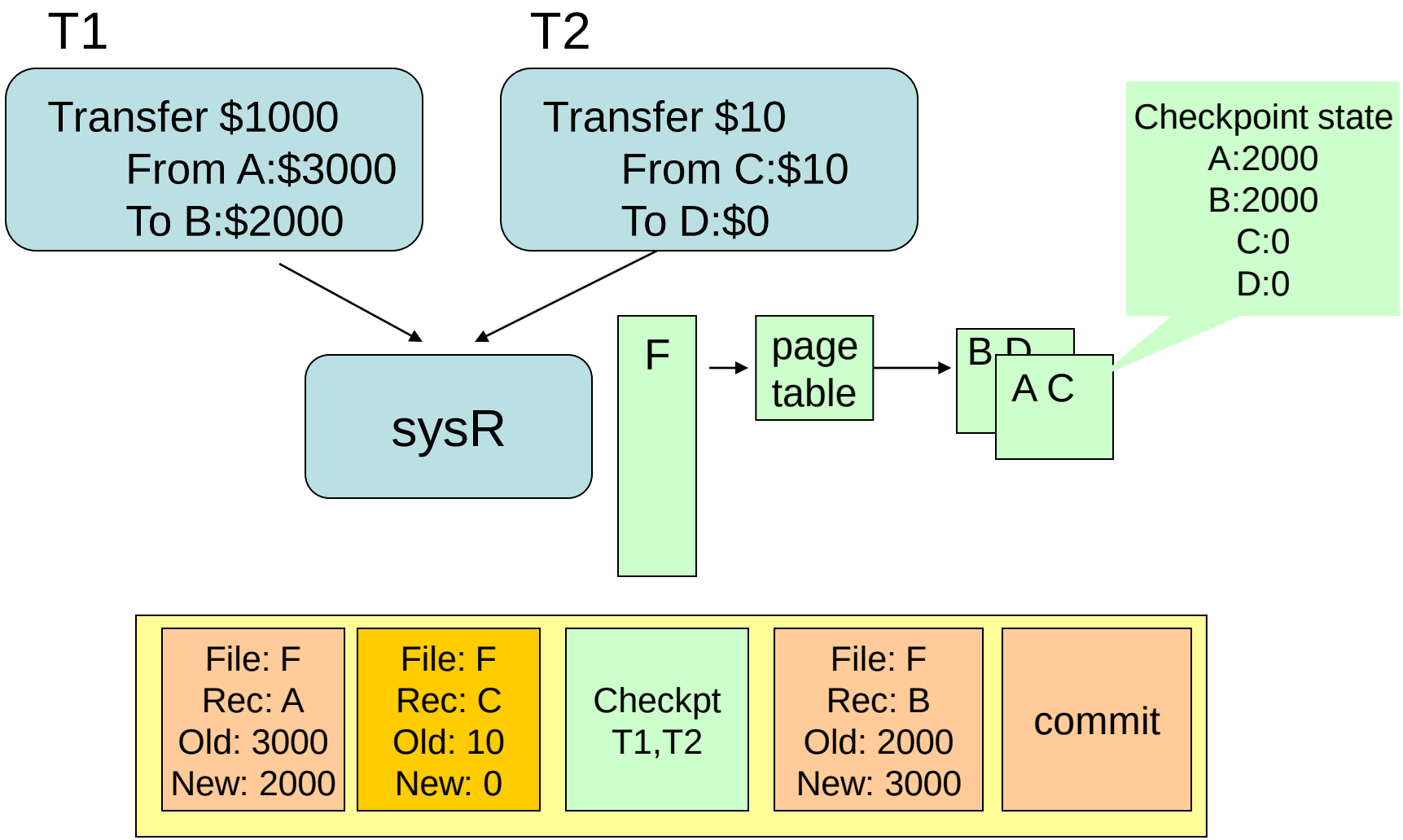
1. Read most recent checkpoint to learn that T2, T4 are ongoing transactions
2. Read log⟶ to learn that T2, T3 are winners and T4 is a loser
3. Read log⟵ to undo loser
4. Read log⟶ to redo winner

How about T5?

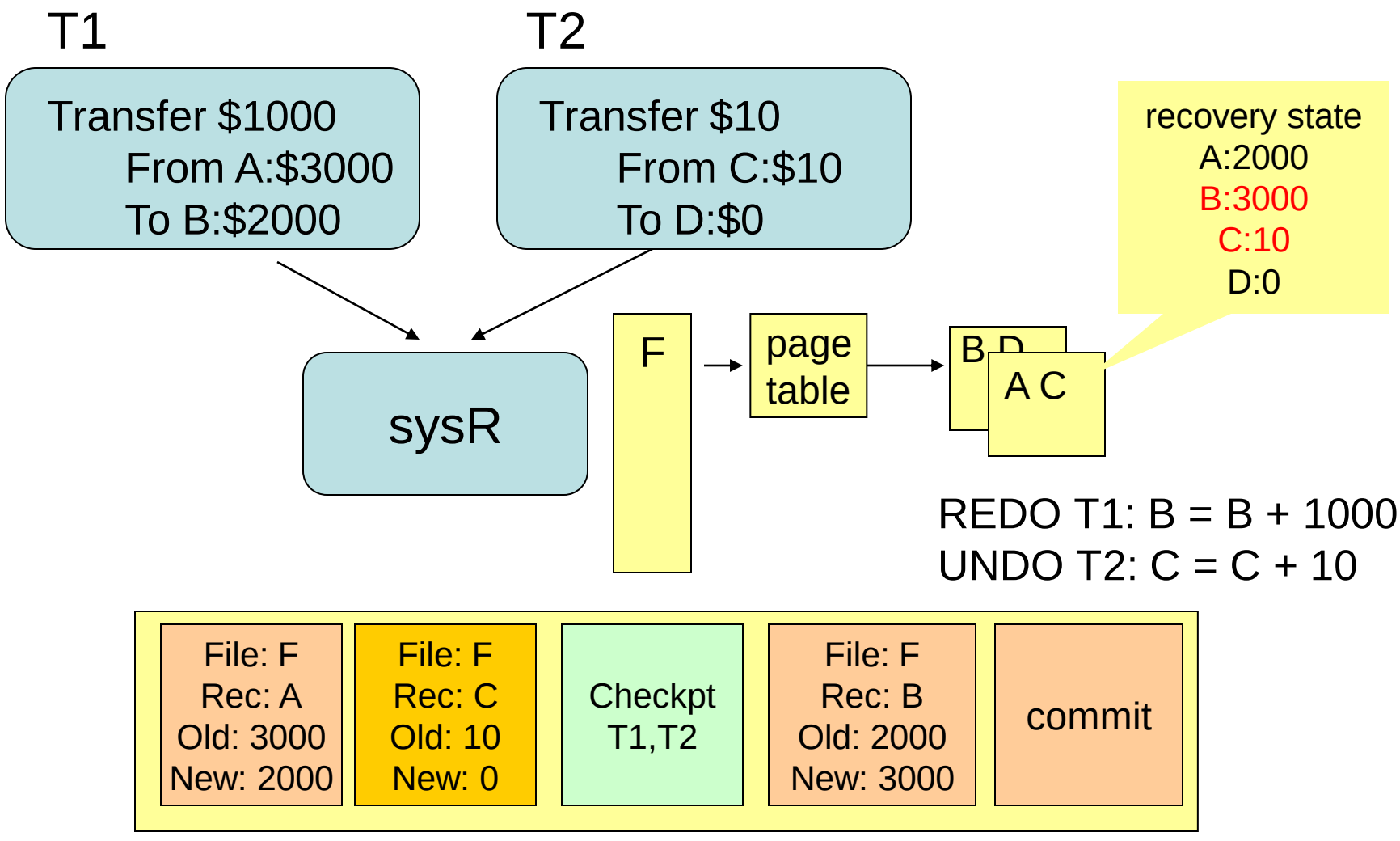
Example using logging



Example recovery



Example recovery



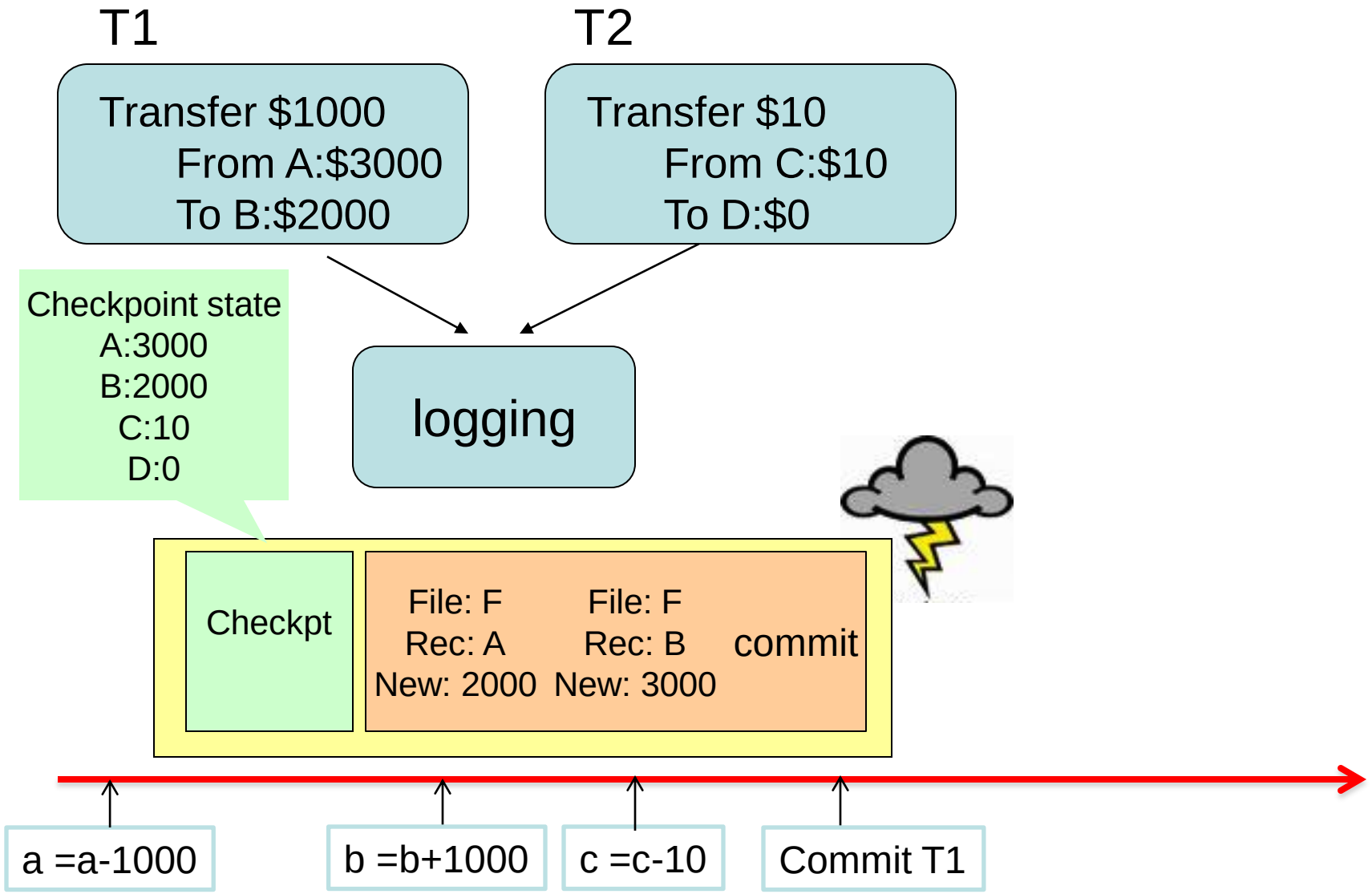
UNDO/REDO logging

- SysR records both UNDO/REDO logs
  - Because a transaction might be very long
    - Must checkpoint w/ ongoing transactions
  - Because a long transaction might be aborted by applications/users
    - Must undo the effects of aborted transactions
- Can we have REDO-only logs for systems w/ “short transactions”?

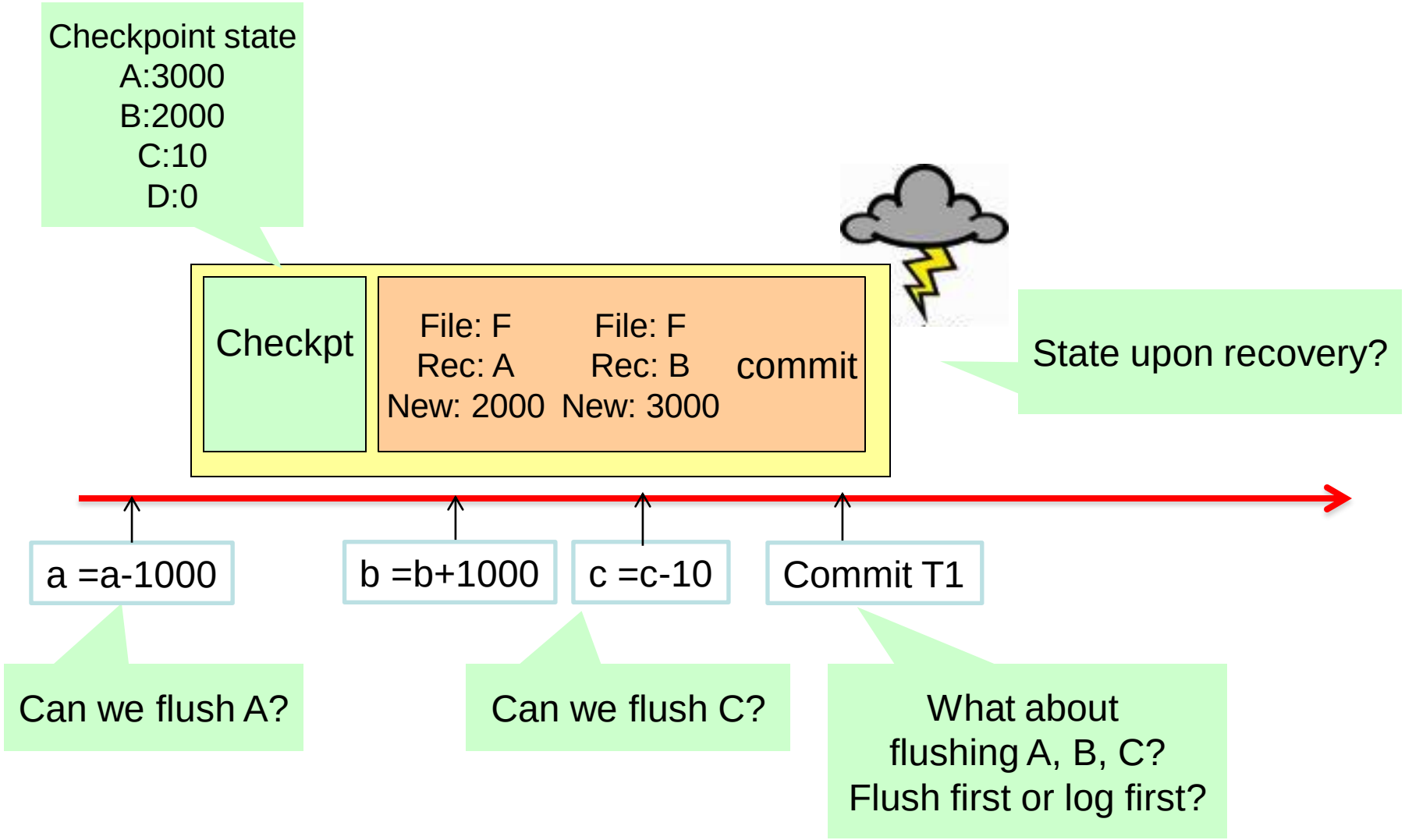
REDO-only logs

- What's the logging rule?
  - Append REDO log records before/after flushing state modification?
  - Can uncommitted transactions flush state?
- How to checkpoint?

Example using REDO-log



Example using REDO-log

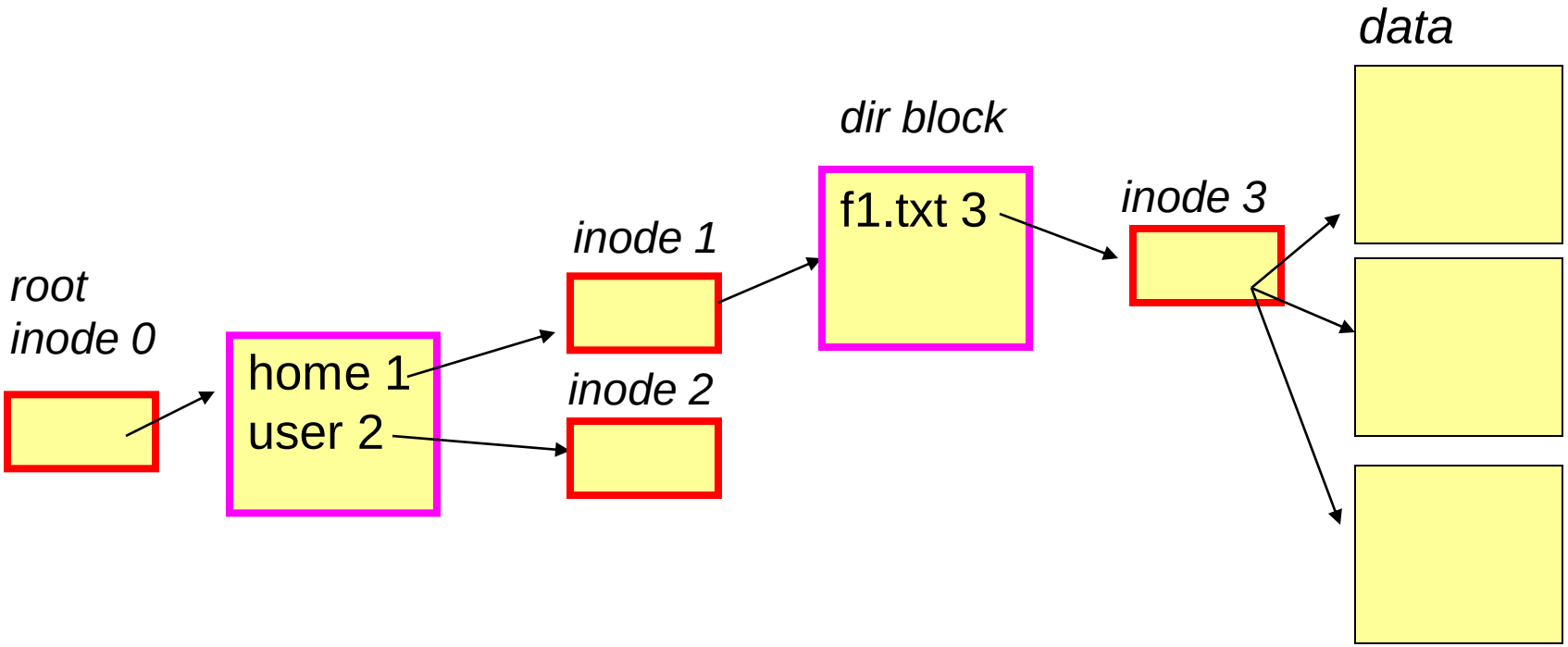


UNDO-only logging

- Can we have UNDO-only logs?
  - What's the logging rule?
  - To checkpoint or not to checkpoint?
  - Problems?
- Compare UNDO/REDO, REDO-only and UNDO-only logging

Case study: disk file systems

FS is a complex data structure



- i-nodes and directory contents are called meta-data
- Also need a free i-node bitmap, a free data block bitmap

Kernel caches used blocks

- Buffer cache holds recently used blocks
- Very effective for reads
  - e.g. access root i-node is extremely fast
- Delay writes
  - Multiple operations can be batched to reduce disk writes
  - Dirty blocks are lost during crash!

Handling crash recovery is hard

- Dangers if crash during meta-data modification
  - Files/dirs disappear completely
  - Files appear when they shouldn't
  - Files have content belonging to different files
- Dangers if crash during file content modification
  - Some writes are lost
  - File content are a mix of old and new data

Goal of FS recovery

- Leave file system in a good state w.r.t. meta-data
  - Recovery complex b-tree operation (e.g. splitting during insert)
- It is okay to lose a few operations
  - To tradeoff for better performance during normal operation
- Recovery speed: no whole-disk scans

Example crash problems

User program

```
fd = create("d/f", 0666);
write(fd, "hello", 5);
```

```
unlink("d/f");
```

File system writes

1. i-node bitmap (Get a free i-node for "f")
2. "f"'s i-node (write owner etc.)
3. "d"'s dir content (add "f" to i-number mapping)
4. "d"'s i-node (update length & mtime)
5. block bitmap (get a free block for f's data)
6. data block
7. "f"'s i-node (add block to list, update mtime & length)

8. "d"s content (remove "f" entry)
9. "d"s i-node (update length, mtime)
10. i-node bitmap
11. block bitmap

FS uses write-back cache

- Synchronous writes for consistency
  - If every write goes to disk, how fast?
  - Writes be performed in a particular order
  - No localized
- Write-back cache
  - FS only writes to cache
  - When cache fills up with dirty blocks, flush some to disk

Can we recover with a write-back cache?

- Write-back cache may write to disk in any order
- Worst case scenarios:
  - A few dirty blocks are flushed to disk, then crash, recover

Example crash problems

```
fd = create("d/f", 0666);
write(fd, "hello", 5);
```

```
unlink("d/f");
```

1. i-node bitmap (Get a free i-node for "f")
2. "f"s i-node (write owner etc.)
3. "d"s dir content (add "f" to i-number mapping)
4. "d"s i-node (update length & mtime)
5. Block bitmap (get a free block for f's data)
6. Data block
7. "f"s i-node (add block to list, update mtime & length)
8. "d"' content (remove "f" entry)
9. "d"' i-node (update length, mtime)
10. i-node bitmap (mark "f"'s i-node as free)
11 block bitmap (mark "f"'s blocks as free)

- Wrote 1-8
- Wrote just 3
- Wrote 1-7 and 10

A more serious crash

```
unlink(“d/f1”);
create(“d/f2”);
```

- Create happens to re-use i-node freed by unlink
- Only second write of “d” content goes to disk
  - #3: update “d”’ content to add “f2” to i-number mapping
- Recovery:
  - Nothing to fix
  - But file “f2” has “f1”’ content
  - Serious undetected inconsistency

Logging File System

- FS needs all-or-nothing meta-data update
- Why aren't synchronous meta-data updates enough?
  - They're slow
  - Recovery may require checking/scanning the whole FS (e.g. fsck)
  - Some operations don't have an obvious single committing write

Logging File System

- Learn from database systems
  - Performance can be gained and consistency achieved by writing updates to a log on stable storage

Although logging may not be a cost-effective technique for the data of a file system, it is effective for the metadata

A Strawman of LFS

- What does a write do?
  - modify block in the data cache
  - append a record to the log
- What does recovery do?
  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

A Strawman of LFS

- When can we write dirty data from cache to disk?
  - Any time
  - Since recovery reconstructs everything
- Why can't we use any of on-disk data's contents during recovery?
  - Don't know if a block is from an uncommitted action
  - The REAL data is in the log!

Problem of Strawmen

- We have to store the whole log forever
- Recovery has to replay from the beginning of time

FSD: WAL + Ckpt. + REDO

- Recovery needs to know a point in log at which it can start a "checkpoint"
  - Pointer into log, stored on disk
- Checkpoint rule:
  - all data writes before ckpt. must be stable on disk
  - Ckpt. may not advance beyond 1st uncommitted
  - Flush a bunch of early writes, update ckpt. ptr

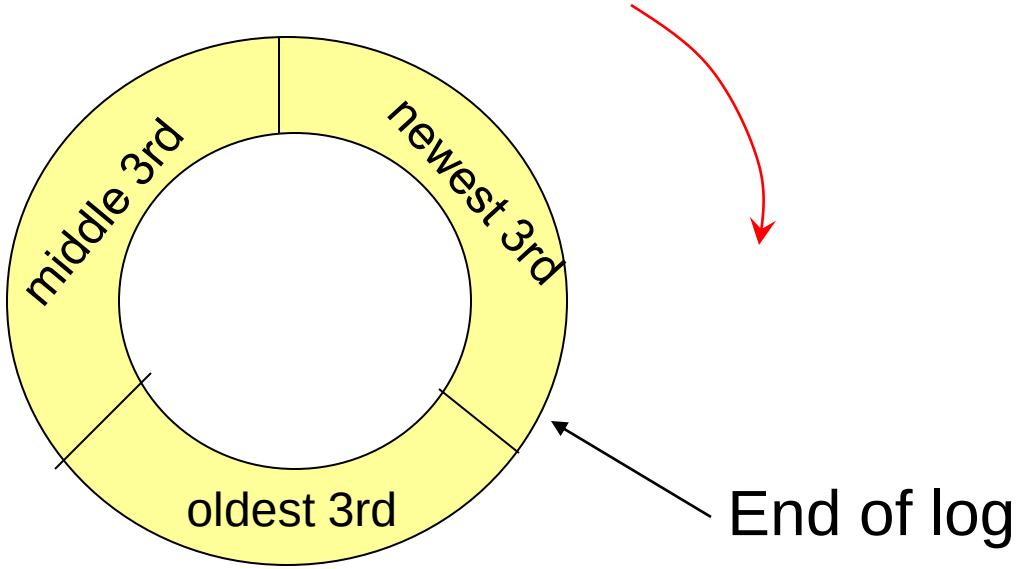
FSD: WAL + Ckpt. + REDO

- 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

FSD's logging

- Log is kept as a circular disk file
  - Append new records to log by synchronously writing it to the file
- When is in-memory log forced to disk?
  - Group commit, every 1/2 second
    - Tolerate little uncertainty in FS
- What if it runs out of log space?
  - When can we reclaim log space?

FSD’s log space reclaimation



- Before reclaiming oldest 3rd, flush all its records to disk if the page is not found in later 3rds

FSD's Recovery

- Recovery re-dos log records
  - Scan log from ckpt. to see what transactions are committed
  - Replay commited updates from ckpt onward
  - How does recovery find the start of the log?
    - On-disk ptr to start of oldest third
    - Written when prev third was re-used

File Transaction

1. write disk cache in memory
2. append log records
3. append commit record
4. wait for all its log records to reach disk
5. write disk cache to disk
6. update checkpoint

(last three in background)

Questions

The paper says that during a one-byte file create FSD writes the leader+data page synchronously to the disk, but records the update to the file name table in memory and only writes it back to disk later.

Why do you suppose the FSD designers decided to write the data page synchronously?

What (if anything) might go wrong if FSD instead wrote the file's data in the in-memory disk cache, and only wrote it to disk later?

Next Classes

- Concurrent Control
  - 2PL, Snapshot Isolation, …
  - Paper: A Critique of ANSI SQL Isolation Levels

Thank you !

Any questions?