Crash Recovering & Logging

Rong Chen

Institute of Parallel and Distributed Systems

Shanghai Jiao Tong University

http://ipads.se.sjtu.edu.cn/rong_chen

Credits: some slides from Jinyang Li (NYU)

Consistency so far

Concurrency forces us to think about meaning of **read/write**

Sequential consistency: all accesses appear executed in a global order by all participants

Eventual consistency: eventual convergence of state and causality preserving (optionally)

How about machine **failure?**

Crash at the Wrong Time

Examples

- Failure during middle of online purchase
- Failure during "**mv** /home/rchen /home/rong"

Problematic

- Pay the bill or not? twice?
- Where and how many files? one / zero / dup?

Who guarantees do applications need?

All-or-nothing Atomicity

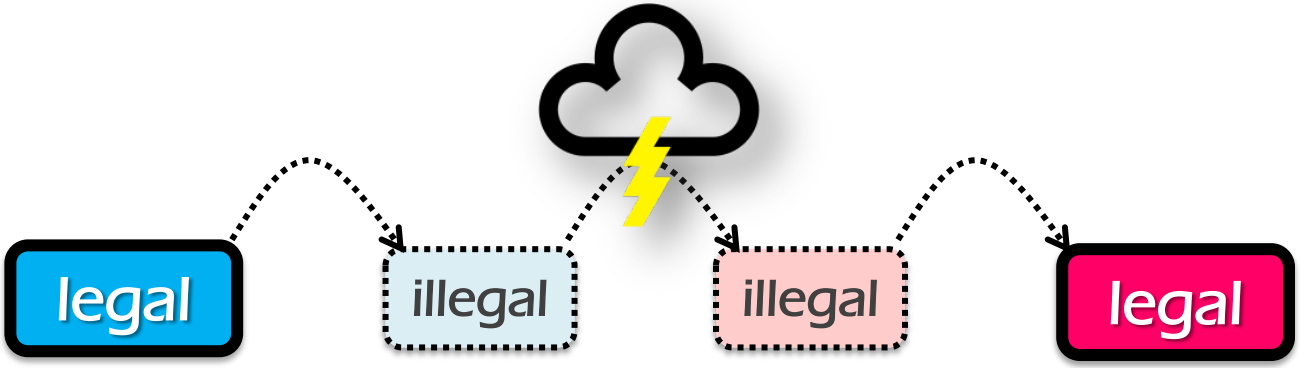
All-or-nothing

- 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 All-or-nothing

Crash may occur at any time

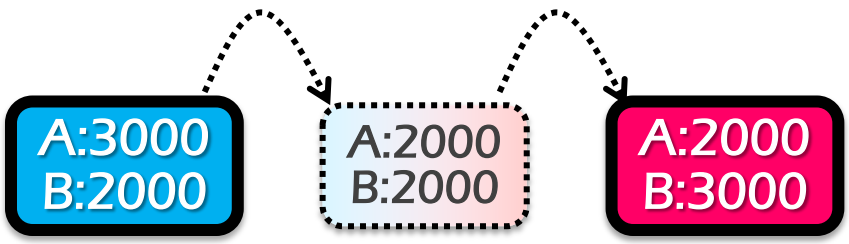


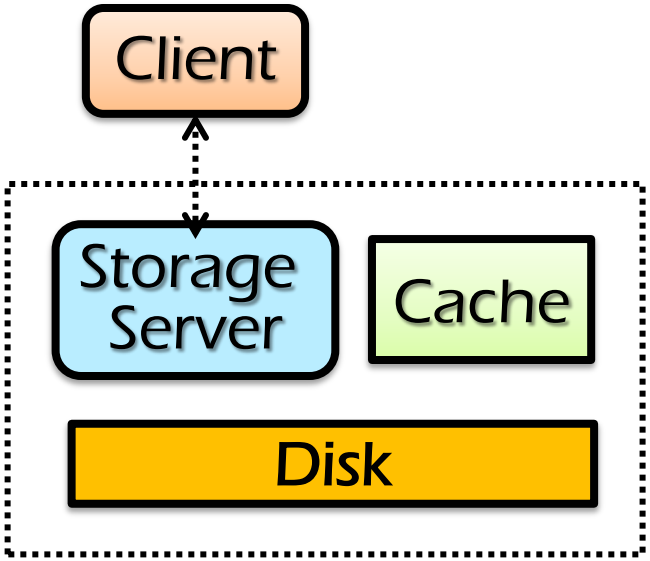
Good normal case performance is desired

- Systems usually cache state

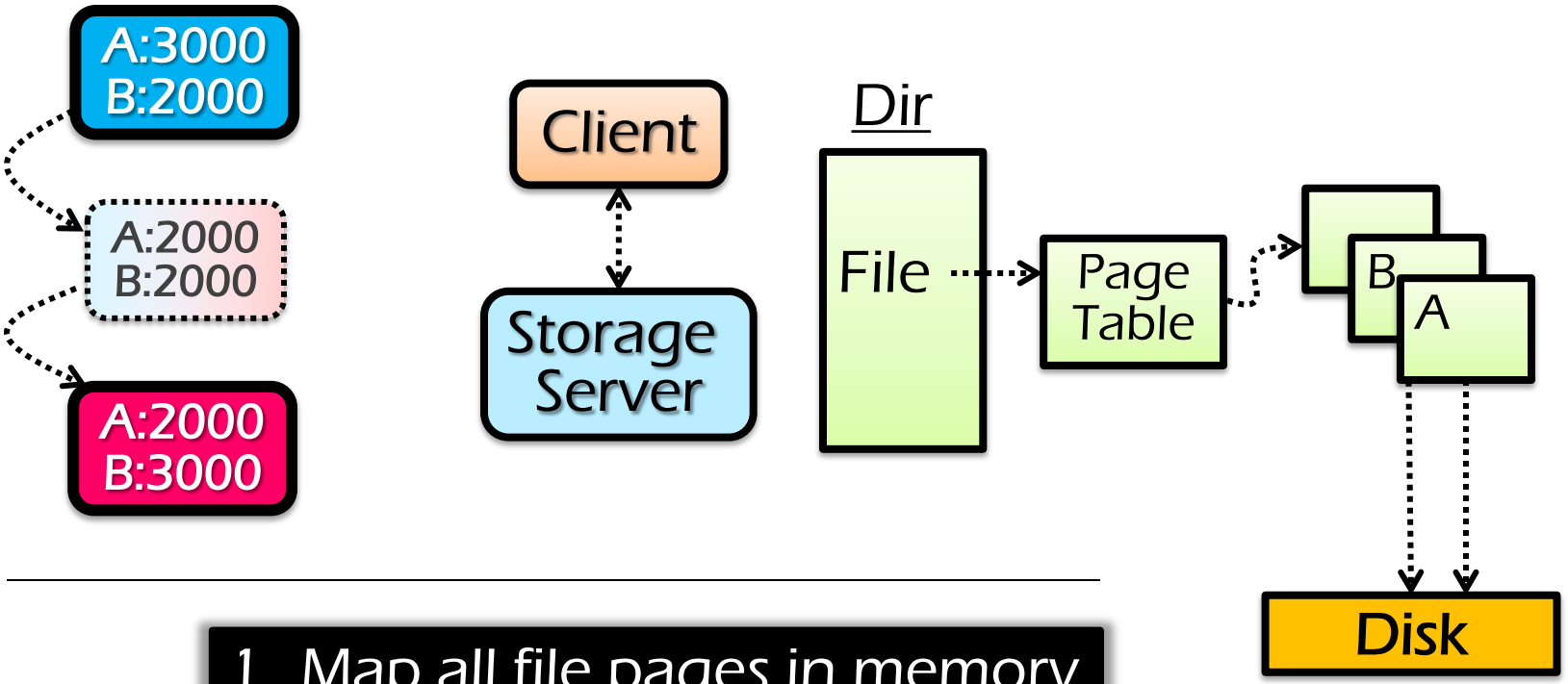
Example

Transfer $1000
From A: $3000
To B: $2000



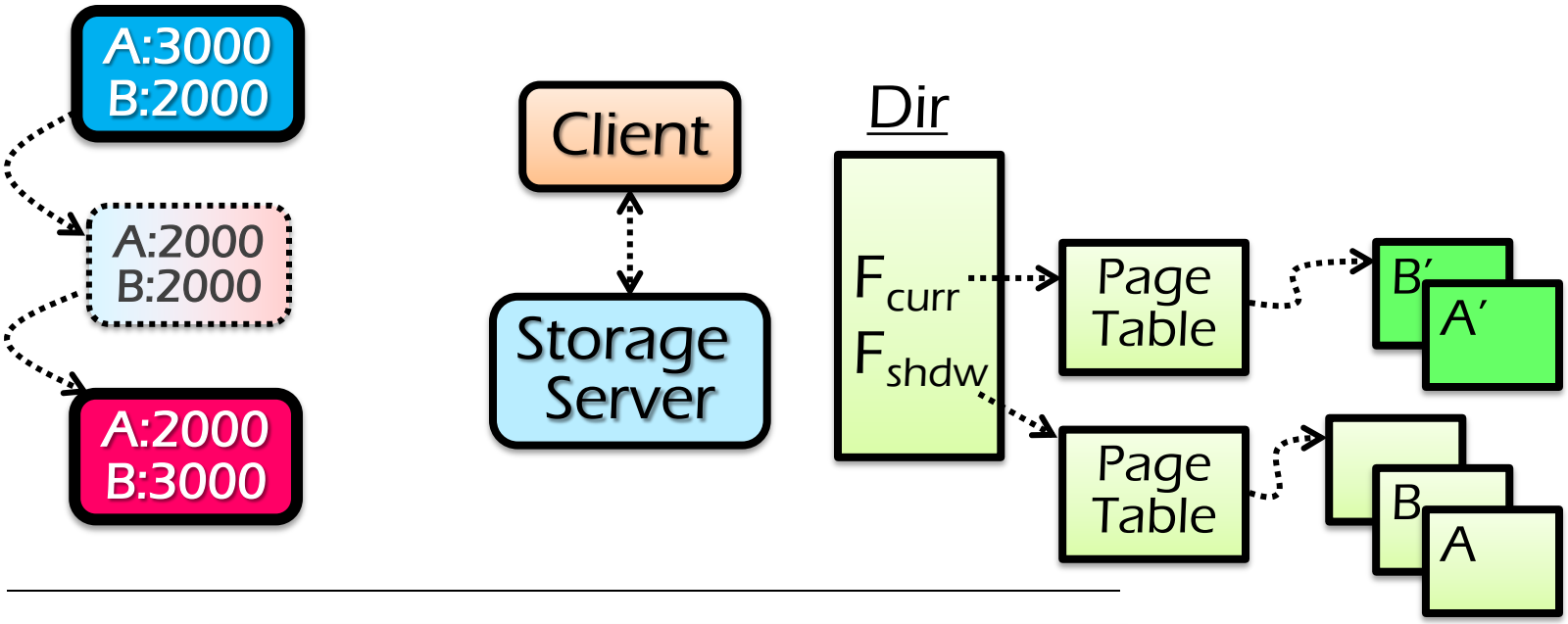


Problem



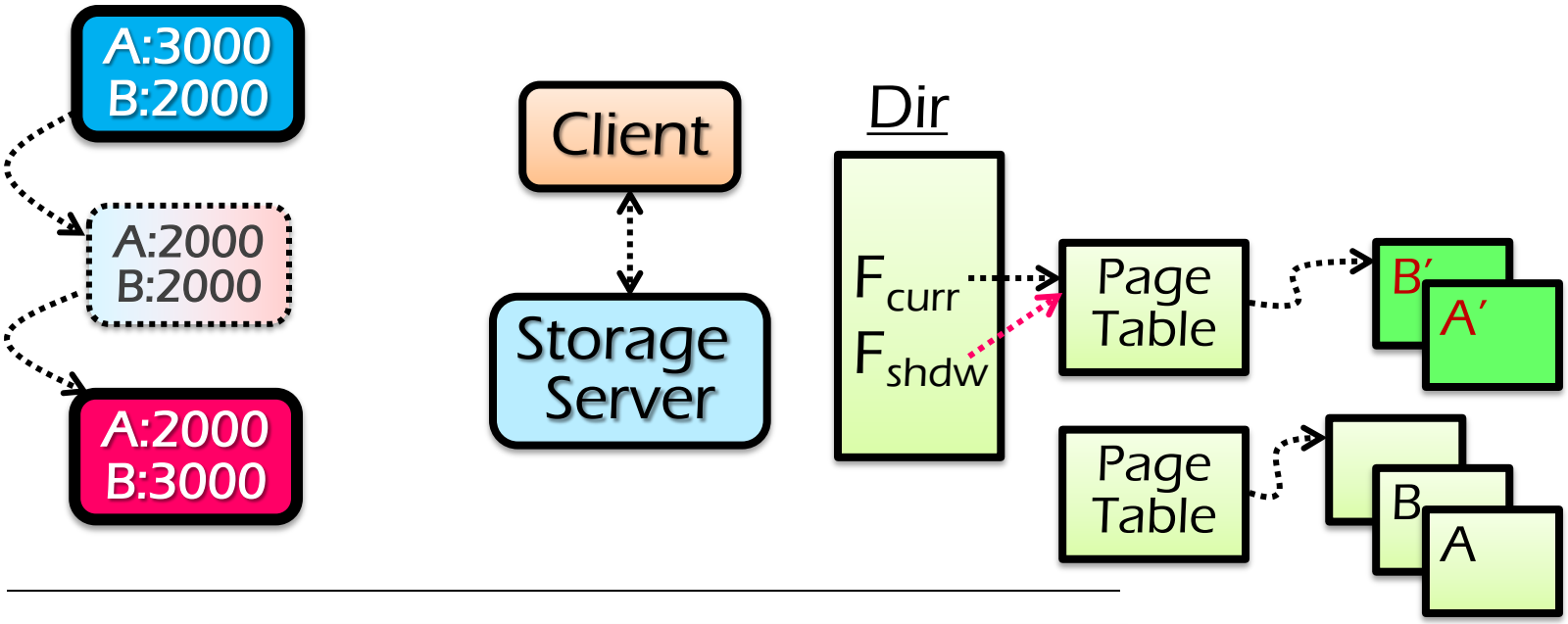
1. Map all file pages in memory
2. Modify A = A – 1000
3. Modify B = B + 1000
4. Write A to disk
5. ~~Write B to disk~~

Straw Man: Atomicity



1. Read A and B from F_{curr}
2. Create F_{shdw}
3. Write A = A – 1000 to F_{curr}
4. Write B = B + 1000 to F_{curr}
5. Replace F_{shdw} with F_{curr}

Straw Man: Atomicity



Atomic Ops

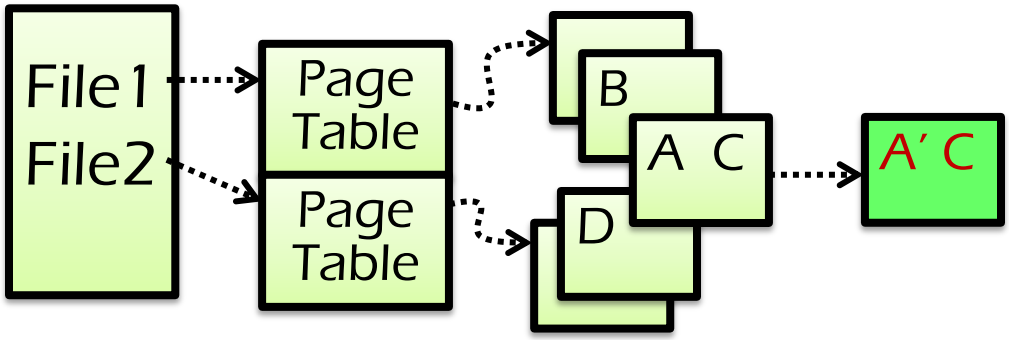
1. Read A and B from F_{curr}
2. Create F_{shdw}
3. Write A = A – 1000 to F_{curr}
4. Write B = B + 1000 to F_{curr}
5. Replace F_{shdw} with F_{curr}

Problems of Straw Man

Multiple transactions share the same file

- Two concurrent transactions: T1 & T2
  1. T1 transfer 1000 from A to B
  2. T2 transfer 10 from C to D
  3. A and C are on the same page
- Commit T1 would (falsely) write intermediate state of T2 to disk

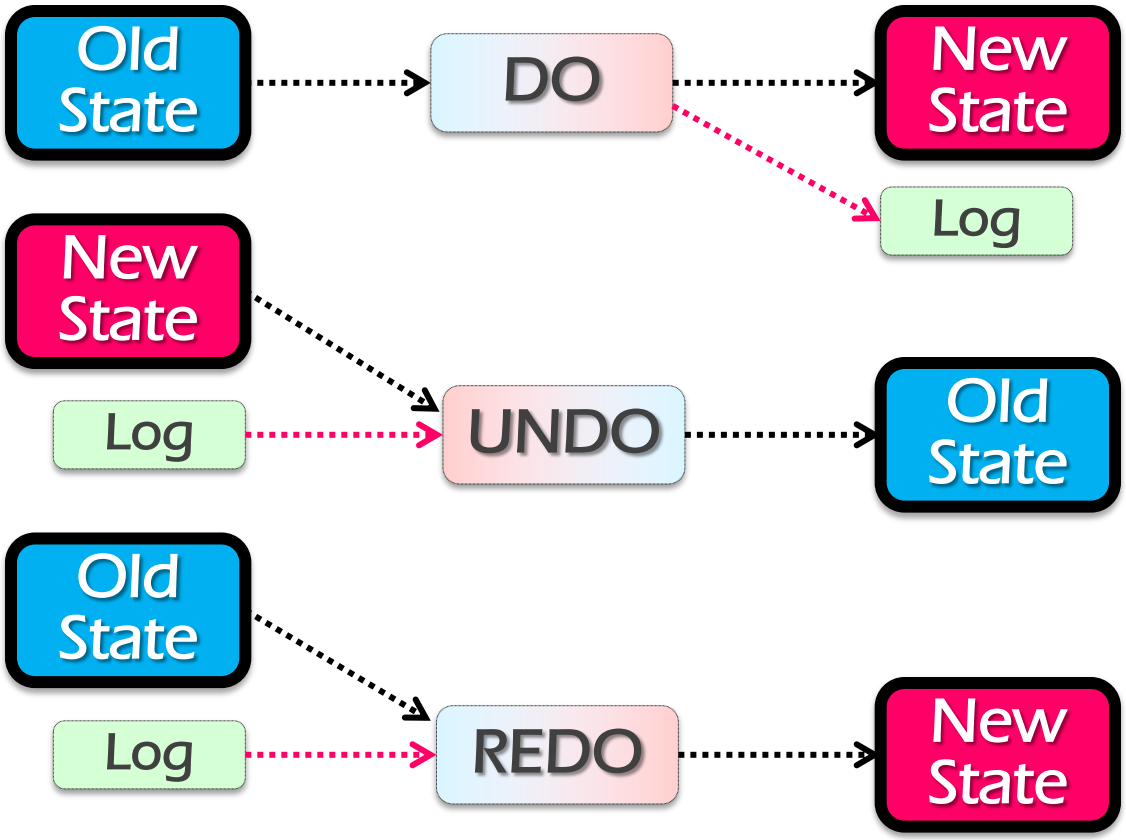
---



Solution: Logging

Keep a log of all update actions Log

Each action has 3 required operations



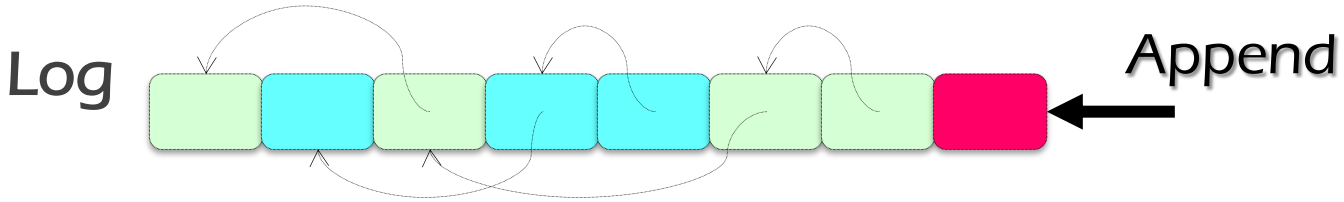
SysR: The Recovery Manager of the System R database Manger

Jim Gray et al, "*SysR: The Recovery Manager of the System R database Manger*". ACM Computing Surveys, 13(2),June **1981**

DO-UNDO-REDO Protocol

Merge **all** transactions into **one** log

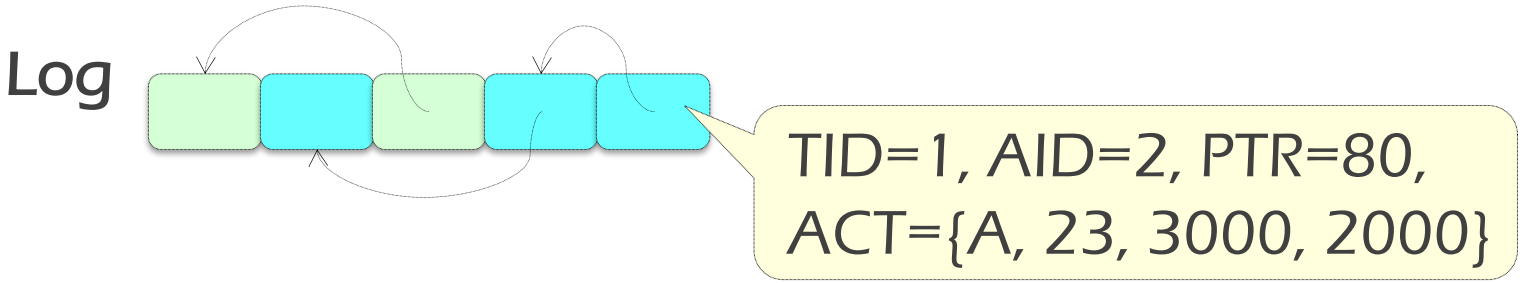
- Append-only → Reduce random access
- Require linked list of actions within on trans



DO-UNDO-REDO Protocol

Each log record consists of

1. Transaction ID
2. Action ID
3. Pointer to previous record in this transaction
4. Action (file name, record ID, old & new value)
5. …



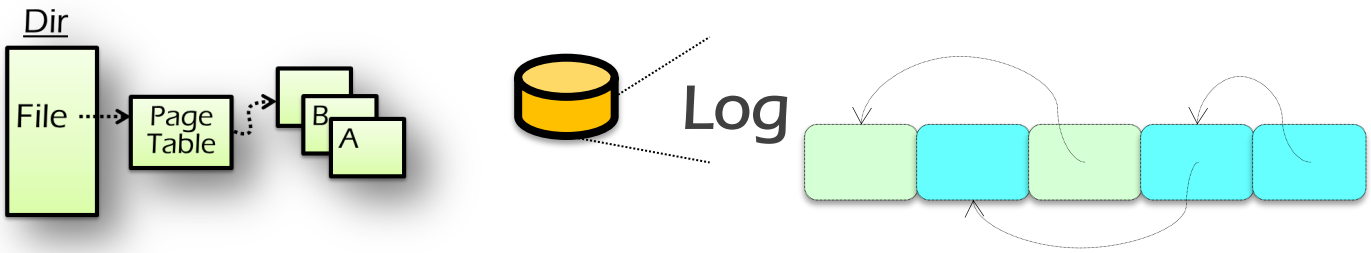
Logging and Recovery

How to **commit** a transaction?

Logging rules

- Write log record to disk before modifying persistent state (e.g., replace $F_{\mathbf{shdw}}$ with $F_{\mathbf{cur}}$)

  → Write Ahead Log (WAL) protocol



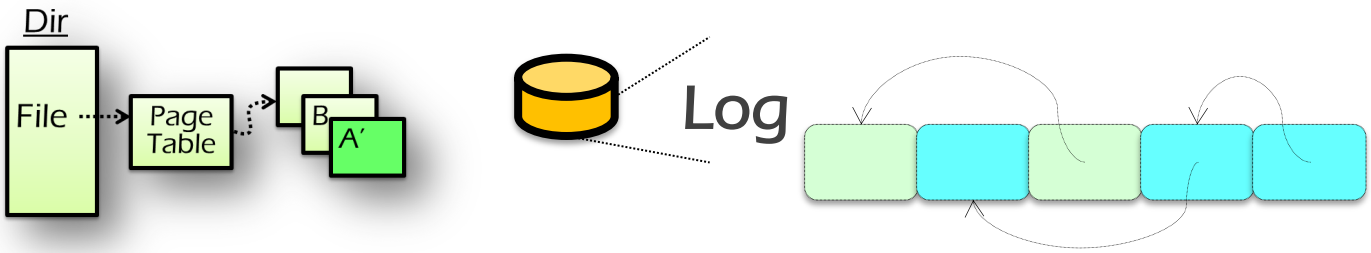
Logging and Recovery

How to **commit** a transaction?

Logging rules

- Write log record to disk before modifying persistent state (e.g., replace $F_{\mathbf{shdw}}$ with $F_{\mathbf{cur}}$)

  → Write Ahead Log (WAL) protocol

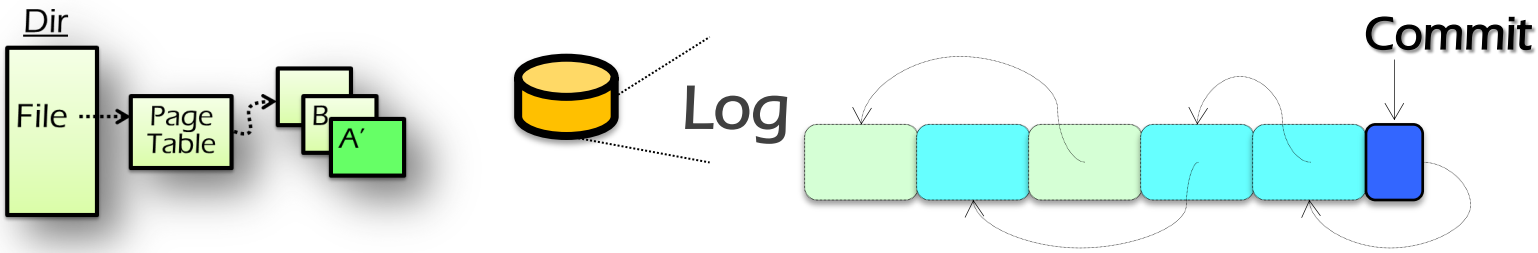


Logging and Recovery

How to **commit** a transaction?

Logging rules

- Write log record to disk before modifying persistent state (e.g., replace F_{shdw} with F_{cur})
  → Write Ahead Log (WAL) protocol
- At commit point, append a commit record to disk at last

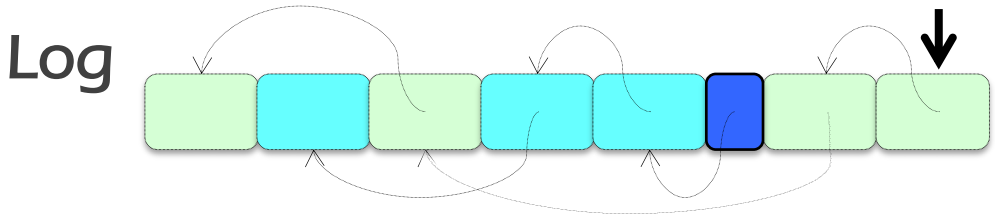


Logging and Recovery

How to **recovery** from a crash?

Recovery rules

- Travel from end to start



Logging and Recovery

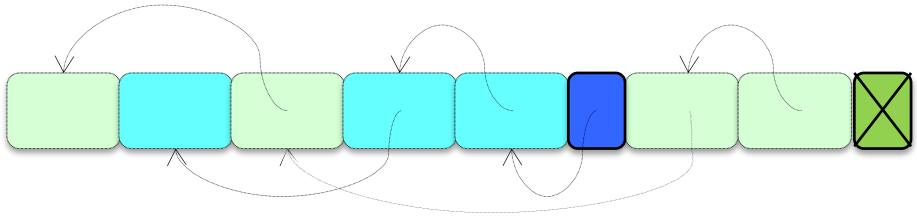
How to **recovery** from a crash?

Recovery rules

- Travel from end to start
- Mark all transaction's log record w/o **CMT** log and append **ABORT** log

---

Log



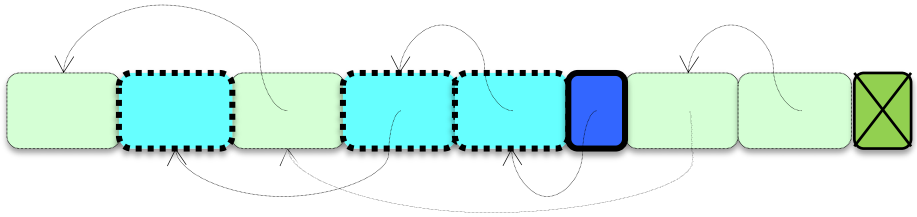
Logging and Recovery

How to **recovery** from a crash?

Recovery rules

- Travel from end to start
- Mark all transaction's log record w/o **CMT** log and append **ABORT** log
- Mark the rest transaction's log record

Log

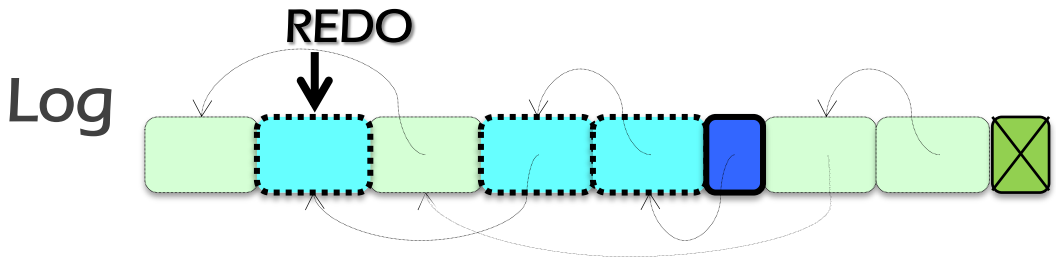


Logging and Recovery

How to **recovery** from a crash?

Recovery rules

- Travel from end to start
- Mark all transaction's log record w/o **CMT** log and append **ABORT** log
- Mark the rest transaction's log record
- **REDO** log records from start to end

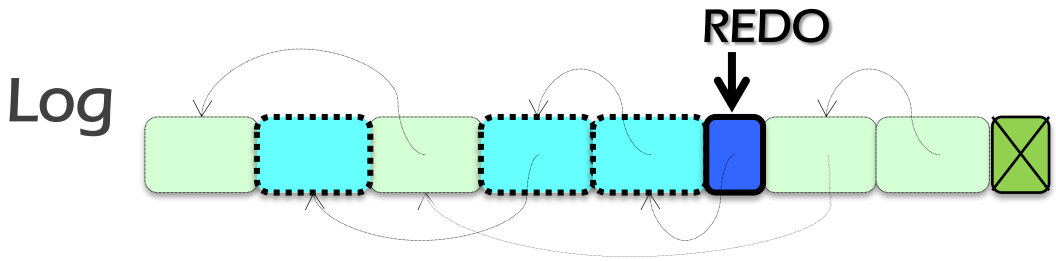


Logging and Recovery

How to **recovery** from a crash?

Recovery rules

- Travel from end to start
- Mark all transaction's log record w/o **CMT** log and append **ABORT** log
- Mark the rest transaction's log record
- **REDO** log records from start to end



Checkpoint

Why we need **checkpoint**?

Avoid aborting **long** transaction

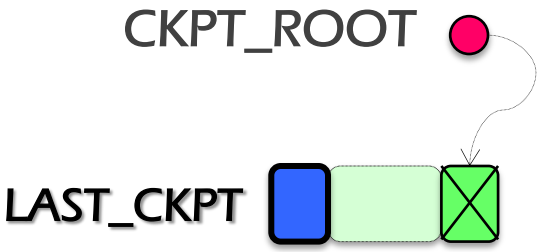
No need to recovery from a **blank** state

...

Checkpoint

How to checkpoint?

1. Wait till no transactions are in progress

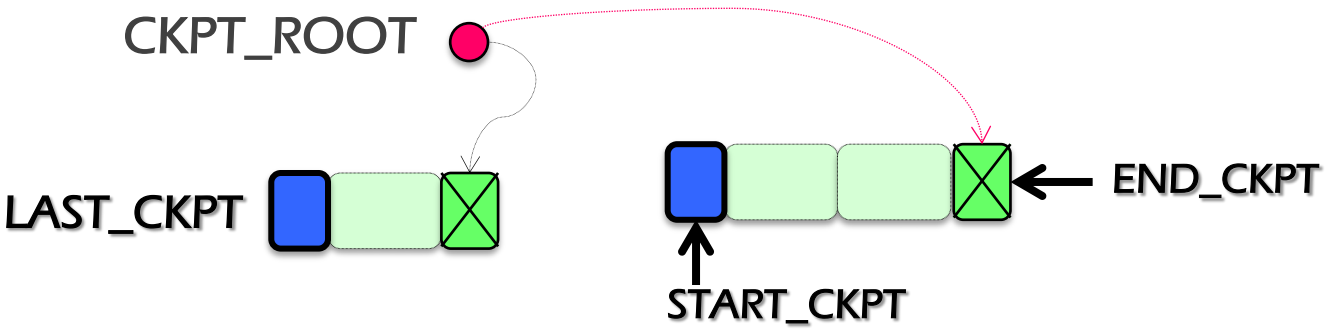


Checkpoint

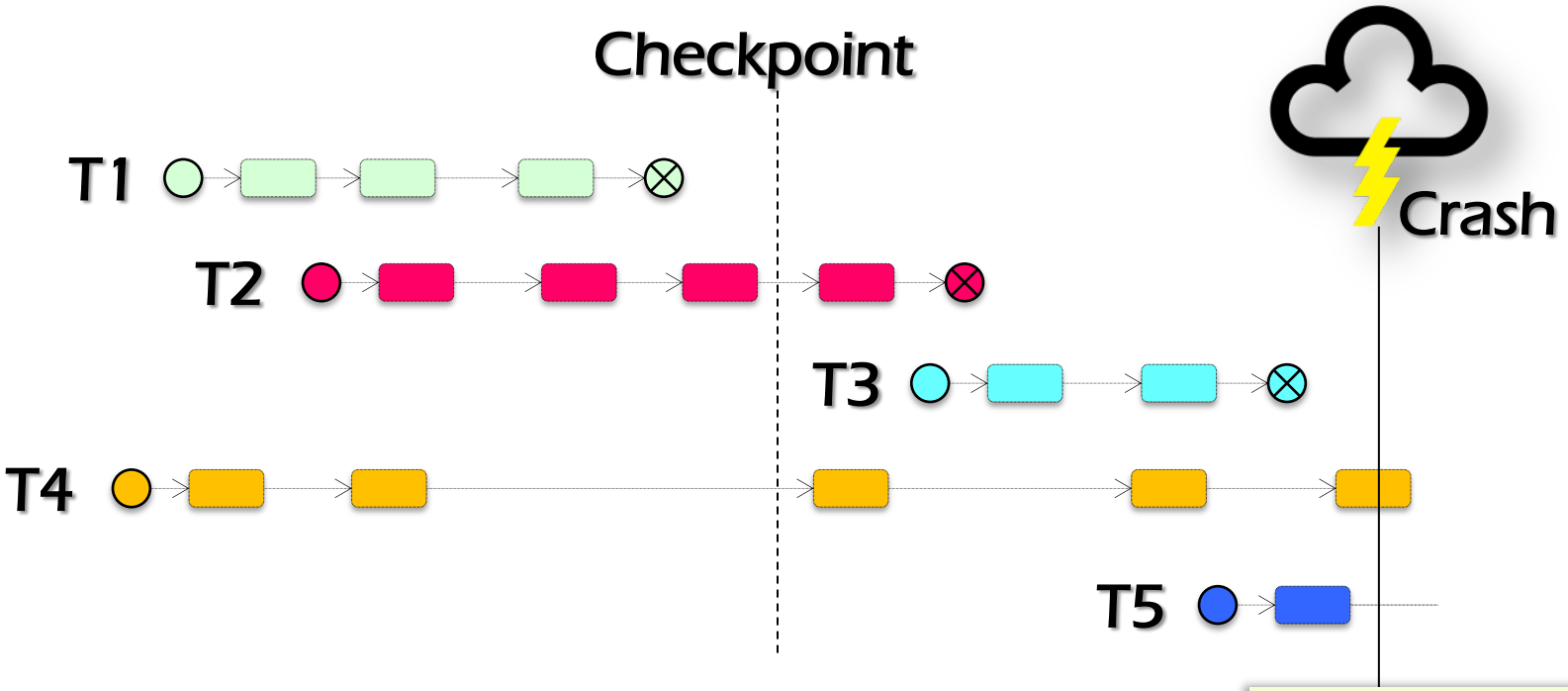
How to checkpoint? actions

1. Wait till no ~~transactions~~ are in progress
2. Write a **CKPT** record to log
   - Contains a list of all transaction in process and pointers to their most recent log records
3. Save all files (dirty cache → disk)
4. Atomically save checkpoint by updating checkpoint root to new **CKPT** record

---

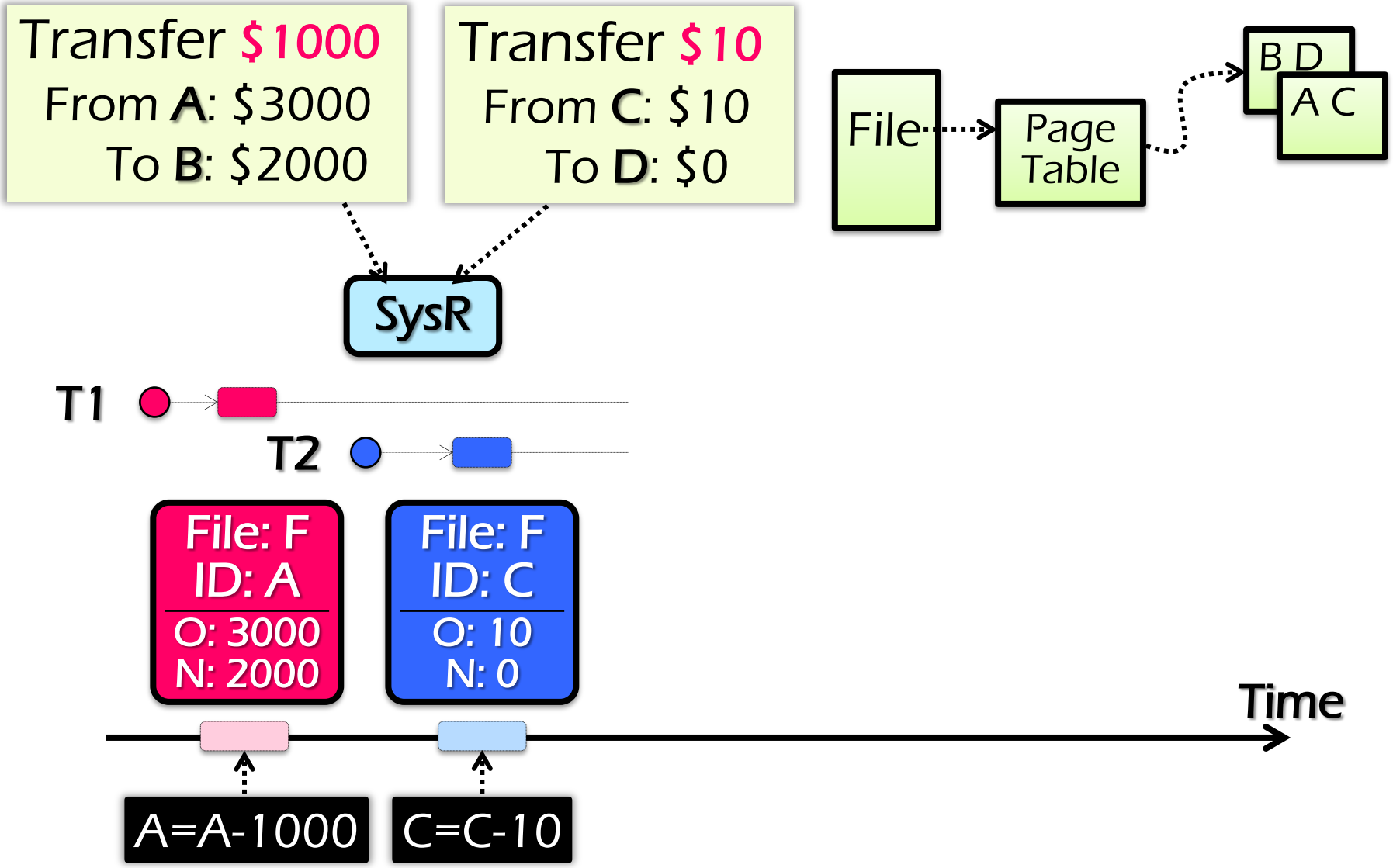


Recovery with Checkpoint

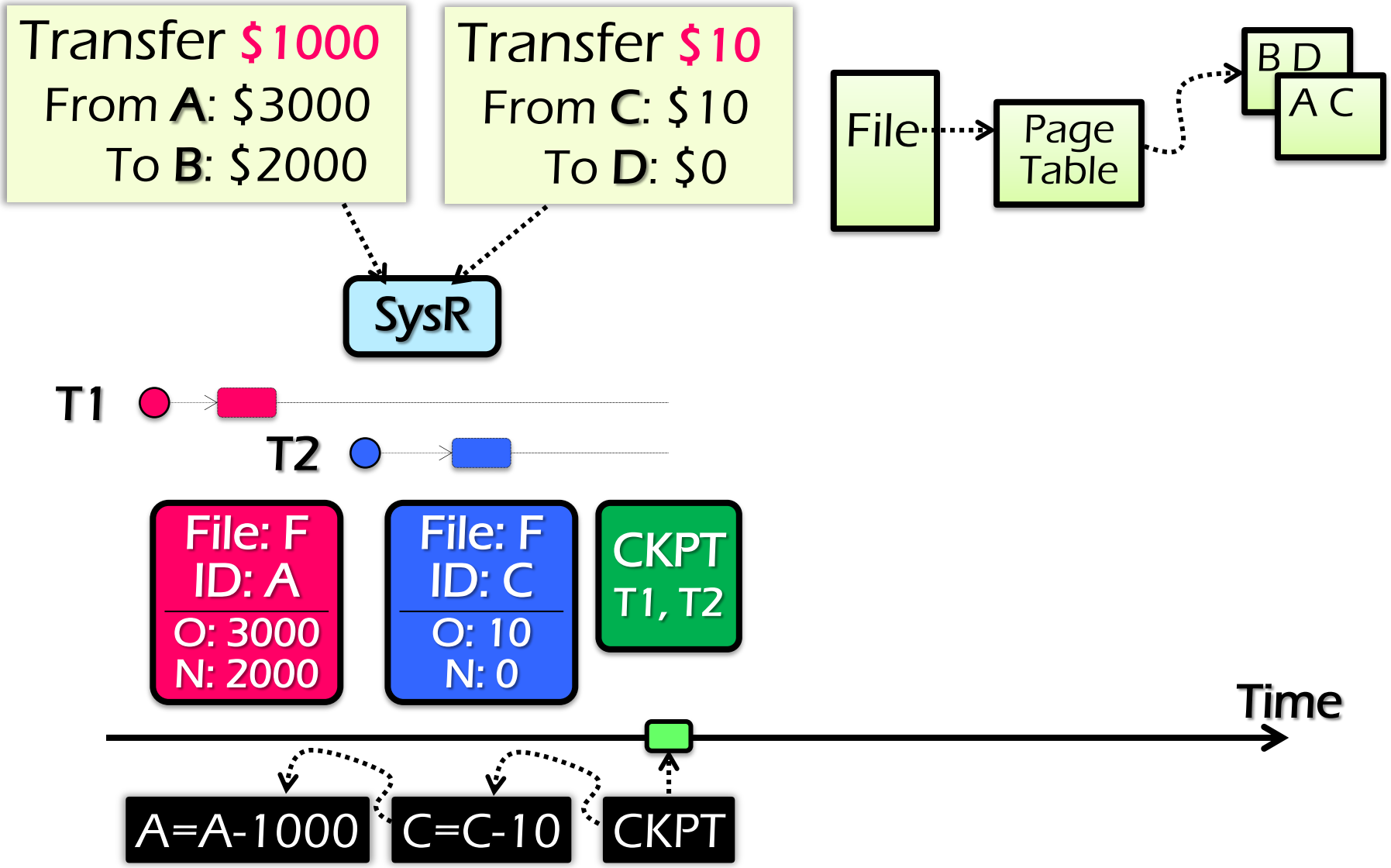


1. Read latest checkpoint
   → T2, T4 are ongoing transactions
2. Read log → T2, T3 are winners and T4 is a loser
3. Read log to **UNDO** loser (T4)
4. Read log to **REDO** winners (T2 and T3)

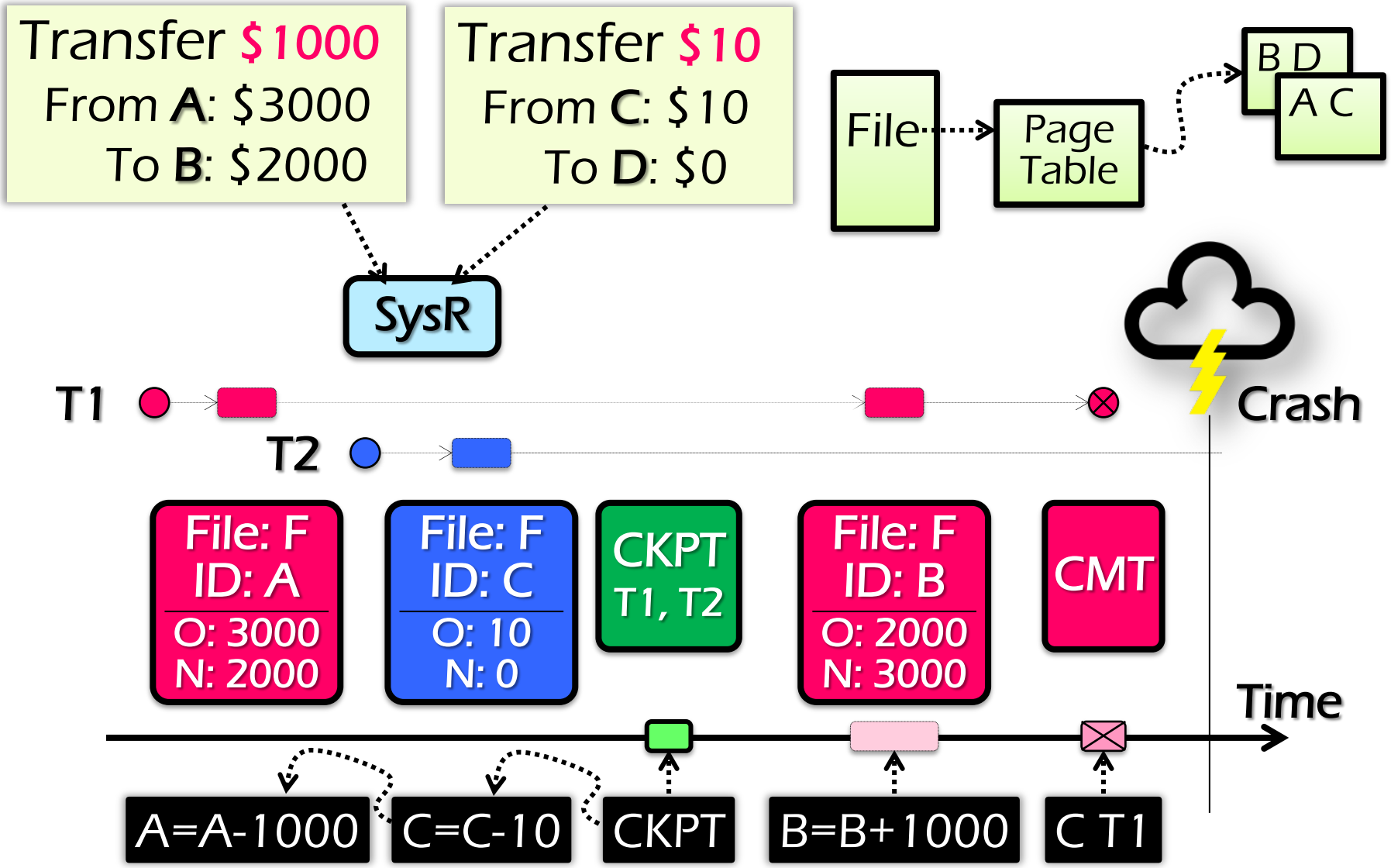
Example of Logging



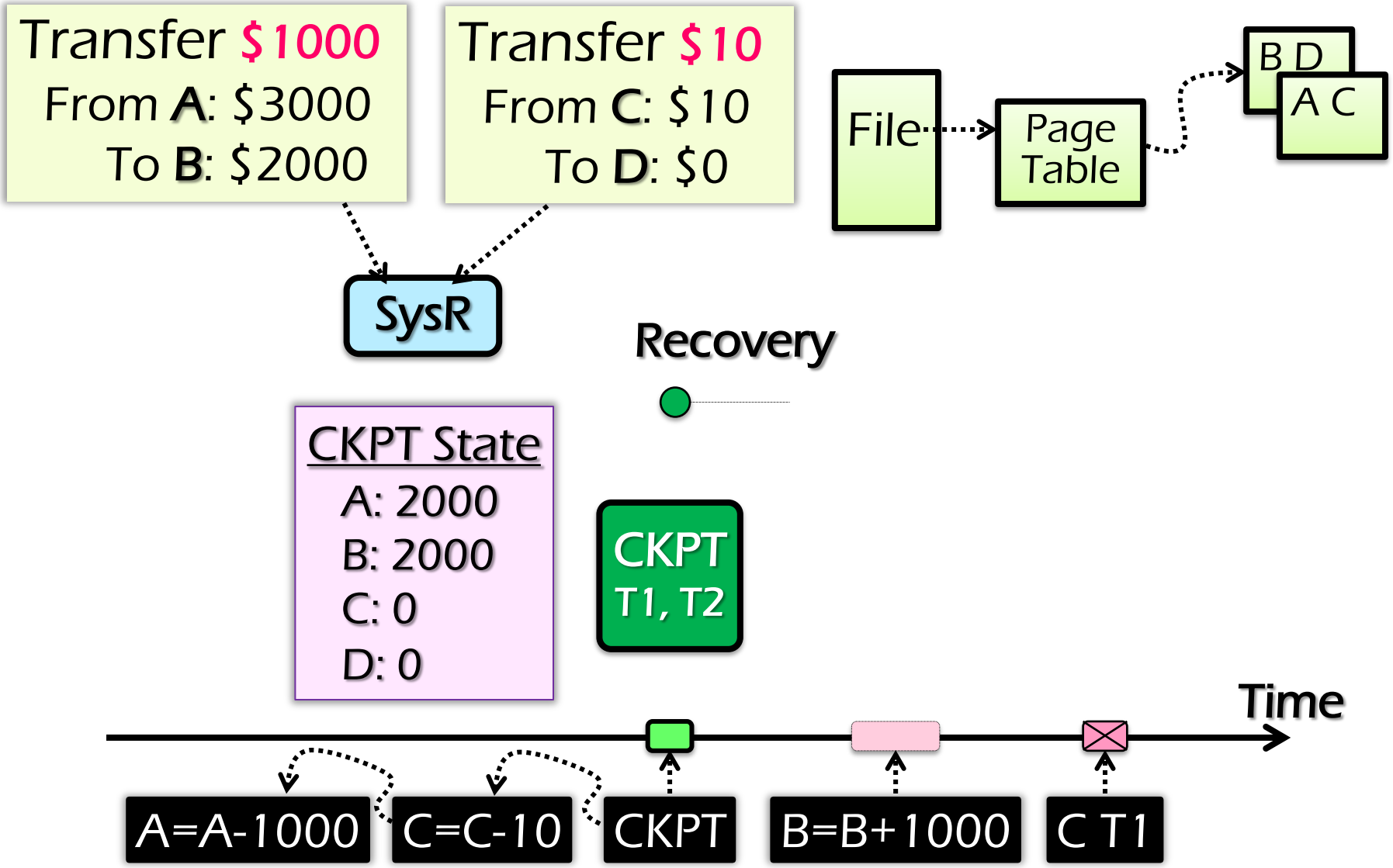
Example of Logging



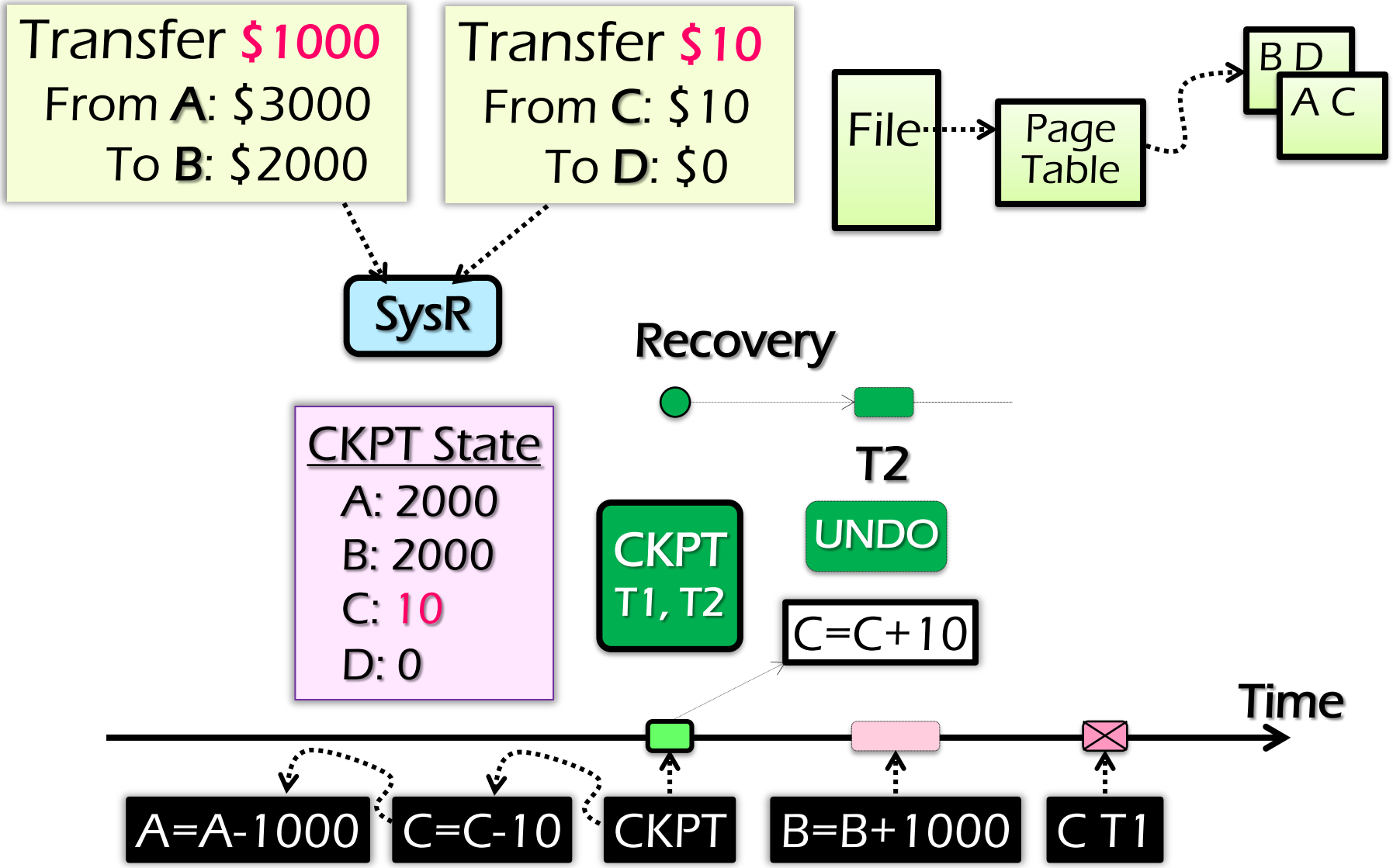
Example of Logging



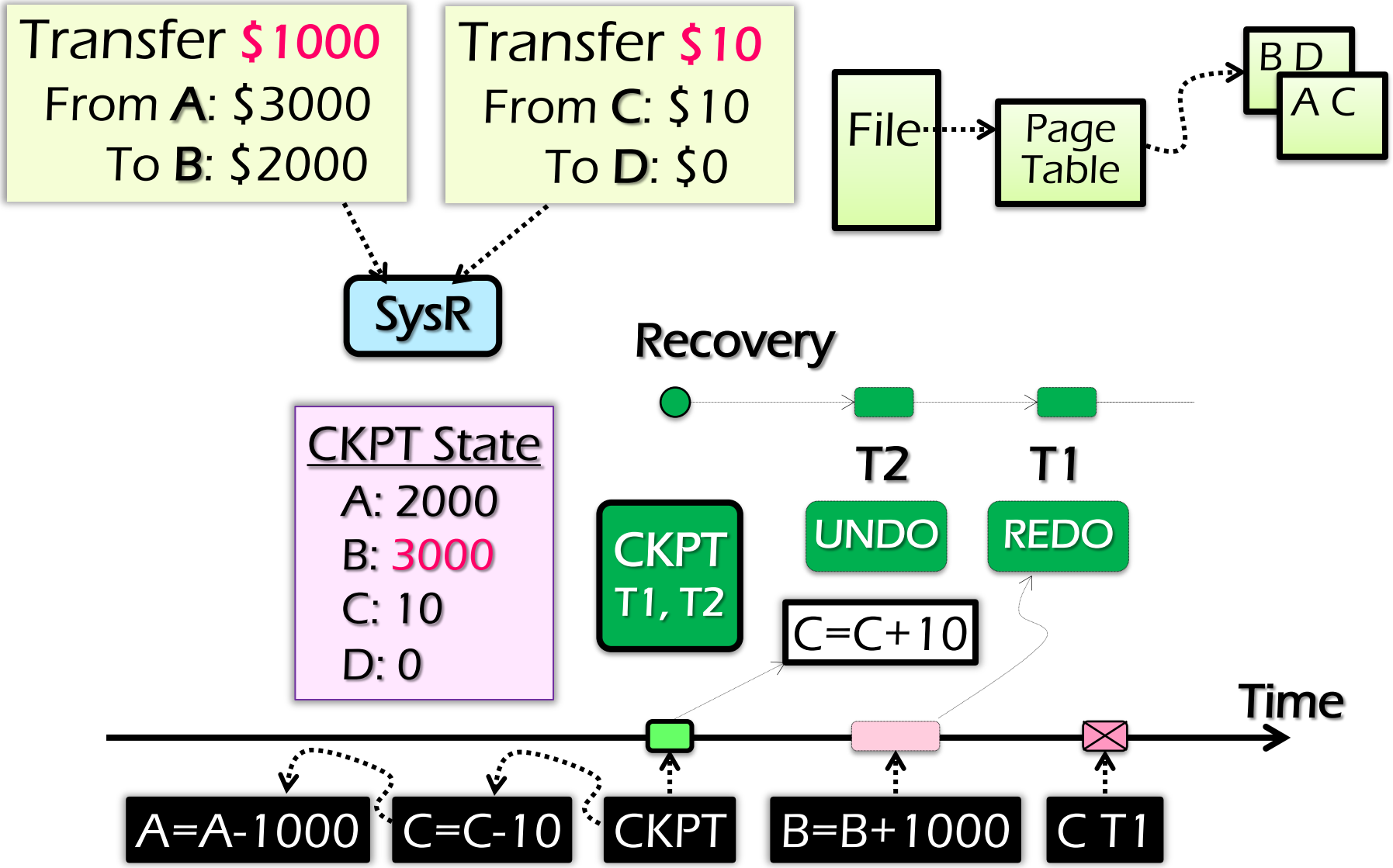
Example of Recovery



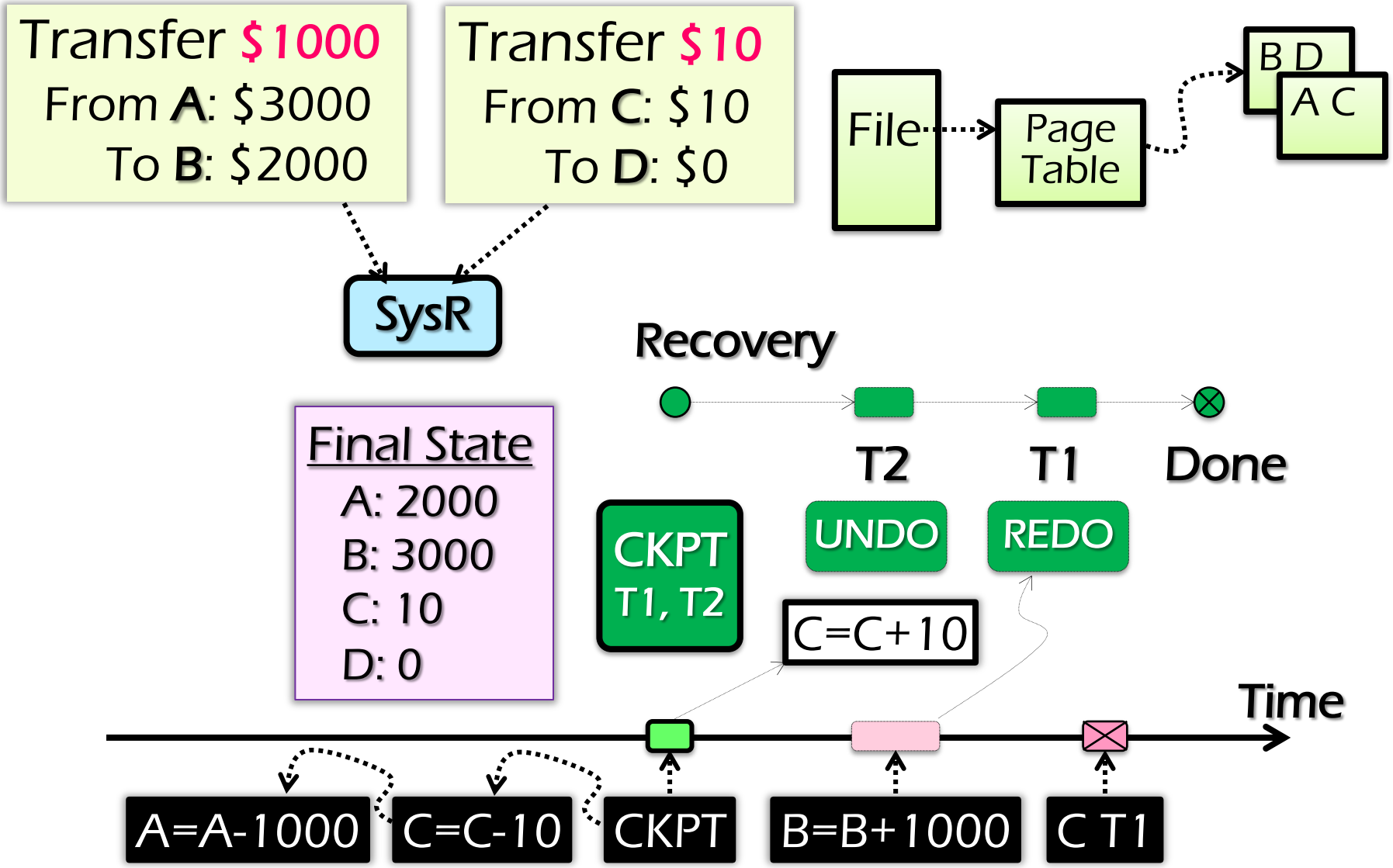
Example of Recovery



Example of Recovery



Example of Recovery



UNDO-REDO Logging

Why records both **UNDO** and **REDO** logs

- A transaction might be very long
  → Must checkpoint w/ ongoing transactions
- A long transaction might be aborted
  → Must UNDO abort state when recovery

Can we have **REDO-only** logs for system w/ "**short** transactions"?

REDO-only Logging

Logging rules

- Append **REDO** log record **before** flushing state modification
- Uncommitted transactions can not flush state (w/o **UNDO**)

How to logging

1. Append **REDO** log record
2. Write **CMT** log record
3. Flush state modification to disk (any time/order)

How to **recovery** from a crash?

REDO-only Logging

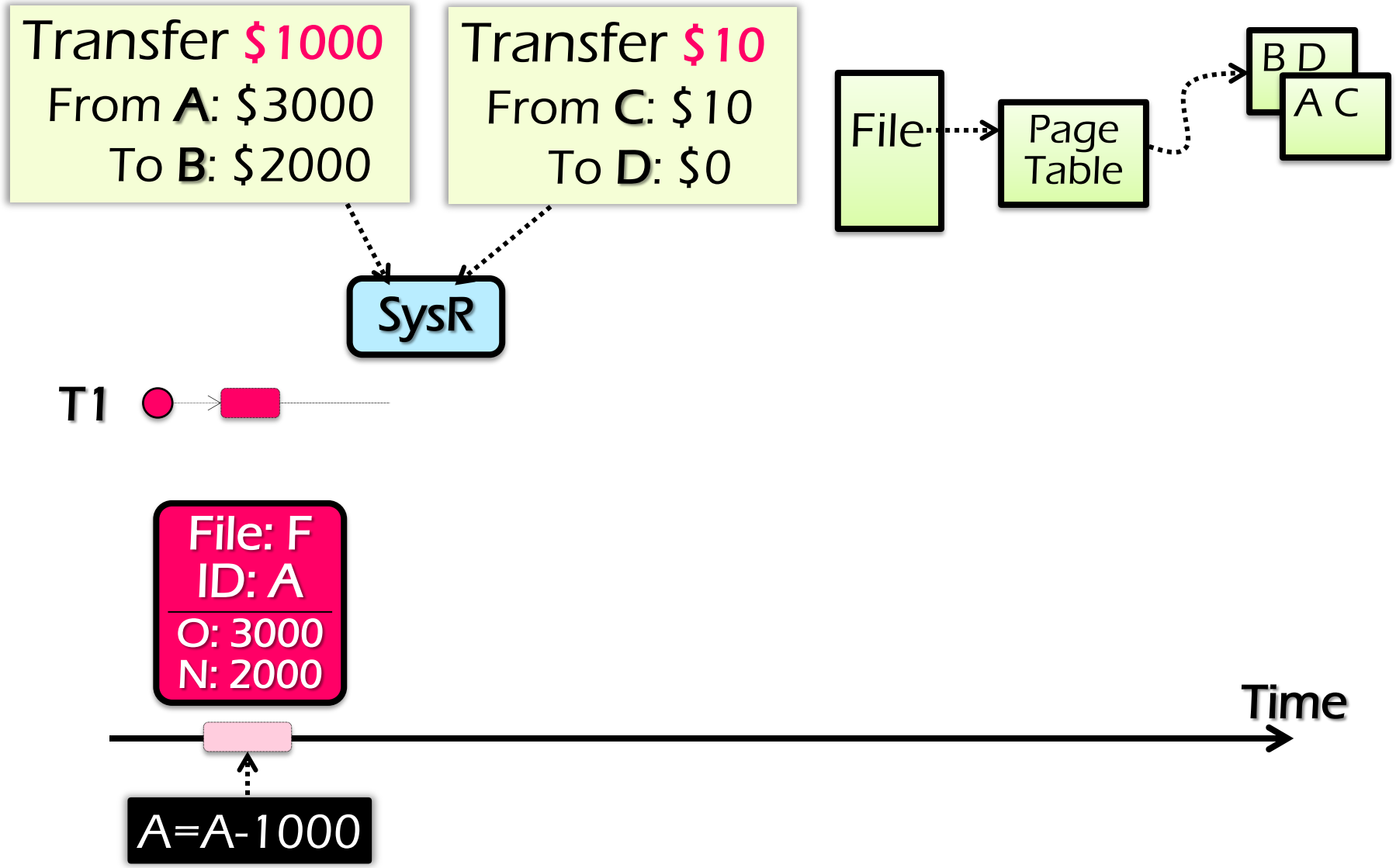
Recovery rules

- **ABORT** all log records w/o **CMT** log
- **REDO** all committed transactions *Why?*

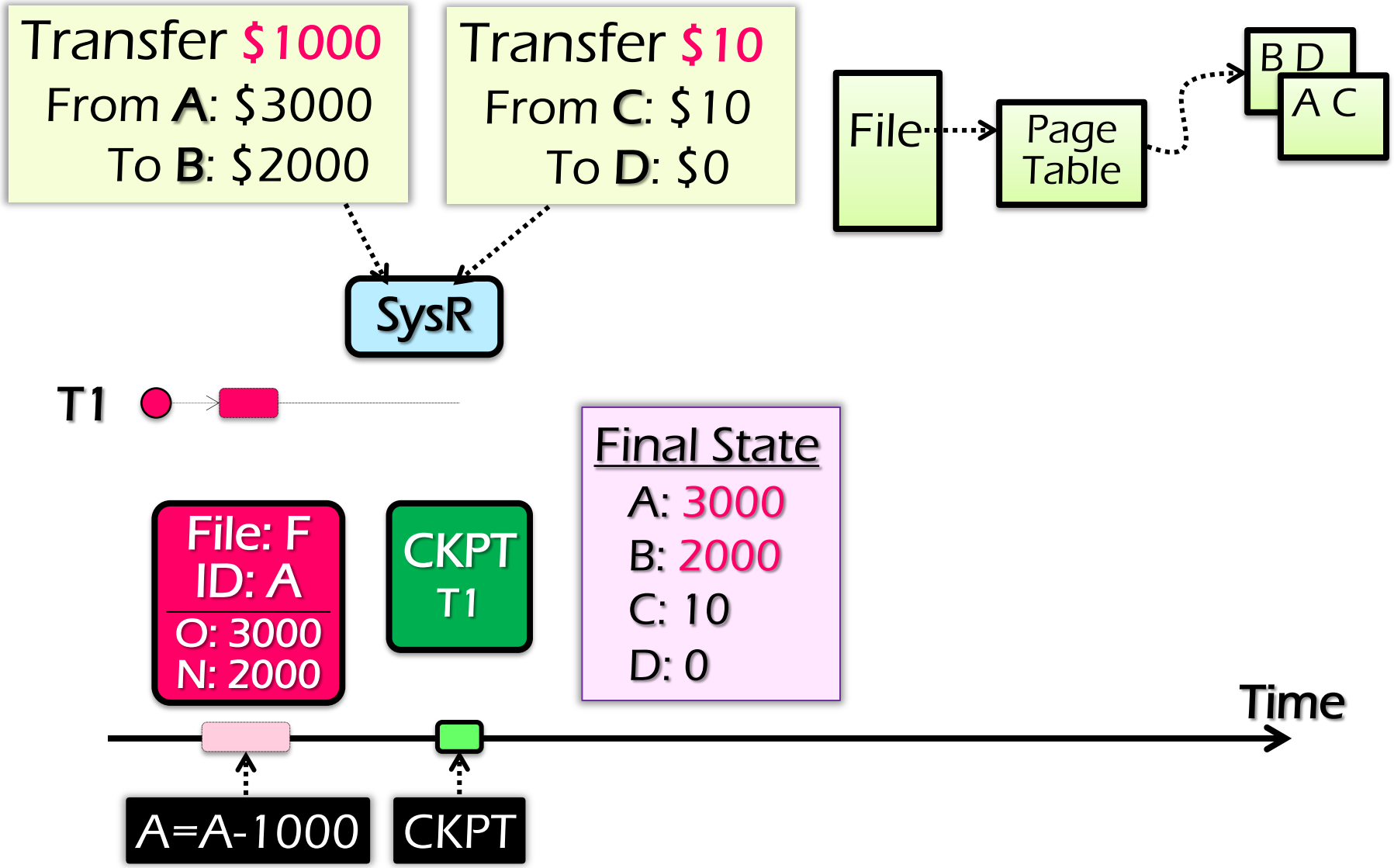
How to checkpoint?

1. Wait till no actions are in progress
2. Write a **CKPT** record to log
3. Save **committed** files
4. Atomically update checkpoint root to new **CKPT** record

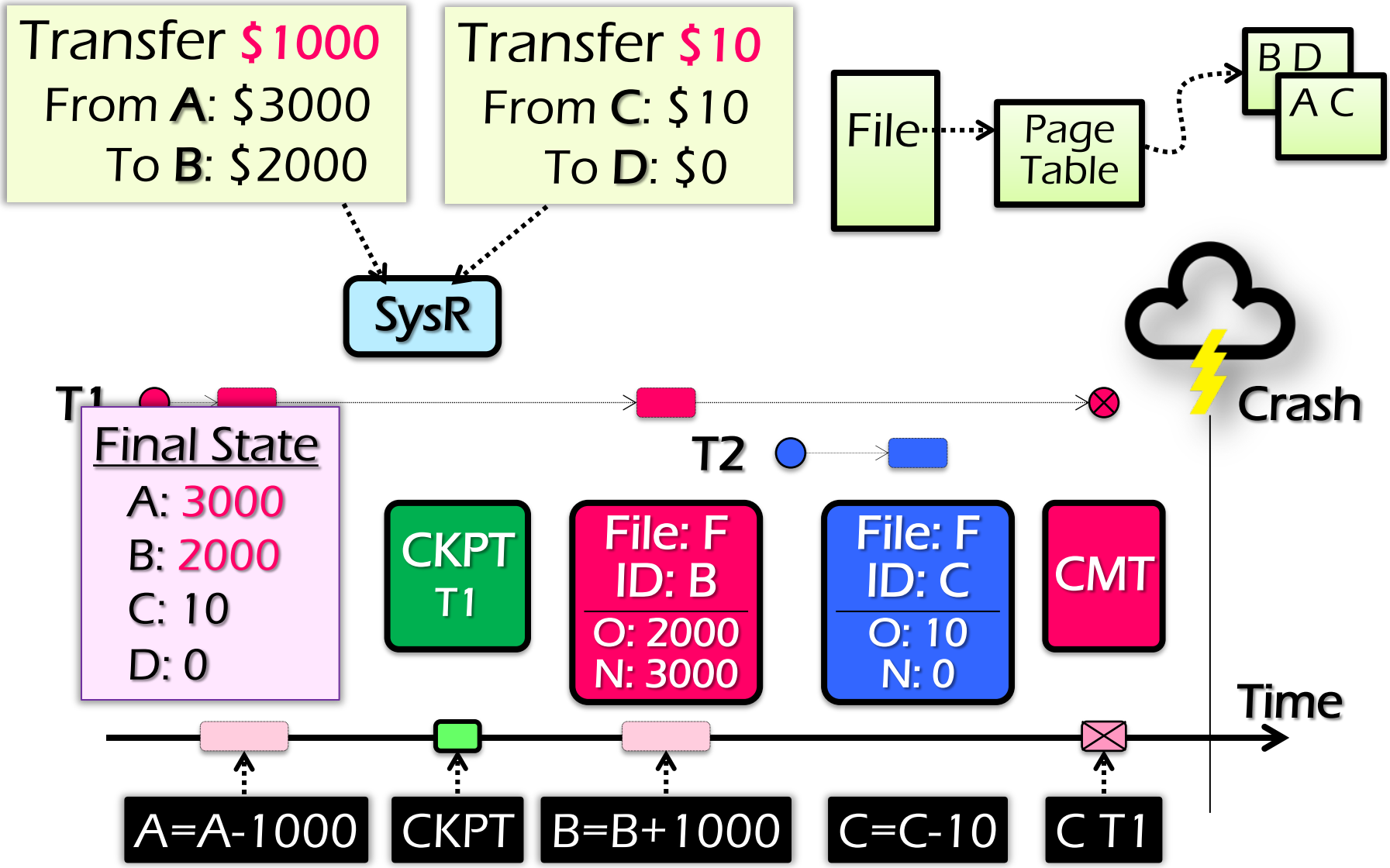
Example of Logging



Example of Logging



Example of Logging



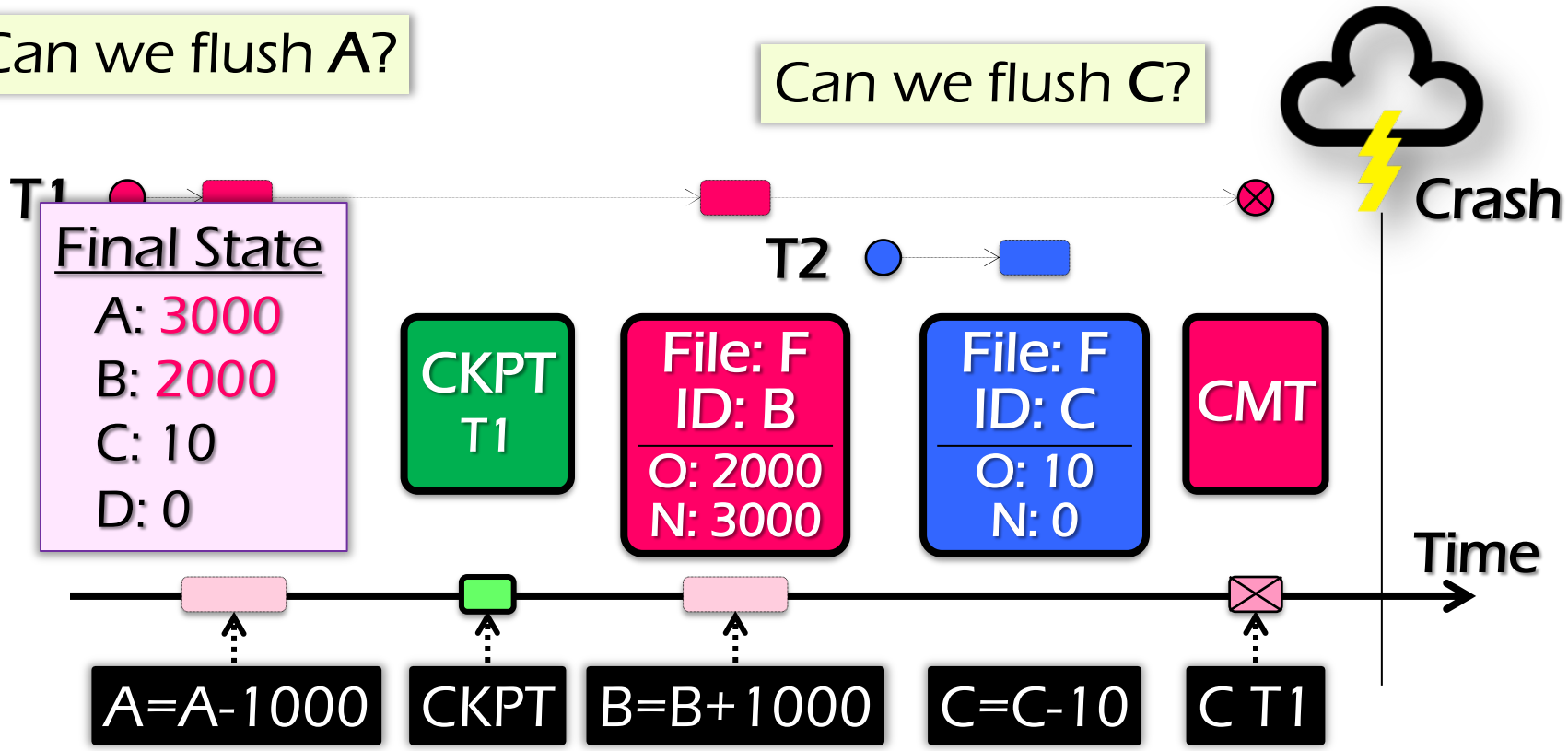
Example of Logging

State upon recovery?

What about flushing **order** of A, B and C?

Can we flush A?

Can we flush C?



UNDO-only Logging

Can we have **UNDO-only** logging?

Logging rules

- Append **UNDO** log record **before** flushing state modification
- State modification must be flushed before transaction committed (w/o **REDO**)

How to logging

1. Append **UNDO** log record
2. Flush state modification to disk (ASAP)
3. Write **CMT** log record

FSD: File System Using Logging and Group Commit

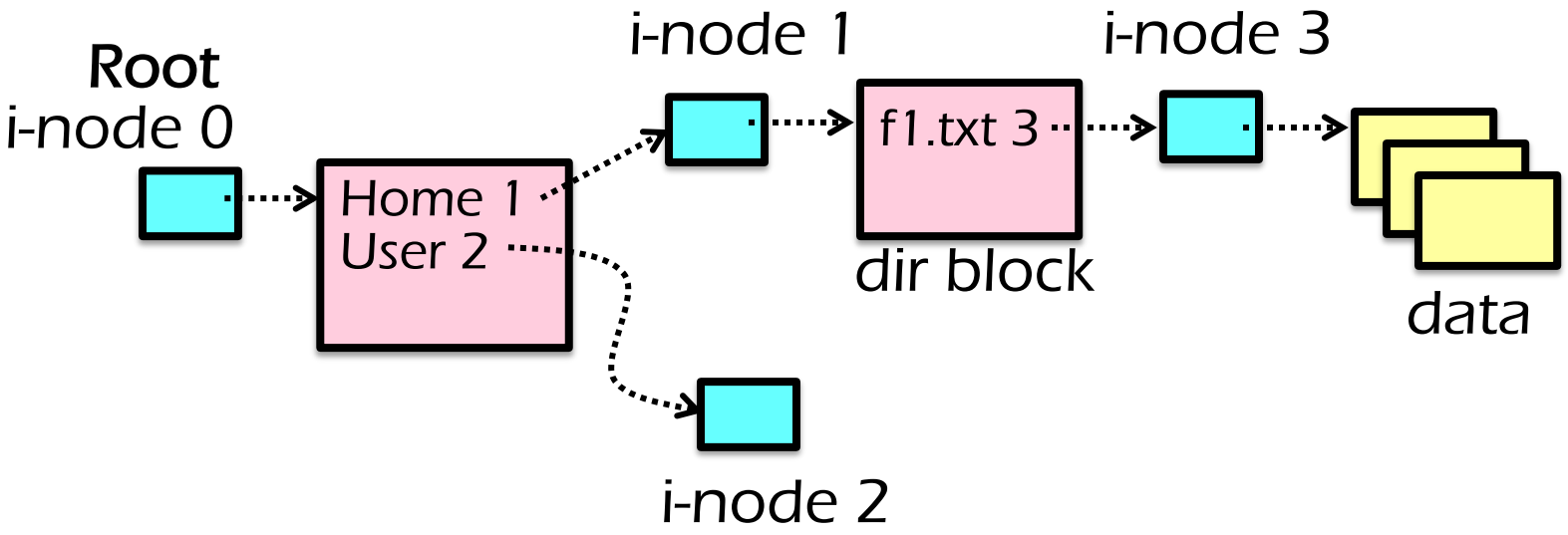
Robert Hagmann, "*Reimplementing the Cedar File System Using Logging and Group Commit*". SOSP, **1987**

File System is Complex

Meta-data: i-nodes and directory contents

Also need a bitmap for **free** i-nodes and a bitmap for **free** dir blocks

---



Kernel Cache

Buffer cache holds recently used blocks

Very effective for **read** operations

- e.g., access **root i-node** is extremely fast

Delay **write** operations

- Multiple operations can be **batched** to reduce disk I/O

*Dirty blocks are **lost** during crash!*

Crash Recovery is Hard

Dangers if crash during **meta-data** update

- File/directories disappear completely
- Files appear when they shouldn't
- Files have content belonging to different files

Dangers if crash during file **content** update

- Some writes are lost
- File content are a mix of old and new data

Goal of File System Recovery

Leave file system in a good state with respect to meta-data

- Recovery complex B-tree operation (e.g., splitting node during insert)

It's OK to lose a few operations

- To tradeoff for better performance during normal operation

Efficient recovery → no whole-disk scans

Example of Crash Problems


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

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

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

8. "d"'s dir content (remove "f")
9. "d"'s i-node (update len & mtime)
10. i-node bitmap (add a free)
11. block bitmap (set a free)

Write-back Cache

Synchronous writes → consistency

- If every write goes to disk → very slow!
- Writes be performed in a fix order
  → No localized
- Redundant write to same location

Write-back cache

- File system only write to cache
- When cache fills up with dirty blocks
  → flush some to disk

Can we recovery with write-back cache?

Example of Crash Problems

Write-back cache may write to disk in **any order**

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

Wrote 1-8

Wrote just 3

Wrote 1-7 and 10

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

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

8. "d"'s dir content (remove "f")
9. "d"'s i-node (update len & mtime)
10. i-node bitmap (add a free)
11. block bitmap (set a free)

A More Serious Crash

```
unlink("d/f1");
create("d/f2", 0666);
```

Create happens to be **re-use** i-node freed by **unlink**

Only 2nd write of "d"'s content goes to disk

3. *"d"'s dir content (add "f2" to i-node mapping)*

Recovery

- Nothing to fix, but file "f2" has "f1"'s content
- Serious undetected inconsistency & insecurity

Worst case: *a few dirty blocks are flushed, then crash*

All-or-nothing Meta-data Update

Why aren't synchronous meta-data updates enough?

- Too slow!
- Recovery may require checking/scanning the whole file system
- Some operations don't have an obvious single committing write

Logging File System

> Although logging may not be a cost-effective technique for the **data** of a file system, it is effective for the **meta-data**.
>
> – Needham, 1987

REDO-only log with Checkpoint

Which the meta-data need to be logged?

- File name table, leader pages

FSD: WAL + CKPT + REDO

What does a write do?

- Modify block in the data cache
- Append a **REDO** record to the log

What does a checkpoint do?

- All **committed** writes before **CKPT** must be stable on disk
- Flush a bunch of early writes, update **CKPT** root pointer

FSD: WAL + CKPT + REDO

When can we flush dirty data to disk?

- After commit log to disk

Why can't we use any of on-disk contents during recovery?

- May be partial
- **The REAL data is in the log!**

Guaranteed on disk	Might be on disk	Cannot be on disk
CKPT	Log on disk	End of in-memory log

FSD's Logging Structure

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 ½ second

FSD's Recovery

Recovery

- Scan log from **CKPT** log to see what transactions are committed
- Replay committed updates from **CKPT** onward

File Transaction

1. Write disk cache in memory
2. Append log records
3. Append commit log record
4. Wait for all its log records to reach disk
5. Write disk cache to disk
6. Update checkpoint

(4-6 in background)

Next Lecture

Topic: Concurrency

THANKS