# EEL-4736/5737 Principles of Computer System Design

Lecture 20

Textbook Chapter 9

Atomicity

# Introduction

- Atomicity – principle widely used in the design of systems
  - Coordination of concurrent activities
    - Before-or-after atomicity
  - Handling failures, exceptions
    - All-or-nothing atomicity
- We will explore design approaches to support atomicity

# Example

- Use Web browser to enter your credit card information and submit request to buy an item from online store
  - Power fails during the request; you would expect either the full order to be processed, or to fail
    - Credit card charged but item not shipped, or item shipped without credit card charged would not be acceptable
  - If only one item in stock and two customers concurrently hit purchase, one successfully completes the purchase ("before") while the other gets an error message ("after")
    - No failure; before-or-after ordering needed to avoid committing one item to two buyers

# Atomicity

- Perform a sequence of steps so they appear to be done as a single, indivisible step
  - Either entire sequence fails or succeeds – not in the middle of a sequence
  - When several atomic operations are ongoing, each appears to take place completely before or after another
- Another key foundation for modularity

# Sweeping simplification

- Systems as state machines
  - Fewer possible states of a system to consider
- Simplifies design and understanding for later developers
  - User of an atomic operation does not need to concern about handling ‘mid’ failures
- Simplifies verification of correctness
- Atomicity provides a basis for both failure recovery and for sequence coordination

# Example - database

Let's consider a simple transfer of funds transaction

Here, GET/PUT read/write from a database

1. **procedure** TRANSFER (*debit_account, credit_account, amount*)
2. GET (*dbdata, debit_account*)
3. *dbdata* ← *dbdata - amount*
4. PUT (*dbdata, debit_account*)
5. GET (*crdata, credit_account*)
6. *crdata* ← *crdata* + *amount*
7. PUT (*crdata, credit_account*)

# Example

1. **procedure** TRANSFER (*debit_account, credit_account, amount*)
2. GET (*dbdata, debit_account*)
3. *dbdata* ← *dbdata - amount*
4. PUT (*dbdata, debit_account*)
5. GET (*crdata, credit_account*)
6. *crdata* ← *crdata + amount*
7. PUT (*crdata, credit_account*)

Thread fails during 4 – if PUT not all-or-nothing, may store partial, corrupted data on disk

Thread fails between 4 and 7 – one account debited, but no account credited

# Example

**1. procedure** TRANSFER (*debit_account, credit_account, amount*)

2. GET (*dbdata, debit_account*)
3. *dbdata* ← *dbdata* - *amount*
4. PUT (*dbdata, debit_account*)
5. GET (*crdata, credit_account*)
6. *crdata* ← *crdata* + *amount*
7. PUT (*crdata, credit_account*)

Two concurrent account transfers:

TRANSFER(A,B,100)

TRANSFER(B,C,200)

Suppose initially, A=500, B=200, C=1000

Would like to see A=400, B=100, C=1200, but

# Example

1. **procedure** TRANSFER (*debit_account, credit_account, amount*)
2. GET (*dbdata, debit_account*)
3. *dbdata* ← *dbdata - amount*
4. PUT (*dbdata, debit_account*)
5. GET (*crdata, credit_account*)
6. *crdata* ← *crdata + amount*
7. PUT (*crdata, credit_account*)

| | |
|---|---|
| T(A,B,100): | GET (crdata, B) <- 200 |
| | crdata <- 200+100 = 300 |
| T(B,C,200): | GET (dbdata, B) <- 200 |
| | dbdata <- 200-200 = 0 |
| T(A,B,100): | PUT(300, B) |
| T(B,C,200): | PUT(0,B) |

# Example – processor ISA

- Complex instructions with multiple steps
  - E.g. x86 MOVS
    - Copy multiple continuous bytes (a string) from a source to a destination in memory
    - Index register for src, dst; counter
  - If an interrupt arises while in execution, e.g. a page fault, and later the instruction is to be restarted, where did the copy stop?
- Out of order execution pipelines
  - Multiple instructions in flight when an interrupt or exception occurs

# All-or-nothing

- A sequence of steps is an all-or-nothing action if, from the point of view of its invoker, the sequence always either
  - completes, or,
  - aborts in such a way that it appears that the sequence had never been undertaken in the first place
- In a layered application, if each action at a lower layer is all-or-nothing, upper layer does not need to concern about intermediate steps taken by the action

# Atomicity

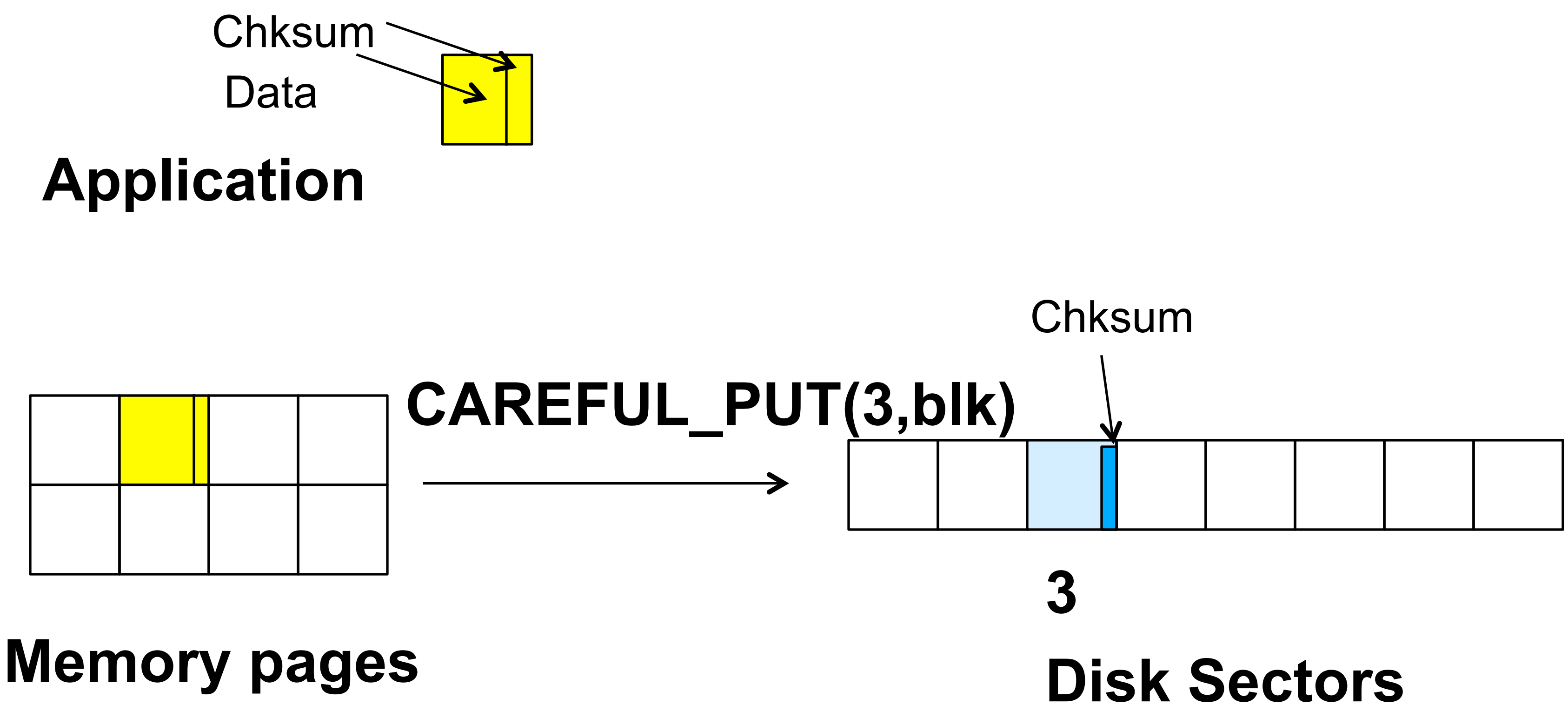
- Hiding internal structure is key to modularity
- For an atomic action, there is no way for a higher layer to discover the internal structure of its implementation
  – Point of view of a procedure invoker, action either completes or not; internal steps are not visible
  – Point of view of concurrent threads, action completes either before or after any other

# All-or-nothing atomicity

- Recall CAREFUL_PUT in chapter 8
  - If write interrupted by power loss, possible to detect with sector checksum
  - If O/S buffer corrupted during a write, need to use application-layer checksum to detect error on next read
  - Still, end up with a corrupted sector
- With all-or-nothing atomicity, upon restart, can recover either old or new value

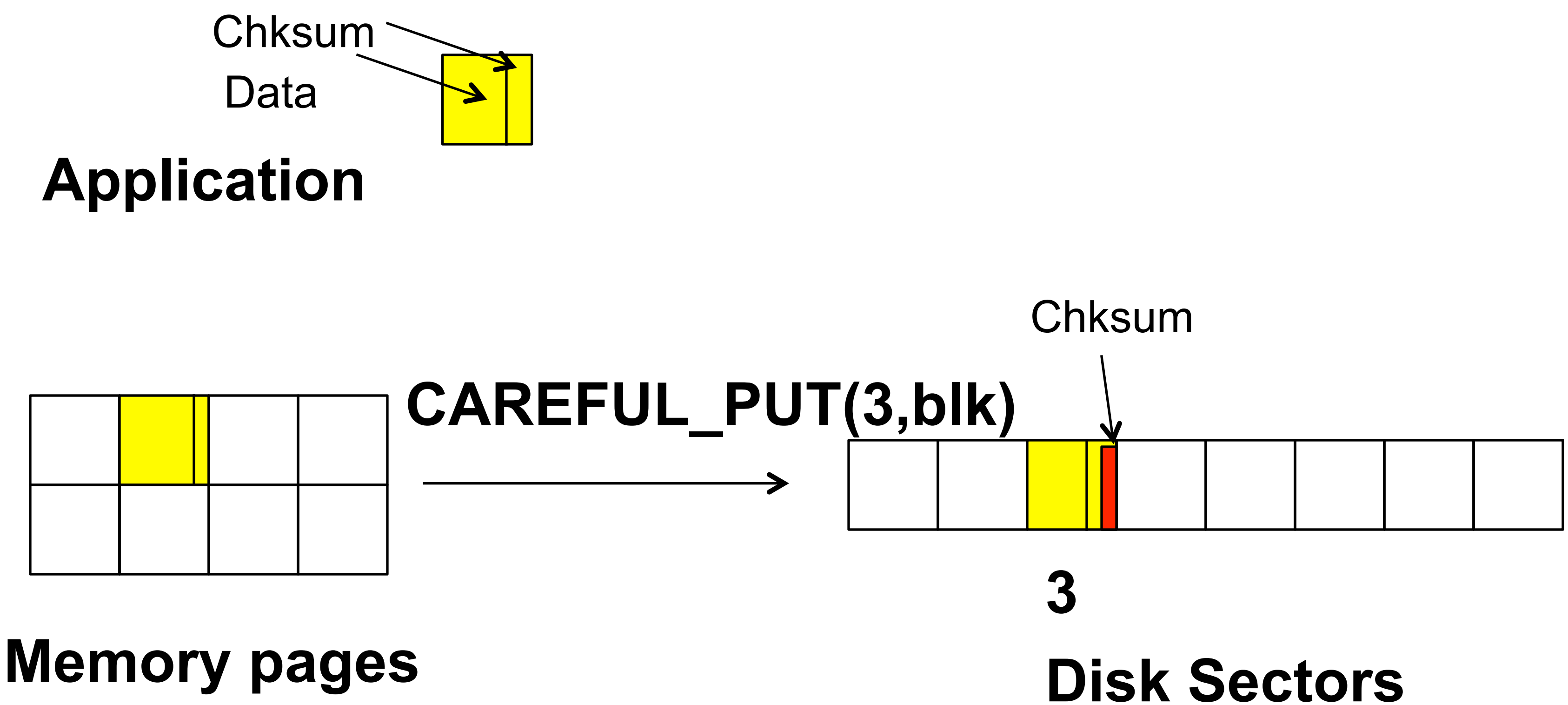
# Example

- Initial state



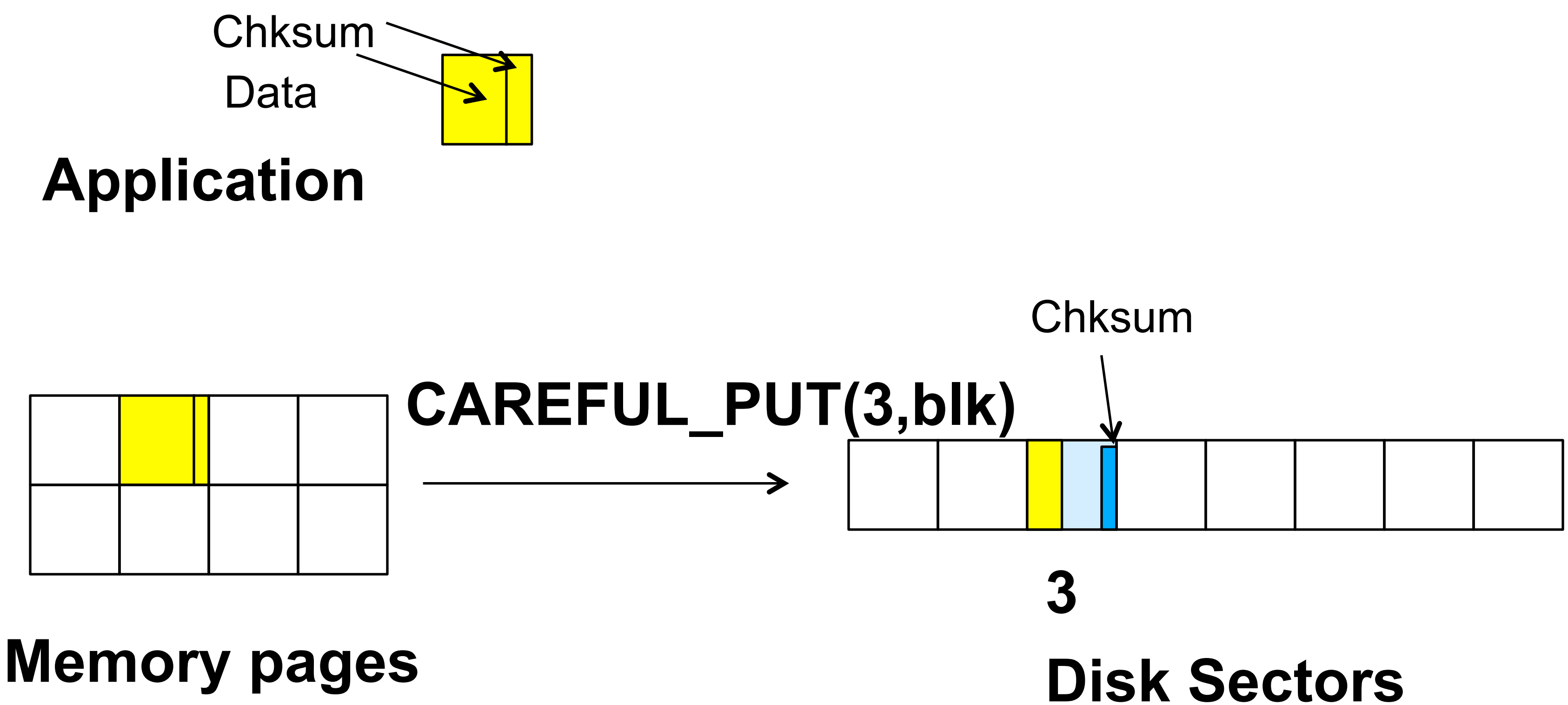
# Example

- Correct operation



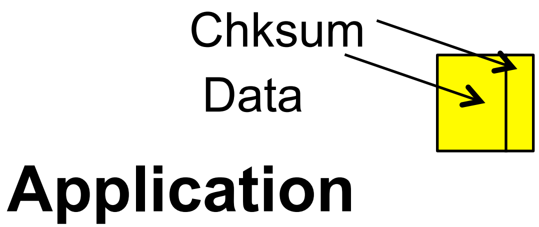
# Example

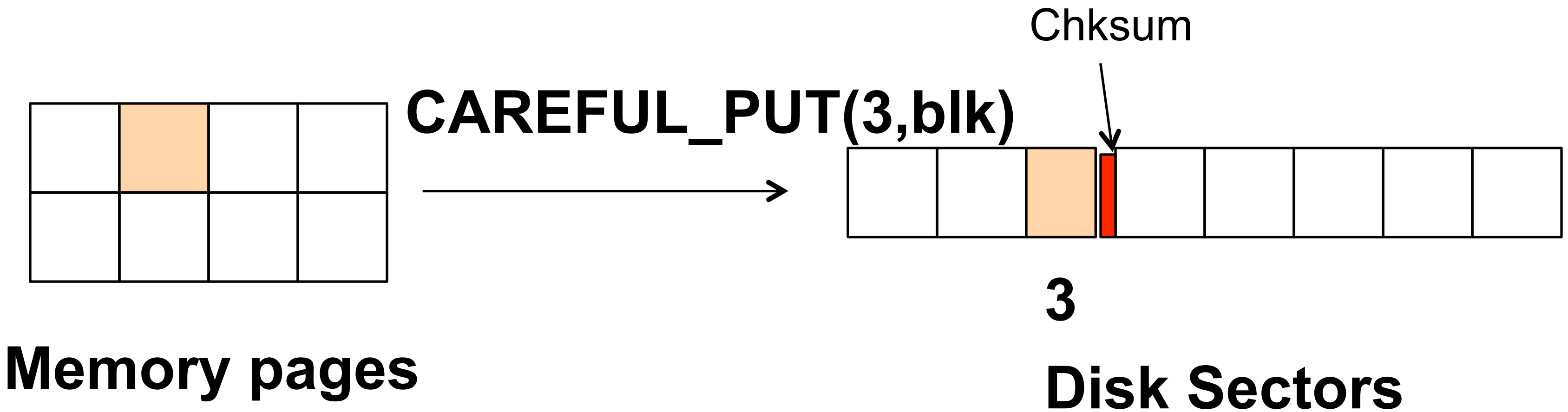
- Power failure



# Example

- Error corrupts data in memory during CAREFUL_PUT



# Error recovery

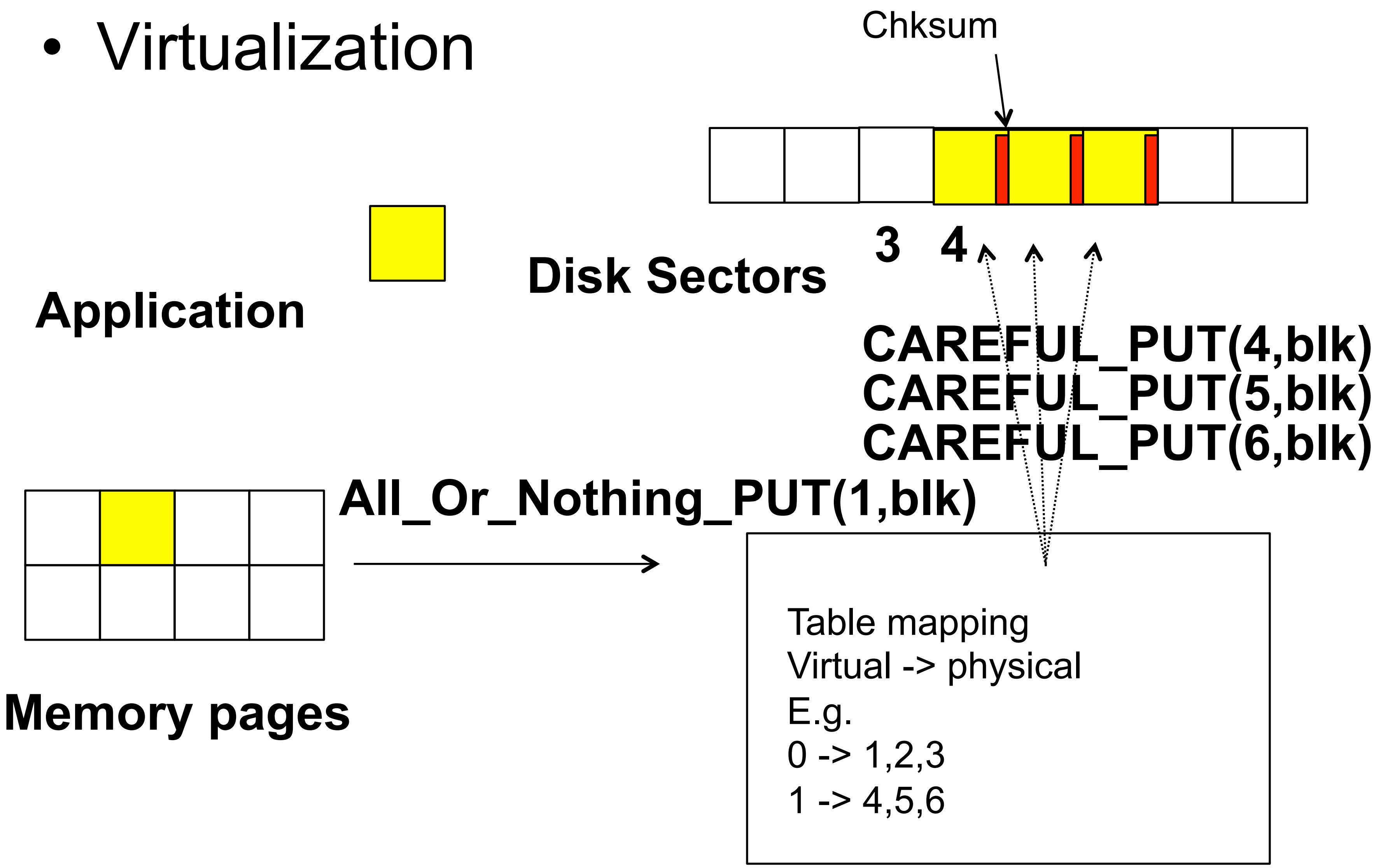
- Adding an application-layer checksum can help in detecting errors
- Would like to be able to recover with all-or-nothing atomicity
  – Apply replication
    - “Don’t modify the only copy”
- Start by addressing single-cell – i.e. updates a single sector
  – We’ll generalize to more complex actions

# All-or-nothing atomicity

- Assume at most one sector with error
- The error can be detected by checksum during a later CAREFUL_GET
- General approach:
  - Virtualize sector – one-to-many mapping
    - Write multiple copies
    - Prevents error in physical cell (sector) from corrupting a virtualized view of cell (sector)
  - Check and repair on access

# Example

- Virtualization



# All-or-nothing atomicity

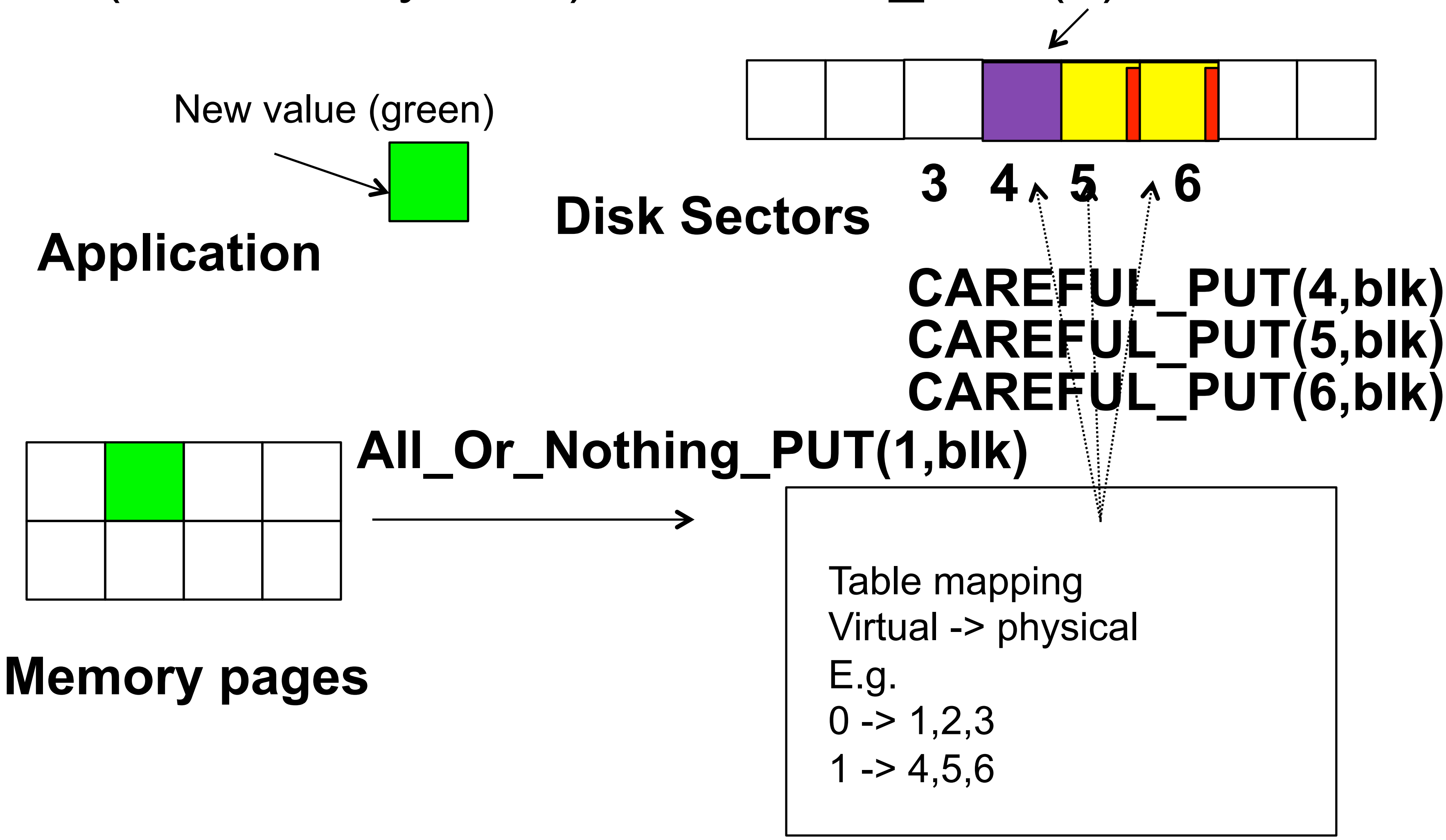
- First iteration:

```
procedure ALMOST_ALL_OR_NOTHING_PUT (data,
   all_or_nothing_sector)
    CAREFUL_PUT (data, all_or_nothing_sector.S1)
    CAREFUL_PUT (data, all_or_nothing_sector.S2)
    CAREFUL_PUT (data, all_or_nothing_sector.S3)

procedure ALL_OR_NOTHING_GET (reference data,
   all_or_nothing_sector)
    CAREFUL_GET (data1, all_or_nothing_sector.S1)
    CAREFUL_GET (data2, all_or_nothing_sector.S2)
    CAREFUL_GET (data3, all_or_nothing_sector.S3)
    if data1 = data2 then data ← data1
      else data ← data3
```

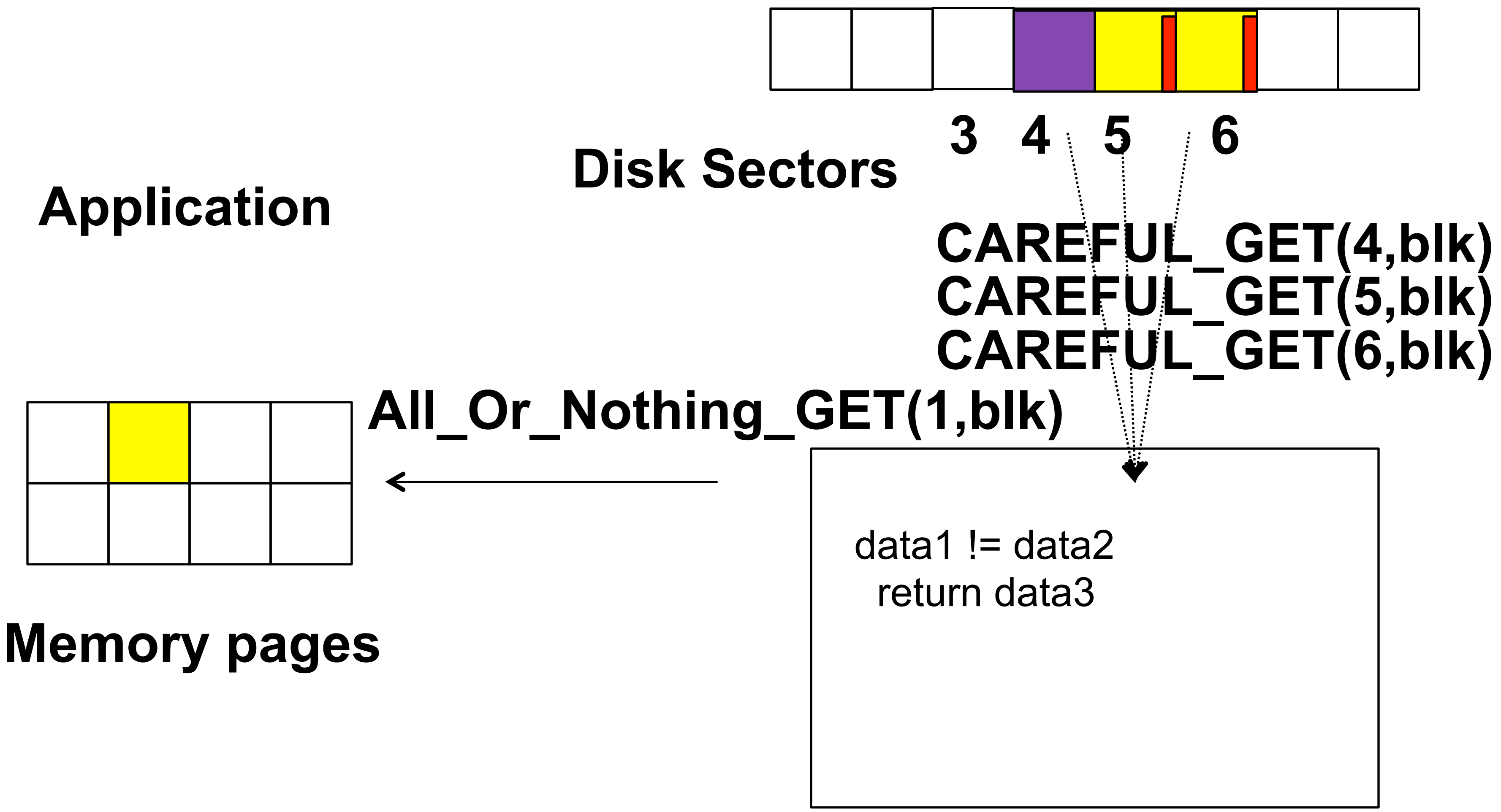
# Example

- (Old value yellow) CAREFUL_PUT(4) fails



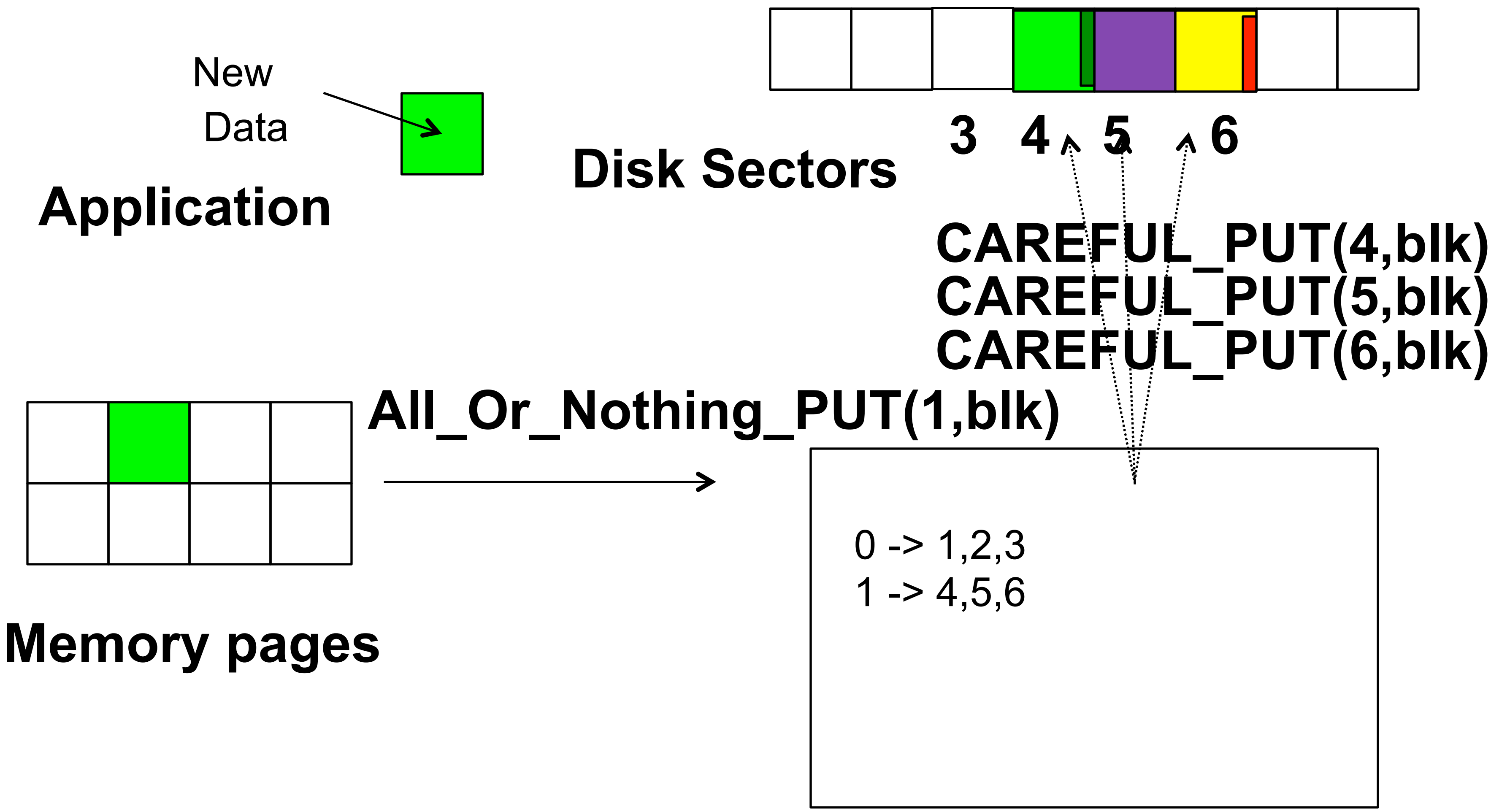
# Example

- AON_GET returns old value (from 6)



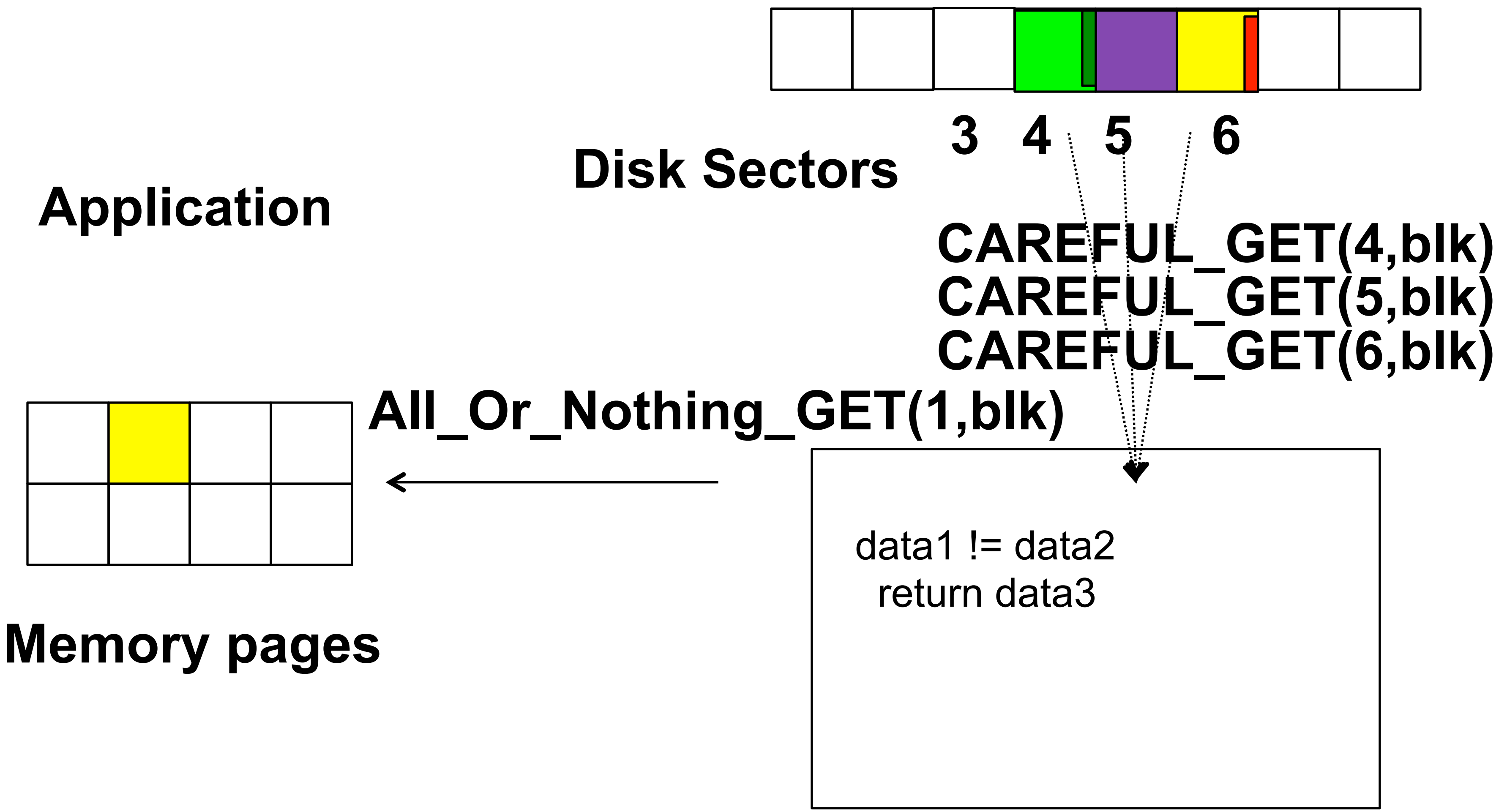
# Example

- CAREFUL_PUT(4) succeeds; C_PUT(5) fails



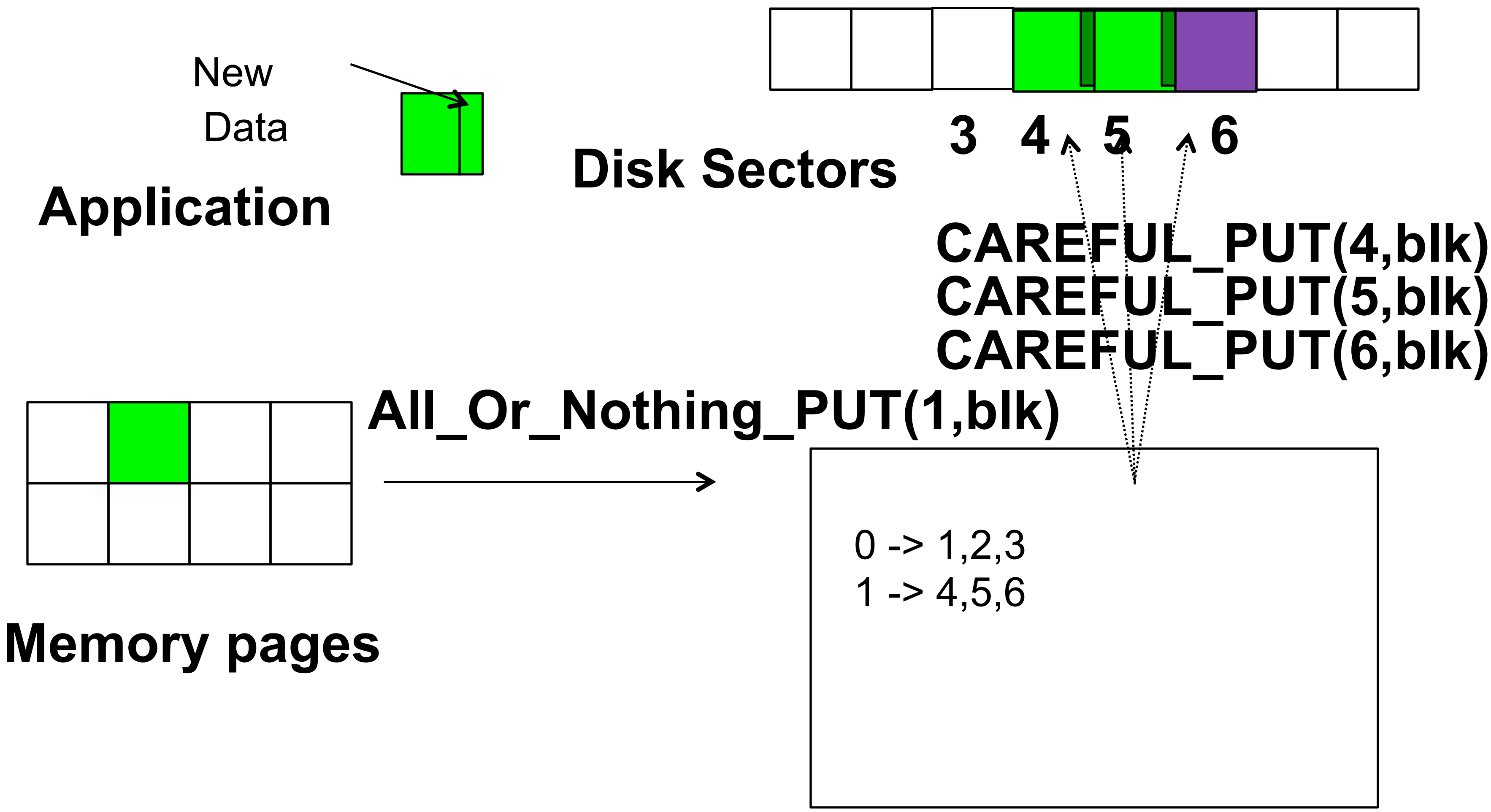
# Example

- AON_GET returns old value (from 6)



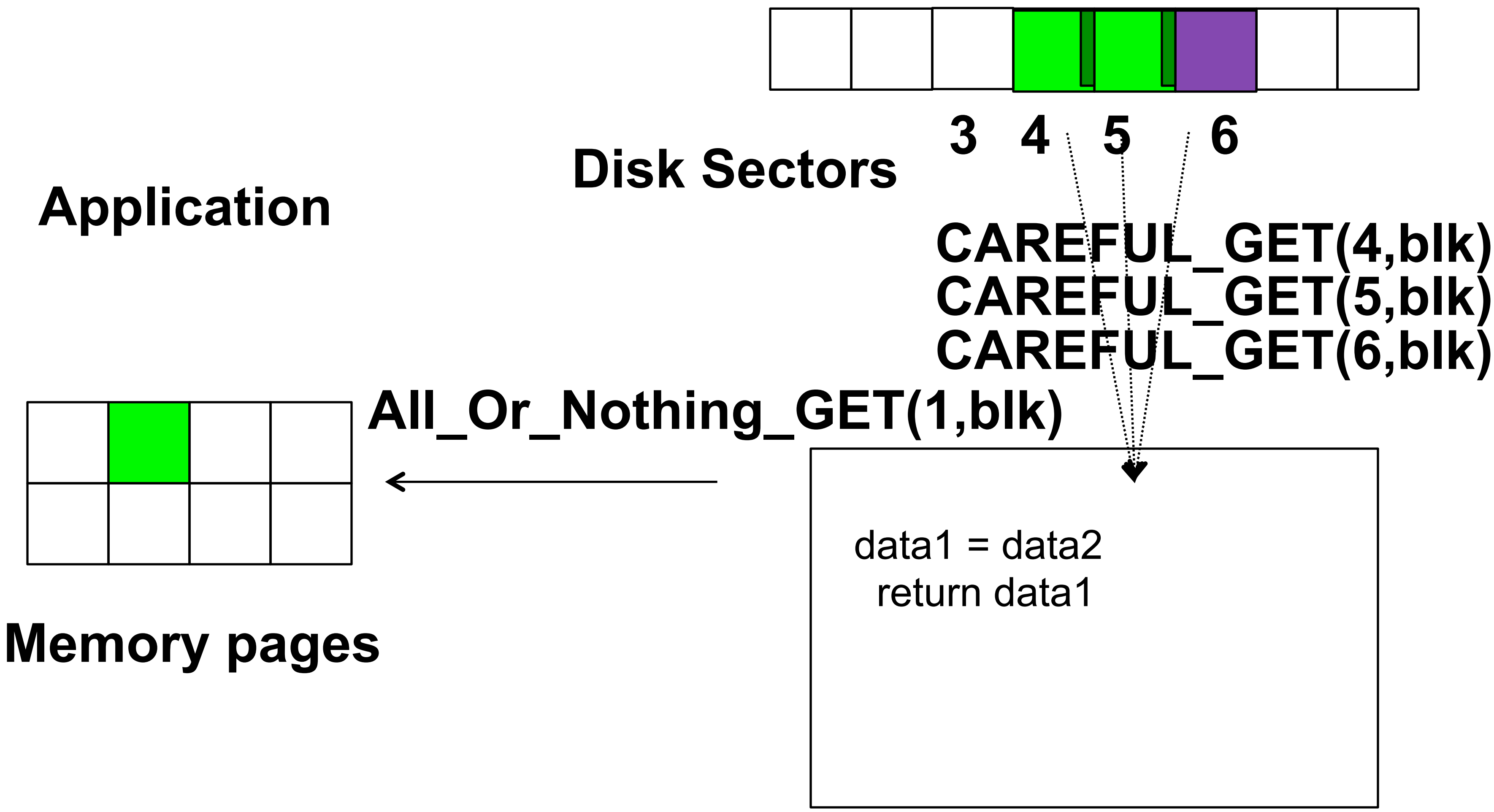
# Example

- CAREFUL_PUT(4,5) succeed; C_PUT(6) fails



# Example

- AON_GET returns newvalue (from 4)



# All-or-nothing atomicity

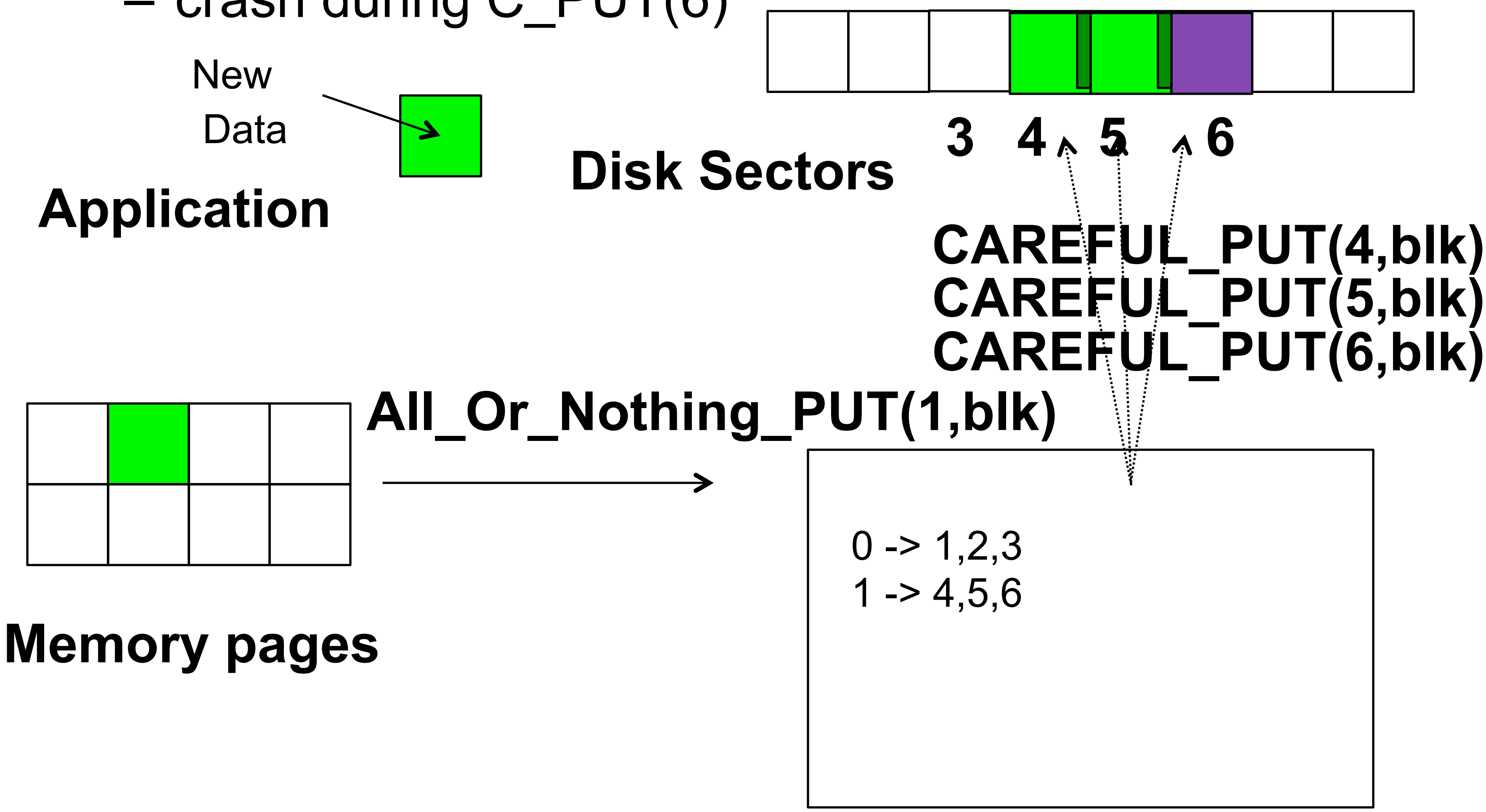
- Problem with this implementation?

```
procedure ALMOST_ALL_OR_NOTHING_PUT (data,
    all_or_nothing_sector)
     CAREFUL_PUT (data, all_or_nothing_sector.S1)
     CAREFUL_PUT (data, all_or_nothing_sector.S2)
     CAREFUL_PUT (data, all_or_nothing_sector.S3)

procedure ALL_OR_NOTHING_GET (reference data,
    all_or_nothing_sector)
     CAREFUL_GET (data1, all_or_nothing_sector.S1)
    CAREFUL_GET (data2, all_or_nothing_sector.S2)
    CAREFUL_GET (data3, all_or_nothing_sector.S3)
     if data1 = data2 then data ← data1
       else data ← data3
```

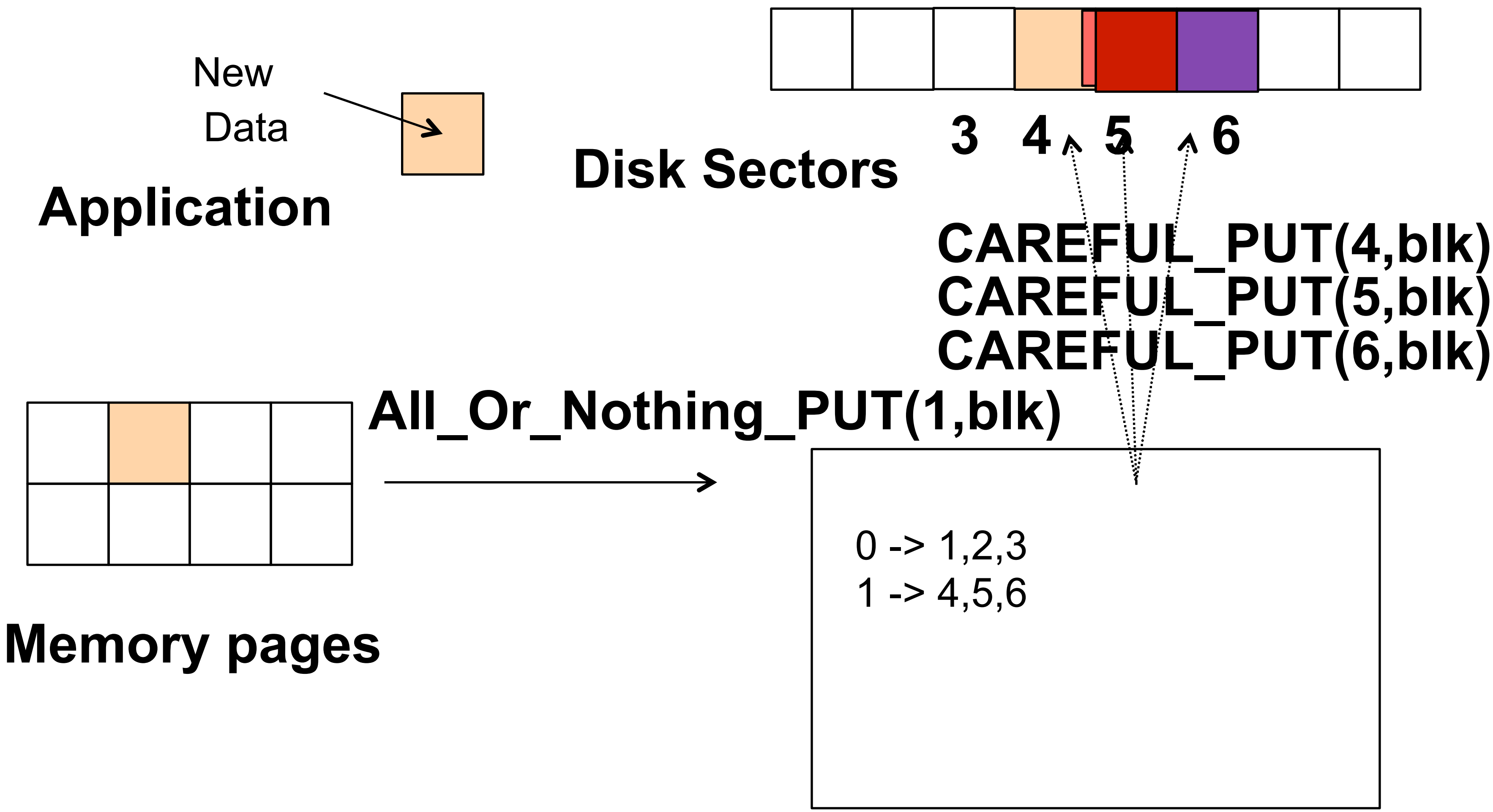
# Example

- CAREFUL_PUT(4,5) succeed
  - crash during C_PUT(6)



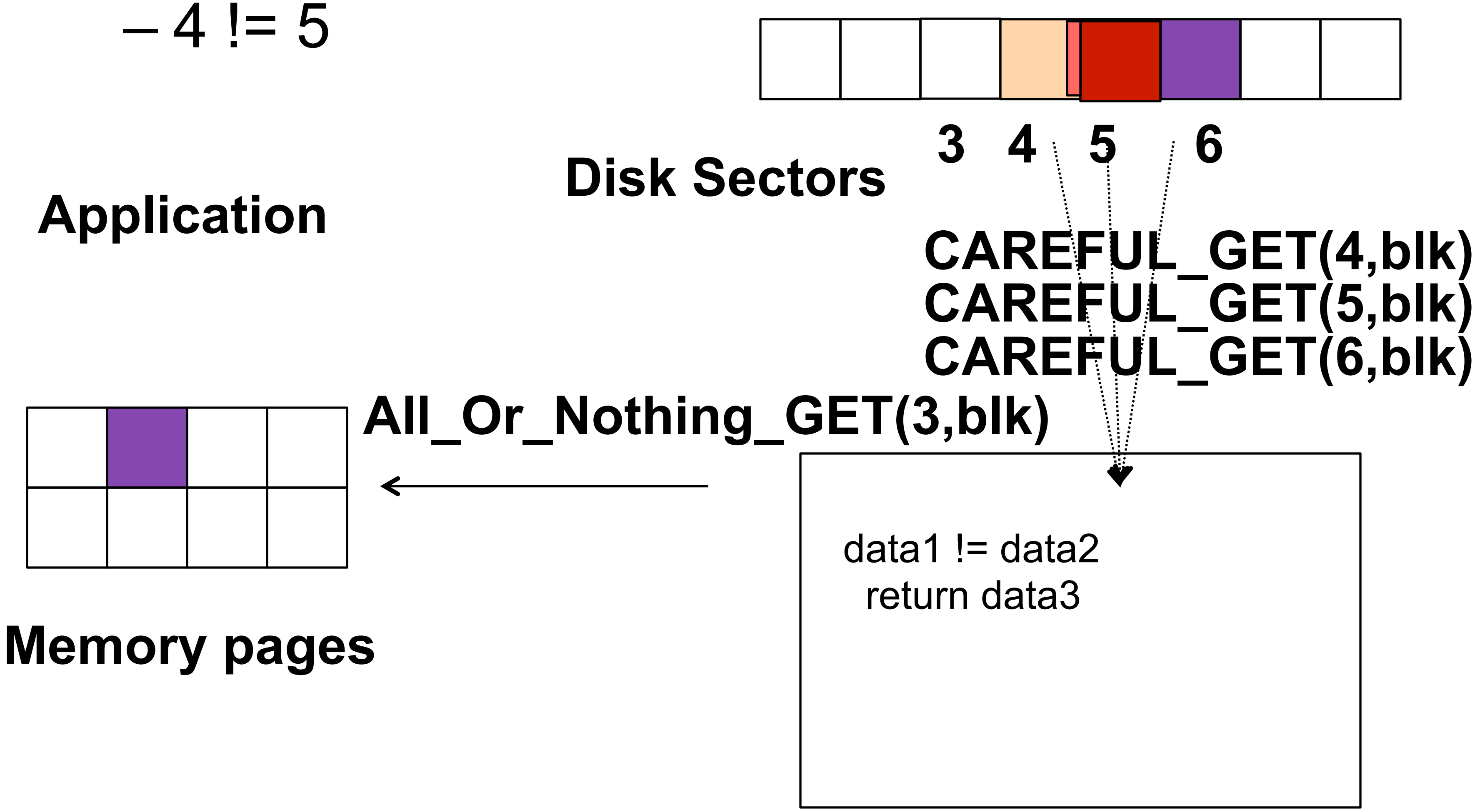
# Example

- Next time, CAREFUL_PUT(5) crashes



# Example

- AON_GET returns wrong value (from 6)
  - 4 != 5



# Fixing All-or-nothing

- Key idea is to maintain that all copies have the same value at the end of an All-or-nothing_PUT
- Check and repair before updates

# Fixing All-or-nothing

- Check and repair – possible states based on (S1, S2, S3) of previous PUT:
  1. (old,old,old)
  2. (bad,old,old)
  3. (new,old,old)
  4. (new,bad,old)
  5. (new,new,old)
  6. (new,new,bad)
  7. (new,new,new)

# Fixing All-or-nothing

- Check and repair – possible states based on sectors (S1, S2, S3) of previous PUT:
  1. (old,old,old) -> returns old
  2. (bad,old,old) -> PUT old into S1
  3. (new,old,old) -> PUT old into S1
  4. (new,bad,old) -> PUT new into S2, S3
  5. (new,new,old) -> PUT new into S3
  6. (new,new,bad) -> PUT new into S3
  7. (new,new,new) -> returns new

# All-Or-Nothing-PUT

```
procedure ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
    CHECK_AND_REPAIR (all_or_nothing_sector)
    ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
procedure CHECK_AND_REPAIR (all_or_nothing_sector)
   CAREFUL_GET (data1, all_or_nothing_sector.S1)
   CAREFUL_GET (data2, all_or_nothing_sector.S2)
   CAREFUL_GET (data3, all_or_nothing_sector.S3)
   if (data1 = data2) and (data2 = data3)  return  // state 7 or 1
   if (data1 = data2)
      CAREFUL_PUT (data1, all_or_nothing_sector.S3) return    // 5 or 6
   if (data2 = data3)
      CAREFUL_PUT (data2, all_or_nothing_sector.S1) return    // 2 or 3
   CAREFUL_PUT (data1, all_or_nothing_sector.S2)      // 4 -> 5
   CAREFUL_PUT (data1, all_or_nothing_sector.S3)      // 5 -> 7
```

# Check and repair

- Failures can happen during CHECK_AND_REPAIR CAREFUL_PUTs
  - Implementation of check_and_repair such that eventually state reaches 1 (old,old,old) or 7 (new,new,new)
    - Assuming no decays
  - E.g.: state 4 (new,bad,old)
    - PUT(S2,new)
      - If it fails; remains in state 4 for next check
      - If it succeeds; goes to state 5 (new,new,old)

# Observations

- Must guarantee only a single thread AON_PUT or AON_GET an AON sector
  - This design by itself does not provide before-or-after atomicity
- Check_and_repair is idempotent
- The successful completion of the line CAREFUL_PUT(S2) in AON_PUT marks a “commit” point
  - All future AON_GETs will retrieve new value

# Generalizing

- All-or-Nothing-PUT addresses the problem of atomic updates of a single sector
- In general, need to support all-or-nothing actions that perform updates to multiple elements

# Example

1. **procedure** TRANSFER (*debit_account, credit_account, amount*)
2. GET (*dbdata, debit_account*)
3. *dbdata* ← *dbdata* - *amount*
4. PUT (*dbdata, debit_account*)
5. GET (*crdata, credit_account*)
6. *crdata* ← *crdata* + *amount*
7. PUT (*crdata, credit_account*)

AON_PUT can prevent corrupted data written to debit account if fails during PUT in line 4

However, does not address the problem that the entire transfer transaction has stopped in the middle

A restart of this thread would debit twice the account

# General all-or-nothing actions

- Ideally:

  begin all-or-nothing action

  …

  arbitrary sequence of lower-layer actions

  end all-or-nothing action

- In practice:

  Pre-commit discipline

  (can back out with no trace)

  Commit point (boundary “all”/“nothing”)

  Post-commit discipline

  (results committed; completion is inevitable, even if fail/restart)

# Abort/commit

- Pre-commit discipline
  - Must be able to abort at any instant
    - Anything that is performed during this step must be such that it is possible to back out leaving no trace
- Post-commit discipline
  - No matter what happens, action must run through completion
    - E.g. through multiple, failures/restart cycles
- AON action has two possible outcomes:
  - Fails before commit point – action “aborts”
  - Passes the commit point – action “commits”

# Post-commit

- The post-commit phase must be such that failures during the phase are tolerated, and committed results externally visible
- An approach that ensures post-commit phase completes *after* a failure/restart is acceptable
  - Design may run recovery action upon restart, or before the next use of data

# Example - shadow copy

- All-or-nothing update of changes to a document in the file system
  - E.g. text editor
- Pre-commit
  - Create a temporary working copy of document
  - Apply all updates to working copy
- Commit
  - Exchange working copy with original
  - E.g. UNIX rename
- Post-commit
  - Release original copy

# All-or-nothing PUT revisited

- procedure ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)

  CAREFUL_PUT (data, all_or_nothing_sector.S1)

  CAREFUL_PUT (data, all_or_nothing_sector.S2) <- commit point

  CAREFUL_PUT (data, all_or_nothing_sector.S3)

- If PUT(S1) fails, or if a crash occurs before commit point, change to S1 will be backed out of before next access, by the check_and_repair procedure

# All-or-nothing PUT revisited

```
procedure ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
    CHECK_AND_REPAIR (all_or_nothing_sector)
    ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
procedure CHECK_AND_REPAIR (all_or_nothing_sector)
   CAREFUL_GET (data1, all_or_nothing_sector.S1)
   CAREFUL_GET (data2, all_or_nothing_sector.S2)
   CAREFUL_GET (data3, all_or_nothing_sector.S3)
   if (data1 = data2) and (data2 = data3)  return  // state 7 or 1
   if (data1 = data2)
      CAREFUL_PUT (data1, all_or_nothing_sector.S3) return    // 5 or 6
   if (data2 = data3)
      CAREFUL_PUT (data2, all_or_nothing_sector.S1) return    // 2 or 3
   CAREFUL_PUT (data1, all_or_nothing_sector.S2)      // 4 -> 5
   CAREFUL_PUT (data1, all_or_nothing_sector.S3)      // 5 -> 7
```

# All-or-nothing PUT revisited

- procedure ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
  CAREFUL_PUT (data, all_or_nothing_sector.S1)
  CAREFUL_PUT (data, all_or_nothing_sector.S2) <- commit point
  CAREFUL_PUT (data, all_or_nothing_sector.S3)

- If PUT(S2) completes, go to state 5
- If PUT(S2) fails before completing, go from state 4 (new,bad,old) to state 5 (new,new,old), possibly through state 6 (if check and repair fails), eventually to state 7 (new,new,new)
  - Commit point is when new value is successfully written to S2

# All-or-nothing PUT revisited

```
procedure ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
    CHECK_AND_REPAIR (all_or_nothing_sector)
    ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
procedure CHECK_AND_REPAIR (all_or_nothing_sector)
   CAREFUL_GET (data1, all_or_nothing_sector.S1)
   CAREFUL_GET (data2, all_or_nothing_sector.S2)
   CAREFUL_GET (data3, all_or_nothing_sector.S3)
   if (data1 = data2) and (data2 = data3)  return  // state 7 or 1
   if (data1 = data2)
      CAREFUL_PUT (data1, all_or_nothing_sector.S3) return    // 5 or 6
  if (data2 = data3)
      CAREFUL_PUT (data2, all_or_nothing_sector.S1) return    // 2 or 3
  CAREFUL_PUT (data1, all_or_nothing_sector.S2)      // 4 -> 5
  CAREFUL_PUT (data1, all_or_nothing_sector.S3)      // 5 -> 7
```

# Golden rule of atomicity

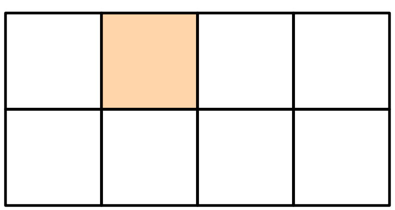
- *Never modify the only copy!*
- But, typical storage systems expose named storage cells that are overwritten
  - PUT(address,value) overwrites cell/sector
    - Previous value stored in address is lost
- All-or-nothing update of a *single* cell is not a problem if PUT is all-or-nothing
  - Memory store, all-or-nothing-put(sector)
- In general, need to update multiple cells – problem if a failure happens between

# Example

- Single operation – all-or-nothing primitive

**Application:**

**account = account + interest**



**All_Or_Nothing_PUT(sector,account)**

**Memory pages**

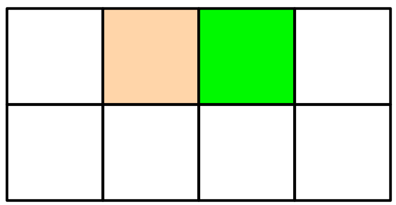
# Example

- All-or-nothing with multiple storage cells to update – failure between all-or-nothing primitives would leave overwritten values behind

**Application:**

**debit = debit - amount**

**credit = credit + amount**



**All_Or_Nothing_PUT(sector1,debit)**

**All_Or_Nothing_PUT(sector2,credit)**

**Memory pages**

# Version histories

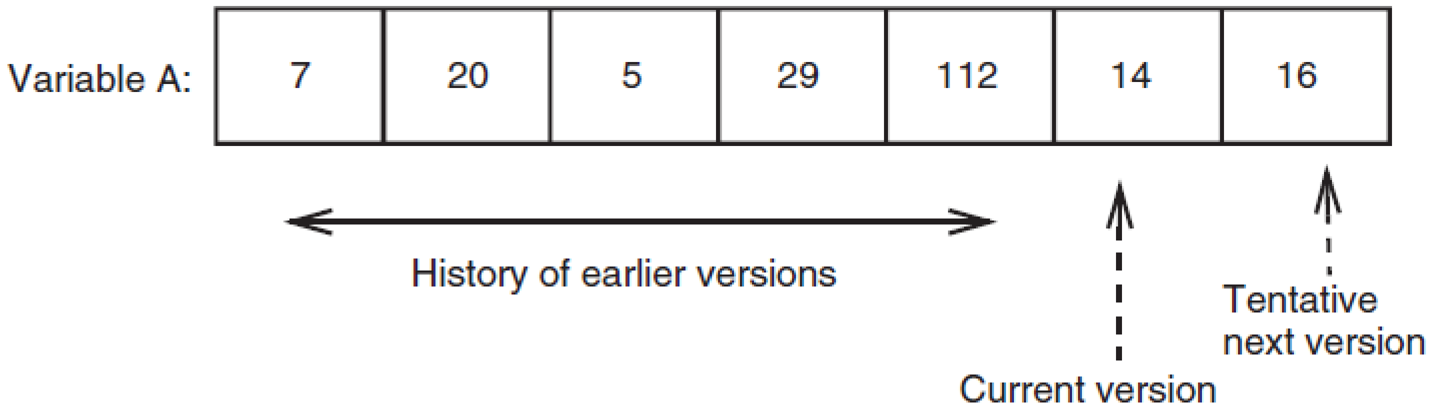
- Dealing with general case of modifying arbitrary data structures
- Model
  - Single thread
    - Let's not worry about before-or-after
  - All modifications in volatile state not committed to durable storage are lost in case of a crash
    - Recall discussion in Chapter 8

# Version histories

- Insight:
  - Updates not visible if bound to a name that is not visible outside of thread
    - E.g. shadow copy of a file
- Provide a storage abstraction different from overwriting a named cell
  - Instead, create a new, tentative version with the updated value, while keeping the current version intact
  - Tentative version not visible outside action until action commits
- "Journal" storage layer on top of cell storage

# Journal storage

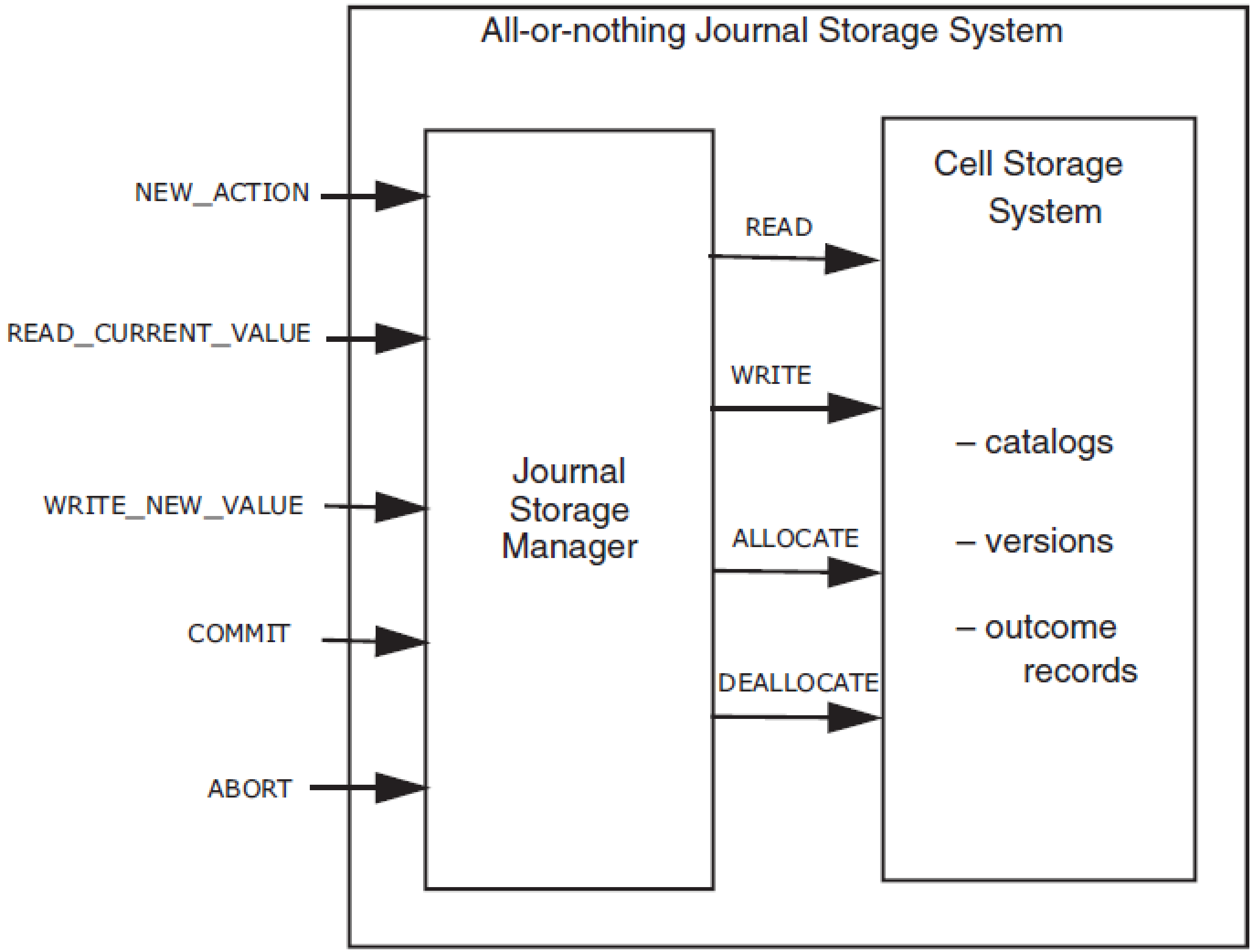
- Associate with every named variable a *list* of cells, instead of a single cell
- Values in the list represent a *history* of values associated with the variable



# Journal storage

- Key insights of the approach:
  - Substitute *overwrite* with *append*
  - Journal storage identifies each all-or-nothing *action* uniquely and associate (multiple) tentative updates with *action ID*
  - Atomically commit all updates in one step, by marking an *action ID* as committed

# Journal storage layer



# Thread's standpoint

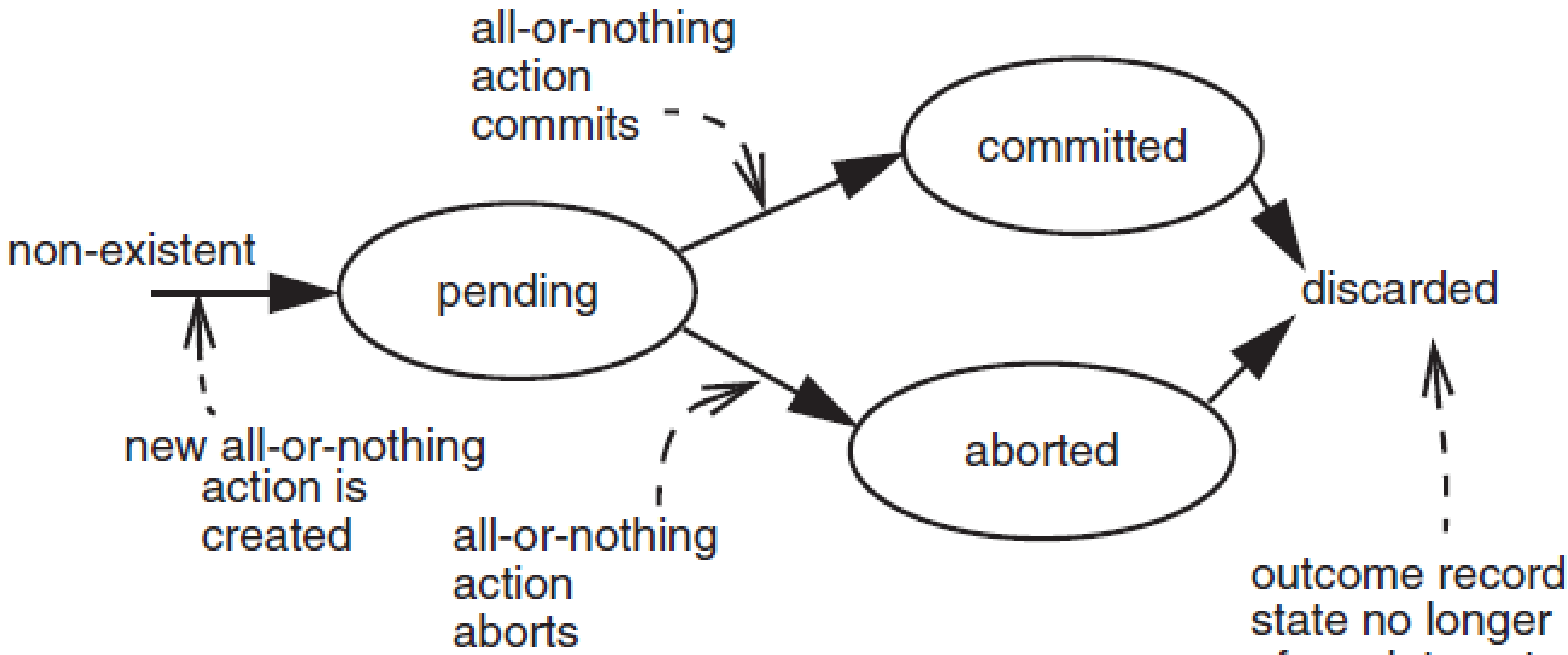
- action_id <- NEW_ACTION
  - WRITE_NEW_VALUE (action_id)
  - READ_CURRENT_VALUE (action_id)
  - WRITE_NEW_VALUE (action_id)
- COMMIT (or ABORT)

- Note, until committed, updated values are only visible to the thread

# Bookkeeping actions

- When a new action is created, it is associated with:
  - A unique action ID (akin to nonce)
  - A state
    - PENDING
    - COMMITTED
    - ABORTED
- This *outcome record* is recorded in non-volatile storage and persists across crashes

# Lifecycle of an action



# Programming interface

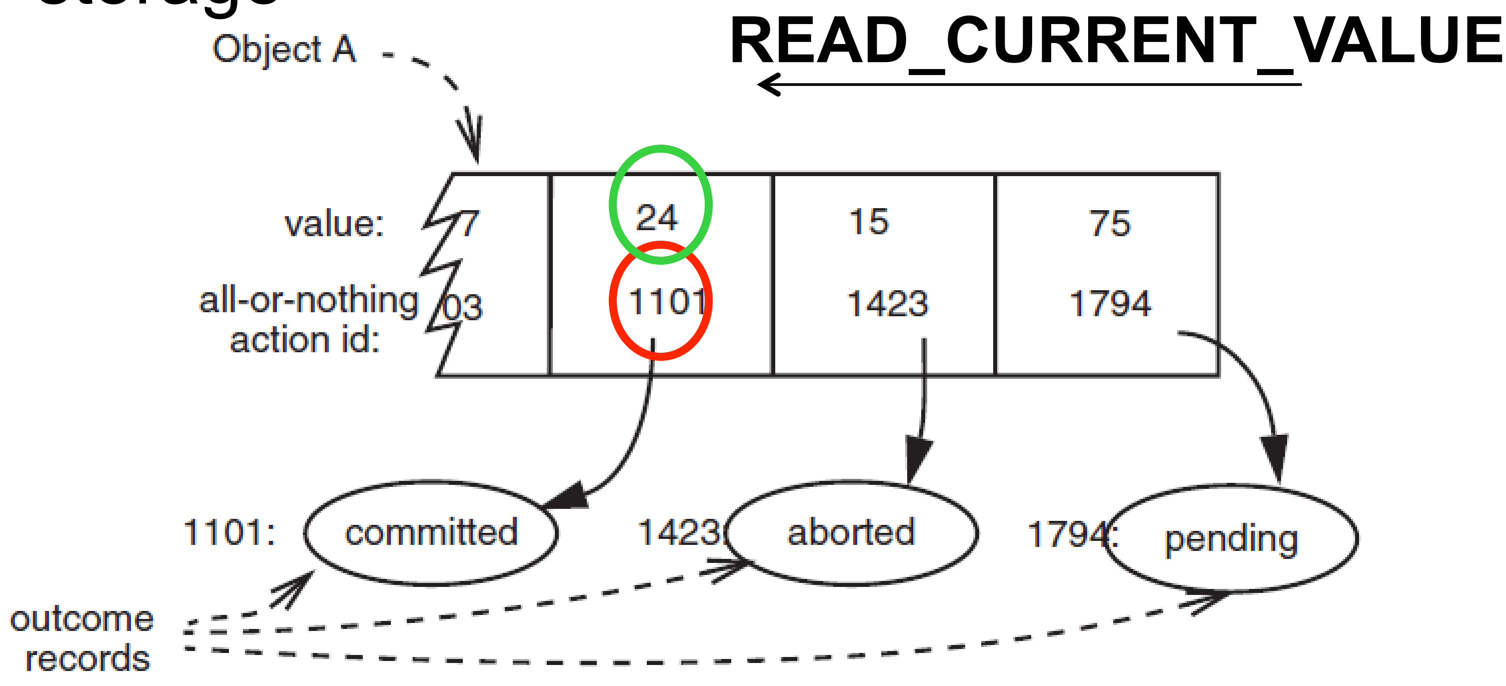
```
procedure NEW_ACTION ()
    id ← NEW_OUTCOME_RECORD ()
    id.outcome_record.state ← PENDING
    return id

procedure COMMIT (reference id)
    id.outcome_record.state ← COMMITTED
    return

procedure ABORT (reference id)
    id.outcome_record.state ← ABORT
    return
```

# Writing new values

- WRITE_NEW_VALUE(actionid,value)
  - Appends new (actionid,value) to the non-volatile list maintained by the journal storage



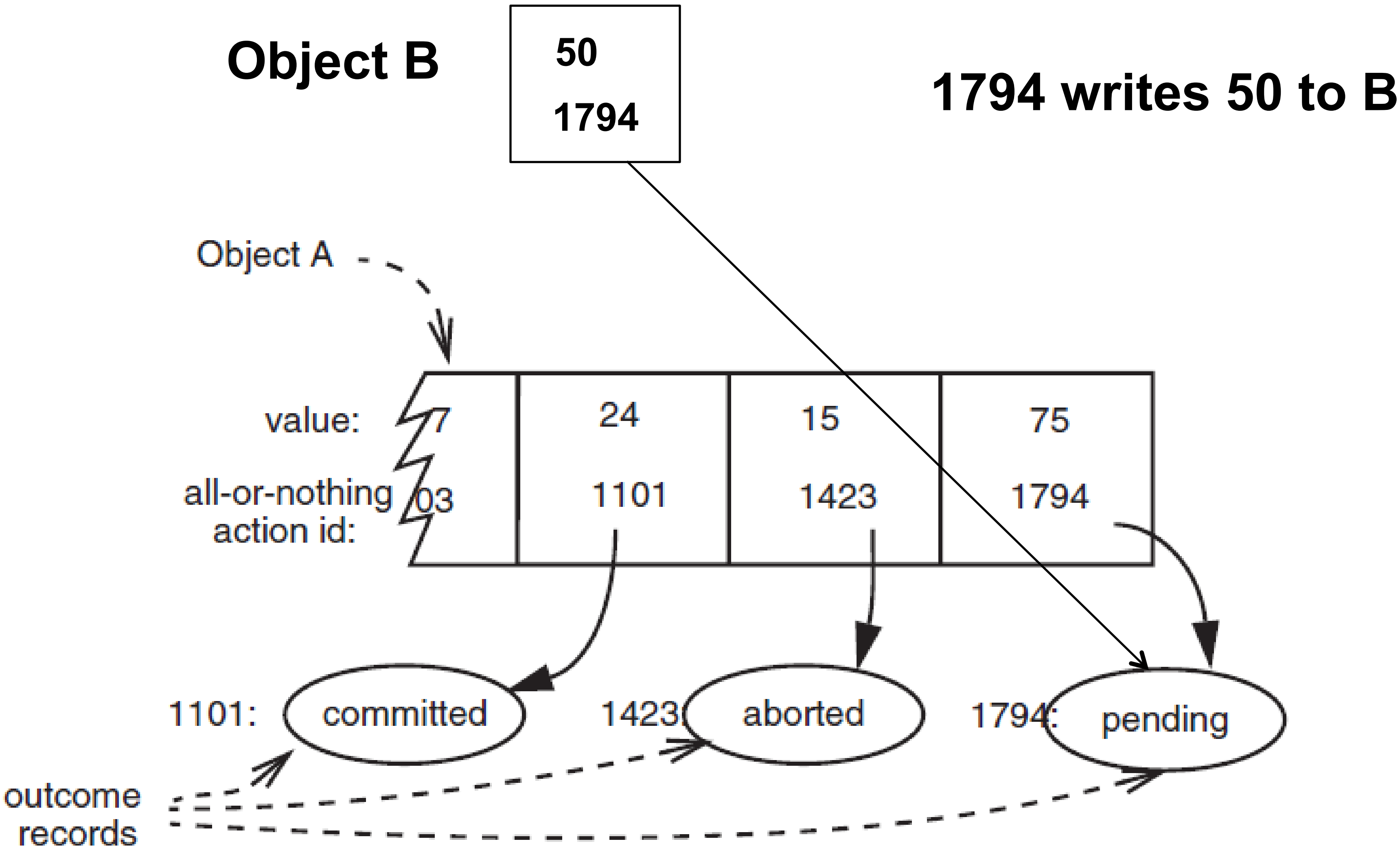
# READ/WRITE API

```
1  procedure READ_CURRENT_VALUE (data_id, caller_id)
2      starting at end of data_id repeat until beginning
3          v ← previous version of data_id        // Get next older version
4          a ← v.action_id  // Identify the action a that created it
5          s ← a.outcome_record.state             // Check action a's outcome record
6          if s = COMMITTED then
7              return v.value
8          else skip v                            // Continue backward search
9      signal ("Tried to read an uninitialized variable!")

10 procedure WRITE_NEW_VALUE (reference data_id, new_value, caller_id)
11     if caller_id.outcome_record.state = PENDING
12         append new version v to data_id
13         v.value ← new_value
14         v.action_id ← caller_id
                   else signal ("Tried to write outside of an all-or-nothing action!")
```

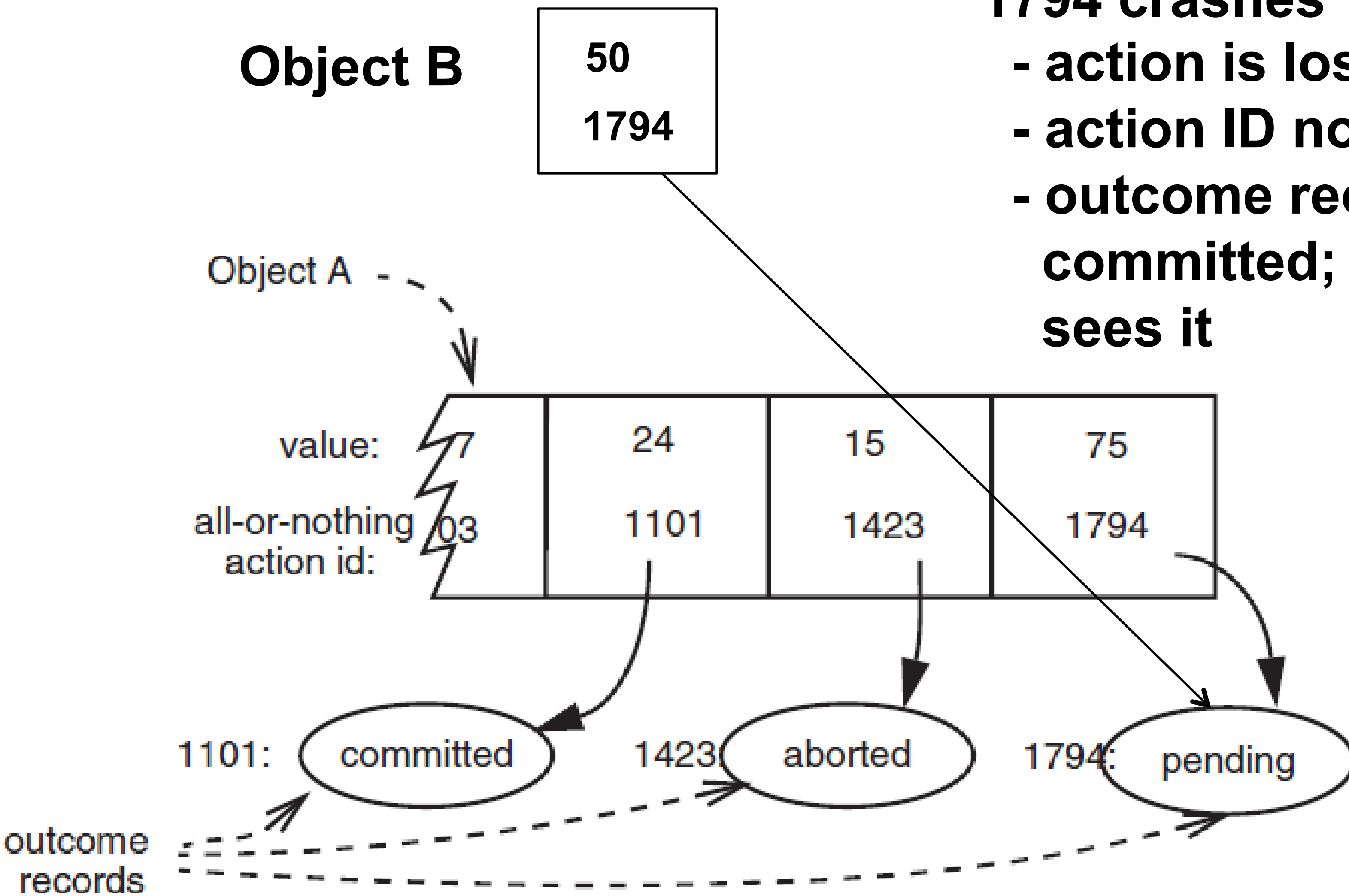
**(READ_MY_PENDING_VALUE may also be exposed; line 6 would test s = PENDING and a = caller_id)**

# Example


Object B
50
1794
1794 writes 50 to B
Object A
value:
7
24
15
75
all-or-nothing action id:
03
1101
1423
1794
1101:
committed
1423:
aborted
1794:
pending
outcome records

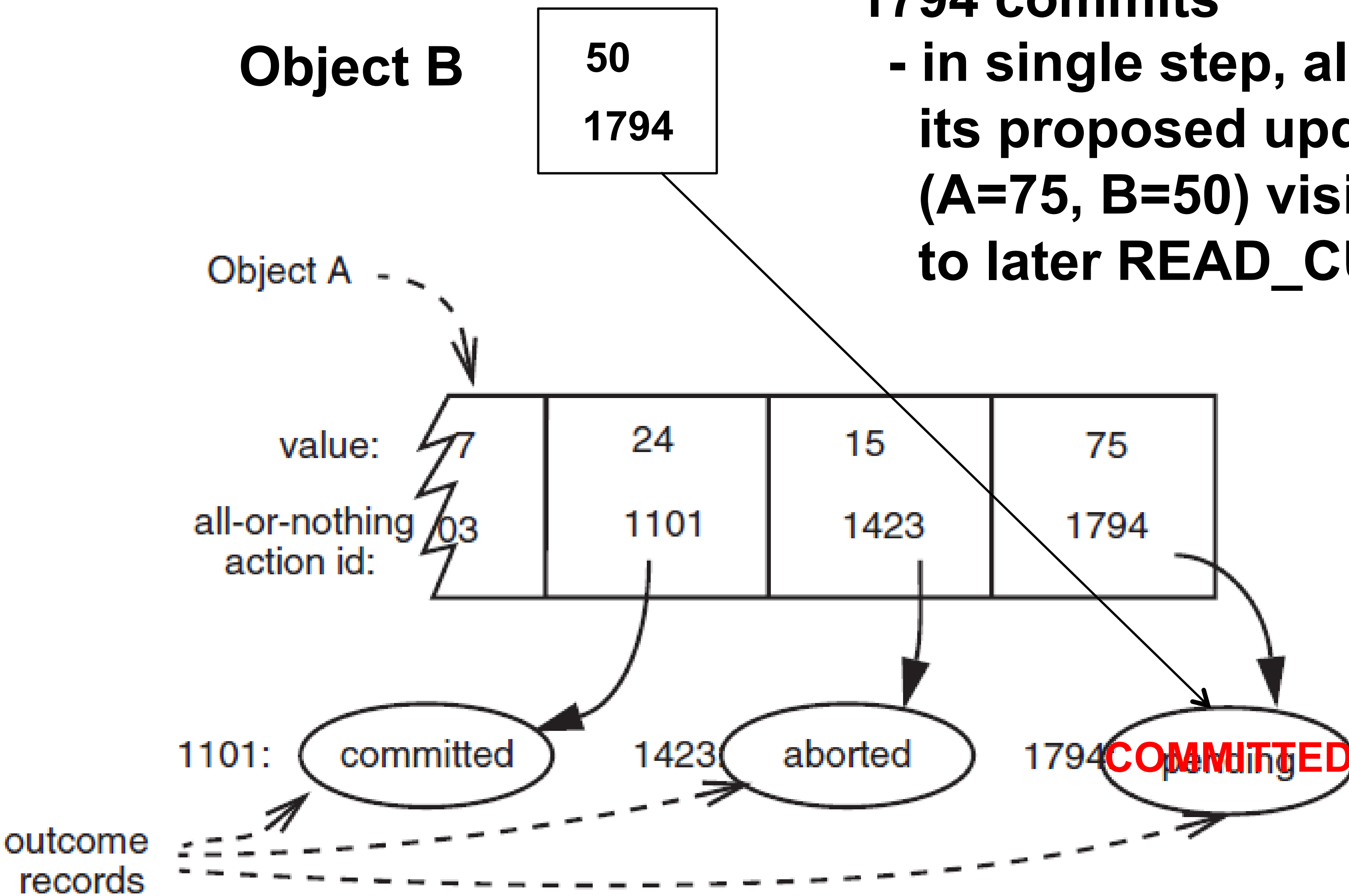
# Example

**1794 crashes**
- **action is lost**
- **action ID not reused**
- **outcome record not committed; no one sees it**



# Example

**1794 commits**
- **in single step, all its proposed updates (A=75, B=50) visible to later READ_CURRENT**



# Account transfer example

```
procedure TRANSFER (reference debit_account, reference credit_account,
                                                                  amount)
    my_id ← NEW_ACTION ()
    xvalue ← READ_CURRENT_VALUE (debit_account, my_id)
    xvalue ← xvalue - amount
    WRITE_NEW_VALUE (debit_account, xvalue, my_id)
    yvalue ← READ_CURRENT_VALUE (credit_account, my_id)
    yvalue ← yvalue + amount
    WRITE_NEW_VALUE (credit_account, yvalue, my_id)
    if xvalue > 0 then
        COMMIT (my_id)
    else
        ABORT (my_id)
        signal("Negative transfers are not allowed.")
```

**All-or-nothing, but not before-or-after**

# Handling updates

- The step of changing an action to "committed" must itself be all-or-nothing
- Adding a new entry to the journal must be all-or-nothing
- Approach:
  – Reduce to the problem of handling the update of single cell
    - E.g. outcome record
    - Pointer in the journal list

# Bootstrapping

- Bootstrapping
  - Recall that we bootstrapped locks with lower-level before-or-after primitives (e.g. RMW)
  - Here, we need a primitive for all-or-nothing
  - Single-sector ALL_OR_NOTHING_PUT

# Applications

- Databases
  – Updates to records
- Journaling file systems
  – Updates to inodes
- Backup/archive systems
  – “Time machine”
- Software revision control systems