All-or-Nothing Atomicity

- An action is atomic
  - if there is no way for a higher layer to discover the internal structure of its implementation
- <> From the point of view of a procedure that invokes an atomic action
  - The atomic action always appears either to complete as anticipated, or to do nothing
  - atomic actions useful in recovering from failures

Before-or-After Atomicity

- <> From the point of view of a concurrent thread
  - An atomic action acts as though it occurs either completely before or completely after every other concurrent atomic action
  - This consequence: makes atomic actions useful for coordinating concurrent threads
- Atomicity hides
  - not just the details of which steps form the atomic action
  - But the very fact that it has structure

All-or-Nothing and Before-or-After Atomicity

- 1. Data abstraction
  - Hide the internal structure of data
- 2. Client/server organization
  - Hide the internal structure of major subsystems
- 3. Atomicity
  - Hide the internal structure of an action
- Enforce industrial-strength modularity
  - Guarantee absence of unanticipated interactions among components of a complex system
- The implementer's point of view
  - Painfully knowing the detail

Atomic actions' benevolent side effects

- audit log
  - atomic actions that run into trouble record
    - the nature of the detected failure and
    - the recovery sequence
  - for later analysis
- Data management system when insert a record
  - rearrange the file into a better physical order
- Cache
- Garbage collection
- They are all hidden from upper levels

Overall system fault tolerance model

- error-free operation不会出错的(不存在的):
  - All work goes according to expectations
  - The user initiates actions and the system confirms the actions by displaying messages to the user
- tolerated error（可容忍错误）:
  - The user who has initiated an action notices that the system failed before it confirmed completion of the action
  - when the system is operating again, checks to see whether or not it actually performed that action

Overall system fault tolerance model

- untolerated error:
  - The system fails without the user noticing
  - the user does not realize that he or she should check or retry an action that the system may not have completed

Disk Storage System Fault Tolerance Model

- A perfect-disk assumption
  - a disk never decays and that it has no hard errors
  - only one thing can go wrong: a system crash at just the wrong time

Disk Storage System Fault Tolerance Model

- The fault tolerance model
  - error-free operation:
    - CAREFUL_GET returns the result of the most recent call to CAREFUL_PUT
    - at sector_number on track, with status = OK.
  - detectable error:
    - The operating system crashes during a CAREFUL_PUT
    - and corrupts the disk buffer in volatile storage
    - and CAREFUL_PUT writes corrupted data on one sector of the disk.

ALL_OR_NOTHING_PUT

```
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)
procedure ALL_OR_NOTHING_GET (reference date,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)    data ← data1
    else  data ← data3
```

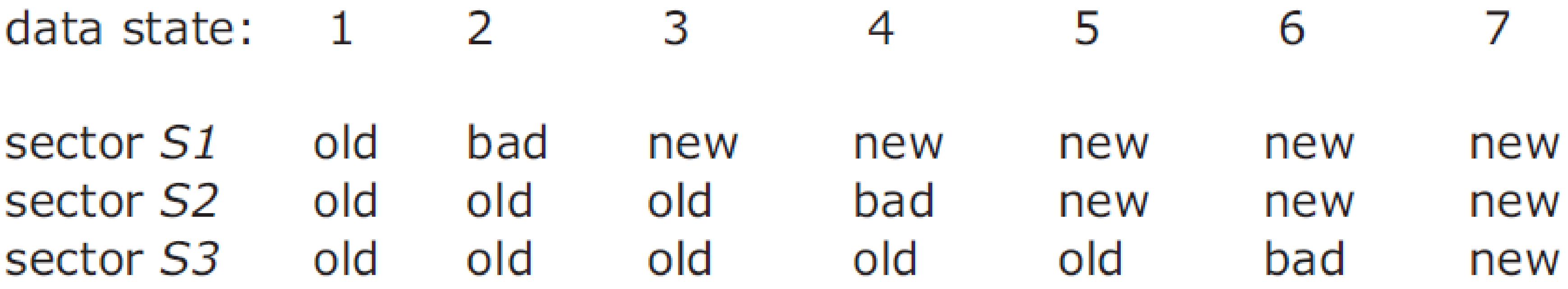
ALL_OR_NOTHING_PUT

```
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)
                                        // Ensure copies match
    CAREFUL_GET (data1, all_or_nothing_sector.S1)
    CAREFUL_GET (data2, all_or_nothing_sector.S2)
    CAREFUL_GET (data3, all_or_nothing_sector.S3)
```

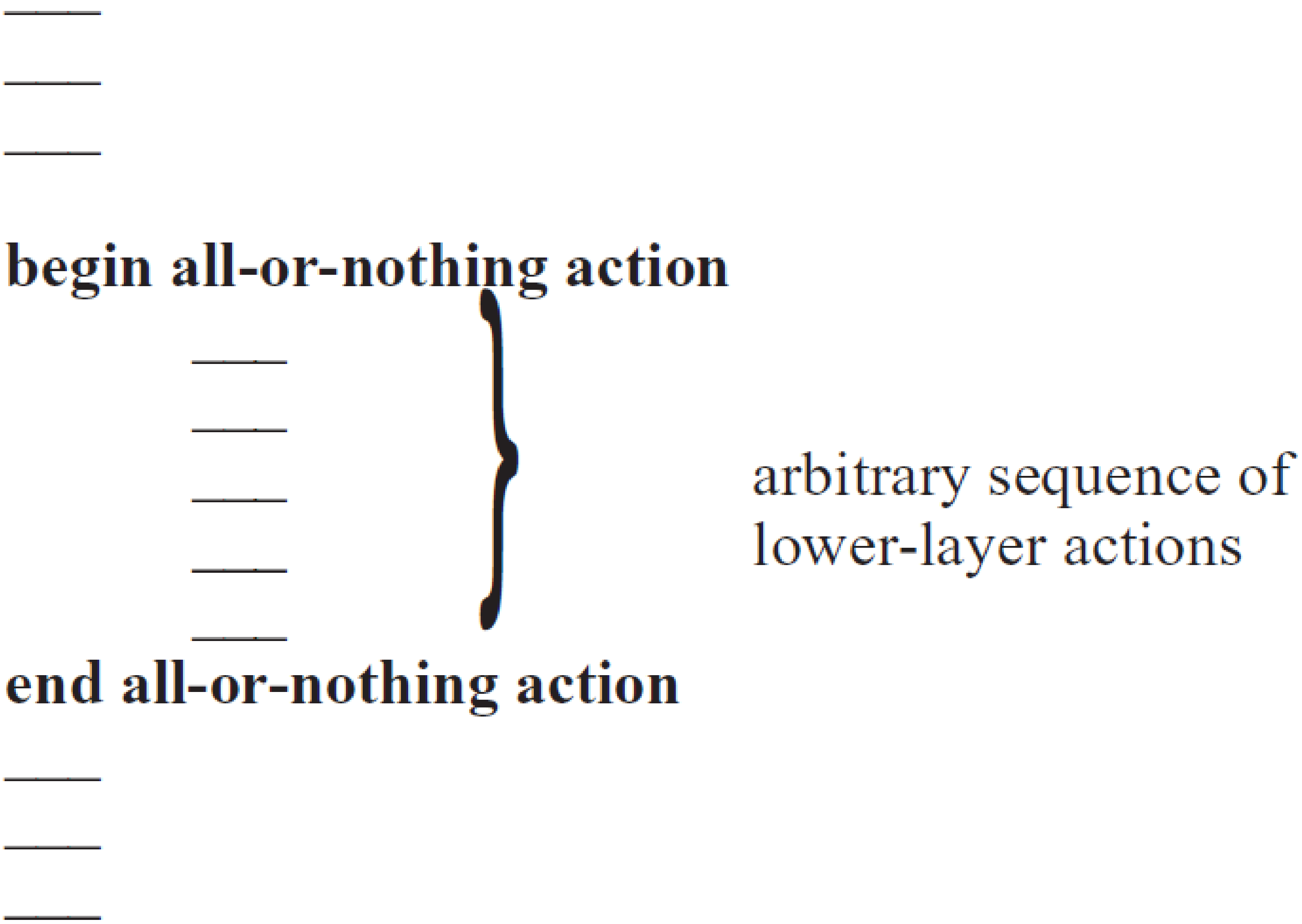
ALL_OR_NOTHING_PUT

```
if (data1 = data2) and (data2 = data3) return    // State 1 or 7, no repair
if (data1 = data2)
    CAREFUL_PUT (data1, all_or_nothing_sector.S3) return  // State 5 or 6.
if (data2 = data3)
    CAREFUL_PUT (data2, all_or_nothing_sector.S1) return  // State 2 or 3.
CAREFUL_PUT (data1, all_or_nothing_sector.S2) // State 4, go to state 5
CAREFUL_PUT (data1, all_or_nothing_sector.S3  // State 5, go to state 7
```



data state:	1	2	3	4	5	6	7
sector *S1*	old	bad	new	new	new	new	new
sector *S2*	old	old	old	bad	new	new	new
sector *S3*	old	old	old	old	old	bad	new

Atomicity



———
———
———

begin all-or-nothing action

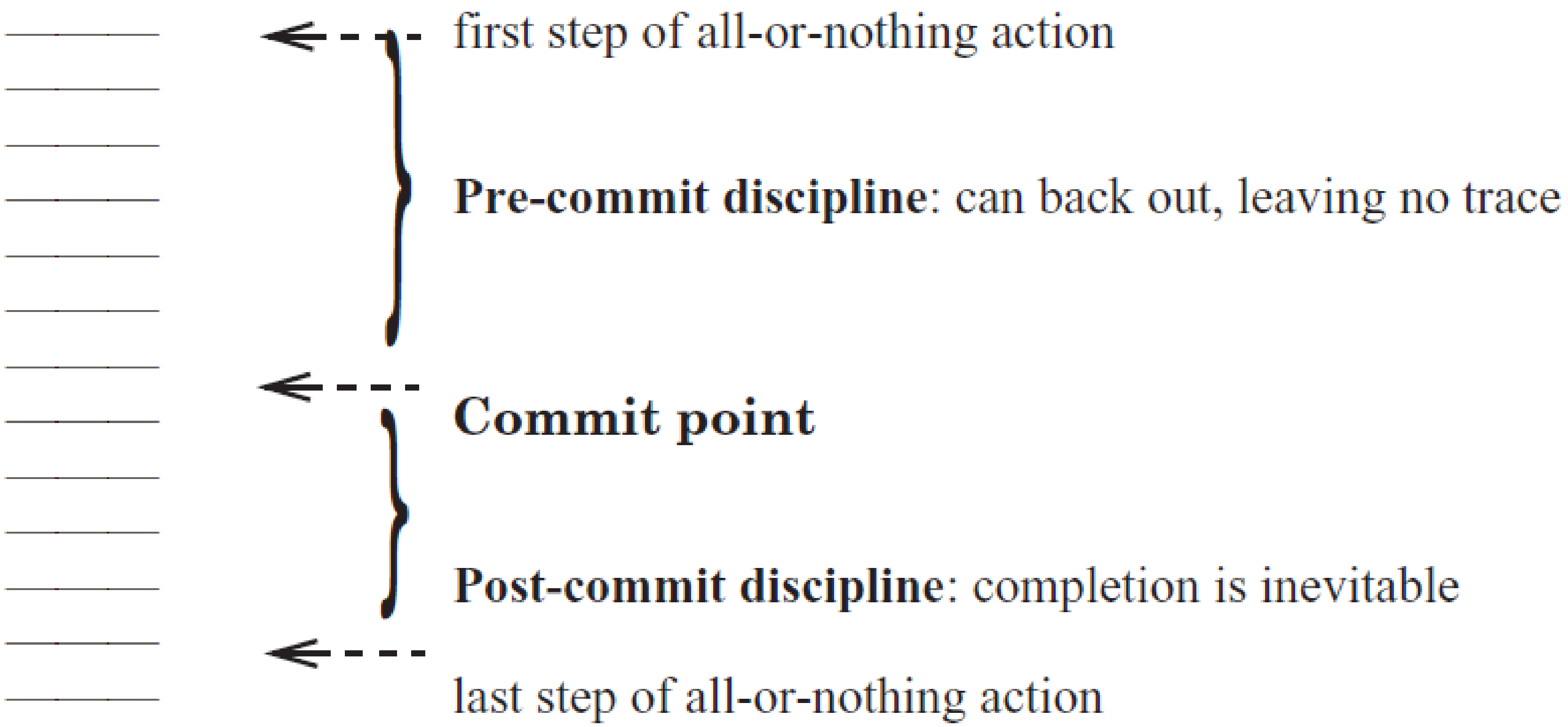
———
———
———
———
———

arbitrary sequence of lower-layer actions

end all-or-nothing action

———
———
———

Commit



first step of all-or-nothing action

Pre-commit discipline: can back out, leaving no trace

Commit point

Post-commit discipline: completion is inevitable

last step of all-or-nothing action

Commit

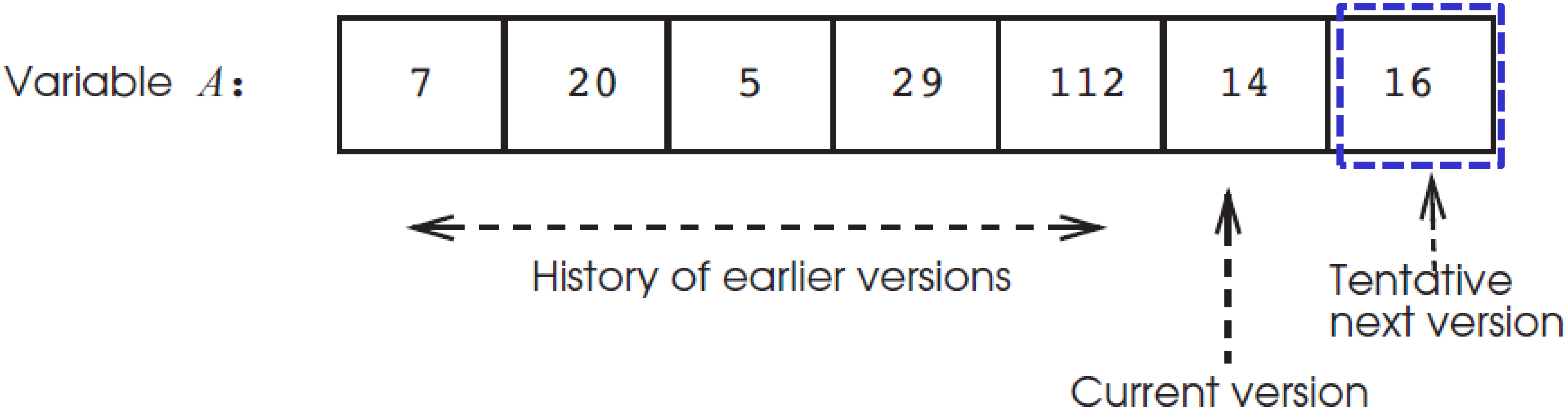
- Pre-commit
  - identify all the resources needed to complete the all-or-nothing action,
  - establish their availability
  - maintain the ability to abort at any instant
    - shared resources, once reserved, cannot be released until the commit point is passed
    - should not do anything externally visible
- Post-commit
  - release reserved resources that are no longer needed
  - perform externally visible actions
  - CANNOT try to acquire additional resources
- Q: where's commit in ALL_OR_NOTHING_PUT?

Shadow Copy

- Pre-commit:
  - Create a complete duplicate working copy of the file that is to be modified
  - make all changes to the working copy
- Commit point:
  - Carefully exchange the working copy with the original
  - Typically this step is bootstrapped
- Post-commit:
  - Release the space that was occupied by the original
- The golden rule of atomicity
  - *Never modify the only copy!*

Journal Storage

- a store operation
  - Not overwrites old data
  - create a new, tentative version of the data
    - remains invisible to any reader outside this all-or-nothing action

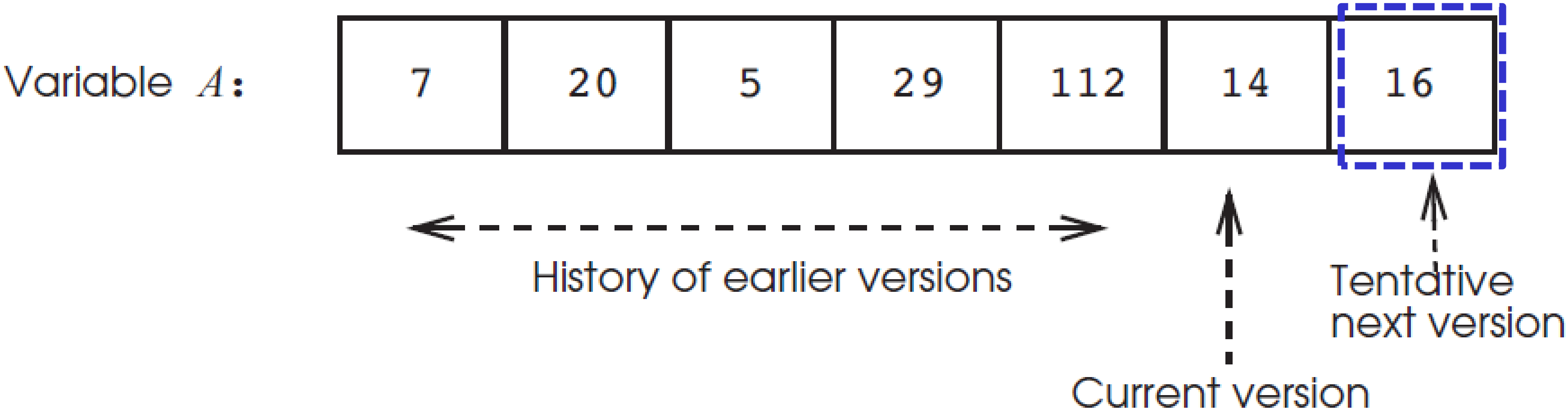


Variable *A*:

7	20	5	29	112	14	16

Journal Storage

- a store operation
  - Not overwrites old data
  - create a new, tentative version of the data
    - remains invisible to any reader outside this all-or-nothing action



Variable *A*:

7 | 20 | 5 | 29 | 112 | 14 | 16

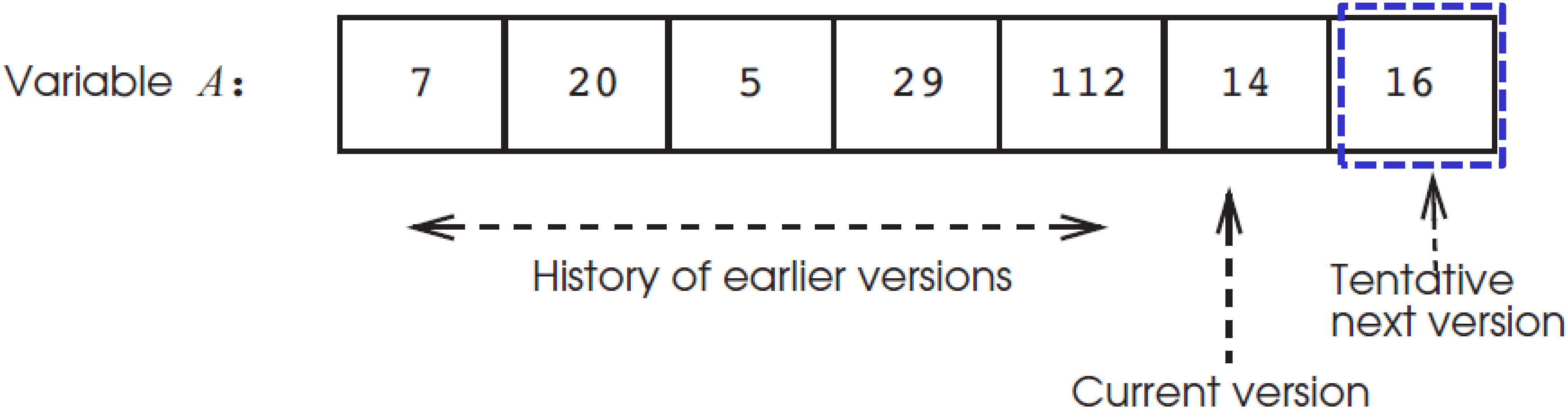
History of earlier versions

Current version

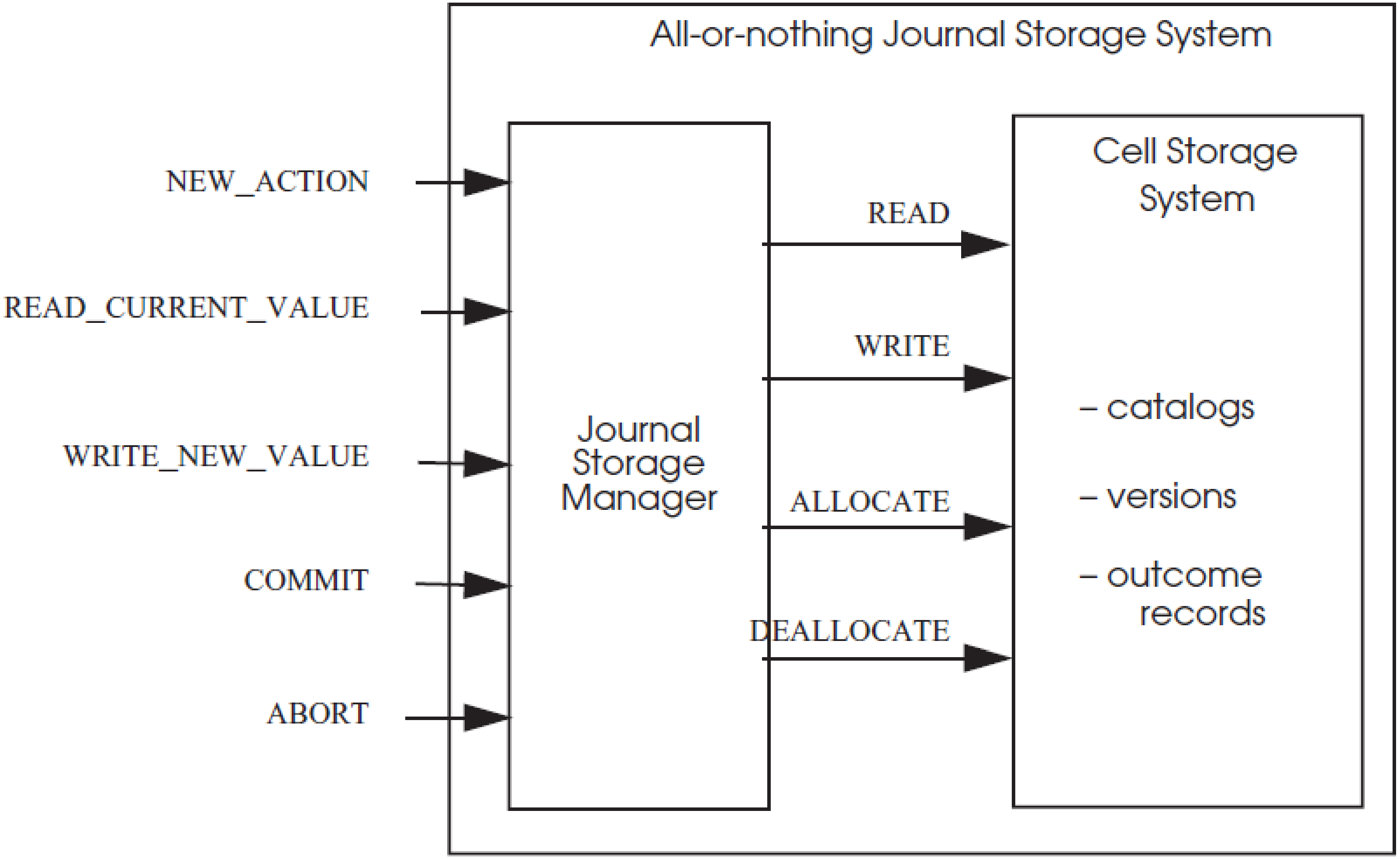
Tentative next version

Journal Storage

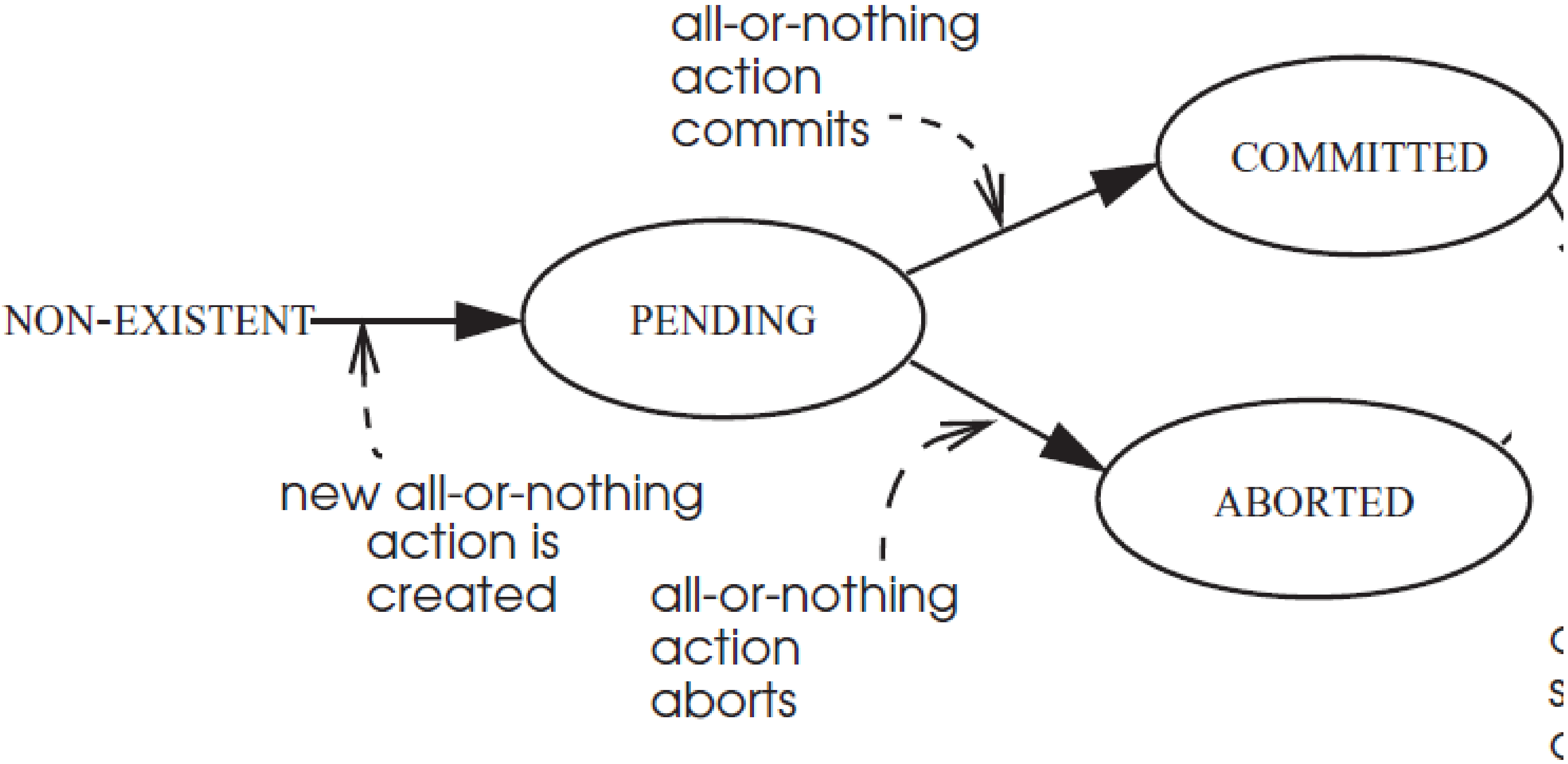
- a store operation
  - Not overwrites old data
  - create a new, tentative version of the data
    - remains invisible to any reader outside this all-or-nothing action



Journal Storage



Journal Storage


all-or-nothing action commits
COMMITTED
NON-EXISTENT
PENDING
ABORTED
new all-or-nothing action is created
all-or-nothing action aborts

Journal Storage

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

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

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

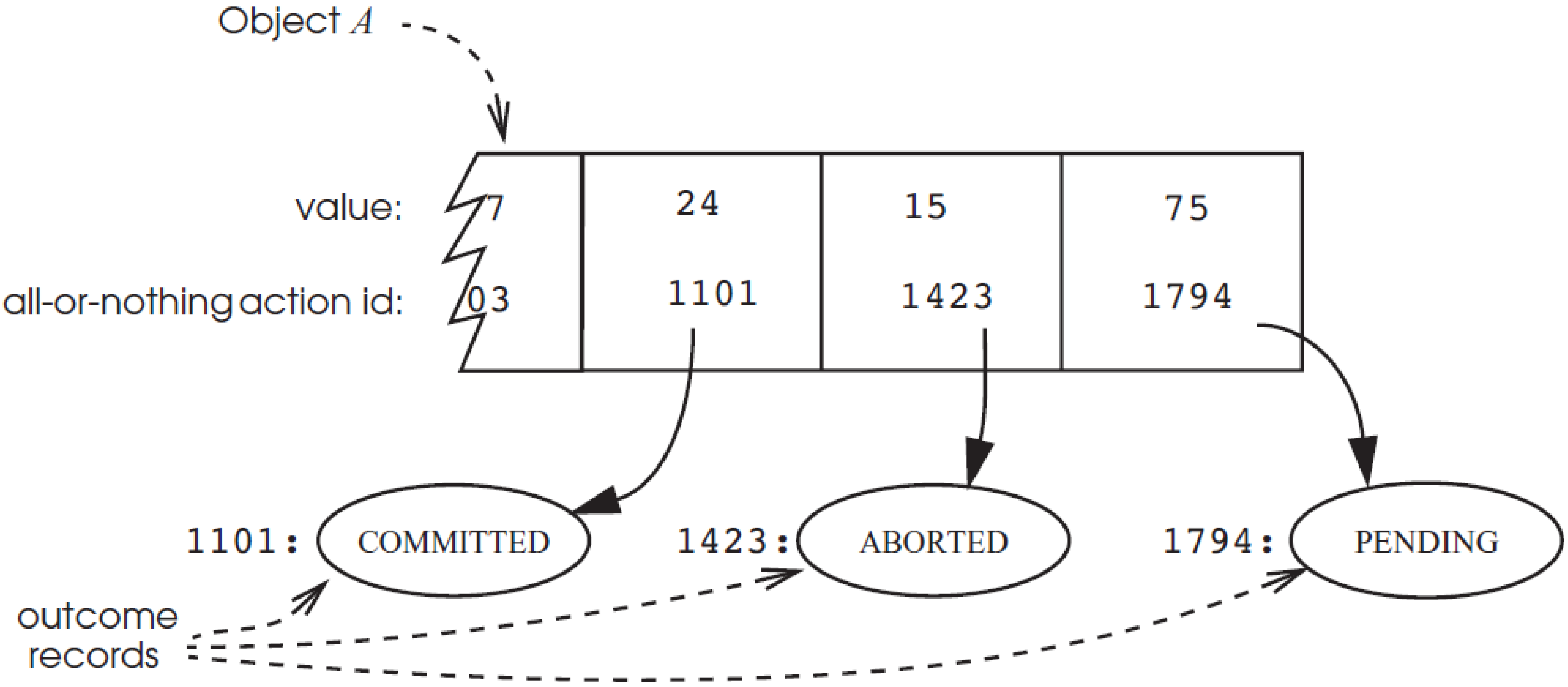
Journal Storage

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

Journal Storage

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

Journal Storage



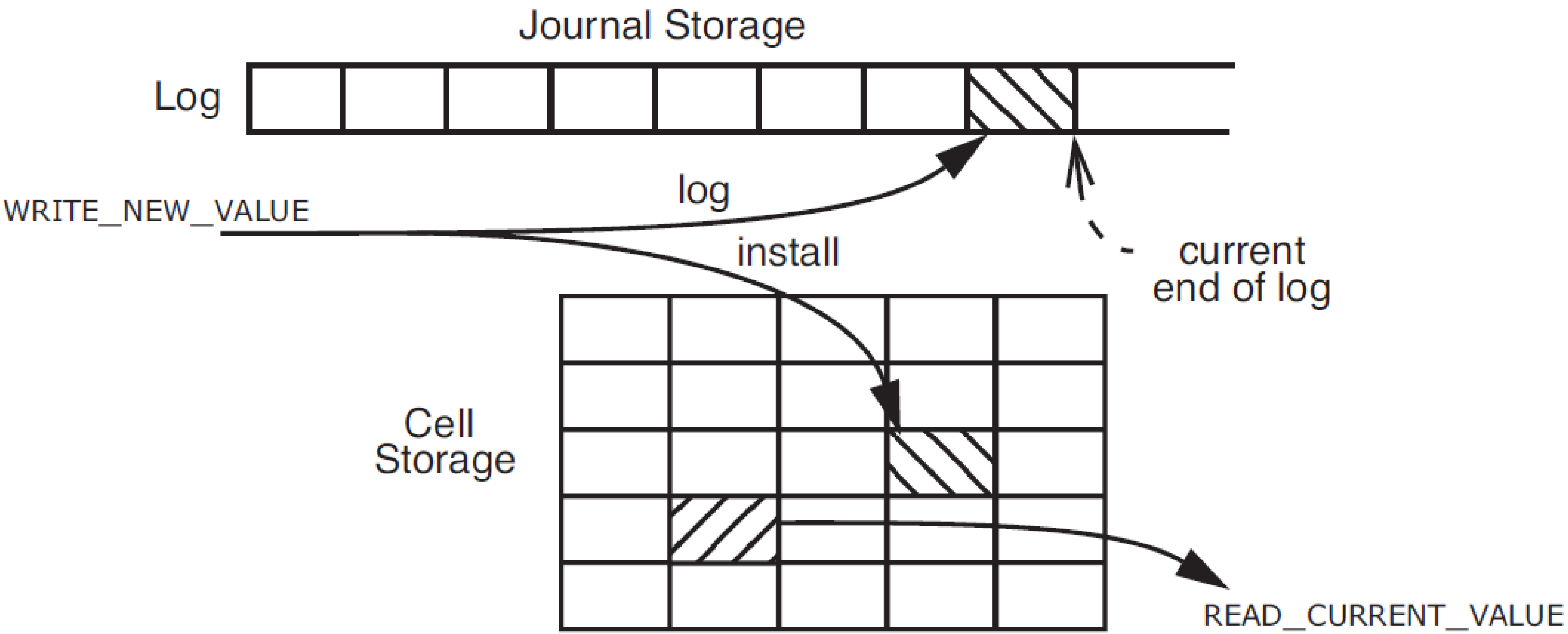
Journal Storage

```
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.")
```

Log

- The application
  - first logs the change in journal storage
  - then it installs the change in cell storage
- separates the reading and writing of data from the failure recovery mechanism
- Minimize the number of storage accesses required for the most common activities
  - Reads, updates
- Rarely-performed activities may not be minimal
  - failure recovery

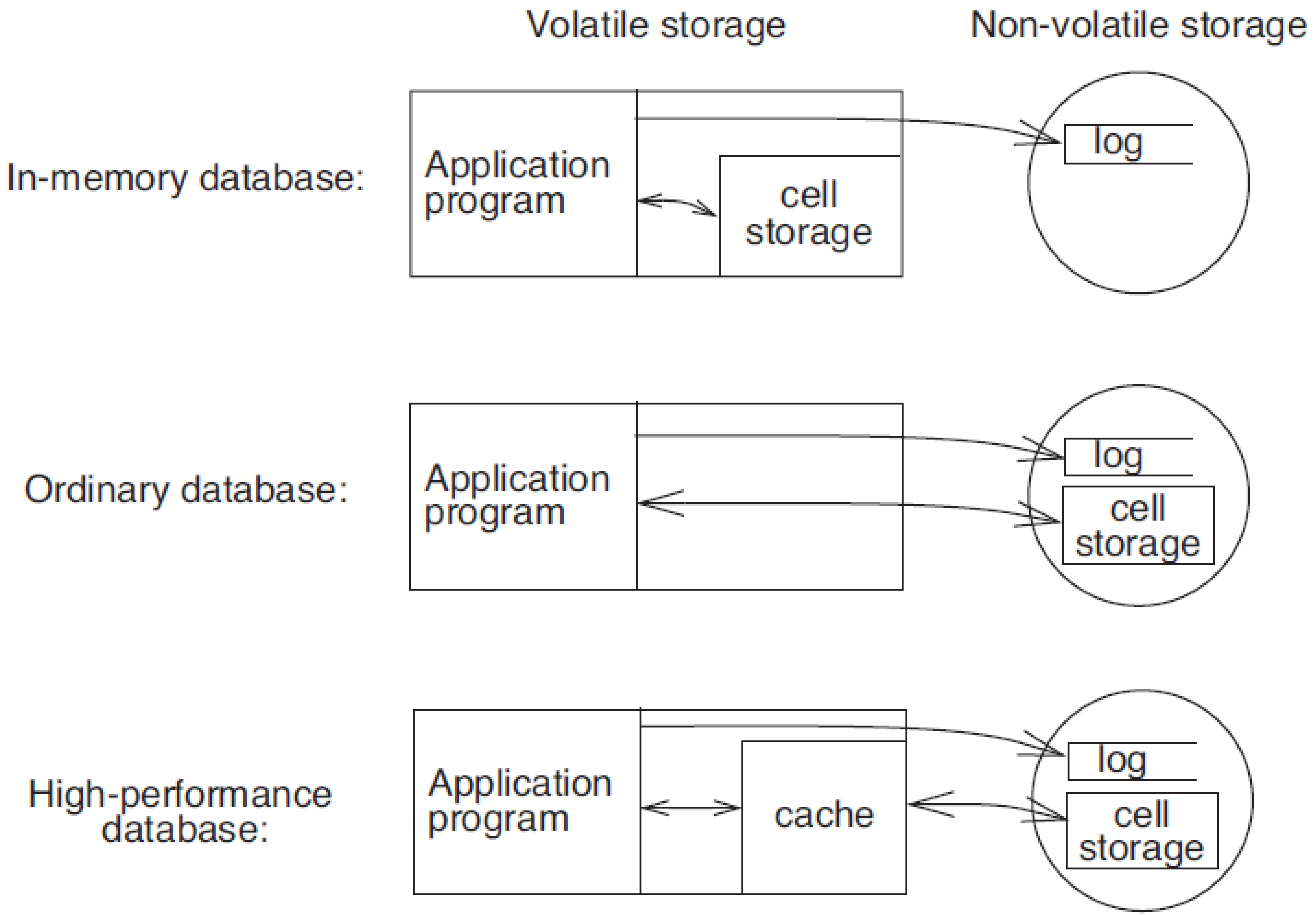
Log



Write-ahead-log protocol (WAL)

Log the update before installing it

Log


Volatile storage
Non-volatile storage
In-memory database:
Application program
cell storage
log
Ordinary database:
Application program
log
cell storage
High-performance database:
Application program
cache
log
cell storage

Logging Protocol

- CHANGE Record
  - The identity of the all-or-nothing action
  - （1）A component action redo
    - installs the intended value in cell storage
      - After commit, if the system crashes , the recovery procedure can perform the install on behalf of the action
  - （2）A second component action undo
    - reverses the effect on cell storage of the install
      - After aborts or the system crashes, it may be necessary for the recovery procedure to reverse the effect

Logging Protocol

- The application
- NEW_ACTION
  - log a BEGIN record that contains just the new identity
  - As the all-or-nothing action proceeds through its pre-commit phase, it logs CHANGE records
- To implement COMMIT or ABORT
  - logs an OUTCOME record
  - commit point

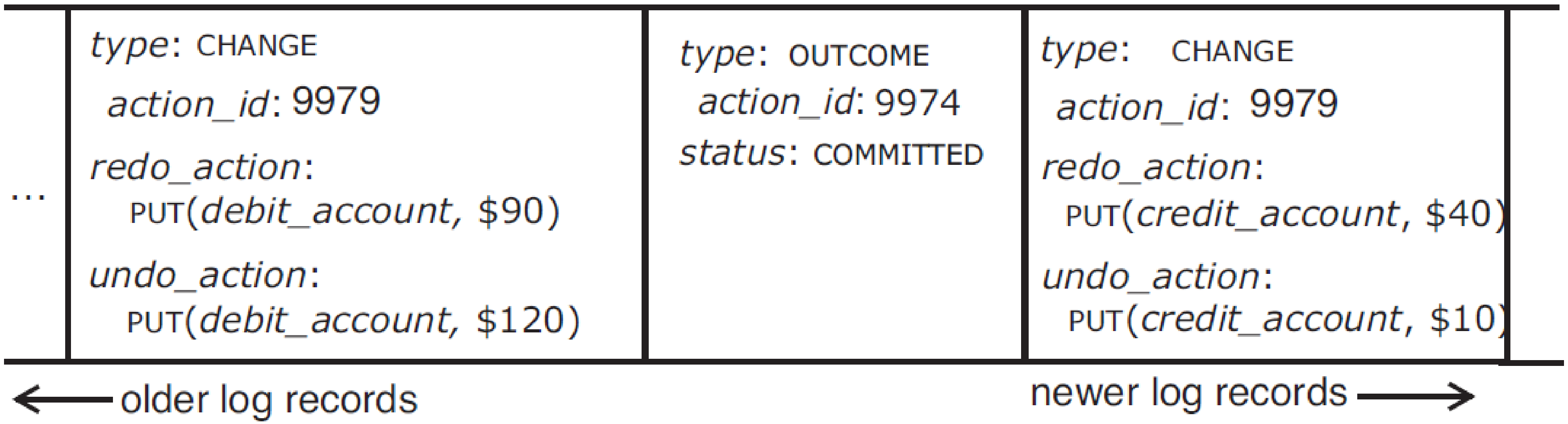
Logging Protocol

```
procedure TRANSFER (debit_account, credit_account, amount)
    my_id ← LOG (BEGIN_TRANSACTION)
    dbvalue.old ← GET (debit_account)
    dbvalue.new ← dbvalue.old - amount
    crvalue.old ← GET (credit_account, my_id)
    crvalue.new ← crvalue.old + amount
    LOG (CHANGE,  my_id,
            "PUT (debit_account, dbvalue.new)", //redo action
            "PUT (debit_account, dbvalue.old)" ) //undo action
```

Logging Protocol ---transfer (cont.)

```
LOG (    CHANGE, my_id,
               "PUT (credit_account, crvalue.new)"    //redo action
               "PUT (credit_account, crvalue.old)"    ) //undo action
PUT (debit_account, dbvalue.new) // install
PUT (credit_account, crvalue.new) // install
if dbvalue.new > 0 then
        LOG ( OUTCOME, COMMIT, my_id)
else
        LOG (OUTCOME, ABORT, my_id)
        signal("Action not allowed. Would make debit account negative.")
LOG (END_TRANSACTION, my_id)
```

Logging Protocol



...

type: CHANGE
action_id: 9979
redo_action:
PUT(*debit_account*, $90)
undo_action:
PUT(*debit_account*, $120)

type: OUTCOME
action_id: 9974
status: COMMITTED

type: CHANGE
action_id: 9979
redo_action:
PUT(*credit_account*, $40)
undo_action:
PUT(*credit_account*, $10)

Logging Protocol

```
procedure ABORT (action_id)
    starting at end of log repeat until beginning
        log_record ← previous record of log
        if log_record.id = action_id then
            if (log_record.type = OUTCOME)
                then signal ("Can't abort an already completed action.")
            if (log_record.type = CHANGE)
                then perform undo_action of log_record
            if (log_record.type = BEGIN)
                then break repeat
    LOG (action_id, OUTCOME, ABORTED) // Block future undos.
    LOG (action_id, END)
```

Logging Protocol: in-memory database

```
procedure RECOVER () // Recovery procedure for a volatile, in-memory database.
    winners ← NULL
    starting at end of log repeat until beginning
        log_record ← previous record of log
        if (log_record.type = OUTCOME)
            then winners ← winners + log_record // Set addition.
    starting at beginning of log repeat until end
        log_record ← next record of log
        if (log_record.type= CHANGE)
            and (outcome_record ← find (log_record.action_id) in winners)
            and (outcome_record.status = COMMITTED) then
            perform log_record.redo_action
```

- Soft state
  - Can be discarded, no need to redo
- Crash during recovery (volitile cell)

Non-volatile logging

- NEW_ACTION 111
- CHANGE 111

  New A→ old A
- CHANGE 111

  New B -> old B

(some install)

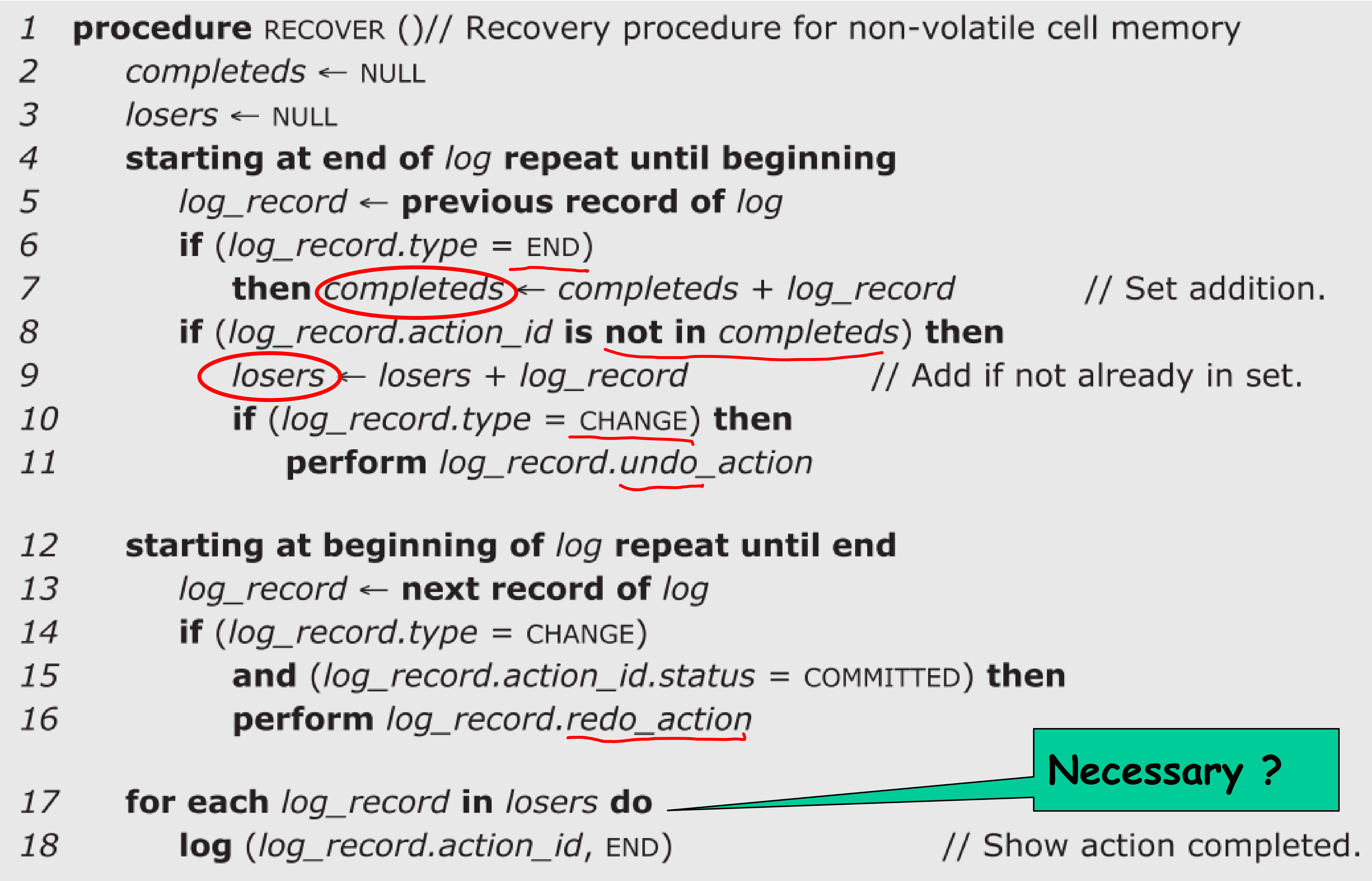
- OUTCOME 111

  COMMIT

(some install)

- END 111

Logging Protocol: non-volatile cell memory



```
procedure RECOVER ()// Recovery procedure for non-volatile cell memory
    completeds ← NULL
    losers ← NULL
    starting at end of log repeat until beginning
        log_record ← previous record of log
        if (log_record.type = END)
            then completeds ← completeds + log_record            // Set addition.
        if (log_record.action_id is not in completeds) then
            losers ← losers + log_record                 // Add if not already in set.
            if (log_record.type = CHANGE) then
                perform log_record.undo_action

    starting at beginning of log repeat until end
        log_record ← next record of log
        if (log_record.type = CHANGE)
            and (log_record.action_id.status = COMMITTED) then
            perform log_record.redo_action

    for each log_record in losers do
        log (log_record.action_id, END)                   // Show action completed.
```

Necessary ?

Undo logging

- NEW_ACTION 114
- CHANGE 114

  New A→ old A

  New B -> old B

(install the cell)

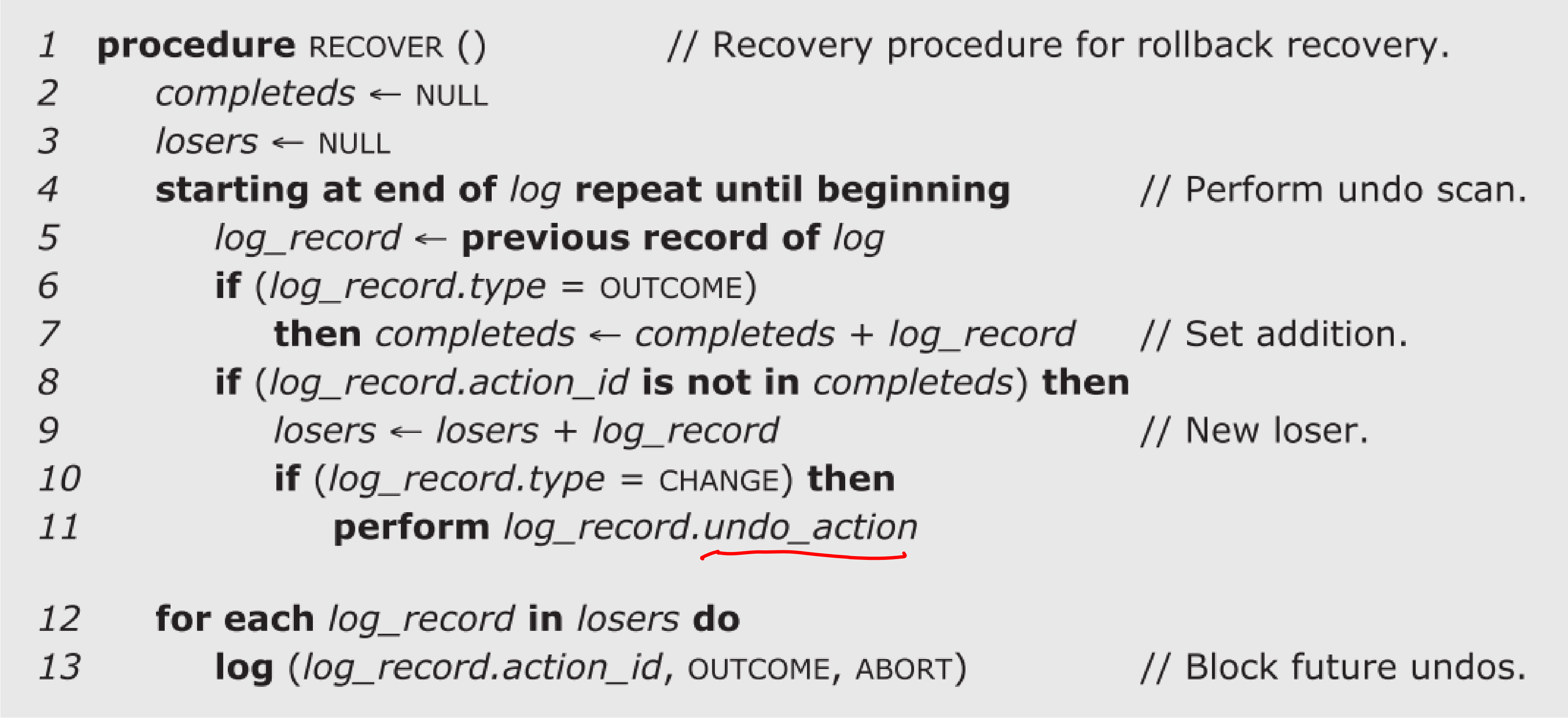
- OUTCOME 114

  COMMIT

- END 114

Undo logging

Logging Protocol: non-volatile cell memory



```
procedure RECOVER ()                              // Recovery procedure for rollback recovery.
    completeds ← NULL
    losers ← NULL
    starting at end of log repeat until beginning           // Perform undo scan.
        log_record ← previous record of log
        if (log_record.type = OUTCOME)
            then completeds ← completeds + log_record       // Set addition.
        if (log_record.action_id is not in completeds) then
            losers ← losers + log_record                    // New loser.
            if (log_record.type = CHANGE) then
                perform log_record.undo_action

    for each log_record in losers do
        log (log_record.action_id, OUTCOME, ABORT)          // Block future undos.
```

redo logging

- NEW_ACTION 114
- CHANGE 114

  New A→ old A

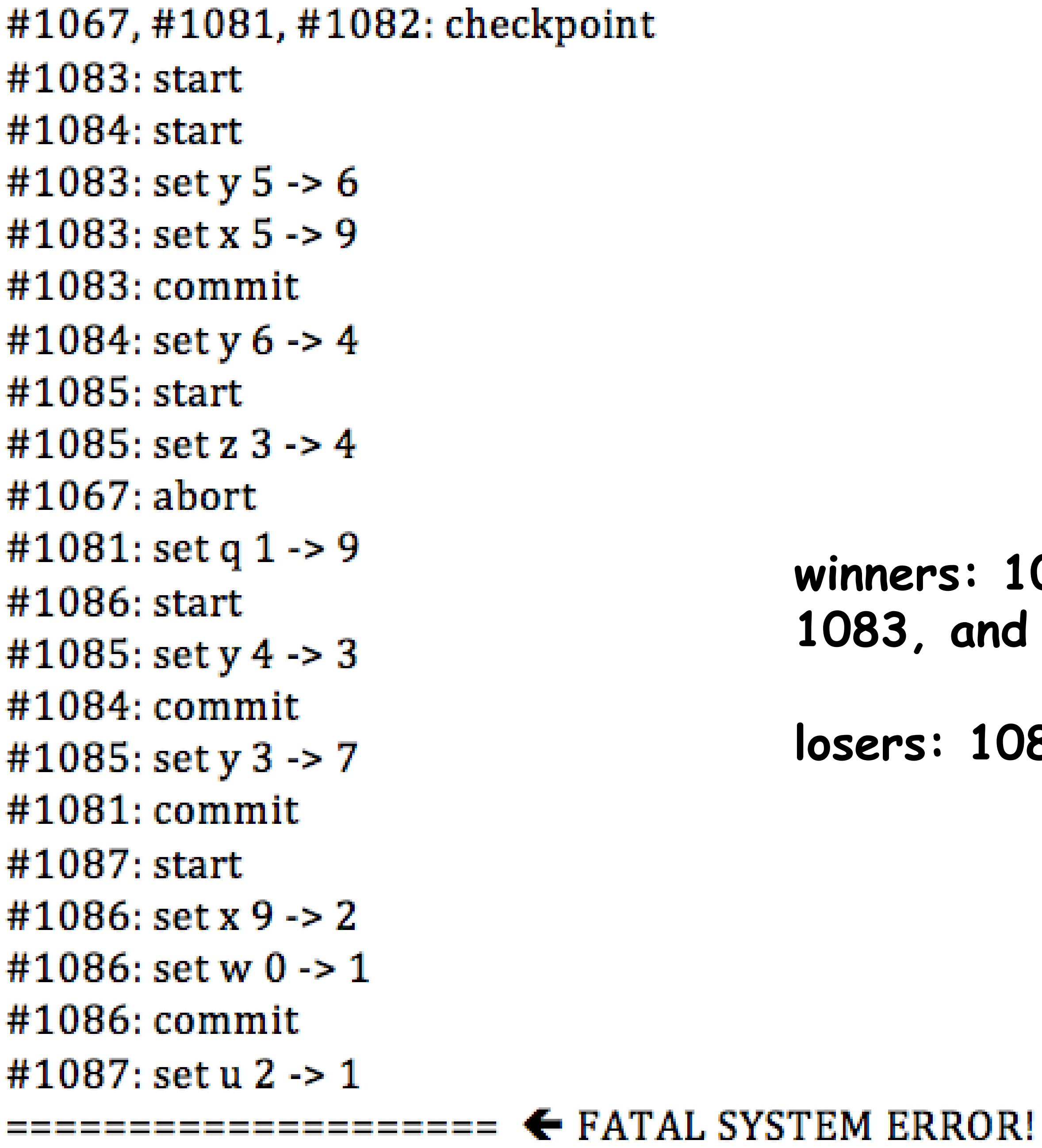
  New B -> old B
- OUTCOME 114

  COMMIT

(install the cell)

- END 114

```
#1067, #1081, #1082: checkpoint
#1083: start
#1084: start
#1083: set y 5 -> 6
#1083: set x 5 -> 9
#1083: commit
#1084: set y 6 -> 4
#1085: start
#1085: set z 3 -> 4
#1067: abort
#1081: set q 1 -> 9
#1086: start
#1085: set y 4 -> 3
#1084: commit
#1085: set y 3 -> 7
#1081: commit
#1087: start
#1086: set x 9 -> 2
#1086: set w 0 -> 1
#1086: commit
#1087: set u 2 -> 1
====================  ← FATAL SYSTEM ERROR!
```



```
#1067, #1081, #1082: checkpoint
#1083: start
#1084: start
#1083: set y 5 -> 6
#1083: set x 5 -> 9
#1083: commit
#1084: set y 6 -> 4
#1085: start
#1085: set z 3 -> 4
#1067: abort
#1081: set q 1 -> 9
#1086: start
#1085: set y 4 -> 3
#1084: commit
#1085: set y 3 -> 7
#1081: commit
#1087: start
#1086: set x 9 -> 2
#1086: set w 0 -> 1
#1086: commit
#1087: set u 2 -> 1
==================== 🡐 FATAL SYSTEM ERROR!
```

winners: 1067, 1081, 1084, 1083, and 1086

losers: 1082, 1085, and 1087