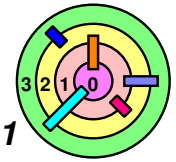
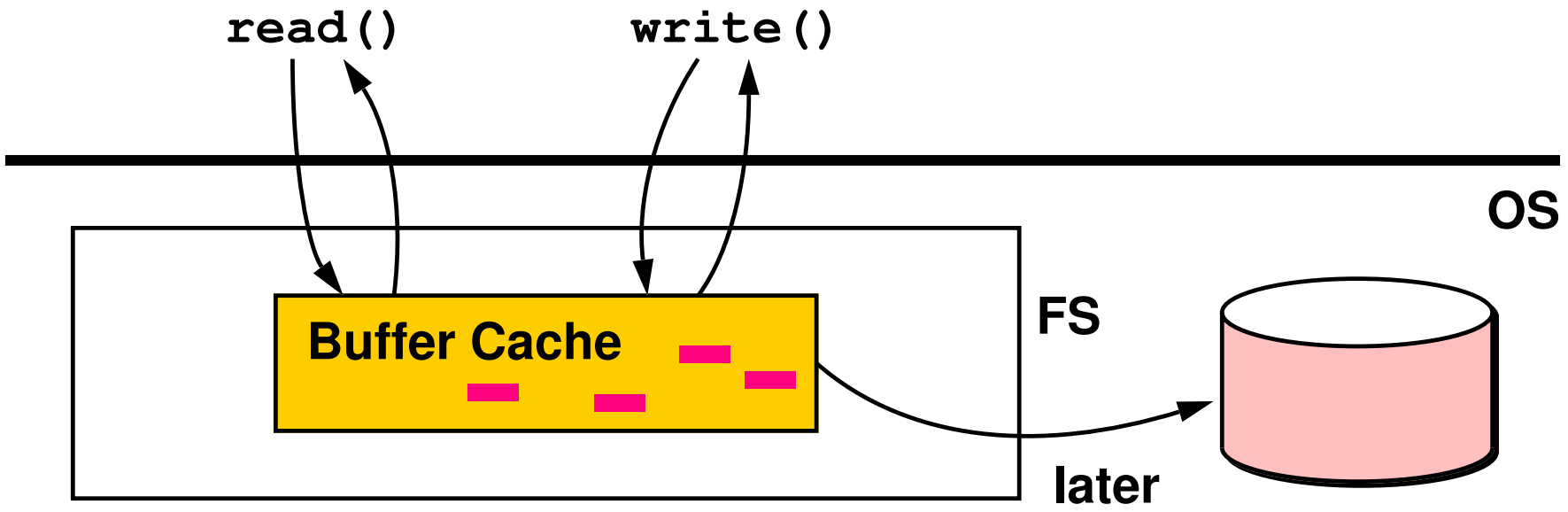
6.2 Crash Resiliency

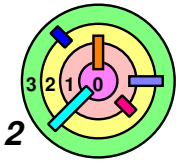
- What Goes Wrong
- Dealing with Crashes



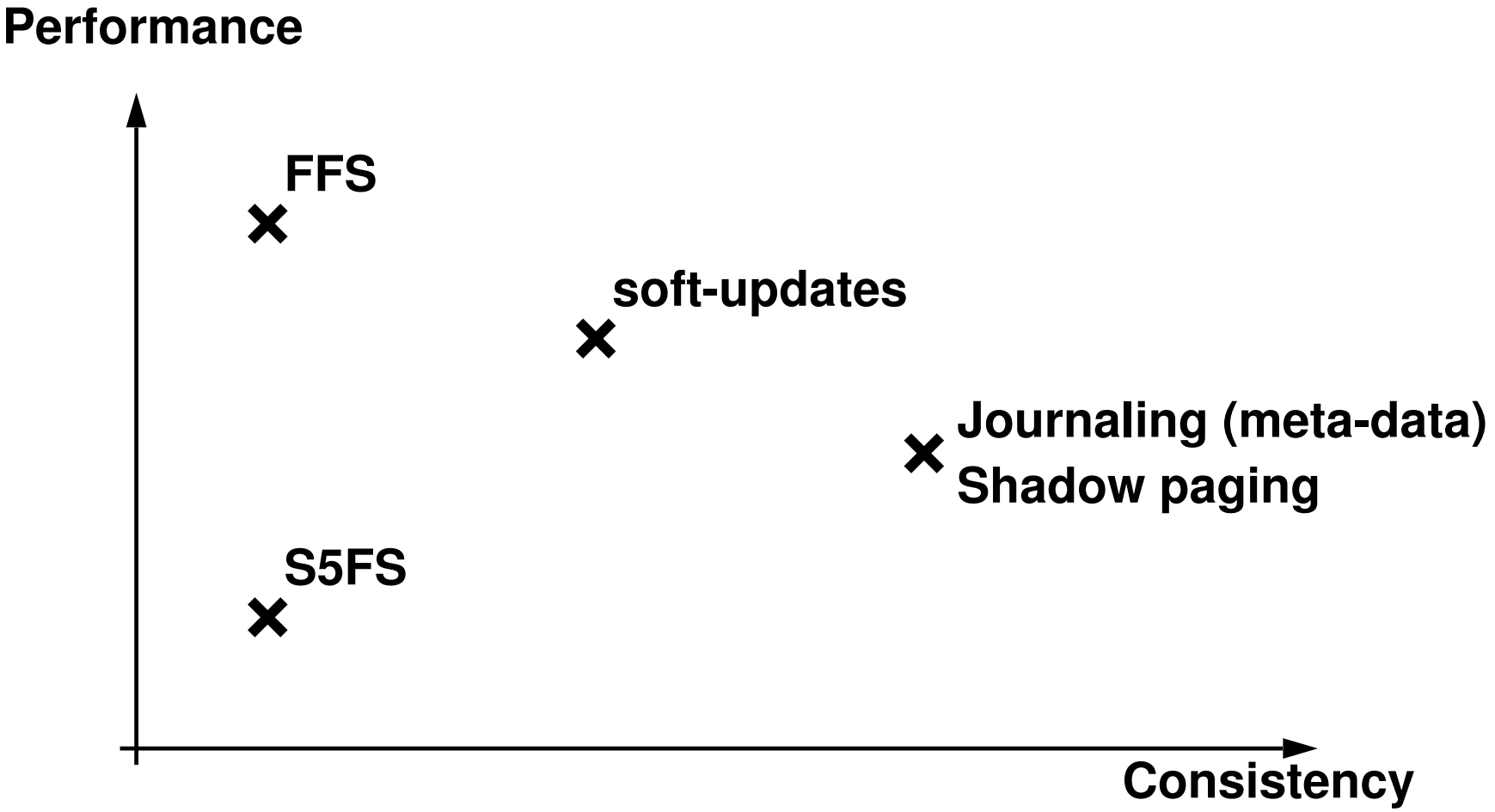
Buffer Cache



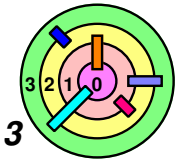
- ***Dirty/modified blocks*** **in buffer cache**
  - **disk blocks are read in and cached in the buffer cache**
    - **originally "clean/unmodified"**
  - **a write operation would modify a disk block in the buffer cache**
    - **the block is labeled *"dirty/modified"***
  - ***disk update:*** **the file system periodically gathers all the dirty blocks, update the disk, and clear the "dirty bits"**
    - **update is done *one disk block at a time***



Overveiw

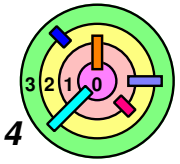


- soft-update provides *recoverable consistency*
- journaling and shadow paging provide *transactional consistency*

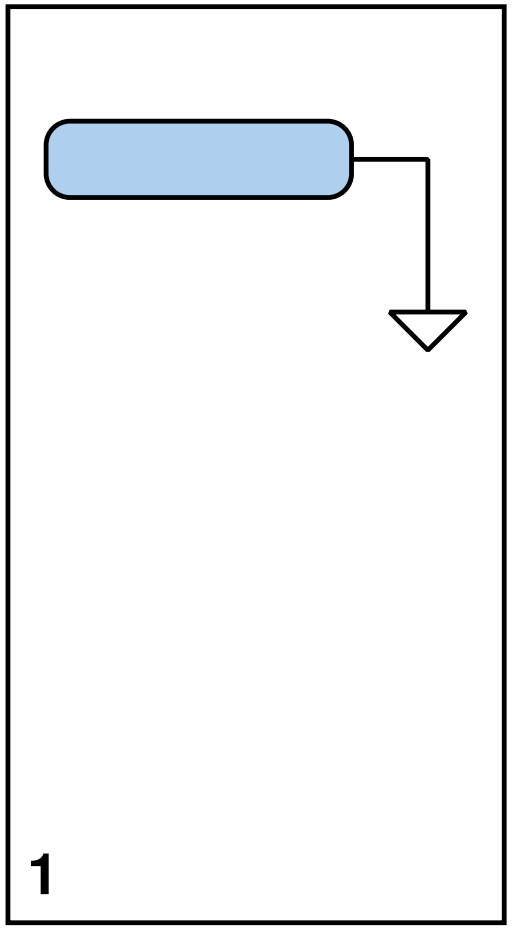


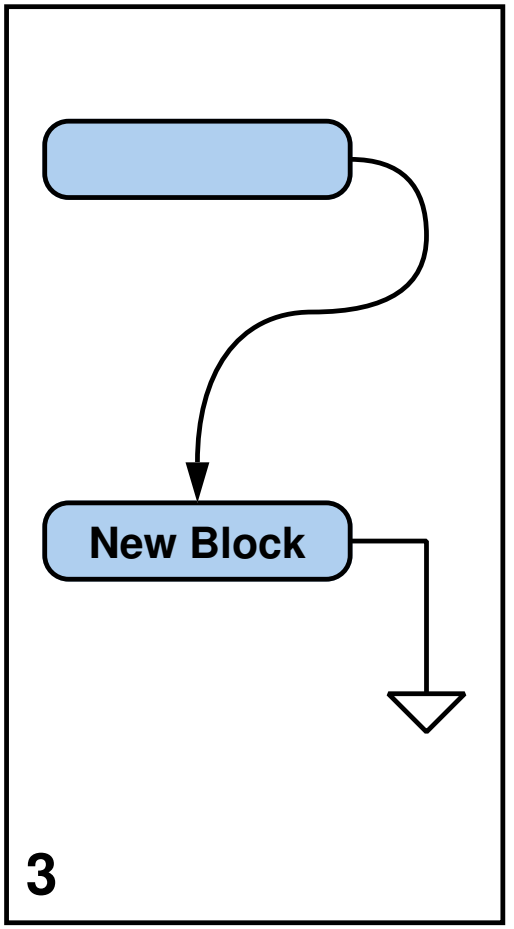
In the Event of a Crash ...

- Most recent updates did not make it to disk
  - is this a big problem?
  - equivalent to crash happening slightly earlier
    - but you may have received (and believed) a message:
      - "file successfully updated"
      - "homework successfully handed in"
      - "stock successfully purchased"
  - there's worse ...

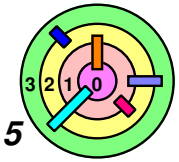


File-System Consistency (1)

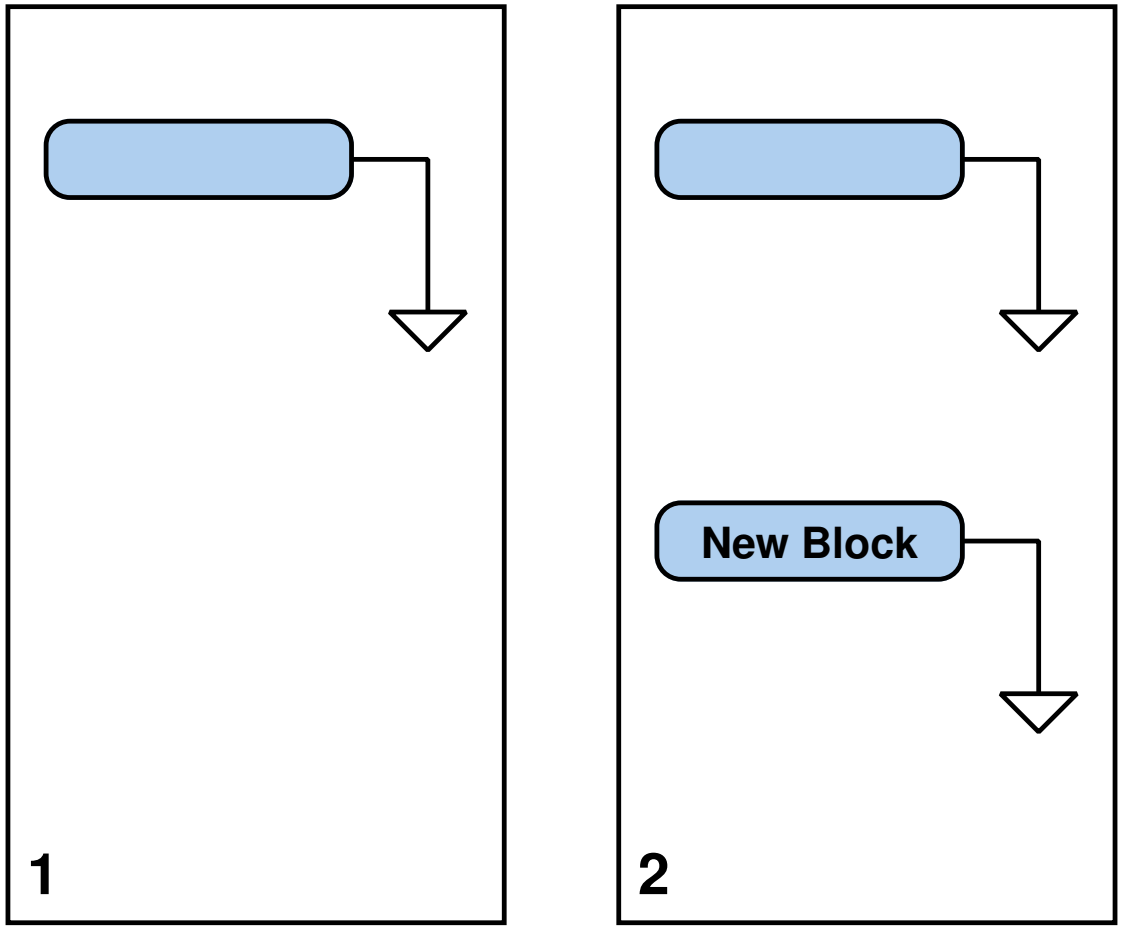




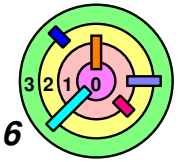
How to go from 1 to 3 *atomically*?



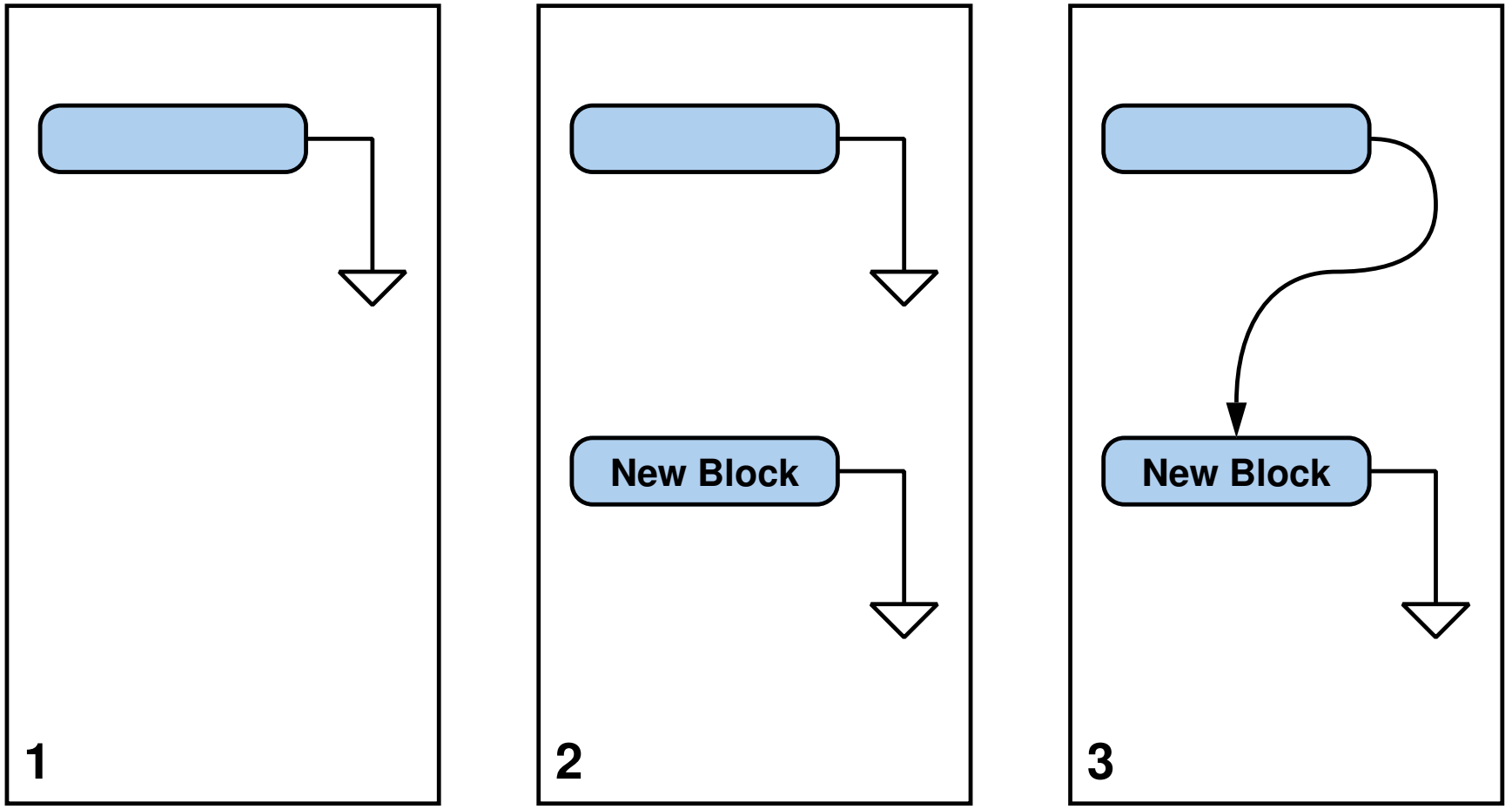
File-System Consistency (1)



- How to go from 1 to 3 *atomically*?
  - write the new block first

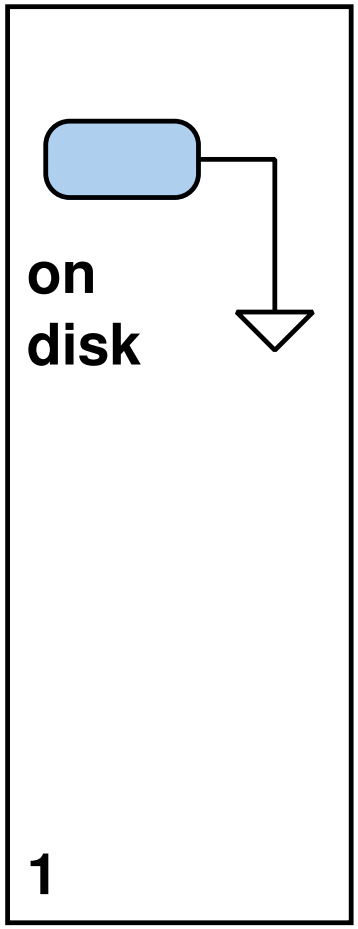


File-System Consistency (1)

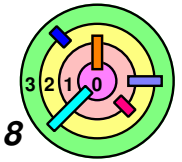


- How to go from 1 to 3 *atomically*?
  - write the new block first
  - then write new values into the old block

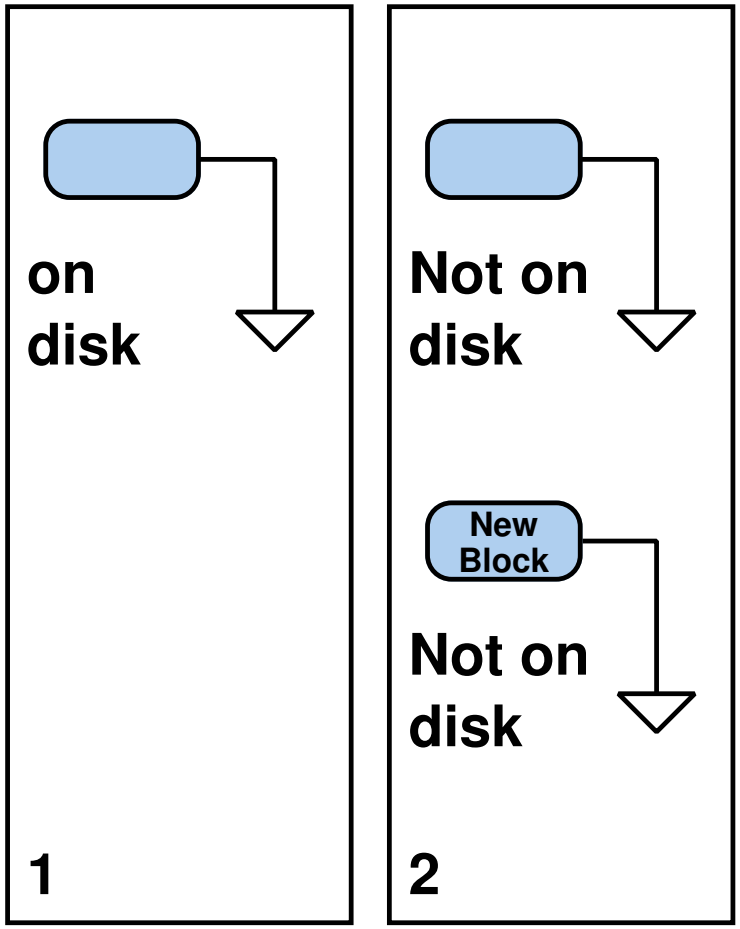
File-System Consistency (2)



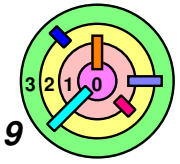
Problem: in S5FS and FFS, the lower level file system can sequence disk writes *in any order*



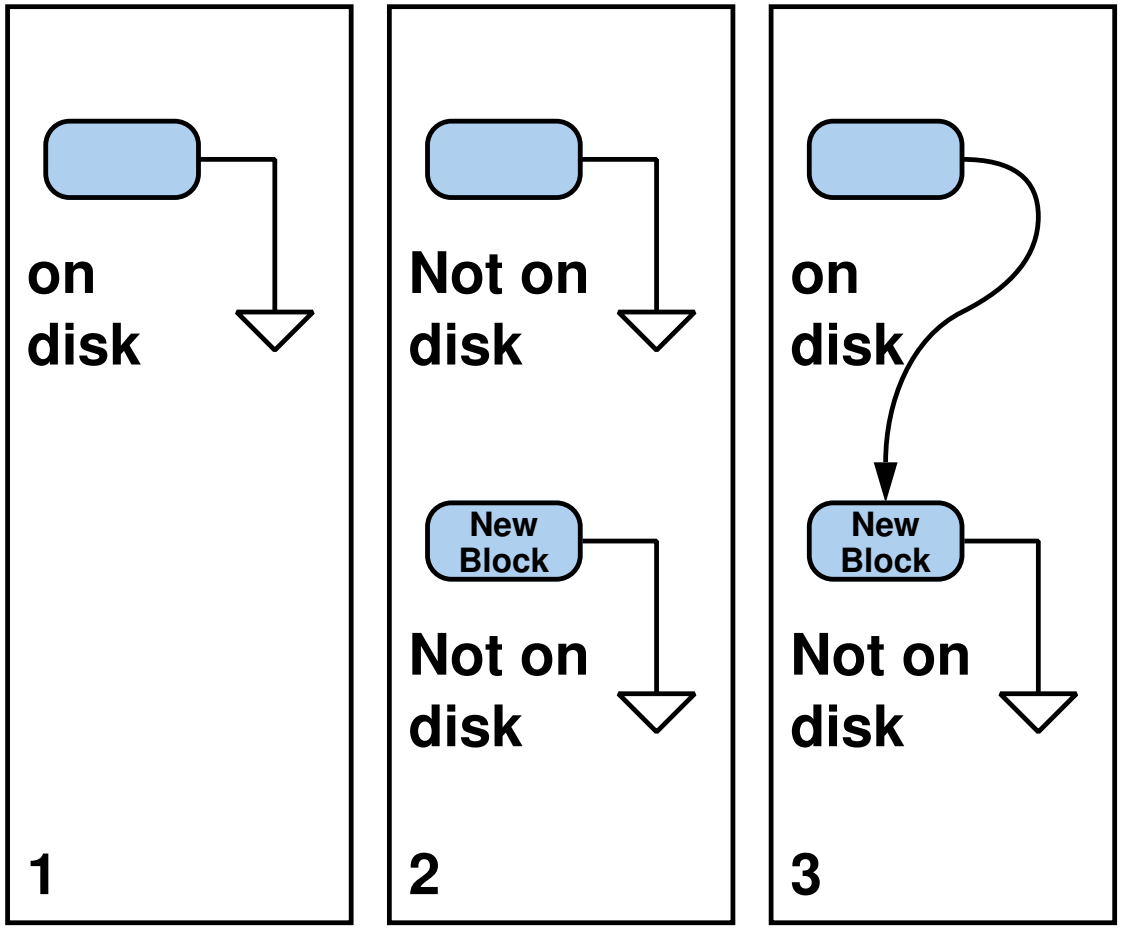
File-System Consistency (2)



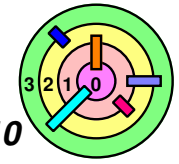
- Problem: in S5FS and FFS, the lower level file system can sequence disk writes *in any order*
  - write new values into the old block first



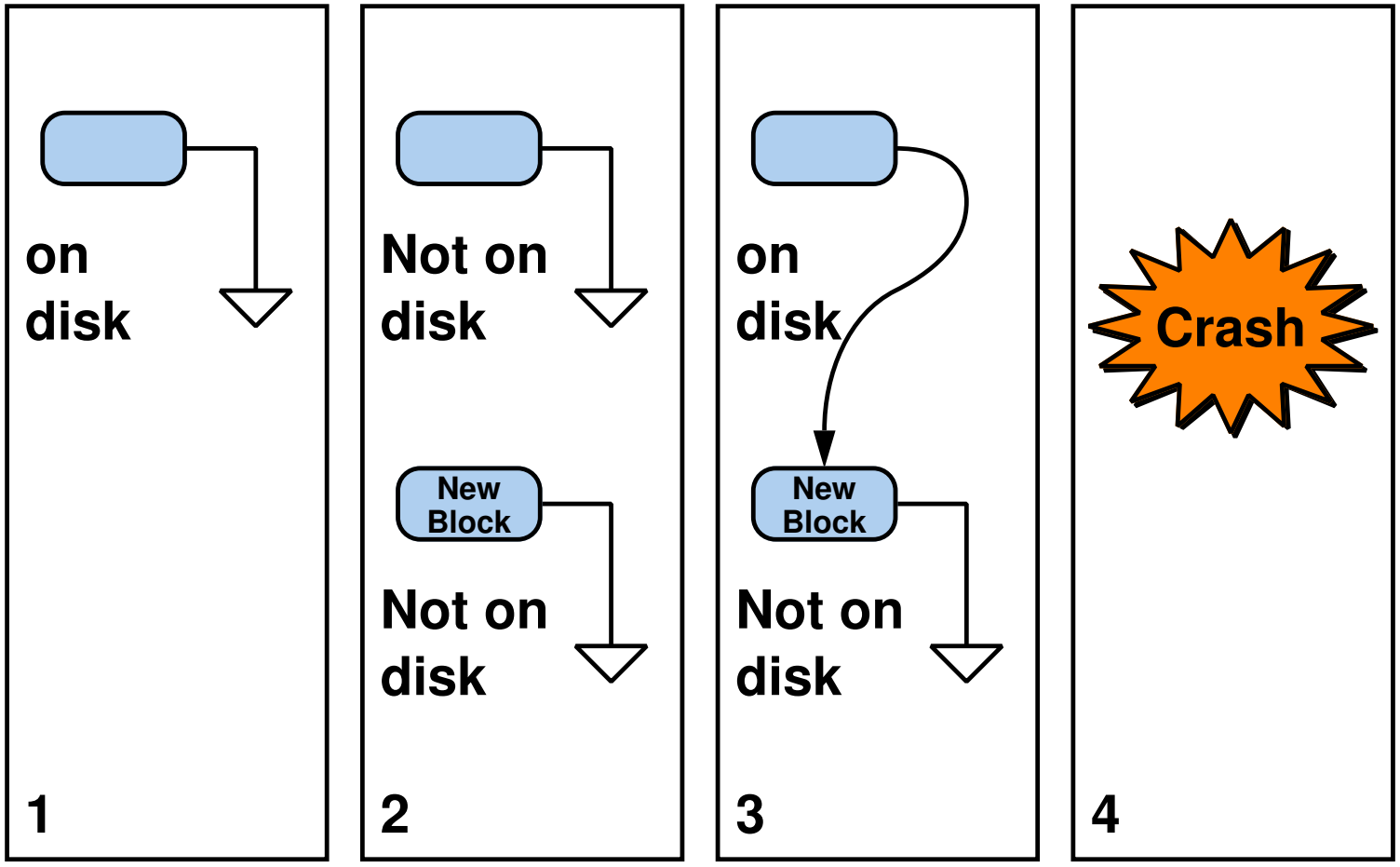
File-System Consistency (2)



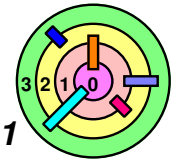
- Problem: in S5FS and FFS, the lower level file system can sequence disk writes *in any order*
  - write new values into the old block first



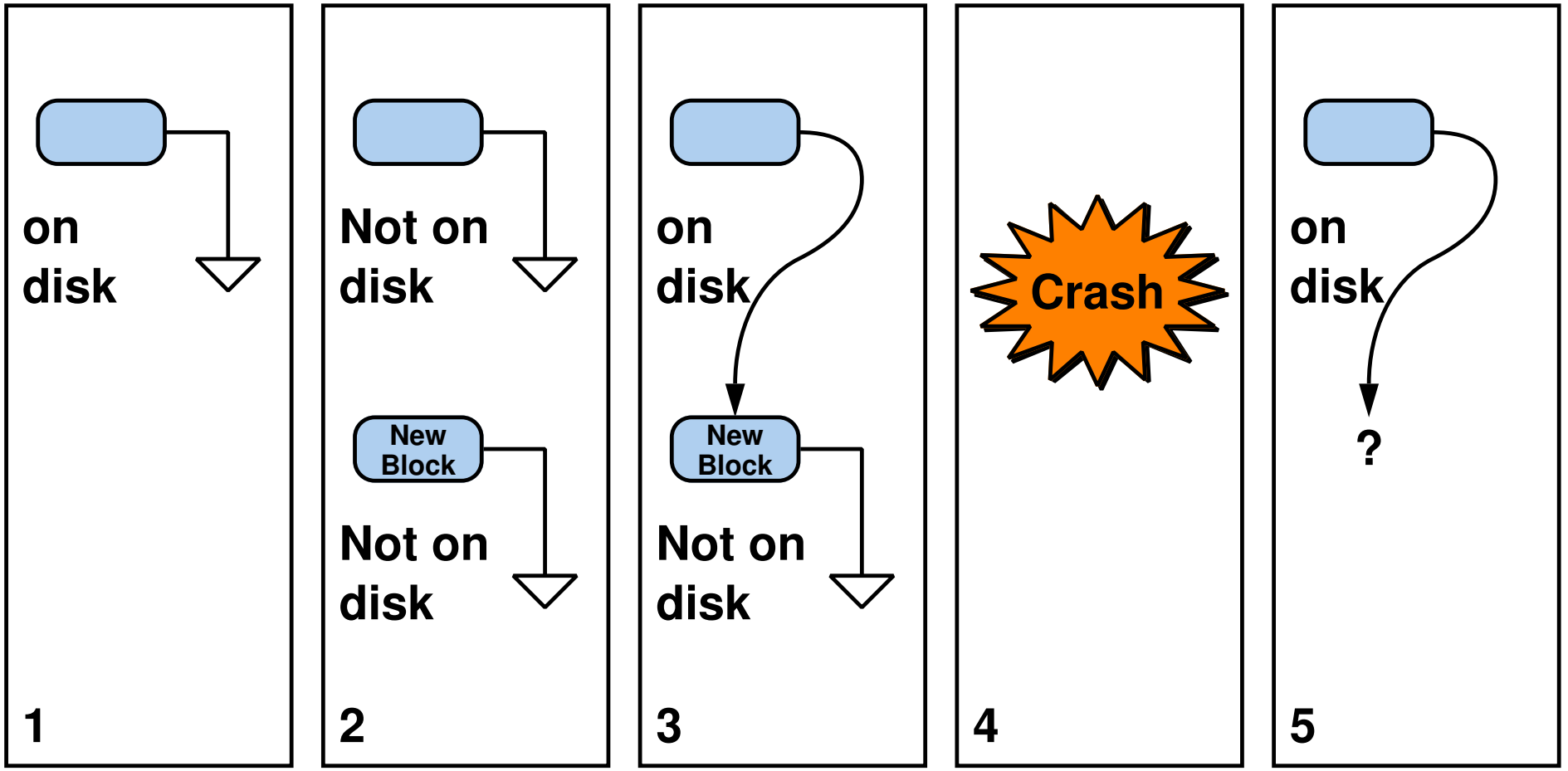
File-System Consistency (2)



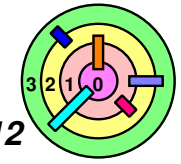
- **Problem: in S5FS and FFS, the lower level file system can sequence disk writes *in any order***
  - **write new values into the old block first**



File-System Consistency (2)

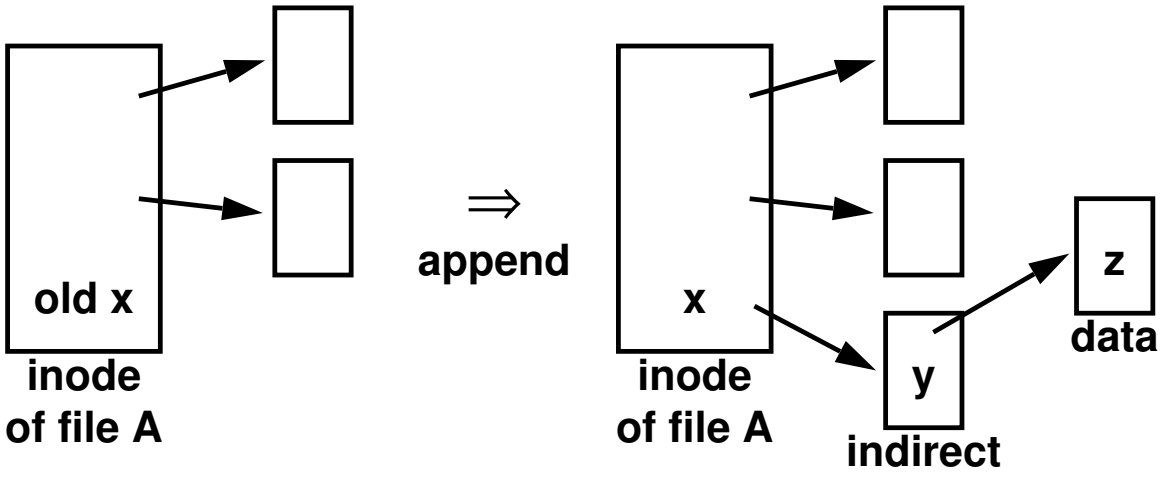


- Problem: in S5FS and FFS, the lower level file system can sequence disk writes *in any order*
  - write new values into the old block first



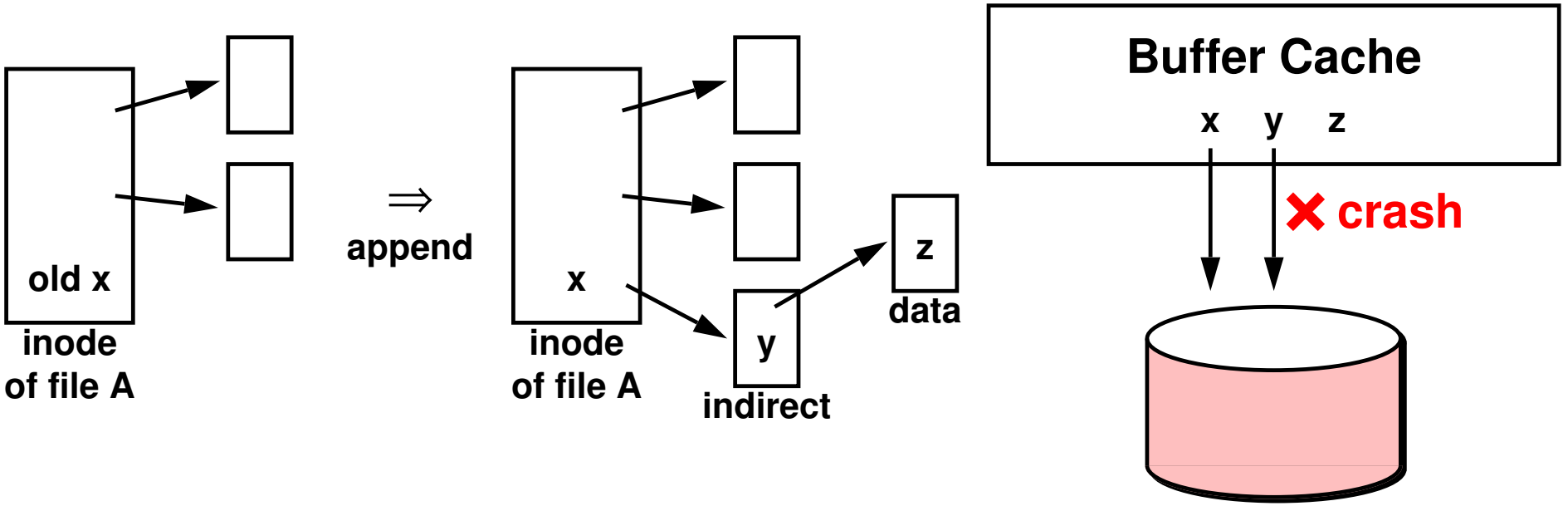
A More Realistic Example

- Let's say that you are appending to file A

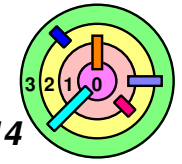


A More Realistic Example

Let's say that you are appending to file A

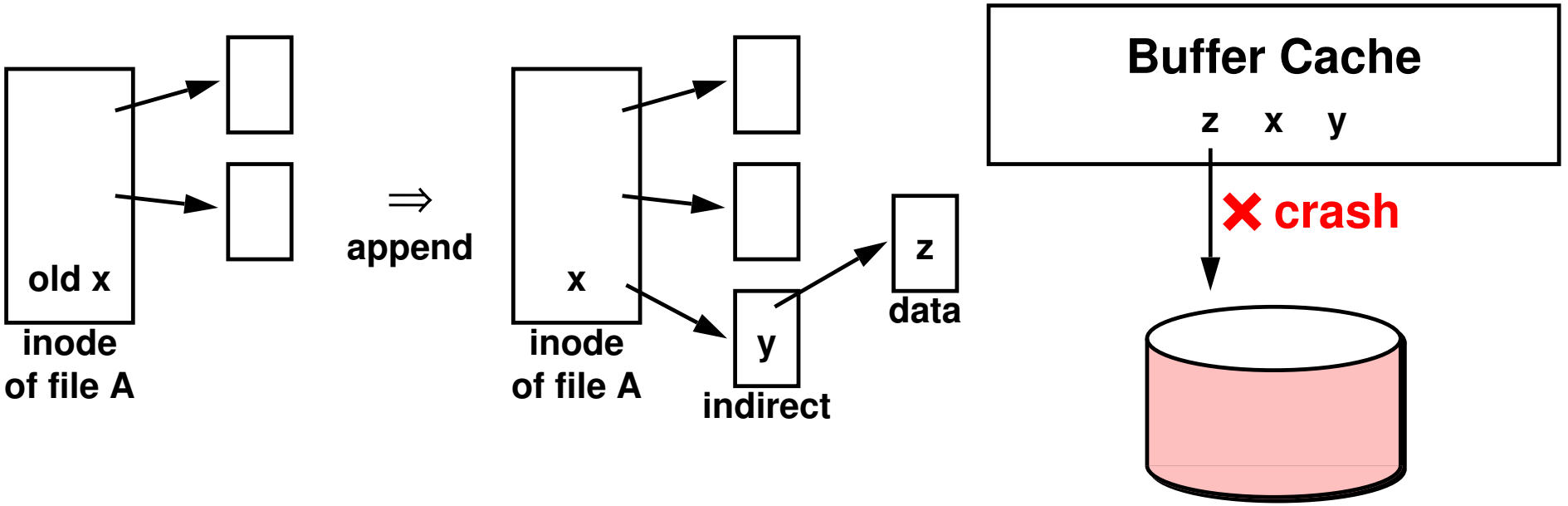


- the buffer cache does not know about the relationship among blocks x, y, and z
- techniques like locking (i.e., lock the disk so that it cannot crash when it's locked) won't work
- it's obvious that the solution is to make the disk update thread aware of the relationship among these blocks
  - but how? there are different approaches

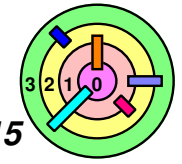


A More Realistic Example

- Let's say that you are appending to file A

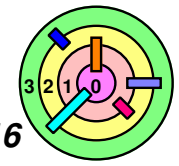


- what about this order and crash timing?
  - what about other combinations?



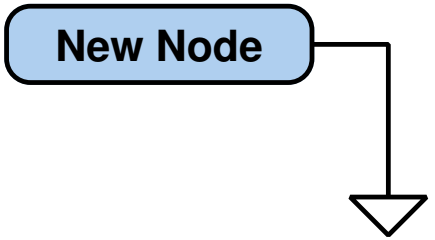
How to Cope ...

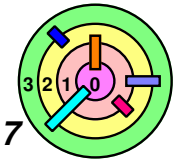
- Don't crash
  - not realistic
- Perform multi-step disk updates in an order such that disk is always consistent, i.e., the *consistency-preserving approach*
- Perform multi-step disk updates as *transactions*, i.e., implemented so that either all steps take effect or none do



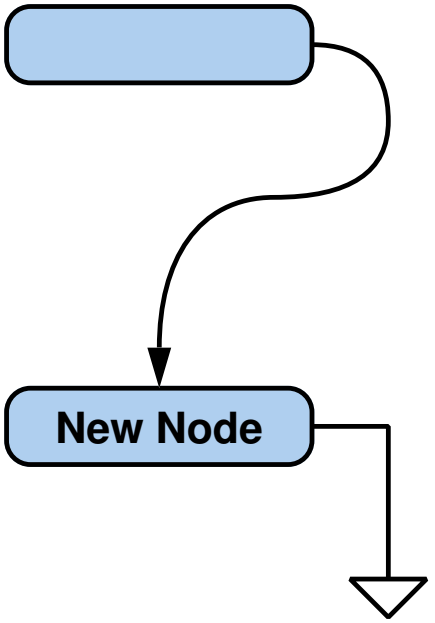
Maintaining Consistency

1) Write this synchronously to disk



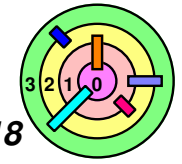


Maintaining Consistency

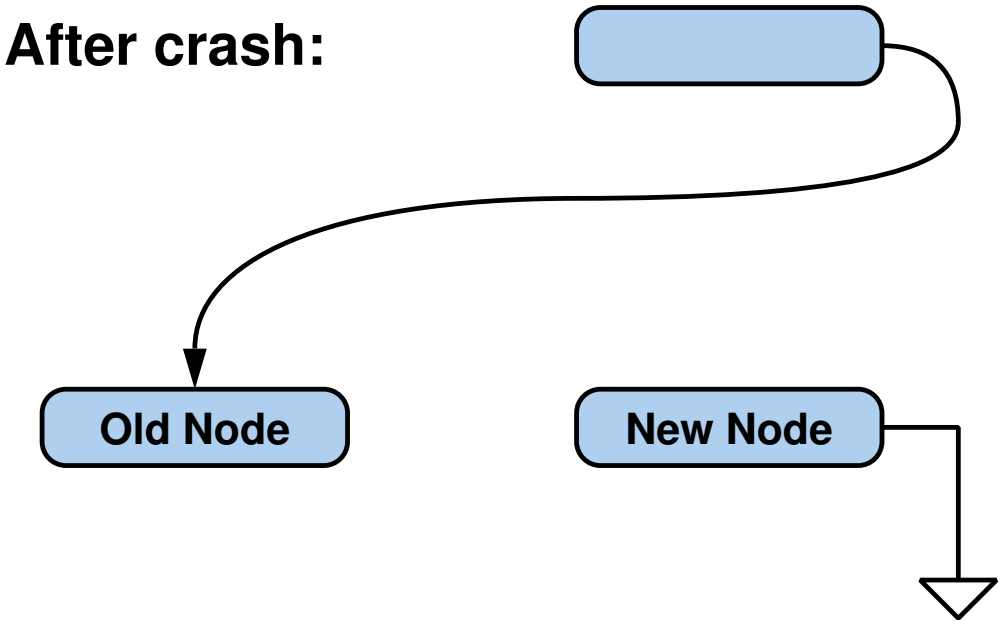


1) Write this *synchronously* to disk

2) Then write this *asynchronously* via the cache

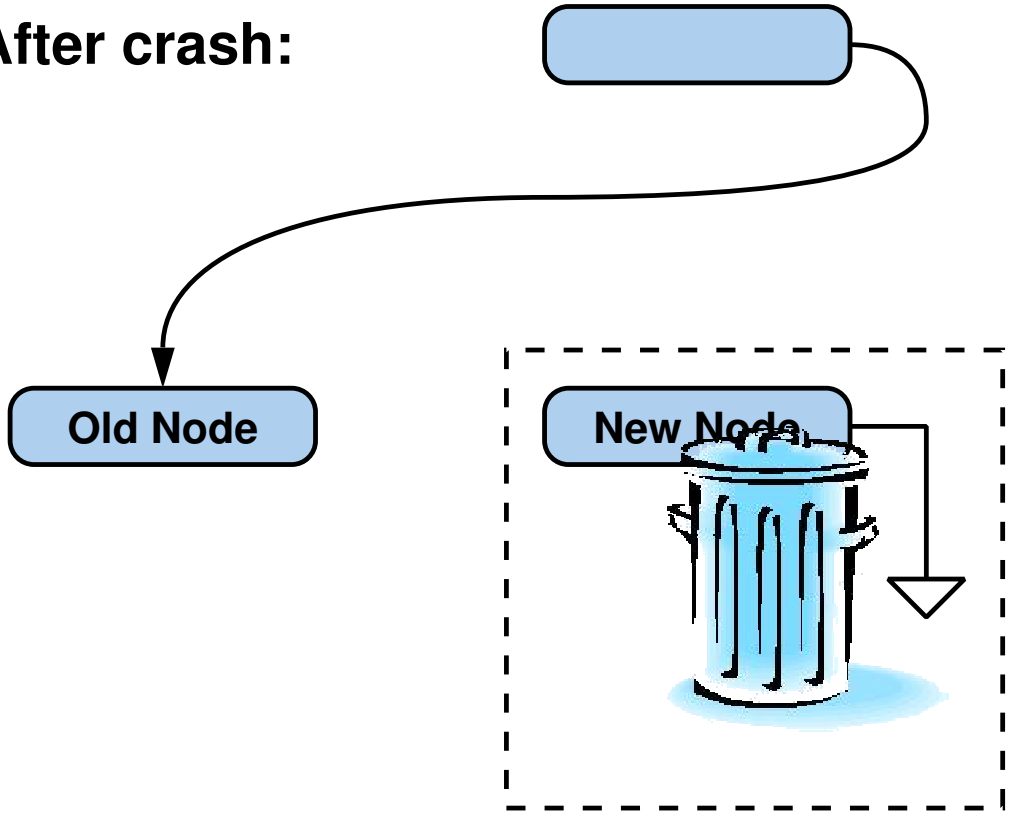


Innocuous Inconsistency



Innocuous Inconsistency

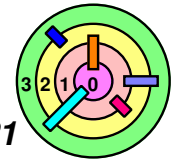
After crash:



- Innocuous inconsistency is acceptable
  - although need to *reclaim lost disk blocks*

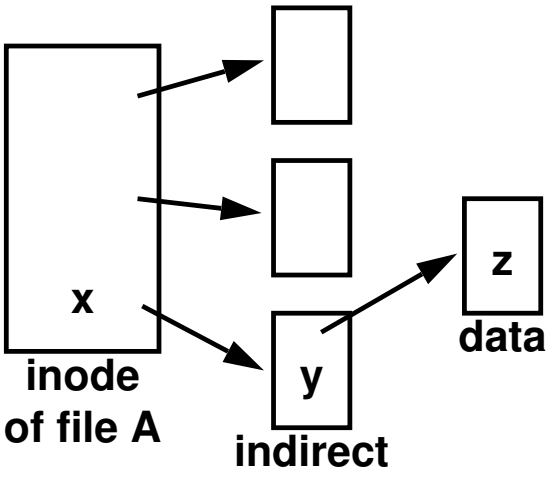
Soft Updates

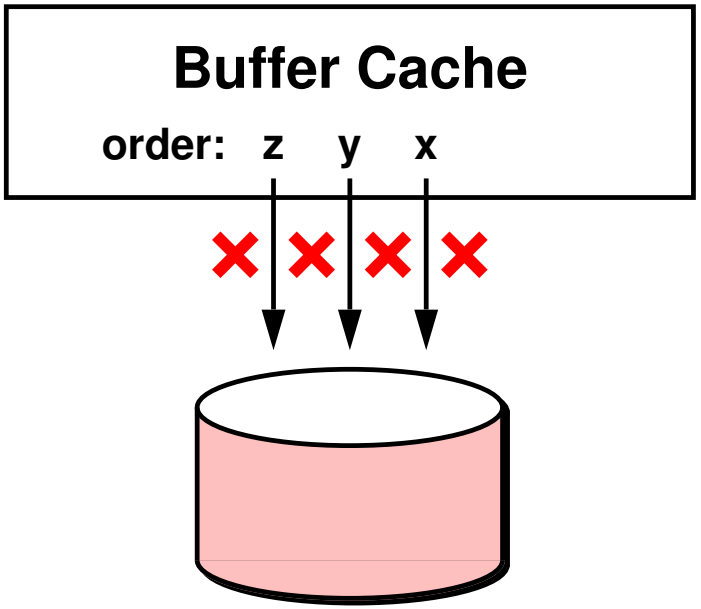
- **Main idea**
  - ***order disk operations*** **to** ***preserve meta-data consistency***
    - **innocuous inconsistency is considered ok**

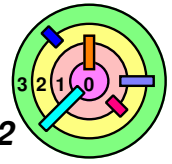


Back To The Example

- Let's say that you are appending to file A

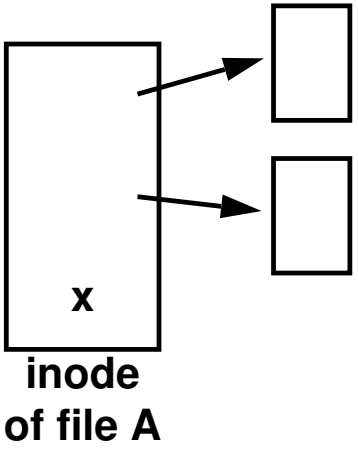


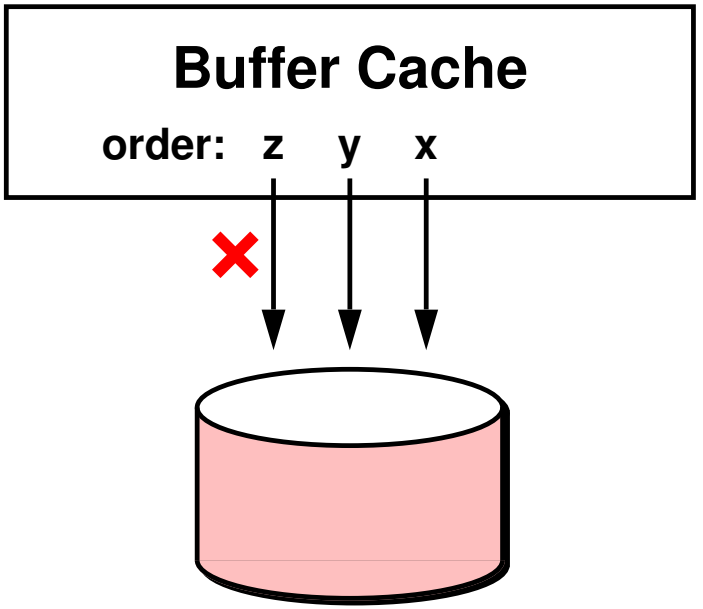




Back To The Example

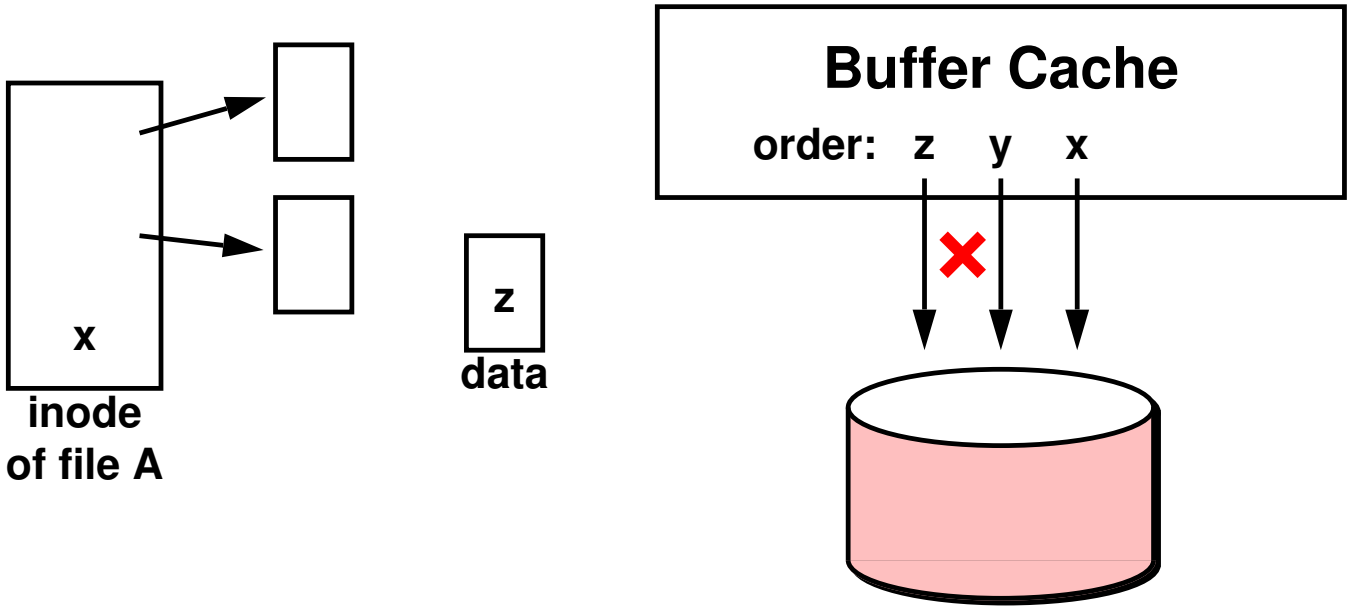
- Let's say that you are appending to file A



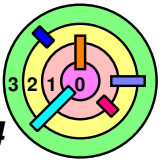


Back To The Example

- Let's say that you are appending to file A

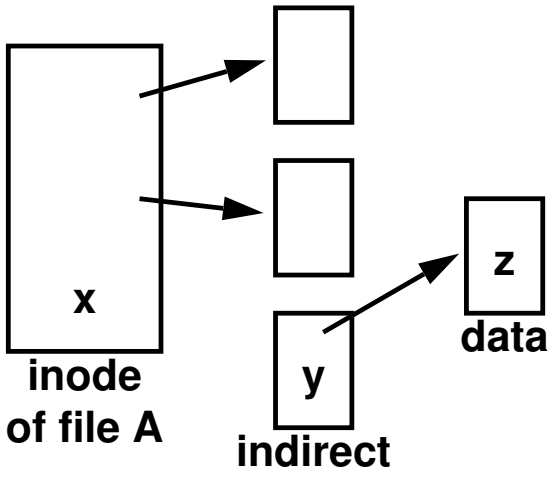


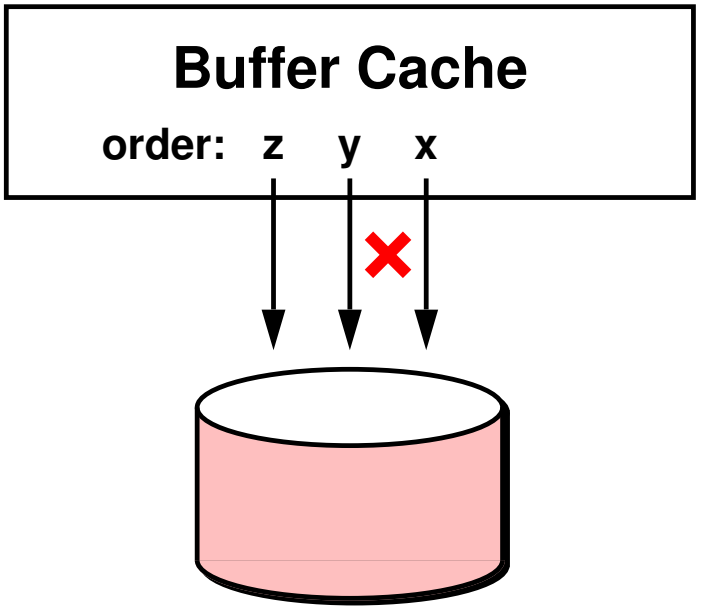
- is this bad?
  - how bad is it?



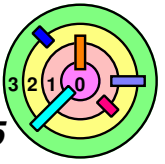
Back To The Example

- Let's say that you are appending to file A



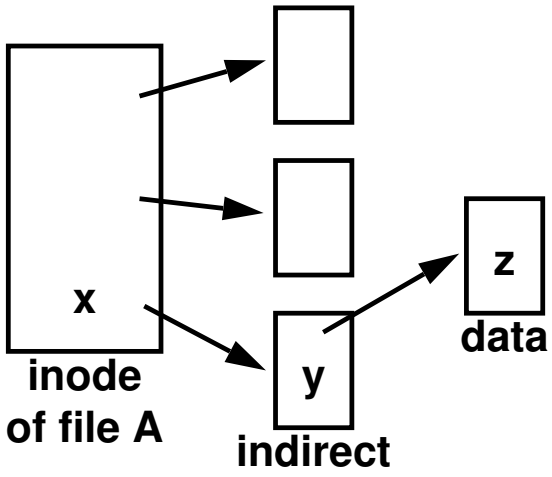


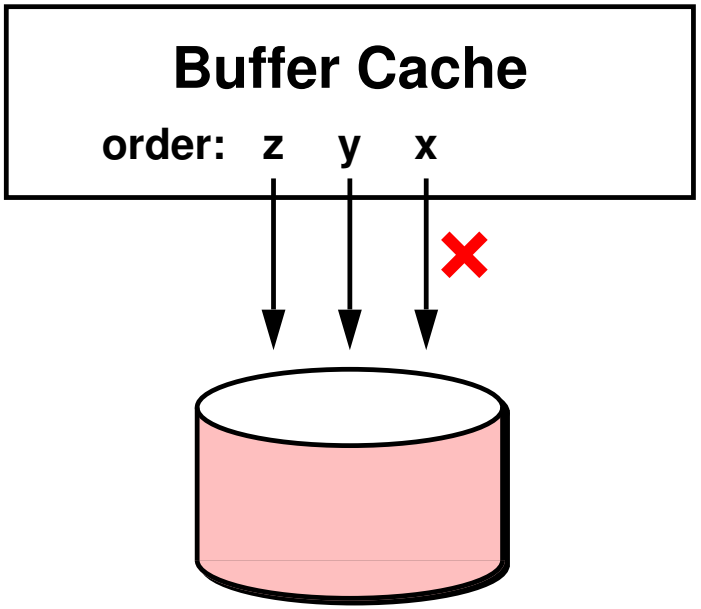
- is this bad?
  - how bad is it?



Back To The Example

- Let's say that you are appending to file A

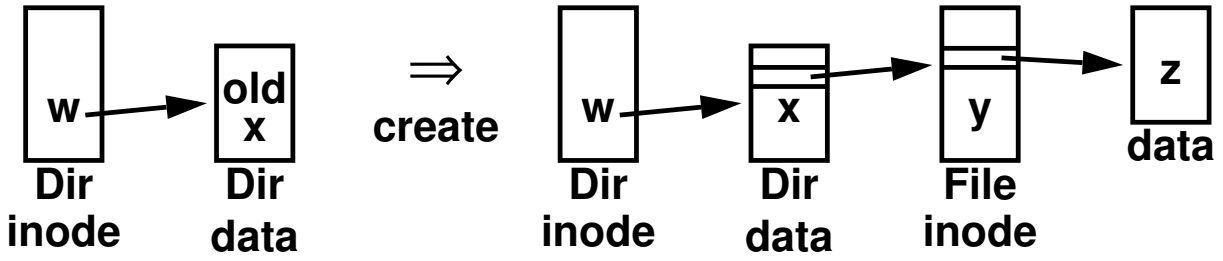




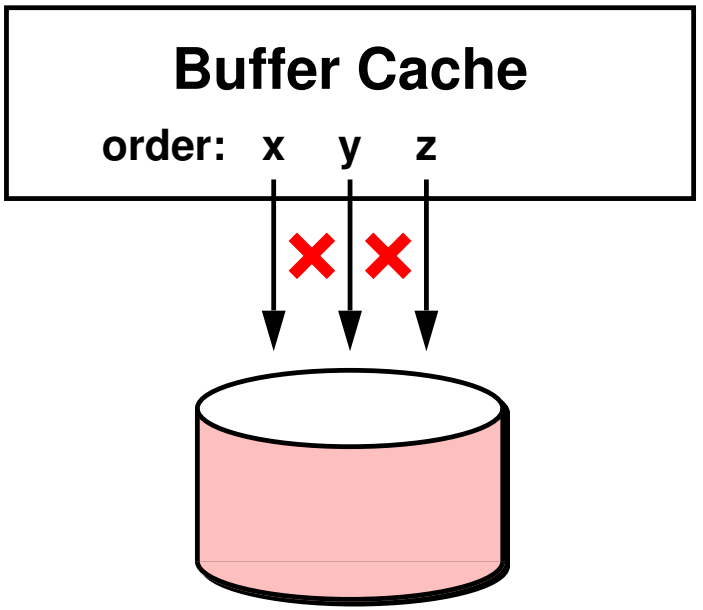
- is this bad?
  - no

Soft Update Example 2

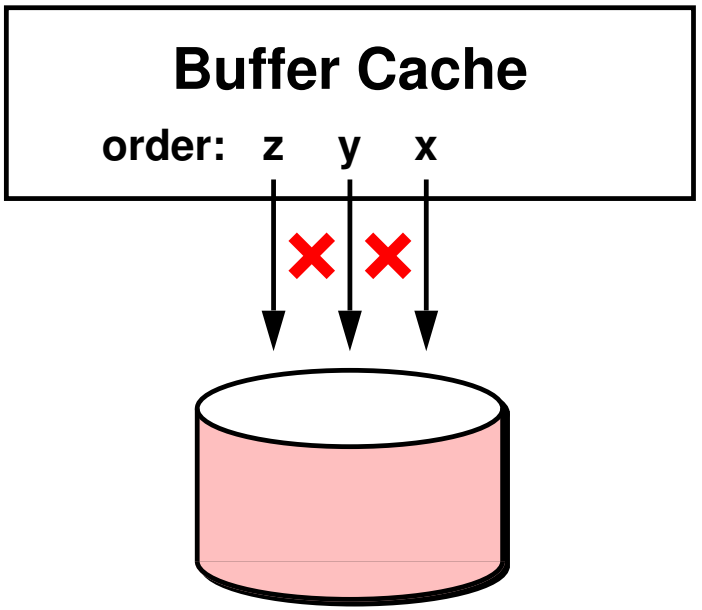
- Create a new file with one data block



Choice 1

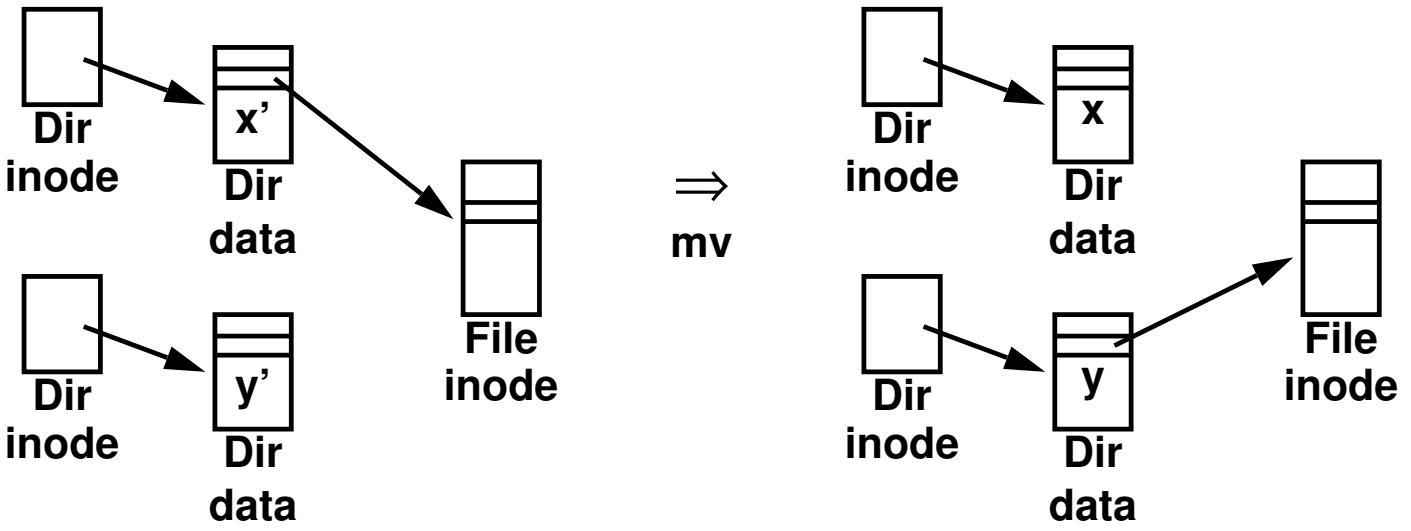


Choice 2

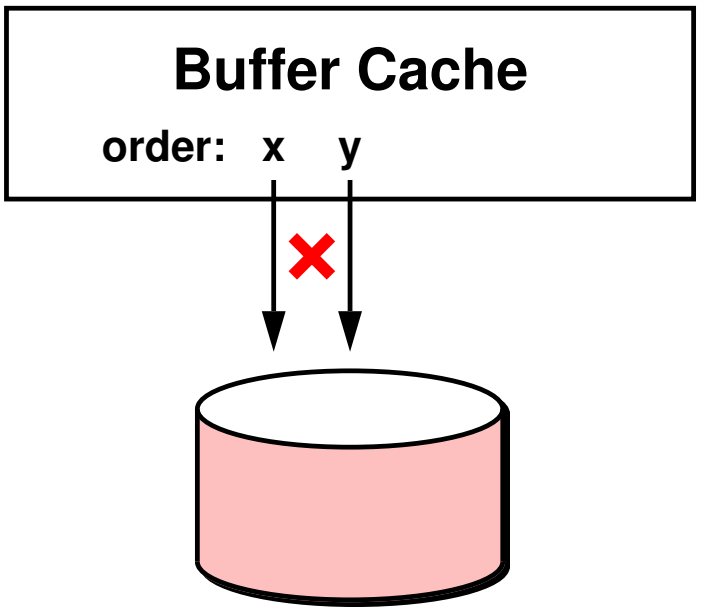


Soft Update Example 3

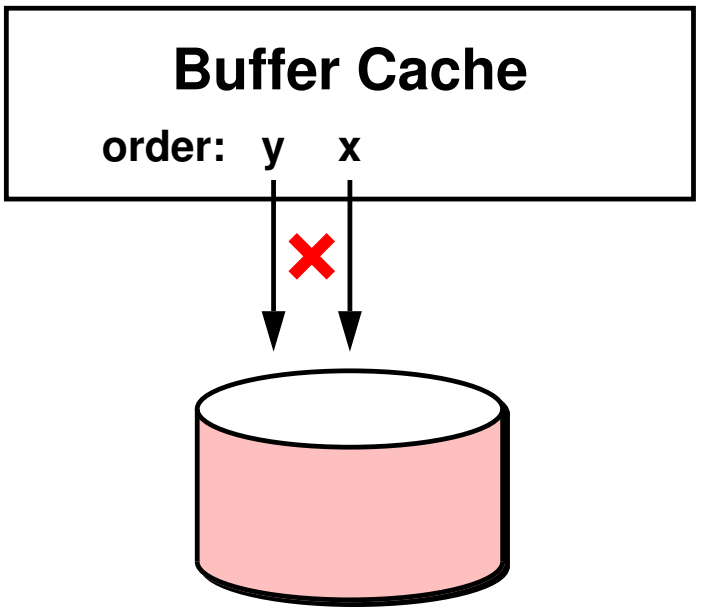
- Move a file

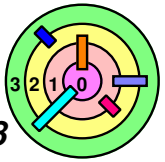


Choice 1



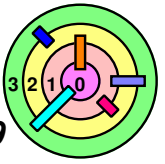
Choice 2



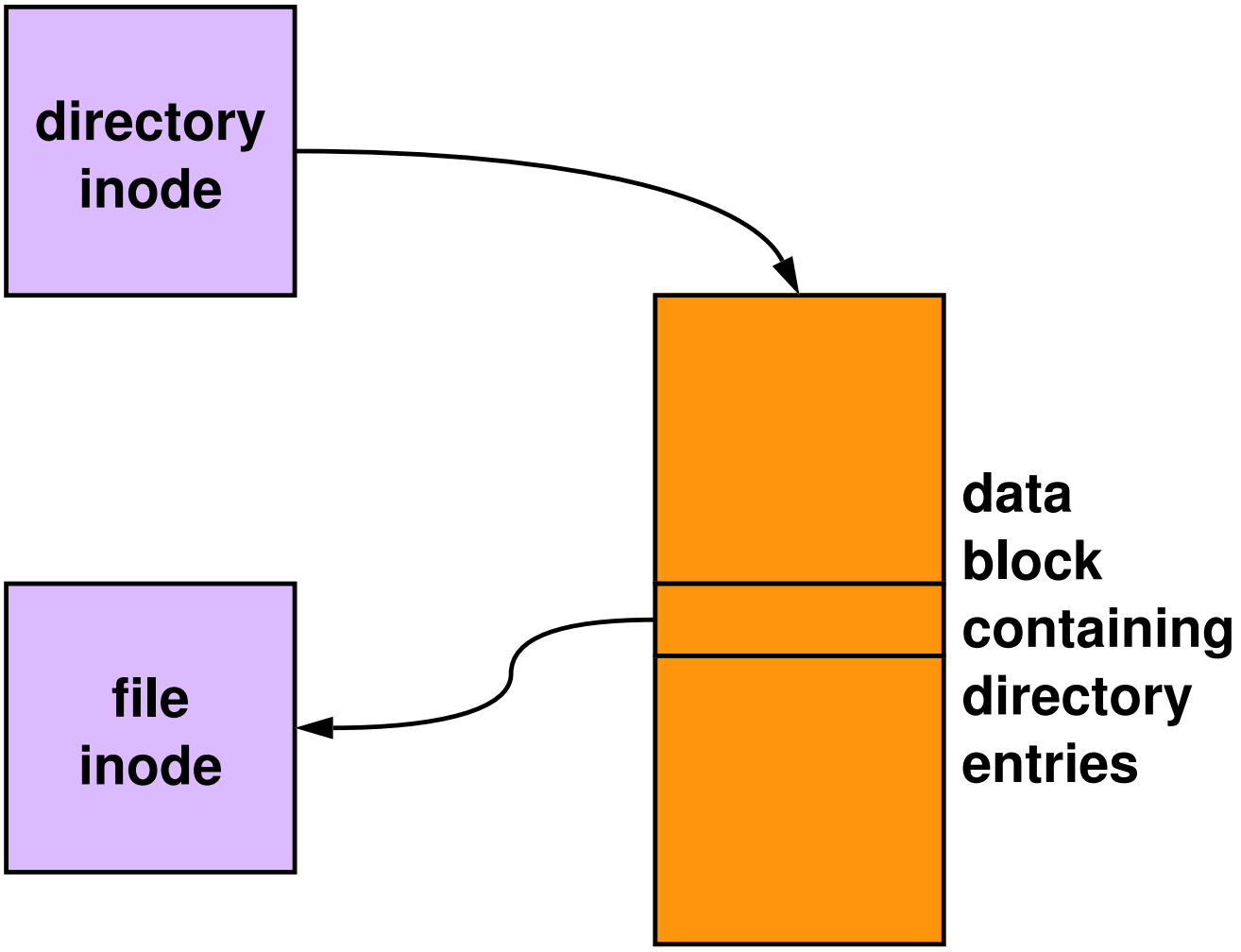


Soft Update

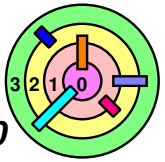
- An implementation of the consistency-preserving approach
  - the idea is simple:
    - update cache in an order that maintains consistency
    - write cache contents to disk in same order in which cache was updated
  - isn't, because reality is more complicated
    - (assuming speed is important)



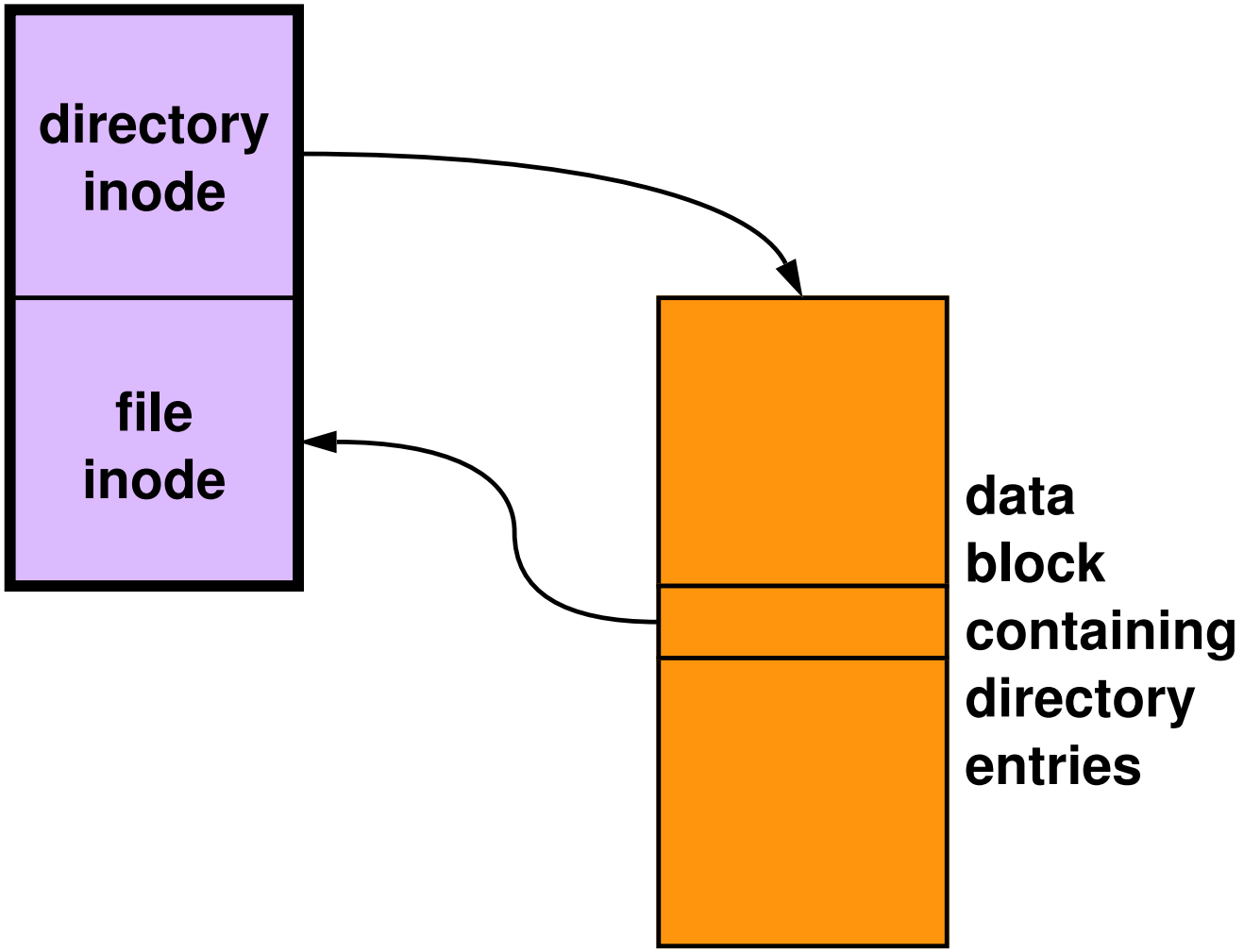
Which Order?



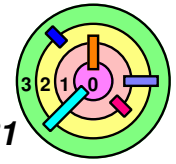
- this is easy
  - just use Topological Sort to figure it out



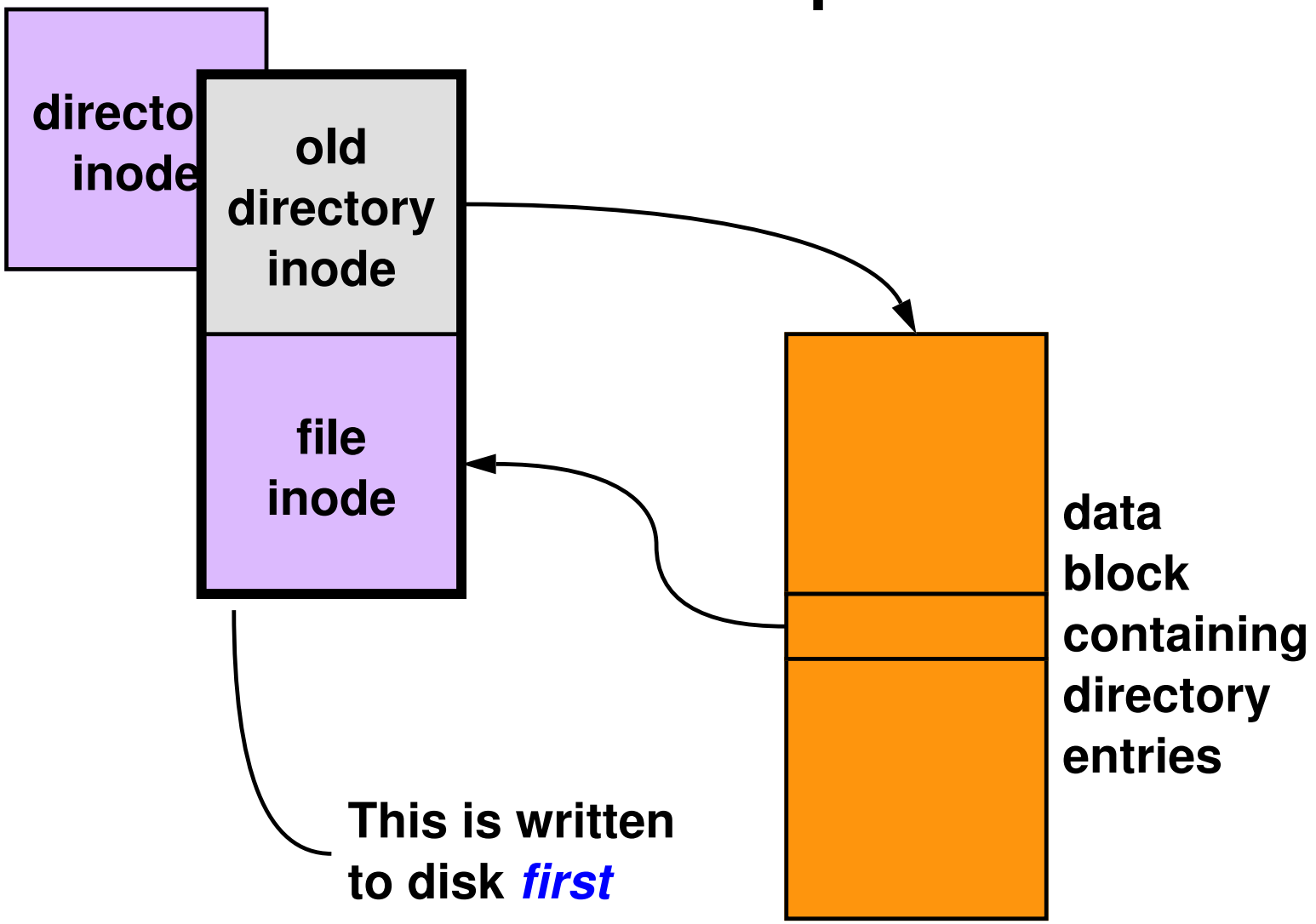
However ...



- circular dependency
  - in reality, in order to *save the number of disk writes*, multiple objects can be *packed* into a disk block



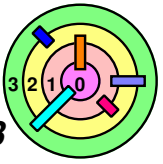
Soft Updates



- breaking circular dependency
  - 3 steps synchronous writes
  - slow

Soft Updates in Practice

- Implemented for FFS in 1994
- Used in FreeBSD's FFS
  - improves performance (over FFS with synchronous writes)
  - disk updates may be *many seconds* behind cache updates
  - need to *reclaim lost disk blocks* as background activity after the system restarts

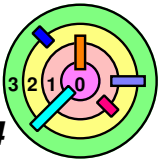


Transactions

- ***Group disk writes*** **into** ***transactions***
- **Classic example: transfer $100 from account 1 to account 2**
  - **need to decrease account 1 balance by $100**
  - **need to increase account 2 balance by $100**
  - **do this while satisfying ACID property**

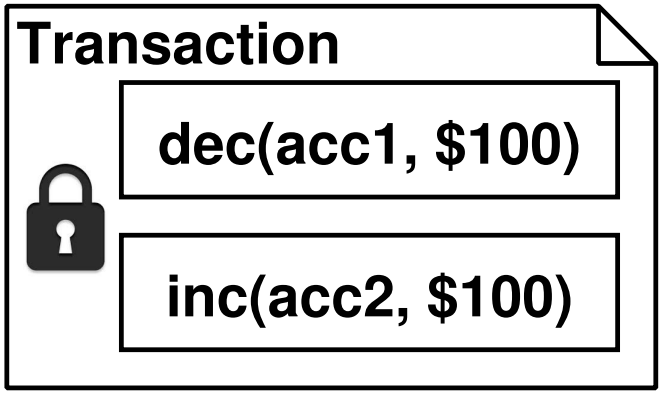
dec(acc1, $100)

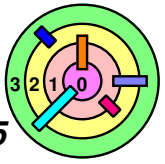
inc(acc2, $100)



Transactions

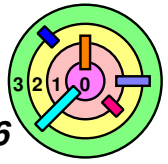
- *Group disk writes* into *transactions*
- Classic example: transfer $100 from account 1 to account 2
  - need to decrease account 1 balance by $100
  - need to increase account 2 balance by $100
  - do this while satisfying ACID property





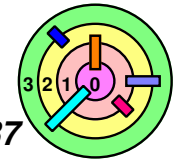
Transactions

- A *transaction* has the *"ACID" property:*
  - *atomic*
    - all or nothing
  - *consistent*
    - take system from one consistent state to another
  - *isolated*
    - have no effect on other transactions until committed
  - *durable*
    - persists



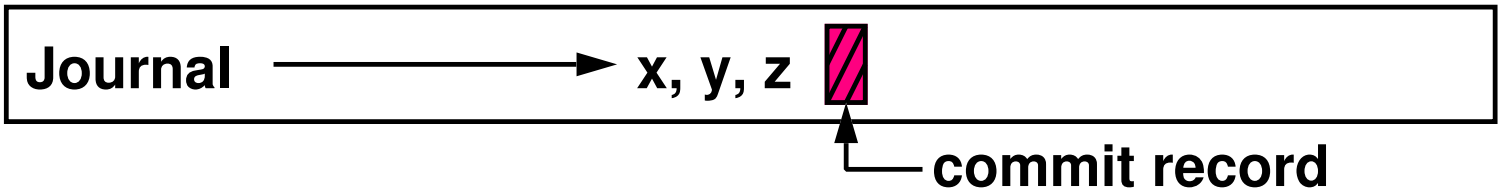
How?

- **Journaling**
  - before updating disk with steps of transaction:
    - record previous contents: *undo journaling*
      - *"before images"* of disk blocks are written into the journal
    - record new contents: *redo journaling*
      - *"after images"* of disk blocks are written into the journal
- **Shadow paging**
  - steps of transaction written to disk, but old values remain
  - single write switches old state to new

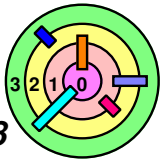


Journaling

- A *journal* is a *separate part of the disk*
  - can add journaling to *any* file system
- A *journal* is *append-only*, like a log
  - for a *redo journal*, append what you are *going to write* to the main part of the disk (i.e., the file system)
  - append a *commit record*
    - a commit record is *one disk block in size*
    - the disk guarantees that a commit record is either written to the disk or not (nothing in between)

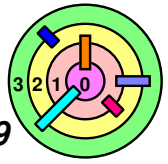


- When it's time to update the file system, *write to journal first*
  - write data to file system only *after* the commit record is written to the journal



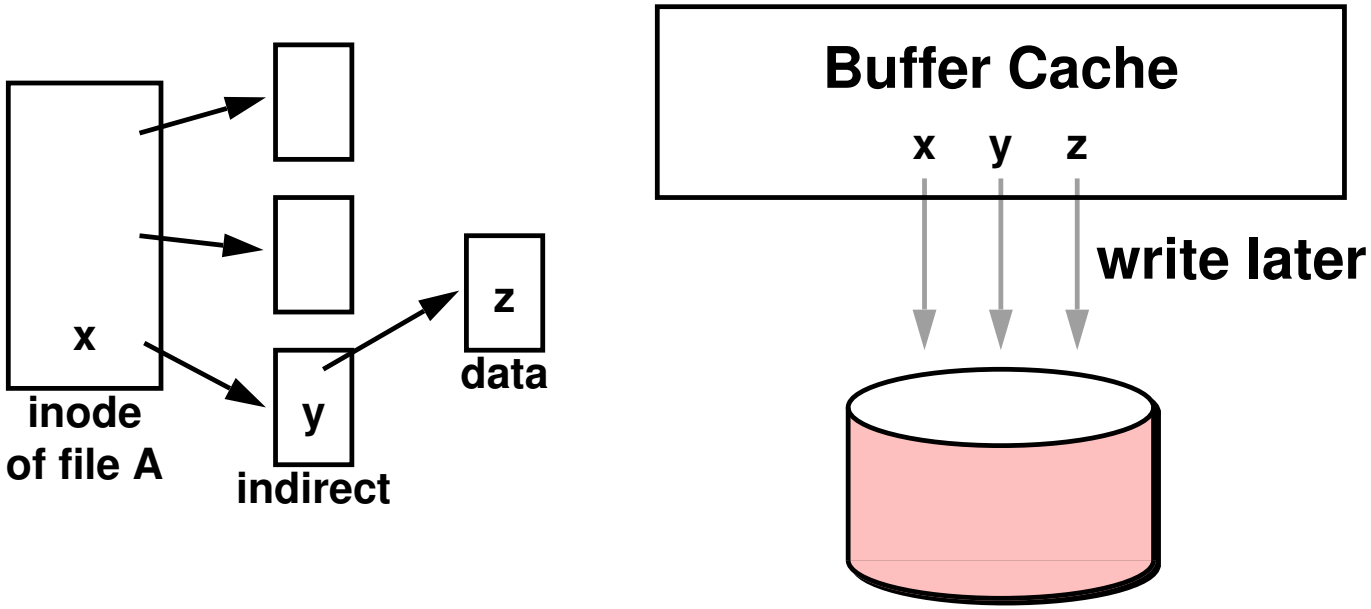
Recovery

- The system can crash at any time
  - data in the file system may be inconsistent when the system reboots
  - recovery will take the file system into a *consistent state*
    - at a *transaction boundary*
- If a *redo journal* is used, recovery involves
  - finding all committed transactions
  - redo (replay) all these transactions
    - if system crashes in the middle of a recovery, no harm is done
    - can perform recovery again and again
      - copying a disk block to the file system is *idempotent*, i.e., doing it twice has the same effect as doing it once
      - dec(acc1, $100) is *not idempotent*
- After recovery, the state of the file system is what it was at the end of the *last committed transaction*
  - therefore, in a consistent state

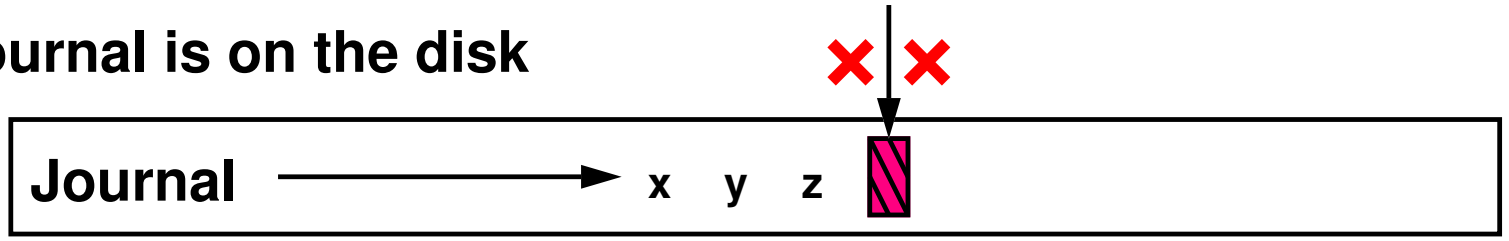


Back To The Example

- Let's say that you are appending to file A



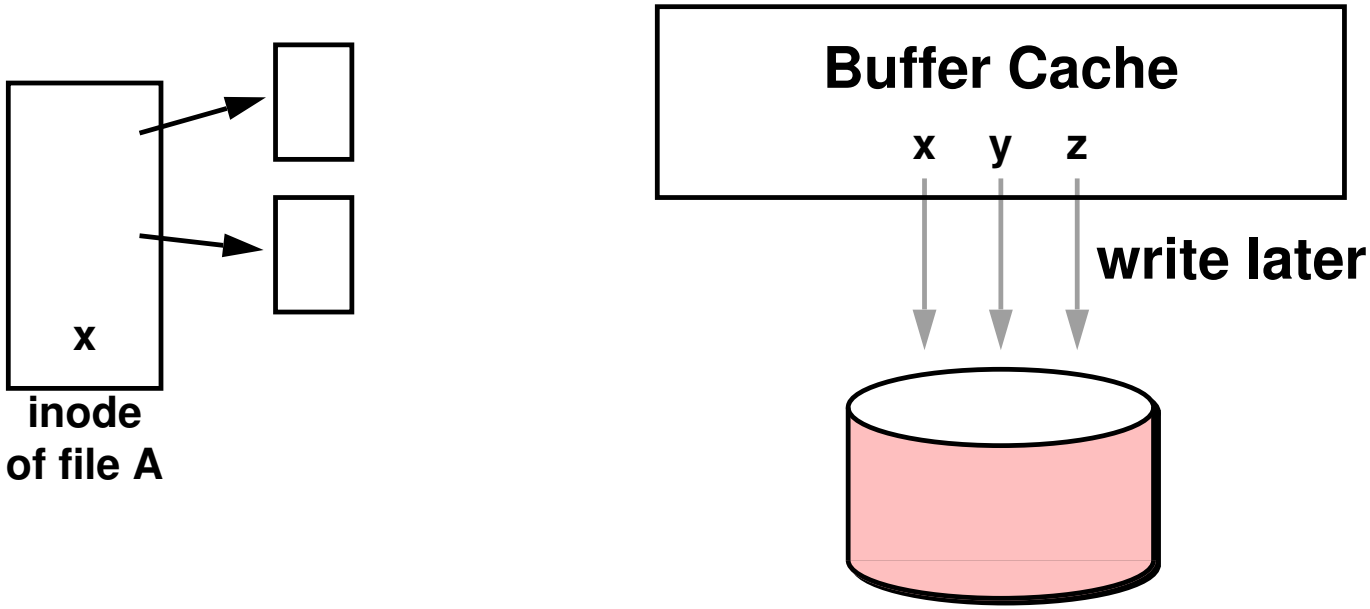
- The journal is on the disk



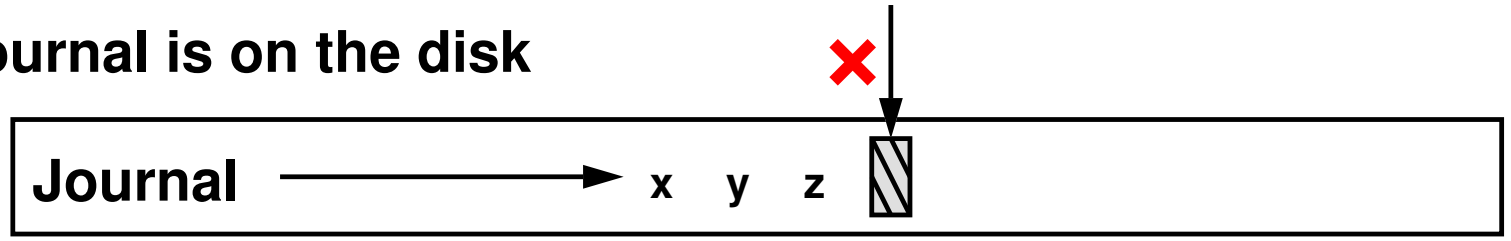
  - question is, did failure happen *before* or *after* the *commit*

Back To The Example

- Let's say that you are appending to file A



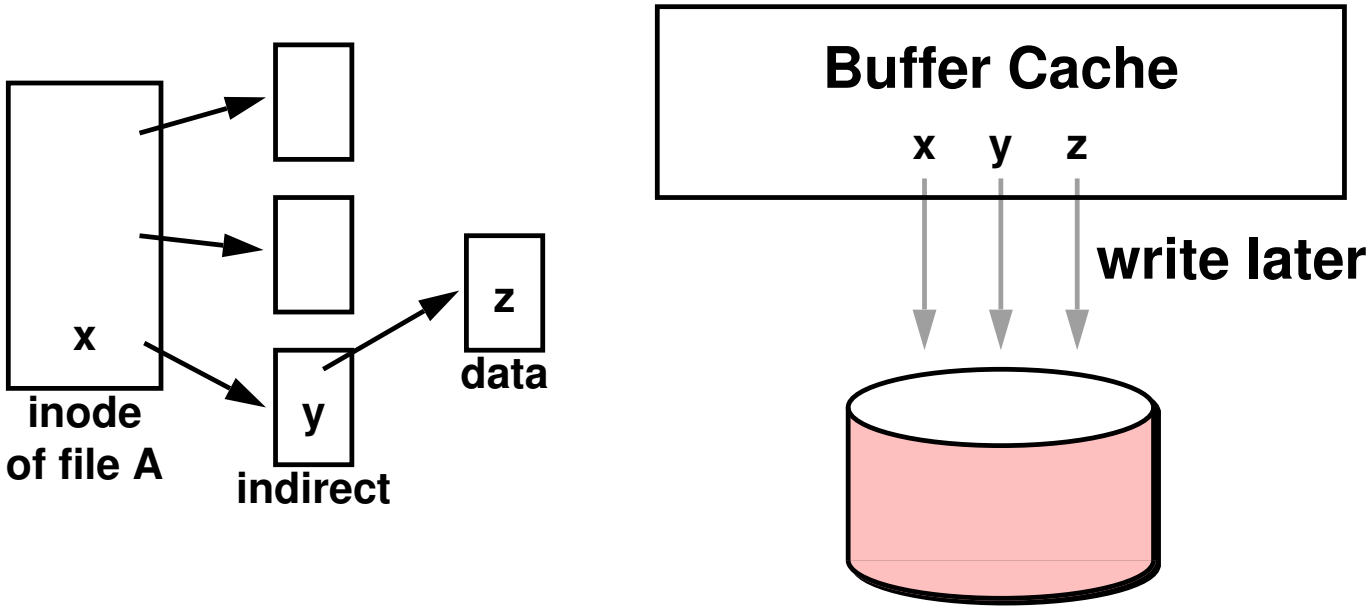
- The journal is on the disk



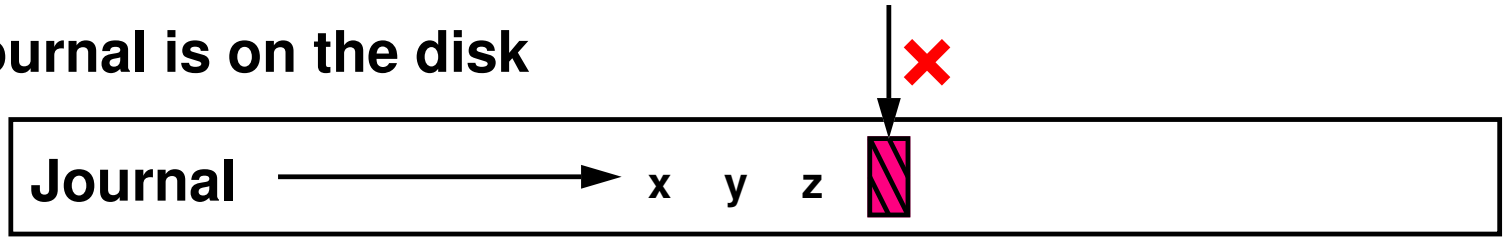
  - question is, did failure happen *before* or *after* the *commit*
  - is this bad?
    - how bad is it?

Back To The Example

- Let's say that you are appending to file A



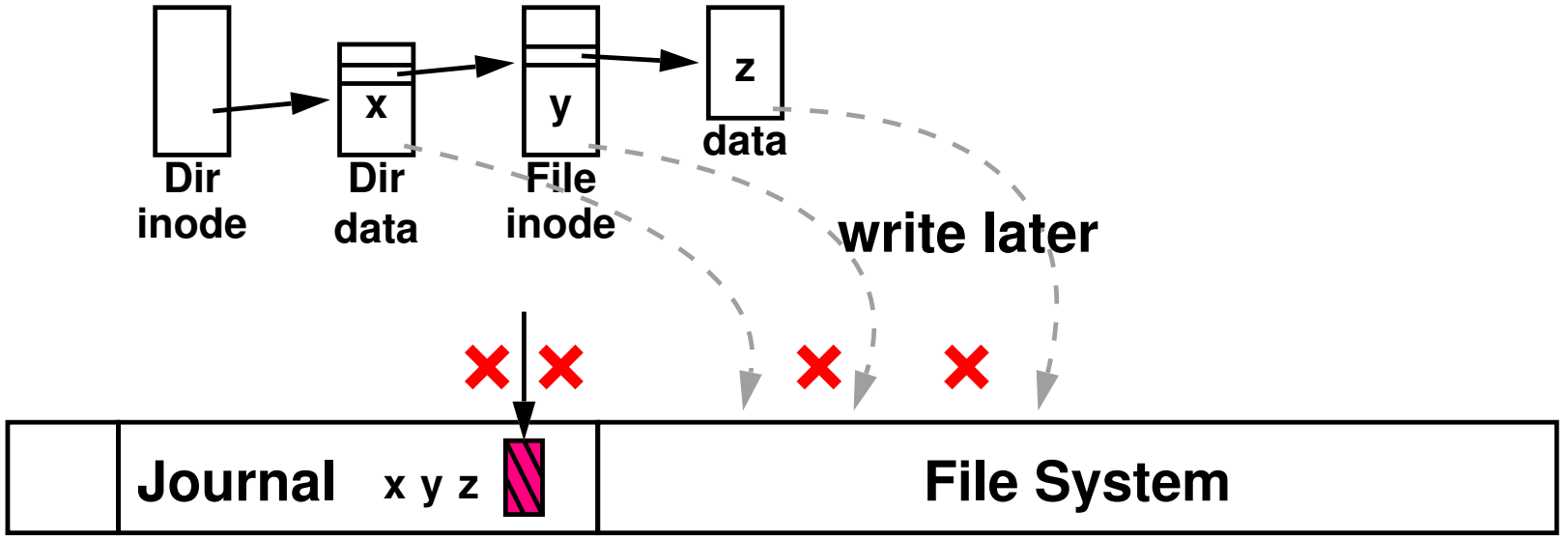
- The journal is on the disk



  - question is, did failure happen *before* or *after* the *commit*
  - is this bad?
    - no

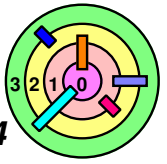
Back To Example 2

- Create a new file with one data block



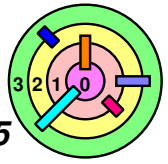
Journaling

- Journaling options
  - journal everything
    - everything on disk made consistent after crash
    - last few updates possibly lost
    - expensive
  - journal metadata only
    - metadata made consistent after a crash
      - *user data not*
    - last few updates possibly lost
    - relatively cheap



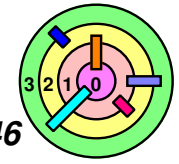
Ext3

- A journaled file system used in Linux
  - same on-disk format as Ext2 (except for the journal)
    - (Ext2 is an FFS clone)
  - supports both full journaling and metadata only journaling
- File-oriented system calls divided into *subtransactions*
  - updates go to file system cache only
  - subtransactions grouped together
- When sufficient quantity collected or 5 seconds elapsed, *commit* processing starts
  - updates (new values) written to journal
  - once entire batch is journaled, end-of-transaction record is written
  - cached updates are then *checkpointed*, i.e., written to file system
  - *journal cleared* after checkpointing completes



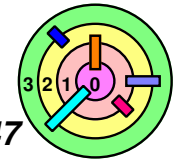
Journaling vs. Log-structured file system

- Some people confuse *journaling* with *log-structured file system*
  - log-structured file system: good *write performance*
    - coarse-grained recovery using *checkpoint file*
    - it's a file system
  - journaling: *crash resiliency*
    - can be *added* to *any existing file system*
    - use *checkpointing* to clear journal

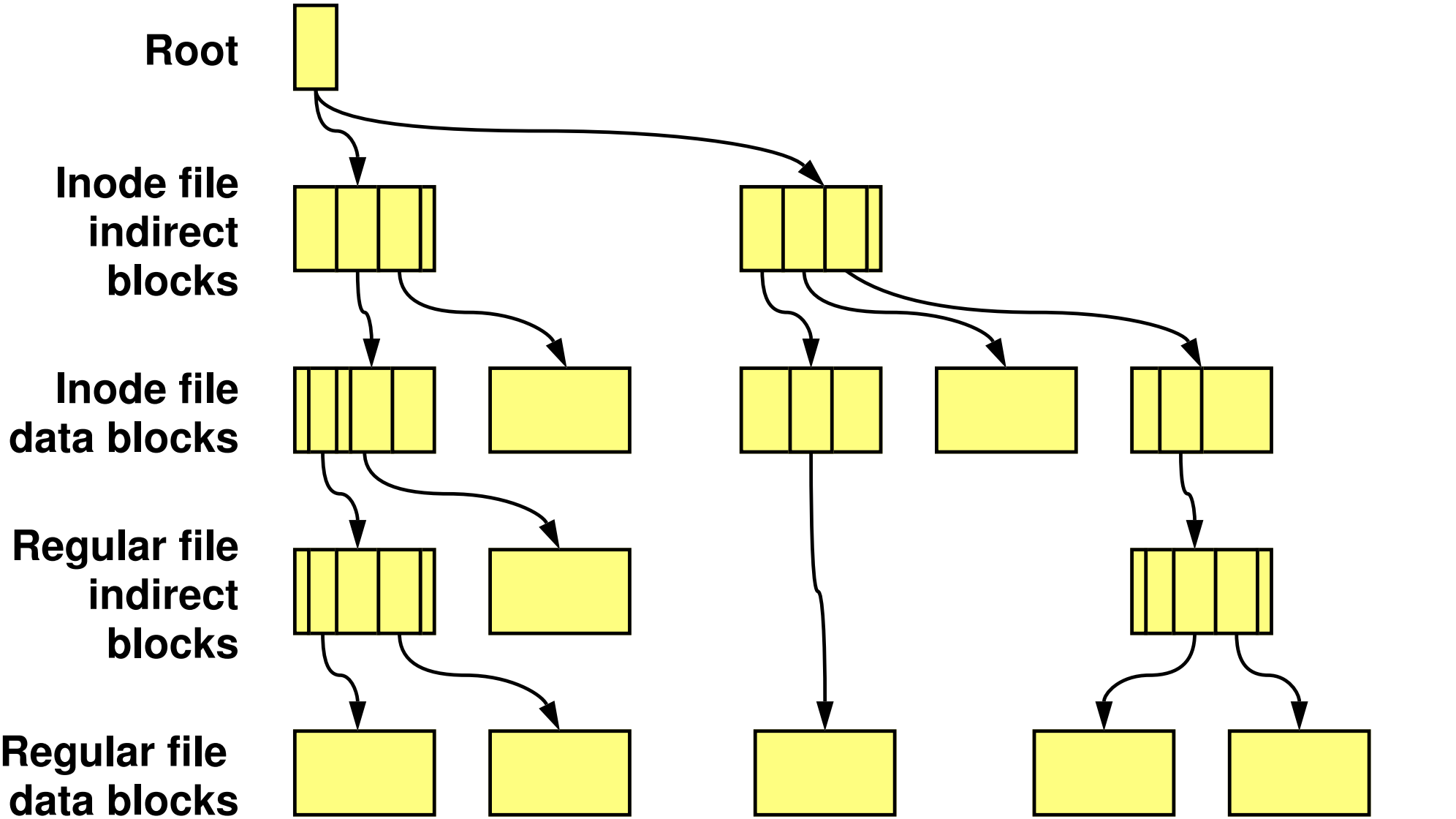


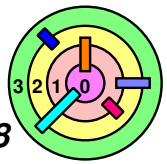
Shadow Paging

- Refreshingly simple
- Based on *copy-on-write* ideas
- Examples
  - WAFL (Network Appliance)
  - ZFS (Sun)

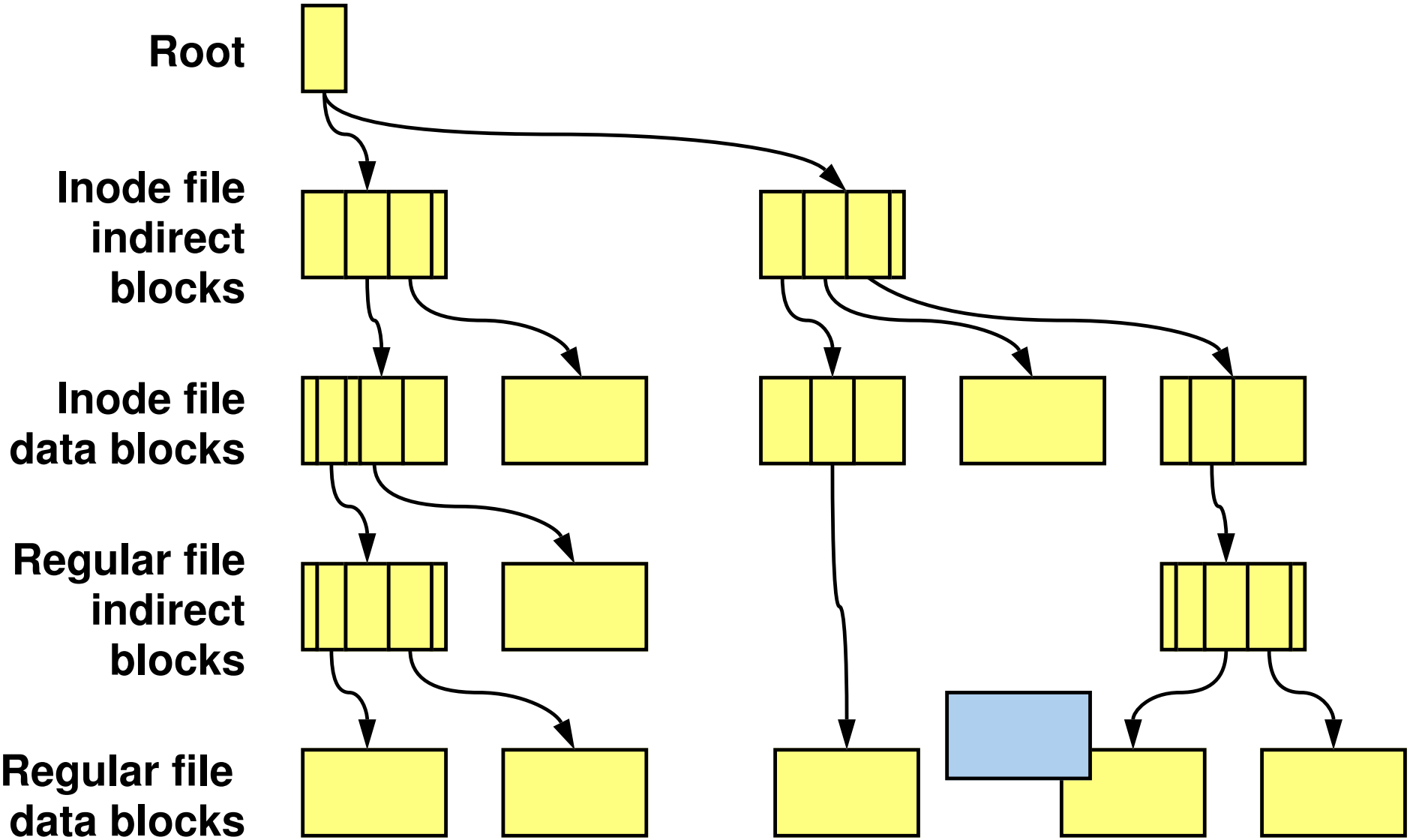


Shadow-Page Tree

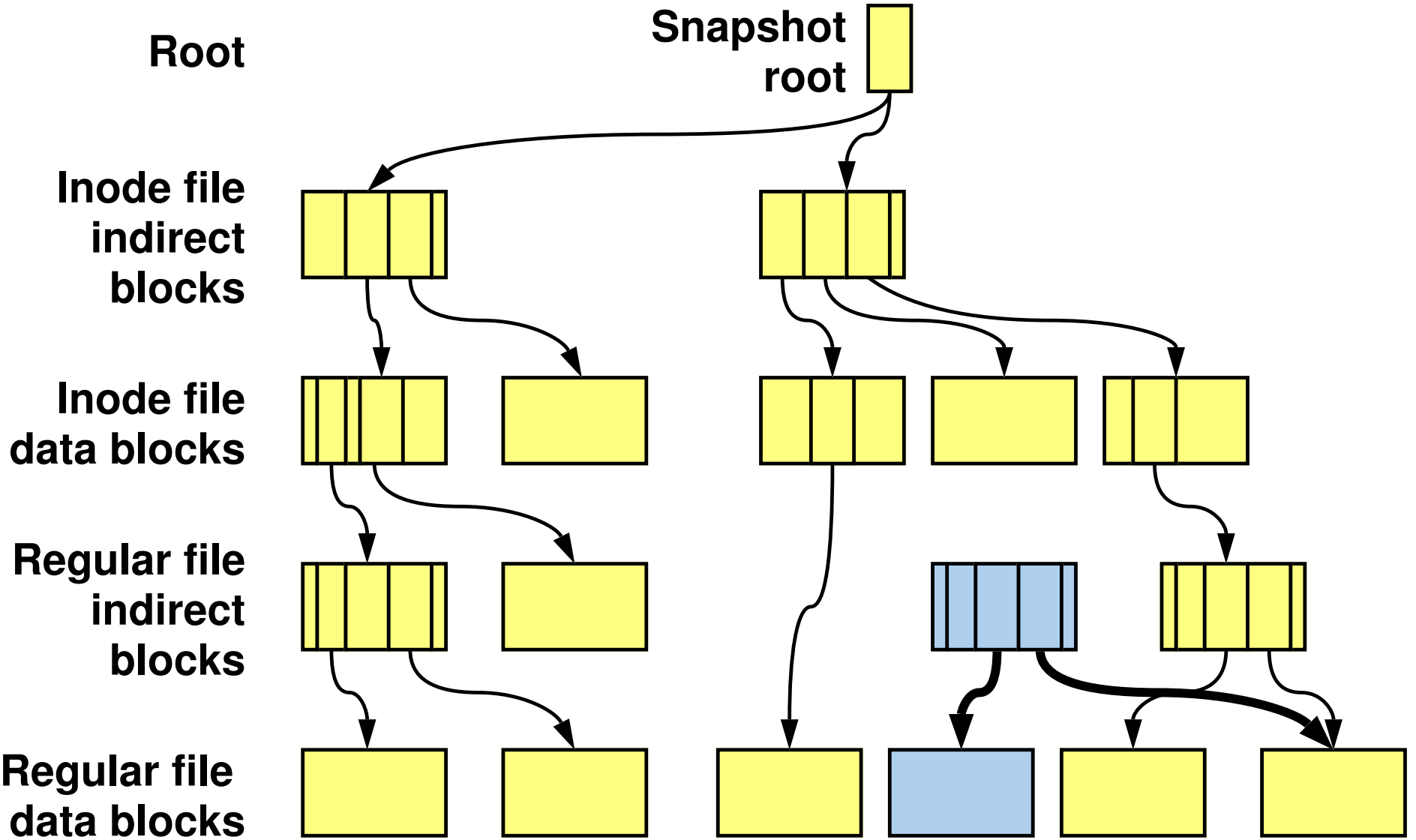




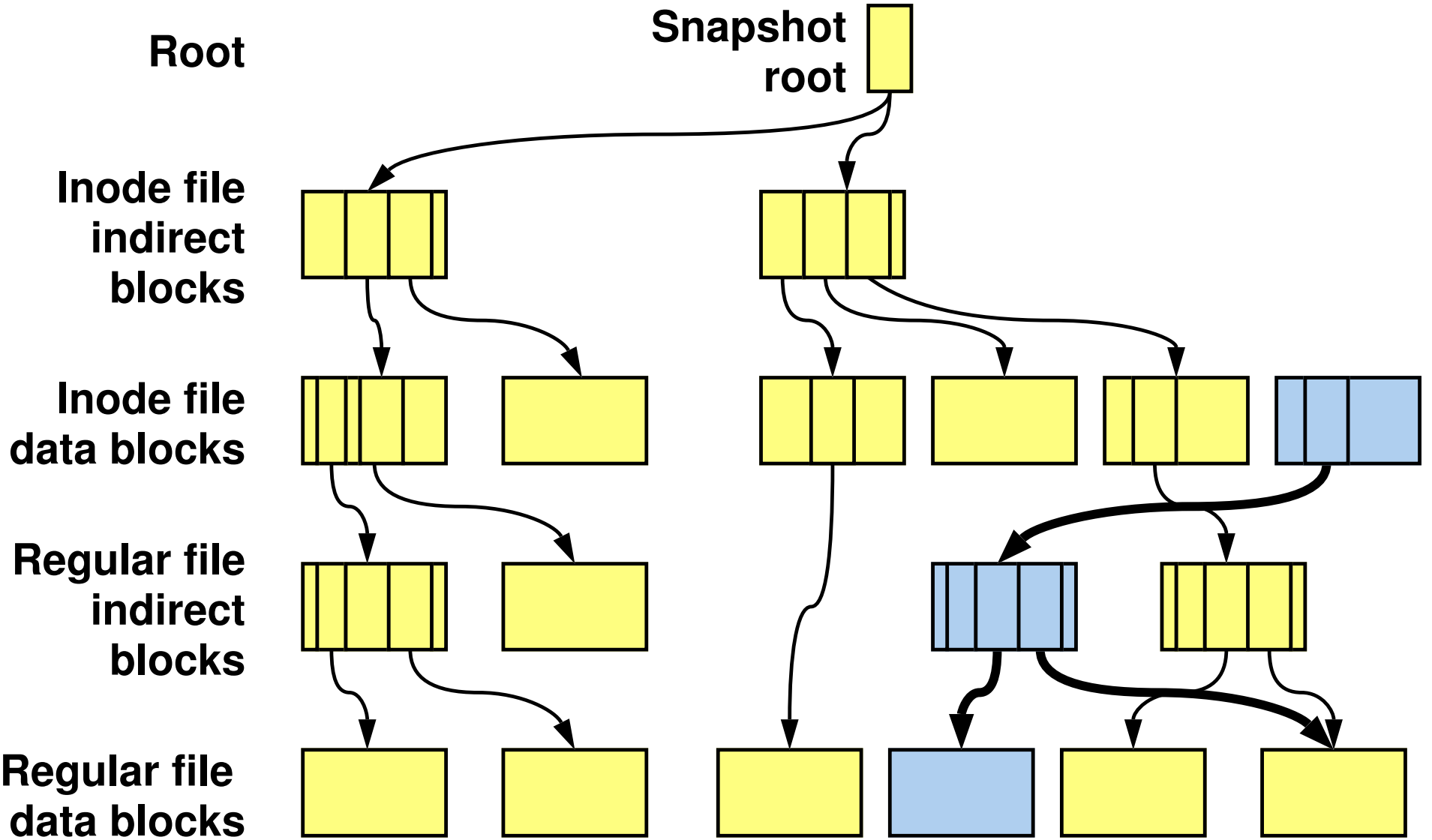
Shadow-Page Tree: Modifying a Node



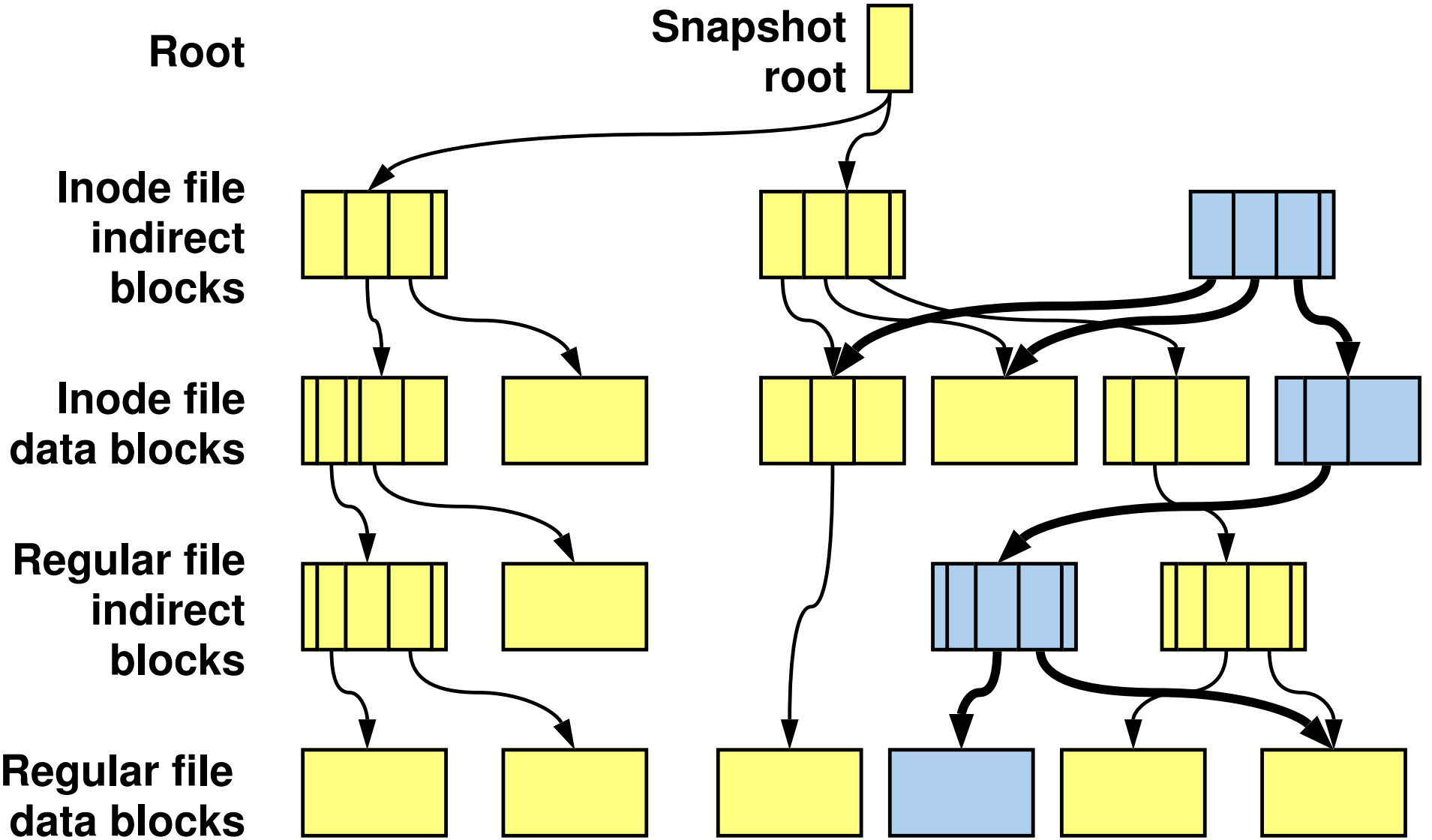
Shadow-Page Tree: Propagating Changes



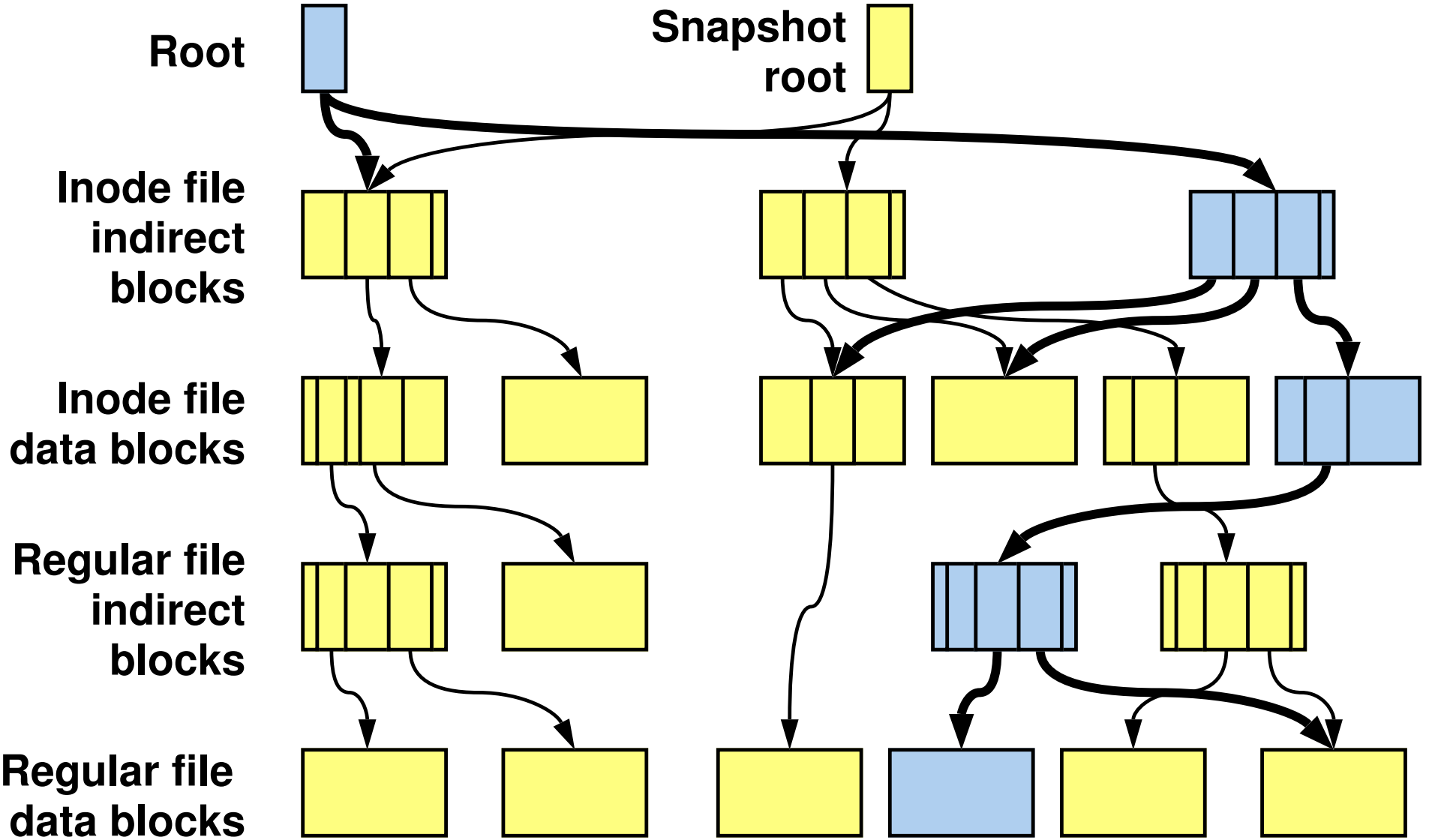
Shadow-Page Tree: Propagating Changes

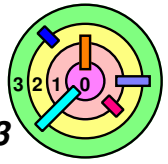


Shadow-Page Tree: Propagating Changes

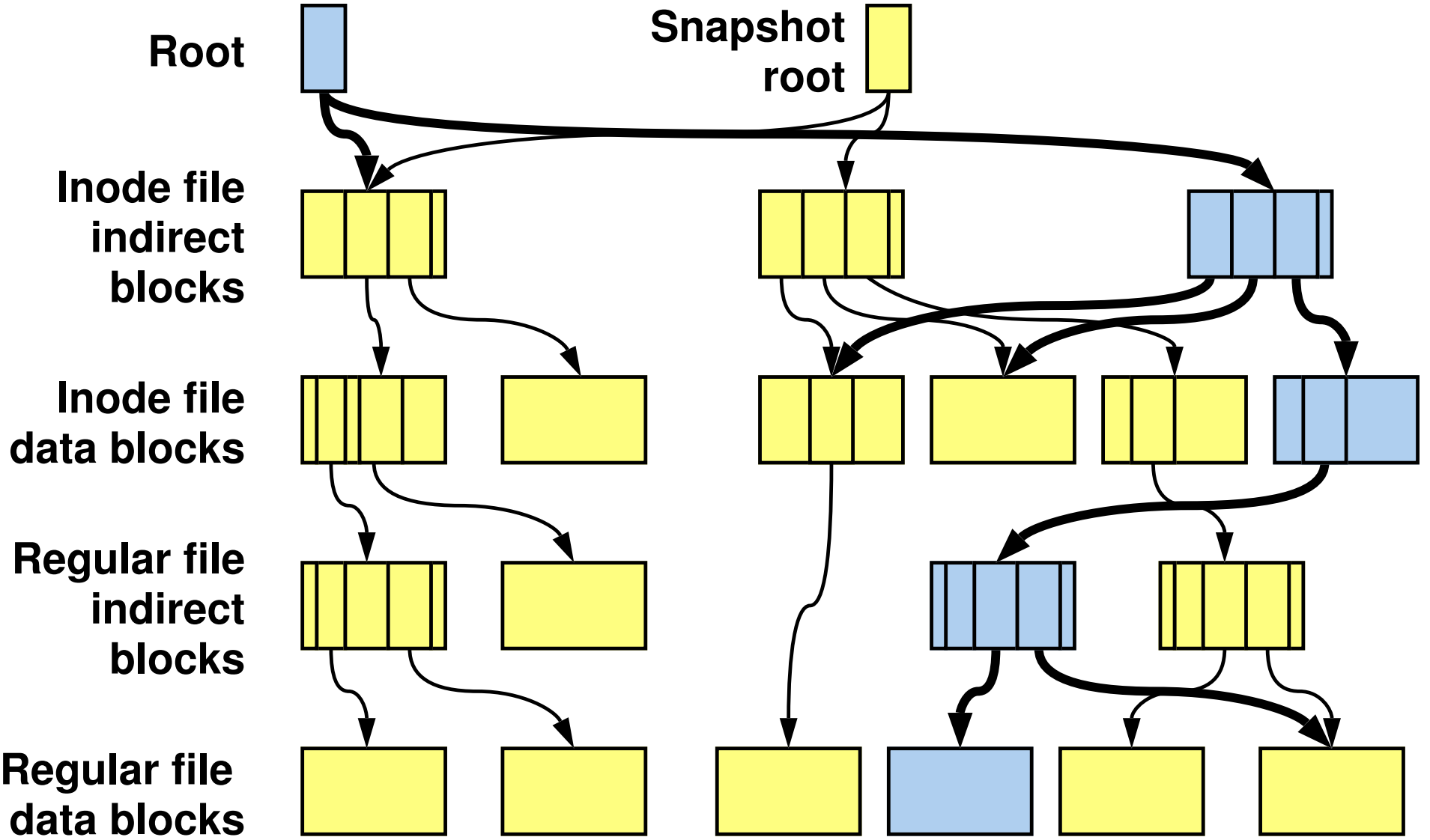


Shadow-Page Tree: Propagating Changes



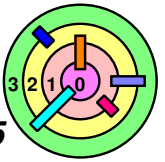


Shadow-Page Tree: Propagating Changes



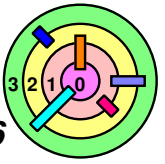
When root location is written to disk, it's like a commit record!

Extra Slides

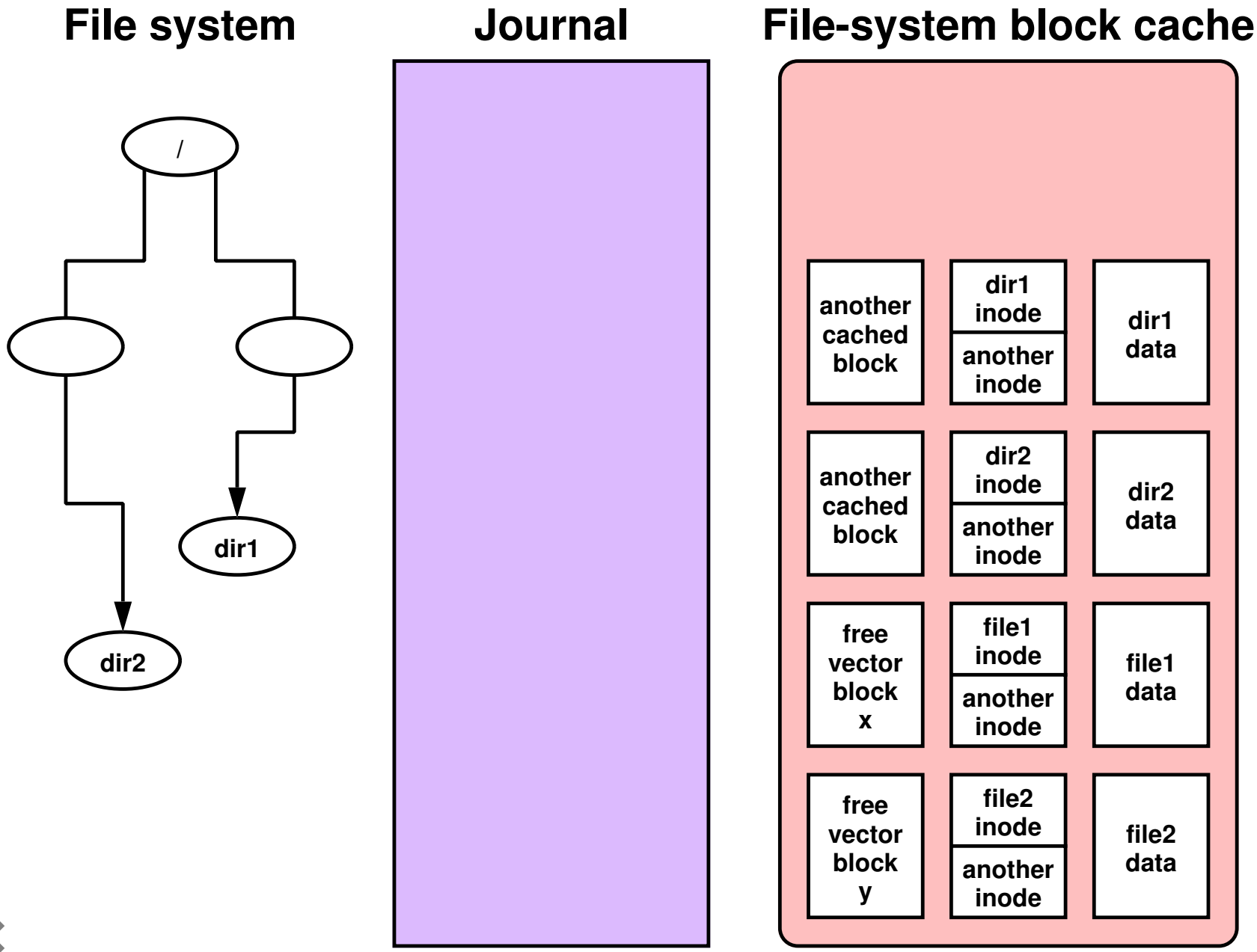


Full Journaling in Ext3

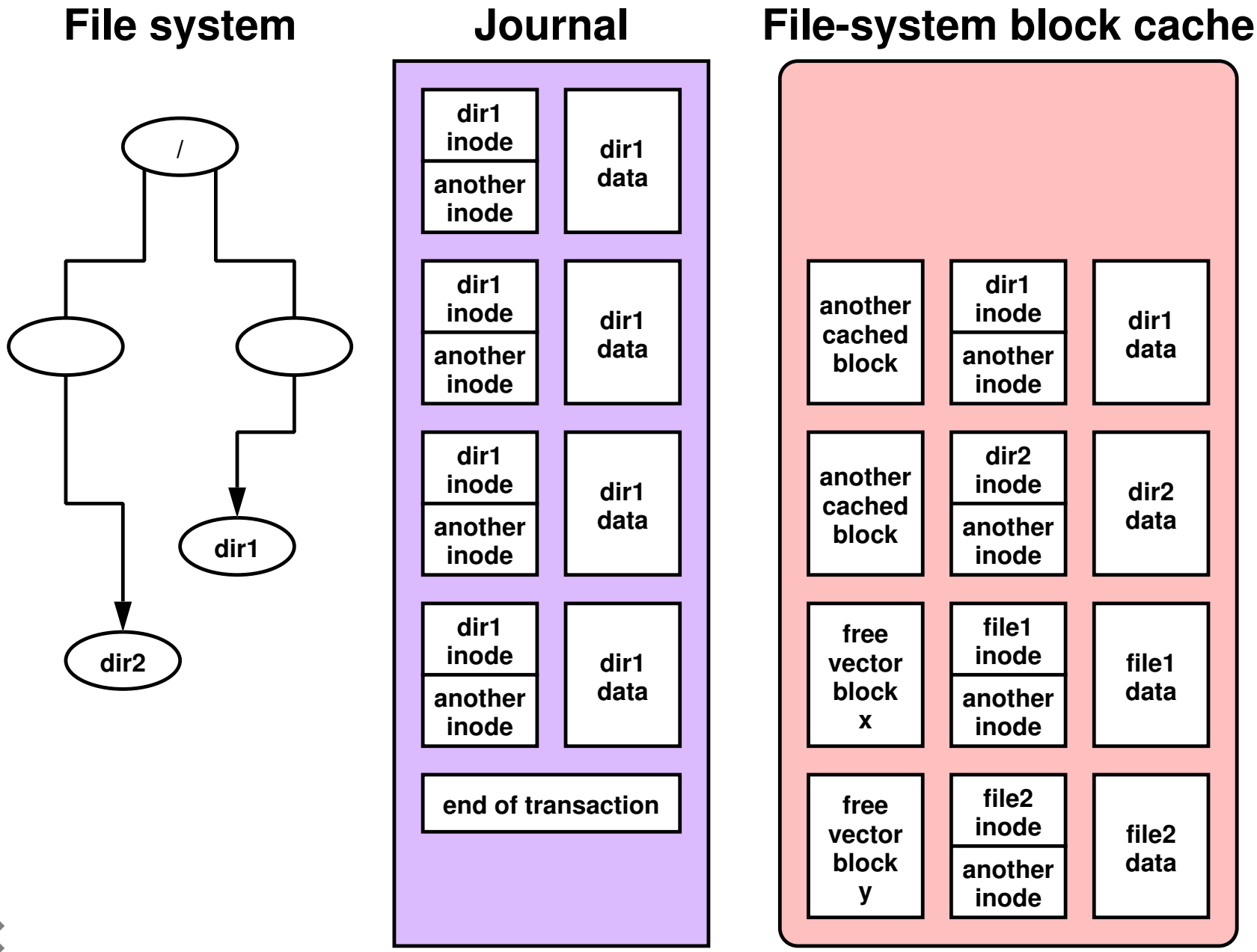
- File-oriented system calls divided into subtransactions
  - updates go to cache only
  - subtransactions grouped together
- When sufficient quantity collected or 5 seconds elapsed, commit processing starts
  - updates (new values) written to journal
  - once entire batch is journaled, end-of-transaction record is written
  - cached updates are then checkpointed, i.e., written to file system
  - journal cleared after checkpointing completes



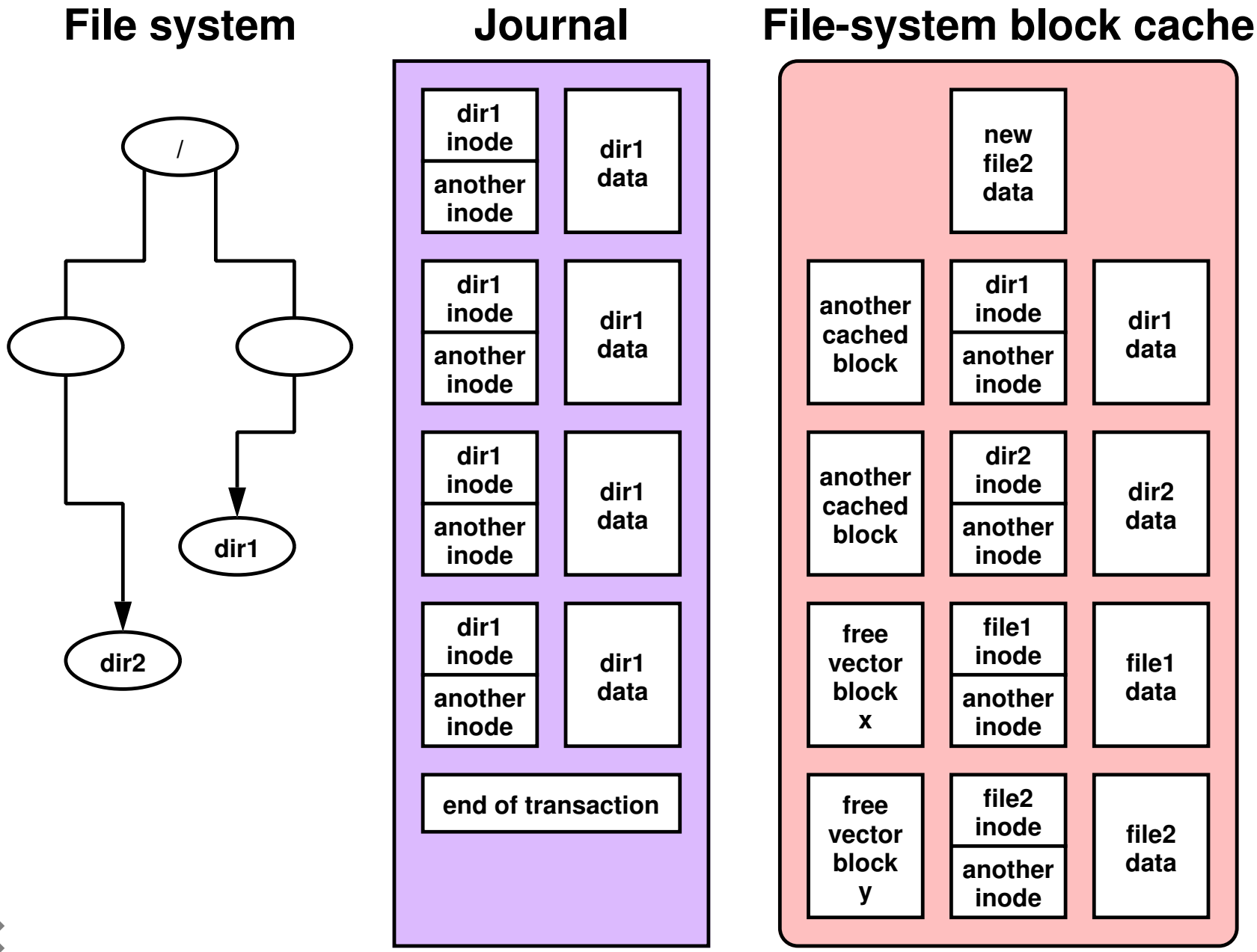
Journaling in Ext3 (part 1)



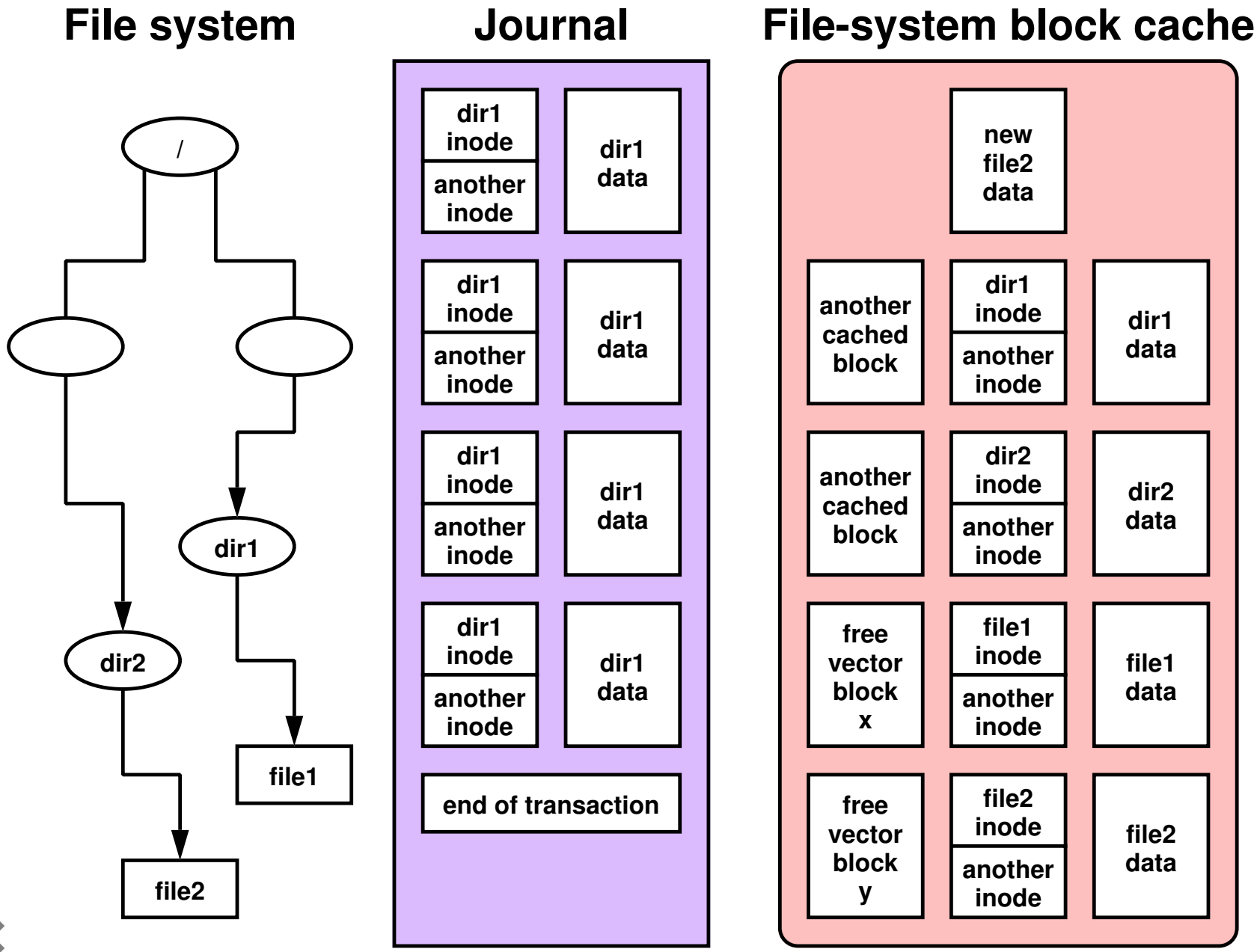
Journaling in Ext3 (part 2)



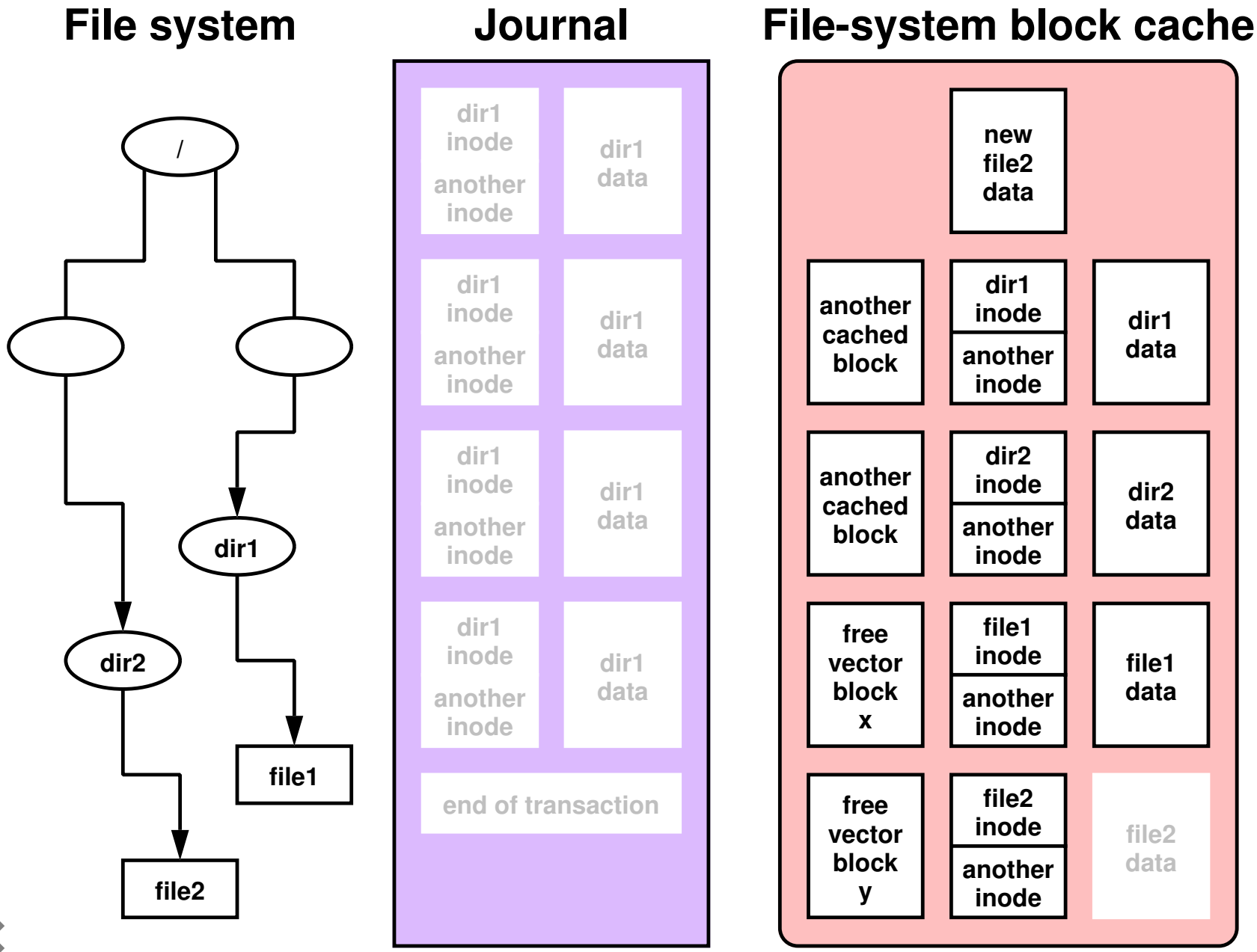
Journaling in Ext3 (part 2)



Journaling in Ext3 (part 3)

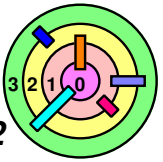


Journaling in Ext3 (part 4)



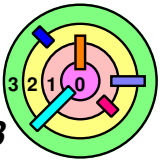
Metadata-Only Journaling in Ext3

- It's more complicated!
- Scenario (one of many):
  - you create a new file and write data to it
  - transaction is committed
    - metadata is in journal
    - user data still in cache
  - system crashes
  - system reboots; journal is recovered
    - new file's metadata are in file system
    - user data is not
    - metadata refer to disk blocks containing other users' data



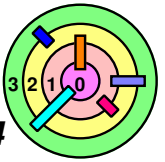
Coping

- Zero all disk blocks as they are freed
  - done in "secure" operating systems
  - expensive
- Ext3 approach
  - write newly allocated data blocks to file system before committing metadata to journal
  - fixed?



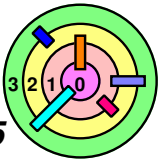
Yes, but ...

- Spencer deletes file A
  - A's data block x added to free vector
- Robert creates file B
- Robert writes to file B
  - block x allocated from free vector
  - new data goes into x
  - system writes newly allocated x to file system in preparation for committing metadata, but ...
- System crashes
  - metadata did not get journaled
    - A still exists; B does not
    - B's data is in A



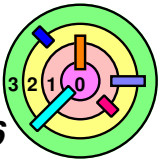
Fixing the Fix

- Don’t reuse a block until transaction freeing it has been committed
  - keep track of most recently committed free vector
  - allocate from it

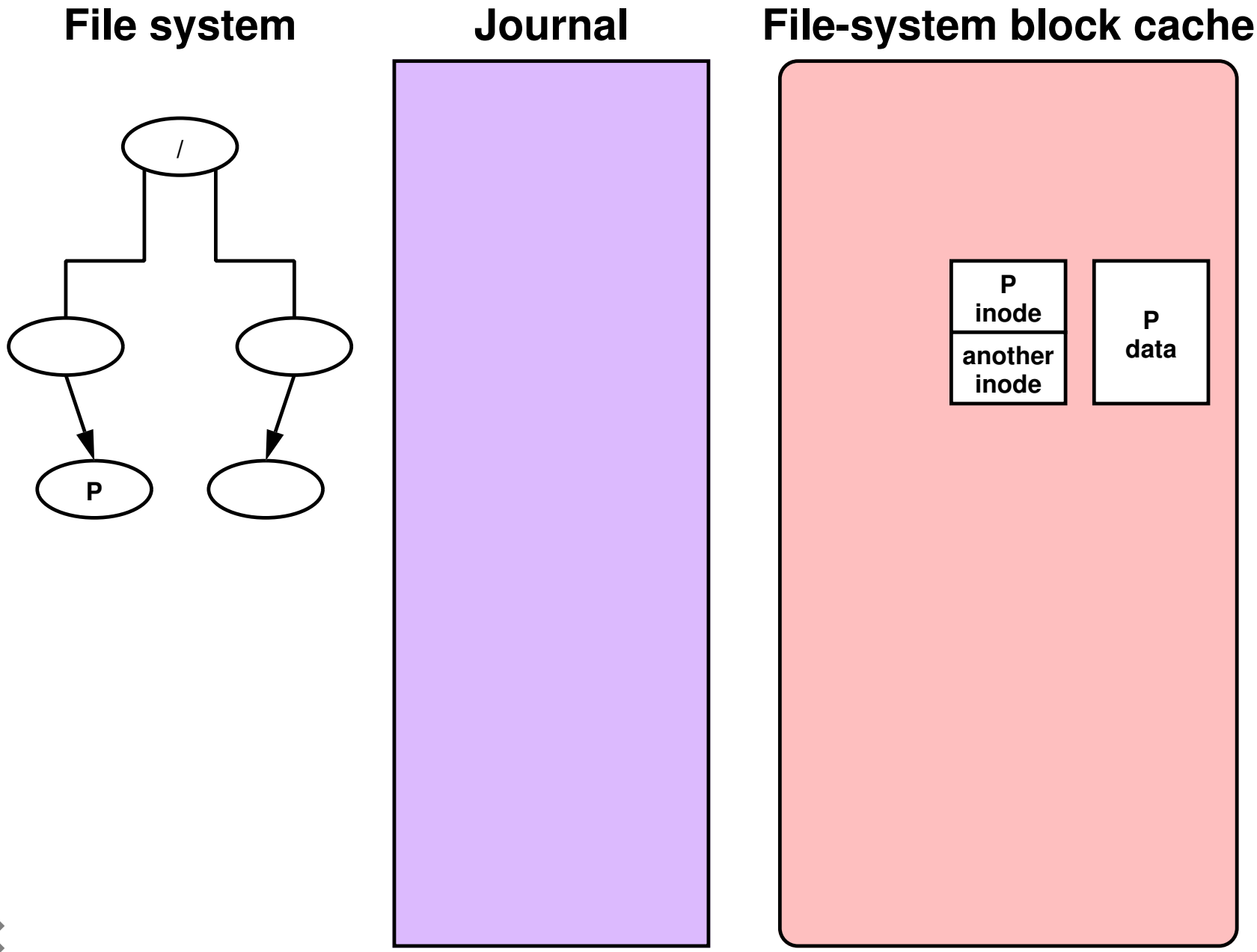


Fixed Now?

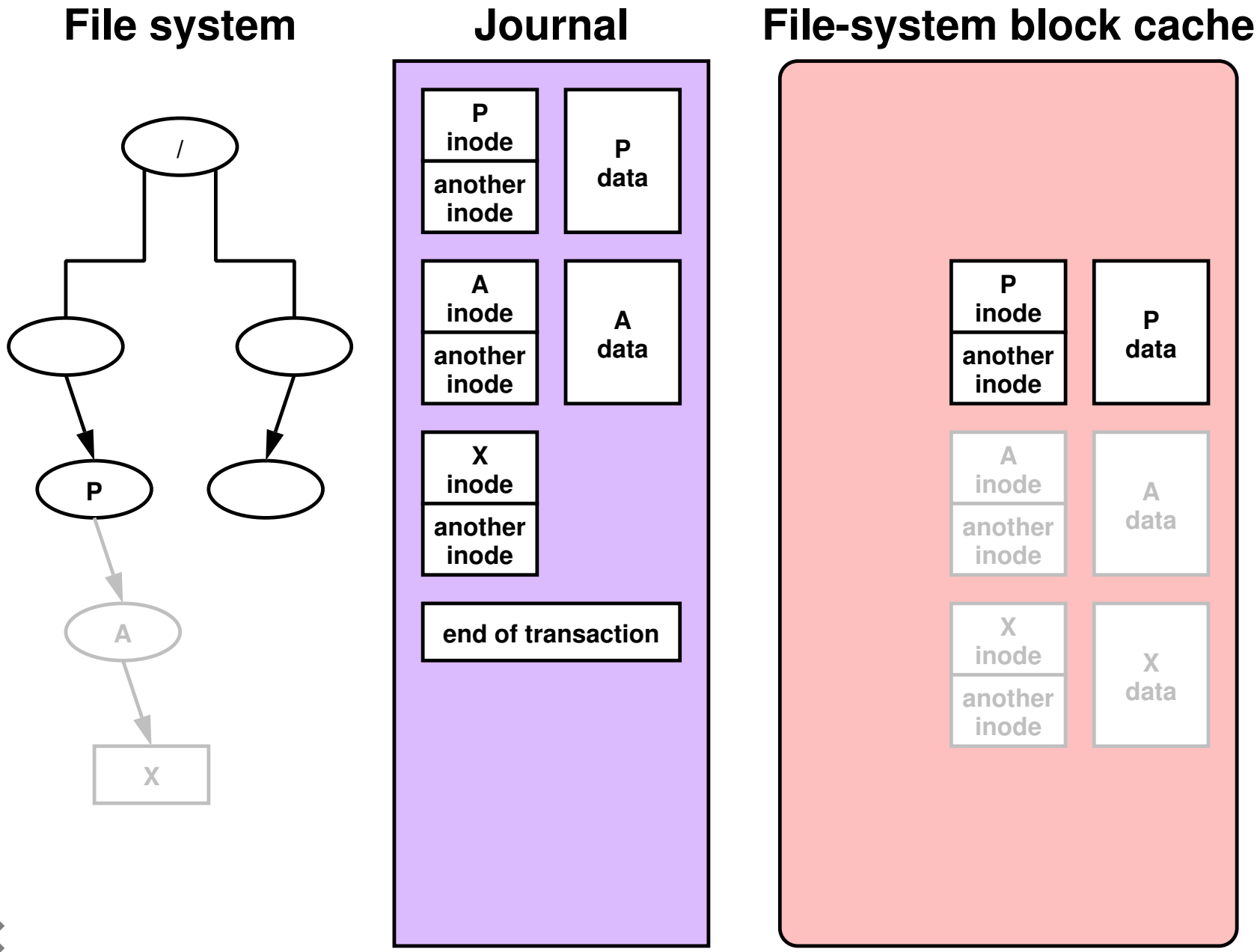
No ...



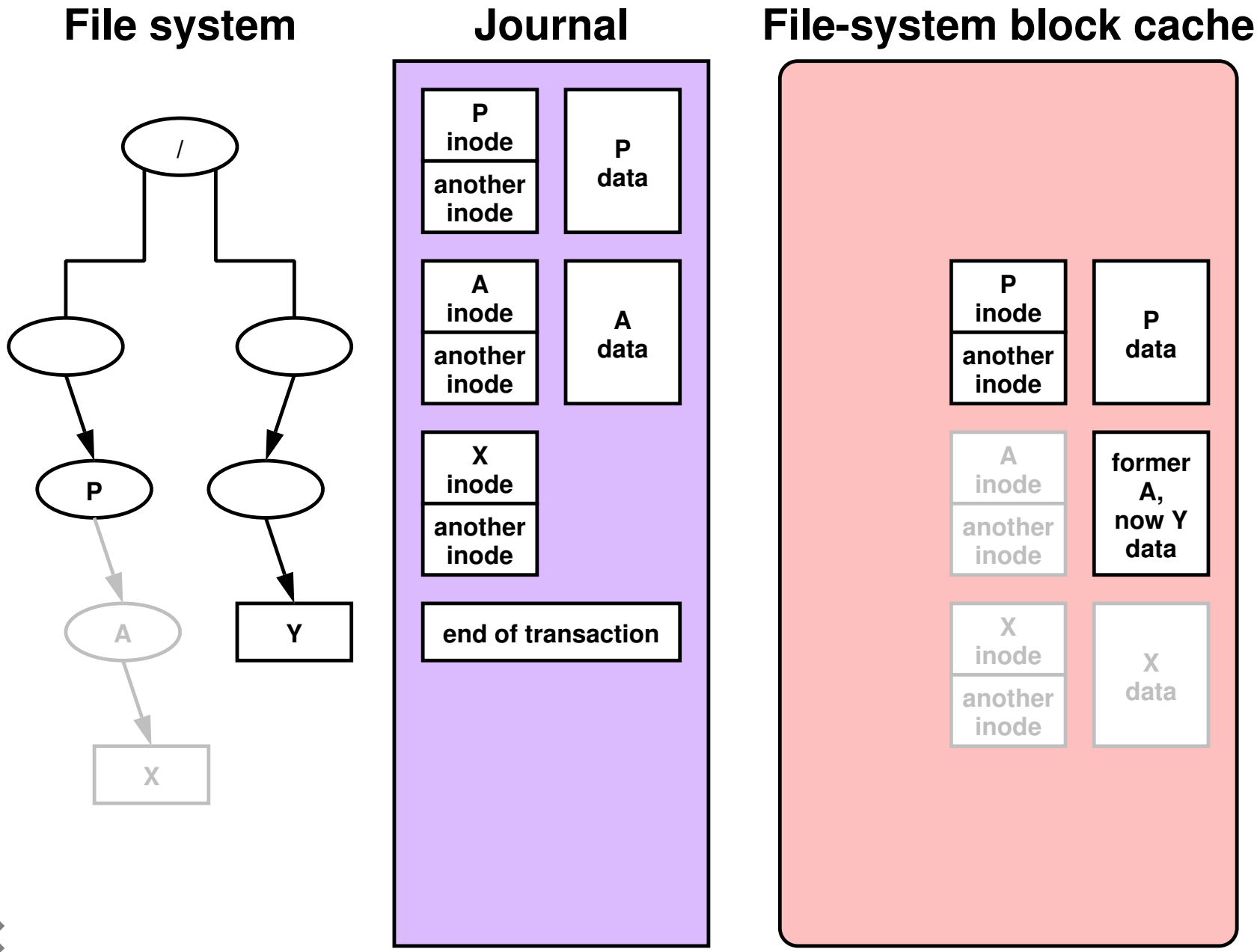
Yet Another Problem (part 1)



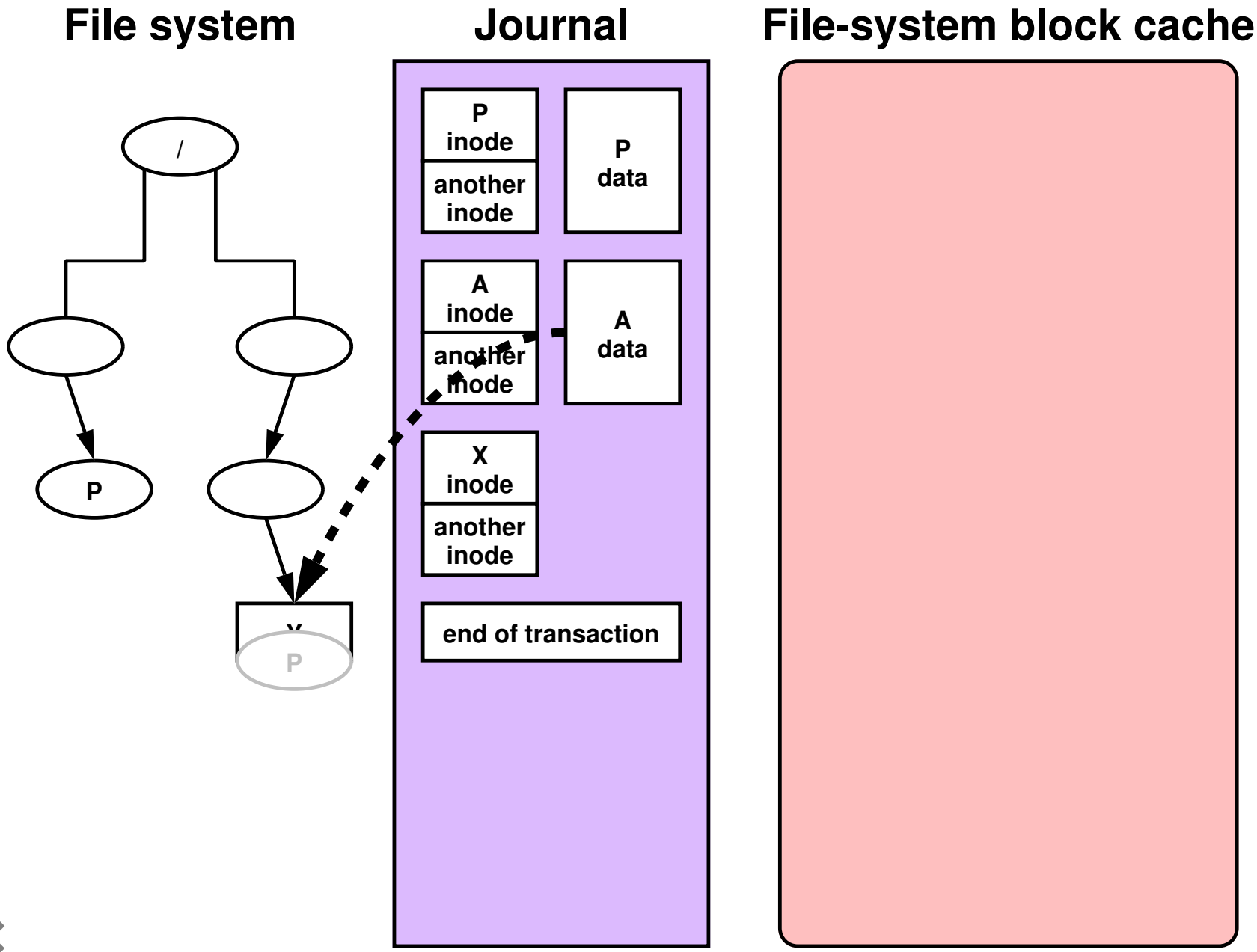
Yet Another Problem (part 2)



Yet Another Problem (part 3)

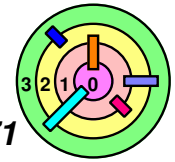


Yet Another Problem (part 3)



The Fix

- The problem occurs because metadata is modified, then deleted
- Don’t blindly do both operations as part of crash recovery
  - no need to modify the metadata!
  - Ext3 puts a "revoke" record in the journal, which means "never mind ..."



Fixed Now?

- Yes!
  - (or, at least, it seems to work ...)