File Systems Part 3

Transactions

- "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*
    - record new contents: *redo journaling*
- Shadow paging
  - steps of transaction written to disk, but old values remain
  - single write switches old state to new

Data vs. Metadata

- Metadata
  - system-maintained data pertaining to the structure of the file system
    - inodes
    - indirect, doubly indirect, triply indirect blocks
    - directories
    - free space description
    - etc.
- Data
  - data written via write system calls

System data structures are **metadata**. What application programs write to files are **data**. From the system's perspective, metadata is what is important – if there are problems with it, then the integrity of the entire file system is in doubt.

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

Committing vs. Checkpointing

- Checkpointed updates
  - written to file system and are thus permanent
- Committed updates
  - not necessarily written to file system, but guaranteed to be written eventually (checkpointed), even if there is a crash
- Uncommitted updates
  - not necessarily written to file system (yet), may disappear if there is a crash

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
    - does redo journaling

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* — written to file system
  - journal cleared after checkpointing completes

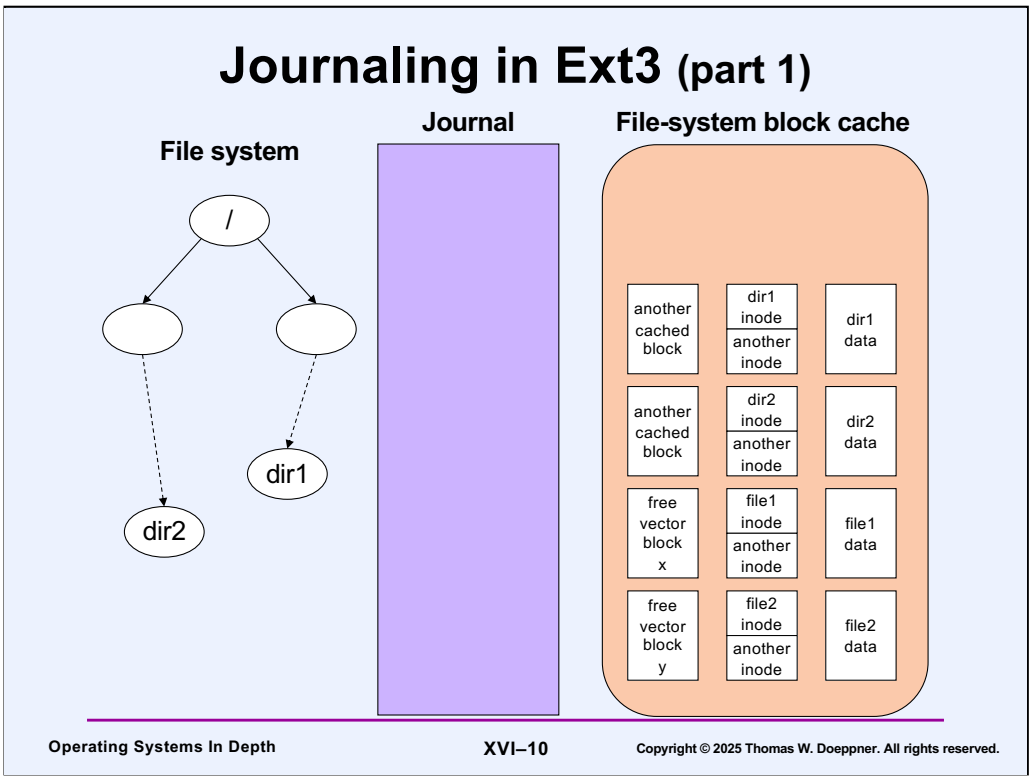
Quiz 1

You have a Linux system with an Ext3 file system with full journaling. You run a script that deletes some files, creates some new files and writes data to them, then renames the new files. This takes two seconds. Immediately after the script finishes, there's a power failure and the system crashes. When it comes back up, after crash recovery:

a) some later parts of the script may have completed, even though earlier parts did not

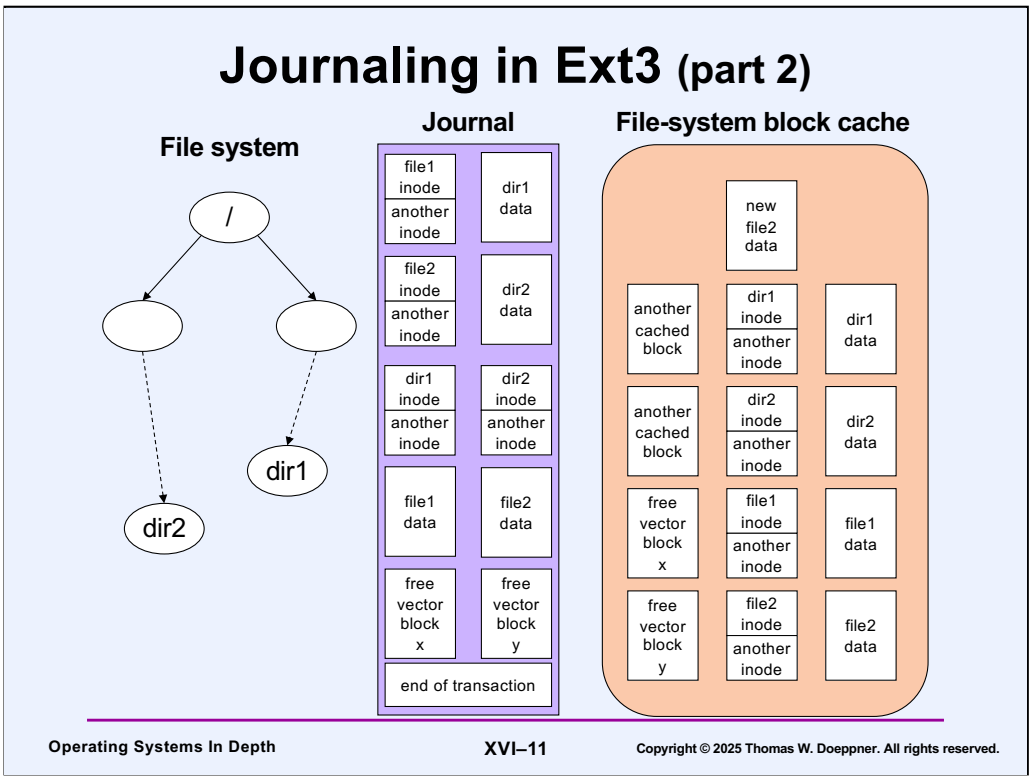
b) it will appear as if the script ran to some point and then terminated (this allows for both none of it and all of it)

c) it will definitely appear as if the script ran to completion



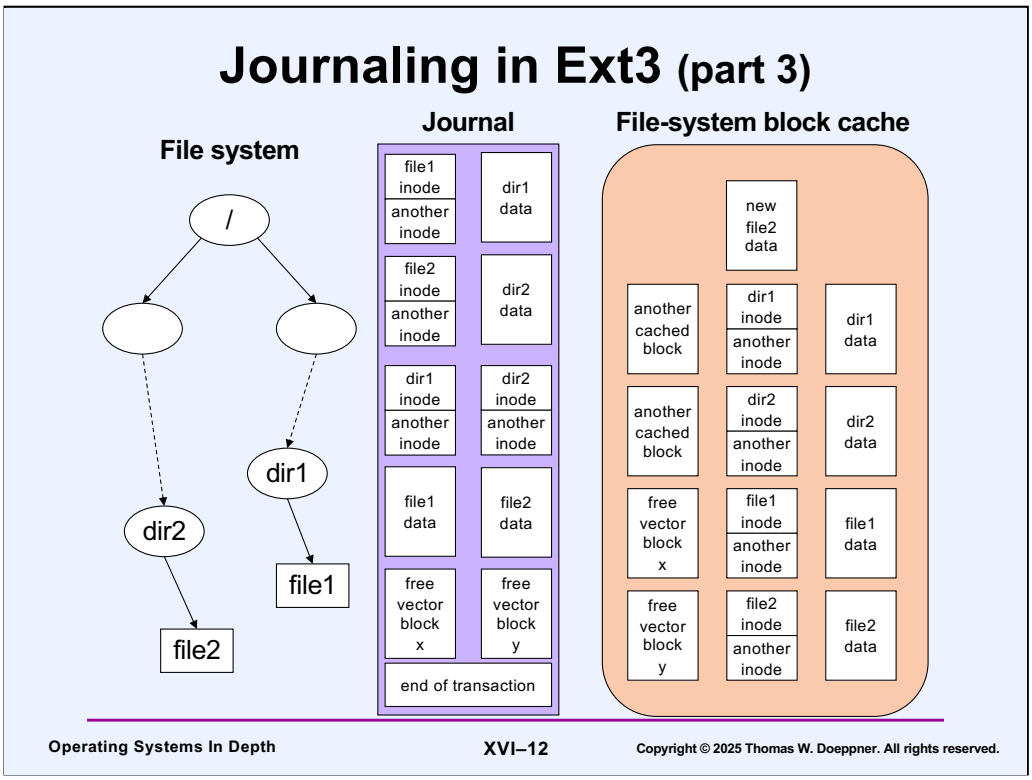
The left-hand portion of the slide shows the file system as it exists on disk. The middle portion shows the contents of the journal. The right-hand portion shows the contents of the cache (in kernel memory). At the moment, a number of operations have been done that affect the file system, but only the cache has been updated. If the system were to crash at this point, the effect would be that nothing has changed in the file system.

Our presentation of Ext3 is partly based on a presentation given by Stephen Tweedie in ~2001.

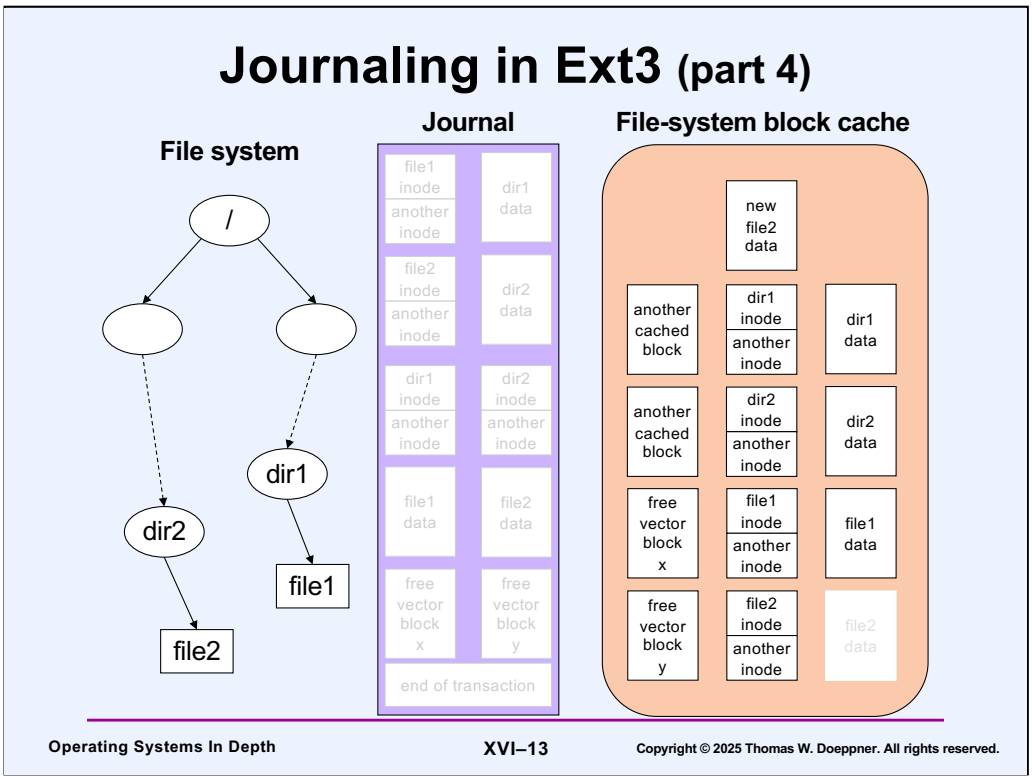


At this point, all the operations have been written to the journal and an "end of transaction" record has been written to the journal as well. This means that the operations (forming a transaction) have been committed. If the system were to crash now, recovery would entail applying the journal to the on-disk file system. After recovery completes, the on-disk file system will contain the effects of all the updates.

Assuming the system doesn't crash, the next step is to copy the data from the cache to the file system (i.e., checkpointing the transaction). Note that checkpointing involves copying from the cache to the file system, not copying from the journal to the file system – the latter requires much more time than the former. If further modifications are made to a block of file after the transaction has been committed, then the new modification, being part of a subsequent transaction, must be added to the cache as an entry separate from the current version of the block, since this older version of the block must continue to exist until it's checkpointed to the file system.



Here, everything that was in the journal has been applied to the on-disk file system. Thus the updates have been **checkpointed**.



Once checkpointing completes, the updates can be removed from the journal, and updated blocks in the cache may be removed.

Quiz 2

You have a Linux system with an Ext3 file system with metadata-only journaling. You run a script that deletes some files, creates some new files and writes data to them, then renames the new files. This takes two seconds. Immediately after the script finishes, there's a power failure and the system crashes. When it comes back up, after crash recovery:

a) **there may be data written to the new files, but no files were deleted**

b) **it will definitely appear as if the script ran to some point and then terminated (this allows for both none of it and all of it)**

c) **the script may appear to have completed, though there's no data written to the new files**

Metadata-Only Journaling in Ext3

- File-oriented system calls divided into subtransactions
  - updates to metadata go to cache only
  - updates to data go to cache; may be written to disk at any time
- When sufficient quantity collected or 5 seconds elapsed, *commit* processing starts
  - metadata updates written to journal
  - once entire batch is journaled, end-of-transaction record is written
- Cached metadata updates are then *checkpointed* — written to file system

Metadata-Only Issues

- 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 are 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 …

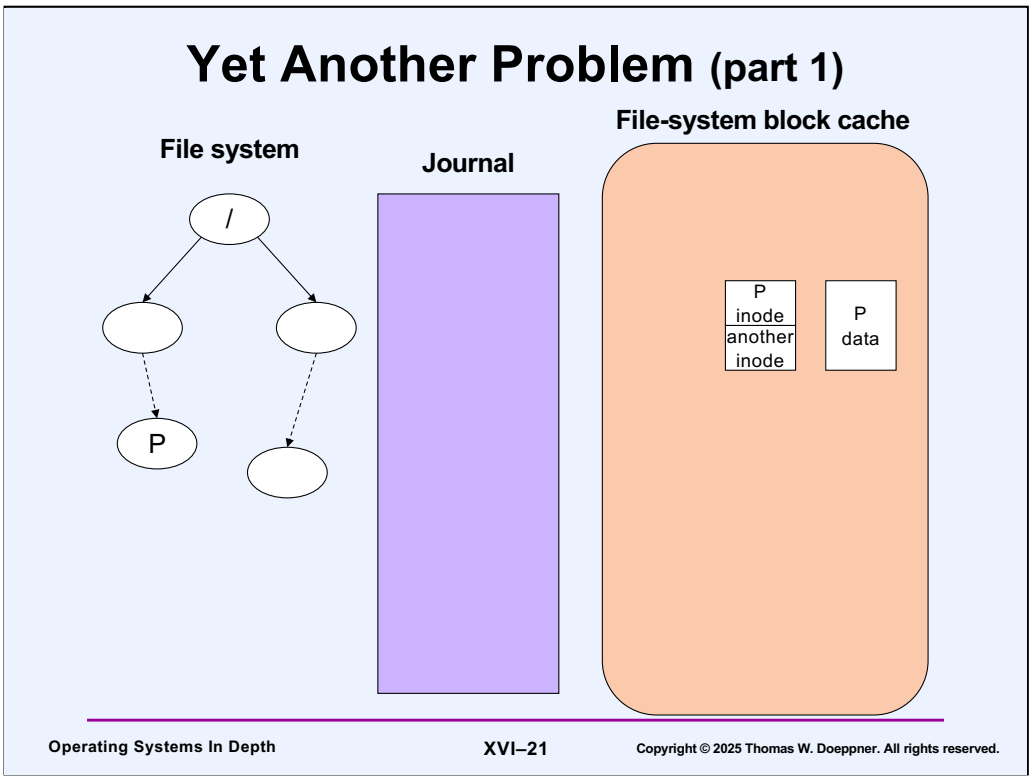
- Mary deletes file A
  - A's data block *x* added to free vector
- Ted creates file B
- Ted writes to file B
  - block *x* allocated from free vector
  - new data goes into *x*
  - system writes newly allocated *x* to file system in preparation for committing metadata, but …
- System crashes
  - metadata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

Fixing the Fix

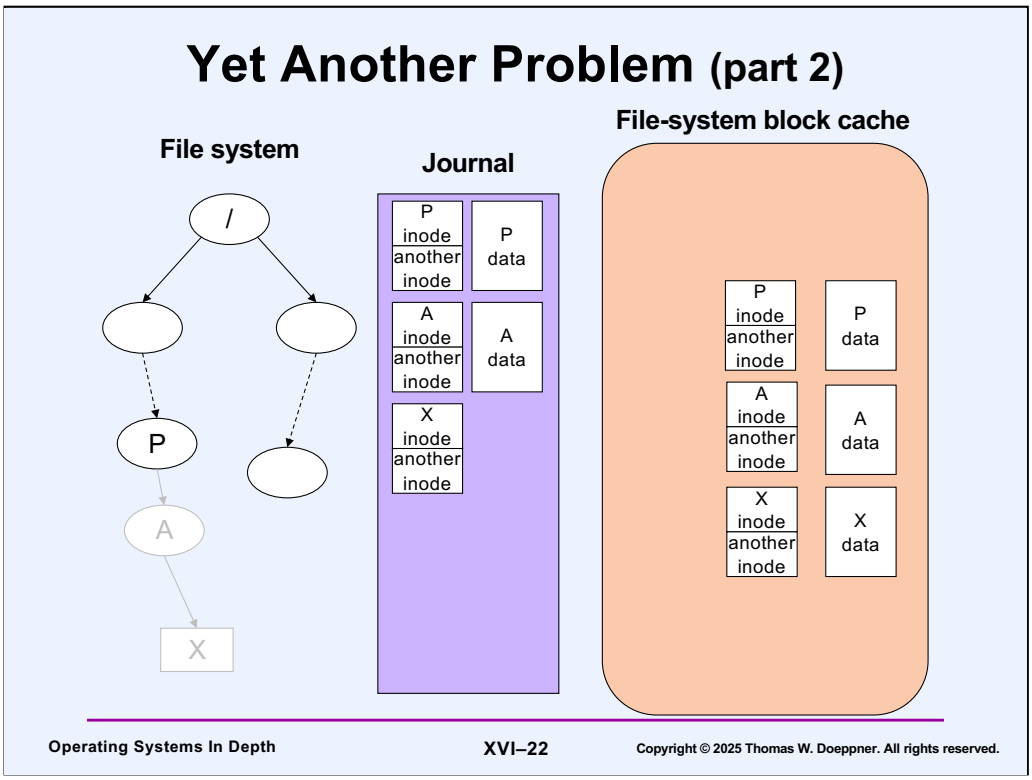
- Don’t reuse a block until transaction freeing it has been committed
  - keep track of most recently committed free vector
  - allocate from it

Fixed Now?

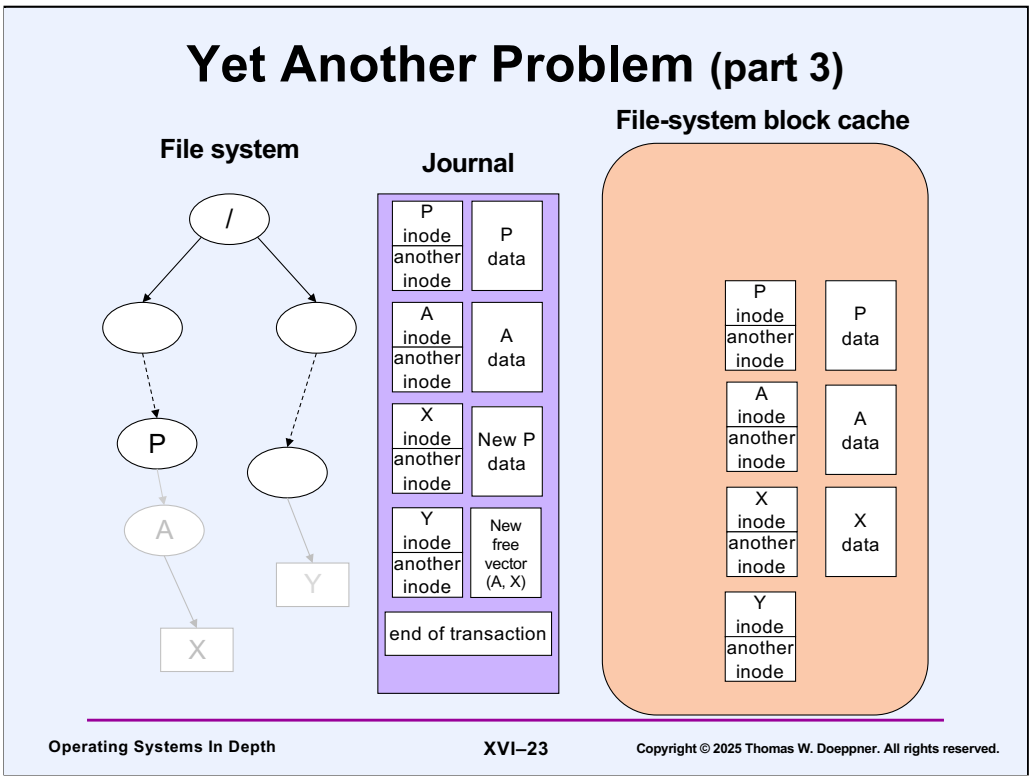
- No …



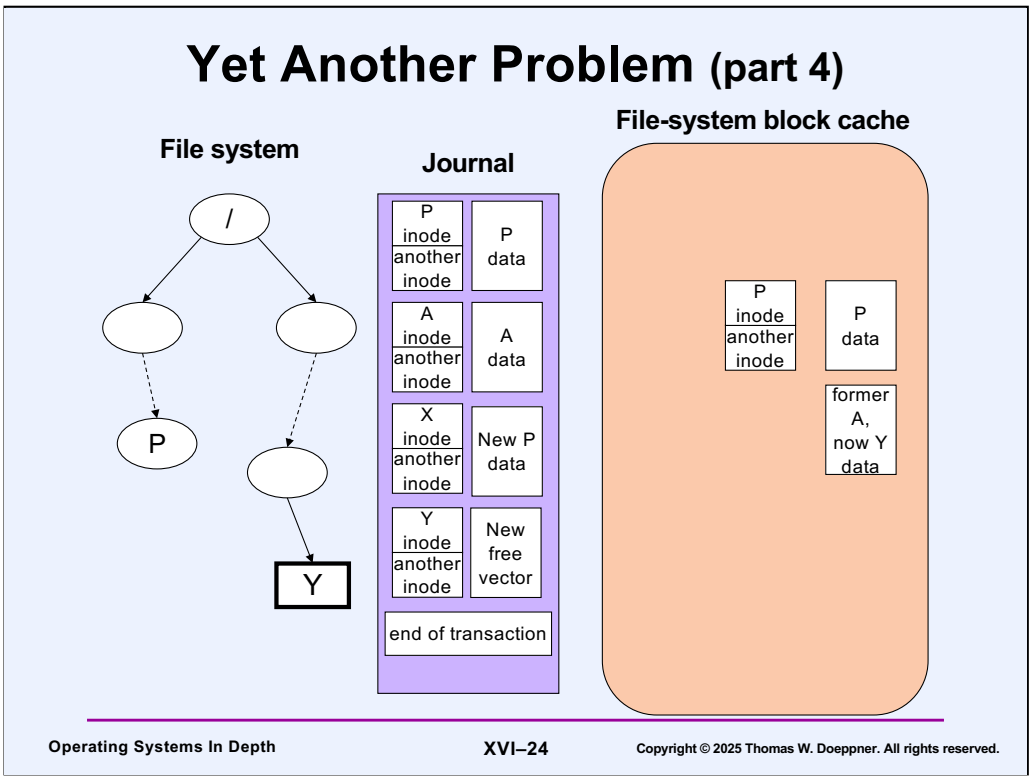
Both directory P's data and inode are in the file system (on disk) and in the block cache.



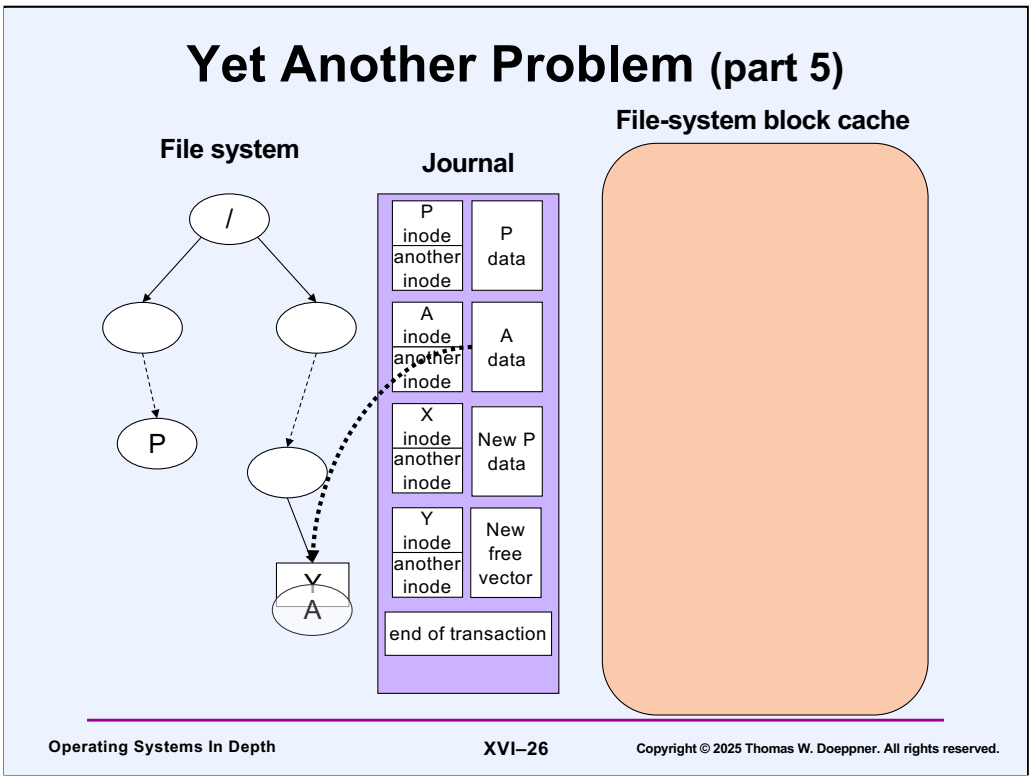
Directory A and file X are created. The cache is updated; the new values of the disk blocks go into the journal, but are not committed yet. (And thus are not checkpointed – A and X are not yet in the file system.)



Directory A and file X are deleted. File Y is created, but no data is written to it. These operations are added to the current transaction, which is then committed, but not checkpointed. However, since it was committed, the blocks occupied by A and X are added to the free vector and available for use.



File Y is written to. Its data block is the one that formerly held the contents of A. This data block is written out to disk (it's not metadata, so there's no restriction on when it may be written to disk).



The system crashes and comes back up. As part of recovery, the contents of the journal are applied to the file system. The journaled contents of the block that originally contained the entries of directory A are copied back into their original location, which now is the first data block of Y. Thus Y's previous contents are destroyed.

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 …)

Ext4

- Latest Linux file system
  - used at Brown CS as local FS on Linux systems
- Retains much of Ext3
  - journaling
    - meta-data only used at Brown CS
  - inodes
- Adds extents
  - four extents in inode
  - if more needed, B-tree is used

Undo Journaling

- Old data written to journal (from cache)
- New data written to cache; written to file system when convenient
- Committed after checkpointing
  - new data is on disk
  - (undo) journal entries removed
- Crash recovery
  - if transaction not committed, all changes to file system are undone by playing back the journal

Undo vs. Redo

- Redo
  - data written to cache
  - data copied to journal
    - committed when entire transaction is in journal
  - data stays in cache until it is checkpointed
- Undo
  - old data written to journal (from cache)
  - new data written to cache
  - new data written to disk
    - committed when all data on disk
  - data may be removed from cache prior to committing

A potential advantage of undo journaling is that data may be removed from the cache sooner than with redo, which may be useful if the kernel is running out of memory. In particular, with redo journaling, data (either the old value or the new value) must stay in the cache until after the transaction is both committed and checkpointed. But with undo journaling, data may be removed from the cache after the old value is written to the journal and the new value is written to the file system on disk. It's not necessary to wait for the entire transaction to be committed or checkpointed.

Quiz 3

If undo journaling is being used, when can a transaction be committed?

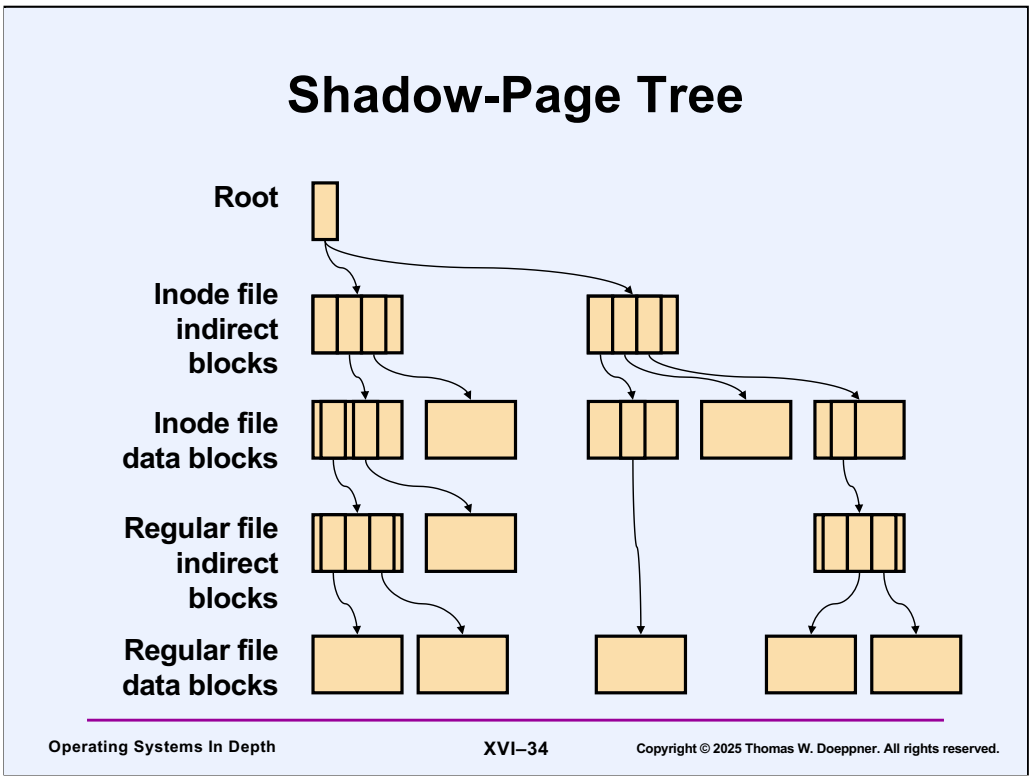
a) when the new contents of all blocks involved in the transaction have been written to the cache

b) when the new contents of all blocks involved in the transaction have been checkpointed to the file system on disk

c) when the old contents of all blocks involved in the transaction have been written to the journal

Shadow Paging

- Refreshingly simple
- Provides historical snapshots
- Examples
  - WAFL (Network Appliance)
  - ZFS (Sun)

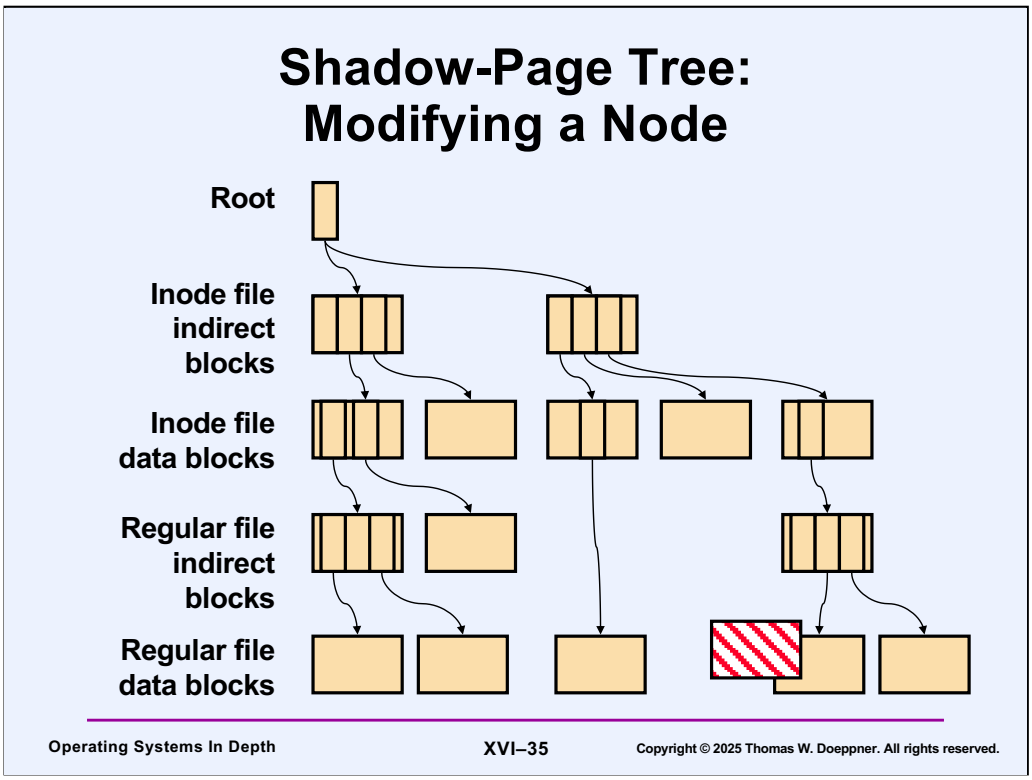


This and the next few slides are based on a description of the WAFL (Write-Anywhere File Layout) file system of Network Appliance Corporation, which is from:

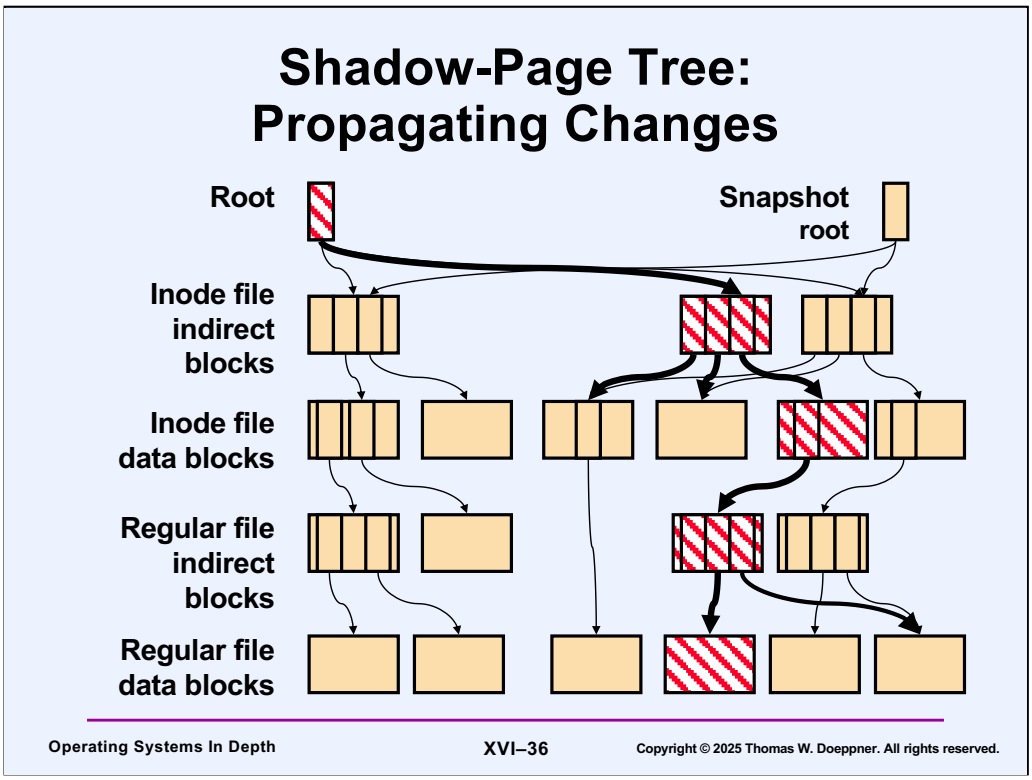
Hitz, D., J. Lau, M. Malcolm (1994). File System Design for an NFS File Server Appliance. Proceedings of USENIX Winter 1994 Technical Conference.

A file system is represented as a tree. Let's assume that the array of inodes is represented as a file, much as things are done in NTFS with its master file table (MFT). Thus we can look up an inode number in this inode file, obtaining the corresponding inode. From an inode we can find the blocks of the corresponding file. Let's assume that files (inode files and regular files) are represented via a disk map consisting of a number of indirect blocks referring to data blocks. In the diagram above, the root node points to the indirect blocks of the inode file. Each of these indirect blocks point to data blocks of the inode file (containing the actual inodes). The inodes contain disk maps for the files they represent, which are indirect blocks referring to data blocks.

Thus the leaves of the tree are the data blocks of all the files in the file system. As we'll see in the next two slides, links between levels of this file-system tree go in both directions (up as well as down).



Step1: A leaf node in a shadow-page tree is modified. A copy is made of the node, and the copy is modified.



Step 2: Copies are made of the leaf node's ancestors all the way up to the root, each modified so as to point to the copied node below it. The original root (snapshot root) points to the tree as it was before the leaf was modified. The new root points to the modified tree.

Benefits

- Each update results in a new shadow-page tree (having much in common with the previous one)
- The current root identifies the current tree
- If the system crashes after an update has been made, but before changes are propagated to the new root, the update is lost
  - a single write (to the root) effectively serves as a commit
- Older roots refer to previous states of the file system – snapshots

In practice, new snapshots aren't created after each update, but after a collection of updates – perhaps hourly.

Quiz 4

When the shadow-page tree is updated:

a) file-system data may be cached (and written asynchronously) as long as the root is written last

b) file-system data may be cached (and written asynchronously) only if all lower parts of the tree are written to disk before upper parts

c) all file-system data must be written to disk synchronously: writes may be cached for the sake of reads, but write system calls may not return until the data is on disk