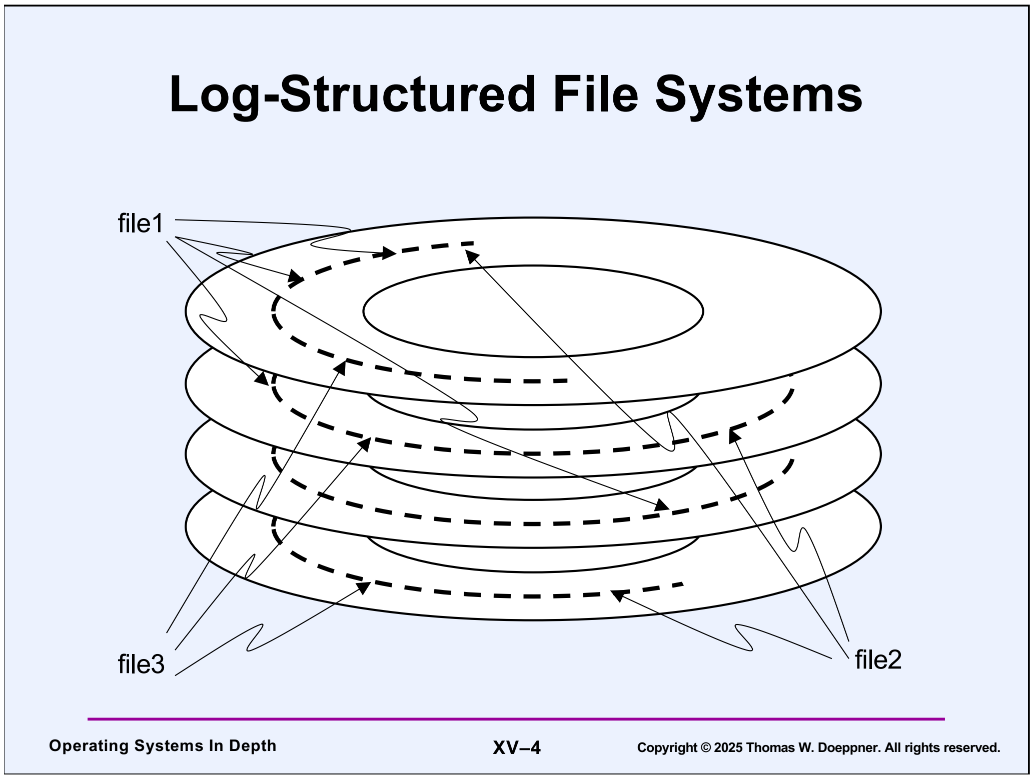
# File Systems Part 3

## File Servers

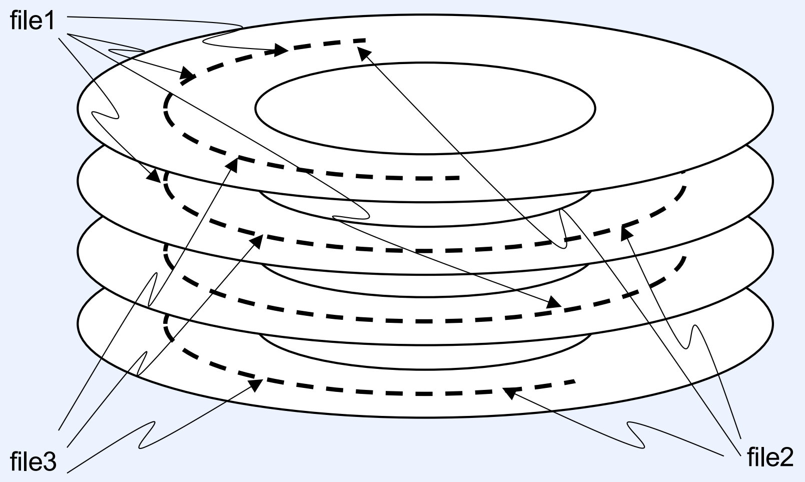
- Servicing many unrelated clients
  - successive requests to file system come from different clients
  - how can we optimize the use of the disk?

## A Different Approach

- We have lots of primary memory
  - enough to cache all commonly used files
- Read time from disk doesn’t matter
- Time for writes does matter



## Log-Structured File Systems

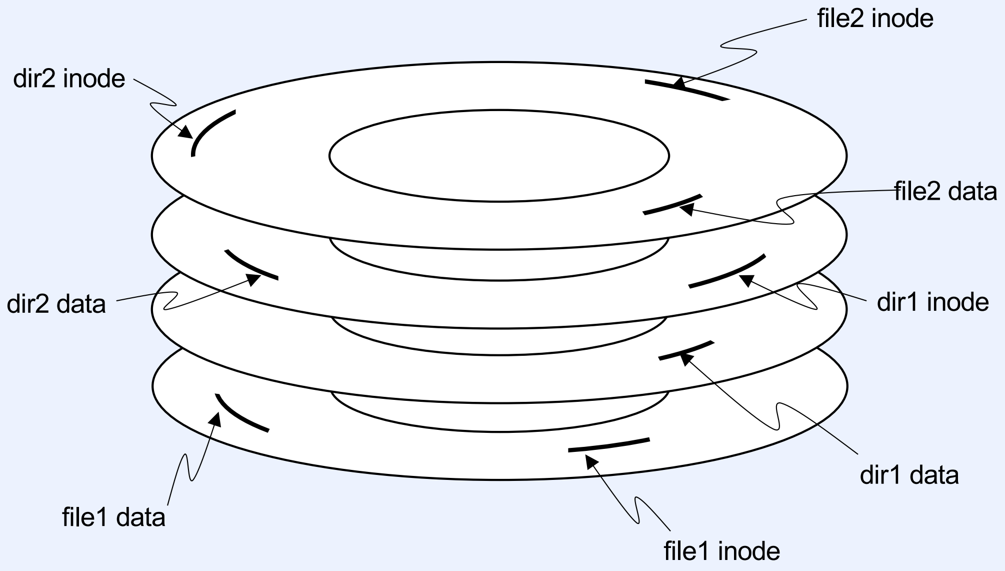


Portions of files are placed in a **log** (shown here as occupying most of one cylinder) in the order in which they are written.

## Example

- We create two single-block files
  - *dir1/file1*
  - *dir2/file2*
- FFS
  - allocate and initialize inode for *file1* and write it to disk
  - update *dir1* to refer to it (and update *dir1* inode)
  - write data to *file1*
    - allocate disk block
    - fill it with data and write to disk
    - update inode
  - six writes, plus six more for the other file
    - seek and rotational delays

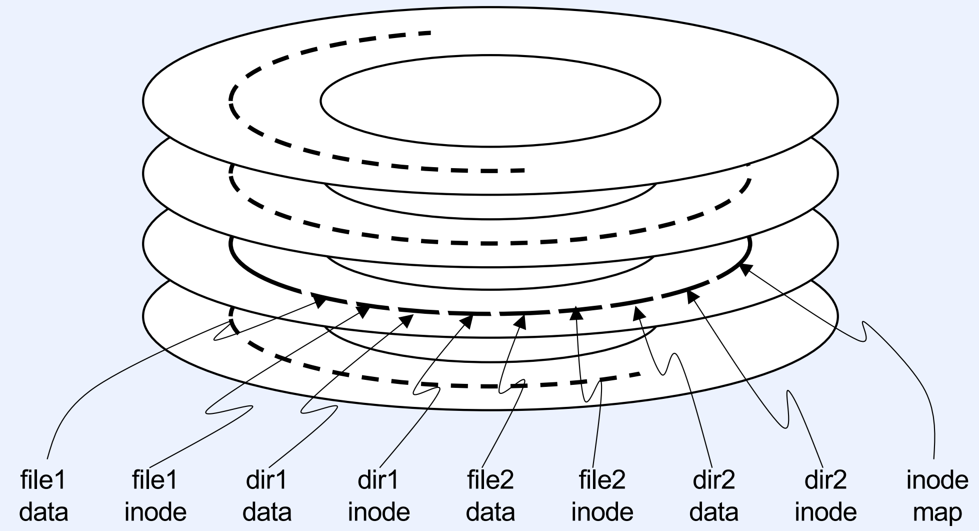
# FFS Picture


file2 inode
dir2 inode
file2 data
dir2 data
dir1 inode
dir1 data
file1 data
file1 inode

## Example (Continued)

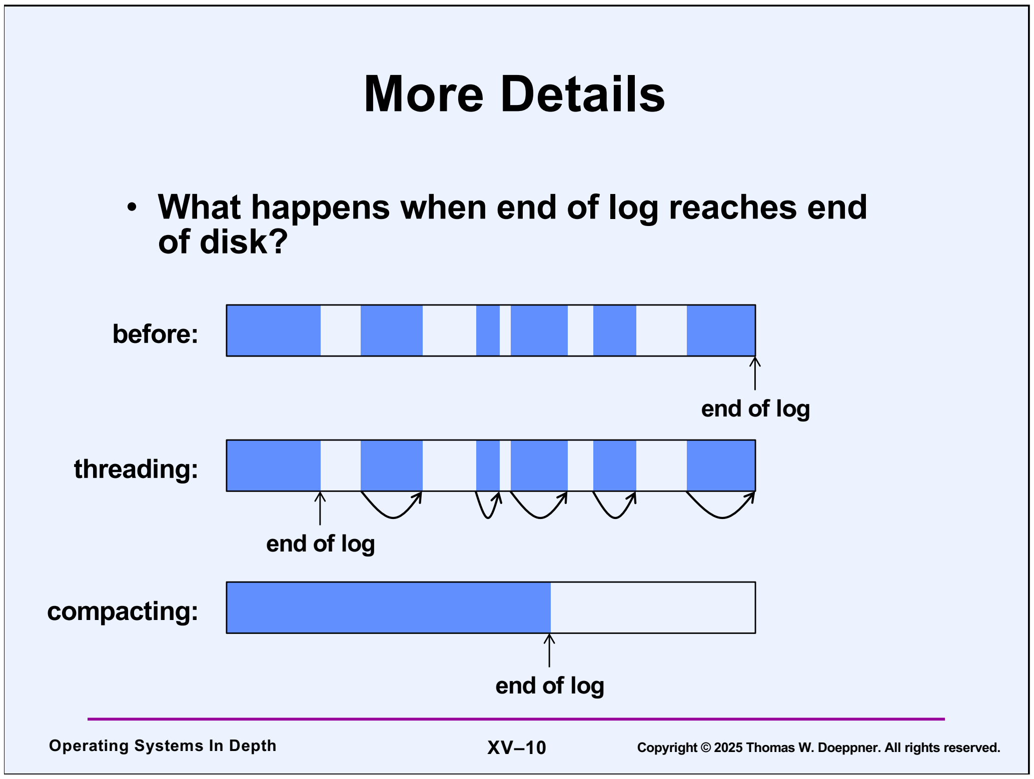
- Sprite (a log-structured file system)
  - one single, long write does everything

## Sprite Picture



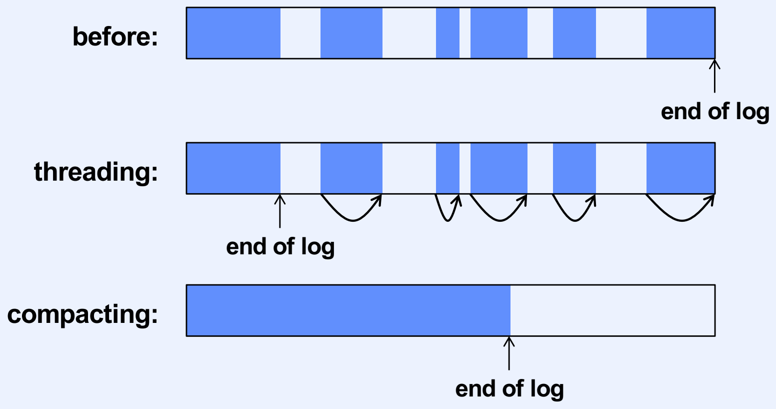
## Some Details

- Inode map cached in primary memory
  - indexed by inode number
  - points to inode on disk
  - written out to disk in pieces as updated
  - checkpoint file contains locations of pieces
    - written to disk occasionally
    - two copies: current and previous
- Commonly/recently used inodes and other disk blocks cached in primary memory



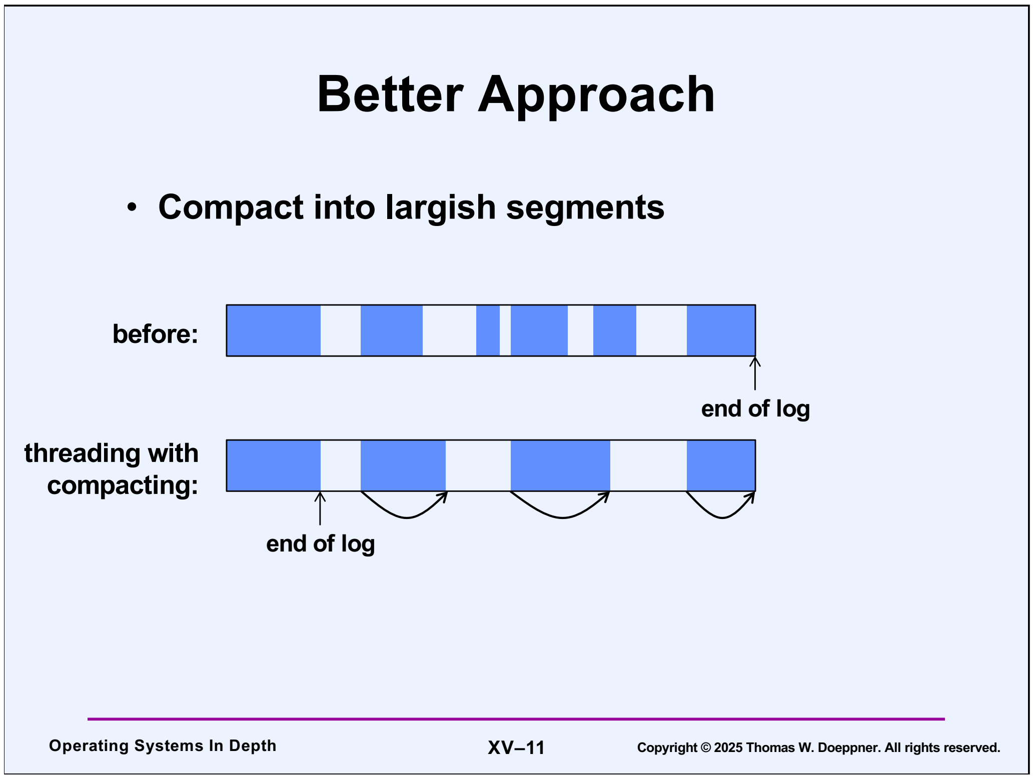
## More Details

- What happens when end of log reaches end of disk?



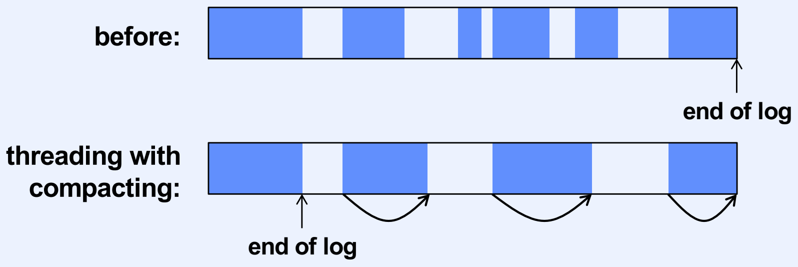
When the log reaches the end of the disk, what should happen? Presumably some space has been freed in the log, so there is space available on the disk. One possibility, known as **threading**, involves setting the end of the log to the beginning of the first region of free space on the disk, and linking together all such regions so the log continues to grow by using up each free region in turn. This is relatively cheap to implement, but the log could end up highly fragmented, negating the benefits of using a log.

Another possibility, known as **compacting**, involves copying the allocated regions of the log towards the beginning of the disk, leaving a single large region of free space at the end. This is time-consuming (because of the copying), but retains the benefits of logging as the log continues to grow.



## Better Approach

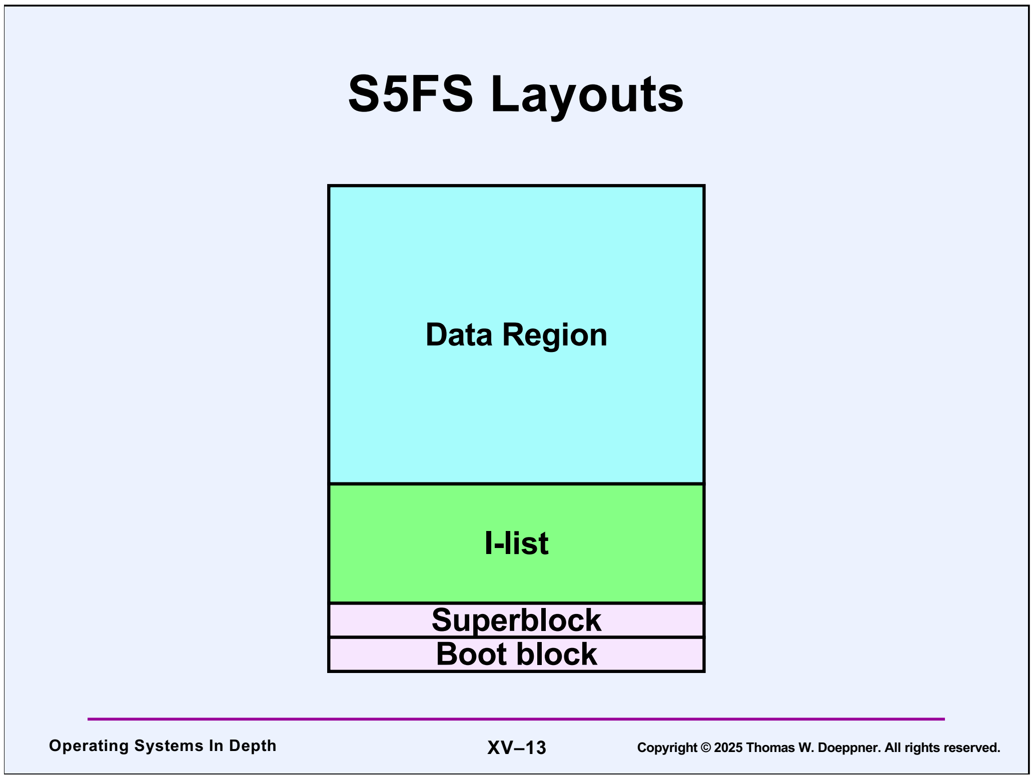
- Compact into largish segments



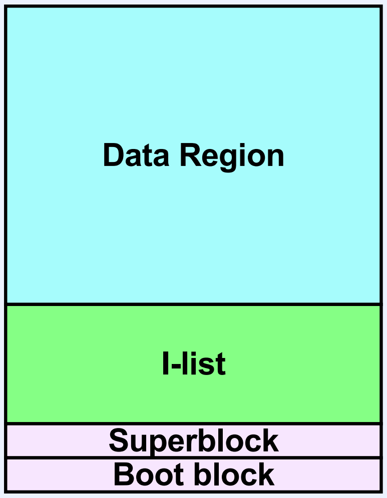
The solution used in Sprite is to compact portions of the log into rather large segments, leaving large gaps between them. The intent is that files that don't change often (or at all), won't have to be copied very often, and the gaps between the segments are large enough so that logging still makes sense.

## LFS

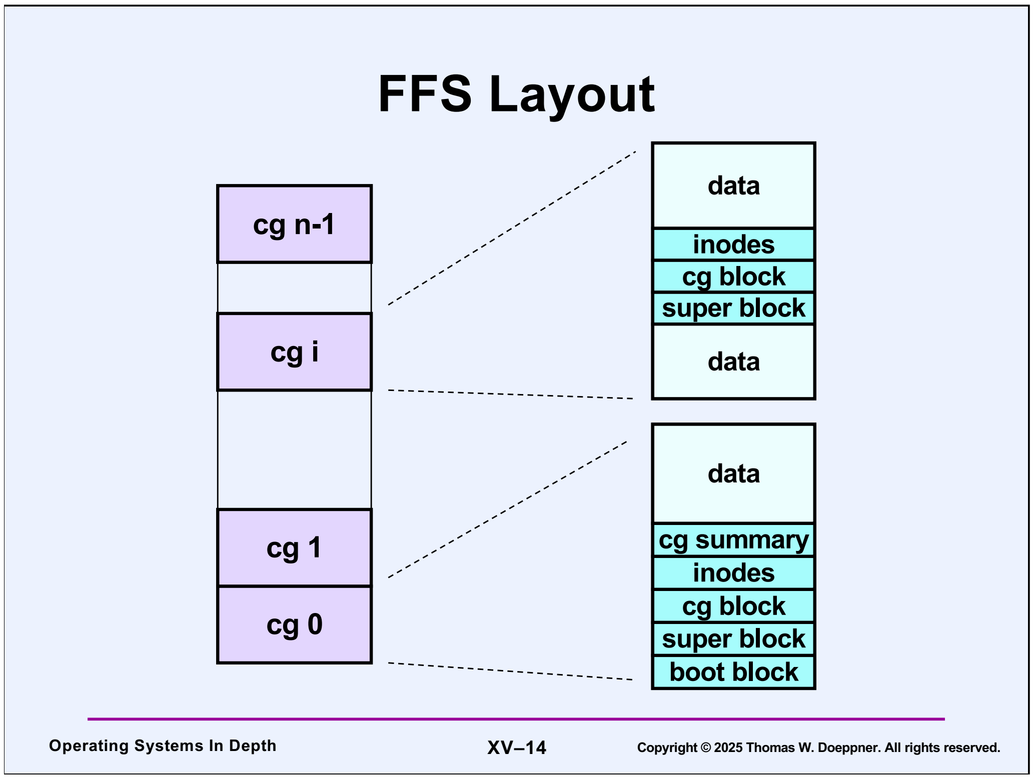
- Sprite was a proof of concept
  - developed by Mendel Rosenblum and John Ousterhout
- LFS (log-structured file system) extended Sprite
  - developed by Margo Seltzer, Keith Bostic, Kirk McKusick, and Carl Staelin
  - added extents
  - better integrated into Unix



## S5FS Layouts

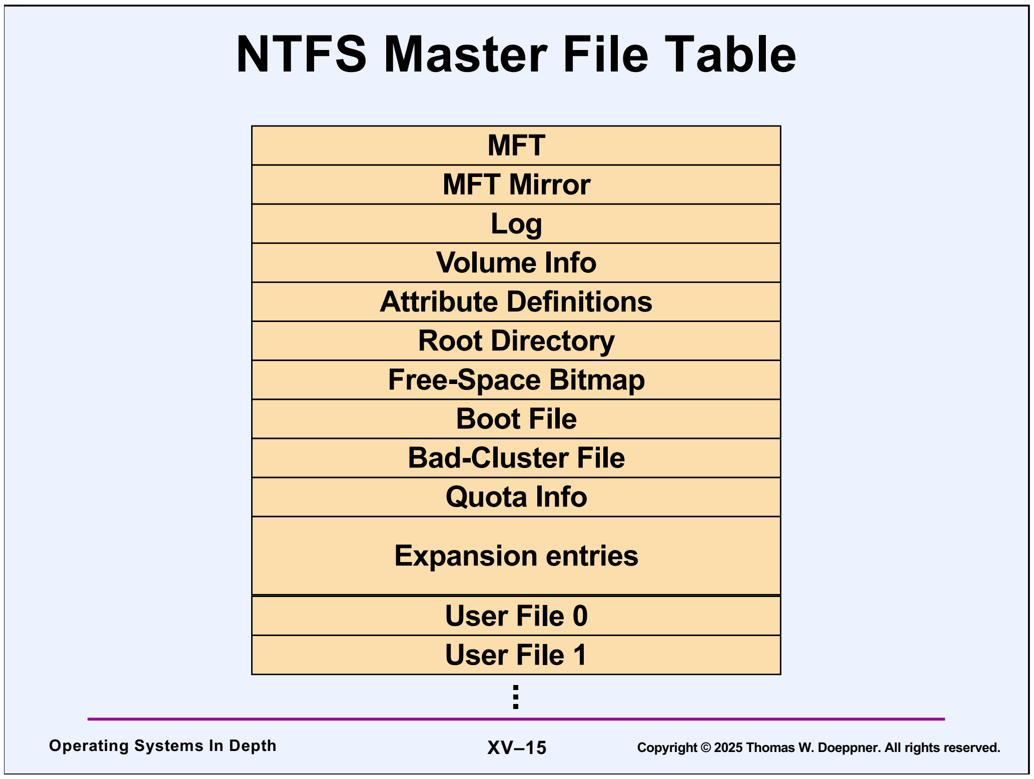


Note that the size of the I-List is fixed. What happens if the i-list is full, but there's plenty of space in the data region, or vice versa? It's far from trivial to reformat the disk so that the i-list gains space at the data region's expense.



## FFS Layout

The FFS format, while improving performance, doesn't solve the problem of having to reformat the disk if the relative amounts of space for inodes and data must be changed.



## NTFS Master File Table

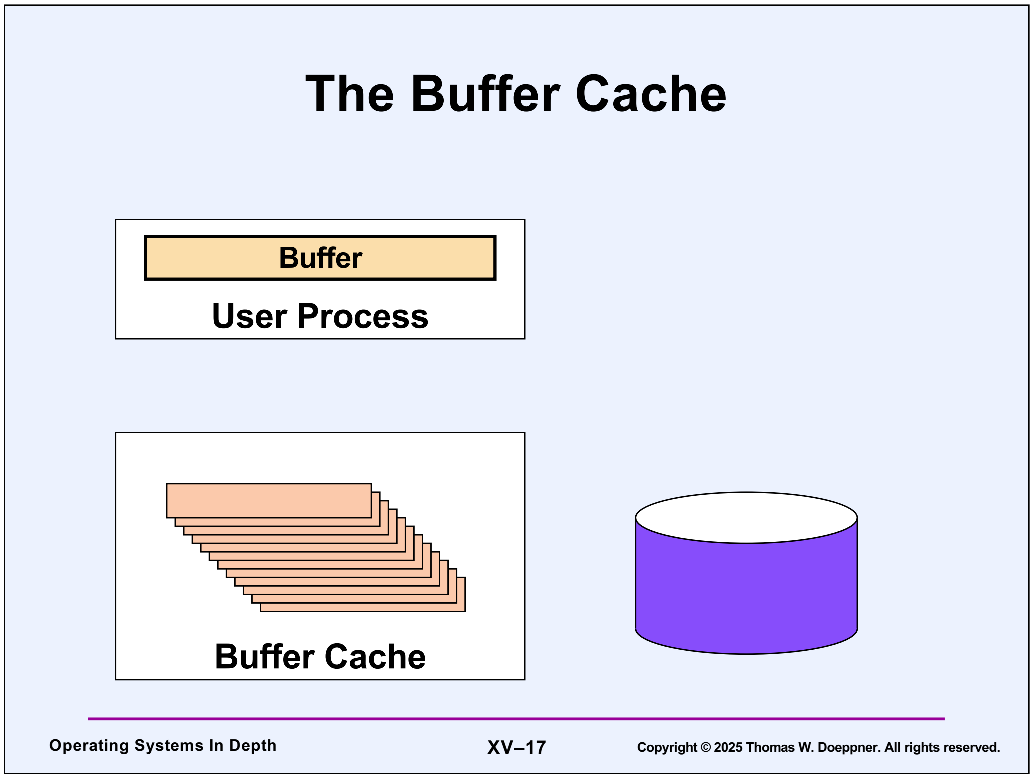
NTFS's Master File Table (MFT). Each entry is one kilobyte long and refers to a file. Since the MFT is a file, it has an entry in itself (the first entry). There's a copy of the MFT called the MFT mirror. There are a number of other entries for system files and metadata files, and then the entries for ordinary files.

Each entry of the MFT plays the role of an inode. But, since the MFT is a file that, like any other file, may grow and shrink dynamically, these inode equivalents are not allocated statically but are allocated dynamically.

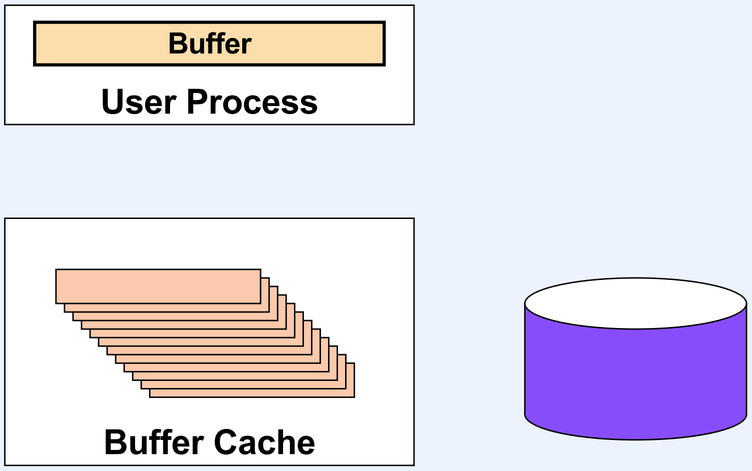
## Quiz 1

Your disk drive is nearing capacity. So you buy a new, larger drive to replace it. You copy your old disk drive to the new one. What else has to be done to take advantage of the larger disk? Assume you're using NTFS.

a) Nothing
b) Modify the free-space bitmap
c) Modify the free-space bitmap and adjust the extent lists for all files
d) All of the above, plus more



## The Buffer Cache



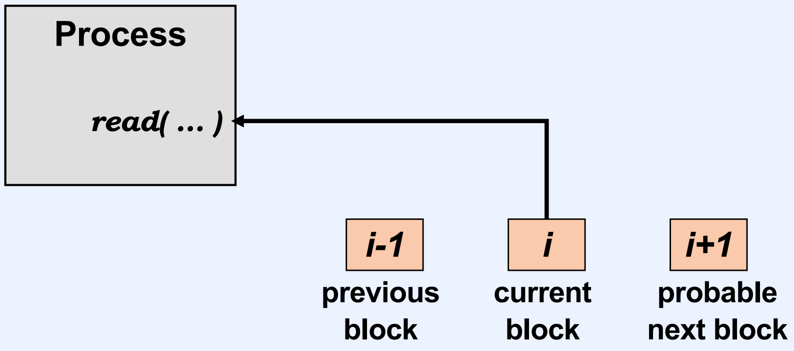
File I/O in Unix, and in most operating systems, is not done directly to the disk drive, but through an intermediary, the **buffer cache**.

The buffer cache has two primary functions. The first, and most important, is to make possible concurrent I/O and computation within a Unix process. The second is to insulate the user from physical block boundaries.

From a user thread's point of view, I/O is **synchronous**. By this we mean that when the I/O system call returns, the system no longer needs the user-supplied buffer. For example, after a write system call, the data in the user buffer has either been transmitted to the device or copied to a kernel buffer—the user can now scribble over the buffer without affecting the data transfer. Because of this synchronization, from a user thread's point of view, no more than one I/O operation can be in progress at a time. Thus user-implemented multibuffered I/O is not possible (in a single-threaded process).

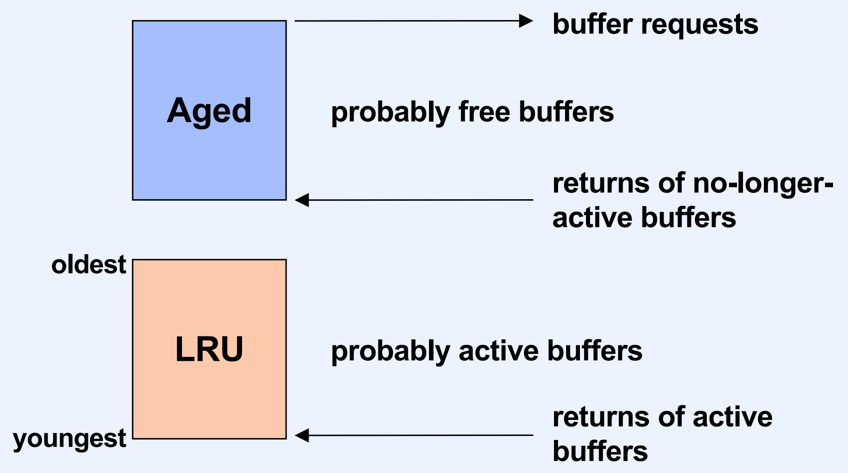
The buffer cache provides a kernel implementation of multibuffered I/O, and thus concurrent I/O and computation are possible even for single-threaded processes.

## Multi-Buffered I/O



The use of **read-ahead**s and **write-behind**s makes possible concurrent I/O and computation: if the block currently being fetched is block *i* and the previous block fetched was block ***i-1***, then block **i+1** is also fetched. Modified blocks are normally written out not synchronously but instead sometime after they were modified, asynchronously.

## Maintaining the Cache



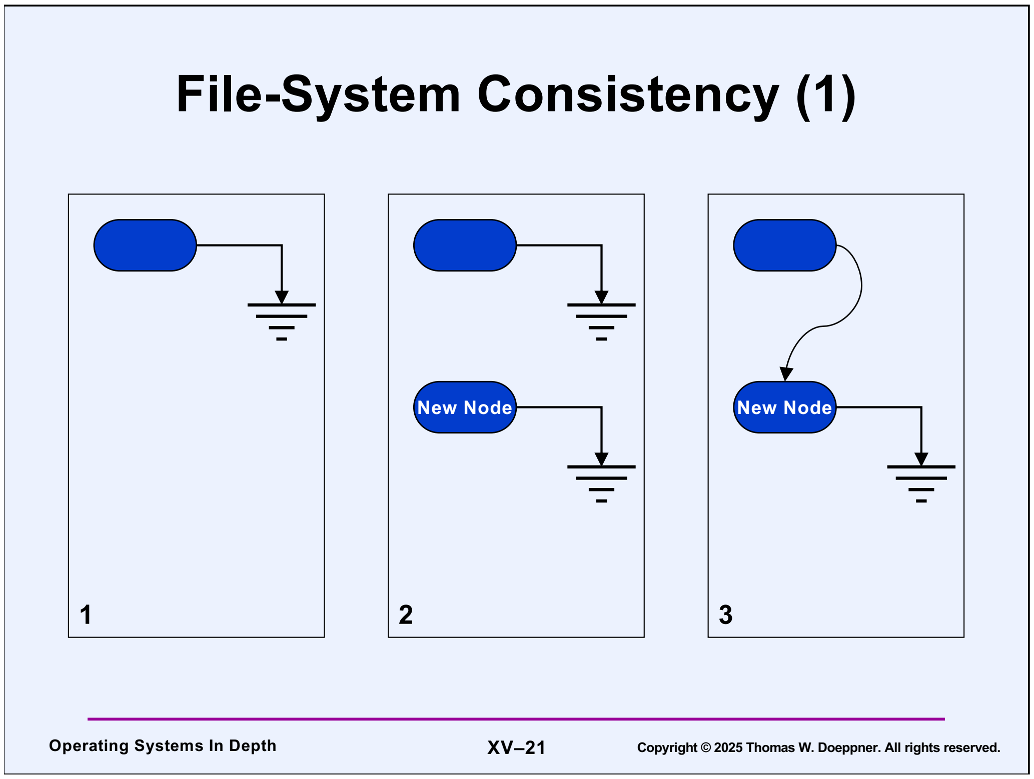
The scheme shown here for maintaining the cache is that used with the FFS file system of Berkeley Unix. Active buffers are maintained in least-recently-used (LRU) order in the system-wide LRU list. Thus, after a buffer has been used (as part of a **read** or **write** system call), it is returned to the end of the LRU list. The system also maintains a separate list of "free" buffers called the aged list. Included in this list are buffers holding no-longer-needed blocks, such as blocks from files that have been deleted.

Fresh buffers are taken from the aged list. If this list is empty, then a buffer is obtained from the LRU list as follows. If the first buffer (least recently used) in this list is clean (i.e., contains a block that is identical to its copy on disk), then this buffer is taken. Otherwise (i.e., if the buffer is dirty), it is written out to disk asynchronously and, when written, is placed at the end of the aged list. The search for a fresh buffer continues on to the next buffer in the LRU list, etc.

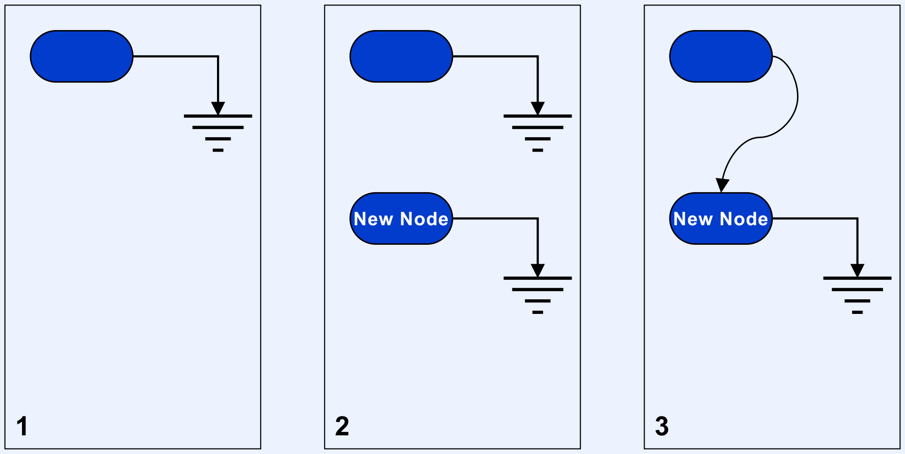
When a file is deleted, any buffers containing its blocks are placed at the head of the aged list. Also, when I/O into a buffer results in an I/O error, the buffer is placed at the head of the aged list.

## In the Event of a Crash …

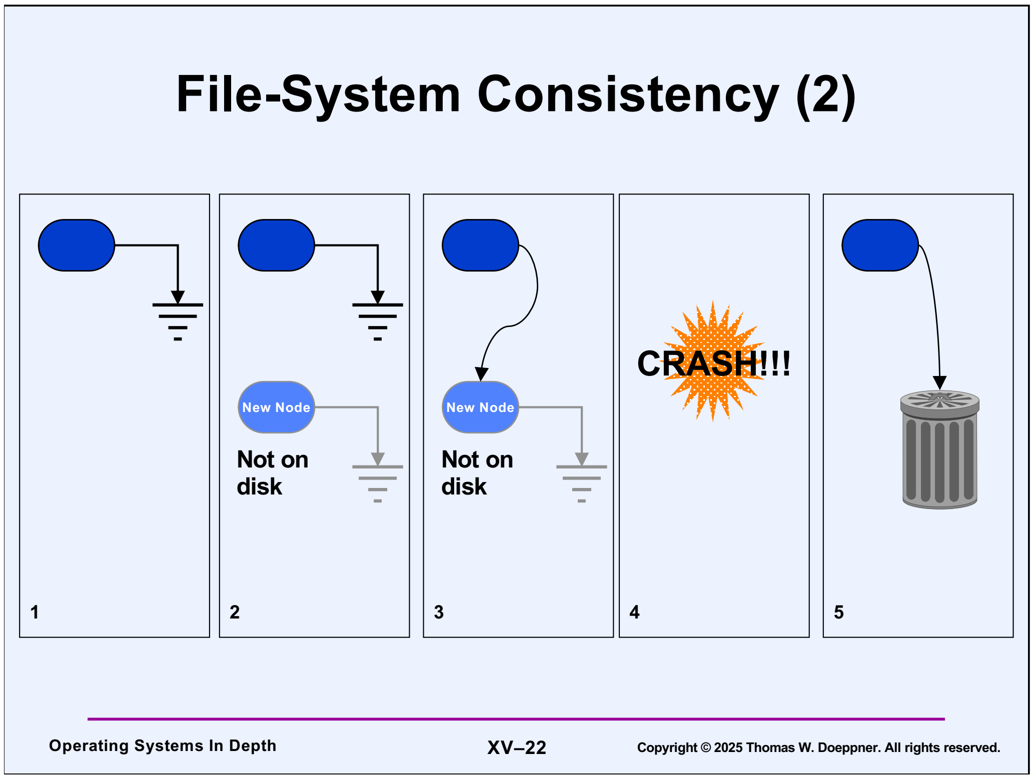
- Most recent updates did not make it to disk
  - is this a big problem?
    - equivalent to crash happening slightly earlier
      - but you may have received (and believed) a message:
        - “file successfully updated”
        - “homework successfully handed in”
        - “stock successfully purchased”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



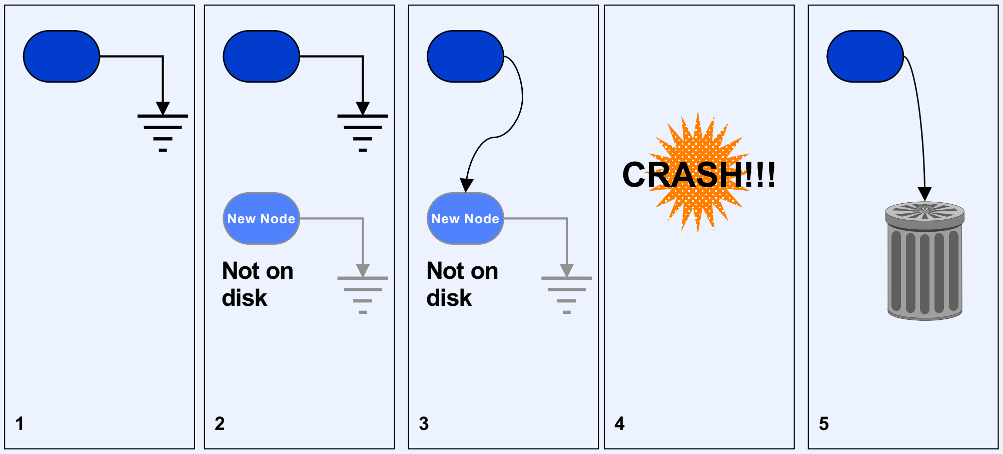
## File-System Consistency (1)



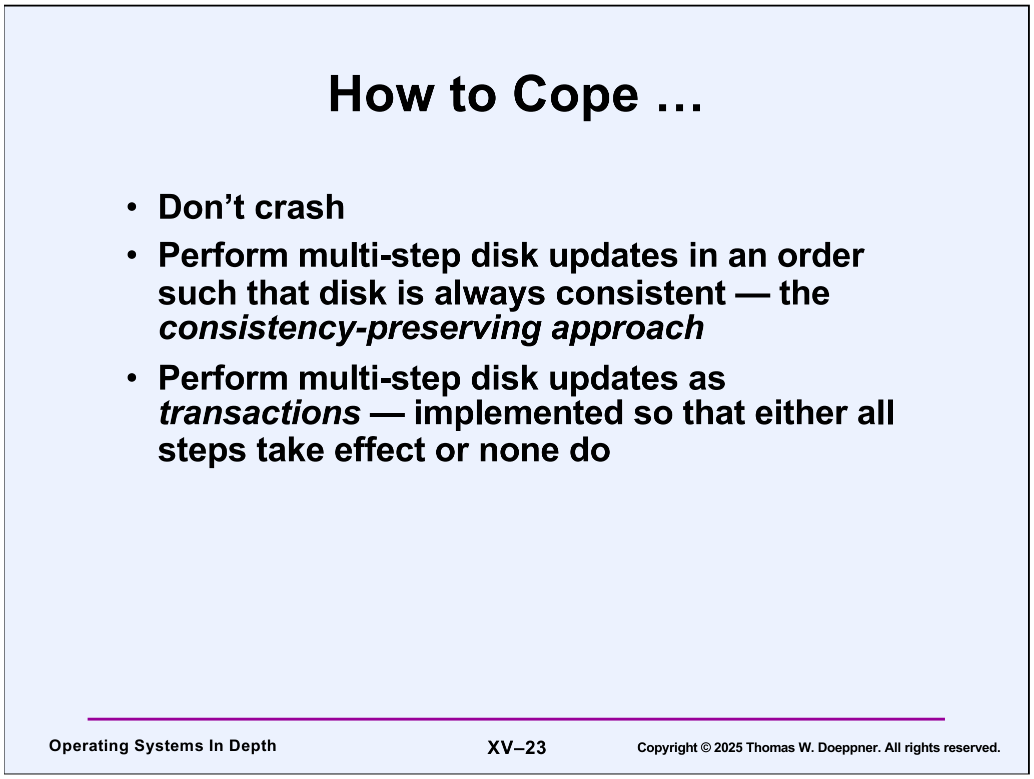
In the event of a crash, the contents of the file system may well be inconsistent with any view of it the user might have. For example, a programmer may have carefully added a node to the end of the list, so that at all times the list structure is well-formed.



## File-System Consistency (2)



But, if the new node and the old node are stored on separate disk blocks, the modifications to the block containing the old node might be written out first; the system might well crash before the second block is written out.

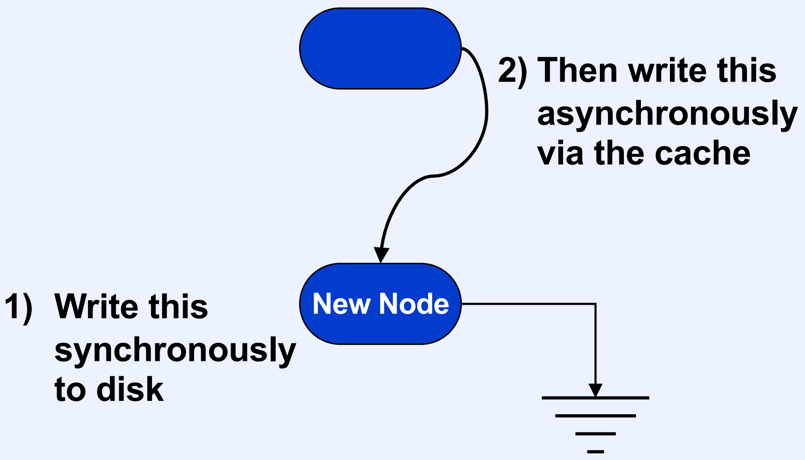


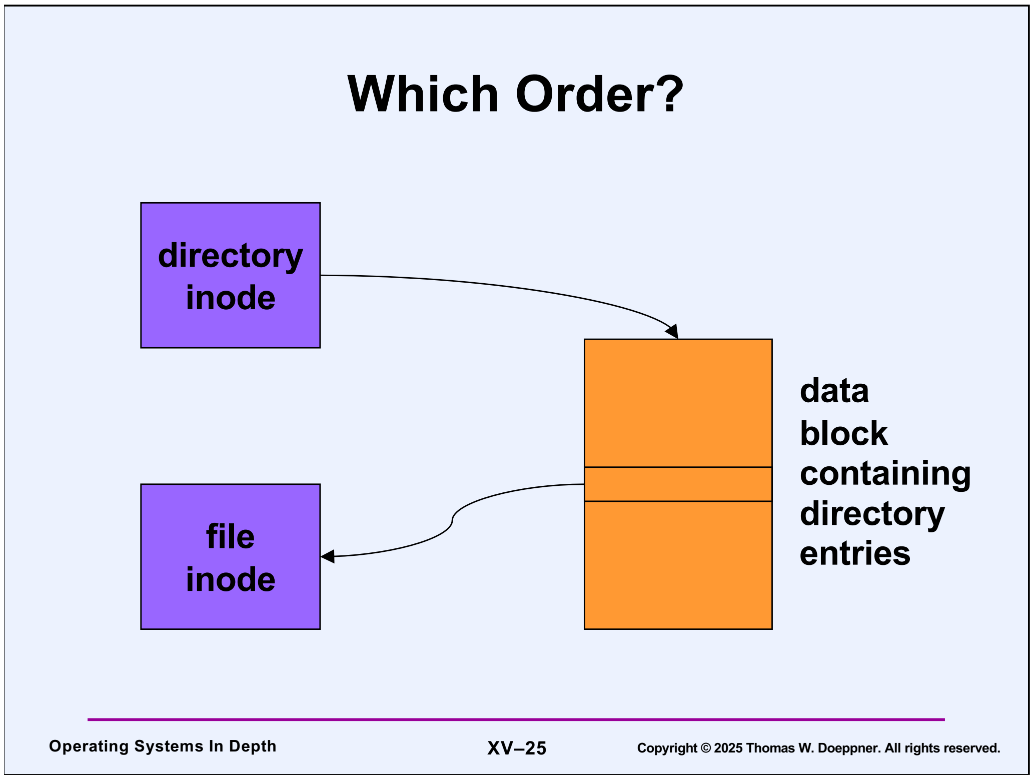
## How to Cope …

- Don’t crash
- Perform multi-step disk updates in an order such that disk is always consistent — the *consistency-preserving approach*
- Perform multi-step disk updates as *transactions* — implemented so that either all steps take effect or none do

The first approach is the most efficient, but tough to implement in practice …

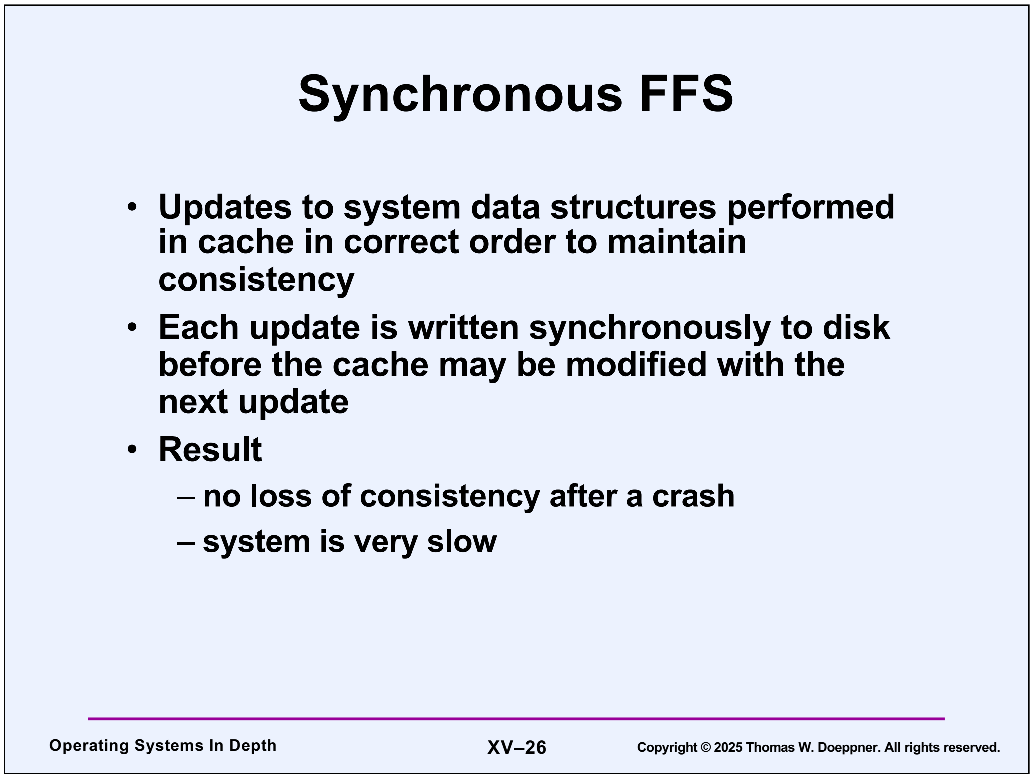
## Maintaining Consistency


2) Then write this asynchronously via the cache
New Node
1) Write this synchronously to disk



## Which Order?

Here we've created a file in a directory, necessitating that a new data block be allocated to hold the directory entry. Thus, as shown in the slide, three items must be written to disk: the file's inode, the directory's new data block, and the directory's inode. In what order should the blocks be written to ensure that no crash will leave the disk in an inconsistent state? Clearly first the file's inode, then the directory's data block, then the directory's inode.



## Synchronous FFS

- Updates to system data structures performed in cache in correct order to maintain consistency
- Each update is written synchronously to disk before the cache may be modified with the next update
- Result
  - no loss of consistency after a crash
  - system is very slow

”Synchronous FFS” is not a system one would want to use, but is strictly a benchmark for comparison purposes.

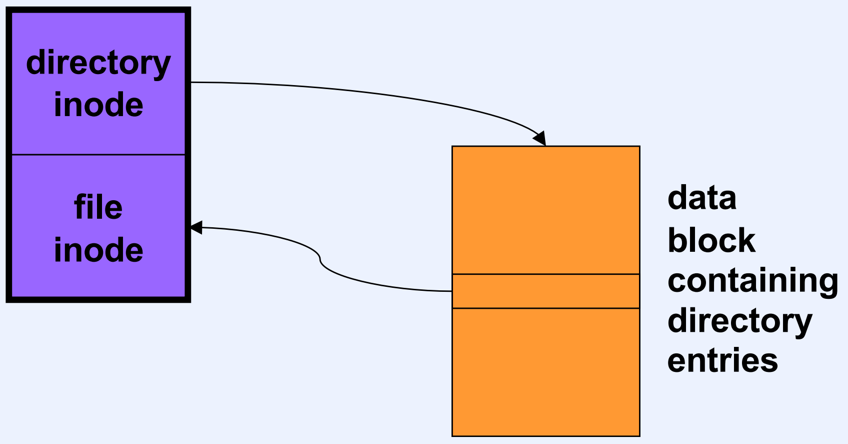
"Written synchronously" means that the next update to the cache can't begin until the disk write completes.

## Soft Updates

- An implementation of the consistency-preserving approach
  - should be simple:
    - update cache in an order that maintains consistency
    - write cache contents to disk in same order in which cache was updated, but without requiring synchronous writes
  - isn't …

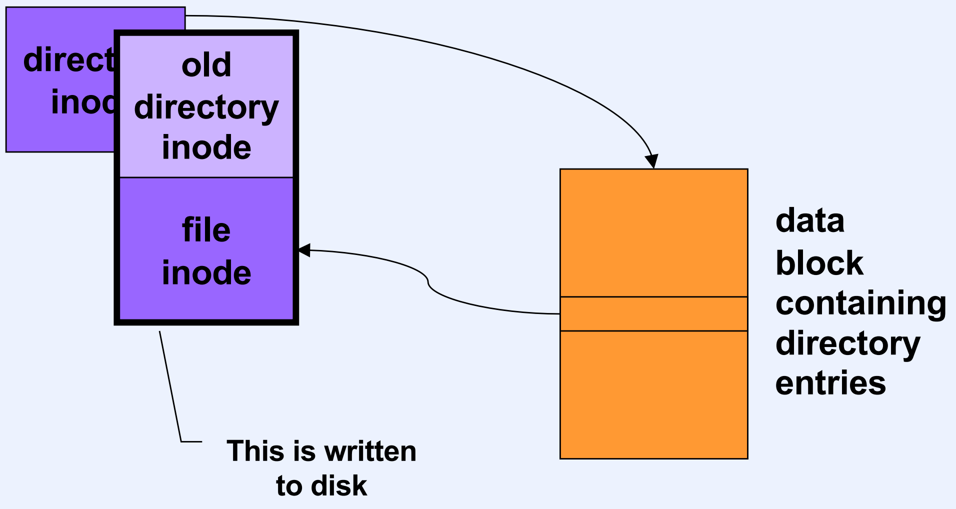
Soft updates is an approach that relaxes the constraints of FFS requiring synchronous updating of system data structures on disk.

## Problem …

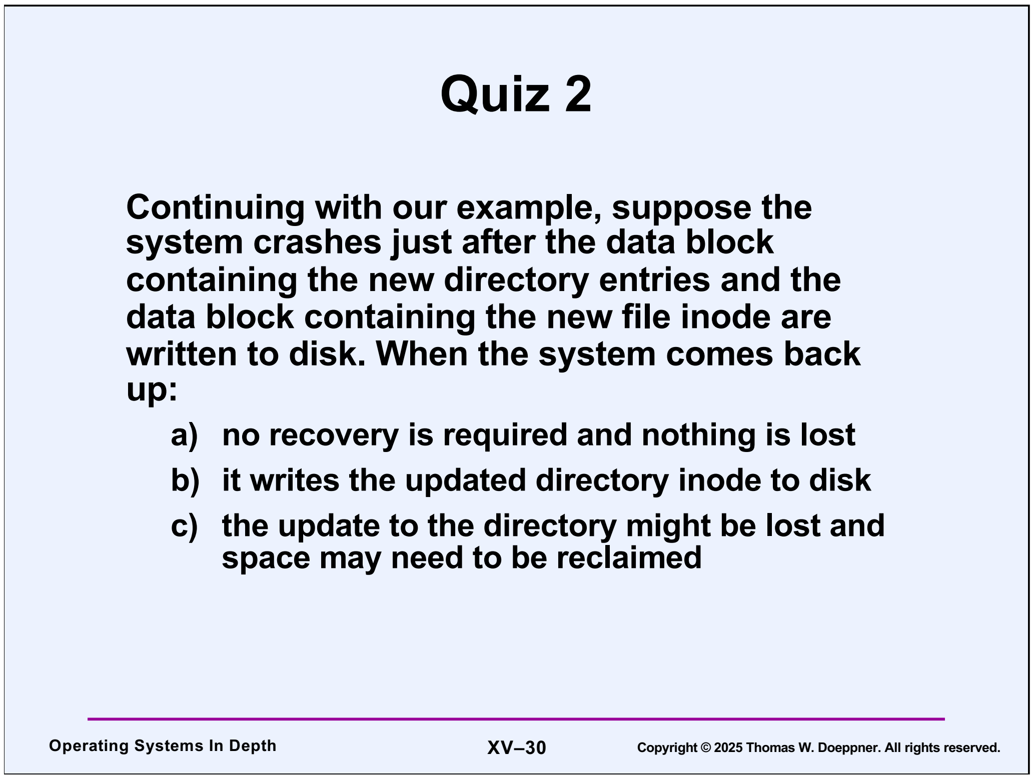


Inodes aren't large, so a number of them are put into a single disk block. It might happen then that the directory's inode is in the same disk block as the file's inode, and thus it's impossible to write one without writing the other. So if the buffer cache contains the block containing both inodes as well as the block containing the data block, and both blocks have not yet been written to disk, it's impossible to update the disk in a way that keeps it always consistent. We have a circular dependency that we hadn't expected.

## Soft Updates



In the soft updates approach, when a cached block must be written out that contains an item that shouldn't be written yet, that item is replaced with its prior contents, the block is written, and then the item is restored in the cache, to be written later.



## Quiz 2

Continuing with our example, suppose the system crashes just after the data block containing the new directory entries and the data block containing the new file inode are written to disk. When the system comes back up:

a) no recovery is required and nothing is lost

b) it writes the updated directory inode to disk

c) the update to the directory might be lost and space may need to be reclaimed

Assume that updating the directory involved adding new entries at its end.

## Quiz 3

When a disk block is allocated, three things happen: the free vector is modified, some data structure (perhaps an inode) is modified to refer to the new block, and the new block is written to. What is the correct order for updating the disk, so as to preserve consistency?

a) inode, free vector, disk block
b) inode, disk block, free vector
c) disk block, inode, free vector
d) free vector, inode, disk block
e) free vector, disk block, inode

## Soft Updates in Practice

- Implemented for FFS in 1994
- Used in FreeBSD’s FFS
  - improved performance (over FFS with synchronous writes)
  - disk updates may be many seconds behind cache updates

## 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

## Example Transactions (1)

- Create file
  - create inode
    - modify free vector/list
    - initialize inode
  - update directory
    - modify contents
      - possibly modify free vector
    - update directory inode

## Example Transactions (2)

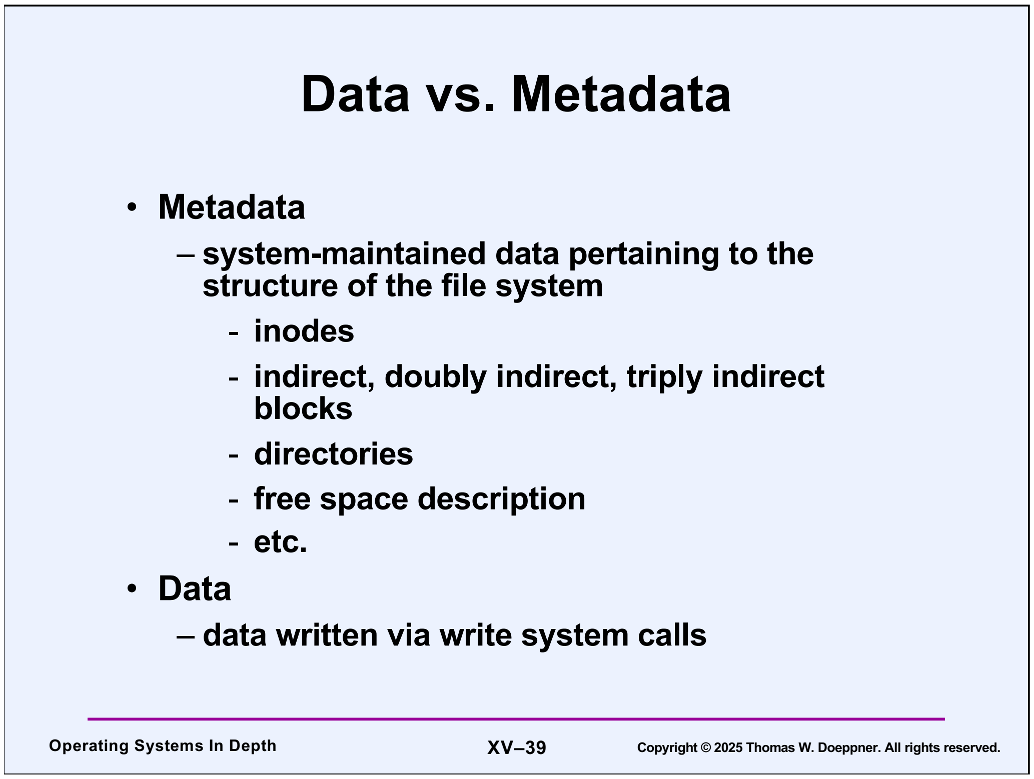
- Rename file
  - update new directory
    - update new directory inode
    - modify new directory
      - update free vector
  - update old directory
    - update old directory inode
    - modify old directory
      - update free vector

## Example Transactions (3)

- Write to a file
  - for each block
    - update free vector
    - copy data to block
  - update inode

## Example Transactions (4)

- Delete a file
  - for each block
    - update free vector
  - update inode free vector/list
  - update directory
    - update directory inode
    - modify directory
      - update free vector



## 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