CS3281 / CS5281

Journaling Filesystems

Will Hedgecock
Sandeep Neema
Bryan Ward

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

VANDERBILT UNIVERSITY

Review of Filesystems

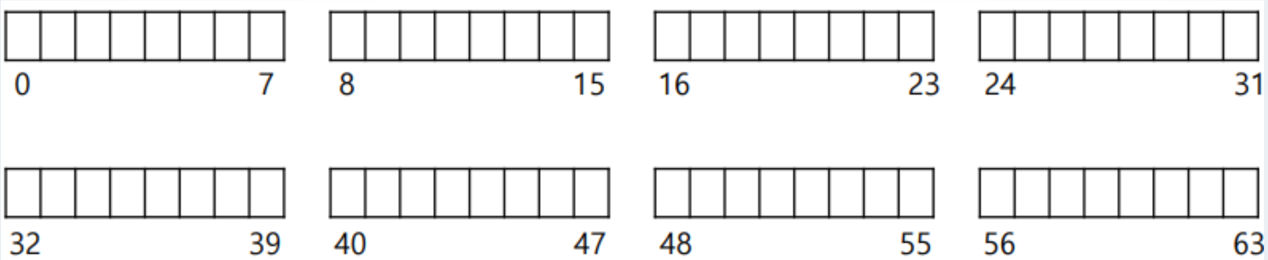
- A filesystem is an organized collection of files and directories
- Linux filesystems use an i-node to store a file's metadata
  - Metadata includes things like file size and permissions
  - Metadata does *not* include a file name
- Today we'll look at journaling and extents

Filesystem Aspects

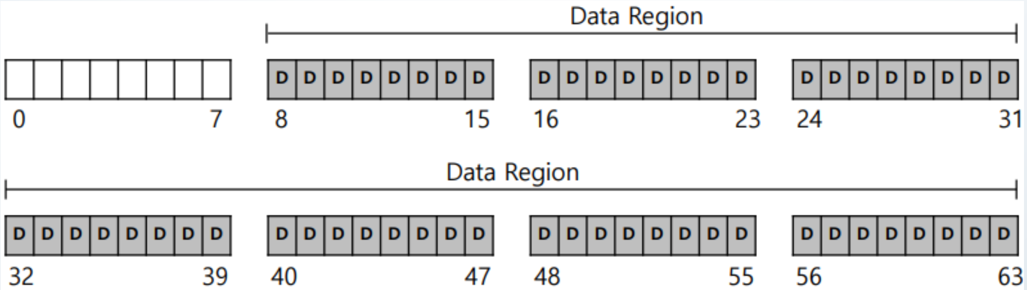
- File system is an on-disk data structure
- Study Very Simple File System (VSFS)
  - A simplified version of UNIX file system
- Two aspects of file system
  - Data structures: on-disk structures to store data and metadata
  - Access methods: mapping system calls, e.g., open(), read(), and write(), to particular structures

Very Simple File System (VSFS) Data Structures

- Divide disk into blocks
- Use one block size (4KB)
- Blocks are addressed from 0 to *N*-1 (N is the number of blocks)

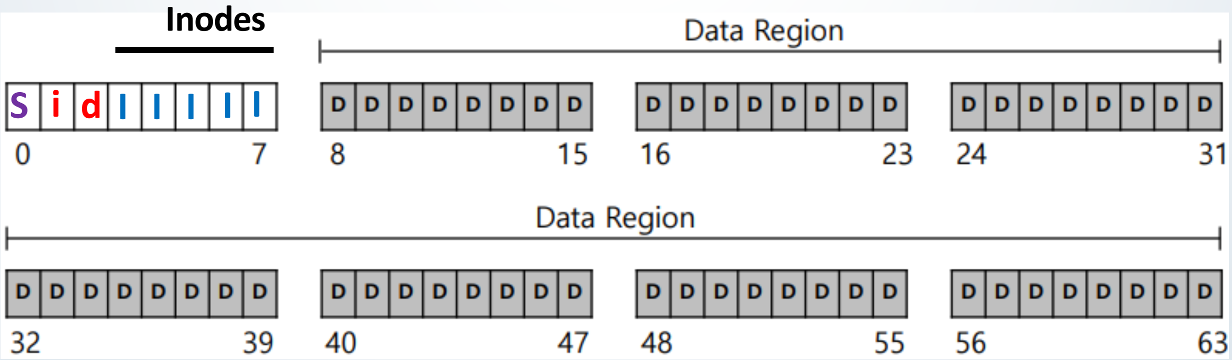


- Store user data in *data region* (e.g., files and directories)



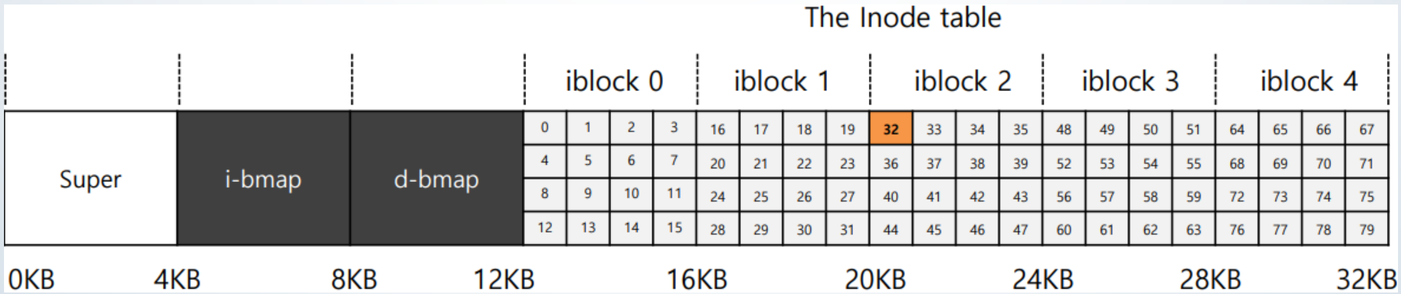
VSFS Data Structures (cont.)

- File system tracks information (metadata) about files.
  - E.g., a subset of data blocks that form a file, file size, access rights
- Metadata is stored in a structure called *inode*
  - inode table is an array of inodes
  - 256-byte inode
  - 16 inodes per block; 80 inodes in 5 blocks
- Use bitmap for tracking free inodes and data blocks
  - A bit indicates whether an inode or a block is free.
- Superblock stores information about a certain file system.
  - E.g., number of inodes and data blocks, the start of inode table



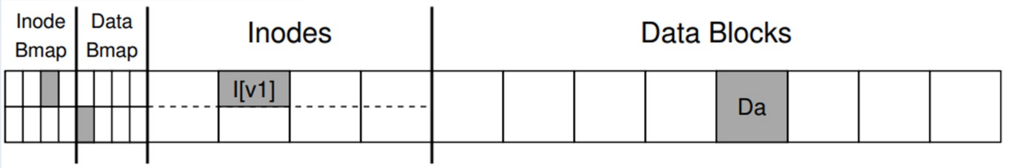
The I-Node

- Index node (inode): array of nodes is indexed
- Each inode is identified by an i-number
  - Used for indexing an array of inodes
- Find the byte address for the inode with i-number 32
  - Compute the offset into the inode table: 32 * sizeof(inode) = 8192 (8KB)
  - Add the offset to start address of inode table: 12KB + 8KB = 20KB
- inodes are fetched using *sectors* (a block consists of sectors)
  - 512-byte sectors
  - Sector number: (20 * 1024)/512 = 40



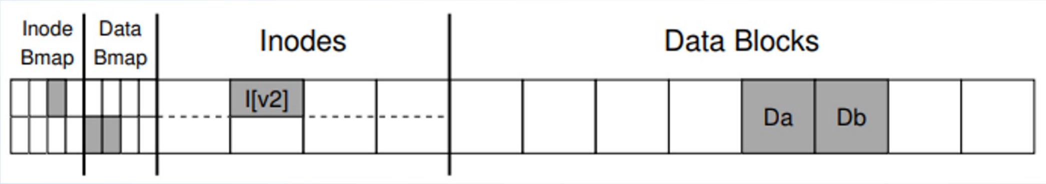
Crash Scenario

- Consider the following simple filesystem scenario from the textbook
  - One file with one allocated data block




```
owner        : remzi
permissions  : read-write
size         : 1
pointer      : 4
pointer      : null
pointer      : null
pointer      : null
```

- Suppose we want to append to this file; we have to
  - (1) write the data to a new data block, (2) update the i-node, (3) update the data bitmap
    - This requires three (separate) writes to disk
  - For example, we want the updated filesystem to look like this:



Crash Scenario (cont.)

- But what if the system crashes after only 1 of the writes is performed?
  - **Just the data block is written**: the data is there, but the i-node doesn't know about it
  - **Just the i-node is updated**: the data isn't there, so we'll read garbage from disk
    - And the filesystem is inconsistent: the i-node points to a data block, but the data block bitmap thinks it hasn't been allocated.
  - **Just the data block bitmap is updated**: no i-node points to the data block, so there's a data leak

- What if the system crashes after 2 writes (out of 3) are performed?
  - **I-node and bitmap updated, data block not updated**: file contains garbage
  - **I-node and data block are written**: the bitmap thinks the block hasn't been allocated
  - **Bitmap and data block are written**: no i-node points to the data (so the data is effectively lost!)

The Crash Consistency Problem

- Crashes cause inconsistency in file system data structures.
  - E.g., space (data) leaks, data block containing garbage
- Disk performs one write at a time.
  - Crashes or power loss occur between writes.

A Solution: Filesystem Check

- A filesystem check (fsck) scans the filesystem for inconsistencies and repairs them
  - Runs before filesystem is made available to users
    - fsck:
      - Check the superblock; if it looks corrupted, you can try to use an alternate copy
      - Scan the i-nodes to see which blocks are in use and resolve inconsistencies in the allocation bitmaps
      - Check that each i-node is ok
      - Scan the entire directory tree (starting at root dir) and verify link counts
      - Check for bad blocks (e.g., data pointer points to an invalid address); just clear the bad pointers
      - Check that directories are consistent (e.g., contain "." and "..", there are i-nodes for each file entry)

Journaling

- A filesystem check can be very slow, especially for large filesystems
- An alternative is journaling, also called write-ahead logging
- Simple idea: before updating the on-disk structures, write information about the update to a *journal* (also called a log)
  - If there's a crash before the disk can be updated, the journal contains enough information to recover
    - You don't have to scan the entire disk!

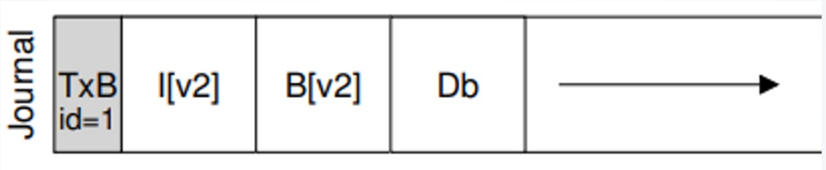
Example: ext2 vs. ext3

- The basic layout of ext2 (no journaling):

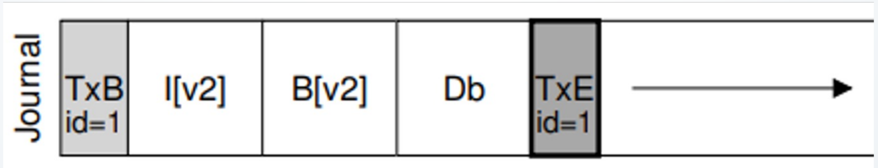
- The basic layout of ext3 (with journaling):

Journaling Phases

- Four basic phases:
  - Journal write: write contents of transaction to journal; wait for these to complete



  - Journal commit: write the transaction commit block; wait for write to complete; transaction is committed



  - Checkpoint: write the contents of the update (metadata and data) to disk
  - Free: mark the transaction as “free” in the journal by updating the journal superblock
    - Done at “some point” in the future

Timeline for Data Journaling

Journal contents				File System	
TxB	(metadata)	(data)	TxE	Metadata	Data
issue	issue	issue			
complete					
	complete				
		complete	issue		
			complete		
				issue	issue
					complete
				complete	

any order

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu
VANDERBILT UNIVERSITY

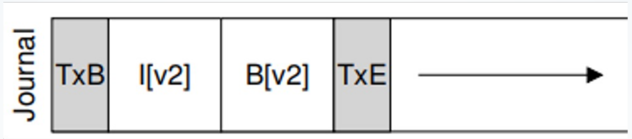
Recovery

- When a crash happens before *journal commit* completes, pending updates in a transaction are skipped.

- When a crash happens after *journal commit* completes and before *checkpoint* completes, transactions are replayed (redo logging)
  - File system finds out transactions that were committed successfully
  - The blocks of those specific transactions are written to their final disk locations again

Ordered (Metadata) Journaling

- Instead of journaling both data and metadata, just do metadata



- In this case, our journaling protocol is:
  - **Data write**: write data to its final location
    - If we crash after this step but before the rest, only the data is lost -- bad for the user, but at least there’s no inconsistency in the filesystem
  - **Journal metadata** write: write metadata to the journal
  - **Journal commit**: write the transaction commit block
  - **Checkpoint metadata**: write the metadata update to final location in filesystem
  - **Free**: sometime later, mark the transaction free in the journal superblock

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

Timeline for Metadata Journaling

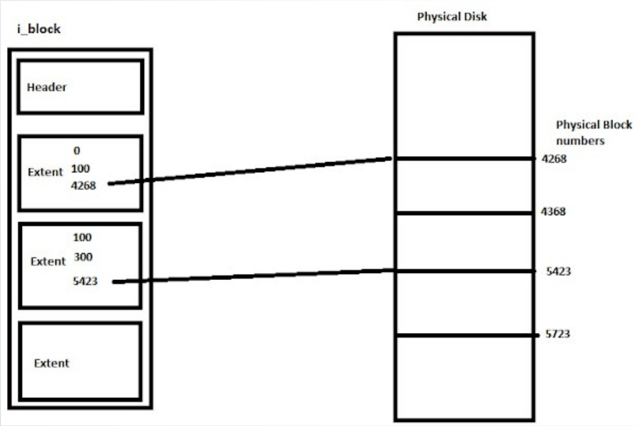
Journal			File System	
TxB	Journal contents (metadata)	TxE	Metadata	Data
issue	issue			Issue
				complete
complete				
	complete			
		issue		
		complete		
			issue	
			complete	

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

VANDERBILT UNIVERSITY

Extents

- Recall that ext2 and ext3 use indirect pointers to data blocks (Figure on right)
  - Can be inefficient!
- The ext4 filesystem uses extents
  - An extent specifies an (1) an initial block address, and (2) the number of blocks in the extent
    - Fragmented file will have multiple extents



http://ext2read.blogspot.com/2010/03/how-ext4-extents-work-earlier-ext2-and.html

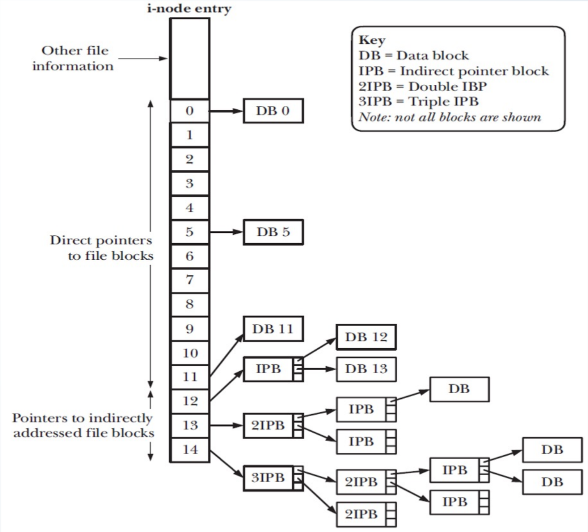


Figure 14-2: Structure of file blocks for a file in an *ext2* file system

*Figure from *The Linux Programming Interface* by Michael Kerrisk