Persistence

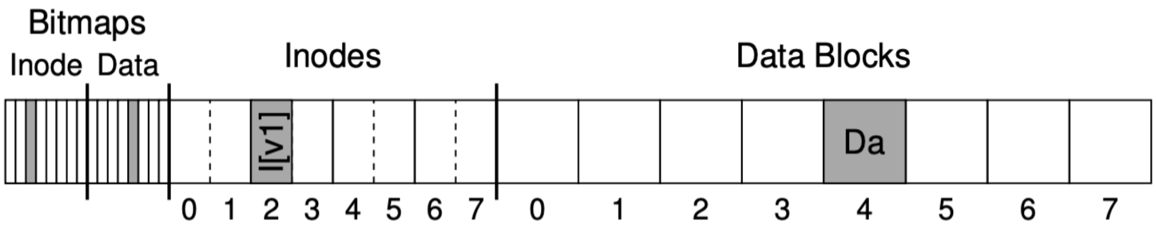
- 36 - I/O Devices
- 37 - Hard Disk Drives
- 38 - RAID
- 39 - File and Directories
- 40 - File System Implementation
- 41 - Locality and the Fast File System
- 42 - Crash Consistency and Journaling
- 43 - Log-structured File Systems
- 44 - Flash-based SSD
- 45 - Data Integrity and Protection

Crash Consistency and Journaling

- How to update the disk despite crashes?
  - how ensure system always in *self-consistent* state, despite partial writes?
- Old systems
  - `fsck` - reads through entire disk, ensuring consistency
    - inodes point to allocated data
    - directories point to allocated, valid inodes
- Newer systems
  - *journaling* (also called *write-ahead logging*)

Example

- Tiny FS, one file (w/ one block) allocated:

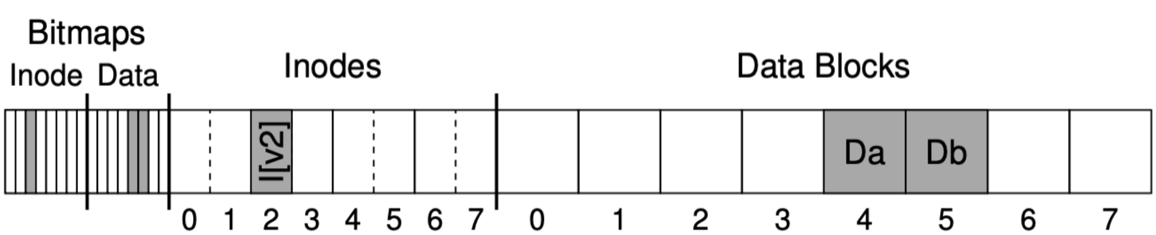


- Inode:

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


Example, cont.

- When we append by adding another block of data…
  - allocate and fill new data block
  - update inode to point to block, change size
  - change data bitmap




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

Note that all of these changes sit in the buffer cache for some unspecified time

Crash scenarios

- just the data block is written
  - not a problem
- just the updated inode (I[v2]) is written to disk
  - block has garbage
  - also, bitmap disagrees w/ inode
- just the updated bitmap is written to disk
  - no pointer to invalid data, but
  - space leak
- inode and bitmap written
  - block has garbage
- inode and data block written
  - all good, except bitmap doesn't know it
- bitmap and data block written
  - bitmap indicates block used, but no idea for what

FFS *Write Ordering*

- Writes
  - file data blocks asynchronous
  - metadata (inodes and directory contents) synchronous
- Implications
  - file `create` call expensive:
    - sync write file inode
    - sync write directory data
    - sync write directory inode
  - asynchronous writes:
    - file data
    - bitmaps can be reconstructed by `fsck`

fsck

- checks superblock, does FS match blocks allocated….
- free blocks: follows inode pointers, ensures all agree w/ bitmaps
- validate inode fields
- validate inode linkcounts (scan entire disk to find hard links)
- look for duplicate pointers to the same block
- look for ptrs outside partition boundaries, etc.
- directory checks : have “.”, “..”, each inode allocated

Very slow, getting worse.

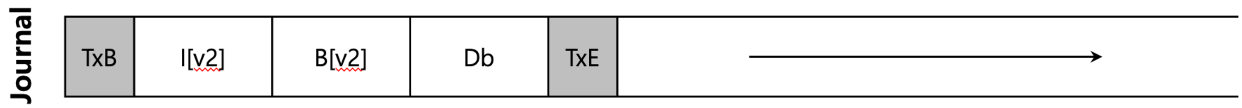
Journaling *write **transactions** to log before final locations*

- write-ahead logging in database world
  - all operations go also to an ordered log
  - write log *before* final locations on disk (bitmaps, inodes, data)
  - log is the ground truth
- ext3
  - on-disk structures mainly the same as ext2
  - but optionally has a journal…

Super	**Journal**	Group 0	Group 1	…	Group N

- Example : our canonical update again
  - We wish to update inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk
  - Before writing them to their final disk locations, we are now first going to write them to the log(a.k.a. journal)

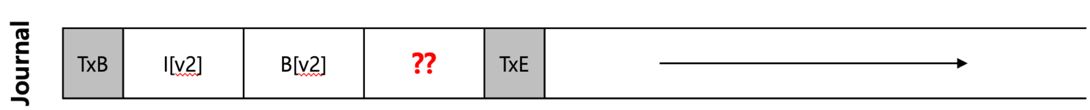
Journaling *transaction structure*



- TxB : transaction begin
  - contains a *transaction identifier* (TID)
- Middle blocks contain actual writes
  - this is *physical* logging, meaning actual writes are in log
  - *logical* logging means some high level representation of the change is used instead (like "+2")
- TxE: transaction end
  - also has TID

Journaling *How to write the transactions?*

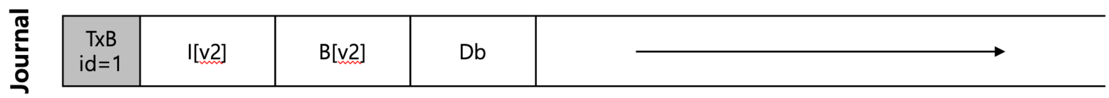
- Could write transactions one at a time
  - wait until one on disk before issuing next
  - this is slow
- Could write all operations at once
  - much faster
  - unsafe : disk might schedule in some other order
  - what if schedule is:
    - (1)TxB, I[v2], B[v2], and TxE and only later (2) write Db
    - and crash between (1) and (2)



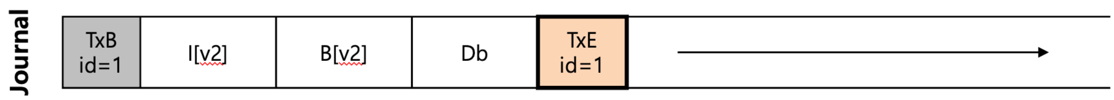
- Looks okay….

Journaling *better approach*

- Write transaction in two steps:
  - First write all blocks *except TxE* to journal



  - Second, write TxE:



  - TxE must be a single sector
    - disk guarantees all or nothing for a single sector
    - TxE must be sector size or less.
  - Crash *before* TxE means transaction has no effect
  - Crash *after* TxE allows transaction to be during *replayed* recovery

Journaling *entire sequence*

- Journal write
  - write all transaction entries except TxE, wait until on disk
- Journal commit
  - write TxE, wait until on disk
- Checkpoint
  - write all pending metadata and data updates to final locations in actual bitmaps, inodes, and data blocks

Journaling *batching*

"Xtion" == "transaction"

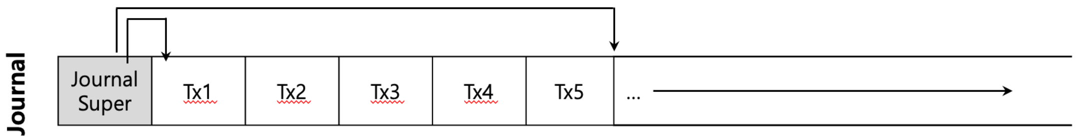
- If we create two files in the same directory
  - modify inode bitmap twice
  - modify data bitmap twice
  - modify directory data twice
  - possibly modify directory inode twice
  - two transactions, each with
    - Xtion write
    - Xtion commit
    - checkpoint
- We can instead *batch* using a single global Xtion
  - just mark all data structures that need to be updated
  - after some timeout, create a Xtion w/ all modified data

Journaling *recovery*

- If crash **before** transaction is written to log
  - pending update **dropped**
- During recovery
  - scan disks for all committed transactions
  - replay in order
- Issues:
  - Works, but recovery is slow.... (like `fsck`)
  - log eventually fills up, FS stops

Journaling *recovery*

- Create a journal *superblock*
    - mark first and last *uncheckpointed* Xtions



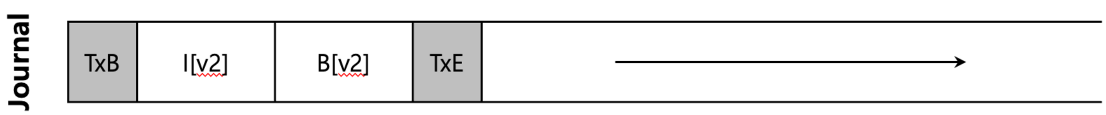
- So complete protocol is:
    - journal write
    - journal commit
    - checkpoint
    - free

    - periodically push free Xtions to journal superblock

Journaling *metadata*

- Still a problem : we are writing every block to disk twice
    - commit to journal
    - checkpoint to on-disk location
    - we've halved our disk bandwidth!
        - (data blocks are majority of journal)

- Metadata journaling
    - data blocks *not* written to journal
    - journal would look like:



- *Instead*: block Db written to final location

Journaling *metadata*

- *When* should we write the data blocks to disk?
  - Write data to disk after transaction
    - file system consistent, but I[v2] might point to garbage
  - Write data to final locations *first*

"write the pointed-to object before the pointer"

- Protocol now:
  - *data write*
  - journal metadata write
  - *wait for completion of first two steps*
  - journal commit
  - checkpoint metadata
  - free at some later time

Journaling *tricksy: block reuse*

- Some metadata really should not be replayed:

1. Directory "foo" is updated

TxB id=1	I[foo] ptr:1000	D[foo] [final addr:1000]	TxE id=1

2. Directory "foo" id deleted. block 1000 freed.

3. User creates file "foobar" using block 1000 for data

TxB id=1	I[foo] ptr:1000	D[foo] [final addr:1000]	TxE id=1	TxB id=2	I[foobar] ptr:1000	TxE id=2

Journaling *tricksy: block reuse*

- Assume crash occurs and all this is in the log:

TxB id=1	I[foo] ptr:1000	D[foo] [final addr:1000]	TxE id=1	TxB id=2	I[foobar] ptr:1000	TxE id=2	→

- During replay, recover process replays everything in the log
  - including the write of directory data to block 1000
  - thereby overwriting the user data from file foobar
- ext3 creates a *revoke record* when the directory is deleted
  - Recovery first scans for revoke records
  - Revoked data not written during recovery

Data Journaling Timeline

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

Metadata Journaling Timeline

Journal TxB	Journal Contents (metadata)	Journal TxE	File System Metadata	File System Data
issue	**issue**			**issue**
				complete
complete				
	complete			
		issue		
		complete		
			issue	
			complete	

t