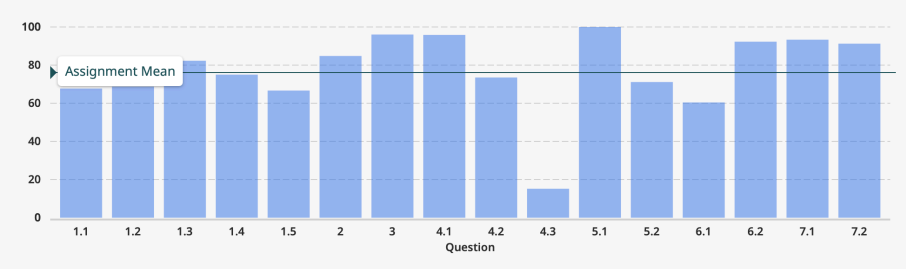
Persistence

- 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

Test #2



exam2 100.0 points

Minimum	Median	Maximum	Mean	Std Dev
47.0%	75.0%	99.0%	75.77%	10.7%

Nope…

```
void increment(int *iptr) {
    int old;
    do {
        old = *iptr;
    } while (TestAndSet(iptr, old + 1) != old);
}
```

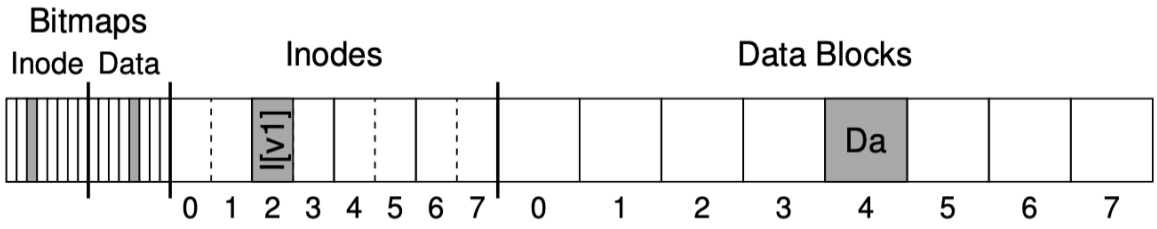
Crash Consistency and Journaling

- How to update the disk despite crashes?
  - how ensure *self-consistent* state, despite partial writes?
    - remember:
      - only individual sectors are atomically written
      - order sectors written ≠ order stable on disk

- 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:

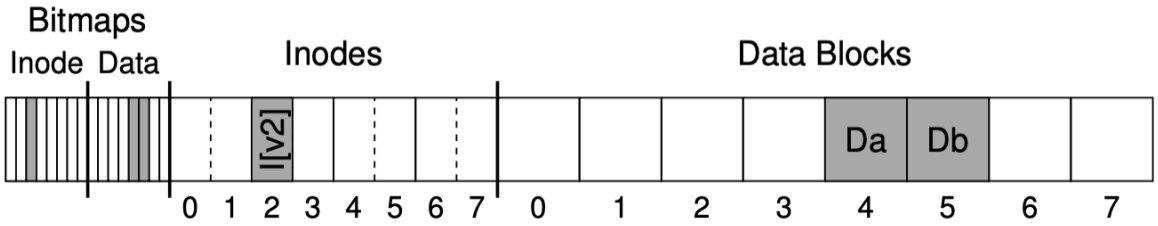


- Inode:

```
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        : 2
pointer     : 4
pointer     : 5
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 (*bad)*
  - also, bitmap disagrees w/ inode (*maybe bad)*
- just the updated bitmap is written to disk
  - no pointer to invalid data, but
  - space leak (sorta *bad)*
- inode and bitmap written
  - block has garbage (*bad)*
- inode and data block written
  - all good, except bitmap doesn't know it (sorta *bad)*
- bitmap and data block written
  - bitmap indicates block used, but no idea for what (*sorta bad)*

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` *after crashing*

- 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 multiple different inodes pointing to the same block
- look for ptrs outside partition boundaries, etc.
- directory checks : have “.”, “..”, each inode allocated, etc.

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…

- 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?*

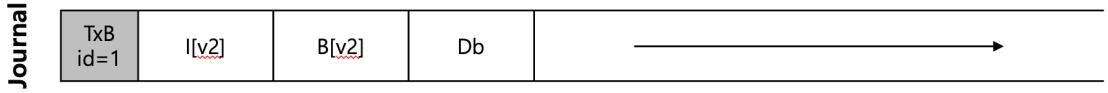
- Could write transactions one at a time
  - wait until one on disk before issuing next
  - this is slow

- 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)

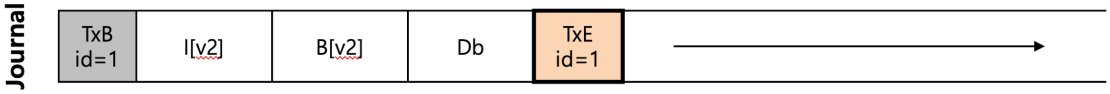
- Looks okay….

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