CS3210: Crash consistency

Taesoo Kim

Administrivia

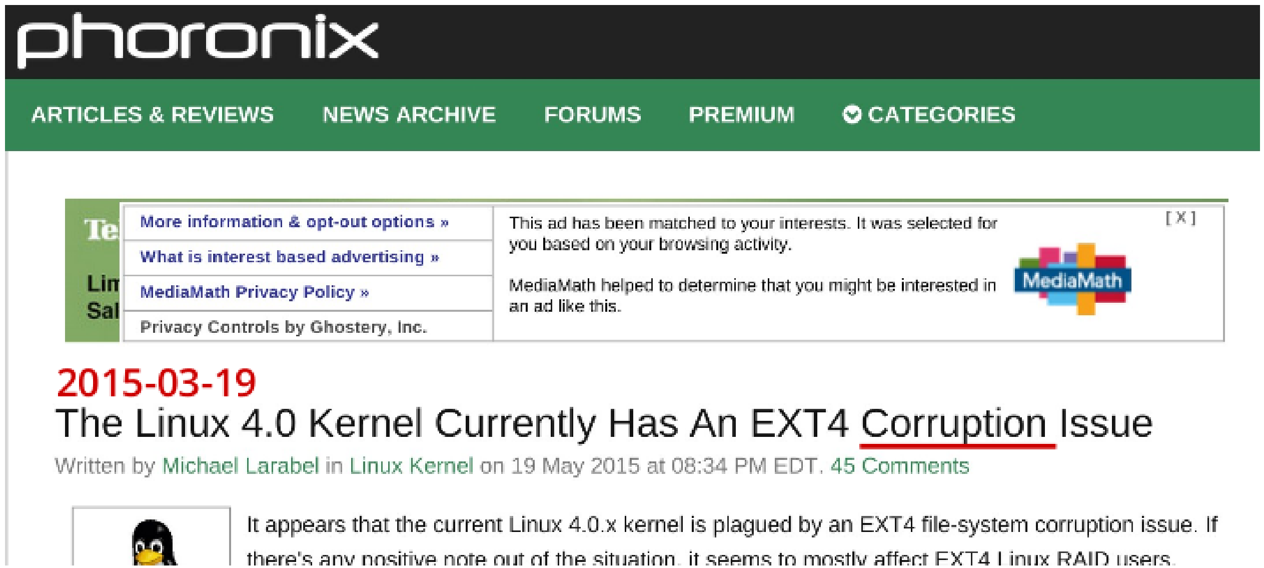
- Quiz #2. Lab3-5, Ch 3-6 (read "xv6 book")
  - Open laptop/book, no Internet

Summary of cs3210

- Power-on → BIOS → bootloader → kernel → user programs
- **OS:** abstraction, multiplexing, isolation, sharing
- **Design:** monolithic (xv6) vs. micro kernels (jos)
- **Abstraction:** process, system calls, files, IPC, networking (lab6)
- **Isolation mechanisms:** CPL, segmentation, paging

Why crash recovery (power failure)?

Why crash recovery (bugs)?



Q: what happens after a FS crash?

- Q: Is it possible that `AAAA` doesn't exist? (yes/no?) Then, `BBBB`?
- Q: Is it possible that `BBBB` contains junks? (yes/no?)
- Q: Is it possible that `BBBB` is empty? (yes/no?)
- Q: Is it possible that `BBBB` contains "hello"? (yes/no?)
- Q: Is it possible that `BB` exists in the current directory? (yes/no?)

```
$ cat AAAA
hello world!
$ cp AAAA BBBB
[panic] ...
[reboot]
```

Why crash recovery?

- Q: Then, is your file system still usable?
- Main problem:
    - crash during multi-step operation
    - leaves FS invariants violated (Q: examples?)
    - can lead to ugly FS corruption

NOTE worse yet, media corruption (very frequent!) is out-of-scope

Example: inconsistent file systems

- Breakdowns of `create()`:
  - create new dirent
  - allocate file inode
- Crash: dirent points to free inode -- disaster!
- Crash: inode not free but not used -- not so bad

Today's Lecture

- Problem: crash recovery
  - crash leads to inconsistent on-disk file system
  - on-disk data structure has "dangling" pointers
- Solutions:
  - synchronous write
  - delayed writes (e.g., write-back cache, soft updates)
  - logging

What can we hope for? (after recovery)

1. FS internal invariants maintained
    - e.g., no block is both in free list and in a file
2. All but last few operations preserved on disk
    - e.g., data written yesterday are preserved
3. No order anomalies
    - `echo 99 > result ; echo done > status`

Simplifying assumption: disk is "fail-stop"

- Disk executes the writes FS sends it, and does nothing else Perhaps doesn't perform the very last write
  - no wild writes
  - no decay of sectors

Correctness vs. performance

- Safety → write to disk ASAP
- Speed → don't write the disk (e.g., batch, write-back cache)
- Two approaches:
  - synchronous meta-data update + fsck (linux ext2)
  - logging (xv6 and linux ext3/4)

meta-data: other than actual file contents (i.e., data block)

Synchronous-write solution

- Synchronous meta-data update:
  - an old approach to crash recovery
  - simple, slow, incomplete
- Most problem cases look like dangling references
  - inode → free block
  - dirent → free inode

Idea: always initialize *on disk* before creating any reference

- "synchronous writes" is implemented by
  1. doing the initialization write
  2. waiting for it to complete
  3. and then doing the referencing write

Example: file creation

- Q: what's the right order of synchronous writes (dirent → free inode)?

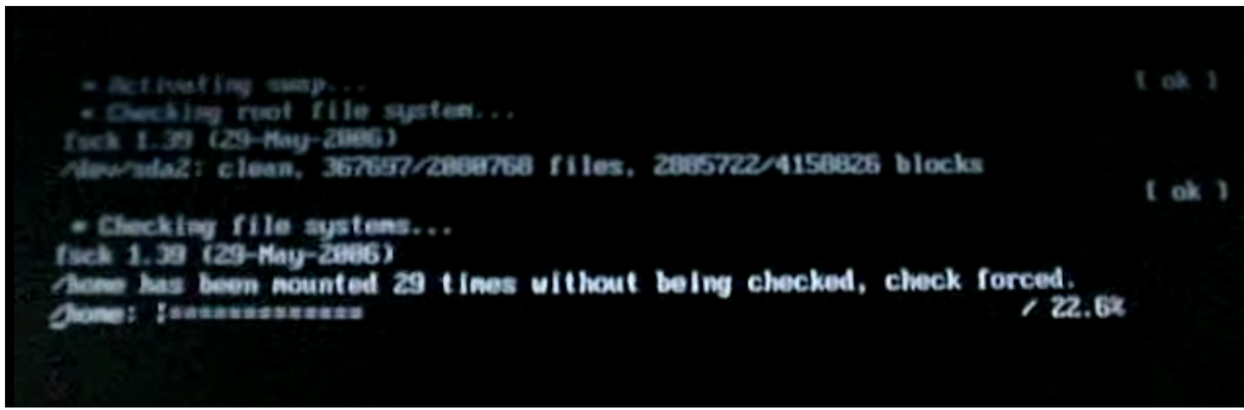
Example: file creation

- Q: what's the right order of synchronous writes (dirent → free inode)?
  1. mark inode as allocated
  2. create directory entry

What will be true after crash+reboot?

- `create()`:
    1. mark inode as allocated ← Q: what if failed after `ialloc()`?
    2. create directory entry

Idea: fix FS when mounting (if crashed)



- To free unreferenced inodes and blocks (orphan)
- To clean-up an interrupted `rename()`

Problems with sync. meta-data update

- Very slow during normal operation (Q: why?)
- Very slow during recovery (Q: why? e.g., 100 MB/sec on 2TB HDD)

How to get better performance?

- Use RAM (e.g., write-back cache)
- Exploit disk sequential throughput (100 MB/sec)
- Keep track of dependencies among buffer caches
  - Q: cycle dependencies?
  - Q: still need slow fsck?

Storage performance

- Q: HDD vs. SSD? faster? bandwidth?
- Q: which one is faster? read vs. write?
- Q: in sequential vs. random?

(ref. http://www.pcgamer.com/hard-drive-vs-ssd-performance/2/)

Chart1: Sequential read

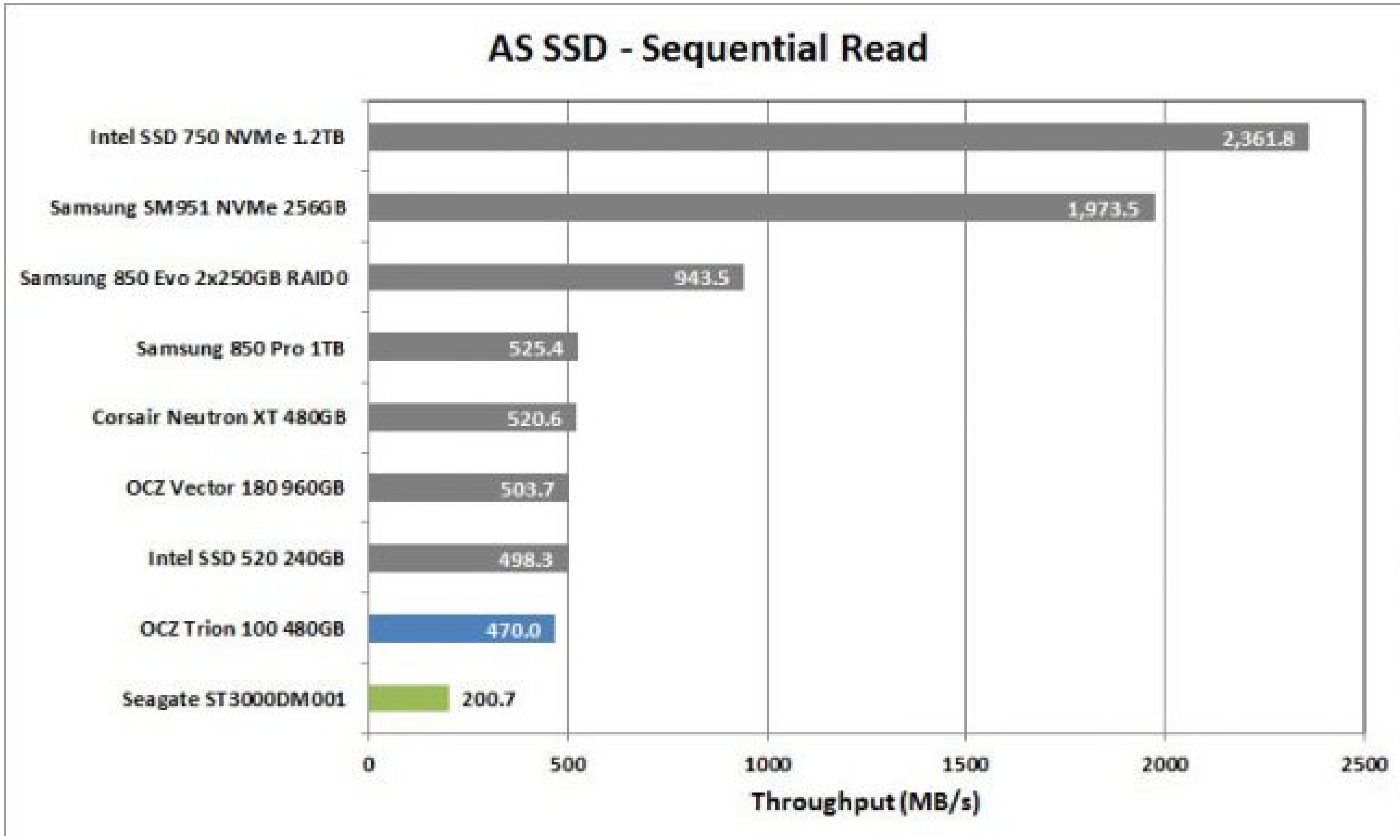


Chart2: Sequential write

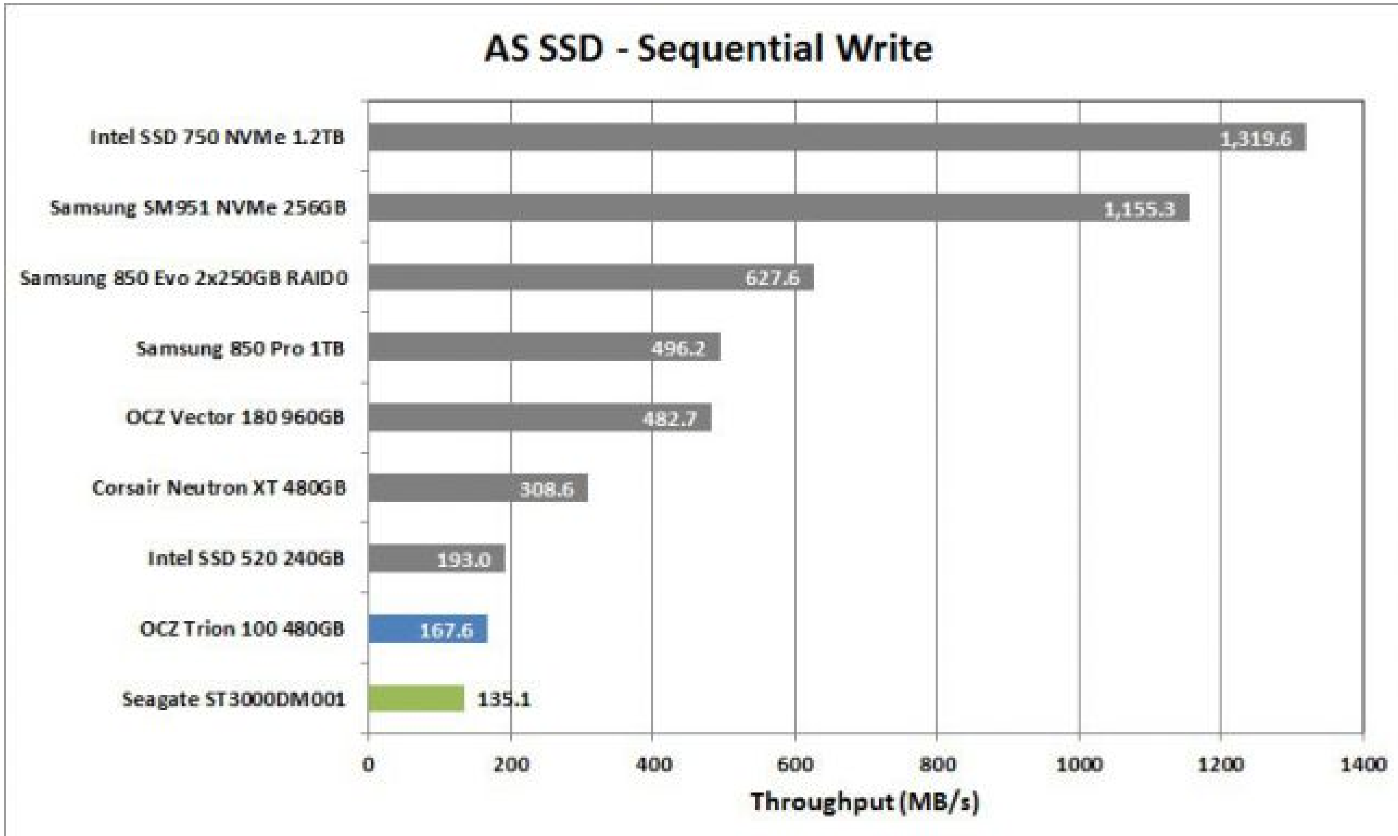


Chart3: Random read

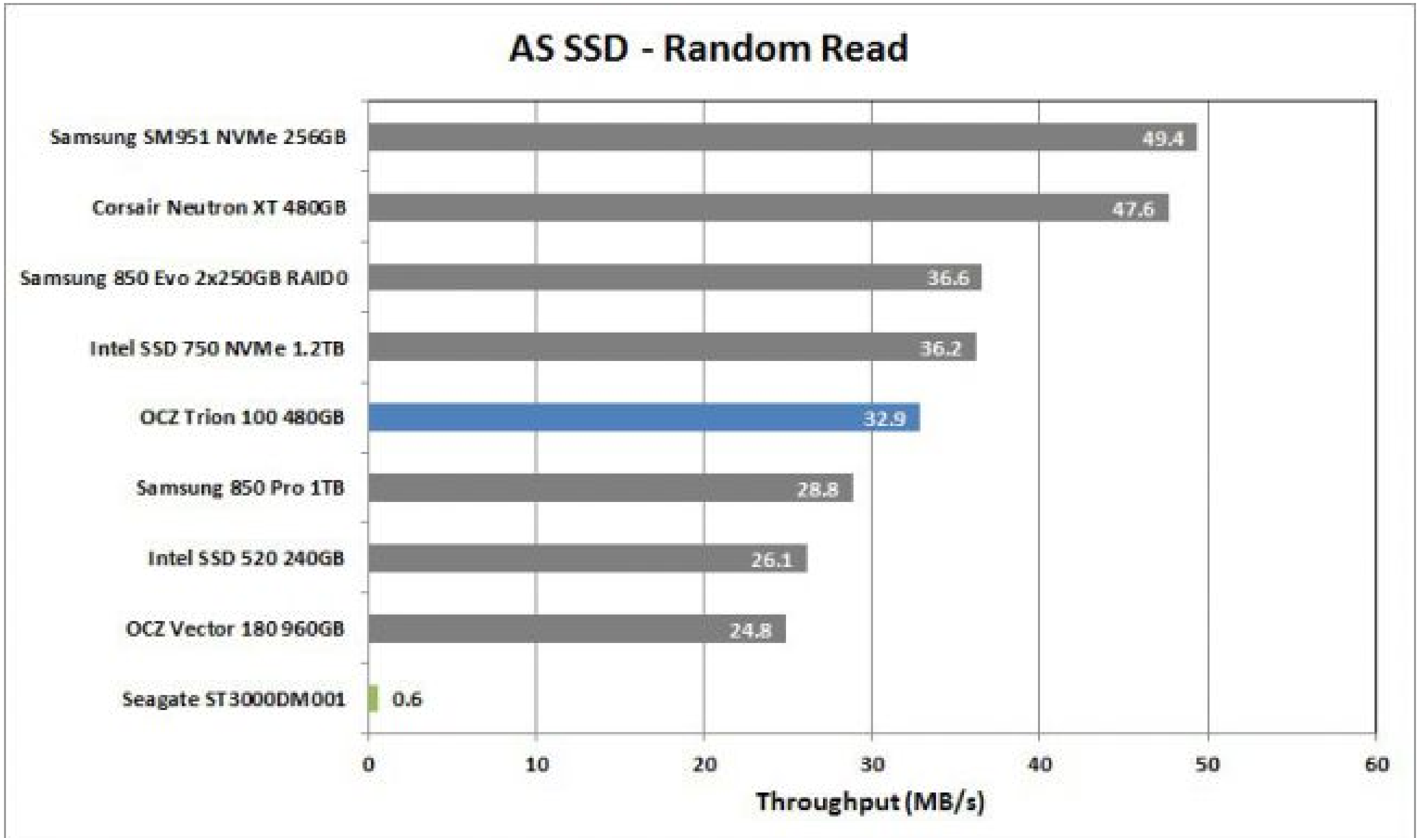
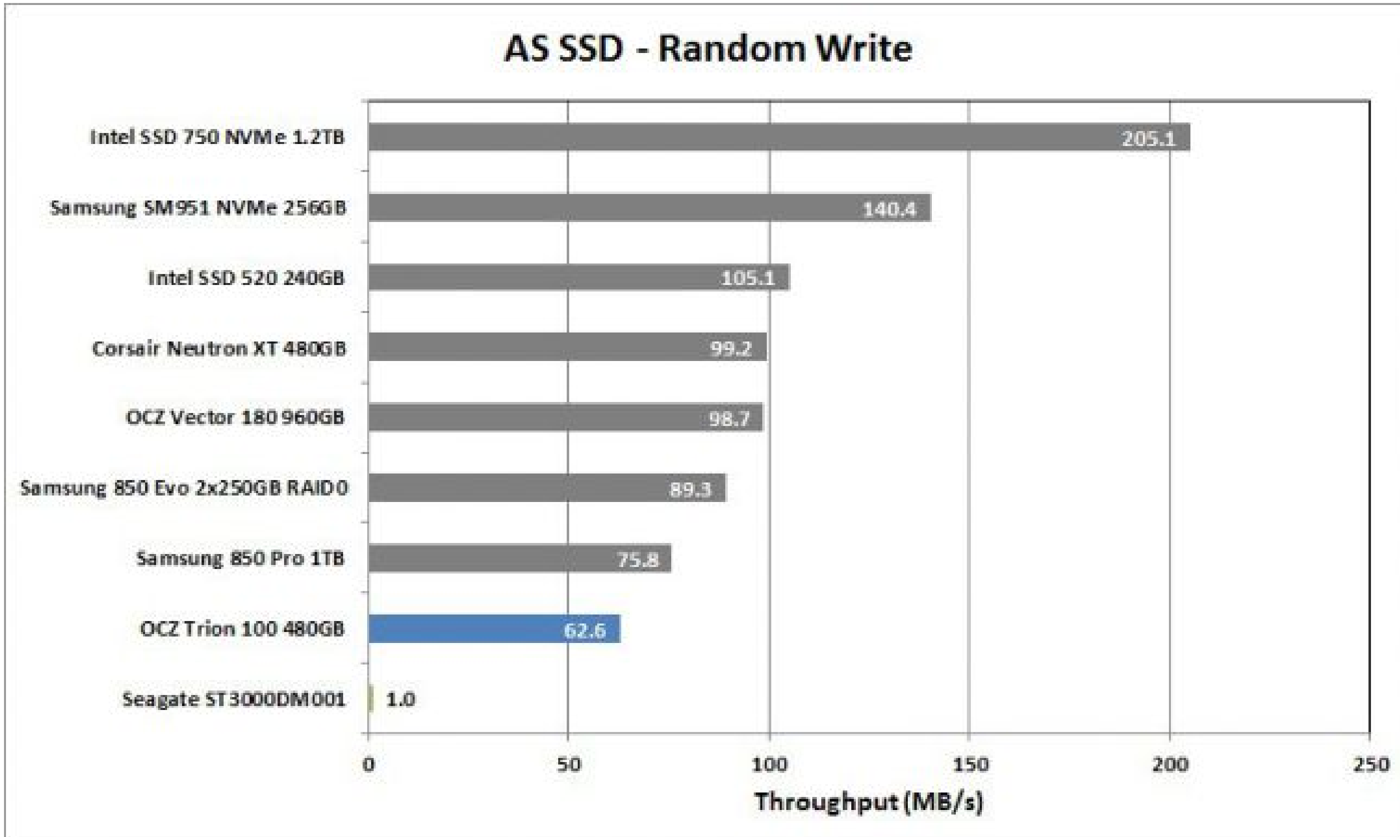


Chart4: Random write



Better idea: "logging"

- How can we get both speed and safety?
  - write only to cache
  - somehow remember relationships among writes
    - e.g., don't send #1 to disk w/o #2 and #3

Goals of logging

1. Atomic system calls w.r.t. crashes
2. Fast recovery (no hour-long `fsck`)
3. Speed of write-back cache for normal operations

Basic approach: "write-ahead" logging

- Atomicity: transaction either fails or succeeds
  1. record all writes to the log
  2. record "done"
  3. do the real writes
  4. clear "done"
- On crash+recovery:
  - if "done" in log, replay all writes in log
  - if no "done", ignore log

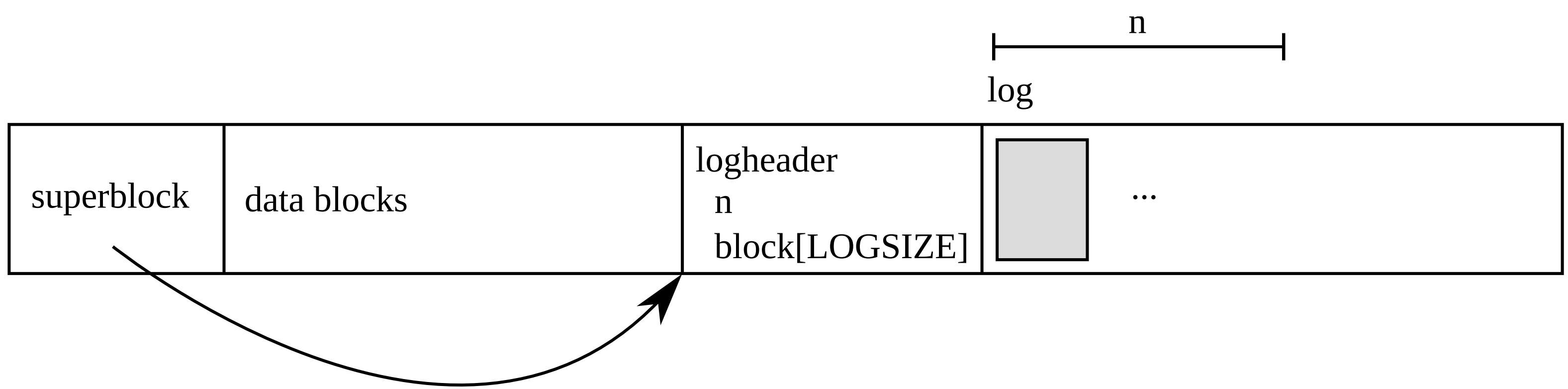
xv6's simple logging

```
+ beg_op();
     bp = bread(dev, bn);
     // modify bp->data[]
-    bwrite(buf);
+    log_write(bp);
     brelse(bp);
+ end_op();
```

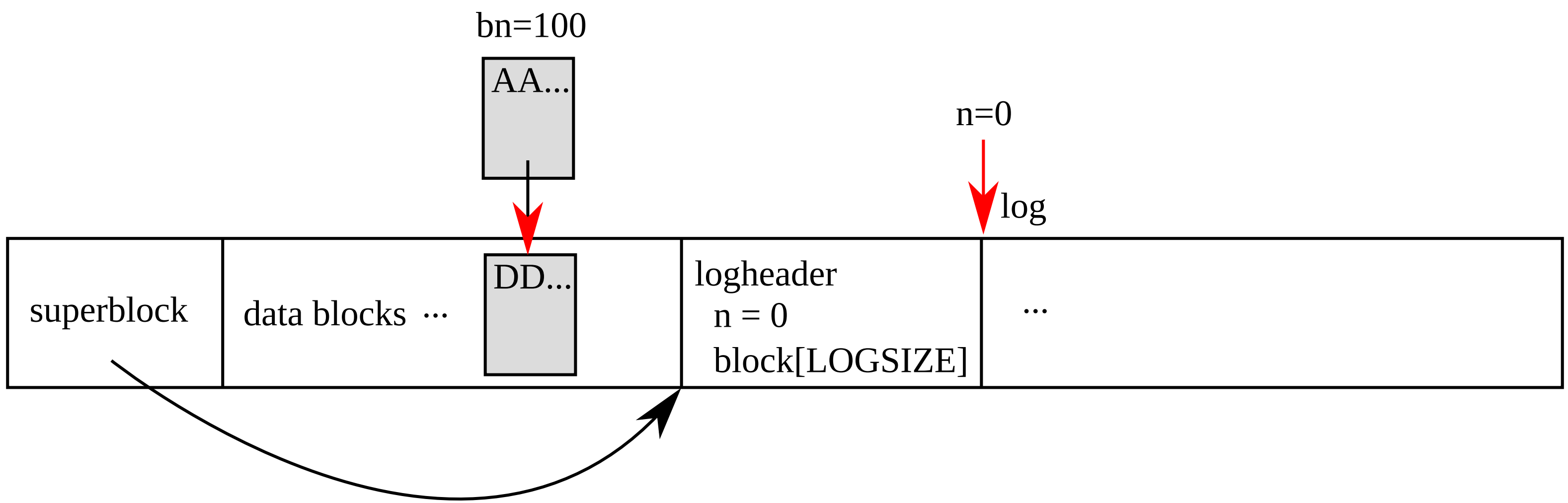
What is good about this design?

- Correctness due to write-ahead log
- Good disk throughput (Q: why? why not?)
- Faster recovery without slow `fsck`
- Q: What about concurrency?
  - xv6: no concurrency to make our life easier

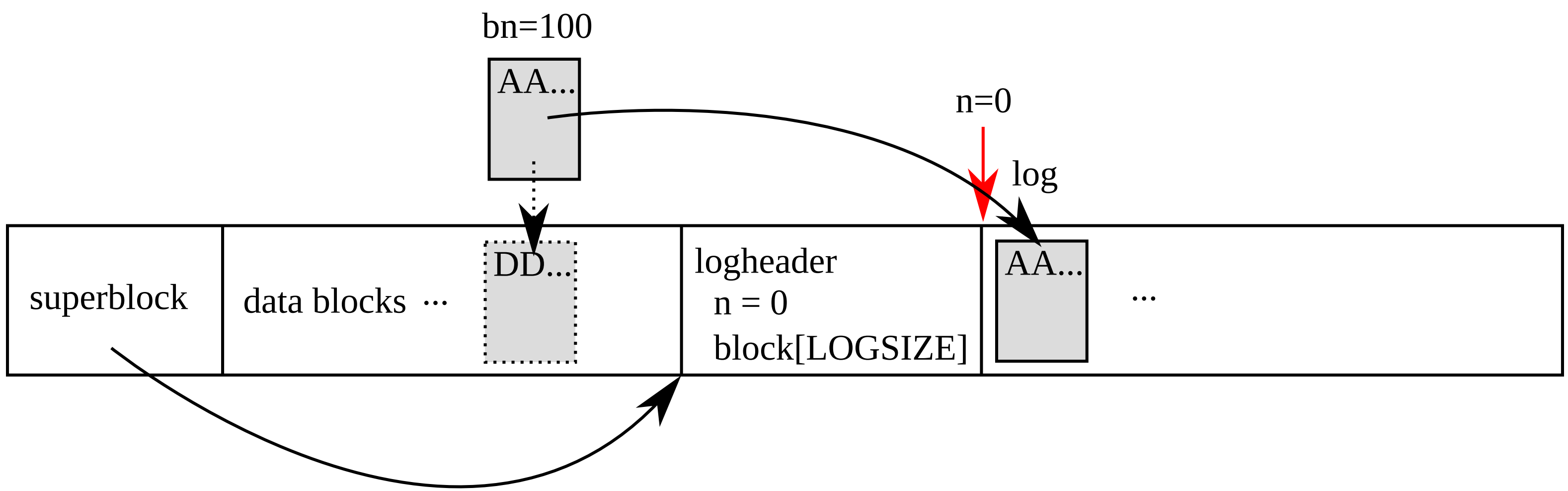
Disk structure for logging



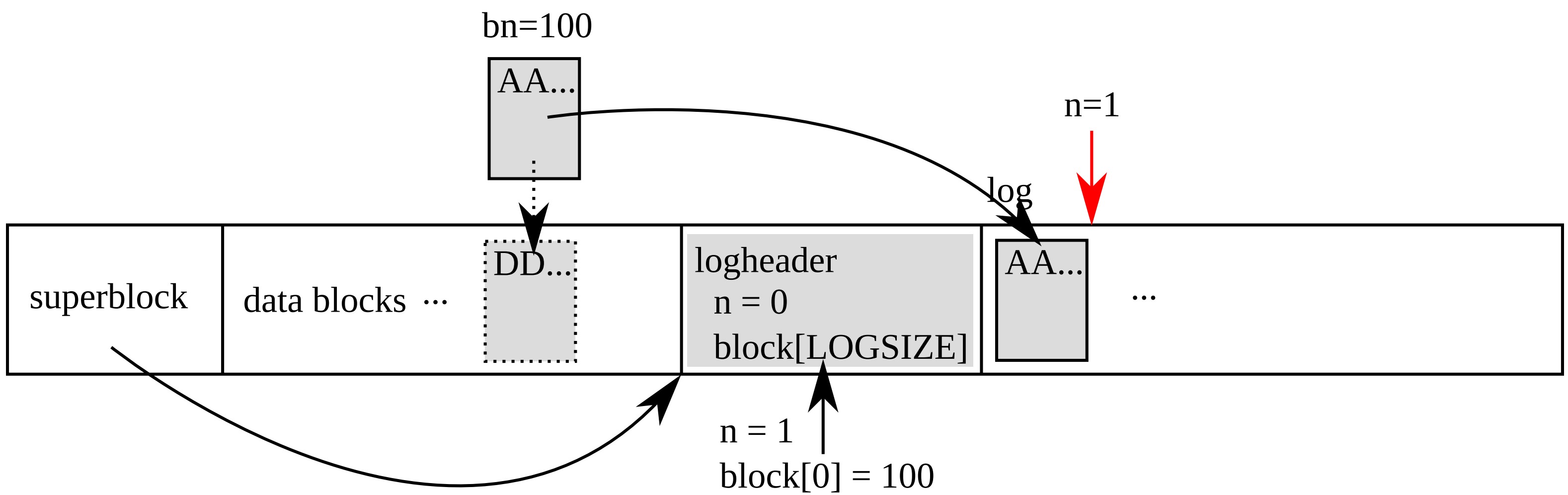
Example: writing a block (bn = 100)



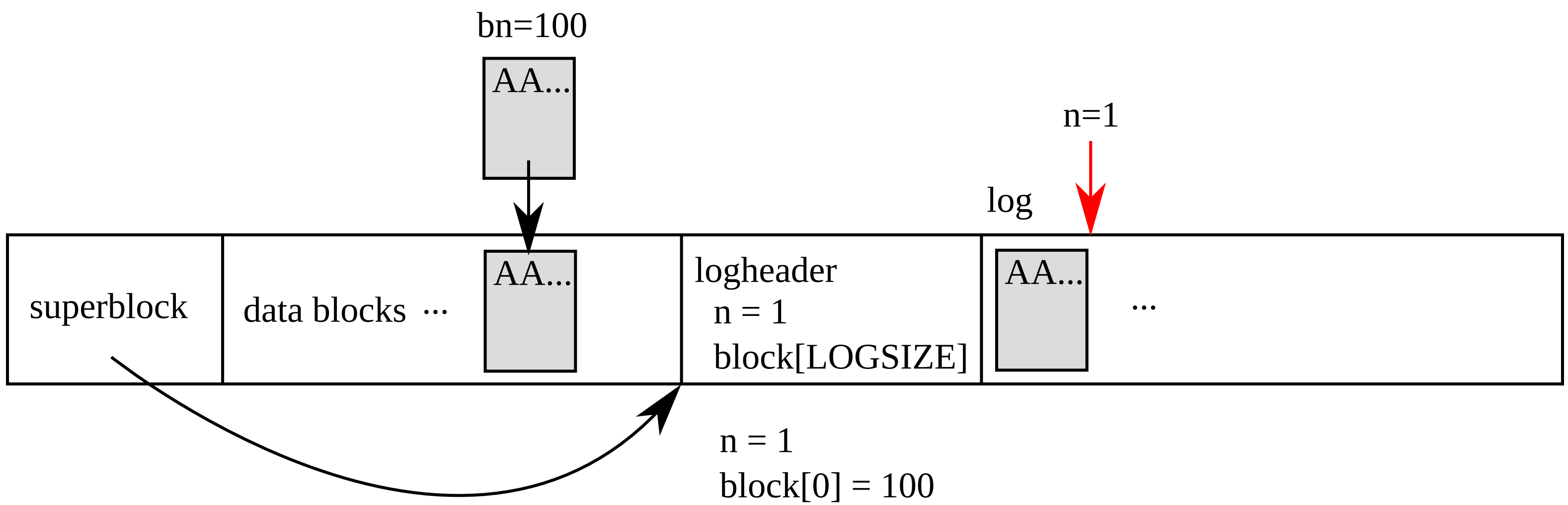
Step1: writing to a log



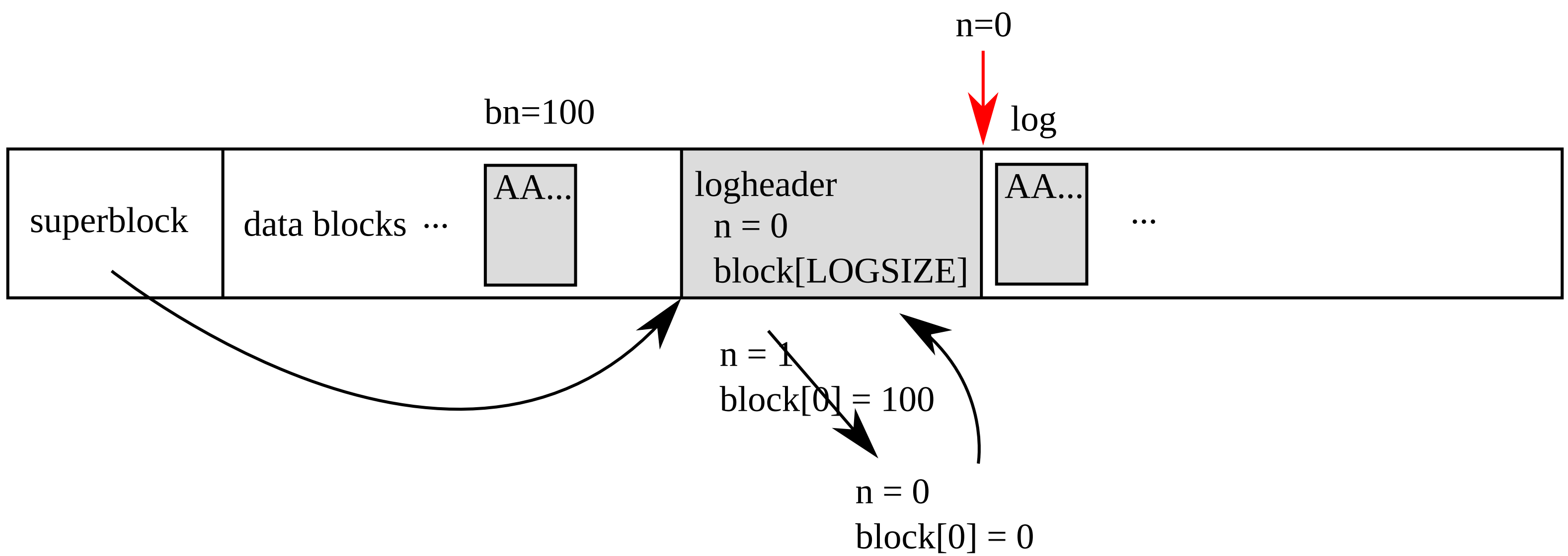
Step2: flushing the logheader (committing)



Step3: overwriting the data block



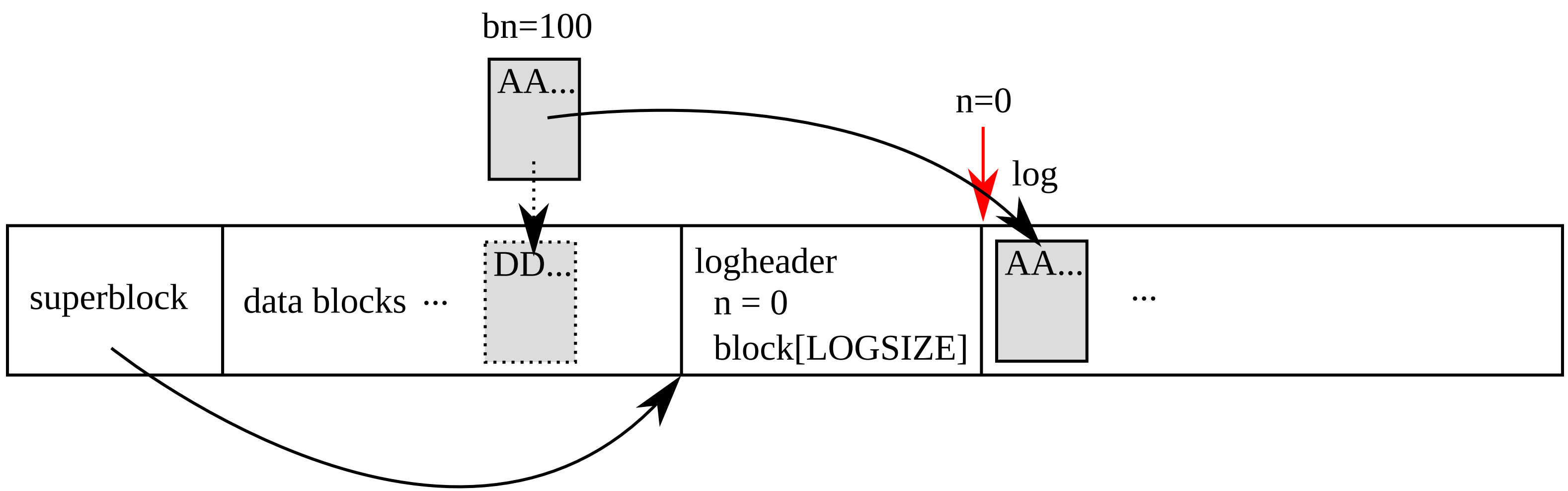
Step4: cleaning up the logheader



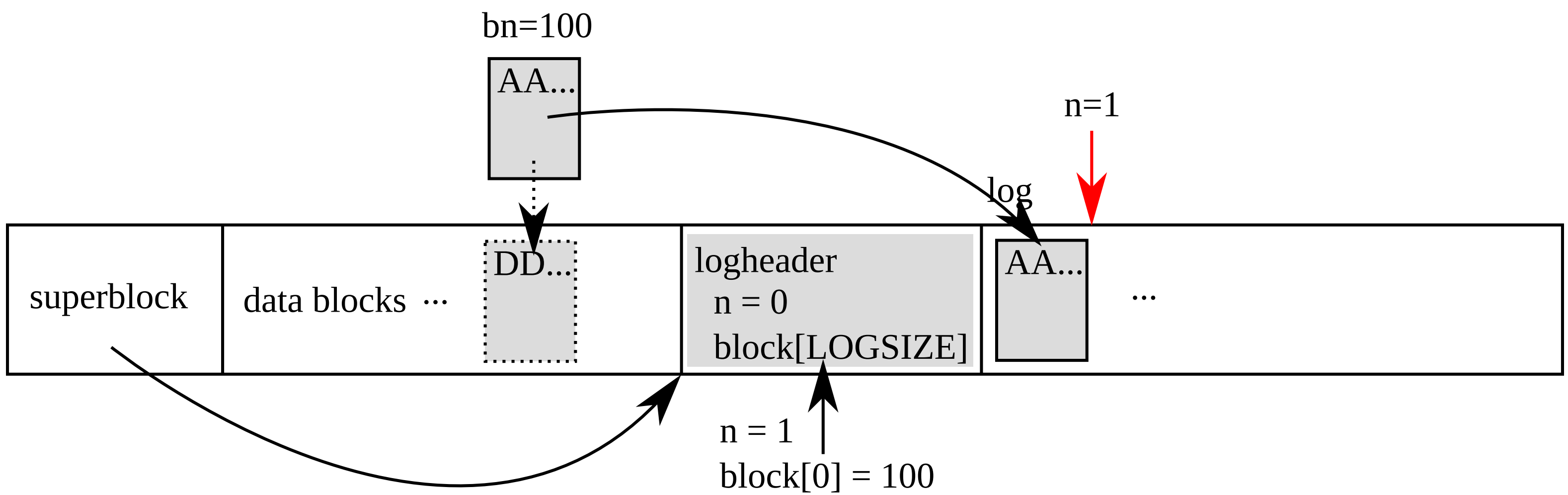
What if failed (say power-off and reboot)?

- Does FS contain "AA.." (❶) or "BB.." (❷)?
  - Q: Step1: writing to a log (❶/❷?)
  - Q: Step2: flushing the logheader (❶/❷?)
  - Q: Step3: overwriting the data block (❶/❷?)
  - Q: Step4: cleaning up the logheader (❶/❷?)

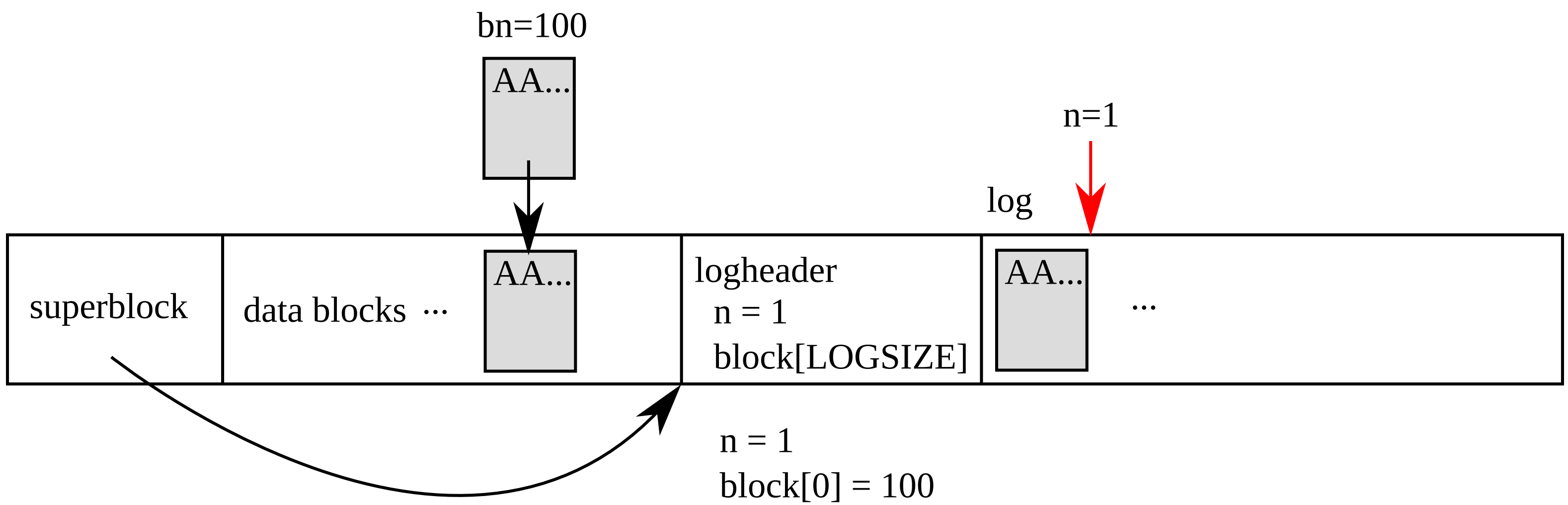
Q? Step1: writing to a log



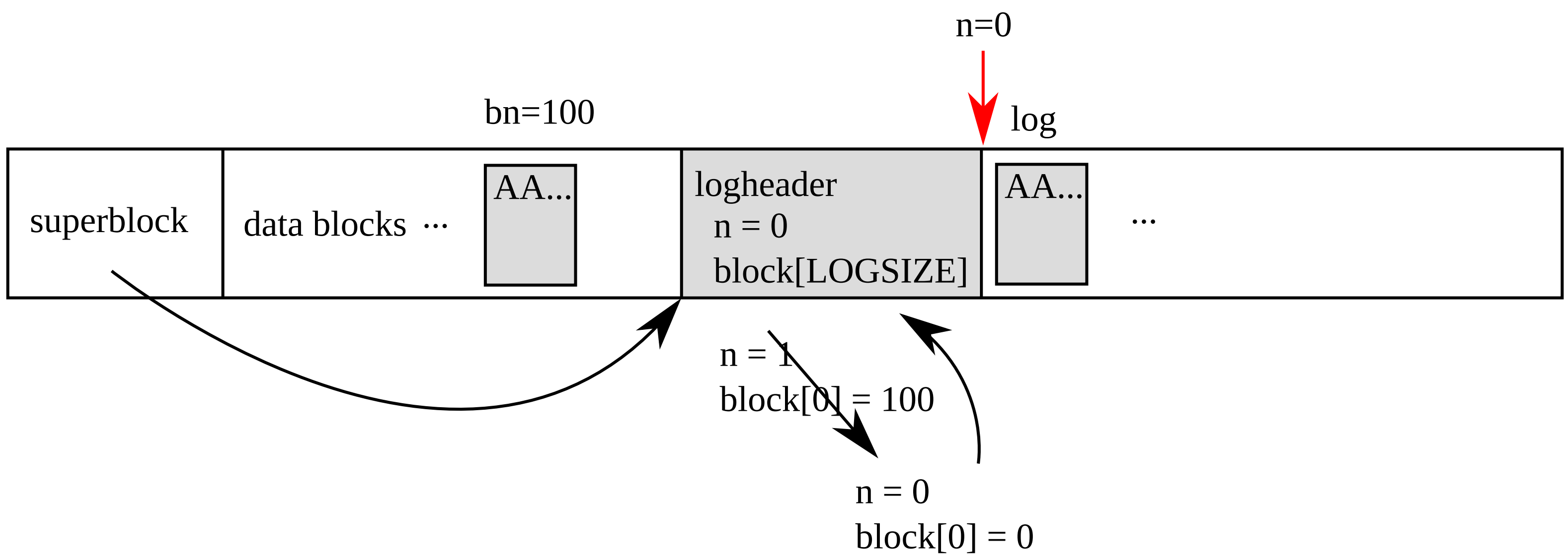
Q? Step2: flushing the logheader



Q? Step3: overwriting the data block



Q? Step4: cleaning up the logheader



DEMO: dumplog.c

```
static void commit() {
  if (log.lh.n > 0) {
    write_log();     // Write modified blocks from cache to log
    // Q1: panic("after writing to log!");
    write_head();    // Write header to disk -- the real commit
    // Q2: panic("after writing the loghead!");
    install_trans(); // Now install writes to home locations
    // Q3: panic("after the transaction!");
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
    // Q4: panic("after cleaning the loghead!");
  }
}
```

A few complications

- How to write larger data that doesn't fit to the log region?
- How to handle concurrency?
- How to avoid 2x writing (redundant)?
- How to log partial data (changes on a few bits)?

References

- Intel Manual
- UW CSE 451
- OSPP
- MIT 6.828
- Wikipedia
- The Internet