

CS3210: Crash consistency

Taeso Kim

Administrivia

- Quiz #2. Lab4-5, Ch 3-6 (read "xv6 book")
 - Open laptop/book, no Internet
 - 3:05pm ~ 4:25-30pm (sharp)
- NOTE Lab6: 10% bonus, a single lab (bump up your grade!)
- NOTE +5pt in quiz2, if you submit an anonymous "token" shown at the end of the course evaluation

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)?

phoronix


ARTICLES & REVIEWS
NEWS ARCHIVE
FORUMS
PREMIUM
♥ CATEGORIES

<p>Te</p> <p>Lin</p> <p>Sal</p>	<p>More information & opt-out options »</p> <p>What is interest based advertising »</p> <p>MediaMath Privacy Policy »</p> <p>Privacy Controls by Ghostery, Inc.</p>	<p>This ad has been matched to your interests. It was selected for you based on your browsing activity. [X]</p> <p>MediaMath helped to determine that you might be interested in an ad like this.</p> 
---------------------------------	---	---

2015-03-19

The Linux 4.0 Kernel Currently Has An EXT4 Corruption Issue

Written by [Michael Larabel](#) in [Linux Kernel](#) on 19 May 2015 at 08:34 PM EDT. [45 Comments](#)



It appears that the current Linux 4.0.x kernel is plagued by an EXT4 file-system corruption issue. If there's any positive note out of the situation, it seems to mostly affect EXT4 Linux RAID users.

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)

```
* Activating swap...                               [ ok ]
* Checking root file system...
fsck 1.39 (29-May-2006)
/dev/sda2: clean, 367657/2808768 files, 2885722/4158826 blocks
                                                    [ ok ]

* Checking file systems...
fsck 1.39 (29-May-2006)
/dev/sda1: /home has been mounted 29 times without being checked, check forced.
Done: |#####          / 22.6%
```

- 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

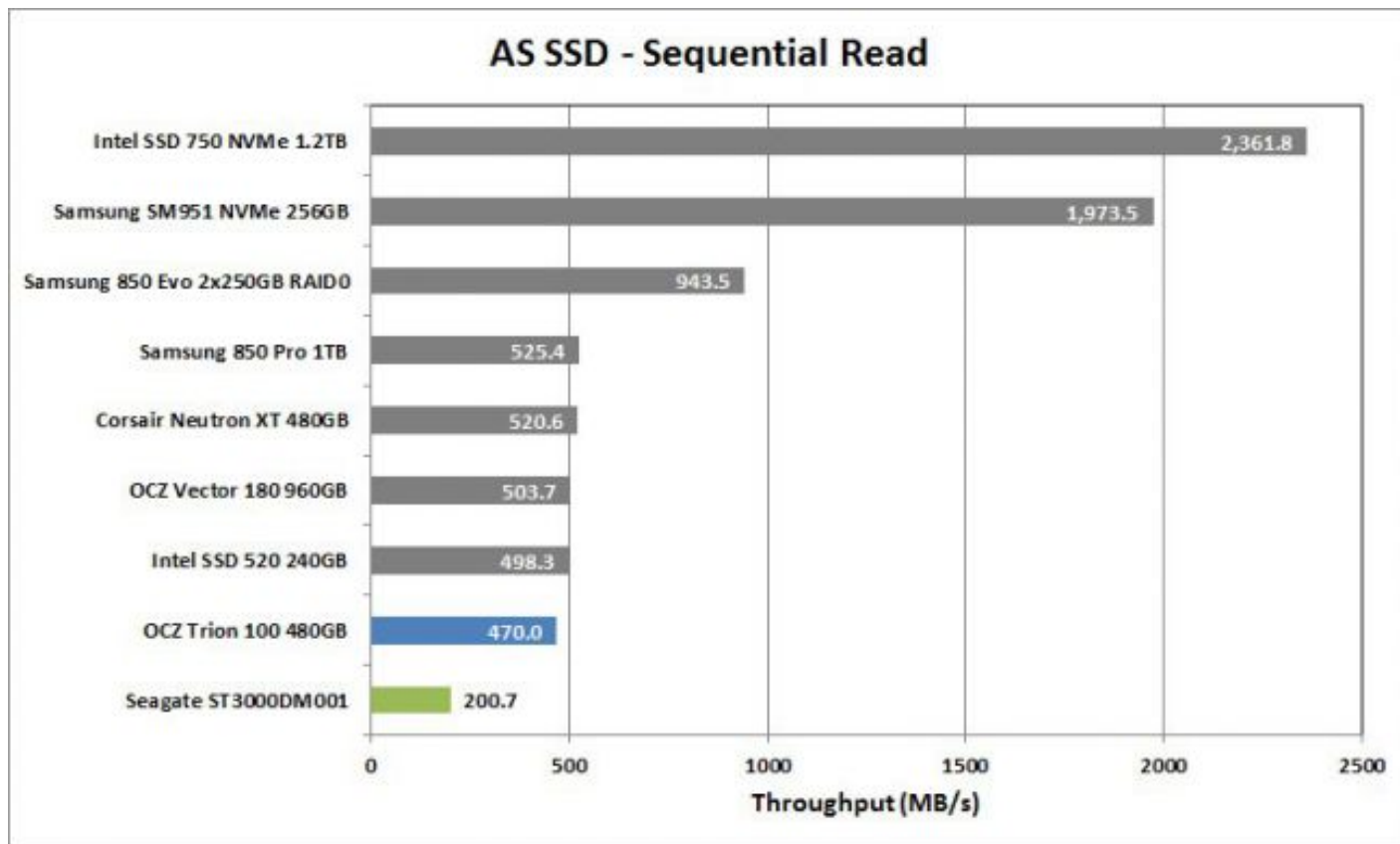


Chart2: Sequential write

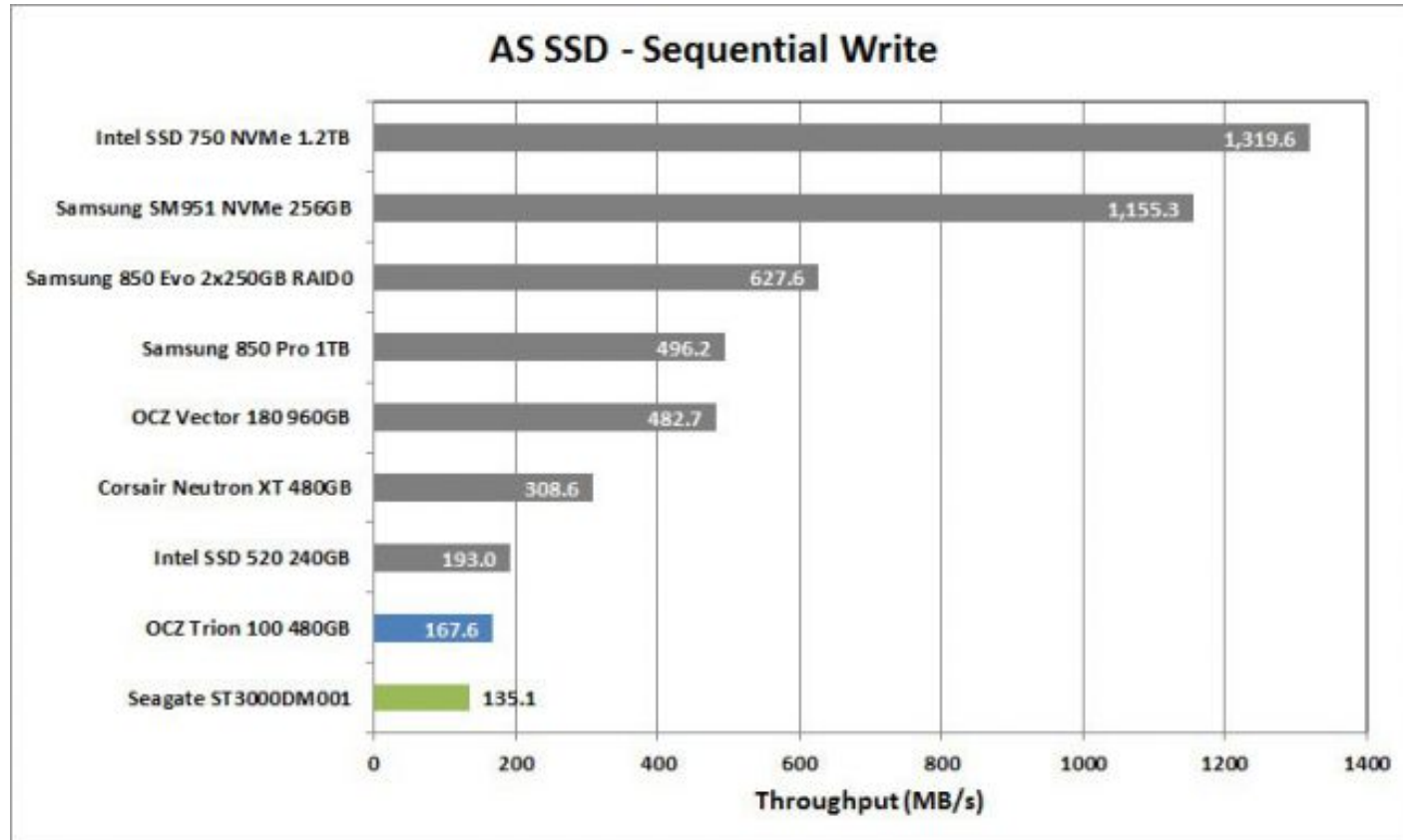


Chart3: Random read

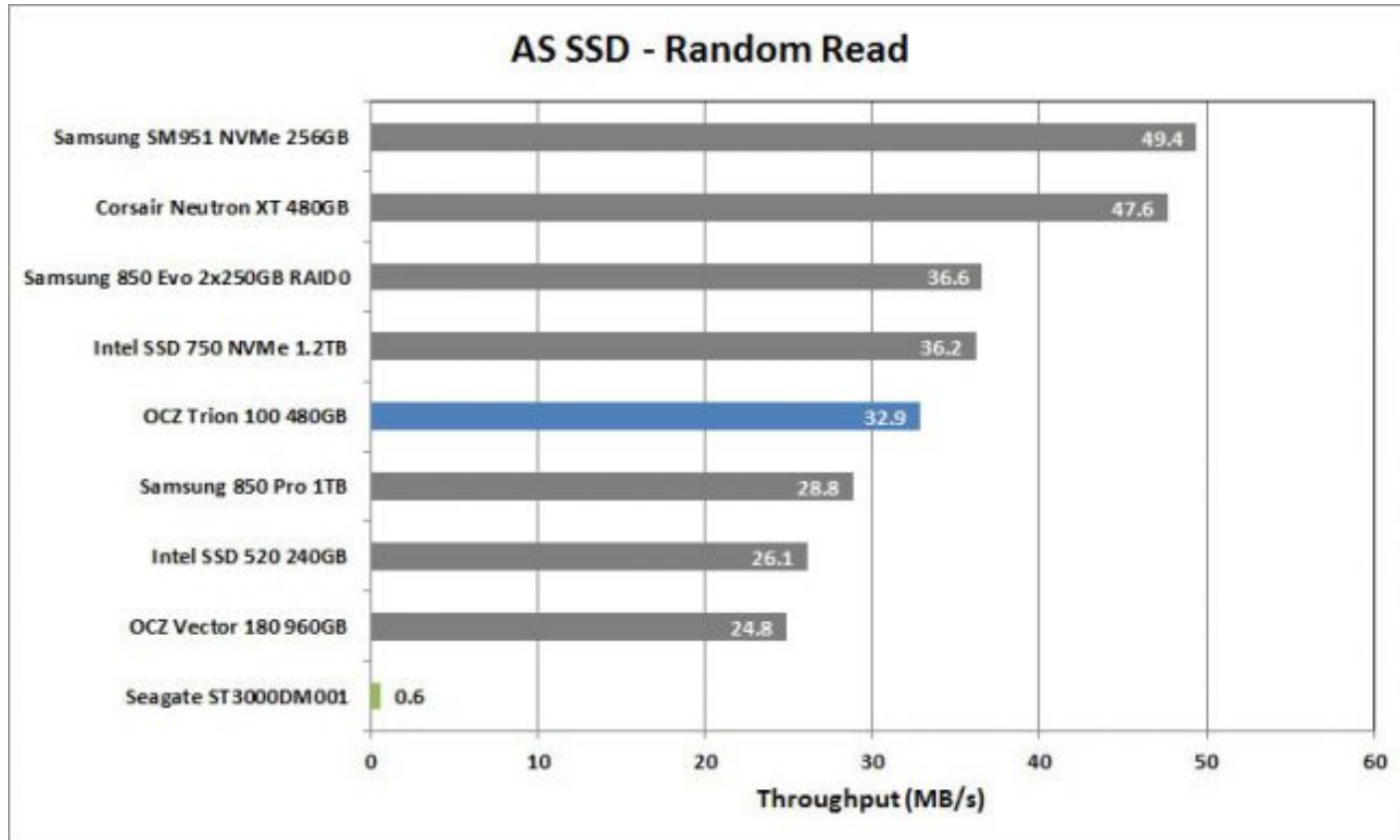
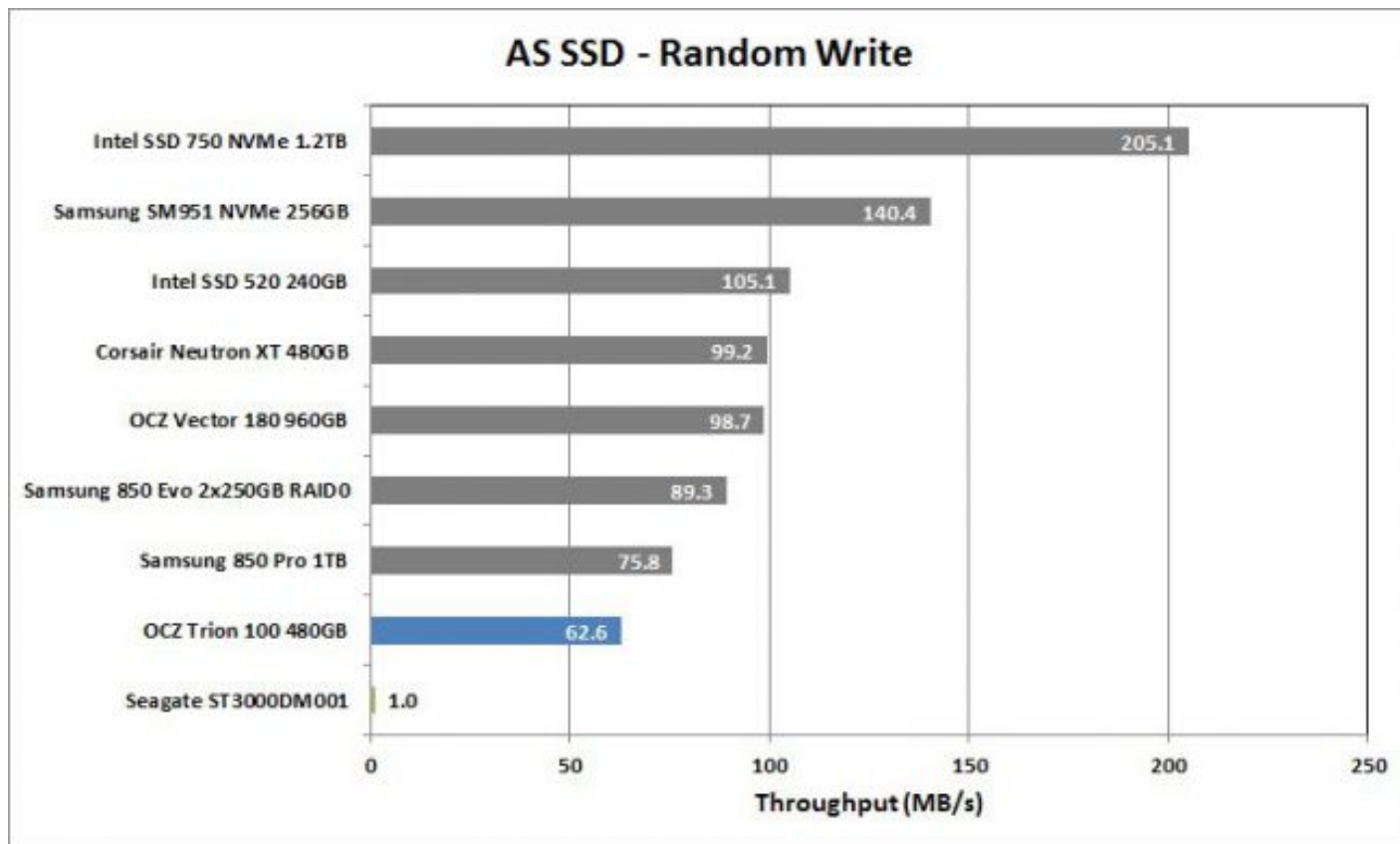


Chart4: Random write



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

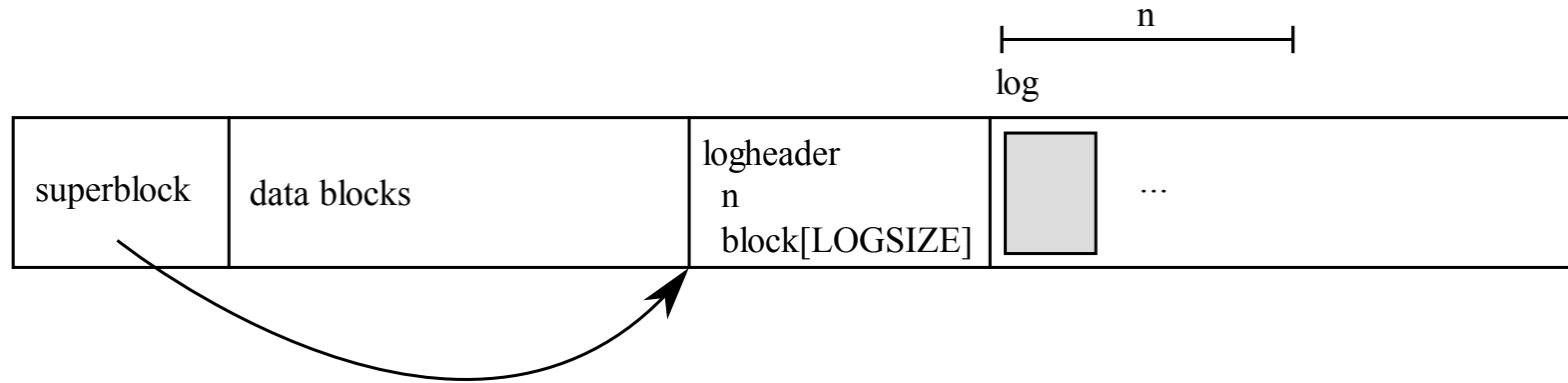
xv6's simple logging

```
01     + beg_op();
02         bp = bread(dev, bn);
03         // modify bp->data[]
04     - bwrite(buf);
05     + log_write(bp);
06         brelse(bp);
07     + end_op();
```

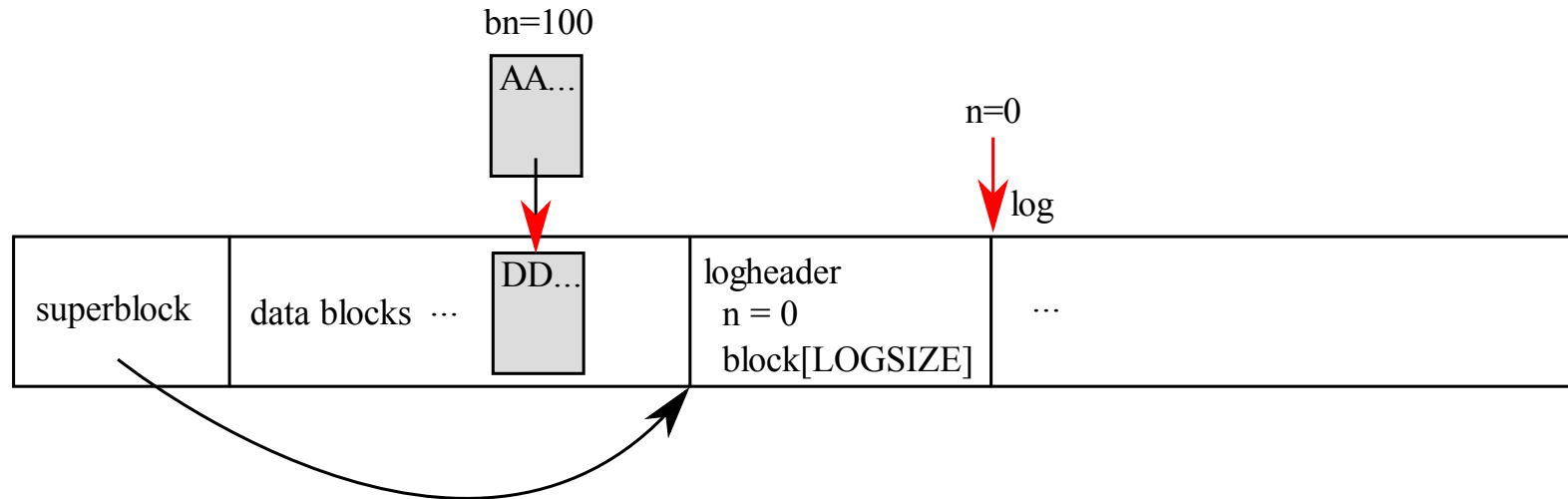
What is good about this design?

- 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

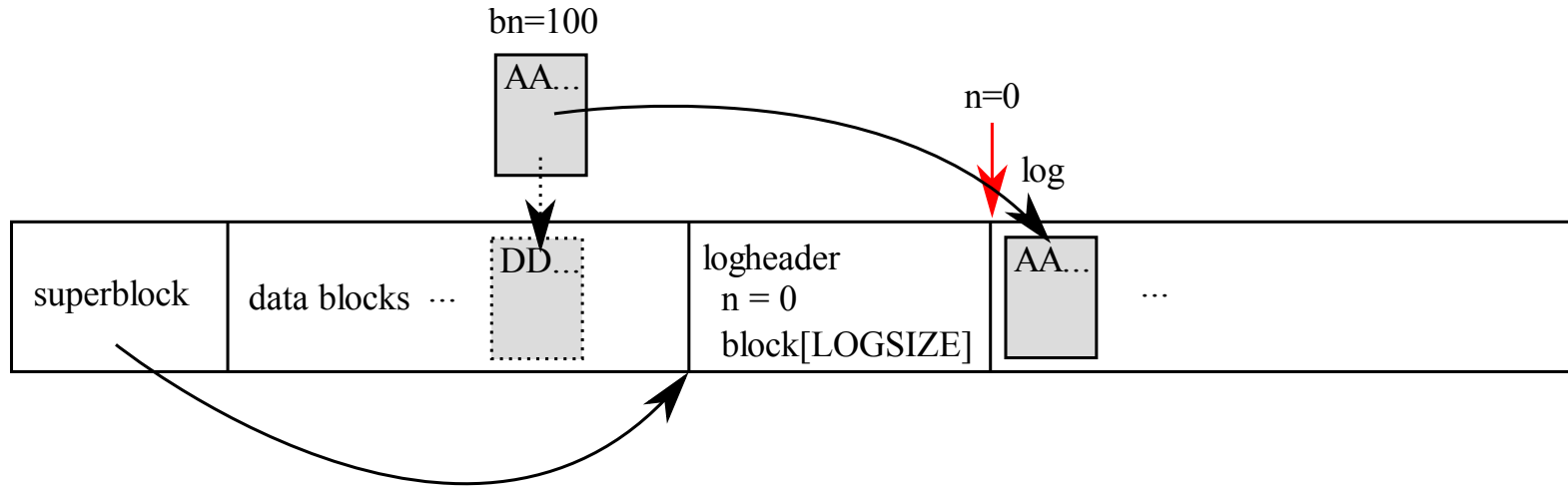
Disk structure for logging



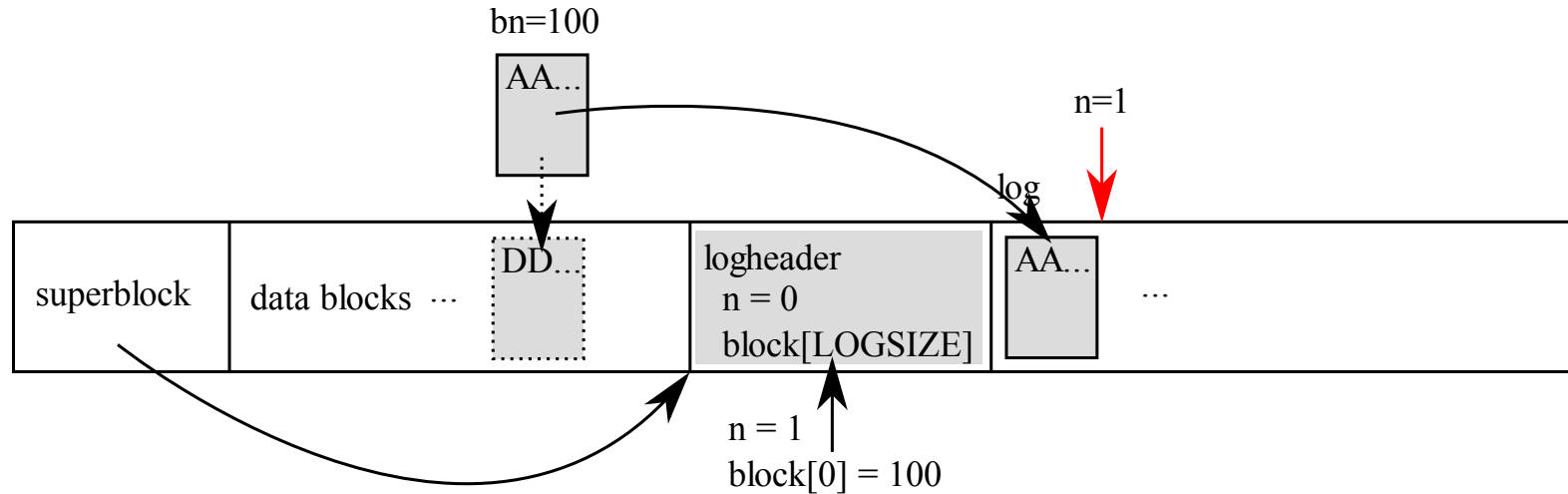
Example: writing a block (bn = 100)



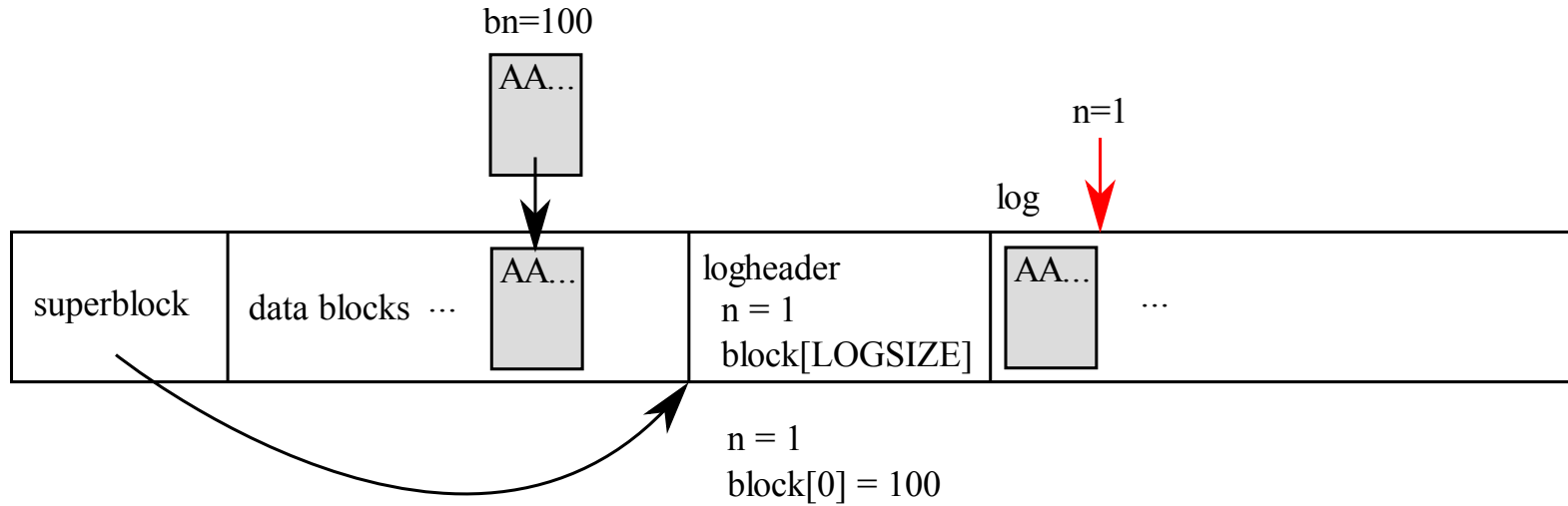
Step1: writing to a log



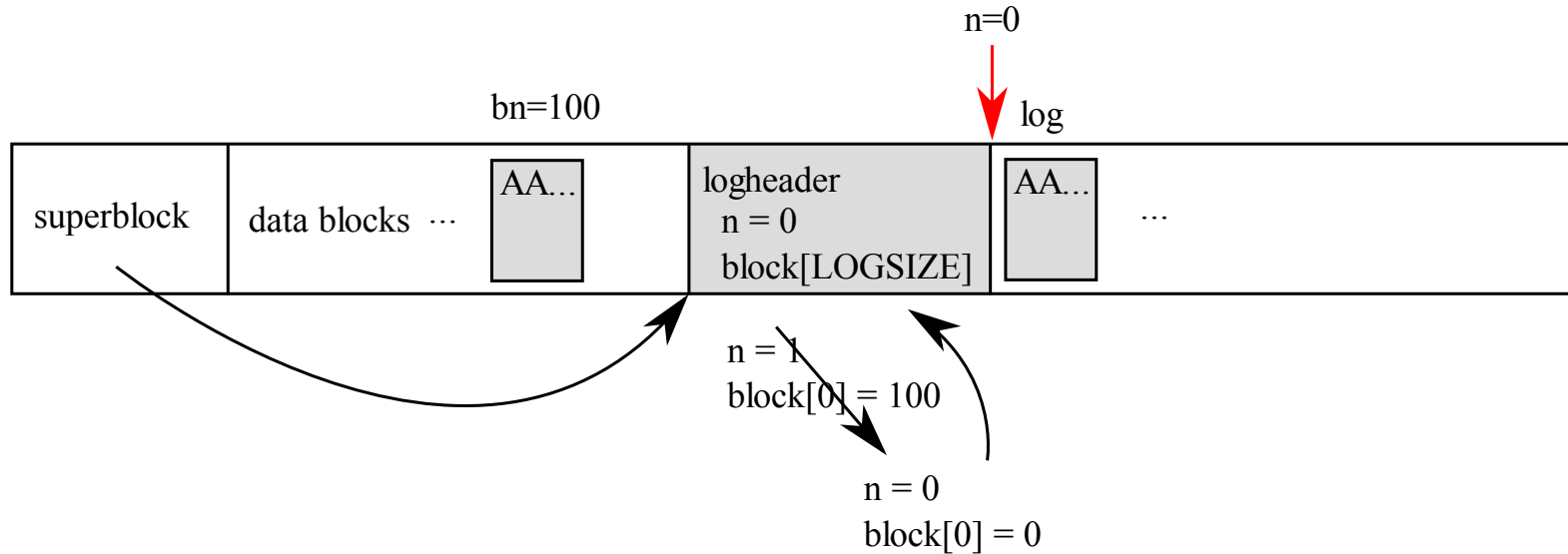
Step2: flushing the logheader (committing)



Step3: overwriting the data block



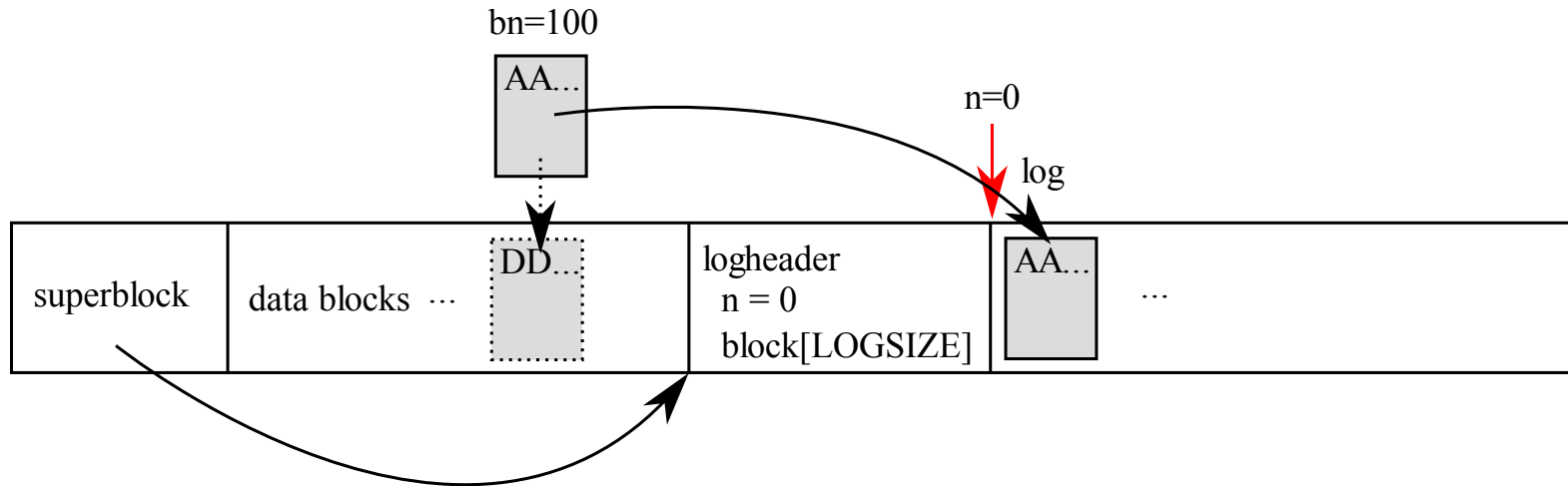
Step4: cleaning up the logheader



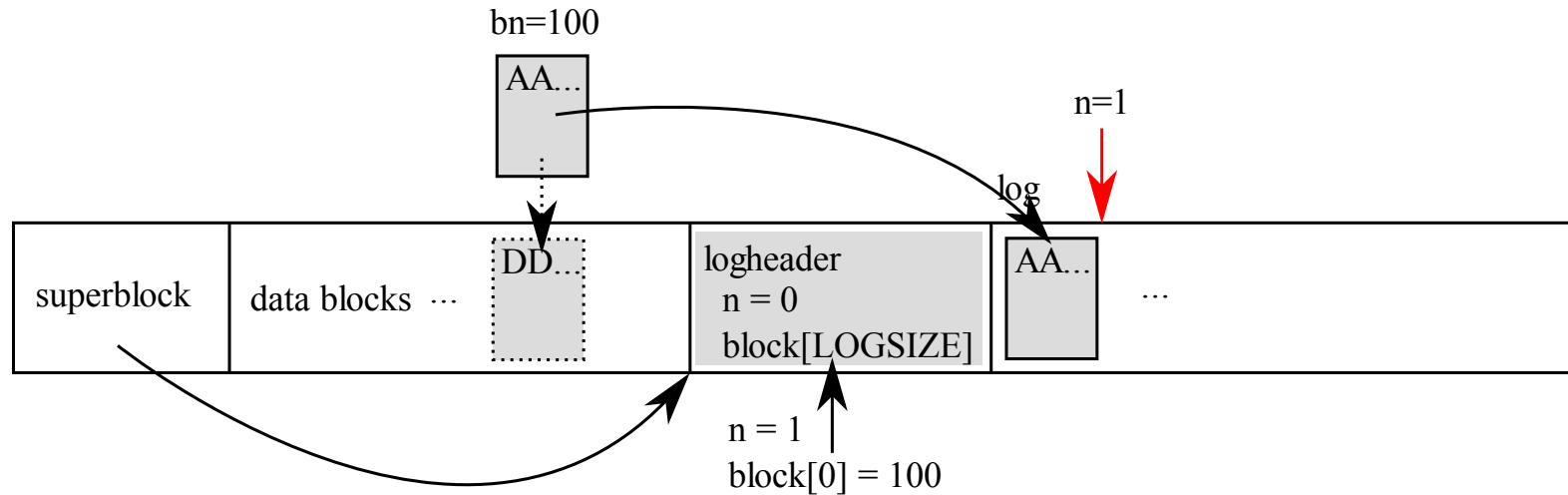
What if failed (say power-off and reboot)?

- 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?)

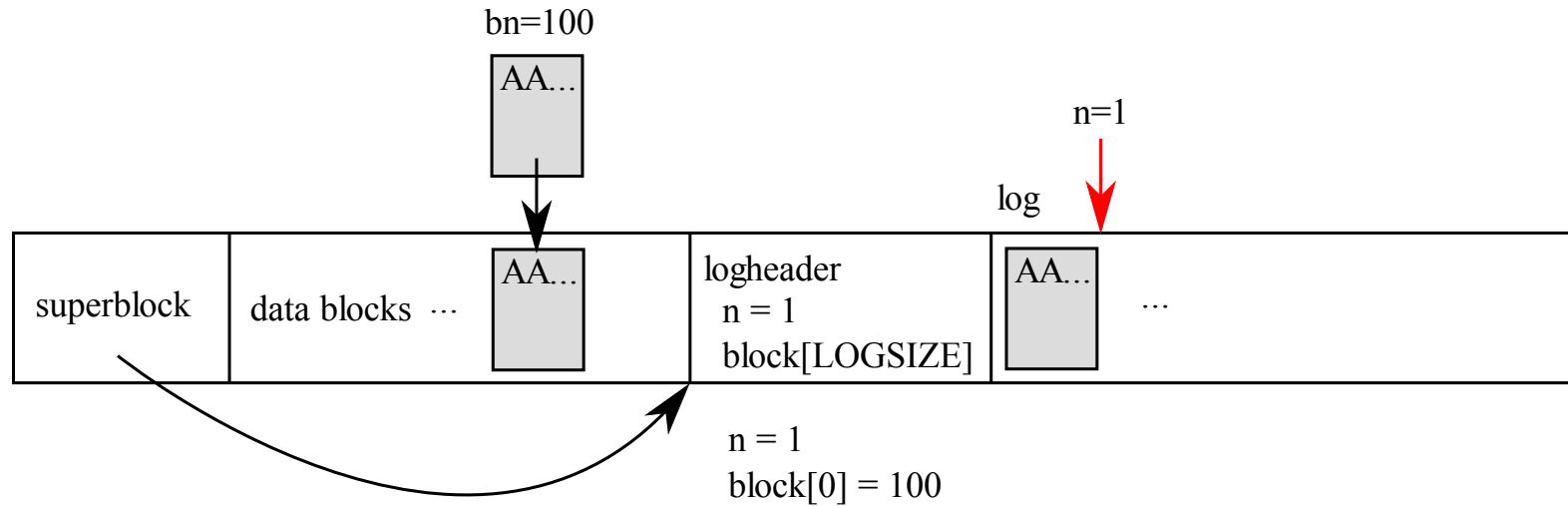
Q? Step1: writing to a log



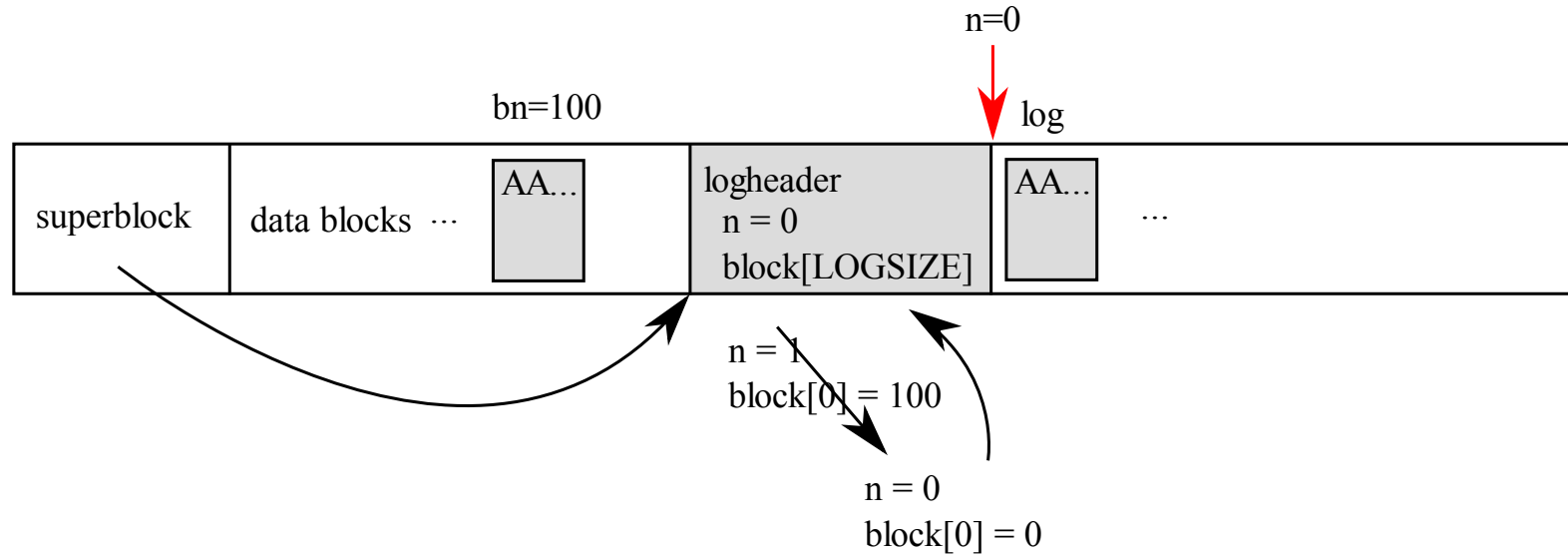
Q? Step2: flushing the logheader



Q? Step3: overwriting the data block



Q? Step4: cleaning up the logheader



DEMO: dumplog.c

```
01     static void commit() {
02         if (log.lh.n > 0) {
03             write_log();      // Write modified blocks from cache to log
04             // Q1: panic("after writing to log!");
05             write_head();    // Write header to disk -- the real commit
06             // Q2: panic("after writing the loghead!");
07             install_trans(); // Now install writes to home locations
08             // Q3: panic("after the transaction!");
09             log.lh.n = 0;
10             write_head();    // Erase the transaction from the log
11             // Q4: panic("after cleaning the loghead!");
12         }
13     }
```

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