Crash-consistent applications

Key takeaways

1. One logical update is many physical writes; a crash exposes the states between.
2. Atomicity comes from a single commit point: an atomic rename or a logged commit.
3. Recovery is safe only when updates are idempotent and writers are locked.

Common misunderstandings of crash-consistent updates

Which claim would you defend right now? Choose before the evidence.

“If close() returns, the bytes are safely on stable storage.”

“The disk persists my writes in the order my program issued them.”

“After a crash, rerunning an append-only update safely finishes the job.”

Keep your choice. Look for the mechanism or counterexample that would make it fail.

Your program will be killed mid-write

Design for the kill, not the happy path

A kill signal, an OOM kill, a power cut: the process never gets to finish. Kill an in-place updater halfway through and the index is left in a state your code never intended to write. The next command reads it.

A crash lands in the middle of your write

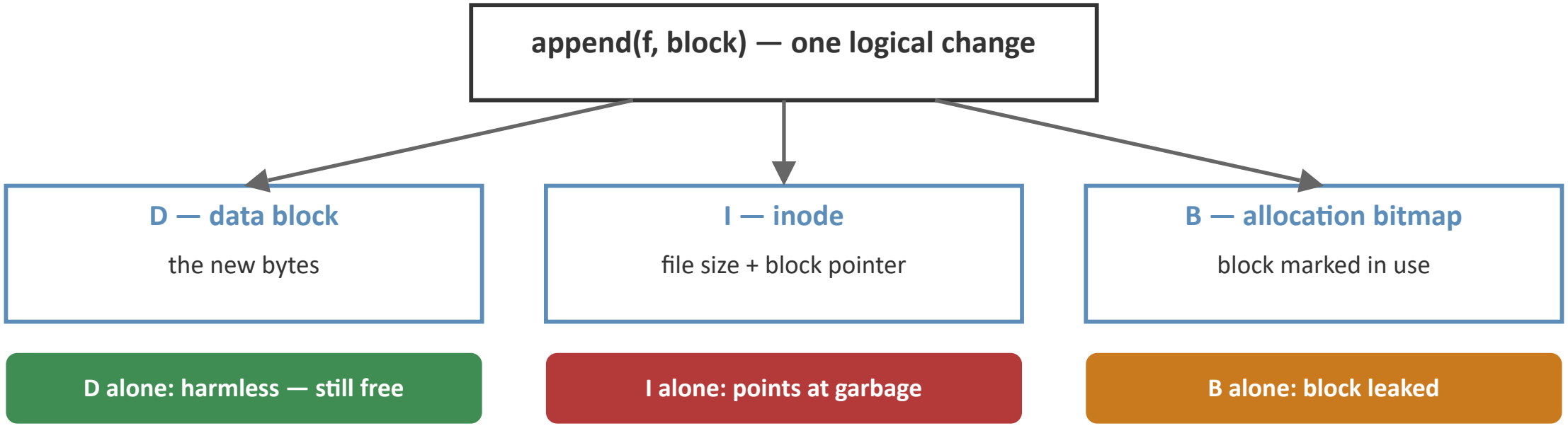
```
$ mindex update --db idx.db mail/    # writing 900 new postings...
[ kill -9 / OOM / power cut ]        # dies after 450

$ mindex search --db idx.db "cxl"    # what does this read now?
#  -> half the postings present? a torn row? an empty file?
```

The crash is the common case. Your on-disk format has to survive it.

One logical update is many physical writes

Even the filesystem has this problem

Appending one block to a file changes three on-disk structures. The kernel writes them back lazily, in any order — a crash persists an arbitrary subset.



Crash consistency: every state a crash can expose must be recoverable to a correct one.

Where a crash lands

An atomic, idempotent update turns every CORRUPT and DUPLICATED point into SAFE.

close() is not durability

close() promises the file descriptor is gone, not that your bytes are on the disk. They can sit in the kernel page cache for seconds. Only fsync forces them to stable storage.

Looks saved — bytes may be RAM-only

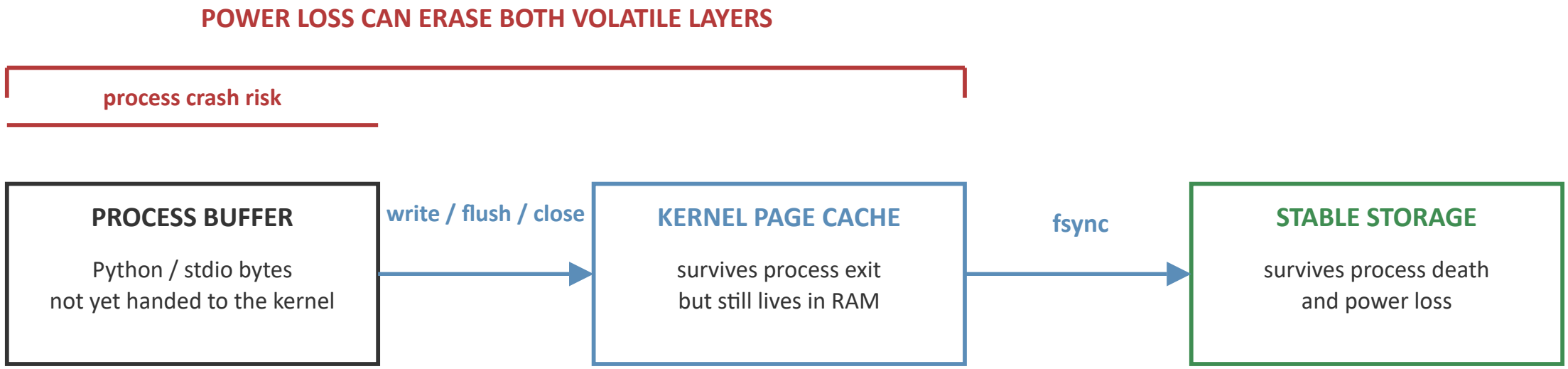
```
with open("idx.db","wb") as f:
    f.write(new_bytes)
# close() returned;
# a power cut here loses it
```

Actually durable

```
with open("idx.db","wb") as f:
    f.write(new_bytes)
    f.flush()
    os.fsync(f.fileno())
# now the platter has it
```

Two failures erase different layers

close() changes ownership; fsync() changes survival



close() can move bytes out of the process; only fsync moves them beyond the power-loss boundary.

Two ways to buy back atomicity

Funnel many writes into ONE commit point: before it all-old, after it all-new

Copy-on-write: publish by rename

```
write_all("idx.db.tmp")
fsync(tmp_fd)
os.replace(tmp, "idx.db")
#   ^ THE commit point
fsync(dir_fd)
```

Write-ahead log: publish by commit record

```
log.append(intents...)
log.append(COMMIT); fsync(log)
#   ^ THE commit point
apply_in_place(intents)
# crash? replay committed log
```

Filesystems journal their own metadata exactly this way (ext4, XFS, NTFS). The fallback when there is no log is fsck: scan the whole disk and repair the invariants — hours of recovery instead of a log replay.

The durable-write recipe

1. write tmp → **2. fsync(tmp)** → **3. rename** → **4. fsync(dir)**

1. write tmp	2. fsync(tmp)	3. rename	4. fsync(dir)
new bytes → idx.db.tmp	force tmp to the platter	os.replace(tmp, idx.db)	force the rename itself
durable: nothing yet — but idx.db stays intact	durable: the new contents survive a crash	durable: readers see all-old OR all-new	durable: the swap survives a crash too

```
tmp = "idx.db.tmp"
write_index(tmp); os.fsync(tmp_fd)   # 1-2  contents on the platter
os.replace(tmp, "idx.db")            # 3    atomic rename: all-or-nothing
os.fsync(dir_fd)                     # 4    the rename itself is durable
```

The live file is only ever swapped in whole. A crash can’t leave it torn.

A database does this for you

SQLite is the write-ahead-log mechanism packaged behind two words: everything between BEGIN and COMMIT lands all-at-once, and a crash before COMMIT rolls back to the old state. That is why the assignment forbids half-measures — the commit is the one durable instant.

ref_mindex.py: all mutations in ONE transaction, crash-before-commit rolls back

```
con.execute("BEGIN")              # open one transaction
apply_all_changes(con)            # inserts + deletes, many rows
#  <-- crash here: next open rolls the whole batch back
con.commit()                      # the single durable instant
```

How the journal/WAL makes that commit atomic: write-ahead logging, W5-Thu.

Idempotent updates recover safely

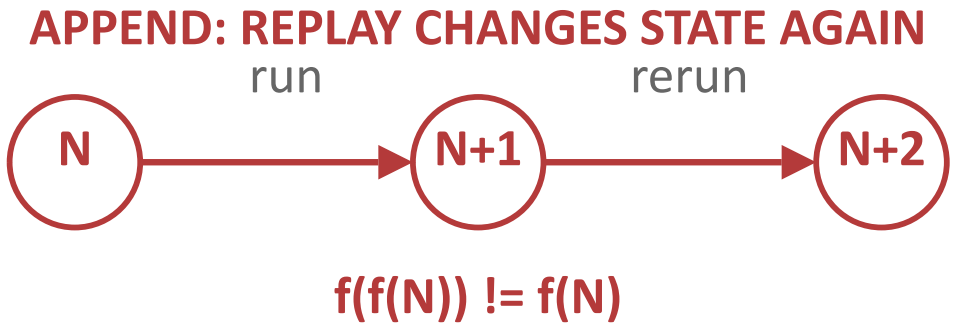
Recovery is just running update again. A full rebuild is fsck for your index — correct, but it punishes every crash. Reconcile to a target instead: update-twice == update-once. Append blindly and the rerun double-indexes what the crash already wrote.

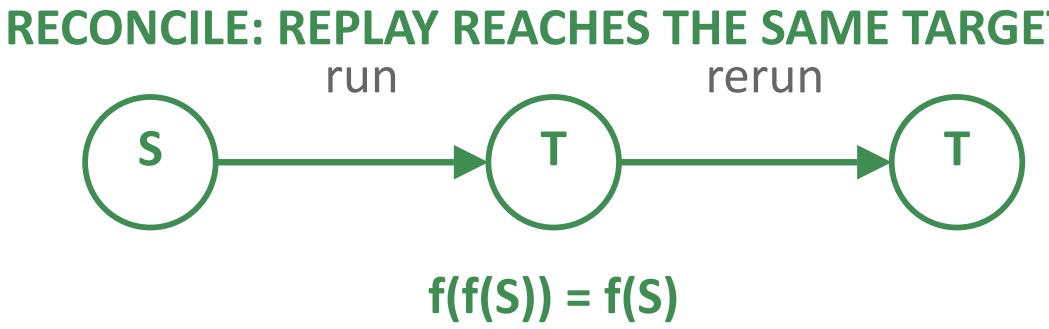
Append-only — rerun duplicates

```
def update(corpus):
    for m in corpus:
        insert_postings(m)
# rerun: every posting twice
```

Reconcile — idempotent

```
def update(corpus):
    add, gone, chg = diff(now)
    delete(gone + chg)
    insert(add + chg)
# rerun: diff empty, a no-op
```





Serialize writers with a lock

Launch update twice and they race on your index's uniqueness constraint. An exclusive flock on <db>.lock makes the second one block (or exit cleanly). Advisory means only code that also asks is held back — and Windows has no flock at all (the cross-platform corner, W2's story).

V3 invariant 10: exclusive flock, block-or-exit, never a torn concurrent read

```
import fcntl
lock = open("idx.db.lock", "w")     # never unlink it: truncate in place
fcntl.flock(lock, fcntl.LOCK_EX)    # advisory, exclusive — 2nd update waits
try:
    reconcile_and_commit(corpus)    # the whole critical section
finally:
    fcntl.flock(lock, fcntl.LOCK_UN)
```

Graders inject the crash

The grader doesn’t wait for a real crash — it makes one. The tool honors MINDEX_CRASH_AFTER=N and dies after N messages, before commit; one clean recovery pass must then equal a fresh build. A random SIGKILL audit catches tools that try to dodge the hook.

The tool honors the crash hook

```
n = os.getenv("MINDEX_CRASH_AFTER")
for i, m in enumerate(batch):
    index_one(m)
    if n and i + 1 >= int(n):
        os._exit(137)  # pre-commit
```

The grader asserts recovery == oracle

```
run(update, env=CRASH_AFTER_5)
run(update)           # recovery
assert index == fresh_build(corpus)
# torn or dup: fails right here
```

Key Takeaways

1. One logical update is many physical writes — a crash exposes the states between
2. Buy atomicity with one commit point: atomic rename or a logged commit (WAL: W5)
3. Make recovery safe: update must be idempotent, and concurrent writers must be locked