Crash consistency – better late than never.

Motivation

- It's a bright, sunny day at Fairbanks Park – I'm using my favorite photo app on my phone.
- **Notification** - Please update the app to the latest version to experience the awesomeness!
- Halfway through updating – the phone hangs and a familiar error message pops up: "Uh no! The app has unexpectedly crashed."
- Oh well, let's download it again – but what the..!? What happened to **all** my preferences (metadata)? I've been using this phone for years, where did **all** my photos (data) go??
- It's all gone! All those memories. ☹

Definition

Crash consistency: making sure an application's data can be recovered if a sudden failure (system/power) occurs.

Perspective

Many application-level crash-consistency problems occur when faced with "bad timing" conditions or the way a file system has been configured.

> *Eg: While executing a git-commit; wait for five seconds; pull the power plug; after rebooting the machine the git repository is probably corrupted.*
> *(All may not be lost should we have a clone of that repository, there can be way to recover.)*

It's strange that we've somehow *accepted this reality* that should our application crash, for whatever reason, data loss/corruption is a probable (inevitable) consequence.

As a safeguard, **Update protocols** have been put into place, which are essentially:

- **System calls** (file writes/renames etc)
- **Update** underlying files and directories in a **recoverable way**.

However, what has happened is:

- some standardized file system interface is being used wide-spread
- makes applications vulnerable to unexpected behaviors based on which type of file system is used and/or the configuration of the file system that's implemented.
- Consequences?
    - The app code might have been tailored to match file system internals but there could be a blatent violation of layering and modularization
    - Time consuming/error prone during backup and restoring, which the users will face.

Conclusion: file system abstraction is broken!

Example: A DBMS that stores data in a single file

- DBMS maintains transaction atomicity during a system crash.
- DBMS uses an update protocol called Undo-logging: before updating the file.
- Essentially: DBMS simply records specific portions (offset) of the file that are about to be updated within a separate log file.

Algo 1: Incorrect Undo-logging Pseudocode

1. creat(log);
2. write(log, “<offset>, <size>, <data>”); # making a backup in the log file
3. write(dbfile, offset, data); # actual update
4. unlink(log); # deleting the log file

NB: The DBMS uses POSIX (Portable Operating System Interface) – standard file system interface used in Unix-like systems.

- When DBMS is started, it rolls back the transaction if a log file exists and is fully written.

Key point: Because file systems buffer write in memory and send them to disk later, from the perspective of an application most file systems can *reorder* the effects of system calls before persisting them on disk.

Eg: File systems such as ext2, ext4, xfs and btrfs) – deletion of the log file can be reordered before the write to the database file.

- Should the system crash in any of the above file systems, log file found might already be deleted from the disk, while the DB has updated only partially.

Some systems [ext2 (default) and ext3/4 (nondefault configs)] behave in nonsensical ways:

- While writing to the log file, a crash might leave garbage data in the newly appended portions of the file.
- During recovery, one can’t tell whether the log file contains garbage or undo information.

Algo 2: Correct Undo-logging Pseudocode for Linux Kernel

1. creat(log);
2. write(log, “<offset>, <chksum>, <size>, <data>”); # <chksum> : log file can end up with garbage values (ext2, ext3-wb ext4-wb)
3. fsync(log); # write(log) && write(dbfile) can be reordered in all configs
4. fsync(./); # creat(log) can be reordered after write(dbfile) – according to Linux man page
5. write(dbfile, offset, data);
6. fsync(dbfile); # write(dbfile) can reorder after unlink(log) – all config, except ext3 (default)
7. unlink(log);
8. fsync(./); # for durability purposes