Term Project: Metadata Journaling

Goal

You are given a **pre-built disk image** named `vsfs.img` containing a small VSFS-like file system. Your task is to add **metadata journaling** so that file-system metadata updates can be made **crash-consistent**. You are also given a sample program `mkfs` that will help you create this disk image. The image is static and has the following layout:

Block Size: 4 KB

Component	Blocks
Superblock	1
Journal	16
Inode Bitmap	1
Data Bitmap	1
Inode Table	2
Data Blocks	64

Total Blocks: 85

You will implement a program named `journal.c` that edits the disk image directly to perform metadata journaling.

Using `mkfs` and Understanding VSFS

You can recreate the `vsfs.img` file system using `mkfs` using the following command:

```
./mkfs
```

This will create a new `vsfs.img` file with the previously specified format. A few things to note:

- The 16 journal blocks are empty. It is your job to implement the journal here.
- Initially `vsfs.img` has only one inode for the root directory, `/`, with `inum = 0`. This also points to the first data block, containing the **directory entries** for the root

directory.

- The inode and data bitmaps are updated accordingly.

Commands and required behavior

1) `./journal create <name>`

Creates a file entry **by logging metadata only**.

What it must do:

- Initialize the journal if it does not exist.
- Read existing metadata to decide what would change:
  - find a free inode in **inode bitmap**
  - choose a free directory entry slot in the root directory
- Compute the *new versions* of the metadata blocks **in memory** (not on disk yet).
- Append one **transaction** into the journal block:
  - log **each** modified metadata block as a DATA record
  - finish the transaction with a COMMIT record
- Write changes only to the **journal block** (and its header).

  Do not modify the inode bitmap / inode table / directory blocks in their home locations during `create`.

Result after `create`:

- The disk image's "real" metadata will still show the old state.
- The journal contains a committed transaction describing the metadata updates that will be applied later by `install`.

---

2) `./journal install`

Applies committed journaled metadata updates to their home locations and then clears the journal.

What it must do:

- If the journal does not exist, it must return a failure.
- Read and parse the journal sequentially.

- For each transaction that has a COMMIT record:
    - replay every logged DATA record by writing its block image to its target home block number
- After replaying all committed transactions:
    - clear (checkpoint) the journal so it becomes empty again

Journal design

Journal block overview

The journal is 16 blocks long and is treated as an **append-only byte array**:

`[ journal_header ][ record ][ record ][ record ] ...`

Journal header (fixed at offset 0)

```
#define JOURNAL_MAGIC 0x4A524E4C

struct journal_header {
    uint32_t magic;         // store JOURNAL_MAGIC
    uint32_t nbytes_used;   // total bytes currently used
};
```

Conceptually:

- `magic` indicates that this is indeed a journal header. We will store the characters `JRNL` in this 4-byte field.
- `nbytes_used` tells you where the next append will go. **You must change** its value **each** time a new entry is appended to/removed from it.
- "empty journal" means `nbytes_used == sizeof(journal_header)`

Journal Record Header

Each journal record has a header that tells:

- the type of the record (DATA/COMMIT)
- the size of the record

```
struct rec_header {
    uint16_t type;   // REC_DATA or REC_COMMIT
    uint16_t size;   // total size of this record in bytes
};
```

Storing the record size is important as we need to know how far we need to traverse in the journal to access the next record. The next subsection mentions how to calculate this size.

DATA record

A DATA record represents **one full filesystem block image** that should be written to a particular home block.

It contains:

- the target home `block_no` (absolute block index in the disk image)
- the full 4096-byte contents that should replace that block

```
struct data_record {
    struct rec_header hdr;   // type = REC_DATA
    uint32_t block_no;       // home block number on disk
    uint8_t  data[4096];     // full block image
};
```

Important constraints:

- The total size of a record is equal to sum of the size of the header, the size of `block_no` and the block size.
- You will have to log **whole blocks** at once. For example, if you need to add a directory entry to the root directory (add a file using the `create` command, you need to add the **entire** updated root directory data block in the journal, not just the new entry.
- You log **only metadata blocks** (bitmap blocks, inode table blocks, directory blocks).

COMMIT record

A COMMIT record seals one transaction.

```
struct commit_record {
    struct rec_header hdr;   // type = REC_COMMIT
};
```

Semantics:

- All `DATA` records since the last `COMMIT` (or since the journal start) belong to the current transaction.
- A transaction is considered valid **only if** its `COMMIT` record is present and fully readable.

Data structures in the file system

Superblock (128 bytes)

```
#define FS_MAGIC 0x56534653  // "VSFS"

struct superblock {
    uint32_t magic;            // FS magic number
    uint32_t block_size;       // 4096
    uint32_t total_blocks;     // total blocks in FS
    uint32_t inode_count;      // total inodes

    uint32_t journal_block;    // block index of journal
    uint32_t inode_bitmap;     // inode bitmap block
    uint32_t data_bitmap;      // data bitmap block
    uint32_t inode_start;      // first inode block
    uint32_t data_start;       // first data block

    uint8_t  _pad[128 - 9*4];  // padding to 128 bytes
};
```

Inode (128 bytes)

```
struct inode {
    uint16_t type;             // 0=free, 1=file, 2=dir
```

```
    uint16_t links;            // link count
    uint32_t size;             // file size in bytes

    uint32_t direct[8];        // 8 direct block pointers

    uint32_t ctime;            // creation time
    uint32_t mtime;            // modification time

    uint8_t  _pad[128 - (2+2+4 + 8*4 + 4+4)]; // nothing here
};
```

Inode and Data Bitmaps

These are just byte arrays with 4096 elements.

Directory Entry (32 bytes)

```
#define NAME_LEN 28

struct dirent {
    uint32_t inode;            // inode number (0 = unused)
    char     name[NAME_LEN];   // null-terminated if shorter
};
```

How journaling works

- `create` does not "apply" metadata; it **writes** the new metadata blocks in the journal and commits.
- `install` replays committed block images from the journal into their home locations.

Operational Constraints

- You may append multiple transactions into the single journal block until it fills.
- If there is insufficient space to append a full new transaction:
    - the command should refuse and instruct the user to run `install`

- Only metadata journaling is required.

Using `validator`

You are provided another program `validator` which you can use to check whether your file system is consistent at any point of time. To use `validator`, you can simply run:

```
./validator
```

Confirm whether the originally created file system is consistent by running the validator once. After a successful `create` or `install`, the image should also be structurally consistent:

- inode bitmap matches which inode slots are in use
- directory entries reference allocated inodes
- no invalid block numbers, no duplicates, no dangling references.

Instructions

Suggested Workflow

- Understand the underlying structure of the filesystem so that it is easy to traverse using struct pointers.
- Implement `create`: remember, this appends individual records for the inode bitmap, inode table and root directory data block.
- Check whether your filesystem is consistent after each `create` by running the validator.
- Implement `install`, keeping in mind the constraints mentioned in the specifications.
- **Important:** Check whether your filesystem is consistent again by running the validator after `install`.

Some Useful Tips

- For debugging, you will require the usage of hex inspectors. These can open a binary file and inspect the data present in it. There are a lot of graphical

inspectors/editors (HxD, Okteta, etc.) and CLI inspectors (xxd, hexdump) available.

- C stores all data structures in **little endian** format. You should read up on endianness if you are not familiar with the term.
- Some sample bitmap mappings are provided below for better understanding:
    - Inode 0 (root): byte 0, bit 0 (LSB)
    - Inode 1: byte 0, bit 1 (from LSB)
    - Inode 8: byte 1, bit 1 (from LSB)
    - Inode 20: byte 3, bit 4 (from LSB)