Homework 3: MiniFS: A Simple User-Space File System in C

BLG312E – Computer Operating Systems

Due Date: June 4, 2025, @23:59

1. Overview

Implement a user-space file system, `MiniFS`, stored in a single disk image (`disk.img`). The image represents a 1 MB disk of 1024 blocks (1024 bytes each). Your implementation must:

- Initialize the disk image (`mkfs`) by writing the superblock, free-block bitmap, inode table, and root directory.
- Provide file operations: `create_fs`, `write_fs`, `read_fs`, `delete_fs`.
- Provide directory operations: `mkdir_fs`, `ls_fs`, `rmdir_fs`.
- Use only standard C file I/O (`fread`, `fwrite`, `fseek`) for block-level access.

2. Disk Layout

Region	Block(s)	Purpose
Superblock	0	Filesystem metadata
Free-block bitmap	1	Tracks free data blocks
Inode table	2–10	File/directory descriptors
Data blocks	11–1023	Content storage

Notes:

(1) Define `#define BLOCK_SIZE 1024` consistently in `disk.h` and `fs.h`.

(2) The free-block bitmap (block 1) tracks only data blocks; free inodes are tracked via the `is_valid` field in each `Inode`, which effectively functions as an inode bitmap.

3. Core Data Structures

```
// Superblock
typedef struct {
  int magic_number;   // filesystem identifier
  int num_blocks;     // total blocks (1024)
  int num_inodes;     // total inodes (e.g., 128)
  int bitmap_start;   // block index of free-block bitmap
  int inode_start;    // block index of inode table
  int data_start;     // block index of first data block
} SuperBlock;
```

```
// Inode
typedef struct {
  int is_valid;            // 0=free, 1=used
  int size;                // bytes (file) or entry count (
     directory)
  int direct_blocks[4];    // direct block pointers
  int is_directory;        // 0=file, 1=directory
  int owner_id;            // your student id number
} Inode;

// DirectoryEntry
typedef struct {
  int inode_number;
  char name[28];           // 27 ASCII chars + null terminator
     (\texttt{\textbackslash 0})
} DirectoryEntry;
```

4. API Specification

Declare in `fs.h` and implement in `fs.c`. Invoke from `main.c`:

```
void    mkfs(const char *diskfile);
int     mkdir_fs(const char *path);
int     create_fs(const char *path);
int     write_fs(const char *path, const char *data);
int     read_fs(const char *path, char *buf, int bufsize);
int     delete_fs(const char *path);
int     rmdir_fs(const char *path);
int     ls_fs(const char *path,
              DirectoryEntry *entries,
              int max_entries);
```

5. Initialization and Path Semantics

- **mkfs:** Create or reset `disk.img`. Initialize superblock, clear bitmap, zero inodes, and allocate the root directory (`is_directory=1`, `size=0`).
- **Path format:** Only absolute paths (begin with `/`); no relative components (“.” or “..”). Parent directories must pre-exist.
  Example: `/src/fs.c`
- **Return conventions:** Success returns 0 (or byte count for write/read); failure returns -1 and prints a descriptive message to `stderr`.

- **Constraints:** Max filename length = 27 ASCII characters (case-sensitive); Unicode not supported.

6. Illustrative Examples

1. **Format the image:**

```
$ ./mini_fs mkfs
```

2. **Directory operations:**

```
$ ./mini_fs mkdir_fs /src
$ ./mini_fs ls_fs /
src/
```

3. **File operations:**

```
$ ./mini_fs create_fs /src/fs.c
$ ./mini_fs write_fs /src/fs.c "Hello MiniFS"
// prints 12
$ ./mini_fs read_fs /src/fs.c
// outputs "Hello MiniFS"
```

4. **Removal:**

```
$ ./mini_fs delete_fs /src/fs.c
$ ./mini_fs rmdir_fs /src
```

7. Implementation Considerations

- Accurate block indexing: prevent metadata/data corruption.
- Bitmap organization: map only data blocks (11–1023).
- Directory capacity: enforce `BLOCK_SIZE/sizeof(DirectoryEntry)` limit.
- Path tokenization: split deterministically; consistent separators.
- Each inode contains only four direct block pointers, so the largest file you can store is 4 × BLOCK_SIZE bytes. If a call to write_fs() would require more than four blocks (i.e. more than 4 × BLOCK_SIZE bytes), the function must refuse to write and return -1.
- Error formatting: print to `stderr`, e.g., “Error: disk is full”.
- Edge cases: zero-length writes, full-disk, empty directories.
- Performance: minimize redundant `fseek`/`fread`.

8. Automated Evaluation Requirements

Students must include:

- A `tests/` directory with:
  - `commands.txt`: CLI command sequence.
  - `expected_output.txt`: expected `stdout`.
- A `check` target in the `Makefile`:

```
check: mini_fs
        ./mini_fs < tests/commands.txt > tests/output.txt 2>/
           dev/null
        diff -u tests/expected_output.txt tests/output.txt \\
          || { echo "Output␣mismatch"; exit 1; }
```

9. Assessment and Grading Criteria (100 pts)

- Disk formatting and superblock layout: 10 pts
- File and directory operations: 20 pts
- Block and inode allocation logic: 20 pts
- Screenshots and terminal proofs: 15 pts
- Explanatory report quality (with answers to the questions): 20 pts
- Code readability and comments: 15 pts

10. Additional Requirements

10.1 Academic Integrity

- Code must be in C using provided **Docker** environment.
- No deadline extensions.
- Plagiarism prohibited; similarity checks applied.
- AI tools may not be used to generate solutions.
- All code, comments, screenshots must be original.

10.3 Functional Demonstration

Provide in `main.c` a sequence to:

- Create a directory.
- Create/write a file.
- Read the file.
- List directory contents.
- Delete file and directory.
- Demonstrate bitmap/inode reuse.

Include `run_log.txt` and `screenshots/`.

10.4 Report Questions

Answer:

1. How are directory entries stored/searched?
2. What happens when the disk or inode table is full?
3. How is double allocation prevented?
4. Describe one encountered error and your solution.

10.5 Submission Instructions

- Submit a zip named `StudentNo_hw3.zip` containing: `fs.c`, `fs.h`, `main.c`, `disk.img`, `README.md`, `run_log.txt`, `screenshots/`, `report.pdf`.
- Submit via Ninova by June 4, 2025 at 23:59.