MiniVSFS: A C-based VSFS Image Generator

You will implement **`mkfs_builder`**, a C program that **creates a raw disk image** for a small, inode-based file system **MiniVSFS**. The program takes some parameters from the command line input, and emits a byte-exact image as a binary **.img** file.

You will also implement **`mkfs_adder`**, another C program that takes a raw MiniVSFS file system image, and a file to be added to that file system. **`mkfs_adder`** will find the file in the working directory and add it to the **root directory (/)** of the file system. Finally, it will save the output image in the new file specified by the output flag.

To understand how to start working on the project, refer to the Project Starting Point section.

MiniVSFS

MiniVSFS, based on VSFS, is fairly simple – a block-based file system structure with a **superblock, inode and data bitmaps, inode tables, and data blocks**. Compared to the regular VSFS, MiniVSFS cuts a few corners:

- Indirect pointer mechanism is not implemented
- Only supported directory is the root (/) directory
- Only one block each for the inode and data bitmap
- Limited size and inode counts

What You'll Build

MKFS_BUILDER

```
mkfs_builder \
  --image out.img \
  --size-kib <180..4096> \
  --inodes <128..512>
```

- image: the name of the output image
- size-kib: the **total** size of the image in kilobytes (multiple of 4)
- inodes: number of inodes in the file system

MKFS_ADDER

```
mkfs_adder \
  --input out.img
  --output out2.img
  --file <file>
```

- input: the name of the input image
- output: name of the output image
- file: the file to be added to the file system

Output

- the updated output binary image with the file added

Program Workflow

1. MKFS_BUILDER

It should perform the following tasks in order:

1. Parse the command line inputs
2. Create the file system according to the provided specifications
3. Save the file system as a binary file with the name specified by the *--image* flag

2. MKFS_ADDER

It should perform the following tasks in order:

1. Parse the command line inputs
2. Open the input image as a binary file
3. Search the file in the present working directory, and add the file to the file system.
4. Update the file system binary image

MiniVSFS Specifications

Block Size = 4096 Bytes
Inode Size = 128 Bytes
Total Blocks = size_kib * 1024 / 4096

Disk Model

All on-disk structures are little endian in format. Disks are divided into equally sized (4096 B) blocks, the blocks are arranged as follows:

Superblock (1 block)	Inode Bitmap (1 block)	Data Bitmap (1 block)	Inode Table	Data

You can assume the following information for the structures:

Superblock

The superblock is a structure to be placed on the first block (block 0) of the image:

field	size (bytes)	default
magic	4	0x4D565346
version	4	1
block_size	4	4096
total_blocks	8	
inode_count	8	
inode_bitmap_start	8	
inode_bitmap_blocks	8	
data_bitmap_start	8	
data_bitmap_blocks	8	
inode_table_start	8	
inode_table_blocks	8	
data_region_start	8	
data_region_blocks	8	
root_inode	8	1
mtime_epoch	8	Build time (Unix Epoch)

flags	8	0
checksum	8	Check discussion on checksum

Skeleton for the superblock has been created as the struct *superblock_t*.

Inodes

Inodes are 128 byte structures with the following format:

field	size (bytes)	default
mode	2	Check discussion on modes
links	2	Check discussion on links
uid	4	0
gid	4	0
size_bytes	8	
atime	8	Build time (Unix Epoch)
mtime	8	Build time (Unix Epoch)
ctime	8	Build time (Unix Epoch)
direct[12]	4 (each)	
reserved_0	4	0
reserved_1	4	0
reserved_2	4	0
proj_id	4	Your group ID
uid16_gid16	4	0
xattr_ptr	8	0
inode_crc	8	Check discussion on checksum

Twelve direct blocks are allowed in MiniVSFS. Elements inside the direct array are **absolute** data block numbers. Skeleton for the inode has been created as *inode_t*.

N.B. Inodes do not have an explicit id/inode number field. They are referred to by their index in the inode table. Note that the *root_inode* field in the superblock struct is set as 1, as inodes are **1-indexed** in the table. Thus, for direct blocks which are unused, you can simply set them to 0 to show that they are unoccupied.

N.B. 1-indexing does not mean the first element of the table is empty, rather it means the first element is indexed as 1. You can simply add 1 to the index of the inode in the table to find its inode number.

Inode Modes

Depending on whether the inode represents a file/directory, the 16-bit mode is calculated:

- $(0100000)_8$ for files
- $(0040000)_8$ for directories.

Bitmaps (inode and data)

- Bit = 1 means allocated, 0 means free
- Bit 0 of byte 0 refers to the first object (inode #1 for inode bitmaps, or the first data block in the data region for data bitmaps)
- Bitmaps occupy entire blocks (zero padded tail)

Directory Entry

A directory entry is a data structure held by each directory containing useful information about its parent and children:

field	size (bytes)	default
inode_no	4	0, if free
type	1	1=file, 2=dir
name	58	
checksum	1	Check discussion on checksum

Skeleton for this entry has been created as *dirent64_t*.

File Allocation Policy

- Inodes and data blocks are placed on a **first-fit** basis: the first available block is allotted to the new resource.
- If a file is too large to be accommodated with 12 direct blocks, you should return a warning message saying accordingly.

Root Directory

One of the first things that you will need to implement is the root directory. The root directory has a **fixed inode number of 1** (as stated earlier), and two members: . and .. , both pointing to itself. It also has the first data block occupied for itself to store its entries.

Links

The link field of the inode contains the number of directories that point to the file/directory. The root directory has 2 links initially (. and ..), and every file inside it has 1 link. When a new file is created, the link count of the root increases by 1 as the new file now refers to the root by ..

Checksum

For each of the skeleton structures of the superblock, the inode, and the directory entry, a checksum is required, which you can compute using the given **superblock_crc_finalize**, **inode_crc_finalize**, and **dirent_checksum_finalize**, which takes in pointers to the respective structures and calculates them accordingly.

Sample code to update the structures is shown below:

```
superblock_t superblock;
inode_t inode;
dirent64_t dirent;

/* proper configuration of the structures */

superblock_crc_finalize(&superblock);
inode_crc_finalize(&inode);
dirent_checksum_finalize(&dirent);
```

Time

You can get the Unix epoch from the *time* library in C, using the **time** function.

```
#include <time.h>

time_t now = time(NULL);
```

Discussion

- You will notice errors if you try to compile the provided skeleton files. Only after you fill the required structures up **accurately** will you be able to compile without any errors.
- Since you will be dealing with a lot of binary data, it is imperative to properly grasp the concept of **pointers** and **typecasting** to figure out how chunks of memory space can be filled up.
- Each of the metadata structures should have a corresponding C struct for organization and packing of data. Some essential **empty** structures have been declared in the skeleton file, but you should create more structures of your own for organization.
- In the metadata structure formats discussed, the fields are unsigned and of sizes 8, 16, 32 and 64 bits. We suggest using the types **uint8_t, uint16_t, uint32_t,** and **uint64_t** to store them respectively. The necessary headers are included in the skeleton .c file.
- Error handling is an essential part of the project. For incompatible starting states of your C program, it should gracefully return an error and exit, rather than exiting due to segmentation faults/other errors.
- As a starting point, we suggest carefully understanding the provided .c file. Some comments have been added for your convenience.
- To debug your output file, we suggest using hexdump or xxd.

Project Starting Point

You will be given the following items in a compressed (.zip) file as a starting point of your project:

- mkfs_builder.c skeleton
- mkfs_adder.c skeleton
- Files to be added to MiniVSFS

Download your assigned .zip file, extract, and start working on the skeleton files.

Project Deliverables

- Complete mkfs_builder.c
- Complete mkfs_adder.c

N.B. You DO NOT need to submit the sample files provided.

Evaluation

- You will be evaluated on the integrity of the file system generated by mkfs_builder, and on the integrity of the file system further generated by

mkfs_adder. You will also be evaluated on your understanding of the delivered code.

- Your code might be subjected to adversarial tests via incompatible CLI parameters and/or input files. Graceful exits from such cases is **mandatory**.

Project Mark Distribution

Task	Mark
CLI Parsing	5
mkfs_builder	15
mkfs_adder	15
Error Handling	5
Viva	60
Total	100