entire code: mkfs_builder.c

```
// Build: gcc -O2 -std=c17 -Wall -Wextra mkfs_minivsfs.c -o mkfs_builder
#define _FILE_OFFSET_BITS 64
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <inttypes.h>
#include <errno.h>
#include <time.h>
#include <assert.h>

#define BS 4096u              // block size
#define INODE_SIZE 128u
#define ROOT_INO 1u

uint64_t g_random_seed = 0; // This should be replaced by seed value from the CLI.

// below contains some basic structures you need for your project
// you are free to create more structures as you require

#pragma pack(push, 1)
typedef struct {
    uint32_t magic;               //32->4 bytes
    uint32_t version;
    uint32_t block_size;
    uint64_t total_blocks;        //64->8 bytes
    uint64_t inode_count;
    uint64_t inode_bitmap_start;
    uint64_t inode_bitmap_blocks;
    uint64_t data_bitmap_start;
    uint64_t data_bitmap_blocks;
    uint64_t inode_table_start;
    uint64_t inode_table_blocks;
    uint64_t data_region_start;
    uint64_t data_region_blocks;
    uint64_t root_inode;
    uint64_t mtime_epoch;         //last time(filesystem was mounted or written to)
    uint32_t flags;
    uint32_t checksum;            // crc32(superblock[0..4091])
} superblock_t;
#pragma pack(pop)
//_Static_assert(sizeof(superblock_t) == 116, "superblock must fit in one block");
```

```
//inode struct
#pragma pack(push, 1)
typedef struct {
    uint16_t mode;                  // 2 bytes - file mode (permissions/type)
    uint16_t links;                 // 2 bytes - file mode (permissions/type)
    uint32_t uid;                   // 4 bytes - owner user ID
    uint32_t gid;                   // 4 bytes - owner group ID
    uint64_t size_bytes;            // 8 bytes - file size in bytes
    uint64_t atime;                 // 8 bytes - last access time(file was last read or accessed, not
modified)
    uint64_t mtime;                 // 8 bytes - last modified time (contents of the file were last
changed)
    uint64_t ctime;                 // 8 bytes - last status change time(inode metadata was
changed)
    uint32_t direct_blocks[12];  // 12 * 4 = 48 bytes - direct data block pointers
    uint32_t reserved_0;          // 4 bytes - reserved
    uint32_t reserved_1;          // 4 bytes - reserved
    uint32_t reserved_2;          // 4 bytes - reserved
    uint32_t proj_id;             // 4 bytes
    uint32_t uid16_gid16;        // 4 bytes
    uint64_t xattr_ptr;           // 8 bytes
    uint64_t inode_crc;           // 8 bytes - low 4 bytes store crc32 of bytes [0..119]; high 4
bytes 0
} inode_t;
#pragma pack(pop)
_Static_assert(sizeof(inode_t)==INODE_SIZE, "inode size mismatch");
```

```
// Directory Entry struct
#pragma pack(push,1)
typedef struct {
    uint32_t inode_no;       // 4 bytes
    uint8_t type;               // 1 bytes
    uint8_t name[58];        // each char is 1 byte
    uint8_t  checksum;       // 1 bytes   - XOR of bytes 0..62
} dirent64_t;
#pragma pack(pop)
_Static_assert(sizeof(dirent64_t)==64, "dirent size mismatch");
```

```
// ==========================DO NOT CHANGE THIS
PORTION==========================
// These functions are there for your help. You should refer to the specifications to see how
you can use them.
//
====================================CRC32============================
========
uint32_t CRC32_TAB[256];
void crc32_init(void){
   for (uint32_t i=0;i<256;i++){
      uint32_t c=i;
      for(int j=0;j<8;j++) c = (c&1)?(0xEDB88320u^(c>>1)):(c>>1);
      CRC32_TAB[i]=c;
   }
}
uint32_t crc32(const void* data, size_t n){
   const uint8_t* p=(const uint8_t*)data; uint32_t c=0xFFFFFFFFu;
   for(size_t i=0;i<n;i++) c = CRC32_TAB[(c^p[i])&0xFF] ^ (c>>8);
   return c ^ 0xFFFFFFFFu;
}
//
====================================CRC32============================
========

// WARNING: CALL THIS ONLY AFTER ALL OTHER SUPERBLOCK ELEMENTS HAVE
BEEN FINALIZED
static uint32_t superblock_crc_finalize(superblock_t *sb) {
   sb->checksum = 0;
   uint32_t s = crc32((void *) sb, BS - 4);
   sb->checksum = s;
   return s;
}

// WARNING: CALL THIS ONLY AFTER ALL OTHER SUPERBLOCK ELEMENTS HAVE
BEEN FINALIZED
void inode_crc_finalize(inode_t* ino){
   uint8_t tmp[INODE_SIZE]; memcpy(tmp, ino, INODE_SIZE);
   // zero crc area before computing
   memset(&tmp[120], 0, 8);
   uint32_t c = crc32(tmp, 120);
   ino->inode_crc = (uint64_t)c; // low 4 bytes carry the crc
}

// WARNING: CALL THIS ONLY AFTER ALL OTHER SUPERBLOCK ELEMENTS HAVE
BEEN FINALIZED
void dirent_checksum_finalize(dirent64_t* de) {
   const uint8_t* p = (const uint8_t*)de;
   uint8_t x = 0;
```

```
    for (int i = 0; i < 63; i++) x ^= p[i];   // covers ino(4) + type(1) + name(58)
    de->checksum = x;
}

// int main()
int main(int argc, char *argv[]) {
    crc32_init();
    // WRITE YOUR DRIVER CODE HERE

    // PARSING CLI PARAMETERS
            char *image_name = NULL;
            uint64_t size_kib = 0;
uint64_t inode_count = 0;

// CLI parser
// ./mkfs_builder --image myfs.img --size-kib 1024 --inodes 128
image_name = argv[2];
size_kib = strtoull(argv[4], NULL, 10);
inode_count = strtoull(argv[6], NULL, 10);

// Validate inputs
if (!image_name || size_kib < 180 || size_kib > 4096 || inode_count < 128 || inode_count >
512) {
    printf("Invalid arguments.\n");
    return 1;
            }

            //SUPERBLOCK INITIALIZATION
            superblock_t sb;
memset(&sb, 0, sizeof(sb));  //for clearing garbage values in sb. Initialize all values with 0

sb.magic = 0x4D565346;
sb.version = 1;
sb.block_size = BS; //constant given in template
sb.total_blocks = (size_kib * 1024) / BS;
sb.inode_count = inode_count;                              //from CLI

sb.inode_bitmap_start = 1;                                 // block after superblock
sb.inode_bitmap_blocks = 1;                                // 1 inode bitmap block;

sb.data_bitmap_start = 2;          //block after inode bmap
sb.data_bitmap_blocks = 1; //1 block for data bmap

sb.inode_table_start = 3;
sb.inode_table_blocks = (inode_count * INODE_SIZE) / BS ;   //floor or celi value?

sb.data_region_start = sb.inode_table_start + sb.inode_table_blocks;
sb.data_region_blocks = sb.total_blocks - sb.data_region_start;
```

```
sb.root_inode = ROOT_INO;
sb.mtime_epoch = time(NULL);
sb.flags = 0;

// Compute superblock checksum
superblock_crc_finalize(&sb);

        //INODE INITIALIZATION
        inode_t *inode_table = malloc(inode_count* sizeof(inode_t));     //malloc or calloc ?
if (!inode_table) { perror("malloc error while inode_table allocation"); return 1; }

// Root inode (#1) = 1st inode = inode_table[0]
inode_table[0].mode = 0x4000;       // directory. For regular file use 0x8000
inode_table[0].links = 2;                  // . and ..
inode_table[0].size_bytes = 128;    // two entries, 64 B each
inode_table[0].atime = sb.mtime_epoch;
inode_table[0].mtime = sb.mtime_epoch;
inode_table[0].ctime = sb.mtime_epoch;
inode_table[0].direct_blocks[0] = 0;             // first data block of root, to be assigned later
inode_table[0].proj_id = 8;              //YOUR_GROUP_ID

// Compute root inode CRC
inode_crc_finalize(&inode_table[0]);

// Other inodes initialized to zero
for (int i = 1; i < inode_count; i++) {
   inode_table[i].proj_id = 8;                        //YOUR_GROUP_ID
   inode_crc_finalize(&inode_table[i]);
}

         // THEN CREATE YOUR FILE SYSTEM WITH A ROOT DIRECTORY========
//DIRECTORY INITIALIZATION
          dirent64_t root_entries[2] = {0};                  // . and ..

//CURRENT DIRECTORY INITIALIZATION

        //entry 0 -> .
        root_entries[0].inode_no = ROOT_INO;         // .
        root_entries[0].type = 2;                              // directory
       // ?? Strncpy is used instead of strcpy because your name field has a fixed size (58
bytes), and      this way you always fill/pad it properly for the filesystem's on-disk format.
Here for ".", 57 empty bytes will be set to null if strncpy is used

        strncpy((char*)root_entries[0].name, ".", 58); //name size = 58 bytes.
        dirent_checksum_finalize(&root_entries[0]);
```

```
        //PARENT DIRECTORY INITIALIZATION
        //entry 1 -> ..
        root_entries[1].inode_no = ROOT_INO;        // ..
root_entries[1].type = 2;
strncpy((char*)root_entries[1].name, "..", 58);
dirent_checksum_finalize(&root_entries[1]);


       //BITMAP INITIALIZATION

uint8_t inode_bitmap[BS] = {0};
inode_bitmap[0] = 1; // root inode = inode_table[0]= 1st inode booked

uint8_t data_bitmap[BS] = {0};
data_bitmap[0] = 1; // root data block allocated, i.e.data block 0 booked.



        // THEN SAVE THE DATA INSIDE THE OUTPUT IMAGE
FILE *fp = fopen(image_name, "wb");
if (!fp) { perror("fopen"); return 1; }

// Superblock bitmap (1 block)
//fwrite(ptr, size, count, fp). count=1 used bc the element in this block is the sb struct, , not just one field. fwrite treats the struct as a single contiguous block of memory.

fwrite(&sb, BS, 1, fp);


//INODE bitmap (1 block)
fwrite(inode_bitmap, BS, 1, fp);

//data bitmap (1 block)
fwrite(data_bitmap, BS, 1, fp);

//INODE TABLE
//Write inode_count elements, each of size INODE_SIZE, starting from inode_table
fwrite(inode_table, INODE_SIZE, inode_count, fp);

// Pad inode table to full blocks
size_t inode_table_size = inode_count * INODE_SIZE;
size_t inode_table_padding = sb.inode_table_blocks * BS - inode_table_size;
uint8_t zero[BS] = {0};
fwrite(zero, inode_table_padding, 1, fp);

// Root directory data block
fwrite(root_entries, sizeof(root_entries), 1, fp);
```

```
// Pad remaining data region
size_t data_region_size = (sb.data_region_blocks - 1) * BS; // first block used by root
fwrite(zero, data_region_size, 1, fp);

fclose(fp);

//The .img file is persistent on disk, so freeing the RAM buffer does not affect it.
free(inode_table);




 return 0;
}
```