Fs.c

```
#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt

#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/module.h>

#include "myfs.h"

/* Mount a myfs partition */
struct dentry *myfs_mount(struct file_system_type *fs_type,
                              int flags,
                              const char *dev_name,
                              void *data)
{
    struct dentry *dentry =
        mount_bdev(fs_type, flags, dev_name, data, myfs_fill_super);
    if (IS_ERR(dentry))
        pr_err("'%s' mount failure\n", dev_name);
    else
        pr_info("'%s' mount success\n", dev_name);

    return dentry;
}

/* Unmount a myfs partition */
void myfs_kill_sb(struct super_block *sb)
{
    kill_block_super(sb);

    pr_info("unmounted disk\n");
}

static struct file_system_type myfs_file_system_type = {
    .owner = THIS_MODULE,
    .name = "myfs",
    .mount = myfs_mount,
    .kill_sb = myfs_kill_sb,
    .fs_flags = FS_REQUIRES_DEV,
    .next = NULL,
};

static int __init myfs_init(void)
{
    int ret = myfs_init_inode_cache();
    if (ret) {
        pr_err("inode cache creation failed\n");
        goto end;
    }

    ret = register_filesystem(&myfs_file_system_type);
    if (ret) {
        pr_err("register_filesystem() failed\n");
        goto end;
    }
    pr_info("module loaded\n");
end:
    return ret;
}

static void __exit myfs_exit(void)
{
    int ret = unregister_filesystem(&myfs_file_system_type);
    if (ret)
        pr_err("unregister_filesystem() failed\n");

    myfs_destroy_inode_cache();

    pr_info("module unloaded\n");
}

module_init(myfs_init);
module_exit(myfs_exit);

MODULE_LICENSE("Dual BSD/GPL");
```

Inode.c

```
#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt

#include <linux/buffer_head.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/module.h>

#include "bitmap.h"
#include "myfs.h"

static const struct inode_operations myfs_inode_ops;
static const struct inode_operations symlink_inode_ops;

/* Get inode ino from disk */
struct inode *myfs_iget(struct super_block *sb, unsigned long ino)
{
    struct inode *inode = NULL;
    struct myfs_inode *cinode = NULL;
    struct myfs_inode_info *ci = NULL;
    struct myfs_sb_info *sbi = MYFS_SB(sb);
    struct buffer_head *bh = NULL;
    uint32_t inode_block = (ino / MYFS_INODES_PER_BLOCK) + 1;
    uint32_t inode_shift = ino % MYFS_INODES_PER_BLOCK;
    int ret;

    /* Fail if ino is out of range */
    if (ino >= sbi->nr_inodes)
        return ERR_PTR(-EINVAL);

    /* Get a locked inode from Linux */
    inode = iget_locked(sb, ino);
    if (!inode)
        return ERR_PTR(-ENOMEM);

    /* If inode is in cache, return it */
    if (!(inode->i_state & I_NEW))
        return inode;

    ci = MYFS_INODE(inode);
    /* Read inode from disk and initialize */
    bh = sb_bread(sb, inode_block);
    if (!bh) {
        ret = -EIO;
        goto failed;
    }
    cinode = (struct myfs_inode *) bh->b_data;
    cinode += inode_shift;

    inode->i_ino = ino;
    inode->i_sb = sb;
    inode->i_op = &myfs_inode_ops;

    inode->i_mode = le32_to_cpu(cinode->i_mode);
    i_uid_write(inode, le32_to_cpu(cinode->i_uid));
    i_gid_write(inode, le32_to_cpu(cinode->i_gid));
    inode->i_size = le32_to_cpu(cinode->i_size);
    inode->i_ctime.tv_sec = (time64_t) le32_to_cpu(cinode->i_ctime);
    inode->i_ctime.tv_nsec = 0;
    inode->i_atime.tv_sec = (time64_t) le32_to_cpu(cinode->i_atime);
    inode->i_atime.tv_nsec = 0;
    inode->i_mtime.tv_sec = (time64_t) le32_to_cpu(cinode->i_mtime);
    inode->i_mtime.tv_nsec = 0;
    inode->i_blocks = le32_to_cpu(cinode->i_blocks);
    set_nlink(inode, le32_to_cpu(cinode->i_nlink));

    if (S_ISDIR(inode->i_mode)) {
        ci->dir_block = le32_to_cpu(cinode->dir_block);
        inode->i_fop = &myfs_dir_ops;
    } else if (S_ISREG(inode->i_mode)) {
        ci->ei_block = le32_to_cpu(cinode->ei_block);
        inode->i_fop = &myfs_file_ops;
        inode->i_mapping->a_ops = &myfs_aops;
    } else if (S_ISLNK(inode->i_mode)) {
        strncpy(ci->i_data, cinode->i_data, sizeof(ci->i_data));
        inode->i_link = ci->i_data;
        inode->i_op = &symlink_inode_ops;
    }

    brelse(bh);

    /* Unlock the inode to make it usable */
    unlock_new_inode(inode);

    return inode;

failed:
    brelse(bh);
    iget_failed(inode);
    return ERR_PTR(ret);
}

/*
 * Look for dentry in dir.
 * Fill dentry with NULL if not in dir, with the corresponding inode if
found.
 * Returns NULL on success.
 */
static struct dentry *myfs_lookup(struct inode *dir,
                                      struct dentry *dentry,
                                      unsigned int flags)
{
    struct super_block *sb = dir->i_sb;
    struct myfs_inode_info *ci_dir = MYFS_INODE(dir);
    struct inode *inode = NULL;
    struct buffer_head *bh = NULL;
    struct myfs_dir_block *dblock = NULL;
    struct myfs_file *f = NULL;
    int i;

    /* Check filename length */
    if (dentry->d_name.len > MYFS_FILENAME_LEN)
        return ERR_PTR(-ENAMETOOLONG);

    /* Read the directory block on disk */
    bh = sb_bread(sb, ci_dir->dir_block);
    if (!bh)
        return ERR_PTR(-EIO);
    dblock = (struct myfs_dir_block *) bh->b_data;

    /* Search for the file in directory */
    for (i = 0; i < MYFS_MAX_SUBFILES; i++) {
        f = &dblock->files[i];
        if (!f->inode)
            break;
        if (!strncmp(f->filename, dentry->d_name.name, MYFS_FILENAME_LEN)) {
            inode = myfs_iget(sb, f->inode);
            break;
        }
    }
    brelse(bh);

    /* Update directory access time */
    dir->i_atime = current_time(dir);
    mark_inode_dirty(dir);

    /* Fill the dentry with the inode */
    d_add(dentry, inode);

    return NULL;
}

/* Create a new inode in dir */
static struct inode *myfs_new_inode(struct inode *dir, mode_t mode)
{
    struct inode *inode;
    struct myfs_inode_info *ci;
    struct super_block *sb;
    struct myfs_sb_info *sbi;
    uint32_t ino, bno;
    int ret;

    /* Check mode before doing anything to avoid undoing everything */
    if (!S_ISDIR(mode) && !S_ISREG(mode) && !S_ISLNK(mode)) {
        pr_err(
            "File type not supported (only directory, regular file and
symlink "
            "supported)\n");
        return ERR_PTR(-EINVAL);
    }

    /* Check if inodes are available */
    sb = dir->i_sb;
    sbi = MYFS_SB(sb);
    if (sbi->nr_free_inodes == 0 || sbi->nr_free_blocks == 0)
        return ERR_PTR(-ENOSPC);

    /* Get a new free inode */
    ino = get_free_inode(sbi);
    if (!ino)
        return ERR_PTR(-ENOSPC);

    inode = myfs_iget(sb, ino);
    if (IS_ERR(inode)) {
        ret = PTR_ERR(inode);
        goto put_ino;
    }

    if (S_ISLNK(mode)) {
        inode_init_owner(inode, dir, mode);
        set_nlink(inode, 1);
        inode->i_ctime = inode->i_atime = inode->i_mtime =
current_time(inode);
        inode->i_op = &symlink_inode_ops;
        return inode;
    }

    ci = MYFS_INODE(inode);

    /* Get a free block for this new inode's index */
    bno = get_free_blocks(sbi, 1);
    if (!bno) {
        ret = -ENOSPC;
        goto put_inode;
    }

    /* Initialize inode */
    inode_init_owner(inode, dir, mode);
    inode->i_blocks = 1;
    if (S_ISDIR(mode)) {
        ci->dir_block = bno;
        inode->i_size = MYFS_BLOCK_SIZE;
        inode->i_fop = &myfs_dir_ops;
        set_nlink(inode, 2); /* . and .. */
    } else if (S_ISREG(mode)) {
        ci->ei_block = bno;
        inode->i_size = 0;
        inode->i_fop = &myfs_file_ops;
        inode->i_mapping->a_ops = &myfs_aops;
        set_nlink(inode, 1);
    }

    inode->i_ctime = inode->i_atime = inode->i_mtime = current_time(inode);

    return inode;

put_inode:
    iput(inode);
put_ino:
    put_inode(sbi, ino);

    return ERR_PTR(ret);
}

/*
```

```
 * Create a file or directory in this way:
 *   - check filename length and if the parent directory is not full
 *   - create the new inode (allocate inode and blocks)
 *   - cleanup index block of the new inode
 *   - add new file/directory in parent index
 */
static int myfs_create(struct inode *dir,
                           struct dentry *dentry,
                           umode_t mode,
                           bool excl)
{
    struct super_block *sb;
    struct inode *inode;
    struct myfs_inode_info *ci_dir;
    struct myfs_dir_block *dblock;
    char *fblock;
    struct buffer_head *bh, *bh2;
    int ret = 0, i;

    /* Check filename length */
    if (strlen(dentry->d_name.name) > MYFS_FILENAME_LEN)
        return -ENAMETOOLONG;

    /* Read parent directory index */
    ci_dir = MYFS_INODE(dir);
    sb = dir->i_sb;
    bh = sb_bread(sb, ci_dir->dir_block);
    if (!bh)
        return -EIO;

    dblock = (struct myfs_dir_block *) bh->b_data;

    /* Check if parent directory is full */
    if (dblock->files[MYFS_MAX_SUBFILES - 1].inode != 0) {
        ret = -EMLINK;
        goto end;
    }

    /* Get a new free inode */
    inode = myfs_new_inode(dir, mode);
    if (IS_ERR(inode)) {
        ret = PTR_ERR(inode);
        goto end;
    }

    /*
     * Scrub ei_block/dir_block for new file/directory to avoid previous data
     * messing with new file/directory.
     */
    bh2 = sb_bread(sb, MYFS_INODE(inode)->ei_block);
    if (!bh2) {
        ret = -EIO;
        goto iput;
    }
    fblock = (char *) bh2->b_data;
    memset(fblock, 0, MYFS_BLOCK_SIZE);
    mark_buffer_dirty(bh2);
    brelse(bh2);

    /* Find first free slot in parent index and register new inode */
    for (i = 0; i < MYFS_MAX_SUBFILES; i++)
        if (dblock->files[i].inode == 0)
            break;
    dblock->files[i].inode = inode->i_ino;
    strncpy(dblock->files[i].filename, dentry->d_name.name,
            MYFS_FILENAME_LEN);
    mark_buffer_dirty(bh);
    brelse(bh);

    /* Update stats and mark dir and new inode dirty */
    mark_inode_dirty(inode);
    dir->i_mtime = dir->i_atime = dir->i_ctime = current_time(dir);
    if (S_ISDIR(mode))
        inc_nlink(dir);
    mark_inode_dirty(dir);

    /* setup dentry */
    d_instantiate(dentry, inode);

    return 0;

iput:
    put_blocks(MYFS_SB(sb), MYFS_INODE(inode)->ei_block, 1);
    put_inode(MYFS_SB(sb), inode->i_ino);
    iput(inode);
end:
    brelse(bh);
    return ret;
}

/*
 * Remove a link for a file including the reference in the parent directory.
 * If link count is 0, destroy file in this way:
 *   - remove the file from its parent directory.
 *   - cleanup blocks containing data
 *   - cleanup file index block
 *   - cleanup inode
 */
static int myfs_unlink(struct inode *dir, struct dentry *dentry)
{
    struct super_block *sb = dir->i_sb;
    struct myfs_sb_info *sbi = MYFS_SB(sb);
    struct inode *inode = d_inode(dentry);
    struct buffer_head *bh = NULL, *bh2 = NULL;
    struct myfs_dir_block *dir_block = NULL;
    struct myfs_file_ei_block *file_block = NULL;
    int i, j, f_id = -1, nr_subs = 0;

    uint32_t ino = inode->i_ino;
    uint32_t bno = 0;

    /* Read parent directory index */
    bh = sb_bread(sb, MYFS_INODE(dir)->dir_block);
    if (!bh)
        return -EIO;
    dir_block = (struct myfs_dir_block *) bh->b_data;

    /* Search for inode in parent index and get number of subfiles */
    for (i = 0; i < MYFS_MAX_SUBFILES; i++) {
        if (strncmp(dir_block->files[i].filename, dentry->d_name.name,
                    MYFS_FILENAME_LEN) == 0)
            f_id = i;
        else if (dir_block->files[i].inode == 0)
            break;
    }
    nr_subs = i;

    /* Remove file from parent directory */
    if (f_id != MYFS_MAX_SUBFILES - 1)
        memmove(dir_block->files + f_id, dir_block->files + f_id + 1,
                (nr_subs - f_id - 1) * sizeof(struct myfs_file));
    memset(&dir_block->files[nr_subs - 1], 0, sizeof(struct myfs_file));
    mark_buffer_dirty(bh);
    brelse(bh);

    if (S_ISLNK(inode->i_mode))
        goto clean_inode;

    /* Update inode stats */
    dir->i_mtime = dir->i_atime = dir->i_ctime = current_time(dir);
    if (S_ISDIR(inode->i_mode)) {
        drop_nlink(dir);
        drop_nlink(inode);
    }
    mark_inode_dirty(dir);

    if (inode->i_nlink > 1) {
        inode_dec_link_count(inode);
        return 0;
    }

    /*
     * Cleanup pointed blocks if unlinking a file. If we fail to read the
     * index block, cleanup inode anyway and lose this file's blocks
     * forever. If we fail to scrub a data block, don't fail (too late
     * anyway), just put the block and continue.
     */
    bno = MYFS_INODE(inode)->ei_block;
    bh = sb_bread(sb, bno);
    if (!bh)
        goto clean_inode;
    file_block = (struct myfs_file_ei_block *) bh->b_data;
    if (S_ISDIR(inode->i_mode))
        goto scrub;
    for (i = 0; i < MYFS_MAX_EXTENTS; i++) {
        char *block;

        if (!file_block->extents[i].ee_start)
            break;

        put_blocks(sbi, file_block->extents[i].ee_start,
                   file_block->extents[i].ee_len);

        /* Scrub the extent */
        for (j = 0; j < file_block->extents[i].ee_len; j++) {
            bh2 = sb_bread(sb, file_block->extents[i].ee_start + j);
            if (!bh2)
                continue;
            block = (char *) bh2->b_data;
            memset(block, 0, MYFS_BLOCK_SIZE);
            mark_buffer_dirty(bh2);
            brelse(bh2);
        }
    }

scrub:
    /* Scrub index block */
    memset(file_block, 0, MYFS_BLOCK_SIZE);
    mark_buffer_dirty(bh);
    brelse(bh);

clean_inode:
    /* Cleanup inode and mark dirty */
    inode->i_blocks = 0;
    MYFS_INODE(inode)->ei_block = 0;
    inode->i_size = 0;
    i_uid_write(inode, 0);
    i_gid_write(inode, 0);
    inode->i_mode = 0;
    inode->i_ctime.tv_sec = inode->i_mtime.tv_sec = inode->i_atime.tv_sec =
0;
    drop_nlink(inode);
    mark_inode_dirty(inode);

    /* Free inode and index block from bitmap */
    put_blocks(sbi, bno, 1);
    put_inode(sbi, ino);

    return 0;
}

static int myfs_rename(struct inode *old_dir,
                           struct dentry *old_dentry,
                           struct inode *new_dir,
                           struct dentry *new_dentry,
                           unsigned int flags)
{
    struct super_block *sb = old_dir->i_sb;
    struct myfs_inode_info *ci_old = MYFS_INODE(old_dir);
    struct myfs_inode_info *ci_new = MYFS_INODE(new_dir);
    struct inode *src = d_inode(old_dentry);
    struct buffer_head *bh_old = NULL, *bh_new = NULL;
    struct myfs_dir_block *dir_block = NULL;
    int i, f_id = -1, new_pos = -1, ret, nr_subs, f_pos = -1;

    /* fail with these unsupported flags */
    if (flags & (RENAME_EXCHANGE | RENAME_WHITEOUT))
        return -EINVAL;
    /* Check if filename is not too long */
    if (strlen(new_dentry->d_name.name) > MYFS_FILENAME_LEN)
        return -ENAMETOOLONG;

    /* Fail if new_dentry exists or if new_dir is full */
    bh_new = sb_bread(sb, ci_new->dir_block);
    if (!bh_new)
        return -EIO;
    dir_block = (struct myfs_dir_block *) bh_new->b_data;
    for (i = 0; i < MYFS_MAX_SUBFILES; i++) {
        /* if old_dir == new_dir, save the renamed file position */
        if (new_dir == old_dir) {
            if (strncmp(dir_block->files[i].filename, old_dentry-
>d_name.name,
                        MYFS_FILENAME_LEN) == 0)
                f_pos = i;
        }
        if (strncmp(dir_block->files[i].filename, new_dentry->d_name.name,
                    MYFS_FILENAME_LEN) == 0) {
            ret = -EEXIST;
            goto relse_new;
        }
        if (new_pos < 0 && dir_block->files[i].inode == 0)
            new_pos = i;
    }
    /* if old_dir == new_dir, just rename entry */
    if (old_dir == new_dir) {
        strncpy(dir_block->files[f_pos].filename, new_dentry->d_name.name,
                MYFS_FILENAME_LEN);
        mark_buffer_dirty(bh_new);
        ret = 0;
        goto relse_new;
    }

    /* If new directory is empty, fail */
    if (new_pos < 0) {
        ret = -EMLINK;
        goto relse_new;
    }

    /* insert in new parent directory */
    dir_block->files[new_pos].inode = src->i_ino;
    strncpy(dir_block->files[new_pos].filename, new_dentry->d_name.name,
            MYFS_FILENAME_LEN);
    mark_buffer_dirty(bh_new);
    brelse(bh_new);

    /* Update new parent inode metadata */
    new_dir->i_atime = new_dir->i_ctime = new_dir->i_mtime =
        current_time(new_dir);
    if (S_ISDIR(src->i_mode))
        inc_nlink(new_dir);
    mark_inode_dirty(new_dir);

    /* remove target from old parent directory */
    bh_old = sb_bread(sb, ci_old->dir_block);
    if (!bh_old)
        return -EIO;
    dir_block = (struct myfs_dir_block *) bh_old->b_data;
    /* Search for inode in old directory and number of subfiles */
    for (i = 0; MYFS_MAX_SUBFILES; i++) {
        if (dir_block->files[i].inode == src->i_ino)
            f_id = i;
        else if (dir_block->files[i].inode == 0)
            break;
    }
    nr_subs = i;

    /* Remove file from old parent directory */
    if (f_id != MYFS_MAX_SUBFILES - 1)
        memmove(dir_block->files + f_id, dir_block->files + f_id + 1,
                (nr_subs - f_id - 1) * sizeof(struct myfs_file));
    memset(&dir_block->files[nr_subs - 1], 0, sizeof(struct myfs_file));
    mark_buffer_dirty(bh_old);
    brelse(bh_old);

    /* Update old parent inode metadata */
    old_dir->i_atime = old_dir->i_ctime = old_dir->i_mtime =
        current_time(old_dir);
    if (S_ISDIR(src->i_mode))
        drop_nlink(old_dir);
    mark_inode_dirty(old_dir);

    return 0;

relse_new:
    brelse(bh_new);
    return ret;
}

static int myfs_mkdir(struct inode *dir,
                          struct dentry *dentry,
                          umode_t mode)
{
    return myfs_create(dir, dentry, mode | S_IFDIR, 0);
}

static int myfs_rmdir(struct inode *dir, struct dentry *dentry)
{
    struct super_block *sb = dir->i_sb;
    struct inode *inode = d_inode(dentry);
    struct buffer_head *bh;
    struct myfs_dir_block *dblock;

    /* If the directory is not empty, fail */
    if (inode->i_nlink > 2)
        return -ENOTEMPTY;
    bh = sb_bread(sb, MYFS_INODE(inode)->dir_block);
    if (!bh)
        return -EIO;
    dblock = (struct myfs_dir_block *) bh->b_data;
    if (dblock->files[0].inode != 0) {
        brelse(bh);
        return -ENOTEMPTY;
    }
    brelse(bh);

    /* Remove directory with unlink */
    return myfs_unlink(dir, dentry);
}

static int myfs_link(struct dentry *old_dentry,
                         struct inode *dir,
                         struct dentry *dentry)
{
    struct inode *inode = d_inode(old_dentry);
    struct super_block *sb = inode->i_sb;
    struct myfs_inode_info *ci_dir = MYFS_INODE(dir);
    struct myfs_dir_block *dir_block;
    struct buffer_head *bh;
    int f_pos = -1, ret = 0, i = 0;

    bh = sb_bread(sb, ci_dir->dir_block);
    if (!bh)
        return -EIO;
    dir_block = (struct myfs_dir_block *) bh->b_data;

    if (dir_block->files[MYFS_MAX_SUBFILES - 1].inode != 0) {
        ret = -EMLINK;
        printk(KERN_INFO "directory is full");
        goto end;
    }

    for (i = 0; i < MYFS_MAX_SUBFILES; i++) {
        if (dir_block->files[i].inode == 0) {
            f_pos = i;
            break;
        }
    }

    dir_block->files[f_pos].inode = inode->i_ino;
    strncpy(dir_block->files[f_pos].filename, dentry->d_name.name,
            MYFS_FILENAME_LEN);
    mark_buffer_dirty(bh);

    inode_inc_link_count(inode);
    d_instantiate(dentry, inode);
end:
    brelse(bh);
    return ret;
}

static int myfs_symlink(struct inode *dir,
                            struct dentry *dentry,
                            const char *symname)
{
    struct super_block *sb = dir->i_sb;
    unsigned int l = strlen(symname) + 1;
    struct inode *inode = myfs_new_inode(dir, S_IFLNK | S_IRWXUGO);
    struct myfs_inode_info *ci = MYFS_INODE(inode);
    struct myfs_inode_info *ci_dir = MYFS_INODE(dir);
    struct myfs_dir_block *dir_block;
    struct buffer_head *bh;
    int f_pos = 0, i = 0;

    /* Check if symlink content is not too long */
    if (l > sizeof(ci->i_data))
        return -ENAMETOOLONG;

    /* fill directory data block */
    bh = sb_bread(sb, ci_dir->dir_block);

    if (!bh)
        return -EIO;
    dir_block = (struct myfs_dir_block *) bh->b_data;

    if (dir_block->files[MYFS_MAX_SUBFILES - 1].inode != 0) {
        printk(KERN_INFO "directory is full\n");
        return -EMLINK;
    }

    for (i = 0; i < MYFS_MAX_SUBFILES; i++) {
        if (dir_block->files[i].inode == 0) {
            f_pos = i;
            break;
        }
    }

    dir_block->files[f_pos].inode = inode->i_ino;
    strncpy(dir_block->files[f_pos].filename, dentry->d_name.name,
            MYFS_FILENAME_LEN);
    mark_buffer_dirty(bh);
    brelse(bh);

    inode->i_link = (char *) ci->i_data;
    memcpy(inode->i_link, symname, l);
    inode->i_size = l - 1;
    mark_inode_dirty(inode);
    d_instantiate(dentry, inode);

    return 0;
}

static const char *myfs_get_link(struct dentry *dentry,
                                     struct inode *inode,
                                     struct delayed_call *done)
{
    return inode->i_link;
}

static const struct inode_operations myfs_inode_ops = {
    .lookup = myfs_lookup,
    .create = myfs_create,
    .unlink = myfs_unlink,
    .mkdir = myfs_mkdir,
    .rmdir = myfs_rmdir,
    .rename = myfs_rename,
    .link = myfs_link,
    .symlink = myfs_symlink,
};

static const struct inode_operations symlink_inode_ops = {
    .get_link = myfs_get_link,
};
```

Super.c

```
#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt

#include <linux/buffer_head.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/statfs.h>

#include "myfs.h"

static struct kmem_cache *myfs_inode_cache;

int myfs_init_inode_cache(void)
{
    myfs_inode_cache = kmem_cache_create(
        "myfs_cache", sizeof(struct myfs_inode_info), 0, 0, NULL);
    if (!myfs_inode_cache)
        return -ENOMEM;
    return 0;
}

void myfs_destroy_inode_cache(void)
{
    kmem_cache_destroy(myfs_inode_cache);
}

static struct inode *myfs_alloc_inode(struct super_block *sb)
{
    struct myfs_inode_info *ci =
        kmem_cache_alloc(myfs_inode_cache, GFP_KERNEL);
    if (!ci)
        return NULL;

    inode_init_once(&ci->vfs_inode);
    return &ci->vfs_inode;
}

static void myfs_destroy_inode(struct inode *inode)
{
    struct myfs_inode_info *ci = MYFS_INODE(inode);
    kmem_cache_free(myfs_inode_cache, ci);
}

static int myfs_write_inode(struct inode *inode,
                                struct writeback_control *wbc)
{
    struct myfs_inode *disk_inode;
    struct myfs_inode_info *ci = MYFS_INODE(inode);
    struct super_block *sb = inode->i_sb;
    struct myfs_sb_info *sbi = MYFS_SB(sb);
    struct buffer_head *bh;
    uint32_t ino = inode->i_ino;
    uint32_t inode_block = (ino / MYFS_INODES_PER_BLOCK) + 1;
    uint32_t inode_shift = ino % MYFS_INODES_PER_BLOCK;
    if (ino >= sbi->nr_inodes)
        return 0;

    bh = sb_bread(sb, inode_block);
    if (!bh)
        return -EIO;

    disk_inode = (struct myfs_inode *) bh->b_data;
    disk_inode += inode_shift;

    /* update the mode using what the generic inode has */
    disk_inode->i_mode = inode->i_mode;
    disk_inode->i_uid = i_uid_read(inode);
    disk_inode->i_gid = i_gid_read(inode);
    disk_inode->i_size = inode->i_size;
    disk_inode->i_ctime = inode->i_ctime.tv_sec;
    disk_inode->i_atime = inode->i_atime.tv_sec;
    disk_inode->i_mtime = inode->i_mtime.tv_sec;
    disk_inode->i_blocks = inode->i_blocks;
    disk_inode->i_nlink = inode->i_nlink;
    disk_inode->ei_block = ci->ei_block;
    strncpy(disk_inode->i_data, ci->i_data, sizeof(ci->i_data));

    mark_buffer_dirty(bh);
    sync_dirty_buffer(bh);
    brelse(bh);

    return 0;
}

static void myfs_put_super(struct super_block *sb)
{
    struct myfs_sb_info *sbi = MYFS_SB(sb);
    if (sbi) {
        kfree(sbi->ifree_bitmap);
        kfree(sbi->bfree_bitmap);
        kfree(sbi);
    }
}

static int myfs_sync_fs(struct super_block *sb, int wait)
{
    struct myfs_sb_info *sbi = MYFS_SB(sb);
    struct myfs_sb_info *disk_sb;
    int i;

    /* Flush superblock */
    struct buffer_head *bh = sb_bread(sb, 0);
    if (!bh)
        return -EIO;

    disk_sb = (struct myfs_sb_info *) bh->b_data;

    disk_sb->nr_blocks = sbi->nr_blocks;
    disk_sb->nr_inodes = sbi->nr_inodes;
    disk_sb->nr_istore_blocks = sbi->nr_istore_blocks;
    disk_sb->nr_ifree_blocks = sbi->nr_ifree_blocks;
    disk_sb->nr_bfree_blocks = sbi->nr_bfree_blocks;
    disk_sb->nr_free_inodes = sbi->nr_free_inodes;
    disk_sb->nr_free_blocks = sbi->nr_free_blocks;

    mark_buffer_dirty(bh);
    if (wait)
        sync_dirty_buffer(bh);
    brelse(bh);

    /* Flush free inodes bitmask */
    for (i = 0; i < sbi->nr_ifree_blocks; i++) {
        int idx = sbi->nr_istore_blocks + i + 1;

        bh = sb_bread(sb, idx);
        if (!bh)
            return -EIO;

        memcpy(bh->b_data, (void *) sbi->ifree_bitmap + i * MYFS_BLOCK_SIZE,
               MYFS_BLOCK_SIZE);

        mark_buffer_dirty(bh);
        if (wait)
            sync_dirty_buffer(bh);
        brelse(bh);
    }

    /* Flush free blocks bitmask */
    for (i = 0; i < sbi->nr_bfree_blocks; i++) {
        int idx = sbi->nr_istore_blocks + sbi->nr_ifree_blocks + i + 1;

        bh = sb_bread(sb, idx);
        if (!bh)
            return -EIO;

        memcpy(bh->b_data, (void *) sbi->bfree_bitmap + i * MYFS_BLOCK_SIZE,
               MYFS_BLOCK_SIZE);

        mark_buffer_dirty(bh);
        if (wait)
            sync_dirty_buffer(bh);
        brelse(bh);
    }

    return 0;
}

static int myfs_statfs(struct dentry *dentry, struct kstatfs *stat)
{
    struct super_block *sb = dentry->d_sb;
    struct myfs_sb_info *sbi = MYFS_SB(sb);

    stat->f_type = MYFS_MAGIC;
    stat->f_bsize = MYFS_BLOCK_SIZE;
    stat->f_blocks = sbi->nr_blocks;
    stat->f_bfree = sbi->nr_free_blocks;
    stat->f_bavail = sbi->nr_free_blocks;
    stat->f_files = sbi->nr_inodes - sbi->nr_free_inodes;
    stat->f_ffree = sbi->nr_free_inodes;
    stat->f_namelen = MYFS_FILENAME_LEN;

    return 0;
}

static struct super_operations myfs_super_ops = {
    .put_super = myfs_put_super,
    .alloc_inode = myfs_alloc_inode,
    .destroy_inode = myfs_destroy_inode,
    .write_inode = myfs_write_inode,
    .sync_fs = myfs_sync_fs,
    .statfs = myfs_statfs,
};

/* Fill the struct superblock from partition superblock */
int myfs_fill_super(struct super_block *sb, void *data, int silent)
{
    struct buffer_head *bh = NULL;
    struct myfs_sb_info *csb = NULL;
    struct myfs_sb_info *sbi = NULL;
    struct inode *root_inode = NULL;
    int ret = 0, i;

    /* Init sb */
    sb->s_magic = MYFS_MAGIC;
    sb_set_blocksize(sb, MYFS_BLOCK_SIZE);
    sb->s_maxbytes = MYFS_MAX_FILESIZE;
    sb->s_op = &myfs_super_ops;

    /* Read sb from disk */
    bh = sb_bread(sb, MYFS_SB_BLOCK_NR);
    if (!bh)
        return -EIO;

    csb = (struct myfs_sb_info *) bh->b_data;

    /* Check magic number */
    if (csb->magic != sb->s_magic) {
        pr_err("Wrong magic number\n");
        ret = -EINVAL;
        goto release;
    }

    /* Alloc sb_info */
    sbi = kzalloc(sizeof(struct myfs_sb_info), GFP_KERNEL);
    if (!sbi) {
        ret = -ENOMEM;
        goto release;
    }

    sbi->nr_blocks = csb->nr_blocks;
    sbi->nr_inodes = csb->nr_inodes;
    sbi->nr_istore_blocks = csb->nr_istore_blocks;
    sbi->nr_ifree_blocks = csb->nr_ifree_blocks;
    sbi->nr_bfree_blocks = csb->nr_bfree_blocks;
```

```
sbi->nr_free_inodes = csb->nr_free_inodes;
sbi->nr_free_blocks = csb->nr_free_blocks;
sb->s_fs_info = sbi;

brelse(bh);

/* Alloc and copy ifree_bitmap */
sbi->ifree_bitmap =
    kzalloc(sbi->nr_ifree_blocks * MYFS_BLOCK_SIZE, GFP_KERNEL);
if (!sbi->ifree_bitmap) {
    ret = -ENOMEM;
    goto free_sbi;
}

for (i = 0; i < sbi->nr_ifree_blocks; i++) {
    int idx = sbi->nr_istore_blocks + i + 1;

    bh = sb_bread(sb, idx);
    if (!bh) {
        ret = -EIO;
        goto free_ifree;
    }

    memcpy((void *) sbi->ifree_bitmap + i * MYFS_BLOCK_SIZE, bh->b_data,
           MYFS_BLOCK_SIZE);

    brelse(bh);
}

/* Alloc and copy bfree_bitmap */
sbi->bfree_bitmap =
    kzalloc(sbi->nr_bfree_blocks * MYFS_BLOCK_SIZE, GFP_KERNEL);
if (!sbi->bfree_bitmap) {
    ret = -ENOMEM;
    goto free_ifree;
}

for (i = 0; i < sbi->nr_bfree_blocks; i++) {
    int idx = sbi->nr_istore_blocks + sbi->nr_ifree_blocks + i + 1;

    bh = sb_bread(sb, idx);
    if (!bh) {
        ret = -EIO;
        goto free_bfree;
    }

    memcpy((void *) sbi->bfree_bitmap + i * MYFS_BLOCK_SIZE, bh->b_data,
           MYFS_BLOCK_SIZE);

    brelse(bh);
}

/* Create root inode */
root_inode = myfs_iget(sb, 0);
if (IS_ERR(root_inode)) {
    ret = PTR_ERR(root_inode);
    goto free_bfree;
    }
    inode_init_owner(root_inode, NULL, root_inode->i_mode);
    sb->s_root = d_make_root(root_inode);
    if (!sb->s_root) {
        ret = -ENOMEM;
        goto iput;
    }

    return 0;

iput:
    iput(root_inode);
free_bfree:
    kfree(sbi->bfree_bitmap);
free_ifree:
    kfree(sbi->ifree_bitmap);
free_sbi:
    kfree(sbi);
release:
    brelse(bh);

    return ret;
}
```

File.c

```
#define pr_fmt(fmt) "myfs: " fmt

#include <linux/buffer_head.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/mpage.h>

#include "bitmap.h"
#include "myfs.h"

/*
 * Map the buffer_head passed in argument with the iblock-th block of the
file
 * represented by inode. If the requested block is not allocated and create
is
 * true,  allocate a new block on disk and map it.
 */
static int myfs_file_get_block(struct inode *inode,
                                   sector_t iblock,
                                   struct buffer_head *bh_result,
                                   int create)
{
    struct super_block *sb = inode->i_sb;
    struct myfs_sb_info *sbi = MYFS_SB(sb);
    struct myfs_inode_info *ci = MYFS_INODE(inode);
    struct myfs_file_ei_block *index;
    struct buffer_head *bh_index;
    bool alloc = false;
    int ret = 0, bno;
    uint32_t extent;

    /* If block number exceeds filesize, fail */
    if (iblock >= MYFS_MAX_BLOCKS_PER_EXTENT * MYFS_MAX_EXTENTS)
        return -EFBIG;

    /* Read directory block from disk */
    bh_index = sb_bread(sb, ci->dir_block);
    if (!bh_index)
        return -EIO;
    index = (struct myfs_file_ei_block *) bh_index->b_data;

    extent = myfs_ext_search(index, iblock);
    if (extent == -1) {
        ret = -EFBIG;
        goto brelse_index;
    }

    /*
     * Check if iblock is already allocated. If not and create is true,
     * allocate it. Else, get the physical block number.
     */
    if (index->extents[extent].ee_start == 0) {
        if (!create)
            return 0;
        bno = get_free_blocks(sbi, 8);
        if (!bno) {
            ret = -ENOSPC;
            goto brelse_index;
        }
        index->extents[extent].ee_start = bno;
        index->extents[extent].ee_len = 8;
        index->extents[extent].ee_block =
            extent ? index->extents[extent - 1].ee_block +
                         index->extents[extent - 1].ee_len
                   : 0;
        alloc = true;
    } else {
        bno = index->extents[extent].ee_start + iblock -
              index->extents[extent].ee_block;
    }

    /* Map the physical block to to the given buffer_head */
    map_bh(bh_result, sb, bno);

brelse_index:
    brelse(bh_index);

    return ret;
}

/*
 * Called by the page cache to read a page from the physical disk and map it
in
 * memory.
 */
static int myfs_readpage(struct file *file, struct page *page)
{
    return mpage_readpage(page, myfs_file_get_block);
}

/*
 * Called by the page cache to write a dirty page to the physical disk (when
 * sync is called or when memory is needed).
 */
static int myfs_writepage(struct page *page, struct writeback_control *wbc)
{
    return block_write_full_page(page, myfs_file_get_block, wbc);
}

/*
 * Called by the VFS when a write() syscall occurs on file before writing the
 * data in the page cache. This functions checks if the write will be able to
 * complete and allocates the necessary blocks through block_write_begin().
 */
static int myfs_write_begin(struct file *file,
                                struct address_space *mapping,
                                loff_t pos,
                                unsigned int len,
                                unsigned int flags,
                                struct page **pagep,
                                void **fsdata)
{
    struct myfs_sb_info *sbi = MYFS_SB(file->f_inode->i_sb);
    int err;
    uint32_t nr_allocs = 0;

    /* Check if the write can be completed (enough space?) */
    if (pos + len > MYFS_MAX_FILESIZE)
        return -ENOSPC;
    nr_allocs = max(pos + len, file->f_inode->i_size) / MYFS_BLOCK_SIZE;
    if (nr_allocs > file->f_inode->i_blocks - 1)
        nr_allocs -= file->f_inode->i_blocks - 1;
    else
        nr_allocs = 0;
    if (nr_allocs > sbi->nr_free_blocks)
        return -ENOSPC;

    /* prepare the write */
    err = block_write_begin(mapping, pos, len, flags, pagep,
                            myfs_file_get_block);
    /* if this failed, reclaim newly allocated blocks */
    if (err < 0)
        pr_err("newly allocated blocks reclaim not implemented yet\n");
    return err;
}

/*
 * Called by the VFS after writing data from a write() syscall to the page
 * cache. This functions updates inode metadata and truncates the file if
 * necessary.
 */
static int myfs_write_end(struct file *file,
                              struct address_space *mapping,
                              loff_t pos,
                              unsigned int len,
                              unsigned int copied,
                              struct page *page,
                              void *fsdata)
{
    struct inode *inode = file->f_inode;
    struct myfs_inode_info *ci = MYFS_INODE(inode);
    struct super_block *sb = inode->i_sb;
    uint32_t nr_blocks_old;

    /* Complete the write() */
    int ret = generic_write_end(file, mapping, pos, len, copied, page,
fsdata);
    if (ret < len) {
        pr_err("wrote less than requested.");
        return ret;
    }

    nr_blocks_old = inode->i_blocks;

    /* Update inode metadata */
    inode->i_blocks = inode->i_size / MYFS_BLOCK_SIZE + 2;
    inode->i_mtime = inode->i_ctime = current_time(inode);
    mark_inode_dirty(inode);

    /* If file is smaller than before, free unused blocks */
    if (nr_blocks_old > inode->i_blocks) {
        int i;
        struct buffer_head *bh_index;
        struct myfs_file_ei_block *index;
        uint32_t first_ext;

        /* Free unused blocks from page cache */
        truncate_pagecache(inode, inode->i_size);

        /* Read ei_block to remove unused blocks */
        bh_index = sb_bread(sb, ci->ei_block);
        if (!bh_index) {
            pr_err("failed truncating '%s'. we just lost %llu blocks\n",
                   file->f_path.dentry->d_name.name,
                   nr_blocks_old - inode->i_blocks);
            goto end;
        }
        index = (struct myfs_file_ei_block *) bh_index->b_data;

        first_ext = myfs_ext_search(index, inode->i_blocks - 1);
        /* Reserve unused block in last extent */
        if (inode->i_blocks - 1 != index->extents[first_ext].ee_block)
            first_ext++;

        for (i = first_ext; i < MYFS_MAX_EXTENTS; i++) {
            if (!index->extents[i].ee_start)
                break;
            put_blocks(MYFS_SB(sb), index->extents[i].ee_start,
                       index->extents[i].ee_len);
            memset(&index->extents[i], 0, sizeof(struct myfs_extent));
        }
        mark_buffer_dirty(bh_index);
        brelse(bh_index);
    }
end:
    return ret;
}

const struct address_space_operations myfs_aops = {
    .readpage = myfs_readpage,
    .writepage = myfs_writepage,
    .write_begin = myfs_write_begin,
    .write_end = myfs_write_end,
};

const struct file_operations myfs_file_ops = {
    .llseek = generic_file_llseek,
    .owner = THIS_MODULE,
    .read_iter = generic_file_read_iter,
    .write_iter = generic_file_write_iter,
    .fsync = generic_file_fsync,
};
```

Dir.c

```c
#define pr_fmt(fmt) "myfs: " fmt

#include <linux/buffer_head.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/module.h>

#include "myfs.h"

/*
 * Iterate over the files contained in dir and commit them in ctx.
 * This function is called by the VFS while ctx->pos changes.
 * Return 0 on success.
 */
static int myfs_iterate(struct file *dir, struct dir_context *ctx)
{
    struct inode *inode = file_inode(dir);
    struct myfs_inode_info *ci = MYFS_INODE(inode);
    struct super_block *sb = inode->i_sb;
    struct buffer_head *bh = NULL;
    struct myfs_dir_block *dblock = NULL;
    struct myfs_file *f = NULL;
    int i;

    /* Check that dir is a directory */
    if (!S_ISDIR(inode->i_mode))
        return -ENOTDIR;

    /*
     * Check that ctx->pos is not bigger than what we can handle (including
     * . and ..)
     */
    if (ctx->pos > MYFS_MAX_SUBFILES + 2)
        return 0;

    /* Commit . and .. to ctx */
    if (!dir_emit_dots(dir, ctx))
        return 0;

    /* Read the directory index block on disk */
    bh = sb_bread(sb, ci->dir_block);
    if (!bh)
        return -EIO;
    dblock = (struct myfs_dir_block *) bh->b_data;

    /* Iterate over the index block and commit subfiles */
    for (i = ctx->pos - 2; i < MYFS_MAX_SUBFILES; i++) {
        f = &dblock->files[i];
        if (!f->inode)
            break;
        if (!dir_emit(ctx, f->filename, MYFS_FILENAME_LEN, f->inode,
                      DT_UNKNOWN))
            break;
        ctx->pos++;
    }
    brelse(bh);

    return 0;
}

const struct file_operations myfs_dir_ops = {
    .owner = THIS_MODULE,
    .iterate_shared = myfs_iterate,
};
```

Myfs.h

```
#ifndef MYFS_H
#define MYFS_H

#define MYFS_MAGIC 0xDEADCELL

#define MYFS_SB_BLOCK_NR 0

#define MYFS_BLOCK_SIZE (1 << 12) /* 4 KiB */
#define MYFS_MAX_EXTENTS \
    MYFS_BLOCK_SIZE / sizeof(struct myfs_extent)
#define MYFS_MAX_BLOCKS_PER_EXTENT 8 /* It can be ~(uint32) 0 */
#define MYFS_MAX_FILESIZE                                       \
    (uint64_t) MYFS_MAX_BLOCKS_PER_EXTENT *MYFS_BLOCK_SIZE \
        *MYFS_MAX_EXTENTS
#define MYFS_FILENAME_LEN 28
#define MYFS_MAX_SUBFILES 128


struct myfs_inode {
    uint32_t i_mode;   /* File mode */
    uint32_t i_uid;    /* Owner id */
    uint32_t i_gid;    /* Group id */
    uint32_t i_size;   /* Size in bytes */
    uint32_t i_ctime;  /* Inode change time */
    uint32_t i_atime;  /* Access time */
    uint32_t i_mtime;  /* Modification time */
    uint32_t i_blocks; /* Block count */
    uint32_t i_nlink;  /* Hard links count */
    union {
        uint32_t ei_block;  /* Block with list of extents for this file */
        uint32_t dir_block; /* Block with list of files for this directory */
    };
    char i_data[32]; /* store symlink content */
};

#define MYFS_INODES_PER_BLOCK (MYFS_BLOCK_SIZE / sizeof(struct myfs_inode))

struct myfs_sb_info {
    uint32_t magic; /* Magic number */

    uint32_t nr_blocks; /* Total number of blocks (incl sb & inodes) */
    uint32_t nr_inodes; /* Total number of inodes */

    uint32_t nr_istore_blocks; /* Number of inode store blocks */
    uint32_t nr_ifree_blocks;  /* Number of inode free bitmap blocks */
    uint32_t nr_bfree_blocks;  /* Number of block free bitmap blocks */

    uint32_t nr_free_inodes; /* Number of free inodes */
    uint32_t nr_free_blocks; /* Number of free blocks */

#ifdef __KERNEL__
    unsigned long *ifree_bitmap; /* In-memory free inodes bitmap */
    unsigned long *bfree_bitmap; /* In-memory free blocks bitmap */
#endif
};

#ifdef __KERNEL__

struct myfs_inode_info {
    union {
        uint32_t ei_block;  /* Block with list of extents for this file */
        uint32_t dir_block; /* Block with list of files for this directory */
    };
    char i_data[32];
    struct inode vfs_inode;
};

struct myfs_extent {
    uint32_t ee_block; /* first logical block extent covers */
    uint32_t ee_len;   /* number of blocks covered by extent */
    uint32_t ee_start; /* first physical block extent covers */
};

struct myfs_file_ei_block {
    struct myfs_extent extents[MYFS_MAX_EXTENTS];
};

struct myfs_dir_block {
    struct myfs_file {
        uint32_t inode;
        char filename[MYFS_FILENAME_LEN];
    } files[MYFS_MAX_SUBFILES];
};

/* superblock functions */
int myfs_fill_super(struct super_block *sb, void *data, int silent);

/* inode functions */
int myfs_init_inode_cache(void);
void myfs_destroy_inode_cache(void);
struct inode *myfs_iget(struct super_block *sb, unsigned long ino);

/* file functions */
extern const struct file_operations myfs_file_ops;
extern const struct file_operations myfs_dir_ops;
extern const struct address_space_operations myfs_aops;

/* extent functions */
extern uint32_t myfs_ext_search(struct myfs_file_ei_block *index,
                                    uint32_t iblock);

/* Getters for superbock and inode */
#define MYFS_SB(sb) (sb->s_fs_info)
#define MYFS_INODE(inode) \
    (container_of(inode, struct myfs_inode_info, vfs_inode))

#endif /* __KERNEL__ */

#endif /* MYFS_H */
```

Extent.c

```
#include <linux/fs.h>
#include <linux/kernel.h>

#include "myfs.h"

/*
 * Search the extent which contain the target block.
 * Retrun the first unused file index if not found.
 * Return -1 if it is out of range.
 * TODO: use binary search.
 */
uint32_t myfs_ext_search(struct myfs_file_ei_block *index,
                             uint32_t iblock)
{
    uint32_t i;
    for (i = 0; i < MYFS_MAX_EXTENTS; i++) {
        uint32_t block = index->extents[i].ee_block;
        uint32_t len = index->extents[i].ee_len;
        if (index->extents[i].ee_start == 0 ||
            (iblock >= block && iblock < block + len))
            return i;
    }
    return -1;
}
```

Bitmap.h

```
#ifndef MYFS_BITMAP_H
#define MYFS_BITMAP_H

#include <linux/bitmap.h>
#include "myfs.h"

/*
 * Return the first bit we found and clear the the following `len`
consecutive
 * free bit(s) (set to 1) in a given in-memory bitmap spanning over multiple
 * blocks. Return 0 if no enough free bit(s) were found (we assume that the
 * first bit is never free because of the superblock and the root inode, thus
 * allowing us to use 0 as an error value).
 */
static inline uint32_t get_first_free_bits(unsigned long *freemap,
                                           unsigned long size,
                                           uint32_t len)
{
    uint32_t bit, prev = 0, count = 0;
    for_each_set_bit (bit, freemap, size) {
        if (prev != bit - 1)
            count = 0;
        prev = bit;
        if (++count == len) {
            bitmap_clear(freemap, bit - len + 1, len);
            return bit - len + 1;
        }
    }
    return 0;
}

/*
 * Return an unused inode number and mark it used.
 * Return 0 if no free inode was found.
 */
static inline uint32_t get_free_inode(struct myfs_sb_info *sbi)
{
    uint32_t ret = get_first_free_bits(sbi->ifree_bitmap, sbi->nr_inodes, 1);
    if (ret)
        sbi->nr_free_inodes--;
    return ret;
}

/*
 * Return `len` unused block(s) number and mark it used.
 * Return 0 if no enough free block(s) were found.
 */
static inline uint32_t get_free_blocks(struct myfs_sb_info *sbi,
                                       uint32_t len)
{
    uint32_t ret = get_first_free_bits(sbi->bfree_bitmap, sbi->nr_blocks,
len);
    if (ret)
        sbi->nr_free_blocks -= len;
    return ret;
}


/* Mark the `len` bit(s) from i-th bit in freemap as free (i.e. 1) */
static inline int put_free_bits(unsigned long *freemap,
                                 unsigned long size,
                                 uint32_t i,
                                 uint32_t len)
{
    /* i is greater than freemap size */
    if (i + len - 1 > size)
        return -1;

    bitmap_set(freemap, i, len);

    return 0;
}

/* Mark an inode as unused */
static inline void put_inode(struct myfs_sb_info *sbi, uint32_t ino)
{
    if (put_free_bits(sbi->ifree_bitmap, sbi->nr_inodes, ino, 1))
        return;

    sbi->nr_free_inodes++;
}

/* Mark len block(s) as unused */
static inline void put_blocks(struct myfs_sb_info *sbi,
                              uint32_t bno,
                              uint32_t len)
{
    if (put_free_bits(sbi->bfree_bitmap, sbi->nr_blocks, bno, len))
        return;

    sbi->nr_free_blocks += len;
}

#endif /* MYFS_BITMAP_H */
```