LAB 3

```
#include <linux/autoconf.h>
#include <linux/version.h>
#ifndef EXPORT_SYMTAB
# define EXPORT_SYMTAB
#endif
#include <linux/module.h>
#include <linux/moduleparam.h>
#include "ospfs.h"
#include <linux/string.h>
#include <linux/slab.h>
#include <linux/file.h>
#include <linux/fs.h>
#include <linux/namei.h>
#include <asm/uaccess.h>
#include <linux/kernel.h>
#include <linux/sched.h>

/*****************************************************************************
 * ospfsmod
 *
 *   This is the OSPFS module!  It contains both library code for your use,
 *   and exercises where you must add code.
 *
 *****************************************************************************/

/* Define eprintk() to be a version of printk(), which prints messages to
 * the console.
 * (If working on a real Linux machine, change KERN_NOTICE to KERN_ALERT or
 * KERN_EMERG so that you are sure to see the messages.  By default, the
 * kernel does not print all messages to the console.  Levels like KERN_ALERT
 * and KERN_EMERG will make sure that you will see messages.) */
#define eprintk(format, ...) printk(KERN_NOTICE format, ## __VA_ARGS__)

// The actual disk data is just an array of raw memory.
// The initial array is defined in fsimg.c, based on your 'base' directory.
extern uint8_t ospfs_data[];
extern uint32_t ospfs_length;

// A pointer to the superblock; see ospfs.h for details on the struct.
static ospfs_super_t * const ospfs_super =
   (ospfs_super_t *) &ospfs_data[OSPFS_BLKSIZE];

static int change_size(ospfs_inode_t *oi, uint32_t want_size);
static ospfs_direntry_t *find_direntry(ospfs_inode_t *dir_oi, const char *name, int namelen);


/*****************************************************************************
 * FILE SYSTEM OPERATIONS STRUCTURES
 *
 *   Linux filesystems are based around three interrelated structures.
 *
 *   These are:
 *
 *   1. THE LINUX SUPERBLOCK.  This structure represents the whole file system.
 *      Example members include the root directory and the number of blocks
 *      on the disk.
 *   2. LINUX INODES.  Each file and directory in the file system corresponds
 *      to an inode.  Inode operations include "mkdir" and "create" (add to
 *      directory).
 *   3. LINUX FILES.  Corresponds to an open file or directory.  Operations
 *      include "read", "write", and "readdir".
 *
 *   When Linux wants to perform some file system operation,
 *   it calls a function pointer provided by the file system type.
 *   (Thus, Linux file systems are object oriented!)
 *
 *   These function pointers are grouped into structures called "operations"
 *   structures.
 *
 *   The initial portion of the file declares all the operations structures we
 *   need to support ospfsmod: one for the superblock, several for different
 *   kinds of inodes and files.  There are separate inode_operations and
 *   file_operations structures for OSPFS directories and for regular OSPFS
 *   files.  The structures are actually defined near the bottom of this file.
 */

// Basic file system type structure
// (links into Linux's list of file systems it supports)
static struct file_system_type ospfs_fs_type;
// Inode and file operations for regular files
static struct inode_operations ospfs_reg_inode_ops;
static struct file_operations ospfs_reg_file_ops;
// Inode and file operations for directories
static struct inode_operations ospfs_dir_inode_ops;
static struct file_operations ospfs_dir_file_ops;
// Inode operations for symbolic links
static struct inode_operations ospfs_symlink_inode_ops;
// Other required operations
static struct dentry_operations ospfs_dentry_ops;
static struct super_operations ospfs_superblock_ops;



/*****************************************************************************
 * BITVECTOR OPERATIONS
 *
 *   OSPFS uses a free bitmap to keep track of free blocks.
 *   These bitvector operations, which set, clear, and test individual bits
 *   in a bitmap, may be useful.
 */

// bitvector_set -- Set 'i'th bit of 'vector' to 1.
static inline void
bitvector_set(void *vector, int i)
{
   ((uint32_t *) vector) [i / 32] |= (1 << (i % 32));
}

// bitvector_clear -- Set 'i'th bit of 'vector' to 0.
static inline void
```

```
bitvector_clear(void *vector, int i)
{
   ((uint32_t *) vector) [i / 32] &= ~(1 << (i % 32));
}

// bitvector_test -- Return the value of the 'i'th bit of 'vector'.
static inline int
bitvector_test(const void *vector, int i)
{
   return (((const uint32_t *) vector) [i / 32] & (1 << (i % 32))) != 0;
}



/*****************************************************************************
 * OSPFS HELPER FUNCTIONS
 */

// ospfs_size2nblocks(size)
// Returns the number of blocks required to hold 'size' bytes of data.
//
//   Input:   size -- file size
//   Returns: a number of blocks

uint32_t
ospfs_size2nblocks(uint32_t size)
{
   return (size + OSPFS_BLKSIZE - 1) / OSPFS_BLKSIZE;
}


// ospfs_block(blockno)
// Use this function to load a block's contents from "disk".
//
//   Input:   blockno -- block number
//   Returns: a pointer to that block's data

static void *
ospfs_block(uint32_t blockno)
{
   return &ospfs_data[blockno * OSPFS_BLKSIZE];
}


// ospfs_inode(ino)
// Use this function to load a 'ospfs_inode' structure from "disk".
//
//   Input:   ino -- inode number
//   Returns: a pointer to the corresponding ospfs_inode structure

static inline ospfs_inode_t *
ospfs_inode(ino_t ino)
{
   ospfs_inode_t *oi;
   if (ino >= ospfs_super->os_ninodes)
      return 0;
   oi = ospfs_block(ospfs_super->os_firstinob);
   return &oi[ino];
}


// ospfs_inode_blockno(oi, offset)
// Use this function to look up the blocks that are part of a file's
// contents.
//
//   Inputs:  oi     -- pointer to a OSPFS inode
//            offset -- byte offset into that inode
//   Returns: the block number of the block that contains the 'offset'th byte
//            of the file

static inline uint32_t
ospfs_inode_blockno(ospfs_inode_t *oi, uint32_t offset)
{
   uint32_t blockno = offset / OSPFS_BLKSIZE;
   if (offset >= oi->oi_size || oi->oi_ftype == OSPFS_FTYPE_SYMLINK)
      return 0;
   else if (blockno >= OSPFS_NDIRECT + OSPFS_NINDIRECT) {
      uint32_t blockoff = blockno - (OSPFS_NDIRECT + OSPFS_NINDIRECT);
      uint32_t *indirect2_block = ospfs_block(oi->oi_indirect2);
      uint32_t *indirect_block = ospfs_block(indirect2_block[blockoff / OSPFS_NINDIRECT]);
      return indirect_block[blockoff % OSPFS_NINDIRECT];
   } else if (blockno >= OSPFS_NDIRECT) {
      uint32_t *indirect_block = ospfs_block(oi->oi_indirect);
      return indirect_block[blockno - OSPFS_NDIRECT];
   } else
      return oi->oi_direct[blockno];
}


// ospfs_inode_data(oi, offset)
// Use this function to load part of inode's data from "disk",
// where 'offset' is relative to the first byte of inode data.
//
//   Inputs:  oi     -- pointer to a OSPFS inode
//            offset -- byte offset into 'oi's data contents
//   Returns: a pointer to the 'offset'th byte of 'oi's data contents
//
// Be careful: the returned pointer is only valid within a single block.
// This function is a simple combination of 'ospfs_inode_blockno'
// and 'ospfs_block'.

static inline void *
ospfs_inode_data(ospfs_inode_t *oi, uint32_t offset)
{
   uint32_t blockno = ospfs_inode_blockno(oi, offset);
   return (uint8_t *) ospfs_block(blockno) + (offset % OSPFS_BLKSIZE);
}


/*****************************************************************************
```

```
 * LOW-LEVEL FILE SYSTEM FUNCTIONS
 * There are no exercises in this section, and you don't need to understand
 * the code.
 */

// ospfs_mk_linux_inode(sb, ino)
// Linux's in-memory 'struct inode' structure represents disk
// objects (files and directories).  Many file systems have their own
// notion of inodes on disk, and for such file systems, Linux's
// 'struct inode's are like a cache of on-disk inodes.
//
// This function takes an inode number for the OSPFS and constructs
// and returns the corresponding Linux 'struct inode'.
//
//   Inputs:  sb  -- the relevant Linux super_block structure (one per mount)
//        ino -- OSPFS inode number
//   Returns: 'struct inode'

static struct inode *
ospfs_mk_linux_inode(struct super_block *sb, ino_t ino)
{
   ospfs_inode_t *oi = ospfs_inode(ino);
   struct inode *inode;

   if (!oi)
      return 0;
   if (!(inode = new_inode(sb)))
      return 0;

   inode->i_ino = ino;
   // Make it look like everything was created by root.
   inode->i_uid = inode->i_gid = 0;
   inode->i_size = oi->oi_size;

   if (oi->oi_ftype == OSPFS_FTYPE_REG) {
      // Make an inode for a regular file.
      inode->i_mode = oi->oi_mode | S_IFREG;
      inode->i_op = &ospfs_reg_inode_ops;
      inode->i_fop = &ospfs_reg_file_ops;
      inode->i_nlink = oi->oi_nlink;

   } else if (oi->oi_ftype == OSPFS_FTYPE_DIR) {
      // Make an inode for a directory.
      inode->i_mode = oi->oi_mode | S_IFDIR;
      inode->i_op = &ospfs_dir_inode_ops;
      inode->i_fop = &ospfs_dir_file_ops;
      inode->i_nlink = oi->oi_nlink + 1 /* dot-dot */;

   } else if (oi->oi_ftype == OSPFS_FTYPE_SYMLINK) {
      // Make an inode for a symbolic link.
      inode->i_mode = S_IRUSR | S_IRGRP | S_IROTH
         | S_IWUSR | S_IWGRP | S_IWOTH
         | S_IXUSR | S_IXGRP | S_IXOTH | S_IFLNK;
      inode->i_op = &ospfs_symlink_inode_ops;
      inode->i_nlink = oi->oi_nlink;
   } else
      panic("OSPFS: unknown inode type!");

   // Access and modification times are now.
   inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;
   return inode;
}


// ospfs_fill_super, ospfs_get_sb
// These functions are called by Linux when the user mounts a version of
// the OSPFS onto some directory.  They help construct a Linux
// 'struct super_block' for that file system.

static int
ospfs_fill_super(struct super_block *sb, void *data, int flags)
{
   struct inode *root_inode;

   sb->s_blocksize = OSPFS_BLKSIZE;
   sb->s_blocksize_bits = OSPFS_BLKSIZE_BITS;
   sb->s_magic = OSPFS_MAGIC;
   sb->s_op = &ospfs_superblock_ops;

   if (!(root_inode = ospfs_mk_linux_inode(sb, OSPFS_ROOT_INO))
       || !(sb->s_root = d_alloc_root(root_inode))) {
      iput(root_inode);
      sb->s_dev = 0;
      return -ENOMEM;
   }

   return 0;
}

static int
ospfs_get_sb(struct file_system_type *fs_type, int flags, const char *dev_name, void *data, struct
vfsmount *mount)
{
   return get_sb_single(fs_type, flags, data, ospfs_fill_super, mount);
}


// ospfs_delete_dentry
//  Another bookkeeping function.

static int
ospfs_delete_dentry(struct dentry *dentry)
{
   return 1;
}


/*****************************************************************************
 * DIRECTORY OPERATIONS
```

```
 *
 * EXERCISE: Finish 'ospfs_dir_readdir' and 'ospfs_symlink'.
 */

// ospfs_dir_lookup(dir, dentry, ignore)
// This function implements the "lookup" directory operation, which
// looks up a named entry.
//
// We have written this function for you.
//
//   Input:  dir    -- The Linux 'struct inode' for the directory.
//           You can extract the corresponding 'ospfs_inode_t'
//           by calling 'ospfs_inode' with the relevant inode number.
//      dentry -- The name of the entry being looked up.
//   Effect: Looks up the entry named 'dentry'.  If found, attaches the
//      entry's 'struct inode' to the 'dentry'.  If not found, returns
//      a "negative dentry", which has no inode attachment.

static struct dentry *
ospfs_dir_lookup(struct inode *dir, struct dentry *dentry, struct nameidata *ignore)
{
   // Find the OSPFS inode corresponding to 'dir'
   ospfs_inode_t *dir_oi = ospfs_inode(dir->i_ino);
   struct inode *entry_inode = NULL;
   int entry_off;

   // Make sure filename is not too long
   if (dentry->d_name.len > OSPFS_MAXNAMELEN)
      return (struct dentry *) ERR_PTR(-ENAMETOOLONG);

   // Mark with our operations
   dentry->d_op = &ospfs_dentry_ops;

   // Search through the directory block
   for (entry_off = 0; entry_off < dir_oi->oi_size;
        entry_off += OSPFS_DIRENTRY_SIZE) {
      // Find the OSPFS inode for the entry
      ospfs_direntry_t *od = ospfs_inode_data(dir_oi, entry_off);

      // Set 'entry_inode' if we find the file we are looking for
      if (od->od_ino > 0
          && strlen(od->od_name) == dentry->d_name.len
          && memcmp(od->od_name, dentry->d_name.name, dentry->d_name.len) == 0) {
         entry_inode = ospfs_mk_linux_inode(dir->i_sb, od->od_ino);
         if (!entry_inode)
            return (struct dentry *) ERR_PTR(-EINVAL);
         break;
      }
   }

   // We return a dentry whether or not the file existed.
   // The file exists if and only if 'entry_inode != NULL'.
   // If the file doesn't exist, the dentry is called a "negative dentry".

   // d_splice_alias() attaches the inode to the dentry.
   // If it returns a new dentry, we need to set its operations.
   if ((dentry = d_splice_alias(entry_inode, dentry)))
      dentry->d_op = &ospfs_dentry_ops;
   return dentry;
}



// ospfs_dir_readdir(filp, dirent, filldir)
//   This function is called when the kernel reads the contents of a directory
//   (i.e. when file_operations.readdir is called for the inode).
//
//   Inputs:  filp -- The 'struct file' structure correspoding to
//            the open directory.
//            The most important member is 'filp->f_pos', the
//            File POSition.  This remembers how far into the
//            directory we are, so if the user calls 'readdir'
//            twice, we don't forget our position.
//            This function must update 'filp->f_pos'.
//        dirent   -- Used to pass to 'filldir'.
//        filldir  -- A pointer to a callback function.
//            This function should call 'filldir' once for each
//            directory entry, passing it six arguments:
//        (1) 'dirent'.
//        (2) The directory entry's name.
//        (3) The length of the directory entry's name.
//        (4) The 'f_pos' value corresponding to the directory entry.
//        (5) The directory entry's inode number.
//        (6) DT_REG, for regular files; DT_DIR, for subdirectories;
//           or DT_LNK, for symbolic links.
//            This function should stop returning directory
//            entries either when the directory is complete, or
//            when 'filldir' returns < 0, whichever comes first.
//
//   Returns: 1 at end of directory, 0 if filldir returns < 0 before the end
//     of the directory, and -(error number) on error.
//
//   EXERCISE: Finish implementing this function.

static int
ospfs_dir_readdir(struct file *filp, void *dirent, filldir_t filldir)
{
   struct inode *dir_inode = filp->f_dentry->d_inode;
   ospfs_inode_t *dir_oi = ospfs_inode(dir_inode->i_ino);
   uint32_t f_pos = filp->f_pos;
   int r = 0;    /* Error return value, if any */
   int ok_so_far = 0;/* Return value from 'filldir' */

   // f_pos is an offset into the directory's data, plus two.
   // The "plus two" is to account for "." and "..".
   if (r == 0 && f_pos == 0) {
      ok_so_far = filldir(dirent, ".", 1, f_pos, dir_inode->i_ino, DT_DIR);
      if (ok_so_far >= 0)
         f_pos++;
   }
```

```
  if (r == 0 && ok_so_far >= 0 && f_pos == 1) {
    ok_so_far = filldir(dirent, "..", 2, f_pos, filp->f_dentry->d_parent->d_inode->i_ino, DT_DIR);
    if (ok_so_far >= 0)
      f_pos++;
  }

  while (r == 0 && ok_so_far >= 0 && f_pos >= 2) {
    ospfs_direntry_t *od;
    ospfs_inode_t *entry_oi;

    if (f_pos > dir_oi->oi_size * OSPFS_DIRENTRY_SIZE) { /* TODO: error cond */
      r = 1;
      break;
    }

    /* Get a pointer to the next entry (od) in the directory.
     * The file system interprets the contents of a
     * directory-file as a sequence of ospfs_direntry structures.
     * You will find 'f_pos' and 'ospfs_inode_data' useful.
     *
     * Then use the fields of that file to fill in the directory
     * entry.  To figure out whether a file is a regular file or
     * another directory, use 'ospfs_inode' to get the directory
     * entry's corresponding inode, and check out its 'oi_ftype'
     * member.
     *
     * Make sure you ignore blank directory entries!  (Which have
     * an inode number of 0.)
     *
     * If the current entry is successfully read (the call to
     * filldir returns >= 0), or the current entry is skipped,
     * your function should advance f_pos by the proper amount to
     * advance to the next directory entry.
     */

    od = ospfs_inode_data(dir_oi, f_pos * OSPFS_DIRENTRY_SIZE);
    entry_oi = ospfs_inode(od->od_ino);

    if(entry_oi != 0) {
      switch(entry_oi->oi_ftype) {
        case OSPFS_FTYPE_REG:
          ok_so_far = filldir(dirent, od->od_name, strlen(od->od_name), f_pos, od->od_ino, DT_REG);
          break;
        case OSPFS_FTYPE_DIR:
          ok_so_far = filldir(dirent, od->od_name, strlen(od->od_name), f_pos, od->od_ino, DT_DIR);
          break;
        case OSPFS_FTYPE_SYMLINK:
          ok_so_far = filldir(dirent, od->od_name, strlen(od->od_name), f_pos, od->od_ino, DT_LNK);
          break;
        default: {
          r=1;
          continue;
        }

      }

      f_pos++;
    }

    filp->f_pos = f_pos;
    return r;
  }


  // ospfs_unlink(dirino, dentry)
  //   This function is called to remove a file.
  //
  //   Inputs: dirino  -- You may ignore this.
  //           dentry  -- The 'struct dentry' structure, which contains the inode
  //                      the directory entry points to and the directory entry's
  //                      directory.
  //
  //   Returns: 0 if success and -ENOENT on entry not found.
  //
  //   EXERCISE: Make sure that deleting symbolic links works correctly.

  static int
  ospfs_unlink(struct inode *dirino, struct dentry *dentry)
  {
    ospfs_inode_t *oi = ospfs_inode(dentry->d_inode->i_ino);
    ospfs_inode_t *dir_oi = ospfs_inode(dentry->d_parent->d_inode->i_ino);
    int entry_off;
    ospfs_direntry_t *od;

    od = NULL;
    for (entry_off = 0; entry_off < dir_oi->oi_size;
         entry_off += OSPFS_DIRENTRY_SIZE) {
      od = ospfs_inode_data(dir_oi, entry_off);
      if (od->od_ino > 0
          && strlen(od->od_name) == dentry->d_name.len
          && memcmp(od->od_name, dentry->d_name.name, dentry->d_name.len) == 0)
        break;
    }

    if (entry_off == dir_oi->oi_size) {
      return -ENOENT;
    }

    od->od_ino = 0;
    oi->oi_nlink--;

    if (oi->oi_ftype != OSPFS_FTYPE_SYMLINK && oi->oi_nlink == 0)
      return change_size(oi, 0);

    return 0;
  }
```

```
/***************************************************************************
 * FREE-BLOCK BITMAP OPERATIONS
 *
 * EXERCISE: Implement these functions.
 */

// allocate_block()
// Use this function to allocate a block.
//
//   Inputs:  none
//   Returns: block number of the allocated block,
//       or 0 if the disk is full
//
//   This function searches the free-block bitmap, which starts at Block 2, for
//   a free block, allocates it (by marking it non-free), and returns the block
//   number to the caller.  The block itself is not touched.
//
//   Note:  A value of 0 for a bit indicates the corresponding block is
//      allocated; a value of 1 indicates the corresponding block is free.
//
//   You can use the functions bitvector_set(), bitvector_clear(), and
//   bitvector_test() to do bit operations on the map.

static uint32_t
allocate_block(void)
{
   void *bitmap = ospfs_block(OSPFS_FREEMAP_BLK);

   int i;
   for (i = 0; i < ospfs_super->os_nblocks; i++) {
      if (bitvector_test(bitmap, i)) {
         bitvector_clear(bitmap, i);
         return i;
      }
   }

   return 0;
}


// free_block(blockno)
// Use this function to free an allocated block.
//
//   Inputs:  blockno -- the block number to be freed
//   Returns: none
//
//   This function should mark the named block as free in the free-block
//   bitmap.  (You might want to program defensively and make sure the block
//   number isn't obviously bogus: the boot sector, superblock, free-block
//   bitmap, and inode blocks must never be freed.  But this is not required.)

static void
free_block(uint32_t blockno)
{
   void *bitmap = ospfs_block(OSPFS_FREEMAP_BLK);
   bitvector_set(bitmap, blockno);
}


/***************************************************************************
 * FILE OPERATIONS
 *
 * EXERCISE: Finish off change_size, read, and write.
 *
 * The find_*, add_block, and remove_block functions are only there to support
 * the change_size function.  If you prefer to code change_size a different
 * way, then you may not need these functions.
 *
 */

// The following functions are used in our code to unpack a block number into
// its consituent pieces: the doubly indirect block number (if any), the
// indirect block number (which might be one of many in the doubly indirect
// block), and the direct block number (which might be one of many in an
// indirect block).  We use these functions in our implementation of
// change_size.


// int32_t indir2_index(uint32_t b)
// Returns the doubly-indirect block index for file block b.
//
// Inputs:  b -- the zero-based index of the file block (e.g., 0 for the first
//      block, 1 for the second, etc.)
// Returns: 0 if block index 'b' requires using the doubly indirect
//        block, -1 if it does not.
//
// EXERCISE: Fill in this function.

static int32_t
indir2_index(uint32_t b)
{
   if (b < OSPFS_NDIRECT + OSPFS_NINDIRECT) {
      return -1;
   }
   else {
      return 0;
   }
}


// int32_t indir_index(uint32_t b)
// Returns the indirect block index for file block b.
//
// Inputs:  b -- the zero-based index of the file block
// Returns: -1 if b is one of the file's direct blocks;
//     0 if b is located under the file's first indirect block;
//     otherwise, the offset of the relevant indirect block within
//     the doubly indirect block.
//
// EXERCISE: Fill in this function.
```

```
static int32_t
indir_index(uint32_t b)
{
   if (b < OSPFS_NDIRECT) {
      return -1;
   }
   else if (b < OSPFS_NDIRECT + OSPFS_NINDIRECT) {
      return 0;
   }
   else {
      return (b - OSPFS_NDIRECT - OSPFS_NINDIRECT) / OSPFS_NINDIRECT;
   }
}


// int32_t indir_index(uint32_t b)
//  Returns the indirect block index for file block b.
//
// Inputs:  b -- the zero-based index of the file block
// Returns: the index of block b in the relevant indirect block or the direct
//      block array.
//
// EXERCISE: Fill in this function.

static int32_t
direct_index(uint32_t b)
{
   if (b < OSPFS_NDIRECT) {
      return b;
   }
   else if (b < OSPFS_NDIRECT + OSPFS_NINDIRECT) {
      return b - OSPFS_NDIRECT;
   }
   else {
      return (b - OSPFS_NDIRECT) % OSPFS_NINDIRECT;
   }
}


// add_block(ospfs_inode_t *oi)
//   Adds a single data block to a file, adding indirect and
//   doubly-indirect blocks if necessary. (Helper function for
//   change_size).
//
// Inputs: oi -- pointer to the file we want to grow
// Returns: 0 if successful, < 0 on error.  Specifically:
//          -ENOSPC if you are unable to allocate a block
//          due to the disk being full or
//          -EIO for any other error.
//          If the function is successful, then oi->oi_size
//          should be set to the maximum file size in bytes that could
//          fit in oi's data blocks.  If the function returns an error,
//          then oi->oi_size should remain unchanged. Any newly
//          allocated blocks should be erased (set to zero).
//
// EXERCISE: Finish off this function.
//
// Remember that allocating a new data block may require allocating
// as many as three disk blocks, depending on whether a new indirect
// block and/or a new indirect^2 block is required. If the function
// fails with -ENOSPC or -EIO, then you need to make sure that you
// free any indirect (or indirect^2) blocks you may have allocated!
//
// Also, make sure you:
//  1) zero out any new blocks that you allocate
//  2) store the disk block number of any newly allocated block
//     in the appropriate place in the inode or one of the
//     indirect blocks.
//  3) update the oi->oi_size field

static int
add_block(ospfs_inode_t *oi)
{
   // current number of blocks in file
   uint32_t n = ospfs_size2nblocks(oi->oi_size);

   // keep track of allocations to free in case of -ENOSPC
   uint32_t allocated[2] = { 0, 0 };
   uint32_t direct, indir;

   if (n < 0)
      return -EIO;
   else if (n < OSPFS_NDIRECT) {
      direct = allocate_block();
      if (!direct)
         return -ENOSPC;
      memset(ospfs_block(direct), 0, OSPFS_BLKSIZE);

      oi->oi_direct[n] = direct;
   }
   else if (n < OSPFS_NDIRECT + OSPFS_NINDIRECT) {
      if (!oi->oi_indirect) {
         allocated[0] = allocate_block();
         if (!allocated[0])
            return -ENOSPC;
         memset(ospfs_block(allocated[0]), 0, OSPFS_BLKSIZE);

         oi->oi_indirect = allocated[0];
      }

      direct = allocate_block();
      if (!direct) {
         if (allocated[0]) {
            free_block(allocated[0]);
            oi->oi_indirect = 0;
         }
         return -ENOSPC;
      }
      memset(ospfs_block(direct), 0, OSPFS_BLKSIZE);
```

```
    ((uint32_t*) ospfs_block(oi->oi_indirect))[direct_index(n)] = direct;
  }
  else if (n < OSPFS_MAXFILEBLKS) {
    if (!oi->oi_indirect2) {
      allocated[0] = allocate_block();
      if (!allocated[0])
        return -ENOSPC;
      memset(ospfs_block(allocated[0]), 0, OSPFS_BLKSIZE);

      oi->oi_indirect2 = allocated[0];
    }

    indir = ((uint32_t *) ospfs_block(oi->oi_indirect2))[indir_index(n)];

    if (!indir) {
      allocated[1] = allocate_block();
      if (!allocated[1]) {
        if (allocated[0])
          free_block(allocated[0]);
        return -ENOSPC;
      }
      memset(ospfs_block(allocated[1]), 0, OSPFS_BLKSIZE);

      indir = allocated[1];
    }

    direct = allocate_block();
    if (!direct) {
      if (allocated[0]) {
        free_block(allocated[0]);
        oi->oi_indirect2 = 0;
      }
      if (allocated[1])
        free_block(allocated[1]);
      return -ENOSPC;
    }
    memset(ospfs_block(direct), 0, OSPFS_BLKSIZE);

    ((uint32_t *) ospfs_block(indir))[direct_index(n)] = direct;
  }
  else {
    return -ENOSPC;
  }


  oi->oi_size += OSPFS_BLKSIZE;

  return 0;
}


// remove_block(ospfs_inode_t *oi)
//   Removes a single data block from the end of a file, freeing
//   any indirect and indirect^2 blocks that are no
//   longer needed. (Helper function for change_size)
//
// Inputs: oi -- pointer to the file we want to shrink
// Returns: 0 if successful, < 0 on error.
//          If the function is successful, then oi->oi_size
//          should be set to the maximum file size that could
//          fit in oi's blocks.  If the function returns -EIO (for
//          instance if an indirect block that should be there isn't),
//          then oi->oi_size should remain unchanged.
//
// EXERCISE: Finish off this function.
//
// Remember that you must free any indirect and doubly-indirect blocks
// that are no longer necessary after shrinking the file.  Removing a
// single data block could result in as many as 3 disk blocks being
// deallocated.  Also, if you free a block, make sure that
// you set the block pointer to 0.  Don't leave pointers to
// deallocated blocks laying around!

static int
remove_block(ospfs_inode_t *oi)
{
  // current number of blocks in file
  uint32_t n = ospfs_size2nblocks(oi->oi_size);

  if (n < 0) {
    return -EIO;
  }
  else if (n == 0) {
    return 0;
  }

  n = n - 1;

  if (n < OSPFS_NDIRECT) {
    free_block(oi->oi_direct[n]);
    oi->oi_direct[n] = 0;
  }
  else if (n < OSPFS_NDIRECT + OSPFS_NINDIRECT) {
    uint32_t* indir = (uint32_t *) ospfs_block(oi->oi_indirect);

    free_block(indir[direct_index(n)]);
    indir[direct_index(n)] = 0;

    if (direct_index(n) == 0) {
      free_block(oi->oi_indirect);
      oi->oi_indirect = 0;
    }
  }
  else if (n < OSPFS_MAXFILEBLKS) {
    uint32_t* indir2 = (uint32_t *) ospfs_block(oi->oi_indirect2);
    uint32_t* indir = (uint32_t *) ospfs_block(indir2[indir_index(n)]);

    free_block(indir[direct_index(n)]);
    indir[direct_index(n)] = 0;
```

```
      if (direct_index(n) == 0) {
         free_block(indir2[indir_index(n)]);
         indir2[indir_index(n)] = 0;
      }

      if (indir_index(n) == 0) {
         free_block(oi->oi_indirect2);
         oi->oi_indirect2 = 0;
      }
   }
   else
      return -EIO;

   oi->oi_size -= OSPFS_BLKSIZE;

   return 0;
}


// change_size(oi, want_size)
// Use this function to change a file's size, allocating and freeing
// blocks as necessary.
//
//   Inputs:  oi-- pointer to the file whose size we're changing
//        want_size -- the requested size in bytes
//   Returns: 0 on success, < 0 on error.  In particular:
//     -ENOSPC: if there are no free blocks available
//     -EIO:    an I/O error -- for example an indirect block should
//          exist, but doesn't
//       If the function succeeds, the file's oi_size member should be
//       changed to want_size, with blocks allocated as appropriate.
//       Any newly-allocated blocks should be erased (set to 0).
//       If there is an -ENOSPC error when growing a file,
//       the file size and allocated blocks should not change from their
//       original values!!!
//            (However, if there is an -EIO error, do not worry too much about
//       restoring the file.)
//
//   If want_size has the same number of blocks as the current file, life
//   is good -- the function is pretty easy.  But the function might have
//   to add or remove blocks.
//
//   If you need to grow the file, then do so by adding one block at a time
//   using the add_block function you coded above. If one of these additions
//   fails with -ENOSPC, you must shrink the file back to its original size!
//
//   If you need to shrink the file, remove blocks from the end of
//   the file one at a time using the remove_block function you coded above.
//
//   Also: Don't forget to change the size field in the metadata of the file.
//        (The value that the final add_block or remove_block set it to
//         is probably not correct).
//
//   EXERCISE: Finish off this function.
```

```
static int
change_size(ospfs_inode_t *oi, uint32_t new_size)
{
   uint32_t old_size = oi->oi_size;
   int r = 0;

   while (ospfs_size2nblocks(oi->oi_size) < ospfs_size2nblocks(new_size)) {
      r = add_block(oi);
      if (r < 0)
         break;
   }

   if (r == -EIO)
      return -EIO;
   else if (r == -ENOSPC) {
      while (ospfs_size2nblocks(oi->oi_size) > ospfs_size2nblocks(old_size))
         r = remove_block(oi);

      oi->oi_size = old_size;
      return -ENOSPC;
   }

   while (ospfs_size2nblocks(oi->oi_size) > ospfs_size2nblocks(new_size)) {
      r = remove_block(oi);
      if (r < 0)
         return r;
   }

   oi->oi_size = new_size;

   return 0;
}


// ospfs_notify_change
// This function gets called when the user changes a file's size,
// owner, or permissions, among other things.
// OSPFS only pays attention to file size changes (see change_size above).
// We have written this function for you -- except for file quotas.

static int
ospfs_notify_change(struct dentry *dentry, struct iattr *attr)
{
   struct inode *inode = dentry->d_inode;
   ospfs_inode_t *oi = ospfs_inode(inode->i_ino);
   int retval = 0;

   if (attr->ia_valid & ATTR_SIZE) {
      if (oi->oi_ftype == OSPFS_FTYPE_DIR)
         return -EPERM;
      if ((retval = change_size(oi, attr->ia_size)) < 0)
         goto out;
   }
```

```
    if (attr->ia_valid & ATTR_MODE)
        oi->oi_mode = attr->ia_mode;

    if ((retval = inode_change_ok(inode, attr)) < 0
        || (retval = inode_setattr(inode, attr)) < 0)
        goto out;

    out:
    return retval;
}


// ospfs_read
// Linux calls this function to read data from a file.
// It is the file_operations.read callback.
//
//   Inputs:  filp -- a file pointer
//            buffer    -- a user space ptr where data should be copied
//            count     -- the amount of data requested
//            f_pos     -- points to the file position
//   Returns: Number of chars read on success, -(error code) on error.
//
//   This function copies the corresponding bytes from the file into the user
//   space ptr (buffer).  Use copy_to_user() to accomplish this.
//   The current file position is passed into the function
//   as 'f_pos'; read data starting at that position, and update the position
//   when you're done.
//
//   EXERCISE: Complete this function.

static ssize_t
ospfs_read(struct file *filp, char __user *buffer, size_t count, loff_t *f_pos)
{
    ospfs_inode_t *oi = ospfs_inode(filp->f_dentry->d_inode->i_ino);
    int retval = 0;
    size_t amount = 0;

    // Make sure we don't read past the end of the file!
    // Change 'count' so we never read past the end of the file.
    if (*f_pos + count > oi->oi_size)
        count = oi->oi_size - *f_pos;

    // Copy the data to user block by block
    while (amount < count && retval >= 0) {
        uint32_t blockno = ospfs_inode_blockno(oi, *f_pos);
        uint32_t n;
        char *data;

        if (blockno == 0) {
            retval = -EIO;
            goto done;
        }

        data = ospfs_block(blockno);

        // Figure out how much data is left in this block to read.
        // Copy data into user space. Return -EFAULT if unable to write
        // into user space.
        // Use variable 'n' to track number of bytes moved.
        n = (count + (*f_pos % OSPFS_BLKSIZE) - amount > OSPFS_BLKSIZE ?
             OSPFS_BLKSIZE - (*f_pos % OSPFS_BLKSIZE) : count - amount);
        retval = copy_to_user(buffer,data,n);

        if (retval < 0) {
            retval = -EFAULT;
            goto done;
        }

        buffer += n;
        amount += n;
        *f_pos += n;
    }

    done:
    return (retval >= 0 ? amount : retval);
}


// ospfs_write
// Linux calls this function to write data to a file.
// It is the file_operations.write callback.
//
//   Inputs:  filp -- a file pointer
//            buffer    -- a user space ptr where data should be copied from
//            count     -- the amount of data to write
//            f_pos     -- points to the file position
//   Returns: Number of chars written on success, -(error code) on error.
//
//   This function copies the corresponding bytes from the user space ptr
//   into the file.  Use copy_from_user() to accomplish this. Unlike read(),
//   where you cannot read past the end of the file, it is OK to write past
//   the end of the file; this should simply change the file's size.
//
//   EXERCISE: Complete this function.

static ssize_t
ospfs_write(struct file *filp, const char __user *buffer, size_t count, loff_t *f_pos)
{
    ospfs_inode_t *oi = ospfs_inode(filp->f_dentry->d_inode->i_ino);
    int retval = 0;
    size_t amount = 0;

    // Support files opened with the O_APPEND flag.  To detect O_APPEND,
    // use struct file's f_flags field and the O_APPEND bit.
    if (filp->f_flags & O_APPEND)
        *f_pos = oi->oi_size;

    if ((*f_pos + count) > oi->oi_size)
        if (change_size(oi, (*f_pos + count)) < 0)
            goto done;
```

```
    // Copy data block by block
    while (amount < count && retval >= 0) {
        uint32_t blockno = ospfs_inode_blockno(oi, *f_pos);
        uint32_t n;
        char *data;

        if (blockno == 0) {
            retval = -EIO;
            goto done;
        }

        data = ospfs_block(blockno);

        // Figure out how much data is left in this block to write.
        // Copy data from user space. Return -EFAULT if unable to read
        // read user space.
        // Keep track of the number of bytes moved in 'n'.
        n = OSPFS_BLKSIZE - (*f_pos % OSPFS_BLKSIZE);

        if (n > count - amount)
            n = count - amount;

        if (copy_from_user(data + (*f_pos % OSPFS_BLKSIZE), buffer, n) != 0)
            return -EFAULT;

        buffer += n;
        amount += n;
        *f_pos += n;
    }

    done:
    return (retval >= 0 ? amount : retval);
}


// find_direntry(dir_oi, name, namelen)
// Looks through the directory to find an entry with name 'name' (length
// in characters 'namelen').  Returns a pointer to the directory entry,
// if one exists, or NULL if one does not.
//
//   Inputs:  dir_oi  -- the OSP inode for the directory
//            name    -- name to search for
//            namelen -- length of 'name'.  (If -1, then use strlen(name).)
//
// We have written this function for you.

static ospfs_direntry_t *
find_direntry(ospfs_inode_t *dir_oi, const char *name, int namelen)
{
    int off;
    if (namelen < 0)
        namelen = strlen(name);
    for (off = 0; off < dir_oi->oi_size; off += OSPFS_DIRENTRY_SIZE) {
        ospfs_direntry_t *od = ospfs_inode_data(dir_oi, off);
        if (od->od_ino
            && strlen(od->od_name) == namelen
            && memcmp(od->od_name, name, namelen) == 0)
            return od;
    }
    return 0;
}


// create_blank_direntry(dir_oi)
// 'dir_oi' is an OSP inode for a directory.
// Return a blank directory entry in that directory.  This might require
// adding a new block to the directory.  Returns an error pointer (see
// below) on failure.
//
// ERROR POINTERS: The Linux kernel uses a special convention for returning
// error values in the form of pointers.  Here's how it works.
// - ERR_PTR(errno): Creates a pointer value corresponding to an error.
// - IS_ERR(ptr): Returns true iff 'ptr' is an error value.
// - PTR_ERR(ptr): Returns the error value for an error pointer.
// For example:
//
// static ospfs_direntry_t *create_blank_direntry(...) {
//     return ERR_PTR(-ENOSPC);
// }
// static int ospfs_create(...) {
//     ...
//     ospfs_direntry_t *od = create_blank_direntry(...);
//     if (IS_ERR(od))
//         return PTR_ERR(od);
//     ...
// }
//
// The create_blank_direntry function should use this convention.
//
// EXERCISE: Write this function.

static ospfs_direntry_t *
create_blank_direntry(ospfs_inode_t *dir_oi)
{
    ospfs_direntry_t *dir_entry;
    int retval;

    uint32_t offset;
    for (offset = 0; offset < dir_oi->oi_size; offset += OSPFS_DIRENTRY_SIZE) {
        dir_entry = ospfs_inode_data(dir_oi, offset);
        if (dir_entry->od_ino == 0)
            return dir_entry;
    }

    retval = add_block(dir_oi);
    if (retval < 0)
        return ERR_PTR(retval);

    dir_entry = ospfs_inode_data(dir_oi, offset);
```

```
  return dir_entry;
}

// ospfs_link(src_dentry, dir, dst_dentry
//   Linux calls this function to create hard links.
//   It is the ospfs_dir_inode_ops.link callback.
//
//   Inputs: src_dentry   -- a pointer to the dentry for the source file.  This
//                           file's inode contains the real data for the hard
//                           linked filae.  The important elements are:
//                             src_dentry->d_name.name
//                             src_dentry->d_name.len
//                             src_dentry->d_inode->i_ino
//           dir          -- a pointer to the containing directory for the new
//                           hard link.
//           dst_dentry   -- a pointer to the dentry for the new hard link file.
//                           The important elements are:
//                             dst_dentry->d_name.name
//                             dst_dentry->d_name.len
//                             dst_dentry->d_inode->i_ino
//                           Two of these values are already set.  One must be
//                           set by you, which one?
//   Returns: 0 on success, -(error code) on error.  In particular:
//              -ENAMETOOLONG if dst_dentry->d_name.len is too large, or
//              'symname' is too long;
//              -EEXIST       if a file named the same as 'dst_dentry' already
//                            exists in the given 'dir';
//              -ENOSPC       if the disk is full & the file can't be created;
//              -EIO          on I/O error.
//
//   EXERCISE: Complete this function.

static int
ospfs_link(struct dentry *src_dentry, struct inode *dir, struct dentry *dst_dentry) {
  ospfs_direntry_t* link;

  if (dst_dentry->d_name.len > OSPFS_MAXNAMELEN)
      return -ENAMETOOLONG;
  if (find_direntry(ospfs_inode(dir->i_ino),
      dst_dentry->d_name.name, dst_dentry->d_name.len))
      return -EEXIST;

  link = create_blank_direntry(ospfs_inode(dir->i_ino));
  if (IS_ERR(link))
      return PTR_ERR(link);

  link->od_ino = src_dentry->d_inode->i_ino;
  memcpy(link->od_name, dst_dentry->d_name.name, dst_dentry->d_name.len);
  link->od_name[dst_dentry->d_name.len] = '\0';

  ospfs_inode(src_dentry->d_inode->i_ino)->oi_nlink++;

  return 0;
}
```

```
// ospfs_create
//   Linux calls this function to create a regular file.
//   It is the ospfs_dir_inode_ops.create callback.
//
//   Inputs:  dir  -- a pointer to the containing directory's inode
//            dentry    -- the name of the file that should be created
//                         The only important elements are:
//                         dentry->d_name.name: filename (char array, not null
//                           terminated)
//                         dentry->d_name.len: length of filename
//            mode  -- the permissions mode for the file (set the new
//            inode's oi_mode field to this value)
//        nd -- ignore this
//   Returns: 0 on success, -(error code) on error.  In particular:
//              -ENAMETOOLONG if dentry->d_name.len is too large;
//              -EEXIST       if a file named the same as 'dentry' already
//                            exists in the given 'dir';
//              -ENOSPC       if the disk is full & the file can't be created;
//              -EIO          on I/O error.
//
//   We have provided strictly less skeleton code for this function than for
//   the others.  Here's a brief outline of what you need to do:
//   1. Check for the -EEXIST error and find an empty directory entry using the
// helper functions above.
//   2. Find an empty inode.  Set the 'entry_ino' variable to its inode number.
//   3. Initialize the directory entry and inode.
//
//   EXERCISE: Complete this function.

static int
ospfs_create(struct inode *dir, struct dentry *dentry, int mode, struct nameidata *nd)
{
  ospfs_inode_t *dir_oi = ospfs_inode(dir->i_ino);
  ospfs_direntry_t *dir_entry;
  ospfs_inode_t *inode;
  uint32_t entry_ino = 0;

  if (dentry->d_name.len > OSPFS_MAXNAMELEN)
      return -ENAMETOOLONG;
  if (find_direntry(dir_oi, dentry->d_name.name, dentry->d_name.len))
      return -EEXIST;

  dir_entry = create_blank_direntry(dir_oi);
  if (IS_ERR(dir_entry))
      return PTR_ERR(dir_entry);

  for (entry_ino = 0; entry_ino < ospfs_super->os_ninodes; entry_ino++) {
      inode = ospfs_inode(entry_ino);
      if (inode->oi_nlink == 0)
          break;
  }
  if (entry_ino == ospfs_super->os_ninodes)
      return -ENOSPC;

  dir_entry->od_ino = entry_ino;
```

```
    memcpy(dir_entry->od_name, dentry->d_name.name, dentry->d_name.len);
    dir_entry->od_name[dentry->d_name.len] = '\0';
    // initialize file
    inode->oi_size = 0;
    inode->oi_ftype = OSPFS_FTYPE_REG;
    inode->oi_mode = mode;
    inode->oi_nlink = 1;

    /* Execute this code after your function has successfully created the
       file.  Set entry_ino to the created file's inode number before
       getting here. */
    {
        struct inode *i = ospfs_mk_linux_inode(dir->i_sb, entry_ino);
        if (!i)
            return -ENOMEM;
        d_instantiate(dentry, i);
        return 0;
    }
}


// ospfs_symlink(dirino, dentry, symname)
//   Linux calls this function to create a symbolic link.
//   It is the ospfs_dir_inode_ops.symlink callback.
//
//   Inputs: dir     -- a pointer to the containing directory's inode
//           dentry  -- the name of the file that should be created
//                      The only important elements are:
//                      dentry->d_name.name: filename (char array, not null
//                          terminated)
//                      dentry->d_name.len: length of filename
//           symname -- the symbolic link's destination
//
//   Returns: 0 on success, -(error code) on error.  In particular:
//              -ENAMETOOLONG if dentry->d_name.len is too large, or
//              'symname' is too long;
//              -EEXIST       if a file named the same as 'dentry' already
//                            exists in the given 'dir';
//              -ENOSPC       if the disk is full & the file can't be created;
//              -EIO          on I/O error.
//
//   EXERCISE: Complete this function.

static int
ospfs_symlink(struct inode *dir, struct dentry *dentry, const char *symname)
{
    ospfs_inode_t *dir_oi = ospfs_inode(dir->i_ino);
    uint32_t entry_ino = 0;
    ospfs_symlink_inode_t* link;

    if (dentry->d_name.len > OSPFS_MAXNAMELEN ||
        strlen(symname) > OSPFS_MAXSYMLINKLEN)
        return -ENAMETOOLONG;
    if (find_direntry(ospfs_inode(dir->i_ino),
        dentry->d_name.name, dentry->d_name.len))
        return -EEXIST;

    entry_ino = ospfs_create(dir, dentry, dir_oi->oi_mode, NULL);
    if (entry_ino < 0)
        return entry_ino;
    entry_ino = find_direntry(ospfs_inode(dir->i_ino),
        dentry->d_name.name, dentry->d_name.len)->od_ino;
    link = (ospfs_symlink_inode_t*) ospfs_inode(entry_ino);

    link->oi_size = strlen(symname);
    link->oi_ftype = OSPFS_FTYPE_SYMLINK;
    link->oi_nlink = 1;
    memcpy(link->oi_symlink, symname, strlen(symname));

    /* Execute this code after your function has successfully created the
       file.  Set entry_ino to the created file's inode number before
       getting here. */
    {
        struct inode *i = ospfs_mk_linux_inode(dir->i_sb, entry_ino);
        if (!i)
            return -ENOMEM;
        d_instantiate(dentry, i);
        return 0;
    }
}


// ospfs_follow_link(dentry, nd)
//   Linux calls this function to follow a symbolic link.
//   It is the ospfs_symlink_inode_ops.follow_link callback.
//
//   Inputs: dentry -- the symbolic link's directory entry
//           nd     -- to be filled in with the symbolic link's destination
//
//   Exercise: Expand this function to handle conditional symlinks.  Conditional
//   symlinks will always be created by users in the following form
//     root?/path/1:/path/2.
//   (hint: Should the given form be changed in any way to make this method
//   easier?  With which character do most functions expect C strings to end?)
//

static void *
ospfs_follow_link(struct dentry *dentry, struct nameidata *nd)
{
    ospfs_symlink_inode_t *oi =
        (ospfs_symlink_inode_t *) ospfs_inode(dentry->d_inode->i_ino);

    if (strncmp(oi->oi_symlink, "root?", 5) == 0) {
        int pivot = strchr(oi->oi_symlink, ':') - oi->oi_symlink;

        if (current->uid == 0) {
            oi->oi_symlink[pivot] = '\0';
            nd_set_link(nd, oi->oi_symlink + 5 + 1);
        }
        else
```

```
        nd_set_link(nd, oi->oi_symlink + pivot + 1);
    }
    else
        nd_set_link(nd, oi->oi_symlink);

    return (void *) 0;
}


// Define the file system operations structures mentioned above.

static struct file_system_type ospfs_fs_type = {
    .owner      = THIS_MODULE,
    .name       = "ospfs",
    .get_sb     = ospfs_get_sb,
    .kill_sb    = kill_anon_super
};

static struct inode_operations ospfs_reg_inode_ops = {
    .setattr= ospfs_notify_change
};

static struct file_operations ospfs_reg_file_ops = {
    .llseek     = generic_file_llseek,
    .read       = ospfs_read,
    .write      = ospfs_write
};

static struct inode_operations ospfs_dir_inode_ops = {
    .lookup     = ospfs_dir_lookup,
    .link       = ospfs_link,
    .unlink     = ospfs_unlink,
    .create     = ospfs_create,
    .symlink    = ospfs_symlink
};

static struct file_operations ospfs_dir_file_ops = {
    .read       = generic_read_dir,
    .readdir    = ospfs_dir_readdir
};

static struct inode_operations ospfs_symlink_inode_ops = {
    .readlink   = generic_readlink,
    .follow_link    = ospfs_follow_link
};

static struct dentry_operations ospfs_dentry_ops = {
    .d_delete   = ospfs_delete_dentry
};

static struct super_operations ospfs_superblock_ops = {
};


// Functions used to hook the module into the kernel!

static int __init init_ospfs_fs(void)
{
    eprintk("Loading ospfs module...\n");
    return register_filesystem(&ospfs_fs_type);
}

static void __exit exit_ospfs_fs(void)
{
    unregister_filesystem(&ospfs_fs_type);
    eprintk("Unloading ospfs module\n");
}

module_init(init_ospfs_fs)
module_exit(exit_ospfs_fs)

// Information about the module
MODULE_AUTHOR("Jonathan Woong");
MODULE_DESCRIPTION("OSPFS");
MODULE_LICENSE("GPL");
```