#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");