Fs.c

#define pr\_fmt(fmt) KBUILD\_MODNAME ": " fmt

#include <linux/fs.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include "myfs.h"

/\* Mount a myfs partition \*/

struct dentry **\***myfs\_mount**(**struct file\_system\_type **\***fs\_type**,**

int flags**,**

const char **\***dev\_name**,**

void **\***data**)**

**{**

struct dentry **\***dentry **=**

mount\_bdev**(**fs\_type**,** flags**,** dev\_name**,** data**,** myfs\_fill\_super**);**

**if** **(**IS\_ERR**(**dentry**))**

pr\_err**(**"'%s' mount failure\n"**,** dev\_name**);**

**else**

pr\_info**(**"'%s' mount success\n"**,** dev\_name**);**

**return** dentry**;**

**}**

/\* Unmount a myfs partition \*/

void myfs\_kill\_sb**(**struct super\_block **\***sb**)**

**{**

kill\_block\_super**(**sb**);**

pr\_info**(**"unmounted disk\n"**);**

**}**

static struct file\_system\_type myfs\_file\_system\_type **=** **{**

**.**owner **=** THIS\_MODULE**,**

**.**name **=** "myfs"**,**

**.**mount **=** myfs\_mount**,**

**.**kill\_sb **=** myfs\_kill\_sb**,**

**.**fs\_flags **=** FS\_REQUIRES\_DEV**,**

**.**next **=** **NULL,**

**};**

static int \_\_init myfs\_init**(**void**)**

**{**

int ret **=** myfs\_init\_inode\_cache**();**

**if** **(**ret**)** **{**

pr\_err**(**"inode cache creation failed\n"**);**

**goto** end**;**

**}**

ret **=** register\_filesystem**(&**myfs\_file\_system\_type**);**

**if** **(**ret**)** **{**

pr\_err**(**"register\_filesystem() failed\n"**);**

**goto** end**;**

**}**

pr\_info**(**"module loaded\n"**);**

end**:**

**return** ret**;**

**}**

static void \_\_exit myfs\_exit**(**void**)**

**{**

int ret **=** unregister\_filesystem**(&**myfs\_file\_system\_type**);**

**if** **(**ret**)**

pr\_err**(**"unregister\_filesystem() failed\n"**);**

myfs\_destroy\_inode\_cache**();**

pr\_info**(**"module unloaded\n"**);**

**}**

module\_init**(**myfs\_init**);**

module\_exit**(**myfs\_exit**);**

MODULE\_LICENSE**(**"Dual BSD/GPL"**);**

Inode.c

#define pr\_fmt(fmt) KBUILD\_MODNAME ": " fmt

#include <linux/buffer\_head.h>

#include <linux/fs.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include "bitmap.h"

#include "myfs.h"

static const struct inode\_operations myfs\_inode\_ops**;**

static const struct inode\_operations symlink\_inode\_ops**;**

/\* Get inode ino from disk \*/

struct inode **\***myfs\_iget**(**struct super\_block **\***sb**,** unsigned long ino**)**

**{**

struct inode **\***inode **=** **NULL;**

struct myfs\_inode **\***cinode **=** **NULL;**

struct myfs\_inode\_info **\***ci **=** **NULL;**

struct myfs\_sb\_info **\***sbi **=** MYFS\_SB**(**sb**);**

struct buffer\_head **\***bh **=** **NULL;**

uint32\_t inode\_block **=** **(**ino **/** MYFS\_INODES\_PER\_BLOCK**)** **+** 1**;**

uint32\_t inode\_shift **=** ino **%** MYFS\_INODES\_PER\_BLOCK**;**

int ret**;**

/\* Fail if ino is out of range \*/

**if** **(**ino **>=** sbi**->**nr\_inodes**)**

**return** ERR\_PTR**(-**EINVAL**);**

/\* Get a locked inode from Linux \*/

inode **=** iget\_locked**(**sb**,** ino**);**

**if** **(!**inode**)**

**return** ERR\_PTR**(-**ENOMEM**);**

/\* If inode is in cache, return it \*/

**if** **(!(**inode**->**i\_state **&** I\_NEW**))**

**return** inode**;**

ci **=** MYFS\_INODE**(**inode**);**

/\* Read inode from disk and initialize \*/

bh **=** sb\_bread**(**sb**,** inode\_block**);**

**if** **(!**bh**)** **{**

ret **=** **-**EIO**;**

**goto** failed**;**

**}**

cinode **=** **(**struct myfs\_inode **\*)** bh**->**b\_data**;**

cinode **+=** inode\_shift**;**

inode**->**i\_ino **=** ino**;**

inode**->**i\_sb **=** sb**;**

inode**->**i\_op **=** **&**myfs\_inode\_ops**;**

inode**->**i\_mode **=** le32\_to\_cpu**(**cinode**->**i\_mode**);**

i\_uid\_write**(**inode**,** le32\_to\_cpu**(**cinode**->**i\_uid**));**

i\_gid\_write**(**inode**,** le32\_to\_cpu**(**cinode**->**i\_gid**));**

inode**->**i\_size **=** le32\_to\_cpu**(**cinode**->**i\_size**);**

inode**->**i\_ctime**.**tv\_sec **=** **(**time64\_t**)** le32\_to\_cpu**(**cinode**->**i\_ctime**);**

inode**->**i\_ctime**.**tv\_nsec **=** 0**;**

inode**->**i\_atime**.**tv\_sec **=** **(**time64\_t**)** le32\_to\_cpu**(**cinode**->**i\_atime**);**

inode**->**i\_atime**.**tv\_nsec **=** 0**;**

inode**->**i\_mtime**.**tv\_sec **=** **(**time64\_t**)** le32\_to\_cpu**(**cinode**->**i\_mtime**);**

inode**->**i\_mtime**.**tv\_nsec **=** 0**;**

inode**->**i\_blocks **=** le32\_to\_cpu**(**cinode**->**i\_blocks**);**

set\_nlink**(**inode**,** le32\_to\_cpu**(**cinode**->**i\_nlink**));**

**if** **(**S\_ISDIR**(**inode**->**i\_mode**))** **{**

ci**->**dir\_block **=** le32\_to\_cpu**(**cinode**->**dir\_block**);**

inode**->**i\_fop **=** **&**myfs\_dir\_ops**;**

**}** **else** **if** **(**S\_ISREG**(**inode**->**i\_mode**))** **{**

ci**->**ei\_block **=** le32\_to\_cpu**(**cinode**->**ei\_block**);**

inode**->**i\_fop **=** **&**myfs\_file\_ops**;**

inode**->**i\_mapping**->**a\_ops **=** **&**myfs\_aops**;**

**}** **else** **if** **(**S\_ISLNK**(**inode**->**i\_mode**))** **{**

strncpy**(**ci**->**i\_data**,** cinode**->**i\_data**,** **sizeof(**ci**->**i\_data**));**

inode**->**i\_link **=** ci**->**i\_data**;**

inode**->**i\_op **=** **&**symlink\_inode\_ops**;**

**}**

brelse**(**bh**);**

/\* Unlock the inode to make it usable \*/

unlock\_new\_inode**(**inode**);**

**return** inode**;**

failed**:**

brelse**(**bh**);**

iget\_failed**(**inode**);**

**return** ERR\_PTR**(**ret**);**

**}**

/\*

\* Look for dentry in dir.

\* Fill dentry with NULL if not in dir, with the corresponding inode if found.

\* Returns NULL on success.

\*/

static struct dentry **\***myfs\_lookup**(**struct inode **\***dir**,**

struct dentry **\***dentry**,**

unsigned int flags**)**

**{**

struct super\_block **\***sb **=** dir**->**i\_sb**;**

struct myfs\_inode\_info **\***ci\_dir **=** MYFS\_INODE**(**dir**);**

struct inode **\***inode **=** **NULL;**

struct buffer\_head **\***bh **=** **NULL;**

struct myfs\_dir\_block **\***dblock **=** **NULL;**

struct myfs\_file **\***f **=** **NULL;**

int i**;**

/\* Check filename length \*/

**if** **(**dentry**->**d\_name**.**len **>** MYFS\_FILENAME\_LEN**)**

**return** ERR\_PTR**(-**ENAMETOOLONG**);**

/\* Read the directory block on disk \*/

bh **=** sb\_bread**(**sb**,** ci\_dir**->**dir\_block**);**

**if** **(!**bh**)**

**return** ERR\_PTR**(-**EIO**);**

dblock **=** **(**struct myfs\_dir\_block **\*)** bh**->**b\_data**;**

/\* Search for the file in directory \*/

**for** **(**i **=** 0**;** i **<** MYFS\_MAX\_SUBFILES**;** i**++)** **{**

f **=** **&**dblock**->**files**[**i**];**

**if** **(!**f**->**inode**)**

**break;**

**if** **(!**strncmp**(**f**->**filename**,** dentry**->**d\_name**.**name**,** MYFS\_FILENAME\_LEN**))** **{**

inode **=** myfs\_iget**(**sb**,** f**->**inode**);**

**break;**

**}**

**}**

brelse**(**bh**);**

/\* Update directory access time \*/

dir**->**i\_atime **=** current\_time**(**dir**);**

mark\_inode\_dirty**(**dir**);**

/\* Fill the dentry with the inode \*/

d\_add**(**dentry**,** inode**);**

**return** **NULL;**

**}**

/\* Create a new inode in dir \*/

static struct inode **\***myfs\_new\_inode**(**struct inode **\***dir**,** mode\_t mode**)**

**{**

struct inode **\***inode**;**

struct myfs\_inode\_info **\***ci**;**

struct super\_block **\***sb**;**

struct myfs\_sb\_info **\***sbi**;**

uint32\_t ino**,** bno**;**

int ret**;**

/\* Check mode before doing anything to avoid undoing everything \*/

**if** **(!**S\_ISDIR**(**mode**)** **&&** **!**S\_ISREG**(**mode**)** **&&** **!**S\_ISLNK**(**mode**))** **{**

pr\_err**(**

"File type not supported (only directory, regular file and symlink "

"supported)\n"**);**

**return** ERR\_PTR**(-**EINVAL**);**

**}**

/\* Check if inodes are available \*/

sb **=** dir**->**i\_sb**;**

sbi **=** MYFS\_SB**(**sb**);**

**if** **(**sbi**->**nr\_free\_inodes **==** 0 **||** sbi**->**nr\_free\_blocks **==** 0**)**

**return** ERR\_PTR**(-**ENOSPC**);**

/\* Get a new free inode \*/

ino **=** get\_free\_inode**(**sbi**);**

**if** **(!**ino**)**

**return** ERR\_PTR**(-**ENOSPC**);**

inode **=** myfs\_iget**(**sb**,** ino**);**

**if** **(**IS\_ERR**(**inode**))** **{**

ret **=** PTR\_ERR**(**inode**);**

**goto** put\_ino**;**

**}**

**if** **(**S\_ISLNK**(**mode**))** **{**

inode\_init\_owner**(**inode**,** dir**,** mode**);**

set\_nlink**(**inode**,** 1**);**

inode**->**i\_ctime **=** inode**->**i\_atime **=** inode**->**i\_mtime **=** current\_time**(**inode**);**

inode**->**i\_op **=** **&**symlink\_inode\_ops**;**

**return** inode**;**

**}**

ci **=** MYFS\_INODE**(**inode**);**

/\* Get a free block for this new inode's index \*/

bno **=** get\_free\_blocks**(**sbi**,** 1**);**

**if** **(!**bno**)** **{**

ret **=** **-**ENOSPC**;**

**goto** put\_inode**;**

**}**

/\* Initialize inode \*/

inode\_init\_owner**(**inode**,** dir**,** mode**);**

inode**->**i\_blocks **=** 1**;**

**if** **(**S\_ISDIR**(**mode**))** **{**

ci**->**dir\_block **=** bno**;**

inode**->**i\_size **=** MYFS\_BLOCK\_SIZE**;**

inode**->**i\_fop **=** **&**myfs\_dir\_ops**;**

set\_nlink**(**inode**,** 2**);** /\* . and .. \*/

**}** **else** **if** **(**S\_ISREG**(**mode**))** **{**

ci**->**ei\_block **=** bno**;**

inode**->**i\_size **=** 0**;**

inode**->**i\_fop **=** **&**myfs\_file\_ops**;**

inode**->**i\_mapping**->**a\_ops **=** **&**myfs\_aops**;**

set\_nlink**(**inode**,** 1**);**

**}**

inode**->**i\_ctime **=** inode**->**i\_atime **=** inode**->**i\_mtime **=** current\_time**(**inode**);**

**return** inode**;**

put\_inode**:**

iput**(**inode**);**

put\_ino**:**

put\_inode**(**sbi**,** ino**);**

**return** ERR\_PTR**(**ret**);**

**}**

/\*

\* Create a file or directory in this way:

\* - check filename length and if the parent directory is not full

\* - create the new inode (allocate inode and blocks)

\* - cleanup index block of the new inode

\* - add new file/directory in parent index

\*/

static int myfs\_create**(**struct inode **\***dir**,**

struct dentry **\***dentry**,**

umode\_t mode**,**

bool excl**)**

**{**

struct super\_block **\***sb**;**

struct inode **\***inode**;**

struct myfs\_inode\_info **\***ci\_dir**;**

struct myfs\_dir\_block **\***dblock**;**

char **\***fblock**;**

struct buffer\_head **\***bh**,** **\***bh2**;**

int ret **=** 0**,** i**;**

/\* Check filename length \*/

**if** **(**strlen**(**dentry**->**d\_name**.**name**)** **>** MYFS\_FILENAME\_LEN**)**

**return** **-**ENAMETOOLONG**;**

/\* Read parent directory index \*/

ci\_dir **=** MYFS\_INODE**(**dir**);**

sb **=** dir**->**i\_sb**;**

bh **=** sb\_bread**(**sb**,** ci\_dir**->**dir\_block**);**

**if** **(!**bh**)**

**return** **-**EIO**;**

dblock **=** **(**struct myfs\_dir\_block **\*)** bh**->**b\_data**;**

/\* Check if parent directory is full \*/

**if** **(**dblock**->**files**[**MYFS\_MAX\_SUBFILES **-** 1**].**inode **!=** 0**)** **{**

ret **=** **-**EMLINK**;**

**goto** end**;**

**}**

/\* Get a new free inode \*/

inode **=** myfs\_new\_inode**(**dir**,** mode**);**

**if** **(**IS\_ERR**(**inode**))** **{**

ret **=** PTR\_ERR**(**inode**);**

**goto** end**;**

**}**

/\*

\* Scrub ei\_block/dir\_block for new file/directory to avoid previous data

\* messing with new file/directory.

\*/

bh2 **=** sb\_bread**(**sb**,** MYFS\_INODE**(**inode**)->**ei\_block**);**

**if** **(!**bh2**)** **{**

ret **=** **-**EIO**;**

**goto** iput**;**

**}**

fblock **=** **(**char **\*)** bh2**->**b\_data**;**

memset**(**fblock**,** 0**,** MYFS\_BLOCK\_SIZE**);**

mark\_buffer\_dirty**(**bh2**);**

brelse**(**bh2**);**

/\* Find first free slot in parent index and register new inode \*/

**for** **(**i **=** 0**;** i **<** MYFS\_MAX\_SUBFILES**;** i**++)**

**if** **(**dblock**->**files**[**i**].**inode **==** 0**)**

**break;**

dblock**->**files**[**i**].**inode **=** inode**->**i\_ino**;**

strncpy**(**dblock**->**files**[**i**].**filename**,** dentry**->**d\_name**.**name**,**

MYFS\_FILENAME\_LEN**);**

mark\_buffer\_dirty**(**bh**);**

brelse**(**bh**);**

/\* Update stats and mark dir and new inode dirty \*/

mark\_inode\_dirty**(**inode**);**

dir**->**i\_mtime **=** dir**->**i\_atime **=** dir**->**i\_ctime **=** current\_time**(**dir**);**

**if** **(**S\_ISDIR**(**mode**))**

inc\_nlink**(**dir**);**

mark\_inode\_dirty**(**dir**);**

/\* setup dentry \*/

d\_instantiate**(**dentry**,** inode**);**

**return** 0**;**

iput**:**

put\_blocks**(**MYFS\_SB**(**sb**),** MYFS\_INODE**(**inode**)->**ei\_block**,** 1**);**

put\_inode**(**MYFS\_SB**(**sb**),** inode**->**i\_ino**);**

iput**(**inode**);**

end**:**

brelse**(**bh**);**

**return** ret**;**

**}**

/\*

\* Remove a link for a file including the reference in the parent directory.

\* If link count is 0, destroy file in this way:

\* - remove the file from its parent directory.

\* - cleanup blocks containing data

\* - cleanup file index block

\* - cleanup inode

\*/

static int myfs\_unlink**(**struct inode **\***dir**,** struct dentry **\***dentry**)**

**{**

struct super\_block **\***sb **=** dir**->**i\_sb**;**

struct myfs\_sb\_info **\***sbi **=** MYFS\_SB**(**sb**);**

struct inode **\***inode **=** d\_inode**(**dentry**);**

struct buffer\_head **\***bh **=** **NULL,** **\***bh2 **=** **NULL;**

struct myfs\_dir\_block **\***dir\_block **=** **NULL;**

struct myfs\_file\_ei\_block **\***file\_block **=** **NULL;**

int i**,** j**,** f\_id **=** **-**1**,** nr\_subs **=** 0**;**

uint32\_t ino **=** inode**->**i\_ino**;**

uint32\_t bno **=** 0**;**

/\* Read parent directory index \*/

bh **=** sb\_bread**(**sb**,** MYFS\_INODE**(**dir**)->**dir\_block**);**

**if** **(!**bh**)**

**return** **-**EIO**;**

dir\_block **=** **(**struct myfs\_dir\_block **\*)** bh**->**b\_data**;**

/\* Search for inode in parent index and get number of subfiles \*/

**for** **(**i **=** 0**;** i **<** MYFS\_MAX\_SUBFILES**;** i**++)** **{**

**if** **(**strncmp**(**dir\_block**->**files**[**i**].**filename**,** dentry**->**d\_name**.**name**,**

MYFS\_FILENAME\_LEN**)** **==** 0**)**

f\_id **=** i**;**

**else** **if** **(**dir\_block**->**files**[**i**].**inode **==** 0**)**

**break;**

**}**

nr\_subs **=** i**;**

/\* Remove file from parent directory \*/

**if** **(**f\_id **!=** MYFS\_MAX\_SUBFILES **-** 1**)**

memmove**(**dir\_block**->**files **+** f\_id**,** dir\_block**->**files **+** f\_id **+** 1**,**

**(**nr\_subs **-** f\_id **-** 1**)** **\*** **sizeof(**struct myfs\_file**));**

memset**(&**dir\_block**->**files**[**nr\_subs **-** 1**],** 0**,** **sizeof(**struct myfs\_file**));**

mark\_buffer\_dirty**(**bh**);**

brelse**(**bh**);**

**if** **(**S\_ISLNK**(**inode**->**i\_mode**))**

**goto** clean\_inode**;**

/\* Update inode stats \*/

dir**->**i\_mtime **=** dir**->**i\_atime **=** dir**->**i\_ctime **=** current\_time**(**dir**);**

**if** **(**S\_ISDIR**(**inode**->**i\_mode**))** **{**

drop\_nlink**(**dir**);**

drop\_nlink**(**inode**);**

**}**

mark\_inode\_dirty**(**dir**);**

**if** **(**inode**->**i\_nlink **>** 1**)** **{**

inode\_dec\_link\_count**(**inode**);**

**return** 0**;**

**}**

/\*

\* Cleanup pointed blocks if unlinking a file. If we fail to read the

\* index block, cleanup inode anyway and lose this file's blocks

\* forever. If we fail to scrub a data block, don't fail (too late

\* anyway), just put the block and continue.

\*/

bno **=** MYFS\_INODE**(**inode**)->**ei\_block**;**

bh **=** sb\_bread**(**sb**,** bno**);**

**if** **(!**bh**)**

**goto** clean\_inode**;**

file\_block **=** **(**struct myfs\_file\_ei\_block **\*)** bh**->**b\_data**;**

**if** **(**S\_ISDIR**(**inode**->**i\_mode**))**

**goto** scrub**;**

**for** **(**i **=** 0**;** i **<** MYFS\_MAX\_EXTENTS**;** i**++)** **{**

char **\***block**;**

**if** **(!**file\_block**->**extents**[**i**].**ee\_start**)**

**break;**

put\_blocks**(**sbi**,** file\_block**->**extents**[**i**].**ee\_start**,**

file\_block**->**extents**[**i**].**ee\_len**);**

/\* Scrub the extent \*/

**for** **(**j **=** 0**;** j **<** file\_block**->**extents**[**i**].**ee\_len**;** j**++)** **{**

bh2 **=** sb\_bread**(**sb**,** file\_block**->**extents**[**i**].**ee\_start **+** j**);**

**if** **(!**bh2**)**

**continue;**

block **=** **(**char **\*)** bh2**->**b\_data**;**

memset**(**block**,** 0**,** MYFS\_BLOCK\_SIZE**);**

mark\_buffer\_dirty**(**bh2**);**

brelse**(**bh2**);**

**}**

**}**

scrub**:**

/\* Scrub index block \*/

memset**(**file\_block**,** 0**,** MYFS\_BLOCK\_SIZE**);**

mark\_buffer\_dirty**(**bh**);**

brelse**(**bh**);**

clean\_inode**:**

/\* Cleanup inode and mark dirty \*/

inode**->**i\_blocks **=** 0**;**

MYFS\_INODE**(**inode**)->**ei\_block **=** 0**;**

inode**->**i\_size **=** 0**;**

i\_uid\_write**(**inode**,** 0**);**

i\_gid\_write**(**inode**,** 0**);**

inode**->**i\_mode **=** 0**;**

inode**->**i\_ctime**.**tv\_sec **=** inode**->**i\_mtime**.**tv\_sec **=** inode**->**i\_atime**.**tv\_sec **=** 0**;**

drop\_nlink**(**inode**);**

mark\_inode\_dirty**(**inode**);**

/\* Free inode and index block from bitmap \*/

put\_blocks**(**sbi**,** bno**,** 1**);**

put\_inode**(**sbi**,** ino**);**

**return** 0**;**

**}**

static int myfs\_rename**(**struct inode **\***old\_dir**,**

struct dentry **\***old\_dentry**,**

struct inode **\***new\_dir**,**

struct dentry **\***new\_dentry**,**

unsigned int flags**)**

**{**

struct super\_block **\***sb **=** old\_dir**->**i\_sb**;**

struct myfs\_inode\_info **\***ci\_old **=** MYFS\_INODE**(**old\_dir**);**

struct myfs\_inode\_info **\***ci\_new **=** MYFS\_INODE**(**new\_dir**);**

struct inode **\***src **=** d\_inode**(**old\_dentry**);**

struct buffer\_head **\***bh\_old **=** **NULL,** **\***bh\_new **=** **NULL;**

struct myfs\_dir\_block **\***dir\_block **=** **NULL;**

int i**,** f\_id **=** **-**1**,** new\_pos **=** **-**1**,** ret**,** nr\_subs**,** f\_pos **=** **-**1**;**

/\* fail with these unsupported flags \*/

**if** **(**flags **&** **(**RENAME\_EXCHANGE **|** RENAME\_WHITEOUT**))**

**return** **-**EINVAL**;**

/\* Check if filename is not too long \*/

**if** **(**strlen**(**new\_dentry**->**d\_name**.**name**)** **>** MYFS\_FILENAME\_LEN**)**

**return** **-**ENAMETOOLONG**;**

/\* Fail if new\_dentry exists or if new\_dir is full \*/

bh\_new **=** sb\_bread**(**sb**,** ci\_new**->**dir\_block**);**

**if** **(!**bh\_new**)**

**return** **-**EIO**;**

dir\_block **=** **(**struct myfs\_dir\_block **\*)** bh\_new**->**b\_data**;**

**for** **(**i **=** 0**;** i **<** MYFS\_MAX\_SUBFILES**;** i**++)** **{**

/\* if old\_dir == new\_dir, save the renamed file position \*/

**if** **(**new\_dir **==** old\_dir**)** **{**

**if** **(**strncmp**(**dir\_block**->**files**[**i**].**filename**,** old\_dentry**->**d\_name**.**name**,**

MYFS\_FILENAME\_LEN**)** **==** 0**)**

f\_pos **=** i**;**

**}**

**if** **(**strncmp**(**dir\_block**->**files**[**i**].**filename**,** new\_dentry**->**d\_name**.**name**,**

MYFS\_FILENAME\_LEN**)** **==** 0**)** **{**

ret **=** **-**EEXIST**;**

**goto** relse\_new**;**

**}**

**if** **(**new\_pos **<** 0 **&&** dir\_block**->**files**[**i**].**inode **==** 0**)**

new\_pos **=** i**;**

**}**

/\* if old\_dir == new\_dir, just rename entry \*/

**if** **(**old\_dir **==** new\_dir**)** **{**

strncpy**(**dir\_block**->**files**[**f\_pos**].**filename**,** new\_dentry**->**d\_name**.**name**,**

MYFS\_FILENAME\_LEN**);**

mark\_buffer\_dirty**(**bh\_new**);**

ret **=** 0**;**

**goto** relse\_new**;**

**}**

/\* If new directory is empty, fail \*/

**if** **(**new\_pos **<** 0**)** **{**

ret **=** **-**EMLINK**;**

**goto** relse\_new**;**

**}**

/\* insert in new parent directory \*/

dir\_block**->**files**[**new\_pos**].**inode **=** src**->**i\_ino**;**

strncpy**(**dir\_block**->**files**[**new\_pos**].**filename**,** new\_dentry**->**d\_name**.**name**,**

MYFS\_FILENAME\_LEN**);**

mark\_buffer\_dirty**(**bh\_new**);**

brelse**(**bh\_new**);**

/\* Update new parent inode metadata \*/

new\_dir**->**i\_atime **=** new\_dir**->**i\_ctime **=** new\_dir**->**i\_mtime **=**

current\_time**(**new\_dir**);**

**if** **(**S\_ISDIR**(**src**->**i\_mode**))**

inc\_nlink**(**new\_dir**);**

mark\_inode\_dirty**(**new\_dir**);**

/\* remove target from old parent directory \*/

bh\_old **=** sb\_bread**(**sb**,** ci\_old**->**dir\_block**);**

**if** **(!**bh\_old**)**

**return** **-**EIO**;**

dir\_block **=** **(**struct myfs\_dir\_block **\*)** bh\_old**->**b\_data**;**

/\* Search for inode in old directory and number of subfiles \*/

**for** **(**i **=** 0**;** MYFS\_MAX\_SUBFILES**;** i**++)** **{**

**if** **(**dir\_block**->**files**[**i**].**inode **==** src**->**i\_ino**)**

f\_id **=** i**;**

**else** **if** **(**dir\_block**->**files**[**i**].**inode **==** 0**)**

**break;**

**}**

nr\_subs **=** i**;**

/\* Remove file from old parent directory \*/

**if** **(**f\_id **!=** MYFS\_MAX\_SUBFILES **-** 1**)**

memmove**(**dir\_block**->**files **+** f\_id**,** dir\_block**->**files **+** f\_id **+** 1**,**

**(**nr\_subs **-** f\_id **-** 1**)** **\*** **sizeof(**struct myfs\_file**));**

memset**(&**dir\_block**->**files**[**nr\_subs **-** 1**],** 0**,** **sizeof(**struct myfs\_file**));**

mark\_buffer\_dirty**(**bh\_old**);**

brelse**(**bh\_old**);**

/\* Update old parent inode metadata \*/

old\_dir**->**i\_atime **=** old\_dir**->**i\_ctime **=** old\_dir**->**i\_mtime **=**

current\_time**(**old\_dir**);**

**if** **(**S\_ISDIR**(**src**->**i\_mode**))**

drop\_nlink**(**old\_dir**);**

mark\_inode\_dirty**(**old\_dir**);**

**return** 0**;**

relse\_new**:**

brelse**(**bh\_new**);**

**return** ret**;**

**}**

static int myfs\_mkdir**(**struct inode **\***dir**,**

struct dentry **\***dentry**,**

umode\_t mode**)**

**{**

**return** myfs\_create**(**dir**,** dentry**,** mode **|** S\_IFDIR**,** 0**);**

**}**

static int myfs\_rmdir**(**struct inode **\***dir**,** struct dentry **\***dentry**)**

**{**

struct super\_block **\***sb **=** dir**->**i\_sb**;**

struct inode **\***inode **=** d\_inode**(**dentry**);**

struct buffer\_head **\***bh**;**

struct myfs\_dir\_block **\***dblock**;**

/\* If the directory is not empty, fail \*/

**if** **(**inode**->**i\_nlink **>** 2**)**

**return** **-**ENOTEMPTY**;**

bh **=** sb\_bread**(**sb**,** MYFS\_INODE**(**inode**)->**dir\_block**);**

**if** **(!**bh**)**

**return** **-**EIO**;**

dblock **=** **(**struct myfs\_dir\_block **\*)** bh**->**b\_data**;**

**if** **(**dblock**->**files**[**0**].**inode **!=** 0**)** **{**

brelse**(**bh**);**

**return** **-**ENOTEMPTY**;**

**}**

brelse**(**bh**);**

/\* Remove directory with unlink \*/

**return** myfs\_unlink**(**dir**,** dentry**);**

**}**

static int myfs\_link**(**struct dentry **\***old\_dentry**,**

struct inode **\***dir**,**

struct dentry **\***dentry**)**

**{**

struct inode **\***inode **=** d\_inode**(**old\_dentry**);**

struct super\_block **\***sb **=** inode**->**i\_sb**;**

struct myfs\_inode\_info **\***ci\_dir **=** MYFS\_INODE**(**dir**);**

struct myfs\_dir\_block **\***dir\_block**;**

struct buffer\_head **\***bh**;**

int f\_pos **=** **-**1**,** ret **=** 0**,** i **=** 0**;**

bh **=** sb\_bread**(**sb**,** ci\_dir**->**dir\_block**);**

**if** **(!**bh**)**

**return** **-**EIO**;**

dir\_block **=** **(**struct myfs\_dir\_block **\*)** bh**->**b\_data**;**

**if** **(**dir\_block**->**files**[**MYFS\_MAX\_SUBFILES **-** 1**].**inode **!=** 0**)** **{**

ret **=** **-**EMLINK**;**

printk**(**KERN\_INFO "directory is full"**);**

**goto** end**;**

**}**

**for** **(**i **=** 0**;** i **<** MYFS\_MAX\_SUBFILES**;** i**++)** **{**

**if** **(**dir\_block**->**files**[**i**].**inode **==** 0**)** **{**

f\_pos **=** i**;**

**break;**

**}**

**}**

dir\_block**->**files**[**f\_pos**].**inode **=** inode**->**i\_ino**;**

strncpy**(**dir\_block**->**files**[**f\_pos**].**filename**,** dentry**->**d\_name**.**name**,**

MYFS\_FILENAME\_LEN**);**

mark\_buffer\_dirty**(**bh**);**

inode\_inc\_link\_count**(**inode**);**

d\_instantiate**(**dentry**,** inode**);**

end**:**

brelse**(**bh**);**

**return** ret**;**

**}**

static int myfs\_symlink**(**struct inode **\***dir**,**

struct dentry **\***dentry**,**

const char **\***symname**)**

**{**

struct super\_block **\***sb **=** dir**->**i\_sb**;**

unsigned int l **=** strlen**(**symname**)** **+** 1**;**

struct inode **\***inode **=** myfs\_new\_inode**(**dir**,** S\_IFLNK **|** S\_IRWXUGO**);**

struct myfs\_inode\_info **\***ci **=** MYFS\_INODE**(**inode**);**

struct myfs\_inode\_info **\***ci\_dir **=** MYFS\_INODE**(**dir**);**

struct myfs\_dir\_block **\***dir\_block**;**

struct buffer\_head **\***bh**;**

int f\_pos **=** 0**,** i **=** 0**;**

/\* Check if symlink content is not too long \*/

**if** **(**l **>** **sizeof(**ci**->**i\_data**))**

**return** **-**ENAMETOOLONG**;**

/\* fill directory data block \*/

bh **=** sb\_bread**(**sb**,** ci\_dir**->**dir\_block**);**

**if** **(!**bh**)**

**return** **-**EIO**;**

dir\_block **=** **(**struct myfs\_dir\_block **\*)** bh**->**b\_data**;**

**if** **(**dir\_block**->**files**[**MYFS\_MAX\_SUBFILES **-** 1**].**inode **!=** 0**)** **{**

printk**(**KERN\_INFO "directory is full\n"**);**

**return** **-**EMLINK**;**

**}**

**for** **(**i **=** 0**;** i **<** MYFS\_MAX\_SUBFILES**;** i**++)** **{**

**if** **(**dir\_block**->**files**[**i**].**inode **==** 0**)** **{**

f\_pos **=** i**;**

**break;**

**}**

**}**

dir\_block**->**files**[**f\_pos**].**inode **=** inode**->**i\_ino**;**

strncpy**(**dir\_block**->**files**[**f\_pos**].**filename**,** dentry**->**d\_name**.**name**,**

MYFS\_FILENAME\_LEN**);**

mark\_buffer\_dirty**(**bh**);**

brelse**(**bh**);**

inode**->**i\_link **=** **(**char **\*)** ci**->**i\_data**;**

memcpy**(**inode**->**i\_link**,** symname**,** l**);**

inode**->**i\_size **=** l **-** 1**;**

mark\_inode\_dirty**(**inode**);**

d\_instantiate**(**dentry**,** inode**);**

**return** 0**;**

**}**

static const char **\***myfs\_get\_link**(**struct dentry **\***dentry**,**

struct inode **\***inode**,**

struct delayed\_call **\***done**)**

**{**

**return** inode**->**i\_link**;**

**}**

static const struct inode\_operations myfs\_inode\_ops **=** **{**

**.**lookup **=** myfs\_lookup**,**

**.**create **=** myfs\_create**,**

**.**unlink **=** myfs\_unlink**,**

**.**mkdir **=** myfs\_mkdir**,**

**.**rmdir **=** myfs\_rmdir**,**

**.**rename **=** myfs\_rename**,**

**.**link **=** myfs\_link**,**

**.**symlink **=** myfs\_symlink**,**

**};**

static const struct inode\_operations symlink\_inode\_ops **=** **{**

**.**get\_link **=** myfs\_get\_link**,**

**};**

Super.c

#define pr\_fmt(fmt) KBUILD\_MODNAME ": " fmt

#include <linux/buffer\_head.h>

#include <linux/fs.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/slab.h>

#include <linux/statfs.h>

#include "myfs.h"

static struct kmem\_cache **\***myfs\_inode\_cache**;**

int myfs\_init\_inode\_cache**(**void**)**

**{**

myfs\_inode\_cache **=** kmem\_cache\_create**(**

"myfs\_cache"**,** **sizeof(**struct myfs\_inode\_info**),** 0**,** 0**,** **NULL);**

**if** **(!**myfs\_inode\_cache**)**

**return** **-**ENOMEM**;**

**return** 0**;**

**}**

void myfs\_destroy\_inode\_cache**(**void**)**

**{**

kmem\_cache\_destroy**(**myfs\_inode\_cache**);**

**}**

static struct inode **\***myfs\_alloc\_inode**(**struct super\_block **\***sb**)**

**{**

struct myfs\_inode\_info **\***ci **=**

kmem\_cache\_alloc**(**myfs\_inode\_cache**,** GFP\_KERNEL**);**

**if** **(!**ci**)**

**return** **NULL;**

inode\_init\_once**(&**ci**->**vfs\_inode**);**

**return** **&**ci**->**vfs\_inode**;**

**}**

static void myfs\_destroy\_inode**(**struct inode **\***inode**)**

**{**

struct myfs\_inode\_info **\***ci **=** MYFS\_INODE**(**inode**);**

kmem\_cache\_free**(**myfs\_inode\_cache**,** ci**);**

**}**

static int myfs\_write\_inode**(**struct inode **\***inode**,**

struct writeback\_control **\***wbc**)**

**{**

struct myfs\_inode **\***disk\_inode**;**

struct myfs\_inode\_info **\***ci **=** MYFS\_INODE**(**inode**);**

struct super\_block **\***sb **=** inode**->**i\_sb**;**

struct myfs\_sb\_info **\***sbi **=** MYFS\_SB**(**sb**);**

struct buffer\_head **\***bh**;**

uint32\_t ino **=** inode**->**i\_ino**;**

uint32\_t inode\_block **=** **(**ino **/** MYFS\_INODES\_PER\_BLOCK**)** **+** 1**;**

uint32\_t inode\_shift **=** ino **%** MYFS\_INODES\_PER\_BLOCK**;**

**if** **(**ino **>=** sbi**->**nr\_inodes**)**

**return** 0**;**

bh **=** sb\_bread**(**sb**,** inode\_block**);**

**if** **(!**bh**)**

**return** **-**EIO**;**

disk\_inode **=** **(**struct myfs\_inode **\*)** bh**->**b\_data**;**

disk\_inode **+=** inode\_shift**;**

/\* update the mode using what the generic inode has \*/

disk\_inode**->**i\_mode **=** inode**->**i\_mode**;**

disk\_inode**->**i\_uid **=** i\_uid\_read**(**inode**);**

disk\_inode**->**i\_gid **=** i\_gid\_read**(**inode**);**

disk\_inode**->**i\_size **=** inode**->**i\_size**;**

disk\_inode**->**i\_ctime **=** inode**->**i\_ctime**.**tv\_sec**;**

disk\_inode**->**i\_atime **=** inode**->**i\_atime**.**tv\_sec**;**

disk\_inode**->**i\_mtime **=** inode**->**i\_mtime**.**tv\_sec**;**

disk\_inode**->**i\_blocks **=** inode**->**i\_blocks**;**

disk\_inode**->**i\_nlink **=** inode**->**i\_nlink**;**

disk\_inode**->**ei\_block **=** ci**->**ei\_block**;**

strncpy**(**disk\_inode**->**i\_data**,** ci**->**i\_data**,** **sizeof(**ci**->**i\_data**));**

mark\_buffer\_dirty**(**bh**);**

sync\_dirty\_buffer**(**bh**);**

brelse**(**bh**);**

**return** 0**;**

**}**

static void myfs\_put\_super**(**struct super\_block **\***sb**)**

**{**

struct myfs\_sb\_info **\***sbi **=** MYFS\_SB**(**sb**);**

**if** **(**sbi**)** **{**

kfree**(**sbi**->**ifree\_bitmap**);**

kfree**(**sbi**->**bfree\_bitmap**);**

kfree**(**sbi**);**

**}**

**}**

static int myfs\_sync\_fs**(**struct super\_block **\***sb**,** int wait**)**

**{**

struct myfs\_sb\_info **\***sbi **=** MYFS\_SB**(**sb**);**

struct myfs\_sb\_info **\***disk\_sb**;**

int i**;**

/\* Flush superblock \*/

struct buffer\_head **\***bh **=** sb\_bread**(**sb**,** 0**);**

**if** **(!**bh**)**

**return** **-**EIO**;**

disk\_sb **=** **(**struct myfs\_sb\_info **\*)** bh**->**b\_data**;**

disk\_sb**->**nr\_blocks **=** sbi**->**nr\_blocks**;**

disk\_sb**->**nr\_inodes **=** sbi**->**nr\_inodes**;**

disk\_sb**->**nr\_istore\_blocks **=** sbi**->**nr\_istore\_blocks**;**

disk\_sb**->**nr\_ifree\_blocks **=** sbi**->**nr\_ifree\_blocks**;**

disk\_sb**->**nr\_bfree\_blocks **=** sbi**->**nr\_bfree\_blocks**;**

disk\_sb**->**nr\_free\_inodes **=** sbi**->**nr\_free\_inodes**;**

disk\_sb**->**nr\_free\_blocks **=** sbi**->**nr\_free\_blocks**;**

mark\_buffer\_dirty**(**bh**);**

**if** **(**wait**)**

sync\_dirty\_buffer**(**bh**);**

brelse**(**bh**);**

/\* Flush free inodes bitmask \*/

**for** **(**i **=** 0**;** i **<** sbi**->**nr\_ifree\_blocks**;** i**++)** **{**

int idx **=** sbi**->**nr\_istore\_blocks **+** i **+** 1**;**

bh **=** sb\_bread**(**sb**,** idx**);**

**if** **(!**bh**)**

**return** **-**EIO**;**

memcpy**(**bh**->**b\_data**,** **(**void **\*)** sbi**->**ifree\_bitmap **+** i **\*** MYFS\_BLOCK\_SIZE**,**

MYFS\_BLOCK\_SIZE**);**

mark\_buffer\_dirty**(**bh**);**

**if** **(**wait**)**

sync\_dirty\_buffer**(**bh**);**

brelse**(**bh**);**

**}**

/\* Flush free blocks bitmask \*/

**for** **(**i **=** 0**;** i **<** sbi**->**nr\_bfree\_blocks**;** i**++)** **{**

int idx **=** sbi**->**nr\_istore\_blocks **+** sbi**->**nr\_ifree\_blocks **+** i **+** 1**;**

bh **=** sb\_bread**(**sb**,** idx**);**

**if** **(!**bh**)**

**return** **-**EIO**;**

memcpy**(**bh**->**b\_data**,** **(**void **\*)** sbi**->**bfree\_bitmap **+** i **\*** MYFS\_BLOCK\_SIZE**,**

MYFS\_BLOCK\_SIZE**);**

mark\_buffer\_dirty**(**bh**);**

**if** **(**wait**)**

sync\_dirty\_buffer**(**bh**);**

brelse**(**bh**);**

**}**

**return** 0**;**

**}**

static int myfs\_statfs**(**struct dentry **\***dentry**,** struct kstatfs **\***stat**)**

**{**

struct super\_block **\***sb **=** dentry**->**d\_sb**;**

struct myfs\_sb\_info **\***sbi **=** MYFS\_SB**(**sb**);**

stat**->**f\_type **=** MYFS\_MAGIC**;**

stat**->**f\_bsize **=** MYFS\_BLOCK\_SIZE**;**

stat**->**f\_blocks **=** sbi**->**nr\_blocks**;**

stat**->**f\_bfree **=** sbi**->**nr\_free\_blocks**;**

stat**->**f\_bavail **=** sbi**->**nr\_free\_blocks**;**

stat**->**f\_files **=** sbi**->**nr\_inodes **-** sbi**->**nr\_free\_inodes**;**

stat**->**f\_ffree **=** sbi**->**nr\_free\_inodes**;**

stat**->**f\_namelen **=** MYFS\_FILENAME\_LEN**;**

**return** 0**;**

**}**

static struct super\_operations myfs\_super\_ops **=** **{**

**.**put\_super **=** myfs\_put\_super**,**

**.**alloc\_inode **=** myfs\_alloc\_inode**,**

**.**destroy\_inode **=** myfs\_destroy\_inode**,**

**.**write\_inode **=** myfs\_write\_inode**,**

**.**sync\_fs **=** myfs\_sync\_fs**,**

**.**statfs **=** myfs\_statfs**,**

**};**

/\* Fill the struct superblock from partition superblock \*/

int myfs\_fill\_super**(**struct super\_block **\***sb**,** void **\***data**,** int silent**)**

**{**

struct buffer\_head **\***bh **=** **NULL;**

struct myfs\_sb\_info **\***csb **=** **NULL;**

struct myfs\_sb\_info **\***sbi **=** **NULL;**

struct inode **\***root\_inode **=** **NULL;**

int ret **=** 0**,** i**;**

/\* Init sb \*/

sb**->**s\_magic **=** MYFS\_MAGIC**;**

sb\_set\_blocksize**(**sb**,** MYFS\_BLOCK\_SIZE**);**

sb**->**s\_maxbytes **=** MYFS\_MAX\_FILESIZE**;**

sb**->**s\_op **=** **&**myfs\_super\_ops**;**

/\* Read sb from disk \*/

bh **=** sb\_bread**(**sb**,** MYFS\_SB\_BLOCK\_NR**);**

**if** **(!**bh**)**

**return** **-**EIO**;**

csb **=** **(**struct myfs\_sb\_info **\*)** bh**->**b\_data**;**

/\* Check magic number \*/

**if** **(**csb**->**magic **!=** sb**->**s\_magic**)** **{**

pr\_err**(**"Wrong magic number\n"**);**

ret **=** **-**EINVAL**;**

**goto** release**;**

**}**

/\* Alloc sb\_info \*/

sbi **=** kzalloc**(sizeof(**struct myfs\_sb\_info**),** GFP\_KERNEL**);**

**if** **(!**sbi**)** **{**

ret **=** **-**ENOMEM**;**

**goto** release**;**

**}**

sbi**->**nr\_blocks **=** csb**->**nr\_blocks**;**

sbi**->**nr\_inodes **=** csb**->**nr\_inodes**;**

sbi**->**nr\_istore\_blocks **=** csb**->**nr\_istore\_blocks**;**

sbi**->**nr\_ifree\_blocks **=** csb**->**nr\_ifree\_blocks**;**

sbi**->**nr\_bfree\_blocks **=** csb**->**nr\_bfree\_blocks**;**

sbi**->**nr\_free\_inodes **=** csb**->**nr\_free\_inodes**;**

sbi**->**nr\_free\_blocks **=** csb**->**nr\_free\_blocks**;**

sb**->**s\_fs\_info **=** sbi**;**

brelse**(**bh**);**

/\* Alloc and copy ifree\_bitmap \*/

sbi**->**ifree\_bitmap **=**

kzalloc**(**sbi**->**nr\_ifree\_blocks **\*** MYFS\_BLOCK\_SIZE**,** GFP\_KERNEL**);**

**if** **(!**sbi**->**ifree\_bitmap**)** **{**

ret **=** **-**ENOMEM**;**

**goto** free\_sbi**;**

**}**

**for** **(**i **=** 0**;** i **<** sbi**->**nr\_ifree\_blocks**;** i**++)** **{**

int idx **=** sbi**->**nr\_istore\_blocks **+** i **+** 1**;**

bh **=** sb\_bread**(**sb**,** idx**);**

**if** **(!**bh**)** **{**

ret **=** **-**EIO**;**

**goto** free\_ifree**;**

**}**

memcpy**((**void **\*)** sbi**->**ifree\_bitmap **+** i **\*** MYFS\_BLOCK\_SIZE**,** bh**->**b\_data**,**

MYFS\_BLOCK\_SIZE**);**

brelse**(**bh**);**

**}**

/\* Alloc and copy bfree\_bitmap \*/

sbi**->**bfree\_bitmap **=**

kzalloc**(**sbi**->**nr\_bfree\_blocks **\*** MYFS\_BLOCK\_SIZE**,** GFP\_KERNEL**);**

**if** **(!**sbi**->**bfree\_bitmap**)** **{**

ret **=** **-**ENOMEM**;**

**goto** free\_ifree**;**

**}**

**for** **(**i **=** 0**;** i **<** sbi**->**nr\_bfree\_blocks**;** i**++)** **{**

int idx **=** sbi**->**nr\_istore\_blocks **+** sbi**->**nr\_ifree\_blocks **+** i **+** 1**;**

bh **=** sb\_bread**(**sb**,** idx**);**

**if** **(!**bh**)** **{**

ret **=** **-**EIO**;**

**goto** free\_bfree**;**

**}**

memcpy**((**void **\*)** sbi**->**bfree\_bitmap **+** i **\*** MYFS\_BLOCK\_SIZE**,** bh**->**b\_data**,**

MYFS\_BLOCK\_SIZE**);**

brelse**(**bh**);**

**}**

/\* Create root inode \*/

root\_inode **=** myfs\_iget**(**sb**,** 0**);**

**if** **(**IS\_ERR**(**root\_inode**))** **{**

ret **=** PTR\_ERR**(**root\_inode**);**

**goto** free\_bfree**;**

**}**

inode\_init\_owner**(**root\_inode**,** **NULL,** root\_inode**->**i\_mode**);**

sb**->**s\_root **=** d\_make\_root**(**root\_inode**);**

**if** **(!**sb**->**s\_root**)** **{**

ret **=** **-**ENOMEM**;**

**goto** iput**;**

**}**

**return** 0**;**

iput**:**

iput**(**root\_inode**);**

free\_bfree**:**

kfree**(**sbi**->**bfree\_bitmap**);**

free\_ifree**:**

kfree**(**sbi**->**ifree\_bitmap**);**

free\_sbi**:**

kfree**(**sbi**);**

release**:**

brelse**(**bh**);**

**return** ret**;**

**}**

File.c

#define pr\_fmt(fmt) "myfs: " fmt

#include <linux/buffer\_head.h>

#include <linux/fs.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/mpage.h>

#include "bitmap.h"

#include "myfs.h"

/\*

\* Map the buffer\_head passed in argument with the iblock-th block of the file

\* represented by inode. If the requested block is not allocated and create is

\* true, allocate a new block on disk and map it.

\*/

static int myfs\_file\_get\_block**(**struct inode **\***inode**,**

sector\_t iblock**,**

struct buffer\_head **\***bh\_result**,**

int create**)**

**{**

struct super\_block **\***sb **=** inode**->**i\_sb**;**

struct myfs\_sb\_info **\***sbi **=** MYFS\_SB**(**sb**);**

struct myfs\_inode\_info **\***ci **=** MYFS\_INODE**(**inode**);**

struct myfs\_file\_ei\_block **\***index**;**

struct buffer\_head **\***bh\_index**;**

bool alloc **=** false**;**

int ret **=** 0**,** bno**;**

uint32\_t extent**;**

/\* If block number exceeds filesize, fail \*/

**if** **(**iblock **>=** MYFS\_MAX\_BLOCKS\_PER\_EXTENT **\*** MYFS\_MAX\_EXTENTS**)**

**return** **-**EFBIG**;**

/\* Read directory block from disk \*/

bh\_index **=** sb\_bread**(**sb**,** ci**->**dir\_block**);**

**if** **(!**bh\_index**)**

**return** **-**EIO**;**

index **=** **(**struct myfs\_file\_ei\_block **\*)** bh\_index**->**b\_data**;**

extent **=** myfs\_ext\_search**(**index**,** iblock**);**

**if** **(**extent **==** **-**1**)** **{**

ret **=** **-**EFBIG**;**

**goto** brelse\_index**;**

**}**

/\*

\* Check if iblock is already allocated. If not and create is true,

\* allocate it. Else, get the physical block number.

\*/

**if** **(**index**->**extents**[**extent**].**ee\_start **==** 0**)** **{**

**if** **(!**create**)**

**return** 0**;**

bno **=** get\_free\_blocks**(**sbi**,** 8**);**

**if** **(!**bno**)** **{**

ret **=** **-**ENOSPC**;**

**goto** brelse\_index**;**

**}**

index**->**extents**[**extent**].**ee\_start **=** bno**;**

index**->**extents**[**extent**].**ee\_len **=** 8**;**

index**->**extents**[**extent**].**ee\_block **=**

extent **?** index**->**extents**[**extent **-** 1**].**ee\_block **+**

index**->**extents**[**extent **-** 1**].**ee\_len

**:** 0**;**

alloc **=** true**;**

**}** **else** **{**

bno **=** index**->**extents**[**extent**].**ee\_start **+** iblock **-**

index**->**extents**[**extent**].**ee\_block**;**

**}**

/\* Map the physical block to to the given buffer\_head \*/

map\_bh**(**bh\_result**,** sb**,** bno**);**

brelse\_index**:**

brelse**(**bh\_index**);**

**return** ret**;**

**}**

/\*

\* Called by the page cache to read a page from the physical disk and map it in

\* memory.

\*/

static int myfs\_readpage**(**struct file **\***file**,** struct page **\***page**)**

**{**

**return** mpage\_readpage**(**page**,** myfs\_file\_get\_block**);**

**}**

/\*

\* Called by the page cache to write a dirty page to the physical disk (when

\* sync is called or when memory is needed).

\*/

static int myfs\_writepage**(**struct page **\***page**,** struct writeback\_control **\***wbc**)**

**{**

**return** block\_write\_full\_page**(**page**,** myfs\_file\_get\_block**,** wbc**);**

**}**

/\*

\* Called by the VFS when a write() syscall occurs on file before writing the

\* data in the page cache. This functions checks if the write will be able to

\* complete and allocates the necessary blocks through block\_write\_begin().

\*/

static int myfs\_write\_begin**(**struct file **\***file**,**

struct address\_space **\***mapping**,**

loff\_t pos**,**

unsigned int len**,**

unsigned int flags**,**

struct page **\*\***pagep**,**

void **\*\***fsdata**)**

**{**

struct myfs\_sb\_info **\***sbi **=** MYFS\_SB**(**file**->**f\_inode**->**i\_sb**);**

int err**;**

uint32\_t nr\_allocs **=** 0**;**

/\* Check if the write can be completed (enough space?) \*/

**if** **(**pos **+** len **>** MYFS\_MAX\_FILESIZE**)**

**return** **-**ENOSPC**;**

nr\_allocs **=** max**(**pos **+** len**,** file**->**f\_inode**->**i\_size**)** **/** MYFS\_BLOCK\_SIZE**;**

**if** **(**nr\_allocs **>** file**->**f\_inode**->**i\_blocks **-** 1**)**

nr\_allocs **-=** file**->**f\_inode**->**i\_blocks **-** 1**;**

**else**

nr\_allocs **=** 0**;**

**if** **(**nr\_allocs **>** sbi**->**nr\_free\_blocks**)**

**return** **-**ENOSPC**;**

/\* prepare the write \*/

err **=** block\_write\_begin**(**mapping**,** pos**,** len**,** flags**,** pagep**,**

myfs\_file\_get\_block**);**

/\* if this failed, reclaim newly allocated blocks \*/

**if** **(**err **<** 0**)**

pr\_err**(**"newly allocated blocks reclaim not implemented yet\n"**);**

**return** err**;**

**}**

/\*

\* Called by the VFS after writing data from a write() syscall to the page

\* cache. This functions updates inode metadata and truncates the file if

\* necessary.

\*/

static int myfs\_write\_end**(**struct file **\***file**,**

struct address\_space **\***mapping**,**

loff\_t pos**,**

unsigned int len**,**

unsigned int copied**,**

struct page **\***page**,**

void **\***fsdata**)**

**{**

struct inode **\***inode **=** file**->**f\_inode**;**

struct myfs\_inode\_info **\***ci **=** MYFS\_INODE**(**inode**);**

struct super\_block **\***sb **=** inode**->**i\_sb**;**

uint32\_t nr\_blocks\_old**;**

/\* Complete the write() \*/

int ret **=** generic\_write\_end**(**file**,** mapping**,** pos**,** len**,** copied**,** page**,** fsdata**);**

**if** **(**ret **<** len**)** **{**

pr\_err**(**"wrote less than requested."**);**

**return** ret**;**

**}**

nr\_blocks\_old **=** inode**->**i\_blocks**;**

/\* Update inode metadata \*/

inode**->**i\_blocks **=** inode**->**i\_size **/** MYFS\_BLOCK\_SIZE **+** 2**;**

inode**->**i\_mtime **=** inode**->**i\_ctime **=** current\_time**(**inode**);**

mark\_inode\_dirty**(**inode**);**

/\* If file is smaller than before, free unused blocks \*/

**if** **(**nr\_blocks\_old **>** inode**->**i\_blocks**)** **{**

int i**;**

struct buffer\_head **\***bh\_index**;**

struct myfs\_file\_ei\_block **\***index**;**

uint32\_t first\_ext**;**

/\* Free unused blocks from page cache \*/

truncate\_pagecache**(**inode**,** inode**->**i\_size**);**

/\* Read ei\_block to remove unused blocks \*/

bh\_index **=** sb\_bread**(**sb**,** ci**->**ei\_block**);**

**if** **(!**bh\_index**)** **{**

pr\_err**(**"failed truncating '%s'. we just lost %llu blocks\n"**,**

file**->**f\_path**.**dentry**->**d\_name**.**name**,**

nr\_blocks\_old **-** inode**->**i\_blocks**);**

**goto** end**;**

**}**

index **=** **(**struct myfs\_file\_ei\_block **\*)** bh\_index**->**b\_data**;**

first\_ext **=** myfs\_ext\_search**(**index**,** inode**->**i\_blocks **-** 1**);**

/\* Reserve unused block in last extent \*/

**if** **(**inode**->**i\_blocks **-** 1 **!=** index**->**extents**[**first\_ext**].**ee\_block**)**

first\_ext**++;**

**for** **(**i **=** first\_ext**;** i **<** MYFS\_MAX\_EXTENTS**;** i**++)** **{**

**if** **(!**index**->**extents**[**i**].**ee\_start**)**

**break;**

put\_blocks**(**MYFS\_SB**(**sb**),** index**->**extents**[**i**].**ee\_start**,**

index**->**extents**[**i**].**ee\_len**);**

memset**(&**index**->**extents**[**i**],** 0**,** **sizeof(**struct myfs\_extent**));**

**}**

mark\_buffer\_dirty**(**bh\_index**);**

brelse**(**bh\_index**);**

**}**

end**:**

**return** ret**;**

**}**

const struct address\_space\_operations myfs\_aops **=** **{**

**.**readpage **=** myfs\_readpage**,**

**.**writepage **=** myfs\_writepage**,**

**.**write\_begin **=** myfs\_write\_begin**,**

**.**write\_end **=** myfs\_write\_end**,**

**};**

const struct file\_operations myfs\_file\_ops **=** **{**

**.**llseek **=** generic\_file\_llseek**,**

**.**owner **=** THIS\_MODULE**,**

**.**read\_iter **=** generic\_file\_read\_iter**,**

**.**write\_iter **=** generic\_file\_write\_iter**,**

**.**fsync **=** generic\_file\_fsync**,**

**};**

Dir.c

#define pr\_fmt(fmt) "myfs: " fmt

#include <linux/buffer\_head.h>

#include <linux/fs.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include "myfs.h"

/\*

\* Iterate over the files contained in dir and commit them in ctx.

\* This function is called by the VFS while ctx->pos changes.

\* Return 0 on success.

\*/

static int myfs\_iterate**(**struct file **\***dir**,** struct dir\_context **\***ctx**)**

**{**

struct inode **\***inode **=** file\_inode**(**dir**);**

struct myfs\_inode\_info **\***ci **=** MYFS\_INODE**(**inode**);**

struct super\_block **\***sb **=** inode**->**i\_sb**;**

struct buffer\_head **\***bh **=** **NULL;**

struct myfs\_dir\_block **\***dblock **=** **NULL;**

struct myfs\_file **\***f **=** **NULL;**

int i**;**

/\* Check that dir is a directory \*/

**if** **(!**S\_ISDIR**(**inode**->**i\_mode**))**

**return** **-**ENOTDIR**;**

/\*

\* Check that ctx->pos is not bigger than what we can handle (including

\* . and ..)

\*/

**if** **(**ctx**->**pos **>** MYFS\_MAX\_SUBFILES **+** 2**)**

**return** 0**;**

/\* Commit . and .. to ctx \*/

**if** **(!**dir\_emit\_dots**(**dir**,** ctx**))**

**return** 0**;**

/\* Read the directory index block on disk \*/

bh **=** sb\_bread**(**sb**,** ci**->**dir\_block**);**

**if** **(!**bh**)**

**return** **-**EIO**;**

dblock **=** **(**struct myfs\_dir\_block **\*)** bh**->**b\_data**;**

/\* Iterate over the index block and commit subfiles \*/

**for** **(**i **=** ctx**->**pos **-** 2**;** i **<** MYFS\_MAX\_SUBFILES**;** i**++)** **{**

f **=** **&**dblock**->**files**[**i**];**

**if** **(!**f**->**inode**)**

**break;**

**if** **(!**dir\_emit**(**ctx**,** f**->**filename**,** MYFS\_FILENAME\_LEN**,** f**->**inode**,**

DT\_UNKNOWN**))**

**break;**

ctx**->**pos**++;**

**}**

brelse**(**bh**);**

**return** 0**;**

**}**

const struct file\_operations myfs\_dir\_ops **=** **{**

**.**owner **=** THIS\_MODULE**,**

**.**iterate\_shared **=** myfs\_iterate**,**

**};**

Myfs.h

#ifndef MYFS\_H

#define MYFS\_H

#define MYFS\_MAGIC 0xDEADCELL

#define MYFS\_SB\_BLOCK\_NR 0

#define MYFS\_BLOCK\_SIZE (1 << 12) /\* 4 KiB \*/

#define MYFS\_MAX\_EXTENTS \

MYFS\_BLOCK\_SIZE / sizeof(struct myfs\_extent)

#define MYFS\_MAX\_BLOCKS\_PER\_EXTENT 8 /\* It can be ~(uint32) 0 \*/

#define MYFS\_MAX\_FILESIZE \

(uint64\_t) MYFS\_MAX\_BLOCKS\_PER\_EXTENT \*MYFS\_BLOCK\_SIZE \

\*MYFS\_MAX\_EXTENTS

#define MYFS\_FILENAME\_LEN 28

#define MYFS\_MAX\_SUBFILES 128

struct myfs\_inode **{**

uint32\_t i\_mode**;** /\* File mode \*/

uint32\_t i\_uid**;** /\* Owner id \*/

uint32\_t i\_gid**;** /\* Group id \*/

uint32\_t i\_size**;** /\* Size in bytes \*/

uint32\_t i\_ctime**;** /\* Inode change time \*/

uint32\_t i\_atime**;** /\* Access time \*/

uint32\_t i\_mtime**;** /\* Modification time \*/

uint32\_t i\_blocks**;** /\* Block count \*/

uint32\_t i\_nlink**;** /\* Hard links count \*/

union **{**

uint32\_t ei\_block**;** /\* Block with list of extents for this file \*/

uint32\_t dir\_block**;** /\* Block with list of files for this directory \*/

**};**

char i\_data**[**32**];** /\* store symlink content \*/

**};**

#define MYFS\_INODES\_PER\_BLOCK (MYFS\_BLOCK\_SIZE / sizeof(struct myfs\_inode))

struct myfs\_sb\_info **{**

uint32\_t magic**;** /\* Magic number \*/

uint32\_t nr\_blocks**;** /\* Total number of blocks (incl sb & inodes) \*/

uint32\_t nr\_inodes**;** /\* Total number of inodes \*/

uint32\_t nr\_istore\_blocks**;** /\* Number of inode store blocks \*/

uint32\_t nr\_ifree\_blocks**;** /\* Number of inode free bitmap blocks \*/

uint32\_t nr\_bfree\_blocks**;** /\* Number of block free bitmap blocks \*/

uint32\_t nr\_free\_inodes**;** /\* Number of free inodes \*/

uint32\_t nr\_free\_blocks**;** /\* Number of free blocks \*/

#ifdef \_\_KERNEL\_\_

unsigned long **\***ifree\_bitmap**;** /\* In-memory free inodes bitmap \*/

unsigned long **\***bfree\_bitmap**;** /\* In-memory free blocks bitmap \*/

#endif

**};**

#ifdef \_\_KERNEL\_\_

struct myfs\_inode\_info **{**

union **{**

uint32\_t ei\_block**;** /\* Block with list of extents for this file \*/

uint32\_t dir\_block**;** /\* Block with list of files for this directory \*/

**};**

char i\_data**[**32**];**

struct inode vfs\_inode**;**

**};**

struct myfs\_extent **{**

uint32\_t ee\_block**;** /\* first logical block extent covers \*/

uint32\_t ee\_len**;** /\* number of blocks covered by extent \*/

uint32\_t ee\_start**;** /\* first physical block extent covers \*/

**};**

struct myfs\_file\_ei\_block **{**

struct myfs\_extent extents**[**MYFS\_MAX\_EXTENTS**];**

**};**

struct myfs\_dir\_block **{**

struct myfs\_file **{**

uint32\_t inode**;**

char filename**[**MYFS\_FILENAME\_LEN**];**

**}** files**[**MYFS\_MAX\_SUBFILES**];**

**};**

/\* superblock functions \*/

int myfs\_fill\_super**(**struct super\_block **\***sb**,** void **\***data**,** int silent**);**

/\* inode functions \*/

int myfs\_init\_inode\_cache**(**void**);**

void myfs\_destroy\_inode\_cache**(**void**);**

struct inode **\***myfs\_iget**(**struct super\_block **\***sb**,** unsigned long ino**);**

/\* file functions \*/

extern const struct file\_operations myfs\_file\_ops**;**

extern const struct file\_operations myfs\_dir\_ops**;**

extern const struct address\_space\_operations myfs\_aops**;**

/\* extent functions \*/

extern uint32\_t myfs\_ext\_search**(**struct myfs\_file\_ei\_block **\***index**,**

uint32\_t iblock**);**

/\* Getters for superbock and inode \*/

#define MYFS\_SB(sb) (sb->s\_fs\_info)

#define MYFS\_INODE(inode) \

(container\_of(inode, struct myfs\_inode\_info, vfs\_inode))

#endif /\* \_\_KERNEL\_\_ \*/

#endif /\* MYFS\_H \*/

Extent.c

#include <linux/fs.h>

#include <linux/kernel.h>

#include "myfs.h"

/\*

\* Search the extent which contain the target block.

\* Retrun the first unused file index if not found.

\* Return -1 if it is out of range.

\* TODO: use binary search.

\*/

uint32\_t myfs\_ext\_search**(**struct myfs\_file\_ei\_block **\***index**,**

uint32\_t iblock**)**

**{**

uint32\_t i**;**

**for** **(**i **=** 0**;** i **<** MYFS\_MAX\_EXTENTS**;** i**++)** **{**

uint32\_t block **=** index**->**extents**[**i**].**ee\_block**;**

uint32\_t len **=** index**->**extents**[**i**].**ee\_len**;**

**if** **(**index**->**extents**[**i**].**ee\_start **==** 0 **||**

**(**iblock **>=** block **&&** iblock **<** block **+** len**))**

**return** i**;**

**}**

**return** **-**1**;**

**}**

Bitmap.h

#ifndef MYFS\_BITMAP\_H

#define MYFS\_BITMAP\_H

#include <linux/bitmap.h>

#include "myfs.h"

/\*

\* Return the first bit we found and clear the the following `len` consecutive

\* free bit(s) (set to 1) in a given in-memory bitmap spanning over multiple

\* blocks. Return 0 if no enough free bit(s) were found (we assume that the

\* first bit is never free because of the superblock and the root inode, thus

\* allowing us to use 0 as an error value).

\*/

static inline uint32\_t get\_first\_free\_bits**(**unsigned long **\***freemap**,**

unsigned long size**,**

uint32\_t len**)**

**{**

uint32\_t bit**,** prev **=** 0**,** count **=** 0**;**

for\_each\_set\_bit **(**bit**,** freemap**,** size**)** **{**

**if** **(**prev **!=** bit **-** 1**)**

count **=** 0**;**

prev **=** bit**;**

**if** **(++**count **==** len**)** **{**

bitmap\_clear**(**freemap**,** bit **-** len **+** 1**,** len**);**

**return** bit **-** len **+** 1**;**

**}**

**}**

**return** 0**;**

**}**

/\*

\* Return an unused inode number and mark it used.

\* Return 0 if no free inode was found.

\*/

static inline uint32\_t get\_free\_inode**(**struct myfs\_sb\_info **\***sbi**)**

**{**

uint32\_t ret **=** get\_first\_free\_bits**(**sbi**->**ifree\_bitmap**,** sbi**->**nr\_inodes**,** 1**);**

**if** **(**ret**)**

sbi**->**nr\_free\_inodes**--;**

**return** ret**;**

**}**

/\*

\* Return `len` unused block(s) number and mark it used.

\* Return 0 if no enough free block(s) were found.

\*/

static inline uint32\_t get\_free\_blocks**(**struct myfs\_sb\_info **\***sbi**,**

uint32\_t len**)**

**{**

uint32\_t ret **=** get\_first\_free\_bits**(**sbi**->**bfree\_bitmap**,** sbi**->**nr\_blocks**,** len**);**

**if** **(**ret**)**

sbi**->**nr\_free\_blocks **-=** len**;**

**return** ret**;**

**}**

/\* Mark the `len` bit(s) from i-th bit in freemap as free (i.e. 1) \*/

static inline int put\_free\_bits**(**unsigned long **\***freemap**,**

unsigned long size**,**

uint32\_t i**,**

uint32\_t len**)**

**{**

/\* i is greater than freemap size \*/

**if** **(**i **+** len **-** 1 **>** size**)**

**return** **-**1**;**

bitmap\_set**(**freemap**,** i**,** len**);**

**return** 0**;**

**}**

/\* Mark an inode as unused \*/

static inline void put\_inode**(**struct myfs\_sb\_info **\***sbi**,** uint32\_t ino**)**

**{**

**if** **(**put\_free\_bits**(**sbi**->**ifree\_bitmap**,** sbi**->**nr\_inodes**,** ino**,** 1**))**

**return;**

sbi**->**nr\_free\_inodes**++;**

**}**

/\* Mark len block(s) as unused \*/

static inline void put\_blocks**(**struct myfs\_sb\_info **\***sbi**,**

uint32\_t bno**,**

uint32\_t len**)**

**{**

**if** **(**put\_free\_bits**(**sbi**->**bfree\_bitmap**,** sbi**->**nr\_blocks**,** bno**,** len**))**

**return;**

sbi**->**nr\_free\_blocks **+=** len**;**

**}**

#endif /\* MYFS\_BITMAP\_H \*/