





#include <linux/time.h>

#include <linux/pagemap.h>

#include <linux/dax.h>

#include <linux/quotaops.h>

#include "myext2.h"

#include "xattr.h"

#include "acl.h"

#include <linux/slab.h>

#include <linux/stat.h>

#include <linux/fcntl.h>

#include <linux/file.h>

#include <linux/uio.h>

#include <linux/fsnotify.h>

#include <linux/security.h>

#include <linux/export.h>

#include <linux/syscalls.h>

#include <linux/splice.h>

#include <linux/compat.h>

#include <linux/mount.h>

#include <linux/fs.h>

#include <asm/uaccess.h>

#include <asm/unistd.h>

typedef ssize\_t (\*io\_fn\_t)(struct file \*, char \_\_user \*, size\_t, loff\_t \*);

typedef ssize\_t (\*iter\_fn\_t)(struct kiocb \*, struct iov\_iter \*);

#ifdef CONFIG\_FS\_DAX

/\*

\* The lock ordering for myext2 DAX fault paths is:

\*

\* mmap\_sem (MM)

\* sb\_start\_pagefault (vfs, freeze)

\* myext2\_inode\_info->dax\_sem

\* address\_space->i\_mmap\_rwsem or page\_lock (mutually exclusive in DAX)

\* myext2\_inode\_info->truncate\_mutex

\*

\* The default page\_lock and i\_size verification done by non-DAX fault paths

\* is sufficient because myext2 doesn't support hole punching.

\*/

static ssize\_t new\_sync\_write(struct file \*filp, const char \_\_user \*buf, size\_t len, loff\_t \*ppos)

{

struct iovec iov = { .iov\_base = (void \_\_user \*)buf, .iov\_len = len };

struct kiocb kiocb;

struct iov\_iter iter;

ssize\_t ret;

init\_sync\_kiocb(&kiocb, filp);

kiocb.ki\_pos = \*ppos;

iov\_iter\_init(&iter, WRITE, &iov, 1, len);

ret = filp->f\_op->write\_iter(&kiocb, &iter);

BUG\_ON(ret == -EIOCBQUEUED);

if (ret > 0)

\*ppos = kiocb.ki\_pos;

return ret;

}

static ssize\_t new\_sync\_read(struct file \*filp, char \_\_user \*buf, size\_t len, loff\_t \*ppos)

{

struct iovec iov = { .iov\_base = buf, .iov\_len = len };

struct kiocb kiocb;

struct iov\_iter iter;

ssize\_t ret;

init\_sync\_kiocb(&kiocb, filp);

kiocb.ki\_pos = \*ppos;

iov\_iter\_init(&iter, READ, &iov, 1, len);

ret = filp->f\_op->read\_iter(&kiocb, &iter);

BUG\_ON(ret == -EIOCBQUEUED);

\*ppos = kiocb.ki\_pos;

return ret;

}

ssize\_t new\_sync\_write\_crypt(struct file \*filp, const char \_\_user \*buf, size\_t len, loff\_t \*ppos)

{

int i;

char\* mybuf = buf;

for(i=0;i<len;i++)//在此处添加对长度为len的buf数据进行加密（简单移位密码，将每个字符值+25）

{

mybuf[i]+=25;

}

printk("haha encrypt %ld\n", len);

return new\_sync\_write(filp, mybuf, len, ppos);//调用默认的写函数，把加密数据写入

}

ssize\_t new\_sync\_read\_crypt(struct file \*filp, char \_\_user \*buf, size\_t len, loff\_t \*ppos)

{

int i;//先调用默认的读函数读取文件数据

ssize\_t ret = new\_sync\_read(filp, buf, len, ppos);

//此处添加对文件的解密（简单移位解密，将每个字符值-25）

for(i=0;i<len;i++)

{

buf[i]-=25;

}

printk("haha encrypt %ld\n", len);

return ret;

}

const struct file\_operations myext2\_file\_operations = {

.llseek = generic\_file\_llseek,

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

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

.unlocked\_ioctl = myext2\_ioctl,

.read =new\_sync\_read\_crypt,

.write =new\_sync\_write\_crypt,

#ifdef CONFIG\_COMPAT

.compat\_ioctl = myext2\_compat\_ioctl,

#endif

.mmap = myext2\_file\_mmap,

.open = dquot\_file\_open,

.release = myext2\_release\_file,

.fsync = myext2\_fsync,

.splice\_read = generic\_file\_splice\_read,

.splice\_write = iter\_file\_splice\_write,

};