#include <linux/version.h>

#include <linux/autoconf.h>

#include <linux/module.h>

#include <linux/moduleparam.h>

#include <linux/init.h>

#include <linux/sched.h>

#include <linux/kernel.h> /\* printk() \*/

#include <linux/errno.h> /\* error codes \*/

#include <linux/types.h> /\* size\_t \*/

#include <linux/vmalloc.h>

#include <linux/genhd.h>

#include <linux/blkdev.h>

#include <linux/wait.h>

#include <linux/file.h>

#include "spinlock.h"

#include "osprd.h"

/\* The size of an OSPRD sector. \*/

#define SECTOR\_SIZE 512

/\* This flag is added to an OSPRD file's f\_flags to indicate that the file

\* is locked. \*/

#define F\_OSPRD\_LOCKED 0x80000

/\* eprintk() 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\_\_)

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

MODULE\_DESCRIPTION("CS 111 RAM Disk");

// EXERCISE: Pass your names into the kernel as the module's authors.

MODULE\_AUTHOR("JONATHAN WOONG");

#define OSPRD\_MAJOR 222

/\* This module parameter controls how big the disk will be.

\* You can specify module parameters when you load the module,

\* as an argument to insmod: "insmod osprd.ko nsectors=4096" \*/

static int nsectors = 32;

module\_param(nsectors, int, 0);

/\* The internal representation of our device. \*/

typedef struct osprd\_info {

uint8\_t \*data; // The data array. Its size is

// (nsectors \* SECTOR\_SIZE) bytes.

osp\_spinlock\_t mutex; // Mutex for synchronizing access to

// this block device

unsigned ticket\_head; // Currently running ticket for

// the device lock

unsigned ticket\_tail; // Next available ticket for

// the device lock

wait\_queue\_head\_t blockq; // Wait queue for tasks blocked on

// the device lock

size\_t n\_readl; // Number of read locks

size\_t n\_writel; // Number of write locks

unsigned desync; //number of interrupted processes

int dead; //whether this will cause a deadlock or not

/\* HINT: You may want to add additional fields to help

in detecting deadlock. \*/

// The following elements are used internally; you don't need

// to understand them.

struct request\_queue \*queue; // The device request queue.

spinlock\_t qlock; // Used internally for mutual

// exclusion in the 'queue'.

struct gendisk \*gd; // The generic disk.

} osprd\_info\_t;

#define NOSPRD 4

static osprd\_info\_t osprds[NOSPRD];

// Declare useful helper functions

/\*

\* file2osprd(filp)

\* Given an open file, check whether that file corresponds to an OSP ramdisk.

\* If so, return a pointer to the ramdisk's osprd\_info\_t.

\* If not, return NULL.

\*/

static osprd\_info\_t \*file2osprd(struct file \*filp);

/\*

\* for\_each\_open\_file(task, callback, user\_data)

\* Given a task, call the function 'callback' once for each of 'task's open

\* files. 'callback' is called as 'callback(filp, user\_data)'; 'filp' is

\* the open file, and 'user\_data' is copied from for\_each\_open\_file's third

\* argument.

\*/

static void for\_each\_open\_file(struct task\_struct \*task,

void (\*callback)(struct file \*filp,

osprd\_info\_t \*user\_data),

osprd\_info\_t \*user\_data);

/\*

\* osprd\_process\_request(d, req)

\* Called when the user reads or writes a sector.

\* Should perform the read or write, as appropriate.

\*/

static void osprd\_process\_request(osprd\_info\_t \*d, struct request \*req)

{

long int sector\_offset;

long int numbytes;

//int err;

if (!blk\_fs\_request(req)) {

end\_request(req, 0);

return;

}

// EXERCISE: Perform the read or write request by copying data between

// our data array and the request's buffer.

// Hint: The 'struct request' argument tells you what kind of request

// this is, and which sectors are being read or written.

// Read about 'struct request' in <linux/blkdev.h>.

// Consider the 'req->sector', 'req->current\_nr\_sectors', and

// 'req->buffer' members, and the rq\_data\_dir() function.

// Your code here.

sector\_offset = req->sector\*SECTOR\_SIZE;

numbytes = req->current\_nr\_sectors\*SECTOR\_SIZE;

if(rq\_data\_dir(req)==READ) {

memcpy(req->buffer, d->data+sector\_offset, numbytes);

}

else if(rq\_data\_dir(req)==WRITE) {

memcpy(d->data+sector\_offset,req->buffer, numbytes);

}

else {

eprintk("Failure to READ/WRITE\n");

}

end\_request(req, 1);

}

// This function is called when a /dev/osprdX file is opened.

// You aren't likely to need to change this.

static int osprd\_open(struct inode \*inode, struct file \*filp)

{

// Always set the O\_SYNC flag. That way, we will get writes immediately

// instead of waiting for them to get through write-back caches.

filp->f\_flags |= O\_SYNC;

return 0;

}

// This function is called when a /dev/osprdX file is finally closed.

// (If the file descriptor was dup2ed, this function is called only when the

// last copy is closed.)

int osprd\_ioctl(struct inode \*inode, struct file \*filp,

unsigned int cmd, unsigned long arg);

static int osprd\_close\_last(struct inode \*inode, struct file \*filp)

{

int r;

r = 0;

if (filp) {

osprd\_info\_t \*d = file2osprd(filp);

int filp\_writable = filp->f\_mode & FMODE\_WRITE;

// EXERCISE: If the user closes a ramdisk file that holds

// a lock, release the lock. Also wake up blocked processes

// as appropriate.

// Your code here.

// This line avoids compiler warnings; you may remove it.

(void) filp\_writable, (void) d;

if (!(filp->f\_flags & F\_OSPRD\_LOCKED))

{

r = -EINVAL;

}

// Otherwise, clear the lock from filp->f\_flags, wake up

// the wait queue, perform any additional accounting steps

// you need, and return 0.

else

{

osp\_spin\_lock(&(d->mutex));

filp->f\_flags &= ~F\_OSPRD\_LOCKED;

d->n\_writel = 0;

d->n\_readl = 0;

d->dead = 0;

if(waitqueue\_active(&d->blockq) == 0) {

d->ticket\_tail += d->desync;

d->desync = 0;

}

osp\_spin\_unlock(&(d->mutex));

wake\_up\_all(&d->blockq);

r = 0;

}

}

return r;

}

void cause\_deadlock(struct file \*filp, osprd\_info\_t \*d){

if (file2osprd(filp) == d){

d->dead++;

}

}

int osprd\_ioctl(struct inode \*inode, struct file \*filp,

unsigned int cmd, unsigned long arg)

{

osprd\_info\_t \*d = file2osprd(filp); // device info

int r = 0;

unsigned local\_ticket;

// is file open for writing?

int filp\_writable = (filp->f\_mode & FMODE\_WRITE) != 0;

// This line avoids compiler warnings; you may remove it.

//(void) filp\_writable, (void) d;

// Set 'r' to the ioctl's return value: 0 on success, negative on error

if (cmd == OSPRDIOCACQUIRE) {

// EXERCISE: Lock the ramdisk.g

//

// If \*filp is open for writing (filp\_writable), then attempt

// to write-lock the ramdisk; otherwise attempt to read-lock

// the ramdisk.

//

// This lock request must block using 'd->blockq' until:

// 1) no other process holds a write lock;

// 2) either the request is for a read lock, or no other process

// holds a read lock; and

// 3) lock requests should be serviced in order, so no process

// that blocked earlier is still blocked waiting for the

// lock.

//

// If a process acquires a lock, mark this fact by setting

// 'filp->f\_flags |= F\_OSPRD\_LOCKED'. You also need to

// keep track of how many read and write locks are held:

// change the 'osprd\_info\_t' structure to do this.

//

// Also wake up processes waiting on 'd->blockq' as needed.

//

// If the lock request would cause a deadlock, return -EDEADLK.

// If the lock request blocks and is awoken by a signal, then

// return -ERESTARTSYS.

// Otherwise, if we can grant the lock request, return 0.

// 'd->ticket\_head' and 'd->ticket\_tail' should help you

// service lock requests in order. These implement a ticket

// order: 'ticket\_tail' is the next ticket, and 'ticket\_head'

// is the ticket currently being served. You should set a local

// variable to 'd->ticket\_head' and increment 'd->ticket\_head'.

// Then, block at least until 'd->ticket\_tail == local\_ticket'.

// (Some of these operations are in a critical section and must

// be protected by a spinlock; which ones?)

// Your code here (instead of the next two lines).

osp\_spin\_lock(&(d->mutex));

local\_ticket = d->ticket\_head;

d->ticket\_head++;

osp\_spin\_unlock(&(d->mutex));

for\_each\_open\_file(current, cause\_deadlock, d);

if (d->dead > 1 && (filp->f\_flags & F\_OSPRD\_LOCKED)) {

return -EDEADLK;

}

if (wait\_event\_interruptible(d->blockq, d->n\_writel == 0

&& (!filp\_writable || d->n\_readl == 0)

&& d->ticket\_tail == local\_ticket))

{

if (d->ticket\_tail == local\_ticket) {

d->ticket\_tail++;

}

else {

d->desync++;

}

return -ERESTARTSYS;

}

osp\_spin\_lock(&(d->mutex));

d->dead = 0;

if (d->mutex.lock>0) {

r = 0;

}

filp->f\_flags |= F\_OSPRD\_LOCKED;

if (filp\_writable) {

d->n\_writel++; d->ticket\_tail++;

}

else {

d->n\_readl++;

}

osp\_spin\_unlock(&(d->mutex));

if (!filp\_writable) {

d->ticket\_tail++;

}

r = 0;

} else if (cmd == OSPRDIOCTRYACQUIRE) {

// EXERCISE: ATTEMPT to lock the ramdisk.

//

// This is just like OSPRDIOCACQUIRE, except it should never

// block. If OSPRDIOCACQUIRE would block or return deadlock,

// OSPRDIOCTRYACQUIRE should return -EBUSY.

// Otherwise, if we can grant the lock request, return 0.

local\_ticket = d->ticket\_head;

if (filp->f\_flags & F\_OSPRD\_LOCKED || d->n\_writel != 0

|| (filp\_writable && d->n\_readl != 0)

|| d->ticket\_tail != local\_ticket)

{

r = -EBUSY;

}

else

{

osp\_spin\_lock(&(d->mutex));

d->ticket\_head++;

filp->f\_flags |= F\_OSPRD\_LOCKED;

if (filp\_writable) {

d->n\_writel++;

}

else {

d->n\_readl++;

}

if(d->ticket\_tail < d->ticket\_head) {

d->ticket\_tail++;

}

osp\_spin\_unlock(&(d->mutex));

r = 0;

wake\_up\_all(&d->blockq);

}

} else if (cmd == OSPRDIOCRELEASE) {

// EXERCISE: Unlock the ramdisk.

//

// If the file hasn't locked the ramdisk, return -EINVAL.

if (!(filp->f\_flags & F\_OSPRD\_LOCKED)) {

r = -EINVAL;

}

// Otherwise, clear the lock from filp->f\_flags, wake up

// the wait queue, perform any additional accounting steps

// you need, and return 0.

else

{

osp\_spin\_lock(&(d->mutex));

filp->f\_flags &= ~F\_OSPRD\_LOCKED;

d->n\_writel = 0;

d->n\_readl = 0;

osp\_spin\_unlock(&(d->mutex));

wake\_up\_all(&d->blockq);

r = 0;

}

}

else {

r = -ENOTTY;

}

return r;

}

// Initialize internal fields for an osprd\_info\_t.

static void osprd\_setup(osprd\_info\_t \*d)

{

/\* Initialize the wait queue. \*/

init\_waitqueue\_head(&d->blockq);

osp\_spin\_lock\_init(&d->mutex);

d->ticket\_head = d->ticket\_tail = 0;

d->n\_readl = 0;

d->n\_writel = 0;

d->dead = 0;

d->desync = 0;

/\* Add code here if you add fields to osprd\_info\_t. \*/

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* THERE IS NO NEED TO UNDERSTAND ANY CODE BELOW THIS LINE! \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Process a list of requests for a osprd\_info\_t.

// Calls osprd\_process\_request for each element of the queue.

static void osprd\_process\_request\_queue(request\_queue\_t \*q)

{

osprd\_info\_t \*d = (osprd\_info\_t \*) q->queuedata;

struct request \*req;

while ((req = elv\_next\_request(q)) != NULL)

osprd\_process\_request(d, req);

}

// Some particularly horrible stuff to get around some Linux issues:

// the Linux block device interface doesn't let a block device find out

// which file has been closed. We need this information.

static struct file\_operations osprd\_blk\_fops;

static int (\*blkdev\_release)(struct inode \*, struct file \*);

static int \_osprd\_release(struct inode \*inode, struct file \*filp)

{

if (file2osprd(filp))

osprd\_close\_last(inode, filp);

return (\*blkdev\_release)(inode, filp);

}

static int \_osprd\_open(struct inode \*inode, struct file \*filp)

{

if (!osprd\_blk\_fops.open) {

memcpy(&osprd\_blk\_fops, filp->f\_op, sizeof(osprd\_blk\_fops));

blkdev\_release = osprd\_blk\_fops.release;

osprd\_blk\_fops.release = \_osprd\_release;

}

filp->f\_op = &osprd\_blk\_fops;

return osprd\_open(inode, filp);

}

// The device operations structure.

static struct block\_device\_operations osprd\_ops = {

.owner = THIS\_MODULE,

.open = \_osprd\_open,

// .release = osprd\_release, // we must call our own release

.ioctl = osprd\_ioctl

};

// Given an open file, check whether that file corresponds to an OSP ramdisk.

// If so, return a pointer to the ramdisk's osprd\_info\_t.

// If not, return NULL.

static osprd\_info\_t \*file2osprd(struct file \*filp)

{

if (filp) {

struct inode \*ino = filp->f\_dentry->d\_inode;

if (ino->i\_bdev

&& ino->i\_bdev->bd\_disk

&& ino->i\_bdev->bd\_disk->major == OSPRD\_MAJOR

&& ino->i\_bdev->bd\_disk->fops == &osprd\_ops)

return (osprd\_info\_t \*) ino->i\_bdev->bd\_disk->private\_data;

}

return NULL;

}

// Call the function 'callback' with data 'user\_data' for each of 'task's

// open files.

static void for\_each\_open\_file(struct task\_struct \*task,

void (\*callback)(struct file \*filp, osprd\_info\_t \*user\_data),

osprd\_info\_t \*user\_data)

{

int fd;

task\_lock(task);

spin\_lock(&task->files->file\_lock);

{

#if LINUX\_VERSION\_CODE <= KERNEL\_VERSION(2, 6, 13)

struct files\_struct \*f = task->files;

#else

struct fdtable \*f = task->files->fdt;

#endif

for (fd = 0; fd < f->max\_fds; fd++)

if (f->fd[fd])

(\*callback)(f->fd[fd], user\_data);

}

spin\_unlock(&task->files->file\_lock);

task\_unlock(task);

}

// Destroy a osprd\_info\_t.

static void cleanup\_device(osprd\_info\_t \*d)

{

wake\_up\_all(&d->blockq);

if (d->gd) {

del\_gendisk(d->gd);

put\_disk(d->gd);

}

if (d->queue)

blk\_cleanup\_queue(d->queue);

if (d->data)

vfree(d->data);

}

// Initialize a osprd\_info\_t.

static int setup\_device(osprd\_info\_t \*d, int which)

{

memset(d, 0, sizeof(osprd\_info\_t));

/\* Get memory to store the actual block data. \*/

if (!(d->data = vmalloc(nsectors \* SECTOR\_SIZE)))

return -1;

memset(d->data, 0, nsectors \* SECTOR\_SIZE);

/\* Set up the I/O queue. \*/

spin\_lock\_init(&d->qlock);

if (!(d->queue = blk\_init\_queue(osprd\_process\_request\_queue, &d->qlock)))

return -1;

blk\_queue\_hardsect\_size(d->queue, SECTOR\_SIZE);

d->queue->queuedata = d;

/\* The gendisk structure. \*/

if (!(d->gd = alloc\_disk(1)))

return -1;

d->gd->major = OSPRD\_MAJOR;

d->gd->first\_minor = which;

d->gd->fops = &osprd\_ops;

d->gd->queue = d->queue;

d->gd->private\_data = d;

snprintf(d->gd->disk\_name, 32, "osprd%c", which + 'a');

set\_capacity(d->gd, nsectors);

add\_disk(d->gd);

/\* Call the setup function. \*/

osprd\_setup(d);

return 0;

}

static void osprd\_exit(void);

// The kernel calls this function when the module is loaded.

// It initializes the 4 osprd block devices.

static int \_\_init osprd\_init(void)

{

int i, r;

// shut up the compiler

(void) for\_each\_open\_file;

#ifndef osp\_spin\_lock

(void) osp\_spin\_lock;

(void) osp\_spin\_unlock;

#endif

/\* Register the block device name. \*/

if (register\_blkdev(OSPRD\_MAJOR, "osprd") < 0) {

printk(KERN\_WARNING "osprd: unable to get major number\n");

return -EBUSY;

}

/\* Initialize the device structures. \*/

for (i = r = 0; i < NOSPRD; i++)

if (setup\_device(&osprds[i], i) < 0)

r = -EINVAL;

if (r < 0) {

printk(KERN\_EMERG "osprd: can't set up device structures\n");

osprd\_exit();

return -EBUSY;

} else

return 0;

}

// The kernel calls this function to unload the osprd module.

// It destroys the osprd devices.

static void osprd\_exit(void)

{

int i;

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

cleanup\_device(&osprds[i]);

unregister\_blkdev(OSPRD\_MAJOR, "osprd");

}

// Tell Linux to call those functions at init and exit time.

module\_init(osprd\_init);

module\_exit(osprd\_exit);