LAB 2

```
#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);


/*
```

LAB 2

```
 * 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);
```