EEL 4732/5733 Advanced Systems Programming
Assignment 6

In this assignment you are going to check a simple character device driver (code snippet given below, minimal code provided on CANVAS, files/assignments/assignment6.c) for concurrency errors: deadlocks and race conditions. The driver aims to support two modes of operation: `MODE1` and `MODE2`. In `MODE1` only one user process can open and access the devices whereas in `MODE2` multiple processes/threads can interact with the device concurrently. Additionally, the mode of operation can be changed at run-time via the `ioctl` function.

We would like you to come up with tests to check for 4 different potential deadlock scenarios of your choice. The tests should include a user-space program (possibly using pthreads) that uses the original or a modified version of the driver. Please note that not every deadlock scenario may require you to insert sleep statements in the driver. So you are required to provide the modified version of the driver only if such modifications are necessary to reproduce the deadlock scenario. You should also provide a README file for each test case explaining which deadlock scenario you're checking by specifying the line numbers of wait statements.

For race conditions, we would like you to come up with 4 pairs of critical regions that may run in parallel and perform a code review rather than crafting a test program. In your review, for each critical region you should identify the data accessed in the critical region, the locks held at the the time it is entered, and your reasoning about the possibility or absence of race conditions.

```
int e2_open(struct inode *inode, struct file *filp)
{
    struct e2_dev *devc = container_of(inode->i_cdev, struct e2_dev, cdev);
    filp->private_data = devc;
    down_interruptible(&devc->sem1);
    if (devc->mode == MODE1) {
        devc->count1++;
        up(&devc->sem1);
        down_interruptible(&devc->sem2);
        return 0;
    }
    else if (devc->mode == MODE2) {
        devc->count2++;
    }
    up(&devc->sem1);
    return 0;
}

int e2_release(struct inode *inode, struct file *filp)
{
    struct e2_dev *devc = container_of(inode->i_cdev, struct e2_dev, cdev);
    down_interruptible(&devc->sem1);
    if (devc->mode == MODE1) {
        devc->count1--;
        if (devc->count1 == 1)
            wake_up_interruptible(&(devc->queue1));
up(&devc->sem2);
    }
    else if (devc->mode == MODE2) {
        devc->count2--;
        if (devc->count2 == 1)
            wake_up_interruptible(&(devc->queue2));
    }
    up(&devc->sem1);
    return 0;
}

static ssize_t e2_read (struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
   struct e2_dev *devc = filp->private_data;
ssize_t ret = 0;
down_interruptible(&devc->sem1);
if (devc->mode == MODE1) {
   up(&devc->sem1);
           if (*f_pos + count > ramdisk_size) {
              printk("Trying to read past end of buffer!\n");
              return ret;
           }
   ret = count - copy_to_user(buf, devc->ramdisk, count);
}
else {
          if (*f_pos + count > ramdisk_size) {
             printk("Trying to read past end of buffer!\n");
             up(&devc->sem1);
             return ret;
          }
          ret = count - copy_to_user(buf, devc->ramdisk, count);
  up(&devc->sem1);
}
return ret;
}


static ssize_t e2_write (struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
    struct e2_dev *devc;
ssize_t ret = 0;
devc = filp->private_data;
down_interruptible(&devc->sem1);
if (devc->mode == MODE1) {
up(&devc->sem1);
        if (*f_pos + count > ramdisk_size) {
            printk("Trying to read past end of buffer!\n");
            return ret;
        }
        ret = count - copy_from_user(devc->ramdisk, buf, count);
}
else {
        if (*f_pos + count > ramdisk_size) {
            printk("Trying to read past end of buffer!\n");
            up(&devc->sem1);
            return ret;
        }
        ret = count - copy_from_user(devc->ramdisk, buf, count);
up(&devc->sem1);
}
return ret;
}

static long e2_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{

       struct e2_dev *devc = filp->private_data;

if (_IOC_TYPE(cmd) != CDRV_IOC_MAGIC) {
pr_info("Invalid magic number\n");
return -ENOTTY;
}
if ( (_IOC_NR(cmd) != 1) && (_IOC_NR(cmd) != 2) ) {
pr_info("Invalid cmd\n");
return -ENOTTY;
}

switch(cmd) {
case E2_IOCMODE2:
down_interruptible(&(devc->sem1));
if (devc->mode == MODE2) {
up(&devc->sem1);
break;
}
if (devc->count1 > 1) {
while (devc->count1 > 1) {
up(&devc->sem1);
    wait_event_interruptible(devc->queue1, (devc->count1 == 1));
down_interruptible(&devc->sem1);
}
}
devc->mode = MODE2;
             devc->count1--;
             devc->count2++;
up(&devc->sem2);
up(&devc->sem1);
break;

case E2_IOCMODE1:
down_interruptible(&devc->sem1);
if (devc->mode == MODE1) {
   up(&devc->sem1);
   break;
```

```
}
if (devc->count2 > 1) {
   while (devc->count2 > 1) {
       up(&devc->sem1);
       wait_event_interruptible(devc->queue2, (devc->count2 == 1));
       down_interruptible(&devc->sem1);
   }
}
devc->mode = MODE1;
      devc->count2--;
      devc->count1++;
down_interruptible(&devc->sem2);
up(&devc->sem1);
break;

default :
pr_info("Unrecognized ioctl command\n");
return -1;
break;
}
return 0;
}

...
```

Please submit all your test programs, modified versions of the driver, and README files on CANVAS.