Critical Region Pair #1

```
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);
        }
...
```

```
case E2_IOCMODE2:
      printk(KERN_INFO "E2_IOCMODE2\n");
        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;
```

Data accessed in critical region #1:	devc->mode, devc->ramdisk
Locks held in critical region #1:	devc->sem1
Data accessed in critical region #2:	devc->mode, devc->count1, devc->count2++
Locks held in critical region #2:	devc->sem1, potentially devc->sem2
Possibility of a race condition:	Since devc->mode is blocked by sem1, there is no chance for a data race.

Critical Region Pair #2

Highlighted section is critical region. d

```
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);
    }
...
```

Since a file descriptor could be passed to a child process or to a different thread, it is possible that this segment of code could be run from two different processes/threads.

Data accessed in both critical regions: copy_to_user(...) , (other data would be on stack)
Locks held in both critical regions: None
Possibility of a race condition: If two separate threads/processes called this function, they could potentially both call copy_to_user(...) at the same time. However, since this is only returning data, and not altering the data in any way, both processes/threads would just receive the same data. Thus, there would be no data race.

Critical Region Pair #3

Highlighted section is critical region. d

```
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);
```

```
    }
...
```

```
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);
        }
```

Data accessed in both critical regions: devc->ramdisk
Locks held in both critical regions: None
Possibility of a race condition: If two separate threads/processes called this function, there would be a data race accessing ramdisk. It is possible for two separate threads/processes to both have access to the same open file in mode 1 since the file descriptor could be passed when the process is forked or passed to both threads. Thus, if copy_from_user executes first, it will overwrite data in the ramdisk, therefore altering what copy_to_user returns.

Critical Region Pair #4

```
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);
        . . .
        else { // MODE 2
                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;` `}`
Same as above

Data accessed in both critical regions: devc->ramdisk, ramdisk_size

Locks held in both critical regions: sem1

Possibility of a race condition: If two separate threads/processes called this function, there would not be a data race accessing ramdisk. Because this critical section is blocked by sem1, it is impossible for there to be a data race to write to ramdisk. However, this is not applicable to the code for mode 1.