ASP ASSIGNMENT 6

KSHITIJ RAJ | UFID: 13584965

RACE CONDITIONS

Race Condition 1: Concurrent Read/Write with no protection - Data Race Condition Mode 1

Consider the following two pieces of code which show the read & write sections of the driver in Mode1:

```
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);
}
```

Imagine the following sequence of operations.

- User app opens the device file in Mode 1
- User app spawns threads and file descriptor is shared
- Thread 1 & 2 acquire & release locks and then they read/write simultaneously to the ramdisk without locks
- This is a case of potential data race

Race Condition 2: Concurrent Writes with no protection. - Data Race Condition Mode 1

Consider the following piece of code which shows the write section of the driver in Mode1.

```
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);
}
```

Imagine the following sequence of operations.

- User app opens the device file in Mode 1
- User app spawns threads and file descriptor is shared
- Thread 1 & 2 acquire & release locks and then they write simultaneously to the ramdisk without locks
- Data gets overwritten and this is a potential data race scenario

Since the driver is in Mode 1, it is under the assumption that there won't be multiple threads accessing the critical regions.

Race Condition 3: Simultaneous updating of pointers by 2 threads:

Consider Mode 2 & simultaneous writes // for write in mode 2

```
down_interruptible(&devc->sem1);
if (devc->mode == MODE1) { ……….
}
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;

● As you can observe updating of file pointer is left to the kernel after the write operation.
● When Two threads sharing a file descriptor perform simultaneous append operation, there is a chance of data getting over-written in the ramdisk if thread1 gets preempted right before the return statement.
● Userapp needs some kind of serialization for its threads in such a scenario.

Race Condition 4: Multiple threads call ioctl to change mode -> mode 2

Consider e2_ioctl() and the case of E2_ IOCMODE2:
Consider the code below:

```
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++;
```

Now:

- We are reading from mode and count1. This critical region is protected by Sem1.
- If multiple threads attempt to change the mode to mode 2, without sem1, it results in a data race condition.
- So we call down(sem1) and then read from count 1 & 2 and after that call up(sem1).