**Deadlocks and Race Conditions**

1. Deadlocks

**No changes have been made driver code except adding print statements to indicate the function and other necessary running status and info.**

1. **userapp1.c**

Number of threads: 3 (T0, T1 & T2)

Deadlock scenario: T0 goes first to change the mode from mode 1 to mode 2 incase current mode was mode 2 and exists. T1 and T2 open the device in mode 1 where T2 opens first and calls IOCTL and tries to change mode but cannot since count1=2. T1 cannot proceed as sem2 is held by T2, resulting in a deadlock.

1. **userapp2.c**

Number of threads: 3 (T0, T1 & T2)

Deadlock scenario: T0 goes first to change the mode from mode 1 to mode 2. T0 finishes and releases. T1 and T2 open the device in mode 2 and both try to change the mode to mode1 but cannot as only one thread must be existing to do. Both the threads wait in queue2 waiting to be woken up resulting in a deadlock.

1. **userapp3.c**

Number of threads: 3 (T0, T1 & T2)

Deadlock scenario: T0 goes first to change the mode from mode 1 to mode 2. T0 finishes and releases. T1 opens and exits without closing, thus not decrementing the count2. T2 opens and tries to change the mode to mode1 again but waits in queue2 as count2 is not equal to 1 (but 2) even when it’s the only thread accessing the device driver. This results in the deadlock.

1. **userapp4.c**

Number of threads: 3 (T0, T1 & T2)

Deadlock scenario: T0 goes first to change the mode from mode 1 to mode 2. T0 finishes and releases. T1 and T2 both require lock1 and lock2 but in an alternate fashion. T1 acquires lock1 and writes and sleeps. Simultaneously, T2 acquires lock2, reads and sleeps.

Now, T1 wakes up and tries to acquire lock2 (held by T2) whereas T2 also tries to acquire lock1 (held by T1). Both threads are waiting for locks and neither of them releases it, causing a deadlock.

1. **userapp\_deadlock\_1.c**

Number of threads: 3 (T0, T1 & T2)

Deadlock scenario: T0 goes first to change the mode from mode 1 to mode 2. T0 finishes and releases. T1 and T2 both have been programmed to change the mode from mode2 to mode1 but doing so requires acquiring a global lock. Depending on who goes in first, for example, T1 acquires the global lock and tries to change the mode but waits for all other processes to release. However, T2 is also waiting on global lock which would be released by T1 only after it changes the mode and exists the IOCTL.

This causes a deadlock as T1 is holding global lock and waiting for T2 to exit but T2 cannot proceed and exit only after it gets the global lock.

1. Code review for potential Race Conditions
2. Protecting mode variable using a semaphore

**---Open Function---**

int e2\_open(struct inode \*inode, struct file \*filp)

{

printk(KERN\_INFO "Device opened\n");

struct e2\_dev \*devc = container\_of(inode->i\_cdev, struct e2\_dev, cdev);

filp->private\_data = devc;

down\_interruptible(&devc->sem1); *//critical region starts*

if (devc->mode == MODE1) {

devc->count1++;

printk(KERN\_ALERT "Count 1 is:%d\n", devc->count1);

up(&devc->sem1);

down\_interruptible(&devc->sem2);

return 0;

}

else if (devc->mode == MODE2) {

devc->count2++;

printk(KERN\_ALERT "Count 2 is:%d\n", devc->count2);

}

up(&devc->sem1); *//critical regions ends*

printk(KERN\_ALERT "Mode: %d\n",dev->mode);

return 0;

}

**---IOCTL function---**

down\_interruptible(&(devc->sem1)); *//critical region starts*

if (devc->mode == MODE2) {

up(&devc->sem1);

printk(KERN\_INFO "Already in mode2\n");

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;

printk(KERN\_INFO "Mode changed to mode2\n");

devc->count1--;

devc->count2++;

up(&devc->sem2);

up(&devc->sem1); *//critical regions ends*

The mode has been protected using a semaphore. Failure to do so would have caused different concurrent running threads to change the mode and the result would be affected depending on which thread changes the mode value resulting in a potential race condition. But this has been avoided using the correctly implemented semaphore.

1. Protecting the concurrent reads and writes

**---read function---**

down\_interruptible(&devc->sem1); *//critical region starts*

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); *//critical regions ends*

**---write function---**

down\_interruptible(&devc->sem1); *//critical regions ends*

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); *//critical regions ends*

Both read and write functions have a lock at the start which is implemented using semaphore sem1. This is necessary has in mode2, more than one device can do a read or a write leading to concurrent writes. Hence, the critical region was protected by the lock semaphore ‘sem1’, which otherwise would have resulted in a race conditions.

1. Potential race condition due to non-updation of position pointer

As seen in point 2, it must be noted that position pointer is not being updated after read or write of data. It must be updated according to the number of bytes read or written as a function like lseek() in the previous assignment would case a race condition as and when the pointer position is modified. Also, this updating of position pointer must be inside the critical section protected by a lock.

1. Change of modes

The device driver changes modes and it is protected using semaphore sem1. This not only eliminates the problem of a potential race condition as explained in point 1 above but also avoids, infinite spinning in case a thread would land in a wait queue as there would no check on number of processes opening and accessing the device. Thus, the condition that only 1 process can be present while changing the mode ensures no race condition. Also, giving up the lock while waiting in the queue allows other processes to atleast, read or write from the device and encourage concurrency.