Shrivinayak Bhat
UFID: 47971819
Advanced Systems Programming
Assignment - 6

Files Included:
Deadlock1.c
Deadlock2.c
Deadlock3.c
Deadlock4.c
Makefile
Assignment6.c

Use make to compile all the files.

Part 1:
Deadlocks

---

1. Userapp tries to open the device twice in Mode1

- Device is set to mode1
- Userapp opens the device for the first time and succeeds if there are no other open devices
- Userapp opens the device for the second time without close it and results in a deadlock since it is trying to acquire sem2 which it already owns
- Since userapp is a single threaded process, there is no way it can execute further
- The file deadlock1.c implements a simple userapp which enters deadlock by trying to open the file twice.
- Userapp should execute fine in mode2
- Stuck on line 49.

2. Two different userapp threads try to change mode 2->1 concurrently

- Userapp sets mode =2 initially
- It spawns two threads and bothof them open the device
- Then, both the threads try to change mode to 1
- They endup in a deadlock since devc->count2 >1 holds true and both the threads give up.

- The file deadlock2.c implements the userapp for the above mentioned behavior.
- Stuck on line 154 & 154.

3. Thread 1 tries changing mode (1-> 2) while Thread 2 is trying to open the file

- Thread1 has opened the file.
- Thread 2 tries opening the file but stalls since it is not able to acquire sem2.
- Thread 1 tries to change mode to 2 but since devc->count1 > 1, it goes into wait queue holding sem2.
- Thread 2 is waiting for sem2 and there is a deadlock.
- The file deadlock3.c implements this scenario
- Stuck on line 49 & 154

4. Two threads same process calling IOCTL

- Two threads are created which try to open the device
- Device is in mode1 and only one threads succeeds in acquiring sem2
- Operation is changed to mode2
- The second thread tries to switch mode back to mode1 and this causes a deadlock since count>1
- The file deadlock4.c implements this scenario

Part 2:
Race Conditions

---

1. Concurrent Read/Write with no protection. - data race condition - Mode1

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

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 wont be multiple threads accessing the critical regions.

3. Concurrent Read-Write - Mode2 - No data race condition

Consider the following piece of code used to read & write data into ramdisk

```
// 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);
    }

// for read 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_to_user(buf, devc->ramdisk, count);
  up(&devc->sem1);
}
return ret;
```

- **Here the Read/Write Operations are protection and are made atomic using the semaphore sem1.**
- **Hence there wont be a data race condition when 2 or more threads/processes try to read/write at the same time.**

4. Concurrent Write-Write - Mode2 - No data race condition

Consider the code shown below

```
// 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);
  }
```

- Here the critical region is protected using semaphore sem1
- There cannot be a data race condition when two threads simultaneously write to the ramdisk

5. Simultaneous updation of file pointer by 2 threads - Potential data race condition

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 updation 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.