Linux Character Device Driver

Operating Systems Assignment 3

Dakshit Babbar | 2020EE30163

May 2024

1 Objective

We are required to design and implement a character device driver for a virtual LIFO device, whose size is practically unbounded, taken to be 1MB here. We must create two virtual LIFO devices that use the same device driver. One of the devices must be read-only, while the other must be write-only.

The characters written to the write-only device can be read only from the read-only LIFO device. The first character read from the read-only LIFO device corresponds to the last character written to the write-only LIFO device.

2 System Setup

The system setup is completed using the *Guide to Hacking the Linux Kernel* by Abhishek Safui. Following is the overview of the steps involved in the setup:

1. Install the source code of the *Yocto Project*(*poky*). This is an embedded Linux platform tool which helps us create a minimal configuration for our kernel to reduce the compilation and testing time.
2. Build *poky* to get the following:
   - Virtual Disk Image
   - Minimal .config file
3. Install the Linux source code from *kernel.org*
4. Build the Linux source code using the minimal .config file obtained from *poky*
5. Boot the compiled kernel on a VM with the Virtual Disk Image obtained from *poky*.

3 Structure

We have created a separate module for the driver in the file lifodriver.c. The driver manages 2 character devices, one that is read-only and another that is write-only. The characters written to the write-only device can be read only from the read-only LIFO device. The first character read from the read-only LIFO device corresponds to the last character written to the write-only LIFO device.

To implement the stack we have maintained a global array along with a top of the stack pointer, which is common to both the devices. The write-only device, writes to that array and increments the top pointer and the read-only device reads from that array and decrements the top pointer. In the case when a reader is trying to read from an empty stack, it will enter a wait queue until signaled by some writter after it writes something to the stack.

A simple driver can handle one reader and one writer easily, but to make the driver more resilient to concurrent accesses of multiple readers and multiple writers, we have utilized locks.

4 Implementation Details

Here we specify the details of the functions implemented for the working of the driver.

- **Virtual Device:**
  Following are the data structures and objects defined in order to create the virtual device which will be used by the two character devices we implement.

```
#define MAX_LIFO_SIZE 1024*1024
char buffer[MAX_LIFO_SIZE];

//one above the top of the stack
unsigned int top;

//define a lock for securing the buffer
static DEFINE_SPINLOCK(dev_lock);

//define a wait queue
static DECLARE_WAIT_QUEUE_HEAD(lifo_dev_wait_queue);

//declare a device nodes
static struct class *lifo_class;
static struct device *lifo_device_ro;
static struct device *lifo_device_wo;
```

- **File Operation Functions:**
  Following is the implementation of the file operation functions for the driver:

```
//open function
static int lifo_open(struct inode *inode, struct file *file){
    //initialize device
    //get the flags given by user
    //give error if reader wants to write
    //give error if writer wnants to read
    return 0;
}

//release function
static int lifo_release(struct inode *inode, struct file *file){
    return 0;
}
```

```
//read function
static ssize_t lifo_read(struct file *file, char *user_buffer, size, offset){
    bool acquired;
    do{
        wait_event_interruptible(lifo_dev_wait_queue, top != 0);
        spin_lock(&dev_lock);
        acquired=true;
        if(top==0){
            acquired=false;
            spin_unlock(&dev_lock);
        }
    } while(!acquired);

    size_t len = MIN(top, size);
    char local_buffer[len];
    //copy the contents of the device from top to bottom in local buffer
    top-=len;
    spin_unlock(&dev_lock);
    //copy from local buffer to user buffer
    return len;
}

//wite function
static ssize_t lifo_write(struct file *file, const char *user_buffer, size, offset){
    spin_lock(&dev_lock);
    size_t len = MIN(MAX_LIFO_SIZE-top, size);
    //copy from user buffer to array
    top+=len;
    spin_unlock(&dev_lock);
    wake_up_interruptible(&lifo_dev_wait_queue);
    return len;
}
```

5 Resilience and Extra Work

Following are some of the specifics related to the resilience of the driver, that were taken care of while implementing the device driver

- Verifying if a reader is trying to write or a writer is trying to read and give errors if necessary.
- Using spinlocks to avoid racing conditions
- In order to make the driver resilient to multiple readers and writers, we used a loop based mechanism. In the case of multiple readers waiting for the stack to get filled up, as soon as a writer fills up the stack, all the readers will be signaled and will move ahead to read which can lead to incorrect results. The loop makes sure that when the process is signaled about the stack being filled, it will try to acquire the lock and will check if the stack is filled or not, if not then it will stay in loop else it will move ahead to read.
- Written multiple tests to check the working and resilience of the driver in multiple cases.