ELL783 Assignment3_hard

Ananya Sharma 2023EET2186

May 2024

1 Character Device Driver

1.1 File Operations Structure

The file operations structure (**struct file_operations)** is fundamental in a character device driver. It defines callback functions that handle various file operations requested by user-space programs. For instance, the read, write, open, and release functions are specified within this structure, allowing the kernel to dispatch these operations to the appropriate driver functions.

my_open and my_release, are standard callback functions in a Linux device driver that handles the opening (open system call) and closing (close system call) of a device file, respectively.

1.2 Initialization and Cleanup Functions

- **my_init function:** The my_init function plays a critical role during module initialization. It dynamically allocates major and minor numbers for the designated devices using **alloc_chrdev_region**. Furthermore, it creates a device class (my device) associated with the kernel module and initializes character device structures **(my_cdev1 and my_cdev2)** with defined file operations (fops). Subsequently, it adds these devices to the system and creates corresponding device nodes **(/dev/my_device0 and /dev/my_device1)** for user-space access.
- **my_exit function:** The **my_exit function** serves as the cleanup routine for the kernel module. It responsibly destroys device nodes **(my_device0 and my_device1)**, unregisters character devices **(my_cdev1 and my_cdev2)**, removes the device class (mydevice), and releases allocated major and minor numbers using appropriate kernel functions. This ensures proper cleanup when the module is unloaded from the kernel.

1.3 Detailed Function Descriptors

- **my_read and my_write** functions handle data reading and writing operations, for the device. **my_read** reads data from the device into a user-space buffer (‘buf‘) with a specified maximum length (‘len‘), while **my_write** writes data from a user-space buffer (‘buf‘) into the device.
- **my_read function: my_read function** handles read operations from the device. It allocates memory (‘output_string1‘) using ‘kmalloc‘ to store a copy of ‘kernel_buffer‘. Then, it copies ‘kernel_buffer‘ into ‘output_string1‘ using ‘strcpy‘, we reverse the content of ‘output_string1‘ in-place

```
static struct file_operations fops = {
    .owner      = THIS_MODULE,
    .read       = my_read,
    .write       = my_write,
    .open       = my_open,
    .release    = my_release,
};
```

Figure 1: File operations

```c
static ssize_t my_read(struct file *filp, char __user *buf, size_t len, loff_t *off);
static ssize_t my_write(struct file *filp, const char __user *buf, size_t len, loff_t *off);
static int my_open(struct inode *inode, struct file *file);
static int my_release(struct inode *inode, struct file *file);
static int __init my_init(void);
static void __exit my_exit(void);
```

Figure 2: Function Prototypes

```c
//DEVICE 1
//creating cdev structure
cdev_init(&my_cdev1,&fops);

//adding character device to the system
if((cdev_add(&my_cdev1,MKDEV(MAJOR(dev), 0),1))<0){
    printk("cannot add device 0 to the system\n");
    class_destroy(dev_class);
    unregister_chrdev_region(dev, DEVICE_COUNT);
    return -1;
}

//creating device under /dev dir
if((device_create(dev_class,NULL,MKDEV(MAJOR(dev), 0),NULL,"my_device0"))==NULL){
    pr_err("cannot create device 0 \n");
    class_destroy(dev_class);
    unregister_chrdev_region(dev, DEVICE_COUNT);
    return -1;
}
```

Figure 3: Adding character device to the system

using two pointers (‘start‘ and ‘end‘). Then, we check if the device is allowed to be read based on the device’s minor number (‘check_minor‘). At last copies the data from ‘output_string1‘ to the user-space buffer (‘buf‘) using **‘copy_to_user‘**, handling potential errors if the copy operation fails and returns the number of bytes read (‘len‘) upon successful completion, indicating the amount of data transferred to the user.

- **my_write function: ‘my_write‘ function** handles write operations to the device. It checks if the device is the correct writing device based on the minor number **(‘check_minor‘)**. Then, it checks if the length of the data to be written (‘len‘) exceeds the maximum allowable data length (‘mem_size‘). If ‘len‘ is greater than ‘mem_size‘, it sets ‘len‘ to ‘mem_size‘ to prevent buffer overrun. It copies data from the user-space buffer (‘buf‘) into the kernel buffer (‘kernel_buffer‘) starting from the current ‘write_start‘ position using **‘copy_from_user‘**. It updates the ‘write_start‘ counter by adding the number of bytes successfully written (‘len‘). Additionally, it ensures that ‘write_start‘ does not exceed the maximum buffer size (‘mem_size - 1‘) to prevent writing beyond the allocated buffer space.

- **my_init function: my_init function** uses **‘alloc_chrdev_region‘** to dynamically allocate major and minor numbers for **‘DEVICE_COUNT‘** devices under the name **”mydevice”**. We use ‘class_create‘ to create a device class named ”mydevice” associated with this kernel module **(‘THIS_MODULE‘)**. Then, we will initialize a character device structure **(‘my_cdev1‘)** with the file operations structure (‘fops‘) defined for the module. This adds the character device **(‘my_cdev1‘)** to the system using **‘cdev_add‘** with the specified major and minor number (‘MKDEV(MAJOR(dev), 0)‘). Then, we use ‘**device_create‘** to create a device node (‘my_device0‘) under the ‘/dev‘ directory associated with the first device **(‘MKDEV(MAJOR(dev), 0)‘).**

  Similarly, initializes a second character device structure **(‘my_cdev2‘)** with the same file operations structure **(‘fops‘)**. We add the second character device **(‘my_cdev2‘)** to the system with a different minor number **(‘MKDEV(MAJOR(dev), 1)‘)**. Then, we create a device node **(‘my_device1‘)** under the **‘/dev‘ directory** associated with the second device. In the end, it returns ‘0‘ to indicate successful initialization (‘my_init‘ function execution).

```
strcpy(device,argv[1]);

strcpy(user_msg,argv[2]);

fd = open(device,O_WRONLY);
 if(fd==-1){
printf("cannot open device file \n");
return -1;
}
printf("opened file descriptor %d\n", fd);

//write to device
ssize_t bytes_written=write(fd, user_msg, strlen(user_msg));
if(bytes_written==-1){
    printf("cannot write to device \n");
close(fd);
return -1;
}
printf("written %zd bytes to device \n",bytes_written);
```

Figure 4: Writer C program

- **my_exit function:** The **‘my_exit‘ function** serves as the cleanup routine for the Linux kernel module. It is executed when the module is being removed or unloaded from the kernel. It calls **‘device_destroy‘** to remove the device nodes **(‘my_device0‘ and ‘my_device1‘)** associated with the module's device class (‘dev_class‘). It uses the major and minor numbers **(‘MKDEV(MAJOR(dev), 0)‘ and ‘MKDEV(MAJOR(dev), 1)‘)** to specify the devices to destroy. It calls ‘cdev_del‘ to remove the character device structures **(‘my_cdev1‘ and ‘my_cdev2‘)** from the system. This operation removes the association between the character devices and their file operations. It calls **‘class_unregister‘** to unregister the device class **(‘dev_class‘)** from the system, removing it from the class hierarchy. It calls **‘class_destroy‘** to destroy the device class (‘dev_class‘) and free associated resources. Then, it Calls **‘unregister_chrdev_region‘** to release the allocated major and minor numbers (‘dev‘) used by the module for its devices.

2 Writer

I have written a C program to write data to a device file (‘devmy_device1‘) representing a writer functionality. we have defined **MAX_BUF_SIZE** as ‘100‘ for defining buffer sizes used in the program. Then, we take command-line arguments ‘device‘ and ‘user_msg‘. Check if exactly two arguments are passed **(‘argc != 3‘)**. Then, we copy the first argument **(argv[1] - device file name)** into the ‘device‘ array., and copy the second argument **(‘argv[2]‘ - message to write)** into the ‘user_msg‘ array. We Use the ‘open‘ system call to open the specified device file (‘device‘) in write-only mode **(‘O_WRONLY‘).**, and check if the file descriptor (‘fd‘) returned by ‘open‘ is ‘-1‘ (indicating failure to open). Then, we use the ‘write‘ system call to write the contents of ‘user_msg‘ to the device file (‘fd‘), with the number of bytes determined by ‘strlen(user_msg)‘.Then, we print the number of bytes successfully written to the device file (‘bytes_written‘) and close the file descriptor **(‘close(fd)‘).** To use this program, compile it and execute it with two command-line arguments:

```
gcc writer.c -o writer
./writer /dev/my_device1 "message_to_write"
```

This will open the device file **‘/dev/my_device1‘**, write the message to it, and then print the number of bytes written before closing the file.

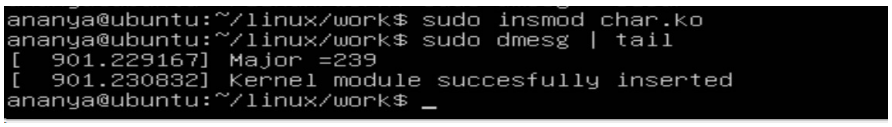

Figure 5: Module Inserted successfully

Figure 6: Character Device Driver files

3 Reader

Reader C program is designed to read data from a device file **(‘devmy_device0‘)**, which acts as a reader. We have defined ‘MAX_BUF_SIZE‘ as ‘1024‘ for defining buffer sizes used in the program. We take a single command-line argument (‘device‘ - the path to the device file to read from), and check if exactly one argument is passed **(‘argc != 2‘)**. Then, we copy the command-line argument (‘argv[1]‘) into the ‘device‘ array. We use the ‘open‘ system call to open the specified device file (‘device‘) in read-only mode **(‘O_RDONLY‘)**, and check if the file descriptor (‘fd‘) returned by ‘open‘ is ‘-1‘ (indicating failure to open). Then, we use the ‘read‘ system call to read data from the device file (‘fd‘) into the ‘user_msg‘ buffer. To ensure space for null-termination, the maximum number of bytes to read is limited by **‘MAX_BUF_SIZE - 1‘**. and we print the data read from the device file (‘user_msg‘) as a string and close the file descriptor **(‘close(fd)‘)**.

To use this program, compile it and execute it with one command-line argument:

```
gcc reader.c -o reader
./reader /dev/my_device0
```

This will open the device file **‘/dev/my_device0‘**, read data from it, display the read data, and then close the file.

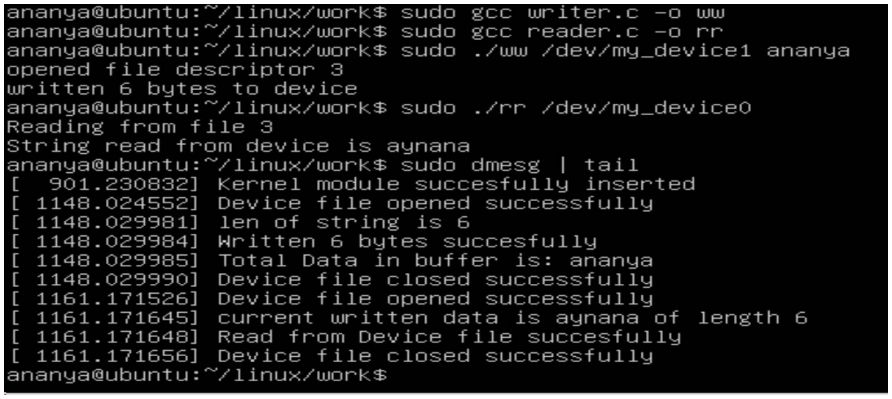

Figure 7: Final Working Output