FINAL CAPSTONE PROJECT: Linux device driver for System metrics

Introduction:

This project involves creating a character device driver for Linux. A character device driver allows the operating system to interact with hardware devices. We will learn how to write and manage this type of driver, including registering it with the kernel. The project includes implementing basic file operations such as read and write. Through this hands-on experience, we aim to gain a deeper understanding of kernel programming and how the Linux operating system functions at a low level.

Motivation:

This project is motivated by a desire to gain practical experience with Linux kernel development and system programming. By working on a character device driver, we can learn how device drivers enable communication between the operating system and hardware. This project will enhance our skills in C programming and debugging complex issues. Additionally, knowledge of kernel programming and device drivers is valuable for many tech careers, making this project beneficial for future job prospects. The challenge and complexity of the project also offer a great opportunity for personal and professional growth.

Software Requirements:

- Linux Operating System: A modern Linux distribution (e.g., Ubuntu, Fedora) with a development environment.
- Linux Kernel Headers: Matching version of the kernel headers installed for compiling the driver.
- GCC Compiler: GNU Compiler Collection (GCC) for compiling the kernel module.
- Make Utility: make tool for managing the build process of the kernel module.
- Text Editor/IDE: Text editors like Vim, Nano, or IDEs like VS Code, for writing and editing code.
- Kernel Debugging Tools: Tools like dmesg for viewing kernel messages and gdb for debugging.
- Version Control System (optional): Git for version control to manage the codebase and track changes.

Source code:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/device.h>
#include <linux/timer.h>
#include <linux/slab.h>

#define DEVICE_NAME "char_device" // Name of the device
#define CLASS_NAME "chardev"      // Name of the device class

// Global variables
static int majorNumber;           // Major number assigned to the device
static char *message;             // Buffer to store messages
static short messageSize;         // Size of the message
static struct class *charClass = NULL; // Device class pointer
static struct device *charDevice = NULL; // Device pointer
static struct timer_list metrics_timer;  // Timer to periodically update metrics
static int cpu_usage;             // Placeholder for CPU usage
static int mem_usage;             // Placeholder for memory usage
static int disk_io;               // Placeholder for disk I/O usage

// Function prototypes
static int metrics_dev_open(struct inode *, struct file *);
static int metrics_dev_release(struct inode *, struct file *);
static ssize_t metrics_dev_read(struct file *, char *, size_t, loff_t *);
static ssize_t metrics_dev_write(struct file *, const char *, size_t, loff_t *);
static void update_metrics(struct timer_list *t);
```

```
// File operations structure
static struct file_operations fops = {
    .open = metrics_dev_open,
    .read = metrics_dev_read,
    .write = metrics_dev_write,
    .release = metrics_dev_release,
};

// Initialization function for the module
static int __init metrics_init(void) {
    printk(KERN_INFO "MetricsDevice: Initializing the MetricsDevice LKM\n");

    // Register the character device
    majorNumber = register_chrdev(0, DEVICE_NAME, &fops);
    if (majorNumber < 0) {
        printk(KERN_ALERT "MetricsDevice failed to register a major number\n");
        return majorNumber;
    }
    printk(KERN_INFO "MetricsDevice: registered correctly with major number %d\n", majorNumber);

    // Create the device class
    charClass = class_create(CLASS_NAME);
    if (IS_ERR(charClass)) {
        unregister_chrdev(majorNumber, DEVICE_NAME);
        printk(KERN_ALERT "Failed to register device class\n");
        return PTR_ERR(charClass);
    }
    printk(KERN_INFO "MetricsDevice: device class registered correctly\n");

    // Create the device
    charDevice = device_create(charClass, NULL, MKDEV(majorNumber, 0), NULL, DEVICE_NAME);
    if (IS_ERR(charDevice)) {
```

```
        class_destroy(charClass);
        unregister_chrdev(majorNumber, DEVICE_NAME);
        printk(KERN_ALERT "Failed to create the device\n");
        return PTR_ERR(charDevice);
    }
    printk(KERN_INFO "MetricsDevice: device class created correctly\n");

    // Initialize and start the timer
    timer_setup(&metrics_timer, update_metrics, 0);
    mod_timer(&metrics_timer, jiffies + msecs_to_jiffies(1000)); // 1 second interval

    // Allocate memory for the message buffer
    message = kmalloc(256, GFP_KERNEL);
    if (!message) {
        printk(KERN_ALERT "Failed to allocate memory for message buffer\n");
        return -ENOMEM;
    }

    return 0;
}

// Exit function for the module
static void __exit metrics_exit(void) {
    // Delete the timer
    del_timer(&metrics_timer);

    // Clean up device and class
    device_destroy(charClass, MKDEV(majorNumber, 0));
    class_unregister(charClass);
    class_destroy(charClass);
    unregister_chrdev(majorNumber, DEVICE_NAME);
```

```
    // Free the allocated memory
    kfree(message);

    printk(KERN_INFO "MetricsDevice: Goodbye from the LKM!\n");
}

// Called when the device is opened
static int metrics_dev_open(struct inode *inodep, struct file *filep) {
    printk(KERN_INFO "MetricsDevice: Device has been opened\n");
    return 0;
}

// Called when the device is closed
static int metrics_dev_release(struct inode *inodep, struct file *filep) {
    printk(KERN_INFO "MetricsDevice: Device successfully closed\n");
    return 0;
}

// Called when the device is read
static ssize_t metrics_dev_read(struct file *filep, char *buffer, size_t len, loff_t *offset) {
    int error_count = 0;

    // Prevent reading the device multiple times
    if (*offset > 0) {
        return 0;
    }

    // Format the metrics into the message buffer
    snprintf(message, 256, "CPU Usage: %d%%, Memory Usage: %d%%, Disk I/O: %d\n", cpu_usage,
mem_usage, disk_io);
    messageSize = strlen(message);

    // Copy the message to user space
```

```
    error_count = copy_to_user(buffer, message, messageSize);
    if (error_count == 0) {
        printk(KERN_INFO "MetricsDevice: Sent %d characters to the user\n", messageSize);
        *offset += messageSize;
        return messageSize;
    } else {
        printk(KERN_INFO "MetricsDevice: Failed to send %d characters to the user\n", error_count);
        return -EFAULT;
    }
}

// Called when data is written to the device
static ssize_t metrics_dev_write(struct file *filep, const char *buffer, size_t len, loff_t *offset) {
    snprintf(message, 256, "%s(%zu letters)", buffer, len);
    messageSize = strlen(message);
    printk(KERN_INFO "MetricsDevice: Received %zu characters from the user\n", len);
    return len;
}

// Timer callback function to update metrics
static void update_metrics(struct timer_list *t) {
    cpu_usage = 20; // Placeholder value for CPU usage
    mem_usage = 30; // Placeholder value for memory usage
    disk_io = 40; // Placeholder value for disk I/O

    // Restart the timer
    mod_timer(&metrics_timer, jiffies + msecs_to_jiffies(1000));
}

module_init(metrics_init);
module_exit(metrics_exit);
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Name");
MODULE_DESCRIPTION("A simple Linux char driver for system metrics");
MODULE_VERSION("0.1");
```

Implementation:

Save the code with filename as MetricsDevice.c

Create a Makefile:

Create a Makefile in the same directory. Here's an example of a simple Makefile for building your module:

```
obj-m += MetricsDevice.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Build the module:

In your terminal, navigate to the directory containing MetricsDevice.c and Makefile, then run:

```
make
```

This Builds the kernel module based on the provided Makefile.

Load the module:

To load the module into the kernel, use the insmod command:

```
sudo insmod your_module.ko
```

This Loads the compiled kernel module into the kernel.

Check if the module is loaded:

You can check if the module is loaded using the lsmod command:

```
lsmod | grep “MetricsDevice”
```

This verifies that the module is loaded by listing it.

Create device node:

You need to create a device node in /dev. First, find the major number assigned to your device. You can check the kernel log using:

```
sudo dmesg | grep “MetricsDevice”
```

Displays kernel messages related to the module for debugging.

Create a device file (if needed):

```
sudo mknod /dev/char_device c <major_number> 0
```

Creates a device file in /dev with appropriate major number. Interact with the device.

for example, if the major number is 240:

```
sudo mknod /dev/char_device c 240 0
```

Interact with the device:

You can now read from and write to the device using basic command-line tools.

- **Writing to the device:**

  ```
  echo "Hello, world!" | sudo tee /dev/char_device
  ```

  After writing to the device, you can read from it and check the kernel messages

- **Read from the Device:**

  ```
  sudo cat /dev/char_device
  ```

  Reads from the device to see the metrics output.

Unload the module:

When you are done, you can unload the module using the rmmod command:

```
sudo rmmod MetricsDevice
```

Unloads the kernel module from the kernel.

Clean up:

Clean up the build files using the make clean command:

```
make clean
```

Removes compiled files and cleans up the build environment.

Challenges Faced:

- Understanding the intricacies of kernel module programming.
- Debugging kernel code with limited error messages and specialized tools.
- Handling concurrent access to the device using synchronization mechanisms.
- Ensuring proper resource cleanup to avoid memory leaks.

- Integrating the driver with the Linux kernel without causing instability.
- Managing dependencies and compatibility with different kernel versions.

Conclusion:

Through this project, we successfully developed a functional character device driver for Linux. We gained practical experience in kernel module programming and system-level development, enhancing our skills in C programming, debugging, and managing interactions between kernel-space and user-space. Despite challenges related to synchronization, resource management, and compatibility, we overcame these hurdles and gained valuable insights into the workings of the Linux operating system. This project has laid a strong foundation for future work, where we can add more features and explore other types of device drivers.