WIPRO NGA Program – LSP Batch

Capstone Project Presentation – 31 July 2024

Project Title Here - Linux System Metrics Device Driver

Presented by – Sumayya Farheen

Introduction

- In modern computing systems, real-time monitoring of system performance and resource utilization is crucial for maintaining system stability, performance, and efficiency.
- With the growing complexity of systems and the increasing demand for high performance, it becomes imperative for administrators and developers to have access to accurate and timely metrics regarding CPU usage, memory utilization, and disk I/O operations.
- This project addresses the need by developing a Linux device driver capable of monitoring and reporting key system metrics through a character device interface. This device driver is designed to be loaded and unloaded dynamically from the kernel, allowing for flexible and on-demand monitoring of system performance.

Introduction

The main objectives of this project are as follows:

- **Dynamic Loading and Unloading:** The driver must be capable of being loaded into and unloaded from the kernel without causing system instability or errors.
- **Device File Management**: Upon loading, the driver will create a device file in the /dev directory, allowing userspace applications to interact with it. When unloaded, the device file should be properly removed.
- **System Metrics Reporting**: The primary functionality of the driver is to collect and provide real-time system metrics, including CPU usage, memory usage, and disk I/O statistics. These metrics will be retrieved and formatted in a user-readable format when the device file is read.
- **Error Handling**: The driver will include robust error handling to manage invalid operations and exceptional cases gracefully, ensuring stability and reliability.
- **Timer-Based Updates**: To ensure metrics are up-to-date, the driver will utilize a timer to periodically collect and update system performance data.

Motivation

- Effective system performance monitoring is critical for optimizing resource usage and troubleshooting potential issues in a timely manner.
- By providing a character device driver that facilitates the retrieval of system metrics, this project aims to empower system administrators and developers with a tool for enhanced visibility into system operations.
- This, in turn, supports better decision-making and system management, ultimately leading to improved system performance and reliability.

Various Applications used in this project

1. System Monitoring Tools

Custom Monitoring Dashboards: Integrate the device driver with graphical dashboards or command-line tools that visualize system metrics in real time. For instance, a custom GUI application could periodically read from the device file and display CPU usage, memory consumption, and disk I/O statistics in graphical charts.

Performance Analytics Software: Use the metrics provided by the driver as input for performance analytics tools that analyze historical data and generate performance reports.

2. System Administration Utilities

Resource Management Tools: Develop utilities that automate resource management based on real-time metrics. For example, a tool could adjust system configurations or alert administrators when certain thresholds are reached.

Various Applications used in this project

Alerting Systems: Integrate with alerting systems to trigger notifications (e.g., emails, SMS) when system metrics indicate potential issues such as high CPU usage or low memory.

3. Educational and Research Applications

Educational Tools: Use the device driver as part of educational tools or demonstrations to teach concepts related to operating systems, kernel programming, and system performance monitoring.

Research Projects: Employ the metrics for research projects focused on system performance analysis, resource utilization patterns, or kernel-level performance optimization.

4. Diagnostic Tools

Troubleshooting Utilities: Incorporate the metrics into diagnostic tools to help identify and resolve performance issues. The metrics can provide insights into resource bottlenecks.

Modules that I have worked in this project

Device Driver Module: This is the core module that implements the character device driver. It handles the loading and unloading of the driver, as well as the interactions with the device file.

Key Components:

Initialization and Cleanup: Functions to initialize the driver (e.g., module_init()) and clean up resources on unload (e.g., module_exit()).

File Operations: Implementations of functions such as open(), read(), write(), and release() to interact with the device file.

Character Device Registration: Registering the device with the kernel and creating a device file in /dev.

Metrics Collection Module: This module is responsible for gathering system metrics such as CPU usage, memory usage, and disk I/O statistics.

Modules that I have worked in this project

Key Components:

System Metrics Retrieval: Functions or routines to collect and format system metrics. For example, reading from /proc or using kernel APIs to get system statistics.

Buffer Management: Handling and formatting of data to be presented to user space applications.

User Interface Module: This module facilitates interaction with the device driver from user space, allowing for the reading of system metrics.

Key Components:

Character Device Interface: The file operations (open(), read(), write(), release()) that provide access to the device file from user space.

Device File Management: Creation and deletion of the device file in the /dev directory.

List of Functions

1. Device Driver Module

Initialization and Cleanup

int __init my_module_init(void);

Initializes the module, registers the character device, and sets up the timer.

void __exit my_module_exit(void);

Cleans up resources, unregisters the device, and deletes the timer.

File Operations

int device_open(struct inode *inode, struct file *file);

Opens the device file.

ssize_t device_read(struct file *file, char __user *buf, size_t len, loff_t *offset);

Reads from the device file, providing system metrics to the user.

int device_release(struct inode *inode, struct file *file);

Closes the device file.

ssize_t device_write(struct file *file, const char __user *buf, size_t len, loff_t *offset);

Handles write attempts to the device file (typically results in an error).

List of Functions

Character Device Management

static int register_device(void);

Registers the character device and creates a device file.

static void unregister_device(void);

Unregisters the character device and removes the device file.

2. Timer Management Module

Timer Setup and Callback

void setup_timer(void);

Initializes and starts the timer for periodic metric updates.

void my_timer_callback(struct timer_list *t);

Callback function executed by the timer to update system metrics.

Timer Cleanup

void cleanup_timer(void);

Stops and cleans up the timer when the module is unloaded.

Challenges

1. Kernel Programming Complexity

Understanding Kernel APIs: Interfacing with kernel APIs can be complex, particularly when dealing with system metrics. Proper understanding of these APIs and how to use them correctly is crucial.

Kernel Module Development: Writing and debugging kernel code requires a deep understanding of kernel internals and the kernel's interaction with hardware and system resources.

2. System Metrics Collection

Accurate Data Retrieval: Collecting accurate and up-to-date system metrics (CPU usage, memory, disk I/O) can be challenging. Ensuring the metrics are precise and reflect real-time usage requires careful implementation.

Performance Overhead: Implementing metrics collection in a way that minimizes performance overhead on the system can be difficult. Excessive or inefficient data collection can impact system performance.