Linux Device Driver Coding Model Evaluation System

Assignment Overview

You are tasked with building an evaluation framework to benchmark coding models specifically for their ability to generate high-quality Linux device drivers.

Problem Statement

Objective: Design and implement an evaluation system that can assess how well AI coding models perform at writing Linux device drivers.

Context: Linux device drivers are critical system components. Writing them requires a deep understanding of:

- Kernel APIs and data structures
- Hardware interaction protocols
- Memory management in kernel space
- Concurrency and synchronization
- Error handling and resource cleanup
- Performance optimization

(prior knowledge of the above is not required for this assignment)

Your evaluation system should be able to test coding models against various device driver scenarios and provide meaningful metrics about their performance. For instance, ask a coding model to generate Linux device driver using prompts like:

"Create a simple character device driver that supports basic read/write operations with a 1KB internal buffer."

Or

"Implement a platform device driver for a memory-mapped GPIO controller with interrupt support."

And measure the output code quality using your custom evaluation metrics automatically.

Requirements

Core Components

1. **Metrics and Scoring System**

    - Quantitative metrics (compilation success rate, test coverage, etc.)
    - Qualitative metrics (code style, best practices adherence)
    - Weighted scoring based on criticality of different aspects
2. **Code Evaluation Engine**
    - Automated scoring of generated code for above metrics using:
        i. Static analysis for common kernel programming patterns
        ii. A script (Bash/Python) or small clang-tidy plugin that scans C source files and flags violations.

Sample Evaluation Metrics

Python
```python
# Example metrics structure
evaluation_metrics = {
    "compilation": {
        "success_rate": 0.85,
        "warnings_count": 12,
        "errors_count": 3
    },
    "functionality": {
        "basic_operations": 0.90,
        "error_handling": 0.75,
        "edge_cases": 0.60
    },
    "security": {
        "buffer_safety": 0.95,
        "race_conditions": 0.80,
        "input_validation": 0.70
    },
    "code_quality": {
        "style_compliance": 0.88,
        "documentation": 0.65,
        "maintainability": 0.75
    },
    "overall_score": 78.5
}
```

Deliverables

1. **Source Code**: Complete implementation on Github repo. Abstain from pushing the whole code at once, your commit history will help us judge your building process.
2. **Documentation**:
    - System architecture document
    - Rubrics explanation used for evaluations
    - User guide for running evaluations
3. **Test Results**: Sample evaluation of linux driver code generated from at least one coding model
4. **Presentation**: 15-minute demo of the system functionality
5. Ideally, spend no more than 2 days working on this. We are primarily interested in the approach you take in ideation and implementation.

Bonus Challenges

- **Advanced Code Evaluation:** Compilation validation against different kernel versions or Runtime testing.
- **Multi-architecture Support**: Test drivers for ARM, x86_64, and RISC-V
- **Regression Testing**: Automated testing against kernel updates
- **Model Fine-tuning**: Suggestions for improving model performance based on evaluation results
- How your approach can scale with an **LLM pipeline**

Resources and Reading

- Linux Kernel Documentation
- Linux Device Drivers Book (3rd Edition)
- Kernel coding style guide: coding-style.rst
- Feel free to use any LLMs to accelerate your learning and implementation, but specify in the final output where you have used them and make sure you understand the details of all critical pieces of the implementation.

Your evaluation Criteria for a prod level Linux kernel code may include (but not limited to)

1. Correctness (40%)

- **Compilation Success**: Does the code compile without errors?
- **Functionality**: Does it perform the required operations?
- **Kernel Integration**: Proper use of kernel APIs and conventions?

2. Security & Safety (25%)

- **Memory Safety**: No buffer overflows, proper bounds checking
- **Resource Management**: Proper allocation/deallocation patterns
- **Race Conditions**: Appropriate use of locking mechanisms
- **Input Validation**: Sanitization of user/hardware inputs

3. Code Quality (20%)

- **Style Compliance**: Adherence to Linux kernel coding style
- **Error Handling**: Comprehensive error checking and recovery
- **Documentation**: Proper comments and function documentation
- **Maintainability**: Clean, readable, and well-structured code

4. Performance (10%)

- **Efficiency**: Optimal use of system resources
- **Scalability**: Performance under load
- **Memory Usage**: Minimal memory footprint

5. Advanced Features (5%)

- **Power Management**: Suspend/resume support where applicable
- **Device Tree Support**: Modern kernel device enumeration
- **Debugging Support**: Integration with kernel debugging facilities

---