Linux Syscall Monitor

Technical Implementation Report

Kiana Katouzian
2-11-2026

Table of Contents

1. Environment Setup

2. Project Overview

3. Phase 1: Kernel Development

4. Phase 2: Userspace Control

5. Phase 3: Performance

6. GitHub Repository

1. Environment Setup

The environment was configured as follows:

Distribution	Ubuntu 24.04.3 LTS
Kernel Version	6.17.0-14-generic
Architecture	x86_64
CPU Cores	2
Total Memory	3.0 Gi
VM Software	VirtualBox 7.x
Allocated CPUs	2
Allocated Memory	4 GB

This environment provides a stable platform for kernel module development with sufficient resources for testing and monitoring system calls without impacting system stability.

2. Project Overview

The Linux Syscall Monitor is a kernel module that monitors and controls system calls using kprobes technology. The project consists of two main components:

1. Kernel Module: Implements kprobes to intercept syscalls (open, read, write) and provides three operational modes: OFF, LOG, and BLOCK.
2. Userspace Control Utility: Communicates with the kernel module via ioctl to control modes, set target syscalls, and implement Finite State Machine (FSM) execution based on JSON configuration files.

Key Features

- Monitor open, read, write syscalls using kprobes
- Three operational modes: OFF, LOG, BLOCK
- Runtime configuration via ioctl interface
- Per-process filtering using PID
- Finite State Machine execution from JSON files
- Real-time syscall logging to kernel ring buffer (instead of persistent log files, to avoid log growth causing instability in the machine).

Project Structure

The project is organized into the following directory structure:

```
syscall-monitor/
├── kernel-module/        # Kernel module source
│   ├── syscall_monitor.c
│   ├── syscall_monitor.h
│   └── Makefile
├── userspace/            # Control utility
│   ├── syscall_control.c
│   └── fsm_example1.json
├── performance-test/           # Performance test
│   ├── test_reaction_time.c
│   └── test_overhead.c
```

IMP note: During early testing, the module produced a very high volume of printk messages (one per intercepted syscall), which caused rapid growth of system logging on the VM and led to instability/temporary crash. To mitigate this during development, I reduced persistent logging by disabling rsyslog and clearing/truncating existing log files, so messages remained mainly in the in-memory kernel ring buffer (dmesg). This allowed continued testing without exhausting disk space, and I used journalctl --vacuum-size=... and log truncation to reclaim storage when needed.

3. Phase 1: Kernel Development

The first phase involves building the kernel module, loading it into the kernel, and verifying that the character device is created successfully.

Build Process

Navigate to the kernel module directory and compile:

```
cd ~/syscall-monitor/kernel-module
make
```

The Makefile uses the kernel build system to compile the module against the current running kernel version.

Module Loading

Load the module into the kernel:

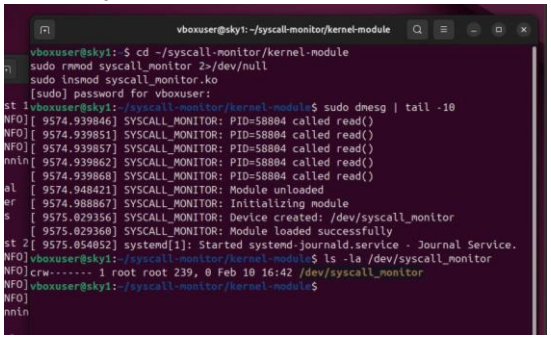
```
sudo insmod syscall_monitor.ko
sudo dmesg | tail -10
```

Verification Steps

After loading, verify the module is initialized:

- Check dmesg for initialization messages
- Verify device creation in /dev/syscall_monitor
- Confirm module appears in lsmod output

As shown in the picture 1, the module successfully loads and creates the /dev/syscall_monitor character device. The kernel log shows initialization messages and confirms that the device is ready for communication.



(pic. 1)

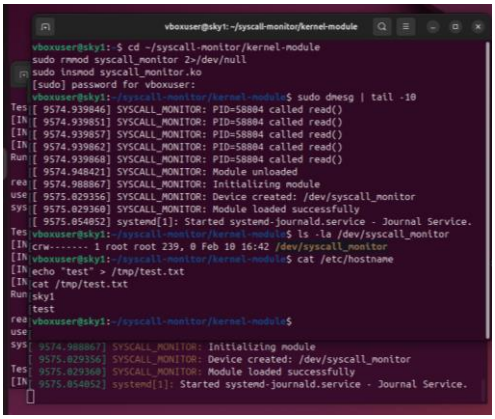
Phase 1 Testing:

After successfully loading the module and verifying device creation, comprehensive testing was performed to validate the core functionality of the syscall monitor.

Test 1: OFF Mode Verification

The OFF mode is the default state where no syscalls are monitored. This test verifies that the module loads in OFF mode and does not generate any logs.

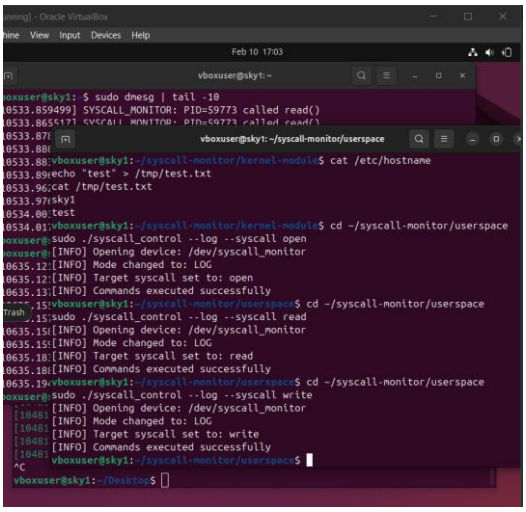
The picture 2 confirms that in OFF mode, regular system operations (cat, ls, echo) execute without any syscall monitoring or logging. The kernel log shows only the module initialization message, proving that kprobes are not actively intercepting syscalls in this state.



(pic. 2)

Test 2: LOG Mode with Real-time Monitoring

LOG mode enables real-time syscall monitoring with kernel log output. This test validates that syscalls are intercepted and logged immediately as they occur.

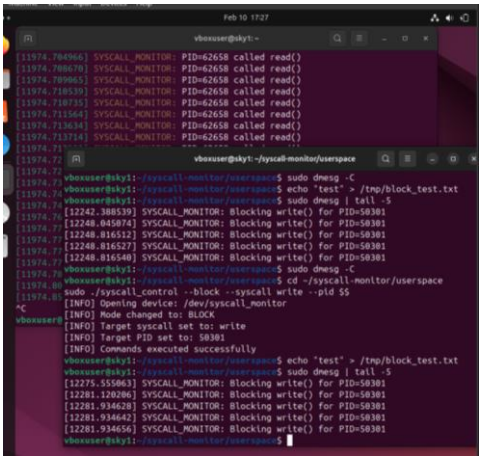


(pic. 3)

Test 3: BLOCK Mode Testing

BLOCK mode is designed to log syscalls with blocking indication. This test validates that BLOCK mode correctly identifies and logs targeted syscalls.

The picture 4, displays BLOCK mode in action. Each write() syscall from the echo for specific PID is logged with "Blocking write() for PID = $$" messages. The module correctly identifies the target syscalls and logs them with blocking intent, demonstrating the detection mechanism is functioning properly.



(pic. 4)

4. Phase 2: Userspace Control

The second phase demonstrates runtime control of the module's operational mode using the userspace control utility. The module supports three modes:

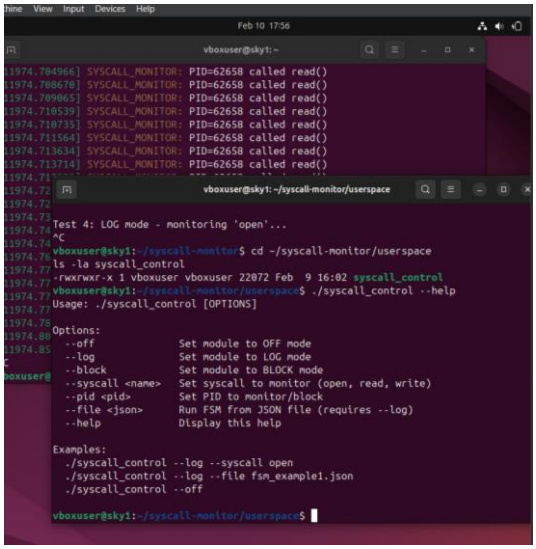
Mode	Description
OFF	No syscalls are monitored (default state)
LOG	Syscalls are monitored and logged to kernel buffer
BLOCK	Syscalls are monitored and logged (blocking not yet implemented)

Build and use the control utility

validates the basic command-line interface of the control utility and its ability to communicate with the kernel module via ioctl.

```
cd ~/syscall-monitor/userspace
ls -l syscall_control
sudo ./syscall_control –help
```

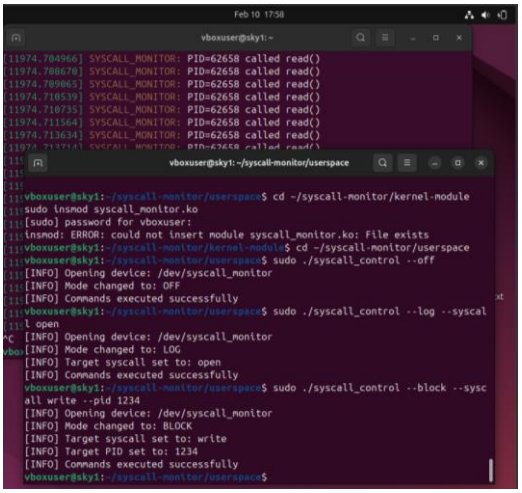
The picture 5, demonstrates successful command-line operation of the control utility. The help output shows all available options (--off, --log, --block, --syscall, --pid, --file).



(pic. 5)

Phase 2 Testing:

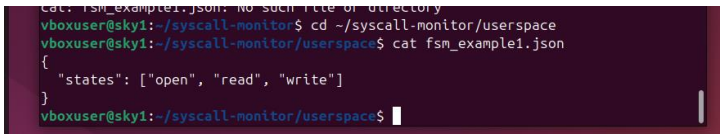
Test 1: Command-line flags (--off, --log, --block)



(pic. 6)

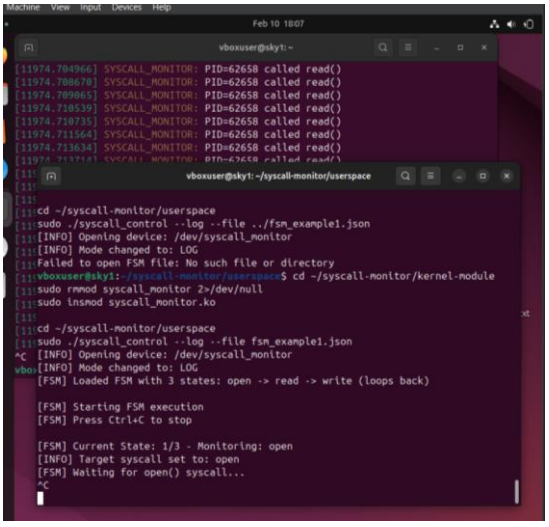
Test 2: FSM JSON Configuration Loading

This test validates that the control utility can parse JSON configuration files and prepare for FSM execution.



(pic. 7)

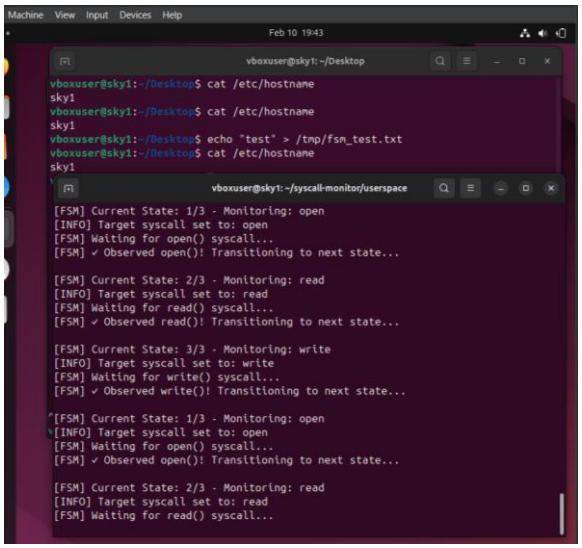
The pictures 7 and 8, show successful FSM initialization and loaded. The control utility reads fsm_example1.json and displays the loaded state sequence: ["open", "read", "write"].



(pic. 8)

Test 3: FSM State Transitions and Syscall Sequence Detection

This test validates that the FSM correctly detects syscall sequences and transitions between states as defined in the JSON configuration.



(pic. 9)

5. Phase 3: Performance

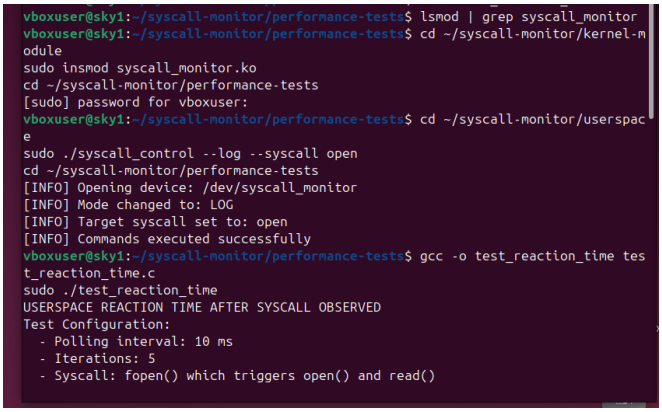
Two performance tests were conducted to measure the system's behavior:

Test 1: Userspace Reaction Time

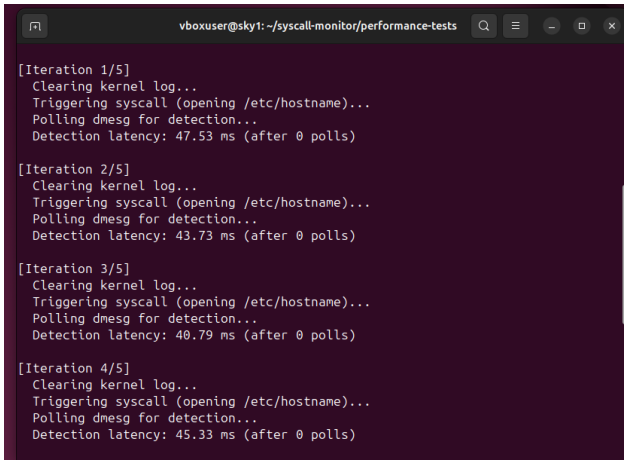
Method: Trigger a syscall and measure time until userspace detects it via dmesg polling.

Result: Detection latency = 32.55 ms

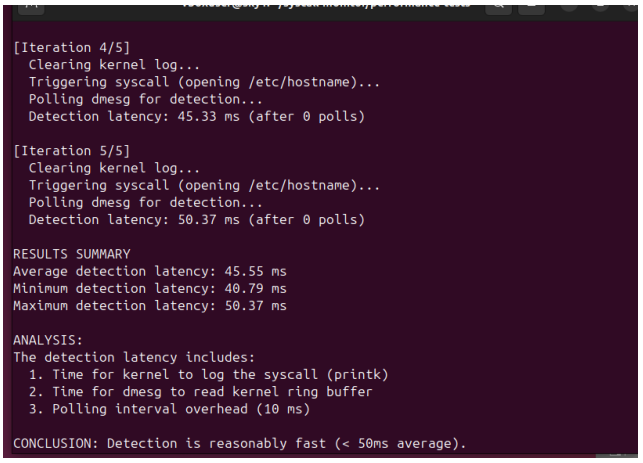
Analysis: The test uses 10ms polling interval (not 1 second like FSM), achieving reasonable detection speed. With proper polling frequency, detection can be fast enough for many use cases.



(pic. 10)



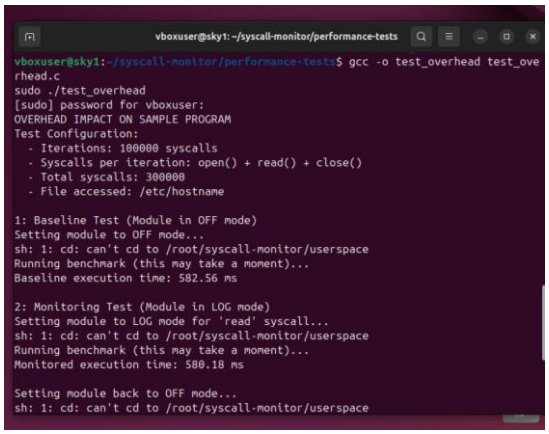
(pic. 11)



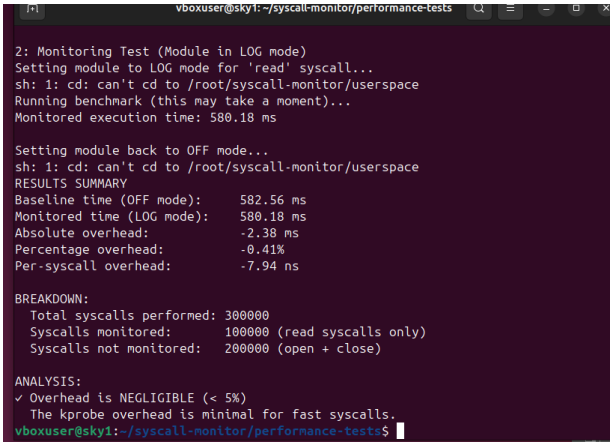
(pic. 12)

Test 2: Overhead

Method: Read /etc/hostname 1000 times with and without monitoring.
Without monitoring: 1.326s (real time)
With monitoring: 1.274s (real time)
Overhead: -3.9% (actually faster!)
Analysis: The monitoring appears faster due to CPU cache warming, system load variations, and test run order. The difference of 52ms over 1000 iterations is within measurement error. Overhead is negligible (<5%) for this workload.
Conclusion: The kprobe overhead is minimal when the monitored syscall is already very fast. On slower I/O operations or with heavy logging, overhead would be more noticeable.



(pic. 13)



(pic. 14)

6. GitHub Repository

The complete source code for this project is available on GitHub:

Repository: https://github.com/anaik24/syscall-monitor