Implementation and Analysis of Custom System Calls in Linux Kernel 6.8: A Comprehensive Study on Ubuntu 24.04 LTS

Anthony Barrantes Jiménez
Escuela de Computación
Instituto Tecnológico de Costa Rica
Cartago, Costa Rica
Email: antbarrantes@estudiantec.cr

Samir Cabrera Tabash
Escuela de Computación
Instituto Tecnológico de Costa Rica
Cartago, Costa Rica
Email: scabrera@estudiantec.cr

***Abstract*—This paper presents a comprehensive implementation and analysis of custom system calls within the Linux kernel 6.8 framework on Ubuntu 24.04 Long Term Support (LTS) environments. Our research establishes a systematic methodology for kernel modification, compilation, and validation through rigorous testing protocols. The implementation encompasses critical kernel file modifications and functionality validation via user-space testing. The resulting custom system call successfully demonstrates the bridge between user-space and kernel-space operations. Our methodology emphasizes systematic troubleshooting approaches with log analysis as the cornerstone for dependency resolution during kernel compilation. Experimental results validate successful integration with zero adverse system impact while maintaining compatibility with existing kernel subsystems and Application Binary Interface (ABI) specifications.**

***Index Terms*—System calls, Linux kernel, Ubuntu LTS, kernel development, operating systems, system programming**

I. Introduction

System calls constitute the fundamental Application Programming Interface (API) layer facilitating controlled communication between user-space applications and kernel-space services in contemporary Operating System (OS) architectures. Within Linux-based systems, these mechanisms provide secure access pathways to kernel functionalities through a standardized ABI [1].

The implementation of custom system calls provides valuable insights into kernel architecture internals and interaction mechanisms governing different privilege levels. This understanding proves essential for system programmers, security researchers, and OS developers seeking to extend kernel capabilities.

Ubuntu 24.04 LTS incorporating Linux kernel 6.8 establishes a robust foundation for kernel development activities. The LTS designation guarantees extended support lifecycles and enhanced stability, rendering it optimal for educational and research endeavors.

This research establishes a systematic framework for implementing custom system calls, emphasizing practical troubleshooting methodologies and log analysis approaches. Our contributions include: (1) demonstrating systematic custom system call implementation, (2) establishing troubleshooting methodologies for kernel compilation, (3) validating system call functionality through testing protocols, and (4) ensuring compatibility with existing kernel subsystems.

II. Related Work

System call implementation has been extensively investigated within OS research communities. Silberschatz et al. [1] provide comprehensive analysis of system call architectures across operating system implementations, emphasizing controlled kernel access mechanisms and security implications.

Love [2] presents detailed examination of Linux kernel development methodologies, including system call implementation strategies. The research emphasizes understanding kernel compilation processes and debugging techniques as fundamental requirements for kernel modifications.

Contemporary research by Kumar [3] documents system call implementation procedures within Linux kernel 6.0.9, offering practical compilation insights. However, this work demonstrates limited emphasis on systematic troubleshooting approaches essential for resolving compilation challenges.

Kerrisk [4] offers extensive documentation of Linux system programming interfaces, providing detailed coverage of system call mechanisms and their practical applications in system programming contexts.

III. Methodology

A. Development Environment Configuration

The development environment utilizes Ubuntu 24.04 LTS with Linux kernel version 6.8. The system configuration incorporates essential development tools including the GNU Compiler Collection (GCC) compiler suite, development libraries, and debugging utilities necessary for kernel compilation and testing.

The environment setup ensures optimal conditions through careful selection of development packages, proper system configuration, and validation of prerequisite components. This approach minimizes compilation issues and ensures consistent development results.

B. Implementation Protocol

The implementation requires systematic modification of three critical kernel components to register and implement the custom system call functionality.

1) System Call Table Registration: The system call table modification involves editing the file at `arch/x86/entry/syscalls/syscall_64.tbl` to register the new system call. This file contains the mapping between system call numbers and their corresponding kernel function names.

```
548    common    hello    sys_hello
```

Listing 1. System Call Table Entry

This registration assigns system call number 548 to our custom function, ensuring proper identification within the kernel's dispatch mechanism. The entry follows standard format where columns represent: system call number, architecture compatibility, system call name, and kernel function name.

2) System Call Implementation: The custom system call implementation utilizes the `SYSCALL_DEFINE0` macro, providing proper parameter handling and security validation mechanisms. The implementation demonstrates fundamental kernel operations while maintaining system security.

The function includes essential kernel headers: `linux/kernel.h` and `linux/syscalls.h`. The implementation uses `SYSCALL_DEFINE0` to create a system call named "Hola mundo!" with no parameters, employs `pr_info` for kernel logging, and returns zero indicating successful execution.

3) Header Declaration: The header modification involves updating `include/linux/syscalls.h` to include the function declaration. This ensures proper symbol resolution during kernel linking and maintains consistency with kernel function declaration standards.

The declaration follows standard format using `asmlinkage` calling convention and long return type characteristic of Linux system calls.

C. Kernel Compilation Framework

The kernel compilation process follows a systematic sequence:

1. Update configuration: `make olddefconfig` 2. Clean source tree: `make mrproper` 3. Generate default configuration: `make defconfig` 4. Parallel compilation: `make -j$(nproc)` 5. Install modules: `make modules_install` 6. Install kernel: `make install`

D. Log Analysis and Troubleshooting

Critical aspects include systematic analysis of compilation logs to identify and resolve dependency issues. Common dependency challenges include missing development headers for ncurses libraries, SSL development packages, and build tools such as bison and flex.

The resolution strategy emphasizes careful analysis of error messages, systematic identification of missing components, and targeted installation of development packages. This approach significantly reduces troubleshooting duration while ensuring successful compilation outcomes.

Fig. 1. Terminal output demonstrating successful system call execution with expected return value 0 and corresponding kernel message verification

E. User-Space Testing Protocol

The validation framework involves creating user-space programs that invoke the custom system call and verify functionality. The test program utilizes standard C library functions with direct system call invocation through the `syscall` function.

Testing combines compilation using `gcc`, execution under controlled conditions, and verification through kernel message logs analysis.

IV. Implementation Results

A. Successful Integration

The implementation achieved complete success with the custom system call properly integrated and functioning according to design specifications. The compilation process completed successfully after resolving dependency issues through systematic log analysis.

Figure 1 demonstrates successful execution, showing the expected return value of 0 and corresponding kernel message in system logs, validating both user-space invocation and kernel-space implementation.

B. System Integration Validation

The system call registration in slot 548 ensures compatibility with existing system calls while maintaining adherence to Linux ABI specifications. The implementation demonstrates proper integration with kernel logging systems through `pr_info` function utilization.

Comprehensive stability testing confirms zero adverse effects on kernel operation or system performance. The custom system call operates within expected parameters while maintaining complete system integrity.

V. Discussion

A. Implementation Success Factors

Success relies fundamentally on systematic approaches to kernel development, emphasizing log analysis and dependency

resolution methodologies. The structured approach to analyzing compilation errors enables rapid identification and targeted resolution of development challenges.

B. Technical Implications

This research provides practical guidance for kernel developers and students learning OS internals. The systematic methodology can be applied to complex kernel modification projects, establishing a foundation for advanced kernel development.

The implementation demonstrates that custom system calls can be successfully integrated into modern Linux kernels with minimal risk when proper procedures are followed.

C. Educational Value

The comprehensive documentation provides valuable educational material for understanding kernel architecture and system call mechanisms. The practical approach enables hands-on learning of complex OS concepts through direct implementation experience.

D. Limitations

While this implementation demonstrates basic system call functionality, more complex implementations may require additional considerations including parameter validation, security mechanisms, and performance optimization strategies.

VI. Conclusion

This research successfully demonstrates comprehensive implementation of custom system calls within Linux kernel 6.8 on Ubuntu 24.04 LTS. The systematic methodology emphasizing log analysis and structured troubleshooting provides practical guidance for kernel development projects.

The successful integration with zero adverse system impact validates the effectiveness of our systematic approach. The implementation serves as a foundation for complex kernel modification projects while providing insights into contemporary Linux kernel development methodologies.

Future research directions may explore sophisticated system call implementations, performance optimization techniques, security validation mechanisms, and advanced kernel debugging methodologies building upon the systematic foundation established through this research.

Acknowledgments

The authors acknowledge the Linux kernel development community for providing comprehensive documentation and the Ubuntu development team for maintaining stable development platforms. We extend gratitude to the open-source software community and the Instituto Tecnológico de Costa Rica for providing computational resources and academic support.

Acronyms

ABI Application Binary Interface. 1, 2

API Application Programming Interface. 1

GCC GNU Compiler Collection. 1

LTS Long Term Support. 1, 3

OS Operating System. 1, 3

References

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. John Wiley & Sons, 2018.

[2] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010.

[3] R. Kumar, "Add a system call to the linux kernel (6.0.9) in ubuntu 22.04 using oracle vm virtualbox," *Medium*, 2023.

[4] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, 2010.