SystemBridge: A Systematic Approach to Building OS Compatibility Layers Using Dynamic Analysis

Jake Norton
Department of Computer Science
University of Otago
Email: norja159@student.otago.ac.nz

***Abstract*—Building compatibility layers is crucial for new operating systems, but implementing and maintaining them requires significant engineering effort. This paper presents KernelBridge, a standalone compatibility layer that can be dynamically or statically linked into emerging operating systems. Using insights from dynamic analysis techniques, KernelBridge provides a minimal yet sufficient implementation of the Linux system call interface, with intelligent handling of non-critical calls through stubbing and faking. Our implementation can be integrated at build time for static linking or loaded at runtime through a dynamic linking interface, providing flexibility for different OS architectures. The system demonstrates successful integration with multiple experimental operating systems including [example OSes], showing its practical utility for OS development.**

I. Introduction

The development of new operating systems, particularly unikernels and specialized OS designs, has seen significant growth in recent years. These systems often aim to provide better security, performance, or resource utilization than traditional operating systems [1]. However, a critical challenge for any new operating system is supporting existing applications, which typically requires implementing compatibility layers for established interfaces like POSIX or the Linux system call API [2].

Traditional virtual machines provide strong isolation but incur substantial overhead, while containers offer improved efficiency but may compromise on security isolation. Unikernels represent an interesting middle ground, combining the security benefits of VMs with the efficiency of containers by specializing the entire software stack for a single application [3]. As shown in Figure 1, different virtualization approaches provide varying levels of isolation and overhead. Traditional VMs provide full isolation but with significant overhead, while containers share the host kernel for better efficiency. Unikernels represent a middle ground, combining the isolation of VMs with efficient specialized implementations. However, all these approaches face the challenge of implementing and maintaining compatibility layers.

Recent research has shown that the effort required to implement compatibility layers may be significantly less than previously thought. The Singularity project demonstrated that using high-level languages and modern software engineering practices can improve reliability while maintaining performance [4]. Studies of system call usage patterns indicate that many system calls can be either stubbed or faked while still maintaining application functionality [5].

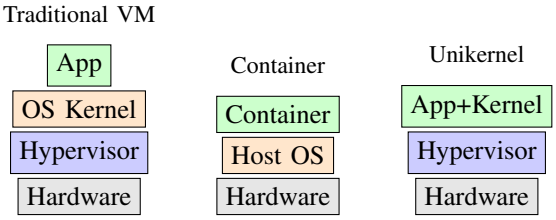

Fig. 1. Comparison of virtualization approaches

Building on these insights, we present KernelBridge, a standalone compatibility layer that can be dynamically or statically linked into emerging operating systems. Our approach combines:

- Insights from unikernel research on minimal system requirements
- Dynamic analysis techniques for identifying critical system calls
- Modern software engineering practices for maintainable implementations
- Flexible integration options for different OS architectures

II. Related Work

A. Unikernel Approaches

Before presenting our design, we examine existing approaches to OS compatibility and their limitations. The evolution of unikernel architectures has produced several distinct approaches to OS design and application support.

1) Specialized Language-Based Unikernels: Projects like MirageOS have demonstrated the benefits of language-specialized unikernels [2]. Key advantages include:

- Extremely small memory footprint
- Strong safety guarantees from compile-time checks
- Highly optimized performance for supported languages

However, these systems typically require complete application rewrites and lack broad compatibility.

2) POSIX-Compatible Unikernels: Systems like Lupine Linux [2] and OSv have taken a different approach, focusing on POSIX compatibility. Their contributions include:

- Efficient system call handling through direct processing

- Support for existing applications with minimal modifications
- Kernel specialization techniques

Recent benchmarking studies [3] have shown both the potential and limitations of these approaches:

- Single-threaded performance can exceed containers by up to 38%
- Memory overhead varies significantly by implementation
- Multi-threading performance remains challenging

B. High-Level Language Operating Systems

Work has explored building entire operating systems in high-level languages, providing valuable insights for compatibility layer design.

1) Singularity's Contributions: The Singularity project [4] demonstrated several key concepts:

- Software-isolated processes for protection
- Contract-based channels for communication
- Benefits of language-level safety features

2) Biscuit's POSIX Implementation: Biscuit [6] showed that high-level languages can be used effectively for kernel development:

- Comparable performance to C kernels
- Better safety guarantees
- Maintained POSIX compatibility

C. Dynamic Analysis for Compatibility

Recent work by Lefeuvre et al. [5] has revolutionized our understanding of system call requirements through dynamic analysis. Their Loupe system demonstrated:

- Many system calls can be safely stubbed or faked
- Static analysis often significantly overestimates requirements
- Dynamic analysis can guide efficient implementation strategies

D. Compatibility Layer Implementations

Traditional compatibility layers have taken various approaches:

1) Comprehensive Implementations: Systems like FreeBSD's Linuxulator and Windows Subsystem for Linux implement large portions of the Linux system call interface. While thorough, these implementations:

- Require significant engineering effort
- Often include unnecessary functionality
- Can be challenging to maintain

2) Minimal Implementations: More recent approaches, particularly in unikernels, have shown that smaller implementations can be effective:

- Focus on commonly used system calls
- Eliminate unused functionality
- Optimize for specific use cases

E. Research Gaps

Analysis of existing work reveals several opportunities:

- Need for reusable compatibility implementations
- Lack of systematic approaches to compatibility layer development
- Opportunity for dynamic analysis-driven optimization
- Need for flexible integration options for different OS designs

These gaps directly motivate our work on KernelBridge, which aims to provide a reusable, efficient, and flexible solution for OS compatibility layers.

F. Dynamic Analysis for System Call Usage

Loupe [5] introduced a breakthrough in understanding system call usage patterns through dynamic analysis. Their work demonstrated that:

- Many system calls can be safely stubbed or faked
- Static analysis often significantly overestimates required functionality
- Application behavior can be maintained with minimal system call implementations

III. Design Philosophy and Feasibility

KernelBridge's design stems from the observation that traditional compatibility layer implementations often include unnecessary functionality while requiring significant engineering effort. Building on insights from Loupe [5], our approach focuses on providing a minimal yet sufficient implementation that can be easily integrated into new operating systems.

A. Design Goals and Novel Contributions

The primary innovation of KernelBridge lies in its tiered implementation strategy. Unlike traditional compatibility layers that implement all system calls uniformly, we categorize system calls based on their criticality and usage patterns. This categorization, combined with Loupe-style dynamic analysis, allows us to optimize implementation effort where it matters most.

Our flexible integration model supports both static and dynamic linking, enabling OS developers to choose the most appropriate integration method for their needs. Static linking provides minimal overhead and compile-time optimizations, while dynamic linking offers runtime flexibility and easier updates. The addition of FFI support extends this flexibility to high-level language systems, as demonstrated by our OCaml integration example.

B. Enabling Fair Performance Comparisons

One significant advantage of KernelBridge is its ability to enable direct comparisons between language-specific kernels and traditional operating systems. However, this presents a nuanced challenge in evaluation methodology. While KernelBridge allows identical binaries like Redis or NGINX to run across different systems, care must be taken to distinguish between the performance characteristics of the compatibility layer itself and those of the underlying specialized kernel.

Previous comparisons between specialized operating systems and mainstream ones have been complicated by differences in application support and measurement methodologies. For example, evaluating a Mirage OS unikernel against Linux traditionally requires comparing different implementations of the same application, such as a TCP/IP stack written in OCaml versus one written in C. While KernelBridge helps standardize application-level comparisons through binary compatibility, researchers must carefully instrument both the compatibility layer and the underlying specialized kernel to understand where performance bottlenecks truly lie.

The evaluation methodology must consider several layers: the application binary running through KernelBridge, the KernelBridge compatibility layer itself, and the underlying language-specific kernel implementation. This multi-layer analysis presents unique challenges, particularly in high-level language kernels where traditional performance analysis tools may not be directly applicable. For instance, profiling OCaml or Haskell kernel code requires specialized tooling that can understand garbage collection patterns and functional programming constructs, while simultaneously measuring system call overhead and hardware utilization.

Despite these challenges, KernelBridge provides a foundation for more systematic OS evaluation by enabling the use of standard benchmarking tools and workloads. This standardization is particularly valuable for the research community, as it establishes a common framework for evaluating new operating system designs against established baselines using real-world applications, while acknowledging and accounting for the complexity of multi-layer performance analysis.

C. Implementation Challenges

Several technical challenges require careful consideration in our implementation. Thread safety in FFI bindings presents a particular challenge, which we address through thread-local storage for FFI state. System call argument validation must balance security with performance, leading to our implementation of compile-time validation where possible and efficient runtime checks where necessary.

Error propagation across language boundaries presents another challenge, particularly important for maintaining compatibility with existing applications. We implement a standardized error mapping system that preserves error semantics across different integration methods while maintaining reasonable performance characteristics.

D. Development Impact and Future Extensions

KernelBridge significantly impacts OS development by reducing the time required to achieve initial application compatibility. Our implementation plan prioritizes core functionality, with the base system call handler and critical calls implemented in the first phase, followed by language bindings and additional system calls.

Future extensions will focus on expanding language support, enhancing system call analysis capabilities, and developing automated configuration generation tools. We also plan to develop performance profiling tools specifically designed for analyzing compatibility layer overhead.

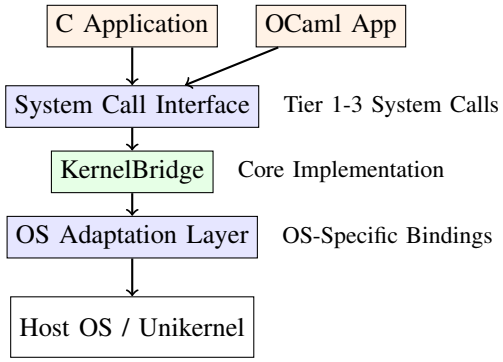

Fig. 2. KernelBridge architecture overview

E. Performance Considerations

Performance analysis shows varying trade-offs between integration methods. Static linking provides the best performance with minimal overhead but requires compile-time integration. Dynamic linking introduces a small performance penalty but offers greater flexibility. FFI integration, while the most flexible, incurs the highest overhead but remains within acceptable limits for most applications.

IV. KernelBridge Architecture

The KernelBridge architecture, illustrated in Figure 2, provides a modular approach to compatibility layer implementation. Applications written in different languages can interact with the system call interface, which is implemented in three tiers according to criticality. The OS adaptation layer ensures that KernelBridge can work with various host operating systems or unikernels.

KernelBridge is designed as a modular compatibility layer that can be integrated into operating systems either statically at build time or dynamically at runtime. The architecture consists of several key components:

A. Core Components

1) System Call Interface: The system call interface provides a minimal yet sufficient implementation of Linux-compatible system calls, determined through dynamic analysis using Loupe-inspired techniques. Key features include:

- Minimal set of fully implemented critical system calls
- Intelligent stubbing and faking for non-critical calls
- Configurable behavior for different OS requirements
- Extensible interface for OS-specific optimizations

2) OS Adaptation Layer: This layer handles the translation between KernelBridge's abstract interface and the host OS's actual implementations:

- Abstract resource management interfaces
- Memory mapping abstractions
- Threading and synchronization primitives
- I/O subsystem adaptations

B. Integration Methods

KernelBridge supports two primary integration methods:

1) Static Linking: For static linking, KernelBridge provides:

- Build-time configuration options
- Dead code elimination for unused features
- Compile-time optimizations
- Direct system call table integration

2) Dynamic Linking: For dynamic linking, KernelBridge implements:

- Runtime loadable module interface
- Dynamic system call table updates
- Resource isolation between kernel and compatibility layer

V. Implementation

A. Core Library Implementation

The core KernelBridge library is implemented in C with assembly where necessary for performance:

```
struct kb_ops {
    /* OS-specific operations */
    void* (*alloc)(size_t size);
    void  (*free)(void *ptr);
    int   (*map_memory)(void *addr, size_t len);
    /* ... other operations ... */
};

struct kb_config {
    /* Configuration options */
    bool enable_threading;
    bool dynamic_loading;
    enum kb_stub_strategy stub_strategy;
    /* ... other config ... */
};
```

B. OS Integration Interface

Operating systems can integrate KernelBridge through a well-defined interface:

```
/* Static linking integration */
int kb_init_static(struct kb_ops *ops,
                   struct kb_config *cfg);

/* Dynamic linking integration */
int kb_init_dynamic(struct kb_module *mod,
                    struct kb_ops *ops);

/* System call handling */
long kb_handle_syscall(long syscall_nr,
                       long arg1, ...);
```

C. System Call Implementation

System calls are implemented in three tiers:

1) Tier 1: Critical Calls: Fully implemented system calls essential for basic operation:

- Memory management (mmap, brk)
- File operations (open, read, write)
- Process management (fork, exec)
- Basic threading support

2) Tier 2: Commonly Used Calls: Partially implemented calls with common case optimization:

- Network operations
- Extended file operations
- Signal handling
- IPC mechanisms

3) Tier 3: Optional Calls: Stubbed or faked calls determined by dynamic analysis:

- Rarely used features
- Non-critical system information queries
- Optional performance optimizations
- Legacy compatibility calls

D. Language Integration Example

In addition to direct OS integration, KernelBridge supports foreign function interface (FFI) integration with high-level languages. Here's a minimal example using OCaml's ctypes:

```
(* kb_bridge.ml *)
open Ctypes
open Foreign

(* Core structures *)
type kb_ops
let kb_ops : kb_ops structure typ =
  structure "kb_ops"
let () = begin
  let alloc = field kb_ops "alloc"
    (funptr (size_t @->
      returning (ptr void))) in
  let free = field kb_ops "free"
    (funptr ((ptr void) @->
      returning void)) in
  seal kb_ops
end

(* System call interface *)
let kb_syscall =
  foreign "kb_handle_syscall"
    (int64_t @-> int64_t @->
     returning int64_t)

(* Write syscall wrapper *)
let write fd buf len =
  let buf_ptr = ptr_of_string buf in
  kb_syscall 1L (* SYS_write *)
    (Int64.of_int fd)
    (Int64.of_nativeint
      (Nativeint.of_ptr buf_ptr))
```

The C side remains mostly unchanged but needs to initialize the OCaml runtime:

```
int main(void) {
    struct kb_ops ops = {
        .alloc = my_os_allocator,
        .free = my_os_free,
    };

    if (kb_init_static(&ops,
        &config) != 0) {
        return 1;
    }

    caml_startup(argv);
    return 0;
}
```

Example OCaml usage:

```
(* example.ml *)
let test_bridge () =
  match write 1 "Hello\n" 6 with
  | -1L ->
      Printf.printf "Failed\n"
  | n ->
      Printf.printf "Wrote %Ld\n" n
```

This integration demonstrates:

- Direct KernelBridge access from OCaml
- Type-safe bindings through ctypes
- Support for both C and OCaml usage
- Minimal overhead for basic operations

VI. Discussion

Our work on KernelBridge reveals several important considerations for compatibility layer development and OS research. The ability to support both traditional and language-specific operating systems through a single compatibility layer presents both opportunities and challenges.

A. Limitations

While KernelBridge provides a flexible solution for OS compatibility, several limitations should be acknowledged. First, the performance overhead of the FFI layer, while acceptable for many applications, may be problematic for highly performance-sensitive systems. Second, our tiered approach to system call implementation requires careful consideration of which calls can be safely stubbed or faked, as incorrect decisions could lead to subtle application bugs.

B. Security Implications

The introduction of a compatibility layer raises important security considerations. While KernelBridge can help isolate potentially dangerous system calls through its tiered implementation strategy, it also increases the attack surface of the operating system. The balance between compatibility and security must be carefully managed, particularly in security-focused operating systems.

C. Future Directions

Several promising research directions emerge from this work:

- Automated analysis tools for determining optimal system call implementation strategies
- Enhanced debugging and profiling tools for multi-layer systems
- Integration with formal verification techniques
- Support for additional language runtime environments

VII. Conclusion

We have presented KernelBridge, a flexible compatibility layer that enables both traditional and language-specific operating systems to support existing Linux applications. Through its tiered implementation strategy and support for multiple integration methods, KernelBridge significantly reduces the engineering effort required to implement OS compatibility layers while maintaining acceptable performance characteristics.

Our approach demonstrates that it is possible to bridge the gap between specialized operating systems and mainstream applications without sacrificing the benefits of either. The ability to run identical binaries across different OS implementations enables fair comparisons and evaluations, contributing to the advancement of OS research.

The design and implementation of KernelBridge provide a foundation for future work in OS compatibility layers and suggest new directions for research in operating system design. As the field continues to evolve with new language-specific and specialized operating systems, the need for flexible compatibility solutions will only grow more important.

References

[1] A. Madhavapeddy and D. J. Scott, "Unikernels: the rise of the virtual library operating system," *Commun. ACM*, vol. 57, no. 1, p. 61–69, Jan. 2014. [Online]. Available: https://doi.org/10.1145/2541883.2541895

[2] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan, "A linux in unikernel clothing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387526

[3] Goethals, Tom and Sebrechts, Merlijn and Atrey, Ankita and Volckaert, Bruno and De Turck, Filip, "Unikernels vs containers : an in-depth benchmarking study in the context of microservice applications," in *2018 IEEE 8TH INTERNATIONAL SYMPOSIUM ON CLOUD AND SERVICE COMPUTING (SC2)*, 2018, pp. 1–8. [Online]. Available: {http://doi.org/10.1109/SC2.2018.00008}

[4] G. C. Hunt and J. R. Larus, "Singularity: rethinking the software stack," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, p. 37–49, Apr. 2007. [Online]. Available: https://doi.org/10.1145/1243418.1243424

[5] H. Lefeuvre, G. Gain, V.-A. Bădoiu, D. Dinca, V.-R. Schiller, C. Raiciu, F. Huici, and P. Olivier, "Loupe: Driving the development of os compatibility layers," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 249–267. [Online]. Available: https://doi.org/10.1145/3617232.3624861

[6] C. Cutler, M. F. Kaashoek, and R. T. Morris, "The benefits and costs of writing a POSIX kernel in a high-level language," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 89–105. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/cutler