Userspace: A Standard Library for Userspace Applications

Rust Implementation for Operating System Interfaces

José Gois
ze.gois.00@gmail.com

Federal University of Rio Grande do Norte (UFRN)

August 31, 2025

Outline

Introduction

- **Userspace** is a Rust-based standard library for userspace applications
- Provides a cross-platform abstraction layer for operating system interfaces
- Built with a `no_std` environment in mind
- Focused on memory safety, portability, and performance
- Aims to provide a consistent API across different architectures and operating systems

Project Objectives

- Create a modern alternative to libc for systems programming
- Ensure memory safety through Rust's ownership model
- Abstract architecture-specific details (currently targeting x86_64)
- Implement ELF format parsing and handling
- Provide low-level memory management utilities
- Enable cross-platform development of userspace applications

Project Architecture

The library is structured in a modular way:

- **Core Library**: Basic types, traits, and utilities
- **Target Layer**: Architecture and OS-specific implementations
- **Memory Management**: Stack handling, page allocation
- **File Formats**: ELF parsing and interpretation
- **Error Handling**: Custom result types

Technology Stack

- **Rust**: Memory-safe systems programming language
- **No Standard Library**: Using `#![no_std]` for bare-metal compatibility
- **Rust Features**:
  - Generic const expressions
  - Trait implementations
  - Unsafe code blocks for low-level operations
- **Build System**: Cargo with custom build scripts
- **Linker Configuration**: Custom linking via linker.ld

Key Implementation Concepts

```rust
// No standard library dependency
#![no_std]
#![allow(incomplete_features)]
#![feature(generic_const_exprs)]

// Core library structure
pub mod macros;
pub mod target;
pub mod file;
pub mod memory;
pub mod traits;
pub mod types;
pub mod result;
```

Memory Management Implementation

- Implemented a stack parsing system that can:
  - Read command-line arguments from stack memory
  - Safely navigate the stack structure
  - Extract environment variables
- Developed page allocation mechanisms
- Created safe wrappers around raw pointers
- Implemented trait-based memory operations

ELF File Format Handling

- Implemented ELF header parsing:

```
// Example from project
if !arg0.pointer.0.is_null() {
  unsafe {
    let cstr = core::ffi::CStr::from_ptr(
      arg0.pointer.0 as *mut i8);
    let self_path = cstr.to_str().unwrap();
    let identifier =
      userspace::file::format::elf::header::Identifier::
          from_path(
        self_path);
  }
}
```

- Created type-safe representations of ELF structures
- Ensured proper endianness handling

Architecture Abstraction

- Implemented architecture-specific traits:
  - Pointer types and operations
  - Register access
  - Memory layout definitions
- Current focus on x86_64 architecture
- Design allows for easy extension to other architectures
- Architecture-specific code isolated in dedicated modules

Cross-Platform Considerations

- Operating system abstractions:
  - System calls
  - File operations
  - Process management
- Platform detection macros
- Feature flags for enabling/disabling functionality
- Consistent error handling across platforms

Challenges Encountered

- Working without the standard library required reimplementation of basic functionality
- Handling architecture-specific details while maintaining a clean API
- Balancing safety and performance in low-level operations
- Creating robust error handling without exceptions
- Managing compile-time features and conditional compilation

Achievements

- Successfully implemented a foundational userspace library in Rust
- Created safe abstractions for low-level system operations
- Developed modular architecture for extensibility
- Implemented ELF parsing capabilities
- Established memory management primitives
- Created cross-platform architecture and OS abstractions

Future Work

- Extend support to additional architectures (ARM, RISC-V)
- Implement more comprehensive file system operations
- Add networking capabilities
- Develop threading and concurrency primitives
- Create higher-level abstractions for common operations
- Improve documentation and examples
- Add comprehensive testing framework

Sustainability Connections (ODS)

This project contributes to the following UN Sustainable Development Goals:

- **Goal 9: Industry, Innovation and Infrastructure**
  - Contributing to technological innovation in systems software
  - Building resilient infrastructure for modern applications
- **Goal 4: Quality Education**
  - Open source nature promotes learning and collaboration
  - Provides educational resources for systems programming

References

- The Rust Programming Language. https://www.rust-lang.org/
- The Rust Embedded Book: A no_std Rust Environment. https://docs.rust-embedded.org/book/intro/no-std.html
- Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification.
- Userspace Project Repository. https://github.com/ze-gois/rust_userspace
- The Rustonomicon: The Dark Arts of Advanced and Unsafe Rust Programming. https://doc.rust-lang.org/nomicon/

Acknowledgments

- Federal University of Rio Grande do Norte (UFRN)
- Research advisor and mentors
- Rust community for their extensive documentation and support
- Open source contributors who inspired this work

Thank you!

Contact: ze.gois.00@gmail.com