Userspace: Uma Biblioteca Padrão para Aplicações em Userspace

José Gois
ze.gois.00@gmail.com

Universidade Federal do Rio Grande do Norte (UFRN)

Orientador: [Nome do Orientador]

31 de agosto de 2025

Resumo

Este trabalho apresenta o desenvolvimento de uma biblioteca padrão chamada *Userspace*, implementada em Rust, com o objetivo de fornecer abstrações seguras para aplicações em nível de sistema operacional. A biblioteca foi projetada para funcionar sem dependência da biblioteca padrão de Rust (`no_std`), tornando-a adequada para ambientes de baixo nível e sistemas embarcados. Implementamos abstrações para diferentes arquiteturas de hardware (com foco inicial em x86_64), gerenciamento de memória, manipulação de formatos de arquivo executáveis (ELF) e um sistema robusto de tratamento de erros. Os resultados mostram que é possível criar abstrações de alto nível para programação de sistemas mantendo a segurança de memória garantida pelo Rust, sem comprometer o desempenho. O projeto contribui para o campo de desenvolvimento de sistemas ao fornecer ferramentas modernas e seguras para programação de baixo nível.

Palavras-chave: Rust. Programação de Sistemas. Segurança de Memória. Arquitetura de Computadores. Formato ELF.

Resumo

This work presents the development of a standard library called *Userspace*, implemented in Rust, aiming to provide safe abstractions for system-level applications. The library was designed to work without depending on Rust's standard library (`no_std`), making it suitable for low-level environments and embedded systems. We implemented abstractions for different hardware architectures (with initial focus on x86_64), memory management, executable file format manipulation (ELF), and a robust error handling system. Results show that it is possible to create high-level abstractions for systems programming while maintaining the memory safety guaranteed by Rust, without compromising performance. The project contributes to the field of systems development by providing modern and secure tools for low-level programming.

Keywords: Rust. Systems Programming. Memory Safety. Computer Architecture. ELF Format.

ODS: Este projeto relaciona-se com o Objetivo 9 (Indústria, Inovação e Infraestrutura) da Agenda 2030 da ONU, contribuindo para o desenvolvimento de infraestrutura tecnológica resiliente e promovendo a inovação em software de sistemas.

1 Introdução

O desenvolvimento de software em nível de sistema operacional tradicionalmente envolve o uso de linguagens como C, que oferecem controle preciso sobre o hardware, mas carecem de garantias modernas de segurança de memória. Isso resulta em vulnerabilidades como buffer overflows, use-after-free e outros problemas que comprometem a segurança e estabilidade dos sistemas. A linguagem Rust surgiu como uma alternativa moderna, oferecendo o mesmo nível de controle de baixo nível com garantias de segurança de memória verificadas em tempo de compilação.

No entanto, o desenvolvimento de aplicações em nível de sistema usando Rust ainda enfrenta desafios. A biblioteca padrão de Rust (`std`) depende de um sistema operacional subjacente, tornando-a inadequada para muitos cenários de programação de sistemas. Além disso, as abstrações existentes para programação em nível de sistema em Rust muitas vezes são fragmentadas ou incompletas.

O projeto *Userspace* visa preencher essa lacuna fornecendo uma biblioteca padrão completa para desenvolvimento de aplicações em userspace, independente do sistema operacional subjacente. A biblioteca implementa:

- Abstrações seguras para diferentes arquiteturas de hardware (inicialmente x86_64)
- Gerenciamento de memória eficiente e seguro
- Manipulação de formatos de arquivo executáveis (como ELF - Executable and Linkable Format)
- Sistema robusto de tratamento de erros
- Interface consistente entre diferentes plataformas

Um dos principais diferenciais do projeto é sua abordagem baseada em traços (traits) e tipos genéricos, que permite uma extensibilidade significativa enquanto mantém a segurança de tipos em tempo de compilação. Isso permite que desenvolvedores adicionem suporte para novas arquiteturas e sistemas operacionais sem modificar o código existente.

A importância deste trabalho reside na crescente necessidade de software de sistema seguro e confiável em uma era onde sistemas embarcados, IoT e infraestrutura crítica são cada vez mais prevalentes. Ao fornecer abstrações seguras para programação de baixo nível, a biblioteca *Userspace* contribui para o desenvolvimento de sistemas mais robustos e menos vulneráveis a falhas de segurança.

2 Método

O desenvolvimento da biblioteca *Userspace* seguiu uma abordagem modular e baseada em componentes, com foco na segurança de tipos e memória. A metodologia adotada pode ser dividida nas seguintes etapas e componentes:

2.1 Ambiente de Desenvolvimento

O projeto foi desenvolvido utilizando Rust 2024 Edition, com recursos experimentais habilitados para suporte a expressões constantes genéricas e outros recursos avançados da

linguagem. Utilizamos o sistema de compilação Cargo para gerenciamento de dependências e compilação.

Configurações específicas foram aplicadas para permitir o desenvolvimento sem a biblioteca padrão de Rust:

```
#![no_std]
#![allow(incomplete_features)]
#![feature(generic_const_exprs)]
#![feature(generic_const_items)]
```

Um script de build personalizado (`build.rs`) foi implementado para gerenciar a compilação de componentes específicos da arquitetura e integração com código Assembly quando necessário.

2.2 Arquitetura do Software

A biblioteca foi estruturada em módulos claramente definidos, cada um responsável por uma área específica de funcionalidade:

- **Core**: Definições básicas e estrutura da biblioteca
- **Target**: Abstrações para arquiteturas de hardware e sistemas operacionais
- **Memory**: Gerenciamento de memória, incluindo stack e alocação
- **File**: Manipulação de formatos de arquivo, com foco em ELF
- **Traits**: Interfaces abstratas definindo comportamentos comuns
- **Types**: Tipos de dados específicos do domínio
- **Result**: Sistema de tratamento de erros

Cada módulo foi desenvolvido com interfaces claras e documentadas, permitindo integração e extensibilidade.

2.3 Implementação de Abstrações de Arquitetura

Para abstrair diferenças entre arquiteturas de hardware, implementamos um sistema baseado em traits:

1. Definição de traits genéricos representando capacidades comuns entre arquiteturas
2. Implementação específica para x86_64 como prova de conceito
3. Utilização de tipos associados e constantes genéricas para representar diferenças entre arquiteturas

Isso permite que o código cliente seja escrito de forma agnóstica à arquitetura subjacente, enquanto mantém a eficiência e especificidade necessárias para programação de sistemas.

2.4 Gerenciamento de Memória

O módulo de memória foi implementado com foco na segurança e eficiência:

1. Manipulação segura da stack, permitindo acesso a argumentos e variáveis de ambiente
2. Sistema de páginas para alocação eficiente de memória
3. Abstrações seguras em torno de ponteiros brutos
4. Implementação de funcionalidades básicas de alocação sem dependência da biblioteca padrão

2.5 Manipulação de Formatos de Arquivo

Para permitir a leitura e interpretação de arquivos executáveis, implementamos suporte ao formato ELF:

1. Parsing de cabeçalhos ELF conforme especificação
2. Abstrações para diferentes versões do formato (32/64 bits)
3. Validação de integridade e consistência
4. Acesso estruturado às seções e segmentos do arquivo

2.6 Metodologia de Teste

Para validar o funcionamento correto da biblioteca:

1. Desenvolvimento de um binário de teste que utiliza as funcionalidades da biblioteca
2. Testes de integração para verificar comportamento em diferentes ambientes
3. Validação manual em ambiente controlado

3 Resultados e Discussão

Os resultados do desenvolvimento da biblioteca *Userspace* demonstram a viabilidade de criar abstrações seguras para programação de sistemas usando Rust. A seguir, apresentamos os principais componentes implementados e discutimos sua relevância e implicações.

3.1 Estrutura da Biblioteca Core

O núcleo da biblioteca foi implementado com sucesso, fornecendo uma base sólida para os demais componentes:

```rust
#![no_std]
#![allow(incomplete_features)]
#![allow(unused_assignments)]
#![feature(generic_const_exprs)]
#![feature(generic_const_items)]

pub struct Origin;

#[macro_use]
pub mod macros;
#[macro_use]
pub mod target;
pub mod file;
pub mod license;
pub mod memory;
pub mod panic;
pub mod result;
pub mod traits;
pub mod types;
pub use result::{Error, Ok, Result};

trait_implement_primitives!();
```

Esta estrutura demonstra uma organização clara e modular, permitindo que usuários importem apenas os componentes necessários para suas aplicações. A ausência de dependência da biblioteca padrão (`#![no_std]`) torna a biblioteca adequada para ambientes sem sistema operacional ou com recursos limitados.

3.2 Abstrações de Arquitetura

A implementação de abstrações de arquitetura demonstrou ser uma abordagem eficaz para lidar com diferenças entre plataformas. O sistema de módulos e traits permite que o código cliente seja escrito de forma portável, enquanto mantém o desempenho otimizado:

```rust
pub mod architecture;
pub mod operating_system;
pub mod result;

pub use architecture as arch;
pub use architecture::Arch;
pub use operating_system as os;
pub use operating_system::Os;

pub use result::{Error, Ok, Result};
```

A arquitetura x86_64 foi implementada como prova de conceito, com suporte para suas características específicas, como manipulação de registradores e convenções de chamada.

3.3 Ponto de Entrada e Manipulação de Stack

Um dos componentes mais importantes da biblioteca é a manipulação segura da stack de programa, permitindo acesso aos argumentos de linha de comando e variáveis de ambiente:

```rust
#[unsafe(no_mangle)]
pub extern "C" fn entry(stack_pointer: crate::target::arch::PointerType)
    -> ! {
```

```
    let stack_pointer = crate::target::arch::Pointer(stack_pointer);

    info!("eXecuting Executable and Linkable Format\n\n");

    let argc = stack_pointer.0 as *const usize;
    info!("argc={:?}\n", unsafe { *argc });
    let stack = userspace::memory::Stack::from_pointer(stack_pointer);
    // stack.print();
    stack.arguments.print();

    let arg0 = stack.arguments.get(0).unwrap();
    let arg0_pointer = arg0.pointer;

    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();
            userspace::info!("\n{:?}\n", self_path);
            let identifier = userspace::file::format::elf::header::
               Identifier::from_path(self_path);
            userspace::info!("{:?}\n", identifier);
        }
    }

    // ...
}
```

Esta implementação permite que programas acessem seus argumentos de linha de comando de forma segura, mesmo sem depender da biblioteca padrão. A abordagem baseada em tipos garante que erros comuns sejam detectados em tempo de compilação.

3.4 Gerenciamento de Memória

O módulo de gerenciamento de memória foi implementado com sucesso, fornecendo:

```
pub mod page;
pub mod stack;
pub use stack::Stack;
pub mod alloc;
pub use alloc::alloc;
```

Este sistema permite alocação e gerenciamento de memória sem depender das facilidades da biblioteca padrão de Rust. As abstrações implementadas garantem segurança de memória enquanto mantêm o controle fino necessário para programação de sistemas.

3.5 Manipulação de Formatos de Arquivo

O suporte ao formato ELF foi implementado permitindo a leitura e interpretação de arquivos executáveis. Isso é crucial para funcionalidades como carregamento dinâmico de código e introspecção de executáveis.

Nossa implementação permite a leitura de cabeçalhos ELF, identificação do tipo de arquivo e extração de metadados importantes, como mostrado no trecho de código do ponto de entrada.

3.6 Desafios e Soluções

Durante o desenvolvimento, enfrentamos vários desafios:

1. **Programação sem biblioteca padrão**: A ausência da biblioteca padrão exigiu re-implementação de funcionalidades básicas. Solucionamos isso com implementações cuidadosas das primitivas necessárias.

2. **Segurança com operações de baixo nível**: Manter a segurança de memória em código de baixo nível foi desafiador. Utilizamos encapsulamento de operações inseguras em APIs seguras para mitigar riscos.

3. **Compatibilidade entre arquiteturas**: Criar abstrações que funcionassem em diferentes arquiteturas exigiu design cuidadoso. O sistema de traits e tipos genéricos permitiu resolver esse desafio.

4. **Tratamento de erros**: Sem exceções ou panics, implementamos um sistema de Result personalizado que preserva informações de erro relevantes.

3.7 Comparação com Trabalhos Relacionados

Comparando com bibliotecas existentes como libc e outras implementações em Rust:

- A biblioteca *Userspace* oferece garantias de segurança de memória superiores às implementações em C
- Em comparação com outras bibliotecas Rust, nosso foco em portabilidade e independência de sistema operacional é distintivo
- A abordagem baseada em traits permite maior extensibilidade que implementações monolíticas

3.8 Limitações Atuais

Reconhecemos algumas limitações na implementação atual:

- Suporte limitado a uma única arquitetura (x86_64)
- Implementação parcial do formato ELF
- Ausência de algumas abstrações de alto nível que facilitariam o uso

Estas limitações serão abordadas em trabalhos futuros.

4 Conclusões

O desenvolvimento da biblioteca *Userspace* demonstra a viabilidade de criar abstrações seguras e eficientes para programação de sistemas utilizando Rust. As principais conclusões deste trabalho são:

1. É possível criar uma biblioteca para programação de sistemas que mantém as garantias de segurança de memória do Rust sem comprometer o desempenho ou a flexibilidade
2. A abordagem baseada em traits e tipos genéricos permite uma extensibilidade significativa, facilitando o suporte a novas arquiteturas e sistemas operacionais
3. Abstrações bem projetadas podem reduzir significativamente a complexidade da programação de baixo nível, mantendo o controle necessário sobre o hardware
4. A independência da biblioteca padrão (`no_std`) permite que a biblioteca seja utilizada em uma ampla variedade de ambientes, desde sistemas embarcados até aplicações de servidor de alto desempenho

Os resultados obtidos têm implicações importantes para o campo de desenvolvimento de sistemas:

- Demonstram a viabilidade de usar linguagens modernas com garantias de segurança para programação tradicionalmente dominada por C
- Fornecem uma base para o desenvolvimento de software de sistema mais seguro e confiável
- Abrem caminho para mais inovação no campo de abstrações de baixo nível

Como trabalhos futuros, pretendemos:

- Estender o suporte para mais arquiteturas, como ARM e RISC-V
- Implementar interfaces completas para chamadas de sistema em diferentes sistemas operacionais
- Desenvolver abstrações para concorrência e paralelismo
- Criar ferramentas para debugging e profiling
- Expandir o suporte para formatos de arquivo e protocolos de rede

A biblioteca *Userspace* representa um passo importante na direção de tornar a programação de sistemas mais segura e acessível, contribuindo para o desenvolvimento de infraestruturas de software mais resilientes e confiáveis.

5 Referências

Referências

[1] The Rust Programming Language Team. *The Rust Programming Language*. Disponível em: https://doc.rust-lang.org/book/. Acesso em: 10 ago. 2023.

[2] The Rust Programming Language Team. *The Embedded Rust Book - A 'no_std' Rust Environment*. Disponível em: https://docs.rust-embedded.org/book/intro/no-std.html. Acesso em: 12 ago. 2023.

[3] Tool Interface Standards Committee. *Executable and Linking Format (ELF) Specification*. Version 1.2, 1995.

[4] The Rust Programming Language Team. *The Rustonomicon*. Disponível em: https://doc.rust-lang.org/nomicon/. Acesso em: 15 ago. 2023.

[5] The Rust Programming Language Team. *Unstable Features*. Disponível em: https://doc.rust-lang.org/unstable-book/. Acesso em: 18 ago. 2023.

[6] The GNU Project. *The GNU C Library (glibc)*. Disponível em: https://www.gnu.org/software/libc/. Acesso em: 20 ago. 2023.

[7] Rich Felker. *musl libc*. Disponível em: https://musl.libc.org/. Acesso em: 22 ago. 2023.

[8] The Rust Embedded Working Group. *The Embedded Rust Book*. Disponível em: https://docs.rust-embedded.org/book/. Acesso em: 25 ago. 2023.

[9] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2023.