2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) | 978-1-6654-2603-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISSREW53611.2021.00075

Rustpi: A Rust-powered Reliable Micro-kernel Operating System

Yuanzhi Liang, Lei Wang, Siran Li, Bo Jiang
School of Computer Science and Engineering, Beihang University,
Beijing 100191, PR China
Email: {liangyz, wanglei, ohmrlsr, jiangbo}@buaa.edu.cn

***Abstract*—Rustpi is a micro-kernel operating system implemented in Rust to explore how modern language features can help to build a reliable operating system. In our system, isolations between micro-kernel servers are achieved by Rust language instead of expensive hardware mechanisms. Moreover, Rust language features such as control-flow integrity and unwinding enable hardware transient fault detection and error recovery without resource leaking. Rustpi creatively integrates these features to enhance its reliability. Moreover, our design is also applicable to other Rust micro-kernel systems or even the Linux kernel.**

***Index Terms*—Operating Systems; reliability; fault tolerance**

I. Introduction

In early days, micro-kernel operating systems made use of hardware isolation mechanisms to achieve fault isolation and recovery. Minix 3[1] is a well-known micro-kernel OS at that time. However, traditional operating system designs cannot meet the increasing demands on reliability and safety. Moreover, the heavy use of unsafe C language worsens the situation. Recently, OS researchers started to build OS with safe programming languages. For example, Biscuit[2] is an operating system written in Go language. Rust is another popular language to build modern kernels. Even the Linux community started to accept Rust as its secondary language[3]. While many operating systems written in Rust emerge, they still fall short in terms of system reliability. Redox is a micro-kernel OS in Rust, but it lacks fault recovery mechanisms. Theseus[4] is an excellent uni-kernel operating system featuring fault recovery. However, as a uni-kernel OS, it cannot disentangle a user program from the kernel module, which makes a malicious user task endanger the whole system.

Therefore, in this paper, we present Rustpi, a micro-kernel operating system written in Rust, which utilizes language features of Rust to enhance the reliability of the whole micro-kernel operating system.

II. System Design

A. Overall Design

As shown in Figure 1, the architecture of Rustpi consists of three parts: kernel, user-space servers, and user-space programs. The kernel of Rustpi provides fundamental primitives such as memory management, address space switching, and thread scheduling. Different from other micro-kernel design, the user-space servers of Rustpi runs in the same address space (called Trusted Address Space, TAS). When an inter-server

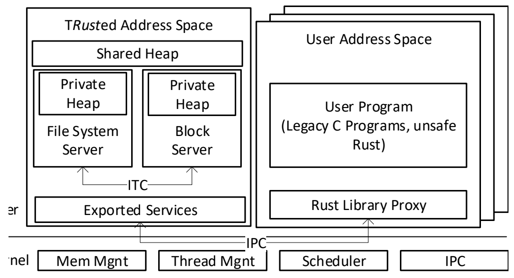

Fig. 1. Overall design of Rustpi

IPC happens, no address space switching or memory copying is needed. Both kernel and user-space servers are written in Rust. Other user-space programs use services exported by the servers in TAS. A C library that provides POSIX file operation APIs is also included.

B. Fault Prevention

Rust is a modern programming language that leverages the ownership model. Typical memory corruption faults and concurrency bugs are prevented by the Rust compiler.

Micro-kernel helps fault prevention from another perspective. In our design, the address spaces of servers or user processes are isolated. As a result, when a memory access fault happens, the micro-kernel can assure that the fault is confined within the address space of the process per se.

Furthermore, Rustpi implements an isolated memory heap for the kernel to enhance overall memory safety and prevent dead-lock error due to heap memory allocation. Traditionally, Rust programs use a global memory allocator to handle all kinds of heap memory allocation. However, if one module of the program corrupts the heap memory, the data of other modules can also be corrupted.

Therefore, Rustpi makes use of the `allocator-api` feature from nightly Rust `std` library and can distinguish allocation requests from different modules. As a result, critical kernel modules no longer use the same heap and the faults of non-critical modules cannot affect critical system modules.

III. Fault Tolerance

While the memory isolation mechanism can stop fault propagation across modules, faults may still occur within a software module or due to hardware environments. Thus, fault

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 27,2022 at 14:44:26 UTC from IEEE Xplore. Restrictions apply.

tolerance is also needed to achieve high reliability, which includes error detection, error recovery, and fault treatment.

A. Error Detection

Both hardware and software can help detect errors. For example, when the running program dereferences a null pointer, a CPU synchronous exception will occur. Rust also generates enormous implicit run-time panics. For instance, for every array indexing, the Rust compiler will generate index range checking that jumps to panic when the check fails.

Detection methods above assume that the underlying hardware environment is working correctly and perfectly. However, under real circumstances, transient hardware faults are inevitable. When such a fault happens, both the control flow and data flow of the running systems can corrupt. Therefore, Rustpi utilizes the control-flow integrity (CFI)[5] technique to detect unexpected function jumps induced by transient hardware error that corrupts return addresses.

When the errors above are detected, the kernel of Rustpi will start an error recovery routine.

B. Error Recovery

Error recovery is to restore the corrupted system to a normal state. There are two types of error recovery, forward and backward. Forward error recovery (FER) is realized by the `Result<T,E>` type in Rust. Operations such as system call return a `Result<T,E>` rather than a value of `T` type. Result is an `enum`, which represents either a successful result with a value of `T` type (`Ok(T)`) or an error of `E` type (`Err(E)`). All system call interfaces of Rustpi return a `Result` type. And the result is eventually encoded in general-purpose registers.

Backward error recovery (BER) is more difficult under the semantics of operating system kernels. Traditional operating system kernels such as Linux use C language without built-in support for BER. However, Rust provides a `panic`/`unwind` mechanism to support BER. Rust compiler generates exception handling routine called landing pads for each of the panics explicitly or implicitly introduced. When the program panics, those routines will take place to do clean-up jobs for every function call in the call stack.

The unwinding of Rust is only available for programs that use `std` library. As a kernel, Rustpi has to implement its unwinding mechanism and provides a similar interface `catch_unwind(|A|R)->Result<R,E>`. By wrapping codes into a closure, any panic happening in the closure can be caught and provided to the caller to handle appropriately. In another word, unwinding converts a BER into a FER.

Rustpi ensures that all routines trapped into the kernel have their subroutines wrapped in a `catch_unwind` closure. Therefore, no panic is a "panic" for the kernel.

C. Fault Treatment

Having the control flow unwinded doesn't mean fault recovery is done. For example, a system call can be unwinded and return an error to user space. However, a system call often consists of multiple steps in which resource allocation may happen. If a step in between throws a panic, all those resources allocated previously need to be taken care of. Luckily, the unwinding of Rust also helps `drop` (release) local variables in the unwinding stack frame. Rustpi implements the `Drop` trait for all kernel resource structs. For example, we allocate two types of resources in a system call. (As shown in Listing 1) If a panic happens between two allocations, the control flow will end early, and `a` will be automatically `drop` (released) in the unwinding process. Therefore, no kernel resource will leak during unwinding. Such exception handling is much more difficult in a C language kernel.

Listing 1. Catch/unwind example

```
let (a, b) = catch_unwind(|| {
    let a = ResourceA::allocate();
    // do something cause panic
    let b = ResrouceB::allocate();
    (a, b)
})?;
```

Micro-kernel servers in Rustpi also make use of this mechanism. When the server receives a request from a client, the server will process the request in a `catch_unwind` closure. When a processing request failed, the server can retry processing the request until it executes successfully, or a persistent error is detected.

In a word, with the help of the `catch_unwind` mechanism of Rust, both systems call interfaces and micro-kernel services can provide reliable service.

IV. Conclusion

In this work, leveraging the language features of Rust, Rustpi provides a set of designs to build a reliable micro-kernel OS. It uses an isolated heap to prevent memory corruption across critical modules. Rustpi also takes advantage of the CFI technique to detect transient hardware faults and have them handled by the recovery process. Furthermore, the drop and unwind feature of Rust are adopted by Rustpi to keep both system call interfaces and server service interfaces reliable. In general, Rustpi demonstrates that the features of Rust help build up a reliable micro-kernel operating system, and its design is also applicable to other micro-kernel operating systems or even the Linux kernel.

References

[1] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Minix 3: A highly reliable, self-repairing operating system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 3, pp. 80–89, 2006.

[2] 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)*, 2018, pp. 89–105.

[3] "Supporting Linux kernel development in Rust [LWN.net]." [Online]. Available: https://lwn.net/Articles/829858/

[4] K. Boos, N. Liyanage, R. Ijaz, and L. Zhong, "Theseus: an experiment in operating system structure and state management," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 1–19.

[5] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 27,2022 at 14:44:26 UTC from IEEE Xplore. Restrictions apply.