

**工程硕士学位论文**

全日制

**面向Rust操作系统的多进程调试方法与系统实现**

学科专业 ：计算机技术

研究方向 ：操作系统

作者姓名 ：

学 号 ：

指导教师 ：

所在学院 ：

二〇二四年四月

Multi-process debugging method and system implementation for Rust operating system

Dissertation Submitted to

Beijing Technology and Business University

in Partial Fulfillment of the Requirement

for Professional Master Degree

of Engineering

by

（Computer Technology）

Dissertation Supervisor：

April 2024

学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。除了文中特别加以标注引用和致谢的地方外，论文中不包含其他个人或集体已经发表或撰写过的研究成果，也不包含为获得 **北京工商大学** 或其他教育机构的学位或证书而使用过的材料。对本文的研究做出重要贡献的个人和集体，均已在文中以明确的方式标明并表示谢意。本声明的法律后果完全由本人承担。

学位论文作者签名： 签字日期： 年 月 日

学位论文版权使用授权书

本人完全了解 北京工商大学 有关保留和使用学位论文的规定，即：研究生在校攻读学位期间学位论文所涉及的知识产权属于北京工商大学。学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许学位论文被查阅和借阅；学校可以公布学位论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存、汇编学位论文。（保密的学位论文在解密后适用本授权书）

学位论文作者签名： 导师签名：

签字日期： 年 月 日 签字日期： 年 月 日

学位论文作者毕业后去向：

工作单位： 电话：

通讯地址： 邮编：

**摘 要**

近年来开源指令集RISC-V以及基于Rust编程语言的开源操作系统（简称RustOS）在全世界范围内获得了广泛的关注。在我国大力发展自主中央处理器（CPU）与自主操作系统的大背景下，RISC-V与RustOS在学术界与产业界有大量的研究与开发工作，许多基于RISC-V指令集定制的RustOS被开发出来满足不同的需求。由于RISC-V与RustOS处于技术发展的初期阶段，研究人员或工程师在开发或调试相关代码时缺少一款适配的源代码级调试工具，使得开发过程艰难繁琐。

在此背景下，本文提出了面向Rust操作系统的多进程调试方法，实现了能够支持Rust语言的源代码级操作系统调试工具。具体而言，操作系统中多进程的切换包括用户进程之间的切换，用户进程和内核进程的切换以及内核进程之间的切换。在操作系统中内核态程序和用户态程序的符号表是分开的，如果程序运行中进行用户态和内核态的转换，符号表要随之切换，调试信息也会随之消失，本研究解决了不同特权级切换时导致的调试信息丢失的问题。对于不同用户进程来说，每个用户进程都有自己的符号表，每次进行用户进程切换的时候都需要通过系统调用进入内核态，更新内存地址空间，然后返回用户态执行新的进程，因此调试器需要获取下一个用户进程的名称，才能进行用户进程的切换，本研究解决了不同用户进程之间切换时的进程标识符获取问题。

为了提高调试器的实用性，在支持GDB断点调试的基础上，引入了eBPF和kprobe等动态调试技术，提出了一种静态断点调试和动态跟踪结合的调试方法。本文设计了eBPF处理模块，实现了运行在操作系统中的eBPF处理模块与GDB的连接与交互，eBPF处理模块具有函数插桩等动态跟踪功能，可以在运行时对内核进行监控，捕获有关函数执行和数据流动等更多的信息。该方法能够使用GDB的静态断点调试，支持单步调试，也能够使用eBPF对运行中的操作系统进行监测，可以对操作系统中运行的多个进程进行调试和动态监测。

最后，本文基于VSCode调试架构，开发能够支持Rust语言的源代码级操作系统的调试器，在调试器中实现了上述两种方法。经过对调试器的测试和使用，该调试器能够在调试操作系统代码漏洞上体现出非常大的作用，并在最后案例研究中，展示使用调试器调试操作系统漏洞的整个过程，通过调试器逐渐缩小代码漏洞的范围，在不需要改动其他源代码的情况下，最终找到并修复了代码漏洞。

关键词：Rust操作系统，操作系统调试器，多进程调试，GDB，eBPF

**ABSTRACT**

In recent years, the open source instruction set RISC-V and the open source operating system based on the Rust programming language (RustOS for short) have gained widespread attention around the world. In the context of my country's vigorous development of independent central processing units (CPUs) and independent operating systems, RISC-V and RustOS have undergone a large amount of research and development work in academia and industry. Many RustOS customized based on the RISC-V instruction set have been Developed to meet different needs. Since RISC-V and RustOS are in the early stages of technological development, researchers or engineers lack an adapted source code-level debugging tool when developing or debugging related codes, making the development process difficult and cumbersome.

Against this background, this article proposes a multi-process debugging method for the Rust operating system and implements a source code-level operating system debugging tool that can support the Rust language. Specifically, the switching of multiple processes in the operating system includes switching between user processes, switching between user processes and kernel processes, and switching between kernel processes. In the operating system, the symbol tables of kernel mode programs and user mode programs are separated. If the user mode and kernel mode are converted during the running of the program, the symbol table will also switch accordingly. After the symbol table is switched, the debugging information will also follow. Disappear, this research solves the problem of debugging information loss caused by switching between different privilege levels. For different user processes, each user process has its own symbol table. Every time a user process is switched, it needs to enter the kernel state through a system call, update the memory address space, and then return to the user state to execute the new process. Therefore The debugger needs to obtain the name of the next user process before switching user processes. This study solves the problem of obtaining process identifiers when switching between different user processes.

In order to improve the practicality of the debugger, on the basis of supporting GDB breakpoint debugging, dynamic debugging technologies such as eBPF and kprobe are introduced, and a debugging method that combines static breakpoint debugging and dynamic tracing is proposed. This article designed the eBPF processing module to realize the connection and interaction between the eBPF processing module running in the operating system and GDB. The eBPF processing module has dynamic tracking functions such as function instrumentation, and can monitor the kernel at runtime and capture relevant function executions. and data flows and more. This method can use GDB's static breakpoint debugging, support single-step debugging, and can also use eBPF to monitor the running operating system, and can debug and dynamically monitor multiple processes running in the operating system.

Finally, based on the VSCode debugging architecture, this article develops a debugger that can support the source code level operating system of the Rust language, and implements the above two methods in the debugger. After testing and using the debugger, the debugger can play a very important role in debugging operating system code vulnerabilities. In the final case study, the entire process of using the debugger to debug operating system vulnerabilities is demonstrated. Narrow down the scope of the code vulnerability and finally find and fix it without changing other source code.

**Keywords**: Rust operating system,operating system debugger,multi-process debugging,GDB,eBPF

目录

[第1章 绪论 1](#_Toc162940130)

[1.1 选题的背景及意义 1](#_Toc162940131)

[1.2 研究内容和主要工作 2](#_Toc162940132)

[1.3 论文组织结构 3](#_Toc162940133)

[第2章 相关工作及技术 5](#_Toc162940134)

[2.1 Rust语言和RISC-V指令集 5](#_Toc162940135)

[2.1.1 Rust语言 5](#_Toc162940136)

[2.1.2 RISC-V指令集 5](#_Toc162940137)

[2.2 操作系统调试技术及调试工具 7](#_Toc162940138)

[2.2.1 GUN调试器和QEMU虚拟机 7](#_Toc162940139)

[2.2.2 eBPF和kprobe技术 9](#_Toc162940140)

[2.2.3 Visual Studio Code 中的调试架构 10](#_Toc162940141)

[2.3 国内外研究现状 11](#_Toc162940142)

[2.3.1 操作系统调试 11](#_Toc162940143)

[2.3.2 静态断点调试 12](#_Toc162940144)

[2.3.3 动态跟踪技术 14](#_Toc162940145)

[2.4 Rust操作系统（rCore-Tutorial-v3）及调试系统架构 15](#_Toc162940146)

[第3章 面向Rust操作系统的多进程调试方法 16](#_Toc162940147)

[3.1 方法概述 16](#_Toc162940148)

[3.2 基于GDB的多进程调试方法 17](#_Toc162940149)

[3.2.1 概述 17](#_Toc162940150)

[3.2.2 断点组管理模块的设计 18](#_Toc162940151)

[3.2.3 调试器中内核态和用户态的切换 19](#_Toc162940152)

[3.2.4 多个用户进程的切换和调试 22](#_Toc162940153)

[3.2.5 小结 26](#_Toc162940154)

[3.3 一种静态断点调试和动态跟踪结合的方法 26](#_Toc162940155)

[3.3.1 概述 26](#_Toc162940156)

[3.3.2 动态跟踪调试的工作机理 27](#_Toc162940157)

[3.3.3 eBPF移植 28](#_Toc162940158)

[3.3.4 eBPF与GDB调试信息的整合 29](#_Toc162940159)

[3.3.5 小结 37](#_Toc162940160)

[3.4 本章小结 37](#_Toc162940161)

[第4章 面向Rust操作系统的调试工具设计与实现 39](#_Toc162940162)

[4.1 整体框架设计 39](#_Toc162940163)

[4.2 服务器部分 40](#_Toc162940164)

[4.2.1 远程连接 40](#_Toc162940165)

[4.2.2 编译和加载 40](#_Toc162940166)

[4.2.3 调试适配器进程 41](#_Toc162940167)

[4.3 网页端部分 42](#_Toc162940168)

[4.4 案例研究 44](#_Toc162940169)

[4.4.1 Bug描述 44](#_Toc162940170)

[4.4.2 代码简述 45](#_Toc162940171)

[4.4.3 调试过程 46](#_Toc162940172)

[4.5 本章小结 48](#_Toc162940173)

[第5章 总结与展望 49](#_Toc162940174)

[5.1 总结 49](#_Toc162940175)

[5.2 展望 50](#_Toc162940176)

[参考文献 51](#_Toc162940177)

# 绪论

## 选题的背景及意义

操作系统（OS）是驱动硬件运行的核心系统，其在硬件和用户之间起到了桥梁的作用。操作系统的性能、可靠性和安全性直接影响到计算机系统的整体表现，以及广泛应用于各行各业的软件和服务的质量。因此，操作系统的开发和维护变得至关重要。在操作系统的开发过程中，调试器是一个不可或缺的工具，它对于保证操作系统的稳定性、提高开发效率、优化系统性能以及支持新技术的应用具有重要意义。

近年来开源指令集RISC-V以及基于Rust编程语言的开源操作系统[1]（简称RustOS）在全世界范围内获得了广泛的关注。在我国大力发展自主中央处理器（CPU）与自主操作系统的大背景下，RISC-V与RustOS在学术界与产业界有大量的研究与开发工作，许多基于RISC-V指令集定制的RustOS被开发出来满足不同的需求。由于RISC-V与RustOS处于技术发展的初期阶段，研究人员或工程师在开发或调试相关代码时缺少一款适配的源代码级调试工具，使得开发过程艰难繁琐。

源代码级调试工具可以帮助开发人员发现和修复操作系统中的各种错误和问题，能够让使用者控制被调试软件的运行，并提供一系列工具实时查看被调试软件运行过程中的中间状态，这样开发人员可以快速定位和分析问题的根源，从而提高开发效率，并确保操作系统的可靠性和稳定性。调试器还可以用于系统性能分析和优化。通过调试器，开发人员可以深入了解操作系统的运行情况，包括各种系统调用的频率、资源的使用情况等，从而发现系统瓶颈并进行优化，提升系统的性能和响应速度。此外，随着技术的不断发展，新的硬件和软件技术不断涌现，操作系统调试器也需要不断更新和优化，以支持新技术的应用和开发。调试工具的不断改进和更新将为操作系统的研发和维护提供强大的支持和保障。

不同于通用应用程序代码开发与调试，操作系统源代码运行状态多、工作逻辑复杂，还有系统资源限制、复杂的数据结构和算法、调试信息的获取和解析以及性能和效率等方面的挑战，进一步增加了调试难度。此外，操作系统调试还面临着Rust编程语言的独特挑战。虽然Rust作为一门现代的系统编程语言在可靠性和内存安全性方面表现出色[2]，但其调试信息可能不如传统C/C++语言那么丰富。这使得在Rust代码中查看变量、数据结构和堆栈跟踪变得更加具有挑战性。

总之，目前的操作系统调试器对操作系统源代码的调试支持不够完善，无法支持用户态和内核态同时调试，和跨用户态和内核态的多进程调试，除此以外，由于Rust语言的调试信息不够完善，导致使用Rust语言编写的操作系统面临更大的困难和挑战。本课题旨在提供一种全新的源代码级调试方法，旨在解决传统源代码级调试器无法跨越内核态和用户态进行调试的问题，以及GDB在Rust语言中的限制，使调试器支持跨越内核态和用户态的操作系统调试，同时结合了静态断点调试和动态跟踪技术，以满足复杂操作系统内核和用户态多进程调试的需求。通过本研究，本文寻求解决传统调试器无法有效应对的内核态和用户态切换问题，提高Rust编程语言的调试支持，为复杂操作系统的开发者提供一款更强大的调试工具。

## 研究内容和主要工作

本研究旨在弥补目前源代码级操作系统调试器的不足，以提高源代码级操作系统调试器的调试能力，提高调试器的适用性和易用性。具体而言，本研究将分析目前源代码级调试器对Rust操作系统进行调试过程中遇到的困难与挑战，并提供了一系列的解决困难的调试方法，基于调试方法开发了面向Rust操作系统的源代码级调试工具。本研究将从以下几个方面进行问题分析并提出问题的解决方法：

（1）一种基于GDB的多进程调试方法。针对操作系统运行状态多、用户态和内核态进程切换频繁，工作逻辑复杂等问题，本课题提出了一种于GDB的多进程调试方法。在该方法中，设置了断点组管理模块，缓存用户设置的不在当前进程地址空间的断点，并在被调试操作系统运行到对应的地址空间的时候再激活断点，实现断点组的切换，解决了跨特权级的源代码断点设置冲突的问题；其次设置了边界检测点，使调试器能够识别被调试操作系统当前在哪个特权级中运行，并根据特权级切换不同的符号表，从而能够让使用者进行跨特权级的调试；最后为了支持多个用户进程的调试，在内核代码中获取下一个要运行进程的标识符，并告知调试器，调试器根据进程标识符来确定下一个要切换的符号表，获取到新进程的调试信息，对新进程进行调试，也能够支持多个用户进程的符号表切换，从而实现了对多个用户进程的调试。

（2）一种静态断点和动态跟踪结合的方法。针对Rust语言的特殊性，调试信息不完善，GDB静断点态调试会暂停被调试操作系统运行，无法动态监测被调试操作系统运行状态等问题，本课题提出了一种静态断点和动态跟踪结合的方法。在该方法中，对当前调试操作系统进行了eBPF移植的移植工作，使操作系统能够支持eBPF程序的运行，设计了eBPF处理模块，它运行在被调试操作系统中，可以对运行中的操作系统进行监控，并获取调试信息，并把获取到的信息通过串口和GDB进行交互，最终展示给用户。该方法能够使用GDB的静态断点调试，支持单步调试，同时也能够使用eBPF对运行中的操作系统进行监测，可以对操作系统中运行的多个进程进行调试和动态监测。

（3）Rust操作系统调试器的系统设计与实现。在目前的调试领域中，并没有一个调试器能够很好地支持Rust操作系统的源代码级调试，因此，本文结合了上述两种方法，开发了一款能够支持Rust语言编写的操作系统调试器，为使用者提供方便的，全面的调试工具。为了让使用者能够加方便地进行调试，本文选择使用调试者和被调试内核分离的设计来实现 QEMU 虚拟机的操作系统远程调试。内核在服务器上运行，用户在浏览器使用VSCode插件发送调试相关的请求。其次，分别介绍了服务器端和用户使用的网页端的设计与实现，最后，介绍了一个调试案例，使用GDB静态断点调试和eBPF动态跟踪调试的功能解决了遇到的程序漏洞。

## 论文组织结构

论文的主要组织结构如图1.1所示。第一章主要介绍了面向Rust操作系统的多进程调试方法与系统实现的研究背景及意义，首先，针对Rust操作系统的发展现状，分析了开发Rust操作系统调试器的重要性，然后，阐述了操作系统调试的复杂性，以及Rust操作系统面对的困难及挑战。接着，阐述了论文的研究内容和主要工作，包括一种基于GDB的多进程调试方法，一种静态断点和动态跟踪结合的方法和Rust操作系统调试器的系统实现。最后，阐述了论文的组织结构，为后续章节提供整体框架。

第二章主要介绍相关工作及技术，主要包括以下内容，首先介绍了Rust语言和RISC-V指令集，及使用Rust语言编写操作系统的优势及RISC-V指令集的现状及特色；其次，介绍了操作系统调试技术及工具，如GUN调试器，eBPF和kprobe等调试技术，VSCode调试器的调试架构；然后介绍了国内外研究现状，主要介绍操作系统调试，静态断点的调试和动态跟踪调试等方面；最后介绍了Rust操作系统和调试器的整体框架，本课题实验中使用的被调试操作系统主要使用的是rCore-Tutorial-v3。

第三章阐述了面向Rust操作系统的多进程调试方法的研究过程，在该章节中，首先对面向Rust操作系统的多进程调试方法进行了概述，简述了目前Rust操作系统调试过程中遇到的困难与挑战及其解决办法，接下来主要阐述了两种调试方法：

1. 一种基于GDB的多进程调试方法，在该小节中首先对操作系统进行多进程调试过程中的问题及难点进行了分析和介绍，接下来，对多进程调试方法的实现进行了展开叙述，分别介绍了断点组管理模块的设计和实现，调试过程中内核态和用户态切换的过程及原理，还有调试器进行多进程切换和调试的过程及实现。
2. 一种静态断点和动态跟踪结合的方法，来对进程进行调试，在该小节中，对实现该方法过程中的问题及难点进行了详细介绍，接下来主要介绍了动态跟踪调试的工作原理，然后，主要对eBPF处理模块的原理和实现进行了介绍。

最后对面向Rust操作系统的多进程调试方法进行了总结。

第四章介绍了Rust操作系统调试器的系统设计与实现，首先介绍了调试器的整体架构，分别介绍了服务器端和用户使用的网页端的设计与实现，最后，介绍了一个调试案例，使用GDB静态断点调试和eBPF动态跟踪调试的功能解决了遇到的程序漏洞。

第五章为总结与展望，对本文提出的两种Rust操作系统调试方法及源代码级操作系统调试工具进行了总结，并对该研究方向上的后续发展方向进行展望。



图 1.1 论文的组织结构

# 相关工作及技术

## Rust语言和RISC-V指令集

### Rust语言

Rust 是一种相对较新的编程语言，自 2015 年 v1.0 发布以来获得了巨大的关注[3]。Rust已经连续七年在Stack Overflow开发者调查的“最受喜爱编程语言”评选项目中折取桂冠，因此未来将有更多的人关注并使用这一编程语言。Rust是为高度安全和并发的系统而设计的，它提供了与C++类似的性能[3]。Rust旨在为安全系统编程提供安全保障和运行时效率，并越来越多地用于构建操作系统内核、web浏览器、数据库和区块链等软件基础设施[4]。

Rust是一种很有前途的系统级编程语言，它可以使用其强类型系统和基于所有权的内存管理方案来防止内存损坏错误[5,6]。Rust旨在不牺牲性能的情况下，实现内存安全和对内存的精细化控制，支持函数式和命令式以及泛型等编程范式的多范式语言。它首要的设计理念主要有以下三个：无垃圾回收的安全内存管理[3]，零成本抽象和支持高并发。因此，使用Rust语言编写操作系统是一个很好的选择。

使用Rust编写操作系统的好处在于：Rust 丰富的类型系统和所有权模型[7]保证了内存安全和线程安全，某种程度上来说Rust完全是内存安全，可以在编译期就能够消除各种各样的错误。Chen, Shao Fu [8]等人使用Rust语言开发了内核模块，利用Rust语言的安全功能来防止在编写内核代码时引入内存错误或同步错误。Liang [9]等人开发了Rustpi，它是一个在Rust中实现的微内核操作系统，探索了使用Rust语言帮助构建可靠的操作系统。

Rust 还拥有出色的文档、友好的编译器和清晰的错误提示信息，集成了一流的工具 —— 包管理器和构建工具。Rust在内存安全和线程安全方面比C/C++语言更加强大，它没有内置的垃圾回收机制，而是从语言的内在机制上去解决C和C++内存安全和线程安全的痛点。

### RISC-V指令集

RISC-V是一种开源、免版税的指令集架构（ISA），开启了处理器创新的新时代。RISC-V具有模块化和可扩展性的特点，明确支持特定领域的自定义扩展。RISC-V是目前流行的嵌入式处理器ISA[10]。RISC-V支持多个平台，同时保持简单性和可靠性，很多流行的工具链和操作系统已经支持RISC-V，对RISC-V的软件支持在过去几年中一直在增加[11]。

RISC-V具有四个显著特点，分别是简单和一致性、学校和产业能够无缝对接、开放的可扩展性以及高编程效率，这些特点正好可以解决嵌入式开发中的许多痛点。因此，越来越多的企业投入到了RISC-V生态建设中，基于RISC-V指令架构开发的芯片和操作系统也日益增多，各大高校也开始了对RISC-V的教学和研发。

RISC-V的指令集架构完全开源，使得任何人都可以查看、使用和修改。与一些专利受限的商业架构不同，RISC-V的免费使用权使其在学术和开发社区中备受欢迎。在教育领域，RISC-V被广泛应用于计算机体系结构和操作系统课程。在嵌入式系统领域，它适用于物联网设备、传感器和各种嵌入式控制系统。一些公司已经将RISC-V引入数据中心和服务器级应用，以追求更大的灵活性和自定义性。

在这个不断蓬勃发展的生态系统中，研究人员和工程师们不断探索RISC-V的潜力，并开发出越来越多的应用。Anders [12]等人对RISC-V处理器的实际部署进行了深入探讨，讨论了最先进的功能和系统级测试解决方案在RISC-V处理器中的应用，以及RISC-V处理器在系统安全方面的作用。Kalapothas [13]等人的研究对RISC-V SoC进行了定量分类，并揭示了未来利用RISC-V开源硬件架构进行机器学习研究的机会。Wali [14]等人评估了操作系统对基于RISC-V的SoC的可靠性，以及配置内存混乱的影响。在未来，RISC-V指令集将在更多的方面和领域得到应用，为新一代的计算机系统和应用提供更大的灵活性和可扩展性。

本课题调试的Rust操作系统即运行在RISC-V上。在实现了 M 模式（机器态）和 U 模式（用户态）的RISC-V处理器上存在一个叫做物理内存保护(PMP，Physical Memory Protection)的功能，允许 M 模式指定 U 模式可以访问的内存地址。PMP 包括几个地址寄存器（通常为 8 到 16 个)和相应的配置寄存器。这些配置寄存器可以授予或拒绝读、写和执行权限。当处于 U 模式的处理器尝试取指或执行 load 或 store 操作时，将地址和所有的 PMP 地址寄存器比较。如果地址大于等于 PMP 地址 i，但小于 PMP 地址 i+1，则 PMP i+1 的配置寄存器决定该访问是否可以继续，如果不能将会引发访问异常。

图2.1显示了 PMP 地址寄存器和配置寄存器的布局。这两者都是 CSR（状态控制寄存器），地址寄存器名为 pmpaddr0 到 pmpaddrN，其中 N+1 是实现的 PMP 个数。地址寄存器右移两位，如果物理地址位宽小于 XLEN-2，则高位为 0。 R、W 和 X 域分别对应读、写和执行权限。A 域设置是否启用此 PMP，L 域锁定了 PMP 和对应的地址寄存器。地址寄存器是右移两位的，这是因为 PMP 以四字节为单位。配置寄存器通过密集地填充在 CSR 中以加速上下文切换，如图2.2所示。对于 RV32（上半部分），16个配置寄存器被分配到 4 个 CSR 中。对于 RV64（下半部分），它们则分配到了两个偶数编号的 CSR 中。PMP 的配置由 R、W 和 X 位组成，它们分别对于 load，store 和 fetch 操作，还有另一个域 A，当它为 0 时禁用此 PMP，当它为 1 时启用。此外，PMP 配置还支持其他的模式，还有加锁等更多功能。

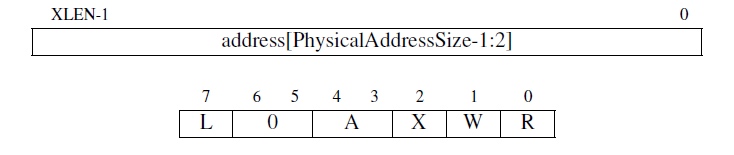


图 2.1 PMP 地址和配置寄存器

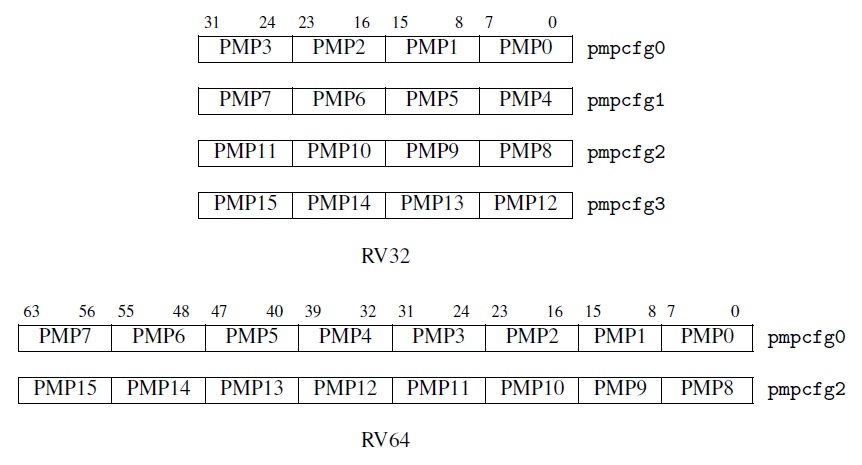


图 2.2 pmpcfg CSR 中 PMP配置的布局

## 操作系统调试技术及调试工具

### GUN调试器和QEMU虚拟机

在软件领域，GNU作为广为人知的开源软件组织，旗下的软件产品丰富、功能完备。基于GNU的调试器（GDB）支持对被调试程序进行断点执行的调试方法。GDB可以通过串口或TCP/IP与实现了RSP协议的运行在远程目标上的GDB服务器进行通信，例如QEMU中的GDB服务器。在GDB中，用户可以在操作系统的用户态随意设置断点，而在内核态设置断点时需要另外一些设置。一旦用户在代码的某处设置断点，便能够观察程序运行到当前代码时的本地变量信息、堆栈信息等。

随着调试器的不断发展，许多人在GDB调试器的基础上进行了扩展和改进。例如，Ji [15]等人基于GNU调试器(GDB)，设计并实现了一个用于新型32位微处理器的可重定向软件调试器。此外，Chatterje [16]等人提出了一个PGDB原型，利用GDB的自动断点增强了对数据争用和死锁检测的支持，用于在PGDB中检测多线程程序执行期间的数据争用和死锁。

系统模拟器通过模拟处理器、内存、外设等硬件资源创建一个完整的虚拟计算机环境，支持运行和调试不同架构的软件，可大大缩短跨架构的软件开发周期[17]。QEMU是一种开源的虚拟机监视器和模拟器，它广泛应用于虚拟化、嵌入式系统开发和仿真等领域。QEMU支持各种架构的仿真，包括x86、ARM、PowerPC、RISC-V等。王涛[17]等人提出了一种基于QEMU的指令追踪技术，该技术面向RISC-V架构，实现了多种场景下的指令序列离线分析，包括指令分类统计、程序热点标记、行为模式分析等。通过gdbstub特性，QEMU提供了GDB服务器，使得GDB可以通过远程连接控制QEMU中运行的操作系统[18]，从而允许用户以与在真实硬件上使用JTAG等低级调试工具类似的方式调试客户代码。

Mihajlović [19]等人对QEMU仿真器的ARM ISA仿真进行了改进，允许连续生成指令级跟踪。使用标准的GDB客户端，可以插入跟踪点以动态记录寄存器和内存地址，而无需更改正在执行的代码。随着虚拟化技术的长期发展，虚拟机的性能已经可以满足很多应用需求，特别是处理器和内存的虚拟化，已经达到了与物理主机相近的性能[20]。QEMU虚拟机提供了非常方便的仿真环境，越来越多的开发人员使用该环境进行设计和开发，Zhang [21]等人提出了一种基于QEMU虚拟机架构的嵌入式可信计算环境的构建方案。另外，Díaz [22]等人的工作分析了当前的软件仿真器，并提出了一种允许QEMU在并行模式下进行外部同步的新方法，构建了硬件/软件协同仿真虚拟平台。他们为QEMU提供了外部同步机制，QEMU是最兼容的开源软件仿真器，同步机制修改了 QEMU 对多线程转换器（并行模式）的原始建议，用于提高协同仿真性能。这些修改支持使用并行化 QEMU 在硬件/软件协同仿真虚拟平台中仿真多核嵌入式设备。

随着技术的进步和应用场景的不断拓展，GDB和QEMU在软件开发、系统调试以及教学领域的作用愈发凸显。未来，随着开源社区和学术界对这些工具的不断改进和优化，本文可以期待更多更强大的功能被引入，以满足日益复杂的软件开发和系统调试需求。因此，持续关注和投入到GDB和QEMU的研究和发展中，将有助于推动软件开发和系统调试技术的进步，为计算机科学领域的发展贡献力量。

### eBPF和kprobe技术

扩展伯克利包过滤器（extended Berkeley Packet Filter，简称 eBPF）是一个允许在内核里安全地执行不受信任的用户定义插件的子系统。它允许用户空间应用程序在 Linux 内核内部执行代码，而无需修改内核代码或插入内核模块[23]。

eBPF程序可以满足各种复杂的监控需求，图2.3展示了将eBPF 用于操作系统跟踪的的一个典型的工作流程。用户程序提供eBPF字节码，通过系统调用加载进内核。这个字节码程序经过verifier验证后交付给eBPF模块执行。eBPF程序可以调用操作系统支持的kprobe等内核监测模块，在内核空间中动态地收集各种类型的信息，并将收集到的数据存储在 eBPF maps 中。



图 2.3 eBPF工作流程

eBPF是 Linux 内核中的指令集和执行环境。eBPF 提高了数据处理的灵活性，通过具有实时 （JIT） 编译器和在内核中运行的解释器的虚拟机实现。它执行用户提供的自定义 eBPF 程序，有效地将内核功能移动到用户空间中。eBPF 已被 Facebook、Netflix 和学术界等公司广泛采用，适用于广泛的应用领域[24]。eBPF允许直接在 Linux 内核中动态加载和运行微程序，而无需重新编译。Miano [25]等人在工作中研究如何在 eBPF 中开发高性能网络测量，他们将草图作为案例研究，因为它们能够支持广泛的任务，同时提供低内存占用和准确性保证。Caviglione [26]等人利用eBPF，以非常有效的编程方式跟踪和监视软件进程的行为。

开发人员在内核或者模块的调试过程中，往往会需要要知道其中的一些函数有无被调用、何时被调用、执行是否正确以及函数的入参和返回值是什么等等。比较简单的做法是在内核代码对应的函数中添加日志打印信息，但这种方式往往需要重新编译内核或模块，重新启动设备之类的，操作较为复杂甚至可能会破坏原有的代码执行过程。kprobes 是一种用于注册断点和相应处理程序的机制[27]。在内核中启用kprobes支持后，用户可以调试任意内核地址的任意指令。利用kprobes技术，用户可以定义自己的回调函数，然后在内核或者模块中几乎所有的函数中（有些函数是不可探测的，例如kprobes自身的相关实现函数，后文会有详细说明）动态的插入探测点，当内核执行流程执行到指定的探测函数时，会调用该回调函数，用户即可收集所需的信息了，同时内核最后还会回到原本的正常执行流程。如果用户已经收集足够的信息，不再需要继续探测，则同样可以动态地移除探测点。因此kprobes技术具有对内核执行流程影响小和操作方便的优点。例如，Sun [28]等人基于kprobe机制开发了一个轻量级、动态的API收集工具：Trace-probe可以跟踪内核的信息，用于分析用户行为和系统软件行为。

### Visual Studio Code 中的调试架构

近年来，具有高度的可定制性轻量级集成开发环境，如Sublime Text、Atom和Visual Studio Code，已经迅速普及。鉴于轻量级集成开发环境的日益普及，经过对操作系统在线调试支持的局限性的观察，本文选择使用Visual Studio Code来设计和实现一个操作系统在线调试环境，Visual Studio Code是最受欢迎和广泛使用的轻量级集成开发环境之一[29]。Visual Studio Code，通常也称为VS Code，是Microsoft为Windows、Linux和macOS开发的源代码编辑器，功能包括支持调试、语法突出显示、智能代码完成、代码片段、代码重构和嵌入式Git，同时，微软也发布了VSCode的在线版本，它在界面和使用方面与桌面版本完全一样。

除此之外，VS Code基于进程做到了物理级别的隔离，把主进程和插件进程分开，这样，这样任何插件性能不佳或者不稳定，都不会直接影响到主进程。Visual Studio Code基于调试适配器协议，实现了一个原生的，非语言相关的调试器UI，它可以和任意后台调试程序通信，调试架构如图2.4所示。通常来讲，GDB等调试器不会实现调试适配器协议，因此需要调试适配器（Debug Adapter）去“适配”这个协议。调试适配器一般而言是一个独立和调试器通信的进程。



图 2.4 Visual Studio Code 中的调试架构

调试适配器协议主要由三个部分组成：

1. Events定义了调试过程中可能发生的事件；

2. Requests定义了VSCode等调试器用户交互界面对调试器的请求；

3. Responds定义了调试适配器对请求的回应。

## 国内外研究现状

### 操作系统调试

在操作系统开发过程中，调试是一个至关重要的环节。操作系统的调试是指对系统代码进行修改和优化，对软件层面的问题进行诊断、定位和修复的过程，以解决发现的问题，这包括检查和调试操作系统内核、驱动程序、应用程序等软件组件的功能、逻辑错误、性能问题等方面的问题。

使用日志等输出信息进行分析是一种常见的调试方法，如周毅[30]等人发明公开一种Linux内核调试系统及方法，有效解决了在操作系统启动过程中无法通过printk调试Linux内核的问题。卓维晨[31]发明公开了一种基于日志打点的Linux内核调试方法,属于Linux内核调试方法,技术方案为：使用打印日志的方法确定内核程序的运行位置,通过添加相关的日志输出接口,根据输出日志的相关信息来实现跟踪内核程序运行的全过程。

随着应用程序和操作系统变得越来越复杂，许多内核跟踪[32]工具在整个软件堆栈中兴起。Gebai [33]等人介绍了 Linux 系统上现代跟踪器在用户空间和内核空间方面的实践比较。近年来，许多研究者提出了各种针对内核故障的实用逆向调试解决方案。Lin [34]等人的工作提出了一种新的内核模糊技术来探索内核错误可能带来的所有可能的错误行为。Xinyang Ge [35]等人提出了Kernel REPT，是第一个针对内核故障的实用逆向调试解决方案，Kernel REPT采用高效的硬件跟踪来记录每个处理器上的内核控制流，根据上下文切换历史识别每个软件线程的控制流，并通过模拟机器指令和硬件事件来恢复其数据流。Bissyandé[36]等人提出了Diagnosys，一种自动构建 Linux 内核调试接口的方法，生成的调试接口提供了有用的日志信息，并且性能损失较低。Zhan [37]等人提出了一种基于系统静态污点分析的方法ErrHunter，来检测 Linux 内核中的多种错误处理错误，提出了一种自动关键变量识别方法来识别错误处理路径中的关键变量和一种静态交叉控制流污点分析方法来构建关键变量控制流图（CCFG），它描述了单独控制流中关键变量的处。ErrHunter 是为内核漏洞检测而设计的，因此它可以处理 Linux 内核的许多特定功能，例如内存管理机制等。

使用虚拟机对操作系统进行调试是一种常见的方法，它允许开发人员在虚拟的硬件环境中运行和调试操作系统，以解决软件层面的问题。虚拟机提供了一个安全、灵活的环境，开发人员可以在其中对操作系统的功能、逻辑错误、性能问题等进行诊断、定位和修复。杨杰[38]等人提出了基于bochs虚拟机的内核调试方法，bochs虚拟机提供了两种调试方式，当使用内部调试器时，其功能相当于一个硬件调试器，具有极为强大的软件调试能力，可以不受任何限制地调试操作系统的所有部分。Takeuchi [39]等人开发了一种使用轻量级虚拟机的新颖的操作系统调试方法，以提高I/O的传输速度。为了保护虚拟机上运行的客户操作系统的安全，Masaya Sato [40]等人提出了一种隐藏虚拟机上运行的程序对调试寄存器的访问的方法。[Masaya Sato](https://www.inderscienceonline.com/doi/abs/10.1504/IJSSC.2019.100007) [41]等人提出了虚拟机监视器（VMM）监视由基本进程调用以访问基本文件的系统调用，使文件对除相应的基本服务之外的所有服务不可见，保证了整个操作系统的安全。Zaidenberg [42]等人提出了一种在开发阶段检测内核漏洞的方法，LgDb是在Lguest之上构建的，Lguest是一个内核模块，它使本机Linux操作系统具有半虚拟化管理程序的能力，它的功能就像是模拟器一样，有了这个模块，在linux内核之上，可以运行多个客户内核，也就是想要调试的内核，这个想要被调试的内核运行在名为Launcher的用户空间中，它可以被主机（也就是Linux内核）控制，随时暂停，因此他就引入了KGDB这个调试工具来调试客户内核，可以让用户更加轻松的进行控制。

综而言之，操作系统调试是确保系统稳定性和性能的关键步骤之一。通过调试，开发人员能够发现、定位和修复软件层面的问题，从而提高系统的可靠性和可维护性。近年来，随着内核调试方法、调试工具和模拟器等工具的不断发展，操作系统调试的效率和精确性得到了显著提升。这些工具的使用不仅加速了问题的解决过程，还为操作系统开发人员提供了更广阔的研究和创新空间。

### 静态断点调试

静态断点调试是指在代码编译时或运行时停止程序执行，以检查程序的状态和执行路径，并查看程序的状态、变量的值、内存的状态等信息的调试方法，通过该方法可以定位和解决程序的问题，包括设置断点、单步执行、查看变量值、查看内存内容等操作。GDB就是常见的静态断点调试工具。

Linux社区开发了Linux操作系统中的内核模块调试工具kgdb，主要的调试方式：本地主机的内核运行kgdb模块，而远程主机需要通过GDB连接到本地主机的kgdb上，从而实现对内核的源码级调试[43]，如图2.5所示；Zaidenberg [42]等人将KGDB连接到内核，当用户在设置断点时，KGDB在那里插入一个中断指令，当操作系统到达断点时，操作系统进行上下文切换，并根据用户的请求进行操作。Chang [44]等人提出了基于GDB的源事务级软硬件协同调试，在软件方面，程序员通过GDB接口输入软件调试命令，然后，GDB将对这些命令进行预处理，并通过RSP协议发送到OpenOCD。最后，这些命令将被转换为JTAG信号并发送到目标处理器的ICE。蒋龙[18]提出的一款基于 GDB 的嵌入式多任务调试系统，该调试方法特别适用于对诸如 eCos、uCos 这类嵌入式微操系统进行多任务调试，为多任务调试做出了贡献，但是只支持嵌入式微内核的片上调试，过程和原理较为繁琐。



图 2.5 GDB远程调试架构

苏嘉玮[45]等人设计了面向RISC-V 微处理器，利用其提供的DM （Debug Mode） 机制，进行调试、仿真设计。采用GDB 调试工具、TCP/IP 协议、JTAG 协议等， 结合DM 机制完成软硬件调试。主机中集成环境通过GDB给某行程序设置断点，GDB 软件通过底层驱动JTAG 接口访问远程处理器的DM， 对其下达命令。DM接受命令，给设置断点的指令替换成breakpoint指令，然后当处理器执行到该指令的时候会进入调试模式的异常服务程序。GDB一直检测DM的状态，得知处理器已经停下来，并显示在软件界面上，这时就可以通过GDB经过底层驱动JTAG 接口访问远程处理器的DM，让DM来执行一些命令，比如查看寄存器的值。

但是静态断点调试对于大型复杂程序，手动设置和管理静态断点可能会变得复杂和繁琐，并且在调试过程中，程序的执行速度可能会受到影响，尤其是在设置大量断点或触发复杂条件下。静态断点调试主要用于检查程序的静态状态，难以捕获程序的动态执行信息。

尽管静态断点调试在过去几十年中一直是主流的调试技术之一，但随着软件开发环境和技术的不断发展，其仍然存在一些挑战和限制。但是，随着软件开发过程的不断优化和工具的改进，静态断点调试仍然会持续发挥重要作用，特别是在定位程序错误、优化性能和提高软件质量方面。

### 动态跟踪技术

动态跟踪是一种实时调试中常用的技术，通过记录程序执行时的信息来监控软件的行为。它能够揭示程序执行路径、参数、函数深度等关键信息，帮助程序员诊断和解决常见问题。动态跟踪技术可以在程序运行时实时监视程序的执行状态，为用户提供丰富的程序执行信息。在Linux系统中，eBPF和kprobe技术被广泛应用于动态跟踪领域。

Lakicevic [46]等人提出了dottobpf工具，利用Linux eBPF进行运行时验证，根据三值LTL规范生成eBPF程序，从而有效地进行系统分析。Ramachandran [47]等人介绍了基于eBPF技术的新型框架FUSE，它可以提供实时故障检测和抑制，有助于防止级联故障。Zhan [48]等人提出了一种基于eBPF技术的多维度网络拥塞检测方法，能够在Linux平台上动态监测网络拥塞状态机的变化。Jiang [49]等人提出了一种用于检测Linux内核环境中竞争条件的新工具，利用动态检测方法和商用处理器上的硬件调试寄存器捕捉动态竞争。Weng [50]等人开发了Kmon，一个基于eBPF的内核透明监控系统，为微服务系统提供多种运行时信息。

在操作系统和安全领域，eBPF技术被广泛应用于实现高性能和低开销的平台级监控。现有的商业eBPF监控系统虽然性能优异，但通常是紧密集成的系统，具有较大的依赖性，且在集成到替代监控系统时缺乏灵活性。为解决这一问题，Zavarella [51]等人提出了一种开发模块化独立eBPF监控系统的方法，该系统可跨各种内核版本、容器网络接口 (CNI) 插件和集群配置进行移植。

同时，动态分析技术对安全实践产生了重大影响，通过自动化一些最繁琐的漏洞检测过程，有助于提高软件的安全性。然而，现有软件工具与许多安全应用程序的需求之间仍然存在巨大差距。在这方面，L [52] i等人介绍了开发跨平台交互式分析工具的工作，该工具利用符号执行和污点跟踪等技术来分析各种平台上的二进制代码。他们的研究为解决安全领域的挑战提供了新的思路和方法。此外，为了提高Linux的实时性能，Shi [53]等人还设计了一种探针型工具，可以测量中断响应时间，输出中断调用堆栈，并分析高中断的原因。这项工作对于深入理解Linux实时性问题并寻找解决方案具有重要意义。

这些研究展示了eBPF技术在平台监控和安全领域的重要作用，同时也反映了动态跟踪技术在软件安全方面的不断发展和创新。随着对实时性能和安全性的需求不断增加，这些领域的研究和技术将持续引领行业的发展方向。

## Rust操作系统（rCore-Tutorial-v3）及调试系统架构

rCore-Tutorial-v3是使用Rust语言编写的操作系统，使用了RISC-V指令集架构，兼容 Linux，支持 x86\_64、RISCV32/64、AArch64 与 MIPS32 平台，目前清华大学的rCore-Tutorial-v3实验教程已经很完善，能够为初学者提供丰富的资源和环境，帮助学生了解rCore的基本结构，以便后续的实验和操作。目前，它的rCore-Tutorial-v3版本能够在QEMU模拟器中运行，在模拟器中完成开发阶段或测试过程，能够节省很多时间和资源。

图2.6是调试系统的一个完整的框架图，使用VSCode开发的调试器并不能够直接对操作系统进行调试，因为操作系统十分复杂，且存在不同的特权级状态。经过调研，本课题选择使用GDB这个更加强大的调试器作为中间组件，协助开发源代码级调试器进行对操作系统调试。当调试适配器接收到一个请求，它就会将请求转换为符合 GDB/MI 接口规范（GDB/MI 是一个基于行的面向机器的 GDB 文本接口，它专门用于支持将调试器用作大型系统的一个小组件的系统的开发。）的文本并发送给 GDB。GDB 在解析、执行完调试适配器发来的命令后，返回符合 GDB/MI 规范的文本信息。调试适配器将 GDB 返回的信息解析后，向调试器返回相应的消息，并展示给用户。



图 2.6 总框架图

# 面向Rust操作系统的多进程调试方法

## 方法概述

随着Rust编程语言的流行和逐渐成熟，越来越多的开发者将其用于构建操作系统。与传统的编程语言相比，Rust具有内存安全性、并发性和性能优势，使其成为开发操作系统的理想选择之一。然而，尽管Rust提供了许多优点，但在实际应用中，针对Rust操作系统的调试仍然面临一些挑战。

对Rust操作系统进行多进程调试的过程中主要面临以下几个问题，首先，操作系统中包括用户态代码和内核态代码，分别对应不同的特权级，不同的特权级又对应不同的符号表。在操作系统运行的过程中，会频繁地进行特权级的切换，导致调试信息的丢失，而目前现有的调试器都无法进行跨特权级的断点调试。其次，调试器和被调试操作系统是两个不同的独立系统，它们分别运行在自己的上下文中，且各自拥有自己的运行时环境。在这种情况下，调试器无法直接获取被调试操作系统当前运行在哪个特权级上面，因为调试器本身并没有“特权级”的概念，因此，想要完成多进程的调试，需要让调试器获取到当前被调试操作系统运行在哪个特权级。最后，操作系统通常有多个用户进程，用户进程切换的主要难题之一是获取下一个要运行的进程的名称，并获取其符号表和调试信息，从而完成对多进程的调试。

针对上述问题，本文提出了基于GDB的多进程调试方法，在该方法中，首先设置了断点组管理模块，缓存不同特权级下地址空间的断点，并在被调试操作系统运行到对应的地址空间的时候再激活断点，实现断点组的切换，解决了特权级切换导致调试信息丢失的问题；其次，设置了边界检测点，使调试器能够检测到当前特权级，并根据特权级切换不同的符号表，从而能够让使用者进行跨特权级的调试；最后，在内核代码中获取下一个要运行进程的标识符，并告知调试器，调试器根据进程标识符来确定下一个要切换进程的符号表，获取到新进程的调试信息，对新进程进行调试，同时也能够支持多个用户进程的符号表切换，从而实现了对多个用户进程的调试。

在完成上述方法研究以后，发现对Rust操作系统的调试并不完善，针对Rust语言的特殊性，多进程调试仍然面对一些挑战。由于Rust语言的静态特性以及操作系统内核的复杂性，导致调试信息不够完善。这使得使用传统的调试工具，如GDB，在调试Rust操作系统时遇到了困难，一些重要的数据结构无法展示。特别是，在静态断点调试模式下，GDB会暂停被调试操作系统的运行，从而阻碍了对操作系统运行状态的动态监测。

针对上述问题，本文提出了一种静态断点的调试和动态跟踪结合的方法来对Rust操作系统进行调试。其中，静态断点调试指基于GDB调试工具，暂停操作系统运行，查看相关调试信息的调试方法，而动态跟踪指的是，使用像eBPF或者kprobe等能够动态观测内核状态信息的工具，对操作系统进行动态的监测。在该方法中，编写了一个eBPF处理模块，它可以在被调试操作系统运行过程中收集调试信息，为了将静态断点调试和动态跟踪结合到一起，本文通过串口的方式将eBPF收集到的调试信息也传送到GDB，信息的传递使用了和静态断点调试一样的RSP协议；接下来，本文将eBPF传递过来的信息和调试适配器进行适配，eBPF传递过来的信息与用户所使用的调试器进行适配，最后通过VSCode调试插件展示给用户。

## 基于GDB的多进程调试方法

### 概述

操作系统逻辑结构复杂，运行过程中会在不同特权级下创建多个进程，在研究基于GDB的多进程调试方法中主要遇到以下问题。首先，操作系统中包括用户态代码和内核态代码，对应不同的特权级，不同的特权级又对应不同的符号表。在操作系统运行的过程中，会频繁地进行特权级的切换，导致调试信息的丢失，而目前现有的调试器都无法进行跨特权级的断点调试。具体来说，特权级的切换主要涉及符号表的切换，符号表包含了编译后的代码中各种变量、函数、数据结构等的名称和地址信息，这些是调试代码所必需的内容，同时符号表也是编译后的代码与源代码之间的桥梁，使调试器能够将二进制代码中的地址映射回源代码的符号名，因此无论调试任何进程，调试器都需要先加载进程的符号表。而在操作系统中内核态程序和用户态程序的符号表是分开的，如果程序运行中进行了用户态和内核态的转换，符号表也要随之切换，符号表切换以后，用户设置的程序断点也会随之消失，比如在内核态设置用户态的断点以后，再进入用户态，用户态的断点将不会被触发。为了解决这个问题，在本文中新增了一个断点组管理模块。断点组管理模块会先缓存设置的异常断点（例如当前操作系统处在内核态代码的运行中，操作人员在用户态代码中设置了断点，该断点即为异常断点），等到特权级切换到对应的状态时，再将缓存的断点信息进行激活，即在用户态运行时，缓存用户设置的内核态断点，等到程序运行到内核态时，再激活缓存的内核态断点；在内核态运行时，缓存用户设置的用户态断点，等运行到用户态时，再激活用户态断点。通过这种方法，操作人员可以在任意状态下对被调试操作系统的任意代码行设置断点，从而解决不同特权级下断点的调试信息丢失问题。

其次，调试器和被调试操作系统是两个不同的独立系统。它们分别运行在自己的上下文中，且各自拥有自己的运行时环境。在这种情况下，调试器无法直接获取被调试操作系统当前运行在哪个特权级上面，因为调试器本身并没有“特权级”的概念。为了解决这个问题，设置一个自动断点机制，在内核态进入用户态和用户态返回内核态时设置断点，分别为内核入口断点和内核出口断点，这两个断点称为边界断点。如果边界断点被触发，就意味着特权级发生了切换，内存地址空间也会发生切换，因此断点组也应该切换。每次断点被触发时，调试适配器都会检测这个断点是否为边界断点。如果是，它会先移除旧断点组中的所有断点，然后设置新断点组的断点。为了确保相关功能正常运行，断点组切换时，符号表文件也应该随着断点组的切换而切换。

最后，操作系统通常允许在用户态创建多个进程对应不同的工作，即用户进程，多个用户进程会抢占有限的中央处理器运行资源，即调度执行或用户进程切换。类似于用户态与内核态，每个用户进程都有自己的符号表，每次进行用户进程切换的时候都需要通过系统调用进入内核态，更新内存地址空间，然后返回用户态执行新的进程。本文设计了通过进程标识的获取对应不同的符号表，并在调试器检测到用户程序切换时对符号表进行切换，以此实现多个用户进程的调试。

综上所述，本方法主要包含以下内容：

1. 提出了断点组管理模块的概念，能够缓存内核态和用户态的断点等调试信息，随着特权级和符号表的切换，断点也随之切换，从而解决不同特权级下断点的调试信息丢失问题。
2. 设置自动断点机制，让调试器能够感知到被调试操作系统进行了特权级切换，从而实现跨特权级的源代码级操作系统调试。
3. 设计了通过调试的方法，获取到不同用户进程的进程标识符，并根据进程标识的获取对应不同的符号表，让调试器根据符号表来进行进程的切换，从而实现了对多个用户进程的调试。

### 断点组管理模块的设计

特权级切换会导致断点等调试信息的丢失，为了保存因为特权级切换而失效的断点调试信息，在本方法中设置了断点组数据结构，断点组会保存不同地址空间中断点等调试信息。它主要使用一个词典缓存了用户要求设置的所有断点（包括内核态和用户态）。词典中的每个元素都是一个键值对，其中键是程序运行所占内存地址空间的代号，值是该代号对应的断点组，包括用户态断点组和内核态断点组，分别保存了用户态和内核态对应内存地址空间内的所有断点。

除此以外，为了满足调试需求，设置了一个当前有效断点组变量，即被调试操作系统当前执行的进程地址空间对应的断点组，只有当前有效断点组中的断点才会被激活，随后会被触发，不是当前有效断点组的断点只会被缓存到其他地址空间的断点组中，不会被触发。

图3.1展示了断点组数据结构包含的主要变量，包含一个地址空间的集合，还有一个当前操作系统所运行进程的地址空间。

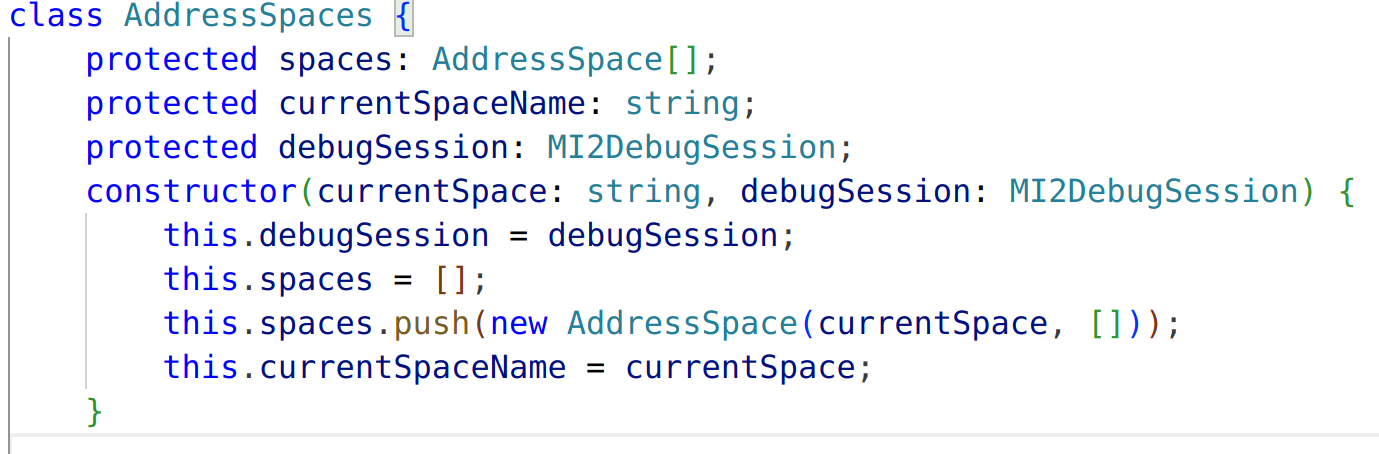


图 3.1 断点组数据结构包含的主要变量

当用户在调试器中设置新断点时，断点组管理模块的执行流程如图3.2。调试器的断点组管理模块会先将这个断点的信息存储在对应的断点组中，然后判断这个断点所在的断点组是否为当前有效断点组。如果是，就立即激活这个断点，程序运行到该断点时就会触发这个断点。如果不是，那么这个断点暂时不会被激活，将会被保存到断点组中，即使程序运行到该断点的时候也不会触发该断点。在这种缓存机制下，用户态断点和内核态断点不会同时被激活，从而避免了内核态和用户态断点的冲突，保存了因特权级切换而丢失的断点调试信息。



图 3.2 断点设置流程图

### 调试器中内核态和用户态的切换

#### 边界断点的实现

断点组管理模块保存了用户态和内核态对应内存地址空间内的所有断点，但由于调试器和被调试操作系统是分开运行的，因此接下来要让调试器知道断点组切换的时机，本文通过以下方法让调试器检测到被调试操作系统的特权级切换：通过人工分析找到被调试操作系统的用户态和内核态切换的边界代码位置，并设置边界断点。当操作系统运行到边界断点时说明马上就会进行特权级的切换，此时调试器自动中断操作系统的运行，并暂停到边界断点的位置，并进行对应符号表的切换以及断点组的切换，最后恢复操作系统的运行。如图3.3所示，当用户进程A想要通过系统调用进入内核态时，会触发边界断点，这时调试器就会检测到被调试操作系统发生了特权级切换，接下来就会进行符号表切换、断点组切换，在切换过程中，调试器会删除原进程对应的地址空间中设置的断点，设置新进程地址空间中断点组的断点，从内核态回到用户态也是一样的流程。边界断点不属于任何断点组，如果设置了边界断点，就会立即被激活然后等待被触发。经过这样处理以后，当操作系统进行特权级转换的时候，调试器通过对边界断点检测和各种处理实现了内核态和用户态的转换。



图 3.3 边界断点及特权级转化过程图

图3.4展示了调试器配置和使用的整体流程，操作者需要先找到边界断点的位置，并进行配置，随后就可以启动调试。在调试过程中，被调试操作系统启动以后会先进入内核态，接着被调试操作系统继续运行，会暂停到边界断点的位置，这时用户就可以在源代码的任意位置设置断点，但是只有内核态中的断点会被激活，随后被触发，最后会暂停到内核出口检测点的位置，调试器进行符号表和断点组的切换。接下来就会进入指定的用户进程，该用户进程中的所有断点都被激活，并暂停到用户进程中设置的第一个断点，在用户进程中，用户可以通过设置好的调试选项，在任意时刻进入内核态，在这个过程中，用户可以在任意时刻、任意代码行设置断点，观测各种变量信息，最终完成整个操作系统的调试。



图 3.4 调试器配置使用流程

#### 用户态进入内核态的调试选项

在操作系统的运行过程中，内核态和用户态的切换非常频繁，在一个用户进程运行过程中可能会多次进行系统调用，如打开一个文件等，如果操作系统每次进行系统调用进入内核态，调试器都要跟随着操作系统进行用户态和内核态的切换是非常繁琐的，可能会出现以下现象，在用户程序运行到一个打开文件的操作时，它就会通过系统调用进入到内核态，而调试器也会随着进入到内核态，经过内核入口断点，内核出口断点最终回到用户进程中，但实际上，使用者并不想观察打开文件这一系统调用的整个过程，这会非常浪费时间，因此本研究为调试器添加了一个单独的调试选项，可以让使用者自己控制调试器的调试流程：是否要跟踪某个系统调用进入内核态。具体操作如下：如果使用者只需要调试用户程序的话，就不需要使用该调试选项；如果使用者想要观察内核的运行状态，使用者首先要在期待进入内核态的代码行设置断点，接下来，等到程序暂停到该行代码的时候设置调试选项，该调试选项会设置边界断点，在设置边界断点以后，程序继续运行，就会触发边界断点，调试器检测到特权级的切换，就会切换符号表和断点组，从用户态进入到内核态，接下来，调试器就会暂停到内核代码中的断点，展示内核运行中的各种调试信息，接着，内核程序执行完毕，运行到内核出口断点，最终就会回到用户态中，暂停到当初进入到内核态的那一行代码。通过设置调试选项，使用者可以选择调试过程中进入内核态的时机，能够提高调试工具的易用性

### 多个用户进程的切换和调试

在上节描述了调试器进行特权级切换的流程，但是，在用户态会有多个用户进程在运行，调试器也需要实现对多个用户进程的调试，也就是实现多个用户进行之间的切换。调试器是根据进程标识符，来进行符号表切换的，本文在调试器中使用一个变量来保存被调试进程的标识符，如果操作系统进行了进程的切换，只需要改变这个变量调试器就能够知道下一个要运行的进程是哪一个，并进行符号表的切换，因此首先要获取进程标识符。在操作系统中，切换用户进程都需要经过系统调用进入内核态，内核态包含了很多重要的数据结构，其中就包括进程控制块，进程控制块中记录了进程标识符，因此需要在进程进入到内核态的时候，获取到下一个运行的进程标识符。操作者需要在被调试操作系统的内核态源代码中，含有进程标识符变量的代码行设置断点，如sys\_exec函数中，通过给GDB发送获取变量信息的命令，并解析返回信息，来获取进程标识符。在调试器获取到进程标识符以后，就可以根据进程标识符来进行符号表的切换，最终实现多进程的调试。

#### 进程控制块信息的获取

要获取进程控制块中的进程标识符需要进行两个方面的修改，首先，在目前调试的rCore操作系统（rCore-Tutorial-v3）中，进程控制块里面是不保存进程标识符信息的，因此先要修改rCore-Tutorial-v3，在进程控制块中添加进程标识符信息，其次，在调试器中获取进程控制块中的进程标识符。

在rCore-Tutorial-v3中get\_exec\_path 函数用于获取进程控制块中进程标识符，在进程控制块中添加进程标识符过程大概分为三个部分：

1. 操作系统创建零号进程initproc 时，在其进程控制块内添加上进程名"initproc"；

2. 在fork() 系统调用中， fork 出的子进程需要赋予和父进程相同的进程名；

3. 在exec() 系统调用的代码中，需要修改当前进程控制块里存储的进程名，因为exec() 系统调用不会创造新的进程控制块，而是覆盖当前资源，基于当前进程控制块来运行新程序。

完成以上三步，进程控制块里就保存了进程名，之后， get\_exec\_path 函数只需要获取到当前进程的进程控制块的独占借用，即可得到需要的进程名。

接下来就是在调试器中获取进程控制块中的进程标识符。在目前调试的rCore操作系统中都是在用户程序进行fork子进程，并调用exec执行子进程的代码中进行多个用户进程的符号表切换的，也就是用户程序中必须含有fork和exec这两个函数的时候才会有新进程的产生，而跟踪exec函数是一个非常好的选择，因为他的函数参数里面包含了下一个要执行的进程的名称也就是进程标识符。

在操作系统中，进程标识符保存在进程控制块这个数据结构中，因此需要获取到进程控制块信息，在操作系统源代码中包含有进程控制块变量的位置设置断点，当程序运行到该断点的时候就可以给GDB发送获取变量信息的命令，并解析返回信息，来获取进程标识符。

在本文开发的调试器中，给GDB发送消息需要调用发送GDB命令的函数，他是一个异步函数，他里面包含一个计数器，和一个promise对象。计数器记录了当前发送的GDB命令的编号，每个命令都会有一个编号，并且这个命令的返回信息也会被赋予这个编号，他们之间有一个显式的对应规则。promise对象代表了未来将要发生的事件，用来传递异步操作的消息。在promise对象中，首先会给GDB发送命令，之后该函数就进入等待状态，直到收到了命令的返回信息且返回信息的处理函数被调用，其中有编号的返回信息会被保存到handlers这个结构体里面，如图3.5所示。handlers结构体的类型是一个数字索引签名，数字索引签名允许使用数字类型的索引来访问对象的属性。索引的值为一个函数类型(info: MINode) => any，函数的参数就是返回的MI类型信息，在函数体里面可以对返回信息进行处理，也就是上面回调函数的处理。



图 3.5 发送命令函数执行流程

在目前的程序中，调试进程给GDB发送命令以后，GDB给出的返回信息中除了发送一条带有编号的表示命令的执行状态和必须携带的基本信息以外，GDB还会返回一些关于命令的其他信息或者一些警告或者提示，例如异步执行输出，异步状态输出，异步通知输出，控制台流输出，目标流输出，日志流输出，他们之中有些是没有编号的，也就是token为undefined，如控制台流输出，对于这些没有编号的信息输出，源程序是没有进行储存的，在调试控制台输出以后就被清理了，如果要用的话就无法获取到，因此要对GDB返回消息处理函数进行修改，将调试器需要的信息保留下来。

在消息处理函数中传入的参数就是GDB返回的信息，可能是一个字符串或字符串数组，处理的过程中，首先要把返回的信息，解析成MI格式的数据，如果该数据的token不是undefined，那么就调用上述handlers数据结构中的处理函数。如果该数据的token为undefined，对于这些数据，程序是没有进行保存，也没有进行任何处理的，但是调试器需要的数据有一部分就是包含在这样的数据中的，因此在调试器中给token为undefined的数据进行了单独的处理。一个GDB命令返回的信息可能会被分成很多行进行返回，如图3.6所示，而在目前的处理程序中，对返回信息都是一行一行来进行处理的，本文要把同一个命令返回的多行信息添加同一个标号，最好的赋值方法就是直接给这些多行信息的编号赋值成获取该信息的GDB命令的编号。本文增加了一个存储MINode格式数据的数组，将这些已经有了token编号的信息保存到该数组中，根据这些编号的时效性，本文为该数组设置了一个回收机制，当数组中的信息超过100条时就全部清空。经过这样处理，想要获取命令返回信息中token为undefined的数据，就可以直接遍历该数组，然后根据命令的编号，获取到对应的信息。



图 3.6 数据格式处理示意图

最终，通过在操作系统的源代码中包含有进程控制块变量的位置设置断点，在该断点处发送GDB命令，并在众多返回信息中解析出了进程控制块信息，在进程控制块信息中就包含了要进行进程切换所需要的进程标识符。

#### 多个用户进程的符号表切换

在获取进程标识符以后，就可以进行用户进程符号表的切换。在被调试操作系统刚刚运行起来的时候都是先从内核态开始运行的，然后会指定一个第一个要运行的用户态进程，如用户进程A，在调试器触发边界断点的时候，调试器就会根据用户进程A的进程标识符进行符号表的切换，切换到指定的用户进程A的符号表，切换断点组，激活在用户进程A中设置的断点。接下来要进行进程切换的话，就需要从当前进程A进入内核态，选择一个合适的位置设置断点，如exec函数，当程序暂停到exec函数那一行的的时候，就设置从用户态进入内核态的调试选项，调试选项设置边界断点，当边界断点被触发时，切换到内核的符号表，切换断点组，在内核态运行的时候，获取到下一个要执行的用户进程标识符，如用户进程C。在获取到用户进程C的标识符以后，赋值给保存进程标识符的变量，这样调试器就能够知道下一个要运行的进程是用户进程C。接下来从内核态进入用户态，在边界断点进行符号表切换，此时要切换的符号表就是用户进程C的符号表，断点组切换，激活用户进程C中设置的断点，最终实现了多个用户进程的符号表切换。



图 3.7 多进程切换示意图

### 小结

在本小节中，首先，详细阐述了目前面向Rust操作系统的多进程调试的主要问题与挑战，接下来，阐述了基于GDB的多进程调试方法的研究过程，在该方法中讲述了使用断点组管理模块，解决特权级切换导师调试信息丢失的问题；使用设边界监测点的方法，解决调试器中内核态和用户态切换的问题；最后，通过获取进程标识符，获取到下一个要运行进程的符号表和调试信息，解决了多个用户进程切换和调试的问题。

## 一种静态断点调试和动态跟踪结合的方法

### 概述

QEMU自带的GDB工具可以对操作系统进行静态断点调试，如上文所述，本文实现了对操作系统的跨特权级的调试和多进程的调试，但是由于 GDB 对 Rust 语言的支持不够完善，像C/C++ 语言的编译器在生成调试信息方面更加成熟。而Rust语言的调试信息可能不如 C/C++ 语言的那么详细，这会影响 GDB 在 Rust 代码中的变量和数据结构查看以及堆栈跟踪的质量。除此以外，GDB 静态断点调试具有一定的局限性，如无法在运行时动态地捕获操作系统中发生的事件，无法查看函数调用参数等。而eBPF和kprobe专门针对内核级别的调试，可以捕获并监视在操作系统内核中发生的事件，例如系统调用、中断等，允许用户在内核运行时执行代码并监视系统级事件。kprobe还允许在运行时通过代码注入的方式动态设置断点，并且灵活捕获感兴趣的事件。本文希望将GDB静态断点调试功能和eBPF、 kprobe的动态跟踪功能结合到一起，这样能够对运行中的操作系统有更加全面的调试和监测。

因此，本文提出了出了一种静态断点调试和动态跟踪结合的方法。本文设计了eBPF处理模块， eBPF处理模块具有函数插桩等动态跟踪功能，可以在运行时对内核进行监控，捕获有关函数执行和数据流动等更多的信息，接下来实现了运行在操作系统中的eBPF处理模块与GDB的连接与交互。从而和 QEMU 虚拟机提供的GDB服务器功能互补，获取到更多内核信息。使调试器具有更大的灵活性，能够让使用者能够更加准确地定位和解决问题，尤其在处理操作系统内核相关的复杂问题时表现突出。

为了实现把基于GDB的多进程断点调试和基于eBPF的动态跟踪调试结合在一起，本文进行了以下工作：首先，本文目前调试的Rust操作系统（rCore-Tutorial-v3）并不支持eBPF和kprobe的运行，因此本文为rCore-Tutorial-v3移植了eBPF和kprobe模块，其次，目前，被调试操作系统运行在QEMU中，它使用GDB服务器与GDB进行信息的传递，最后再由GDB通过调试适配器进程和用户本地的调试器进行通信；因此为了将用户的调试需求传递给eBPF，并把eBPF在被调试操作系统中收集到的数据整合起来，并传递到用户界面上，本文为eBPF在 QEMU上添加了串口进行输出，并为eBPF设计了单独的处理模块，负责接收用户的调试需求并将获取到的数据传递到GDB中，再由GDB发送到用户本地的调试界面上；最后，本文要在用户本地的调试器中适配eBPF，包括调试信息的处理和发送，还有数据的展示。

综上所述，本小节的主要内容如下：

1. 提出了一种静态断点调试和动态跟踪结合的方法，在本文目前调试的Rust操作系统（rCore-Tutorial-v3）中移植eBPF和kprobe模块，使rCore-Tutorial-v3支持eBPF和kprobe的运行；
2. 为eBPF调试信息实现基于串口的通信机制，设计eBPF处理模块，负责接受用户的调试请求，并把获取的调试信息传递给GDB，并在串口之上实现了RSP协议，统一GDB服务器和eBPF处理模块与GDB的通信方法，最后，在用户本地的调试器中适配eBPF，包括调试信息的处理和发送，还有数据的展示。

### 动态跟踪调试的工作机理

动态调试是一种在程序运行时观察和修改其行为的强大技术。动态调试允许开发者实时观察程序的执行状态，包括变量值、函数调用栈等，可以在程序运行时发现和修复错误，而不需要重新编译和重新启动。

kprobe是Linux内核中强大的动态跟踪工具，它允许在内核函数的入口或出口设置断点，并可以附加处理程序来监视函数的调用或返回。它通过在内核代码中插入断点来捕获指定函数的调用，并在发生时执行预定义的操作。eBPF可用于编写内核扩展程序，eBPF 提供了一种安全且高效的方式，允许用户向内核注入代码来执行特定任务。用户可以使用C语言手动编写eBPF程序，里面包含eBPF指令，在该程序中，用户可以使用特定的函数在内核代码的任意指令行放置探测点，同时也可以获取到该探测点的相关信息，当该指令被执行的时候就会触发eBPF程序。

图3.8展示了将 eBPF 用于操作系统动态跟踪调试的一个典型的工作流程。用户程序提供eBPF字节码，通过系统调用加载进内核。这个字节码程序经过verifier验证后交付给eBPF模块执行。通过kprobe内核监测模块，eBPF程序可以在运行到特定地址或函数时执行，从而在内核地址空间中动态地收集调试信息，并将收集到的数据存储在 eBPF maps 中，使用者可以编写用户程序对存储在 eBPF maps 中调试信息进行获取，分析。



图 3.8 eBPF工作流程

### eBPF移植

在目前本文调试的操作系统中，rCore-Tutorial-v3并没有eBPF的支持，因此首先要进行的就是eBPF移植工作，让rCore-Tutorial-v3能够支持eBPF程序的运行。rCore-Tutorial-v3的一个分支rCore-Tutorial-2022A已经有了eBPF支持，且将相关代码封装成了三个模块。因此，为了实现eBPF 处理模块，本文将这三个模块移植到了rCore-Tutorial-v3上。

移植过程中遇到的主要障碍之一是rCore-Tutorial-v3 的中断处理流程中使用了较新的Trap::Breakpoint模块，而rCore-Tutorial-2022A用的是Trap::Exception模块，需要在这两个模块之间进行转换。其次，考虑到用户在大部分情况下都是使用函数名、变量名等符号而不是具体的地址来设置断点，eBPF模块需要有将符号转换为地址的功能。rCore-Tutorial-2022A的eBPF模块预留了这个功能的接口但并没有实现，而本文在rCore-Tutorial-v3的eBPF模块中实现了这个功能。

实现符号转地址功能的第一步是获取完整的符号表。rCore-Tutorial-v3 是用 cargo 工具创建的。一般而言，用cargo工具创建的 Rust 项目可用发布模式和调试模式两种模式编译、运行。在这两种模式中，发布模式对代码进行较高等级的优化，删除较多调试相关的信息，而调试模式则对代码进行较弱等级的优化并保留了更多调试相关的信息，比较符合本文的需求。但是由于 rCore-Tutorial-v3 项目本身的设计缺陷，这个项目不支持使用调试模式进行编译。因此，本文需要修改发布模式的配置文件，让编译器在发布模式下也像在调试模式下一样关闭代码优化，保留调试信息。在获取到更加完整的调试信息以后，本文首先通过内嵌汇编的方式，在内核的数据段中分配了3MiB的连续空间用于存放符号表；其次修改用于构建内核的Makefile脚本，使得Makefile在构建内核后通过调用nm工具从内核镜像中提取内核符号信息并保存为文本文件；最后利用dd命令，将符号文件注入到内核镜像中。

需要注意的是，Rust编译器会对函数名进行名字改编（Name Mangling），这会对eBPF模块的符号解析功能造成障碍。在早期版本的Rust工具链中，可以通过添加rustflags = ["-Zsymbol-mangling-version=v0"] 编译参数关闭名字改编；在近期版本的Rust工具链中这个编译参数不再有效，需要用rustfilt工具将保存有内核符号信息的文本文件中的所有被名字改编的符号还原为原本的符号。在有了完整且可解析的符号表并成功注入内核后，本文编写了一个工具函数用于在符号表中搜索特定的符号并返回符号对应的地址，从而实现了预留的符号解析接口。

在完成上述工作以后，rCore-Tutorial-v3操作系统已经能够支持eBPF程序的运行，在操作系统执行的过程中，可以实现对调试信息的收集。

### eBPF与GDB调试信息的整合

#### eBPF处理模块的设计与实现

本文使用了一种静态断点调试和动态跟踪结合的方法，来提高调试器的调试能力。运用两种调试技术，GDB调试技术和eBPF跟踪技术，同时调试同一个目标，即虚拟机中运行的操作系统。

GDB具有可以改变操作系统运行状态的控制能力，而eBPF只负责收集信息，不影响内核的状态。如表3.1所示，该表详细展示了GDB和 eBPF功能与局限，可以看出，二者形成了很好的互补。

表 3.1 eBPF和GDB的功能对比

|  |  |  |
| --- | --- | --- |
|  | eBPF | GDB |
| 读内存，读寄存器 | 可以 | 可以 |
| 写内存，写寄存器 | 不可以 | 可以 |
| 获取进程控制块等内核信息 | 方便 | 繁琐 |
| 停下（halt） | 不可以 | 可以 |
| 单步调试 | 不可以（原因是不能停下） | 可以 |
| 查看断点 | 不可以（原因是不能停下） | 可以 |
| 跟踪函数调用关系 | 优点：查看函数调用的参数 | 优点：查看函数调用栈 |
| 断点 | 触发后被调试的操作系统不能停下，主要起辅助作用 | 断点触发后被调试的操作系统会停下，这对于静态分析功能来说是必不可少的 |
| 跟踪异步函数 | 可以编写帮助函数，较方便 | 较繁琐 |

为了实现运行在操作系统中的eBPF程序与GDB的连接与交互，本文实现了eBPF处理模块。

在Rust操作系统能够支持eBPF程序运行以后，本文就可以使用eBPF和kprobe收集被调试操作系统运行中的调试信息，接下来，本文要将eBPF收集到的数据，传递到用户本地的调试插件中。在原始的调试架构中，GDB服务器运行在QEMU中，与GDB通过RSP协议进行数据的传递，再通过GDB与用户本地的调试插件进行通信，因此，本文希望eBPF收集到的数据使用和GDB服务器的数据使用一样的传递路径，需要把收集到的调试信息通过RSP协议传递给GDB，再由GDB传递给VSCode调试插件，也就是调试器，被调试操作系统及用户本地调试插件之间的数据流图如图3.9所示。



图 3.9 数据流图

利用 GDB 自带的远程调试功能，本文很容易就能建立QUME中的GDB服务器和 GDB 的连接。接下来的主要问题就是如何让 GDB 在连接到GDB服务器的同时也连接到eBPF处理模块。

#### eBPF处理模块与GDB基于串口的通信机制

GDB 和 GDB服务器 用 TCP 协议通信，但是由于主线版本的 rCore-Tutorial-v3 暂未提供稳定的网络协议栈和网卡驱动支持（不论是真实网卡还是 QEMU 虚拟机的 virtio 虚拟设备）且 eBPF处理模块是运行在操作系统里的，因此若要让 eBPF 处理模块用TCP协议连接到 GDB ，实现难度比较大。本研究在调研了各种调试器与调试器服务器通信的方案后，选择用串口进行二者的通信，将调试信息、调试命令的传输和一般的输出语句分离，通过一个专有的串口进行调试信息，调试命令的传输。这样，eBPF 调试功能就不会影响操作系统的运行状态。

操作系统能够支持eBPF程序运行以后，要实现eBPF处理模块和GDB的交互。操作系统进行编译后，运行在QEMU虚拟机上，当用户在调试窗口提供的调试选项中开启eBPF功能后，eBPF处理模块就开始运行。本文使用了一种基于串口的 GDB 与 eBPF处理模块的通信机制。

由于 rCore-Tutorial-v3 的终端已经占用了一个串口用于文字输入输出，为了不影响操作系统的运行状态，eBPF 处理模块需要用另一个专属的串口来和 GDB 通信。为了给QEMU虚拟机添加串口，本文修改了 QEMU虚拟机的源代码，为新串口分配了 MMIO 地址和 IRQ（中断号），主要修改内容如图3.10，3.11，3.12所示。

**表格

低可信度描述已自动生成**

图 3.10 修改头文件，添加新串口的枚举名和中断号

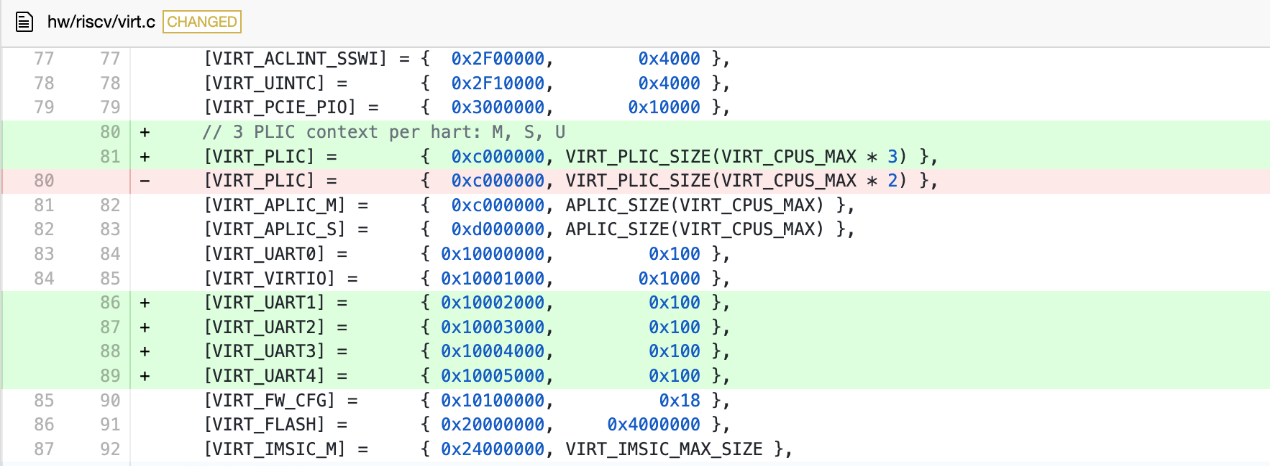


图 3.11 为新串口分配 MMIO 地址

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图 3.12 新串口初始化

还修改了设备树初始化函数，如图3.13所示：



图 3.13 设备树初始化函数

经过修改后的 QEMU 中的 rCore-Tutorial-v3 操作系统可以给第二个串口发送消息，但不能接收基于中断机制的消息。这是因为在 RISC-V 中，存在用于保护物理地址的寄存器 pmpcfg 和 pmpaddr 。而 rCore-Tutorial-v3 的 SBI（rustsbi-QEMU）通过设置这两个寄存器的值，使得只有 SBI 部分和 OS 所在的地址空间可以使用，而PLIC、串口等设备所在的物理地址不可以使用。为了调试方便，本文修改了SBI对 pmpcfg 和 pmpaddr 的设置，使得所有物理地址都可以被使用。

同时，为了支持第二个串口的通信，本文参考原有的串口相关的代码，对 rCore-Tutorial-v3 做了尽可能少的修改，修改内容包括：添加第二个串口的初始化例程、修改中断处理例程、添加用于在第二个串口收发单字节的系统调用和 eBPF 帮助函数（helper functions）。至此，实现了基于中断的多串口数据收发，使得 GDB 与 eBPF处理模块可以通过串口通信。

#### eBPF处理模块与GDB之间的RSP协议实现

在实现GDB 与 eBPF处理模块的通信以后，需要规定两者进行数据传递的协议。在串口或网络之上，GDB 和GDB服务器之间用 RSP 高层协议进行通信，因此，在本文中，eBPF处理模块与GDB的通信也选择使用RSP 高层协议进行通信。

RSP协议规定的基本消息单位是由ASCII字符组成的数据包（Packet），数据包的格式如图3.14所示，其中，“$”用于标识数据包的开头，packet-data是传送的实际数据，checksum（即校验值）是“$”和“#”之间所有字符的模 256 和，它是八位无符号整数，编码成ASCII字符后一定占用两个字符，因此“checksum”这三个字符标示了一个数据包的结束。通信的双方在接收到数据包后，可以发送单个“+”字符表示数据包接收成功，或发送“-”表示数据包接收失败，要求重发。

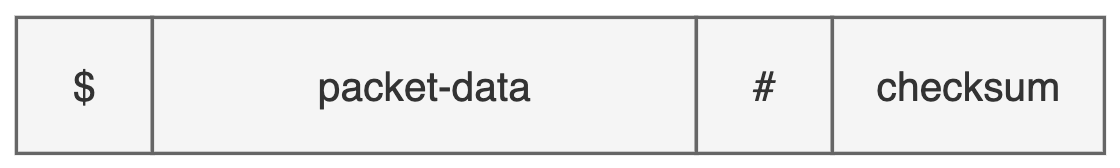


图 3.14 RSP协议的数据包格式

除此之外，还有一种通知数据包（Notification Packet），它和普通的数据包的区别有两个：1）普通数据包的交流机制是同步的，而通知数据包是异步的，常用于通知 GDB 某个事件的发生，2）收到通知数据包后无需像普通数据包一样发送“+”或“-”。通知数据包的格式如图3.15所示，可以看到，通知数据包以字符“%”标识开头，而其余的格式和普通的数据包是一致的：

图形用户界面, 文本, 应用程序

描述已自动生成

图 3.15 RSP协议的通知数据包格式

尽管GDB服务器和eBPF处理模块都使用RSP协议与GDB进行通信，但在实际通信中，GDB服务器主要以同步方式收发消息。这是由于QEMU的GDB服务器调试机制是同步的。通常情况下，QEMU的GDB服务器会在断点被触发并暂停被调试的操作系统后才开始收集信息。相比之下，eBPF处理模块的跟踪调试功能主要依赖内核插桩机制，在插桩触发之后 eBPF处理模块收集数据，收集完毕后eBPF程序立即退出，操作系统继续运行。eBPF处理模块不会为了和 GDB 通信而让操作系统停下。因此大部分的信息都会以异步的方式传送给GDB。这种异步的消息处理方式可以提供更高的并发性和响应性，然而，异步消息可能会与同步消息重合，这就要求和eBPF处理模块通信的 GDB 子模块具有较好的鲁棒性，能恰当地处理同步信息的字节流被异步信息的字节流打断的情况。

RSP 协议规定，同步消息以字符“#”开头，而异步消息以字符“%”开头，根据这个特性，本文设计一个消息处理流程，如图3.16所示，可以确保消息被有序处理：GDB中负责和eBPF处理模块通信的子模块逐字节接收来自eBPF程序的消息，默认情况下按同步信息处理，如果发现接收到字符“%”，则接下来接收到的字节都放入异步消息处理例程，直到接收到“#”符号后，再返回原来的同步消息处理流程继续从串口接收同步信息。这样，就算同步消息被异步消息打断，同步消息和异步消息都能被完整地接收。

表格

描述已自动生成

图 3.16 消息处理例程

由于多个 eBPF 程序不会并发运行，因此异步消息流之间是按顺序发送的，不会互相重叠；运行 eBPF 程序时，操作系统其他部分是不运行的，操作系统中负责收发同步消息的用户态进程也不运行，直到 eBPF 程序发送完异步消息后，这个用户态进程才会继续运行，继续同步消息的发送，这种机制可以确保异步消息不会被同步消息打断。

这套协议比较简明，也非常适合GDB 与 eBPF处理模块进行通信，因此本文在GDB中增加一个子模块，让这个子模块使用 RSP 协议和 eBPF处理模块进行通信。考虑到这个子模块将会有一个持续从串口接收字节流的线程，且有很多字符串处理流程，本文选择用 python 语言来编写这个模块，因为用 python 语言创建和管理线程比较简单，且已经有功能强大的 pyserial 库能够便捷地处理串口消息的收发，图3.17为异步消息处理例程的主要代码，图3.18为同步消息处理例程的主要代码。GDB 允许在不修改源代码的情况下支持 python 语言编写的扩展脚本，然而这个特性在本文使用的 Ubuntu20.04 的包管理器自带的GDB，和RISC-V工具链提供的 GDB 可执行文件中都是关闭的，因此本文需要自行编译一份支持 python 扩展的GDB。在 GDB 中运行的 python 脚本可以使用 GDB 库，通过继承这个库中的 MICommand 类，本文将这个和eBPF处理模块通信的子模块封装成一个 GDB/MI 命令供调试适配器调用，这种封装也使得在后续在VSCode调试插件中适配eBPF处理模块变得容易。

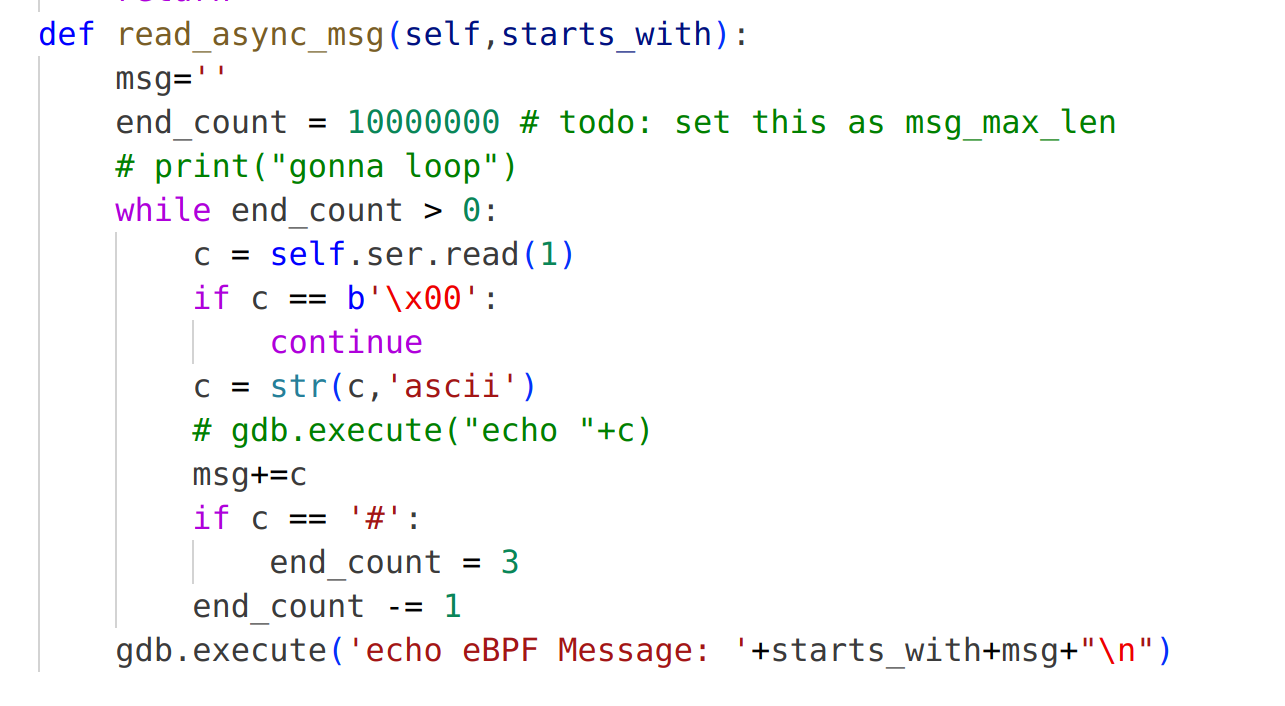


图 3.17 异步消息处理例程的主要代码

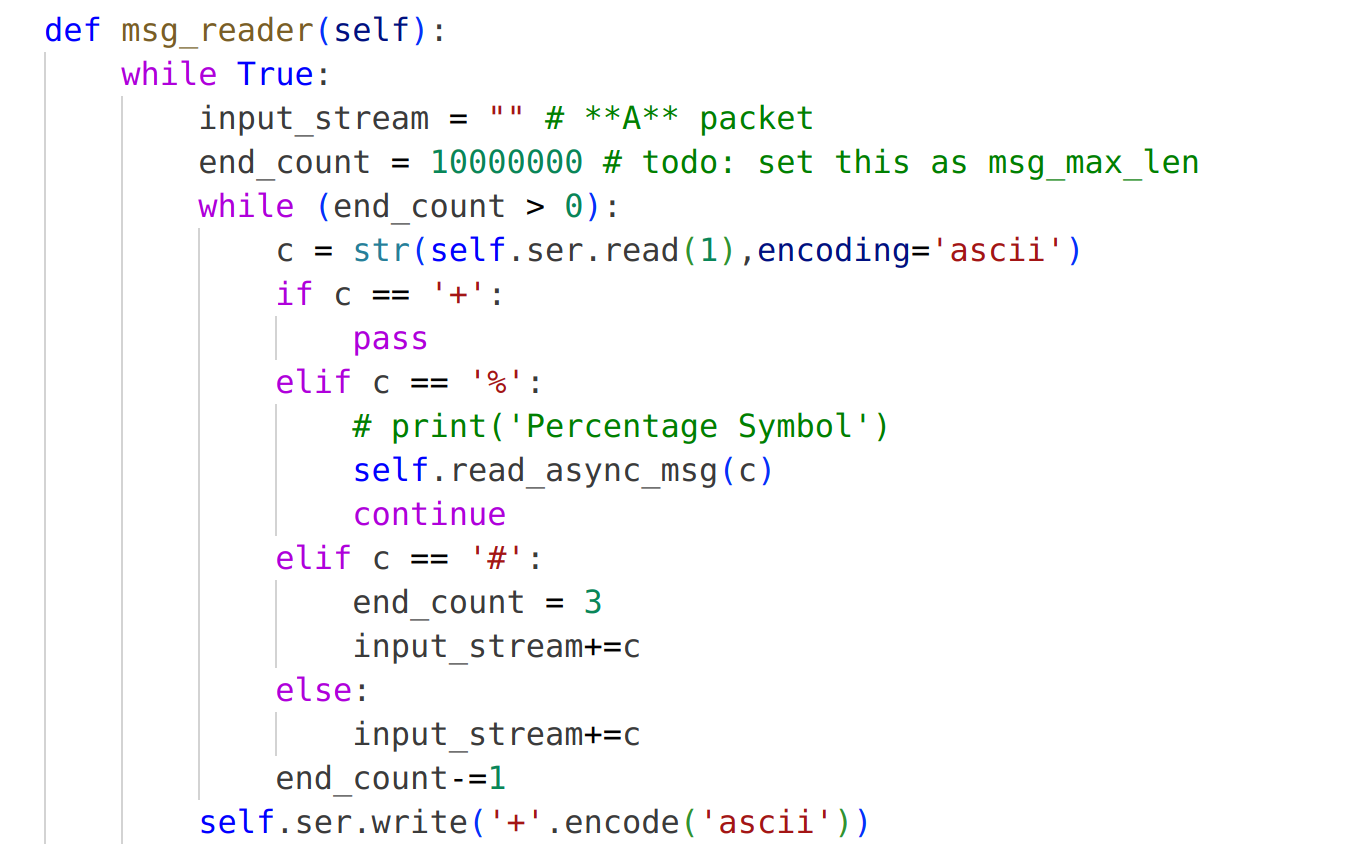


图 3.18 同步消息处理例程的主要代码

经过上述流程eBPF处理模块和GDB可以通过RSP 协议进行正常通信，GDB服务器和 eBPF处理模块与GDB都可以通过RSP协议进行通信。

#### eBPF调试信息的适配

在完成GDB服务器和 eBPF处理模块与GDB的连接以后，需要在调试适配器中适配 eBPF处理模块，调试适配器是用户所使用的VSCode调试器与GDB沟通的桥梁，如2.2.3节所讲述的，是一个独立和调试器通信的进程，。

在 GDB 的层面上，和 eBPF处理模块的所有交互都是通过指定命令进行的。这个命令的规范如下：

// 连接到eBPF 程序的串口.

-side-stub target remote /dev/tty1

// 在某地址设置断点，然后收集寄存器信息

-side-stub break 0x8020xxxx then-get register-info

// 收集函数参数

-side-stub arguments <function-name>

从调试适配器的角度来说，适配主要分两部分，第一个部分是修改用于判断 GDB/MI 消息类别的正则表达式，使得 GDB 传来的 GDB/MI 消息能被正确地处理；第二个部分是，如果调试器端请求执行一些和 eBPF处理模块有关的行为，需要将这些行为翻译成对应的 GDB/MI 消息并发送给 GDB。目前，已经适配两个命令。

最后，需要在用户使用的调试器插件中适配 eBPF处理模块。与调试适配器类似，eBPF处理模块的适配工作也分为两部分。第一部分是添加与eBPF处理模块相关的用户界面，并将这些用户界面的相关事件绑定到调试适配器请求发送函数上。第二部分是解析调试适配器传递的事件和回复信息，并将这些信息更新到相应的用户界面元素上，涉及eBPF处理模块的相关界面如图3.19所示。

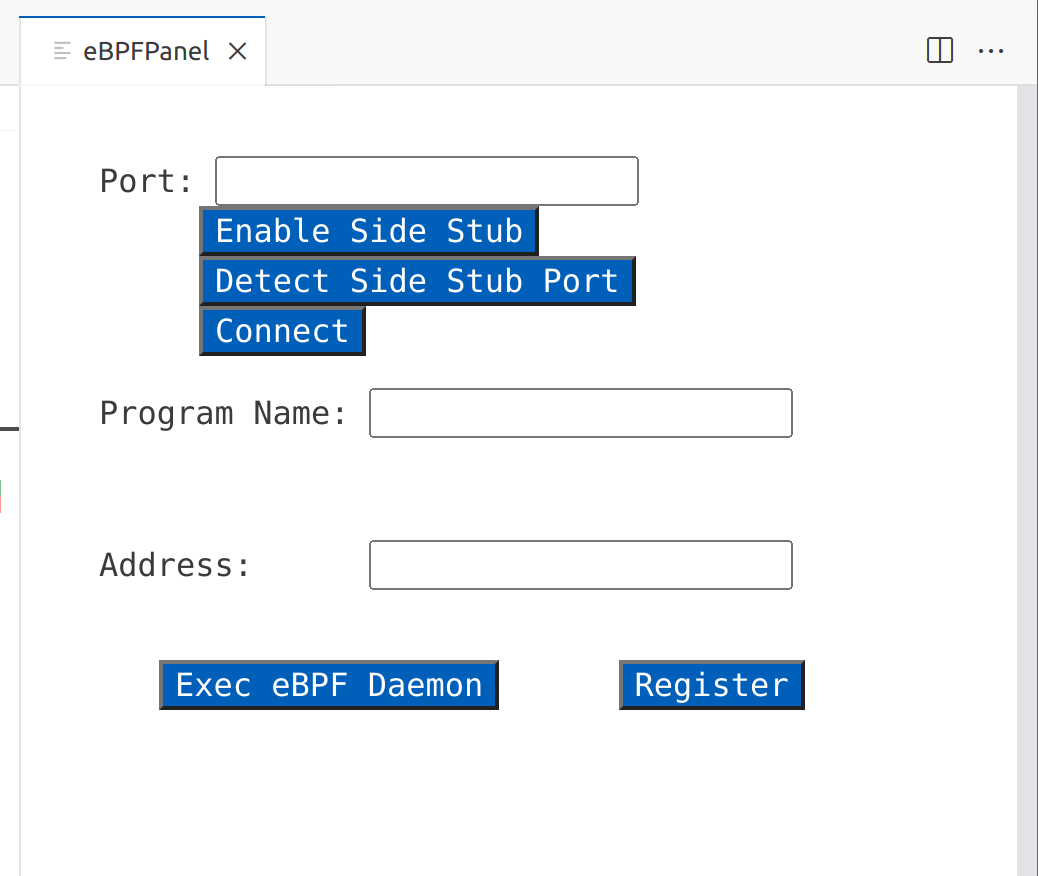


图 3.19 eBPF调试选项

需要注意的是，在用户的使用流程上，GDB服务器和eBPF处理模块的区别在于，eBPF处理模块要提前指定好插桩触发后应执行的行为。二者的使用流程具体如下：

GDB服务器的使用流程：

1. 用户在调试插件中中设置断点。
2. 断点触发，操作系统暂停运行。
3. GDB 等待调试适配器传来的用户的指令，并据此执行信息收集，控制操作系统等行为。

eBPF调试的使用流程：

1. 用户在调试插件中设置断点并提前指定断点触发后的操作。
2. 操作系统中的 eBPF 模块注册相关的 eBPF 程序。
3. 断点触发，eBPF 程序执行这些操作，返回信息，操作系统继续运行。操作系统的状态和 eBPF 程序触发之前保持一致。

经过上述修改，GDB服务器和eBPF处理模块都能够连接到GDB上，同时两者传递过来的调试信息能够与调试适配器和调试插件进行传递和展示。

### 小结

在本小节中，首先，详细描述了在实现基于GDB的多进程调试方法后仍存在的主要问题与挑战，接下来，详细阐述了静态断点调试和动态跟踪结合的方法的实现原理及过程，其中，介绍了如何对 eBPF进行移植，以及移植过程中遇到的主要问题及解决思路与实现；介绍了eBPF处理模块的设计与实现的过程，主要有eBPF处理模块与GDB如何进行通信，以及在调试适配器和调试插件中对eBPF处理模块的适配工作。本小节实现了静态断点调试和动态跟踪结合的调试技术，能够观察到更多内核数据，同时能够动态地操作系统运行过程中的调试信息，能够更大程度地满足用户地调试需求。

## 本章小结

在这一章节中，主要介绍了两种面向Rust操作系统调试方法的设计与实现。首先深入探讨了针对Rust操作系统的多进程调试所面临的核心问题与挑战。接着，详细介绍了基于GDB的多进程调试方法的研究过程。在这个方法中，通过采用断点组管理模块，成功解决了特权级切换导致调试信息丢失的难题；同时，借助设边界监测点的方法，有效应对了调试器中内核态和用户态切换的复杂情况；最后，通过获取进程标识符，顺利获取了下一个要运行进程的符号表和调试信息，成功解决了多个用户进程切换和调试的问题。

随后，对实现基于GDB的多进程调试方法后依然存在的主要问题与挑战进行了详细论述。在此基础上，本文提出了静态断点调试和动态跟踪结合的方法，并阐述了该方法的实现原理及步骤。具体而言，探讨了如何将 eBPF 进行移植，以及在移植过程中所面临的关键问题与应对策略；同时，详细描述了 eBPF 处理模块的设计与实现过程，包括了与 GDB 的通信机制以及在调试适配器和调试插件中对 eBPF 处理模块的适配工作。

通过本章的实践，成功实现了基于GDB的多进程调试方法和静态断点调试和动态跟踪相结合调试方法，不仅能够全面观察内核数据，还能够实时监测操作系统运行过程中的调试信息，极大地满足了用户的调试需求。

# 面向Rust操作系统的调试工具设计与实现

方便的源代码级调试工具，对监测程序运行状态和理解程序的逻辑十分重要，尤其是相对复杂的内核代码以及用户态、内核态的系统调用交互。高效的Rust语言跟踪能力，是Rust操作系统内核开发的必要工具，对基于Rust的操作系统实验和开发很有帮助。Rust操作系统调试的环境配置通常十分复杂，虽然，现在模拟器技术已经广泛应用，被调试操作系统不需要在硬件上运行，但是现有 RISC-V、Rust环境搭建成本高，上手难度大，不利于初学者的内核学习与调试工作。基于本文提出的基于GDB的多进程调试方法及静态断点调试和动态跟踪结合的调试方法，实现了一种基于VSCode以及云服务器的内核源代码远程调试工具：在云服务器中部署QEMU虚拟机并运行Rust操作系统，通过QEMU提供GDB接口与用户本地的网页或安装版VSCode进行连接，提供一种对用户友好的Rust操作系统的在线调试方法

## 整体框架设计

VSCode提供了在线集成环境，基于该在线集成环境，开发了一个在线调试系统。本文通过调试者和被调试内核分离的设计来实现 QEMU 虚拟机或真实系统上的操作系统远程调试。内核在服务器上运行，用户在浏览器使用VSCode插件发送调试相关的请求。

在线调试系统的整体框架如图4.1所示，在远程部署上，被调试操作系统含有待编译的操作系统的源代码，当用户发出编译请求时，服务器中的 Rust 工具链会通过特定的编译参数编译操作系统源代码，生成满足操作系统调试要求的调试信息文件。如果用户接下来发出调试请求， GDB 会加载调试信息文件并连接至 QEMU 的GDB服务器。如果被调试的操作系统中含有eBPF程序，在用户启用了 eBPF 跟踪功能后，相关的eBPF模块会随着GDB的启动而激活，提供更加强大和灵活的动态跟踪调试功能。

图4.1中本地环境中VSCode插件调试和调试扩展前端，用户可以在VSCode调试插件中可以接收到调试过程中产生的相关信息，在扩展前端界面中可以设置断点，观察到运行过程中的一些本地信息，进行一些基本的调试操作，VSCode调试插件则负责处理在扩展界面发送的一些请求，并给用户进行反馈。

调试适配器是运行在服务器中的独立进程，负责处理调试插件发送来的请求。GDB一旦成功加载调试信息文件并连接至 QEMU的GDB服务器，调试适配器进程将启动并开始接收调试插件发送的请求。调试适配器会将请求转换为 GDB 指令发送给 GDB。GDB 在执行完 GDB 指令后将 GDB/MI 格式的信息返回给调试适配器。调试适配器解析后将结果返回给调试插件，最终展示给用户。



图 4.1 系统框架图

## 服务器部分

### 远程连接

OpenVSCode Server 是 VS Code 的一个分支，它在 VSCode 原有的五层架构的基础上增加了服务器层，使其可以提供一个和 VSCode 功能相近的，通过浏览器即可访问的在线集成环境。这个在线集成环境可以和服务器上的开发环境、调试环境通信。

用户可以在在线环境上编辑项目源代码，同时可以远程连接到服务器上的终端。本文在服务器里配置好了 QEMU 虚拟机和 GDB、Rust 工具链。用户可以自行通过终端命令使用 QEMU、GDB 等工具手动调试自己编写的操作系统，也可以通过在线环境中的操作系统调试模块进行更便利的调试。

如果用户选择用操作系统调试模块进行调试，操作系统调试模块做的第一步是编译内核并获取操作系统镜像文件和调试信息文件。接下来本文以 rCore-Tutorial-v3操作系统为例，阐述如何获取这两类文件。

### 编译和加载

在使用默认编译参数的情况下，rCore-Tutorial-v3 编译出的操作系统镜像和调试信息文件难以用于操作系统调试。这是由于几乎所有现代生产软件都是经过优化编译的。优化编译器发出调试信息（例如，DWARF 信息）[54]以支持源代码调试器。同样的，rCore-Tutorial-v3 操作系统基于Rust语言编写，使用Rustc编译器。在默认情况下，Rustc编译器会对代码进行比较激进的优化，例如内联函数，删除大量有助于调试的符号信息。因此，本文需要修改编译参数，以尽量避免编译器的优化操作。

此外，rCore-Tutorial-v3 为了提升性能，修改了用户态程序的链接脚本，使得包含调试信息的DWARF 段在链接时被忽略。这些段对调试用户态程序非常重要，因此本文需要修改链接脚本，移除这种忽略。在修改了链接脚本后，为了让链接脚本生效，需要用 cargo clean 命令清空缓存。

在修改了编译参数、链接脚本后，编译出的可执行文件占用的磁盘空间显著增加，导致 rCore-Tutorial-v3 操作系统的 easy-fs 文件系统无法正常运作，例如在加载文件时崩溃，栈溢出等。因此，本文调整了这个文件系统的 easy-fs-fuse 磁盘打包程序的磁盘大小等参数。此外，由于可执行文件中保留了大量符号信息，用户程序在运行时占用的内存也显著增加，因此需要调整操作系统的用户堆栈大小和内核堆栈大小。

在编译完成后，服务器上的 QEMU 会加载操作系统镜像，并开启一个 GDB服务器。接着，GDB 加载编译时生成的符号信息文件并连接到 QEMU 提供的GDB服务器。如果用户开启了 eBPF 跟踪功能，QEMU中运行的操作系统通过其专属的调试用串口连接到GDB上的 eBPF 调试处理模块。

GDB 与GDB服务器、eBPF处理模块通过 GDB 远程串行协议 (RSP) 进行通信。RSP 是一个通用的、高层级的协议，用于将 GDB 连接到任何远程目标。只要远程目标的体系结构（例如在本项目中是RISC-V）已经被 GDB 支持，并且远程目标实现了支持 RSP 协议的服务器端，那么 GDB 就能够远程连接到该目标。

### 调试适配器进程

调试适配器是一个独立的进程，负责协调在线集成环境和 GDB。在 GDB 准备就绪后，调试适配器进程会启动，并开始监听在线环境中扩展前端模块发送来的各种调试请求。

如图4.2所示，一旦调试适配器接收到一个请求，它就会将请求（调试适配器请求）转换为符合 GDB/MI 接口规范（GDB/MI 是一个基于行的面向机器的 GDB 文本接口，它专门用于支持将调试器用作大型系统的一个小组件的系统的开发。）的文本并发送给 GDB。GDB 在解析、执行完调试适配器发来的命令后，返回符合 GDB/MI 规范的文本信息。调试适配器将 GDB 返回的信息解析后，向扩展前端返回调试适配器协议的回复消息。此外，调试过程中发生的特权级切换、断点触发等事件会通过调试适配器协议的事件消息发送给扩展前端。



图 4.2 调试适配器和GDB、扩展前端的通信机制

## 网页端部分

在用户浏览器上运行的在线调试环境中，一个被称作扩展前端的模块负责和和服务器上的调试适配器通信。它监听调试适配器接收和发出的消息并做出反馈，如更新用户界面、根据用户请求发送请求、响应回复和事件等。扩展前端会解析接收到的回复和事件并将需要的信息转发至用户调试界面。如果用户调试界面向扩展前端传递了某个消息，扩展前端也会将这个消息转换为请求发送给调试适配器。这种传递信息的方式有比较高的自由度。

不同类型的数据的更新策略是不一样的，具体见表4.1：

表 4.1 不同类型数据的更新策略

|  |  |  |
| --- | --- | --- |
| **名称** | **功能** | **更新策略** |
| 寄存器信息 | 显示寄存器名及寄存器值 | 触发断点或暂停时更新 |
| 断点信息 | 显示当前设置的断点以及暂未设置的，缓存的其他内存空间下的断点（比如在内核态时某用户程序的断点） | 触发断点或暂停时更新 |

本文通过 VSCode 提供的几个重要的原生请求接口来展示数据，比如variablesRequest，其功能是在在线调试窗口左侧的调试标签页中，顶部VARIABLES 标签栏里展示变量的名字与值。每当代码调试因触发断点等原因发生了暂停，在线调试环境都会自动发送一个变量请求向调试适配器请求变量数据。本文添加了一个自定义的 variablesRequest 获取到寄存器数据，从而在更贴近原生界面的树视图里展示寄存器数据，如图4.3所示。

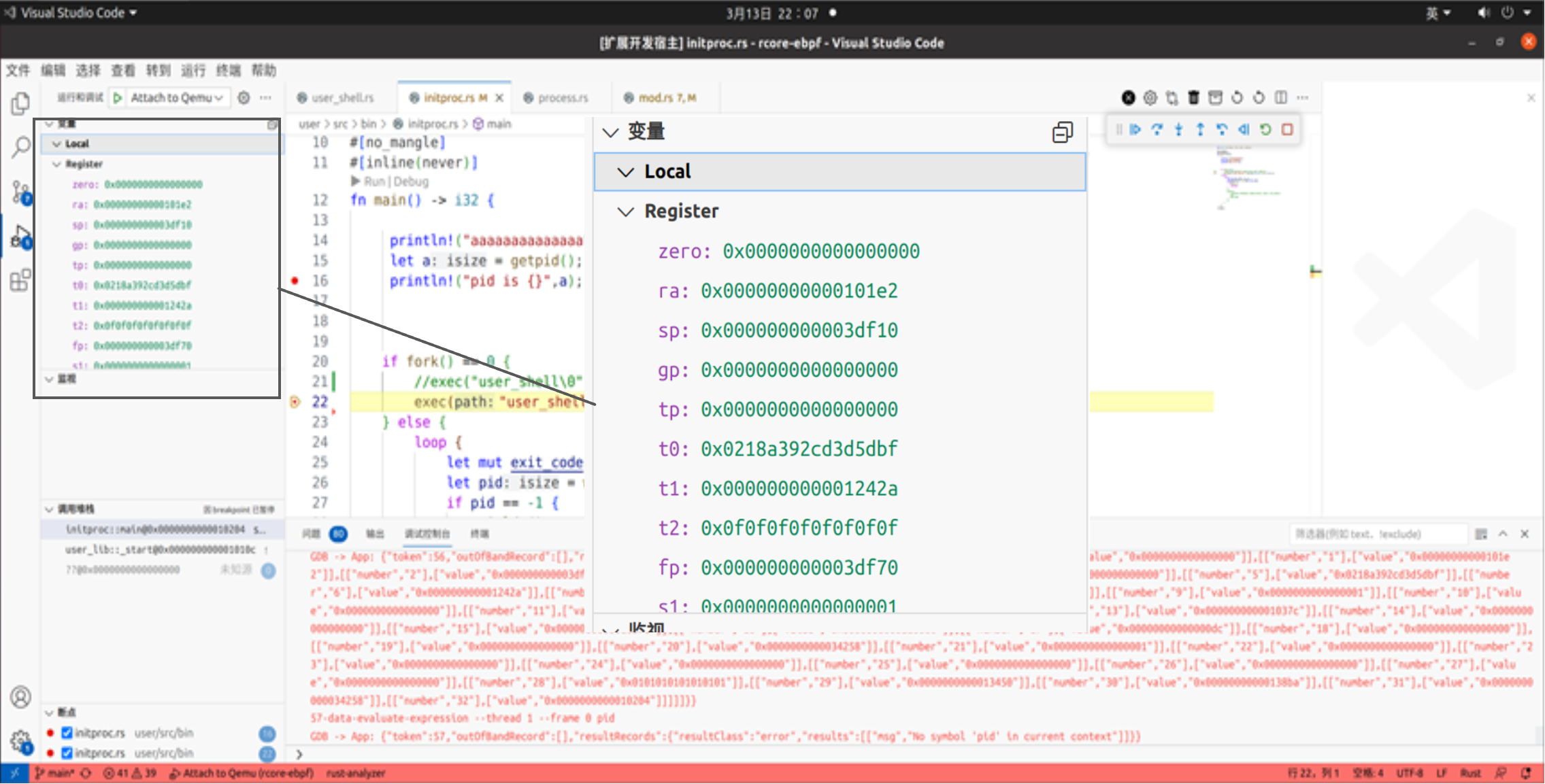


图 4.3 通过树视图展示数据

提供了功能按钮如图4.4.所示：

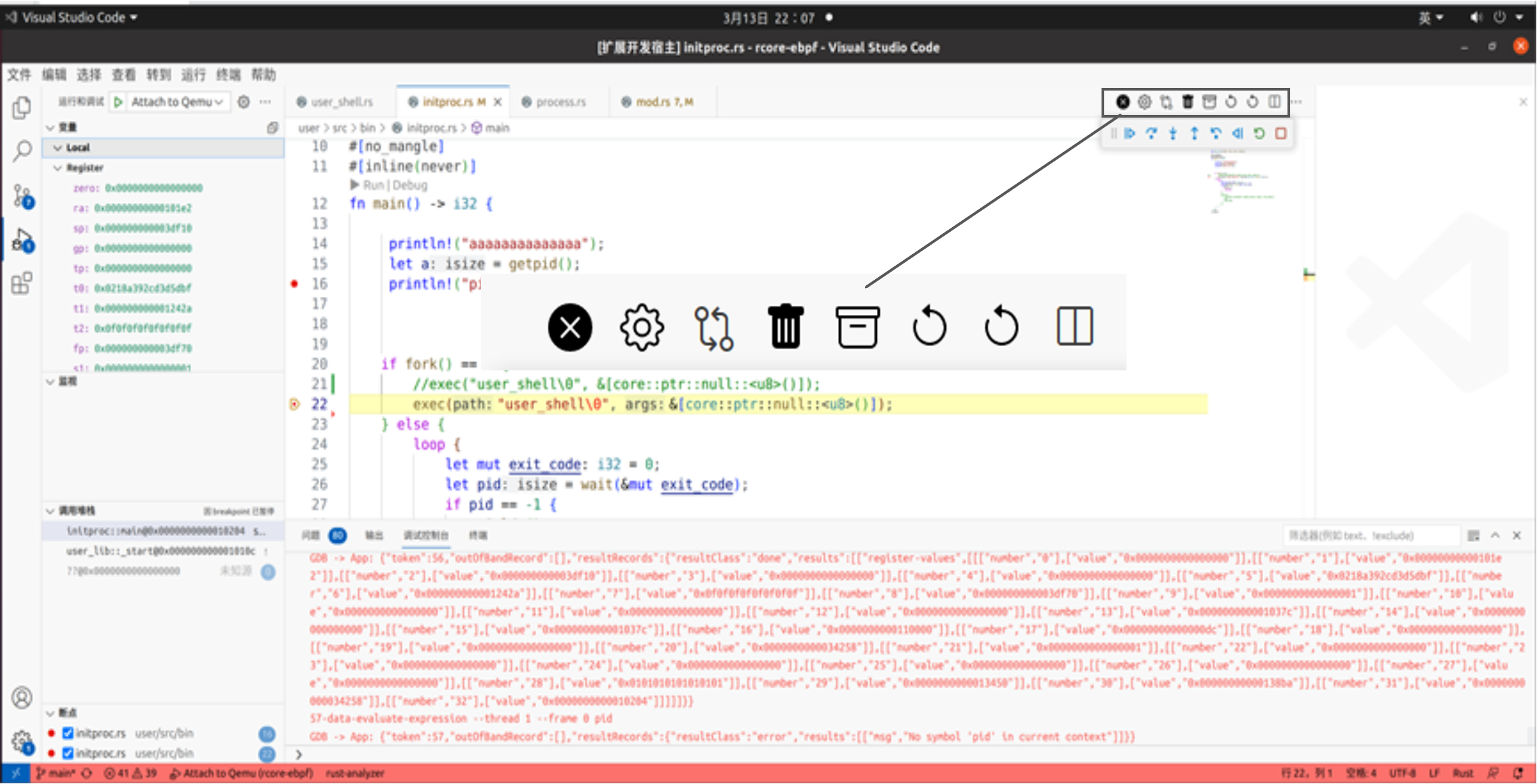


图 4.4 原生样式的命令按钮

按钮的功能如表4.2所示。

表 4.2 调试界面按钮的名称及功能

|  |  |
| --- | --- |
| **名称** | **功能** |
| gotokernel | 在用户态设置内核态出入口断点，从用户态重新进入内核态 |
| setKernelInBreakpoints | 设置用户态到内核态的边界处的断点 |
| setKernelOutBreakpoints | 设置内核态到用户态的边界处断点 |
| removeAllCliBreakpoints | 重置按钮。清空编辑器，调试适配器， GDB中所有断点信息 |
| disableCurrentSpaceBreakpoints | 令GDB清除当前设置的断点且不更改调试适配器中的断点信息 |
| updateAllSpacesBreakpointsInfo | 手动更新断点信息表格 |

此外本文还支持了VSCode自带的继续、单步等常见的调试功能按钮，如图4.5所示：

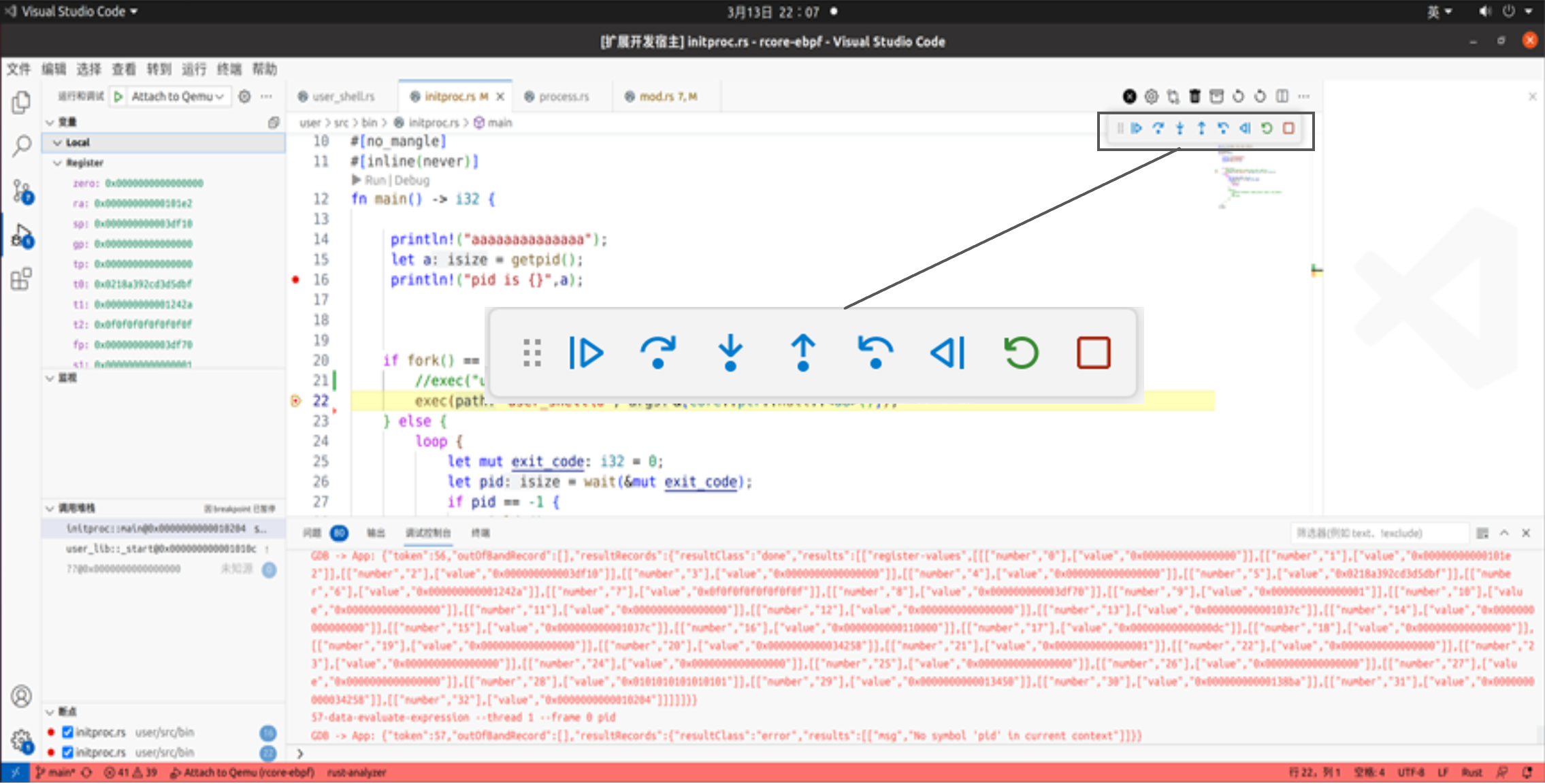


图 4.5 调试功能按钮

## 案例研究

接下来本文同时运用GDB断点和eBPF调试功能，用本文的调试器调试 rCore-Tutorial 自带的 http 服务器，从中可以看出 GDB加eBPF 带来的调试上的方便。

### Bug描述

tcp-simplehttp是 rCore-Tutorial 自带的一个简单的 HTTP 服务器。服务器启动之后，在火狐浏览器访问对应的网页地址即可获得服务器返回的静态页面。但是，如果本文在浏览器里打开多个标签页，每打开一个标签页，就在这个标签页里访问服务器的URL，就会发现一个奇怪的现象：一部分标签页成功显示出了网页，另一部分则一直在加载中，始终无法显示网页。而且，加载成功的标签页和加载失败的标签页是交替出现的，如图4.6所示。



图 4.6 tcp\_simplehttp服务器程序未返回所有请求的网页

作为对比，本文用同样的方式重复打开全国大学生计算机系统能力大赛的官网，结果是所有标签页都正常地打开了如图4.7所示。



图 4.7 全国大学生计算机系统能力大赛的官网成功返回所有请求的网页

接下来，本文尝试在操作系统调试器的帮助下找到出错的原因并修复这个错误。

### 代码简述

在开始使用调试器之前，查看了这个服务器的源代码，从而大致了解了它的工作流程：当服务器启动，进入到main函数后，main函数就会调用 listen() 库函数，在80端口开始监听。如果接收到客户端的连接，就调用handle\_tcp\_client 函数处理来自客户端的请求。handle\_tcp\_client 函数会先读取请求并检查是否为有效的 HTTP GET 请求，如果是的话，就从请求中提取出网页地址，并返回网页地址对应的网页。此外，如果用户访问的是页面，服务器会在返回一个表示服务器关闭的网页后终止连接并关闭服务器自身。

### 调试过程

首先，查看服务器是否接收到了所有的 HTTP 请求。因此本文在服务器的函数设置一个eBPF 断点。这个 eBPF 断点会返回当前的所有寄存器值。从而获得accept 函数的返回值和参数。

设置断点后，本文打开服务器，重复访问同一个网址。发现了两个异常现象：

1. 在浏览器尚未访问网址时， accept() 函数就被调用了一次。
2. 浏览器打开六个标签页（每一个标签页都向服务器请求同一个网址），只有四个标签页正确显示出了内容。accept()函数在这段时间内只触发了四次断点，正常状况下应该是六个断点才对。

这个初步的尝试显示，问题可能出在内核没有成功接收到所有的 HTTP 请求，或者内核接收到了所有的请求，却没有全部传送给应用程序。

为了确认具体的出错位置，本文从内核网络栈的代码到系统调用,再到用户态程序上设置了多个内核和应用程序的eBPF断点，看看是哪个环节出了问题，跟踪的内核函数如表4.3所示。

表 4.3 eBPF跟踪的函数及地址

|  |  |
| --- | --- |
| 函数名 | 地址 |
| receive | 000000008021c0ca |
| net\_interrupt\_handler | 0000000080212c24 |
| sys\_accept | 0000000080216e5a |

在sys\_accept() 之上还有syscall() 函数。但是由于用 eBPF 跟踪syscall() 函数会造成死循环（原因是 eBPF 系统调用也会调用这个函数）本文用GDB跟踪 syscall() 函数。跟踪的用户函数如表4.4所示：

表 4.4 GDB跟踪的函数及地址

|  |  |
| --- | --- |
| 函数名 | 地址 |
| listen | 0x000109fe |
| accept | 0x00010a1c |

设置完断点后，仍然重复访问网页六次，发现六个网页只正常打开了四个，eBPF跟踪点返回信息只有四次，如图4.8所示。回到 VSCode 查看调试信息和GDB断点，发现 listen 被调用一次（这是正常的），其他函数都被调用四次，那说明问题并不出在网络协议栈函数的调用流程上。

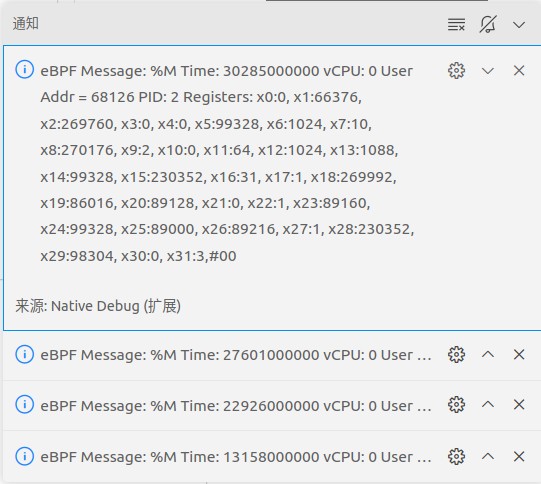


图 4.8 eBPF检查点触发，弹出调试信息

于是，怀疑另一个原因可能是本文过于频繁地访问网页，而这个服务器程序在处理过于频繁的请求时会出错。为了验证这个猜想，本文在sys\_accept、write函数设置GDB断点，这样每次处理accept时操作系统都会停下来，如图4.9所示，减缓了服务器程序的处理速度。然而，同样的异常现象（六个网页只打开四个）还是出现。

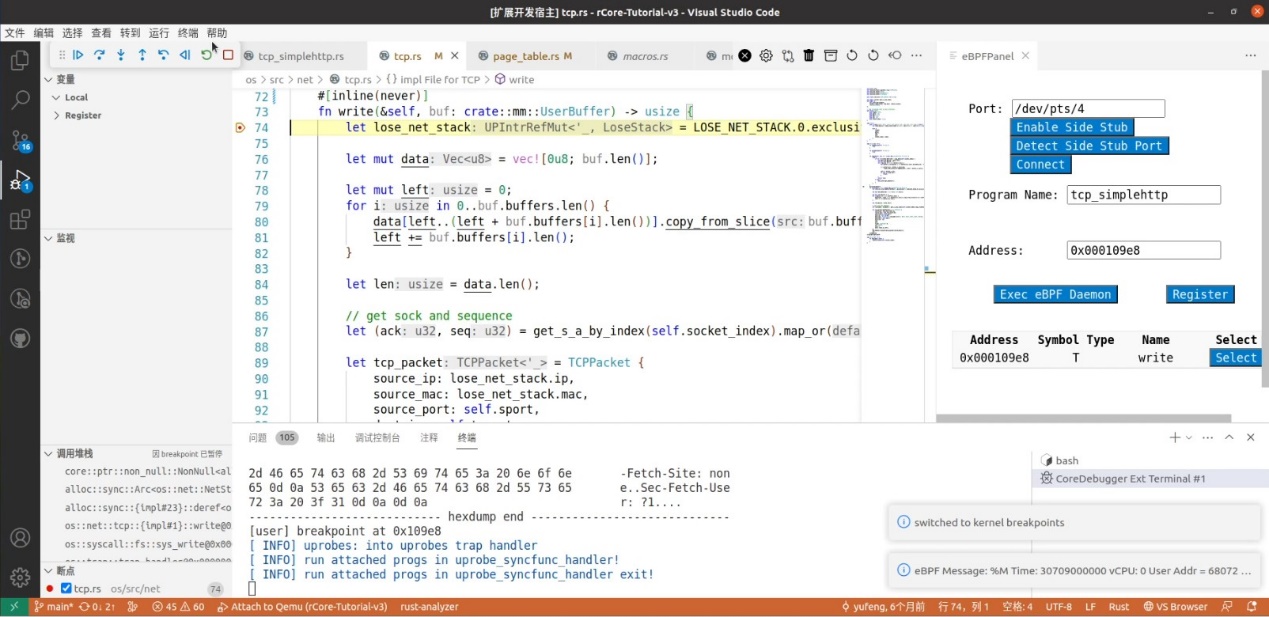


图 4.9 `write`函数的GDB断点触发

之后又考虑到，如果流程是正确的，就需要考虑发送的内容是否有错误。于是，本文检查了服务器返回信息的函数，终于发现了出错的原因：在服务器发送的http response中， Connection:Close 中的字母t被遗漏了，变成了Connecion:Close ，如图4.10所示，导致前一个连接没有被正确关闭，因此后一个连接无法成功建立。

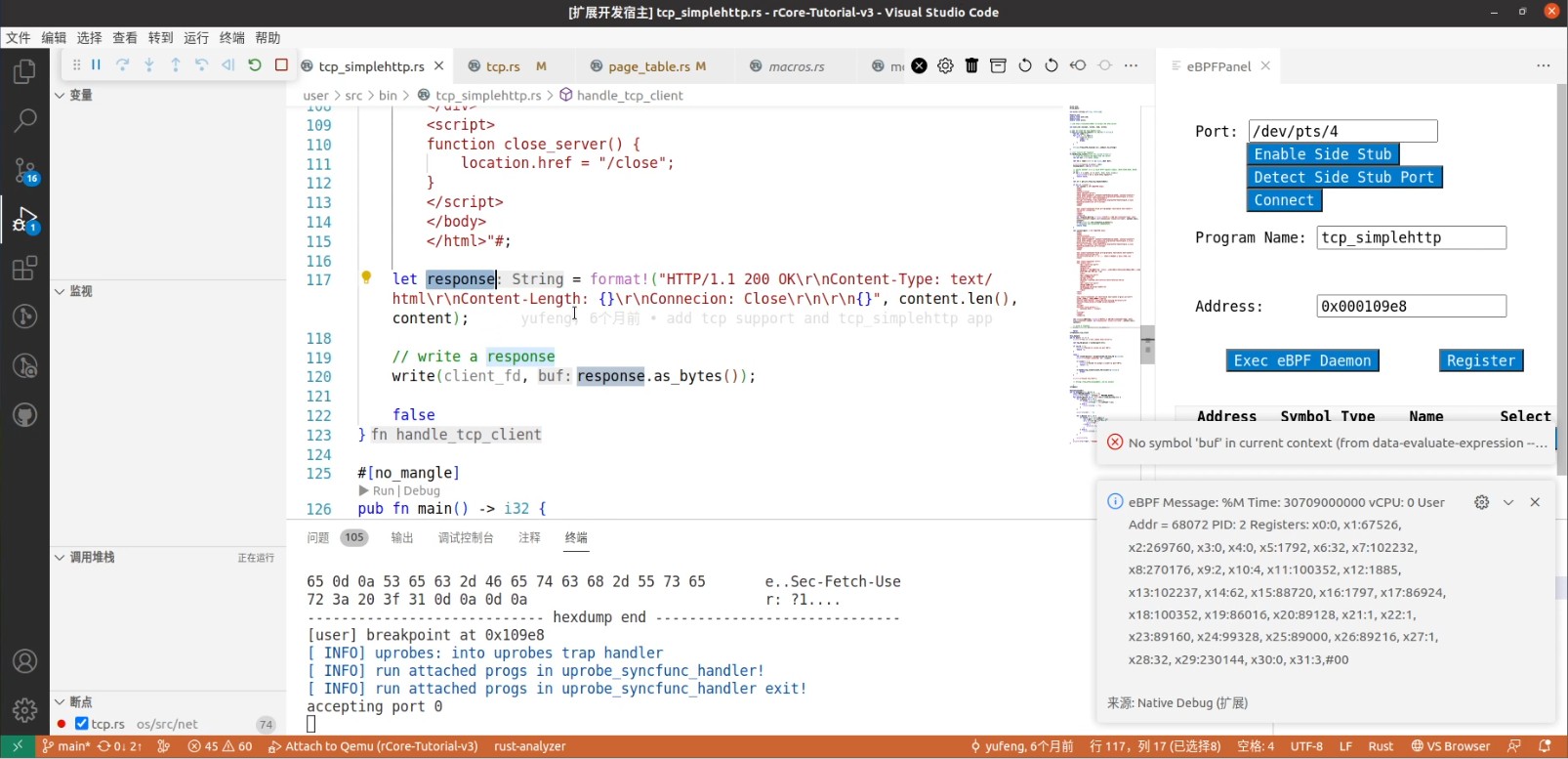


图 4.10 出错的代码

这样，结合 eBPF 和 GDB 两种调试手段，本文根据调试器提供的线索快速定位并修复了这个Bug。除了这一处代码，在整个调试的过程中，本文不需要出于调试目的而改动任何其他的源代码。

## 本章小结

本章主要介绍了支持Rust语言的源代码级操作系统调试工具，基于VSCode开发了一个调试插件，在云服务器中部署QEMU虚拟机并运行Rust操作系统，由GDB服务器和eBPF处理模块进行收集调试信息，处理命令等工作，通过QEMU提供GDB接口与用户本地的网页或安装版VSCode进行交互。在各个小节中，分别介绍了用户本地调试插件架构和服务器中的调试架构，以及两者之间如何进行通信，并展示了调试器的功能及调试界面。在最后案例研究中，展示了使用调试器调试操作系统漏洞的整个过程，通过调试器逐渐缩小代码漏洞的范围，最终找到并修复了代码漏洞。

面向Rust操作系统的调试工具，可以促进Rust语言在操作系统领域的应用和发展。Rust作为一种安全性和性能兼备的系统编程语言，其在操作系统开发中具有广阔的前景。而完善的调试工具可以提升Rust操作系统的开发体验，并提升Rust操作系统的开发效率和质量，解决调试中的挑战，促进Rust语言在操作系统领域的广泛应用和发展。

# 总结与展望

## 总结

近年来开源指令集RISC-V以及基于Rust编程语言的开源操作系统，在全世界范围内获得了广泛的关注，但是Rust操作系统的调试方法各个方面都不够成熟，而且不同于通用应用程序代码开发与调试，操作系统源代码运行状态多、工作逻辑复杂，进一步增加了调试难度。在目前的调试领域中，并没有一款在线调试器能够很好地支持使用Rust语言编写的操作系统的调试。本文提出了两种操作系统调试器的调试方法，并开发了一款源代码级Rust操作系统调试器。本文的主要工作包括了以下几个方面：

（1）在基于GDB的多进程调试方法中，介绍对调试器添加多进程调试功能的具体实现。首先，本文建立了一个断点组管理模块，该模块能够缓存用户设置的断点，待被调试的操作系统运行到相应地址空间时再激活断点，从而解决了跨特权级的源代码断点设置冲突的问题。其次，本文引入了边界检测点，以识别被调试操作系统当前的特权级，并据此切换不同的符号表，使得用户能够进行跨特权级的调试。最后，在3.4小节中，阐述了如何实现多个用户进程进行切换和调试，本文通过内核代码获取下一个要运行的进程标识符，并将其传递给调试器，使调试器能够准确获取新进程的调试信息，从而实现了对多个用户进程的调试。通过这些步骤，本文成功实现了对调试器多进程调试功能的扩展，并为用户提供了更加灵活和全面的调试工具。

（2）在一种静态断点调试和动态跟踪调试方法中，旨在探讨在rCore-Tutorial-v3操作系统中进行eBPF处理模块设计与实现的过程中所面临的挑战以及解决方法。首先，本文在rCore-Tutorial-v3中移植了eBPF模块，以便在被调试的操作系统中收集调试信息。在确保rCore-Tutorial-v3支持eBPF程序运行后，本文将重点放在了将静态断点调试和动态跟踪结合起来的工作上。为此，本文采用了通过串口将eBPF模块收集到的调试信息传送到GDB的方法，这一过程中使用了与静态断点调试相同的RSP协议。接着，本文对来自eBPF模块的信息进行了适配，以便与调试适配器进行交互。最后，本文将eBPF模块传递的信息与在线集成开发环境（IDE）进行适配，通过VSCode插件将调试信息展示给用户。通过这些步骤，本文成功地实现了在rCore-Tutorial-v3操作系统中支持静态断点调试和动态跟踪调试的目标。

（3）在Rust操作系统调试器的设计与实现方面，本课题基于VSCode的在线集成开发环境和调试架构，开发了支持Rust语言的源代码级操作系统调试工具，使用者只需要在在线VSCode中加载插件，就可以对云服务器中的Rust操作系统进行调试，调试器能够支持使用GDB对Rust操作系统进行静态断点调试，和使用eBPF和kprobe对Rust操作系统进行动态跟踪调试。使用者可以在在线调试器中进行跨特权级的断点设置，查看各种本地变量，查看寄存器信息，查看系统调用的函数参数等。并在5.4小节中展示了一个调试案例，使用上述提到的各种调试方法缩小了可能出错代码的范围，通过分析最终找到了问题代码。

## 展望

目前能够支持Rust操作系统的调试方法和调试器已经取得了一定的成果，但是，为了更好地满足使用的调试需求，在现有工作的基础上，本文还希望对调试工具进行下列改进：

1. 将eBPF处理模块和GDB服务器更紧密地结合使用，例如由eBPF处理模块解析复杂的特权级指令捕获条件，由GDB服务器进行跟踪；
2. 增强信息获取的能力，目前本文已经将kprobe跟踪工具添加到了调试器中，但是它收集到的信息更多的是内核态调试信息，uprobe可以收集到用户态的调试信息，添加以后可以使调试器的信息获取能力更加强大；
3. 基于真实系统(FPGA 或 RISC-V 开发板)的远程实验与调试系统，可以在真实环境中进行实验和调试，观察到更加真实的数据；
4. 适配更多的操作系统，目前系统只支持单一操作系统的调试，对要调试的操作系统限制较高，本文将让系统能够支持更多操作系统的调试，并提供简单的配置手段，以此提高系统的易用性。

参考文献

1. Lankes S, Breitbart J, Pickartz S. Exploring rust for unikernel development[C]//Proceedings of the 10th Workshop on Programming Languages and Operating Systems. 2019: 8-15.
2. Balasubramanian A, Baranowski M S, Burtsev A, et al. System programming in rust: Beyond safety[C]//Proceedings of the 16th workshop on hot topics in operating systems. 2017: 156-161.
3. Pearce D J. A lightweight formalism for reference lifetimes and borrowing in Rust[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2021, 43(1): 1-73.
4. Hu S, Hua B, Wang Y. Comprehensiveness, Automation and Lifecycle: A New Perspective for Rust Security[C]//2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2022: 982-991.
5. Li Z, Wang J, Sun M, et al. Detecting cross-language memory management issues in Rust[C]//European Symposium on Research in Computer Security. Cham: Springer Nature Switzerland, 2022: 680-700.
6. Zhang Y, Kundu A, Portokalidis G, et al. On the dual nature of necessity in use of Rust unsafe code[C]//Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2023: 2032-2037.
7. Crichton W, Gray G, Krishnamurthi S. A Grounded Conceptual Model for Ownership Types in Rust[J]. Proceedings of the ACM on Programming Languages, 2023, 7(OOPSLA2): 1224-1252.
8. Chen S F, Wu Y S. Linux kernel module development with rust[C]//2022 IEEE Conference on Dependable and Secure Computing (DSC). IEEE, 2022: 1-2.
9. Liang Y, Wang L, Li S, et al. Rustpi: A Rust-powered Reliable Micro-kernel Operating System[C]//2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2021: 272-273.
10. Cui E, Li T, Wei Q. Risc-v instruction set architecture extensions: A survey[J]. IEEE Access, 2023, 11: 24696-24711.
11. Mezger B W, Santos D A, Dilillo L, et al. A survey of the RISC-V architecture software support[J]. IEEE Access, 2022, 10: 51394-51411.
12. Anders J, Andreu P, Becker B, et al. A survey of recent developments in testability, safety and security of risc-v processors[C]//2023 IEEE European Test Symposium (ETS). IEEE, 2023: 1-10.
13. Kalapothas S, Galetakis M, Flamis G, et al. A survey on risc-v-based machine learning ecosystem[J]. Information, 2023, 14(2): 64.
14. Wali I, Sánchez-Macián A, Ramos A, et al. Analyzing the impact of the operating system on the reliability of a RISC-V FPGA implementation[C]//2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS). IEEE, 2020: 1-4.
15. Ji J H, Woo G, Park H B, et al. Design and implementation of retargetable software debugger based on gdb[C]//2008 Third International Conference on Convergence and Hybrid Information Technology. IEEE, 2008, 1: 737-740.
16. Chatterjee N, Majumdar S, Sahoo S R, et al. Debugging multi-threaded applications using pin-augmented gdb (pgdb)[C]//International conference on software engineering research and practice (SERP). Springer. 2015: 109-115.
17. 王涛,秦宵宵,徐学政,等.基于QEMU的高效指令追踪技术[J].计算机系统应用,2023,32(11):3-10.DOI:10.15888/j.cnki.csa.009330.
18. 蒋龙. 基于GDB的嵌入式多任务调试器的设计实现与集成[D].浙江:浙江大学,2014.-
19. Mihajlović B, Žilić Ž, Gross W J. Dynamically instrumenting the QEMU emulator for Linux process trace generation with the GDB debugger[J]. ACM Transactions on Embedded Computing Systems (TECS), 2014, 13(5s): 1-18.
20. Rong X. Design and Implementation of Operating System in Distributed Computer System Based on Virtual Machine[C]//2020 International Conference on Advance in Ambient Computing and Intelligence (ICAACI). IEEE, 2020: 94-97.
21. Zhang L, Kong X. Embedded trusted computing environment build based on QEMU virtual machine architecture[C]//2014 Seventh International Symposium on Computational Intelligence and Design. IEEE, 2014, 1: 193-196.
22. Díaz E, Mateos R, Bueno E J, et al. Enabling parallelized-QEMU for hardware/software co-simulation virtual platforms[J]. Electronics, 2021, 10(6): 759.
23. Mohamed M H N, Wang X, Ravindran B. Understanding the Security of Linux eBPF Subsystem[C]//Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems. 2023: 87-92.
24. Gunturu M, Aluguri R. Towards Performance Evaluation and Future Applications of eBPF[J]. 2024.
25. [Miano S, Chen X, Basat R B, et al. Fast in-kernel traffic sketching in EBPF[J]. ACM SIGCOMM Computer Communication Review, 2023, 53(1): 3-13..
26. Caviglione L, Mazurczyk W, Repetto M, et al. Kernel-level tracing for detecting stegomalware and covert channels in Linux environments[J]. Computer Networks, 2021, 191: 108010..
27. Krishnakumar R. Kernel korner: kprobes-a kernel debugger[J]. Linux Journal, 2005, 2005(133): 11.
28. Sun J, Li Z, Zhang X, et al. The study of data collecting based on kprobe[C]//2011 Fourth International Symposium on Computational Intelligence and Design. IEEE, 2011, 2: 35-38.
29. Fan H, Li K, Li X, et al. CoVSCode: a novel real-time collaborative programming environment for lightweight IDE[J]. Applied Sciences, 2019, 9(21): 4642.
30. 周毅,程石林,陆暤冉,等. 一种Linux内核调试系统及方法[P]. 江苏省：CN106227653B,2018-10-30.
31. 卓维晨. 一种基于日志打点的Linux内核调试方法[P]. 山东省：CN104657277B,2017-12-22.
32. Desnoyers M, Dagenais M R. Synchronization for fast and reentrant operating system kernel tracing[J]. Software: Practice and Experience, 2010, 40(12): 1053-1072.
33. Gebai M, Dagenais M R. Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead[J]. ACM Computing Surveys (CSUR), 2018, 51(2): 1-33.
34. Lin Z, Chen Y, Wu Y, et al. GREBE: Unveiling exploitation potential for Linux kernel bugs[C]//2022 IEEE Symposium on Security and Privacy (SP). IEEE, 2022: 2078-2095..
35. Ge X, Niu B, Cui W. Reverse debugging of kernel failures in deployed systems[C]//2020 USENIX Annual Technical Conference (USENIX ATC 20). 2020: 281-292.
36. [36] BISSYANDÉ T F, RÉVEILLÈRE L, LAWALL J L, 等. Ahead of time static analysis for automatic generation of debugging interfaces to the Linux kernel[J/OL]. Automated Software Engineering, 2016, 23(1): 3-41[2024-03-23]. https://doi.org/10.1007/S10515-014-0152-4.
37. Bissyandé T F, Réveillère L, Lawall J L, et al. Ahead of time static analysis for automatic generation of debugging interfaces to the linux kernel[J]. Automated Software Engineering, 2016, 23: 3-41.
38. 杨杰,成新民.一种基于虚拟机的操作系统内核调试方法[J].科技风,2010(06):170-171.DOI:10.19392/j.cnki.1671-7341.2010.06.149.
39. Takeuchi T. OS debugging method using a lightweight virtual machine monitor[C]//Design, Automation and Test in Europe. IEEE, 2005: 1058-1059.
40. Sato M, Taniguchi H, Nakamura R. Virtual machine monitor-based hiding method for access to debug registers[C]//2020 Eighth International Symposium on Computing and Networking (CANDAR). IEEE, 2020: 209-214.
41. Sato M, Taniguchi H, Yamauchi T. Design and implementation of hiding method for file manipulation of essential services by system call proxy using virtual machine monitor[J]. International Journal of Space-Based and Situated Computing, 2019, 9(1): 1-10.
42. Wenyu C, Dongpu H, Dongcheng T, et al. A model of remote debugger supporting multiple types of connection[C]//2011 International Conference on Electronics, Communications and Control (ICECC). IEEE, 2011: 642-645.
43. 李红卫,李翠萍,韩红宇.kgdb调试Linux内核的剖析与改进[J].微型机与应用,2004(10):7-10.
44. Chang T H, Hou S C, Huang J. A unified GDB-based source-transaction level SW/HW co-debugging[C]//2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS). IEEE, 2016: 506-509.
45. 苏嘉玮关宁,刘强,孙国飞,王欢. 基于RISC-V微处理器的软硬件调试方法研究与实现[J]. 航天标准化, 2020(02): 12-15.
46. Lakicevic N. Runtime Verification with Linux eBPF[D]. Norway: University of Oslo, 2023.
47. Ramachandran G S, McDonald L, Jurdak R. FUSE: Fault Diagnosis and Suppression with eBPF for Microservices[C]//International Conference on Service-Oriented Computing. Cham: Springer Nature Switzerland, 2023: 243-257.
48. Dong X, Liu Z. Multi-dimensional detection of Linux network congestion based on eBPF[C]//2022 14th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA). IEEE, 2022: 925-930.
49. Jiang Y, Yang Y, Xiao T, et al. Kernel data race detection using debug register in Linux[C]//2014 IEEE COOL Chips XVII. IEEE, 2014: 1-3.
50. Weng T, Yang W, Yu G, et al. Kmon: An in-kernel transparent monitoring system for microservice systems with ebpf[C]//2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence). IEEE, 2021: 25-30.
51. Zavarella T D. A methodology for using eBPF to efficiently monitor network behavior in Linux Kubernetes clusters[D]. America: Massachusetts Institute of Technology, 2022.
52. Li L, Wang C. Dynamic analysis and debugging of binary code for security applications[C]//International Conference on Runtime Verification. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013: 403-423.
53. Shi Q, Chen L J, Qiao Z. A Measuring Tool for Interrupt Latency Based on Linux[C]//Journal of Physics: Conference Series. IOP Publishing, 2023, 2476(1): 012050.
54. Li Y, Ding S, Zhang Q, et al. Debug information validation for optimized code[C]//Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. 2020: 1052-1065.