北京工商大学

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

工程领域 ： 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 ： 计算机与人工智能学院

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

RISC-V 与 Rust 操作系统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 Rust 操作系统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。操作系统与普通应用程序不同，在操作系统中为了分配和管理资源，设置了不同的特权级，通常有用户态和内核态，目前的源代码及调试器都无法感知特权级切换，无法进行不同特权级下多个进程的调试。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，并实现了能够支持 Rust 语言的源代码级操作系统调试工具。本文的主要工作包括以下内容：

由于操作系统特有的特权级机制，其中不同特权级会对应不同的符号表，而符号表保存了调试器所需要的调试信息。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。本文提出了一种基于 GDB 的多进程调试方法，在该方法中，设置了断点组管理模块，缓存不同进程地址空间的调试信息，解决了切换特权级调试信息丢失的问题；其次设置了边界检测点，使调试器能够感知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名称，调试器根据进程名称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，实现了对多个用户进程的调试。

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，本文提出了一种静态断点调试和动态跟踪结合的调试方法。本文实现了运行在操作系统中的 eBPF 与 GDB 的连接与交互，可以在运行时对内核进行监控，捕获有关函数执行等更多的信息。该方法支持 GDB 的静态断点调试，同时也支持使用 eBPF 对运行中的操作系统进行动态跟踪，可以对操作系统中运行的多个进程进行调试并获取更多函数信息。

本文提出了面向 Rust 操作系统的调试工具，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，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，找到代码漏洞。

关键词：Rust 操作系统；操作系统调试器；多进程调试；GDB；eBPF

ABSTRACT

RISC-V and the Rust operating system are in the early stages of technological development. Researchers or engineers lack an adapted source code-level debugging tool when developing or debugging the Rust operating system, making the development process difficult and cumbersome. The operating system is different from ordinary applications. In order to allocate and manage resources, different privilege levels are set up in the operating system. There are usually user mode and kernel mode. The current source code and debugger cannot sense the privilege level switching and cannot perform different privilege levels. Debugging of multiple processes under privileged levels. Against this background, this thesis 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. The main work of this thesis includes the following contents:

Due to the unique privilege level mechanism of the operating system, different privilege levels correspond to different symbol tables, and the symbol table stores the debugging information required by the debugger. During the operation of the operating system, privilege levels are frequently switched, resulting in the loss of debugging information. Currently, existing debuggers are unable to perform breakpoint debugging across privilege levels. This thesis proposes a multi-process debugging method based on GDB. In this method, a breakpoint group management module is set up to cache debugging information in different process address spaces, which solves the problem of losing debugging information when switching privilege levels; secondly, boundary detection is set up point, so that the debugger can sense which privilege level the debugged operating system is currently running in, and switch different symbol tables according to the privilege level, allowing users to debug across privilege levels; finally, in order to support the debugging of multiple user processes , obtain the name of the next process to be run in the kernel code, the debugger determines the next symbol table to be switched based on the process name, obtains the debugging information of the new process, debugs the new process, and realizes the debugging of multiple user processes debugging.

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. This thesis proposes a debugging method that combines static breakpoint

debugging and dynamic tracing. This thesis realizes the connection and interaction between eBPF and GDB running in the operating system, which can monitor the kernel at runtime and capture more information about function execution. This method supports GDB's static breakpoint debugging, and also supports the use of eBPF to dynamically track the running operating system. It can debug multiple processes running in the operating system and obtain more function information.

This thesis proposes a debugging tool for the Rust operating system, develops a debugger for the source code-level operating system that can support 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. The debugger can gradually reduce the scope of code vulnerabilities and find the code without changing other source codes. loopholes.

KEY WORDS: Rust operating system, operating system debugger, multi-process debugging, GDB, eBPF.

目录

第1章 绪论

1.1 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。

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，大多数调试器都无法感知到特权级状态的变化，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，无法感知操作系统中的特权级切换，所以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[3]，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感知被调试操作系统特权级切换的问题，

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，使调试器能够对运行中的操作系统进行实时监测，为复杂操作系统的开发者提供一款更强大的调试工具。

1.2 研究内容和主要工作

本研究旨在弥补目前操作系统源代码级调试器的不足，解决源代码级调试器无法感知特权级切换和不同特权级下多进程切换等问题，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。针对源代码级调试器无法感知被调试操作系统特权级切换的问题，以及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，使调试器能够感知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，调试器根据进程名称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 静态断点调试会暂停被调试操作系统运行，无法动态获取被调试操作系统调试信息等问题，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，设计了 eBPF 处理模块，它运行在被调试操作系统中与 GDB 进行通信，负责接受使用者的调试需求并解析，从而获取 eBPF 的调试信息，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并动态获取调试信息。

（3）面向 Rust 操作系统的调试工具设计与实现。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，展示了一个调试案例，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。

1.3 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的发展现状；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：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，和调试器进行多进程切换和调试的过程及实现。接下来介绍了第二种调试方法：一种静态断点和动态跟踪结合的方法，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，为了实现两种调试技术的结合，本文对 eBPF 与 GDB 调试信息的数据流整合进行了介绍，主要有 eBPF 处理模块的设计与实现，eBPF 处理模块与 GDB 之间的 RSP 协议的实现以及 eBPF 调试信息的适配。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，展示了一个调试案例，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。

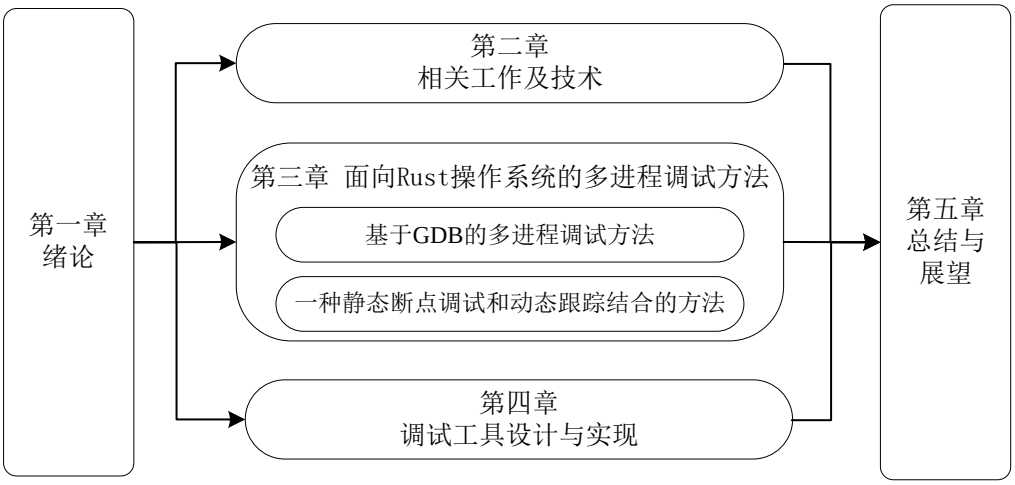

图 1.1　论文的组织结构

第2章 相关工作及技术

2.1 Rust 语言和 RISC-V 指令集

2.1.1 Rust 语言

Rust 是一种相对较新的编程语言，自 2015 年发布以来获得了巨大的关注[4]。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[5]。

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[6-7]。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[4]，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保证了内存安全和线程安全[8]，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[9]等人利用 Rust 语言创建了内核模块，并通过 Rust 语言的安全特性来避免在编写内核代码时产生内存或同步的错误。Liang [10]等人开发了 Rustpi，这是一个在 Rust 平台上运行的微内核操作系统，他们致力于利用 Rust 语言来构建一个更为可靠的操作系统。

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。

2.1.2 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[11]，它支持多个平台，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[12]。

RISC-V 展现了四大突出特性：简洁性与一致性、学校与产业之间的无缝连接、开放性的扩展能力以及高效的编程性能，这些都为嵌入式开发中的众多挑战提供了解决方案。因此，RISC-V 生态建设吸引了越来越多的企业参与，基于 RISC-V 指令架构开发的芯片和操作系统也在逐渐增加，各大高校也开始投入到 RISC-V 的教学和研发工作中。

在 RISC-V 领域中，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[13]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[14]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[15]等人评估了操作系统对基于 RISC-V 的片上系统的可靠性，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。

2.2 操作系统调试技术及调试工具

2.2.1 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 可以用于本地调试程序，它允许程序员查看程序的运行状态、检查变量和内存、设置断点等，以便在代码中找到和修复问题。GDB 服务器是 GDB 的另一部分，用于远程调试，它允许在目标计算机上运行一个小型的 GDB 服务器，然后在本地计算机上运行 GDB 与之连接。通过 GDB 服务器，可以在远程目标机上调试程序，而不需要将整个 GDB 调试器放在目标系统上，如图 2.1 所示。

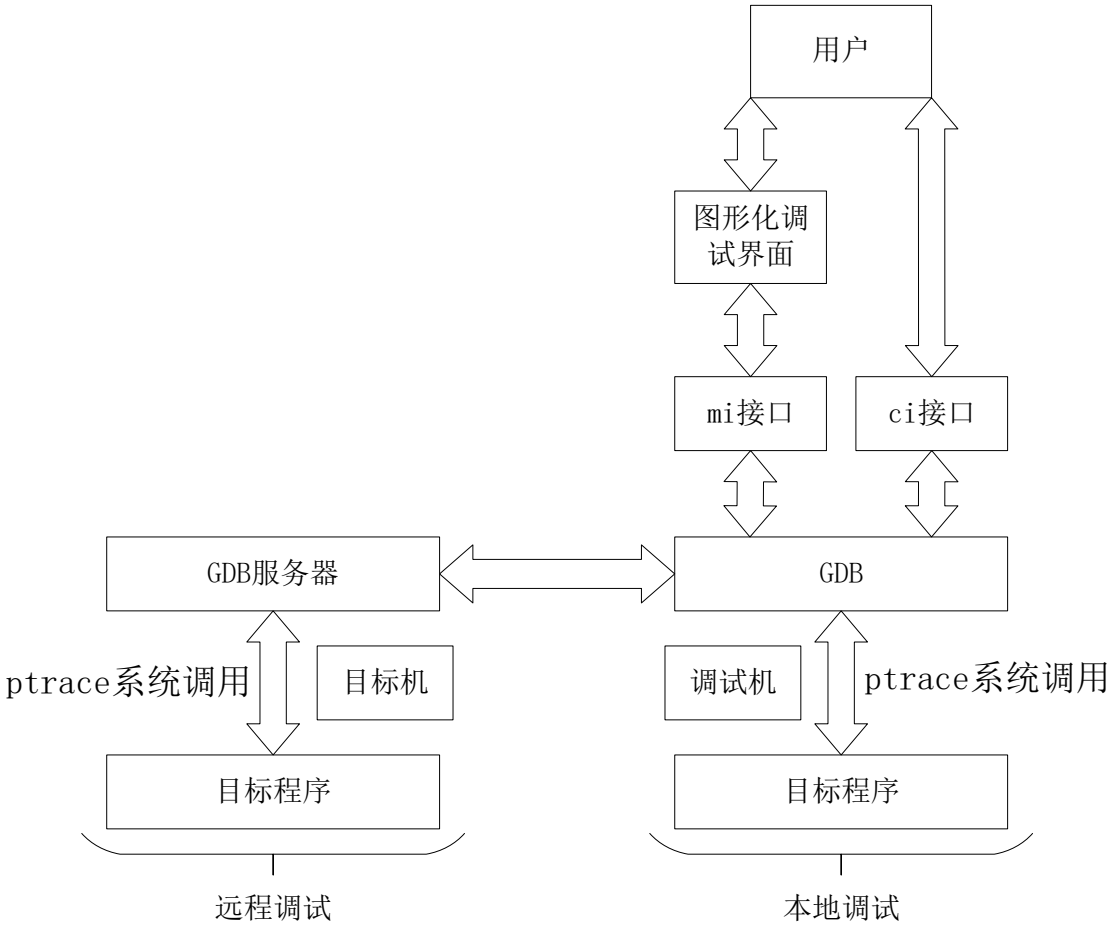

图 2.1　GDB 调试的组成架构图

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[16]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[17]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[18]。QEMU 是一种开源的虚拟机监视器和模拟器，能够模拟一台能够独立运行操作系统，其架构图如图 2.2 所示，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[18]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[19]，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。

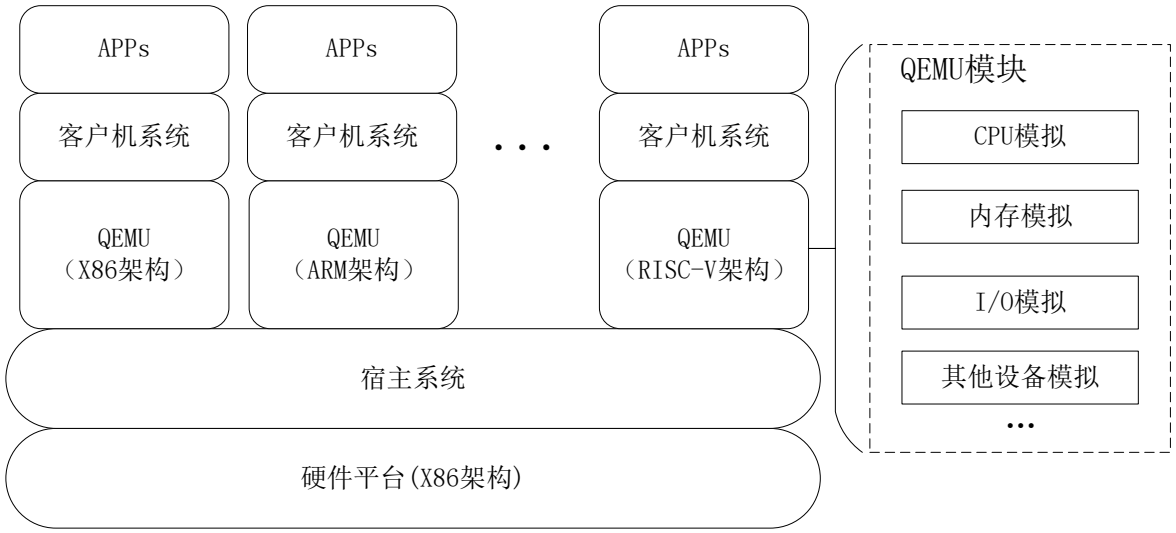

图 2.2　QEMU 架构图

Mihajlović[20]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[21]。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[22]等人提出了一个方案，该方案利用 QEMU 虚拟机架构来构建嵌入式可信计算环境。另外，Díaz[23]等人的研究对当前的软件仿真器进行了深入分析，并提出了一种创新方法，允许 QEMU 在并行模式下进行外部同步。他们构建了一个硬件/软件协同仿真虚拟平台，为 QEMU 引入了外部同步机制。作为最兼容的开源软件仿真器，QEMU 在该研究中发挥了关键作用。这一同步机制的修改改进了 QEMU 对于多线程转换器（并行模式）的原始建议，从而提高了协同仿真的性能。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。本文中开发的操作系统调试器，就使用了 GDB 的远程调试技术，基于 QEMU 中的 GDB 服务器来实现对操作系统的调试。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。

2.2.2 eBPF 和 kprobe 技术

在内核或模块的调试过程中，了解函数的调用情况、执行状态以及参数和返回值等信息至关重要。一种简单的方法是通过在函数中添加日志打印信息，但这种方法通常需要重新编译内核或模块，并重新启动设备，操作较为复杂且可能破坏原有的代码执行过程。

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[24]。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。

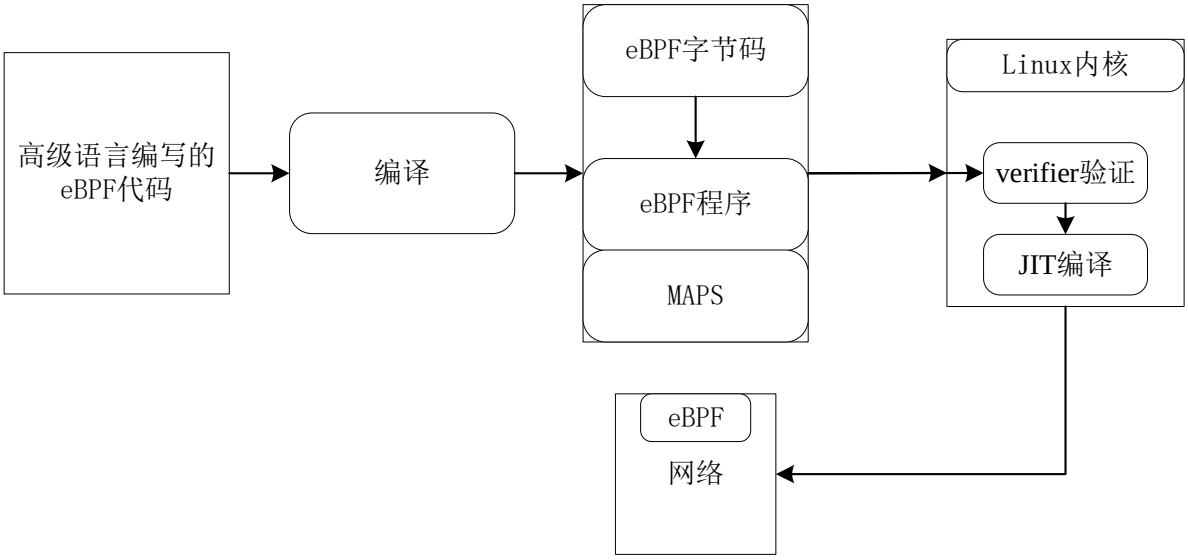

图 2.3　eBPF 工作流程

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 技术提供跟踪和分析、可观测性和监控、服务网格、企业安全等关键技术，为操作系统赋予新的能力[25]，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[26]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[27]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。

kprobes 是一种用于注册断点和相应处理程序的机制[28]。用户启用了 kprobes 支持后，可以在任意内核地址的任意指令上设置断点，从而实现对内核代码的调试。使用 kprobes 技术，用户可以自己定义回调函数，并在内核各个模块的几乎所有函数中动态地插入探测点，以便在执行流程到达指定的探测函数时调用该回调函数。通过这种方式，用户可以收集所需的信息，并在完成调试后动态地移除探测点，使内核恢复正常执行流程。kprobes 技术对内核执行流程的影响很小，因为探测点的插入和移除是动态的，并且不需要修改内核源代码或重新编译内核。其次，它操作方便灵活，用户可以根据需要随时添加或移除探测点，而无需停止内核运行或重启系统。这使得 kprobes 技术成为内核调试和性能分析的强大工具。例如，Sun[29]等人基于 kprobe 机制开发了一个轻量级、动态的应用程序编程接口收集工具：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。

2.2.3 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[30]。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。

VSCode 提供了插件架构，如图 2.4 所示，VS Code 是通过 Electron 实现跨平台的，而 Electron 则是基于 Chromium 和 Node.js，比如 VS Code 的界面，就是通过 Chromium 进行渲染的。同时， VS Code 是多进程架构，当 VS Code 第一次被启动时会创建一个主进程，然后每个窗口，都会创建一个渲染进程。与此同时，VS Code 会为每个窗口创建一个进程专门来执行插件，也就是插件进程。除了这三个主要的进程以外，还有一种调试进程，VS Code 为调试器专门创建了调试适配器进程，渲染进程会通过调试适配器协议跟调试适配器进程通讯。

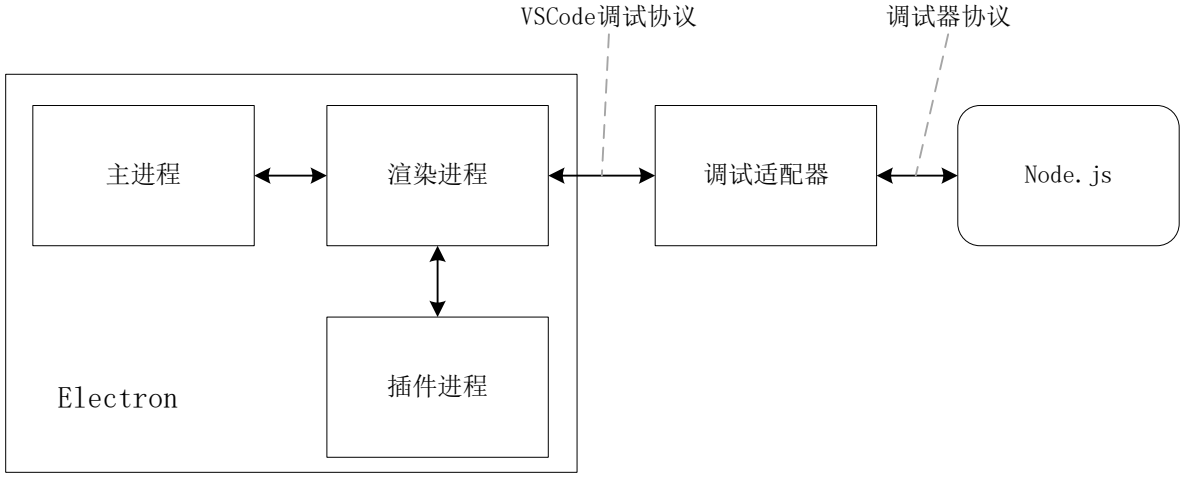

图 2.4　VSCode 插件架构

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 2.5 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。

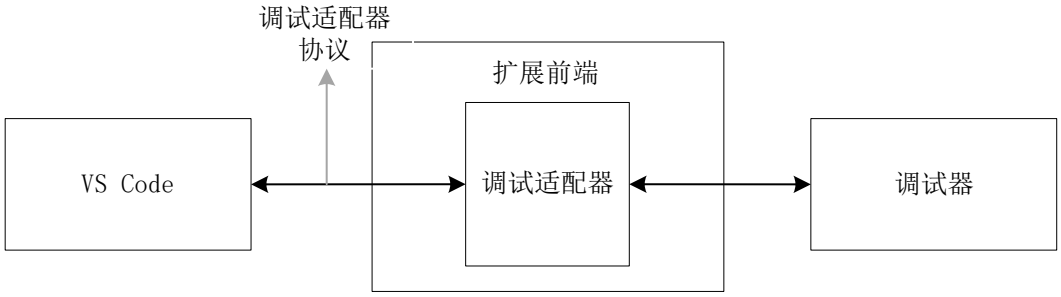

图 2.5　Visual Studio Code 中的调试架构

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。

2.3 国内外研究现状

2.3.1 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，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[31]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[32]发明了一种针对 Linux 内核的调试方法。该方法基于日志记录，旨在确定内核程序的执行位置，具体方案包括通过添加相应的日志输出接口，实现对内核程序运行全过程的跟踪。

随着应用程序和操作系统变得越来越复杂，简单地使用日志打印信息的方法已经无法满足对复杂内核代码的调试需求。因此，许多便捷的调试方法和内核跟踪[33]工具在整个软件堆栈中兴起。Gebai[34]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[35]等人提出了一项新的内核模糊技术，旨在探索内核错误可能带来的各种错误行为。Xinyang Ge[36]等人提出了 Kernel REPT，作为首个针对内核故障的实用逆向调试解决方案。该解决方案利用高效的硬件跟踪记录每个处理器上的内核控制流，并根据上下文切换历史来识别每个软件线程的控制流，进而通过模拟机器指令和硬件事件来恢复其数据流。Bissyandé[37]等人提出了 Diagnosys，这是一种自动构建 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[38]等人提出了 ErrHunter，一种基于系统静态污点分析的方法，用于检测 Linux 内核中的多种错误处理错误。该方法提出了自动识别错误处理路径中关键变量的方法，并使用静态交叉控制流污点分析构建关键变量控制流图。ErrHunter 旨在为内核漏洞检测提供支持，能够处理 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[39]等人提出了基于 bochs 虚拟机的内核调试方法，使用内部调试器时，其功能类似于硬件调试器，具备强大的软件调试功能，可以对操作系统的所有部分进行无限制的调试。Takeuchi[40]等人开发了一种使用轻量级虚拟机的新型操作系统调试方法，旨在提高 I/O 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[41]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[42]等人还提出了虚拟机监视器（VMM）监视基本进程调用以访问基本文件的系统调用，以确保文件对除相应的基本服务之外的所有服务不可见，从而保证了整个操作系统的安全性。Zaidenberg[43]等人提出了一种在开发阶段检测内核漏洞的方法，该方法利用 LgDb 在 Lguest 之上构建，Lguest 是一个内核模块，使本机 Linux 操作系统具有半虚拟化管理程序的能力。这个方法的关键在于引入了 KGDB 这个调试工具来调试客户内核，从而让用户更轻松地进行控制。

调试是确保操作系统稳定和性能的关键步骤之一。通过对操作系统进行调试，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。

2.3.2 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[44]，如图 2.6 所示。Chang[45]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电路中的仿真器从而实现软硬件协同调试。蒋龙[19]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 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。

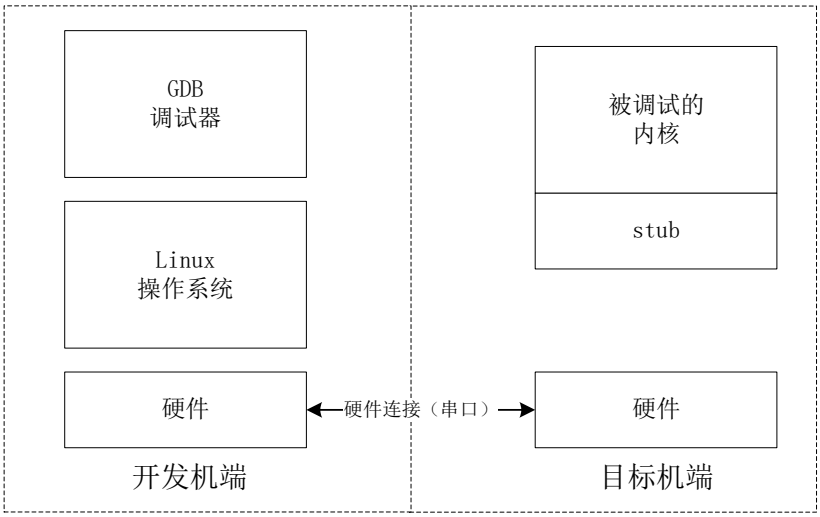

图 2.6　GDB 远程调试架构[43]

苏嘉玮[46]等人针对 RISC-V 微处理器设计了调试和仿真方案，利用其提供的 DM（Debug Mode）机制。采用了 GDB 调试工具、TCP/IP 协议以及 JTAG 协议等，结合 DM 机制完成软硬件调试。在主机中的集成环境通过 GDB 给特定行程序设置断点，然后 GDB 软件通过底层驱动 JTAG 接口访问远程处理器的 DM，向其下达

命令。DM 接收命令后，将设置断点的指令替换成 breakpoint 指令，随后当处理器执行到该指令时，将进入调试模式的异常服务程序。GDB 持续监测 DM 的状态，一旦发现处理器已经停止，便在软件界面上显示相应信息。此时，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 执行一些命令，比如查看寄存器的值。

但是对于大型复杂程序，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，在设置大量断点或触发复杂条件下，程序的执行速度可能会受到影响。除此以外，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，静态断点调试在定位程序错误、优化性能和提高软件质量方面仍然持续发挥重要作用。

2.3.3 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 [47]等人提出了 dottobpf 工具，利用 Linux eBPF 进行运行时验证，通过生成 eBPF 程序根据三值线性时间逻辑规范，有效地进行系统分析。Ramachandran [48]等人介绍了一种基于 eBPF 技术的新型框架 FUSE，旨在提供实时故障检测和抑制，有助于预防级联故障。Zhan [49]等人提出了一种基于 eBPF 技术的多维度网络拥塞检测方法，在 Linux 平台上能够动态监测网络拥塞状态机的变化。Jiang [50]等人提出了一种新工具，用于检测 Linux 内核环境中的竞争条件，利用动态检测方法和商用处理器上的硬件调试寄存器捕获动态竞争。Weng [51]等人开发了 Kmon，这是一个基于 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 [52]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，Li[53]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[54]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。

综上所述，eBPF 和 kprobe 等动态跟踪技术在操作系统调试方面发挥着重要作用。通过这些技术，开发者可以更加全面地监测操作系统运行中的状态，包括系统调用、进程活动、网络通信等各个方面的细节，有助于快速定位和解决操作系统中的各种问题，提高系统的稳定性和可靠性。

2.4 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 2.7 是本文开发的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。

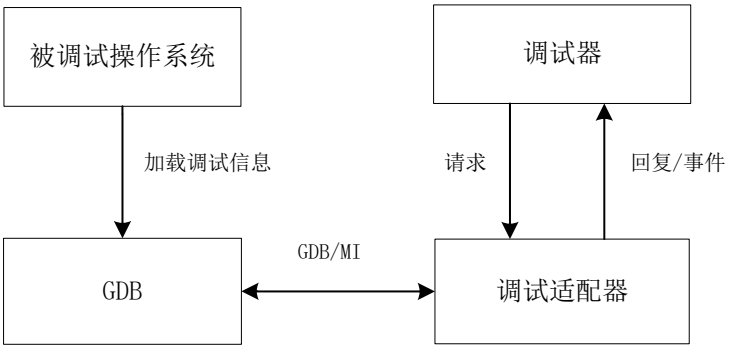

图 2.7　总框架图

2.5 本章小节

在本章节中介绍了本文涉及的相关工作及技术，首先介绍了 Rust 语言和 RISC-V 指令集架构的特点及其在操作系统开发领域的发展现状，其次介绍了操作系统调试技术及调试工具，其中包括 QEMU 虚拟机和静态断点调试工具 GDB，动态跟踪技术 eBPF 和 kprobe，以及本文使用的调试器开发工具 VSCode，然后介绍了操作系统调试方面的国内外研究现状，最后简单介绍了本文要调试的 Rust 操作系统即 rCore-Tutorial-v3，以及本文开发的调试器整体系统架构。

第3章 面向 Rust 操作系统的多进程调试方法

3.1 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。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。随着各大高校和各个领域中 Rust 操作系统的开发，对其调试的需求也日益凸显。然而，在实际应用中，针对 Rust 操作系统的调试仍然面临一些挑战[3]。

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，调试器根据进程名称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，本文设计了 eBPF 处理模块，该模块负责接受处理用户发送的调试请求，并将 eBPF 收集到的调试信息传送到 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。

3.2 基于 GDB 的多进程调试方法

3.2.1 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名称，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。

3.2.2 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。

```
class AddressSpaces {
    protected spaces: AddressSpace[];
    protected currentSpaceName: string;
    protected debugSession: MI2DebugSession;
    constructor(currentSpace: string, debugSession: MI2DebugSession) {
        this.debugSession = debugSession;
        this.spaces = [];
        this.spaces.push(new AddressSpace(currentSpace, []));
        this.currentSpaceName = currentSpace;
    }
```

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。

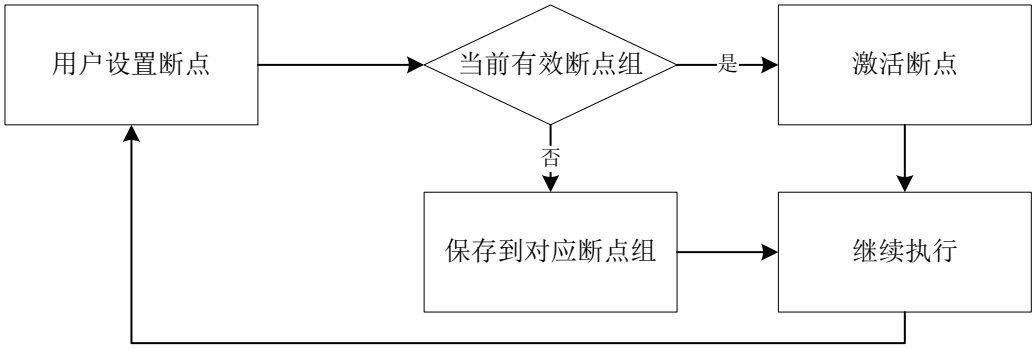

图 3.2　断点设置流程图

3.2.3 调试器中内核态和用户态的切换

3.2.3.1　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。

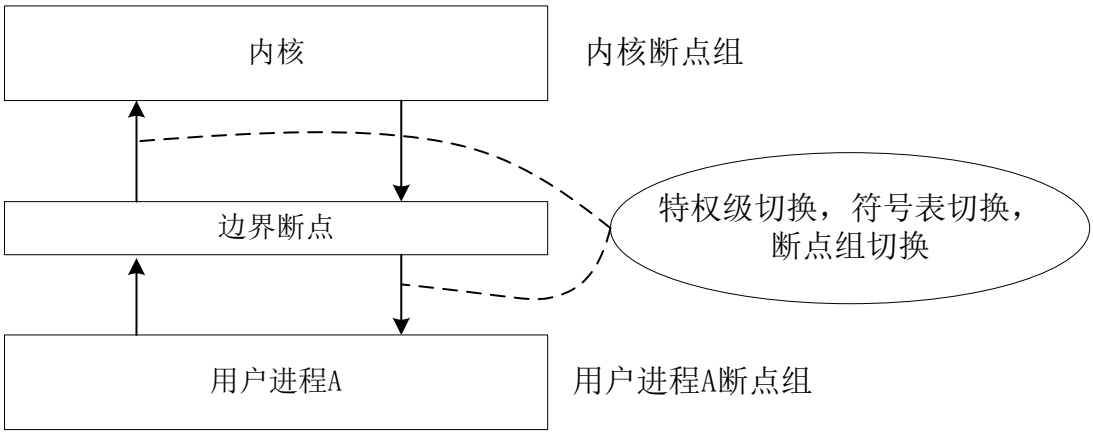

图 3.3　边界断点及特权级转化过程图

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。

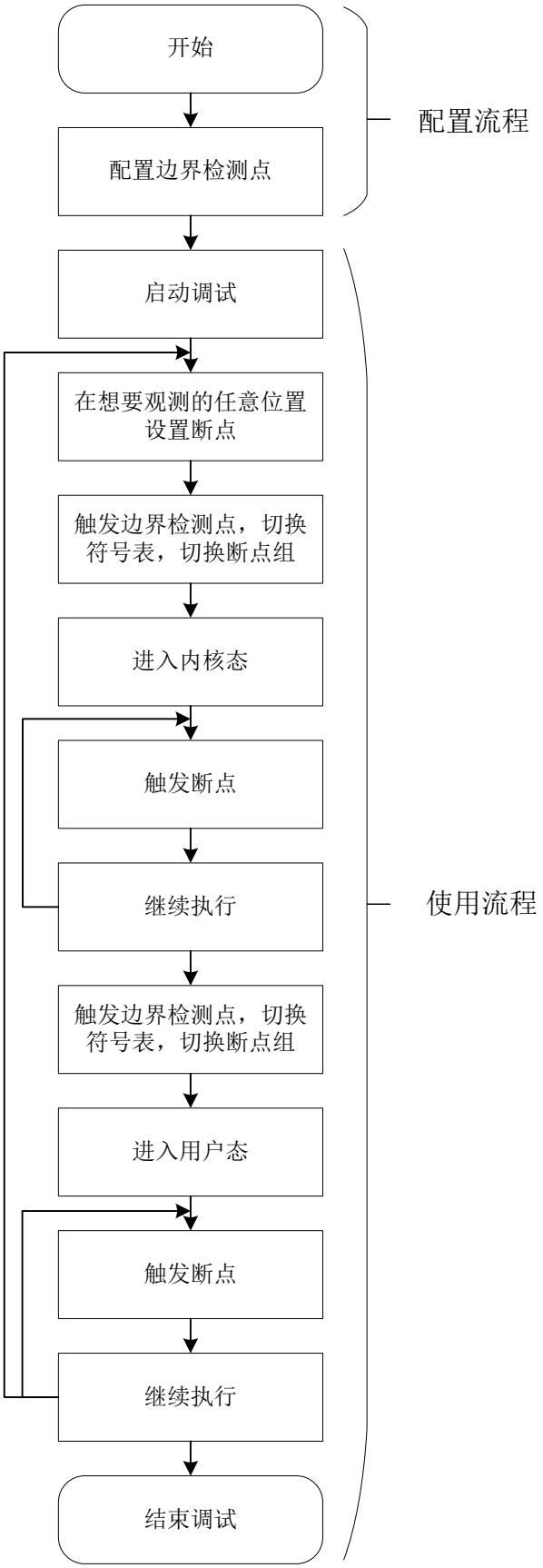

图 3.4　调试器配置使用流程

3.2.3.2 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

3.2.4 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。调试器是根据进程名称，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名称。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名称，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名称。操作者需要在被调试操作系统的内核态源代码中，含有进程名称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，来获取进程名称。在调试器获取到进程名称以后，就可以根据进程名称来进行符号表的切换，最终实现多进程的调试。

3.2.4.1 进程控制块信息的获取

要获取进程控制块中的进程名称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名称信息的，因此先要修改 rCore-Tutorial-v3，在进程控制块中添加进程名称信息，其次，在调试器中获取进程控制块中的进程名称。

在 rCore-Tutorial-v3 中 get_exec_path 函数用于获取进程控制块中进程名称，在进程控制块中添加进程名称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。

接下来需要在调试器中获取进程控制块中的进程名。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。

在操作系统中，进程名称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，来获取进程名称。

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。

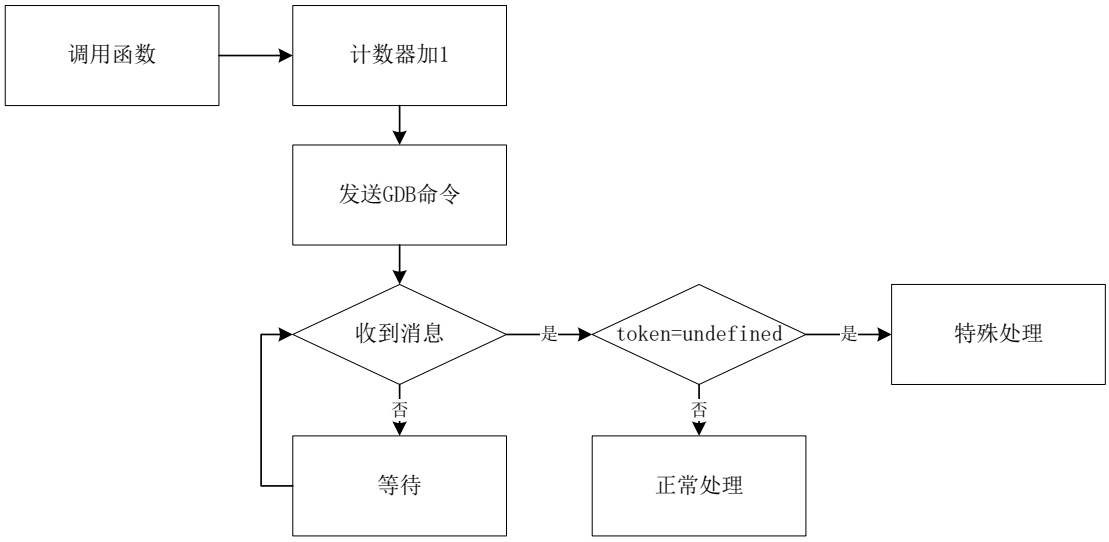

图 3.5　发送命令函数执行流程

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。

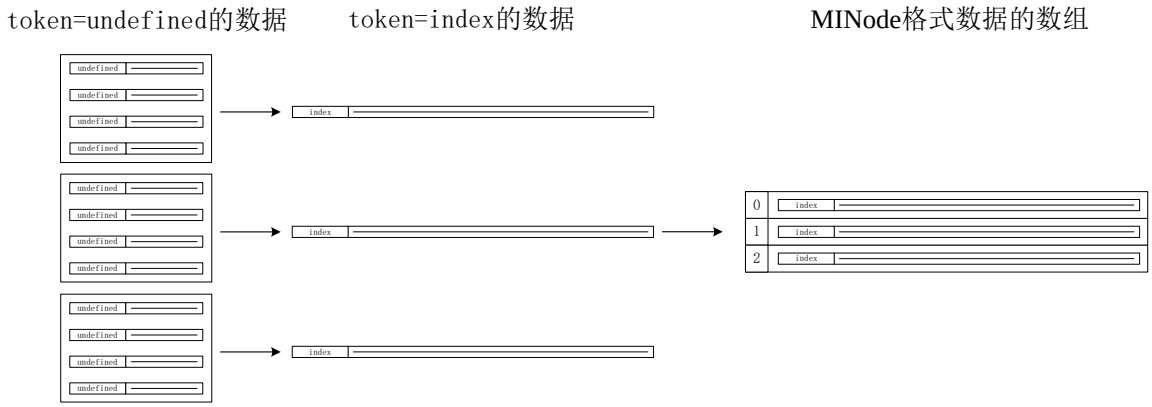

图 3.6　数据格式处理示意图

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，获取到的进程控制块信息如图 3.7 所示，可以观察到，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名称，如图 3.8 所示。

图 3.7　进程控制块信息

图 3.8　进程名称信息

3.2.4.2　多个用户进程的符号表切换

在获取进程名称以后，就可以进行用户进程符号表的切换，如图 3.9 所示。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 的进程名称进行符号表的切换，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名称，如用户进程 C。在获取到用户进程 C 的进程名称以后，赋值给保存进程名称的变量，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。

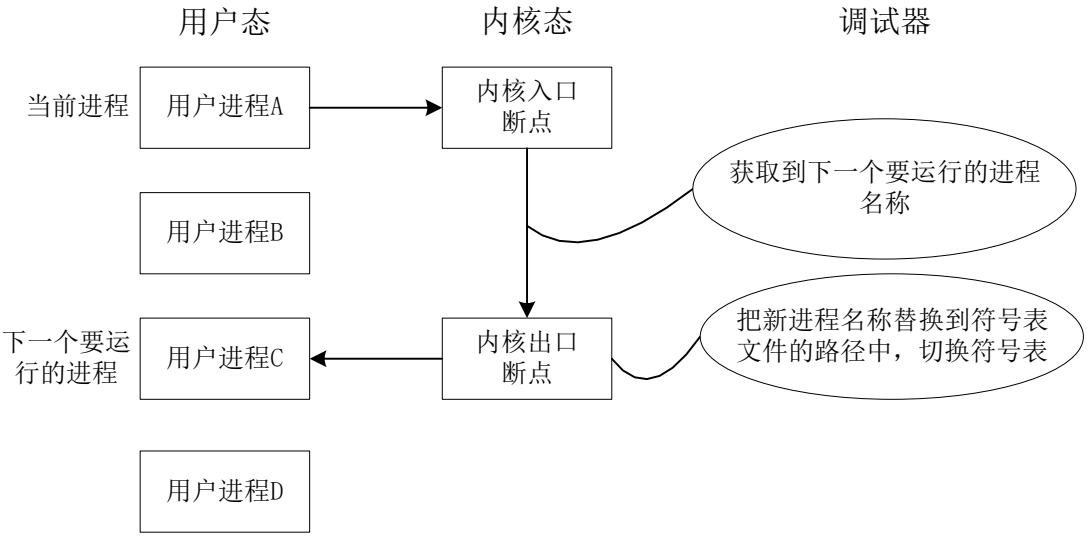

图 3.9　多进程切换示意图

3.2.5 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；使用设边界检测点的方法，解决调试器中内核态和用户态切换的问题；最后，通过获取进程名称，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。

3.3 一种静态断点调试和动态跟踪结合的方法

3.3.1 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，但是单独使用 GDB 对操作系统进行调试仍然不够完善，具体来说，主要有以下几点。首先， GDB 对 Rust 语言的支持不够完善，不如像 C/C++ 语言的编译器在生成调试信息方面成熟，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。其次，GDB 静态断点调试需要暂停被调试操作系统的运行，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。最后，GDB 支持在虚拟环境中进行调试，如果 Rust 操作系统需要运行在真实环境中，并投入生产的时候，单独的 GDB 调试无法满足调试需求。因此本文引入了 eBPF 和 kprobe 两种调试技术，它们作为内核级别的调试工具，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，将 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：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，传递到用户界面上，本文为 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：为 eBPF 调试信息实设计 eBPF 处理模块，负责接受用户的调试请求，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，和数据的展示。

3.3.2 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 3.10 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。

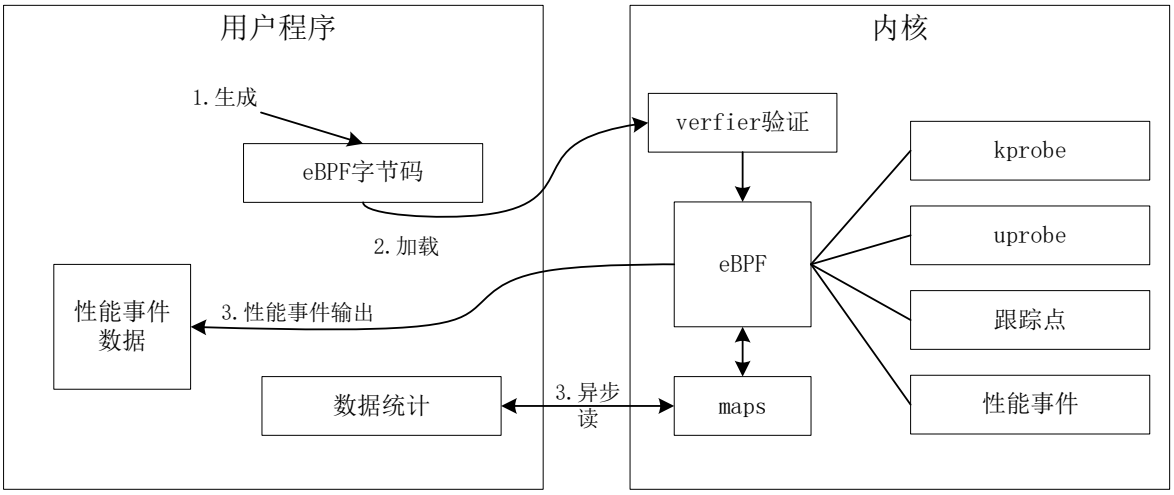

图 3.10　eBPF 工作流程

3.3.3 eBPF 与 GDB 调试信息的数据流整合

3.3.3.1　eBPF 处理模块的设计与实现

本文提出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 QEMU 种运行的操作系统。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

在 Rust 操作系统能够支持 eBPF 程序运行以后，使用者就可以发送命令，使用

eBPF 和 kprobe 收集被调试操作系统运行中的调试信息，但是 eBPF 收集到的调试信息保存在操作系统内部，本文要将 eBPF 收集到的数据，传递到用户本地的调试插件中。

如 2.2.1 小节所讲述的，利用 GDB 自带的远程调试功能，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 服务器的同时也获取到操作系统中 eBPF 的调试信息。

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，操作系统运行在 QEMU 中，通过一个专有的串口进行调试信息，调试命令的传输。调试器端可以通过 GDB 向这个串口传入调试命令，获取调试信息。但是，操作系统中并没有负责接收从串口传来的调试命令的进程，也无法根据调试命令进行 kprobe 注册和 eBPF 的相关操作。本文的解决方案是由一个用户态进程进行这些操作，并把这个用户态进程称做 eBPF 处理模块，本文设计了 eBPF 处理模块，该模块运行在操作系统中，负责接受用户传递过来的调试请求，并在请求中解析出函数插桩所需要的信息，之后调用 eBPF 对应的函数，在操作系统中进行插桩等操作。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 3.11 所示。

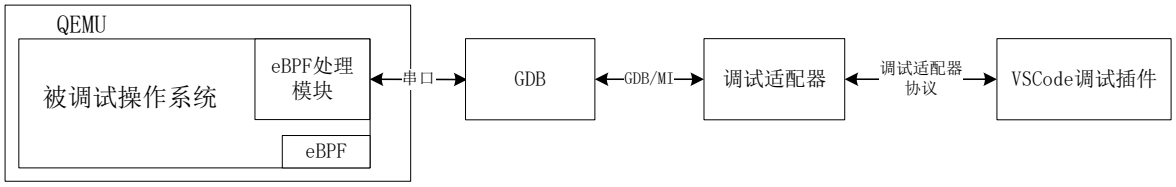

图 3.11　eBPF 处理模块数据流图

3.3.3.2　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，为了统一 GDB 和 eBPF 调试信息的流向，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 3.12 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 3.12　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 3.13 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 3.13　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 3.14 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。

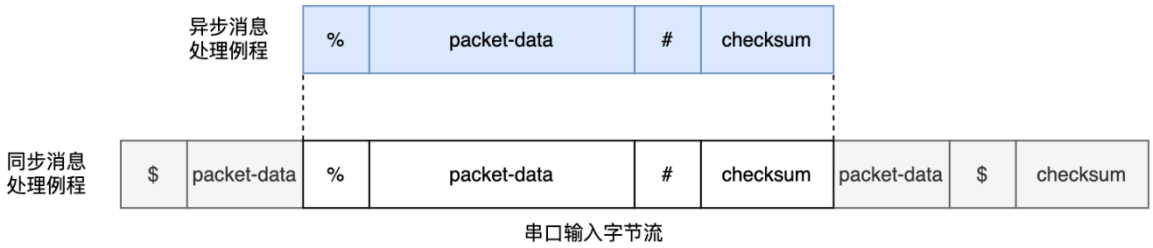

图 3.14　消息处理例程

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 3.15 为异步消息处理例程的主要代码，图 3.16 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。

```python
def read_async_msg(self,starts_with):
    msg=''
    end_count = 10000000 # todo: set this as msg_max_len
    # print("gonna loop")
    while end_count > 0:
        c = self.ser.read(1)
        if c == b'\x00':
            continue
        c = str(c,'ascii')
        # gdb.execute("echo "+c)
        msg+=c
        if c == '#':
            end_count = 3
        end_count -= 1
    gdb.execute('echo eBPF Message: '+starts_with+msg+"\n")
```

图 3.15　异步消息处理例程的主要代码

```
def msg_reader(self):
    while True:
        input_stream = "" # **A** packet
        end_count = 10000000 # todo: set this as msg_max_len
        while (end_count > 0):
            c = str(self.ser.read(1),encoding='ascii')
            if c == '+':
                pass
            elif c == '%':
                # print('Percentage Symbol')
                self.read_async_msg(c)
                continue
            elif c == '#':
                end_count = 3
                input_stream+=c
            else:
                input_stream+=c
            end_count-=1
        self.ser.write('+'.encode('ascii'))
```

图 3.16　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，数据流图如图 3.17 所示。

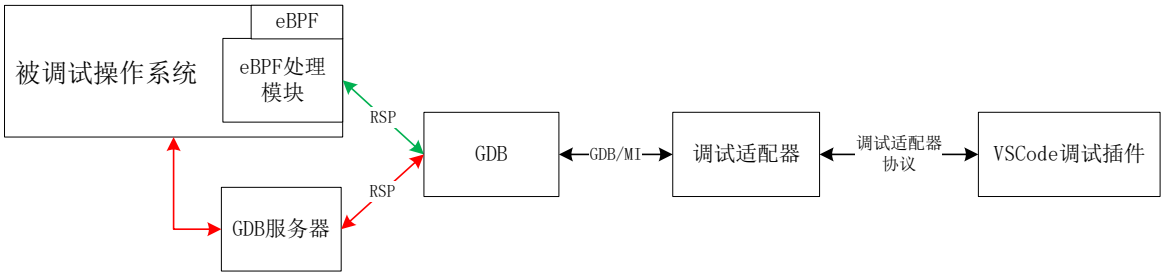

图 3.17　数据流图

3.3.3.3　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，它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 3.18 所示。

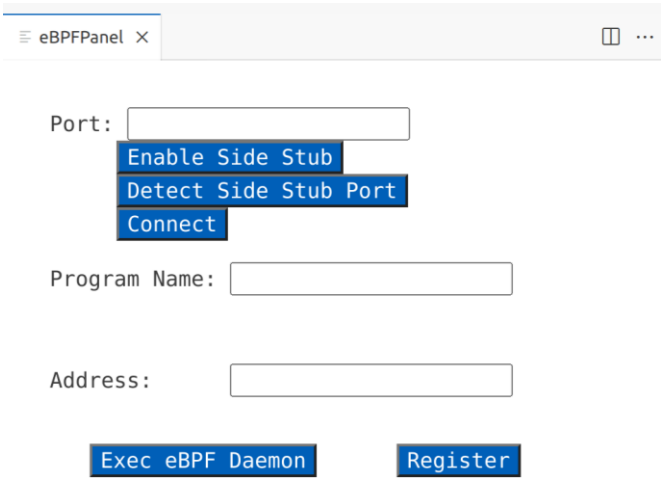

图 3.18　eBPF 调试选项

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。

3.3.4 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 eBPF 处理模块的设计与实现的过程，eBPF 处理模块与 GDB 之间 RSP 协议的实现，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，同时能够动态地获取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。

3.4 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。首先深入探讨了目前 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，借助设置边界检测点的方法，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，通过获取进程名称，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，在该方法中进行了 eBPF 与 GDB 调试信息的数据流整合，详细描述了 eBPF 与 GDB 之间基于串口的通信的实现过程， eBPF 处理模块与 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 的多进程调试方法以及静态断点调试和动态跟踪相结合的调试方法，不仅能够对不同特权级下的多进程进行调试，还能够在不影响操作系统运行的情况下实时获取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。

第4章 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 GDB 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。

4.1 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，本课题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，用户在本地浏览器使用 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。

在用户使用的本地环境中包括 VSCode 调试插件和扩展前端，用户可以在 VSCode 调试插件中接收到调试过程中产生的相关信息，在扩展前端界面中会加载被调试操作系统的源代码，在该窗口中可以设置断点，观察到被调试操作系统运行过程中的一些本地信息，进行调试操作，VSCode 调试插件则负责处理在扩展界面发送的调试请求，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。

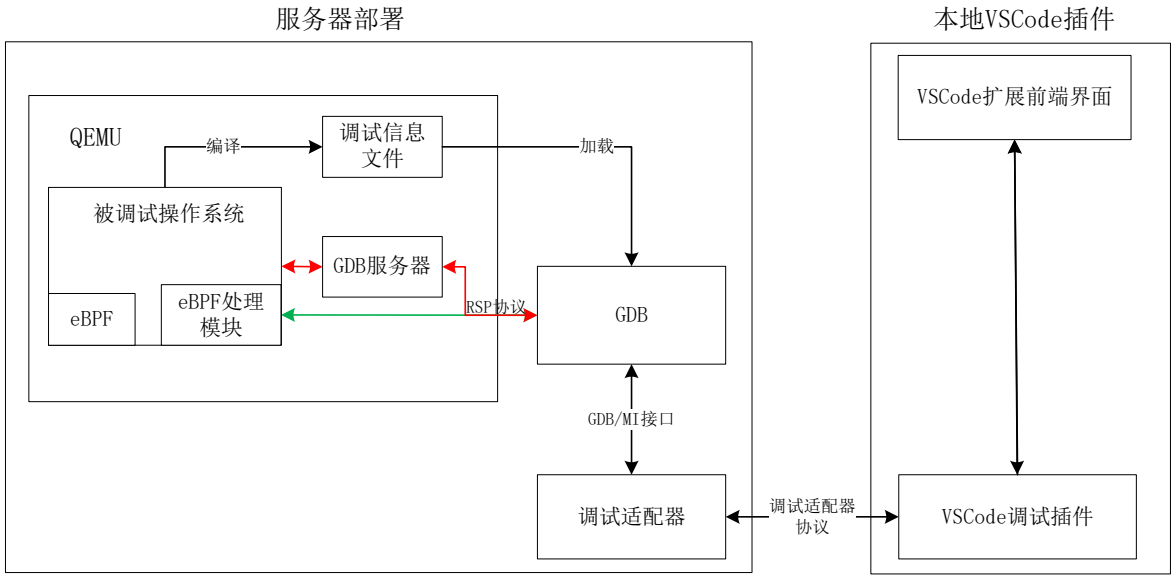

图 4.1　系统框架图

4.2 服务器部分

4.2.1 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，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。

4.2.2 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]难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。

4.2.3 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调试插件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，向调试插件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调试插件。

图 4.2　调试适配器和 GDB、调试插件的通信机制

4.3 网页端部分

在用户浏览器上运行的在线调试环境中，调试插件负责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。调试插件会解析接收到的回复和事件并将需要的信息转发至用户扩展前端界面。如果用户扩展前端界面向调试插件传递了某个消息，调试插件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。

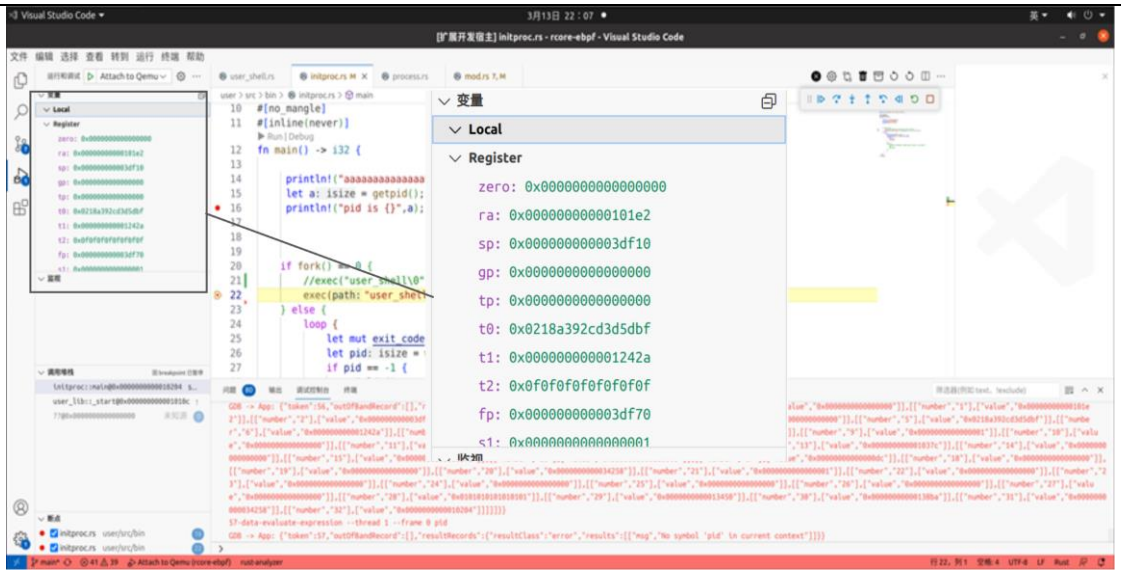

图 4.3　通过树视图展示数据

提供了功能按钮如图 4.4.所示：

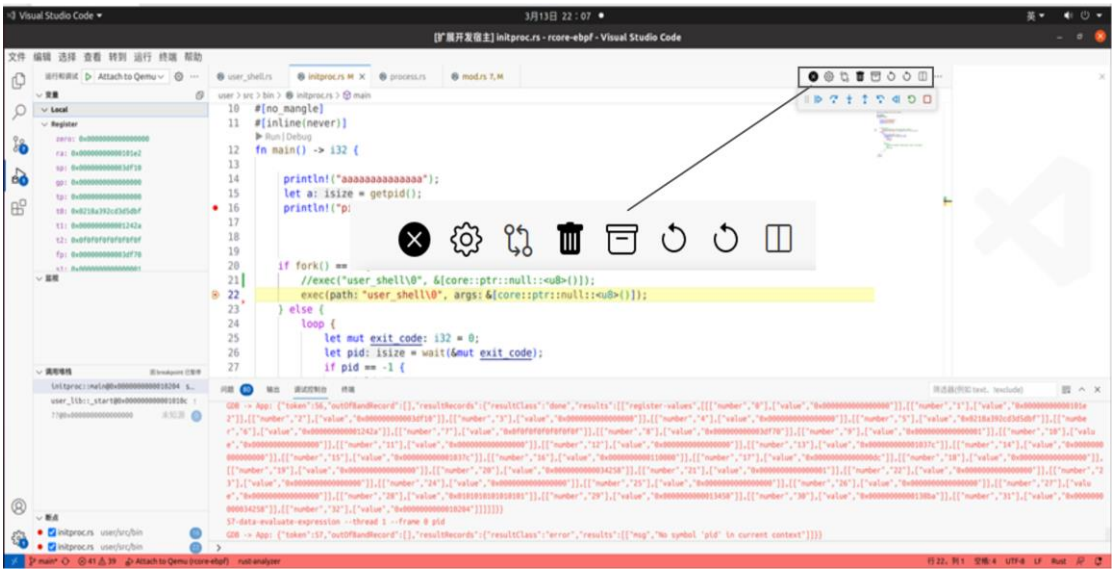

图 4.4　原生样式的命令按钮

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

名称	功能
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：

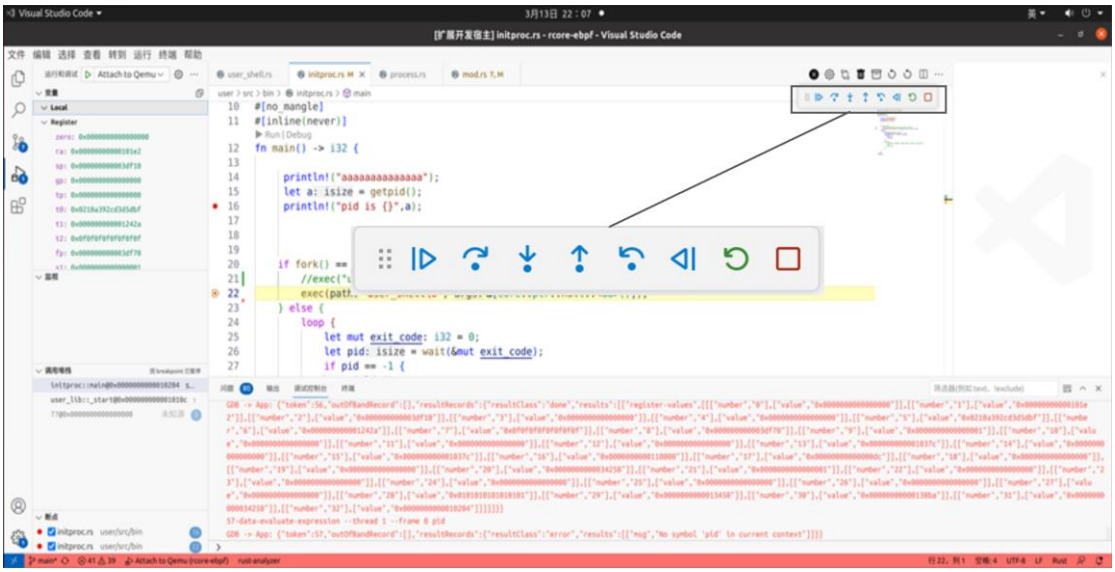

图 4.5　调试功能按钮

4.4 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。

4.4.1 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。

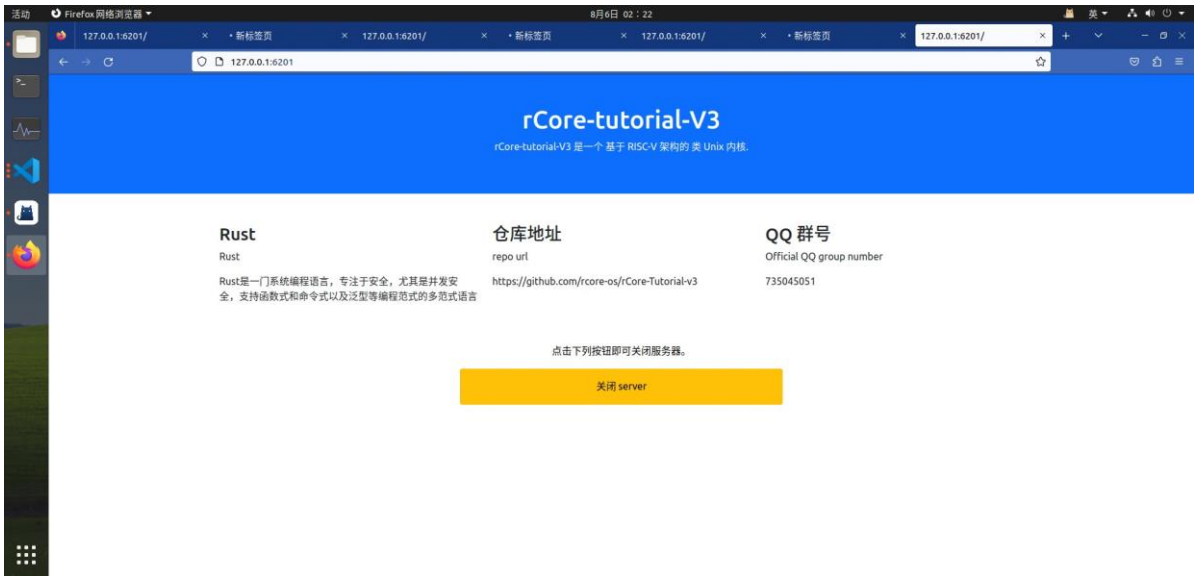

图 4.6　tcp_simplehttp 服务器程序未返回所有请求的网页

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。

4.4.2 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。

4.4.3 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。

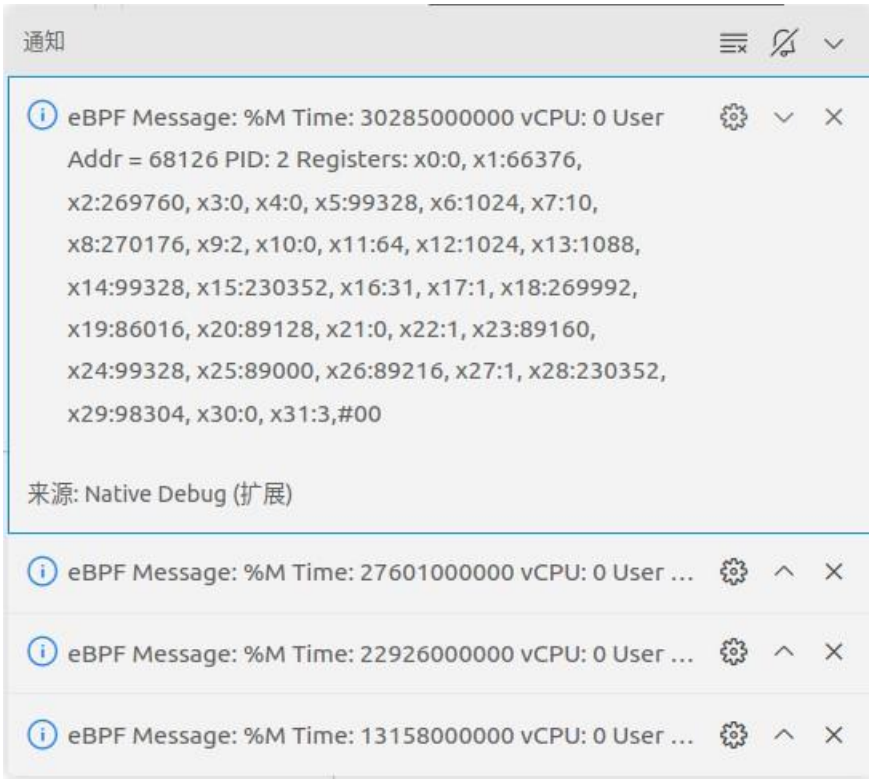

图 4.8　eBPF 检查点触发，弹出调试信息

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。

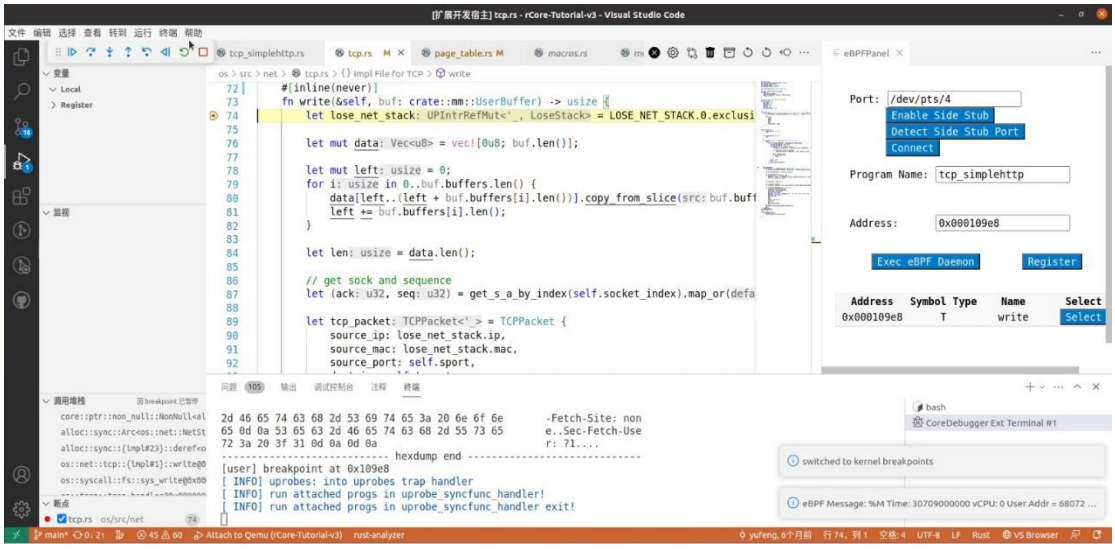

图 4.9　`write`函数的 GDB 断点触发

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。

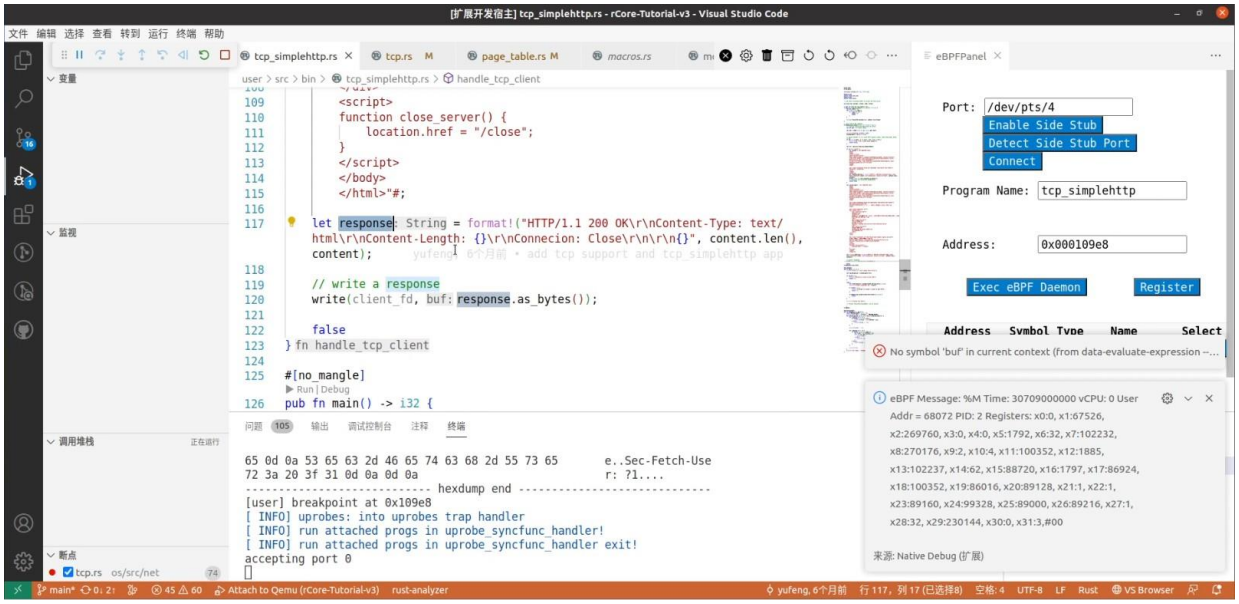

图 4.10　出错的代码

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。

4.5 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 处理模块负责收集调试信息，处理命令等工作，通过远程环境中提供的 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，在不修改操作系统代码的情况下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。

第5章 总结与展望

5.1 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。本文提出了面向 Rust 操作系统的多进程调试方法，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：

针对目前大多数源代码级调试器无法感知操作系统特权级切换以及无法进行不同特权级下多个进程调试的问题，提出了基于 GDB 的多进程调试方法。在该方法中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名称，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。在此方法的基础上，为了使调试器的功能更加全面，提出了一种静态断点调试和动态跟踪调试方法，在该方法中结合了 GDB 和 eBPF 两种调试技术，主要探讨了在 rCore-Tutorial-v3 操作系统中进行 eBPF 处理模块设计与实现及 GDB 和 eBPF 处理模块数据流整合的过程中所遇到的问题以及解决方法。首先，在该方法中设计了 eBPF 处理模块，负责接受用户发送的调试请求，并将收集到的调试信息传送到 GDB，这一过程中使用了与静态断点调试相同的 RSP 协议。接着，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，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。

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，在不修改操作系统代码的情况下，通过调试和分析最终找到了问题代码。

5.2 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] 吴竞邦,张露元,陈志扬,等. 跨内核态和用户态的操作系统源代码调试方法及装置[P]. 北京市：CN117707931A,2024-03-15.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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: 1224-1252.

[9] 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.

[10] 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.

[11] Cui E, Li T, Wei Q. Risc-v instruction set architecture extensions: A survey[J]. IEEE Access, 2023, 11: 24696-24711.

[12] 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.

[13] 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.

[14] Kalapothas S, Galetakis M, Flamis G, et al. A survey on risc-v-based machine learning ecosystem[J]. Information, 2023, 14(2): 64.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 蒋龙. 基于 GDB 的嵌入式多任务调试器的设计实现与集成[D].浙江大学,2014.

[20] Mihajlović B, ŽilićŽ, 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 高巍.基于操作系统 eBPF 在云原生环境下的技术研究[J].电子技术与软件工程,2022(17):70-74.

[26] 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.

[27] 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.

[28] Krishnakumar R. Kernel korner: kprobes-a kernel debugger[J]. Linux Journal, 2005, 2005(133): 11.

[29] 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.

[30] 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.

[31] 周毅,程石林,陆暻冉,等. 一种 Linux 内核调试系统及方法[P]. 江苏省：CN106227653B,2018-10-30.

[32] 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.

[33] Desnoyers M, Dagenais M R. Synchronization for fast and reentrant operating system kernel tracing[J]. Software: Practice and Experience, 2010, 40(12): 1053-1072.

[34] 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.

[35] 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.

[36] 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.

[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] Zhan D, Yu X, Zhang H, et al. ErrHunter: Detecting Error-Handling Bugs in the Linux Kernel Through Systematic Static Analysis[J]. IEEE Transactions on Software Engineering, 2022, 49(2): 684-698.

[39] 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.

[40] Takeuchi T. OS debugging method using a lightweight virtual machine monitor[C]//Design, Automation and Test in Europe. IEEE, 2005: 1058-1059.

[41] 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.

[42] 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.

[43] Zaidenberg N J, Khen E. Detecting kernel vulnerabilities during the development phase[C]//2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing. IEEE, 2015: 224-230.

[44] 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.

[45] 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.

[46] 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.

[47] Lakicevic N. Runtime Verification with Linux eBPF[D]. Norway: University of Oslo, 2023.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] Zavarella T D. A methodology for using eBPF to efficiently monitor network behavior in Linux Kubernetes clusters[D]. America: Massachusetts Institute of Technology, 2022.

[53] 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.

[54] 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.

在学期间发表的学术论文与研究成果

[1] On the design and implementation of a real-time testbed for distributed TDMA-based MAC protocols in VANETs[J]. IEE Access, 2021, 9: 122092-122106.

[2] A Multi-Secret Reputation Adjustment Method in the Secret Sharing for Internet of Vehicles[J]. Security and Communication Networks, 2022, 2022.

[3] 支持跨内核态和用户态的操作系统源代码级调试方法. 发明专利申请号：2023115942620（已公开）