目录

摘要

第1章      绪论

1.1选题的背景及意义

1.2国内外研究现状

1.3论文主要研究内容和组织结构

1.4本章小结

第2章      相关工作及技术

2.1

系统架构总览

第3章     基于GDB的多进程调试方法

3.1问题概述

3.2断点组管理模块的设计

3.3调试器中内核态和用户态的切换

         3.3.1边界断点的实现

         3.3.2用户态进入内核态的调试选项

3.4多个用户进程的切换和调试

         3.4.1进程控制块信息的获取及展示

         3.4.2多个用户进程的符号表切换

3.5本章小结

第4章      一种静态断点调试和动态跟踪结合的方法

4.1问题概述

4.2动态调试的工作机制

4.3eBPF移植

4.4 eBPF处理模块的设计与实现

4.3.1eBPF处理模块基于串口的通信机制及协议实现

4.3.2eBPF处理模块的适配

4.4本章小结

第5章      调试工具设计与实现

5.1整体框架设计

5.2服务器部分

5.3网页端

5.4本章小结

时间安排

十一月底进行中期答辩，明年 三月份可能就要完成大论文，字数不少于2万，页数在60页左右，参考文献不少于50篇

总页数65页，目录摘要，致谢之类的有13页，第一章第二章大概14页

论文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。我们提出了一种支持跨特权级的操作系统调试方法，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程序的调试需求。

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。

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。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。源代码级调试工具的主要功能是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，为使用者定位程序问题以及修改代码提供帮助，是相关软件开发中不可缺少的一种工具。

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，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。

针对上述问题，本课题旨在提供一种全新的源代码级调试方法，支持跨越内核态和用户态的操作系统调试。通过本研究，我们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更强大的调试工具。

1.2国内外研究现状

**操作系统调试技术及调试工具的发展历程**

1. 初始阶段：手动调试与 printf 调试

在早期计算机时代，程序员主要依赖手动排查错误。当程序出现问题时，程序员需要仔细检查代码，逐行排查问题。这种方式显然效率低下，尤其是在大型项目中。

另一种方法是使用 printf 进行调试。程序员在代码中插入 printf 语句，输出变量值或关键信息，以便理解程序的执行流程。这种方法简单，但输出信息有限，也会干扰程序的正常执行。

2. 进入硬件级调试：硬件断点和监控

随着硬件技术的进步，出现了硬件级调试技术。硬件断点允许程序在特定地址停止执行，这对于精确定位 bug 非常有帮助。硬件监控技术允许程序员监视内存和寄存器的状态，从而更深入地了解程序的运行情况。

举例：使用硬件断点调试器，如JTAG（Joint Test Action Group）可以在芯片级别实现硬件级调试。

3. 引入软件断点：调试器的崛起

调试器是一种能够与运行中程序交互的工具。通过引入软件断点，调试器可以在程序执行到指定位置时暂停，允许开发者查看变量、调用栈等信息。这种方法大大提高了调试的效率。

举例：GDB（GNU Debugger）是一款常用的调试器，支持多种编程语言，包括C、C++、Rust等。

4. 操作系统级调试：内核调试技术的发展

随着操作系统的发展，对于内核级别的调试需求逐渐增加。内核调试技术允许在内核态进行调试，监视和修改内核数据结构、中断服务例程等。

举例：KGDB是一个为 Linux 内核设计的调试工具，可以通过串口等方式实现内核的远程调试。

5. 动态追踪技术：DTrace、SystemTap等

动态追踪技术允许在程序运行时动态监视程序状态，包括函数调用、系统调用等。这为开发者提供了更深入地了解程序执行过程的手段。

举例：DTrace是Solaris和一些BSD系统上的动态追踪工具，可以实时监控和跟踪系统运行。

6. 虚拟机调试：适应云计算时代

随着云计算的兴起，虚拟机调试成为关键的技术。虚拟机调试器可以在虚拟环境中进行调试，监视和修改虚拟机中的程序。

举例：QEMU是一款支持虚拟化的开源模拟器，同时也提供了调试功能。

7. eBPF 的兴起：更强大的内核级调试

eBPF（extended Berkeley Packet Filter）技术在 Linux 内核中崭露头角。它允许在内核中插入小型程序，实现动态追踪和监控内核的各个方面。

举例：eBPF可以用于实现高效的动态追踪，例如监视系统调用、网络数据包等。

多进程调试工具，调试原理

Linux社区开发了Linux操作系统中的内核模块调试工具kgdb[1]，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 attach到本地主机的kgdb上，从而实现对内核的源码级调试，但是它是Linux操作系统中的一个模块，只能对Linux进行调试；蒋龙[2]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；殷绍剑[3]等人为aCoral 设计实现了一个远程调试代理系统，aCoral是一款嵌入式实时操作系统，它运行在目标机上，然后使用调试代理来与宿主机上的GDB进行沟通，从而达到调试的目的；

模拟器调试。通常使用的Simulator 是指令级的模拟器(IIS)，它相当于在宿主机上虚拟了一台目标机。该目标机可以是和宿主机的CPU不同的类型。利用指令集模拟器进行的交叉调试是一种完全软件模拟的调试方法，根本不需要目标板的支持，就连I/0等设备也都是软件模拟的。比如Qemu就是一款非常优秀的开源模拟器，可以对应用程序和系统程序进行模拟执行，并且可以通过基于RSP实现的软件调试接口与宿主机GDB进行调试会话，实现调试。使用模拟器调试的有点在于，这种调试是一种纯软件的模拟调试，其可扩展性非常好，常常可以根据需求很快的变动，这在嵌入式软件开发的起步阶段是非常有实用价值的，可以大大降低研发成本。

Zaidenberg[4]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，所以他就引入了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；Meghana D K[5]等人提出了把调试信息从二进制文件中分离出来，并把它保存到云存储空间，从而减少二进制可执行文件的大小，提高调试的效率的方法，由于内核代码量较大，调试启动花费的时间确实较多，可以考虑把该方法引用到我们目前的项目中。

但是上面提到的工具都需要进行环境的配置，其过程及原理都很复杂，对于初学者来说可能比较困难，更好的方法是为用户提供一种在线的工具，让用户能够在浏览器的网页上就直接能够进行代码的修改和调试。现在市面上已经有相关产品，例如github codespaces, 它能够支持Rust代码的在线编辑，但是并没有开放源代码，使用、部署的限制比较大，且不具备操作系统相关的调试功能。

1.4论文主要研究内容和组织结构

第二章主要介绍相关工作及技术

第三章实现了一种基于GDB的多进程调试方法，在该方法中，我们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，我们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。

第四章主要实现了一种静态断点和动态跟踪结合的方法，来对进程进行调试，首先进行了对当前调试操作系统进行了eBPF移植，使操作系统能够支持eBPF程序的运行，然后设计了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，并获取一些调试信息，并把获取到的信息通过串口和GDB进行交换，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，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。

第五章介绍了系统实现

第2章 相关工作及技术

RISC-V指令集架构

RISC-V是一个基于精简指令集（RISC）原则的开源指令集架构（ISA）。RISC-V具有4个特点，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编程效率高，正好可以解决嵌入式开发中的很多痛点。所以越来越多的企业也投入到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越来越多，各大高校也开始了对RISC-V 的教学和开发。RISC-V的指令集架构是完全开源的，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。RISC-V广泛应用于多个领域。在教育中，它被广泛用于计算机体系结构和操作系统的课程。在嵌入式系统中，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，追求更大的灵活性和自定义性。

调试工具：GDB

在软件领域，GNU是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，如QEMU中的GDB服务器。GDB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的一些设置。用户在代码的某处设置断点以后，能够看到程序运行到当前代码时，本地的变量信息、堆栈信息等。

Rust语言

Rust语言是一门系统编程语言，最初由 Graydon Hoare 于2006 年创造和发布，Rust是近年来兴起的系统级编程语言，由火狐浏览器的开发商Mozilla研发和维护。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，所以未来将有更多的人关注并使用这一编程语言。

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，零成本抽象和支持高并发。所以，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，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。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。

kprobe技术

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。而利用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。

eBPF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。它依赖于静态分析来保护内核免受有漏洞的、恶意的插件的破坏。 eBPF程序可以满足各种复杂的监控需求，下图展示了将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在 eBPF maps 中。

集成开发环境：VS Code

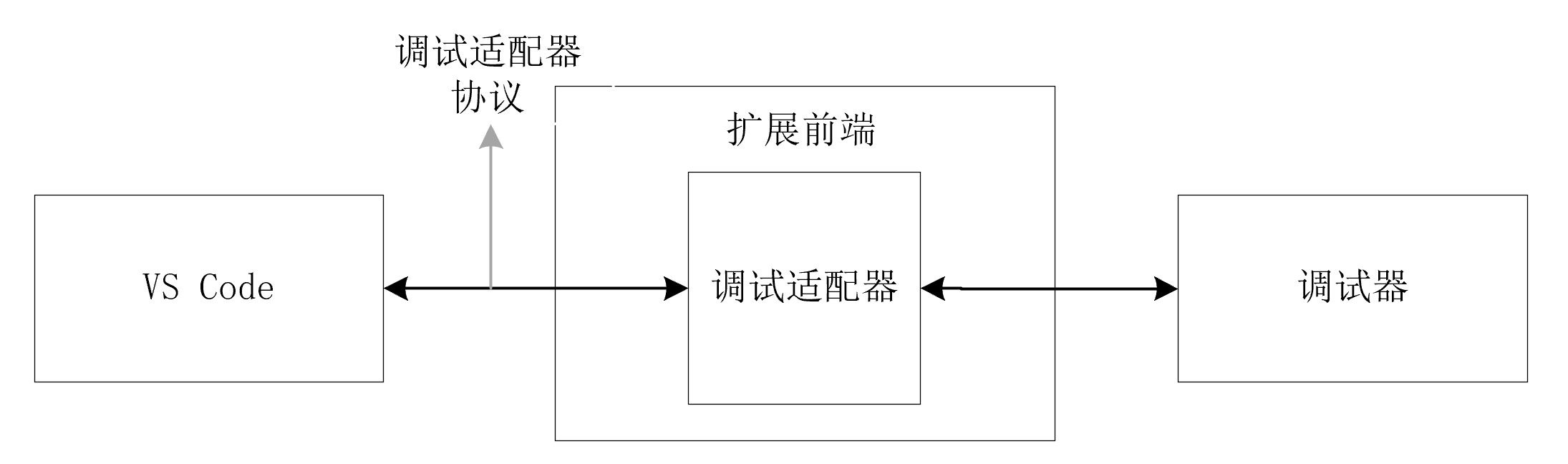
近年来，具有高度的可定制性轻量级集成开发环境，如Sublime Text、Atom和Visual Studio Code，已经迅速普及。然而，轻量级IDE对操作系统在线调试的支持非常有限。鉴于轻量级集成开发环境的日益普及，以及我们对操作系统在线调试支持的局限性的观察，我们选择使用Visual Studio Code来设计和实现一个操作系统在线调试环境。VS Code基于进程做到了物理级别的隔离，把主进程和插件进程分开，这样，这样任何插件性能不佳或者不稳定，都不会直接影响到主进程。

基于 Visual Studio Code 的在线调试方案

近年来，具有高度的可定制性的轻量级集成开发环境，如 Sublime Text、Atom 和 Visual Studio Code，已经迅速普及。然而，轻量级 IDE 对操作系统在线调试的支持非常有限。鉴于轻量级集成开发环境的日益普及，以及我们对操作系统在线调试支持的局限性的观察，我们选择使用Visual Studio Code来设计和实现一个操作系统在线调试环境。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。在线调试的优势是，本地计算机性能要求不高，无需在本地配置开发环境，便于分享协作，可以构建高效的程序开发平台。

Visual Studio Code 中的调试架构

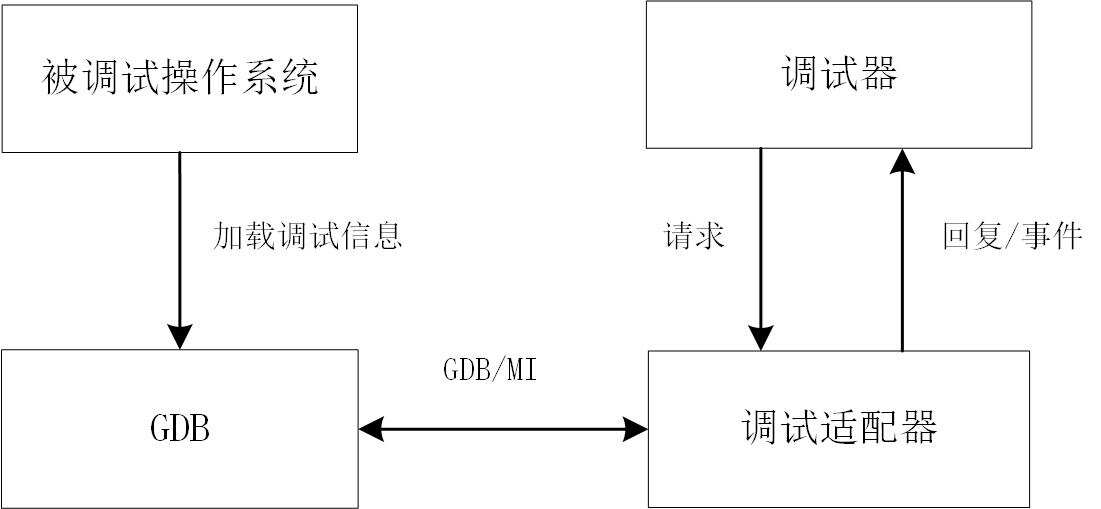
Visual Studio Code基于调试适配器协议，实现了一个原生的，非语言相关的调试器UI，它可以和任意后台调试程序通信。通常来讲，GDB等调试器不会实现调试适配器协议，因此需要调试适配器（Debug Adapter）去“适配”这个协议。调试适配器一般而言是一个独立和调试器通信的进程。



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。

系统架构总览



总框架图

上图是调试系统的一个完整的框架图，我们使用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。经过我们的调研，我们选择使用GDB这个更加强大的调试器作为中间组件，协助我们开发的源代码级调试器进行对操作系统的调试。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。

第3章 基于GDB的多进程调试方法

3.1问题概述

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，所以无论调试任何进程，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，我们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，我们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。我们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，本章的贡献点主要如下：

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。

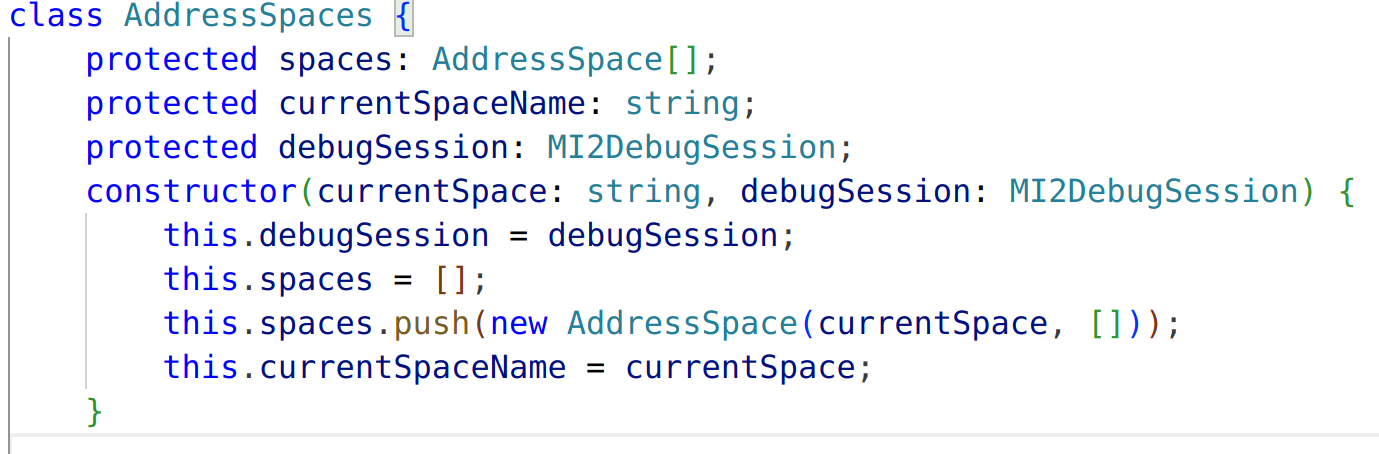
3.       设计了通过调试的方法，获取到不同用户进程的进程标识符，并根据进程标识的获取对应不同的符号表，让调试器根据符号表来进行进程的切换，从而实现了对多个用户进程的调试。

3.2断点组管理模块的设计

为了保存因为特权级切换失效的断点调试信息，设置了断点组数据结构：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。

以下是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，

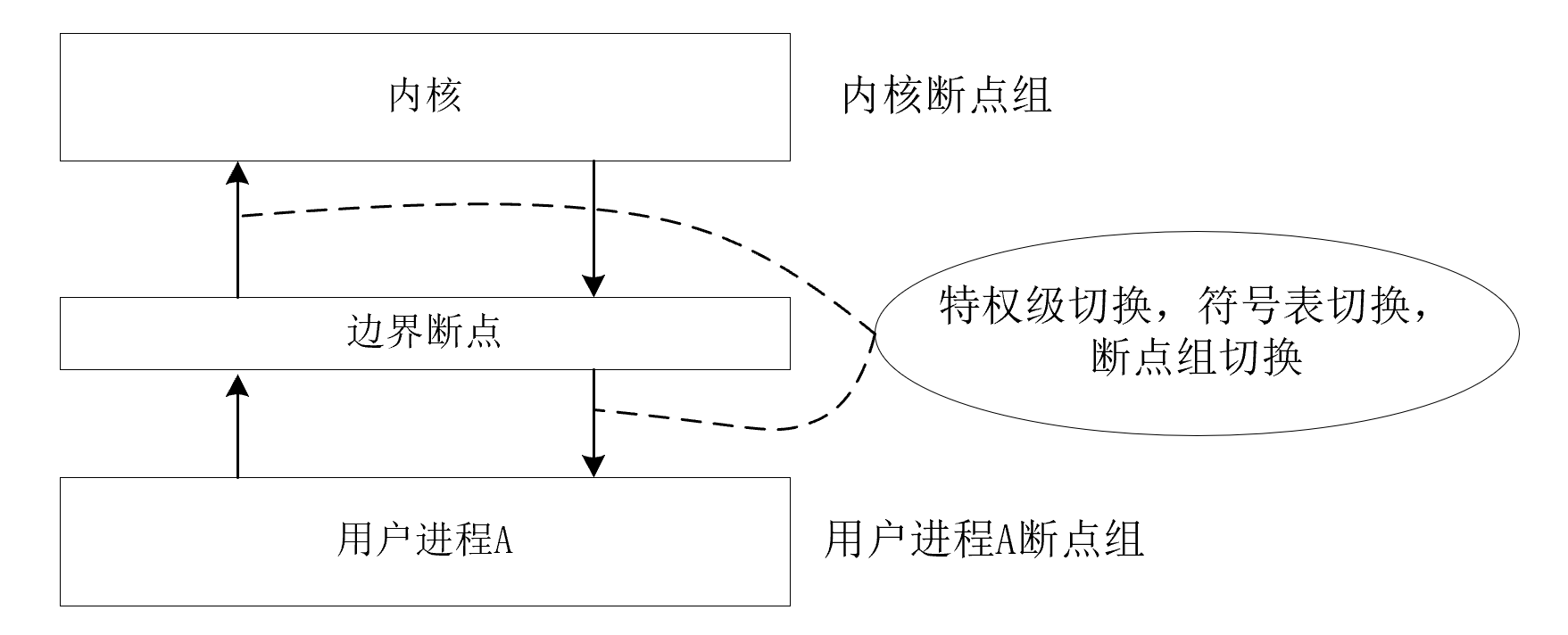


当用户在调试器中设置新断点时，调试器的断点组管理模块会先将这个断点的信息存储在对应的断点组中，然后判断这个断点所在的断点组是否为当前有效断点组。如果是，就立即激活这个断点。如果不是，那么这个断点暂时不会被激活，将会被保存到断点组中。在这种缓存机制下，用户态断点和内核态断点不会同时被激活，从而避免了内核态和用户态断点的冲突。

3.3调试器中内核态和用户态的切换

3.3.1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，接下来要让调试器知道断点组切换的时机，我们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。如下图所示，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。

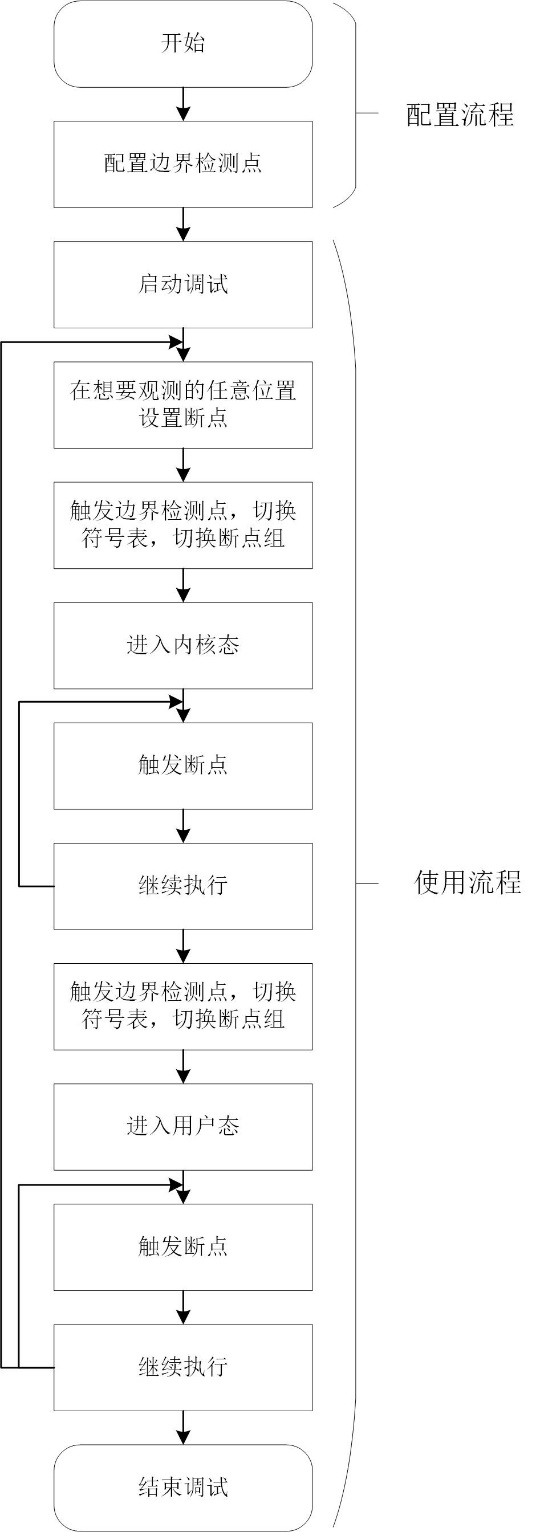


边界断点级特权及转化过程图

3.3.2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，这会导致非常浪费时间，所以我们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。

如下面流程图所展示的，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。



调试器配置、使用流程图

3.4多个用户进程的切换和调试

上述过程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，我们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，我们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，所以我们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，所以我们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。

3.4.1进程控制块信息的获取

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。

在操作系统的数据结构中，进程标识符保存在进程控制块这个数据结构中，所以我们需要获取到进程控制块信息，我们需要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。

在插件中，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。它的类型是一个数字索引签名，数字索引签名允许我们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。

protected handlers: { [index: number]: (info: MINode) => any } = {};

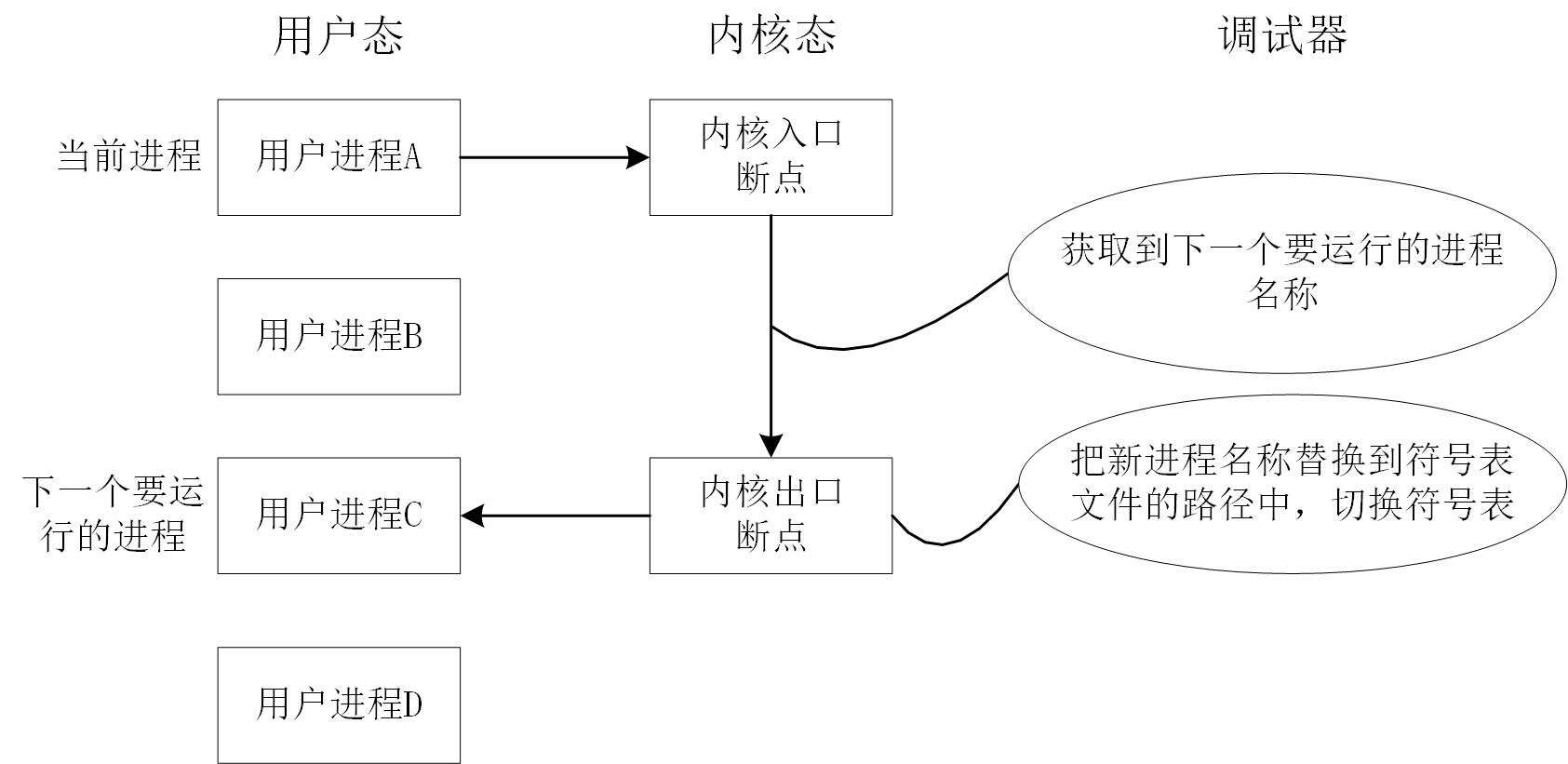
现在程序中，调试进程给GDB发送命令以后，GDB给出的返回信息中除了发送一条带有编号的表示命令的执行状态和必须携带的基本信息以外，GDB还会返回一些关于命令的其他信息或者一些警告或者提示，例如异步执行输出，异步状态输出，异步通知输出，控制台流输出，目标流输出，日志流输出，他们之中有些是没有编号的，如控制台流输出，对于这些没有编号的信息输出，源程序是没有进行储存的，在调试控制台输出以后就被清理了，如果要用的话就无法获取到，所以我们要对GDB返回消息处理函数进行修改，将我们需要的信息保留下来。

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，但是我们需要的数据有一部分就是包含在这样的数据中的，所以我们给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，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，我们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该条命令的编号。我们增加了一个存储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，我们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。

最终，我们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。

3.4.2多个用户进程的符号表切换

在获取了进程标识符以后，我们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。



3.5本章小结

在本章节中，我们为调试器添加了多进程的调试功能，首先，我们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，我们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，接下来就可以对新进程进行调试，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。

第4章 一种静态断点调试和动态跟踪结合的方法

4.1问题概述

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，我们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。

因此，我们采用了一种静态断点调试和动态跟踪结合的方法。我们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。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的多进程的断点调试和基于eBPF的动态跟踪调试结合在一起，我们进行了以下工作：

首先，我们目前调试的Rust操作系统（rCore-Tutorial-v3）并不支持eBPF和kprobe的运行，所以我们为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和用户本地的调试器通过调试适配器进程来进行通信；所以为了将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，我们为数据的输出在qemu上添加了单独的串口，并为eBPF设计了单独的处理模块，将获取到的数据同样传递到GDB中，再由GDB发送到用户本地的调试界面上；最后，我们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。

4.2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。eBPF程序通过clang编译成eBPF字节码，再通过llvm编译成机器码，这个机器码通过系统调用加载到内核代码中，通过验证以后就会在内核中等待执行。当用户编写的代码中检测的事件发生时，就会调用该eBPF程序，反馈给用户用的信息。eBPF程序加载到内核中以后，通过 kprobe 在内核函数的入口或出口设置断点，并将这些断点与特定的 eBPF 程序关联起来。当断点被触发时，与之关联的 eBPF 程序会执行，对所捕获的内核上下文或数据进行处理、过滤或记录。最后，eBPF 程序可以将结果数据传回用户空间或与其他系统共享，供后续分析或处理。这种方式使得开发者能够在不修改内核源码的情况下实现高效的内核级监控和跟踪，保障了系统的安全性和稳定性。

4.3eBPF移植

在目前我们调试的操作系统中，rCore并没有eBPF的支持，所以首先要进行的就是eBPF移植工作，让rCore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。所以，为了实现eBPF Server，我们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。

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，而我们在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。在2.2.1节中我们已经详细描述了如何做到在编译、链接时保留符号信息且操作系统仍能正常运行，在此基础上，我们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，我们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。

4.4 eBPF处理模块的设计与实现

4.1.1eBPF处理模块的设计

我们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。下表详细展示了GDB和 eBPF功能与局限，可以看出，二者形成了很好的互补：

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 | 类似tracepoint，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，这对于第二章所述的一些静态分析功能来说是必不可少的 |
| 跟踪异步函数 | 由于可以编写 帮助函数，因此较方便 | 较繁琐 |

利用 GDB 自带的远程调试功能，我们很容易就能建立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。

操作系统要能够支持eBPF技术，运行eBPF程序，这样才能实现eBPF和GDB的交互，操作系统进行编译后，运行在QEMU虚拟机上，用户开启eBPF功能，eBPF处理模块就开始运行。我们使用了一种基于串口的 GDB 与 eBPF处理模块的通信机制，eBPF处理模块需要用另一个专属的串口来和 GDB 通信。

4.4.2eBPF处理模块基于串口的通信机制及协议实现

操作系统能够支持eBPF技术以后，运行eBPF程序，这样才能实现eBPF和GDB的交互，操作系统进行编译后，运行在QEMU虚拟机上，用户开启eBPF功能，eBPF处理模块就开始运行。我们使用了一种基于串口的 GDB 与 eBPF处理模块的通信机制，eBPF处理模块需要用另一个专属的串口来和 GDB 通信。

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，我们修改了 Qemu 虚拟机的源代码，为新串口分配了 MMIO 地址和 IRQ（中断号）。同时，为了支持第二个串口的通信，我们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，我们实现了基于中断的多串口数据收发，使得 GDB 与 eBPF服务器可以通过串口通信。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。这套协议也非常适合GDB 与 eBPF处理模块进行通信，所以我们在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。

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，我们设计一个如下的消息处理流程，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。

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。

至此，GDB可以同时连接到GDB服务器和 eBPF处理模块，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>

4.4.3eBPF调试信息的适配

我们需要在调试适配器中适配 eBPF处理模块。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，如果在线 IDE 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，需要在在线集成开发环境中适配 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。

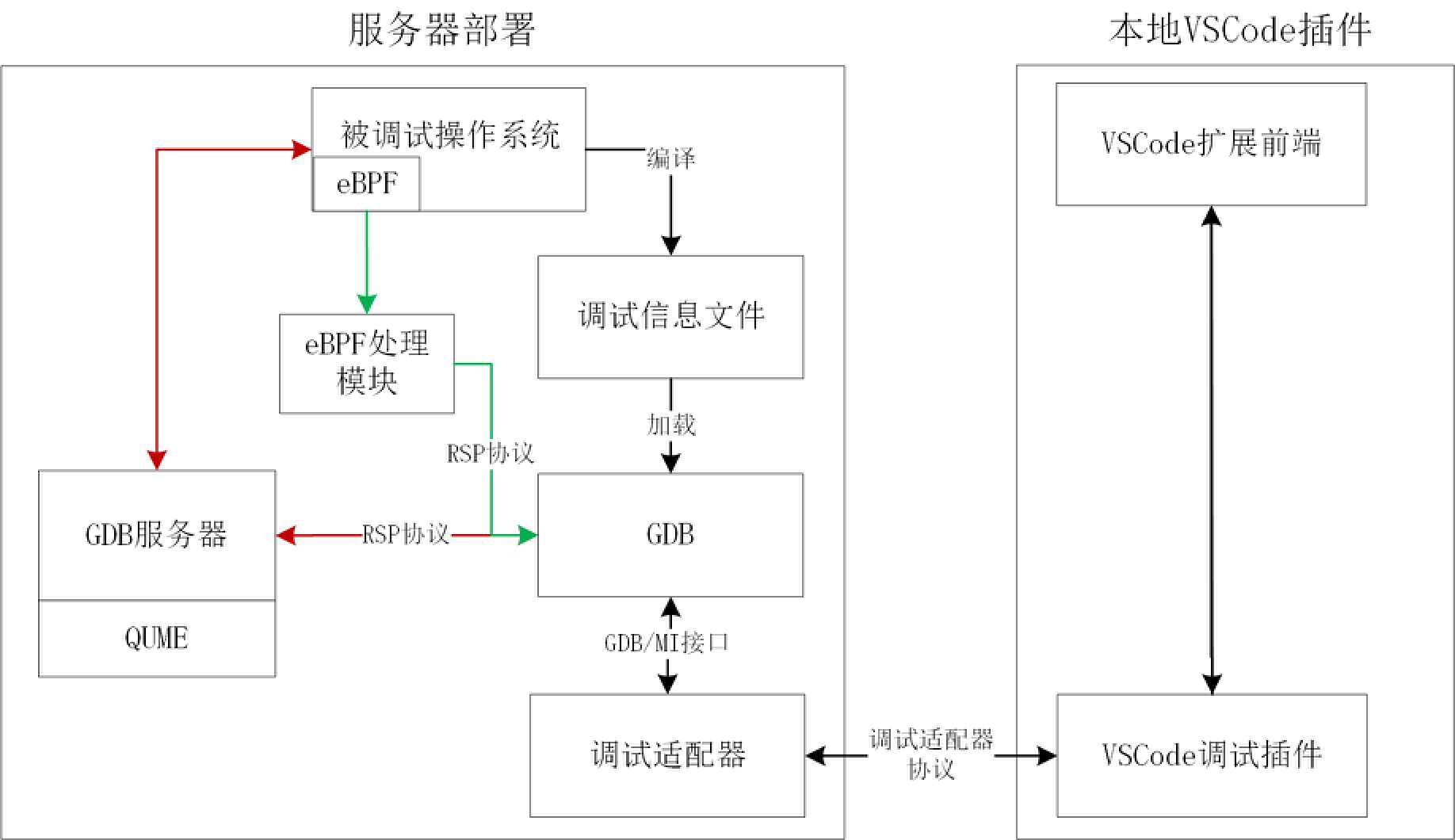
经过上述修改，GDB服务器和eBPF处理模块都能够连接到GDB上，然后通过调试适配器与调试插件进行信息传递，实现了静态断点调试和动态跟踪结合的调试技术，能够观察到更多内核数据，更大程度地满足用户地调试需求。

**本章小结**

本章主要介绍了在rCore-Tutorial-v3操作系统中eBPF移植过程中的难点和解决办法，我们编写了一个eBPF处理模块让eBPF在被调试操作系统中能够手机调试信息，在rCore-Tutorial-v3支持eBPF程序运行之后，为了将静态断点调试和动态跟踪结合到一起，我们通过串口的方式将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，我们将eBPF传递过来的信息和调试适配器进行适配，eBPF传递过来的信息与在线集成开发环境进行适配，最后展示给用户。

第5章      调试工具设计与实现

5.1整体框架设计



系统框架图

在线调试系统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。

上图中，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。

5.2服务器部分

5.2.1在线 VSCode

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IDE。这个在线IDE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。我们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。接下来我们以 rCore-Tutorial-v3操作系统为例，阐述如何获取这两类文件。

5.2.2编译

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。这是因为 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，例如内连函数，删除大量有助于调试的符号信息。因此，我们需要修改编译参数，以尽量避免编译器的优化操作。

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release, debug 两种模式编译、运行。在这两种模式中， release 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 debug 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，比较符合我们的需求。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 debug 模式进行编译。因此，我们需要修改 release 模式的配置文件，让编译器在 release 模式下也像在 debug 模式下一样关闭代码优化，保留调试信息。

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 .debug\_info 等包含调试信息的DWARF 段在链接时被忽略。这些段对调试用户态程序非常重要，因此我们需要修改链接脚本，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，我们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。

5.2.3 Qemu 和 GDB

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 gdbserver。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中运行的操作系统会启动基于 eBPF 的调试服务器。这个eBPF服务器会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服务器通过 GDB 远程串行协议 (RSP) [5]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。

5.2.4 调试适配器

调试适配器是一个独立的进程，负责协调在线 IDE 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。

如下图所示，一旦 Debug Adapter 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 Debug Adapter 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。



调试适配器和GDB、扩展前端的通信机制

5.3网页端部分

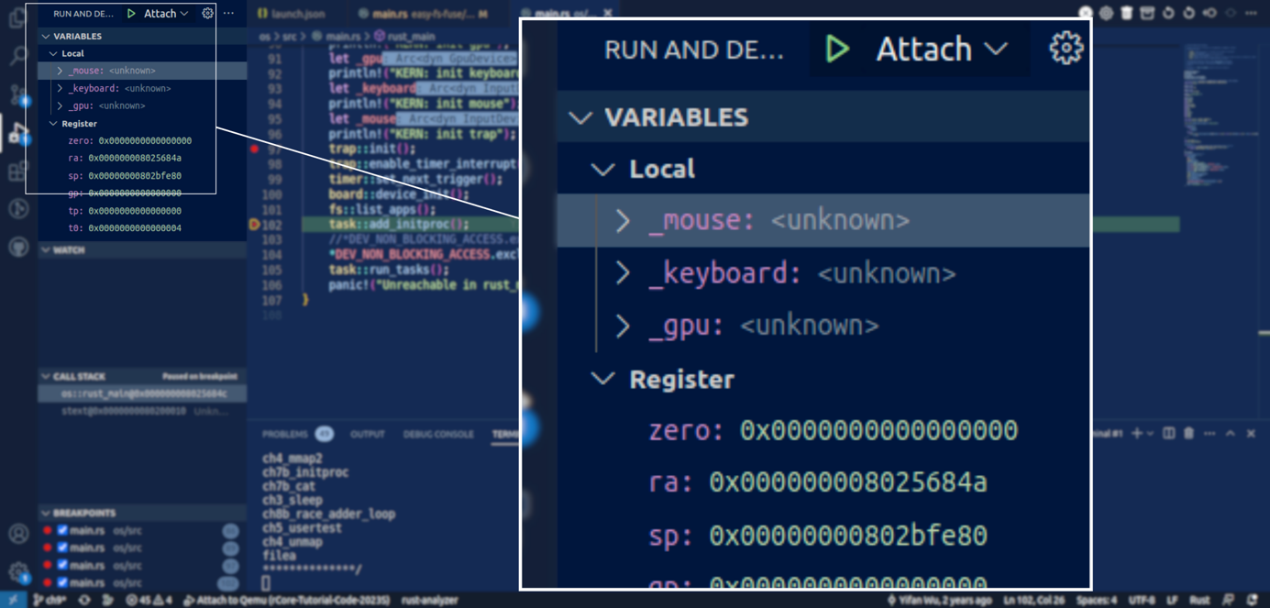
在用户浏览器上运行的在线调试环境中，一个被称作扩展前端的模块负责和和服务器上的调试适配器通信。它监听调试适配器接收和发出的消息并做出反馈，如更新用户界面、根据用户请求发送请求、响应回复和事件等。扩展前端会解析接收到的回复和事件并将需要的信息转发至用户调试界面。如果用户调试界面向扩展前端传递了某个消息，扩展前端也会将这个消息转换为请求发送给调试适配器。这种传递信息的方式有比较高的自由度。

不同类型的数据的更新策略是不一样的，具体见下表：

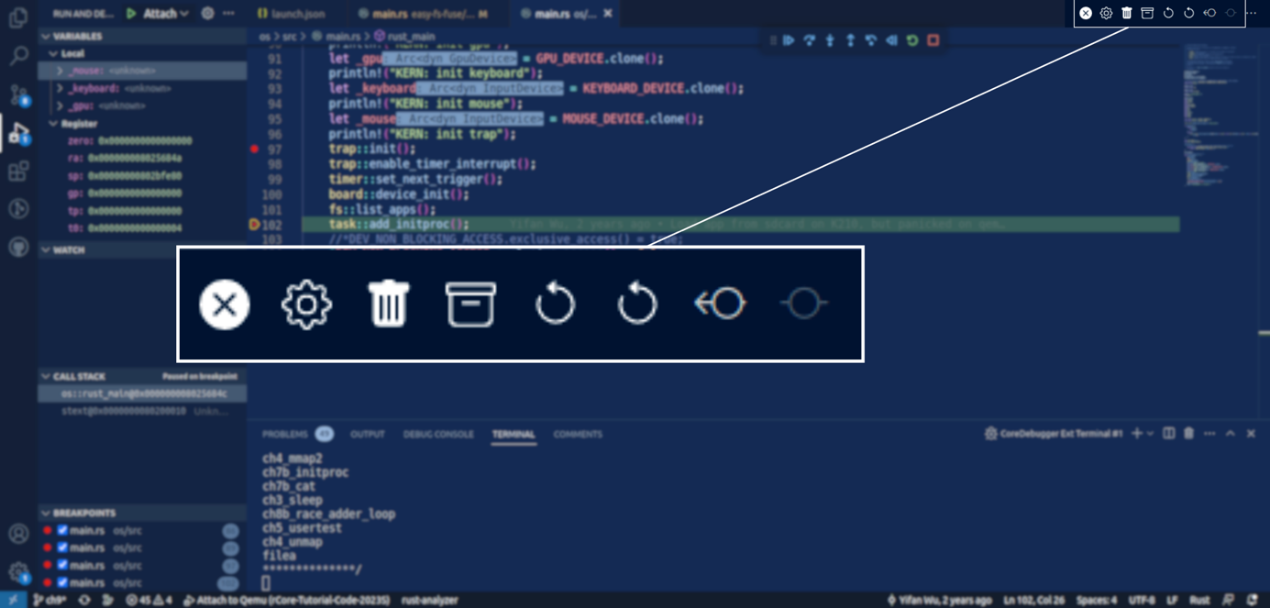
不同类型数据的更新策略

|  |  |  |
| --- | --- | --- |
| **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 |
| 内存信息 | 显示指定位置和长度的内存信息，可增删 | 触发断点、暂停、用户修改请求的内存信息时更新 |
| 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 |

我们通过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 variablesRequest 向调试适配器请求变量数据。我们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。



通过TreeView展示数据



原生样式的命令按钮

其中按钮的功能如下表所示：

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，Debug Adapter, GDB中所有断点信息 |
| **disableCurrentSpaceBreakpoints** | 令GDB清除当前设置的断点且不更改Debug Adapter中的断点信息 |
| **updateAllSpacesBreakpointsInfo** | 手动更新断点信息表格 |

此外我们还支持了VSCode自带的继续、单步等常见的调试功能按钮，如下图所示：



调试功能按钮

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，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。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。

**案例研究**

接下来我们同时运用GDB断点和eBPF调试功能，用我们的调试器调试 rCore-Tutorial 自带的 http 服务器，从中可以看出 GDB+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 Firefox 浏览器访问对应的 URL 即可获得服务器返回的静态页面。但是，如果我们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。



图5.16 tcp\_simplehttp服务器程序未返回所有请求的网页

作为对比，我们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：



图5.17 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，我们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。

5.4.1 代码简述

在开始使用调试器之前，我们阅读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 URL ，并返回 URL 对应的 HTML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并关闭服务器自身。

5.4.2 Debug过程

首先，我怀疑服务器是否接收到了所有的 HTTP 请求。因此我们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，我们打开服务器，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，我们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。

跟踪的内核函数如下

|  |  |
| --- | --- |
| 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）我们用GDB跟踪 syscall() 函数。

跟踪的用户函数如下：

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

设置完断点后，我们和刚才一样，重复访问网页六次，发现六个网页只正常打开了四个。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。

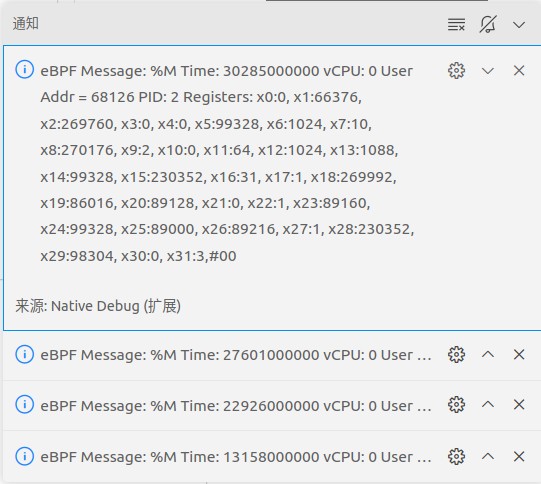


图5.18 eBPF检查点触发，弹出调试信息

于是，我们又怀疑，原因可能是我们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，我们在sys\_accept 、 write 函数设置GDB断点，这样每次处理accept时 OS都会停下来，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。

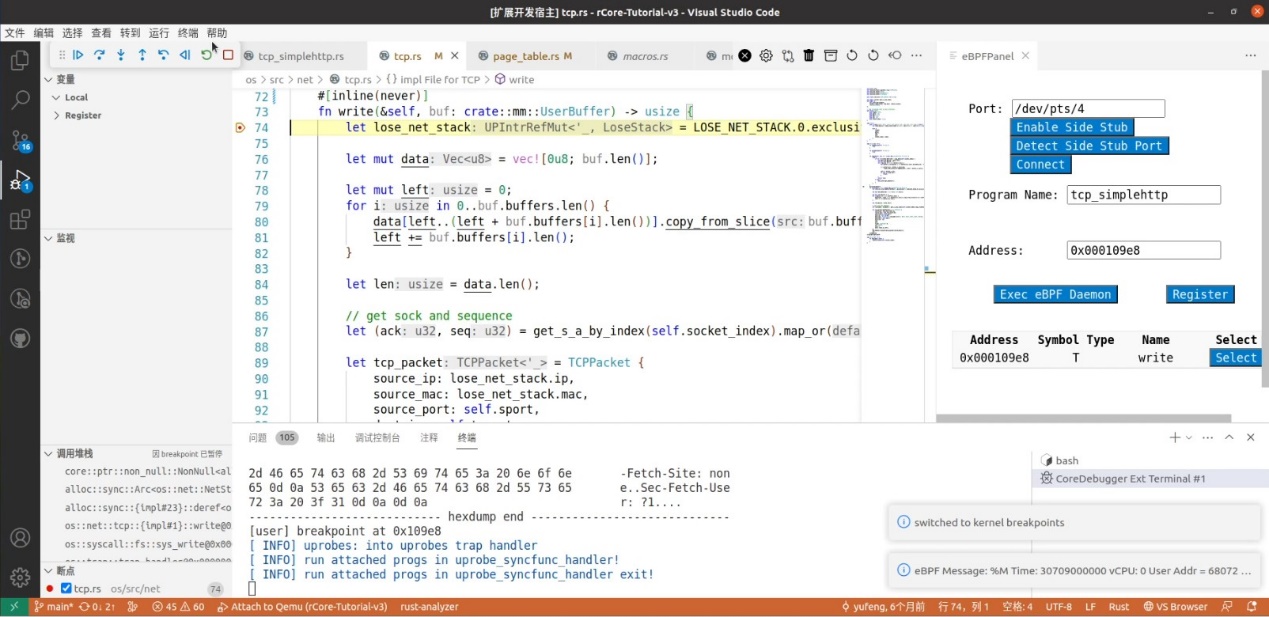


图5.19 `write`函数的GDB断点触发

我们又想到，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，我们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 ，导致前一个连接没有被正确关闭，所以后一个连接无法成功建立。

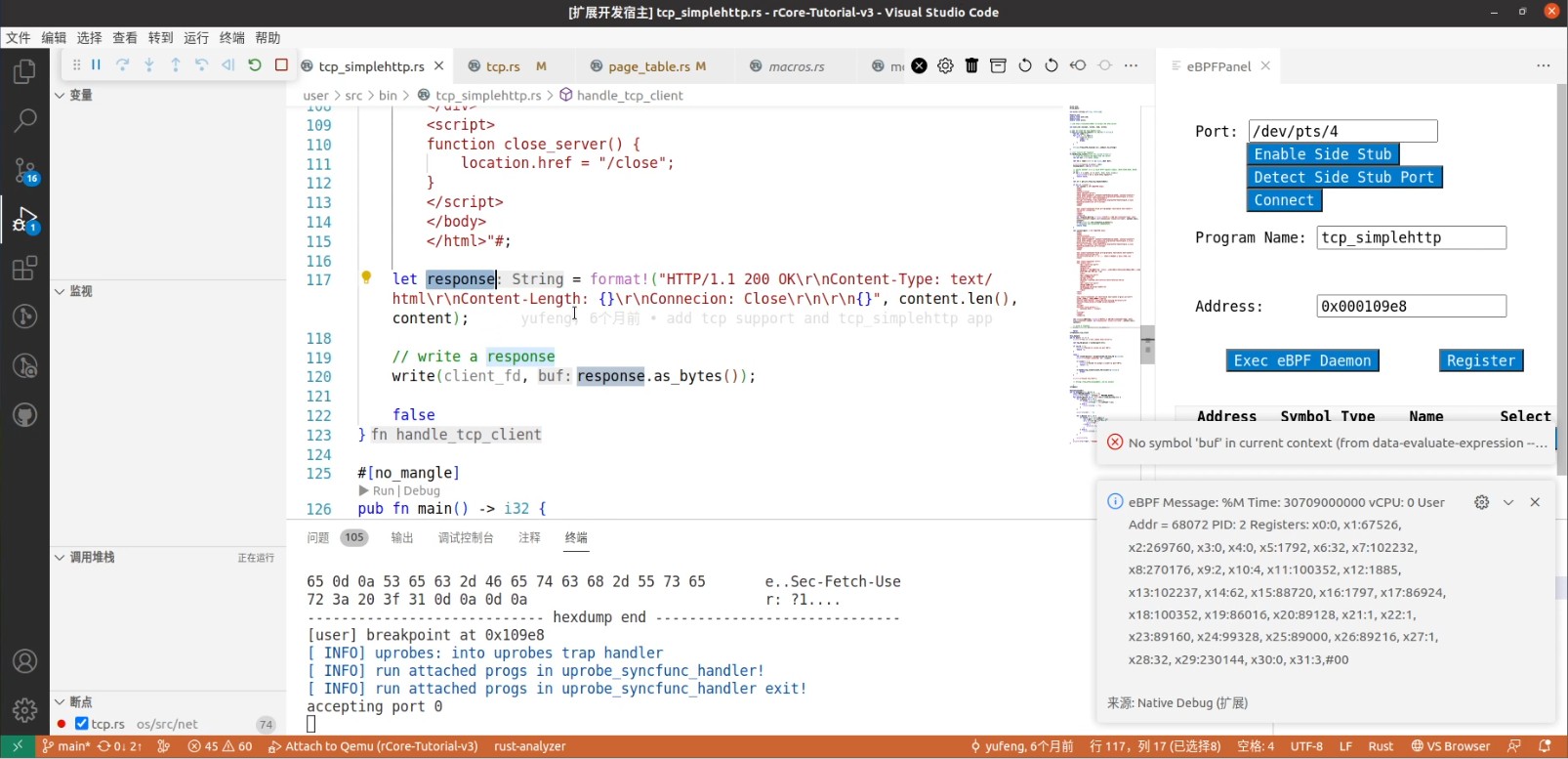


图5.20 出错的代码

这样，结合 eBPF 和 GDB 两种调试手段，我们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Debug的过程中，我们不需要出于调试目的而改动任何其他的源代码。

参考文献

[1]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.

[2]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.

[3]殷绍剑. 嵌入式多线程远程调试器研究与实现[D].电子科技大学,2013.

[4] W. Chen, D. Han, D. Tang. et al.A model of remote debugge r supporting multiple types of connection[Cl. 2011 Interna ational Conference on Electronics. Communications and Control 9-11 Sept.2011,Piscataway, NJ,USA,2011,642-5

[5]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.DOI:10.19314/j.cnki.1009-234x.2020.02.004.

[6]王敬宇,范昊.一种面向通用众核CPU的软件调试器设计[J].计算机工程与科学,2014,36(10):1854-1859.

[7]王琼. 嵌入式Linux软件调试技术研究与实现[D].华中科技大学,2007.

[8] T. -H. Chang, S. -C. Hou and I. -J. Huang, "A unified GDB-based source-transaction level SW/HW co-debugging," 2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), 2016, pp. 506-509, doi: 10.1109/APCCAS.2016.7804015.

[9]K. Goossens, B. Vermeulen, R. Van Steeden, and M Bennebroek, "Transaction-based communication- centric debug," in 2007. NOCS 2007. First International Symposium on Networks-on-Chip2007,pp.95-106.

[11]S. Pasricha and N. Dutt, On-Chip Communication Architectures: System on Chip Interconnect, 1st ed.2008.

[12] A. M. Gharehbaghi and M. Fujita, "Transaction-based debugging of system-on-chips with patterns," in 2009. ICCD 2009. IEEE International Conference on Computer Design, 2009, pp.186-192.

[13]N. J. Zaidenberg and E. Khen, "Detecting Kernel Vulnerabilities During the Development Phase," 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, 2015, pp. 224-230, doi: 10.1109/CSCloud.2015.91.

[14]Tobias Holl, Philipp Klocke, Fabian Franzen, and Julian Kirsch. 2018. Kernel-Assisted Debugging of Linux Applications. In Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium (ROOTS '18). Association for Computing Machinery, New York, NY, USA, Article 3, 1–9.

[15]Meghana D K and J. G. Reddy, "Cloud-based approach to increase the performance of execution of binary by using the separate debug file," 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), 2016, pp. 743-746, doi: 10.1109/ICATCCT.2016.7912098.

[16] Michael J. Eager, Eager Consulting, “Introduction to the DWARF Debugging Format" April 2012.

[17]Bill Karakostas "Towards Autonomic Cloud Configuration and Deployment Environments" International Conference onCloud and Autonomic Computing ,IEEE,2014

[18]Robayet Nasim, Andreas J. Kassler, "Deploying OpenStack:Virtual Infrastructure or Dedicated Hardware",38th Annual International Computers. software and Applications Conference Workshops IEEE 2014.

[19] Bojan Mihajlovic, Zeljko Zilic, and Warren J. Gross. Dynamically instrumenting the QEMU emulator for linux process trace generation with the GDB debugger. ACM Trans. Embedd. Comput. 2014. Syst. 13,5s Article 167 (November 2014), 18 pages.

[20] Javier Alonso, Michael Grottke, Allen P. Nikora, and h ishor S. Trivedi. An empirical investigation of fault repairs and mitigations in space mission s system software. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems an d Networks (DSN'13). 2013. 1-8.

[21]Matteo Bordin, Cyrille Comar, Tristan Gingold, Jerom e Guitton, Olivier Hainque, Thomas Quinot, Julien Delange, Jerome Hugues, and Laurent Pautet. Couverture: An innovative open framework for coverage analysis of safety critical applications.Ad a User J. 2009.30,4,248-255.

[22]赵辉,史永辉,张静.基于虚拟环境的嵌入式软件调试工具[J].科技展望,2016,26(11):126-127.

[23]Fan, H.; Li, K.; Li, X.; Song, T.; Zhang, W.; Shi, Y.; Du, B. CoVSCode: A Novel Real-Time Collaborative Programming Environment for Lightweight IDE. Appl. Sci. 2019, 9, 4642.