北京工商大学

硕士学位论文中期报告

论 文 题 目：面向 Rust 操作系统的多进程调试方法与系统实现

研究生姓名：张露元

学　　号：2130072064

专　　业：计算机技术

导 师 姓 名：吴竞邦

论文工作的
起 止 时 间：2022.09-2023.12

2023 年　12 月 28　日　填写

一、课题目前进展情况（已完成的研究工作与研究成果，与开题报告所定研究内容和进展是否相符）

根据实际研究的进展，对开题报告的研究内容做了改变，确定论文主要研究内容为：基于 QEMU 和 GDB，提供多进程调试方法，支持对 Rust 操作系统的多进程调试；基于 eBPF 和 GDB，提供一种静态断点的调试和动态跟踪的方法，支持对多进程的跟踪和监测；基于 VSCode 构建远程开发环境，开发调试器，支持断点调试与进程监测的功能结合。

1. 基于 GDB 的多进程调试方法

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，除此以外，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。在操作系统进行编译以后，就会创建不同的符号表，每个用户进程和特权级都会对应不同的符号表，而在操作系统运行过程中会进行特权级的切换和多个用户进程的切换，这些操作就会对应符号表的切换，而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，符号表的切换就会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，如图 1 所示。

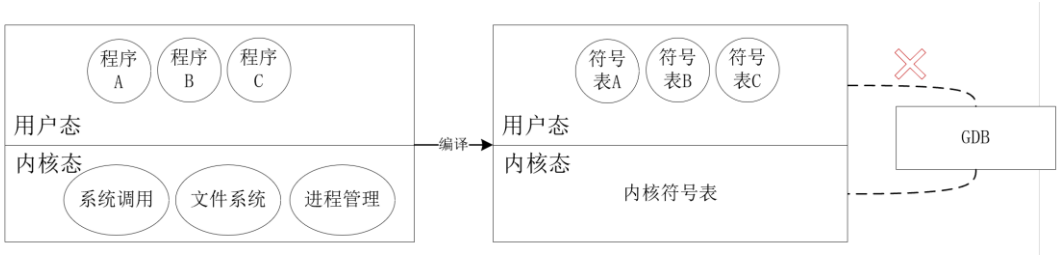

图 1 编译符号表

为了保存因为特权级切换或进程切换失效的断点调试信息，我们在调试器中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，这样在被调试操作系统进行进程切换或者是特权级切换的时候，就会缓存原进程的调试信息，再加载新进程的调试信息，从而达到调试不同特权级下的进程的目的。

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。

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。

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。

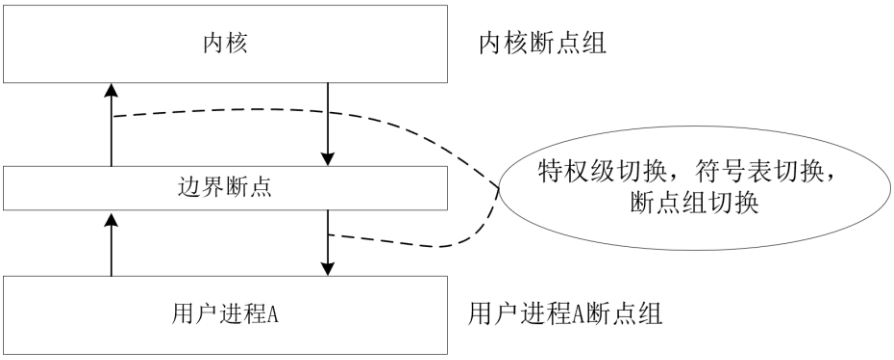

图 2 特权级切换

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。

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，如图所示。

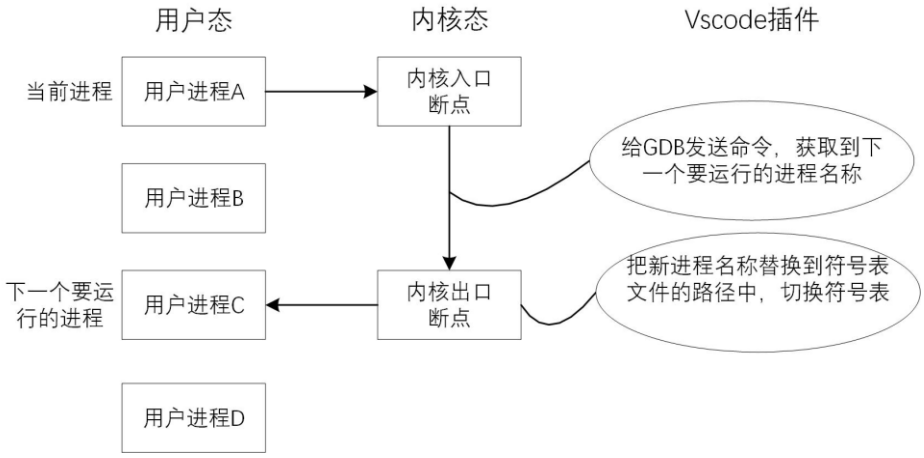

图 3 多进程切换

2. 一种静态断点的调试和动态跟踪的方法

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。

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。

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。

接下来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。表 1 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 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	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。

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。

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。

最后，需要在在线 IDE 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。

3. 支持 Rust 语言的源代码级操作系统调试工具

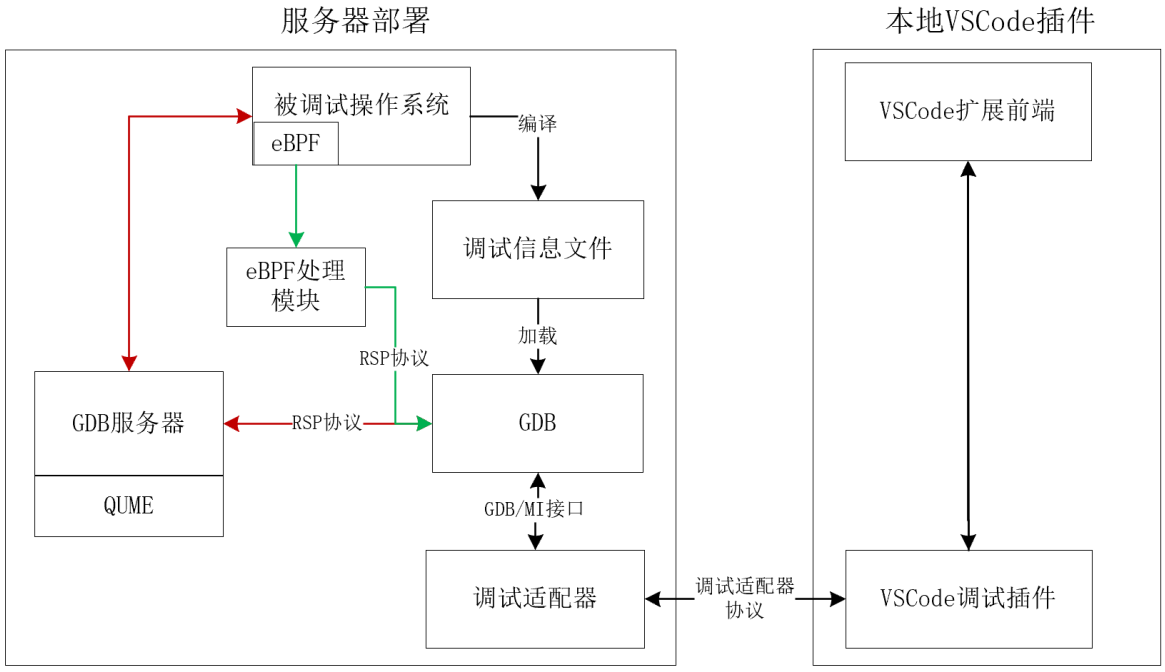

图 4 系统框架图

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。

图 4 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。

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。

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[4]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。

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。

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

图 6 调试适配器和 GDB、扩展前端的通信机制

网页端部分

在用户浏览器上运行的在线环境中，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：

表 2 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

以下是在用户界面实现的功能按钮：

表 3 功能按钮

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

下图是人机交互界面的整体展示，能够对代码进行十分方便的展示和浏览，使用者也可以在调试界面左侧的信息展示部分看到被调试程序的各种信息，还可以在源代码直接手动设置断点，断点信息会随时更新到信息展示的部分，右上角是系统的调试功能按钮，能够让使用者十分方便的完成各种调试需求。

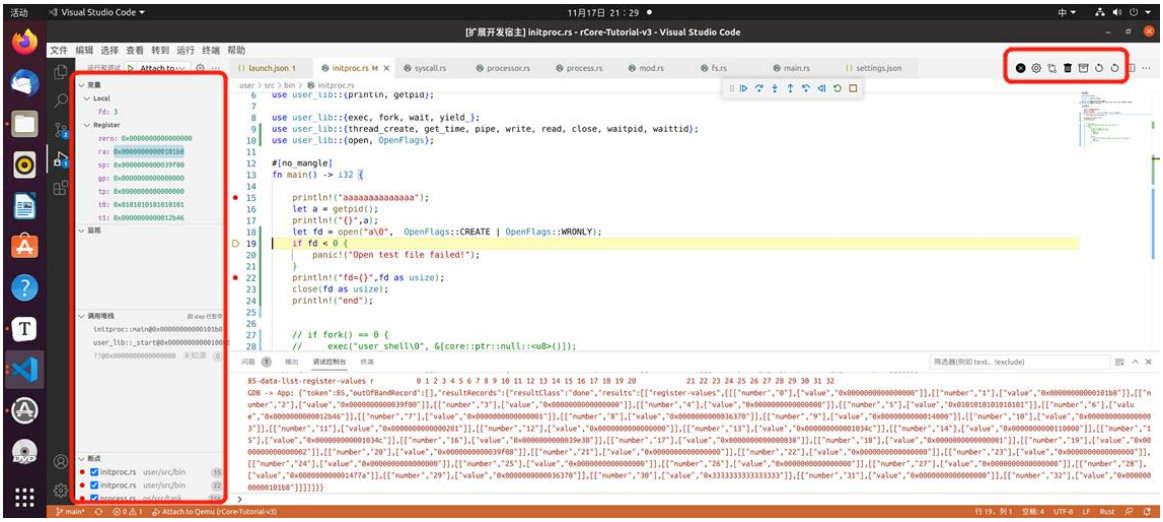

图 6 系统实现界面

二、课题研究存在的问题与难点，拟采用的解决方案。

1. 在调试器的开发过程中，我们也希望对调试器插件进行一个调试，code-debug 插件是分为两部分的，扩展和调试适配器，这两部分是由两个进程来控制，所以如果调试的话应该是启动两个调试配置，一个是 launch extension，另一个是 server。

前者是调试 extension 的部分，更具体地说是 extension.ts 文件，用它调试就会启动一个新窗口（扩展开发宿主)；后者是调试调试适配器的部分，也就是除了 extension.ts 文件地其他文件，这部分的调试需要进行一个配置，在 code-debug 中的 launch.json 已经配置好了，是一个叫做 code-debug sever 的调试配置，里面有一个 4711 的端口号，启动这个配置以后，他会监听这个端口号，所以，在我们要调试的项目（rCore-Tutorial-v3）中，也要添加一个 "debugServer": 4711,的配置，使两者可以传递信息。

2. GDB 对 rust 语言支持的局限性，很多复杂的数据结构都不支持查看。

要从被调试操作系统 rCore 入手，在 rCore 中找到需要的变量，解析他的数据结构，追踪到他保存最基础的数据结构的地方,查看他的首地址以及所占内存大小,然后让 GDB 去读取该内存中的数据，并展示出来。

3. 进程标识符包含在进程控制块中，所以要获取进程控制块，但是如问题 1 所说，复杂的数据结构不支持查看，同时，插件对 GDB 返回信息的处理功能并没有把我们想要的而数据返回，所以要对信息的处理流程进行完善，主要使用了以下方法：进程控制块是内核中的数据结构，所以只能在程序运行到内核态时进行获取，需要在有进程控制块变量的地方设置断点，然后给 GDB 发送 print 命令，让 GDB 返回相应的信息。该命令返回的信息中就存在 token 为命令编号和 token 为 undefined 的情况，所以需要像上述方法一样，通过发送的命令的编号，在保存 MINode 信息的数组中进行查找，然后把相应编号的几条信息单独拿出来再进行处理。获取到的信息被分成了几条返回到了插件，信息类型是字符串类型，并且里面的内容不是很规则，所以采取的方法是使用正则表达式进行分割，根据进程控制块中的变量，从返回的信息中获取对应的值。由于现在的进程控制块相关信息都是字符串类型，不好操控，所以创建了一个新的类，来保存进程控制块的信息，包含一些基本信息，如其中的变量名称，类型，以及它的值。对于含有指针的变量，返回信息中只会有储存真正信息的地址，需要先获取地址以后，再通过插件中获取内存的函数，根据获取到的地址去读取内存中的内容，从而获取到真正的值。

三、课题下一步的计划与安排（科研论文工作是否能达到预期研究成果、是否留有足够的时间完成全部论文工作）

根据目前进度和研究结果，科研论文工作能够达到预期研究成果，目前留有足够的时间完成全部论文工作，课题下一步的具体时间安排如表所示:。

计划	时间	进度
文献调研	2021.10-2022.10	已完成
开题报告	2022.11-2022.12	已完成
基于 GDB 的多进程调试方法	2022.10-2023.06	已完成
一种静态断点调试和动态跟踪结合的方法	2023.06-2023.10	已完成
系统实现	2022.10-2023.10	已完成
撰写论文	2023.10-2024.04	在进行
论文答辩	2024.05	/

四、已正式发表论文的情况

论文：

1. 吴竞邦，张露元等， VANET 中基于分布式 TDMA 的 MAC 协议实时测试平台的设计与实现；J. Wu, L. Zhang and Y. Liu, "On the Design and Implementation of a Real-Time Testbed for Distributed TDMA-Based MAC Protocols in VANETs," in IEEE Access, vol. 9, pp. 122092-122106, 2021, doi: 10.1109/ACCESS.2021.3108346.（已录用）
2. 吴竞邦，陈熙禹，张露元等，车联网秘密共享中的一种多秘密信誉调整方法；Jingbang Wu, Xiyu Chen, Luyuan Zhang, Shufen Zhou, Daoshun Wang, "A Multi-Secret Reputation Adjustment Method in the Secret Sharing for Internet of Vehicles", Security and Communication Networks, vol. 2022, Article ID 1413976, 13 pages, 2022. https://doi.org/10.1155/2022/1413976（已录用）

专利：

3. 支持跨内核态和用户态的操作系统源代码级调试方法；吴竞邦，张露元（已提交）

五、导师意见

导师签名：　　年　　月　　日

北 京 工 商 大 学

硕士学位论文中期检查表

年级 **2021 级** 专业名称 计算机技术 姓名 张露元 指导教师 吴竞邦

<table>
<tr><td>论文题目
（开题题目）</td><td colspan="4">支持 rust 语言的源代码级内核调试工具</td></tr>
<tr><td>变更后论文题目</td><td colspan="4">面向 Rust 操作系统的多进程调试方法与系统实现</td></tr>
<tr><td rowspan="6">检查小组成员</td><td>姓 名</td><td>职 称</td><td>工 作 单 位</td><td>本人签名</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td rowspan="2">检查小组评议</td><td colspan="4">成 绩（百分制）</td></tr>
<tr><td colspan="4">评议意见：

检查小组组长签名： 年 月 日</td></tr>
<tr><td>学院意见</td><td colspan="4">主管院长签名： 年 月 日</td></tr>
</table>