

泉城实验室学习总结

Syscall组 陈志扬

目录

泉城实验室学习总结

目录

1 工作内容概要

2 提升调试插件的适用范围

2.1 调试器插件原理概述

2.2 基于 Debug Adapter 的调试工具设计与实现

2.3 解决内核态用户态的 GDB 断点冲突

2.4 获取更多调试信息

2.5 适配 ArceOS 和 Starry (内核态)

2.5.1 概述

2.5.2 RISC-V+QEMU+ArceOS unikernel

2.5.3 x86-64+QEMU+ArceOS unikernel

2.5.4 适配Starry

3 优化使用体验

3.1 概述

3.2 简化安装流程

3.3 整理代码

3.4 提供更加友好的帮助信息

致谢

1 工作内容概要

方便的源代码级调试工具，对监测程序运行状态和调试程序十分重要，尤其是复杂的内核代码。现有 Rust 操作系统开发领域缺少一款源代码级调试工具，导致相关实验环境搭建成本高，上手难度大，不利于 Rust 操作系统内核学习与开发工作。针对上述需求，我构建了一款支持 Rust 操作系统内核开发的源代码级调试工具，该工具具备如下特征（1）基于 QEMU 和 GDB，支持跨内核态和用户态的源代码跟踪调试；（2）基于 eBPF，支持开发板上跨内核态和用户态的性能分析检测；（3）基于 VScode 构建了远程开发环境，支持断点调试与性能检测的功能结合。

在这个月的泉城实验室集中开发中，我主要的工作是提升了调试插件的适用范围，支持了 Qemu 上 ArceOS、Starry 的内核态调试，且支持了 x86_64 和 RISC-V 两个平台。同时，我对调试器插件的源代码做了很多的整理工作，方便未来其他同学进行修改。同时我也发展了一些调试器的真实用户，他们给我很多宝贵的反馈意见，我根据这些反馈意见大大优化了插件的安装、使用体验。

2 提升调试插件的适用范围

2.1 调试器插件原理概述

在代码执行流上，和一般的应用程序相比，操作系统的特点在于频繁的特权级切换。能否跨特权级跟踪操作系统代码的执行（尤其是及用户态、内核态的系统调用交互）是衡量操作系统调试器调试能力的重要指标。然而，如果用户想利用 GDB 同时跟踪内核态和用户态代码，在 GDB 中同时设置内核态和用户态代码的断点，会发现由于特权级切换时页表进行了刷新，地址空间改变，而 GDB 断点又是依赖于地址空间的，所以内核态，用户态 GDB 断点无法全部生效。因此，原生 GDB 不具备内核态与用户态方便的切换跟踪功能。

为了让GDB能够同时跟踪内核态和用户态代码，我们设计了一套名为“断点组切换”的机制，其核心的原理是，将用户设置的GDB断点按断点所在的地址空间分为若干组，在任意时刻下，只有当前地址空间对应的断点组是生效的。如果调试器检测到地址空间发生了切换，就会令 GDB 立即切换到切换后的地址空间对应的断点组和符号表，从而使得在任意时刻下，GDB 都只设置当前运行的指令所在的地址空间的断点，这样就避免了断点失效的问题，保证了用户设置的用户态，内核态的断点均可生效。

实现这套机制的难点在于，地址空间切换后，断点组要立即进行切换，然而 GDB 又如何检测到地址空间是否切换？而且，GDB 只有在断点触发，被调试OS因断点触发而暂停之后，才可以进行断点设置、符号表切换等工作（即 GDB 在大部分情况下进行的都是静态调试），而 OS 的运行又是动态的，瞬息万变的，那么又如何进行断点组，符号表的动态切换？我们将在本章中详细讨论这些问题，以及“断点组切换”机制具体的代码实现。

2.2 基于 Debug Adapter 的调试工具设计与实现

如前文所述，我们通过断点组和符号表的动态切换来保证内核态，用户态断点均能正确触发。然而，我们在实现“断点组切换”机制时发现，如果想通过修改 GDB 源代码的方式来实现这种灵活的切换机制，不仅修改 GDB 源码的工作量很大，而且 GDB 和 VSCode 代码编辑器的交互（我们最终的目的是想在 VSCode 等集成开发环境里用 GUI 进行 OS 调试）也要从头编写。因此我们查找资料，想寻找一种更高效的“断点组切换”实现方式。我们发现，在 VSCode 上，VSCode 一般并不直接和各种调试器进行信息传输和控制交互，而是通过一个叫调试适配器（Debug Adapter, DA）的“中间人”将不同调试器提供的不同接口协议（比如，GDB 提供的 GDB/MI 接口和 LLDB 提供的 LLDB/MI 接口）都通过 Debug Adapter 抽象成统一的调试适配器协议（Debug Adapter Protocol, DAP），而 VSCode 已经完整支持调试适配器协议，还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，所以开发者只需要以 VSCode 插件的形式实现一个能和自己的调试器进行交互的调试适配器，并且将调试适配器对接到 VSCode 就可以了。

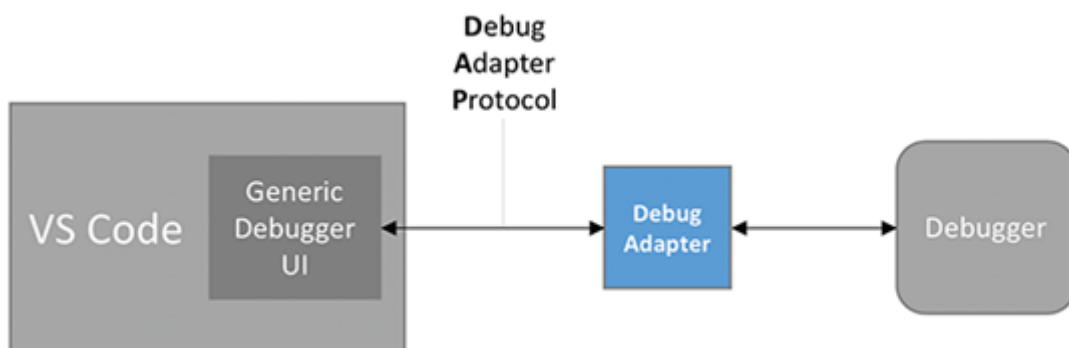


图2.1 基于Debug Adapter的调试架构

在研究了调试适配器协议和GDB的 GDB/MI 接口后，我们认为，编写一个对下通过 GDB/MI 接口控制 GDB，对上通过 DAP 协议和 VSCode 调试界面进行交互的调试适配器，不仅能实现我们的“断点组切换”功能，而且代码抽象层次高，实现后续各种功能（包括对用户界面的扩展，对 eBPF 的支持等）都比较高。于是，我们采用了这种实现方案，在 Debug Adapter 中实现“断点组切换”。

2.3 解决内核态用户态的 GDB 断点冲突

前面已经讲到，要实现操作系统调试功能的关键问题在于同时设置内核态、用户态的断点，但是我们在“双页表”的OS（比如 rCore-Tutorial-v3）上实际测试后发现，用户态、内核态的断点设置是冲突的。这是由于 GDB 根据内存地址设置断点，因此是依赖于页表、快表的，但是 OS 从内核态切换到用户态时执行了 risc-v 处理器的 `sfence.vma` 等指令，使得 TLB 刷新成用户进程的页表。所以，如果在 OS 运行在内核态时，令 GDB 设置用户态程序的断点，这个用户态的断点无法被触发。

解决这个问题的核心思路是，缓存设置后会造成异常情况的断点，待时机合适再令 GDB 设置这些断点。在用户态运行时，缓存内核态断点；在内核态运行时，缓存用户态断点。为此，我们在 Debug Adapter 中新增了一个断点组管理模块。

断点组管理模块用一个词典缓存了用户要求设置的（包括内核态和用户态）所有断点。词典中某个元素的键是内存地址空间的代号，元素的值是这个代号对应的断点组，即这个内存地址空间里的所有断点。当任何一个断点被触发时，Debug Adapter 都会检测当前触发的这个断点属于哪个断点组。我们将这些包含了最新触发断点的断点组称为当前断点组（Current Breakpoint Group）。

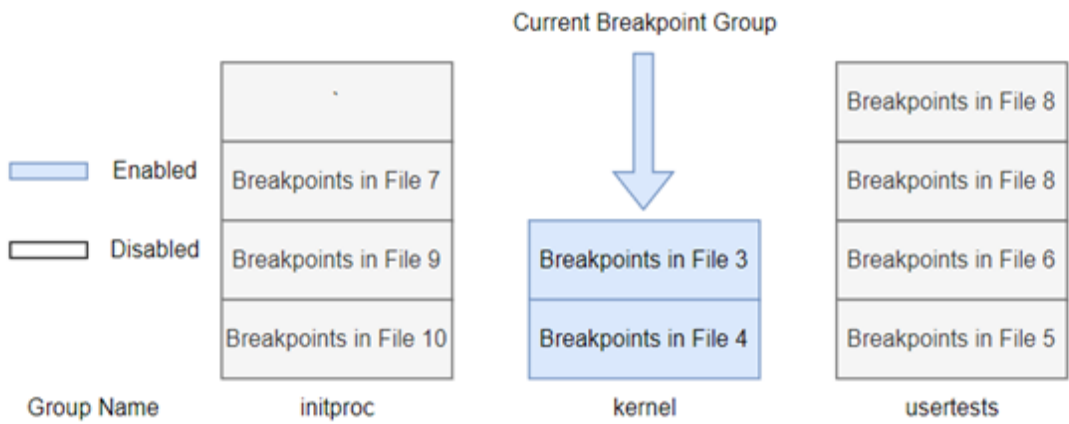


图2.2 断点组

如上一节所述，我们通过 Debug Adapter 来实现这个机制。Debug Adapter 和 GDB 的交互比较简单，我们只要实现以下四个功能即可：

1. Debug Adapter 通过向 GDB 发送 `set-breakpoint` 设置一个断点，GDB 返回断点设置的结果。
2. Debug Adapter 通过向 GDB 发送 `add-symbol-file` 添加符号表，GDB 返回符号表添加的结果。
3. Debug Adapter 通过向 GDB 发送 `remove-breakpoint` 去除一个断点，GDB 返回断点去除的结果。
4. Debug Adapter 通过向 GDB 发送 `remove-symbol-file` 去除符号表，GDB 返回去除的结果。

然而，Debug Adapter 和 VSCode 的交互略显复杂。Debug Adapter 和 VSCode 用调试适配器协议进行交互，该协议主要由以下三个部分组成：

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UI对Debug Adapter的请求；
3. Responds 定义了Debug Adapter对请求的回应。

当用户在 VSCode 编辑器中设置新断点时，VSCode 会向Debug Adapter 发送一个请求设置断点的 Request：

```

onDidSendMessage: (message) => {
    if (message.command === "setBreakpoints"){
//如果Debug Adapter设置了一个断点
        vscode.debug.activeDebugSession?.customRequest("update");
    }
    if (message.type === "event") {
        //如果（因为断点等）停下
        if (message.event === "stopped") {
            //更新寄存器和断点信息
            vscode.debug.activeDebugSession?.customRequest("update");
        }
    }
}

```

Debug Adapter 中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不是当前断点组，如果是的话，就令 GDB 当即设置这个断点。反之，如果不是，那么这个断点暂时不会令 GDB 设置（由于API名比较冗长，为了不占用过多篇幅，我们用截图来展示这份关键代码）：

```

//src/mibase.ts-MI2DebugSession-setBreakPointsRequest
protected setBreakPointsRequest(
    response: DebugProtocol.SetBreakpointsResponse,
    args: DebugProtocol.SetBreakpointsArguments
): void {
    this.miDebugger.clearBreakPoints(args.source.path).then(
        () => {
            //清空该文件的断点
            const path = args.source.path;
            const spaceName = this.addressSpaces.pathToSpaceName(path);
            //保存断点信息，如果这个断点不是当前空间的（比如还在内核态时就设置用户态的
            //断点），暂时不通知GDB设置断点。
            //如果这个断点是当前地址空间，或者是内核入口断点，那么就通知GDB立即设置断点
            if ((spaceName === this.addressSpaces.getCurrentSpaceName())
                || (path === "src/trap/mod.rs" && args.breakpoints[0].line === 30))
            {
                // TODO rules can be set by user
                this.addressSpaces.saveBreakpointsToSpace(args, spaceName);
            }
            else {
                this.sendEvent({
                    event: "showInformationMessage",
                    body: "Breakpoints Not in Current Address Space. Saved",
                } as DebugProtocol.Event);
                this.addressSpaces.saveBreakpointsToSpace(args, spaceName);
                return;
            }
        }
    );
    //令GDB设置断点
    const all = args.breakpoints.map((brk) => {
        return this.miDebugger.addBreakPoint({
            file: path,
            line: brk.line,
            condition: brk.condition,
            countCondition: brk.hitCondition,
        });
    });
    //...
    //更新断点信息
    this.customRequest("update", {} as DebugAdapter.Response, {});
}

```

图2.3 断点组缓存代码

在这样的缓存机制下，GDB 不会同时设置内核态和用户态断点，因此避免了内核态用户态的断点冲突。接下来需要一个机制，在合适的时机进行断点组的切换，保证某个断点在可能被触发之前就令 GDB 设置下去。显然，**利用特权级切换的时机是理想的选择**。因此，我们令 Debug Adapter 自动在内核态进入用户态以及用户态返回内核态处，设置断点。我们称这两个断点为**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，进而内存地址空间也会发生切换，因此断点组也应当切换。我们令 Debug Adapter 每次断点被触发时都检测这个断点是否是边界断点。如果是的话，先移除旧断点组中的所有断点，再设置新断点组的断点（图2.4）：

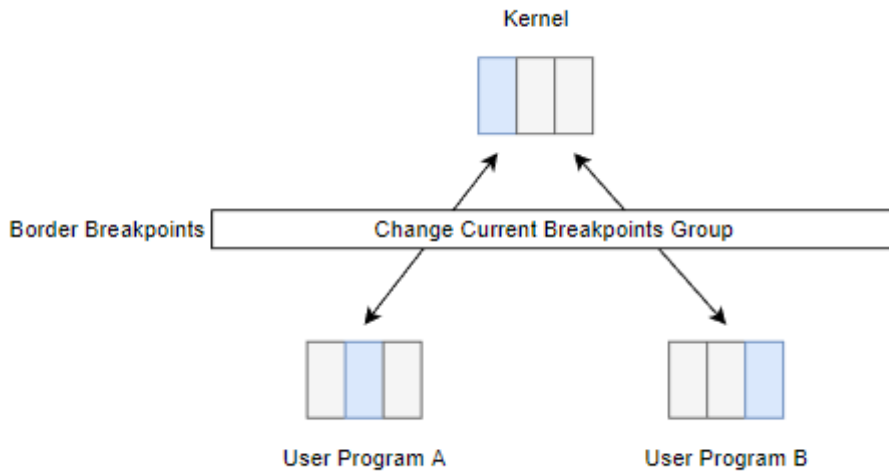


图2.4 断点组切换

断点组切换的代码如下：

```
protected handleBreakpoint(info: MINode) {
    if (this.addressSpaces.pathToSpaceName(
info.outOfBandRecord[0].output[3][1][4][1])
=== 'kernel'
){//如果是内核即将trap入用户态处的断点
    this.addressSpaces.updateCurrentSpace('kernel');
    this.sendEvent({ event: "inKernel" } as DebugProtocol.Event);
    if (info.outOfBandRecord[0].output[3][1][3][1] === "src/trap/mod.rs"
&& info.outOfBandRecord[0].output[3][1][5][1] === '135') {
        this.sendEvent({ event: "kernelToUserBorder" }
as DebugProtocol.Event);//发送event
    }
}
}
```

为了保证相关功能正常运作，断点组切换时，符号表文件也应随着断点组的切换而切换：

```
//extension.ts
else if (message.event === "kernelToUserBorder") {
    //到达内核态->用户态的边界
    // removeAllCliBreakpoints();
    vscode.window.showInformationMessage("will switched to " + userDebugFile + "
breakpoints");
    vscode.debug.activeDebugSession?.customRequest("addDebugFile", {
        debugFilepath:
            os.homedir() +
            "/rCore-Tutorial-v3/user/target/riscv64gc-unknown-none-elf/release/"
+
            userDebugFile,
    });
    vscode.debug.activeDebugSession?.customRequest(
        "updateCurrentSpace",
        "src/bin/" + userDebugFile + ".rs"
    );
}
```



```
);
```

目前，我们没有做多核处理器的适配工作。不过，从调试工具的角度来讲，多核处理器的适配是比较简单的，只需要根据进程的CPU号进行断点组的细分即可。

2.4 获取更多调试信息

上一章所述的“断点组切换”机制用到了两个关键的“边界断点”。然而，在使用默认编译参数的情况下，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等，导致边界断点无法设置。因此，如果不修改编译参数，编译出的操作系统镜像和调试信息文件就难以用于基于“断点组切换”机制的操作系统调试。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。

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，我们修改了 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，一些 OS 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，因此在这些 OS 上需要修改链接脚本，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。好在 ArceOS 和 Starry 不需要做这个工作。

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，导致文件系统无法正常运转，例如在加载文件时崩溃，栈溢出等。为了解决这个问题，需要调整了这个文件系统的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，我们也调整了操作系统的用户堆栈大小和内核堆栈大小。

2.5 适配 ArceOS 和 Starry (内核态)

2.5.1 概述

考虑到 ArceOS 没有用户态和内核态的切换，适配起来比较简单，我先适配 RISC-V 平台上的 ArceOS。通过调整 launch.json 配置文件，切换 GDB 版本，我初步支持了单核的 arceos(unikernel) 在 riscv 上的调试。以下是大概的调整过程：首先根据 arceos 的输出，找到 arceos 的 qemu 和 gdb 启动参数，再查阅 arceos(unikernel) 的 makefile，发现他们用的是 gdb-multiarch 而不是 riscv64-unknown-elf-gdb。好在 launch.json 配置文件已经允许用户切换 gdb 的版本。在配置文件中切换 gdb 版本之后，调试例程成功启动了。但是没有停在第一行代码，而是 arceos 自己执行完应用程序然后关机了。怀疑是代码被内联，于是修改编译参数。但是，在添加了 [profile.release] debug = true 参数后，arceos 启动不了了。发现 qemu 必须要用 stripped 的 bin 文件(elf 跑不了)，GDB 则加载 elf 文件。在适配 x86 平台的 ArceOS 时，我在尝试将 ArceOS 以 Debug 模式编译时遇到一些困难，最终得以解决。在适配 Starry 的过程中，我发现了插件代码中特权级切换时符号表切换的逻辑问题，对插件源代码做出了一些修改，在配置文件中添加了可配置参数，使得今后适配其他 OS 更加容易。

2.5.2 RISC-V+QEMU+ArceOS unikernel

用户如果想要用调试器插件调试 ArceOS，流程如下：

首先我们需要下载 gdb-multiarch：

```
sudo apt install gdb-multiarch
```

接着我们运行一下 ArceOS，从而生成 bin 和 elf 文件。这里以 RISC-V 上的单核 arceos-helloworld 为例：

```
make A=apps/helloworld/ ARCH=riscv64 LOG=info SMP=1 run
```

在 ArceOS 的输出中，我们发现了 QEMU的启动参数:

```
qemu-system-riscv64 -m 128M -smp 1 -machine virt -bios default -kernel  
apps/helloworld//helloworld_riscv64-qemu-virt.bin -nographic
```

我们将这些启动参数转移到配置文件 `launch.json` 中:

```
//launch.json  
{  
  "version": "0.2.0",  
  "configurations": [  
    {  
      "type": "gdb",  
      "request": "launch",  
      "name": "Attach to Qemu",  
      "executable": "${userHome}/arceos/apps/helloworld/helloworld_riscv64-  
qemu-virt.elf",  
      "target": ":1234",  
      "remote": true,  
      "cwd": "${workspaceRoot}",  
      "valuesFormatting": "parseText",  
      "gdbpath": "gdb-multiarch",  
      "showDevDebugOutput": true,  
      "internalConsoleOptions": "openOnSessionStart",  
      "printCalls": true,  
      "stopAtConnect": true,  
      "qemuPath": "qemu-system-riscv64",  
      "qemuArgs": [  
        "-M",  
        "128m",  
        "-smp",  
        "1",  
        "-machine",  
        "virt",  
        "-bios",  
        "default",  
        "-kernel",  
        "apps/helloworld/helloworld_riscv64-qemu-virt.bin",  
        "-nographic",  
        "-s",  
        "-S"  
      ],  
  
      "KERNEL_IN_BREAKPOINTS_LINE": 65, // src/trap/mod.rs中内核入口行号。可能要修  
改  
      "KERNEL_OUT_BREAKPOINTS_LINE": 124, // src/trap/mod.rs中内核出口行号。可能要  
修改  
      "GO_TO_KERNEL_LINE": 30, // src/trap/mod.rs中，用于从用户态返回内核的断点行号。  
在rCore-Tutorial-v3中，这是set_user_trap_entry函数中的stvec::write(TRAMPOLINE as  
usize, TrapMode::Direct);语句。  
    },  
  ],  
}
```

```
}
```

我们在 `qemuArgs` 中添加了 `-s -S` 参数, 这样qemu在启动的时候会打开gdb调试功能并且停在第一条指令处, 方便我们设置断点。


此外, 应当注意 `executable` 参数指向包含符号表的elf文件, 而不是去除符号表后的bin文件。

由于ArceOS是unikernel, 没有用到用户态, 因此以下这三个参数不需要填写:

```
"KERNEL_IN_BREAKPOINTS_LINE":65, // src/trap/mod.rs中内核入口行号。可能要修改
"KERNEL_OUT_BREAKPOINTS_LINE":124, // src/trap/mod.rs中内核出口行号。可能要修改
"GO_TO_KERNEL_LINE":30, // src/trap/mod.rs中, 用于从用户态返回内核的断点行号。
在rCore-Tutorial-v3中, 这是set_user_trap_entry函数中的stvec::write(TRAMPOLINE as
usize, TrapMode::Direct);语句。
```

最后我们再次按f5开始调试ArceOS. 我们发现Qemu虚拟机启动, ArceOS停在了第一条指令

```
oslab@oslab:~/arceos$ qemu-system-riscv64 -M 128m -smp 1 -machine virt -bios
default -kernel apps/helloworld/helloworld_riscv64-qemu-virt.bin -nographic -s -S
```

接下来我们设置断点。比如我们在Hello, World输出语句打一个断点, 然后按"".我们会发现断点触发了。

2.5.3 x86-64+QEMU+ArceOS unikernel

首先根据arceos主页安装 `cargo-binutils` `libclang-dev` `cross-musl-based` 等工具。

然后我们安装riscv64,x86_64,aarch64虚拟机:

```
cd qemu-7.0.0/
./configure --target-list=aarch64-softmmu,riscv64-softmmu,x86_64-softmmu,aarch64-
linux-user,aarch64_be-linux-user,riscv64-linux-user,x86_64-linux-user
make -j$(nproc)
```

在你的终端配置文件中添加:

```
export PATH=$PATH:/home/oslab/qemu-7.0.0/build
```

然后重启终端。我们之前已经运行过riscv的了(<https://github.com/rcore-os/blog/pull/228/files>), 因此尝试运行x86_64:

```
make A=./apps/helloworld/ ARCH=x86_64 LOG=info run
```

遇到错误:


```
Running on qemu...
qemu-system-x86_64 -m 128M -smp 1 -machine q35 -kernel
./apps/helloworld/helloworld_x86_64-qemu-q35.elf -nographic -cpu host -accel kvm
Could not access KVM kernel module: No such file or directory
qemu-system-x86_64: -accel kvm: failed to initialize kvm: No such file or
directory
make: *** [Makefile:162: justrun] Error 1
```

原因是我的Linux是跑在VMWare虚拟机里的，不支持KVM. 因此去掉 `-cpu host -accel kvm` 参数:

```
qemu-system-x86_64 -m 128M -smp 1 -machine q35 -kernel
./apps/helloworld/helloworld_x86_64-qemu-q35.elf -nographic
```

运行成功. 接着我们把这个qemu启动参数搬到 `launch.json` 里. 由于这里qemu直接启动elf而不是bin文件, `launch.json` 也做对应的调整:

```
//launch.json
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "gdb",
      "request": "launch",
      "name": "Attach to Qemu",
      "executable": "${userHome}/arceos/apps/helloworld/helloworld_x86_64-
qemu-q35.elf",
      "target": ":1234",
      "remote": true,
      "cwd": "${workspaceRoot}",
      "valuesFormatting": "parseText",
      "gdbpath": "gdb-multiarch",
      "showDevDebugOutput": true,
      "internalConsoleOptions": "openOnSessionStart",
      "printCalls": true,
      "stopAtConnect": true,
      "qemuPath": "qemu-system-x86_64",
      "qemuArgs": [
        "-M",
        "128m",
        "-smp",
        "1",
        "-machine",
        "q35",
        "-kernel",
        "apps/helloworld/helloworld_x86_64-qemu-q35.bin",
        "-nographic",
        "-s",
        "-S"
      ],
      "KERNEL_IN_BREAKPOINTS_LINE": 65, // src/trap/mod.rs中内核入口行号。可能要修
改
```

```
"KERNEL_OUT_BREAKPOINTS_LINE":124, // src/trap/mod.rs中内核出口行号。可能要修改
"GO_TO_KERNEL_LINE":30, // src/trap/mod.rs中, 用于从用户态返回内核的断点行号。
在rCore-Tutorial-v3中, 这是set_user_trap_entry函数中的stvec::write(TRAMPOLINE as
usize, TrapMode::Direct);语句。
    },
]
}
```

arceos-httpserver也跑通了。只需要注意qemu开放的tcp端口不要和gdb占用的端口有重叠即可。

2.5.4 适配Starry

在适配 x86-64 Starry 的过程中, 我经过很长时间的尝试, 发现Starry编译出的elf只有 linking symbol 没有debugging symbol, 所以不可以用“文件名-行号”的方式打断点, 只能通过直接指定函数名的方式打断点。这可能是因为Starry在和C程序链接的时候把这些信息丢弃了。此类问题的根本解决办法是让编译器在编译的时候就把符号信息单独放在某一文件里(而不是放进elf文件)但是目前做不到。

我与Starry开发者们交流了这个问题, 刚开始怀疑是内核栈的问题, 由于debug模式代码优化比较差, 当运行第一个应用程序的时候切换到应用的内核栈, 但是这个栈比较小, 第一个应用程序进行fork的时候就会导致溢出, 解决方法就是可以将应用程序的栈开大一点。不过这没能解决问题, 最后找到了问题的根源: debug模式下内存布局的一些改变导致了RAMDISK插入的时候并没有插入到 4K 页面的起始段。通过 .align 4096 进行对齐即可。这样就可以在debug编译模式下调试内核了(不需要调整内核栈大小)。

至此, 内核态的Starry可以用调试器调试了, 然后我尝试让调试器插件支持调试Starry的用户态程序。首先遇到了一个问题: 插件之前的逻辑(用户打**位置靠前**的用户态断点并被插件暂存-用户continue-边界断点触发-插件进行断点组切换-用户点continue-用户态断点触发)有局限性: 在rCore-Tutorial-v3的特权级切换的时候有用跳板页所以这招可以用, 但不是所有OS都有用跳板页。在拜读了lab2的GDB调试章节, 找杨金博讨论了一下熟练的工程师使用GDB跳到用户态的方式, 最后任务应该改成这种模式: 用户continue-边界断点触发-插件不停地单步, 检测是否到达用户态-如果到达用户态, 进行断点组切换。而这又衍生出了另一个问题: 如果stopped事件之后跟了一个事件, 而这个事件会让插件请求GDB做一些只有GDB停下时才能做的事情, 但是stopped事件之后就立即单步运行了, 可能会导致这个事件没被正确地处理。举个例子, 比如Debug Adapter按顺序传来:

1. 事件A:stopped
2. 事件B:请插件委托Debug Adapter向GDB请求PC寄存器。

插件处理事件A, 然后继续单步运行, 直到再次停下来了, 才处理事件B。而事件B本来要在单步运行之前进行处理才对。在这种情况下, 我们通过事件B获得的PC寄存器值就不是正确的。

因此, 我只能把“不停地单步”的功能转移到Debug Adapter, 这就牵一发而动全身, 导致很多代码需要重构。

3 优化使用体验

3.1 概述

在各种 OS 开发者、学习者社区中, 我们注意到, 尝试用 GDB 进行 OS 调试的往往是初学者。因为他们尚处于入门阶段, 对于操作系统的各种抽象概念和复杂的代码执行流不熟悉, 因此需要通过 GDB 进行操作系统的单步调试, 信息收集, 来熟悉这些内容。对于初学者来说, 学习 OS 的各种机制、策略, 读懂错综复杂的 OS 代码, 理解OS的编译流程、掌握OS和硬件的交互机制带来的负担已经很大, 不应该再在调试器的配置, 使用上增加门槛。因此, OS调试工具很有必要提供一个无需命令行操作的, 融入现有集成开发环境的友好用户界面。

我们之前已经有了一个用户界面，但是还有很多可以改进的地方。因此，我们有必要吸引一些用户来使用我们的插件，并根据他们的反馈来改进插件的使用体验。

3.2 简化安装流程

我首先找到的用户是杨金博，他在调试x86-64 Starry时感觉用GDB Debug很繁琐，想使用我这个插件。首先遇到的问题是插件安装起来很繁琐：需要先clone 插件的代码，编译插件的代码后，在弹出的新窗口中调试OS。他觉得这样很影响使用体验，应当将插件打包成安装包，让用户直接用安装包安装即可。我编译出了安装包，又觉得每一次都手动编译很不方便，于是通过Github Actions做了一个自动编译发布包的功能。这样，只要在commit中加上一个特定的tag，github就会自动生成 vsix 格式的安装包。这样做对开发者和使用者来说都方便不少。

3.3 整理代码

插件可以使用之后，杨金博想往插件里添加一个功能，但是发现难以下手，因为代码比较杂乱。我觉得这是一个整理代码的好时机。开源项目的代码通常都应做到清晰可读且符合规范，方便他人理解和修改代码。因此，我对源代码做了这些整理的工作：

- 将用户需要关注的核心代码放在同一处
- 添加单元测试
- 删除一些"hack"，比如不再用文件名来判断断点的特权级，而是由内存地址来判断特权级
- 连续十几个的switch-case 改成hashmap
- 将一些冗长的代码抽象成函数
- 统一注释的规范

3.4 提供更加友好的帮助信息

目前经过多次的迭代，仓库主页的Readme已经包含了用户在安装、使用、改造调试器的过程中需要知道的大部分内容。但是这个Readme已经变得冗长，很多人看到密密麻麻的文字说明望而却步，因此我写了一个简约版本的mdbook文档（链接：<https://chenzhiy2001.github.io/code-debug-documentation/>），不仅使用了大家熟悉的mdbook界面，而且只保留最核心的内容，将用户上手的难度降到最低。比如，在仓库主页的Readme中，我们讲述了安装插件的多种方案，但是在这个帮助文档中我们只保留了最简单快捷的那一种。

除了文档，调试器插件运行过程中输出的调试信息也是需要经常被用户分析的，因此这些输出的可读性也很重要。我们不仅对这些输出的格式做了缩进，颜色方面的改进，使得可读性大大增加，而且在帮助文档中告诉用户如何阅读这些调试信息。

致谢

感谢泉城实验室的小伙伴们和老师们！