支持 Rust 语言的源代码级内核调试工具

OSATC 学生分会场报告

陈志扬

指导教师：吴竞邦

北京工商大学计算机科学与技术系

2023 年 3 月 23 日

项目背景与目标

- 项目背景
  - rust 操作系统相关实验上手难度较高
  - 环境配置繁琐
  - GDB TUI 不方便

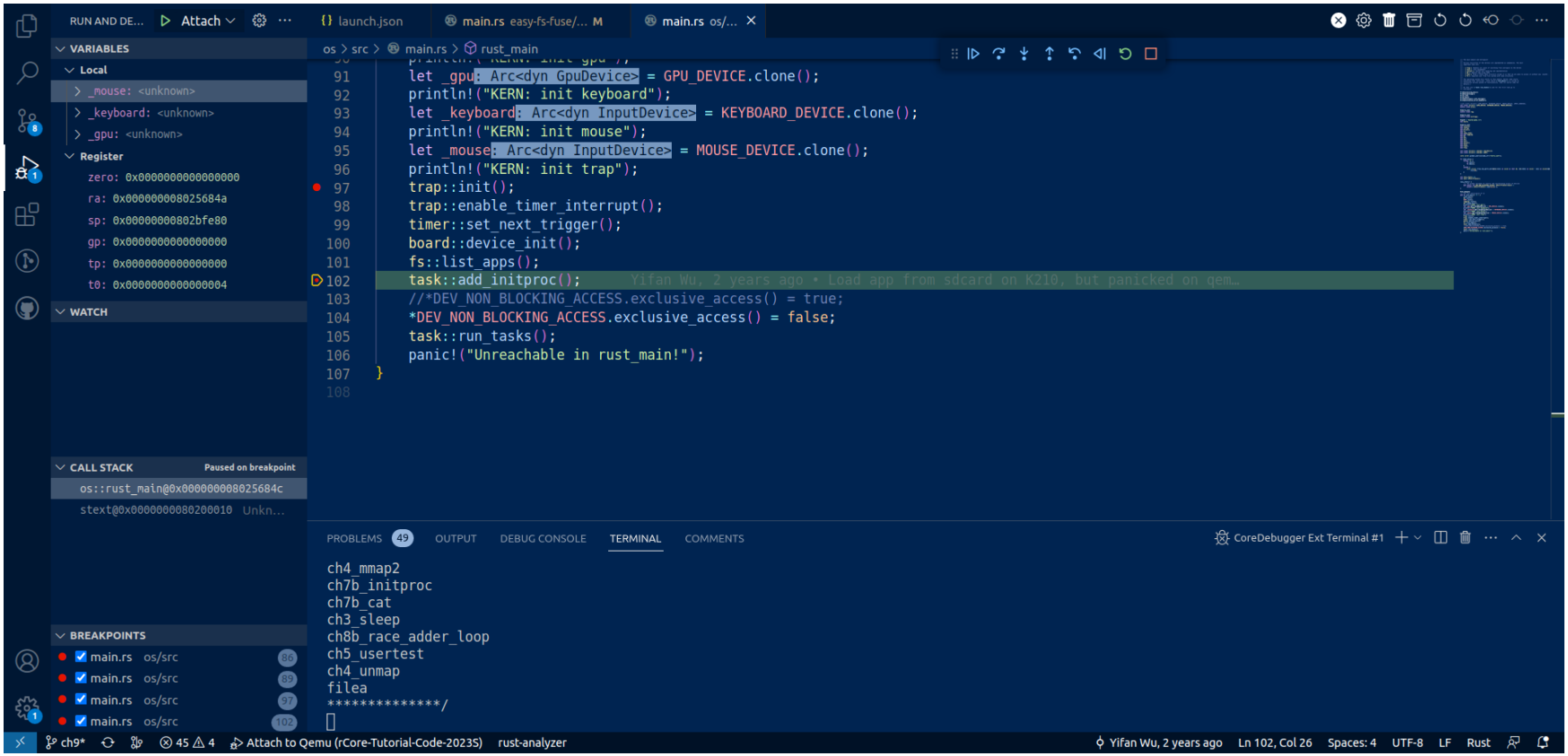
项目背景与目标

- 项目背景
  - rust 操作系统相关实验上手难度较高
  - 环境配置繁琐
  - GDB TUI 不方便
- 已完成工作
  - 基于 Qemu 的 rust 内核在线调试工具
  - 支持基于 GDB 的单步断点、内存查看、寄存器查看功能
  - 内核态与用户态方便的切换跟踪

项目背景与目标

- 项目背景
  - rust 操作系统相关实验上手难度较高
  - 环境配置繁琐
  - GDB TUI 不方便
- 已完成工作
  - 基于 Qemu 的 rust 内核在线调试工具
  - 支持基于 GDB 的单步断点、内存查看、寄存器查看功能
  - 内核态与用户态方便的切换跟踪
- 目前的进展与安排
  - 支持基于 eBPF 的单步断点、内存查看、寄存器查看功能
  - 函数调用动态跟踪
  - 基于真实系统（FPGA 或 RISC-V 开发板）的远程实验与调试系统

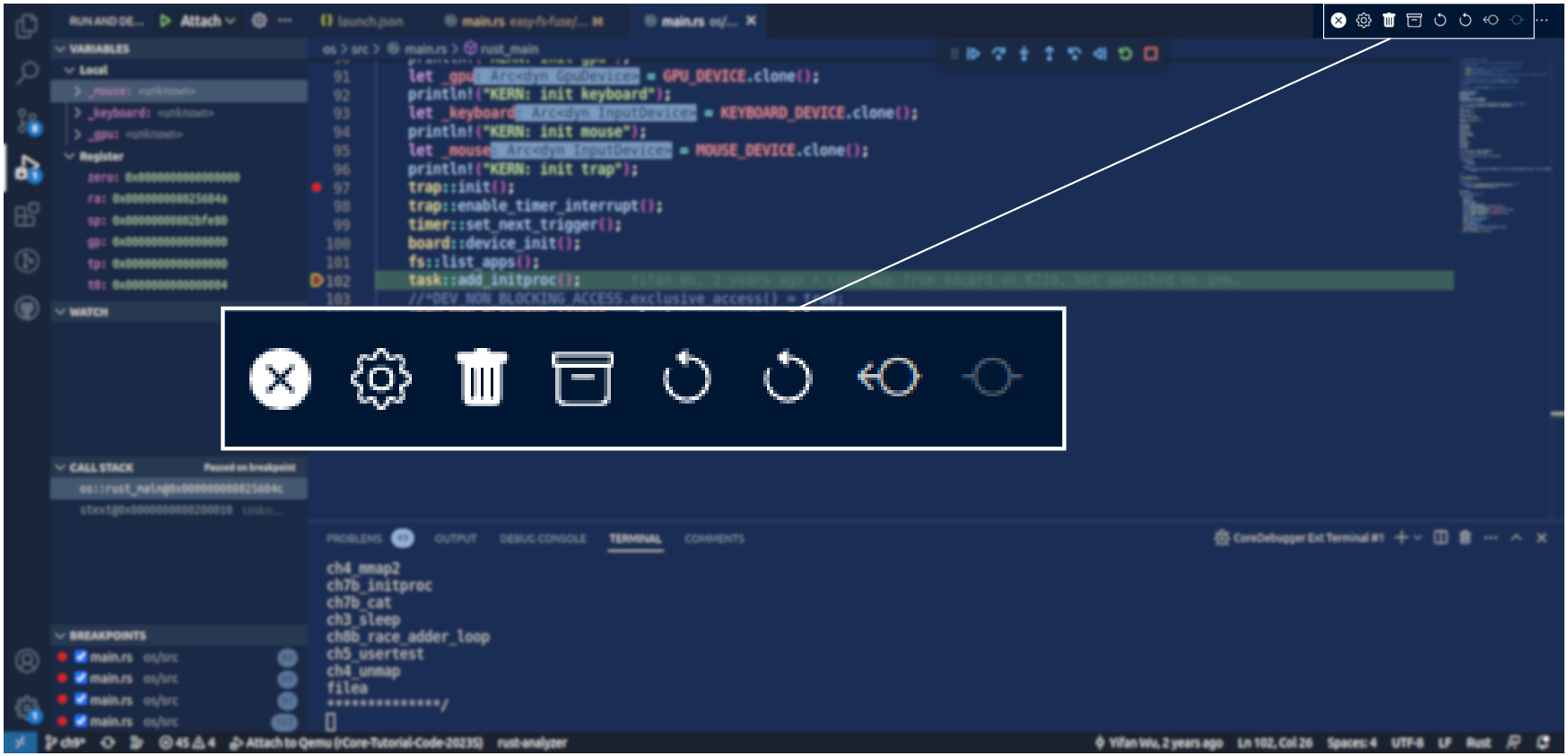
用户界面


RUN AND DE... Attach
launch.json
main.rs easy-fs-fuse/... M
main.rs os/...
os > src > main.rs > rust_main
VARIABLES
Local
_mouse: <unknown>
_keyboard: <unknown>
_gpu: <unknown>
Register
zero: 0x0000000000000000
ra: 0x000000008025684a
sp: 0x00000000802bfe80
gp: 0x0000000000000000
tp: 0x0000000000000000
t0: 0x0000000000000004
WATCH
CALL STACK Paused on breakpoint
os::rust_main@0x000000008025684c
stext@0x0000000080200010 Unkn...
BREAKPOINTS
main.rs os/src 86
main.rs os/src 89
main.rs os/src 97
main.rs os/src 102
let _gpu: Arc<dyn GpuDevice> = GPU_DEVICE.clone();
println!("KERN: init keyboard");
let _keyboard: Arc<dyn InputDevice> = KEYBOARD_DEVICE.clone();
println!("KERN: init mouse");
let _mouse: Arc<dyn InputDevice> = MOUSE_DEVICE.clone();
println!("KERN: init trap");
trap::init();
trap::enable_timer_interrupt();
timer::set_next_trigger();
board::device_init();
fs::list_apps();
task::add_initproc(); Yifan Wu, 2 years ago • Load app from sdcard on K210, but panicked on qem…
//*DEV_NON_BLOCKING_ACCESS.exclusive_access() = true;
*DEV_NON_BLOCKING_ACCESS.exclusive_access() = false;
task::run_tasks();
panic!("Unreachable in rust_main!");
}
PROBLEMS 49 OUTPUT DEBUG CONSOLE TERMINAL COMMENTS
CoreDebugger Ext Terminal #1
ch4_mmap2
ch7b_initproc
ch7b_cat
ch3_sleep
ch8b_race_adder_loop
ch5_usertest
ch4_unmap
filea
**************/
ch9* 45 4 Attach to Qemu (rCore-Tutorial-Code-2023S) rust-analyzer
Yifan Wu, 2 years ago Ln 102, Col 26 Spaces: 4 UTF-8 LF Rust

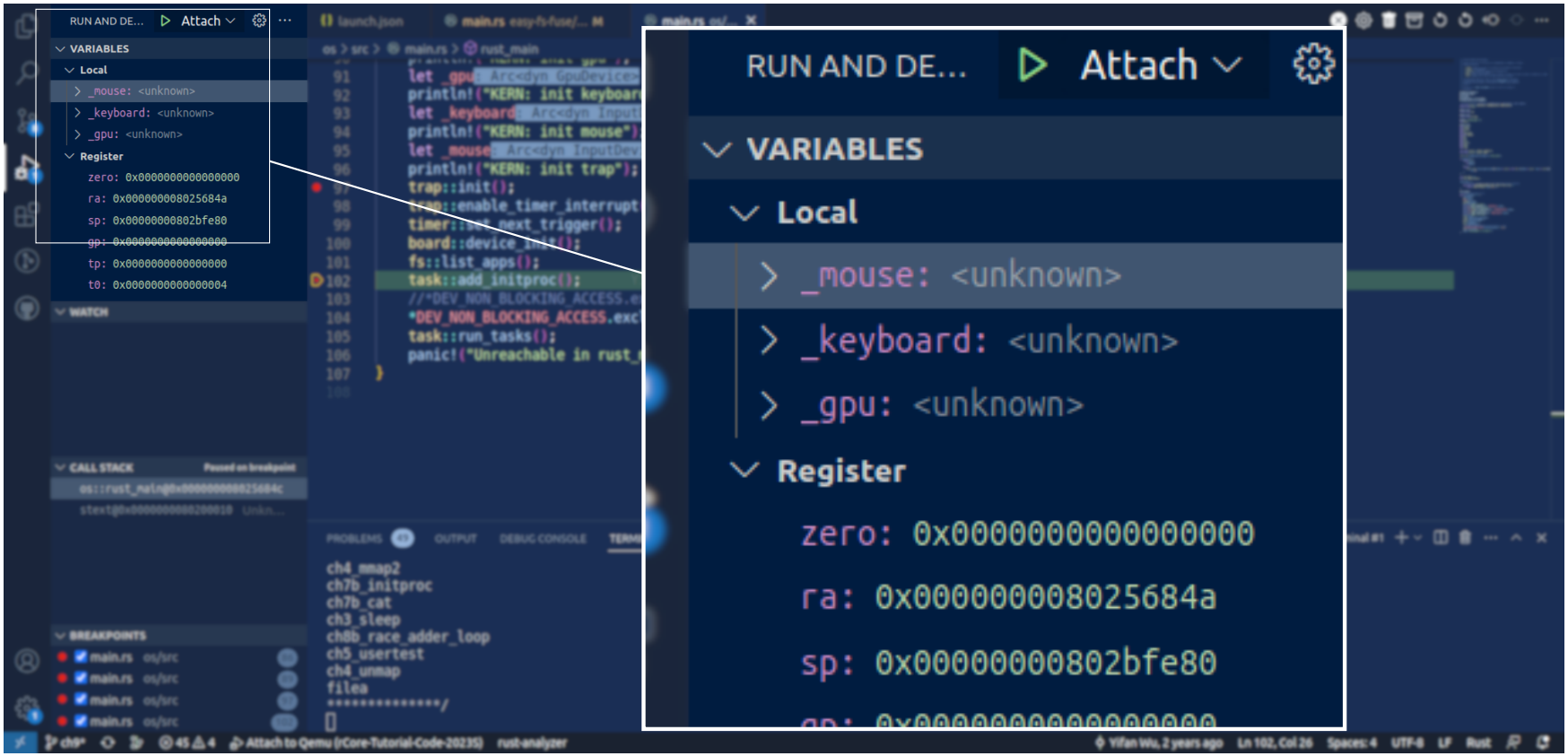
用户界面

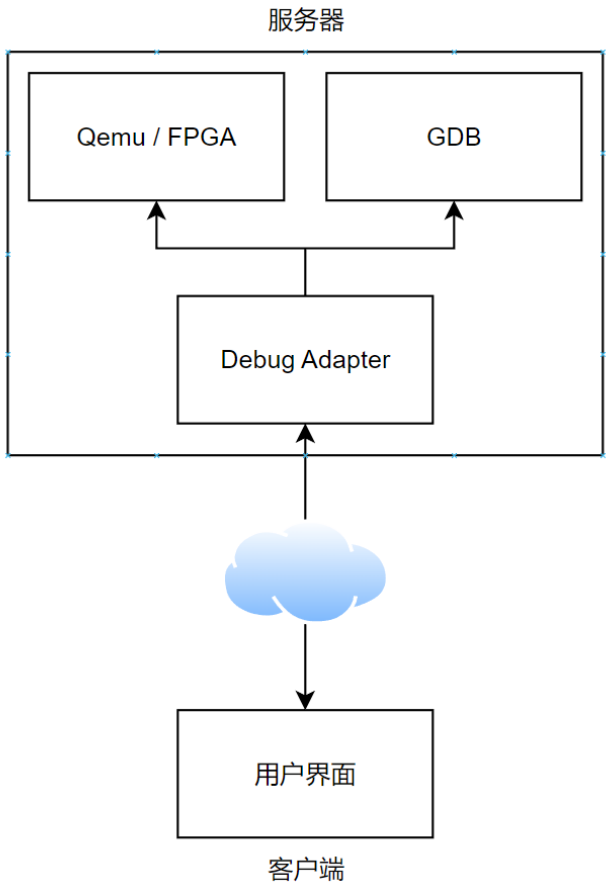

let _gpu: Arc<dyn GpuDevice> = GPU_DEVICE.clone();
println!("KERN: init keyboard");
let _keyboard: Arc<dyn InputDevice> = KEYBOARD_DEVICE.clone();
println!("KERN: init mouse");
let _mouse: Arc<dyn InputDevice> = MOUSE_DEVICE.clone();
println!("KERN: init trap");
trap::init();
trap::enable_timer_interrupt();
timer::set_next_trigger();
board::device_init();
fs::list_apps();
ch4_mmap2
ch7b_initproc
ch7b_cat
ch3_sleep
ch8b_race_adder_loop
ch5_usertest
ch4_unmap
filea
**************/

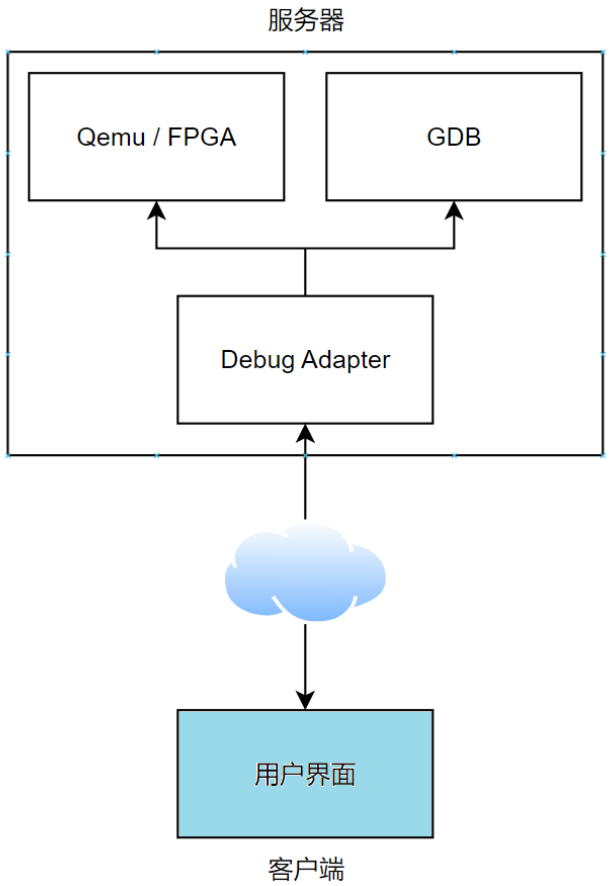
用户界面

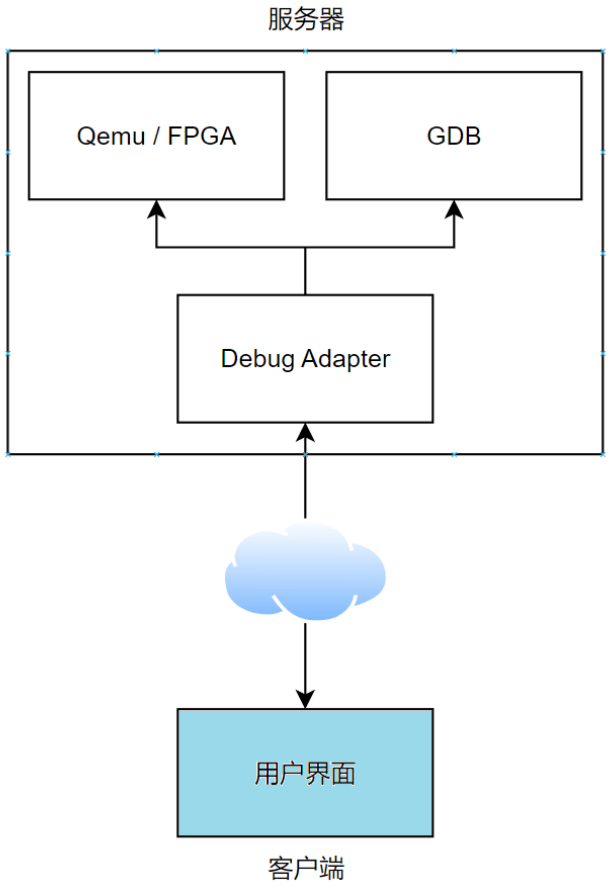

let _gpu: Arc<dyn GpuDevice> = GPU_DEVICE.clone();
println!("KERN: init keyboard");
let _keyboard: Arc<dyn InputDevice> = KEYBOARD_DEVICE.clone();
println!("KERN: init mouse");
let _mouse: Arc<dyn InputDevice> = MOUSE_DEVICE.clone();
println!("KERN: init trap");
trap::init();
trap::enable_timer_interrupt();
timer::set_next_trigger();
board::device_init();
fs::list_apps();
task::add_initproc();
ch4_mmap2
ch7b_initproc
ch7b_cat
ch3_sleep
ch8b_race_adder_loop
ch5_usertest
ch4_unmap
filea
**********/

用户界面

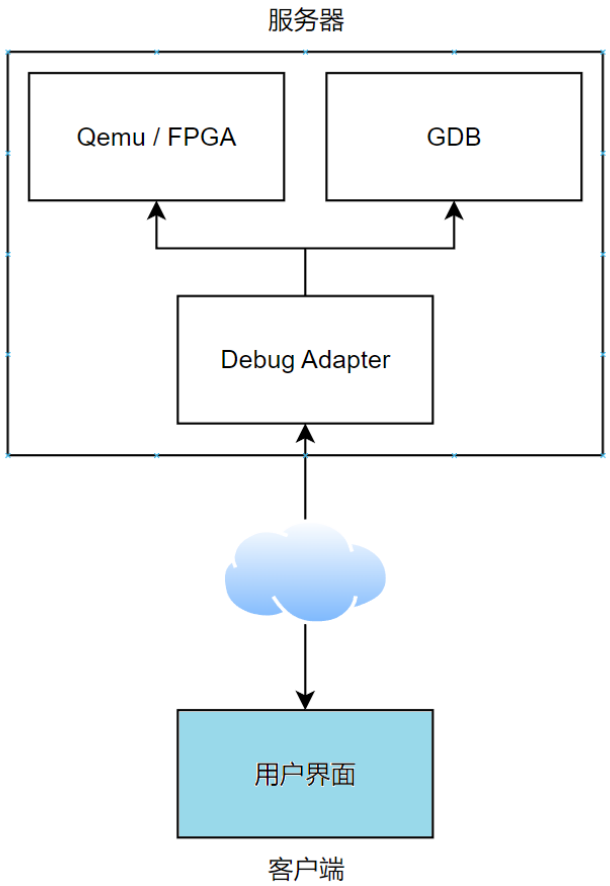

RUN AND DE...
Attach
VARIABLES
Local
_mouse: <unknown>
_keyboard: <unknown>
_gpu: <unknown>
Register
zero: 0x0000000000000000
ra: 0x000000008025684a
sp: 0x00000000802bfe80


服务器
Qemu / FPGA
GDB
Debug Adapter
用户界面
客户端

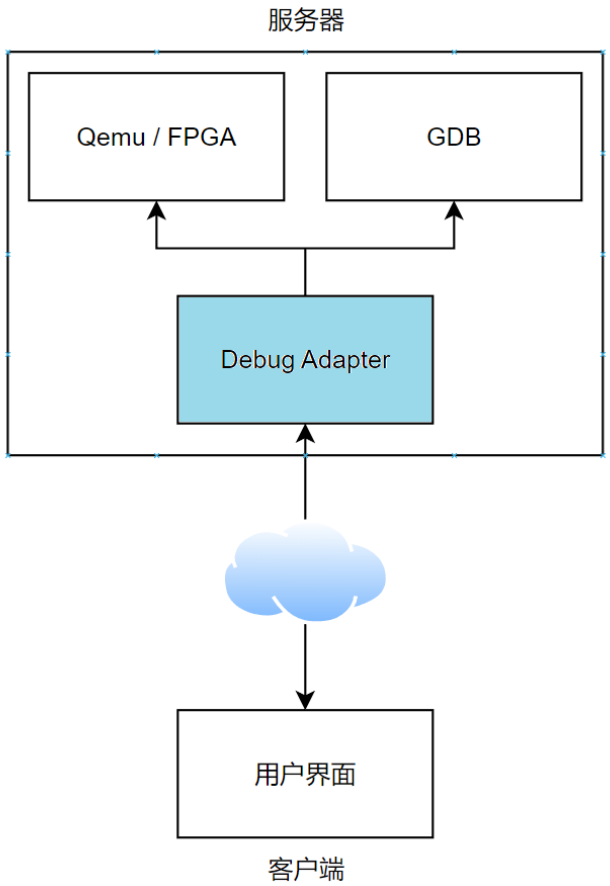

服务器
Qemu / FPGA
GDB
Debug Adapter
用户界面
客户端

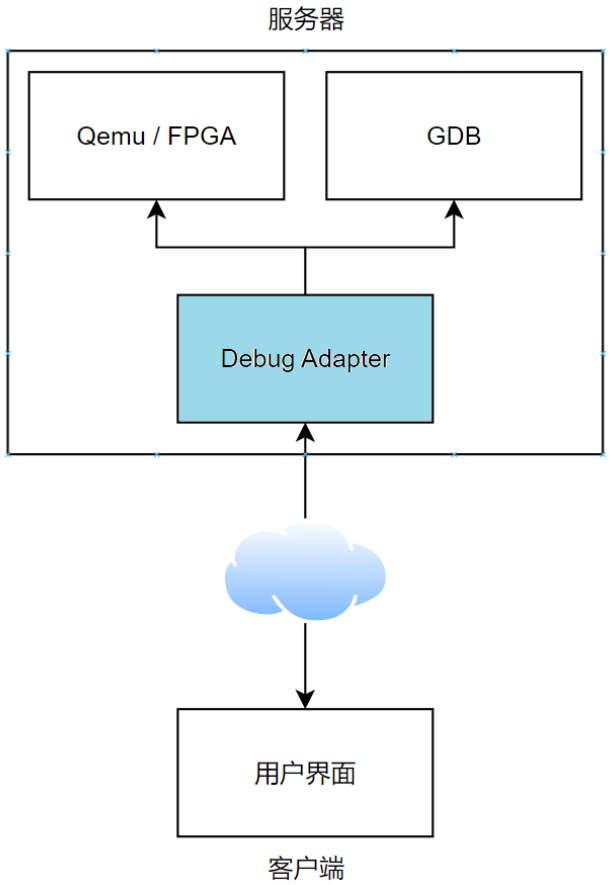


- 客户端是一个浏览器页面，打开即用，类似 github classroom

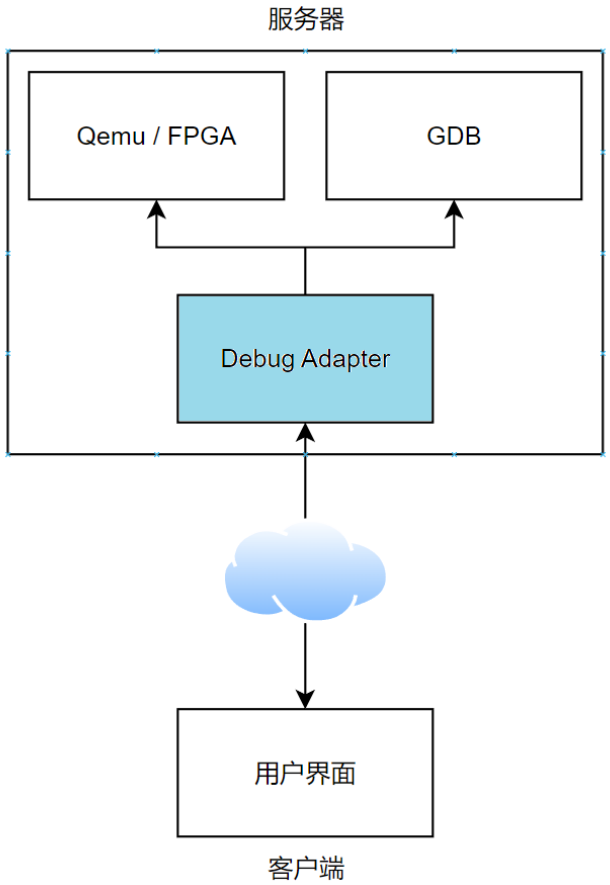


- 客户端是一个浏览器页面，打开即用，类似 github classroom
  - 调试者与被调试内核分离

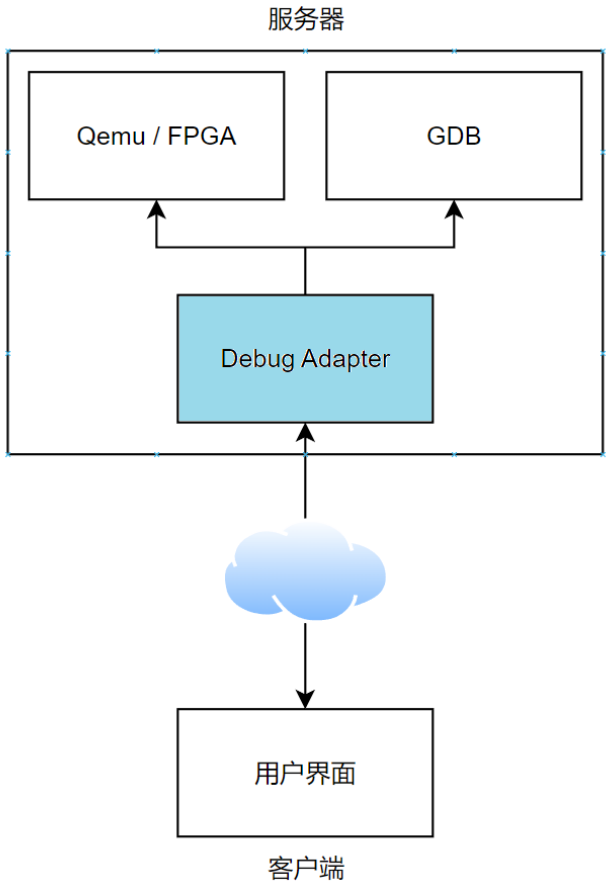

服务器
Qemu / FPGA
GDB
Debug Adapter
用户界面
客户端



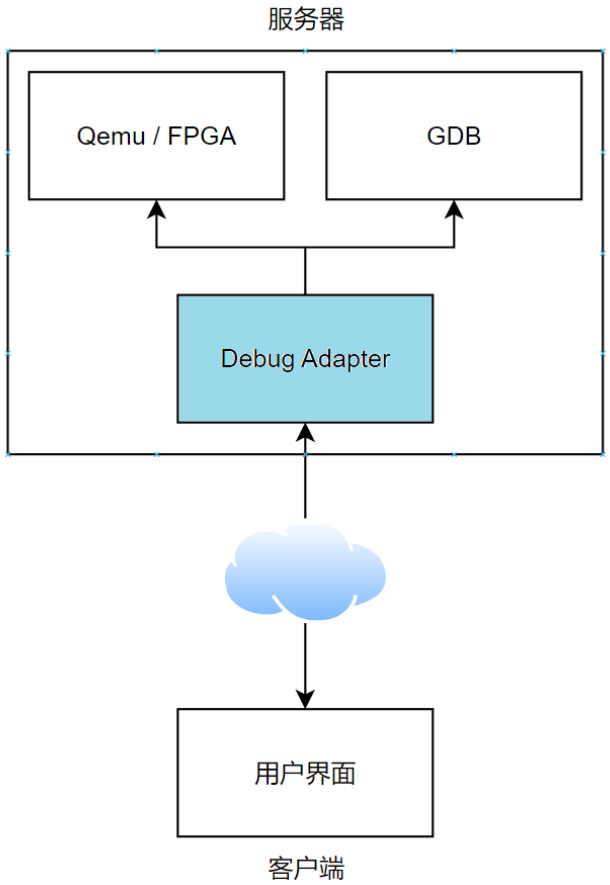
- Debug Adapter 提供操作系统相关的调试功能



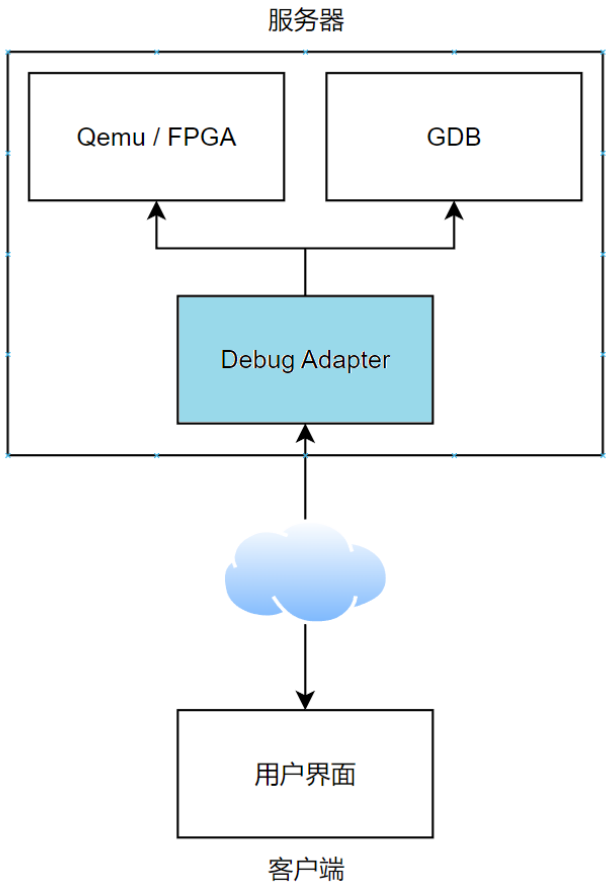
- Debug Adapter 提供操作系统相关的调试功能
  - 获取寄存器、内存、变量信息



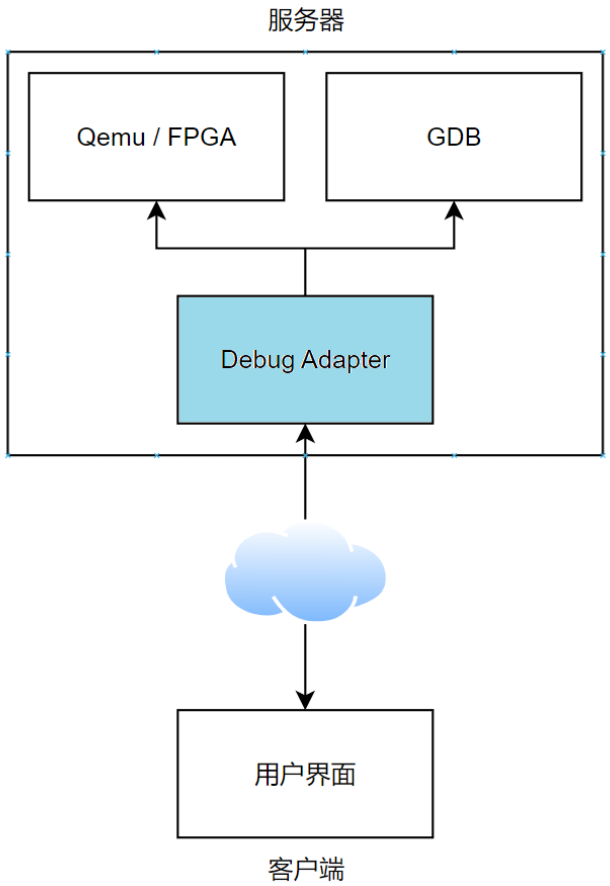
- Debug Adapter 提供操作系统相关的调试功能
  - 获取寄存器、内存、变量信息
  - 跟踪系统调用



- Debug Adapter 提供操作系统相关的调试功能
  - 获取寄存器、内存、变量信息
  - 跟踪系统调用
  - 准确获取当前特权级

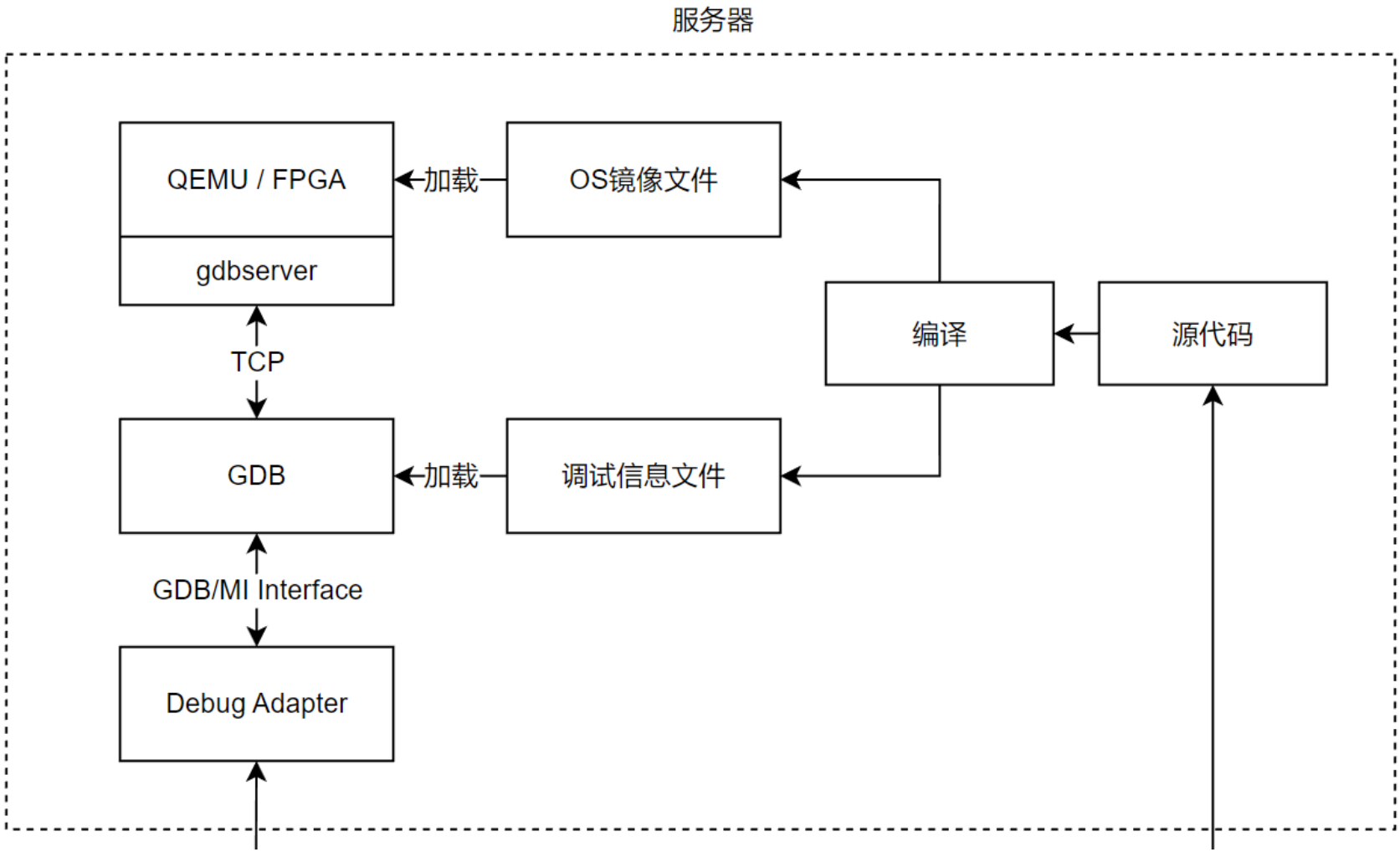


- Debug Adapter 提供操作系统相关的调试功能
  - 获取寄存器、内存、变量信息
  - 跟踪系统调用
  - 准确获取当前特权级
  - 支持在内核态设置用户态程序的断点



- Debug Adapter 提供操作系统相关的调试功能
  - 获取寄存器、内存、变量信息
  - 跟踪系统调用
  - 准确获取当前特权级
  - 支持在内核态设置用户态程序的断点
  - 自动加载、更换符号信息文件
- Debug Adapter 以 VSCode 插件的形式提供

自动编译、加载内核并启动调试



遇到的问题

- 问题 1：GDB 无法设置断点

遇到的问题

- 问题 1：GDB 无法设置断点
  - 解决办法：修改编译参数，从而保留符号表等调试信息，并关闭编译器的优化

遇到的问题

- 问题 1：GDB 无法设置断点
  - 解决办法：修改编译参数，从而保留符号表等调试信息，并关闭编译器的优化
- 问题 2：没有操作系统相关的调试功能

遇到的问题

- 问题 1：GDB 无法设置断点
  - 解决办法：修改编译参数，从而保留符号表等调试信息，并关闭编译器的优化
- 问题 2：没有操作系统相关的调试功能
  - 解决办法：修改 Debug Adapter

通过修改编译参数解决问题 1

- 修改 ‘cargo.toml‘ 配置文件中的编译参数

通过修改编译参数解决问题 1

- 修改 ‘cargo.toml‘ 配置文件中的编译参数
  - ‘debug=true‘ 保留调试信息

通过修改编译参数解决问题 1

- 修改 ‘cargo.toml‘ 配置文件中的编译参数
  - ‘debug=true‘ 保留调试信息
  - ‘opt-leve1=0‘ 最低优化等级

通过修改编译参数解决问题 1

- 修改 ‘cargo.toml‘ 配置文件中的编译参数
  - ‘debug=true‘ 保留调试信息
  - ‘opt-leve1=0‘ 最低优化等级
- ‘linker.ld‘ 保留 *.debug 段

通过修改编译参数解决问题 1

- 修改 ‘cargo.toml‘ 配置文件中的编译参数
  - ‘debug=true‘ 保留调试信息
  - ‘opt-leve1=0‘ 最低优化等级
- ‘linker.ld‘ 保留 *.debug 段
- 修改后带来的问题

通过修改编译参数解决问题 1

- 修改 ‘cargo.toml‘ 配置文件中的编译参数
  - ‘debug=true‘ 保留调试信息
  - ‘opt-leve1=0‘ 最低优化等级
- ‘linker.ld‘ 保留 *.debug 段
- 修改后带来的问题
  - 应用程序占用的磁盘空问显著增加，导致 easy-fs-fuse（用于将应用程序打包为文件系统镜像）崩溃

通过修改编译参数解决问题 1

- 修改 ‘cargo.toml‘ 配置文件中的编译参数
  - ‘debug=true‘ 保留调试信息
  - ‘opt-leve1=0‘ 最低优化等级
- ‘linker.ld‘ 保留 *.debug 段
- 修改后带来的问题
  - 应用程序占用的磁盘空问显著增加，导致 easy-fs-fuse（用于将应用程序打包为文件系统镜像）崩溃
    - 因此，需要将磁盘镜像的空间调大

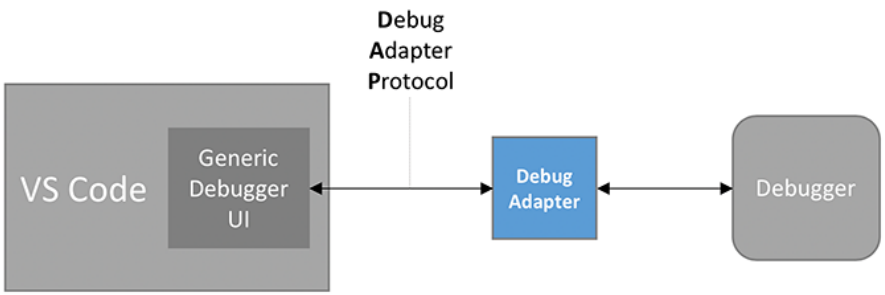
通过修改编译参数解决问题 1

- 修改 ‘cargo.toml‘ 配置文件中的编译参数
  - ‘debug=true‘ 保留调试信息
  - ‘opt-leve1=0‘ 最低优化等级
- ‘linker.ld‘ 保留 *.debug 段
- 修改后带来的问题
  - 应用程序占用的磁盘空问显著增加，导致 easy-fs-fuse（用于将应用程序打包为文件系统镜像）崩溃
    - 因此，需要将磁盘镜像的空间调大
  - 用户栈溢出

通过修改编译参数解决问题 1

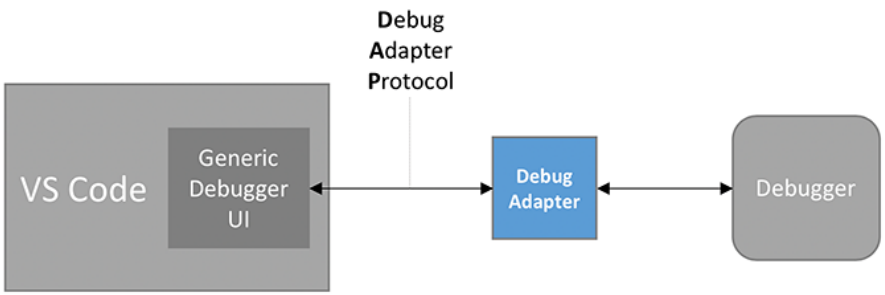
- 修改 ‘cargo.toml‘ 配置文件中的编译参数
  - ‘debug=true‘ 保留调试信息
  - ‘opt-leve1=0‘ 最低优化等级
- ‘linker.ld‘ 保留 *.debug 段
- 修改后带来的问题
  - 应用程序占用的磁盘空问显著增加，导致 easy-fs-fuse（用于将应用程序打包为文件系统镜像）崩溃
    - 因此，需要将磁盘镜像的空间调大
  - 用户栈溢出
    - 调整 ‘USER_STACK_SIZE‘ 等参数

通过改进 Debug Adapter 解决问题 2



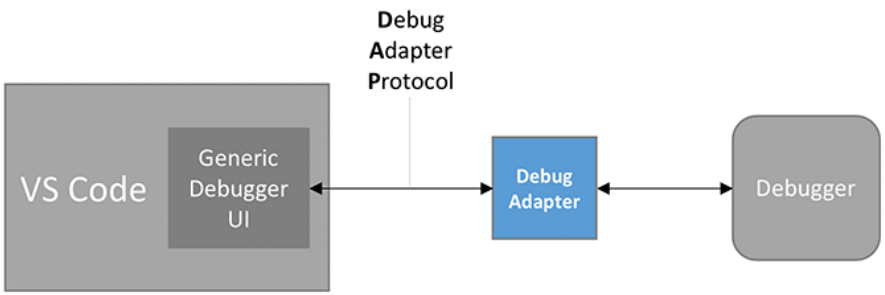
- 负责协调 VS Code 和 GDB 的独立进程

通过改进 Debug Adapter 解决问题 2



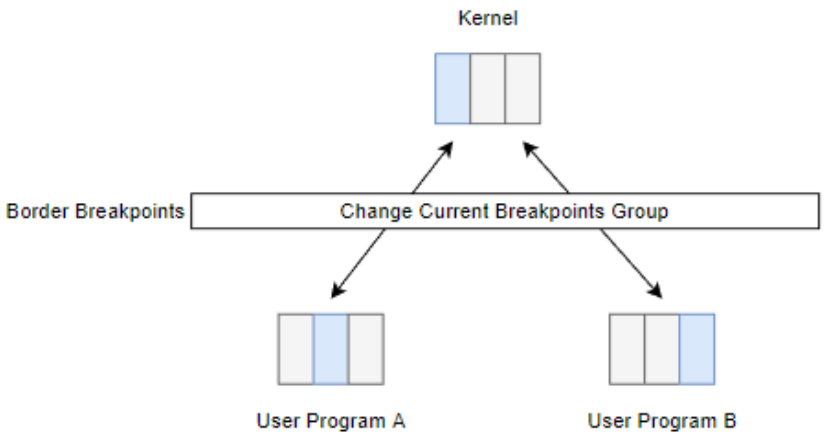
- 负责协调 VS Code 和 GDB 的独立进程
- DA 和 VSCode 通信：DAP
  - Request-Response-Event

通过改进 Debug Adapter 解决问题 2

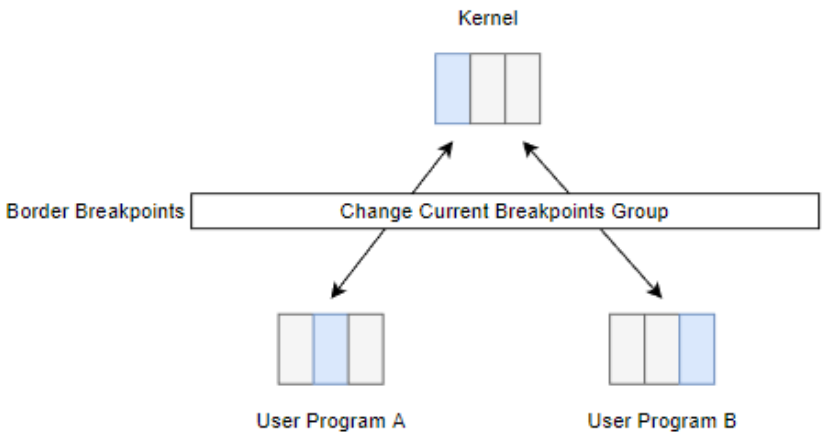


- 负责协调 VS Code 和 GDB 的独立进程
- DA 和 VSCode 通信：DAP
  - Request-Response-Event
- DA 和 GDB 通信：GDB/MI

解决内核态用户态的断点冲突

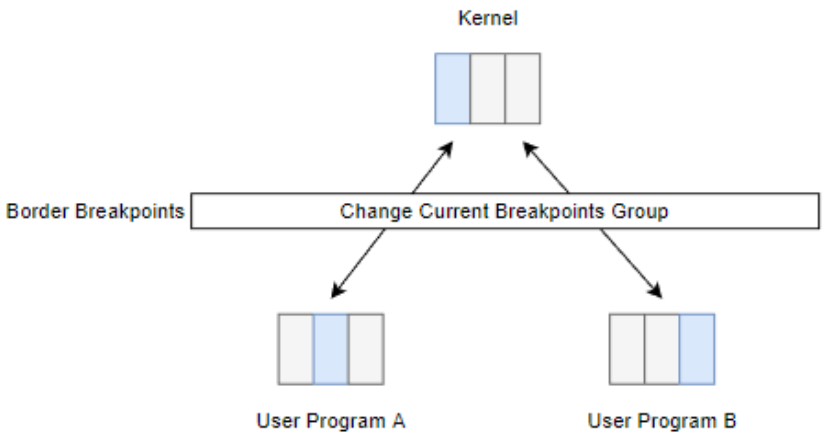


解决内核态用户态的断点冲突



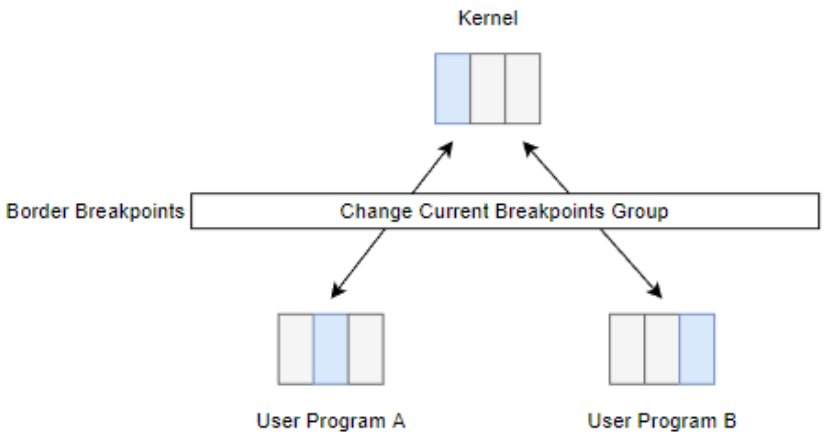
- 存在的问题：由于 GDB 限制，无法在内核态设置用户态代码的断点

解决内核态用户态的断点冲突



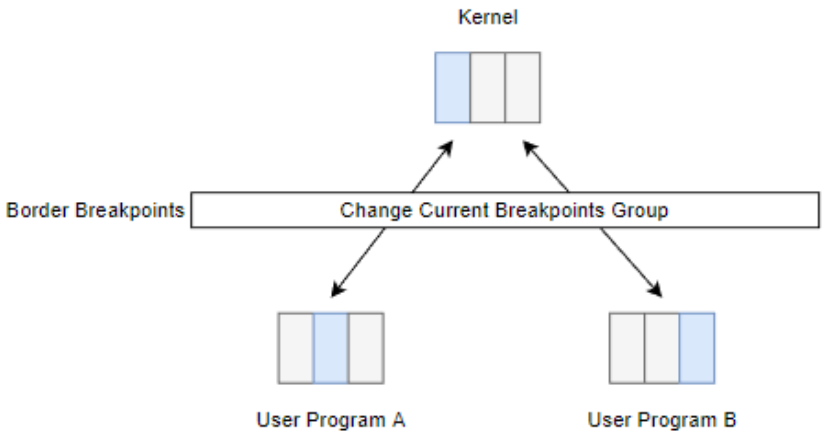
- 存在的问题：由于 GDB 限制，无法在内核态设置用户态代码的断点
- 原因：特权级切换时，TLB 刷新

解决内核态用户态的断点冲突



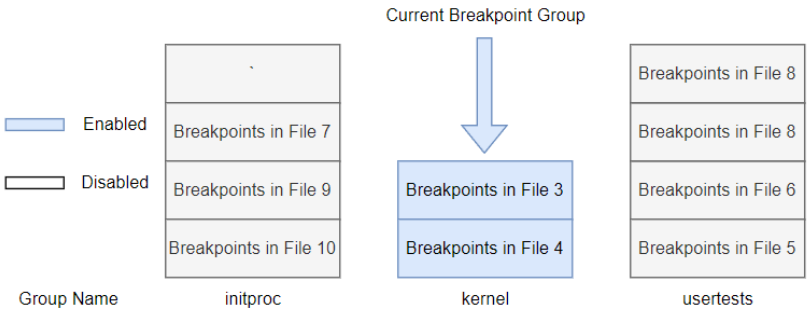
- 存在的问题：由于 GDB 限制，无法在内核态设置用户态代码的断点
- 原因：特权级切换时，TLB 刷新
- 解决方法：暂存断点，待时机合适再设置断点

解决内核态用户态的断点冲突



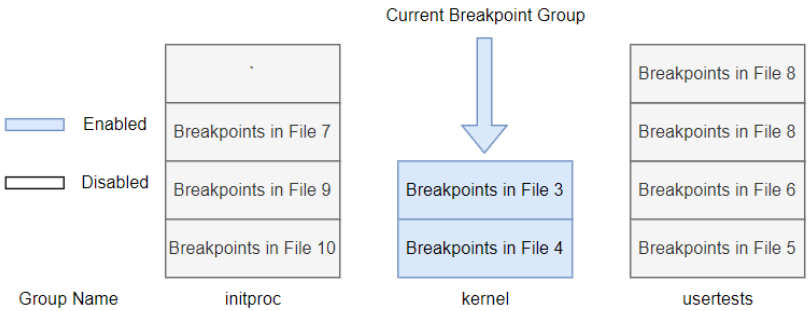
- 存在的问题：由于 GDB 限制，无法在内核态设置用户态代码的断点
- 原因：特权级切换时，TLB 刷新
- 解决方法：暂存断点，待时机合适再设置断点
- 关键问题：暂存断点的策略，恢复断点的时机

断点组管理模块

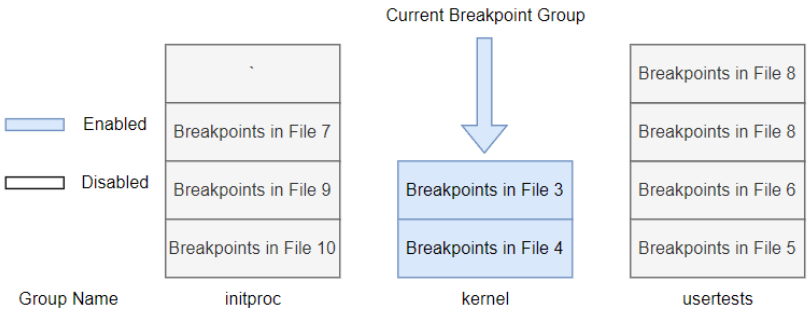

Current Breakpoint Group
Enabled
Disabled
Breakpoints in File 7
Breakpoints in File 9
Breakpoints in File 10
Breakpoints in File 3
Breakpoints in File 4
Breakpoints in File 8
Breakpoints in File 8
Breakpoints in File 6
Breakpoints in File 5
Group Name
initproc
kernel
usertests

断点组管理模块

- 分组缓存所有断点的信息
  - 当前断点组

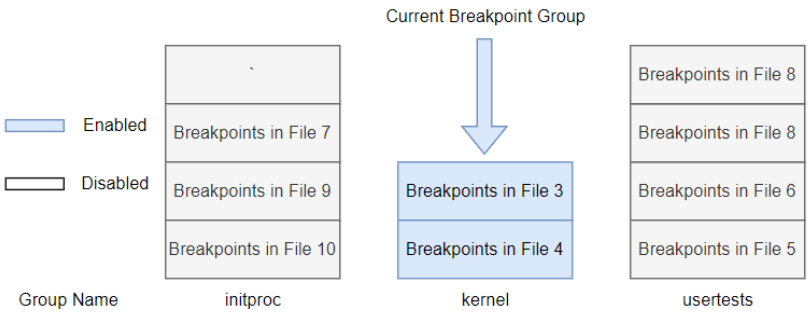


断点组管理模块



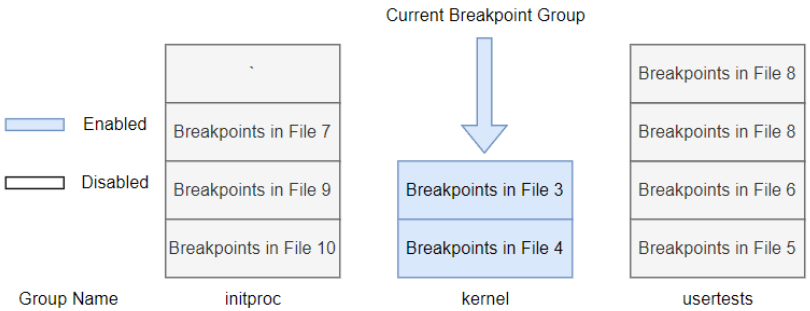
- 分组缓存所有断点的信息
  - 当前断点组
- 若用户设置的断点不属于当前断点组，不令 GDB 设置

断点组管理模块



- 分组缓存所有断点的信息
  - 当前断点组
- 若用户设置的断点不属于当前断点组，不令 GDB 设置
- 在特权级切换时切换符号表文件、进行断点组切换

断点组管理模块



- 分组缓存所有断点的信息
  - 当前断点组
- 若用户设置的断点不属于当前断点组，不令 GDB 设置
- 在特权级切换时切换符号表文件、进行断点组切换
- 这种策略还可以应用于多处理机、多线程、多协程…

如何判断特权级是否已经切换？

- 由于 risc-v 处理器无寄存器能反映当前特权级，所以需要用一些变通的办法

如何判断特权级是否已经切换？

- 由于 risc-v 处理器无寄存器能反映当前特权级，所以需要用一些变通的办法
- 在特权级切换的代码附近设置断点，若断点触发则特权级将切换

如何判断特权级是否已经切换？

- 由于 risc-v 处理器无寄存器能反映当前特权级，所以需要用一些变通的办法
- 在特权级切换的代码附近设置断点，若断点触发则特权级将切换
  - 这些“边界断点”可以由 Debug Adapter 自动设置，无需用户手动设置

如何判断特权级是否已经切换？

- 由于 risc-v 处理器无寄存器能反映当前特权级，所以需要用一些变通的办法
- 在特权级切换的代码附近设置断点，若断点触发则特权级将切换
  - 这些“边界断点”可以由 Debug Adapter 自动设置，无需用户手动设置
- 同时借助内存地址空间、文件名辅助判断

利用 eBPF 进行跟踪

- 为了用上调试器，得改编译参数，改内核代码... 比较繁琐.

利用 eBPF 进行跟踪

- 为了用上调试器，得改编译参数，改内核代码... 比较繁琐.
- GDB 无法跟踪 rCore 的一些重要的内核数据结构，对 rust 语言的支持也不是特别好.

利用 eBPF 进行跟踪

- 为了用上调试器，得改编译参数，改内核代码... 比较繁琐.
- GDB 无法跟踪 rCore 的一些重要的内核数据结构，对 rust 语言的支持也不是特别好.
- 因此，我们想用 eBPF 技术来实现跟踪功能.

利用 eBPF 进行跟踪

- 为了用上调试器，得改编译参数，改内核代码... 比较繁琐.
- GDB 无法跟踪 rCore 的一些重要的内核数据结构，对 rust 语言的支持也不是特别好.
- 因此，我们想用 eBPF 技术来实现跟踪功能.
  - eBPF 技术使得用户可以在内核执行用户自定义的程序.

利用 eBPF 进行跟踪

- 为了用上调试器，得改编译参数，改内核代码... 比较繁琐.
- GDB 无法跟踪 rCore 的一些重要的内核数据结构，对 rust 语言的支持也不是特别好.
- 因此，我们想用 eBPF 技术来实现跟踪功能.
  - eBPF 技术使得用户可以在内核执行用户自定义的程序.
  - eBPF 程序的可移植性比较好.

利用 eBPF 进行跟踪

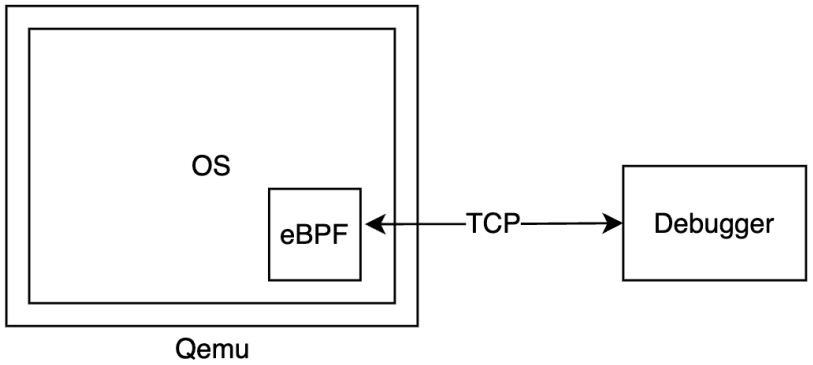
- 为了用上调试器，得改编译参数，改内核代码... 比较繁琐.
- GDB 无法跟踪 rCore 的一些重要的内核数据结构，对 rust 语言的支持也不是特别好.
- 因此，我们想用 eBPF 技术来实现跟踪功能.
  - eBPF 技术使得用户可以在内核执行用户自定义的程序.
  - eBPF 程序的可移植性比较好.
- 我们想要基于 eBPF 实现的功能：函数调用动态跟踪、单步断点、内存查看、寄存器查看、异步函数跟踪

利用 eBPF 进行跟踪

- 为了用上调试器，得改编译参数，改内核代码... 比较繁琐.
- GDB 无法跟踪 rCore 的一些重要的内核数据结构，对 rust 语言的支持也不是特别好.
- 因此，我们想用 eBPF 技术来实现跟踪功能.
  - eBPF 技术使得用户可以在内核执行用户自定义的程序.
  - eBPF 程序的可移植性比较好.
- 我们想要基于 eBPF 实现的功能：函数调用动态跟踪、单步断点、内存查看、寄存器查看、异步函数跟踪

基于 eBPF 的 gdbserver

- 我们最终希望做到的效果是，只要内核支持 eBPF，就可以在不修改内核代码的情况下进行基于 eBPF 程序的跟踪调试.



Thanks!