

# 开发记录和知识库

---

## 20220605

- 写文档

## 20220604

1. 自动化
  - a. 识别不同的断点=> 内核入口出口检测
  - b. 自动清除断点
  - c. 自动添加调试信息文件
  - d. 一键启动:
    - i. 打开panel
    - ii. make run
    - iii. start debugging
2. 断点的保存, 恢复先不弄, 现在先直接清空全部断点。等有办法判断接下来运行的是哪个应用程序了再添加这个功能。
3. bugs/todos
  - a. Sets multiple breakpoints for a single source and clears all previous breakpoints in that source.
  - b. 入口出口用setInstructionBreakPoints要更好 (插件里没有, DAP有)
  - c. <https://shimo.im/docs/hRQk6dXkxHp9pR3T#anchor-IKAf>
  - d. 关debugsession的时候应把webview和qemu也关了
  - e. trap\_handler前几行不能设断点
  - f. 等待窗口加载完毕且设置完断点后再launch
  - g. 显示隐藏的断点
  - h. `vscode.commands.executeCommand("workbench.debug.viewlet.action.removeAllBreakpoints");`  
无法删除隐藏的断点。 `vscode.debug.removeBreakpoints` 也有bug。要第二次调用才会生效。最好的办法是终端 `del` (清除所有断点) 加上  
`vscode.commands.executeCommand("workbench.debug.viewlet.action.removeAllBreakpoints");`  
(清除界面上显示的断点)
- 4.

## 20220602

1. 用户态时可以在内核态设断点, 但是有时函数被内联了所以会设置失败。
2. 由于测试的步骤比较繁琐, 我每一次尝试运行插件的时候流程都略有不同, 这导致错误复现不出来或者错误时而出现时而不出现。故统一测试流程:
  - a. 打开panel, make run
  - b. 设置入口 (如 `src/trap/mod.rs:135`。此时stvec等csr已修改完毕) 和 (非必须) 出口 (如`trap_return`) 断点。
  - c. continue。在出口附近停止。

- d. 删除全部断点,。
- e. 切换为initproc的调试信息文件。
- f. inkernel=false,
- g. 要求用户设置用户态的断点。
- i. 奇怪现象：此时13行可设置断点，14，15，和后面的部分均不行（-break-insert - f）。如果只设置13行断点，并执行到13行，可以设置14，15行的断点。在gdb或vscode里都是这样

```

1 //initproc.rs
2 11 fn main() -> i32 {
3 12
4 13 println!("aaaaaaaaaaaaa");
5 14 let a = getpid();
6 15 println!("{}",a);

```

- i. 奇怪现象2：我是用file命令切换到initproc的用户态文件的，但是在用户态居然能设置内核态断点，应该是pending breakpoints，所以设置了也没用。改用add-symbol-file解决了这个问题
  - ii. 不奇怪现象3：workbench.debug.viewlet.action.removeAllBreakpoints可以，workbench.debug.viewlet.action.disableAllBreakpoints导致直接跑飞
3. 接下来严格按照这个流程进行自动化

## 20220531

```

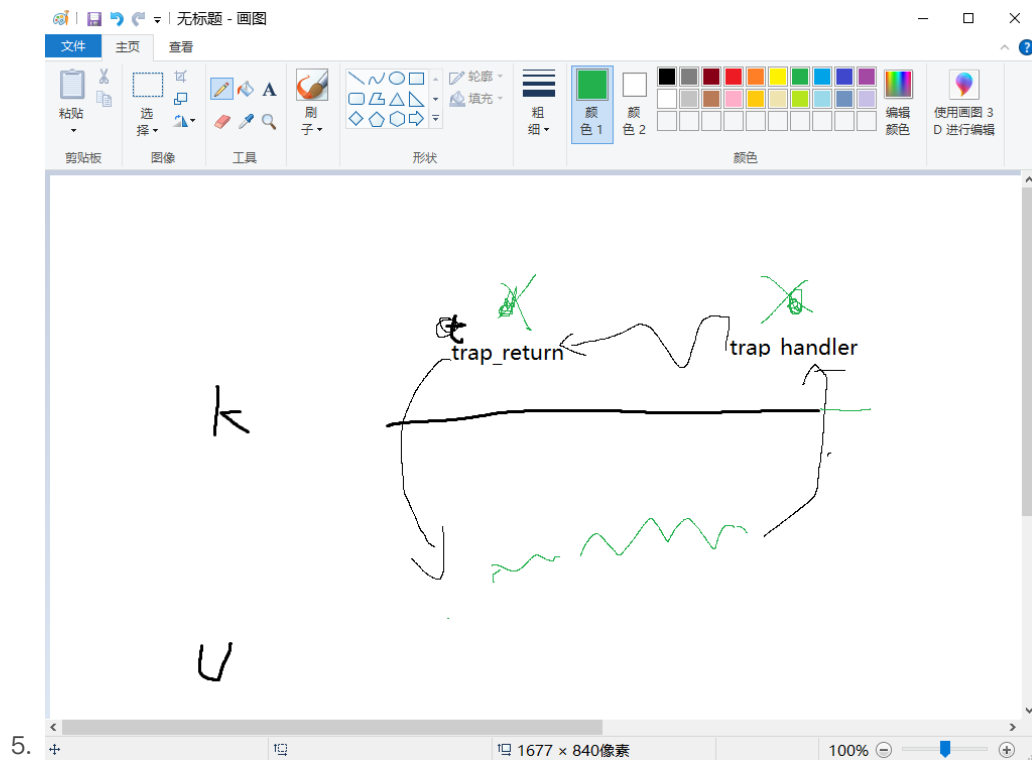
1 (gdb) si
2 0x00000000080209040      135      asm!(
3 (gdb) si
4 0xffffffffffffffff060 in ?? ()
5 (gdb) si
6 0xffffffffffffffff064 in ?? ()
7 (gdb) si
8 Warning:
9 Cannot insert breakpoint 3: Cannot access memory at address 0x80209004
10
11 Command aborted.
12
13 (gdb) b user_lib::_start
14 Cannot access memory at address 0x10014
15 (gdb) b trap_handler
16 Cannot access memory at address 0x80208d58
17 (gdb) info reg pc
18 pc      0xffffffffffffffff064      0xffffffffffffffff064
19 (gdb)

```

## 20200528-组会结论：

1. 将能展示的部分（处于用户态时设置内核态的断点）进行排查 分析成功和未成功的原因
2. 将错误分而治之 一步一步进行优化
3. 将能设置断点处和不能设置断点处分析明白，不能设置断点处直接指出

4. 每搞定一步，就在石墨文档写结果，开始下一步



5.

6.

20220527

## 崩溃原因

简单来说就是不能在用户态设置内核态的断点

RISC-V中文手册：……这条 `sfence.vma` 会通知处理器，软件可能已经修改了页表，于是处理器可以相应地刷新转换缓存……

推测是执行`sfence.vma`之后，TLB刷新成用户进程的页表，导致内核地址空间的断点无法被设置。

在vscode里，我在`trap_return`函数设断点，不断在debug cosole步进（用`ni`或`si`。只能一步一步来，不能用`si 100`。图形界面哪个是step into不是step instruction）直到报错“无法设置断点”。此时TLB已经刷新为用户进程的页表。此时取消内核态的断点，添加`initproc`的调试信息文件（`add-symbol-file`）添加用户态程序的断点（未报错或退出），`continue`。接下来步进，可以顺利到达`ecall`。在文字终端的gdb中，`ecall`之后再步进会出错，可此时在vscode里却回`trap_handler`了、

解决办法：[实现特权级的切换 — rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 \(rcore-os.github.io\)](#)

当 CPU 执行完一条指令（如 `ecall`）并准备从用户特权级陷入（`Trap`）到 S 特权级的时候，硬件会自动完成如下这些事情……CPU 会跳转到 `stvec` 所设置的 Trap 处理入口地址，并将当前特权级设置为 S，然后从 Trap 处理入口地址处开始执行……而当 CPU 完成 Trap 处理准备返回的时候，需要通过一条 S 特权级的特权指令 `sret` 来完成，这一条指令具体完成以下功能……CPU 会跳转到 `sepc` 寄存器指向的那条指令，然后继续执行。所以，似乎可以利用watchpoint？

断点缓存先不做，先弄断点删除，这样至少能跑

0. panel

1. os启动
2. 设置入口（如 `src/trap/mod.rs:135`。此时stvec等csr已修改完毕）和（非必须）出口（如 `trap_return`）断点。
3. continue。在出口附近停止。
4. 删除全部断点,。
5. 设置stvec为断点。切换为用户程序（哪个用户程序？这得改rcore）的调试信息文件。
6. `inkernel=false`,
7. 要求用户步进（我等下把step into 改成 si）或设置（恢复）用户态的断点。

当触发入口断点时，切换为os的调试信息文件，删除全部断点，`inkernel=false`,

问题：在用户态时，不论是trap\_handler还说stvec里的值，均为Cannot access memory at address....

## 获取当前停留的断点的信息

### 梳理流程

搞了几天还是一直崩溃，用户态的就是不稳定。我看先回到终端gdb，终端gdb弄出一套流程，然后vscode严格照着这个流程走，就肯定能行。除了崩溃，还有个问题就是设了的断点并不会在那里停下来。我看原因很可能是那个断点根本没设置成功。

- stopped
  - a. 更新信息
    - i. 寄存器
    - ii. 内存
    - iii. 停留在哪个断点
      1. 断点信息发送给网页，以此更新特权级信息
- 特权级切换
  - 在边界停下来
  - 删除旧断点
  - 步进
  - 进入用户态程序
  - 判断是哪个应用程序
  - 切换对应的符号表文件（用add-symbol-file。这样在用户态还能设置内核的断点）
  - enable用户态程序的断点
  - 如果回用户态，再回内核态，这个symbol file如何处理？明天再弄。

### 尝试添加gdbgui

1. 测试gdbgui是否能正常运作
2. 不行，比我这个插件还不稳定
- 3.

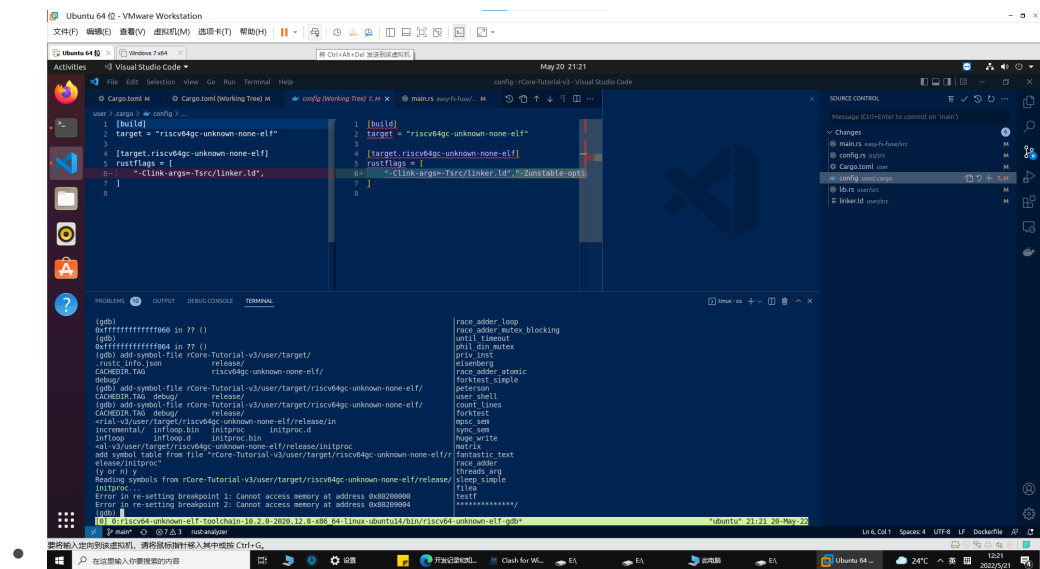
## 用户态跟踪

- `debug=true` 已经被加入repo了
- `os` 包含调试信息但是所有的应用程序都没有。在gdb中打开均显示no debugging symbol。
- 用 `file` 命令看了下, `os` with `debug_info`而用户程序没这个。
- 看了file的源码, 是通过查看有没有`.debug_info` 段来判断的。
- 通过 `rust-readobj` , 发现os有`.debug_info`, `.debug_arange`等段, 而用户程序没有。
- `opt-level=0` (显然) 没用。
- rcore群里问了, 没人理我的话明天去比赛群问
- 群里大佬说linker.ld里面故意discard了debug段 (文档里没说这事, 所以绕了一大圈都没发现) 去掉就好了。
- 然后生成fs.img磁盘镜像的时候出问题。将磁盘镜像改大 (`easy-fs-fuse/src/main.rs`的60和64行16->64) 就好了。在这里出现了一个很奇怪的报错, 没人 (包括吴一凡) 遇到过, 折腾了一阵子发现是user/目录要先 `make clean` 再编译, 修改过的linkerscript才会生效。
- 编译通过了, 但是用户程序没法在os上跑, Heap allocation error。这很奇怪, 新增的这些debug段明明会被 `objcopy` 的 `strip-all` 命令去除的。我换成 `strip-all-gnu` 等, 也没用。
- heap改大点可以, 要改很多地方, 而且k210上可能就跑不了了。
- 这里面涉及一些关于链接的知识我不太熟。这个问题先放着吧。
- rustc有一个`split-debuginfo`参数, 但是不稳定:

```
1 rustflags = [  
2     "-Clink-args=-Tsrc/linker.ld", "-Zunstable-options", "-Csplit-debuginfo=ur  
3 ]  
4 ==>  
5 Compiling bare-metal v0.2.5  
6 Compiling buddy_system_allocator v0.6.0  
7 error: A dwo section may not contain relocations  
8 error: could not compile `bare-metal` due to previous error  
9 warning: build failed, waiting for other jobs to finish...  
10 error: could not compile `spin` due to previous error  
11 error: could not compile `log` due to previous error
```

问rust社区, 答: 几百兆的exe才能显示你的实力.....

- 没辙了, 直接把堆开大吧, k210跑不了就跑不了吧
- 堆放大了四倍, 能跑了, 好耶



## 支持非通用寄存器

界面余叶弄，我先把数据弄进前端页面。获取不了会直接报错，得用 `try...catch` 。

## 后续安排

- 自动化
- 修bug，改进使用体验
  - inKernel
- 写文档

## 特权级信息准确获取

## 20220513

- webview重构
  - webview里面不要有太多逻辑。变成一个纯粹的输入输出页面，逻辑全部放到插件里面
- 支持CSR等非通用计算器
- 用户态的跟踪
  - 先用一般的gdb
  - 跟踪到用户态的入口
  - 进入用户态
  - 切换二进制文件
  - 尝试读取一些信息
  - 在vscode中提供支持（可能要调用vscode api）
- 特权级信息准确获取 除bug
  - 自动地检测到断点->修改特权级信息->继续执行
- ni->si

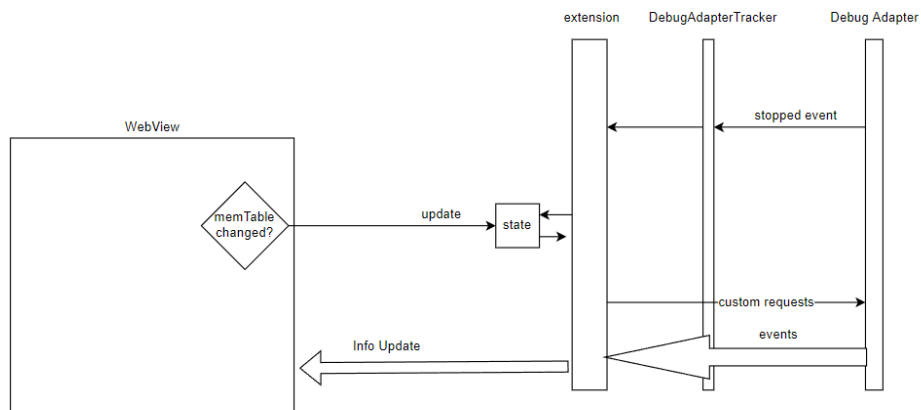
## 20220513 v0.0.4

- WebView重构
  - 退回到粗糙界面版本方便调试
  - 简化流程，不在WebView内部保留状态。需要时全部去插件外请求。
- 寄存器
  - 先弄U->S。S->M和SBI关系比较大。
  - 在gdb里，尝试找到它。
  - 实现customRequest

- 资料
  - qemu文档
  - gdb文档
  - qemu源码
- 特权级信息准确获取
  - 简单办法：
    - 在临界处设置断点
    - 监听断点事件，做出对应判断
- 修改符号表
  - `ChangeBinaryRequest`
- 传github并发布
  - os比赛的文档

## 20220507 v0.0.3

- 使用bootstrap.js提供更友好的界面
- 展示任意内存地址的信息。可自行添加
- DebugAdapter
- 文档
  - DebugAdapter CustomRequests

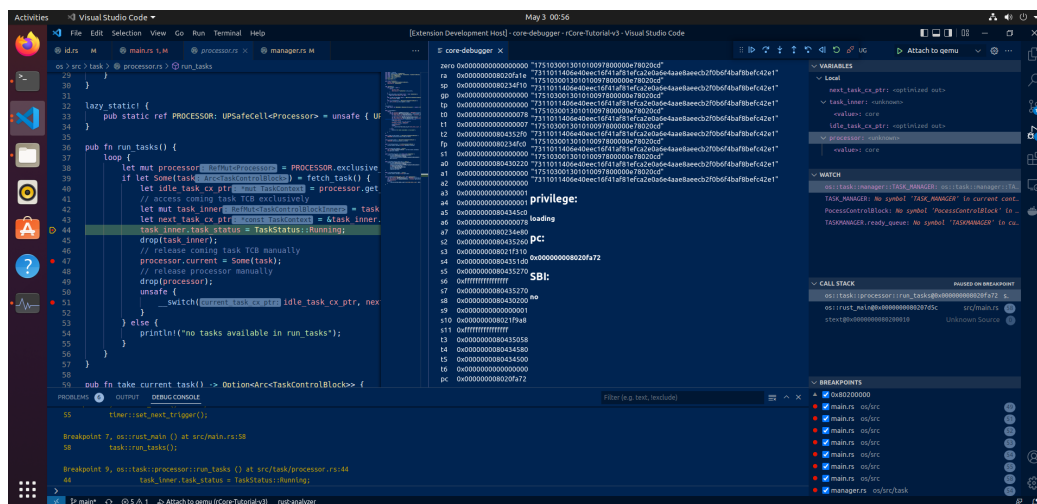


- 
- 传github

## 下周工作

1. 先搞定关键的寄存器和内存的数据获取；
2. 搞定当前特权级信息的准确获取；
3. 搞一个例子：在USM三态修改符号表，并获取内存单元信息；

## 20220430-速通 v0.0.1



- 已实现功能
  - 即时更新（当可指定的特定事件，如断点、暂停、步进发生时自动更新所有信息）：
    - 寄存器名和寄存器值
    - 是否处于SBI（根据pc位置判定）
    - 断点信息
    - 展示部分本地变量名
    - 内核栈信息
    - 展示指定内存地址的信息（图上privilege上面的那些，因为界面会复杂些还没做完，所以直接把信息输出出来了。测试请求：

```
1 [{from:0x80200000,length:16},{from:0x80201000,length:32}]
```

- bugs:
  - 编译器会优化掉TASK\_MANAGER内部的内容
    - 尝试过：
      - `black_box()`
      - `opt-level =0或1`
      - `#[Debug]`
  - breakpoints窗口如果直接指定内存地址的断点，请求无效。这个功能可以移动到webView panel上实现
- todo:
  - 根据地址空间判断特权级
  - 改进界面
  - 切换符号表
- frontend
  - Webview Panel
    - 消息传送机制
      - 传入
        - `vscode.debug.onDidReceiveDebugSessionCustomEvent` [How to send custom event from Debug Adapter to vscode client? · Issue #113538 · microsoft/vscode \(github.com\)](#) 当Debug Adapter产生事件时（如断点，异常，步进完成）
        - `debugSessionEventToWebview` 获取所需寄存器，内存的相关信息
        - vscode仅提供 `customRequest` 接口，并未提供代码实现。
        - `currentPanel.webview.postMessage` 传入webview进行前端展示
      - 流程图
      - 传出



- Debug Adapter
  - Debug Adapter Protocol 未包含请求寄存器的Request. 因此自己添加一个
    - 添加Tracker
    - `vscode.debug.activeDebugSession?.customRequest(command)`` 发送请求
    - `` 接收请求, 返回结果 (通过一个customEvent)
    - `getRegisters()` 通过终端命令得到寄存器信息
    - 传入WebView Panel
  -

```

1 impl fmt::Debug for Point {
2     fn fmt(&self, f: &mut fmt::Formatter<'_,>) -> fmt::Result {
3         f.debug_struct("Point")
4             .field("x", &self.x)
5             .field("y", &self.y)
6             .finish()
7     }
8 }

```

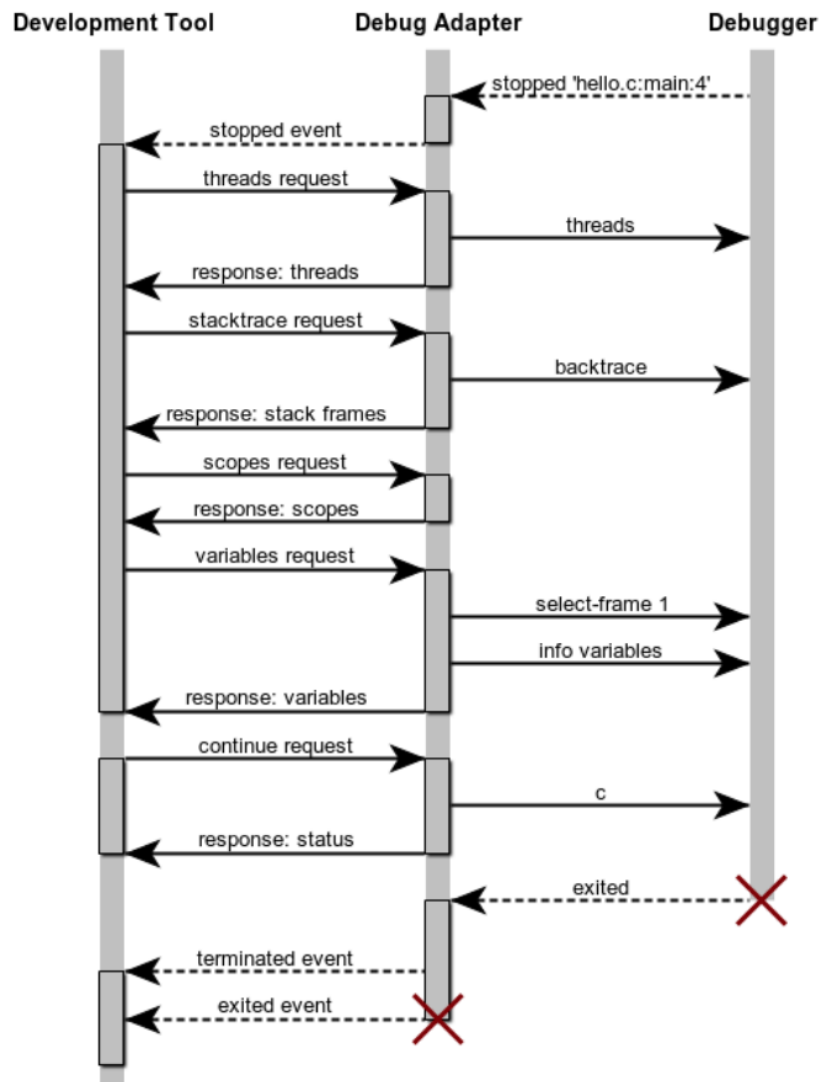
- rust-gdb
- 其它
  - 如果google不到任何文档信息, 可以直接在github里搜代码
- 待完善:
  - 添加StepInstruction
  - 第一次断点没数据的bug
  -

## 20220430-计划

1. 速通
2. 写分析文档

## 20220430-进展

1. 半成品
  - a. 读取符号表->设置断点->
  - b. debugger 插件分析
    - i. 消息传送机制
      1. vscode webview
        - a. `WebViewView` 仅有示例代码
      2. Debug Adapter [Overview \(microsoft.github.io\)](https://microsoft.github.io/debugger-protocol-1.56/)
        - a. 要素: `Requests` 和 `Respond`



b.

c. 封装

i. 有时可能需另起炉灶

d. PrivilegeRequest->MemoryRequest, RegisterRequest, .....

c. 设置断点

i. BreakpointRequest

d. 前端展现获取的信息

20220422

## 进展

1. 二进制代码完整跟踪 ✓ 原因：内核态断点

2. 状态信息获取

a. SBI的状态：地址 0x80200000 之前

a. 特权级：sstatus 的 SPP 字段不一定表示当前的特权级，保存的是trap之前的特权级

i. 解决办法：执行特定指令时？

b. 进程控制块信息

3. 插件分析

a.



code-debug.drawio  
2KB

预览

## 计划

只要有指令，寄存器，内存信息，就可以知道操作系统的大多数状态（特权级，内核进程，多个用户进程.....）。

1. 做一个半成品
  - a. 符号表读取
2. 利用操作系统自身的信息（主要是内存地址空间）来判断特权级等信息
  - a. 内核态->用户态：在 `trap_return` 附近设置断点
  - b. 用户态->内核态：（uCore, rCore都可以）在跳转代码附近设置断点
  - c. M->S：函数，syscall
3. should be specifically configured for different OSes

## 20220415

1. 编写一个文档，其展示了最终成品具有的功能
2. 修改插件，使其能够读取并解析符号文件的信息（配合objdump）

## 20220409-后续工作安排

1. 二进制代码的完整跟踪：获取二进制信息；

```
(gdb)
0x0000000080210f2e in os::trap::trap_return () at src/trap/mod.rs:101
101         let user_satp = current_user_token();
(gdb)
108         asm!(
(gdb)
0x0000000080210f34      108      asm!(
(gdb)
0x0000000080210f38      108      asm!(
(gdb)
0x0000000080210f3c      108      asm!(
(gdb)
0x0000000080210f40      108      asm!(
(gdb)
0x0000000080210f44      108      asm!(
(gdb)
0x0000000080210f48      108      asm!(
(gdb)
0x0000000080210f4a      108      asm!(
(gdb)
0x0000000080210f4e      108      asm!(
(gdb)
0xffffffffffffffff060 in ?? ()
(gdb)
0xffffffffffffffff064 in ?? ()
(gdb)
Warning:
Cannot insert breakpoint 3: Cannot access memory at address 0x80200000
Command aborted.
a. (gdb) si
```

```

0x00000000080210f48      108      asm!(
(gdb)
0x00000000080210f4a      108      asm!(
(gdb)
0x00000000080210f4e      108      asm!(
(gdb)
0xfffffffffffff060 in ?? ()
(gdb)
0xfffffffffffff064 in ?? ()
(gdb)
Warning:
Cannot insert breakpoint 3: Cannot access memory at address 0x80200000

Command aborted.
(gdb) d
Delete all breakpoints? (y or n) y
(gdb) si
0xfffffffffffff068 in ?? ()
(gdb)
0xfffffffffffff06c in ?? ()
(gdb)
0xfffffffffffff06e in ?? ()
(gdb)
b. "97a616a34ebc" 04:12 15-Apr-22 [28/1865]
(gdb)
0xfffffffffffff054 in ?? ()
(gdb)
0xfffffffffffff056 in ?? ()
(gdb)
0xfffffffffffff05a in ?? ()
(gdb)
0xfffffffffffff05e in ?? ()
(gdb)
os::trap::trap_handler () at src/trap/mod.rs:42
42      pub fn trap_handler() -> ! {
(gdb)
riscv::register::stvec::write (bits=<optimized out>)
    at /home/workspace/.cargo/git/checkouts/riscv-ab2abd16c438337b/11d43cf/src/register/ma
cros.rs:105
105      () => core::arch::asm!("csrrw x0, {1}, {0}", in(reg) bits, const $
csr_number),
(gdb)
0x00000000080210bb4      105      () => core::arch::asm!("csrrw x0, {1}, {0$
", in(reg) bits, const $csr_number),
(gdb)
0x00000000080210bb6      105      () => core::arch::asm!("csrrw x0, {1}, {0$
", in(reg) bits, const $csr_number),
C.

```

d. 把断点删掉就可以内核态-> `trap_return()` ->用户态-> `trap_handler()` ->内核态了。寄存器可以看, `bt` 无信息。吴老师说可能是地址被转换过了

2. : 特权级、进程控制块信息、SBI的状态、

- 特权级:
- 进程控制块信息:
- SBI的状态: pc

3. 在用户态、内核态和SBI中在指定位置设置断点;

4. Debugger插件的分析;

5. 在前端展现获取的信息;

## 4月1日组会

1. 测试应用程序是否能正常输出

- 能.usertests能跑, 但是gdb追踪不到。是哪一部追踪不到? 在哪个地方切换回应用态?
  - 改用SI追踪到了

2. 测试应用程序是否可以调用系统调用

- usertests跑通了。可以的。只是gdb追踪不到
  - 改用si追踪到了

3. 能否跳过rCore-Tutorial, 直接debug应用程序?

- 有qemu-riscv64但根据rCore-Tutorial-Book, 兼容性较差。

4. 除了gdbserver, qemu是否提供了其他调试方案

- 没找着

5. 有可能是gdb和rCore-Tutorial的shell（调用了 `getchar()` ）共用一个管道导致冲突。试着排查该问题。
- a. 改用tmux解决了。但是为啥？

## 计划

### ☒ 复现gdbgui

☒ 问题：docker镜像中给出的gdbgui的ip和实际用的ip不一致，修改[这里](#)应该就能解决。

☒ 开启docker镜像后还要在docker容器中再输入一次命令。很不方便。在Dockerfile中添加CMD，和修改[这里](#)应该就能解决。

### ☒ 参考rCore-Tutorial实现gdb调试rCore on qemu

☒ 观察符号信息在哪里？

### ☒ 写文档

### ☒ 实现通过gdbgui调试(gdb=>rCore on qemu)

☒ rCore-Tutorial的debug功能

☒ wyf说理论上能用的，试试照着实验指导书中的指令看看

### ☒ 编写主站

☒ 设计html

### ☒ 发布github仓库然后发给向老师和群里

☒ 删去不必要的文件

☒ 一个markdown的使用指南

☒ Dockerfile

☒ 工具链可能要用LFS或者让用户自己去下载,gitignore要忽略掉

☒ 上传镜像到docker-hub

☒ 让dockerfile好看点

### ☐ 添加便捷功能

☒ - `$foldername` -

☒ - `fast_git` -

☒ 一键gdbgui（就是加三行启动命令）

☒ 复位功能

☒ 将makefile修改为全局均可执行

☒ 解决rust-analyzer failed to discover workspace

### ☒ 提升稳定性。杜绝跑飞

☒ 随便加断点。

☒ 先试试Hello-World

☒ 进入shell等待用户输入

☒ scanf阻塞

☒ gdb绷不住了，找不到user目录下程序的符号信息。切换到用户程序时出的问题

☒ idle绷不住。换成vscode插件后不会绷不住了。是gdbgui的问题

☒ virt是否是单线程？

☒ 找到跑飞的最后一个程序。找到了，是shell。之前在shell之前崩溃是gdbgui的问题

☒ gdbgui本身的调试机制

☒ python部分

☒ node.js部分

☒ 自己做一遍rCore实验，多使用gdbgui。要是使用时出了什么问题，使用这个debug机制看看出了什么bug。

### ☒ 融合

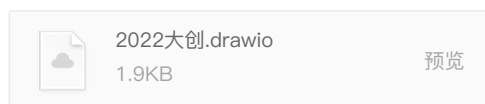
☒ 搞个插件，把vscode和gdbgui合在一起。

☒ 弄懂[Debugger Extension | Visual Studio Code Extension API](#)

- ☒ 写Debug Adapter。可能可以抄gdbgui的代码。
- ☐ 没有找到源代码的地方，show assembly instead
- ☐ core库源码无法显示。可能和编译方式有关。可以问问群里大佬
- ☒ vscode设置 允许随意设置断点（是否需要？）
- ☒ 添加堆栈查看
- ☒ 排查问题。是gdb没生成堆栈信息还是gdbgui没解析？结果：如果将指令在开启gdbgui时自动启动，堆栈就可以解析，应该是gdbgui的bug。
- ☐ 写论文（4月开始写，5月初投稿）
  - ☐ 内容
    - ☐ 设计，算法实现，一组能展示之实验
    - ☐ 实验平台，如何设计，问题，用到什么模块，什么都是自己搞的，搭建，使用，测试
- ☐ 高级功能
  - ☐ 利用mozilla rr 提供回退功能（见gdbgui文档）
  - ☐ 尝试解决[Rust \(Debugging with GDB\) \(sourceware.org\)](#)里提到的问题
  - ☐ 参考rust-gdb见[Debugging Rust apps with GDB – LogRocket Blog](#)
  - ☐ 多用户
  - ☐ 用户文件存储
  - ☐ 花哨的图形界面
- ☐ 支持uCore等其他代码仓库
  - ☐ 符号表问题
- ☐ 将qemu替换成实体板子

## 开发记录

### 流程图和文档（12月22日）



这个图是半成品。FPGA调试要再去熟悉一下，然后再改一改。  
文档还没写。期末考完再写。

->文档<https://shimo.im/docs/TwHhy6VwpWYg3crQ>

### 1月12日

文档和流程图开了个头

### 1月18日之前

- czy 编写实验平台的构思的文档<https://shimo.im/docs/TwHhy6VwpWYg3crQ>，然后跑通树莓派的项目
- lzf 实现实验平台的前端和后端服务器程序
- zwz 17日之前实现<https://rcore-os.github.io/rCore-Tutorial-Book-v3/chapter0/5setup-devel-env.html>

### gdbgui

编写一个Dockerfile. 参考了[docker x gdbgui 在docker中debug c++ 代码](#)

```
1 FROM ubuntu:18.04
2 # LABEL maintainer="dinghao188" \
```

```

3 #         version="1.1" \
4 #         description="ubuntu 18.04 with tools for tsinghua's rCore-Tutorial-\
5 RUN sed -i 's/archive.ubuntu.com/mirrors.ustc.edu.cn/g' /etc/apt/sources.lis
6 RUN apt-get update && apt-get install -y \
7 vim \
8 zsh \
9 git \
10 g++ \
11 gdb \
12 python3 \
13 python3-pip \
14 python3-venv \
15 wget \
16 make
17 # userpath append not working
18 RUN export LC_ALL=C.UTF-8 && export LANG=C.UTF-8 && python3 -m pip install -
19 RUN ~/.local/bin/pipx install gdbgui

```

构建这个容器：

```

1 docker build -t gdbgui-test .

```

运行：

```

1 docker run --cap-add=SYS_PTRACE --security-opt seccomp=unconfined \
2 -it --rm \
3 -v $PWD:/sharedFolder \
4 --name gdbgui-test-1 \
5 -p 5000:5000 \
6 gdbgui-test /bin/zsh

```

进入docker后：

```

1 ~/.local/bin/gdbgui -r

```

在主机浏览器进入127.0.0.1:5000即可看到gdbgui界面。

## gdb如何调试rCore-Tutorial on qemu

rCore-Tutorial的仓库里有，在os/Makefile里。

<https://github.com/rcore-os/rCore-Tutorial-v3/blob/c358424faed90cab0a0f0ee92b6c594f0f577045/os/Makefile#L103>

```

1 debug: build
2     @tmux new-session -d \
3         "qemu-system-riscv64 -machine virt -nographic -bios $(BOOTLOADER)
4         tmux split-window -h "riscv64-unknown-elf-gdb -ex 'file $(KERNEL
5         tmux -2 attach-session -d

```

第三行中，

qemu-system-riscv64 -machine virt -nographic -bios \$(BOOTLOADER) -device loader,file=\$(KERNEL\_BIN),addr=\$(KERNEL\_ENTRY\_PA)开启qemu， 开启 (stub? ) gdb， 暂停qemu虚拟的程序的执行。

不懂看这：

关键是这一行：

```
1 riscv64-unknown-elf-gdb -ex 'file $(KERNEL_ELF)' -ex 'set arch riscv:rv64' -
```

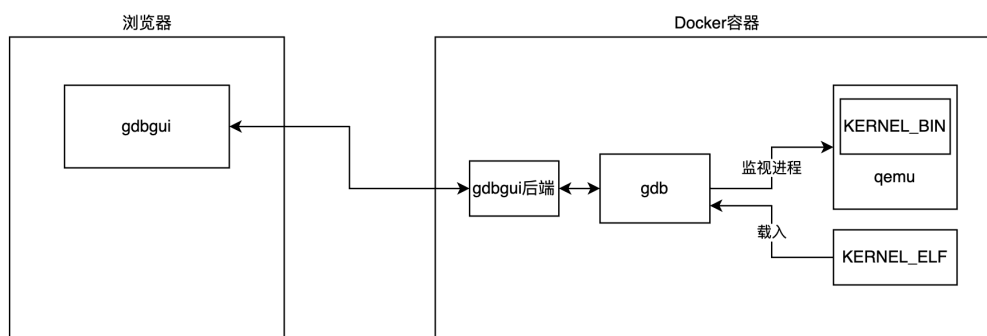
`-ex` 表示执行一个gdb指令。<https://shimo.im/docs/hRQk6dXkxHp9pR3T#anchor-RSAs>

`file` 表示载入一个文件。

gdb载入了KERNEL\_ELF，而qemu载入了KERNEL\_BIN。KERNEL\_BIN是KERNEL\_ELF去除了符号信息后（具体在老rCore-Tutorial-v3有，得找过来）得到的。

看来我们的实验平台也要这么做。gdb需要一个带有符号信息的可执行文件。

计划这样做：



gdbgui-gdb-qemu-kernel...  
1.5KB

预览

## gdbgui调试[ rCore-Tutorial on qemu ]

首先要编译一遍rCore-Tutorial，得到KERNEL\_BIN和KERNEL\_ELF。

为此先在Dockerfile中添加相关的依赖：

```
1 FROM ubuntu:18.04
2 # RUN sed -i 's/archive.ubuntu.com/mirrors.ustc.edu.cn/g' /etc/apt/sources.list
3 # installing gdbgui
4 RUN apt-get update && apt-get install -y \
5 vim \
6 zsh \
7 git \
8 g++ \
9 gdb \
10 python3 \
11 python3-pip \
12 python3-venv \
13 wget \
14 make
15 # userpath append not working
16 RUN export LC_ALL=C.UTF-8 && export LANG=C.UTF-8 \
17     && python3 -m pip install --upgrade pip \
18     && pip3 install gdbgui
19     # && python3 -m pip install --user userpath urllib3 pipx \
20     # && python3 -m userpath append ~/.local/bin
```



```

21 # RUN ~/.local/bin/pipx upgrade-all
22 # RUN ~/.local/bin/pipx install gdbgui
23 #install some deps for rCore-Tutorial
24 RUN set -x \
25     && apt-get update \
26     && apt-get install -y curl wget autoconf automake autotools-dev curl libk
27         gawk build-essential bison flex texinfo gperf libtool patchutils
28         zlib1g-dev libexpat-dev pkg-config libgl2.0-dev libpixman-1
29
30 #install rust and qemu
31 RUN set -x; \
32     RUSTUP='/root/rustup.sh' \
33     && cd $HOME \
34     #install rust
35     && curl https://sh.rustup.rs -sSf > $RUSTUP && chmod +x $RUSTUP \
36     && $RUSTUP -y --default-toolchain nightly --profile minimal \
37     #compile qemu
38     && wget https://ftp.osuosl.org/pub/blfs/conglomeration/qemu/qemu-5.0.0.t
39     && tar xvjf qemu-5.0.0.tar.xz \
40     && cd qemu-5.0.0 \
41     && ./configure --target-list=riscv64-sofmmu,riscv64-linux-user \
42     && make -j$(nproc) install \
43     && cd $HOME && rm -rf qemu-5.0.0 qemu-5.0.0.tar.xz
44 #for chinese network
45 RUN set -x; \
46     APT_CONF='/etc/apt/sources.list'; \
47     CARGO_CONF='/root/.cargo/config'; \
48     BASHRC='/root/.bashrc' \
49     && echo 'export RUSTUP_DIST_SERVER=https://mirrors.ustc.edu.cn/rust-stat
50     && echo 'export RUSTUP_UPDATE_ROOT=https://mirrors.ustc.edu.cn/rust-stat
51     && touch $CARGO_CONF \
52     && echo '[source.crates-io]' > $CARGO_CONF \
53     && echo "replace-with = 'ustc'" >> $CARGO_CONF \
54     && echo '[source.ustc]' >> $CARGO_CONF \
55     && echo 'registry = "git://mirrors.ustc.edu.cn/crates.io-index"' >> $CAF

```

sifive官网只提供了x86的工具链，所以m1 mac不好使了，去整个x86 ubuntu

电池鼓包顶到键盘了，导致CTRL松不开，离谱

然后写了一个Makefile，它可以做上边图片里的事：启动qemu（暂时先直接用rCore-Tutorial里的make run）、启动gdbgui后端（gdbgui后端会启动gdb）。用户访问gdbgui网站后手动载入KERNEL\_ELF、监视qemu进程（用户手动操作）。

1月24日：以上基本都弄好了，但是发现这些小玩意缝合起来很费劲。我再加把劲乱糊一通也是能凑合用的，但是对后续开发极不利。此时应设计一个简单的规范，以后所有代码仓库都依照这个规范进行提交。

这个规范里有：

- 代码在哪里（直接上传，仓库拉取）
- 在进入gdbgui之前先编译
- 指出符号文件的位置
- 指出内核文件的位置
- 指出用哪款gdb
-

再画一个图，说明这个规范：

最后脚本如下：

1

最后修改了Dockerfile使得Mekefile和工具链在构建docker镜像时就被复制进docker镜像的文件系统的home（即 `/root`）目录。

最后Dockerfile如下：

1

## 1月29日

编译的时候想保留调试信息有两种办法，刚开始我选择直接把`--release`选项去掉，结果惹了一堆麻烦。后来换了第二种，在Cargo.toml里添加`debug=true`属性就可以了。

此外犯了一些低级错误，比如gdbgui里有个↓按钮，我想当然地以为这就是步进按钮，但实际上那个按钮的功能是执行到下一函数调用。按了一下自然就一泻千里.....

还有一个低级错误，第二种保留调试信息的办法其实几天前就尝试过，但是当时打开gdbgui一看全是汇编，就以为这办法不行.....os启动的那几行代码本来就是汇编写的，当然没有源代码可供显示😂启动之后进入的是rustsbi，rCore-Tutorial是直接把编译好的rustsbi二进制文件缝合进内核的，显然也不会有源代码。之前gdb调试起来有问题，si会卡在一处地址不动，去问了wyf大佬，他建议严格照着实验指导书执行以下试试，然后今天理解了rCore-Tutorial-v3-Book的[相关章节](#)中的gdb指令和执行结果（顺便学了学怎么用gdb）才发现是怎么回事.....

现在可以停在第一个函数调用并且显示源代码了。保留调试信息有两种办法，刚开始我直接把`--release`选项去掉，结果因为rust编译器对release和debug模式的代码的检查规则是不一样的，东西越改越多，后来换了第二种办法，保留`--release`不变，在Cargo.toml里添加`debug=true`属性就可以了，

## Debug Adapter

[How to send custom event from Debug Adapter to vscode client? · Issue #113538 · microsoft/vscode \(github.com\)](#)

## rCore-Tutorial-v3

- debug模式编译

The image shows a side-by-side comparison of two Makefile configurations. The left side is the original configuration for release mode, and the right side is the modified configuration for debug mode. The changes are highlighted with red and green boxes.

Original Makefile (Left)	Modified Makefile (Right)
1 # Building	1 # Building
2 TARGET := riscv64gc-unknown-none-elf	2 TARGET := riscv64gc-unknown-none-elf
3- MODE := release	3+ MODE := debug
4 TARGET_DIR := target/\${TARGET}/\${MODE}	4 TARGET_DIR := target/\${TARGET}/\${MODE}
5 APPS := \$(wildcard \${APP_DIR}/*.rs)	5 APPS := \$(wildcard \${APP_DIR}/*.rs)
6 patsubst \${APP_DIR}/%.rs, \${TARGET_DIR}/%, \$(APPS)	6 ELFS := \$(patsubst \${APP_DIR}/%.rs, \${TARGET_DIR}/%, \$(APPS))
7 patsubst \${APP_DIR}/%.rs, \${TARGET_DIR}/%.bin, \$(APPS)	7 BINS := \$(patsubst \${APP_DIR}/%.rs, \${TARGET_DIR}/%.bin, \$(APPS))
8	8
9 OBJDUMP := rust-objdump --arch-name=riscv64	9 OBJDUMP := rust-objdump --arch-name=riscv64
10 OBJCOPY := rust-objcopy --binary-architecture=riscv64	10 OBJCOPY := rust-objcopy --binary-architecture=riscv64
11	11
12 elf: \$(APPS)	12 elf: \$(APPS)
13- @cargo build --release	13+ @cargo build

- launch.json

- 注意rCore-Tutorial将os和user改为debug模式后会导致无法执行用户程序（包括shell）

- 仅用户程序debug模式

- 

## rust工具链

- [rust – How to rustup update when permission is denied? – Stack Overflow](#)
- Blocking waiting for file lock on package cache
- `cargo clean`
- ``cargo run -s`` 报错
  - 改为 ``cargo run -- -s``
  - 注意Core-Tutorial将os和user改为debug模式后会导致无法执行用户程序（包括shell）！
- rust-readobj, rust-objdump等
  - 先 `cargo install cargo-binutils` 再 `rustup component add llvm-tools-preview`

## makefile

### include

- 在 Makefile 使用 include 关键字可以把别的 Makefile 包含进来，这很像 C 语言的 `#include`，被包含的文件会原模原样的放在当前文件的包含位置。

### 赋值符号的区别

#### Makefile 中:= ?= += =的区别 – wanqi – 博客园 (cnblogs.com)

在Makefile中我们经常看到 `=` `:=` `?=` `+=` 这几个赋值运算符，那么他们有什么区别呢？我们来做个简单的实验...

 <https://www.cnblogs.com/wanqieddy/archive/2011/09/21/2184257.html>

## docker

### 将本地文件复制到docker中

#### docker将本地文件添加到镜像中\_怡宝2号-CSDN博客\_docker添加镜像文件

有两种实现方案1. `docker cp`命令`docker cp`只能拷贝到容器中，不能直接拷贝到镜像1.1 启动一个容器do...

 <https://blog.csdn.net/u011622208/article/details/103491095>

注意ADD \*.tar.gz 会自动解压这个文件。

### EXPOSE 暴露端口

格式为 `EXPOSE <端口1> [<端口2>...]`。

[使用帮助](#)

[确定](#)

EXPOSE 指令是声明容器运行时提供服务的端口，这只是一个声明，在容器运行时并不会因为这个声明应用就会开启这个端口的服务。在 Dockerfile 中写入这样的声明有两个好处，一个是帮助镜像使用者理解这个镜像服务的守护端口，以方便配置映射；另一个用处则是在运行时使用随机端口映射时，也就是 `docker run -P` 时，会自动随机映射 EXPOSE 的端口。

要将 EXPOSE 和在运行时使用 `-p <宿主端口>:<容器端口>` 区分开来。`-p`，是映射宿主端口和容器端口，换句话说，就是将容器的对应端口服务公开给外界访问，而 EXPOSE 仅仅是声明容器打算使用什么端口而已，并不会自动在宿主进行端口映射。

## 环境变量

我在dockerfile中添加CMD ...后，老是报错找不到rustup，但是在这个容器的shell里输入rustup是有反馈的。

[linux – When installing Rust toolchain in Docker, Bash `source` command doesn't work – Stack Overflow](#)

显示所有环境变量：`printenv`

## **gdb**

[gdb QuickStart \(umich.edu\)](#)

[File Options \(Debugging with GDB\) \(sourceware.org\)](#)

[Debugging with GDB – Summary \(gnu.org\)](#)

[QEMU/Debugging with QEMU – Wikibooks, open books for an open world](#)

[Debugging with GDB – GDB/MI Program Control \(gnu.org\)](#)

### **–ex**

Execute a single GDB command.

This option may be used multiple times to call multiple commands. It may also be interleaved with ‘-command’ as required.

```
1 gdb -ex 'target sim' -ex 'load' \  
2 -x setbreakpoints -ex 'run' a.out
```

## **rust支持**

[Rust \(Debugging with GDB\) \(sourceware.org\)](#)

## **好用的指令**

```
1 display/4i $pc
```

## **gdbgui**

### **–r**

run on a server and host on 0.0.0.0. Accessible to the outside world as long as port 80 is not blocked.

### **use gdb binary not on your \$PATH**

`gdbgui --gdb-cmd build/mygdb`

### **KeyError: WERKZEUG\_SERVER\_FD**

<https://github.com/cs01/gdbgui/issues/425>

## **shell**

## **修改系统代理**

Linux修改系统代理 – JoXrays's Blog

linux系统代理可以通过shell修改，其效果作用于当前shell或者所有shell，对于像chrome,firefox等桌面浏览...

 <https://www.joxrays.com/linux-system-proxy/>

## gitlab

### 命令行指引

您还可以按照以下说明从计算机中上传现有文件。

#### Git 全局设置

```
1 git config --global user.name "ATOM"
2 git config --global user.email "chenzhiy2001@qq.com"
```

#### 创建一个新仓库

```
1 git clone git@gitlab.eduxiji.net:chenzhiy/project788067-87777.git
2 cd project788067-87777
3 touch README.md
4 git add README.md
5 git commit -m "add README"
6 git push -u origin master
```

#### 推送现有文件夹

```
1 cd existing_folder
2 git init
3 git remote add origin git@gitlab.eduxiji.net:chenzhiy/project788067-87777.git
4 git add .
5 git commit -m "Initial commit"
6 git push -u origin master
```

#### 推送现有的 Git 仓库

```
1 cd existing_repo
2 git remote rename origin old-origin
3 git remote add origin git@gitlab.eduxiji.net:chenzhiy/project788067-87777.git
4 git push -u origin --all
5 git push -u origin --tags
```

## github

### 访问github加速

提高国内访问 github 速度的 9 种方法! – SegmentFault 思否

<https://segmentfault.com/a/1190000038298623>

无需代理直接加速各种 GitHub 资源拉取 | 国内镜像赋能 | 助力开发 – Frytea's Blog

## git

### git设置代理

## tmux

[Getting Started · tmux/tmux Wiki \(github.com\)](#)

```
tmux new
```

```
C-b % Split window
```

```
C-b [ or C-b ] 切换窗口
```

```
C-b + [ Scroll mode. q to quit.
```

## 复现软硬件协同的用户态中断时遇到的错误

- <https://shimo.im/docs/PgwQPxkhG8tJgQ3t/read>

## qemu

```
1 ERROR: glib-2.48 gthread-2.0 is required to compile QEMU
2 https://github.com/Xilinx/qemu/issues/40
3 $ apt install libglib2.0-dev
```

```
1 ERROR: pixman >= 0.21.8 not present.
2 Please install the pixman devel package.
3 $ apt install libpixman-1-dev
```

已经编译完成。

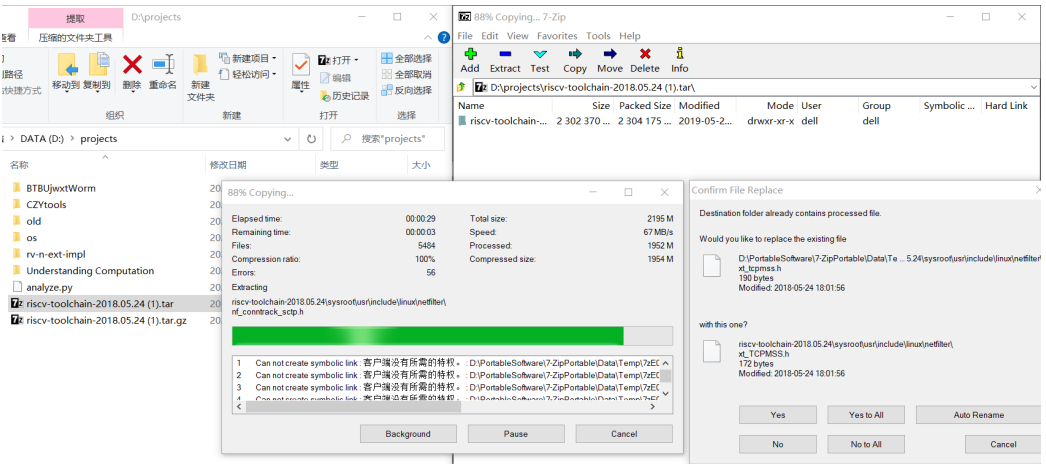
## rCore-N on qemu

没遇到啥障碍，一次跑通。

```
/dev/tty2
[rustsbi] RustSBI version 0.2.0-alpha.4
[rustsbi] Implementation: RustSBI-QEMU Version 0.0.1
[rustsbi-dtb] Hart count: cluster0 with 4 cores
[rustsbi] misa: RV64A
CDFIMNSU
[rustsbi] mideleg: usoft, utimer, uext, ssoft, stimer, sext (0x333)
[rustsbi] medeleg: ima, ia, bkpt, la, sa, uecall, ipage, lpage, spage (0xblab)
n 64 cfigs_in_pmpcfg 8 pmpcfg_max_id 2
i 0 j 0 pmpaddr_id 0
i 0 j 1 pmpaddr_id 1
i 0 j 2 pmpaddr_id 2
i 0 j 3 pmpaddr_id 3
i 0 j 4 pmpaddr_id 4
i 0 j 5 pmpaddr_id 5
i 0 j 6 pmpaddr_id 6
i 0 j 7 pmpaddr_id 7
[rustsbi] pmp0: 0x0 ..= 0x3fffffffffffffff (rwx)
[rustsbi] enter supervisor 0x80200000
[hart 0]satp: 0x8000000000000000
54b, sp: 0x80319ed0
[hart 0]Hello
[hart 1]satp: 0x8000000000000000, sp: 0x80329ed0
[hart 3]satp: 0x8000000000000000, sp: 0x80349ed0
[hart 2]satp: 0x8000000000000000, sp: 0x80339ed0
[hart 1]Hello
[hart 3]Hello
[hart 2]Hello
```

# FPGA环境配置

## 不应该在windows下解压工具链



## object not found

在这一步

```
1 # Build RISC-V image
2 # Build riscv-pk (which contains the bootloader), riscv-linux and riscv-rootfs
3 mkdir build
4 make -j sw # change 16 to the number of cores according to your host
```

会出现object not found的问题。这是手动终止make过程导致的。把整个仓库删掉重来能解决。

## 不要混用工具链

此外，

```
1 hello.c:1:10: fatal error: stdio.h: No such file or directory
2     1 | #include <stdio.h>
3       |           ^~~~~~
4 compilation terminated.
```

是因为混用了apt提供的riscv工具链和文档推荐的工具链造成的。一个简单粗暴的办法是卸载apt提供的工具链。

## 链接器不支持动态链接

```
1 make[1]: Entering directory '/mnt/d/projects/rv-n-ext-impl/sw/riscv-linux'
2 CALL scripts/checksyscalls.sh
3 <stdin>:1335:2: warning: #warning syscall rseq not implemented [-Wcpp]
4 CHK include/generated/compile.h
5 VDSO arch/riscv/kernel/vdso/vdso.so.dbg
6 /mnt/d/projects/riscv-toolchain-2018.05.24/bin/../lib/gcc/riscv64-unknown-elf/7.5.0/collect2: error: ld returned 1 exit status
```

陈乐：那个链接器不支持把.o链接成共享库，只能是可执行文件。关于这个问题，我发邮件问过做相关工作的人，得到的回答是不是riscv的问题，编译成.o时设置为位置无关代码，是

可以链接成共享库的。

我将`rv-n-ext-impl\sw\riscv-linux\arch\riscv\kernel\vdso\Makefile`中的`-shared`删掉，成功编译出了`linux.bin`文件。这样做可能会为未来留下隐患，好在这个项目不使用`riscv-linux`。

☐ 试试换一个链接器

## python?

```
1 /usr/bin/env: 'python': No such file or directory
```

For ubuntu 20.04 you can use following package to python command. And it is python 3.

```
sudo apt-get install python-is-python3
```

没完没了.....

```
1 spike-dasm: error while loading shared libraries: libriscv.so: cannot open s
2 ch file or directory
```

这头先暂缓。先试试fpga吧。

## lrv-rust-bl

- 搞定。得到 `lrv-rust-bl.bin` 二进制文件。
- 一些常用的工具，文档里没写，默认读者已经安装过了。包括：
  - 跑通`rCore-Tutorial-v3`所需要的工具
  - `objdump`
  - `gcc-riscv64-unknown-elf`
  - `proxychains`不是必须的，但是挺好用

## rCore-N on FPGA

先将`rCore-N`在`qemu`上跑通，再尝试生成`bin`文件，就不会报错。成功得到 `rcore-n.bin` 二进制文件。

## 粗糙的构想

```
1 编写硬件代码并上传---编译--->二进制文件---烧录----->
2                                     FPGA<---调试--->用户
3 编写软件代码并上传---编译--->二进制文件---运行于--->
```


一键编译还是一步一步编译？

详细的构想：

<https://shimo.im/docs/TwHhy6VwpWYg3crQ>

## 简单的编译，组成原理，qemu知识

内核第一条指令（原理篇） — `rCore-Tutorial-Book-v3 3.6.0-alpha.1` 文档

 <https://rcore-os.github.io/rCore-Tutorial-Book-v3/chapter1/3first-instruction-in-kernel1.html>

关于编译的部分写得比较空泛，去找个具体的例子看看就能看懂了。



