操作系统课程设计中期报告：URVirt

dram

项目简介

项目简介

- 在 U-mode 运行的一个 trap-and-emulate hypervisor
- 目标：可以在 Unmatched 平台 (SiFive Freedom U740) 上的 Linux 中，作为用户程序运行 rCore
- rCore 的源码尽量少修改

项目目标

- 使用相同的 kernel binary 文件，可以在 QEMU 上运行，也可以用 URVirt 在用户态虚拟运行
- 绝大部分指令直接在 CPU 上执行，保证效率
- 最终可以正常运行 rCore 的基础功能

相关工作

RVirt

https://github.com/mit-pdos/RVirt

- 在 S-mode 运行的 trap-and-emulate hypervisor
- 本项目的名字 URVirt 即是受 RVirt 的名字启发

RVirt

https://github.com/mit-pdos/RVirt

- 在 S-mode 运行的 trap-and-emulate hypervisor
- 本项目的名字 URVirt 即是受 RVirt 的名字启发

主要区别：

- RVirt 在 S-mode 运行，URVirt 在 U-mode 运行
- RVirt 的底层功能使用 RISC-V 的 S-mode 的 CPU 功能；URVirt 使用 Linux 的提供的功能（系统调用）。

QEMU

https://www.qemu.org/

QEMU 是一个知名的支持指令集模拟的虚拟机

- QEMU 在用户态运行的时候，也有一定的在用户态模拟特权指令的功能

QEMU

https://www.qemu.org/

QEMU 是一个知名的支持指令集模拟的虚拟机

- QEMU 在用户态运行的时候，也有一定的在用户态模拟特权指令的功能

主要区别：

- QEMU 进行指令集模拟，暂时不能在 RISC-V 的平台上运行；URVirt 不进行非特权指令的模拟，只能在 RISC-V 平台运行
- QEMU 支持 M-mode 的模拟；URVirt 只能模拟 S-mode 以上

目前进度

目前进度概述

- 实现了使用 Linux 的虚拟内存功能模拟 “物理” 内存
- 实现了内核加载到指定 “物理” 地址
- 实现了 SBI call 的模拟
- 实现了基本的特权级（暂时未实现指令）

SBI call 演示

```
static inline uintptr_t sbi_call(
  size_t which, uintptr_t arg0, uintptr_t arg1, uintptr_t arg2) {
    register uintptr_t a0 asm ("a0") = (uintptr_t)(arg0);
    register uintptr_t a1 asm ("a1") = (uintptr_t)(arg1);
    register uintptr_t a2 asm ("a2") = (uintptr_t)(arg2);
    register uintptr_t a7 asm ("a7") = (uintptr_t)(which);
    asm volatile ("ecall" : "+r" (a0)
                          : "r" (a1), "r" (a2), "r" (a7)
                          : "memory");
    return a0;
}


static inline void sbi_console_putchar(size_t c) {
    sbi_call(SBI_CONSOLE_PUTCHAR, c, 0, 0);
}
```

SBI call 演示

```
const char str[] = "I wrote this with SBI calls\n";

void kernel_main() {
    clear_bss();
    for (const char *p = str; *p; p ++) {
        sbi_console_putchar(*p);
    }
    sbi_shutdown();
}
```

SBI call 演示


```
[root@fedora-riscv src]# urvirt-loader/urvirt-loader urvirt-stub/urvirt-stub.bin test-kernel/kernel.bin
[urvirt] Entering urvirt-stub
[urvirt] Jumping to kernel ...
I wrote this with SBI calls
[urvirt] SBI Shutdown!
[root@fedora-riscv src]#
```

图 1: 在 URVirt 中运行的使用 SBI call 的 S-mode 内核程序

内部实现

启动流程

- `memfd_create` 创建 “物理” 内存对应的 fd
- 打开内核二进制文件
- `mmap` 将 stub 载入内存，并开辟一个栈空间
- 跳转到 stub 的起始地址 entrypoint
- 准备地址空间
  - stub 将除了自己的代码以及栈空间以外的地址空间全部 `munmap`
  - 切换到新的栈空间
  - 将物理内存的 memfd 给 `mmap` 到期望的内存地址处

启动流程

- 加载内核
  - 将内核复制到期望的起始地址
- 注册虚拟内存 / SBI 调用 / 特权指令模拟相关：
  - 注册 signal handler
  - 注册 `sigaltstack` 为刚刚分配的栈（因为 entrypoint 执行完毕之后就不再有用）
  - 注册 seccomp filter
- 跳转到内核起始地址开始执行

Seccomp BPF

- seccomp 是 Linux 内核中的一个用户程序系统调用过滤机制，可以限制或捕获一个用户程序所能使用的系统调用
- seccomp 可以加载一个 BPF 程序进行更详细的过滤，比如分析参数和地址

Seccomp BPF

- seccomp 是 Linux 内核中的一个用户程序系统调用过滤机制，可以限制或捕获一个用户程序所能使用的系统调用
- seccomp 可以加载一个 BPF 程序进行更详细的过滤，比如分析参数和地址
- 使用一个 BPF 程序过滤系统调用，使得只有指令的地址在 stub 的范围内的系统调用才被允许，在范围外的产生 SIGSYS

Seccomp 踩坑

- seccomp(2) 的 SECCOMP_SET_MODE_FILTER 用的是 BPF 而不是 eBPF
- BPF 没有 64 位数的操作指令
- QEMU user 并不支持 seccomp 的 BPF 模式

Signal handler

各种需要模拟的情况，都通过 Linux 的信号来完成：

- 在程序执行特权指令的时候，会发生非法指令异常，在 Linux 中表现为一个 SIGILL 信号
- 当访问非法地址的时候，会发生 page fault，在 Linux 中表现为一个 SIGSEGV 信号
- 使用 seccomp 捕获 ecall 指令，会产生 SIGSYS 信号

Signal handler

各种需要模拟的情况，都通过 Linux 的信号来完成：

- 在程序执行特权指令的时候，会发生非法指令异常，在 Linux 中表现为一个 SIGILL 信号
- 当访问非法地址的时候，会发生 page fault，在 Linux 中表现为一个 SIGSEGV 信号
- 使用 seccomp 捕获 ecall 指令，会产生 SIGSYS 信号
- 使用 SA_SIGINFO flag 可以获得扩展的 handler 信息

```
void handler(int sig, siginfo_t *info, ucontext_t *ucontext);
```

在 ucontext 参数中，可以获得发生异常处的寄存器信息，对用户态状态进行修改，实现模拟

Signal handler 相关踩坑

- SECCOMP_RET_TRAP 产生的 SIGSYS，传给 handler 的 ucontext_t 里的寄存器 a7 好像并不是系统调用编号，需要在 siginfo_t 的 si_syscall field 里找到。
- SECCOMP_RET_TRAP 产生的 SIGSYS，pc 已经是 ecall/c.ecall 指令后面的指令
- 把 vdso 给 munmap(2) 之后 signal handler 返回的时候会 SIGSEGV，因为内核执行 signal handler 的时候将 ra 设置为 vdso 里的 __vdso_rt_sigreturn 函数的地址，也就是恢复栈指针之后要调用系统调用 rt_sigreturn 返回到信号发生的地方，所以我们可以复刻一份这个函数然后绕过 vdso 里面的那份（参见 https://elixir.bootlin.com/linux/latest/source/arch/riscv/kernel/vdso/rt_sigreturn.S#L10）

```
__vdso_rt_sigreturn:
    li a7, __NR_rt_sigreturn
    ecall
```

后续安排

后续安排

- RISC-V 指令解码
- 虚拟内存功能实现
- 特权级别实现
- Block device 驱动实现

exit_group(0)