

proj178-kvm-in-rust

清华大学, Kaito, proj178

项目简介

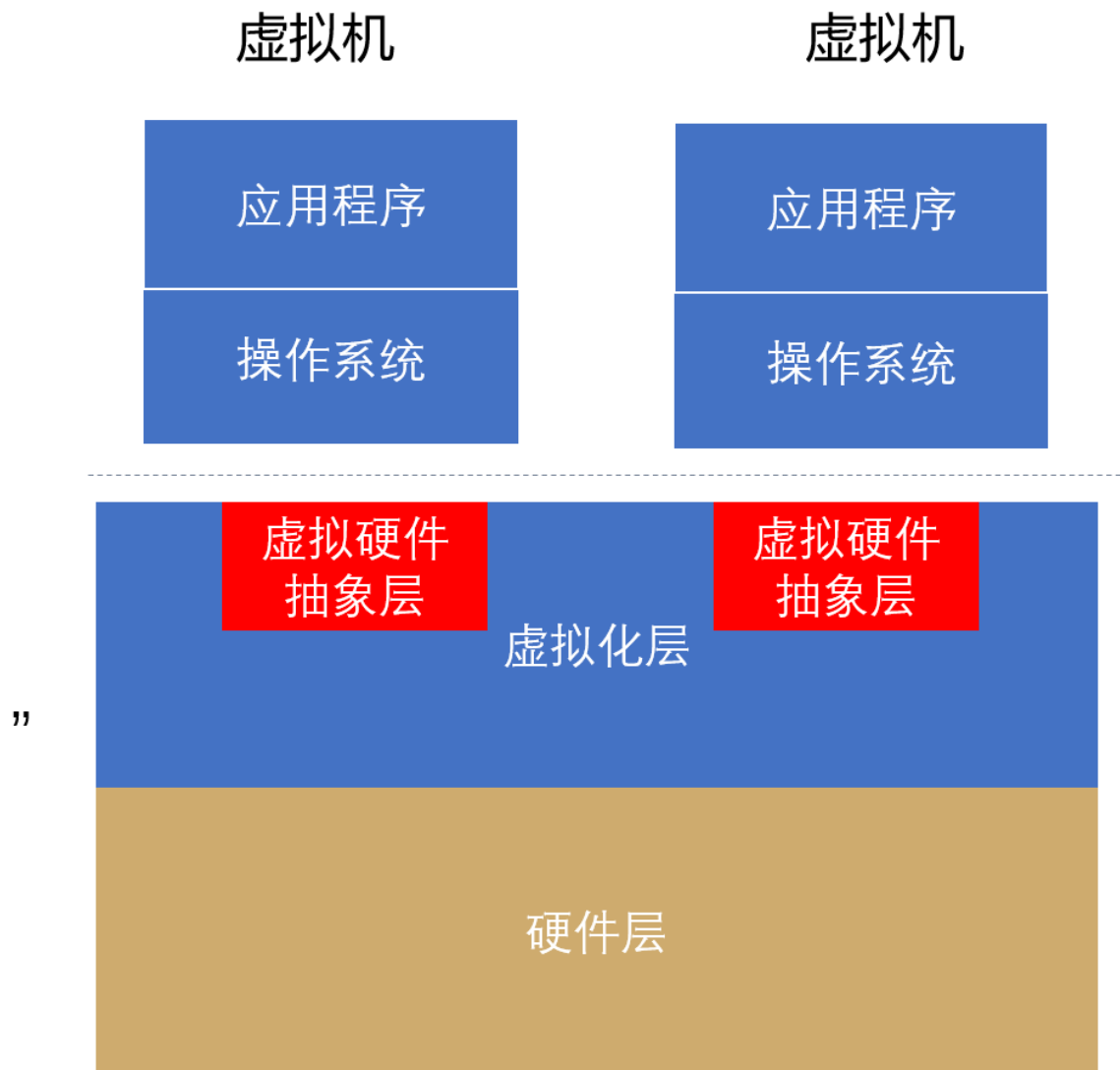
基于内核的虚拟机 Kernel-based Virtual Machine (KVM) 是一种内建于 Linux® 中的开源虚拟化技术。具体而言, KVM 可帮助您将 Linux 转变为虚拟机监控程序, 使主机计算机能够运行多个隔离的虚拟环境, 即虚拟客户机或虚拟机 (VM)。Linux 2.6.20 或更新版本包括 KVM。KVM 于 2006 年首次公布, 并在一年后合并到主流 Linux 内核版本中。

本项目以Linux Kernel中的KVM为基础, 拟采用Rust语言重新设计与实现它。KVM 是 Linux Kernel 的一部分。Linux 2.6.20 或更新版本包括 KVM。KVM 于 2006 年首次公布, 并在一年后合并到主流 Linux 内核版本中。由于 KVM 属于现有的 Linux 代码, 因此它能立即享受每一项新的 Linux 功能、修复和发展, 无需进行额外工程。Rust语言与C语言不同, 其作为一门现代的系统编程语言, 表达能力更强, 语义更丰富, 所以在用Rust语言参与本项目时, 从某种角度上需要我们将一些规则和设计显式地"写"在代码中, 这同时也促使了我们去思考。

项目前期调研分析

硬件虚拟化

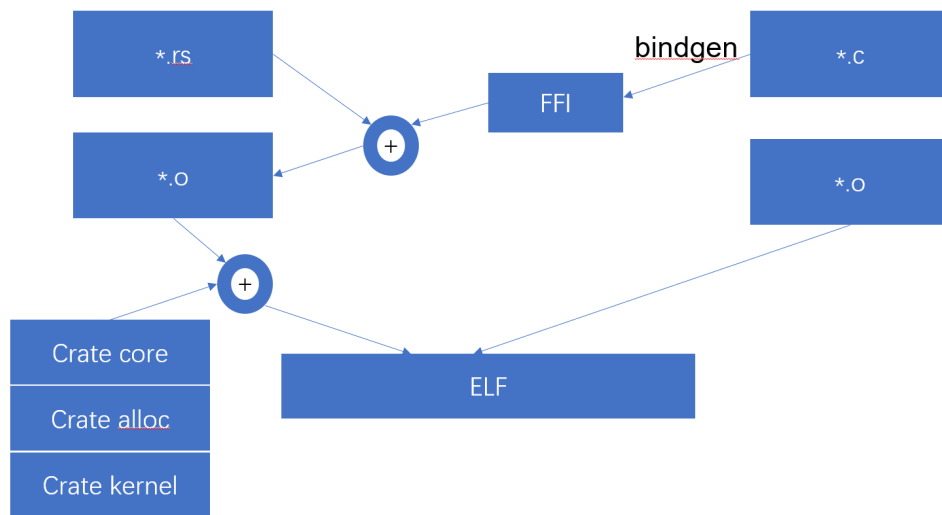
硬件虚拟化方面, 我参考学习了RVM项目的文档, 对内存、CPU虚拟化的架构进行了学习, 同时也学习了硬件虚拟化的流程。另一方面, 通过研究Linux kernel KVM module的源代码和文档了解了KVM的运行流程。KVM通过创建文件的方式, 以ioctl的方式来作为一个dev与应用(QEMU)进行通信, 因此实现的核心, 就是相应的初始化函数以及KVM规范下的ioctlAPI。另一方面, 为了面向x86架构, 研究Intel白皮书中的VMX相关内容, 在开发过程中也经常使用到。



Rust For Linux

另一方面，对于使用rust语言来实现Linux内核的支持，参考了github上[Rust-for-Linux](#)团队所给出的支持。通过使用他们封装的Rust语言支持，可以很好的实现Rust语言实现的module与kernel的联合编译，同时也让Rust语言的开发过程中可以使用Linux内核中实现的接口和结构体，可以极大方便开发过程。

Rust For Linux中将Rust代码写入Linux内核的开发方式如下：



Rust实现module的价值

关于使用Rust重写Linux kernel module的价值，阅读了华盛顿大学的论文：[Practical Safe Linux Kernel Extensibility](#)。文中指出，rust语言的内存管理方式可以帮助Linux内核解决约93%的UB类型漏洞。因此，本项目对于Linux的设计也具有不错的价值，同时也意味着在开发过程中应当尽量避免使用rust中的unsafe关键词，来保持rust语言的特性。

代码设计

作为一个rust语言重写的linux内核模块，首先应当保证对外接口与linux原版的模块一致，因此模块对外暴露的接口是以linux IOCTL的方式提供的。

rust_kvm

核心的数据结构是rust_kvm，对应linux下的kvm结构体，基于Rust For Linux提供的支持，实现了作为内核模块所需的一些操作（包括FileOperation、MiscDevice、Drop），并且在 `IoctlHandler` 中实现了主要的IOCTL API：

- `IOCTL_KVM_CREATE_VM`：用于创建分配一段虚拟内存
- `IOCTL_KVM_CREATE_VCPU`：用于创建一个虚拟处理器用来运行Guest程序
- `IOCTL_KVM_VCPU_RUN`：用于运行Vcpu，运行其中的Guest程序
- `IOCTL_KVM_SET_USER_MEMORY_REGION`：用于将用户内存分配给Guest程序，并建立hpa到gpa的映射
- `IOCTL_KVM_SET/GET_REGS`、`IOCTL_KVM_SET/GET_SREGS`：用于设置以及获取当前Vcpu的寄存器状态，由于目前实现只支持单个VCPU、单个Guest程序，因此不需要指定具体CPU和Guest

Vcpu

Vcpu部分的代码，实现了KVM中所运行的Vcpu的状态描述以及所需要的操作行为。Vcpu结构体的设计如下：

```
pub(crate) struct Vcpu {
    pub(crate) guest: Ref<GuestWrapper>,
    pub(crate) guest_state: Pin<Box<GuestState>>,
    pub(crate) host_state: Pin<Box<HostState>>,
    // DefMut trait for UniqueRef, List use it
    pub(crate) mmu: UniqueRef<RkvmMmu>,
    pub(crate) run: RkvmPage,
    pub(crate) vmcs: RkvmPage,
    pub(crate) vcpu_id: u32,
    pub(crate) launched: bool,
}
```

结构体的成员变量基本与C语言的源代码一致，主要是针对于单Vcpu进行了一定程度的简化。Vcpu相比常规的cpu属性之外，还需要保存宿主机的状态 `host_state` 以及用户程序（应用程序或GuestOS），以保证KVM对于Vcpu的状态管理。对于state的设计采用了Pin的封装方式，保证内存位置的固定性，防止在对于Vcpu的操作中错误地改变用户程序的状态，保证了代码的安全性。

对Vcpu进行封装后实现了以下API：

- `init`：用于对于Vcpu的状态以及存储进行初始化，同时将宿主机的RSP寄存器设置为用户程序的状态
- `get_run`：用于获取Vcpu运行的虚拟地址
- `vcpu_exit_handler`：用于处理Vcpu的各种退出情况和状态，并将推出原因转发给硬件的VMX以及KVM的主函数
- `vcpu_run`：用于运行装载在Vcpu上的用户程序，同时跟踪运行状态、开关中断，返回错误信息
- `set_regs/get_regs`，`set_sregs/get_sregs`：用于设置以及获取Vcpu的寄存器状态，

Guest

Guest是KVM虚拟机上所运行的用户程序，设计如下：

```
pub(crate) struct Guest {
    pub(crate) mm: *const bindings::mm_struct,
    pub(crate) memslot: RkvmMemorySlot,
    pub(crate) nr_slot_pages: u64,
}
```

对于用户程序本身的状态维护由Vcpu进行，因此在Guest部分的代码主要是针对用户程序所需要的内存空间的分配以及调度。在KVM中，实现的方式是将内存单元切分为基本的内存槽。在分配一块新的内存空间时，将相应的物理内存地址进行映射后写入结构体中。其中，关于系统具体将哪一内存分配给谁，是通过直接调用Linux的接口的方式来进行的。通过Rust For Linux提供的bindgen来将C语言实现的include变为Rust语言可用的 `bindings crate` 来进行使用，设置相应的 `mm_struct`。

Guest程序的内存映射方式采用的是EPT扩展页表的形式。根据Intel VT-X的标准实现了一个4级页表，存储在 `mmu.rs` 之中。程序的退出（对应KVM中的VM_EXIT）则是实现在了 `exit.rs`，包含不同异常的处理以及退出方式。

MMU

mmu的设计在本项目中是一个难点。在目前的设计中，R-KVM采用了扩展页表EPT的方式来进行实现，通过设置VMX中的EPT相关寄存器来实现gpa、hpa、vpa之间的相互映射。具体的数据结构设计如下：

```
pub(crate) struct RkvmMmu {
    pub(crate) root_hpa: u64,
    pub(crate) root_mmu_page: Ref<RkvmMmuPage>,
    mmu_root_list: List<Ref<RkvmMmuPage>>,
    mmu_pages_list: List<Ref<RkvmMmuPage>>,
    pub(crate) spte_flags: Ref<EptMasks>,
}
```

mmu节点以及页表的组织采用了自定义链表的方式来进行，通过调用Linux内核的接口来将宿主机的内存页进行映射和分配，组成用户程序所使用的内存单元。使用链表的优势在于，本项目在设计时面对的是X86_64架构的硬件，因此一般常见的GuestOS都需要对于4级页表有着充分的支持。如果不使用链表，由于Rust语言对于内存安全性的要求，对于每一级页表项，都需要实现复杂的内存锁获取释放以及拷贝的过程，否则就会引入大量的unsafe代码。链表的使用让多级页表的处理相对更加简单、安全，同时也保证了代码对于更多页表支持的可扩展性，方便后续进一步的开发工作。

运行流程

和常规的KVM一样，R-KVM采用了IOCTL的方式来给其余应用程序提供硬件虚拟化的服务。以test_misc.c为例：

```
int fd, ret;
struct kvm_sregs sregs;
struct kvm_run * run;
unsigned long mem = 0;
char buf[4096];
fd = open("/dev/rust_kvm", O_RDWR);
if (fd < 0) {
    printf(" open fd failed \n");
    exit(1);
}
printf(" open success \n");

ret = ioctl(fd, KVM_CREATE_VM, 0);
if (ret < 0) {
    printf(" ioctl failed \n");
    exit(1);
}
```

将作为内核模块安装好的rust_kvm设备打开，之后通过ioctl来与rust_kvm设备进行交互，使用相应的服务。首先需要创建一段虚拟内存，分配内存进行映射。在此之后，根据需要将希望放在虚拟机上运行的用户程序存入分配好的虚拟内存中。之后，创建一个Vcpu并运行这一段内存，根据需求进行动态的虚拟内存分配。

决赛开发日志

2022.6.25：整合了部分代码，进行了一些警告信息的移除，阅读了项目导师所补充的新内容。

2022.7.1：本周继续添加了一些rkvm的exit reason，包括用户退出的原因捕捉以及读写内存异常。同时，将此前遗留的裸指针、变量可变性等警告信息进行了完善，修复了之前遗留的一个mmu根节点创建问题，更新了页的定义。

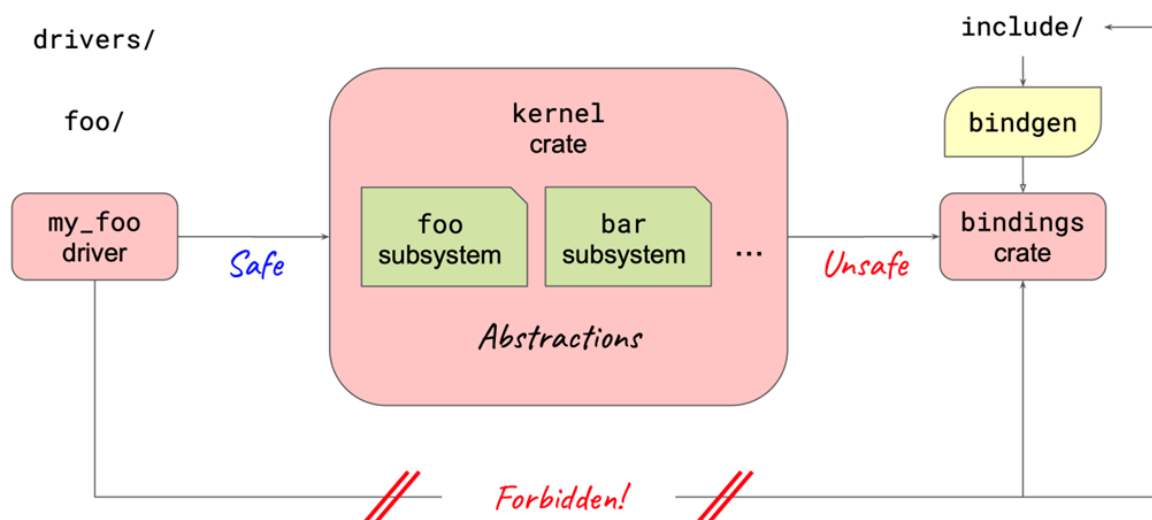
2022.7.8：确定进一步测试的目标OS为xv6，对该操作系统的支持平台以及相应的中断控制硬件进行了分析，并研究了进一步运行xv6所需要实现的任务。xv6采用apic的中断控制，需要对此进行模拟，同时，面对ucore的四级页表结构，也需要进一步完善mmu的设计。

2022.7.22: 基本完成mmu四级页表的设计, 开始设计apic中断控制的相关代码。

2022.7.29: 使用kvmtool运行了xv6-x86_64版本, 调试了用户程序保证正确性。

2022.8.6: 将GuestOS替换为rCore-Tutorial-x86_64版本, 调试了用户程序保证正确性。

项目成果及创新性



目前, 本项目已经在依赖kvmtool这一工具的条件下, 完成了对于典型的简单C语言操作系统 (xv6) 以及Rust语言操作系统 (rCore-Tutorial) 的运行支持, 并提供了一个简单的用户程序测例作为对IOCTL API正确性的检验。

代码链接: <https://github.com/KaitoD/linux>

安全性

本项目作为用Rust语言所设计的Linux内核模块, 其核心竞争力主要也是Rust语言的安全性上。如上图所示, 在将Rust代码写入Linux内核的过程中, 对于unsafe的使用是有着较强的约束性的, 对于需要保证代码所暴露出的接口必须通过safe的方式, 转化为一个 `kernel crate` 的抽象才能够接入到Linux的内核当中。

在本项目中, 3242行核心代码中包含有67处unsafe的使用, 可以分成以下两类:

- 由于需要引入C语言版代码的接口而加入的 (或者为了使用汇编代码所加入的), 这一部分由于安全性并非rust来确保, 因此不会给内核引入更多UB漏洞。这一类unsafe共有14处

```
let spt = unsafe { Some(bindings::page_address(page.pages) as u64) };
```

- 其余由于结构体构造、指针拷贝在代码设计中没能避免或暂时没有更阿红解决方案所引入的unsafe代码。这一部分unsafe代码共有55处, 相比代码整体规模, 这些unsafe代码的引入从整体安全性上来看是可以接受的

可扩展性

本项目的代码设计主要是针对了单个Vcpu (即不支持多核)、单个Guest程序 (即不支持嵌套虚拟化以及在虚拟机上同时运行多个OS)、x86_64架构的条件下所开发的, 因此在一些常量的设置上相比完整的KVM要更简略。但是在一些核心结构体的设计上, 由于链表等数据结构的引入和使用, 使得在进一步开发中不需要对已有的代码 (尤其是核心数据结构) 上进行大规模的重构, 保证了代码的可扩展性。

后续开发方向

- qemu的兼容：由于qemu对于kvm所依赖的IOCTL要更为丰富，因此目前本项目仅支持kvmtool作为应用程序作为用户级的衔接。后续将对接口进行丰富化以兼容功能更为强大的qemu。
- 多核、多GuestOS的支持：增加对于多个Vcpu同时运行的支持，来对多核OS提供支持，同时也可以支持一台虚拟机上并行运作多个OS。