# 五一假期大作业动员

## 目前 rvisor-kernel 的开发状况

- 内核内部
  - 实现了简单的系统调用替换。
  - 实现了内核模块与用户通过 ioctl 交互。
- 内核外部
  - 抽取了部分 runc 的代码，能够与 containerd 做简单的交互。

## 基本原理

### C 与 rust 的交互

最基本的交互方法如下：

```
extern "C" fn a_fn_call_by_c(str: *const u8) {
    ...
}

extern "C" {
    fn c_fn_called_by_rust(...);
}
```

但是第二个参数一多就会比较麻烦，所以我们可以用bindgen来自动生成

在 linux_kernel_module::binding 中可以直接使用linux函数

```
linux_kernel_module::binding::set_fs()
```

### 系统调用替换

内核API：

```
syscall_table = (void **) kallsyms_lookup_name("sys_call_table"); // 获取系统调用表
```

```
    cr0 = disable_wp(); // 关闭内存的写保护
    syscall_table[syscall_num] = syscall_fn; // 替换系统调用函数
    restore_wp(cr0);
```

自己编写的rustAPI：

| | |
|---|---|
| cleanup | 退出的时候调用 |
| init | init的时候调用 |
| replace_clear ⚠ | recover the replace |
| replace_init ⚠ | init syscall replacer (find where the syscall is) |
| replace_syscall ⚠ | replace the syscall (here we replace open for test) |
| safe_replace_syscall | replace_syscall 的安全包装 |

（不安全的其实都是私有的）

init中会调用多次safe_replace_syscall

## ioctl 交互

### 如何使用

rvisor 本身作为 linux系统的一个设备，提供一个系统设备号。安装了rvisor时候可以通过 `cat /proc/devices | grep rvisor` 来查询主设备号。

首先 mknod 一个设备：

```
1    mknod --mode=a=rw /dev/rvisor c <主设备号> 0
```

然后在程序中使用ioctl访问该设备，即可：

```
1        int fd = open("/dev/rvisor", O_RDWR);
2        int i = ioctl(fd, <command>, <arg>);
```

rvisor在进一步实现的时候会使用这个功能提供用户空间对内核的控制。

### 原理与实现

首先，我在linux_module_rust中专门编写了一个模块来处理ioctl的文件

实现图中这两个trait就相当于实现一个ioctl文件

**Traits**

| | |
|---|---|
| FileOperations | `FileOperations` corresponds to the kernel's `struct file_operations`. You implement this trait whenever you'd create a `struct file_operations`, and also an additional trait for each function pointer in the `struct file_operations`. File descriptors may be used from multiple threads (or processes) concurrently, so your type must be `Sync`. |
| Ioctl | |

在这里实现了rvisor的输入输出文件：

**Struct rvisor_kernel::iodev::IoDeviceFile** [-][src]

[+] Show declaration

[-]输入输出文件，

**Trait Implementations**

[+] `impl FileOperations for IoDeviceFile` [src]

[+] `impl Ioctl for IoDeviceFile` [src]

然后将相关的设备做一个定义：

```
        // 登记设备名，和设备文件struct
        let _chrdev_registration =
                chrdev::builder(cstr!("rvisor"), 0..1)?
                    .register_device::<iodev::IoDeviceFile>()
                    .build()?;
```

具体细节参考项目文档。

## 与containerd的交互

### go 语言

由于go语言那大道至简的特性，还是要简单说一下go工程的布置方法。

对于使用惯rust的大家来说，总是随便就 `cargo new` 一个工程。这在我们的大道至简语言中是太过复杂的。大道至简语言要求我们，所有的go代码都要放在 `$GOPATH` 下。通常用 `~/go` 作为 GOPATH。

```
    $GOPATH
      ├─<src>
          ├─<libs>
              |--<event>
                  hh.go
          ├─<demo>
              ├─demo.go
              ├─demo_hh.go
              ├─demo_test.go
      ├─<bin>#可执行文件目录
      ├─<pkg> # 包将被安装到此处
```

所有项目都在$GOPATH/src下找个文件夹放就可以了，文件夹下想要用别的库，创建一个vendor然后放在下面即可。

### containerd的安装和shim运行时的编译

```
  sudo apt install containerd
```

然后在 `/containerd-shim-rvisor-v1` 目录下 `go build` ,得到 `containerd-shim-rvisor-v1*` 可执行文件，然后将可执行文件放到 `/usr/bin` 下，运行时就安装好了。

然后配置containerd，向 `/etc/containerd/config.toml` 中加入：

```
  disabled_plugins = ["restart"]
  [plugins.cri.containerd.runtimes.rvisor]
      runtime_type = "io.containerd.rvisor.v1"
```

然后重启containerd,并随便找一个镜像运行就行了。

```
  sudo systemctl restart containerd
  sudo ctr run --rm --runtime io.containerd.rvisor.v1
  docker.io/denverdino/hellowasm:latest sad
```

然后containerd就会与rvisor-shim的go代码做交互（采用gRPC，大家有必要了解一下），我们在go代码中设置容器就行了。（顺带一提：这里的代码是contaienrd-shim的模版代码，需要在上面对一些特定的功能进行修改。）

# 下一步

五一假期的任务：

- 实现chroot的功能，以简单但不安全的方式实现容器功能。
- 基本完成runrsc（python），提供一个客户端对系统做出控制
- 基本完成rvisor-shim（go），可以运行一个进程的程序。

## 内核提供的接口（暂定）

```
ioctl(fd, RVISOR_CREATE, const char * rootfs) = errno
ioctl(fd, RVISOR_ADD_PROC, int pid) = errno
ioctl(fd, RVISOR_KILL, int pid) = errno
```

## 运行时 runrsc

运行时提供：

```
runrsc create root_path -> 将模块加载如内核，并完成输入输出的配置（insmod + mknod + ioctl(CREATE)）
runrsc exec guest_path -> 启动一个容器内进程
runrsc ps -> 输出所有运行的进程
runrsc shutdown -> 结束所有进程
```

## Shim

shim能够：

- 以chroot方式运行 FROM scratch 的镜像