Kernel

1 构建基本的内核容器系统

1.1 搭建 Rust + Linux 内核模块开发环境

- 借助 fjw 助教提供的 VLab KVM 虚拟机远程共同开发。
- 使用 Github 项目 linux-kernel-module-rust 进行 LKM + Rust 开发。
- 借助 `vscode + rust-analyser` 搭建开发环境。
- 使用 `bindgen` 生成 C 和 Rust 的调用接口。

1.2 系统调用的替换

使用 `kallsyms_lookup_name` 得到系统调用表后，编写一个简单的 C-Shim, 实现对系统调用的替换。

由于 Linux 的内存保护机制，需要暂时禁止掉内存的写保护。

同时，还要暂时关闭内核抢占 `preempt_disable()`。

```
int replace_syscall(unsigned int syscall_num, long (*syscall_fn)(void)) {
    .......

    cr0 = disable_wp(); // 关闭内存写保护，同时关闭内核抢占。
    syscall_table[syscall_num] = syscall_fn; // 替换相应的系统调用
    restore_wp(cr0); // 恢复内存写保护，开启内核抢占。

    return 0;
}
```

1.3 系统调用的使用

- 在执行系统调用时，虽然我们是在内核态，但是我们的线程没有改变，我们的线程仍然是用户线程，故可以直接在内核调用系统调用，产生与用户态完全相同的效果。
- 虽然 Linux 的具体实现中，内核空间和用户空间的线性地址是共享的，但是用户传入的指针在内核中还是不能直接使用。（其他的系统
- 内核无法信任用户的指针，因而要对传入的指针进行检查，可以使用 `__copy_from_user` 和 `__copy_to_user` 宏来对用户空间进行操作。同时为了避免出现bug，不允许内核直接访问用户态数据。

- 所以我们在系统调用的时候，需要先使用 `__copy_from_user` 来检查这个指针是否是来自用户态的，然后就可以对数据进行修改了，但是系统调用内部还会对我们传入的指针做一次多余的检查，因此，可以通过修改 fs 寄存器来关闭这一个检查机制。

```
#[inline]
pub unsafe fn sys_read_kern(fd : i32, buf: * const u8, count: isize) -> usize {
    let old_fs = get_fs();
    set_fs(KERNEL_DS);
    let ret = read(fd, buf, count);
    set_fs(old_fs);
    ret
}

#[inline]
pub fn read_kern(fd : i32, buf: &[u8]) -> usize {
    unsafe {
        sys_read_kern(fd, buf.as_ptr(), buf.len())
    }
}
```

1.4 实现安全的用户态指针

用户态切片，使用 `PhantomData` 来区分三种策略 `In`，`Out`，`InOut`。同时定义了 `Read`、`Write` 两种 trait。

- `UserInSlice<T> = UserSlicePtr<T,In>` 实现了 `Read` trait。
- `UserOutSlice<T> = UserSlicePtr<T,Out>` 实现了 `Write` trait。
- `UserInOutSlice<T> = UserSlicePtr<T,InOut>` 同时实现了 `Read` trait 和 `Write` trait。

```
// PhantomData: Zero-sized type used to mark things that "act like" they own a T.
pub struct UserSlicePtr<T, P : Policy>(*mut T, usize, PhantomData<P>);

impl<T, P: Policy> UserSlicePtr<T, P> {
    pub unsafe fn new( ptr: *mut T, length: usize) -> Self{
        Self(ptr, length, PhantomData)
    }
}

pub trait Policy {}
pub trait Read: Policy {}
pub trait Write: Policy {}

pub enum In {}
pub enum Out {}
pub enum InOut {}

impl Policy for In {}
impl Policy for Out {}
```

```
impl Policy for InOut {}

impl Read for In {}
impl Write for Out {}
impl Read for InOut {}
impl Write for InOut {}

pub type UserInSlice<T> = UserSlicePtr<T, In>;
pub type UserOutSlice<T> = UserSlicePtr<T, Out>;
pub type UserInOutSlice<T> = UserSlicePtr<T, InOut>;

impl<T, P: Read> UserSlicePtr<T, P> {
    pub fn read(&self) -> Result<&[T]> {
        // __copy_from_user
        ....
    }
}
.....
```

1.5 实现内核与用户的交互

实现内核与用户的交互的其他方式：系统调用方式（`seccomp`）、文件系统方式（`cgroup`）

在系统中添加一个虚拟设备节点类型 `rvisor`，使用 `mknod` 创建节点后，可以通过 `ioctl` 系统调用进行交互。

命令列表如下

- `create` 新建一个容器环境，并返回容器 id。
- `addself` 为特定 id 的容器，增加调用进程
- `remove` 删除特定 id 的容器

```
    /// 对用户空间的iotcl调用做出反应
    /// * create 命令新建一个容器环境
    /// * addproc 增加一个进程
    /// * remove 删除一个进程
    fn ioctl(&self, cmd:u32, arg: u64) -> KernelResult<i64> {
        info!("ioctl cmd={} arg={}", cmd, arg);
        let mut container = Container::get_container();
        let cmd = IoctlCmd::try_from(cmd)?;
        match cmd {
            IoctlCmd::Create => {
                ......
            }
            IoctlCmd::AddSelf => {
                ......
            }
            IoctlCmd::Remove => {
                ......
            }
```

```
        }
    }
```

1.6 在 Rust 中实现内核同步

Linux 内核中使用一种自旋锁：`spinlock_t` 来完成基本的同步功能，自旋锁就在内容被其他线程占用的时候忙等，直到资源释放。

使用通常的信号量往往会带来上下文切换的开销，所以在内核的开发中，我们经常使用自旋锁。

Rust 提供了 `spin` 这个 crate，便于我们在 `no_std` 的环境下使用自旋锁。

```
use spin::{
    Mutex,  // 使用方法与 std::Mutex 相同
    RwLock, // 使用方法与 std::RwLock 相同
};
```

1.7 使用Rust实现简易 chroot 功能

有了上面的方法，我们就可以在内核安全地编写容器了。劫持一些像 `openat`、 `fork` 等基本的系统调用，就可以实现简易的容器了。

```
// 一旦有了内核同步，就可以使用安全的静态可变变量了。
lazy_static!{
    static ref CONTAINER : Mutex<BTreeMap<i64, Task>> = {
        Mutex::new(BTreeMap::new())
    };
}
pub fn get_thread<'a>() -> Option<Task> {...}
pub fn create(s : String) -> KernelResult<usize> {...}
pub fn run(id: usize) -> KernelResult<usize> {...}
pub fn remove(id: usize) -> KernelResult<usize> {...}

pub extern "C" fn openat(a0: u64, a1: *const u8, a2: u32, a3: u32) -> isize {
    ......
}
```

2 为容器系统添加内核

2.1 zCore

由于我们还有容器方面的任务，编写完整的、支持Linux的内核对我们而言时间不是特别够，所以我们选择使用一个现有的内核。

zCore 是清华大学操作系统课程实验的一个项目，由多届助教和学生共同开发、重构、和完善。

zCore 是一个用 Rust 编写的 Zircon 微内核，Zircon 是Google 推出的 Fuchsia OS 的微内核，zCore 的基本设计方法都是参照 Fuchsia 设计的。

清华大学与 zCore 相似的项目还有 rCore（Rust 编写的内核）uCore（C 编写的内核）等。

2.2 Why zCore?

1. zCore 相比用 Rust编写的成熟的微内核操作系统 RedoxOS 更简单，更利于我们学习。
2. rCore 使用的是传统的内核，相比 zCore，不容易实现容器隔离。
3. zCore 的代码可复用性高。
4. 个人原因。

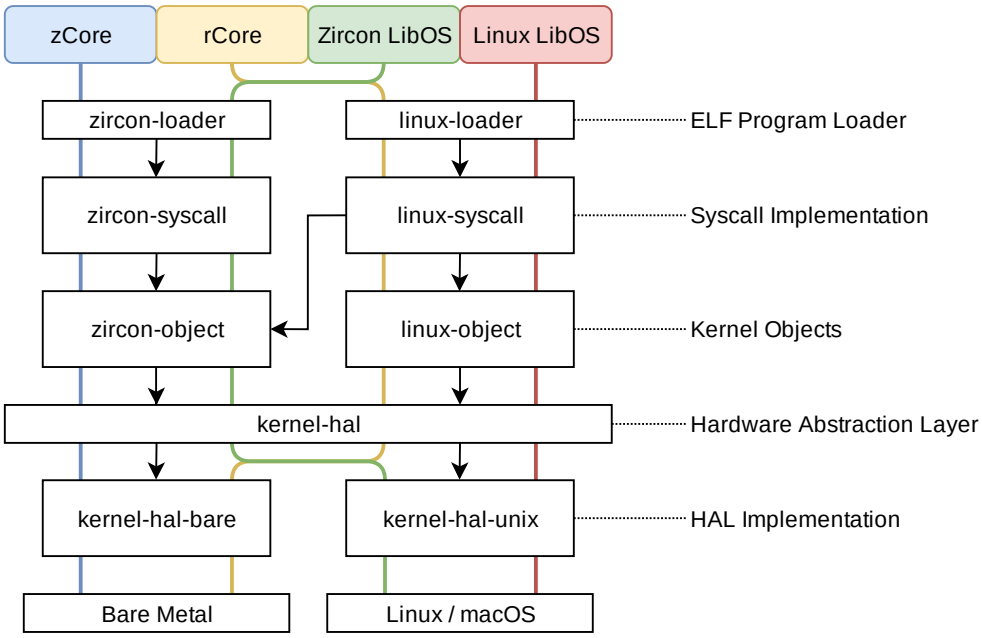
```
/// Linux specific process information.
pub struct LinuxProcess {
    /// The root INode of file system
    root_inode: Arc<dyn INode>,
    /// Parent process
    parent: Weak<Process>,
    /// Inner
    inner: Mutex<LinuxProcessInner>,
}

#[derive(Default)]
struct LinuxProcessInner {
    /// Execute path
    execute_path: String,
    /// Current Working Directory
    ///
    /// Omit leading '/'.
    current_working_directory: String,
    /// Opened files
    files: HashMap<FileDesc, Arc<dyn FileLike>>,
    /// Futexes
    futexes: HashMap<VirtAddr, Arc<Futex>>,
    /// Child processes
    children: HashMap<KoID, Arc<Process>>,
}
```

2.3 zCore 结构

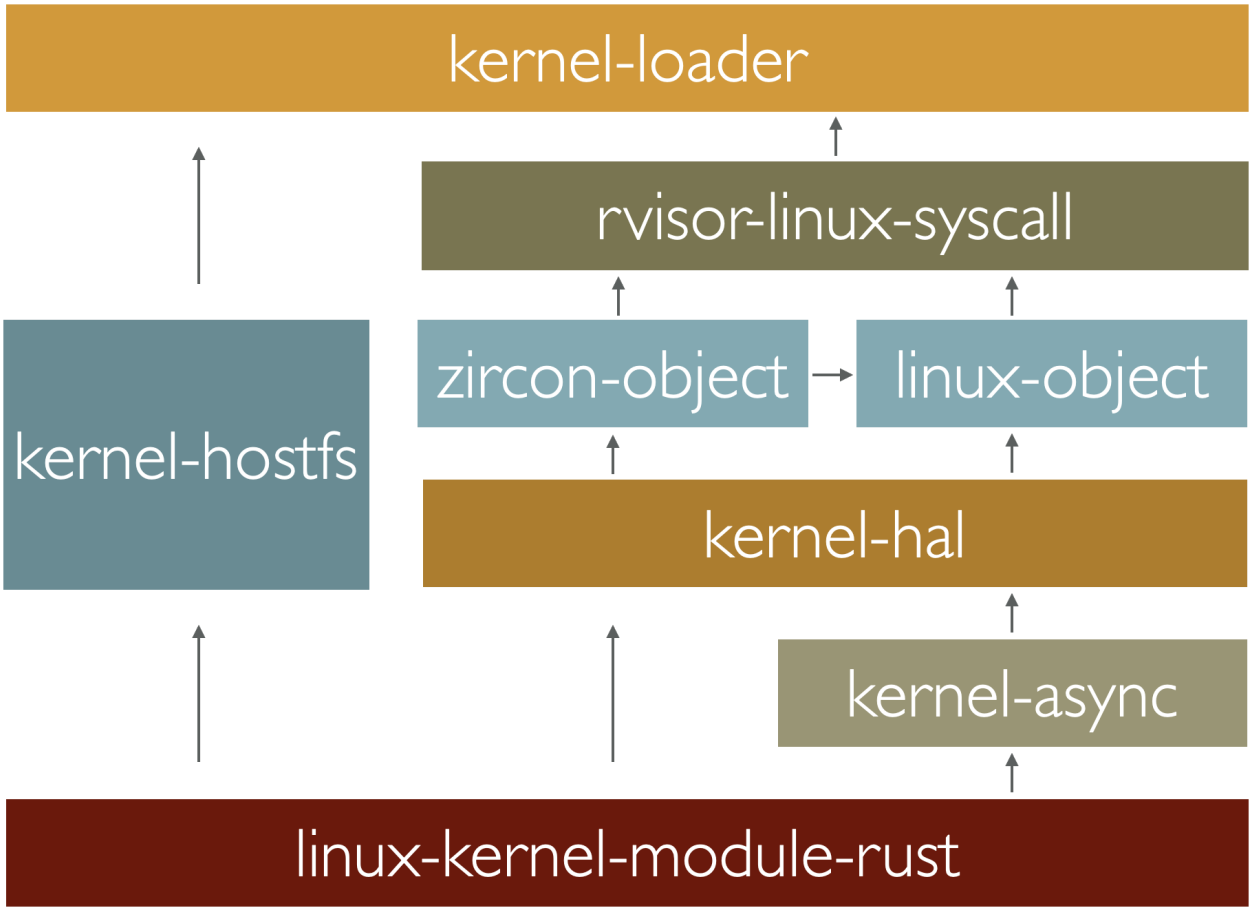
- linux-syscall linux系统调用实现。（部分参考）

- linux-object zCore 对 Linux 支持的部分。（完整使用）
- zircon-object 是 Zircon 内核最核心的部分。（完整使用）
- kernel-hal 由于内核中有一些安全要求，这一部分必须要重写，当然也包括下面HAL的实现。
- 除了这张图之外，还有一些库需要我们自己实现。



2.4 rVisor-zCore 内部架构设计

- `kernel-loader` 内核模块，实现容器架构，系统调用劫持
- `linux-kernel-module` 修改过的原 Github 项目，添加了安全的用户指针，和 logger 等部件。
- `kernel-hostfs` 用来在内核中实现 HostFs（本机文件系统）
- `kernel-async` 在内核中添加对 Rust async 语法的支持。
- `zircon-object` 、 `linux-object` 原 zCore 模块。
- `rvisor-linux-syscall` 所有劫持的 syscall 的实现。



2.5 async/await 在内核的实现

现代语言往往都有支持并发、异步的特殊语法，比如 go语言的 `goroutine` 和 `channel` ，就利用协程（`coroutine`）大大地简化了并发程序的开发。

在 Rust 中，相应地，有 `async/await` 可以实现异步编程。但 Rust 中的 `async/await` 是一个抽象的、0开销的语法。

zCore 的线程使用 `async/await` 实现，因此，必须利用 Rust `async/await` 的抽象特性，建立起Linux 线程与 zCore 线程的对应关系。

```
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    async {
        let x: u8 = foo().await; // 对foo进行等待，这期间线程有可能会去做其他事情
                                 // 直到 foo Ready 了才会返回。
        x + 5
    }
}
```

Future Trait

- async 返回一个 Future Trait 类型。
- Future 包含一个 poll 方法，可以从外部查看当前的状态：正在计算 / 已经完成。并且可以唤醒其他任务。

```
pub enum Poll<T> {
    Ready(T),  // 已经结束
    Pending,   // 正在进行中
}

trait Future {
    type Output;

    /// Attempt to resolve the computation to a final value, registering
    /// the current task for wakeup if the value is not yet available.
    fn poll(
        mut self: Pin<&mut Self>, // Pin 表示地址位置固定
        cx: &mut Context<'_>      // 包含 waker，可以唤醒其他任务
    ) -> Poll<Self::Output>;
}
```

Executor 和 Task

- Executor 完成线程的调度，对各个 Future 的状态进行查询。
- Task 保存当前线程的Future，唤醒状态等。

```
pub struct Task {
    pub future: Mutex<Pin<Box<dyn Future<Output = ()> + Send + 'static>>>,
    state: Mutex<bool>,
}
// task 可以被其他的 task/executor 唤醒
impl Woke for Task {
    fn wake_by_ref(task: &Arc<Self>) {
        task.mark_ready();
    }
}

pub struct Executor {
    tasks: VecDeque<Arc<Task>>, // 线程池
}

impl Executor {
    pub fn spawn(future: impl Future<Output=()>) -> Task {...} // 在线程池生成线程
    pub fn run() {...} // 开始事件循环，反复 poll 各个 future。
}
```

建立 Linux 线程与 zCore 之间的关联

```
/// 提供两种线程，一种是 zCore 内部的线程，一种是 Linux 用户线程。
pub enum TaskInner {
    Kern(Mutex<Pin<Box<dyn Future<Output = ()> + Send + 'static>>>),
    // 用户调用线程，返回 isize。
    UserTop(
        Mutex<Pin<Box<dyn Future<Output = isize> + Send + 'static>>>,
        Arc<LxThread>
    ),
}

pub struct Task {
    pub inner: TaskInner,
    state: Mutex<bool>,
}

/// 每个 Linux 用户线程的事件循环（event loop）。
/// 所有的轻量级 zCore 线程由所有在执行 Linux 系统调用的线程共同处理。
/// 等到处理完成之后，返回系统调用的返回值。
pub fn run_user_top(thread : Arc<LxThread>) -> isize {
    loop {
        if let Some(task) = { || GLOBAL_EXECUTOR.lock().pop_runnable_task() }() {
            match task.inner {
                TaskInner::Kern(future) => {...} // future.poll()
                TaskInner::User(future, thr) => {
                    ... // future.poll()
                    if Polling::Ready(ret) = ret {
                        *thr.ret_val() = Some(ret);
                    }
                }
            }
        }
        if let Some(ret) = *thread.ret_val() {
            return ret;
        }
    }
}
```

2.6 在内核中通过 mmap 实现内存映射

`mmap` 系统调用可以将文件的内容映射到内存上，一般用来分配内存、共享内存、加载动态库等。

这里使用 `mmap` 将文件作为物理地址，然后将用户空间作为虚拟地址，使用 `mmap` 进行虚拟地址的映射，从而允许 zCore 管理虚拟地址的映射关系，使 zCore 与 Linux 主机完全解耦。

不过 `mmap` 的内存与文件的同步是有损耗的，每次与文件同步都需要执行一次同步系统调用 `fsync` 才能完成同步。我们需要使用 tmpfs 作为我们的物理地址，tmpfs 是在内存上的文件系统，读/写没有很大的损耗。

```
// kernel-hal 硬件抽象层
/// Map the page of `vaddr` to the frame of `paddr` with `flags`.
#[export_name = "hal_pt_map"]
pub fn map(&mut self, vaddr: VirtAddr, paddr: PhysAddr, flags: MMUFlags) -> Result<(), ()> {
    ......
    syscall::mmap(.....)
    Ok(())
}

/// 使用 read 系统调用读取物理地址
#[export_name = "hal_pmem_read"]
pub fn pmem_read(paddr: PhysAddr, buf: &mut [u8]) {
    ensure_mmap_pmem();
    syscall::read(FRAME_FILE.fd, buf.as_mut_ptr(), buf.len());
}
```

2.7 实现多个容器系统的隔离

- 为了保证多个容器直接不互相影响，从安全方面考虑，应当为每个容器设置一个物理地址空间。
- 同时，zCore 中的空闲帧表是静态的，需要为每个容器配备一个。

```
struct Container {
    id : usize,
    phys_file: PathBuf, // PathBuf 是自己实现的
    available_frame : VecDeque<usize>,
}
```

2.8. rvisor-linux-syscall 的实现

rvisor-linux-syscall 实现的总结

```
基本完整地实现：read write openat close fstat fstatat lseek ioctl pread pwrite readv writev
sendfile fcntl fsync fdatasync truncate ftruncate getdents64 getcwd chdir
renameat mkdirat linkat unlinkat readlinkat faccessat copy_file_range sync
clone execve exit exit_group wait4 set_tid_address clock_gettime getpid gettid
uname getppid open stat lstat access dup2 fork vfork rename mkdir rmdir link
unlink readlink (53 个)

简单地转换后进入 Linux：mmap mprotect munmap brk (4 个)

使用的系统调用：
    stat lseek read write fdatasync fsync openat close mkdir rename（文件系统）
    fork clone execve exit wait4 （进程管理）

内核函数/宏：
```

```
    printk BUG preempt_disable preempt_enable
    __copy_from_user __copy_to_user strncpy_from_user set_fs get_fs
```