zCore 整体设计

zCore Summer of Code Day2

王润基

2020.08.04 @ 鹏城实验室

提纲

- zCore 项目背景和开发历史
- zCore 整体结构
- HAL 硬件抽象层的设计实现
- async Rust 原理和设计模式
- Rust 工程项目实践经验

zCore 立项背景

rCore 快速发展

- 支持各种 Linux 系统调用
- 支持四种指令集和诸多物理硬件

代码日益臃肿

- 内部设计缺乏推敲
- 希望探索微内核架构

反思 rCore 的问题

Rust 应该怎么写?

设计风格仍有 C 语言痕迹，存在大量 unsafe 代码

代码应该如何组织划分?

内核代码增长到 2 万行，模块化不够彻底，不够社区规范

Linux 是不是最佳的设计?

宏内核，不区分线程与进程，存在 fork、signal 等历史糟粕

=> 需要一次彻底的重构

zCore 开发历史

调研阶段

- 2019.09：在 OSTrain 课上提出探索微内核的设想
  - 锁定目标：Fuchsia / Zircon
- 2019.10：开始对 Zircon 的调研学习工作
  - 微观：内核对象的设计
  - 宏观：系统的启动和运作流程

验证阶段

- 2019.12：尝试用 Rust 编写基础的 Zircon 内核对象
  - 实现进程和内存管理相关的内核对象
  - 在单元测试的驱动下，逐渐形成了**硬件抽象层**
  - 进一步验证了 **用户态** OS 的可行性
- 2020.01：zCore 整体框架定型
  - 实现信号等待，验证了 async **异步机制** 的可行性
  - 基于 Zircon 内核对象实现了 Linux **系统调用**

开发阶段

- 2020.02：
  - “移植” 回裸机环境
  - 针对第一个用户进程 userboot 实现系统调用
- 2020.03:
  - 理解用户程序行为，推进系统调用实现
- 2020.04：
  - 支持运行 Shell！宣告核心功能开发基本完成

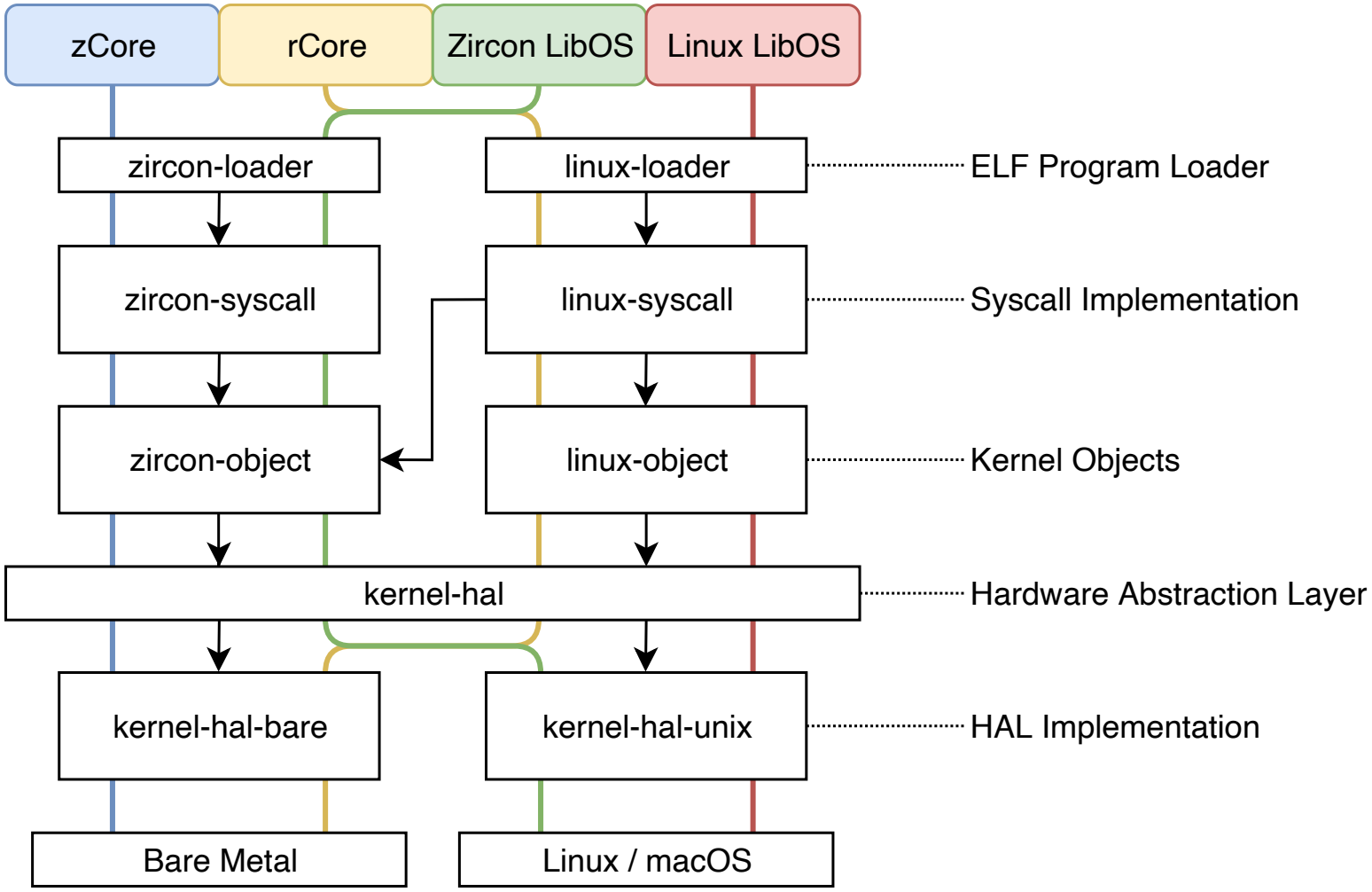
完善阶段

- 2020.05: 支持驱动程序
  - 实现了 DDK 驱动相关内核对象和系统调用
  - 使用 microbenchmark 进行性能分析和优化
- 2020.06-07：继续实现缺失功能
  - 实现异常处理机制
  - 基于 RVM 实现 Hypervisor
  - x86 多核支持，开始 ARM64 的移植工作

zCore: A Next Gen "Rust" OS!

- 可以完全在用户态开发、测试、运行
- 由若干 crates 组装而成，符合 Rust 社区标准
- 使用 Rust async 机制，内核协程化
- 继承 rCore，同时支持 Linux 和 Zircon 系统调用

整体架构

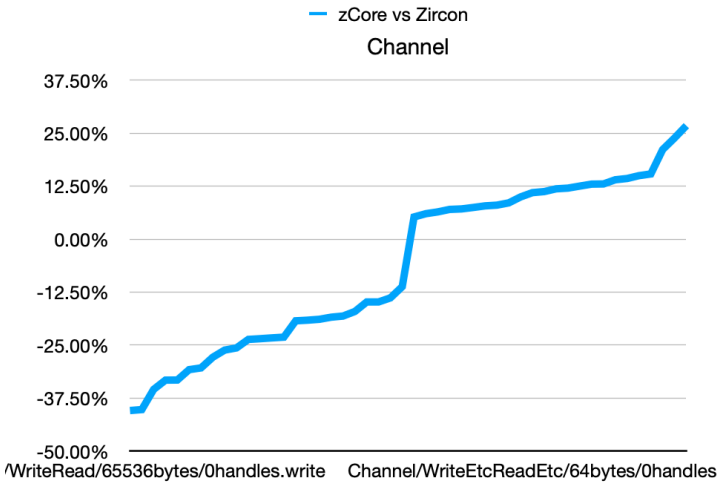

zCore
rCore
Zircon LibOS
Linux LibOS
zircon-loader
linux-loader
ELF Program Loader
zircon-syscall
linux-syscall
Syscall Implementation
zircon-object
linux-object
Kernel Objects
kernel-hal
Hardware Abstraction Layer
kernel-hal-bare
kernel-hal-unix
HAL Implementation
Bare Metal
Linux / macOS

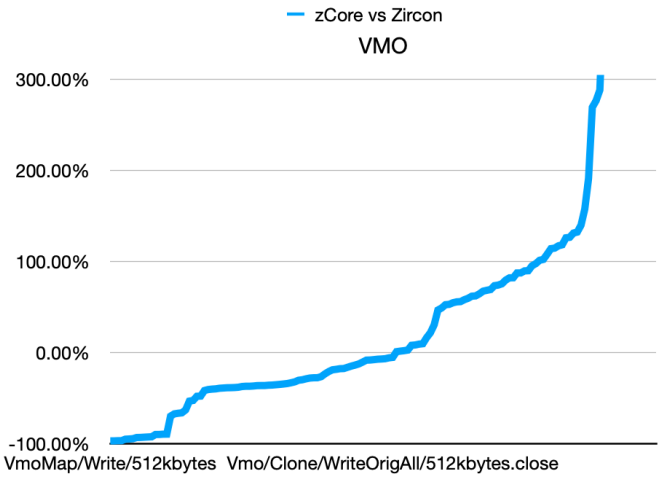
完成度： Core Tests

Test	Status
Bti	⚠️ 7/8
ConditionalVariableTest	✅ 3/3
C11MutexTest	✅ 5/5
C11ThreadTest	✅ 6/6
ChannelInternalTest	✅ 2/2
ChannelTest	✅ 38/38
ChannelWriteEtcTest	✅ 27/27
ClockTest	✅ 2/2
ProcessDebugUtilsTest	✅ 1/1
ProcessDebugTest	❌ 0/4
ExecutableTlsTest	✅ 12/12
EventPairTest	✅ 8/8
FifoTest	✅ 9/9
FPUTest	✅ 1/1
FutexTest	✅ 14/14
HandleCloseTest	⚠️ 2/3
HandleDup	✅ 4/4
HandleInfoTest	✅ 4/4
HandleTransferTest	✅ 2/2
HandleWaitTest	✅ 2/2
InterruptTest	⚠️ 6/7
JobTest	⚠️ 8/26
MemoryMappingTest	⚠️ 5/8
ObjectChildTest	✅ 1/1
ObjectGetInfoTest	✅ 4/4
JobGetInfoTest	⚠️ 24/39
ProcessGetInfoTest	⚠️ 25/69
TaskGetInfoTest	⚠️ 11/12
ThreadGetInfoTest	⚠️ 26/41
VmarGetInfoTest	⚠️ 19/21
ObjectWaitOneTest	✅ 5/5
ObjectWaitManyTest	✅ 5/5
Pager	❌ 0/76
PortTest	⚠️ 10/18
ProcessTest	⚠️ 8/26
SchedulerProfileTest	❌ 0/14
Pthread	✅ 6/6
PThreadBarrierTest	✅ 3/3
PthreadTls	✅ 1/1
Resource	❌ 0/11
StackTest	✅ 2/2
StreamTestCase	❌ 0/11
SyncCompletionTest	✅ 11/11
SyncCondition	✅ 2/2
SyncMutex	✅ 3/3
SystemEvent	❌ 0/9
Threads	⚠️ 12/36
TicksTest	✅ 1/1
Vmar	⚠️ 4/33
VmoCloneTestCase	✅ 6/6
VmoClone2TestCase	⚠️ 32/41
VmoCloneDisjointClonesTests	✅ 2/2
VmoCloneResizeTests	⚠️ 2/4
ProgressiveCloneDiscardTests	✅ 2/2
VmoSignalTestCase	✅ 3/3
VmoSliceTestCase	⚠️ 15/15
VmoZeroTestCase	✅ 12/12
VmoTestCase	⚠️ 13/27

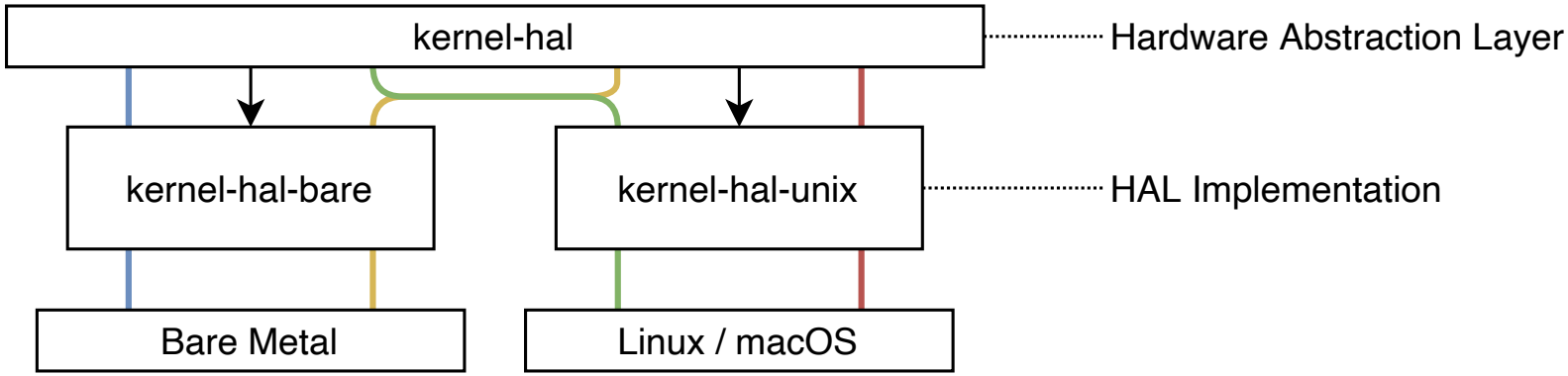
Benchmark

基于 Ubuntu 20.04，QEMU-KVM 1 CPU 测试





HAL 硬件抽象层的设计实现



需求：内核对象单元测试

- 测试对象：线程 `Thread`，内存映射 `VMO`， `VMAR`
- 但 `cargo test` 只能在开发环境用户态运行
- 思考：能否在用户态**模拟**页表和内核线程?

方案：用户态模拟内核机制

- 内核线程：等价于用户线程 `std::thread`
- 内存映射：Unix `mmap` 系统调用
  - 用一个文件代表全部物理内存
  - 用 `mmap` 将文件的不同部分映射到用户地址空间

潜在问题

- 用户线程难以细粒度调度
- “页表”共享同一个地址空间

进一步思考：用户态 OS

既然每个内核对象都能在用户态完成其功能，

那么整个 OS 可不可以完全跑在用户态呢?

潜在好处

充分利用用户态丰富的开发工具，降低开发难度：
IDE + gdb + cargo + perf ...

现有解决方案

Library OS，User-mode Linux

最后的技术难点：用户-内核态切换 🧙‍♀️

用户程序和内核都运行在用户态……

- 控制流转移：系统调用 -> 函数调用
  - `int 80` / `syscall` -> `call`
  - `iret` / `sysret` -> `ret`
  - 需要修改用户程序代码段！
- 上下文恢复：寻找 "scratch" 寄存器
  - 用户程序如何找到内核入口点？内核栈？
  - 利用线程局部存储 TLS，线程指针 fsbase
  - macOS 无法设置 fsbase 怎么办？

实现：接口函数 + 弱链接

`kernel-hal` 定义了全部接口函数：

```rust
#[linkage = "weak"]
#[export_name = "hal_pmem_read"]
pub fn pmem_read(_paddr: PhysAddr, _buf: &mut [u8]) {
    unimplemented!()
}
```

`kernel-hal-bare` 和 `kernel-hal-unix` 分别提供它们在裸机环境和 Unix 系统上的实现：

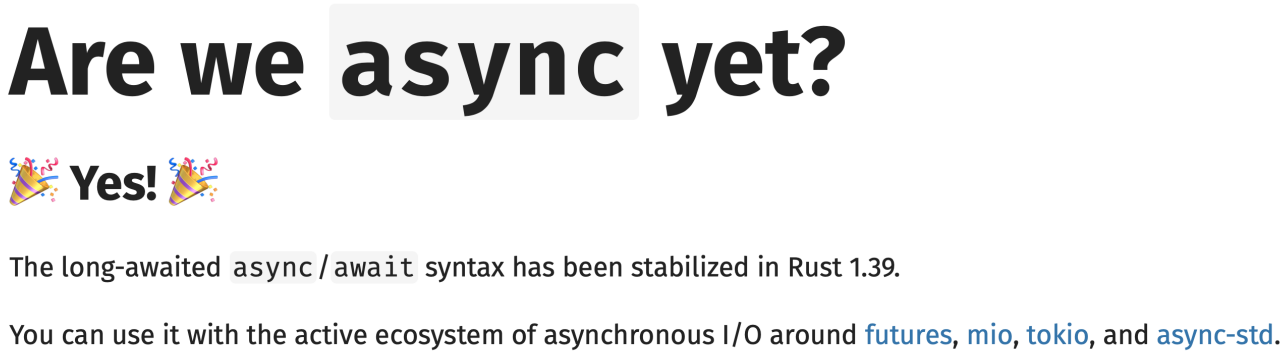
```rust
#[export_name = "hal_pmem_read"]
pub fn pmem_read(_paddr: PhysAddr, _buf: &mut [u8]) {...}
```

使用时引用其中某一个实现库，用强符号覆盖弱符号。

HAL API 举例

- 内核线程：`hal_thread_spawn`
- 物理内存：`hal_pmem_{read,write}`
- 虚拟内存：`hal_pt_{map,unmap}`
- 用户态：`hal_context_run`
- 定时器：`hal_timer_{set,tick}`
- 输入输出：`hal_serial_{read,write}`
- 设备管理：`hal_irq_{enable,handle}`

async Rust 原理和设计模式



👍 推荐阅读：《使用 Rust 编写操作系统》#12 Async/Await

`async-await`：用同步风格编写异步代码

- 本质：无栈协程，协作式调度
- 适用于高并发 IO 场景

应用情况：

- 需要编译器的特殊支持：函数 => 状态机对象
- 主流编程语言均已支持：C#，JavaScript，Python，C++
- 几乎没有在 bare-metal 中应用

Rust Async 时间线

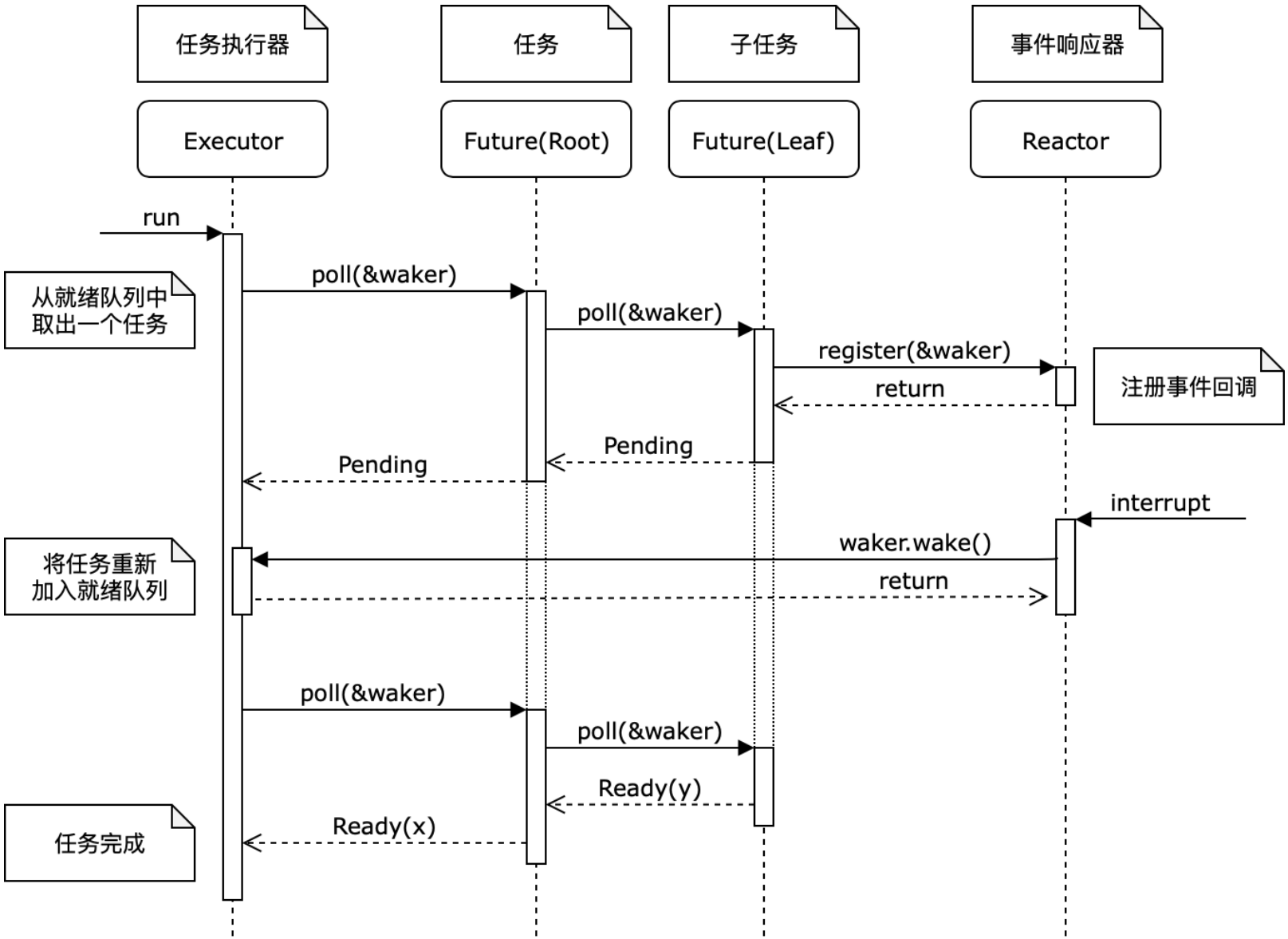
- 2019.11.07：async 进入 stable Rust 1.39
  宣告 async 在用户程序中正式可用！
- 2020.01.16：在 zCore 中第一次引入 async
  验证了用 async 编写内核逻辑的可行性
- 2020.03.21：Rust PR#69033 进入主线
  宣告 async 在 no_std 中正式可用！
- 2020.03.28：BlogOS 发布 Async/Await 文章
  标志着 async 在 嵌入式 / OS 领域即将得到广泛应用

Sync

```
fn handler(mut stream: TcpStream) -> Result<()> {
    let mut buf = [0; 1024];
    let len = stream.read(&mut buf)?; // may block
    stream.write_all(&buf[0..len]))?; // may block
}
```

Async

```
// fn handler(...) -> impl Future<Output = Result<()>>
async fn handler(mut stream: TcpStream) -> Result<()> {
    let mut buf = [0; 1024];
    let len = stream.read(&mut buf).await?;
    stream.write_all(&buf[0..len])).await?;
}
```


任务执行器
任务
子任务
事件响应器
Executor
Future(Root)
Future(Leaf)
Reactor
run
从就绪队列中
取出一个任务
poll(&waker)
poll(&waker)
register(&waker)
return
注册事件回调
Pending
Pending
interrupt
waker.wake()
将任务重新
加入就绪队列
return
poll(&waker)
poll(&waker)
Ready(y)
Ready(x)
任务完成

底层：手动构造 Future

```rust
// 例：在内核对象上等待信号
fn wait_signal(&self, signal: Signal) -> WaitSignal {
    // 定义一个状态机结构
    struct WaitSignal {...}
    // 实现 Future trait 的 poll 函数
    impl Future for WaitSignal {
        type Output = Signal;
        fn poll(self: Pin<&mut Self>, cx: &mut Context)
            -> Poll<Self::Output>
        {
            // 若目标事件已发生，直接返回 Ready
            if self.signal().contains(signal) {
                return Poll::Ready(signal);
            }
            // 尚未发生：注册回调函数，当事件发生时唤醒自己
            let waker = cx.waker().clone();
            self.add_signal_callback(move || waker.wake());
            Poll::Pending
        }
    }
    // 返回状态机对象
    WaitSignal {...}
}
```

中层：用 async-await 组合 Future

```
async fn sys_object_wait_signal(...) -> Result {
    ...
    let signal = kobject.wait_signal(signal).await;
    ...
}
```

高级用法：用 `select` 组合子实现 超时处理 和 异步取消

```
async fn sys_object_wait_signal(..., timeout) -> Result {
    ...
    let signal = select! {
        s = kobject.wait_signal(signal) => s,
        _ = delay_for(timeout) => return Err(Timeout),
        _ = cancel_token => return Err(Cancelled),
    };
    ...
}
```

上层：Executor 运行 Future

- libos：`tokio` / `async-std`，支持多线程，可以模拟多核
- bare：`rcore-os/executor`，简易单核
- 未来：期望嵌入式社区的 `async-nostd`？

进出用户态问题

async 要求保持内核上下文（即内核栈）

- 传统 OS：User call Kernel 风格，平时内核栈清空
- zCore：Kernel call User 风格，保留内核上下文

Rust 工程项目实践经验

- 代码质量控制：`cargo fmt + cargo clippy`
- 文档和单元测试：`cargo doc + cargo test + grcov`
- crate 的拆分和发布流程：`cargo publish`
- 持续集成和自动测试：GitHub Actions
- 社区合作开发：GitHub issue + PR

zCore 自动测试项目

- `#![deny(warnings)]`：警告报错
- cargo fmt && clippy：检查代码格式和风格
- cargo build：保证所有平台编译通过
- cargo test：用户态单元测试，报告测试覆盖率
- core-test：内核态集成测试，维护通过测例列表
- （TODO）cargo bench：性能测试

上述测试全部通过才允许合入 master

模块化：rCore OS 生态

拆成小型 no_std crate，每个专注一件事：

- `trapframe-rs`：用户-内核态切换
- `rcore-console`：在 Framebuffer 上显示终端
- `naive-timer`：简单计时器
- `executor`：单线程 Future executor
- ……

完成功能后，继续完善文档、测试、examples

基本稳定后，发布到 crates.io，遵守语义化版本号更新

总结

期望大家通过 zCore 项目：

- 了解最新工业级操作系统内核的设计模式
- 从更高层视角把握操作系统的整体结构
- 更深入、更规范、更优雅地使用 Rust
- 继续推进 Rust 在系统领域的应用

谢谢大家

PS：感谢康总大力支持！

陈康

zCore是我的，一定加上感谢我！

陈康

我支持zCore!!!