zCore 简介

王润基

2021.02.25

提纲

- 整体介绍
- 主要技术：用户态 OS，async 异步
- Fuchsia 与 Zircon 内核对象简介

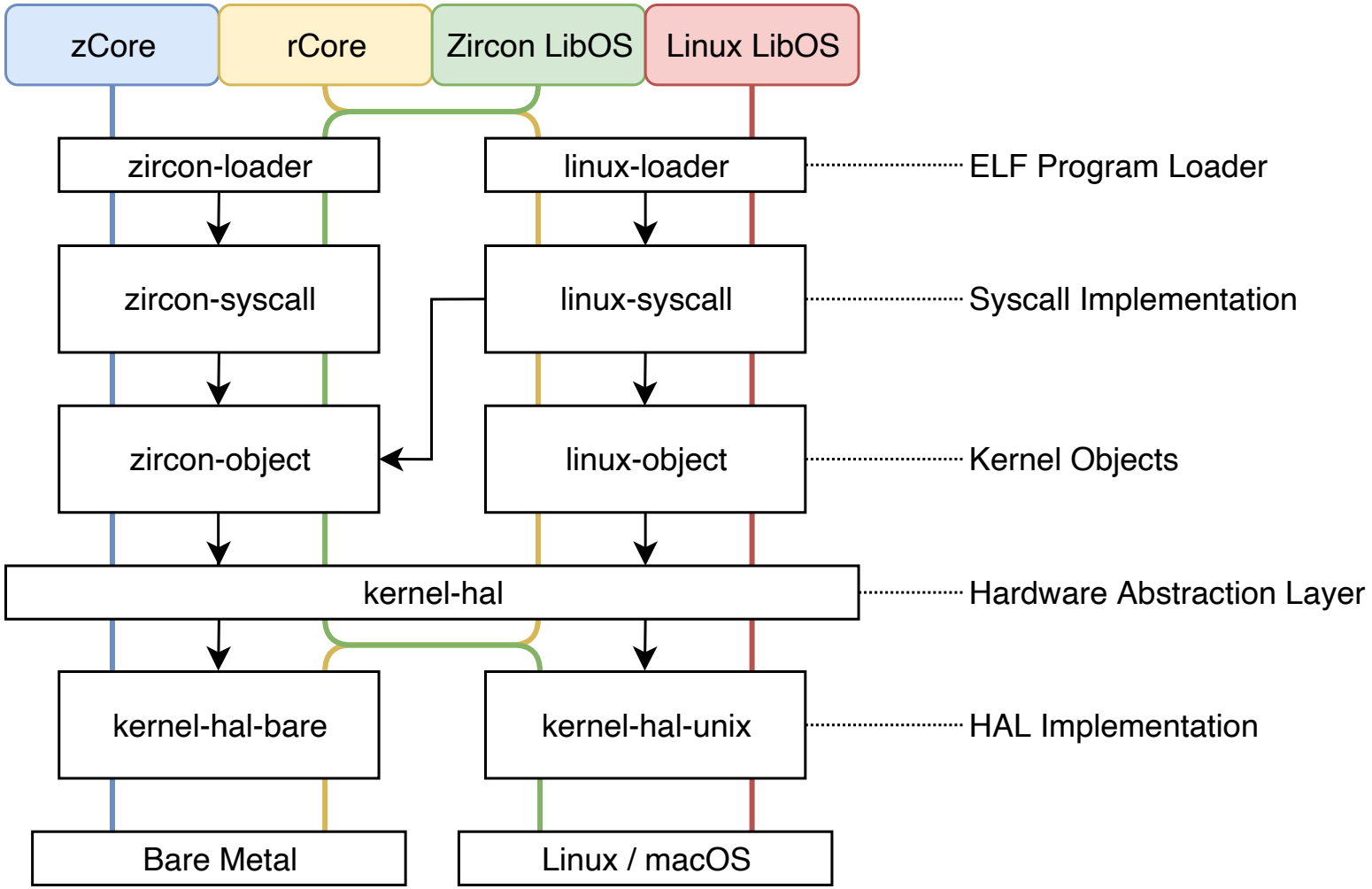
zCore: A Next Gen Rust OS!

Rust 语言编写的“混合”操作系统内核

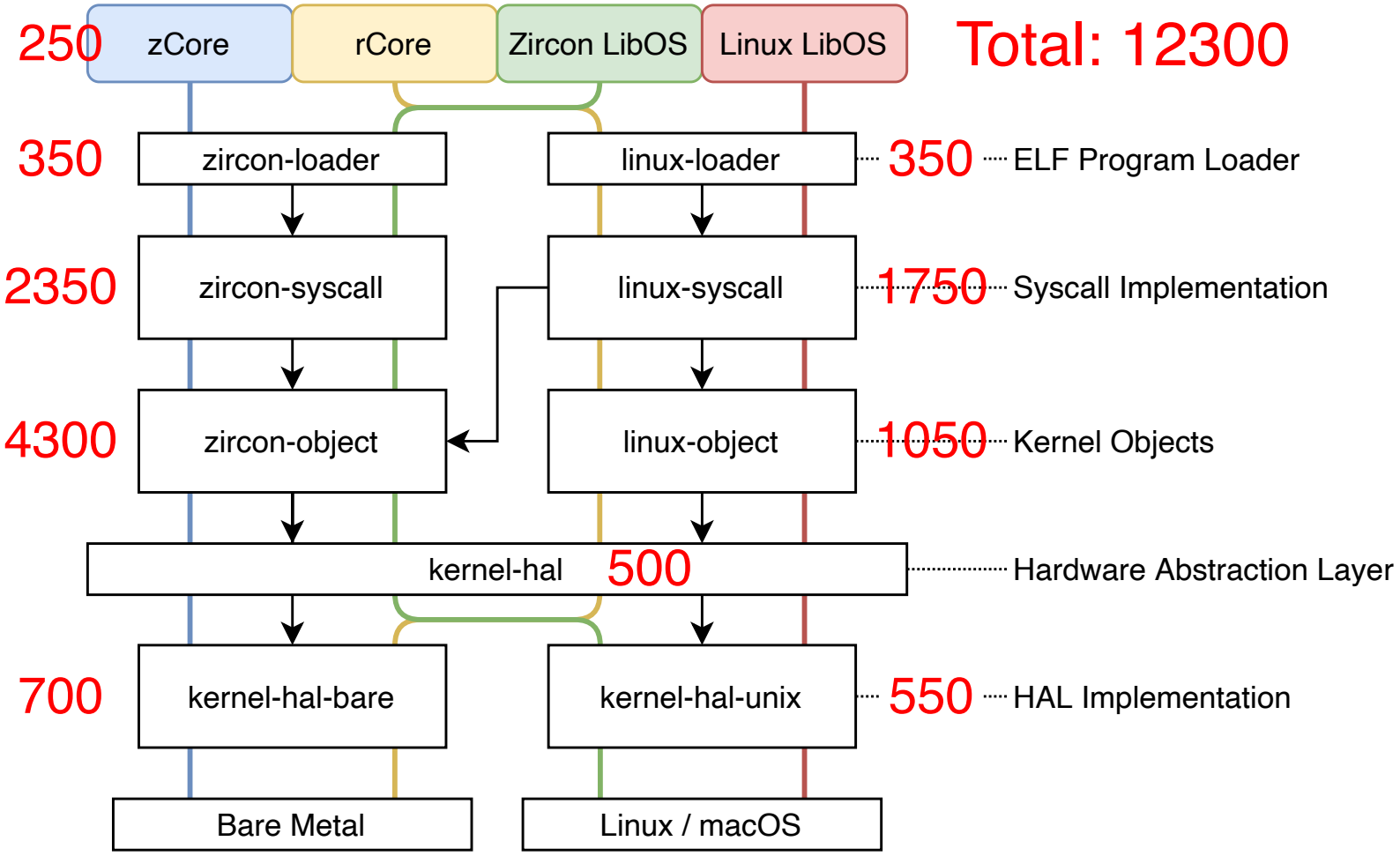
- 同时支持 Linux 和 Zircon 系统调用
- 同时支持 LibOS 和 裸机 OS 形式

  可以完全在用户态开发、测试、运行
- 符合 Rust 风格，项目模块化
- 使用 Rust async 机制，内核协程化

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

项目规模

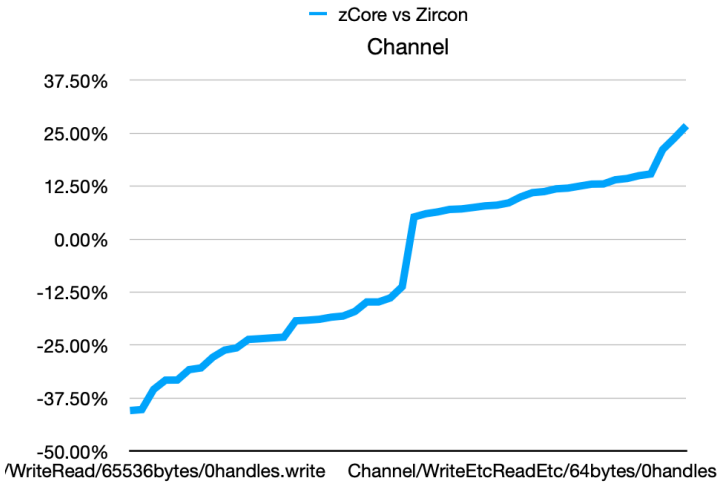

250
zCore
rCore
Zircon LibOS
Linux LibOS
Total: 12300
350
zircon-loader
linux-loader
350
ELF Program Loader
2350
zircon-syscall
linux-syscall
1750
Syscall Implementation
4300
zircon-object
linux-object
1050
Kernel Objects
kernel-hal
500
Hardware Abstraction Layer
700
kernel-hal-bare
kernel-hal-unix
550
HAL Implementation
Bare Metal
Linux / macOS

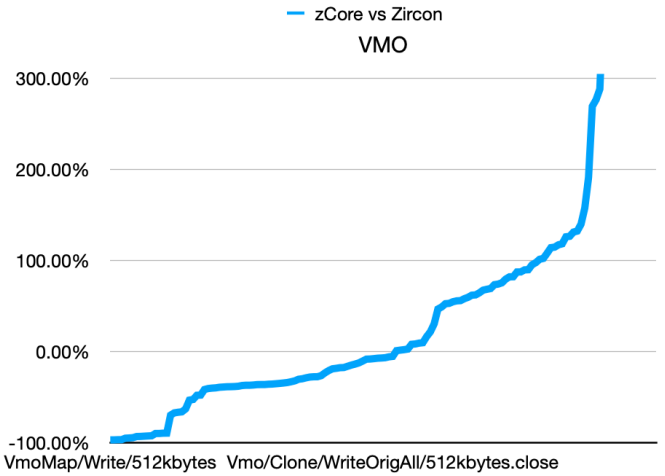
完成度：Zircon 官方系统调用测试集

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





软件工程：自动测试

- `#![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

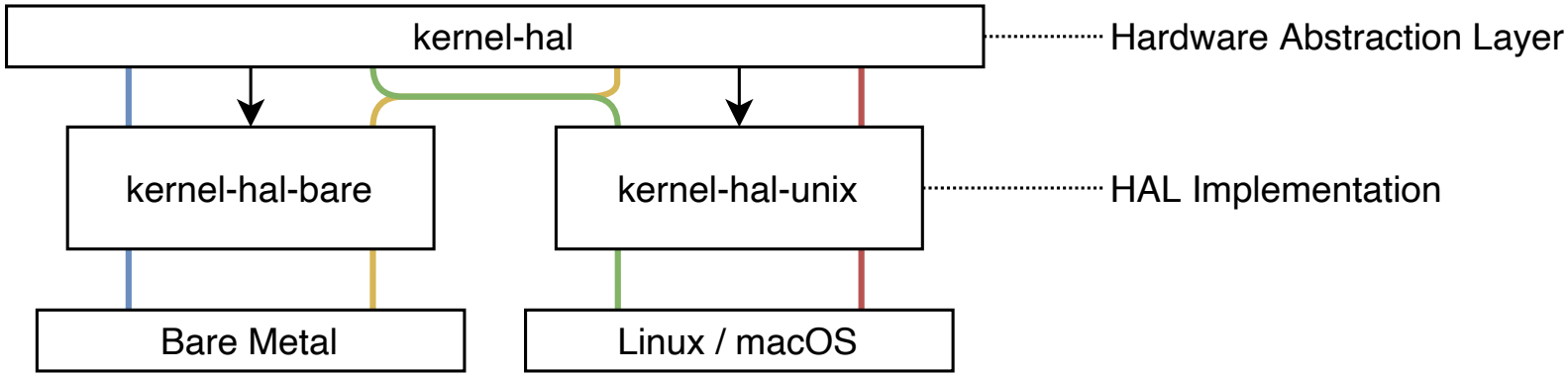
基本稳定后，发布到 crates.io，遵守语义化版本号更新

zCore 开发历史

- 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
- 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**
- 2020.02：支持第一个用户进程 userboot
- 2020.03：更多系统调用实现
- 2020.04：支持运行 Shell！宣告核心功能开发基本完成
- 2020.05：支持驱动程序
- 2020.06-07：实现异常处理和 Hypervisor

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？

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

Are we `async` yet?

🎉 Yes! 🎉

The long-awaited `async` / `await` syntax has been stabilized in Rust 1.39.

You can use it with the active ecosystem of asynchronous I/O around futures, mio, tokio, and async-std.

👍 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：

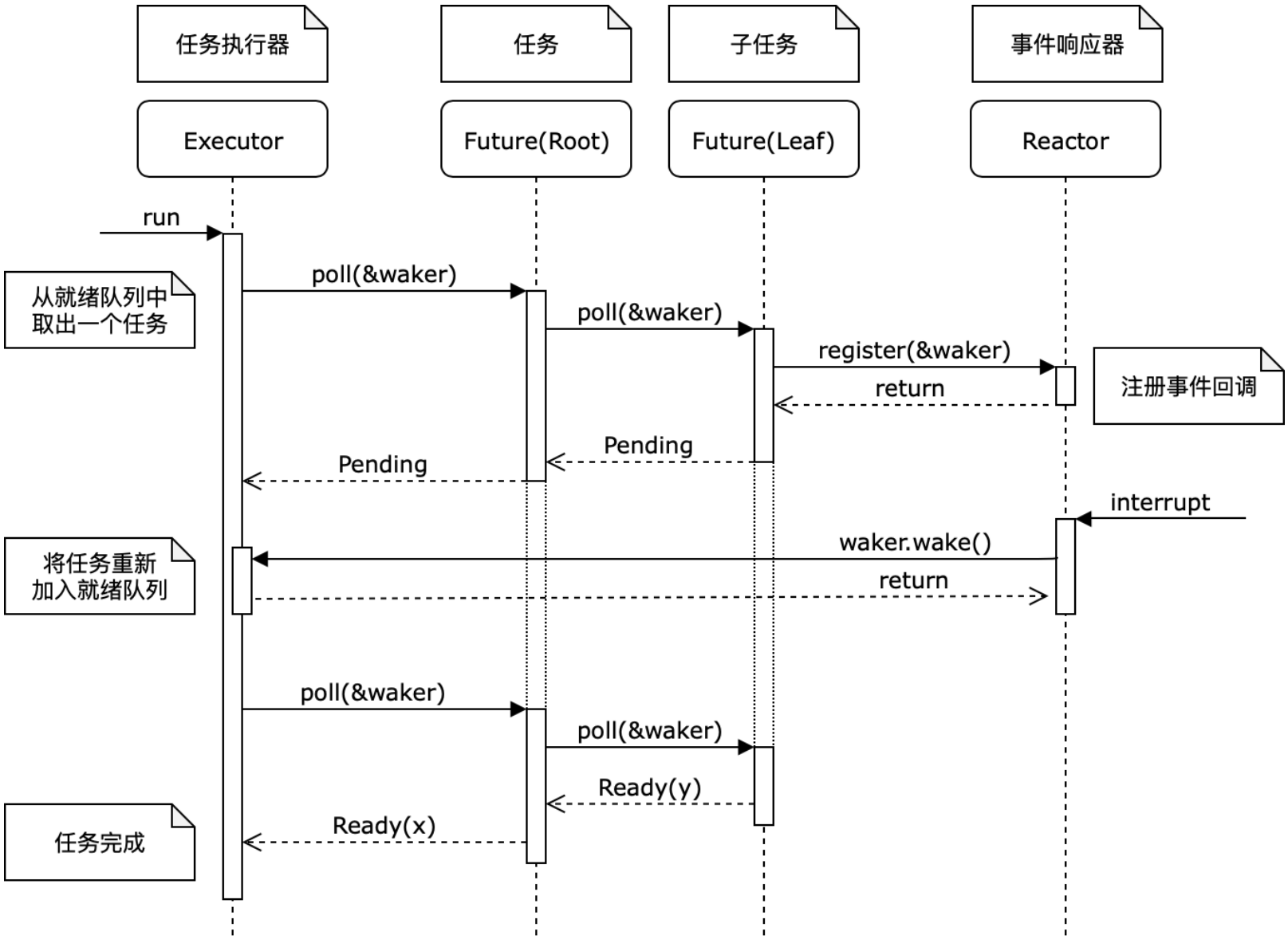
- 需要编译器的特殊支持：函数 => 状态机对象
- 主流编程语言均已支持：C#，JavaScript，Python，C++
- 几乎没有在 bare-metal 中应用

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
将任务重新
加入就绪队列
waker.wake()
return
poll(&waker)
poll(&waker)
Ready(y)
任务完成
Ready(x)

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

Fuchsia 与 Zircon 内核对象

The Fuchsia layer cake

Topaz

implements interfaces defined by underlying layers & contains four major categories of software:
modules, agents, shells, and runners.

- modules include the calendar, email, and terminal modules;
- shells include the base shell and the user shell;
- agents include the email and chat content providers;
- runners include the Web, Dart, and Flutter runners.

Peridot

provides the services needed to create a cohesive, customizable, multi-device user experience assembled from modules, stories, agents, entities, and other components.

device, user, and story runners. and the ledger and resolver, as well as the context and suggestion engines

Garnet

provides device-level system services for software installation, administration, communication with remote systems, and product deployment.

network, media, and graphics services. and the package management and update system

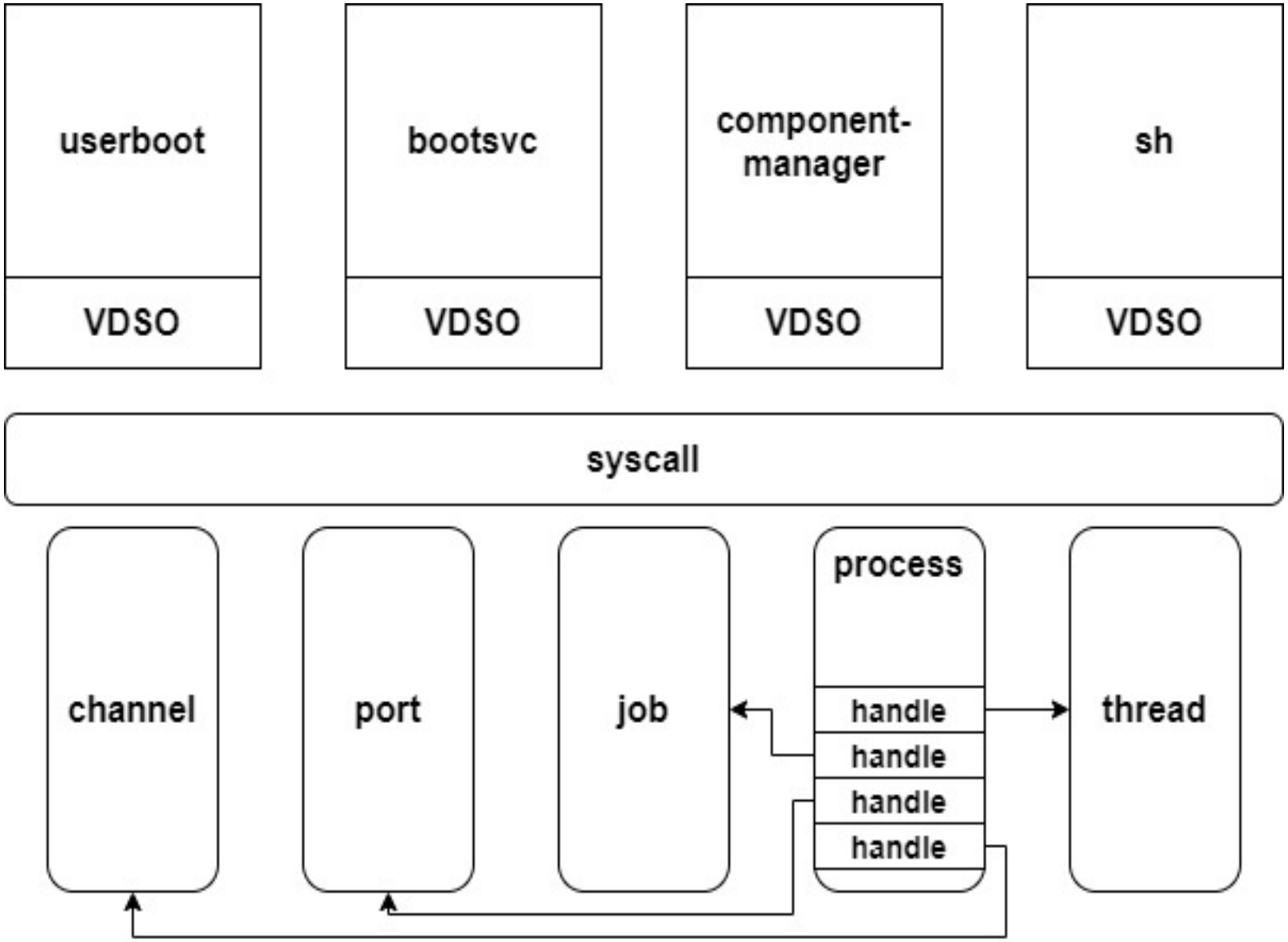
Zircon

mediates hardware access, implements essential software abstractions over shared resources, and provides a platform for low-level software development.

kernel, device manager, most core and first-party device drivers, and low-level system libraries, such as libc and fdio. and defines the Fuchsia IDL (FIDL)

Zircon 内核特点

- 实用主义微内核
- 使用 C++ 实现，支持 x86_64 和 ARM64
- 面向对象：将功能划分到内核对象
- 默认隔离：使用 Capability 进行权限管理
- 安全考量：强制地址随机化，使用 vDSO 隔离系统调用


userboot
bootsvc
component-
manager
sh
VDSO
VDSO
VDSO
VDSO
syscall
channel
port
job
process
handle
handle
handle
handle
thread

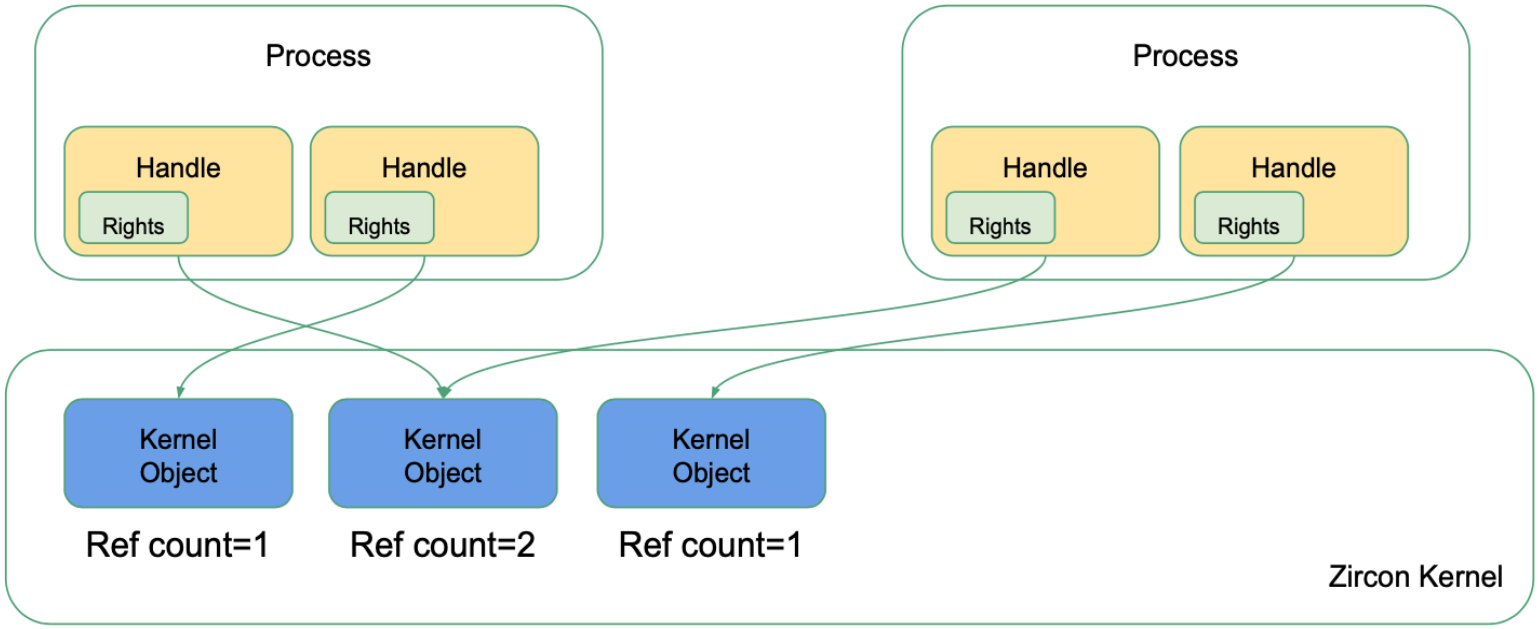
Zircon 内核对象

Everything can be KernelObject

- 任务：Job, Process, Thread, Exception
- 内存：VMAR, VMO, Pager, Stream
- IPC：Channel, FIFO, Socket
- 信号：Event, Timer, Port, Futex
- 驱动：Resource, Interrupt, PCI ...

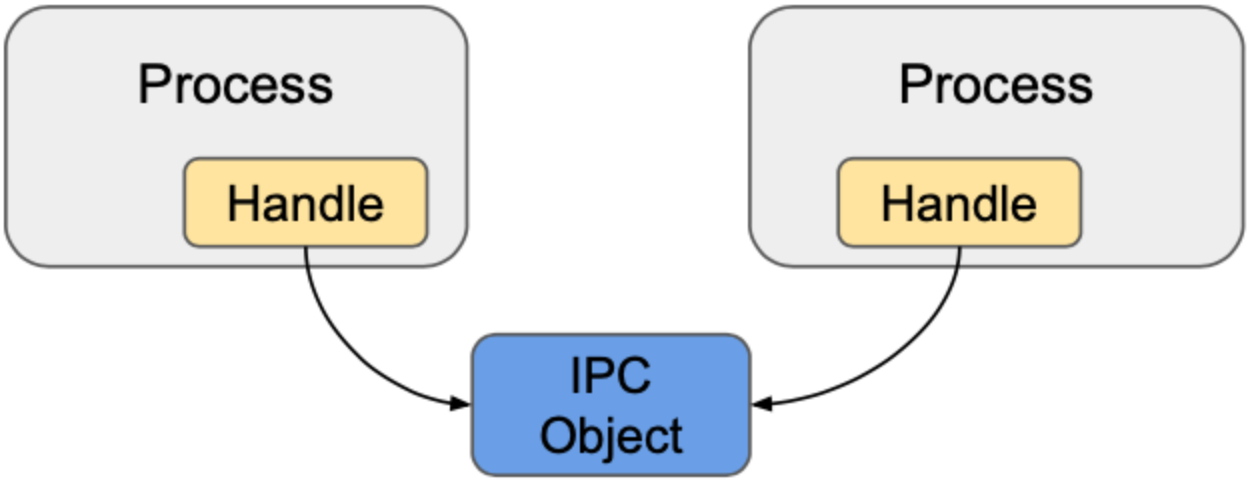
Object

- Object：内核对象
- Rights：对象访问权限
- Handle = Object + Rights：对象句柄（类似 fd）



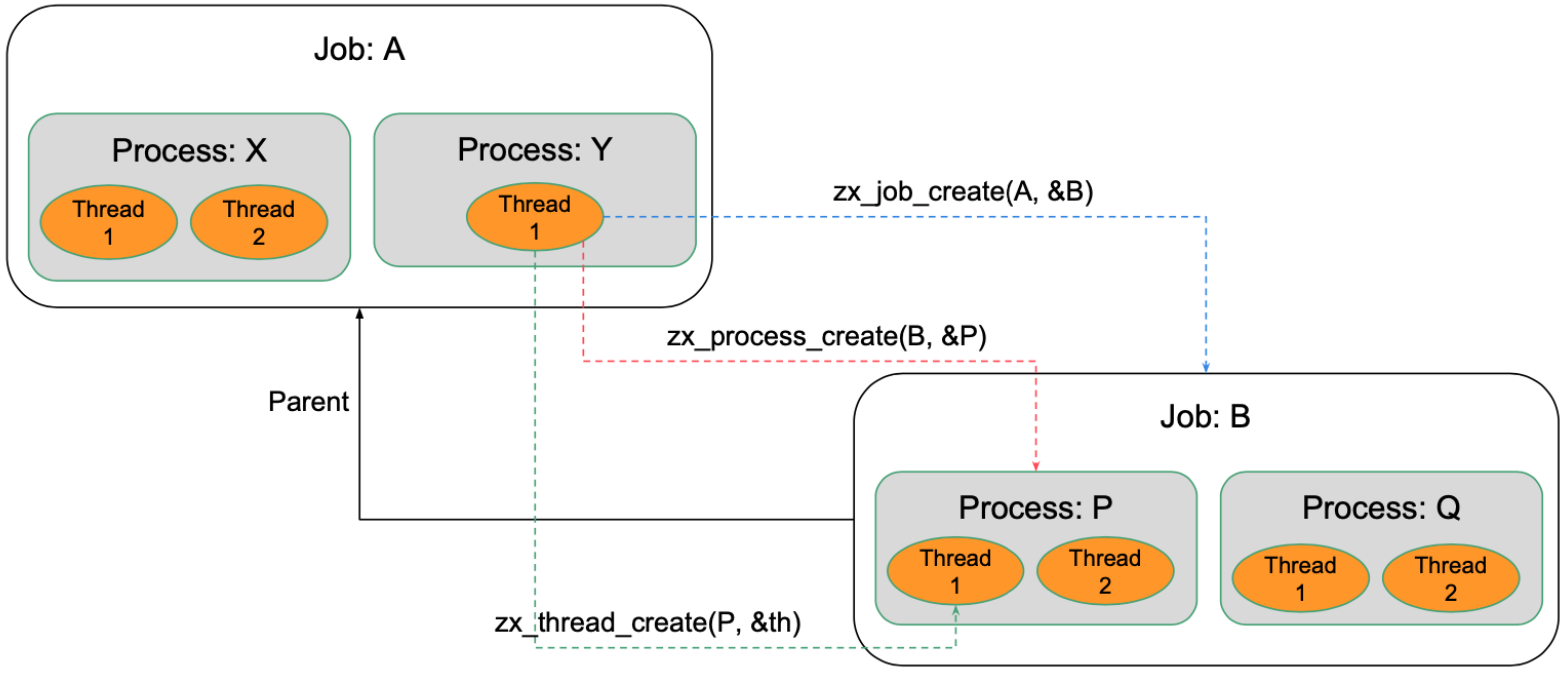
IPC

- Channel：进程间通信基础设施，可以传递数据和 handle
- FIFO：报文数据传输
- Socket：流数据传输



Tasks

- Job：作业，负责控制权限（类似容器）
- Process：进程，负责管理资源
- Thread：线程，负责调度执行



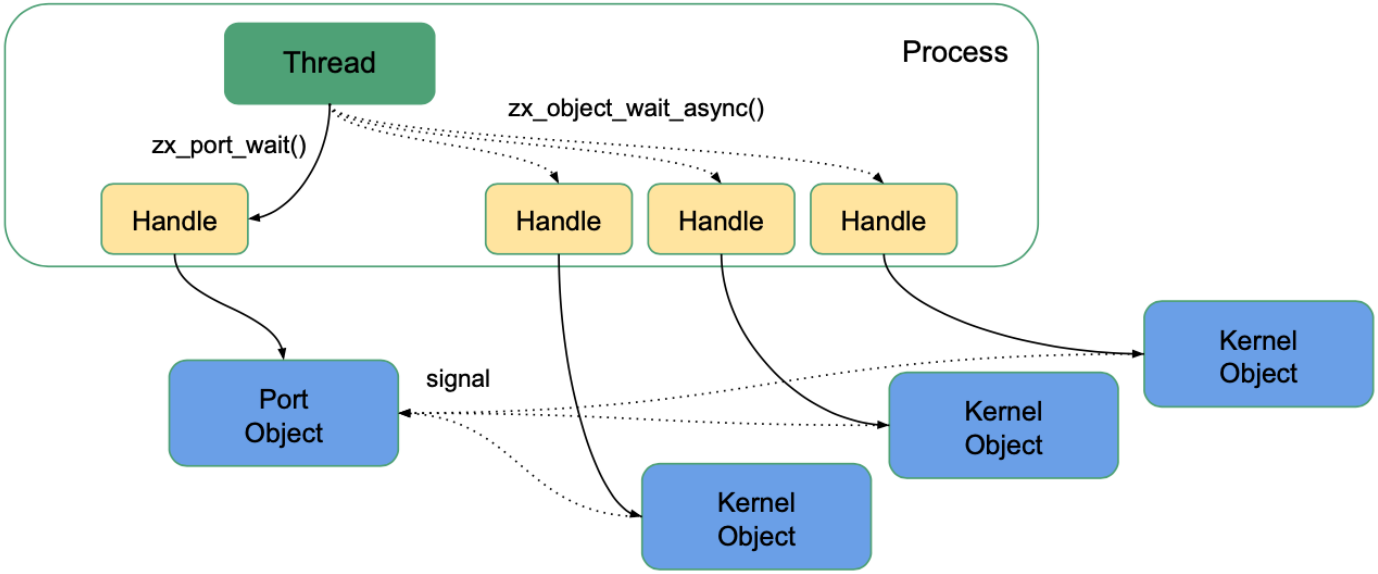
Memory and address space

- VMO: Virtual Memory Object
  - Paged：分页物理内存，支持写时复制
  - Physical：连续物理内存
- VMAR: Virtual Memory Address Region
  - 代表一个进程的虚拟地址空间
  - 树状结构
- Pager：用户态分页机制

Signaling and Waiting

每个 Object 有 32 个信号位，用户程序可以阻塞等待。

- Event (Pair)：事件源/对
- Timer：计时器
- Futex：用户态同步互斥机制
- Port：事件分发机制（类似 epoll）



谢谢