

zCore 入门导引

张译仁

2022.3.19

提纲

- 总体介绍
- 各模块实现

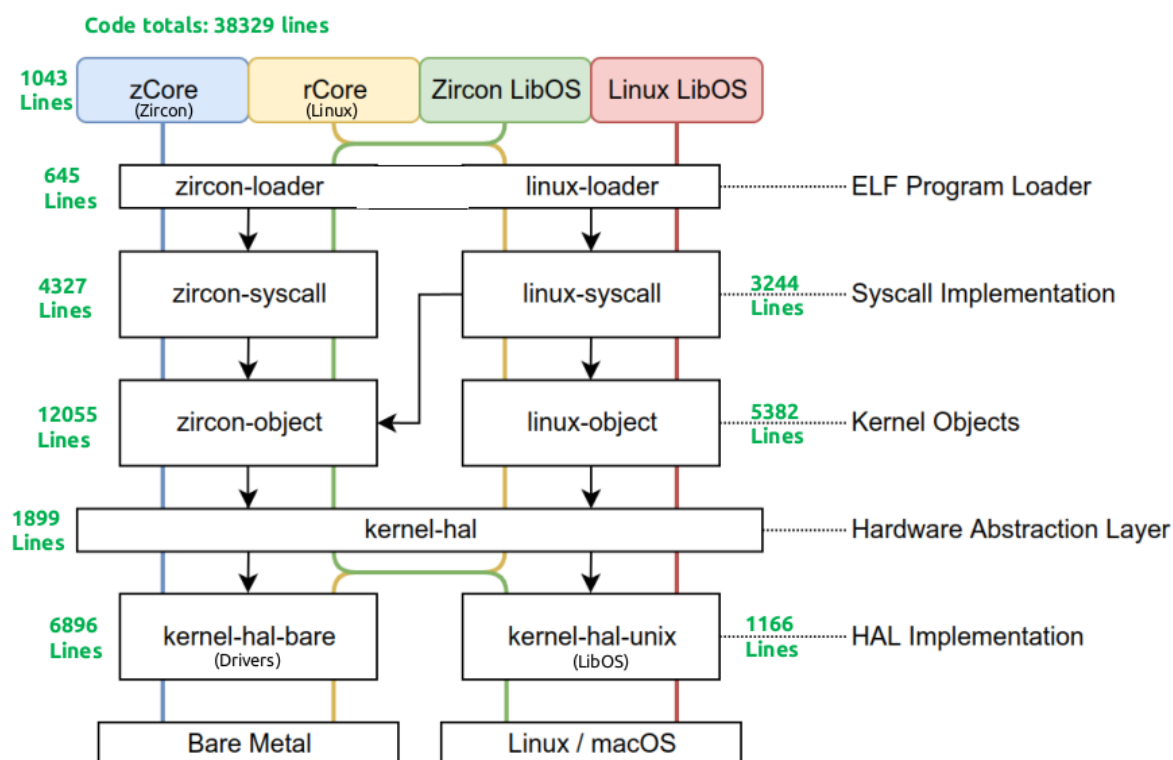
总体介绍 -- 特征

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
- 内核协程化
 - 扩展 Rust Async 机制
 - 重新思考 syscall 以及调度

总体介绍 - 项目规模

目前大约 3.8 万行左右代码量

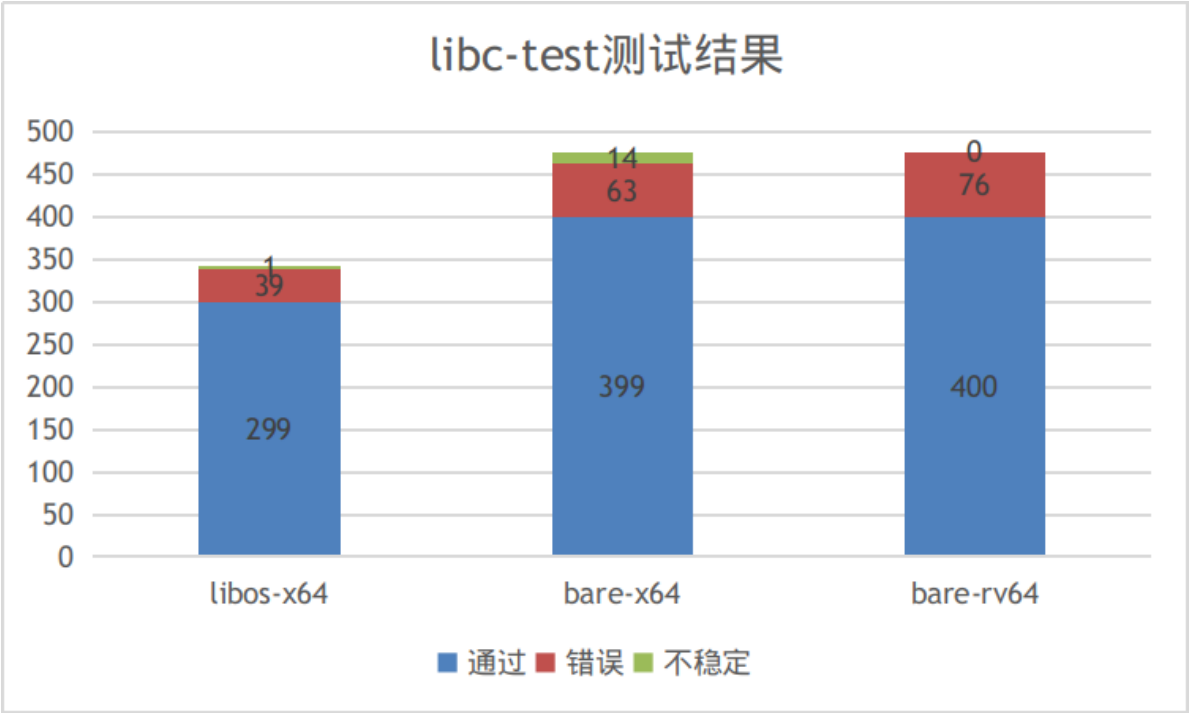


总体介绍 - 完成度

Zircon 官方系统调用测试集

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

总体介绍 - 完成度



总体介绍 - zCore架构设计文档

zCore 架构设计文档

简明 zCore 教程

zCore 整体结构和设计模式

Fuchsia OS 和 Zircon 微内核

1. 内核对象

1.1. 初识内核对象

1.2. 对象管理器：Process 对象

1.3. 对象传送器：Channel 对象

2. 任务管理

2.1. Zircon 任务管理体系

2.2. 进程管理：Process 与 Job 对象

2.3. 线程管理：Thread 对象

3. 内存管理

3.1. Zircon 内存管理模型

3.2. 物理内存：VMO 对象

3.3. 物理内存：按页分配的 VMO

3.4. 虚拟内存：VMAR 对象

4. 用户程序

4.1. Zircon 用户程序

4.2. 上下文切换

4.3. 系统调用

5. 信号和等待

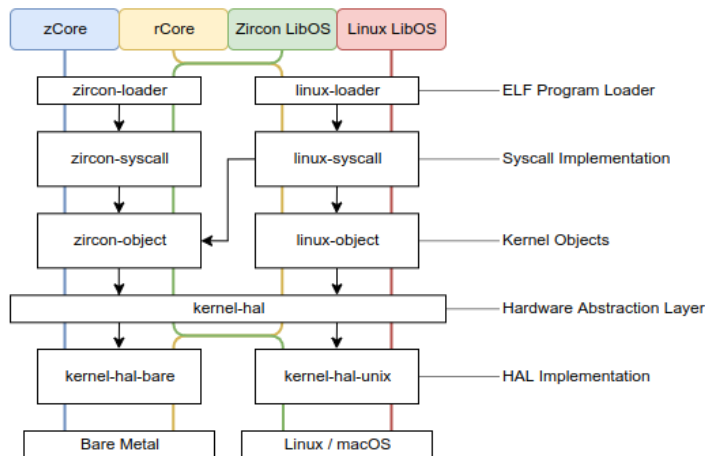


简明 zCore 教程



zCore 的整体结构

zCore 的整体结构/项目设计图如下：



zCore的设计主要有两个出发点：

- 内核对象的封装：将内核对象代码封装为一个库，保证可重用
- 硬件接口的设计：使硬件与内核对象的设计相对独立，只向上提供统一、抽象的API接口

项目设计图如下：项目设计图如下：项目设计图如下

- zCore 接口文档



Crate zcore_loader

Version 0.1.0

All Items

Modules

Crates

kernel_hal

linux_object

linux_syscall

zcore_drivers

zcore_loader

zircon_object

zircon_syscall

Click or press 'S' to search, '?' for more options...



Crate zcore_loader

source · [-]

[-] Linux and Zircon user programs loader and runner.

Modules

linux linux

Run Linux process and manage trap/interrupt/syscall.

zircon

Run Zircon user program (userboot) and manage trap/interrupt/syscall.

zircon

总体介绍 - 自动测试

文档与CI/CD集成测试

- `#![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

总体介绍 - 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
 -
-

提纲

- 总体介绍
 - 各模块实现
 - 初始化与第一个用户程序
 - 两套 `syscall` 接口
 - 内核设计 -- 用户态HAL
 - 内核协程
 - 其他
-

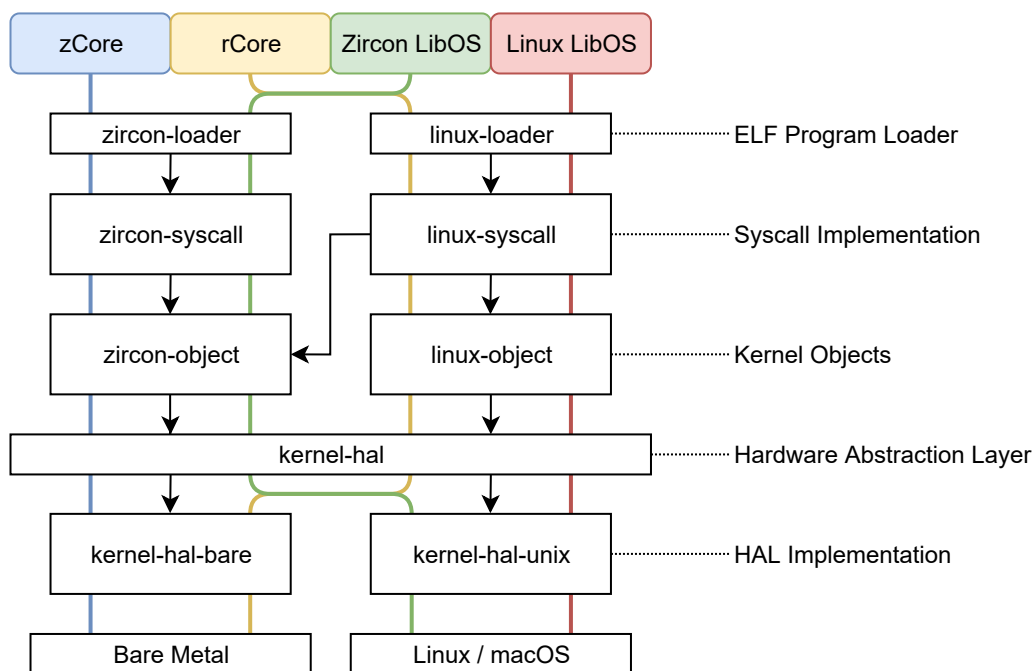
初始化与第一个用户程序

- `zCore`: 调用 `kernel-hal` 接口，完成 内存 / 中断 等的初始化
 - `loader`：加载第一个用户程序并运行
 - [看代码]
-

提纲

- 各模块实现
 - 初始化与第一个用户程序
 - 两套 `syscall` 接口
 - 内核设计 -- 用户态HAL
 - 内核协程
 - 其他
-

两套 syscall 接口



- [看代码：目录结构]

两套 syscall

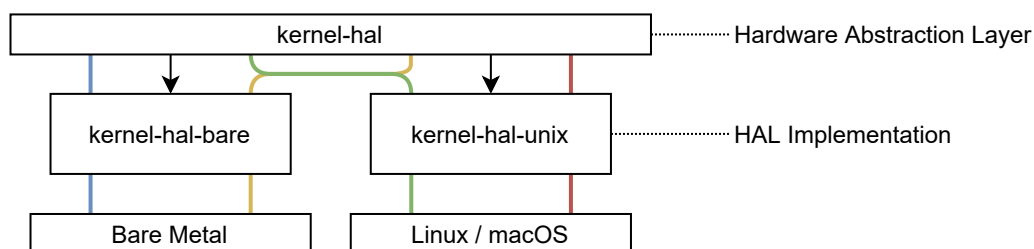
- zircon 文档: <https://fuchsia.dev/fuchsia-src/reference/syscalls>
- zcore tutorial: <https://rcore-os.github.io/zCore-Tutorial/>
- syscall 入口: [trapframe-rs](#) / loader

提纲

- 各模块实现
 - 初始化与第一个用户程序
 - 两套 syscall 接口
 - **内核设计 -- 用户态HAL**
 - 内核协程
 - 其他

内核设计 -- 用户态HAL

HAL 硬件抽象层的设计实现



- Mode: bare-metal vs **libos**
- Arch: x86-64 vs riscv
- Platform: qemu vs d1

内核设计 -- 用户态HAL

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？

内核设计 -- 用户态HAL

方案：用户态模拟内核机制

- 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
- log: `stdin` / `stdout`
- 磁盘：文件

进一步思考：

- 时钟中断？
- 测试驱动 / 测试页表内容？

内核设计 -- 用户态HAL

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}`
- [看代码]

内核设计 -- 用户态HAL

Challenge：用户-内核态切换 🐞

- 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`

提纲

- 各模块实现
 - 初始化与第一个用户程序
 - 两套 syscall 接口
 - 内核设计 -- 用户态HAL
 - **内核协程**
 - 其他
-

内核设计 -- 异步调度

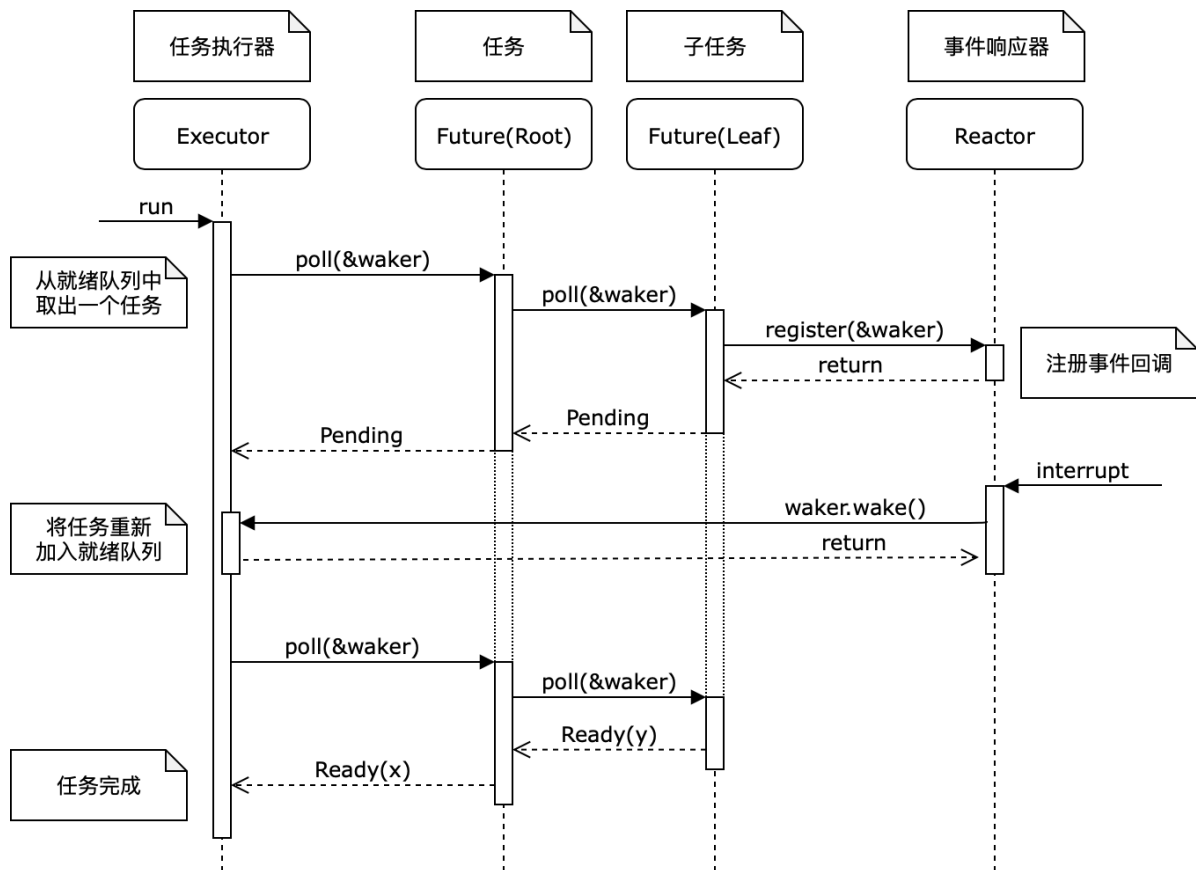
```
1 fn handler(mut stream: TcpStream) -> Result<()> {
2     let mut buf = [0; 1024];
3     let len = stream.read(&mut buf)?; // may block
4     stream.write_all(&buf[0..len])?; // may block
5 }
```

协程化

```
1 // fn handler(...) -> impl Future<Output = Result<()>>
2 async fn handler(mut stream: TcpStream) -> Result<()> {
3     let mut buf = [0; 1024];
4     let len = stream.read(&mut buf).await?;
5     stream.write_all(&buf[0..len]).await?;
6 }
```

内核设计 -- 异步调度

异步调度：底层 --> 中层 --> 上层



内核设计 -- 异步调度

底层：中断/异常事件处理：手动构造 Future

```

1 // 例：在内核对象上等待信号
2 struct waitSignal {...} // 定义一个状态机结构
3 impl Future for waitSignal { // 实现 Future trait 的 poll 函数
4     type Output = Signal;
5     fn poll(self: Pin<&mut Self>, cx: &mut Context)
6     -> Poll<Self::Output>
7     {
8         if self.signal().contains(signal) { // 若目标事件已发生，直接返回 Ready
9             return Poll::Ready(signal);
10        }
11
12        let waker = cx.waker().clone(); // 尚未发生：注册回调函数，当事件发生时唤醒自己
13        self.add_signal_callback(move || waker.wake());
14        Poll::Pending
15    }
16 }
17
18 fn wait_signal(&self, signal: Signal) -> impl Future<Output = Signal> {
19     waitSignal {...} // 返回状态机对象
20 }
  
```


内核设计 -- 异步调度

中层：事件组合处理：用 async-await 组合 Future

```
1  async fn sys_object_wait_signal(...) -> Result {
2      ...
3      let signal = kobject.wait_signal(signal).await;
4      ...
5  }
```

内核设计 -- 异步调度

中层：事件组合处理：用 async-await 组合 Future 用法：用 `select` 组合子实现 超时处理 和 异步取消

```
1  async fn sys_object_wait_signal(..., timeout) -> Result {
2      ...
3      let signal = select! {
4          s = kobject.wait_signal(signal) => s,
5          _ = delay_for(timeout) => return Err(Timeout),
6          _ = cancel_token => return Err(Cancelled),
7      };
8      ...
9  }
```

内核设计 -- 异步调度

上层：异步内核函数：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
- [看代码: [zCore/utils.rs](#)]

提纲

- 各模块实现
 - 初始化与第一个用户程序
 - 两套 syscall 接口
 - 内核设计 -- 用户态HAL
 - 内核协程
 - 其他

其他

- zircon-user: 用户编译 zircon 用户程序
- linux-user ?
- [测试与运行](#)

附录：Zircon 内核对象

Fuchsia 和 Zircon

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

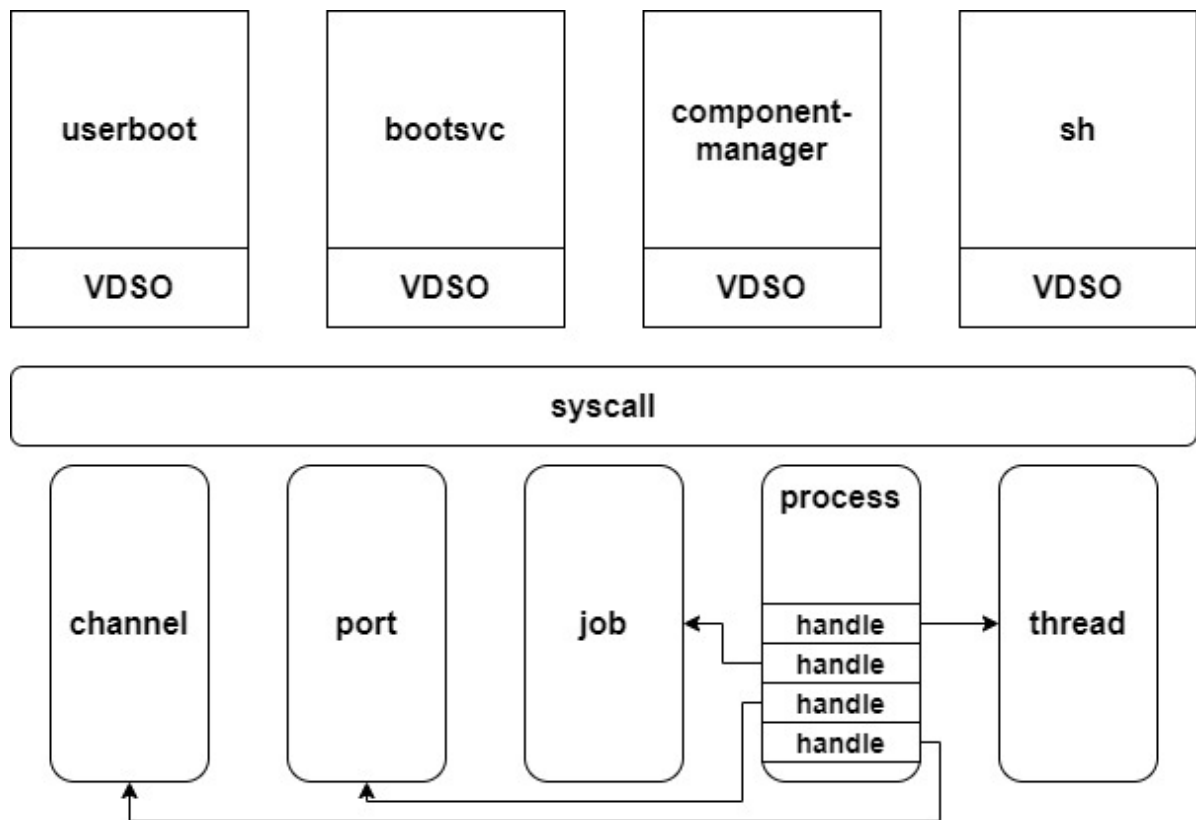
mediates hardware access, implements essential software abstractions over shared resources, and provides a platform for low-level software development.

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 内核对象 -- 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

Zircon 内核对象 -- 用户执行环境

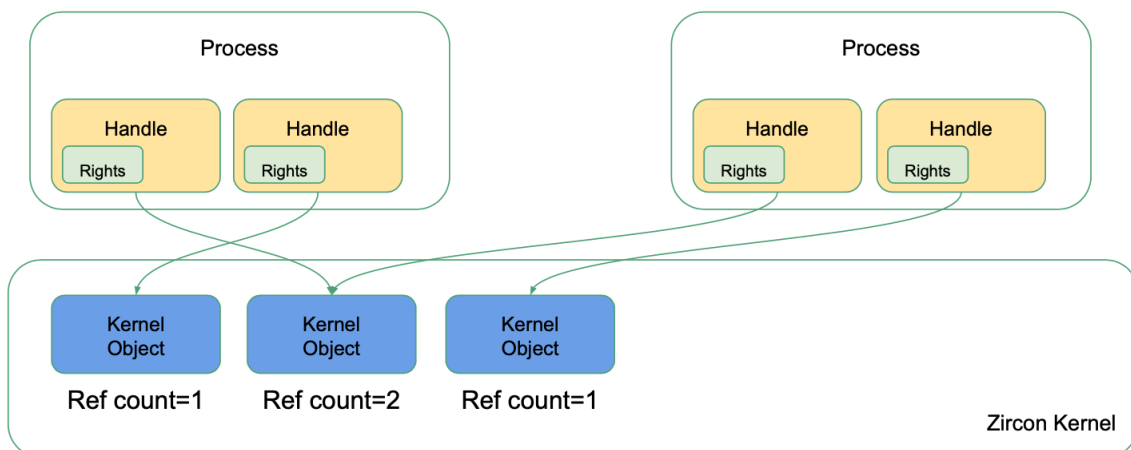


Zircon 内核对象 -- 主要组成

- 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 ...

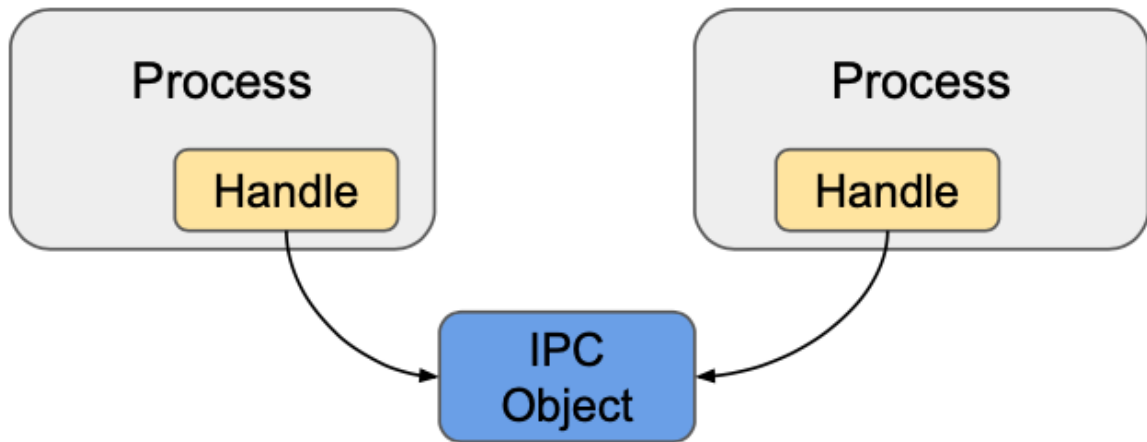
Zircon 内核对象 -- Object

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



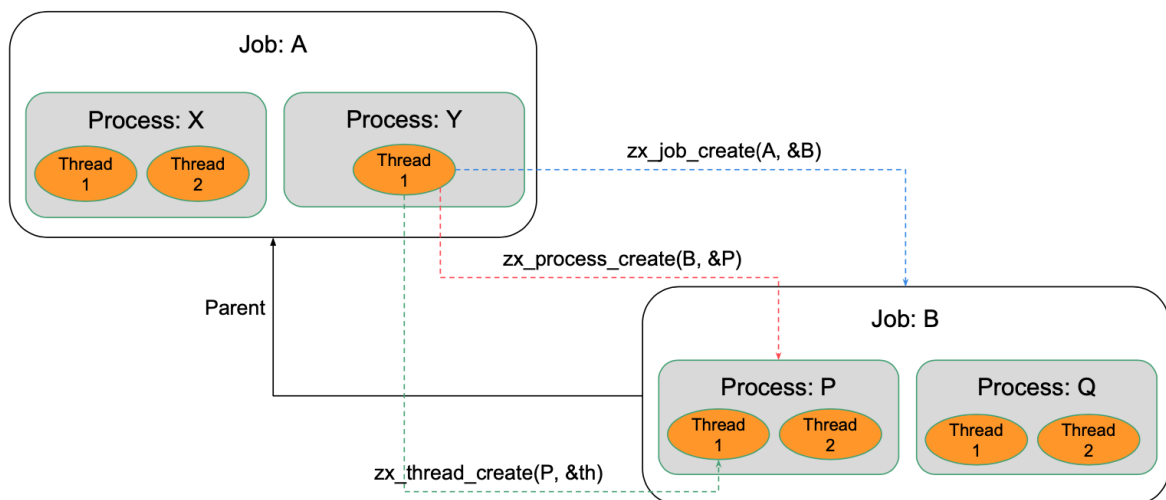
Zircon 内核对象 -- IPC

- 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



Zircon 内核对象 -- 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



Zircon 内核对象 -- MAS

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 <!-- - 一段连续的虚拟内存, 可以用于在进程之间、内核和用户空间之间共享内存
 - 在内核中被维护为类似线段树的数据结构, 支持从一个VMO中创建新的VMO -->
- VMAR: Virtual Memory Address Region
 - 代表一个进程的虚拟地址空间

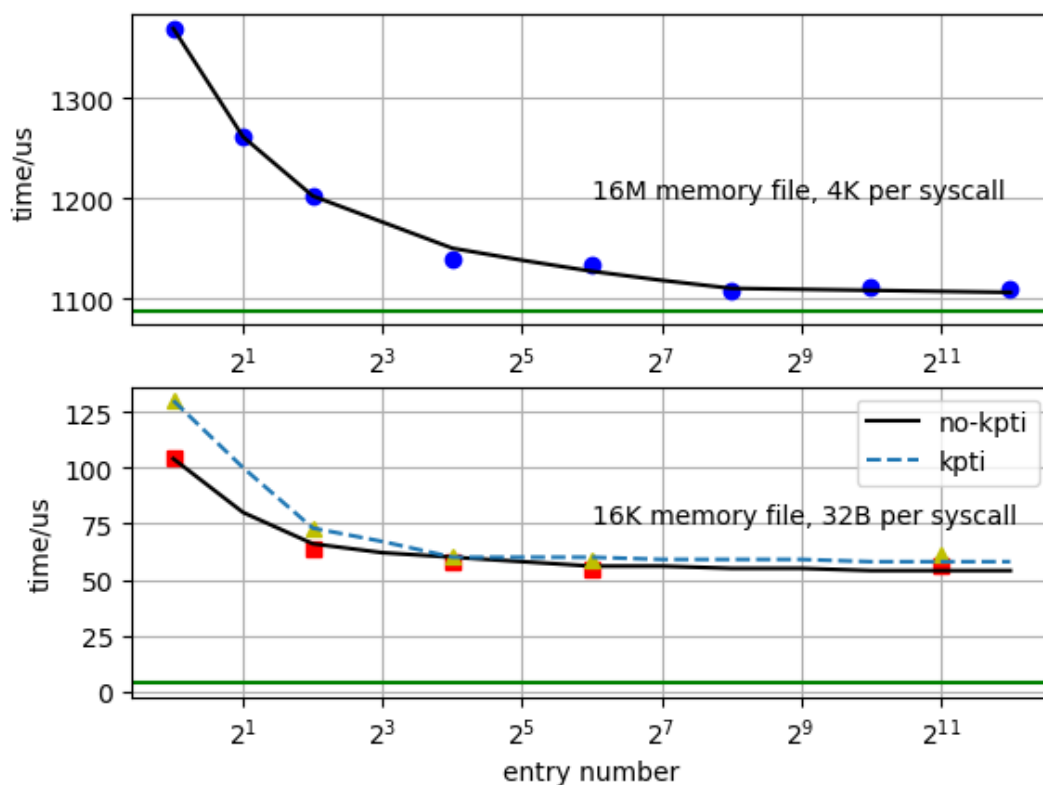
- 树状结构
- Pager: 用户态分页机制

Zircon 内核对象 -- Signaling /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)

附录：异步 syscall 简单测试



- allwinner nezha riscv

异步 syscall 的感知

- 如何感知异步 syscall ?

	CPU占用	Latency	Locality	备注
syscall	👍	👍	😞	
polling	😞	👍	👍	适合高速设备
free-polling	👍	😞	👍	
uintr	👍	👍	👍	需要硬件支持

zCore: free-polling + uintr

可抢占协程

- 异步执行 latency 的累加
 - 用户提交 => 内核感知, 内核提交 => 用户感知：uintr or polling
 - interrupt => 内核响应：优先级，可抢占协程
- 可退化的协程 = 可抢占的协程
 - 在中断 handler 里新建一个线程
 - 协程的线程间迁移

协程的调度

- 协程的类型
 - 内核协程：处理异步 syscall 的协程，polling 协程等
 - 用户协程：用户线程封装而成
- 核专用化
 - 内核核：内核协程，切换次数多，切换开销小
 - 用户核：用户协程，切换考校大，切换次数少
 - 如果更进一步：磁盘 IO 核，网络核
- 问题：load balance

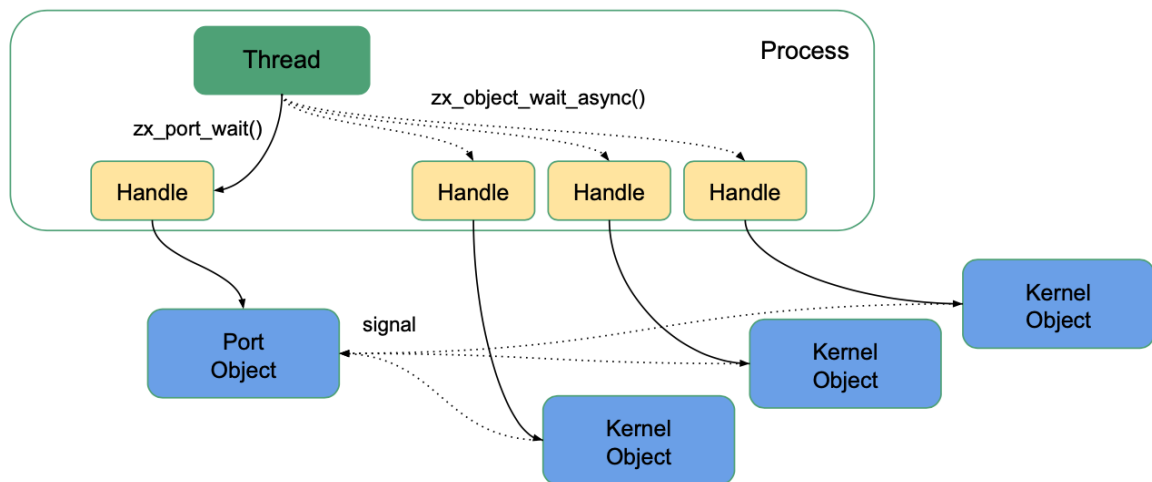
协程切换开销

- thread switch => function return and call
- lmbench lat-ctx

```
1 Context switching - times in microseconds - smaller is better
2 -----
3 Host          OS  2p/0K 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K
4 -----
5 proc switch      0.3900 0.8800 0.8900 1.9200 2.2700 2.11000 2.25000
6 coroutine switch 0.0330 0.0390 0.0390 0.0600 0.0600 0.06800 0.06800
```

Zircon 内核对象 -- Signaling /Waiting

每个 Object 有 32 个信号位，用户程序可以阻塞等待。



谢谢