# zCore 项目进展汇报

*Github.com/rcore-os/zCore*

萧络元

清华大学

2022.07.05

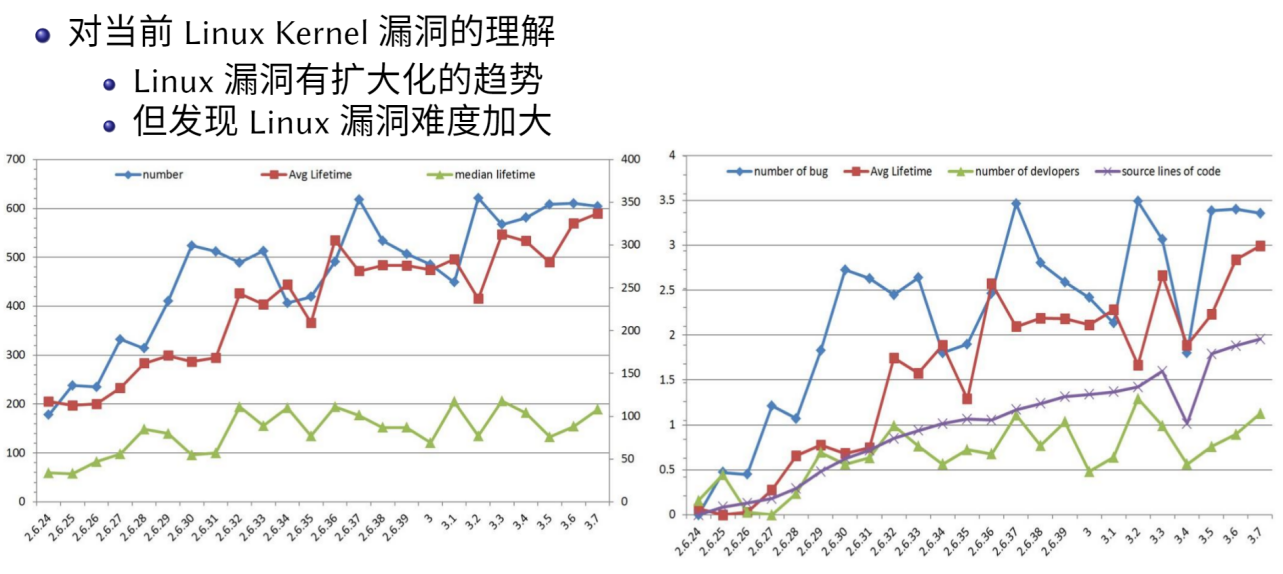
### 提纲

* **相关背景**
* 总体介绍
* 内核设计
  + 用户态HAL
  + 内核异步调度
  + Zircon 内核对象

### 

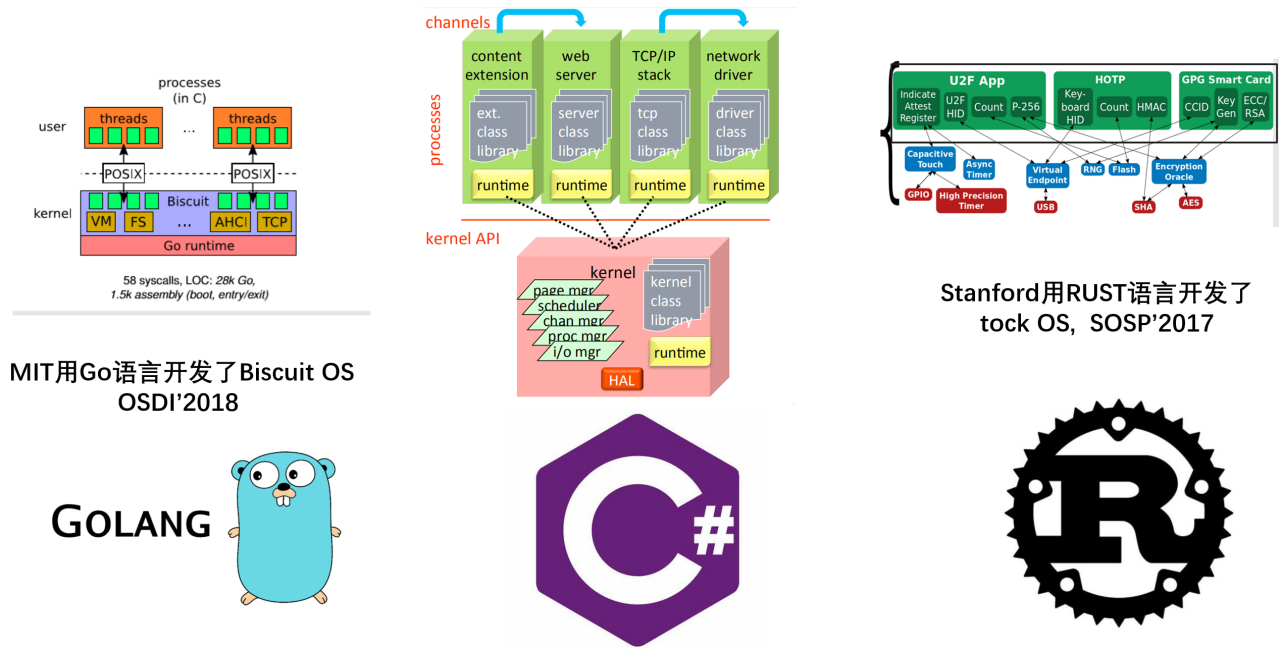
### 相关背景

* 操作系统安全问题

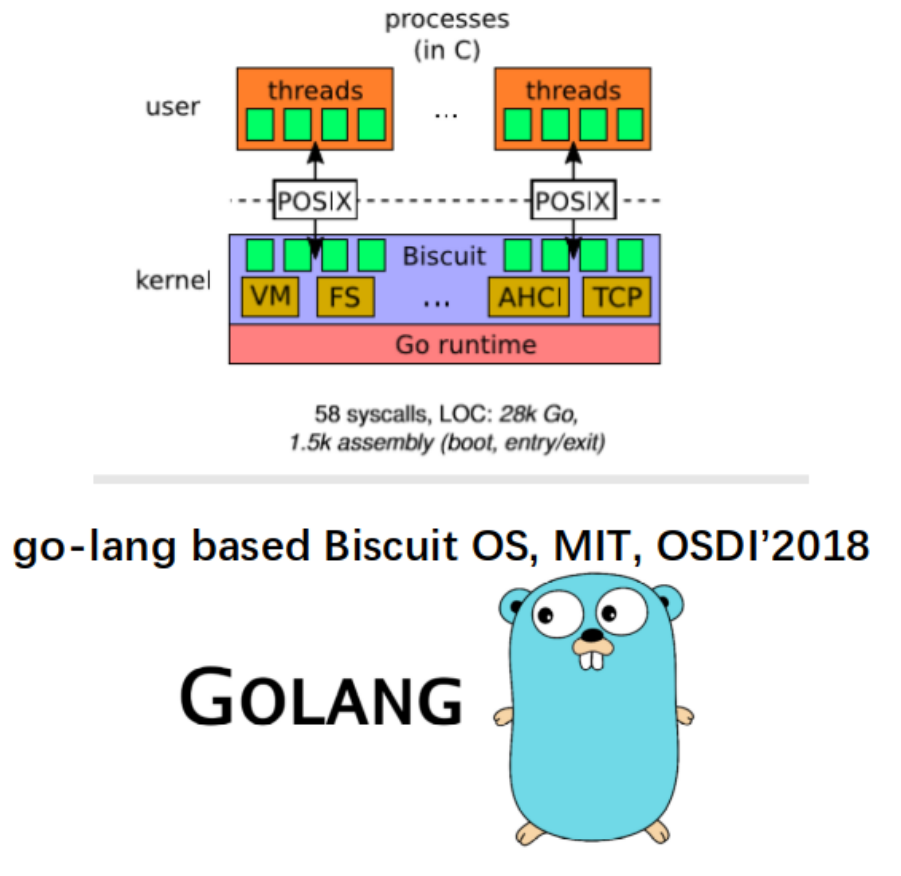


### 相关背景

* 基于新高级语言的OS



### 相关背景

* 基于新高级语言的OS  
  

### 提纲

* 相关背景
* **总体介绍**
* 内核设计
  + zCore内核对象
  + 用户态HAL
  + 内核异步调度

### 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 - 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
* ……

### 总体介绍 - 整体架构

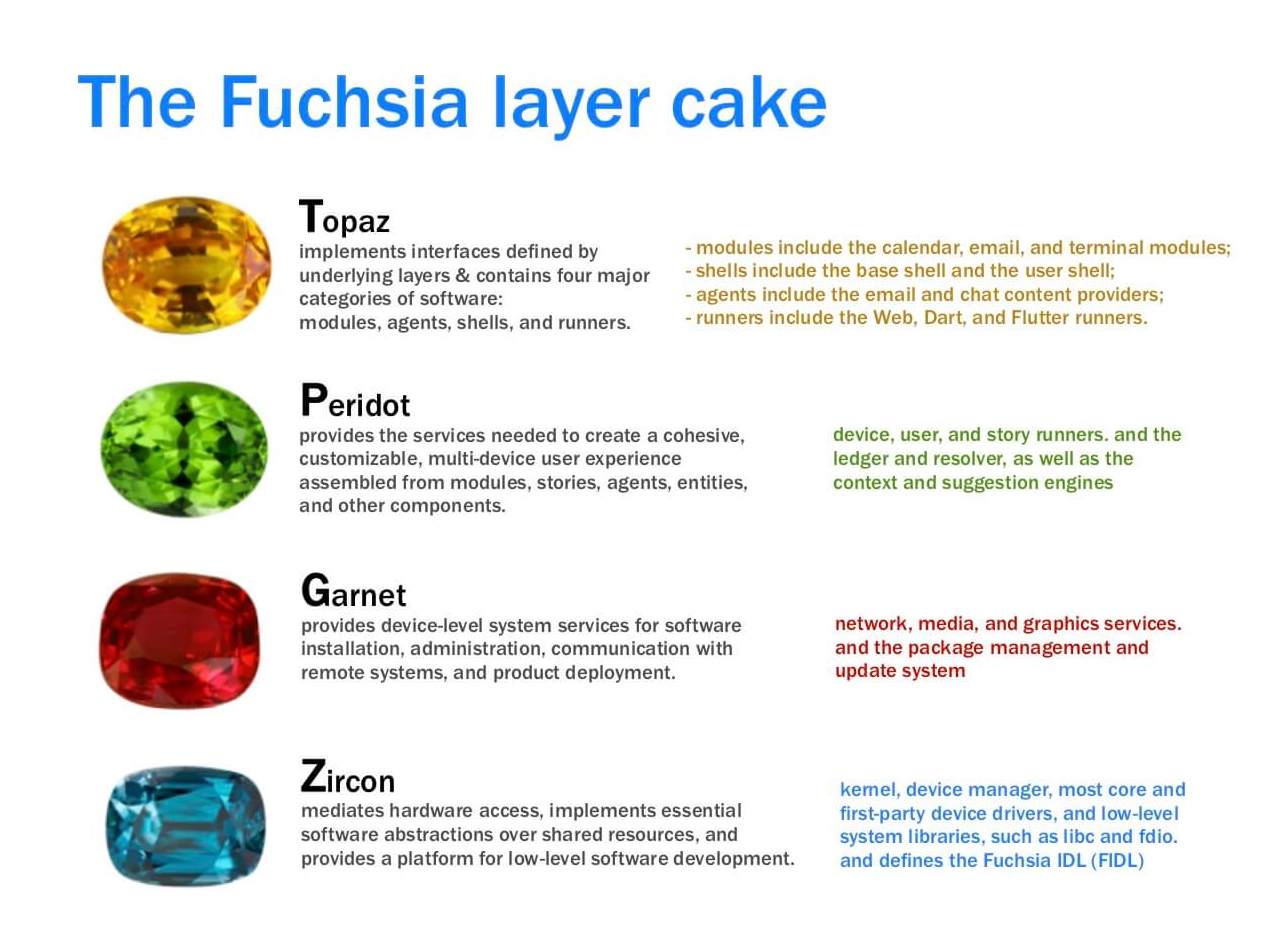


### 提纲

* 相关背景
* 总体介绍
* **内核设计**
  + zCore 内核对象
  + 用户态HAL
  + 内核异步调度

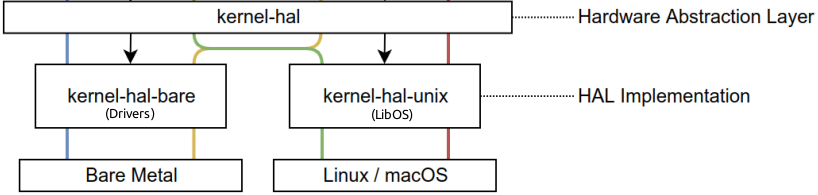
### zCore 内核对象

Fuchsia 和 Zircon



### 内核设计 -- 用户态HAL

HAL 硬件抽象层的设计实现



### 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

### 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}

### 内核设计 -- 用户态HAL

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

### 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

### 内核设计 -- 用户态HAL

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

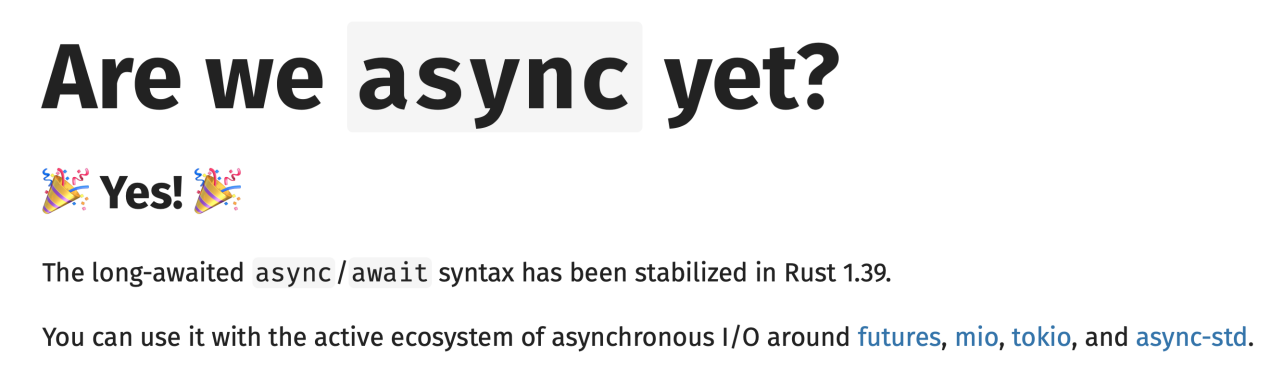
Library OS，User-mode Linux

### 提纲

* 相关背景
* 总体介绍
* **内核设计**
  + 用户态HAL
  + **内核异步调度**

### 内核设计 -- 异步调度

async Rust 原理和设计模式



### 内核设计 -- 异步调度

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

### 内核设计 -- 异步调度

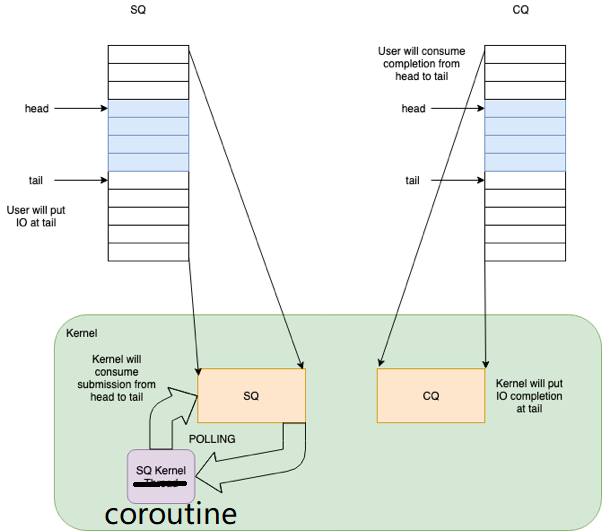
async-await：用同步风格编写异步代码

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

### 内核设计 -- 异步调度

* 异步 syscall 与内核协程调度
* Linux 5引入的一套新的syscall接口，用于支持异步IO: io\_uring 与异步 syscall
* 异步 syscall测试



### 内核设计 -- 异步调度

zCore: free-polling + uintr

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | CPU | Latency | Locality | 备注 |
| syscall | Y | Y | X |  |
| polling | X | Y | Y | 高速设备 |
| free-polling | Y | X | Y |  |
| uintr | Y | Y | Y | 硬件支持 |

### 内核设计 -- 异步调度

可抢占协程

* 异步执行 latency 的累加
  + 用户提交 => 内核识别, 内核提交 => 用户识别
    - uintr or polling
  + interrupt => 内核响应
    - 优先级，可抢占协程
* 协程不可抢占 => 重新回到线程
  + 在中断 handler 里新建一个线程 （线程池）
  + 协程的线程间迁移

### 内核设计 -- 异步调度

* 协程的类型
  + 内核协程
  + 用户协程
* CPU Core专用化
  + 内核Core：运行内核协程，切换开销小
  + 用户Core：运行用户协程，切换开销大
  + 如果更进一步：IO Core，Net Core
* 问题：load balance

### 内核设计 -- 异步调度

协程切换开销

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

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

* 协程的内存开销 $\approx$ Max{ 同时存活的状态之和 }

### 内核设计 -- 异步调度

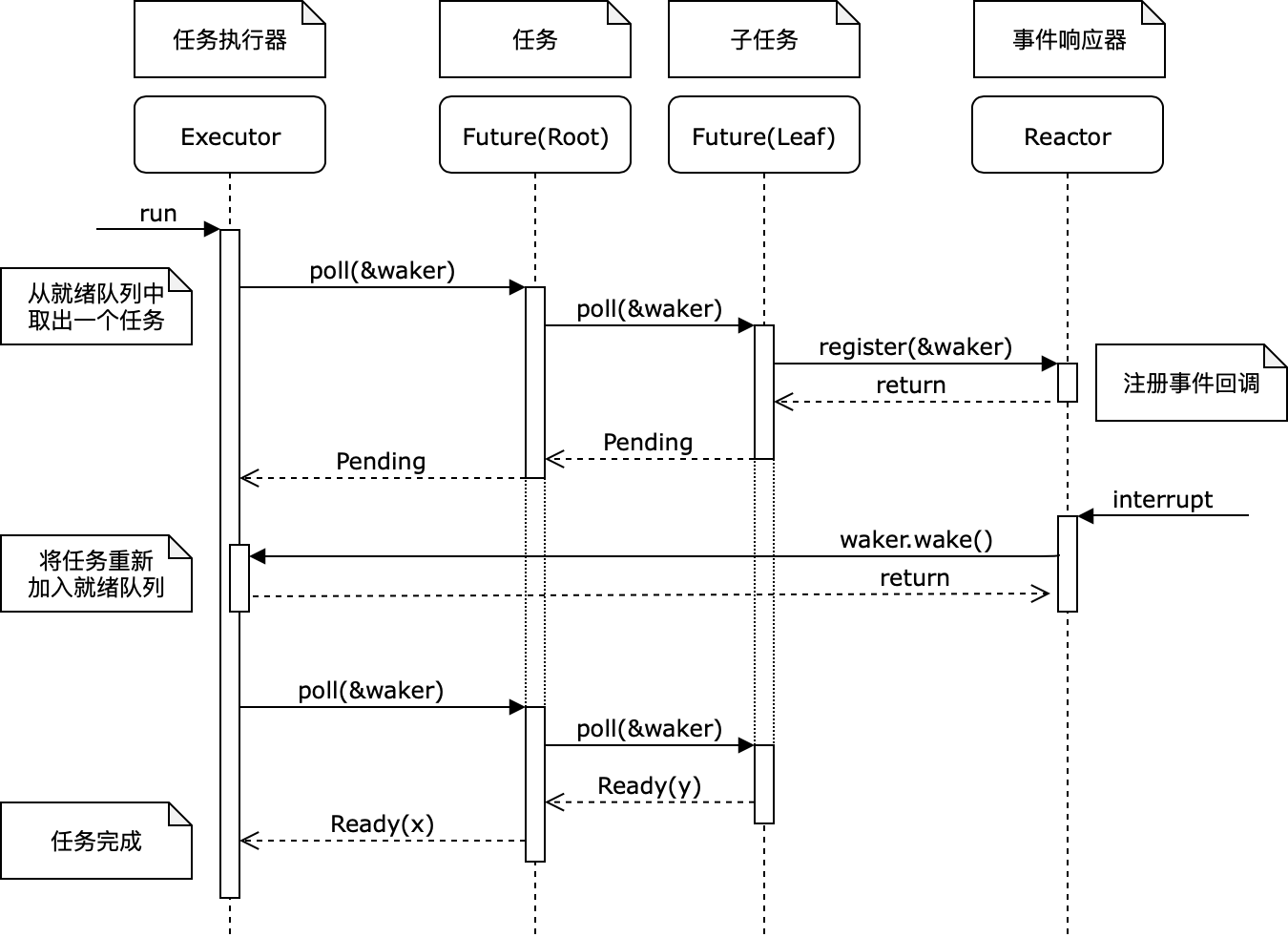
fn handler(mut stream: TcpStream) -> Result<()> {  
 let mut buf = [0; 1024];  
 let len = stream.read(&mut buf)?; // may block  
 stream.write\_all(&buf[0..len]))?; // may block  
}

协程化

// 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?;  
}

### 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

// 例：在内核对象上等待信号  
fn wait\_signal(&self, signal: Signal) -> WaitSignal {  
  
 struct WaitSignal {...} // 定义一个状态机结构  
 impl Future for WaitSignal { // 实现 Future trait 的 poll 函数  
 type Output = Signal;  
 fn poll(self: Pin<&mut Self>, cx: &mut Context)  
 -> Poll<Self::Output>  
 {  
 if self.signal().contains(signal) { // 若目标事件已发生，直接返回 Ready  
 return Poll::Ready(signal);  
 }  
   
 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  
 self.add\_signal\_callback(move || waker.wake());  
 Poll::Pending  
 }  
 }  
 WaitSignal {...} // 返回状态机对象  
}

### 内核设计 -- 异步调度

中层：事件组合处理：用 async-await 组合 Future

async fn sys\_object\_wait\_signal(...) -> Result {  
 ...  
 let signal = kobject.wait\_signal(signal).await;  
 ...  
}

### 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

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),  
 };  
 ...  
}

### 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
* 未来：期望嵌入式社区的 async-nostd？

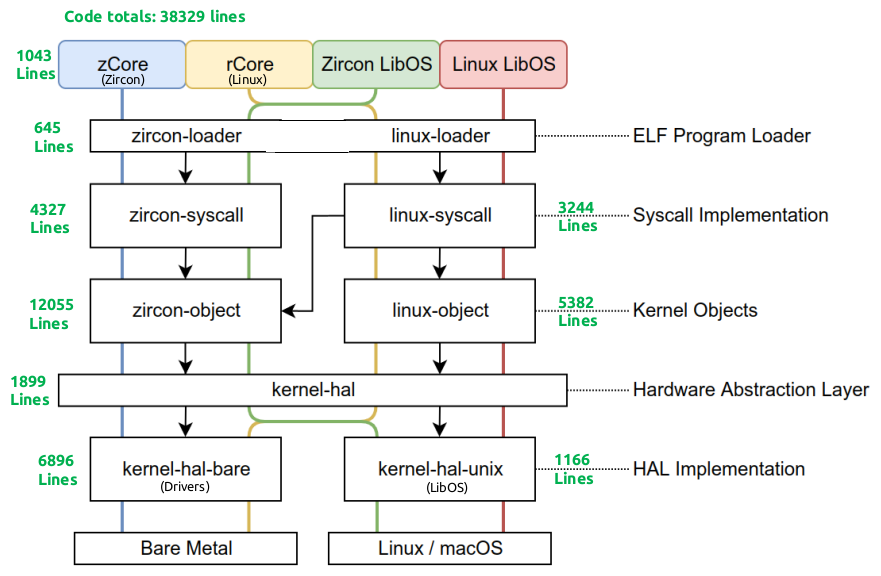
### 内核设计 -- 异步调度

挑战：进出用户态问题

async 要求保持内核上下文（即内核栈）

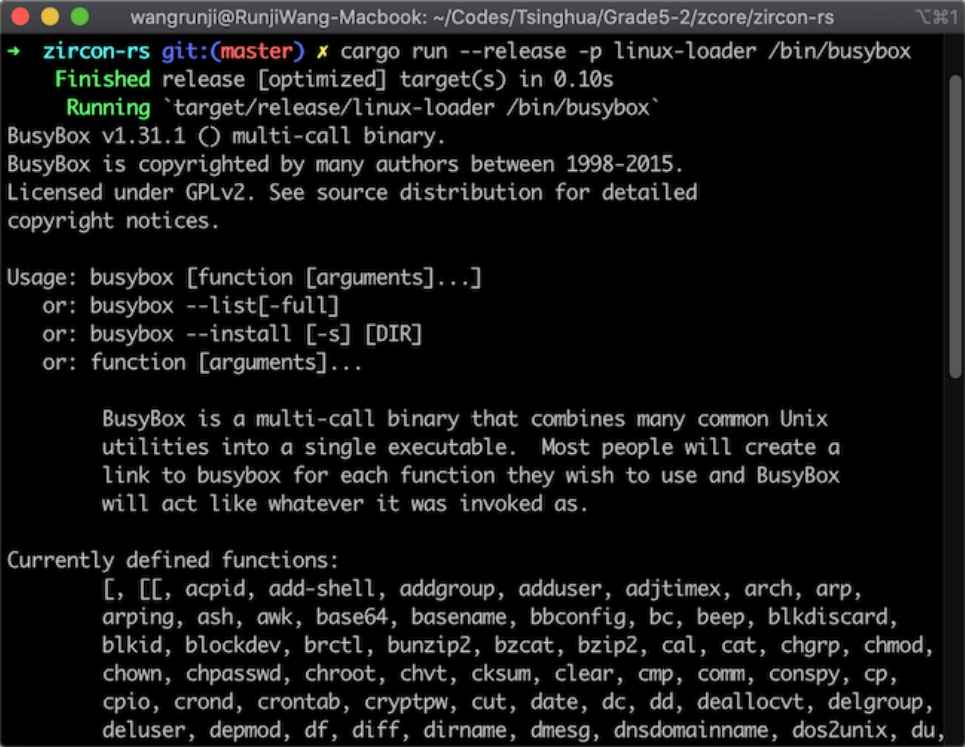
* 传统 OS：User调用Kernel，平时内核栈清空
* zCore：Kernel调用User，保留内核上下文

### zCore- 项目规模

目前大约接近4万行的代码量  


### 

### 总体介绍 - 运行展示

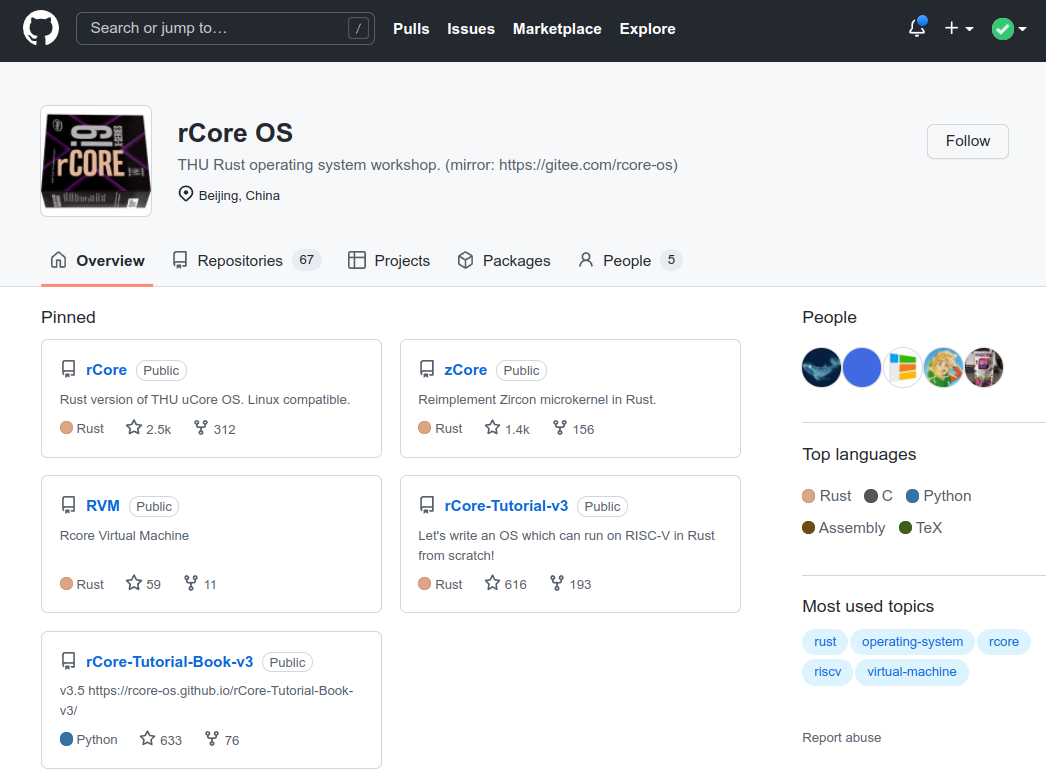
zCore: Linux LibOS on macOS  


### 总体介绍 - 开发社区

zCore项目属于清华大学陈渝老师的操作系统教学社区rCore OS

有着众多的操作系统爱好者贡献代码

其中zCore拥有一千多颗Star， rCore有两千多颗Star

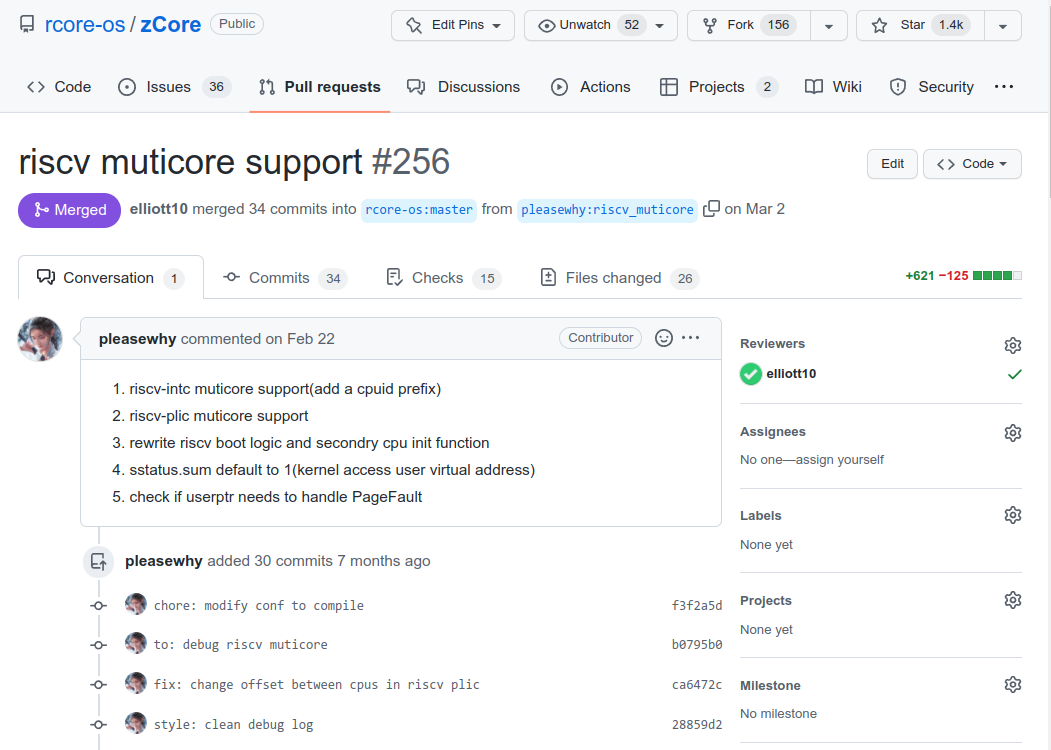


### 总体介绍 - 开发社区

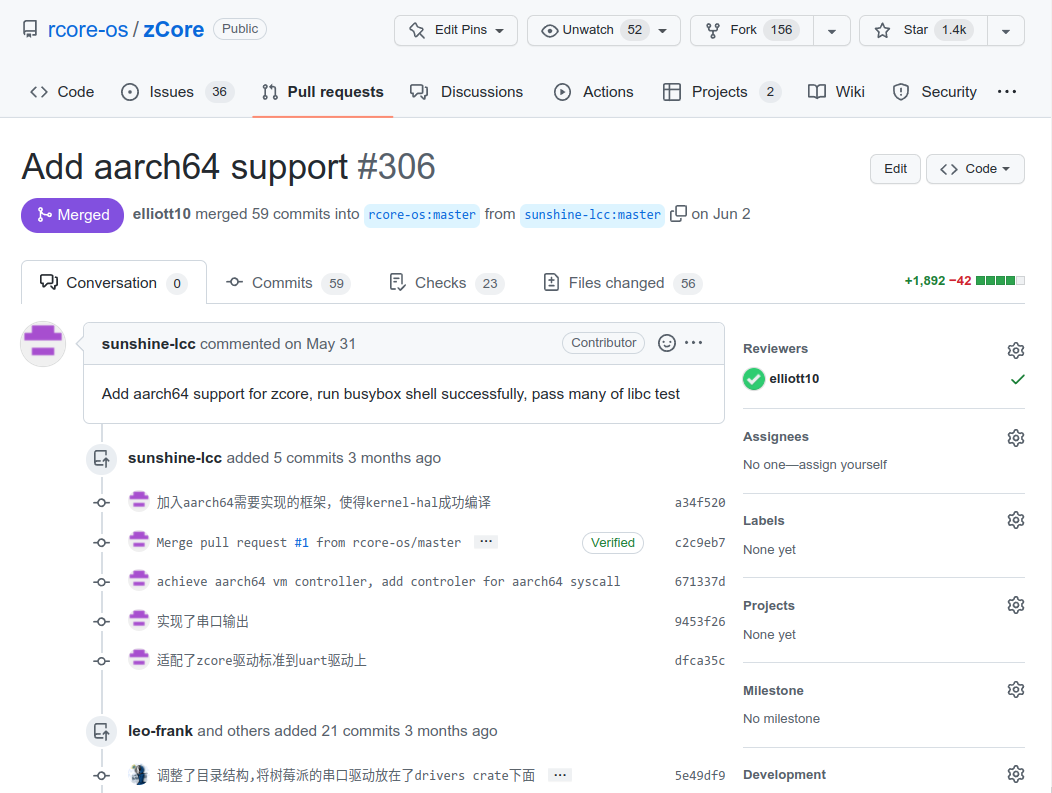
**开发社区活跃**

* 持续适配功能复杂的应用程序，如图像识别的opencv库，二维码识别的C++程序aruco；
* 持续修复发现的问题，如网络协议栈出现的时钟Bug；
* 持续添加新功能，如串口驱动，PCI扫描，网卡驱动；
* 新增多核的支持；
* 启动引导的持续优化，如设备树的轻量化解析；
* 持续适配新的硬件平台，如全志D1, Sifive U740

今年二月份，爱好者添加了RISCV多核的支持；



如前几月爱好者新增加了ARM 64架构的支持，提交了PR



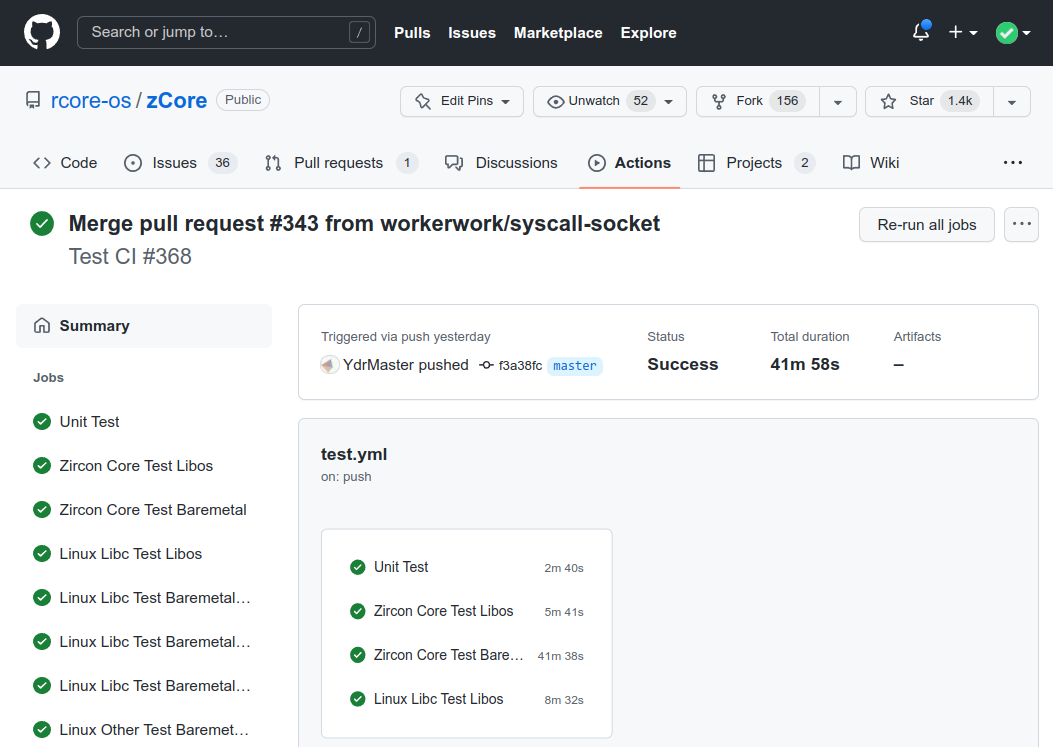
### 总体介绍 - 自动测试

文档与CI/CD集成测试

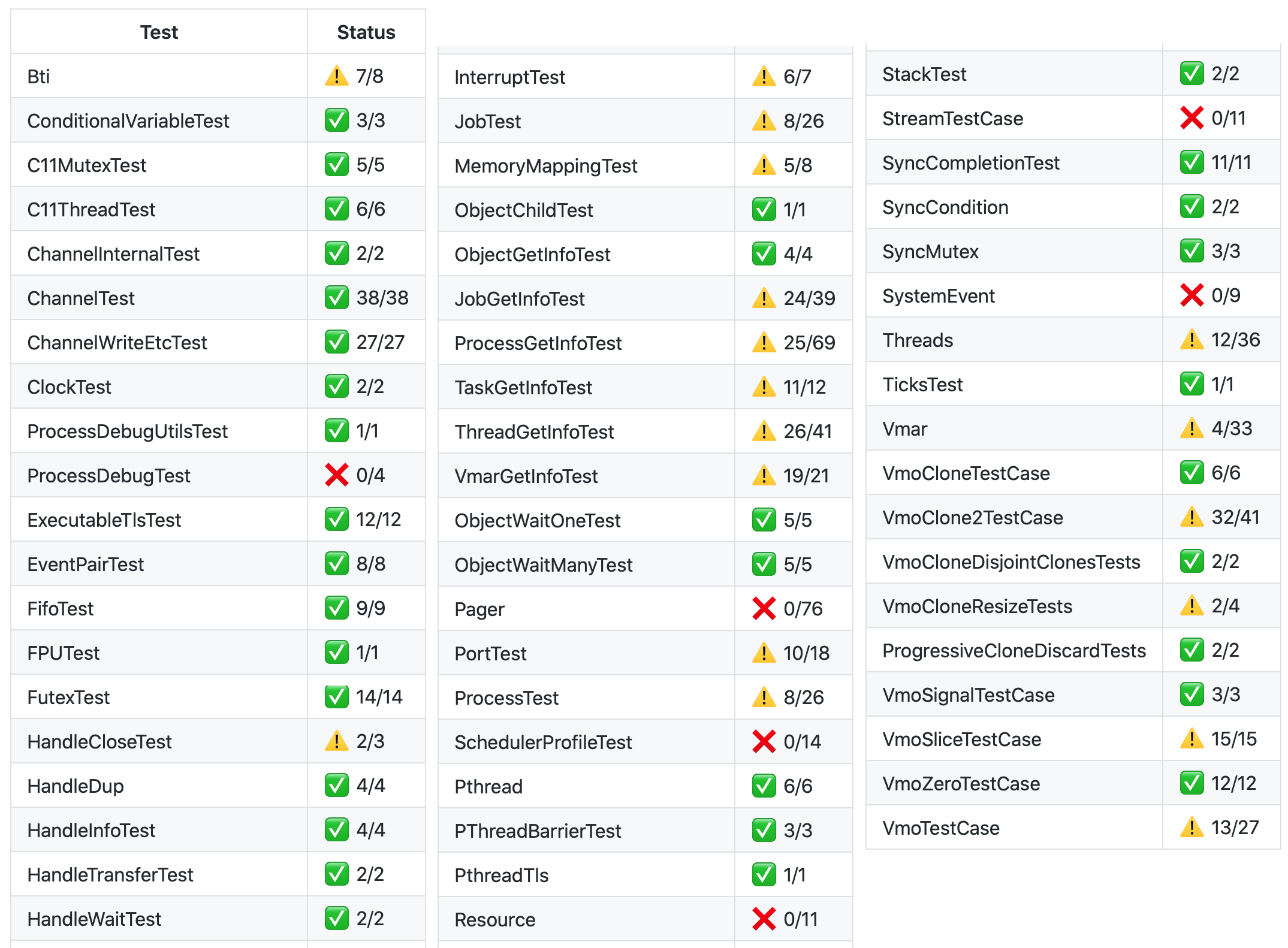
* #![deny(warnings)]：警告报错
* cargo fmt && clippy：检查代码格式和风格
* cargo build：保证所有平台编译通过
* cargo test：用户态单元测试，报告测试覆盖率
* core-test：内核态集成测试，维护通过测例列表
* （TODO）cargo bench：性能测试

上述测试全部通过才允许合入 master

zCore项目社区开发者提交PR，自动触发Github CI测试



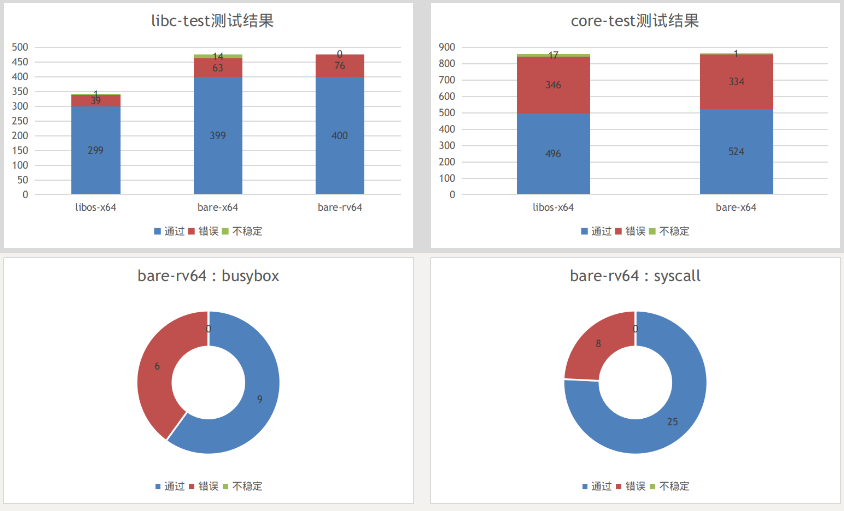
### 总体介绍 - 完成度

zCore完成的Zircon 官方系统调用测试集  


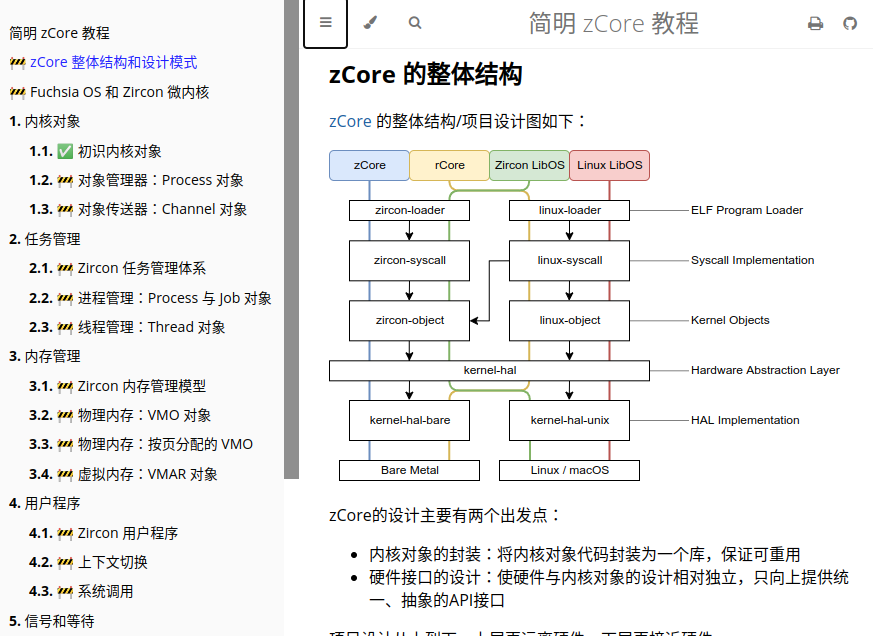
### 总体介绍 - 完成度

zCore完成的Zircon 官方系统调用测试集

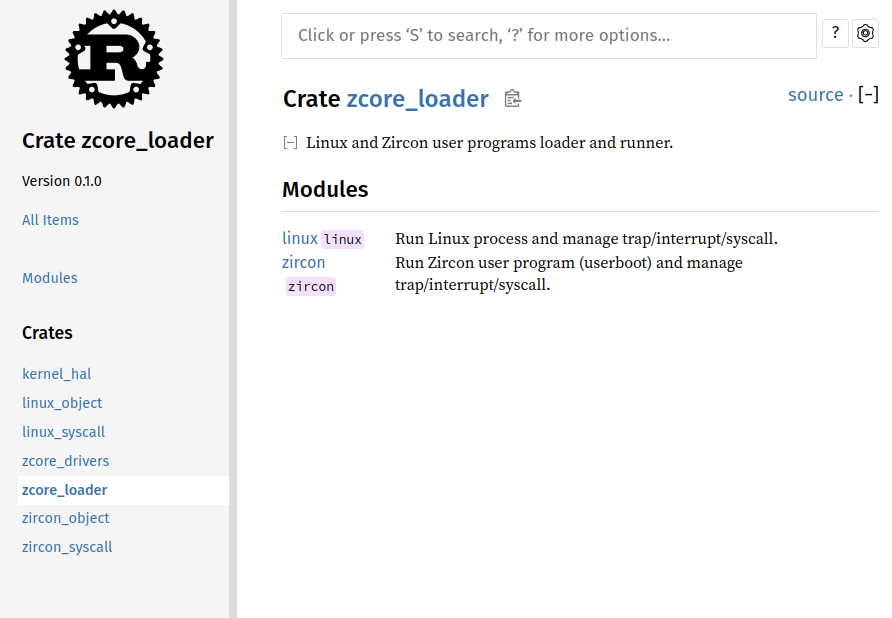
### 总体介绍 - 对系统调用和标准库的支持情况



### 总体介绍 - zCore架构设计文档

zCore 架构设计文档  


### 总体介绍 - zCore系统调用文档

zCore 系统调用文档  


# 谢谢

*Github.com/rcore-os/zCore*