异步编程、Rust语言和异步操作系统

1. 异步编程
2. Rust语言的并发和异步支持
3. 异步操作系统

向勇 清华大学计算机系

20210927

提纲

1. 异步编程
2. Rust语言的并发和异步支持
3. 异步操作系统

1. 异步编程

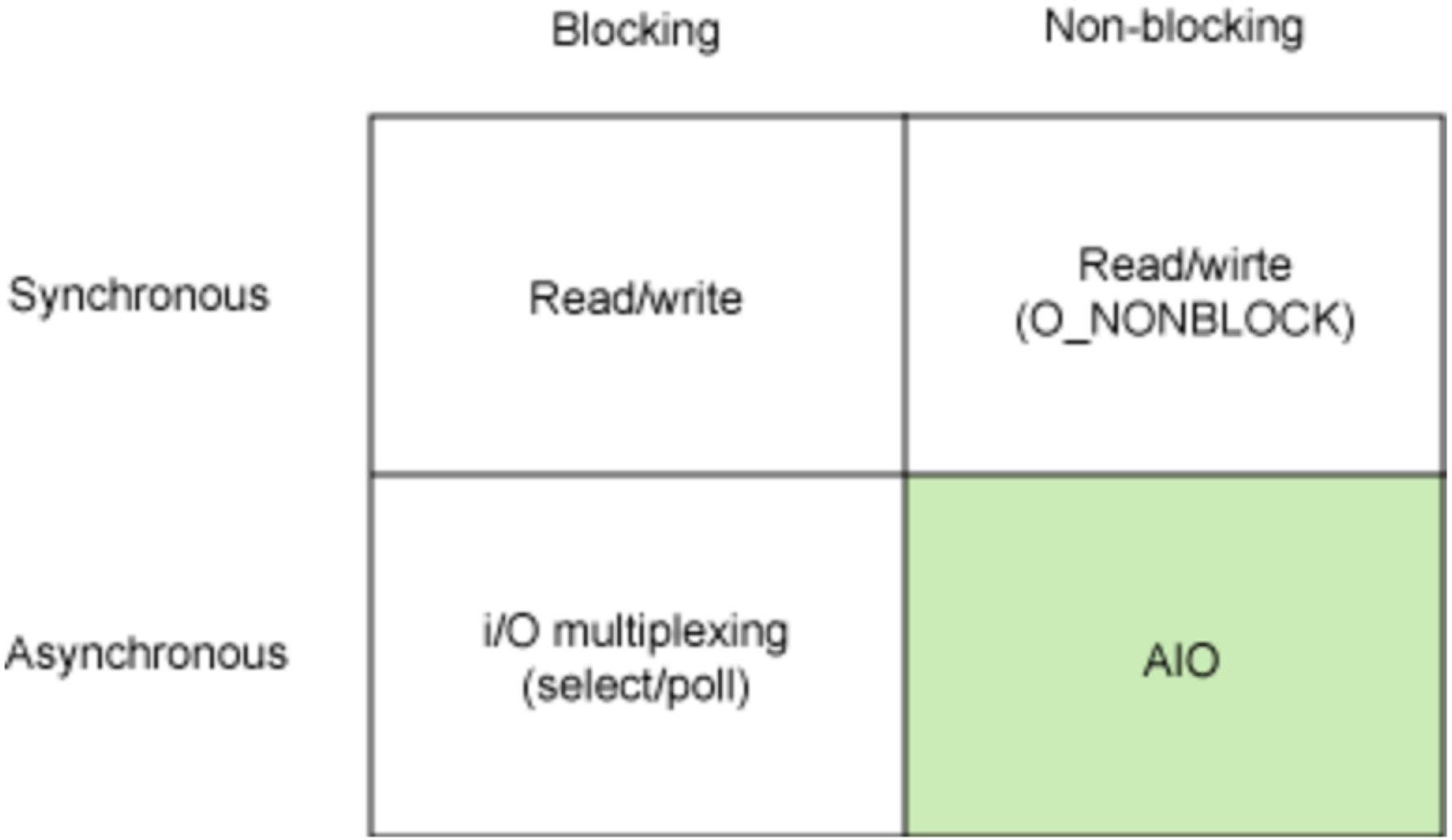
1.1 基本概念和原理

1.2 OS Aproach

1.3 Programming Aproach

1.1 基本概念和原理

基本 Linux IO 模型



阻塞式IO模型

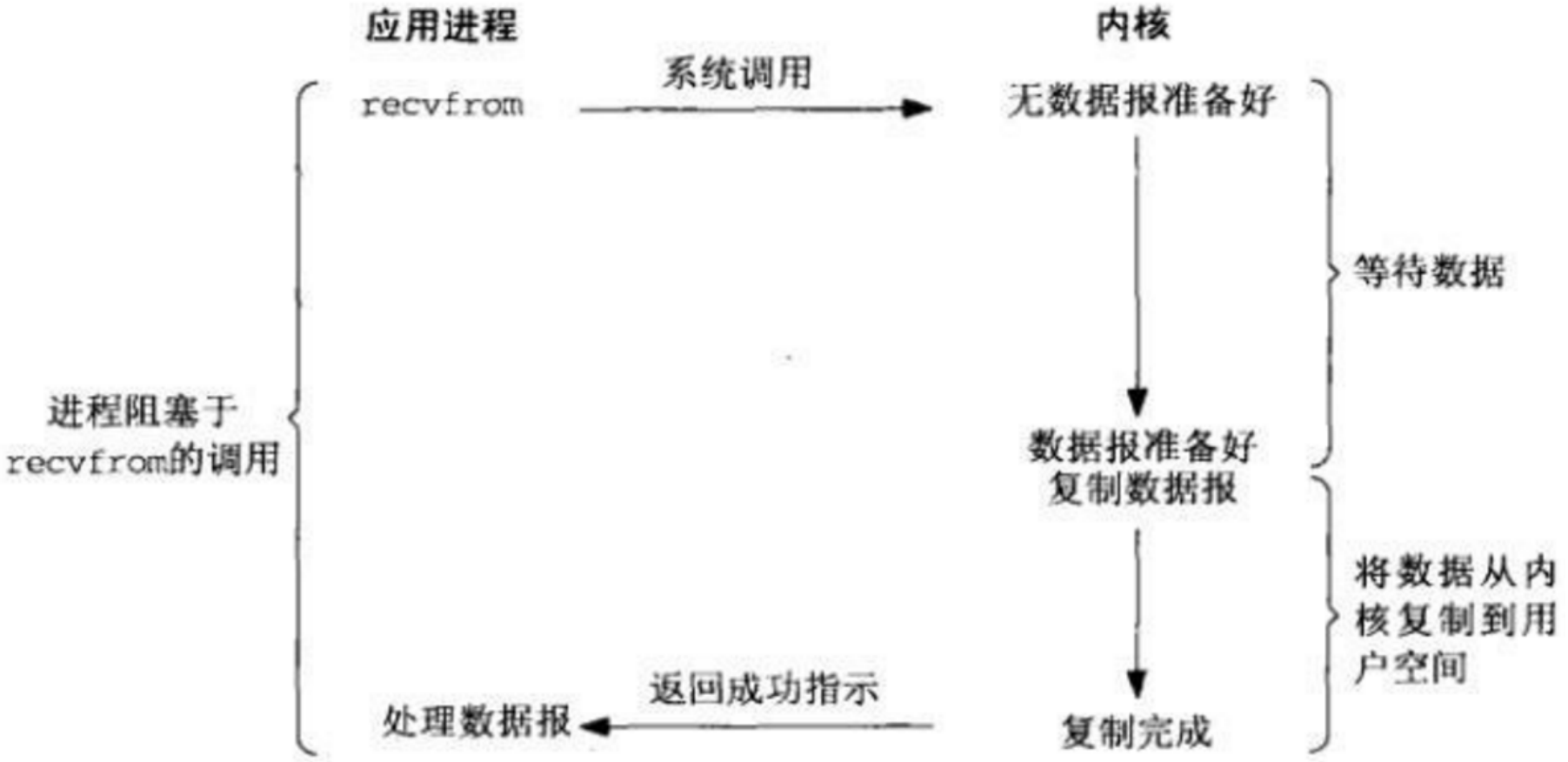


图6-1　阻塞式I/O模型

非阻塞式IO模型

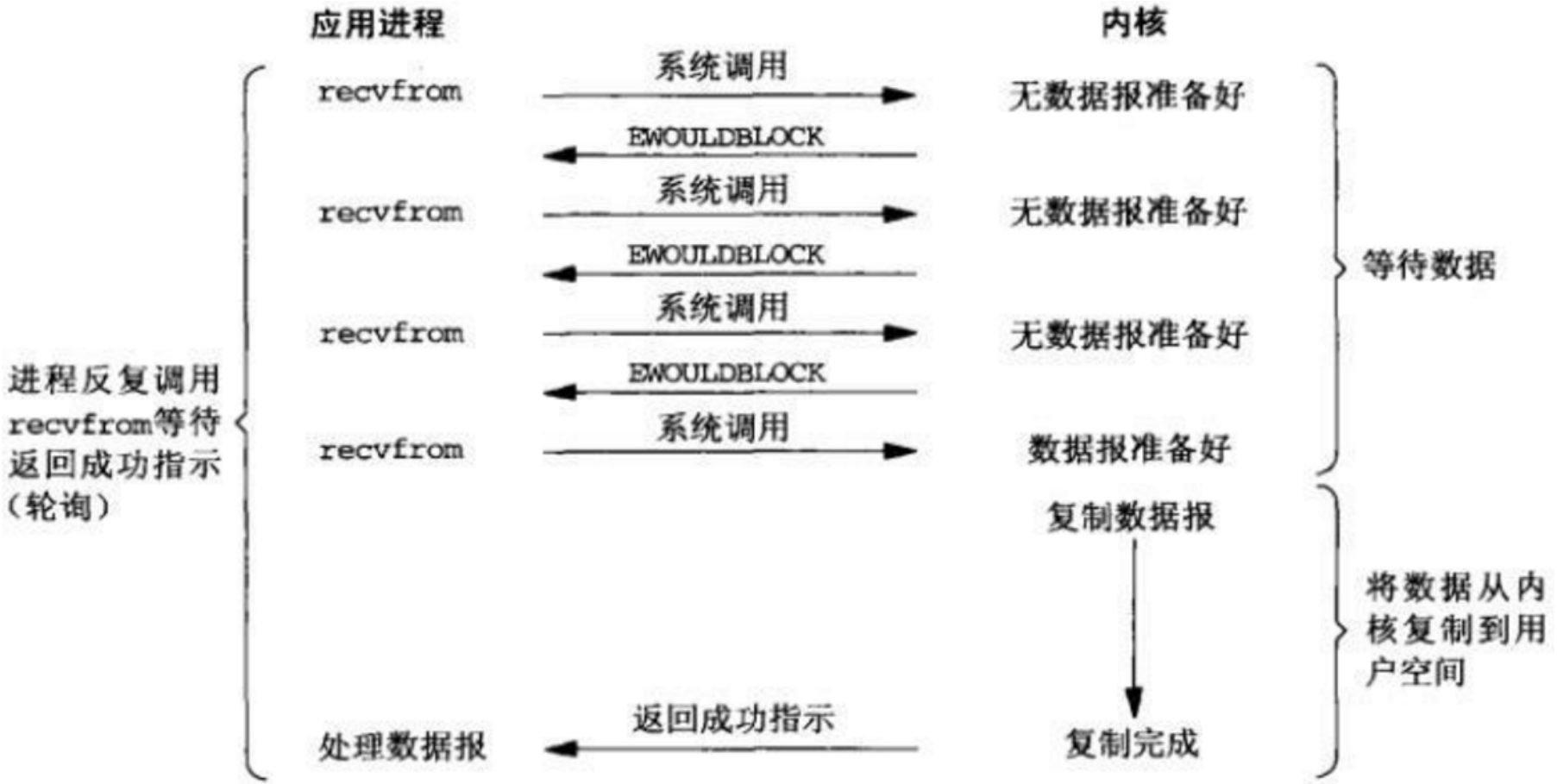


图6-2　非阻塞式I/O模型

IO复用模型

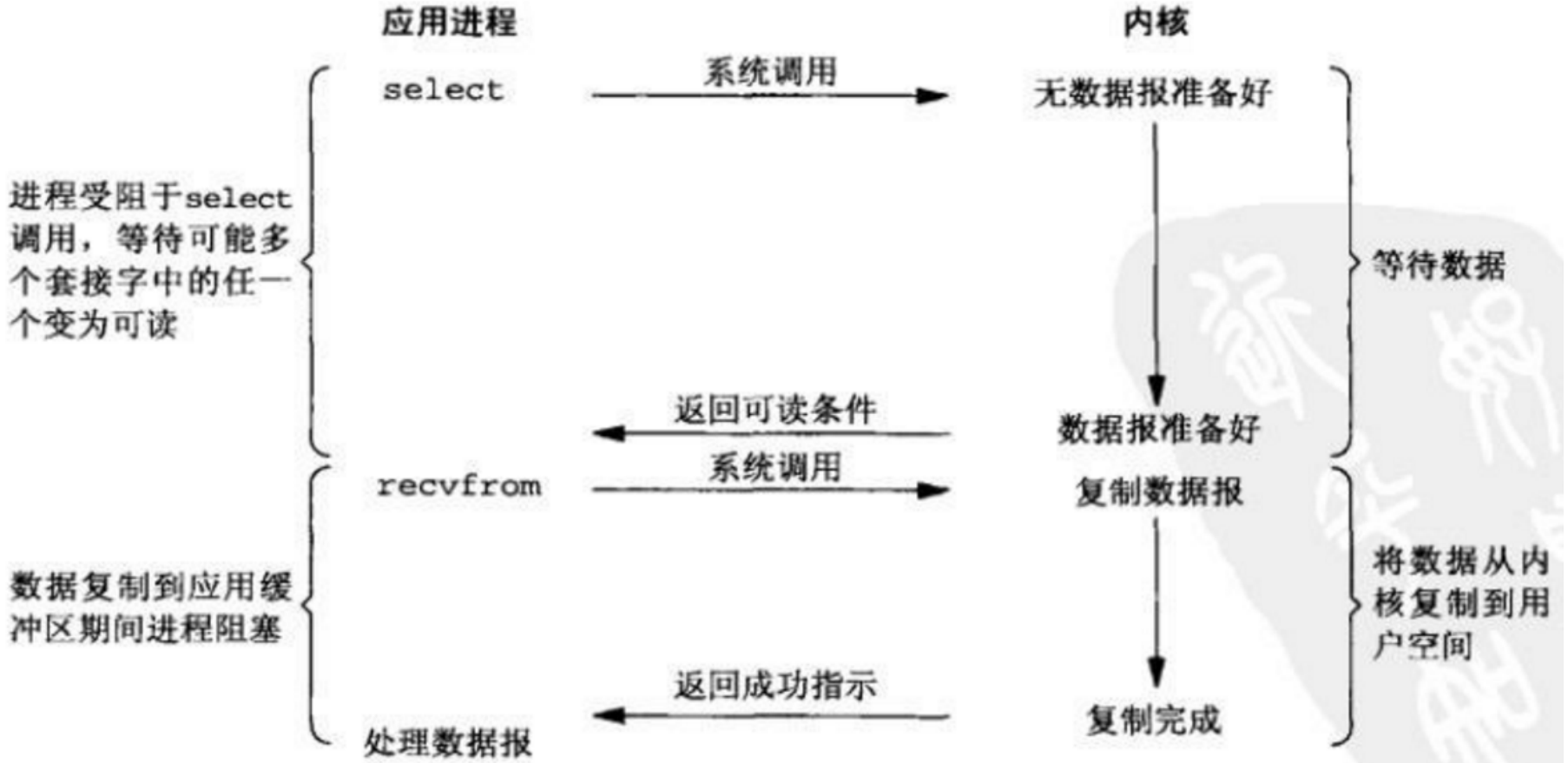


图6-3　I/O复用模型

信号驱动IO模型

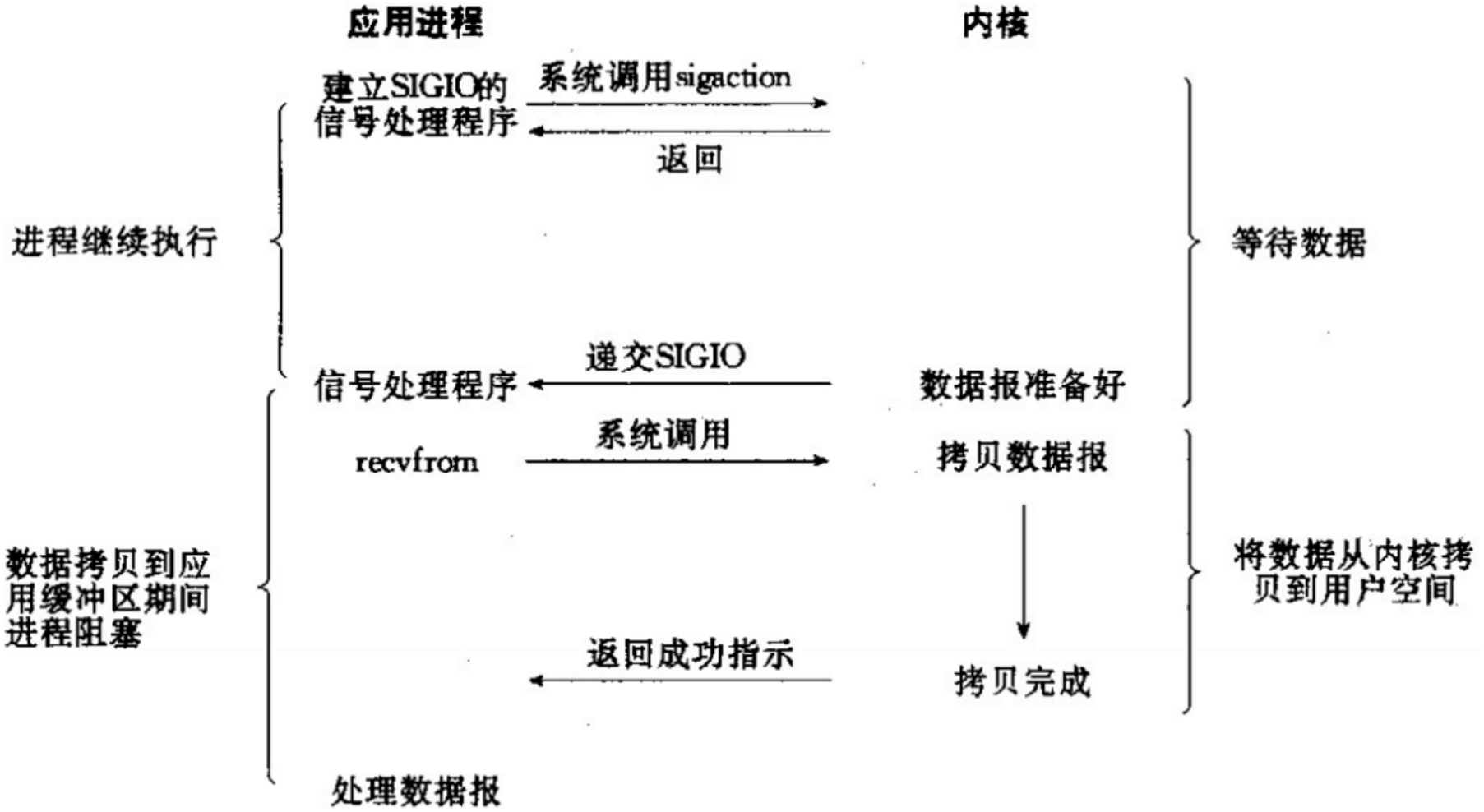


图 6.4　信号驱动 I/O 模型

异步IO模型

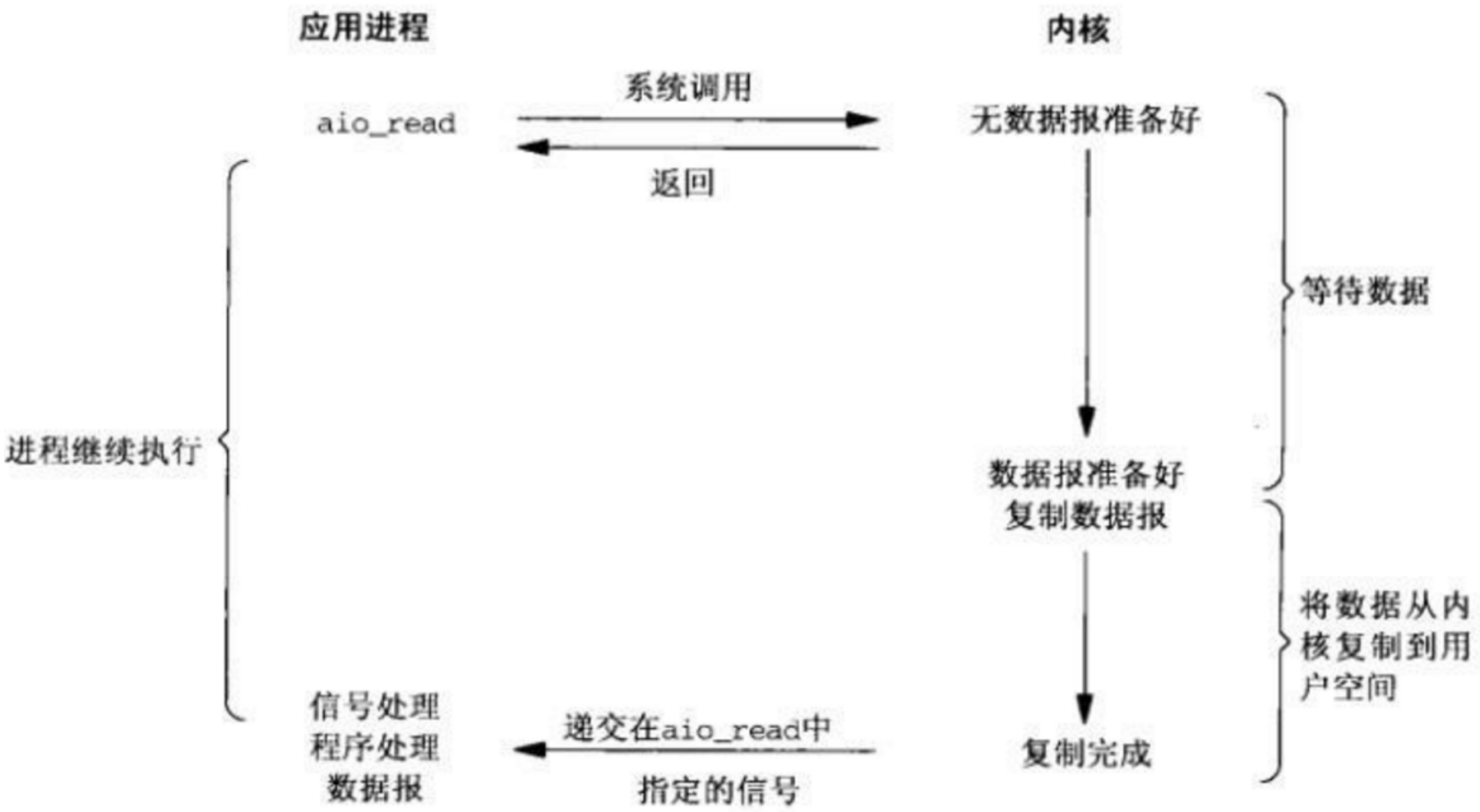


图6-5　异步I/O模型

五个IO模型的比较

图 6.6　五个 I/O 模型的比较

1.2 OS Aproach

Multitasking

- Non-Preemptive multitasking
  - The programmer `yielded` control to the OS
  - Every bug could halt the entire system
  - Example: Windows 95
- Preemptive multitasking
  - OS can stop the execution of a process, do something else, and switch back
  - OS is responsible for scheduling tasks
  - Example: UNIX, Linux

User-level Thread

- Advantages
  - Simple to use
  - A "context switch" is reasonably fast
  - Each stack only gets a little memory
  - Easy to incorporate *preemption*
- Drawbacks
  - The stacks might need to grow
  - Need to save all the CPU state on every switch
  - Complicated to implement correctly if you want to support many different platforms
  - Example: Green Threads

Kernel-supported Threads

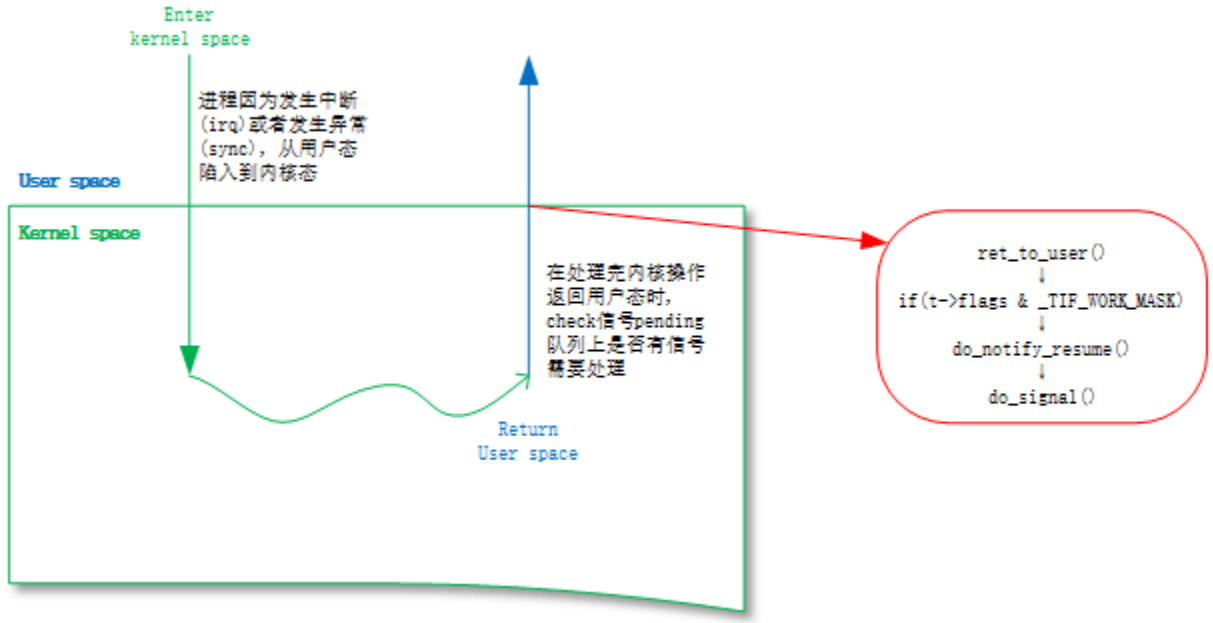
- **Advantages**
  - Easy to use
  - Switching between tasks is reasonably fast
  - Geting parallelism for free
- **Drawbacks**
  - OS level threads come with a rather large stack
  - There are a lot of syscalls involved
  - Might not be an option on some systems, such as http server

Example: Using OS threads in Rust

信号（Signal(用户态的异步处理机制)）

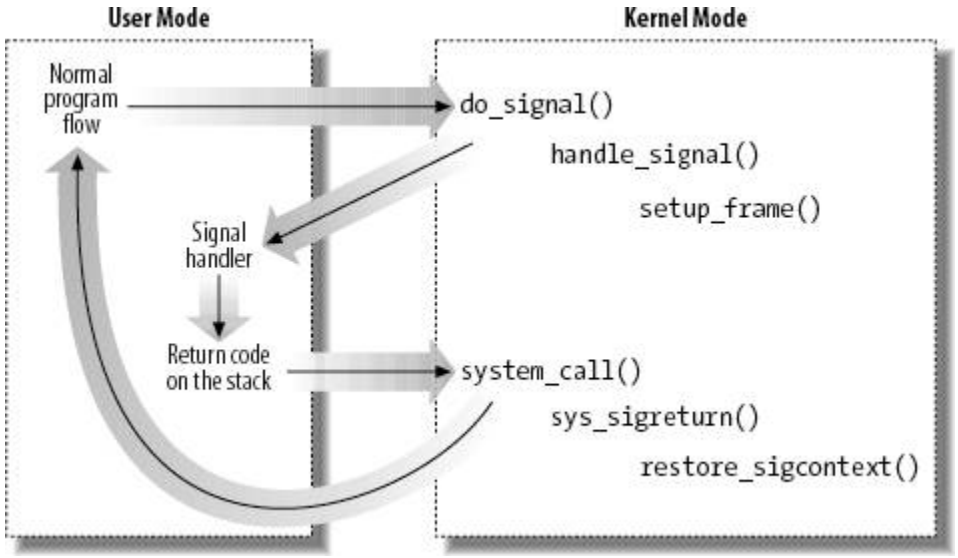
信号（Signal）响应时机

- 发送信号并没有发生硬中断，只是把信号挂载到目标进程的信号 pending 队列
- 信号执行时机：进程执行完异常/中断返回到用户态的时刻



信号处理

- 用户注册的信号处理函数都是用户态的
  - 先构造堆栈，返回用户态去执行自定义信号处理函数
  - 再返回内核态继续被信号打断的返回用户态的动作。



1.3 Programming Aproach

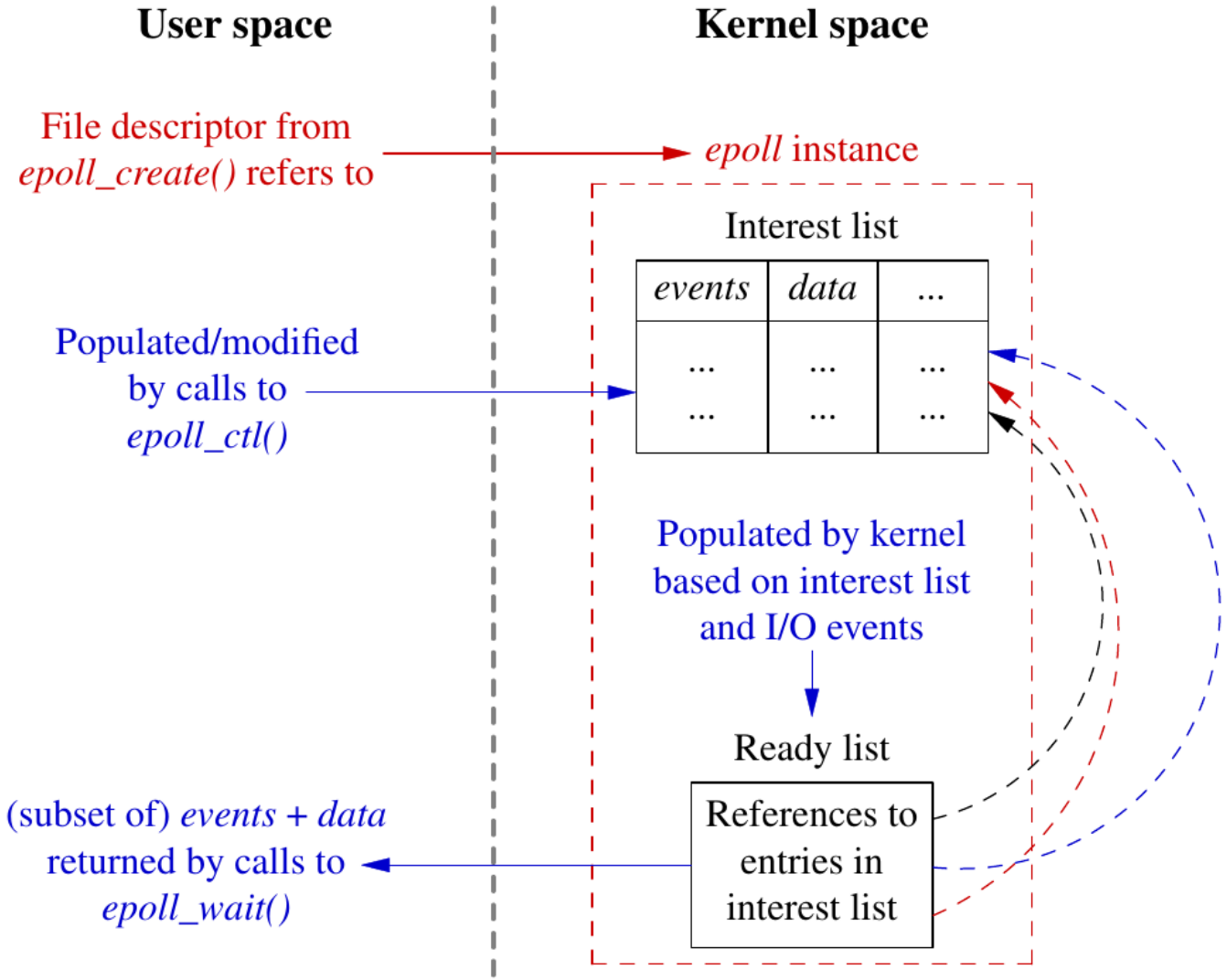
Callback based approaches

- Save a pointer to a set of instructions to run later together.
  - Advantages
    - Easy to implement in most languages
    - No context switching
    - Relatively low memory overhead
  - Drawbacks
    - Memory usage grows linearly with the number of callbacks
    - Callback hell: Hard to debug
    - Require a substantial rewrite to go from a "normal" program flow to one that uses a "callback based" flow

Event queue: Epoll, Kqueue and IOCP

- Epoll
  - Epoll is the Linux way of implementing an event queue
  - Epoll works very efficiently with a large number of events
- Kqueue
  - Kqueue is the MacOS way of implementing an event queue, which originated from BSD
  - In terms of high level functionality, it's similar to Epoll in concept but different in actual use
- IOCP
  - IOCP or Input Output Completion Ports is the way Windows handles this type of event queue

Epoll


User space
Kernel space
File descriptor from
epoll_create() refers to
epoll instance
Interest list
events
data
...
Populated/modified
by calls to
epoll_ctl()
Populated by kernel
based on interest list
and I/O events
Ready list
References to
entries in
interest list
(subset of) events + data
returned by calls to
epoll_wait()

io_uring

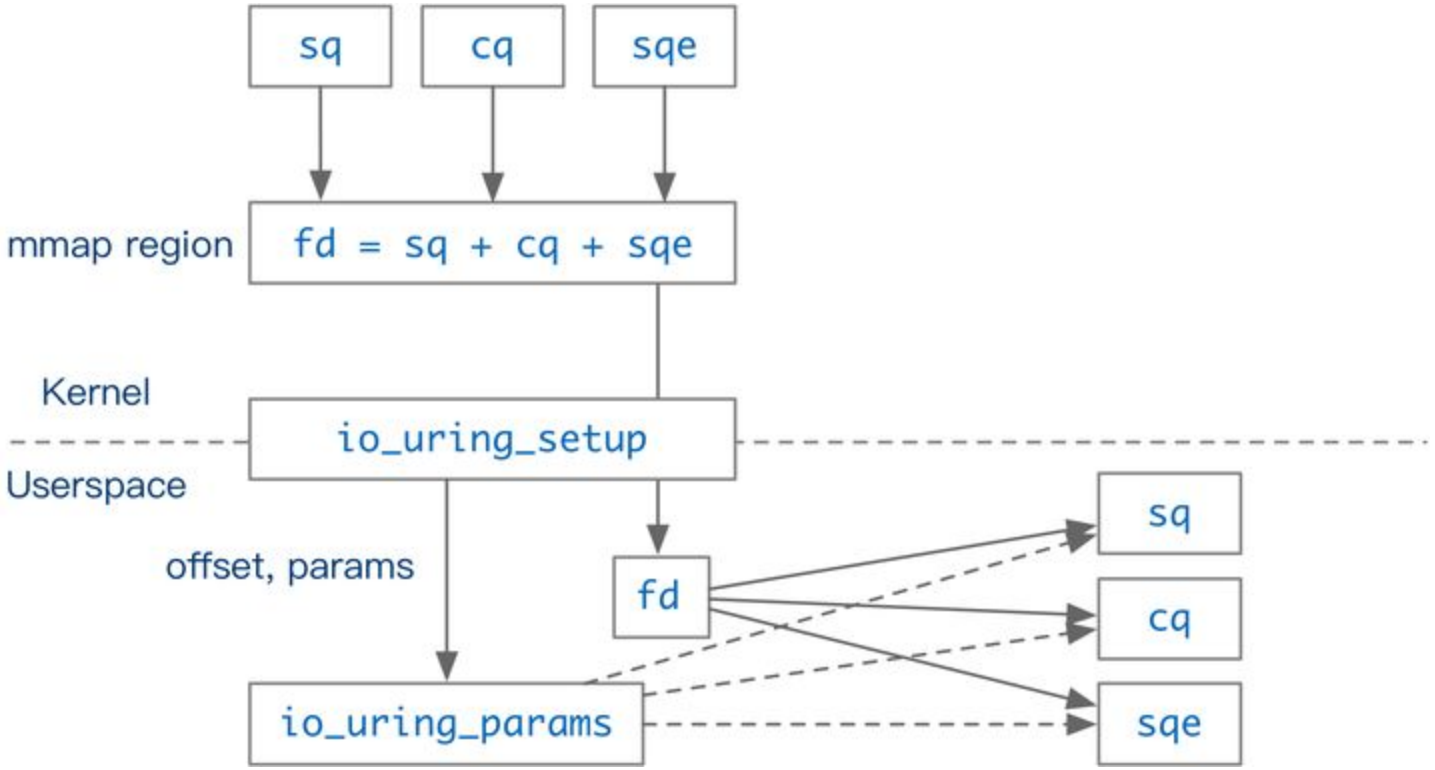
- io_uring 的用户态 API: `io_uring` 的实现仅仅使用了三个 syscall
  - `io_uring_setup`：设置 `io_uring` 上下文
  - `io_uring_enter`：提交并获取完成任务
  - `io_uring_register`：注册内核用户共享的缓冲区

io_uring 的IO过程

缩略语	英语	中文
SQ	Submission Queue	提交队列
CQ	Completion Queue	完成队列
SQE	Submission Queue Entry	提交队列项
CQE	Completion Queue Entry	完成队列项
Ring	Ring	环

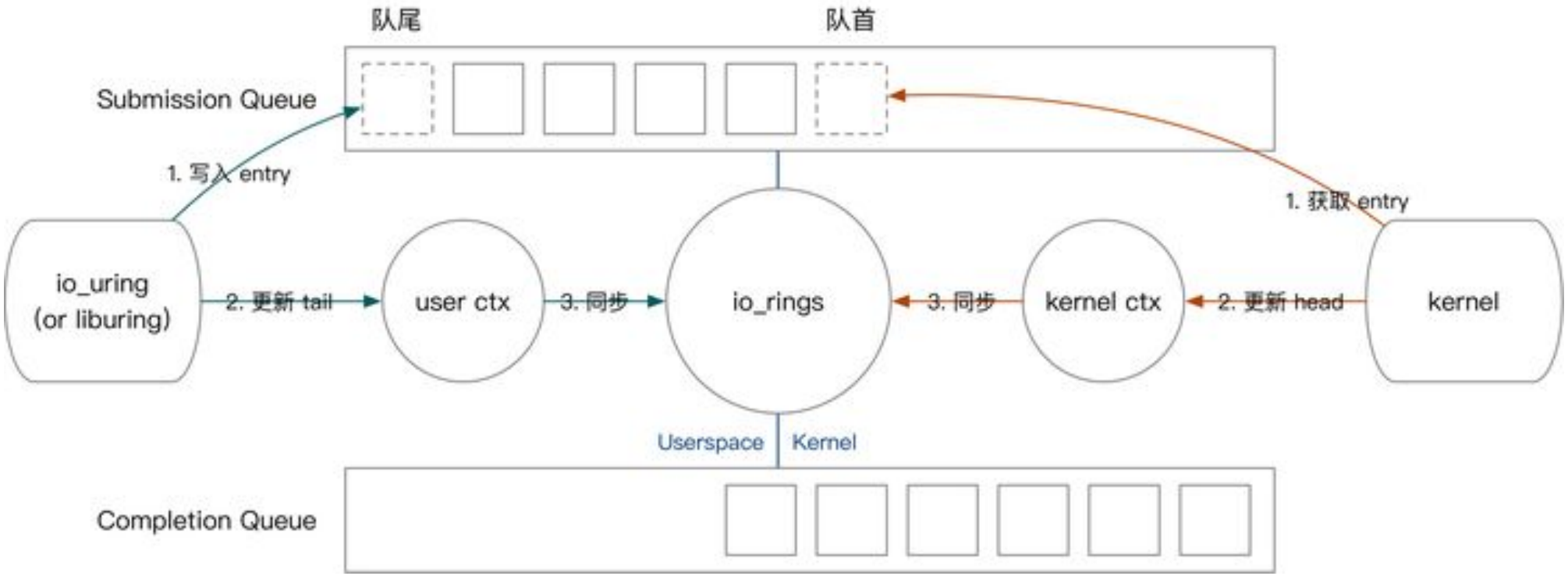
初始化 io_uring

```
long io_uring_setup(u32 entries, struct io_uring_params __user *params)
```



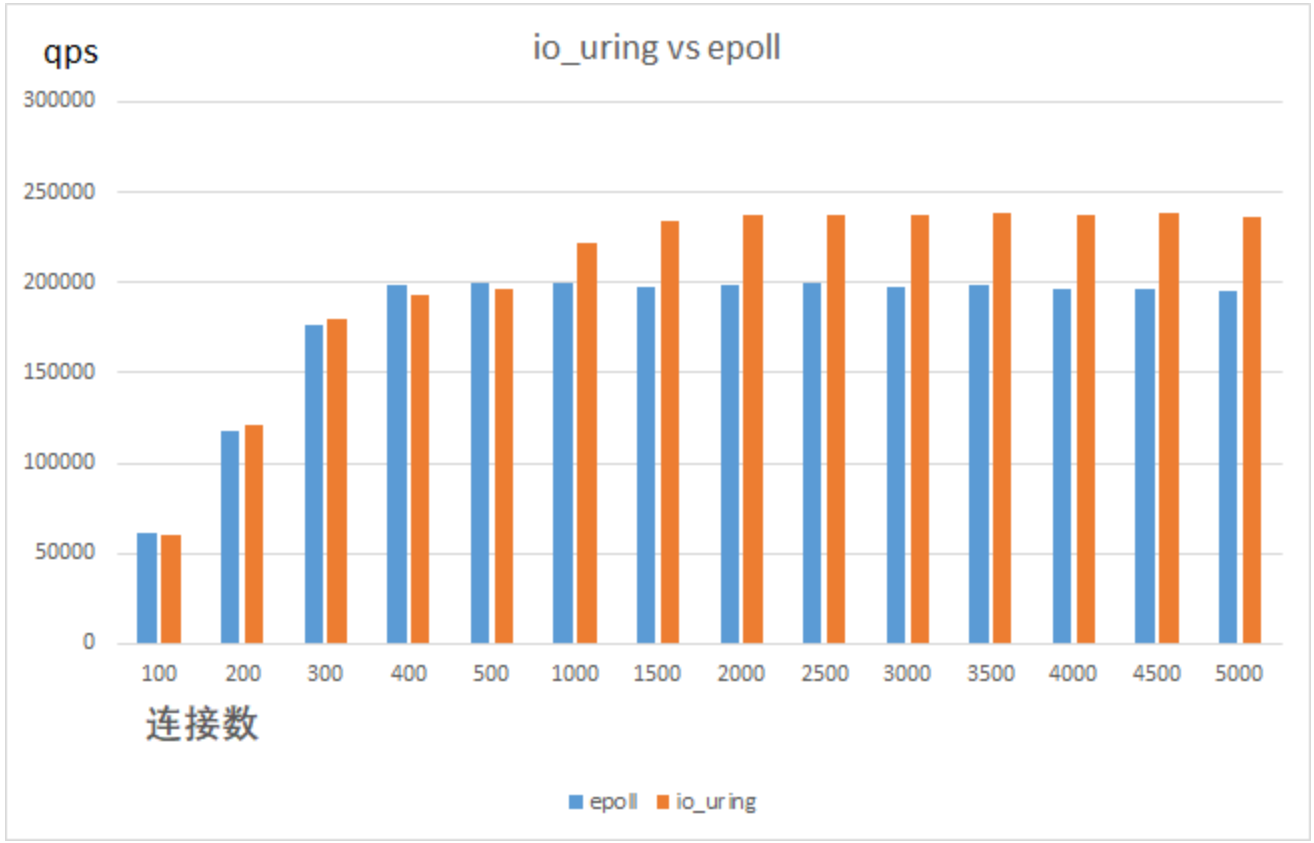
任务的提交与完成

- `io_uring` 通过环形队列和用户交互。

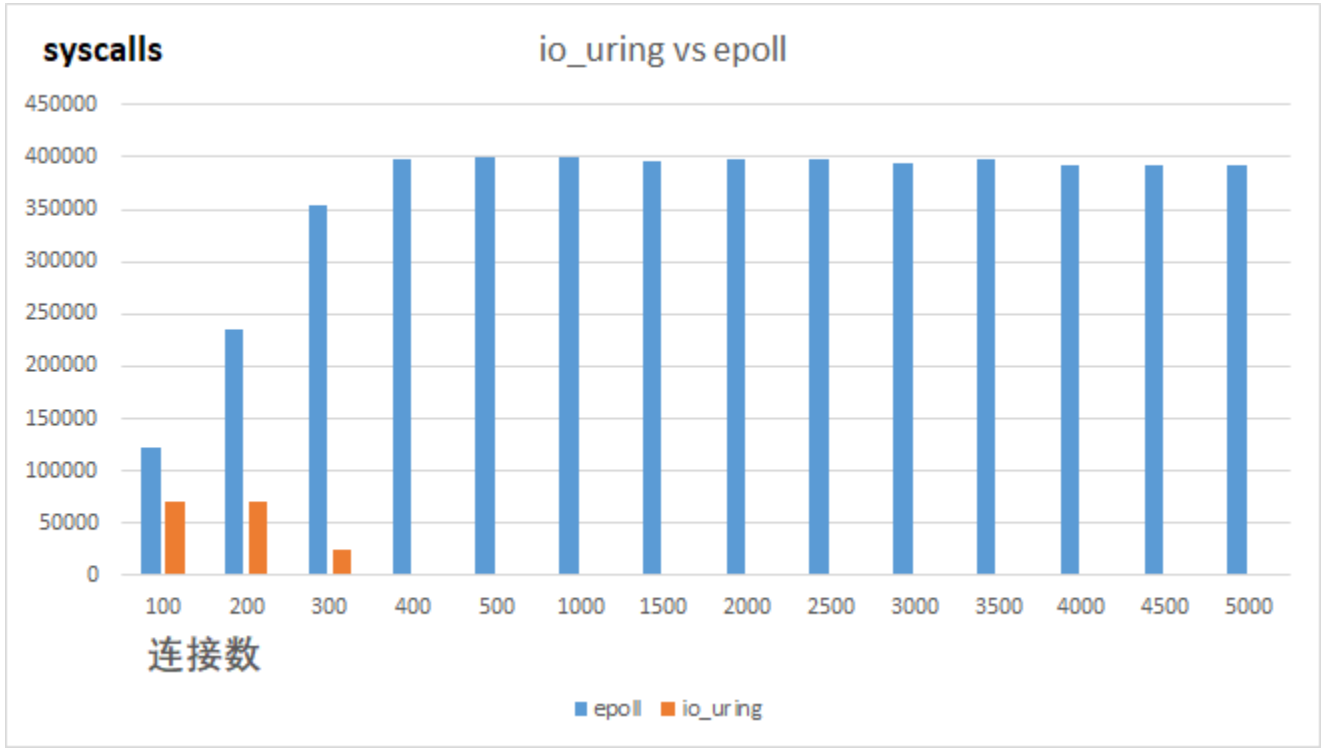


io_uring的性能

io_uring和epoll在echo_server场景下qps数据对比



io_uring和epoll在echo_server场景下系统调用上下文切换数量的对比



2. Rust语言的并发和异步支持

2.1 Rust的并发编程

2.2 Future in Rust

2.3 Function colors in Rust

2.1 Rust的并发编程

Rust的设计目的

两个棘手问题

1. 怎样才能安全地进行系统编程?
2. 怎样才能容易地使用并发?

ownership

1. 每一个值都具有一个“拥有域(owning scope)”；
2. 传递或返回一个值会转移ownership(移动它)到新的域(scope)中；
3. 当一个域(scope)结束时，如果域所拥有的值还没销毁，此时将自动销毁；

借出(borrow)

Rust会检查所有借出的值，确保它们的寿命不会超过值本身的寿命；
每一个引用仅在一个有限的域(scope)中有效，编译器会自动判定；

- 不可变引用`&T`，可以共享但不能被改变。
- 可变引用`&mut T`，可以被改变但不能共享。

通道(channel)

```
fn send<T: Send>(chan: &Channel<T>, t: T);
fn recv<T: Send>(chan: &Channel<T>) -> T;
```

- 通道中传输的数据类型是泛型的(`<T:Send>`是API的一部分);
- 只要传递一个`T`给函数`send`就意味着会转移它的ownership;

锁(Locks)

- `Mutex`是一个类型`T`的泛型类型，`T`是锁要保护的数据。
  当你在创建一个`Mutex`时，会将数据的ownership转移到mutex中，并立即放弃对它的访问。
- 调用`lock`函数来阻塞线程直到获取到锁。它会返回一个值，`MutexGuard<T>`。 当`MutexGuard<T>`销毁时，它会自动释放锁。
- 访问数据的唯一方式是函数`access`，它将可变引用`MutexGuard<T>`转换为一个可变引用`T`（临时借用）。

2.2 Future in Rust

Future的设计目标

- 调用 I/O 时，系统调用会立即返回，然后你可以继续进行其他工作
- I/O完成时，回到调用该异步 I/O 暂停的那个任务线上
- **一种通过对异步 I/O 的良好抽象形成的基于库的解决方案**
  - 它不是语言的一部分，也不是每个程序附带的运行时的一部分，只是可选的并按需使用的库

From callbacks to futures (deferred computation)

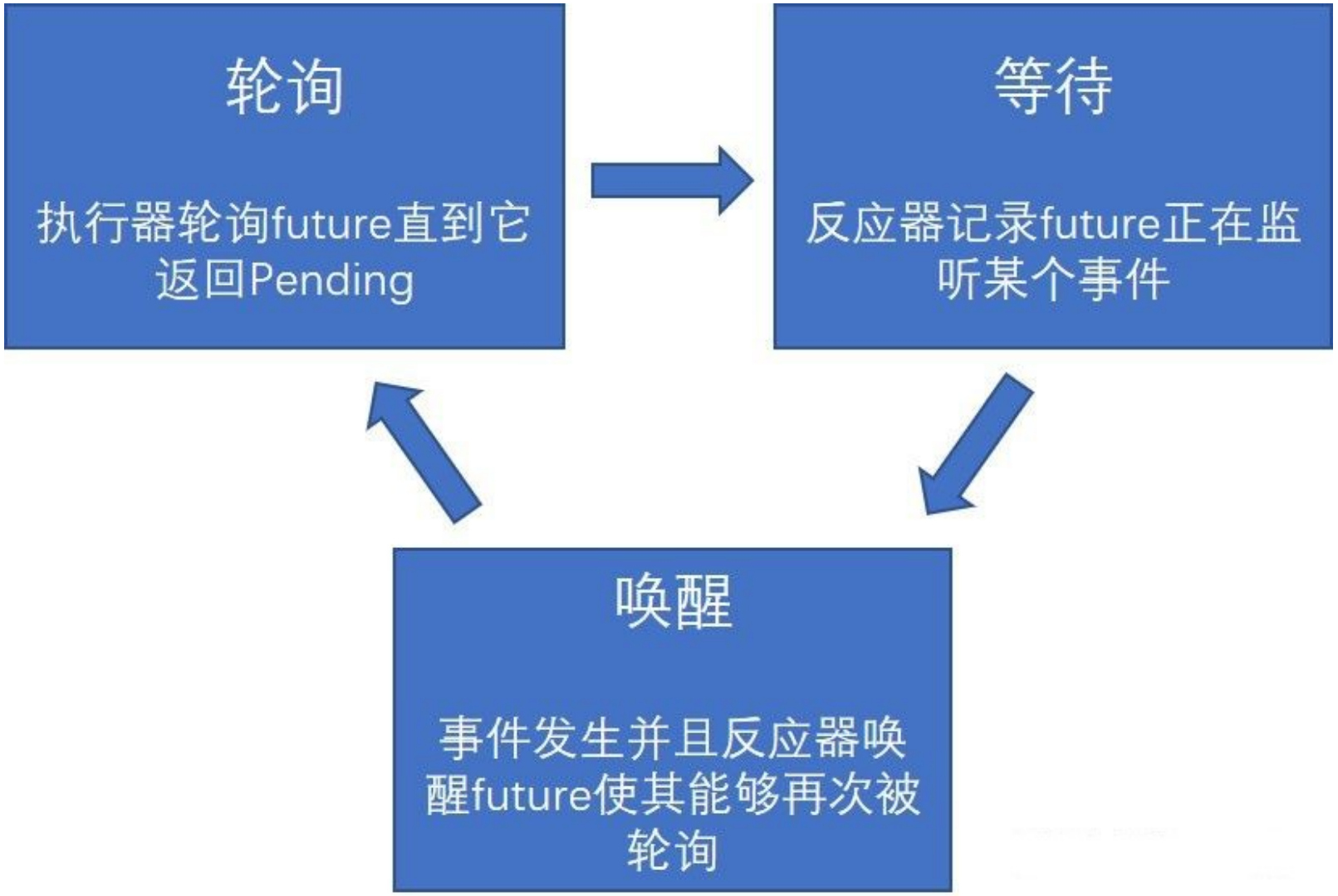
- Future is one way to deal with the complexity which comes with a callback based approach.

```
async function run() {
    await timer(200);
    await timer(100);
    await timer(50);
    console.log("I'm the last one");
}
```

Concept of Future

- Three phases in asynchronous task:
    i. **Executor**: A Future is polled which result in the task progressing
        - Until a point where it can no longer make progress
    ii. **Reactor**: Register an event source that a Future is waiting for
        - Makes sure that it will wake the Future when that event is ready
    iii. **Waker**: The event happens and the Future is woken up
        - Wake up to the executor which polled the Future
        - Schedule the future to be polled again and make further progress

基于轮询的 Future的异步执行过程



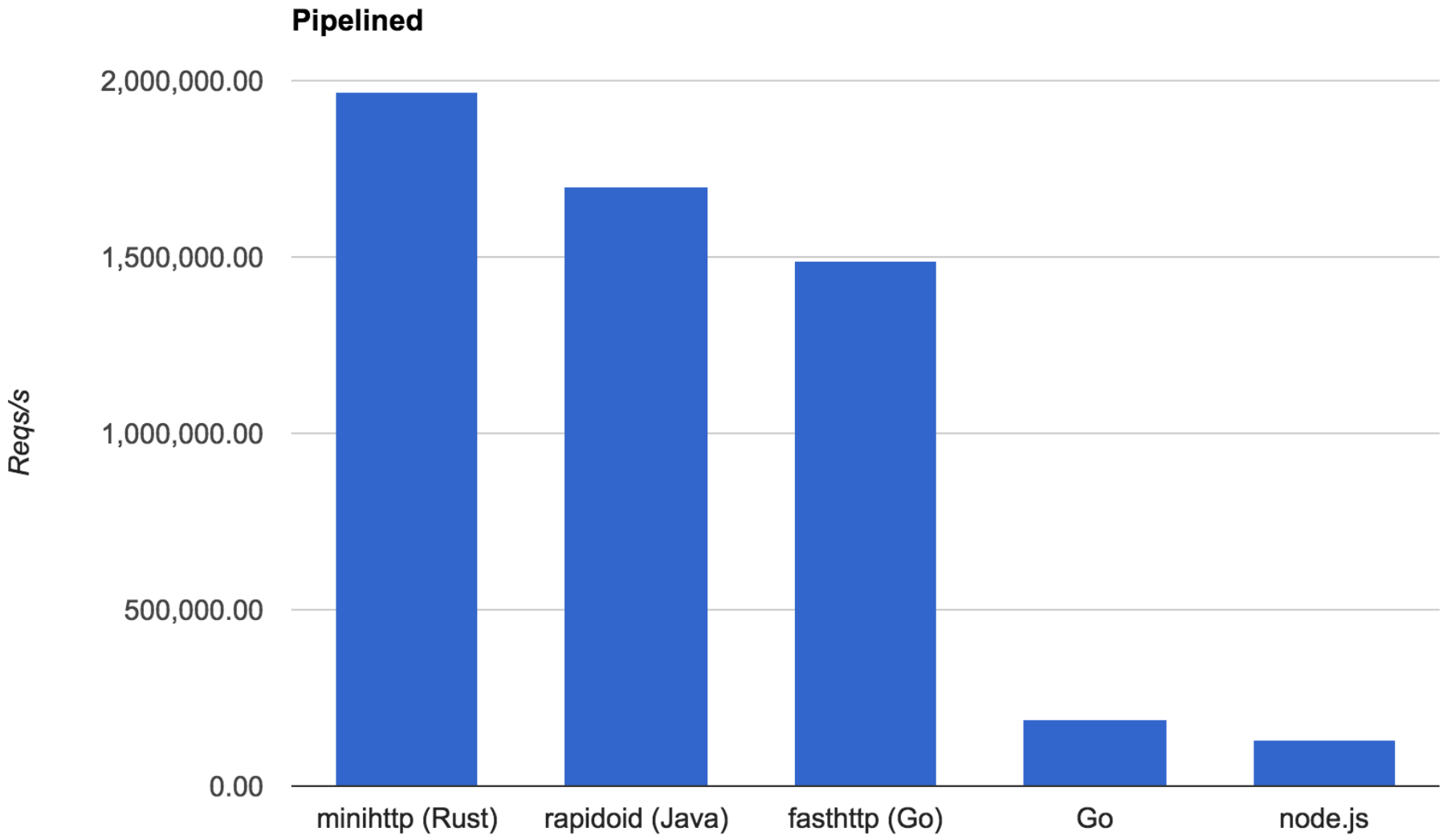
Runtimes

- Languages like C#, JavaScript, Java, GO and many others comes with a runtime for handling concurrency
- Rust uses a library for handling concurrency
- The two most popular runtimes for Futures:
    - async-std
    - Tokio

Zero-cost futures in Rust

- Build up a big `enum` that represents the state machine
  - There is one allocation needed per “task”, which usually works out to one per connection
- When an event arrives, only one dynamic dispatch is required
- There are essentially no imposed synchronization costs

Here are the results, in number of “Hello world!"s served per second on an 8 core Linux machine.



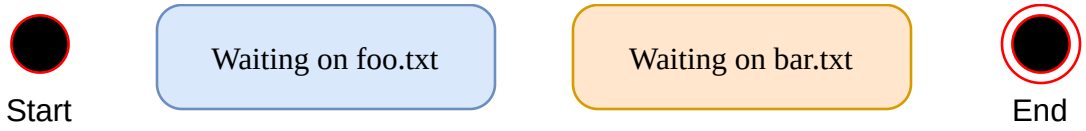
Generators and async/await

Concurrency in Rust

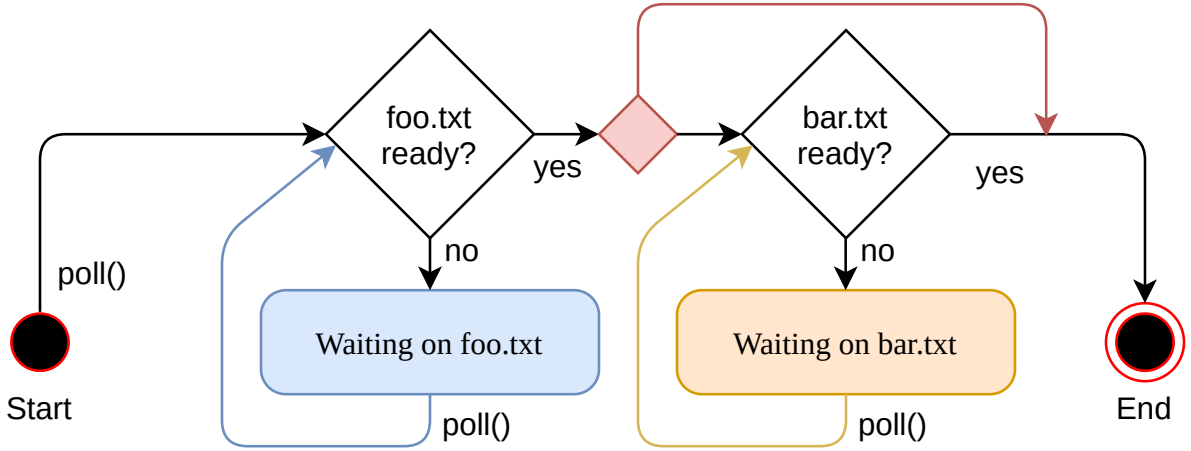
1. Stackful coroutines, better known as green threads.
2. Stackless coroutines, better known as generators.

State Machine Transformation in Future

- Each state represents a different pause point of the function



- Arrows represent state switches and diamond shapes represent alternative ways



Self-Referential Structs & Pin

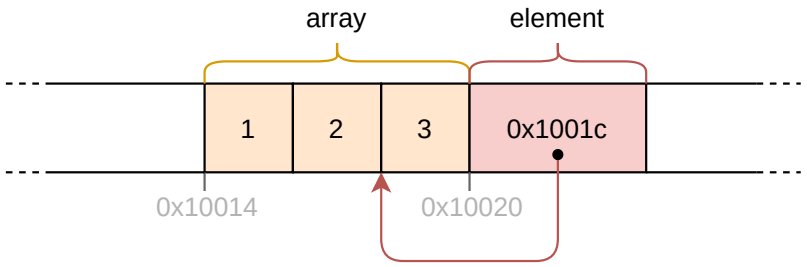
```
async fn pin_example() -> i32 {
    let array = [1, 2, 3];
    let element = &array[2];
    async_write_file("foo.txt", element.to_string()).await;
    *element
}
```

The struct for the "waiting on write" state

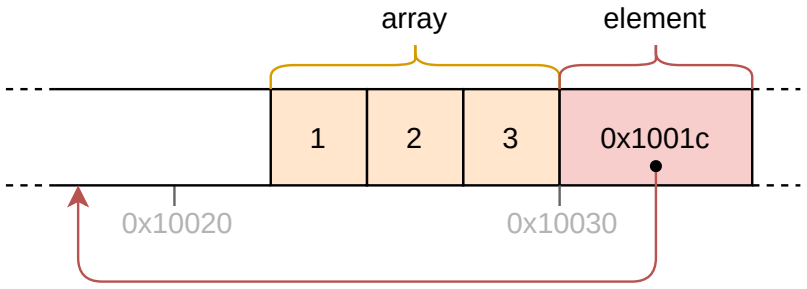
```
struct WaitingOnWriteState {
    array: [1, 2, 3],
    element: 0x1001c, // address of the last array element
}
```

The Problem with Self-Referential Structs

- memory layout of self-referential struct



- After moving this struct to a different memory address



Defination of Pin

- Pin wraps a pointer. A reference to an object is a pointer
  - **Reference type**. In order to break apart a large future into its smaller components, and put an entire resulting future into some immovable location, we need a reference type for methods like `poll`
- Pin gives some guarantees about the *pointee* (the data it points to)
  - **Never to move before being dropped**. To store references into itself, we decree that by the time you initially `poll`, and promise to never move an immobile future again

Waker

- The `Waker` type allows for a loose coupling between the reactor-part and the executor-part of a runtime
- By having a wake up mechanism that is *not* tied to the thing that executes the future, runtime-implementors can come up with interesting new wake-up mechanisms
- Creating a `Waker` involves creating a `vtable` which allows us to use dynamic dispatch to call methods on a *type erased* trait object we construct our selves

Reactor

- To actually abstract over this interaction with the outside world in an asynchronous way
  - Receive events from the operating system or peripherals
  - Forward them to waiting tasks
- Mio: Library of reactors in Rust
  - Provide non blocking APIs and event notification for several platforms

2.3 Function colors in Rust

- Rust async executors provide a `block_on()` primitive that invokes an async function from a non-async context and blocks until the result is available
- Rust async provides `spawn_blocking()` which invokes a blocking sync function from an async context, temporarily suspending the current async function without blocking the rest of the async environment.

Rust中的异步支持库

- Tokio
- async-nostd

3. 异步操作系统

3.1 osblog

3.2 Drogue IoT

3.3 基于Rust的异步操作系统AsyncOS构想

3.1 osblog: RISC-V OS using Rust

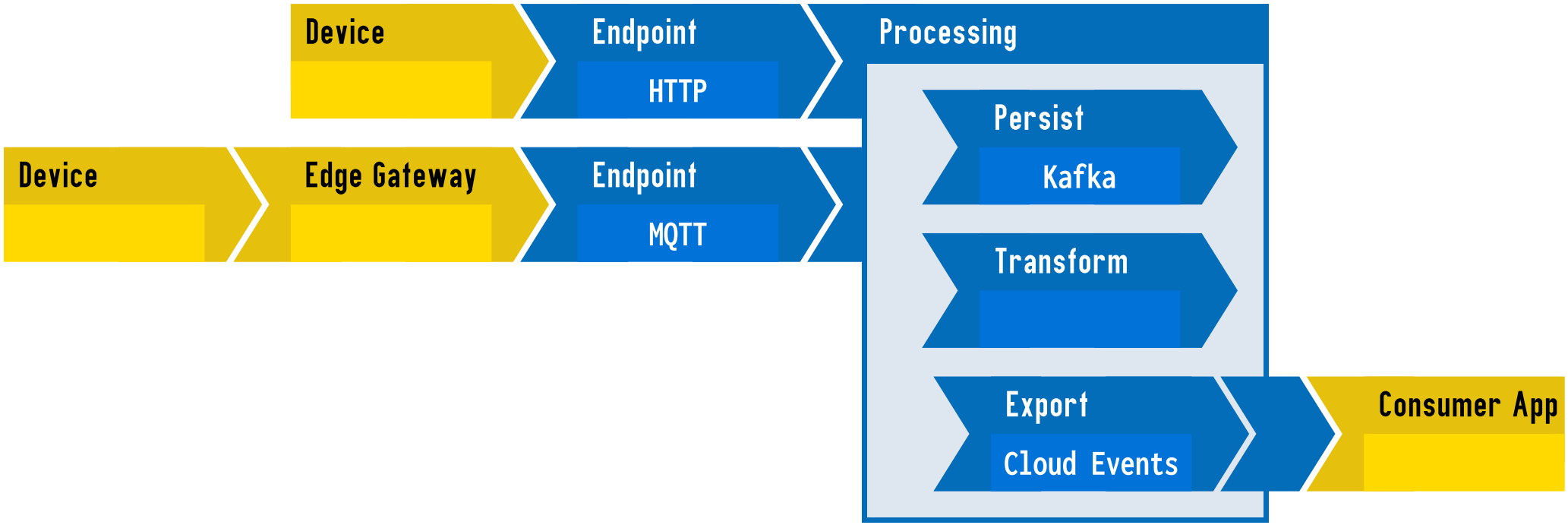
This tutorial will progressively build an operating system from start to something that you can show your friends or parents -- if they're significantly young enough.

The Road Ahead...

+ Chapter 0: Setup and pre-requisites (UPDATED 2020: Rust out-of-the-box!)
+ Chapter 1: Taking control of RISC-V
+ Chapter 2: Communications
+ Chapter 3.1: Page-grained memory allocation
+ Chapter 3.2: Memory Management Unit
+ Chapter 4: Handling interrupts and traps
+ Chapter 5: External interrupts
+ Chapter 6: Process memory
+ Chapter 7: System calls
+ Chapter 8: Starting a process
+ Chapter 9: Block driver
+ Chapter 10: Filesystems
+ Chapter 11: Userspace Processes

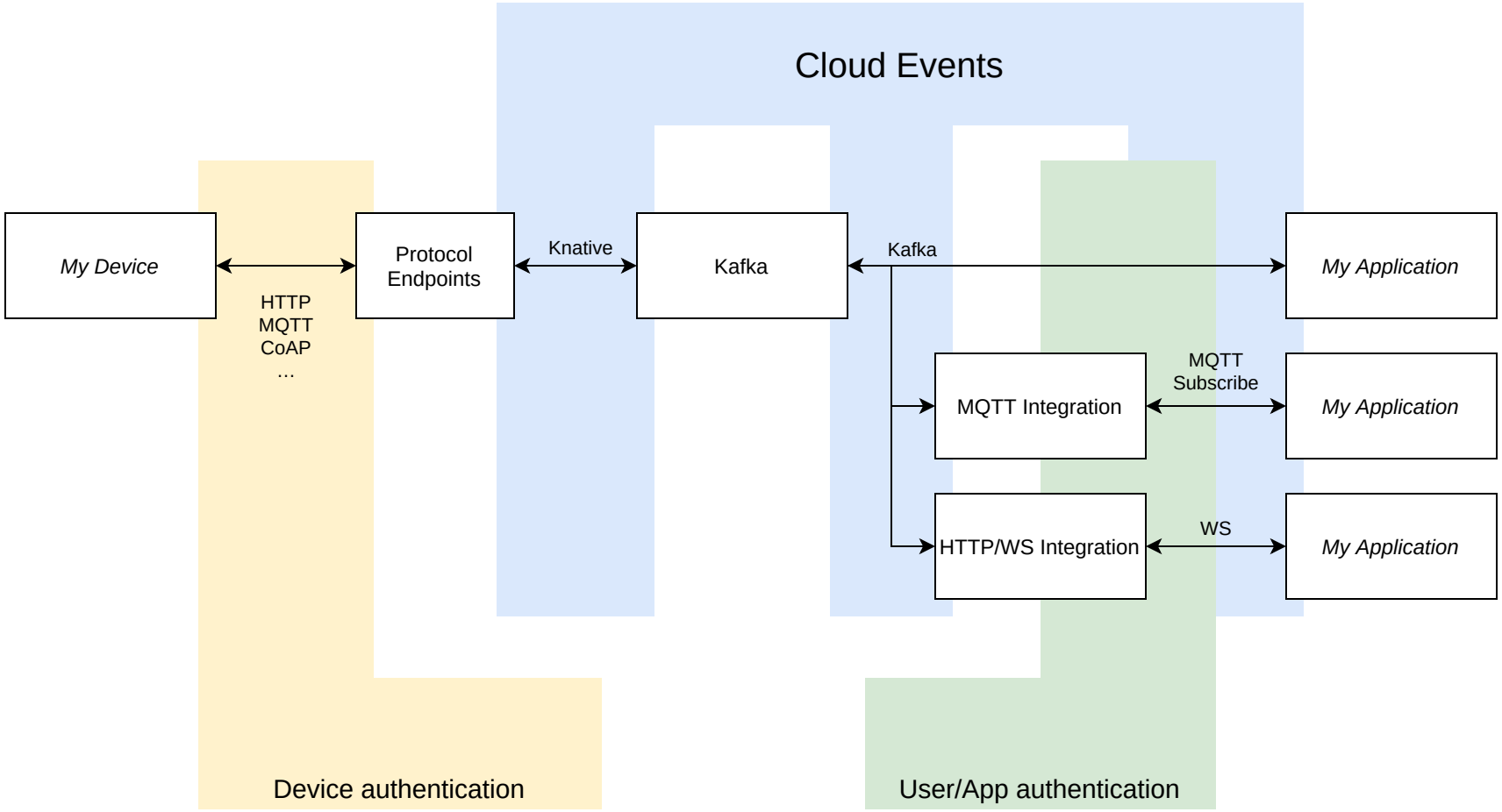
3.2 Drogue IoT

Drogue Cloud



abstraction of protocols in Drogue

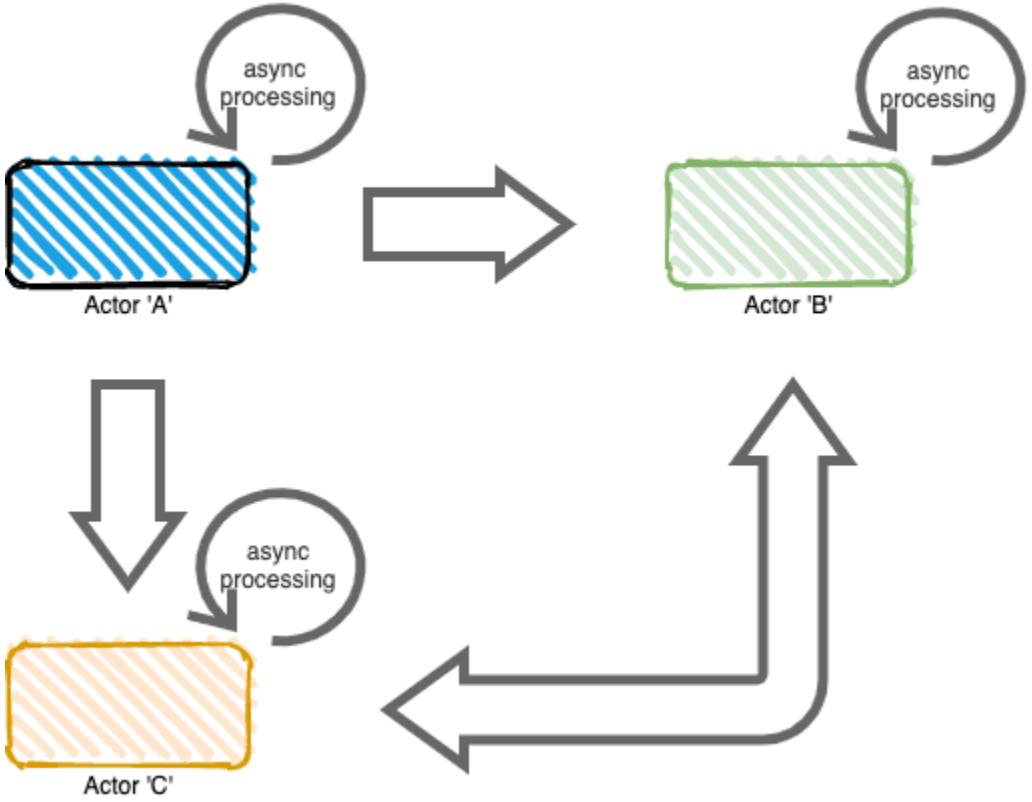
Data plane in Drogue



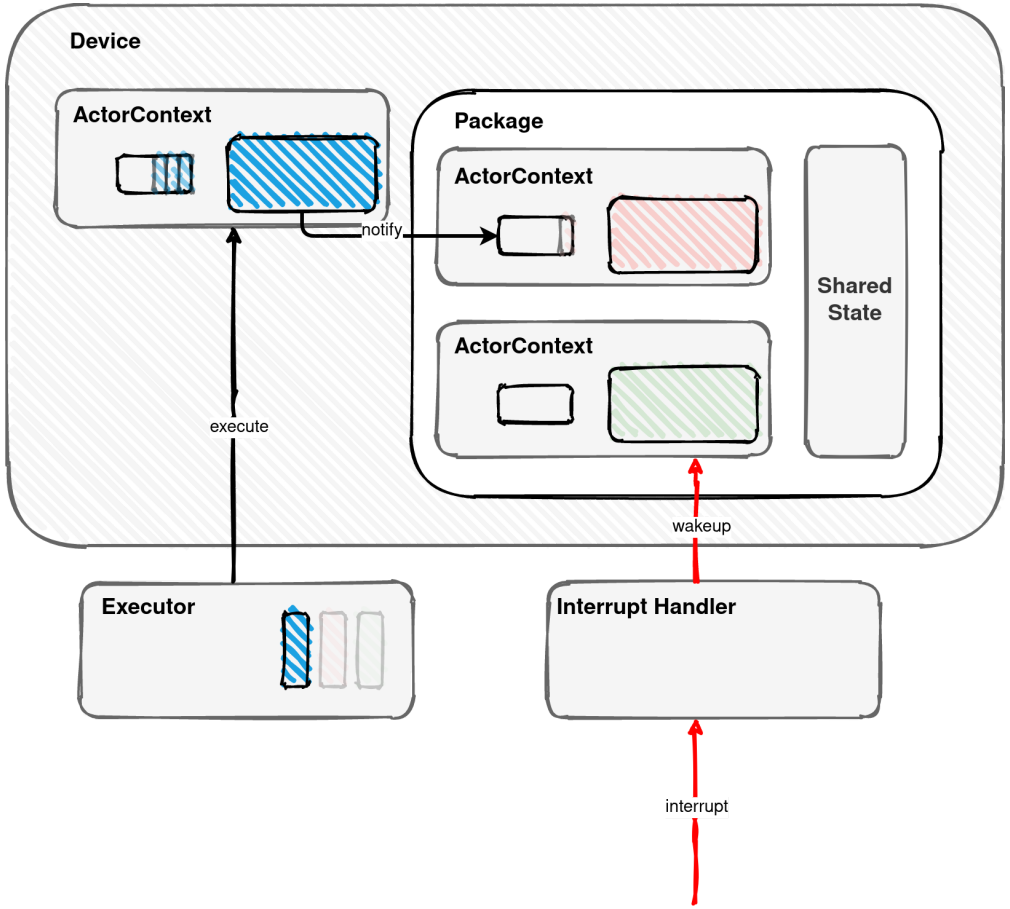
Introducing Drogue Device

- Actor-based: State is held by an actor, accessed/mutated only by that actor, in response to messages.
- Cooperative Scheduling: Using Rust's async/await support, actors attempt to be non-blocking and share the processor.
- Message-Passing: Support notifications, requests/responses, and an event-bus.

Actor System



Drogue Actor Model



3.3 基于Rust的异步操作系统AsyncOS构想

在RISC-V平台上设计并实现一个基于Rust语言的异步操作系统。

1. 在操作系统内核中实现细粒度的并发安全、模块化和可定制特征；
2. 利用Rust语言的异步机制，优化操作系统内核的并发性能；
3. 完善操作系统的进程、线程和协程概念，统一进程、线程和协程的调度机制；
4. 利用RISC-V平台的用户态中断技术，向应用程序提供的异步系统调用接口；
5. 开发原型系统，设计用户态测试用例库和操作系统动态分析跟踪工具，对异步操作系统的特征进行定量性的评估。

任务管理：进程、线程与协程

- 进程：每个进程有独立的地址空间，进程切换将导致页表切换；
  - 内核是一个独立的进程（内核进程），它仅运行在内核态；内核进程实现系统调用服务实现和资源管理；
  - 用户进程的地址空间分成用户地址空间和内核地址空间两部分；内核地址空间仅包括支持进程切换的必要功能；
  - 系统调用将导致进程切换，从而统一系统调用和进程间通信；依赖用户态中断，可实现不通过内核进程的进程间通信；

任务管理：进程、线程与协程

- 协程：作为CPU调度的基本单位，协程是基于状态转移的异步函数执行流；
  - 协程在主动让出CPU时，解除与栈的绑定关系；
  - 主动让出CPU的就绪协程进入运行状态前需要与空闲栈绑定；
  - 由于被抢占而让出CPU时，协程将继续占用所绑定的栈；
- 线程：栈与协程必须绑定后才能在CPU上执行。栈与协程的绑定形成了线程的概念。
  - 相同进程内主动让权时的协程切换，可使用相同的栈；
  - 相同进程内被抢占协程的切换，会导致栈的切换；

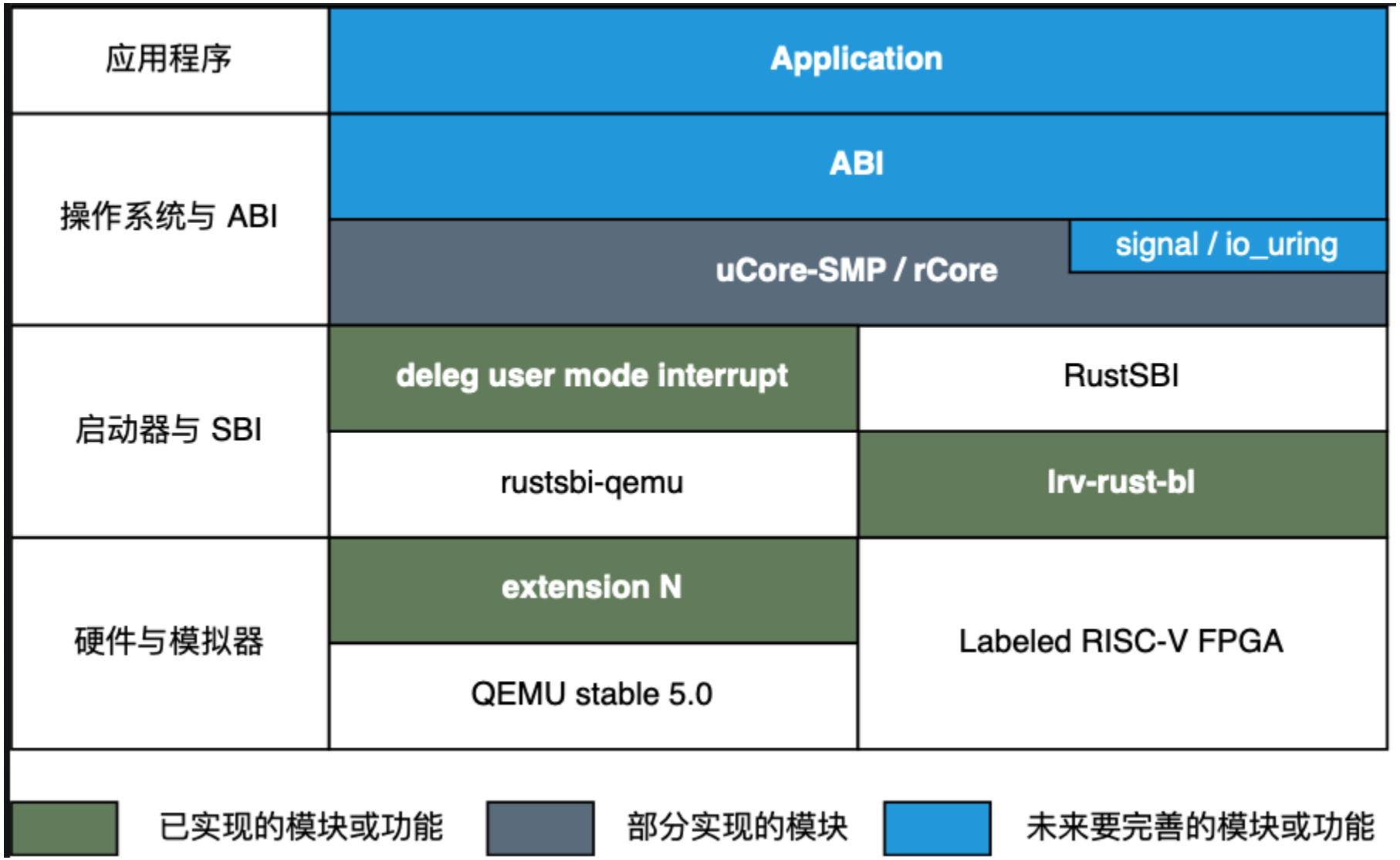
协程、线程和进程的调度

- 已有工作：华中科大蒋周奇、车春池：飓风内核的共享调度器
- 共享调度器的特征
  - 协程是CPU调度的基本单位；
  - 协程具备优先级属性；
  - 整个系统内的所有协程依据优先级进行统一调度；
  - 协程切换：相同进程内的主动让权协程间的切换；
  - 线程切换：相同进程内被抢占协程的切换；
  - 进程切换：不同进程间的协程切换；
  - 调度器代码由内核进程维护，可在用户态执行；

用户态中断机制

- 用户态中断是由硬件实现
- 协程可通过ecall指令向指定进程发送用户态中断请求；
- 在软件的参与下，硬件可直接向指定进程发送用户态中断请求；
- 处于运行状态的进程会立即响应和处理发给自己的用户态中断；
- 处于暂停状态的进程会在进入运行状态时优先响应和处理发给自己的用户态中断；

用户态中断实现


应用程序
Application
操作系统与 ABI
ABI
uCore-SMP / rCore
signal / io_uring
启动器与 SBI
deleg user mode interrupt
RustSBI
rustsbi-qemu
lrv-rust-bl
硬件与模拟器
extension N
Labeled RISC-V FPGA
QEMU stable 5.0
已实现的模块或功能
部分实现的模块
未来要完善的模块或功能

异步系统调用

- 系统调用分为同步和异步两种
  - 同步系统调用：用户进程发出系统调用请求后进入等待状态；内核进程执行系统调用服务功能后唤醒用户进程；用户进程获取系统调用结果并继续执行；
  - 异步系统调用：用户进程中的某协程发出系统调用请求后主动让权（可继续执行其他协程）；内核进程执行系统调用服务功能后通过用户态中断唤醒用户进程；用户进程响应用户态中断，获取系统调用结果并继续对应协程执行；

异步系统调用的执行过程

- 第一次异步系统调用时：
  i. 用户进程准备系统调用参数、发出系统调用请求；
  ii. 内核进程将映射共享内存、发起相应服务协程的异步执行；
  iii. 内核进程执行完服务协程后，在响应队列保存返回值，并通过用户态中断通知应用进程；
- 第二次异步系统调用时：
  i. 用户进程在请求队列准备系统调用参数；在共享内存的响应队列中查看第一次系统调用的结果；
  ii. 内核进程在完成第一个服务协程后，在共享内存的响应队列中保存返回值，主动查询新的系统调用请求，并执行；如果没有新的请求，则让出CPU；