操作系统专题训练

第二讲: 并发与处理器、操作系统和编程语言

向勇
清华大学计算机系
xyong@tsinghua.edu.cn
2022年9月

提纲

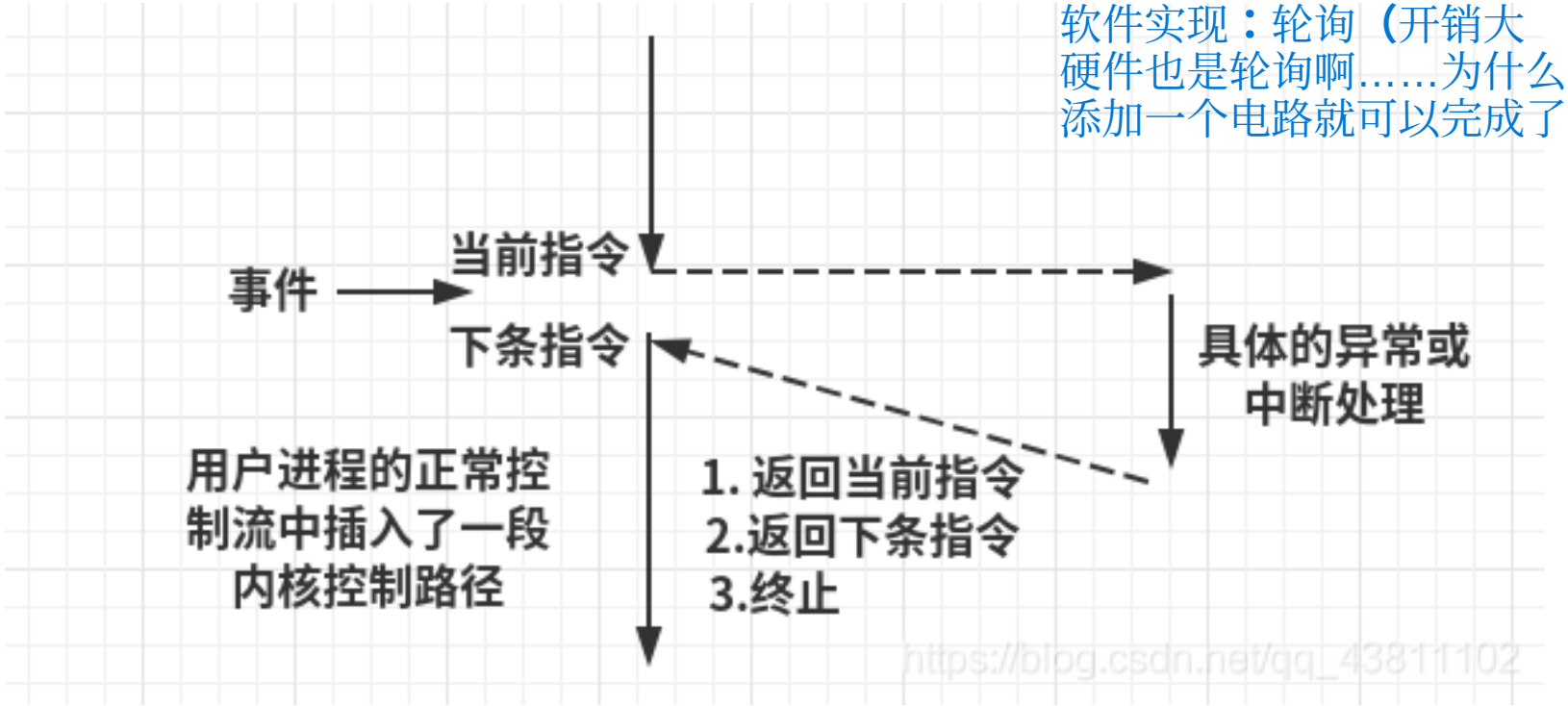
1. CPU硬件对并发的支持

2. 操作系统对并发的支持
3. Rust语言对并发的支持
4. 异步操作系统

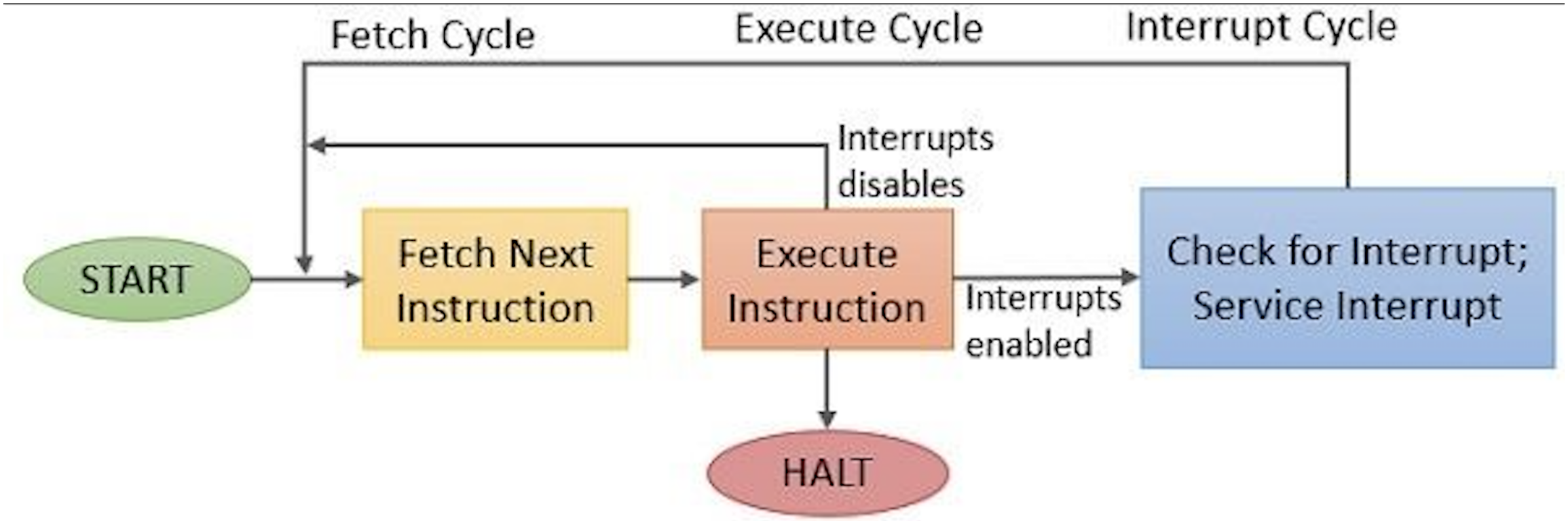
中断概念

程序执行中，由于**异常情况**，CPU上正在执行的程序会被“中断”，转到处理异常情况或特殊事件的程序去执行，结束后再返回到原被“中断”的程序处继续执行。

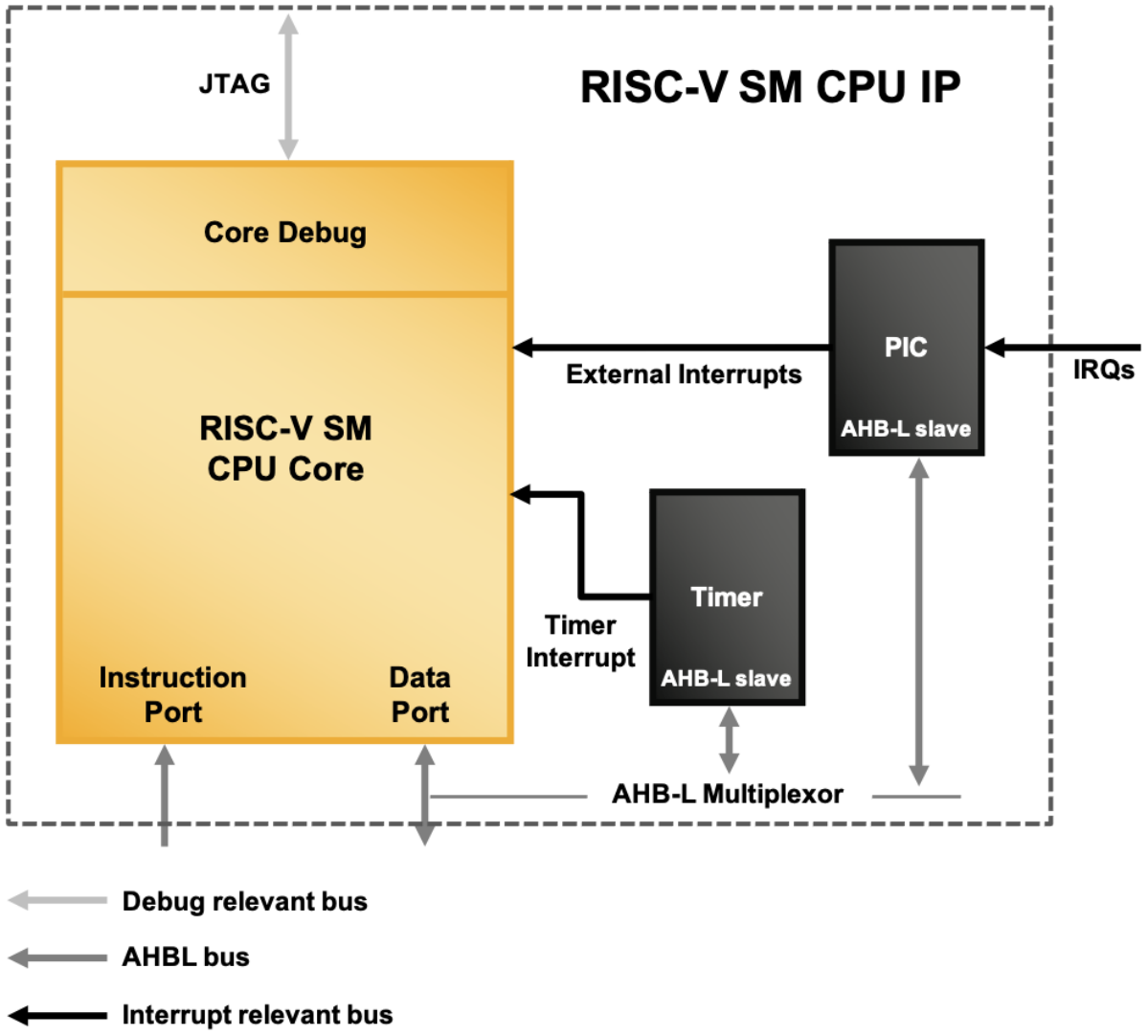
中断如何完成的？
软件实现：轮询（开销大
硬件也是轮询啊……为什么开销不大呢？
添加一个电路就可以完成了



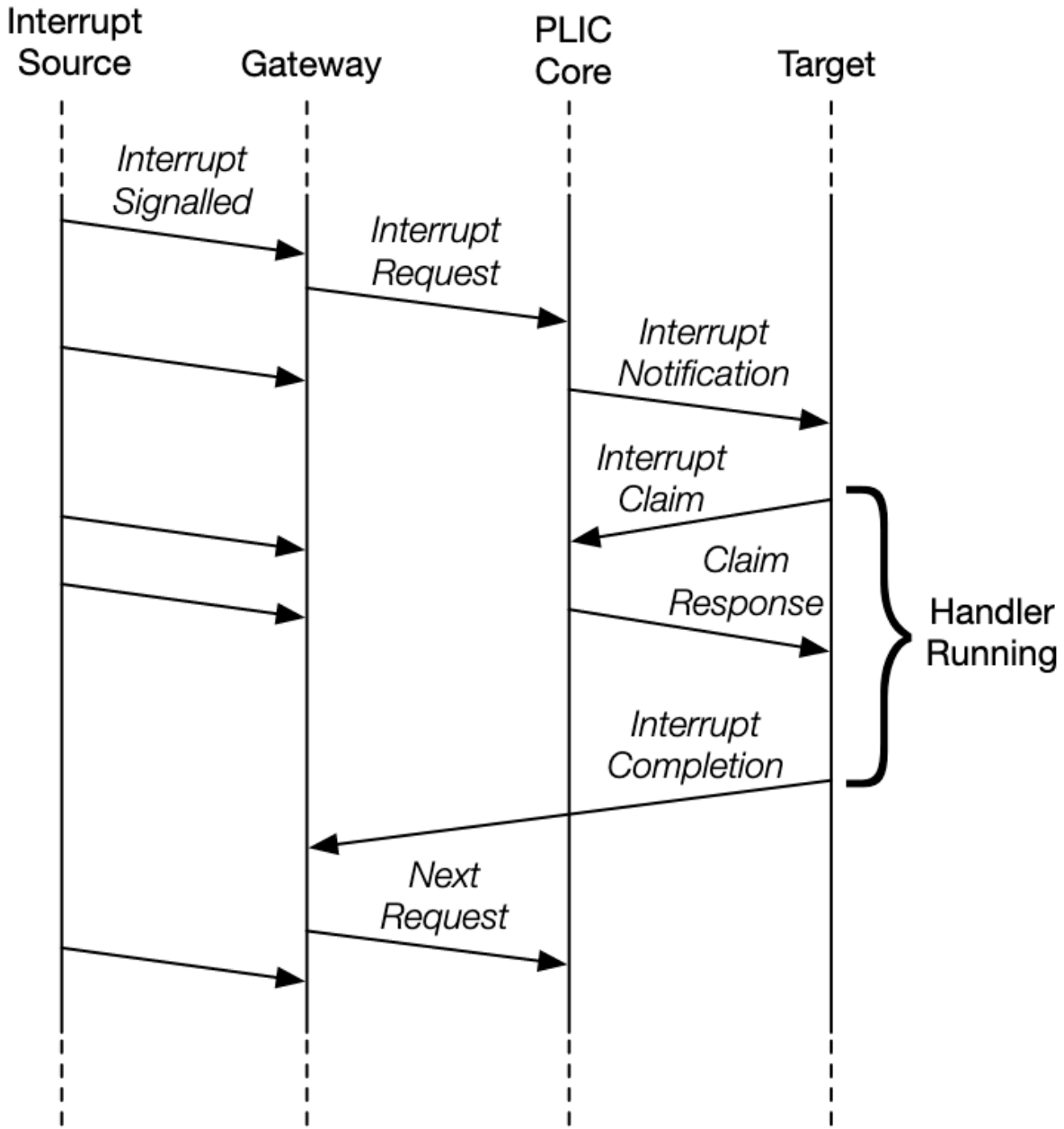
Instruction Cycle with Interrupts



RISC-V中断


JTAG
RISC-V SM CPU IP
Core Debug
RISC-V SM
CPU Core
Instruction
Port
Data
Port
External Interrupts
PIC
AHB-L slave
IRQs
Timer
Interrupt
Timer
AHB-L slave
AHB-L Multiplexor
Debug relevant bus
AHBL bus
Interrupt relevant bus

RISC-V PLIC interrupt Flow



RISC-V N-Extention

信号机制的用户态迁移

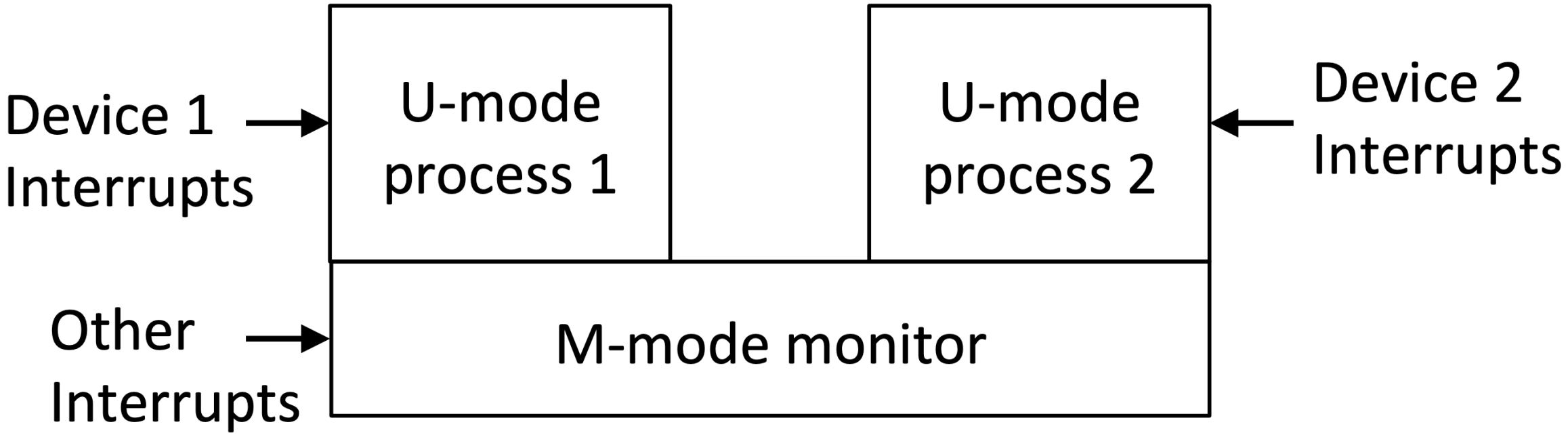
"N" Standard Extension for User-Level Interrupts

Objective:

- secure embedded systems with only M-mode and U-mode present
- user-level trap handling.
  - replace conventional signal handling
  - generate user-level events such as garbage collection barriers, integer overflow, floating-point traps

Ref: Krste Asanović, RISC-V interrupt

Interrupts in Secure Embedded Systems



User-Level Interrupts “N” - User Trap Setup

- Natural extension of interrupt model into user permissions
- Adds user CSRs, and **uret** instruction

Number	Privilege	Name	Description
0x000	URW	ustatus	User status register.
0x004	URW	uie	User interrupt-enable register.
0x005	URW	utvec	User trap handler base address.

麻烦之处在哪里？**原本的中断是OS完成的，**current process**在**CPU**上。**
现在呢？用户态可不止一个process，发给谁？哪个正在执行（可能就没在执行

User-Level Interrupts “N” - User Trap Handling

- Natural extension of interrupt model into user permissions
- Adds user CSRs, and **uret** instruction

Number	Privilege	Name	Description
0x040	URW	uscratch	Scratch reg. for user trap handlers.
0x041	URW	uepc	User exception program counter.
0x042	URW	ucause	User trap cause.
0x043	URW	ubadaddr	User bad address.
0x044	URW	uip	User interrupt pending.

Interrupts in Secure Embedded Systems

(M, U modes)

- M-mode runs secure boot and runtime monitor
- Embedded code runs in U-mode
- Physical memory protection on U-mode accesses
- Interrupt handling can be delegated to U-mode code
- Provides arbitrary number of isolated subsystems

Intel UINTR

- In the User Interrupts hardware architecture, a receiver is always expected to be a user space task.
- A user interrupt can be sent by another user space task, kernel or an external source (like a device).

Intel UINTR

The average normalized latency for a 1M ping-pong IPC notifications with message size=1.

IPC type	Relative Latency (normalized to User IPI)
User IPI	1.0
Signal	14.8
Eventfd	9.7
Pipe	16.3
Domain	17.3

Kernel managed architectural data structures for UINTR

- UPID: User Posted Interrupt Descriptor
- UITT: User Interrupt Target Table

The interrupt state of each task is referenced via MSRs which are saved and restored by the kernel during context switch.

Instructions for UINTR

- `senduipi <index>` - **send** a user IPI to a target task based on the UITT index.
- `clui` - **Mask** user interrupts by clearing UIF (User Interrupt Flag).
- `stui` - **Unmask** user interrupts by setting UIF.
- `testui` - **Test** current value of UIF.
- `uiret` - return from a **user interrupt handler**.

API for UINTR

1)A receiver can **register/unregister** an interrupt handler using the Uintr receiver related syscalls.

```
uintr_register_handler(handler, flags)
```

```
uintr_unregister_handler(flags)
```

2)A syscall also allows a receiver to **register a vector** and create a user interrupt file descriptor - uintr_fd.

```
uintr_fd = uintr_create_fd(vector, flags)
```

API for UINTR

3)Any sender with access to uintr_fd can use it to **deliver events** (in this case - interrupts) to a receiver.

```
uipi_index = uintr_register_sender(uintr_fd, flags)
```

4a)After the initial setup, a sender task can use the SENDUIPI instruction along with the uipi_index to **generate user IPIs** without any kernel intervention.

```
SENDUIPI <uipi_index>
```

4b) If the **sender is the kernel or a device**, the uintr_fd can be passed onto the related kernel entity to allow them to setup a connection and then generate a user interrupt for event delivery.

提纲

1. CPU硬件对并发的支持

2. 操作系统对并发的支持

3. Rust语言对并发的支持
4. 异步操作系统

操作系统服务

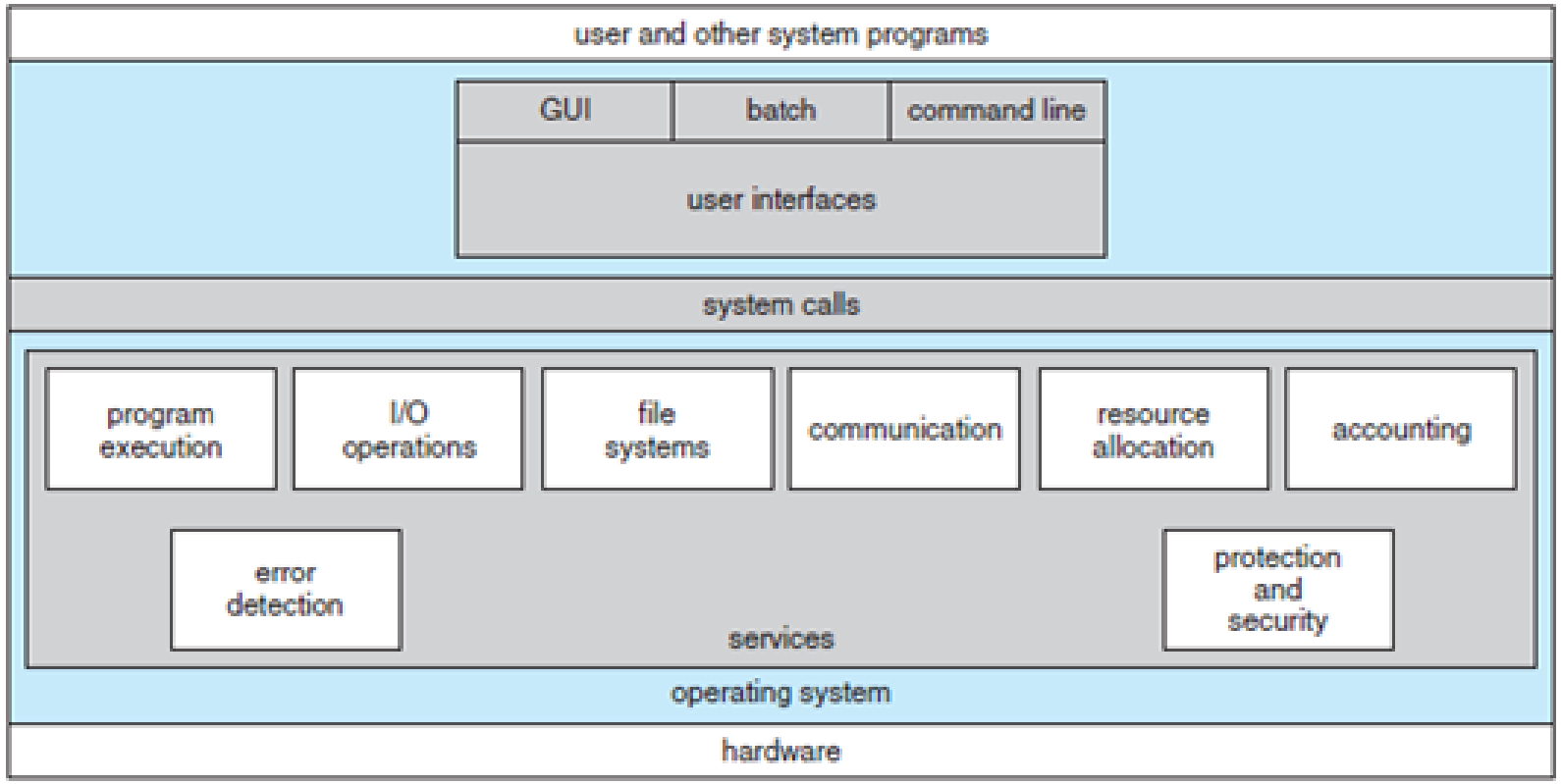
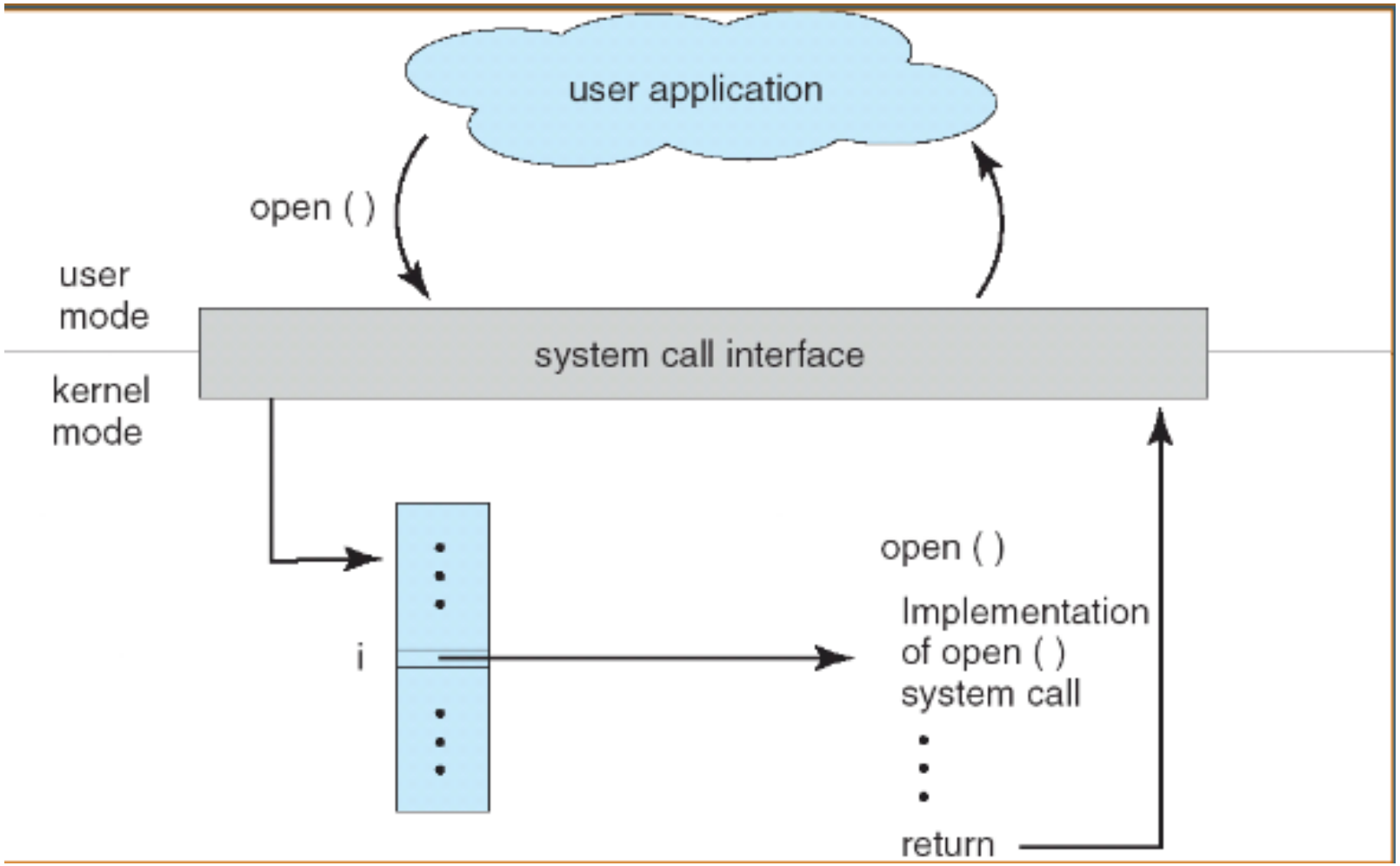
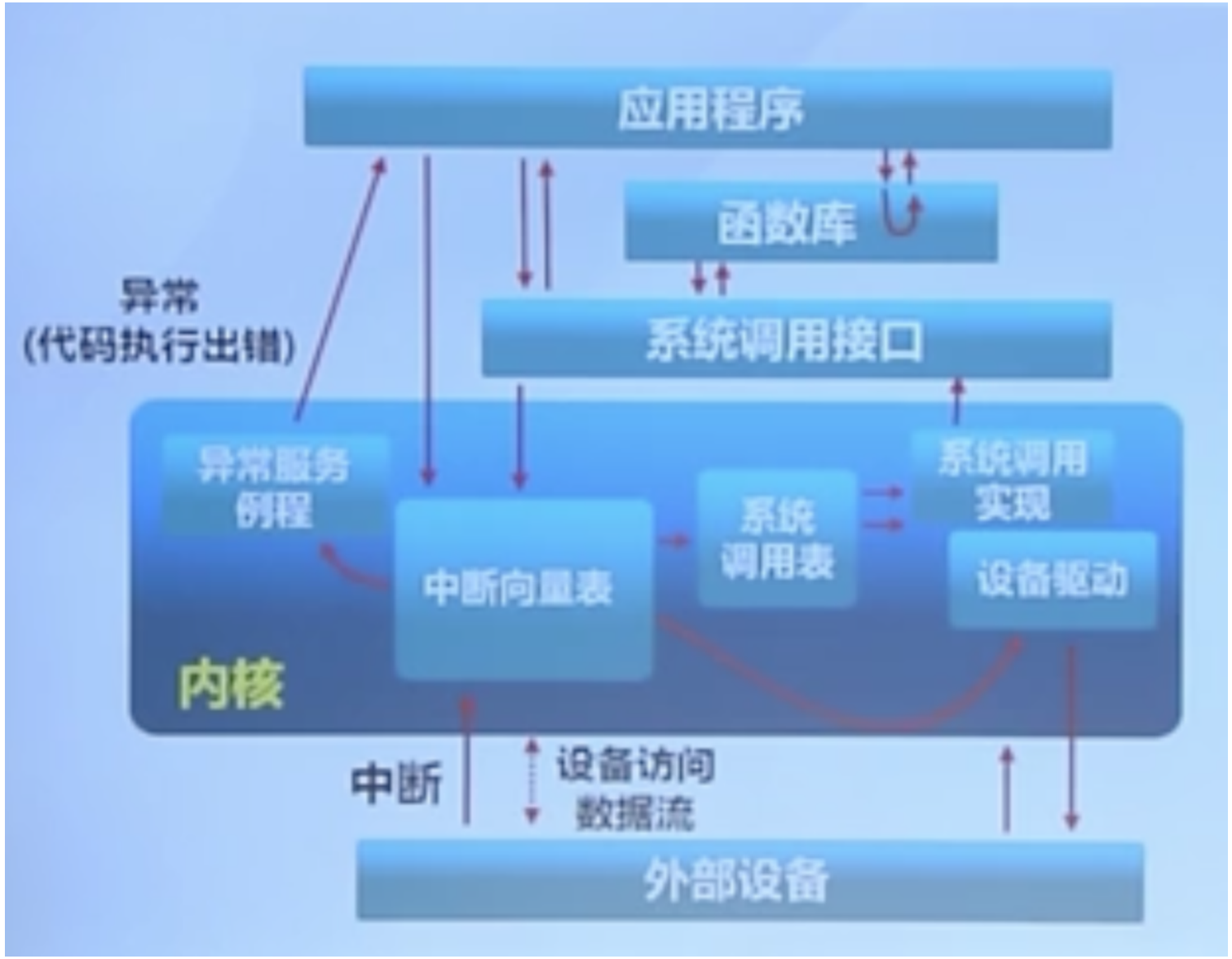


Figure 2.1 A view of operating system services.

系统调用

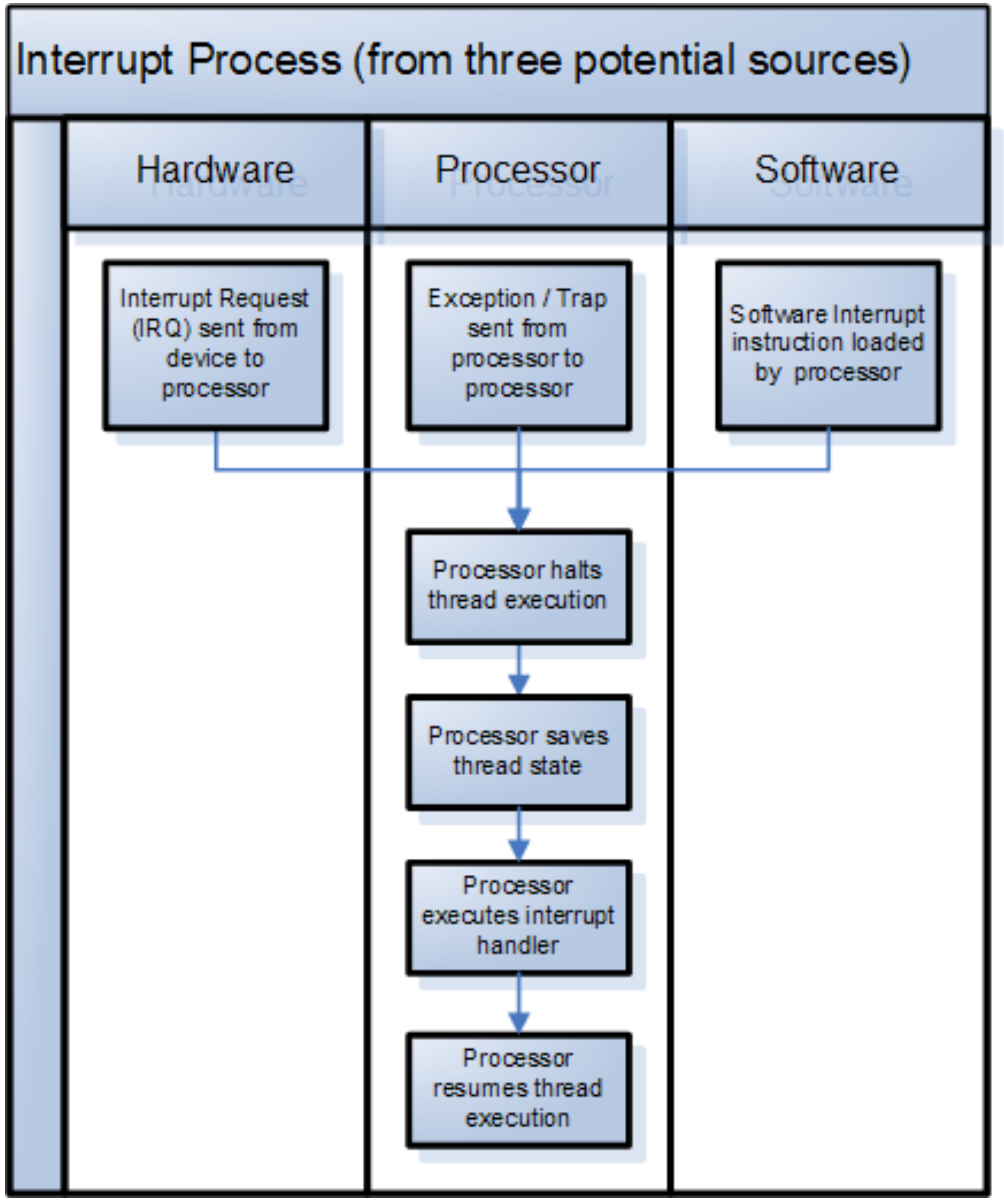

user application
open ( )
user mode
system call interface
kernel mode
i
open ( )
Implementation of open ( ) system call
return

内核的进入与退出


应用程序
函数库
异常
(代码执行出错)
系统调用接口
异常服务
例程
中断向量表
系统
调用表
系统调用
实现
设备驱动
内核
中断
设备访问
数据流
外部设备

硬中断的响应过程

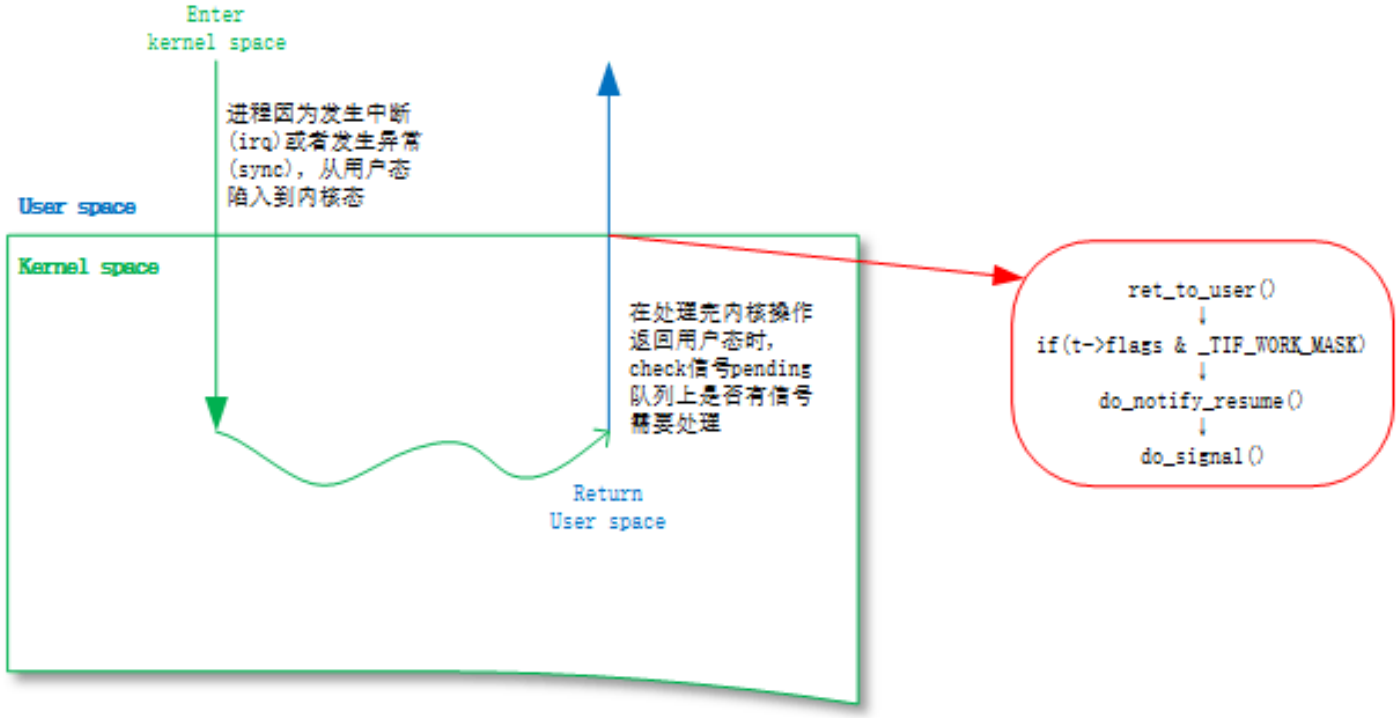
操作系统的中断响应过程



信号（Signal)

用户态的异常处理机制

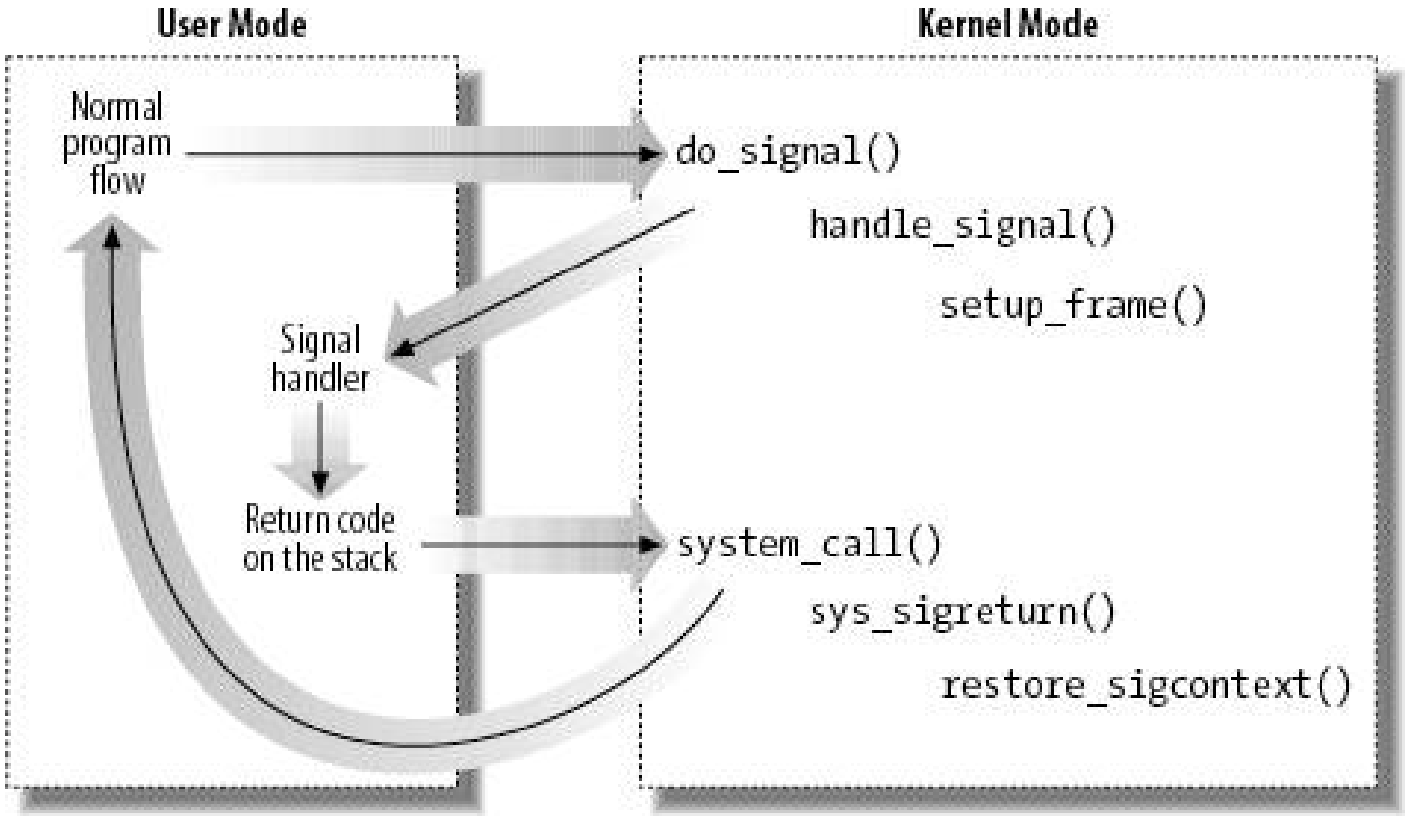
- 信号（Signal）响应时机
  - 发送信号并没有发生硬中断，只是把信号挂载到目标进程的信号 pending 队列
  - 信号执行时机：进程执行完异常/中断返回用户态的时刻



信号处理

为什么不能在内核态完成？
对于信号怎么处理是由用户态说了算的
比如Ctrl + C，登入程序也是可以响应的，**但是不能被中断**

- 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。



进程、线程和协程

Multitasking

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

User-level Thread

早期的OS只是支持了多进程，并没有支持多线程
在用户态以封装函数库的形式实现用户态线程之间的切换
这样不用进内核

- 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
- 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

并发模型与调度

并发机制：

- 内核线程：内核实现
- 用户线程：用户库实现、语言支持
- 协程：用户库实现、语言支持

并发模型与调度

上下文切换与调度器：执行流控制

- 中断上下文保存与恢复：基于中断
- 进程切换：基于时钟中断、主动让权
- 线程切换：基于时钟中断、主动让权
- 协程切换：主动让权

异常和错误处理

- 内核中断机制：硬件与操作系统协作
  - 用户态中断：硬件、操作系统和应用协作管理
- rust中的option：程序设计语言管理
- 信号：操作系统和应用协作管理

提纲

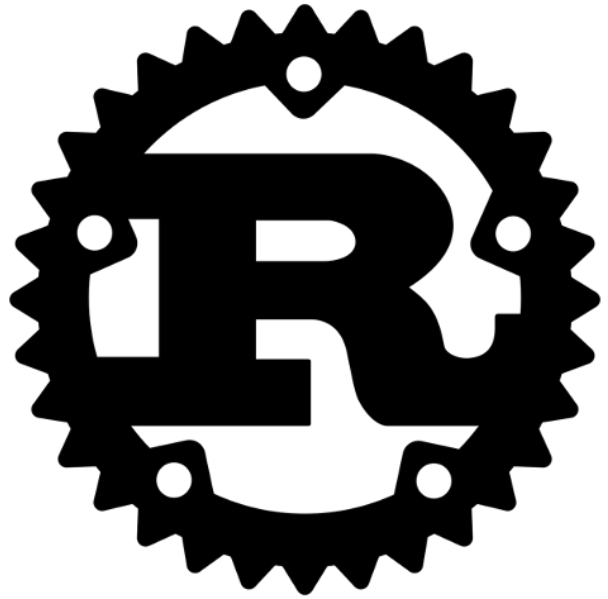
1. CPU硬件对并发的支持
2. 操作系统对并发的支持

3. Rust语言对并发的支持

4. 异步操作系统

Rust 语言历史

1. 2008年开始由Graydon Hoare私人研发
2. 2009年得到Mozilla赞助，2010年首次发布0.1.0版本，用于Servo引擎的研发
3. 2015年5月15号发布1.0版本
4. 2018年发布2018 Edition
5. 2021年2月9号，Rust基金会宣布成立。



从 2016 年开始，截止到 2021年，Rust 连续六年成为StackOverflow 语言榜上最受欢迎的语言

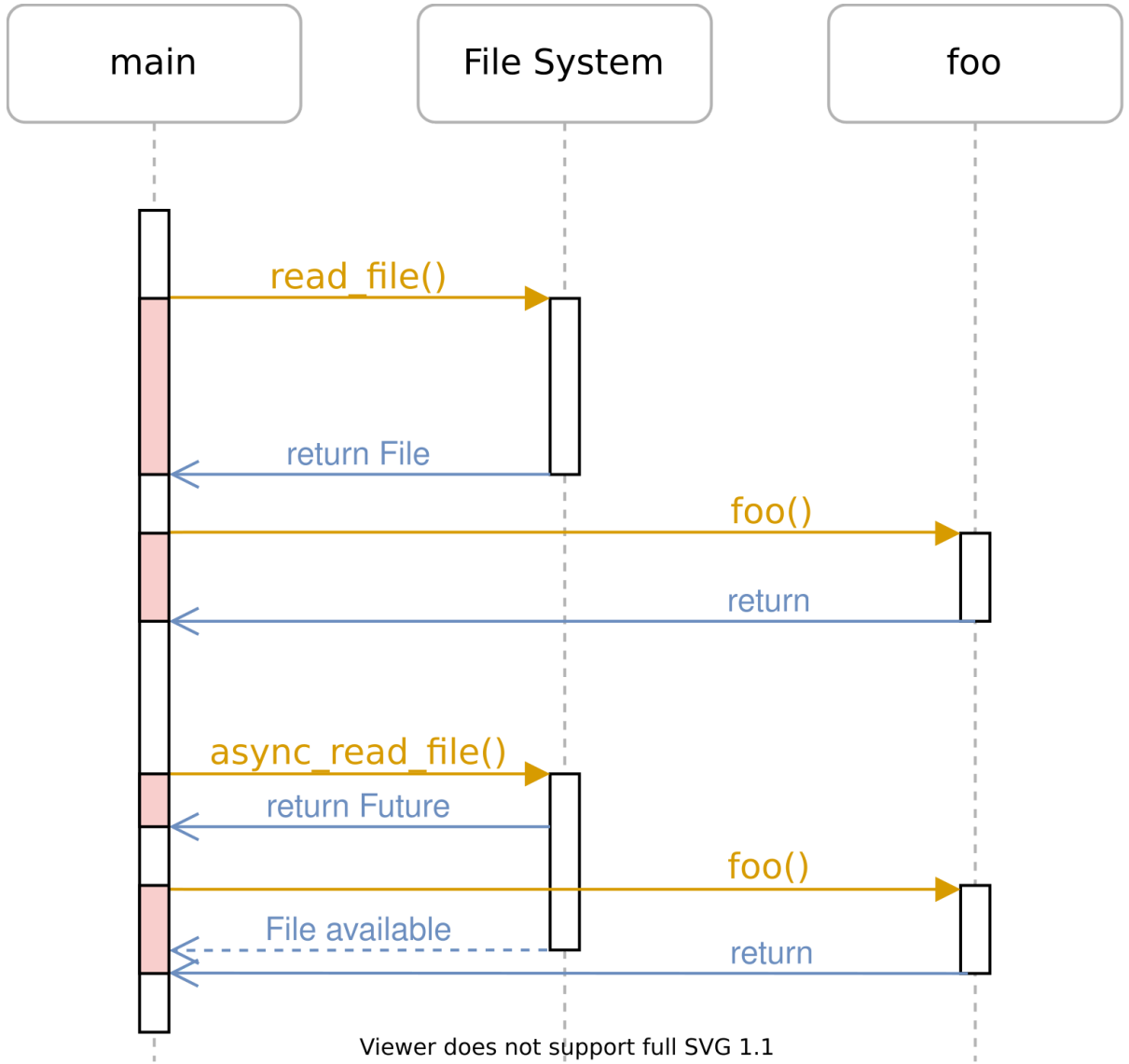
Rust 语言的目标

融合优秀语言特征，创造一门安全和性能兼备的语言。

1. 内存安全（内存访问模型抽象、类型安全的类型系统）
2. 性能（所有的细节都是可以高效控制）
3. 线程安全（并发执行模型抽象）

Concept of Future

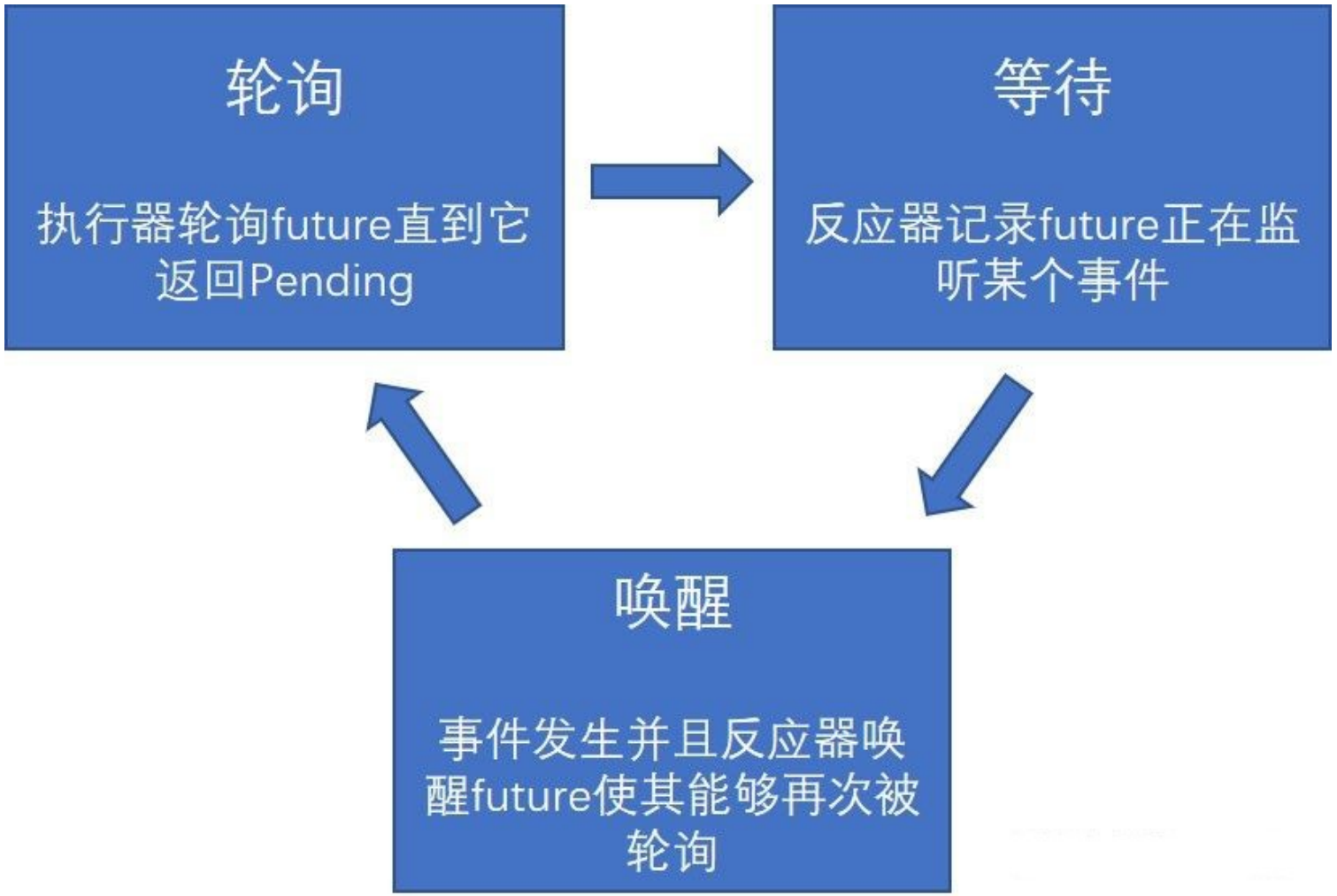
A future is a representation of some operation which will **complete in the future**.



Concept of Future

- Three phases in asynchronous task:
  1. **Executor**: A Future is **polled** which result in the task progressing
     - Until a point where it can no longer make progress
  2. **Reactor**: Register an **event source** that a Future is waiting for
     - Makes sure that it will wake the Future when event is ready
  3. **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

异步机制：green thread, coroutine (future)



提纲

1. CPU硬件对并发的支持
2. 操作系统对并发的支持
3. Rust语言对并发的支持

4. 异步操作系统

异步操作系统设计的整体目标

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. 向应用程序提供**异步系统调用**接口，优化系统调用访问性能；
4. 结合LLVM中Rust语言编译器的异步支持技术，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**；
5. 利用RISC-V**用户态中断**，优化操作系统的信号和进程通信性能；
6. 开发**原型系统**，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：有独立的**地址空间**，存有页表切换开销；
  - 在异步操作系统中，内核是一个独立的进程，有自己的页表；
  - 系统调用过程会变成一种特殊和优化的进程切换。
  - 进程切换代码是在所有进程的内核态共享的。
- 线程：有独立的**堆栈**，切换时需要保存和恢复全部寄存器。
  - 由于内核与用户线程不在一个地址空间中，每个用户线程只有用户栈，不存在对应的内核栈；
  - 每个内核线程只有内核栈，不存在对应的用户栈。

任务管理：进程、线程与协程

- 协程：可以理解为**状态机**转移函数，执行时可共用同一个栈。
  - 每个线程内可以有多个协程。
  - 编译器将 async 函数变换成状态机时，函数中需要跨越 await 的变量将存放在 Future 对象中（一般在堆上），其它变量只需放在栈上或寄存器中。

把栈的开销简化掉，只有在执行的时候才会需要使用栈，如果不执行，那就不会使用栈。
栈是空的就可以使用，栈里面有占了东西，那就没办法了

理想的协程切换过程（协程、线程和进程的调度）

协程切换可表现为不同进程和不同线程中的下一个**就绪协程选择**。

1. 协程切换：**同一进程**中**主动让权**协程间的切换；栈没内容
   - 由编译器自动生成的有限状态机切换代码完成协程切换；
2. 线程切换：**同一进程**中由于**被抢占**让权协程间的切换；栈有内容，那就不能视之为协程切换
   - 切换需要进行用户堆栈保存和切换；
   - 由内核提供的线程切换代码完成线程切换；
3. 进程切换：**不同进程**中由于被抢占让权协程间的切换；基于地址空间
   - 保存当前用户堆栈，切换到内核态，完成进程地址空间；

切换到另外一个协程时还要考虑其他进程是否有协程比目标协程的优先级高等等

理想的异步系统调用

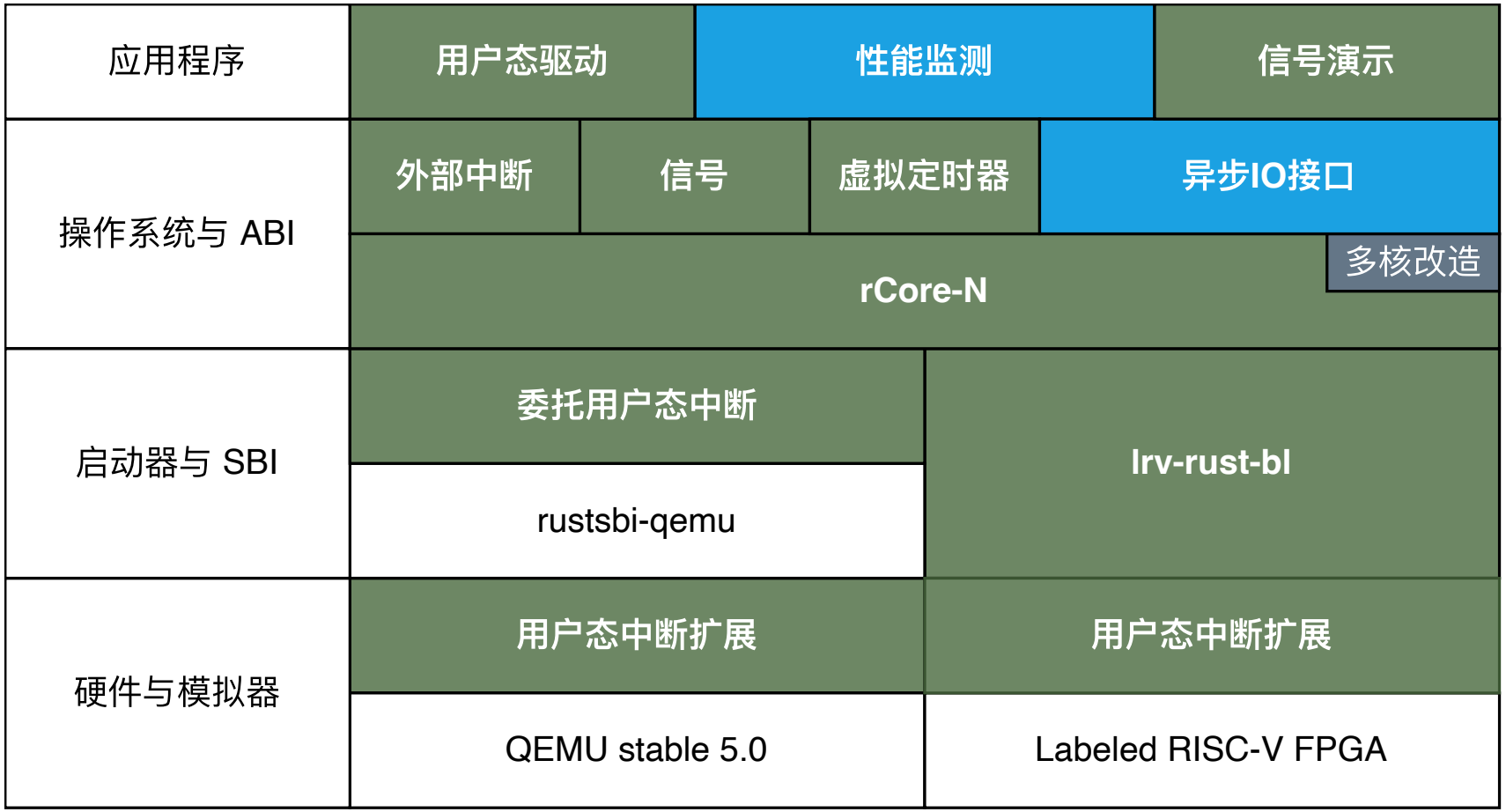
- 用户态的异步系统调用会执行**编译器自动生成**相应的系统调用请求代码，维护协程控制块数据结构；
- 在第一次系统调用请求时和最后一次系统调用完成时，需要**进入内核**；快速完成就返回结果（和正常系统调用一样），时间长返回future指针
- 中间的各次系统调用只进行系统调用的**请求提交和结果查询**，并进行进程、线程或协程切换。**跑在不同的**CPU**上，**比较好理解
- 在当前协程的系统调用还没有结果返回且**没有新的可执行用户任务**时，才会进行协程切换。

目前进展

尤予阳、贺锟鹏：RISC-V的用户态中断扩展

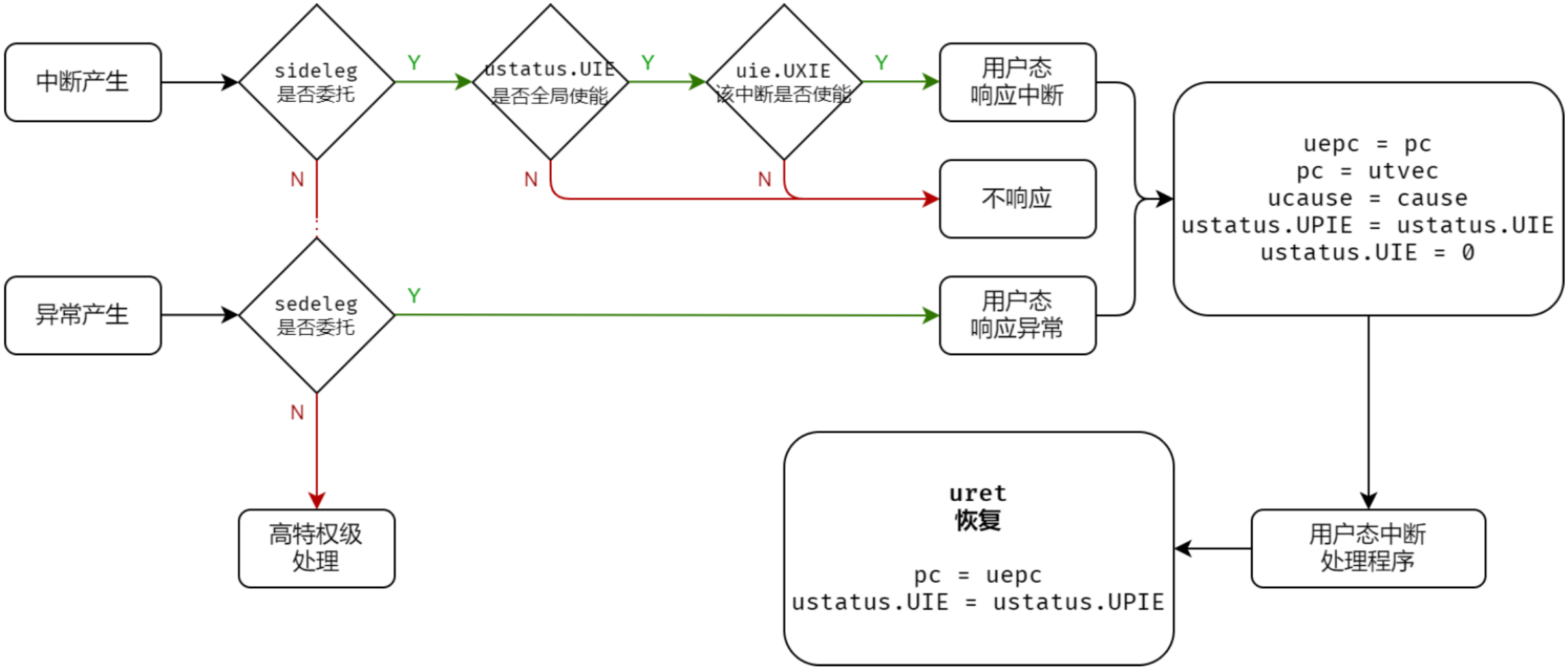
- 在QEMU和FPGA上**初步**实现用户态中断的支持：
  - 用户态中断的注册通过系统调用在内核进程进行处理。
  - 用户态中断支持的信号处理过程，可完全在用户态完成，不需要内核的参与。
- 面临的挑战：
  - 中断编号：需要标识中断的类型、中断源和中断目的方标识；
  - 中断请求和响应的硬件支持：中断目的方可能处于暂停或在不同CPU上运行；

软硬件协同的用户态中断扩展



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

用户态陷入的硬件处理流程

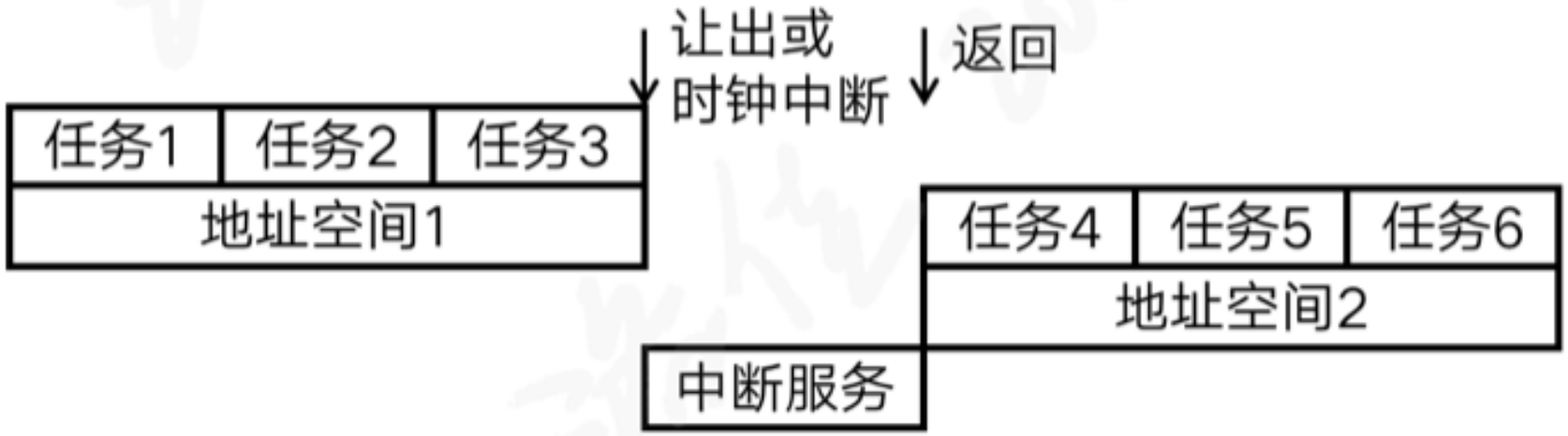


线程的实现是用户线程和内核线程，协程再这样干就没意义了。
用户态的协程切换用“编译器”，“函数库”来实现

车春池、蒋周奇：共享调度器

在操作系统层面提供协程支持

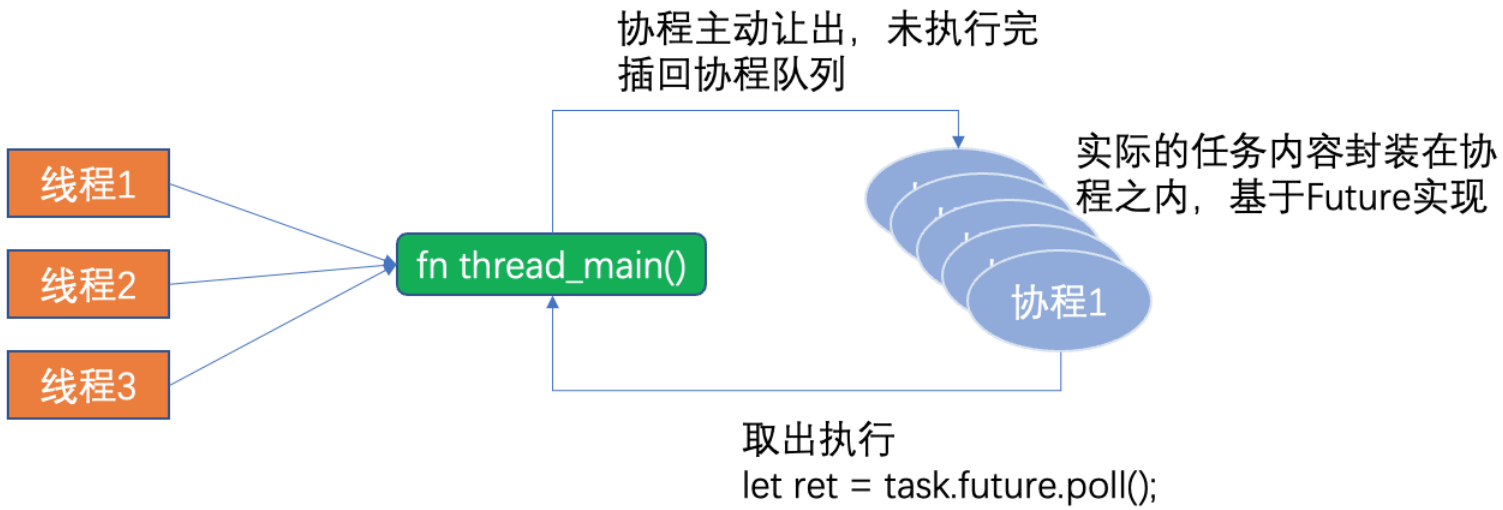
- 共享调度器直接将所使用的代码、任务池资源都共享到用户，**用户运行和内核相同的代码**。
- 用户进程与内核以相同的逻辑处理任务池中的任务。



王文智：线程与协程的统一调度

能够感知到协程的存在

1. 协程与线程灵活绑定；
2. 实现协程（future）在单CPU上并发执行；可在多CPU上并行执行；
3. 线程和协程可采取不同的调度策略；
4. 沿用线程中断的处理过程，**协程可被强制中断**；



吴非凡：异步系统调用

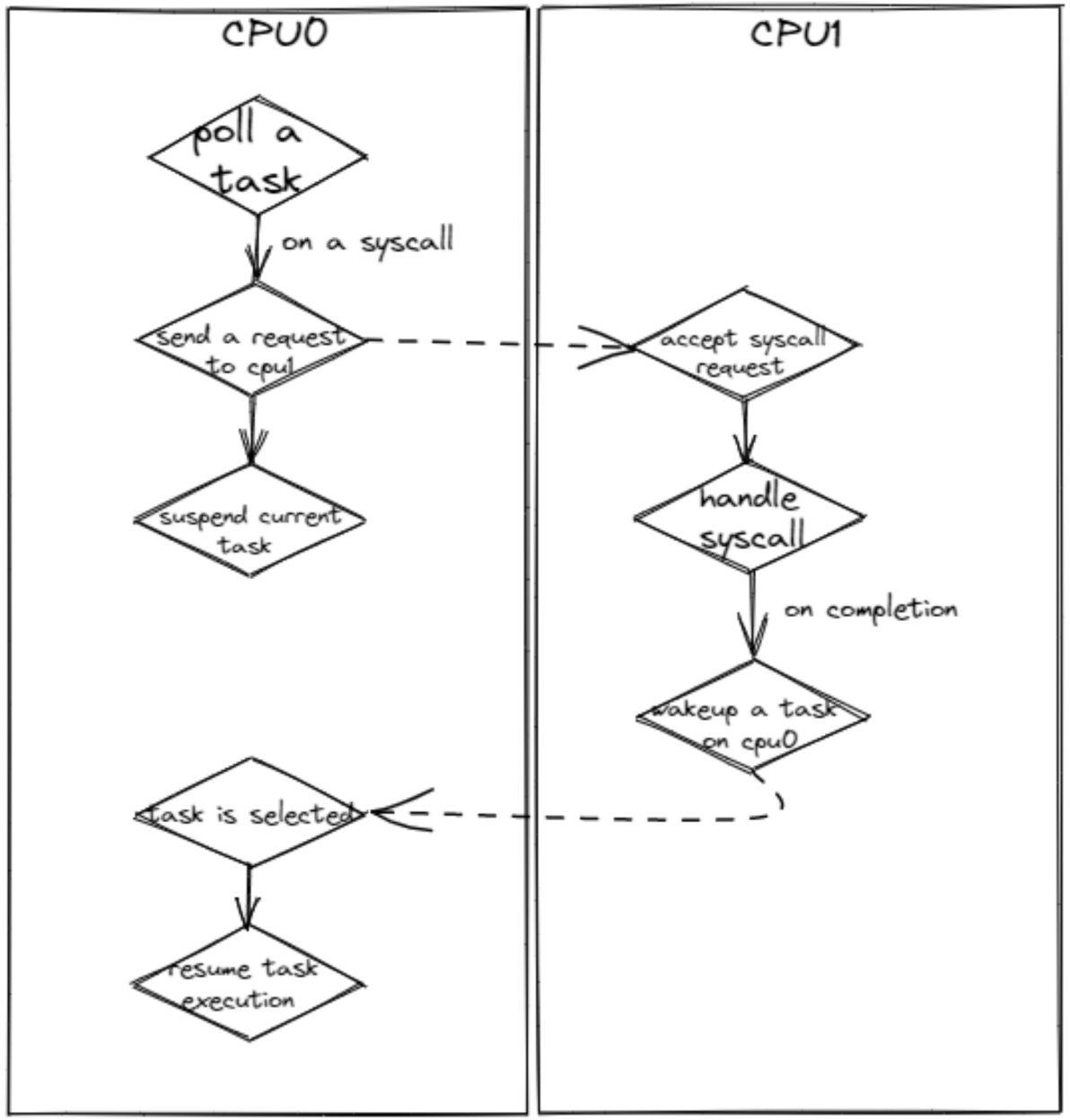
异步系统调用的用户视图

1. 用户利用异步系统调用函数和 Rust 的 async/await ,生成对应的 Future 树,交由对应的 UserExecutor 进行处理。
2. 对于其中的 Leaf Future ,在 UserExecutor 的执行流中,会发送**系统调用**,陷入内核,在内核简单注册后立即返回Pending。
3. 内核完成后,会向用户发送**用户态中断**
4. 用户态中断处理程序向 UserReactor 发送事件唤醒对应的UserTask

异步系统调用的内核视图

1. 内核陷入程序判断是 UserEnvTrap 在将寄存器参数和执行流交由内核中的 syscall 函数处理。
2. 对于有异步扩展的 syscall 函数首先判断系统调用的异步参数(编码后的用户任务号)是同步还是异步系统调用
3. 异步版本的系统调用会将生成的 Future 交给 KernelExecutor,并**返回 Future 的注册信息**(成功与否) 。
4. 陷入函数退出。

吴一凡：异步内核模块


CPU0
CPU1
poll a task
on a syscall
send a request to cpu1
accept syscall request
suspend current task
handle syscall
on completion
wakeup a task on cpu0
task is selected
resume task execution

总结

- 对于操作系统，**开源与协作**将是极具潜力的发展策略
  - 用十一年时间，RISC-V从大学项目开始，做到目前的认可程度；
  - 用十三年时间，Rust从个人项目开始，做到目前广泛关注；
- 操作系统与CPU指令集和编程语言的协作有可能带来创新的思路
  - 提高操作系统的性能
  - 降低操作系统的开发难度
  - 减少操作系统的漏洞
- 在硬件技术和编译技术的协作下，异步编程技术有可能深入影响操作系统的发展。

谢谢！