基于用户态中断的异步系统调用

开源操作系统夏令营

Kevin Axel

2021 年 8 月 29 日

RISC-V N 拓展

- RISC-V 社区提出了 RISC-V N 拓展草案

RISC-V N 拓展

- RISC-V 社区提出了 RISC-V N 拓展草案
- 贺鲲鹏、尤予阳同学基于该草案进一步完善，提出了一种符合规范的模拟器、FPGA 实现

RISC-V N 拓展

- RISC-V 社区提出了 RISC-V N 拓展草案
- 贺鲲鹏、尤予阳同学基于该草案进一步完善，提出了一种符合规范的模拟器、FPGA 实现
- GitHub 项目地址位于 https://github.com/Gallium70/rv-n-ext-impl

异步操作系统设计方案

- rCore 社区的前辈们对异步操作系统的设计

异步操作系统设计方案

- rCore 社区的前辈们对异步操作系统的设计
- github.com/async-kernel

异步操作系统设计方案

- rCore 社区的前辈们对异步操作系统的设计
- `github.com/async-kernel`
- 在内核中实现细粒度的并发安全、构件化和可定制特征

异步操作系统设计方案

- rCore 社区的前辈们对异步操作系统的设计
- `github.com/async-kernel`
- 在内核中实现细粒度的并发安全、构件化和可定制特征
- 利用 Rust 的异步无栈协程机制，优化内核的并发性能

异步操作系统设计方案

- rCore 社区的前辈们对异步操作系统的设计
- github.com/async-kernel
- 在内核中实现细粒度的并发安全、构件化和可定制特征
- 利用 Rust 的异步无栈协程机制，优化内核的并发性能
- 异步系统调用接口

异步操作系统设计方案

- rCore 社区的前辈们对异步操作系统的设计
- github.com/async-kernel
- 在内核中实现细粒度的并发安全、构件化和可定制特征
- 利用 Rust 的异步无栈协程机制，优化内核的并发性能
- 异步系统调用接口
- 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

异步操作系统设计方案

- rCore 社区的前辈们对异步操作系统的设计
- `github.com/async-kernel`
- 在内核中实现细粒度的并发安全、构件化和可定制特征
- 利用 Rust 的异步无栈协程机制，优化内核的并发性能
- 异步系统调用接口
- 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
- 利用 RISC-V 平台的用户态中断技术，优化操作系统的信号和进程间通信性能

异步操作系统设计方案

- rCore 社区的前辈们对异步操作系统的设计
- `github.com/async-kernel`
- 在内核中实现细粒度的并发安全、构件化和可定制特征
- 利用 Rust 的异步无栈协程机制，优化内核的并发性能
- 异步系统调用接口
- 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
- 利用 RISC-V 平台的用户态中断技术，优化操作系统的信号和进程间通信性能
- 开发原型系统

异步系统调用

第一次系统调用

- 用户：准备系统调用参数、发出系统调用请求
- 内核：映射共享内存、获取系统调用参数、发起相应服务协程的异步执行、返回共享内存中的服务响应队列信息给用户进程；
- 内核进程执行完服务协程后，在响应队列保存返回值；

第二次系统调用

- 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；
- 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；

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. UserExecutor 取出一个 UserTask，判断是否已经注册到 UserReactor 。对没注册的任务 poll 一次后，若为 pending 则注册到 UserReactor。对于其他注册的任务，查询 UserReactor，若准备就绪就 Poll 一次并更新在 UserReactor 的状态。
3. 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。
4. 内核完成后，会向用户发送用户态中断
5. 用户态中断处理程序向 UserReactor 发送事件唤醒对应的 UserTask

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. 对于有异步拓展的 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）是否为 0. 0 代表是同步系统调用，非零则代表是异步系统调用
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。

其他情况

- 对于单核情况。用户注册完异步系统调用并陷入内核后，由于内核优先级高，内核会不断处理 KernelTask 在处理完毕后返回用户时，直接会触发用户态中断并唤醒对应的 UserTask
- 对于多核情况。内核和用户进程位于两个硬件线程，用户进程利用异步机制，在发起系统调用后，可以避免阻塞等待，运行其他的任务。而内核也可以同时完成系统调用的内容。从而实现，用户进程和内核系统调用同时高负载运行。

进度

- ☑ 内核态运行时
- ☑ 用户态运行时
- ☑ 系统调用实现
  - ☑ pipe
  - ☑ close
  - ☑ read
  - ☑ write
- ☒ 测例
- ☒ 文档