

SharedScheduler 共享调度器

技术报告

项目成员：赵方亮、廖东海
指导老师：向勇

目录

摘要	1
第一章 背景介绍	3
1.1 用户态中断	3
1.2 Rust 语言协程机制	3
第二章 项目整体介绍	5
2.1 项目结构	5
2.2 环境配置	5
第三章 任务调度与状态转换模型	5
3.1 任务调度	5
3.1.1 进程、线程调度	6
3.1.2 协程调度	6
3.2 状态转换模型	7
3.2.1 线程内部协程状态转换	7
3.2.2 线程状态转换	8
第四章 异步系统调用	8
4.1 用户态异步系统调用接口改造	9
4.2 内核系统调用改造	10
第五章 vDSO 共享机制	10
5.1 共享方式	10
5.2 动态链接	11
5.3 全局堆分配器	11
第六章 协程调度框架	12
第七章 功能评价	12
7.1 串行的 pipe 环	13
7.1.1 同步读写环	14
7.1.2 异步读写环	15
7.2 独立的 pipe 模拟网络中的长连接	15
7.2.1 线程与协程对比	17
7.2.2 优先级设置对性能的影响	18

第八章 未来展望	19
参考文献	20
第零章 附录 在 FPGA 上的实验测试结果	21
1.1 模拟 WebServer, 线程、协程对比结果	21
1.2 优先级设置对性能的影响, 实验结果	22

图 目录

图 1.1 传统的信号机制与用户态中断的对比示意图	4
图 1.2 Rust 异步编程模型	4
图 3.1 内核、用户地址空间以及两种调度	6
图 3.2 线程、协程状态模型	7
图 6.1 协程调度框架	12
图 7.1 串行管道环实验示意图	13
图 7.2 同步读写环性能对比	14
图 7.3 模拟 WebServer 实验原理图	16
图 7.4 线程、协程模型的消息时延对比	17
图 7.5 线程、协程模型的请求吞吐量对比	18
图 7.6 不同优先级连接的时延和抖动	19
图 7.7 不同优先级连接的时延分布	20
图 A.1 线程、协程模型的时延分布	21
图 A.2 线程、协程模型的吞吐量	22
图 A.3 不同优先级连接的时延分布	23

表 目录

表 5.1 SharedScheduler 编程接口	11
表 7.1 异步读写环实验性能数据	15
表 A.1 板上 RISC-V 子系统配置	21

摘要

多线程作为解决并发问题的经典模型，存在着诸多优点且应用广泛，但也存在着一定的局限性，例如线程切换存在这开销，并且线程不能无限增加。如果 I/O 操作非常耗时，多线程很有可能满足不了高效率、高质量的需求。因此我们利用 Rust 语言对于协程机制的支持，构建了线程/协程并发模型，将协程定义为最小的任务单元，并引入内核之中，为内核和用户进程提供一套协调的任务调度机制（共享调度器）。在此基础上，我们利用了用户态中断技术与协程机制对部分 I/O 系统调用进行改造，为用户进程提供了一个完全异步的环境。此外，我们还在 Qemu 和 fpga 上进行了线程与协程的对比测试，协程在切换上的开销远小于线程。

关键字： 并发 任务调度 线程 协程 异步

Abstract

As a classical model to solve concurrent problems, multi-thread has many advantages and is widely used, but it also has some limitations, such as the overhead of thread switching, and the number of threads cannot be increased infinitely. If I/O operations are time-consuming, multi-thread may not meet the need for high efficiency and quality. Therefore, by using Rust language's support for coroutine mechanism, we built a thread/coroutine concurrency model, defined coroutines as the smallest task unit, and imported them into the kernel to provide a set of coordinated task scheduling mechanism (SharedScheduler) for kernel and user processes. On this basis, we use the user space interrupt technology and coroutine mechanism to transform part of I/O system calls, and provide a completely asynchronous environment for the user process. In addition, we also test thread versus coroutine on Qemu and fpga, coroutine switching overhead is much less than thread.

Keywords: Concurrency Task Scheduling Thread Coroutine

第一章 背景介绍

1.1 用户态中断

在传统的操作系统中，我们常常使用内核转发的方式模拟信号传输，来进行进程间通信。发送信号的进程需要进入内核态进行操作，特权级的切换造成的性能开销将导致效率低下，而接收信号的进程往往需要等待下一次被调度时才能响应中断，这意味着这信号不能被及时响应，导致延迟较高。用户态中断技术的出现，使得高效的信号传输成为可能。

“中断”的术语在之前常常用于描述源自硬件并在内核中处理的信号。即使是我们常用的软件中断（即软件产生的信号）也是在内核中处理的。但近几年出现的一种设计，希望处于用户空间的各个任务可以相互之间不通过内核，直接发送中断信号并在用户态处理。2020 年，Intel 在 Sapphire Rapids 处理器中首次发布 x86 的用户态中断功能，而 2021 年的 Linux 在 RFC Patch 中提交了对用户态中断支持的代码，我们可以以用户态的信号传输为例，对用户态中断的优势进行分析。

在传统的多核系统中，运行在两个核心的用户进程通过信号进行通信时，发送端进程需要陷入内核，向接收端进程的信号队列中插入信号，等待接受进程下一次被调度时才能响应信号；而在支持用户中断的多核系统中，发送端可以直接通过硬件接口向接收端发送信号，而接收端在核上调度时会立即响应。

从图1.1可以看出，在最好的情况下（即发送端和接收端都在核上时），用户态中断下的信号传递不需要进入内核，而且接收端也会及时响应。

1.2 Rust 语言协程机制

Rust 在早期曾支持过有栈协程，但为了更好地避免内存泄漏问题，以及更好地进行调试，Rust 在 1.0 版本不再支持有栈协，而转向了基于 `async/await` 的异步编程模型，2017 年开始在 nightly 版本对无栈协程进行支持，后续所提到的协程都指无栈协程。

Rust 编译器支持将异步函数转化为协程的能力。使用 `async/await` 语法，编译器可以将异步函数通过宏展开成一个状态机函数，该状态机在异步操作未完成时挂起，在异步操作完成时进行恢复。这种转换在编译期完成，保证了运行时的高效性和安全性。

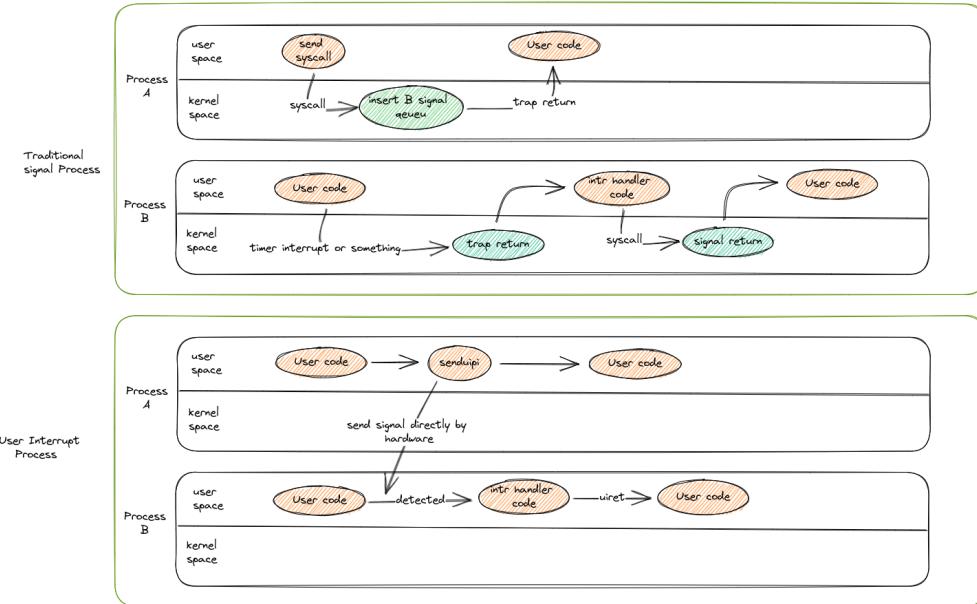


图 1.1 传统的信号机制与用户态中断的对比示意图

值得注意的是，Rust 语言本身不提供异步运行时，仅仅提供支持运行时的任务抽象 Future、异步唤醒的抽象 Waker 和异步函数转换成状态机的语法糖 async/await。第三方库往往借助 Rust 语言提供的简单抽象来实现自己的异步运行时。

Rust 异步编程模型如图1.2所示。首先用户创建任务并将任务加入全局的任务队列中，然后 Executor 中的 worker 线程会不断从任务队列中取出任务并执行，并且会创建一个 waker 用于保存任务的执行状态。当任务需要等待异步消息时，则 woker 会将对应的 waker 丢到相应的异步消息的 Reactor 中，然后执行其他任务。而在 Reactor 中当异步消息来临时，则会找到对应的 Waker 并通过 wake 操作将对应的阻塞任务重新加入到任务队列中。

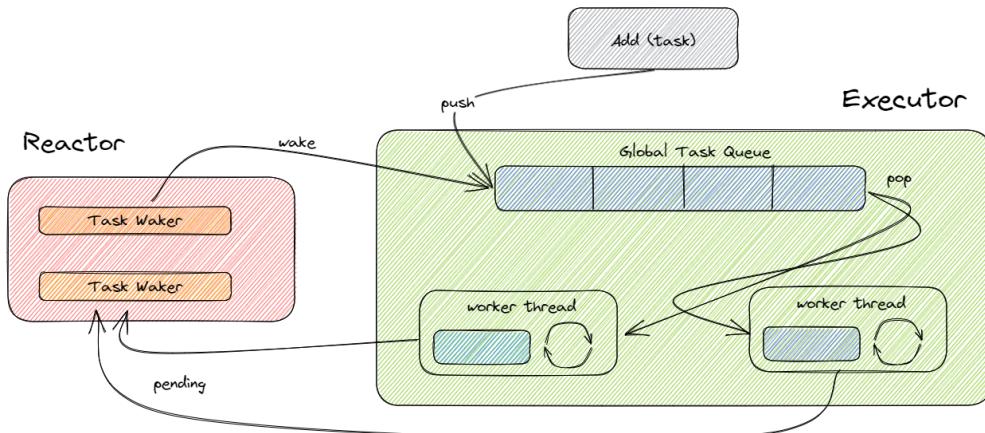


图 1.2 Rust 异步编程模型

第二章 项目整体介绍

2.1 项目结构

整个项目采取模块化的形式，由以下几个部分组成：

- os：内核代码。
- lib_so：内核与用户进程依赖的动态链接库，目前只有共享调度器依赖。
- syscall：内核态与用户态系统调用接口层，定义了内核需要实现的 syscall trait，向用户暴露系统调用接口，并以宏的形式将同步系统调用与异步系统调用统一。
- user：用户程序，包括了本文中提到的测试。

2.2 环境配置

本项目利用了用户态中断技术，因此必须运行在支持用户态中断的 Qemu 中。具体的环境部署请参考https://gallium70.github.io/rv-n-ext-impl/ch6_0_user_guide.html#qemu。

第三章 任务调度与状态转换模型

3.1 任务调度

我们保留进程控制块（PCB）、线程控制块（TCB）等数据结构不变，进程仍然负责地址空间隔离，而线程不再与特定的某个任务（函数）进行绑定，而是被视为分配给某个进程的虚拟 CPU，负责运行协程的调度函数，提供给协程代码运行所需要的栈。

同时，使用协程控制块（CCB）来描述任务信息。图3.1展示了内核和用户进程地址空间内的数据结构。其中，PCB 和 TCB 处于内核地址空间中，由内核进行管理；CCB 处于内核和用户进程地址空间内的 Executor 数据结构中，由内核和用户进程各自对其进行管理。由于这些数据结构所处的地址空间不同，而要对任务进行调度，因此整个系统的运行需要进行两种调度：(1) 由内核完成进程、线程调度；(2) 内核、用户进程内部的协程调度。图3.1展示了这两种调度。

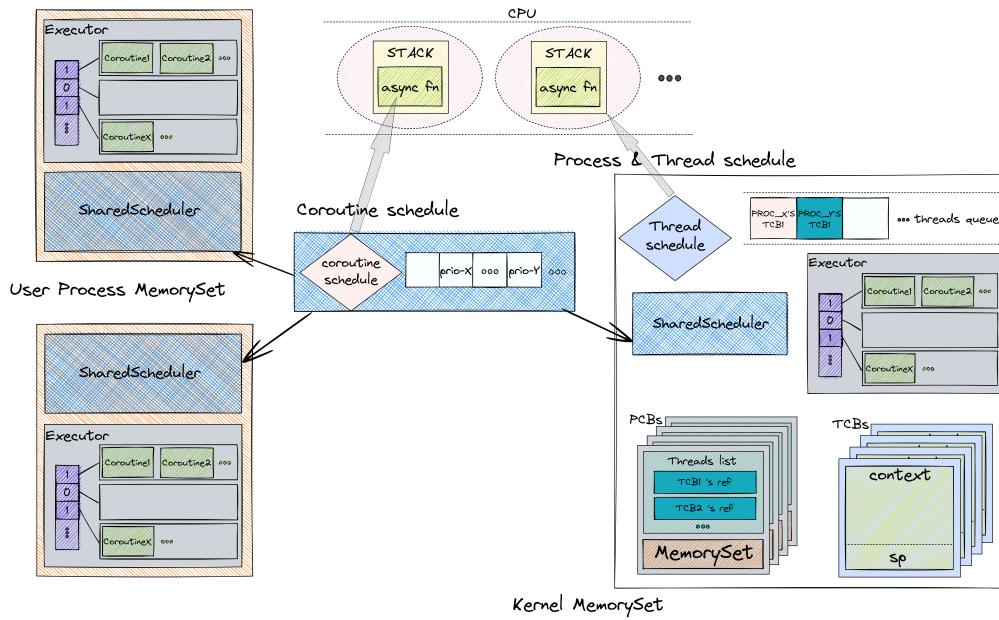


图 3.1 内核、用户地址空间以及两种调度

3.1.1 进程、线程调度

一般来说，进程、线程都需要进行调度，需要根据一定的策略，将 CPU 分配给处于就绪队列中的某个进程中的某个线程。在没有引入协程之前，线程绑定了特定的任务（函数），这个任务同样存在优先级，因此在进程调度完成之后还需要进行一次线程调度。

但是，基于上述对线程概念变化的描述，线程并不与某个具体的任务进行绑定，**TCB** 中不需要 **prio** 字段记录任务的优先级，所有线程都是等价的。因此，这一阶段不需要额外进行调度。我们将这两种调度合并在一起，由内核来完成，从所有线程的就绪队列中找到属于目标进程的位置最靠前的线程即可，如图3.1所示。

总体来说，进程、线程调度仍然需要以优先级作为依据，在进程初始化时，我们将进程的优先级设置为默认优先级，一旦进程开始运行，其优先级会动态变化，这种变化将会在协程调度中进行介绍。

3.1.2 协程调度

利用 Rust 语言提供的 `async/await` 关键字，我们可以轻易的利用协程来描述任务，同时也是由于 Rust 语言的机制，协程必须依靠异步运行时（`AsyncRuntime`）才能运行。为此，我们以共享库的形式将 `AsyncRuntime` 的依赖提供给内核和用户进程，即共享调度器（`SharedScheduler`）。

如图3.1所示，协程作为任务单元，每个协程都应该有各自的优先级，保存在协程控制块的 prio 字段中，SharedScheduler 采用了优先级位图来进行调度，位图中的每位 bit 对应着一个优先级队列，1 表示这个优先级的队列中存在就绪协程，0 表示这个优先级队列内不存在就绪协程。进行协程调度时，根据优先级位图，取出优先级最高的协程并更新位图，使其与进程内的协程的优先级始终保持一致。

上文提到的进程优先级动态变化与协程的调度相关。目前的实现是将进程优先级的实际意义理解为进程内协程的最高优先级，它记录在 SharedScheduler 内提供了一段共享内存中，实际上是优先级数组。在完成对内核或者用户进程内的协程的一次调度之后，SharedScheduler 会在优先级数组上对应的位置更新进程内协程的最高优先级。

3.2 状态转换模型

引入协程之后，任务的调度发生了变化，因此，相应的状态转换模型也产生了一系列变化。由于内核不感知协程，只负责进程、线程调度，线程的状态模型仍然是成立的。除此之外，我们还为协程建立了和线程类似的状态模型。由于协程调度建立在线程调度完成的基础上，因此协程的状态转换依附于线程的状态转换，却又会导致线程的状态发生变化，二者相互影响。如图3.2所示。

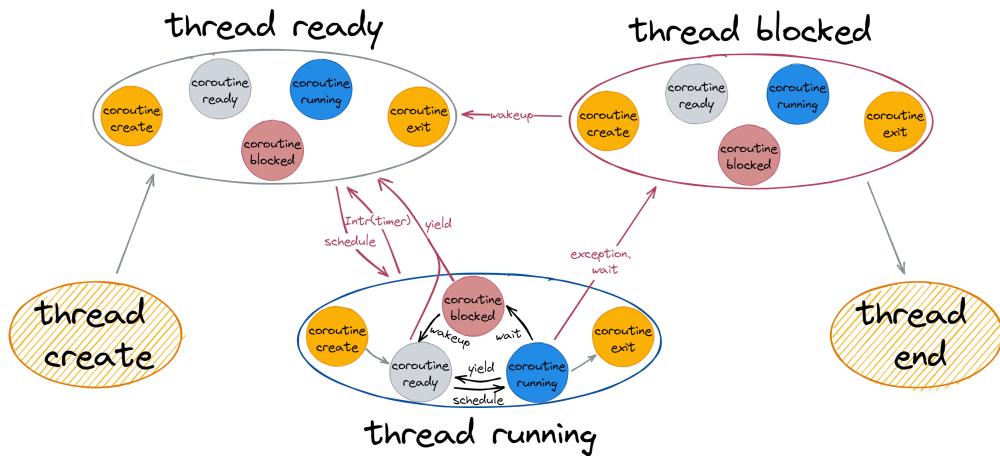


图 3.2 线程、协程状态模型

3.2.1 线程内部协程状态转换

当线程处于就绪和阻塞的状态时，其内部协程的状态不会发生变化。只有在线程处于运行状态，协程才会发生状态转换。通常，线程会由于内核执行了某些处理过程，

从阻塞态恢复到就绪态时，线程内部的协程所等待的事件也已经处理完毕，此时协程理应处于就绪的状态，因此，需要用某种方式将协程的状态从阻塞转换为就绪态。我们通过用户态中断完成了这个过程，但是，用户态中断唤醒协程目前的实现实际上は新开了一一个线程专门执行唤醒的操作，因此，协程状态发生变化还是在线程处于运行的状态。这里存在着二者相互影响的逻辑关系。

协程在创建之后进入就绪态，经过调度转变为运行态；在运行时，当前协程可能会因为等待某一事件而进入到阻塞态，也可能因为检测到存在其他优先级而让权，也可能会执行完毕进入退出状态；当协程处于阻塞的状态时，这时只能等待某个事件发生之后被唤醒从而进入就绪的状态。

3.2.2 线程状态转换

在图3.2中，不仅仅描述了线程内部的协程状态模型，同时还描述了线程状态模型以及线程、协程状态转换的相互影响关系。从图中我们可以观察到，线程仍然具有创建、就绪、阻塞、运行和结束五种状态，其状态转换与上述3.2.1节类似。需要注意的是，尽管从内核的视角看来，造成线程发生状态转移的事件仍然是调度、让权、等待以及唤醒这几类。但是，线程内部的协程状态转换给这几类事件增加了新的起因。因此，造成线程状态变化的原因具体可划分为以下几类：

1. 与内部协程无关的外部事件。例如，无论内部的协程处于何种状态，都会因为时钟中断被抢占进入就绪态。
2. 与内部协程相关的外部事件。由于内核执行了某些处理过程，完成了线程内部协程所等待的事件，使得协程需要被重新调度，线程将会从阻塞态转换为就绪态。
3. 内部协程自身状态变化。一方面是协程主动让权，当上一个执行完毕的协程处于就绪或阻塞态，需要调度下一个协程时，若此时检测出其他进程内存在更高优先级的协程，这时会导致线程让权进入就绪态；另一方面是协程执行过程中产生了异常，导致线程进入阻塞态，等待内核处理异常。

然而，无论何种原因，一旦线程状态发生变化，就会发生线程切换。这种切换需要保存 CPU 上的所有寄存器信息，是比较低效的。

第四章 异步系统调用

引入协程机制，是为了利用其异步的特性。若两个任务 A 和 B 被描述成同一线程中的两个协程，且 A 依赖 B，它们不需要进入到内核之中，仅仅在用户态完成。即

使 A 的优先级较高，先执行，也可以通过暂停，使得 B 执行相应的任务之后再唤醒 A 继续执行。这个过程体现了协程的好处。

当上述条件不变，任务 A 和 B 仍然是同一线程中的两个协程，两者都需要进入内核且 A 依赖 B。这时存在严格的执行顺序限制，A 必须在 B 执行结束之后才能执行。否则 A 一旦先进入内核执行同步系统调用等待 B 完成。在这种场景下，两个任务将会阻塞无法运行。然而，当我们完成对系统调用的改造之后，这种强的执行顺序限制也不复存在，A 先执行系统调用进入内核之后马上返回用户态并暂停，B 开始执行它的任务，结束之后再唤醒 A 继续执行。需要注意，当 A 和 B 被描述成两个线程时，上述场景提到的阻塞问题将不复存在。

我们对系统调用的改造涉及两个方面。一方面，需要将同步和异步系统调用接口进行统一，这需要对用户态系统调用接口进行形式上和功能上的修改；另一方面，内核中的系统调用处理流程需要修改。

4.1 用户态异步系统调用接口改造

对用户态系统调用接口的异步化改造，一方面需要考虑到功能上的差距，另一方面需要考虑到形式上的统一，尽可能缩小和同步系统调用的区别。此外还需要考虑改造过程中的自动化。

为了使得系统调用能够支持异步的特征，需要增加 AsyncCall 辅助的数据结构，并按照 Rust 语言的要求为其实现 Future trait，完成这个工作之后才能够在调用系统调用时使用 await 关键字。

我们力求缩小形式上的差距，考虑到异步系统调用更加复杂，因此，采取的策略是同步系统调用向异步看齐，我们利用了 Rust 语言中的过程宏，帮助生成同步系统调用和异步系统调用。

```
#[async_fn(true)]
pub fn read(fd: usize, buffer: &mut [u8], key: usize, cid: usize) -> isize {
    sys_read(fd, buffer.as_mut_ptr() as usize, buffer.len(), key, cid)
}

#[async_fn]
pub fn write(fd: usize, buffer: &[u8], key: usize, cid: usize) -> isize {
    sys_write(fd, buffer.as_ptr() as usize, buffer.len(), key, cid)
}
```

最终，同步和异步系统调用在形式上达到高度统一，唯一的区别在于参数的不

同。

4.2 内核系统调用改造

除了在用户层系统调用接口上的形式一致，我们还追求内核系统调用处理接口上的一致。最终，内核通过判断系统调用参数，选择执行同步或异步处理逻辑。其中异步的处理逻辑是内核采取某种方式使得任务立即返回用户态，而不需要同步的等待相应的处理流程执行完毕再返回用户态。在内核异步的完成相应处理过程之后，再唤醒对应的用户协程。

以 `read` 系统调用为例，进入内核之后，若执行异步处理，则是将这些同步的处理过程封装成协程，将这个内核协程添加到内核的 `Executor` 中，再返回到用户态。

以异步读管道操作为例，异步系统调用返回到用户态之后，内核的处理流程被封装成协程，但并未执行，需要等待内核完成处理之后，才会被唤醒执行。内核执行这个协程，完成相应的处理过程之后，会发起用户态中断，传递需要唤醒的用户协程 ID，由用户态中断处理函数唤醒对应的协程。

第五章 vDSO 共享机制

若要用户进程能够实现协程调度运行，最直白的方式是以静态库的形式向用户进程提供依赖，但这种方式存在着弊端：编译时，这些依赖会被编译进 ELF 镜像中，使得内存空间开销增大。这显然是不明智的做法。因此，我们以 vDSO 的方式将 SharedScheduler 共享给内核和用户进程使用。它对外完全透明，仅仅暴露出一组接口供内核和用户进程使用，其内部集成了维护 `Executor` 数据结构的操作和协程调度算法。然而，在裸机的环境下，实现 vDSO 存在着一些挑战。

5.1 共享方式

根据 vDSO 官方手册上的描述，vDSO 是由内核自动链接进用户进程地址空间的共享库。然而，Rust 无法编译出 `riscv64gc-unknown-none-elf` 架构的共享库文件。但是，其中的关键点是内核先查找符号表，再进行链接。

因此，我们将 SharedScheduler 编译成可执行文件格式，内核将其加载进自己的地址空间中进行维护，并且能够查找符号表信息找到对应的接口信息。在创建进程时，由内核将它映射到用户进程地址空间中，使得用户进程拥有执行相应的接口的权限。这种实现方式的缺陷在于必须将 SharedScheduler 映射到同一段虚拟地址上，牺

牲了灵活性。

5.2 动态链接

在标准库环境下，若在 Rust 中使用共享库，需要先声明外部函数，最后再进行链接。我们仿照这种方式，利用 `get_libfn!` 宏来声明外部函数，使得在编写用户程序时能够使用 SharedScheduler 暴露出来的共享库。

这个过程宏实际上会针对每个声明的外部函数，创建带有 `VDSO_` 前缀的符号，内核创建用户进程时，解析进程对应的符号表，查找其中带有 `VDSO_` 前缀的符号，再从内核维护的 SharedScheduler 符号表中查找到对应的符号地址并进行链接。这种方式不仅仅能够支持更多的共享模块，也能支持标准的共享目标文件。最终，我们通过不同的 feature 向内核和用户暴露出不同的接口，见表5.1。

表 5.1 SharedScheduler 编程接口

接口	功能	feature
<code>user_entry</code>	用户进程入口	<code>kernel</code>
<code>spawn</code>	添加协程	<code>kernel</code> 、 <code>user</code>
<code>current_cid</code>	获取当前正在运行的协程 ID	<code>kernel</code> 、 <code>user</code>
<code>wake</code>	唤醒目标协程	<code>kernel</code> 、 <code>user</code>
<code>reprio</code>	调整目标协程优先级	<code>kernel</code> 、 <code>user</code>

5.3 全局堆分配器

Rust 使用默认的全局分配器（global allocator）在堆上进行分配、释放内存，并且它将协程上下文固定在堆上避免自引用问题。因此，SharedScheduler 与堆之间的联系非常紧密，必须能够动态的完成对内核或用户进程堆的管理。存在两种方式能够达到这种目的。第一种方式是显式的将堆指针当作参数传递给 SharedScheduler 接口；第二种方式是采用某种方式使得全局分配器能够动态的指向内核和进程各自的堆。对于普通的数据，第一种操作可以很好的解决问题，但是对于堆这种系统分配的数据，用户不能对其进行显式的操作。因此只能采取第二种方式，我们将内核和用户进程的堆映射到同一个虚拟地址，最终解决了动态操作数据的问题。同理，我们以相同的方式解决了动态使用 Executor 的问题。

第六章 协程调度框架

内核和所有的用户进程能够访问 SharedScheduler 暴露的共享库。但这些操作最终会落实到对其内部的 Executor 数据结构的操作上。为了避免在内核和用户库中重复定义 Executor 数据结构，我们在独立的 lib 中定义 Executor，并为其实现某些成员方法。

按照 Rust 的特性，仅仅在内核和用户进程中创建 Executor 数据结构而不使用其成员方法，那么内核和用户程序编译生成的 ELF 文件中将不会包含这些成员方法。而在 SharedScheduler 中，我们通过调用 Executor 的成员方法完成对它内部的操作。这样一种特性，使得我们既能保留住共享的特点，又能方便的实现模块化的协程调度框架，从而通过不同的 feature 使用不同的协程调度算法，见图6.1。

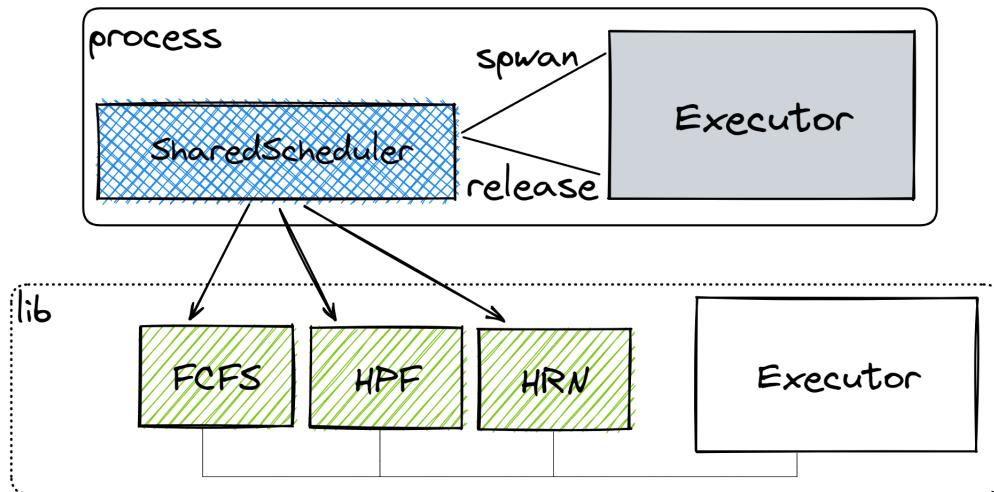


图 6.1 协程调度框架

第七章 功能评价

我们的项目脱胎于 AmoyCherry 的 AsyncOS 开源项目，采用了相同的优先级的调度算法，但在调度器的内核与用户态复用、异步协程唤醒、内存分配方面有了不同程度的改进，同时也扩展了调度器对多核多线程的支持。为了衡量这些因素对性能的影响，我们设置了串行的 pipe 环实验与 AsyncOS 进行性能对比。

此外，为了展示协程相比于线程所拥有的低时延的切换开销，我们利用协程和线程搭建了两个不同模型的 WebServer 来测试服务器的吞吐量、消息时延和抖动的情况，由于当前的实验框架暂不支持 socket 网络通信，我们使用 Pipe 通信来模拟网络中的 TCP 长连接。

最后，我们还将通过 WebServer 的实验展示协程优先级对任务的吞吐量、消息时延和抖动的影响，展示优先级在有限资源下保障某些特定任务实时性上的巨大作用。

测试代码版本：rCore-N

7.1 串行的 pipe 环

仿照 AsyncOS 的串行管道环实验，我们将若干个管道的读写端首尾相连，形成一个环状结构，如图7.1所示，我们将一读一写封装成一个协程，然后将第一个协程叫做 server 0 端，负责第一次写入输入来启动整个管道环，其余的负责传递消息的协程叫做 server i 端，server 0 经过第一次点火写入之后，由各个 server 端协程进行消息传递，最终回到 server 0 的读端。

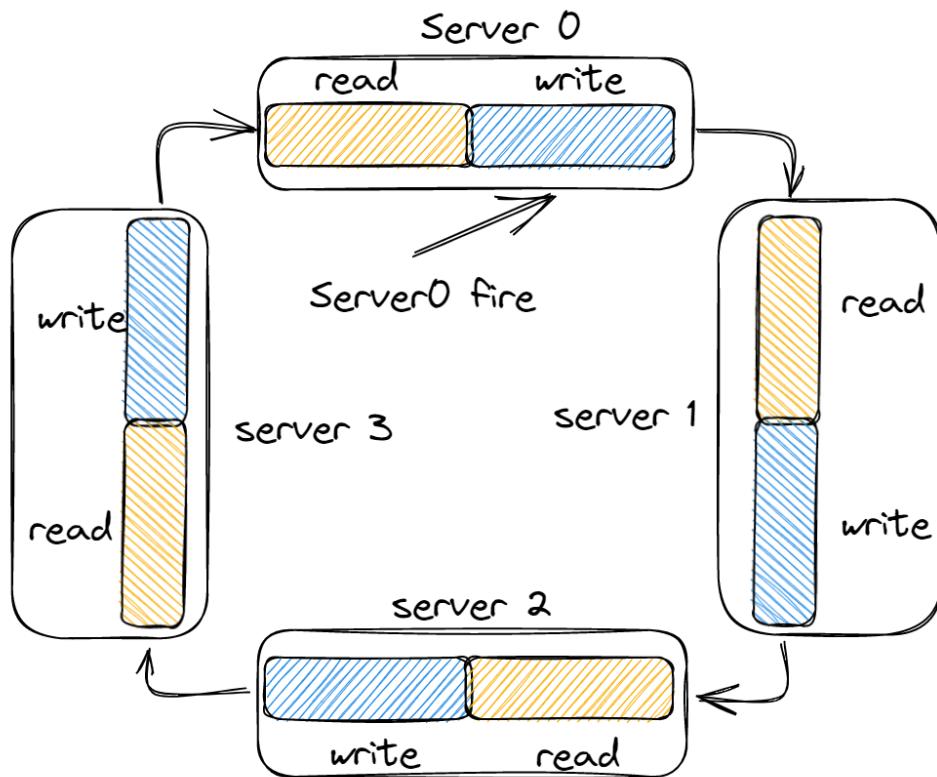


图 7.1 串行管道环实验示意图

我们的内核同时支持同步的系统调用和异步的系统调用，根据协程中管道的读写方式，我们将上面的环分为同步读写环（读写都使用原始的系统调用接口，直到系统调用完成后才返回用户态）和异步读写环（异步读将创建内核协程任务，创建完内核协程任务后立刻返回用户态，而内核协程由内核在空闲时执行）。

7.1.1 同步读写环

我们在单 CPU 的情况下，分配 1 个线程处理协程，测量了 1 个串行 pipe 环的执行时间，并与 AsyncOS 的性能进行对比。实验对应的测试文件为 user/src/bin/async_pipe.rs，变化参数包括：

- 管道环的长度
- 传输数据的大小
- 读写方式（同步和异步）

同步读写环的性能对比结果如图 7.2 所示，其中 AsyncOS-228 表示在 AsyncOS 中实现同步读写环并在环中传递长度为 228 字节的数据，SharedScheduler-912 表示在共享调度器中实现同步读写环并在环中传递长度为 912 字节的数据，其他以此类推。

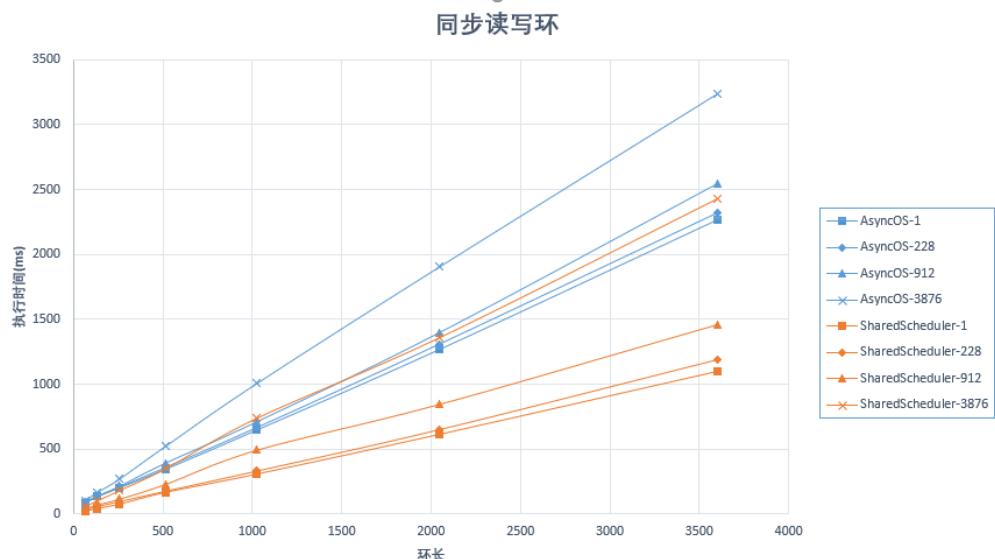


图 7.2 同步读写环性能对比

可以从中看出，传递的时间随着环长的增加而线性增加。同时，传递消息越长，会导致系统调用中的读写时间越长，导致执行时间越长。

此外，我们可以看出，在环长和传递消息长度相同的情况下，共享调度器的执行时间总是小于 AsyncOS 的。在单线程（不涉及多核）和同步读写（不涉及内核协程和异步唤醒）的场景下，当前的性能差距体现了调度算法本身的性能差距。主要体现在以下两点：

- 在 AsyncOS 中，每次时钟中断都会去收集所有的进程的优先级然后更新全局的优先级位图，即使进程的优先级没有发生变化；而在共享调度器中，更新全局位图在用户态进行添加协程、删除协程等会改变当前进程优先级的操作时才会进行，保

证了按需更新的原则，避免了无谓的时间消耗。

- 在 AsyncOS 中，在每次时钟中断陷入内核时都会去检查内核的协程调度器是否有任务，而检查的方法是遍历所有优先级的就绪队列是否为空，如果有 8 个优先级则至少遍历 8 次；而在我们在共享调度器中维护了当前调度器中最高优先级，只需检查最高优先级是否为 PRIO::MAX 即可判空。

由于时钟中断的发生频率极高，因此这两点是共享调度器和 AsyncOS 在此场景下的性能差距的关键。

7.1.2 异步读写环

异步读写环的性能对比和上图类似，只是共享调度器和 AsyncOS 的性能差距被拉大，表7.1是在异步读写环中传递长度为 3876 字节的数据的性能数据：

表 7.1 异步读写环实验性能数据

环长	16	32	64	128	256	512
AsyncOS-3876	202ms	376ms	715ms	1401ms	2767ms	5502ms
SharedScheduler-3876	43ms	61ms	90ms	181ms	294ms	573ms

从中可以看出，AsyncOS 和共享调度器的性能差距扩大为近 10 倍。在异步读的场景下，涉及到内核协程和异步唤醒的处理，因此除了上面提到的两点影响因素外，还有如下的因素：

- AsyncOS 的内核唤醒用户态协程，将用户态协程的编号插入到 callback 队列里，等到用户态调度器在获取新的协程之前，总是要先遍历自己的 callback 队列，把 callback 队列中的协程编号插回就绪队列中；而共享调度器则是通过用户态中断来将唤醒的协程编号插回就绪队列中，无需再每次调度时检查 callback 队列。
- AsyncOS 在用户态调度器无法取到就绪的协程任务时，会陷入忙等，直到时间片用完；而共享调度器则会主动让权，将计算资源交给内核协程和其他进程。

7.2 独立的 pipe 模拟网络中的长连接

前文对比并分析了我们的共享调度器与 AsyncOS 的性能差异，为了展示我们的协程模型相比于线程模型所拥有的低切换开销的优势，在本节中我们用线程模型和协程模型模拟实现了一个用于长连接通信的 WebServer 服务器并对这两种模型的性能进行了对比，实验框架如图7.3所示：

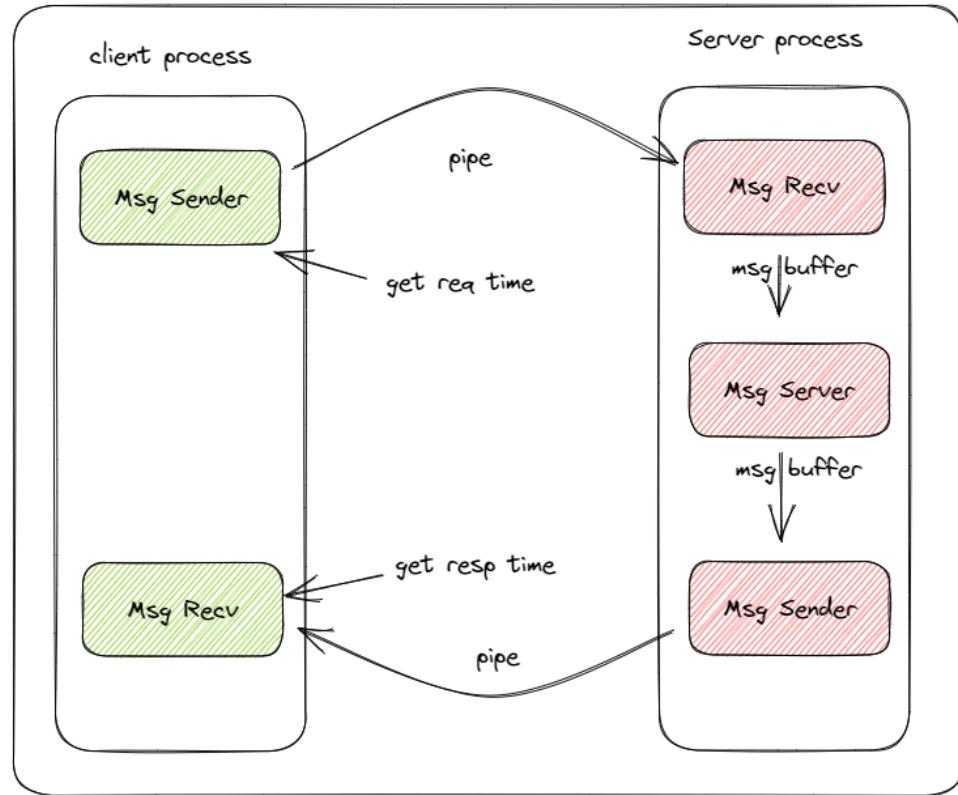


图 7.3 模拟 WebServer 实验原理图

由于当前的开发环境暂不支持 socket 通信，我们使用两个 pipe 来模拟一个连接的双工通信，client 和 server 是两个不同的进程，通过 pipe 实现跨进程通信。图中展示了一个长连接的组成部分。在客户端，Msg Sender 负责定时发送一定长度的数据到 Server Process，并生成一个消息的定时器来进行计时，Msg Recv 则负责从服务端接收响应并根据定时器统计响应时间。而服务端这边一个连接则是由三个部分组成，分别是消息接收（Msg Recv，负责将接收到的消息展开成矩阵）、消息处理（Msg Server，负责进行矩阵运算）和消息响应（Msg Sender，负责将矩阵转成字节流消息返回客户端），这三者之间通过共享的消息缓冲区传递消息。

我们的目的是为了测试服务端的性能，因此为了减少客户端对整个系统的影响，我们选用占用内核资源较少的协程模型来实现客户端，同时为了保证 client 的定时发送准确以及接收及时，我们对 Client process 的两个部分的协程都设置最高优先级。

值得注意的是，我们衡量的是切换的开销，因此我们将从一开始就建立好连接，并将协程和线程创建好后添加到框架中。下面，我们将从 5s 内吞吐量、消息时延和时延抖动三个方面来分析服务端的性能。

7.2.1 线程与协程对比

为了衡量协程与线程在切换上的开销，我们分别用线程模型和协程模型实现了 Server Process 的三个部分。我们用 Thread-1 表示线程模型下处理 1×1 矩阵请求的测试结果，用 Coroutine-20 表示协程模型下处理 20×20 矩阵请求的测试结果，其他以此类推。实验在 4 个物理 CPU 上进行，客户端分配 2 个虚拟 CPU，以 50ms 为周期发送请求，服务端分配 4 个虚拟 CPU。对应的测试文件为 user/src/bin/connect_test.rs 和 user/src/bin/connect_thread_test.rs。变化的参数为：

- 服务端的实现模型（协程模型和线程模型）。
- 连接数量。

消息时延的测试结果如图7.4所示。可以看出，线程模型在连接数较少时，消息时延和协程模型持平，甚至略低于协程模型，这是因为在连接数较少时内核直接调度线程来执行任务，而协程模型多了一层调度器的同步互斥，导致时延略高；而随着连接数的增加，线程模型的延迟迅速上升，远高于协程模型，这是由于协程的切换大都不需要陷入内核，只需要切换函数栈即可，切换开销远小于线程；对比相同模型下的不同规模的矩阵请求，发现请求的矩阵规模越大，时延越高，这是由于大规模的矩阵请求代表着消息发送和接收的开销变大，消息处理的开销也变大，时延理所应当增加，但我们更应该注意到，协程模型下，随着矩阵规模增加导致时延增加的幅度小于线程模型，这是因为线程模型下的同步读写遇到阻塞时是将阻塞的线程重新插入调度的队尾，等待下一次调度时再判断是否还是阻塞等待，判断任务是否就绪采用的是不断轮询的方法，这将导致许多无效的切换，而协程模型则是由 reactor 通知调度器等待事件发生，一个阻塞点只会判断两次（第一次阻塞，第二次唤醒），避免了无效的轮询和切换。消息抖动的测试结果与时延的结果类似。

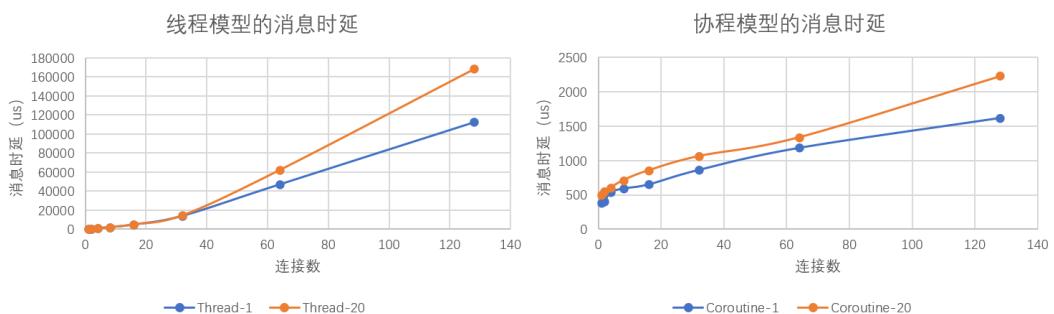


图 7.4 线程、协程模型的消息时延对比

图7.5展示了不同模型下的不同矩阵请求规模的总吞吐量的测试结果。可以看出

协程模型下的服务端随着连接数的增加，总的吞吐量也随之线性增加，即使矩阵请求的规模增加，由于负载远远没有达到峰值（客户端每 50ms 发送一个请求，而上图所示的时延最高不过 3ms），所以吞吐量取决于连接数和客户端的请求频率。而对于线程模型下的服务器，在连接数较小时尚能与协程模型持平，但是连接数增加之后，切换开销迅速增大，导致了总吞吐量增加趋势减缓，而对于 Thread-20，在连接数为 64 左右的负载就几乎到达了峰值（上图的消息时延为 60ms 左右，客户端每 50ms 发送一个请求）。

线程和协程模型的吞吐量对比

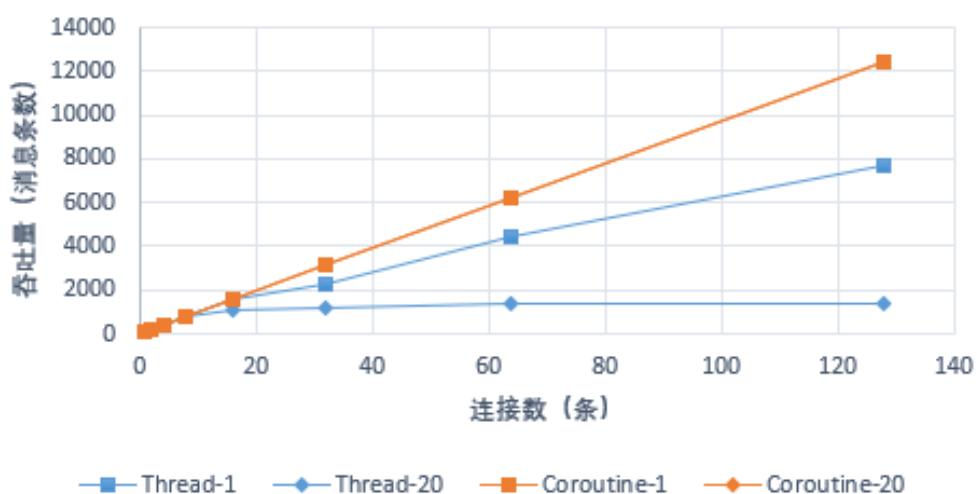


图 7.5 线程、协程模型的请求吞吐量对比

7.2.2 优先级设置对性能的影响

在计算机系统中，CPU 资源和 IO 资源总是有限的，在有限的资源条件下，我们可以通过设置优先级来优先保证某一些服务。在 WebServer 的场景中，我们可以将各个连接的优先级按照阶梯式进行设置，来保证某些连接的更低的时延和更小的抖动。我们的实验在 4 个物理 CPU 上进行，客户端分配 1 个虚拟 CPU，以 100ms 为周期发送 1600 字节的请求消息（即矩阵规模为 40×40 ），服务端以协程的形式实现。客户端与服务端建立了 126 个连接（均分为 7 个优先级），分别测试在 CPU 资源不同的情况下不同优先级的连接的性能。实验对应的测试文件为 user/src/bin/connect_test_with_prio.rs，变化的参数为：

- 不同优先级的连接。
- 分配给服务端的虚拟 CPU 个数。

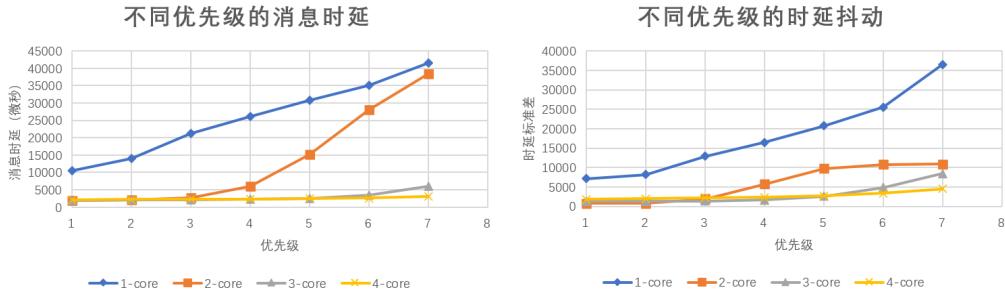


图 7.6 不同优先级连接的时延和抖动

测试结果如图7.6所示，图例中的 x-core 表示给服务端分配了 x 个虚拟核心（内核线程）。从中可以看出，在资源紧张的时候，只能优先保证优先级高（优先级高的连接，优先级数小）的连接，随着优先级的降低，时延和抖动都会增大。随着资源数量的增加，系统也能够保证优先级相对较低的连接有较低的时延和抖动，但仍然遵循最高优先级有最低时延和抖动的要求。同时，我们还注意到，随着虚拟核心的增加，在核心数较少时就能保证的高优先级连接会出现一点性能下降，但仍在可以忍受的范围内，这是由于虚拟核心的增加造成了调度器的同步互斥的加剧，在未来通过引入多级就绪队列可以缓解这个问题。

我们进一步分析，在资源十分有限的情况下，各个优先级连接的消息时延的分布情况如图7.7所示。可以看出，最高优先级（如优先级为 1 和 2）的连接的消息时延分布十分集中，该服务得到了有效的保证，然后优先级较低的连接的时延分布比较分散，波峰随着优先级降低而延迟出现。

第八章 未来展望

目前的共享调度器还停留在初级的阶段，麻雀虽小，但五脏俱全，在这个项目的基础上，我们还可以进行横向与纵向的扩展。

一方面，可以把共享调度器构建出的异步执行环境应用到底层的驱动上，实现更高效的设备驱动，也可以考虑将协程用于用户进程的异常处理，从这些方面进行横向扩展。

另一方面，目前的进程、线程和协程调度算法还存在着很大的扩展空间，可以考虑从调度算法的性能上进行纵向的扩展。

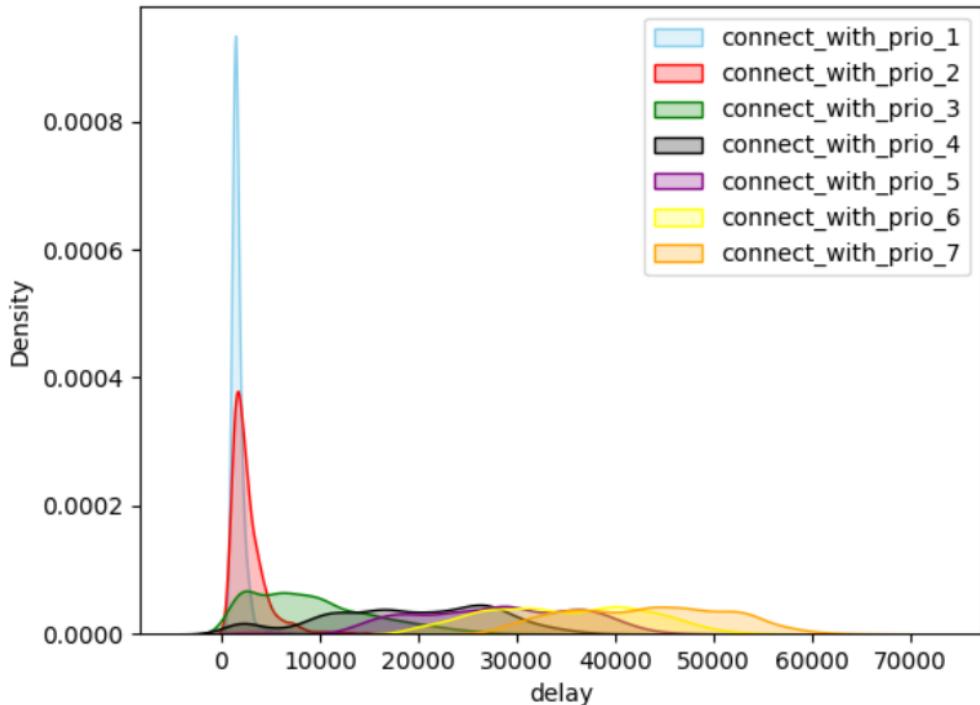


图 7.7 不同优先级连接的时延分布

参考文献

- [1] gallium70. Risc-V Extension N Implementation [EB/OL].<https://gallium70.github.io/rv-n-ext-impl/intro.html>.
- [2] stevenbai.Introduction-Futures Explained in 200 Lines of Rust [EB/OL]. <https://cfsamson.github.io/books-futures-explained/introduction.html>.
- [3] The Rust RFC Book [EB/OL].<https://rust-lang.github.io/rfcs/2033-experimental-coroutines.html>.
- [4] x86 User Interrupts support - Sohil Mehta [EB/OL].<https://lore.kernel.org/lkml/20210913200132.3396598-1-sohil.mehta@intel.com>.

附录

第一章 在 FPGA 上的实验测试结果

实验板型号为 zcu102, 搭载的 FPGA 型号为 Zynq UltraScale+ XCZU9EG-2FFVB1156 MPSoC。板上 RISC-V 子系统配置参数见下表:

表 A.1 板上 RISC-V 子系统配置

cores	Frequency	BTB entries	L2 cache size	Memory size
4	100 MHz	40	2 MB	2 GB

测试代码: rCore-N下 utest 目录。

1.1 模拟 WebServer, 线程、协程对比结果

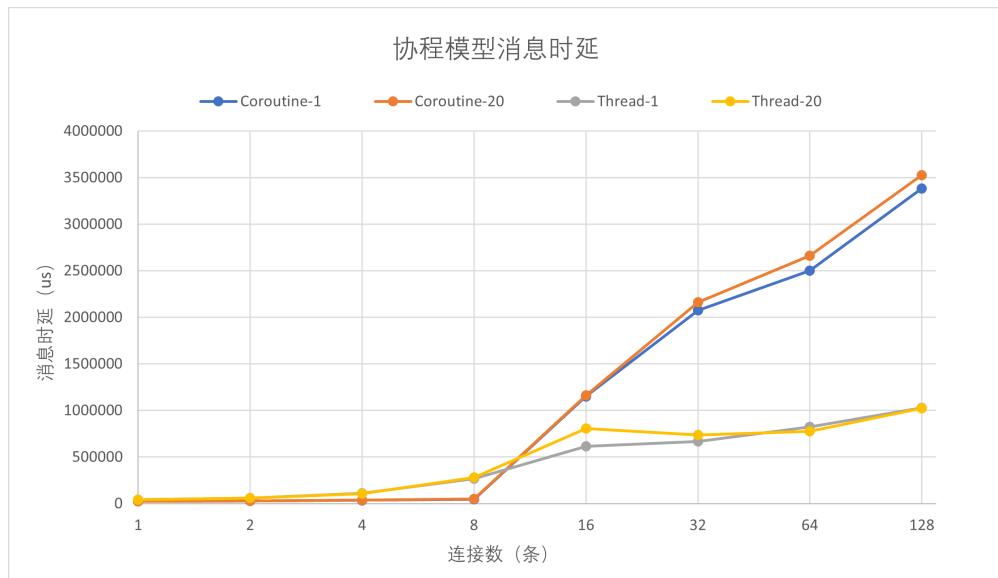


图 A.1 线程、协程模型的时延分布

与在 Qemu 中的测试相同, 实验在 4 个物理 CPU 上进行, 客户端分配 2 个虚拟 CPU, 以 50ms 为周期发送请求, 服务端分配 4 个虚拟 CPU。对应的测试文件为 utest/src/bin/ct.rs 和 utest/src/bin/ctt.rs。变化的参数为:

- 服务端的实现模型 (协程模型和线程模型)。
- 连接数量。

见图A.1和图A.2。连接数较少时，线程模型和协程模型的吞吐量没有区别。当连接数为1时，线程模型的消息时延是协程模型的1.75X，随着连接数增加至8，线程模型的消息时延达到最高，为协程模型的5.7X。因此，实验充分体现出协程在切换上的开销远低于线程。

由于FPGA上的时钟频率比Qemu低，所以在连接数超过8时，实验的负载较重，但线程模型的消息时延反而比协程低。我们进一步的分析之后，得出的结论是：由于在内核中的I/O协程的优先级最高，因此内核会在处理完I/O任务之后才进行调度，从而影响到客户端和服务端的线程被调度的次数。协程模型相较于线程模型会增加内核I/O协程数目，因此会导致客户端无法及时被调度从而导致消息时延增加。线程模型下，内核的I/O协程较少，因此客户端和服务端被调度的次数会增多，从而导致消息时延减少。

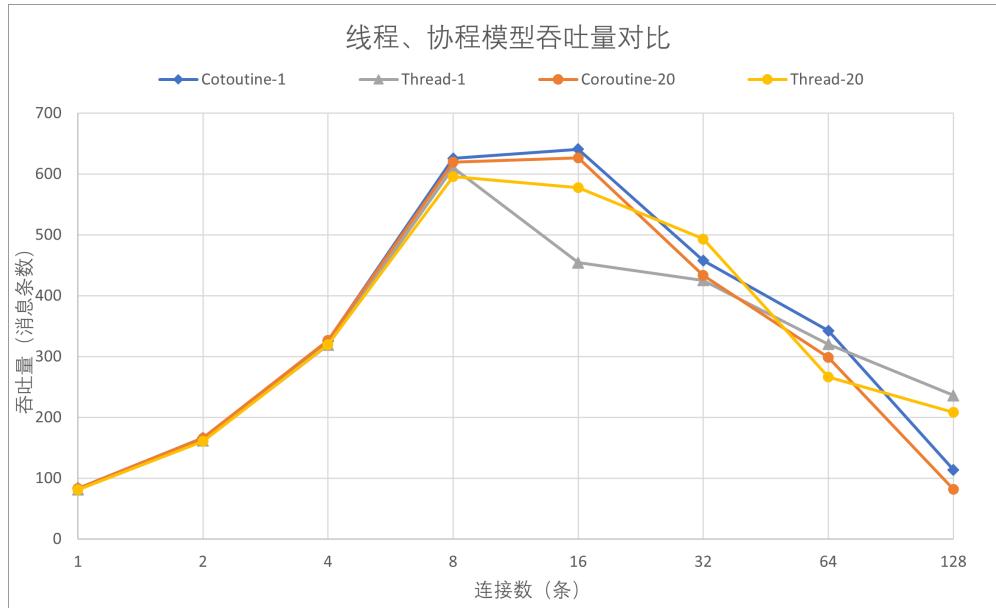


图 A.2 线程、协程模型的吞吐量

1.2 优先级设置对性能的影响，实验结果

在FPGA上的测试更加突出在资源有限的条件下，优先级较高的任务能够得到保障。我们的实验在4个物理CPU上进行，客户端分配1个虚拟CPU，以200ms为周期发送1600字节的请求消息（即矩阵规模为 40×40 ），服务端以协程的形式实现。客户端与服务端在每个优先级下建立1条连接，分别测试在CPU资源不同的情况下不同优先级的连接的性能。实验对应的测试文件为`utest/src/bin/cwpt.rs`，变化的参数为：分配给服务端的虚拟CPU资源数。

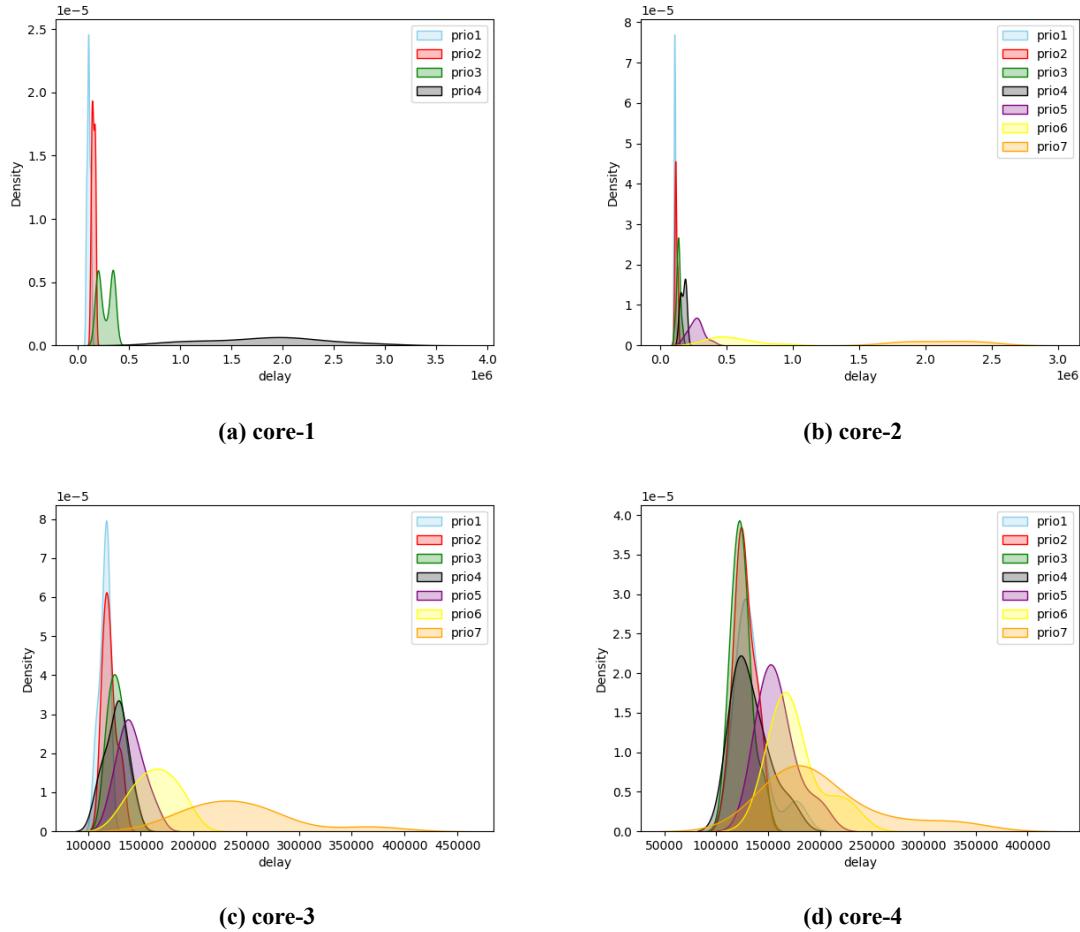


图 A.3 不同优先级连接的时延分布

只分配给服务端 1 个虚拟 CPU 时，只能保证优先级为 1、2、3、4 的任务能够执行。CPU 资源增加对于优先级高的任务的影响较小，但使得优先级低的任务被响应的时间提前。