分 类 号： TP391 密 级： 无

单位代码： 10028 学 号： 2201002036

首都师范大学硕士学位论文

轻量级的操作系统基本调度单位的设计与实现

研 究 生： 王文智

指导教师： 孙卫真

学科专业： 计算机科学与技术

学科方向： 操作系统

2023 年 5 月 26 日

首都师范大学学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　年　月　日

首都师范大学学位论文授权使用声明

本人完全了解首都师范大学有关保留、使用学位论文的规定，学校有权保留学位论文并向国家主管部门或其指定机构送交论文的电子版和纸质版。有权将学位论文用于非赢利目的的少量复制并允许论文进入学校图书馆被查阅。有权将学位论文的内容编入有关数据库进行检索。有权将学位论文的标题和摘要汇编出版。保密的学位论文在解密后适用本规定。

学位论文作者签名：

日期：　年　月　日

首都师范大学学位论文使用授权声明

本人已经认真阅读首都师范大学的“研究生学位论文著作权管理规定”，同意本人所撰写学位论文的使用授权遵照学校的管理规定：

学校作为申请学位的条件之一，学位论文著作权拥有者须授权所在大学拥有学位论文的部分使用权，即：

1) 已获学位的研究生必须按学校规定提交印刷版和电子版学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；

2) 为教学和科研目的，学校可以将公开的学位论文或解密后的学位论文作为资料在图书馆、资料室等场所或在校园网上供校内师生阅读、浏览。

同意论文提交后发布□；滞后：□半年；□一年发布。

作者签名： 导师签名：

年 月 日 年 月 日

本人已经认真阅读“CALIS 高校学位论文全文数据库发布章程”，同意将本人的学位论文提交“CALIS 高校学位论文全文数据库”中全文发布，并可按“章程”中的规定享受相关权益。

同意论文提交后滞后：□半年；□一年；□二年发布。

作者签名： 导师签名：

年 月 日 年 月 日

摘 要

调度是计算机科学的关键问题。进程一般是操作系统分配资源的基本单位，而线程是操作系统的基本调度单位。协程（coroutine）是一种新兴的轻量级任务调度单位，由各个进程独立管理，占有的系统资源少于进程和线程，所以拥有较小的切换开销。这个特点使得协程常作为并发编程的优化工具使用，可以改善系统的整体运行效率。但是，协程的调度受制于所在的进程和线程，无法直接被操作系统感知，这限制了协程作为一种调度单位的表达力和应用范围，无法像线程一样被操作系统统一调度，在系统内的所有线程中选取一个合适的线程执行。

本文提出了一种将协程作为系统级基本调度单位的调度模型。该模型没有为协程本身引入额外的重量级系统资源（栈，上下文等），保留了协程作为轻量级调度单位的低开销特征。该模型为协程引入了优先级属性，使其可以完成优先级调度。该模型还设计了进程的优先级位图和操作系统内核的优先级位图，使得操作系统内核可以通过位图感知进程内协程的存在，并在一个全局的视角下调度运行系统内的所有协程，从而把协程升级为操作系统的基本调度单位。

本文基于 Rust 语言提供了模型的参考实现。首先，本文实现了一个相对独立的协程运行库，用于处理进程内的协程创建，调度和运行，可以被不同的操作系统引入使用。其次，本文修改了开源操作系统 rCore 的内核，结合协程库，完整实现了本文提出的调度模型，使得该操作系统可以将协程作为系统级的基本调度单位使用。由于协程模型只要求增量修改，所以操作系统的原有功能不会受到影响。处于本文调度框架之下的协程，虽然比传统的局限于进程内的协程增加了管理上的开销，但是本文通过实验验证了在高并发场景下，本调度框架的协程性能明显优于线程，依然保持了协程低开销的特征。

关键词：协程；异步编程；操作系统；调度器；Rust

Abstract

Task scheduling is a key issue in concurrent programming. A process is generally the basic unit for resource allocation by the operating system, and a thread is the basic scheduling unit of the system. A coroutine is an emerging lightweight task scheduling unit that is independently managed by each process and occupies less system resources than processes and threads, so it has a small switching overhead. This feature makes coroutines often used as an optimization tool for concurrent programming, which can improve the overall operating efficiency of the system. However, the scheduling of coroutines is limited by the processes and threads in which they reside, and cannot be directly perceived by the operating system. This limits the expressive power and scope of application of coroutines as a scheduling unit, and cannot be uniformly scheduled by the operating system like threads. In the system Select a suitable thread to execute from all the threads in it.

This paper proposes a scheduling model that uses coroutines as the basic scheduling unit at the system level. This model does not introduce additional heavyweight system resources (stack, context, etc.) for the coroutine itself, and retains the low-overhead feature of the coroutine as a lightweight scheduling unit. This model introduces a priority attribute for coroutines, enabling them to complete priority scheduling within the process. The model also designs the priority bitmap of the process and the priority bitmap of the operating system kernel, so that the operating system kernel can perceive the existence of coroutines in the process through the bitmap, and schedule all the coroutines in the operating system from a global perspective. Coroutine, upgrade the coroutine to the basic scheduling unit of the operating system.

This article provides a reference implementation of the model based on the Rust language. First of all, this paper implements a relatively independent coroutine runtime library, which is used to handle the creation, scheduling and operation of coroutines in the process, which can be introduced and used by different operating systems. Secondly, this paper modifies the kernel, node and coroutine library of the open source operating system rCore, and fully implements the scheduling model proposed in this paper, so that the operating system can use coroutines as the basic scheduling unit at the system level. Since the coroutine model only requires incremental modification, the original functions of the operating system will not be affected. Although the coroutines under the scheduling framework of this paper have increased management overhead compared with the

traditional coroutines limited to the process, this paper has verified through experiments that in high concurrency scenarios, the coroutine performance of this scheduling framework is significantly better than Threads still maintain the low-overhead characteristics of coroutines.

Key words: Coroutines, Asynchronous Programming, Operating System, Scheduler, Rust

目 录

图 索 引

表 索 引

第 1 章 绪论

1.1 课题的背景和意义

截止 2022 年，Google 的服务器每天承受着 893[1] 亿次的访问量，平均每秒需要处理 800 万个请求，这对于硬件和软件都是一个极大的挑战。中国互联网络信息中心发布的第三十九次《中国互联网络发展状况统计报告》表明，截至 2016 年 12 月 31 日，我国网民数量已经达到 7.31 亿[2]，其规模已经相当于欧洲人口的总量[3]。截止 2021 年，全球互联网使用数也已达 46.6 亿人次。互联网提供的工作和生活方式重塑着人类社会，全球一半以上的人每天都要通过互联网通信，建立连接，发送请求。计算机技术如何高效快速地构建网络服务是学术界和工业界的一大课题。

计算机处理网络服务，本质上是在有限的硬件之上通过各种软件技术来优化海量任务的处理和执行速度。业界有一个著名的几乎每一家互联网厂商都无法规避 C10k 问题，描述的就是一个网络服务器以同时处理 10000 个大量客户端并发连接的场景。对于软件层面，多线程技[4] 就是一个常见的通过减少调度单元的创建与切换开销，使得有限的硬件可以同时处理大量任务的一个解决方案。但是，基于庞大的用户基数，随着互联网提供的各类业务复杂性和计算量的陡增，业界急需一种更强大的任务执行单位为人类社会的高效运作保驾护航。

协程是一种功能强大且用途广泛的程序结构，一般可以实现为线程的创建与切换开销更小的调度单位。首先，协程可应用于高并发程序，其中多个任务可以并发执行而无需显式同步。与目前广泛应用于并发编程的线程相比，协程可以避免上下文切换带来的负面影响，从而提升系统的性能。而并发编程是目前计算机领域中极其重要的一项技术，世界上的服务器每天都要处理海量的请求并向用户提供尽量快速的答复。在高并发条件下，非线形增长的线程切换开销会成为系统的性能瓶颈[5]，而协程就是一种应对高并发的有效解决方案[6]。

协程常作为效率工具和优化手段出现在一些对于性能要求极为苛刻的场景中。它是一种重要的编程结构，为开发人员提供了强大的工具来构建高性能、并发和响应迅速的应用程序。协程在以高性能为目标的编程语言中必不可少，尤其是在游戏开发和网络计算等领域。因为大量的 I/O 事件往往不能及时达到时，协程就可以作为异步机制的载体进行系统优化[7]。它们使开发人员能够创建可以并发执行的非阻塞函数，从而减少等待 I/O 操作所花费的时间。此外，协程允许开发人员处理多个执行流，从而提供了出色的并发模型。通过使用协程，开发人员可以创建更易于调试和维护的应用程序，并提供更好的用户体验。因此，协程已

成为现代开发人员必不可少的工具，可带来更高效和有效的编程。

虽然目前实验室和工业界有很多各种语言编写的协程工具可以直接使用，但它们都有一个显著的缺点：调度算法不够灵活。协程的主要应用场景之一是高并发编程，而在处理大量任务调度时拥有一个高效灵活的调度算法是必要的。

此外，协程也可用作异步机制的上层抽象，向系统提供高效易用的异步调度单元。

内核在处理 I/O 和网络时常需要等待数据的到来，从而使得 CPU 无法完成实际的计算工作，通常称之为忙等。而异步操作可以允许多个任务同时执行，而不会相互阻塞，这可以消除或减轻忙等带来的性能损耗。近年的研究不断印证着异步机制对于内核的重要意义，但是向内核提供异步机制面临着两个问题，一是设计和实现具有一定的复杂性，二是程序复用率很低，通常只能专门为一个系统设计一套内核异步机制。所以，通过协程向内核提供异步支持，并以函数库的形式提供使用接口可以有效解决这两个问题。

1.2 国内外研究进展

任务调度的性能优化是计算机科学的关键问题，尤其是在互联网飞速发展的今天。目前关于轻量级调度单位的研究进展及其不足可划分为以下两类。

（1）无法参与系统级的调度：在有限的硬件资源限制下，具有更高性能的协程常用来作为多线程技术的优化方案。但是目前主流的协程技术主要应用于进程之内，其创建、调度与运行由各个进程控制，各个进程的协程无法被操作系统统一管理。从性能优化的角度出发，在高并发环境下有必要从一个全局的视角去进行计算资源的分配，但目前主流的协程方案无法做到这一点，它们的特点如下所述。

Golang[8] 是 Google 在 2007 开发一种静态强类型、编译型语言，因其近似 C 语言的执行性能和近解析型语言的开发效率早已风靡全球。它强大的性能很大程度上是因为它在语言层面提供的协程 Goroutine，它是 Go 中最基本的执行单元。每一个 Go 程序至少有一个 Goroutine：main Goroutine。Goroutine 建立在操作系统线程基础之上，它与操作系统线程之间实现了一个多对多 (M:N) 的两级线程模型。每一个 Goroutine 都拥有自己的栈保存执行状态，这会给轻量级的调度单位带来一定的性能损耗，这种协程方案常称为有栈协程[9](stackful)。

C++20[10] 中引入了协程这一全新特性，目的是以更灵活、简便的方式提供生成器和异步操作，并使其更易读和更易维护。C++ 的协程可以使用 co_yield 和 co_await 这两条语句来暂停或恢复协程的执行，它的设计理念源自于对 C++ 函数的泛化，即在原先 C++ 普通函数的基础上增加了三个附加操作：暂停，恢复和销

毁。线程和协程都可以作为 C++ 协程的调用者，当它们调用一个协程的时候会先创建一个协程帧，协程帧会构建 promise 对象，再通过 promise 对象产生返回值。调用者可以通过协程帧的句柄 std::coroutine_handle 来访问协程帧，这在协程恢复运行的时候用到。

Rust[11] 中协程的原理是基于 async/await 的概念，它允许在单线程程序中进行多个异步函数的切换和运行。Rust 的协程通过 Futures 和 Tasks 这两个机制实现，它们是 Rust 中异步编程的基本模块。Future 代表一个计算单元，它最终会产生值或返回一个错误提示。Futures 是惰性的，意思是着它们只在被轮询时才开始执行，而轮询 Future 会返回一个 Poll 类型的结果，指示 Future 是否已完成、仍在运行或遇到错误。

研究人员利用 C++20 和 Rust 提供的无栈协程[9](stackless) 对系统性能做了很多优化工作，但它们都不是系统级的调度单位，无法在全局的视角下分配计算资源。Corobase[12] 采用 coroutine-to-transaction 模型来规避数据获取期间的数据停顿（data stall）。Zhu[13] 等人通过引入 coroutine-to-SQL 模型来减少交互式事务的等待开销。FaSST[14] 和 Grappa[15] 使用 C++ 协程库来减少 RDMA 网络延迟。Nelson[16] 等人为协程开发了一个运行时系统，它支持足够高的并发量以减少其大型高带宽全局内存系统的长时间延迟。Knoche[17] 提出了一种基于协程的方法来减轻迁移到微服务时批处理作业的潜在性能损失。

（2）缺乏内核之中通用的轻量级异步任务调度方案：内核除了捕获应用程序的执行错误之外，还需要处理高特权级的指令，如设备读写，所以内核同样需要轻量级的任务调度机制进行性能优化。此外，内核中等待的读写事件不一定能马上到来，所以内核中的调度单位还需要具有异步运行的特征。近年来有较多关于利用轻量级的异步任务调度单位优化内核性能的工作，但是它们都是分别独立构造了一套功能相近调度框架作为研究工具，业界缺乏通用的轻量级异步任务调度方案提供软件复用，导致了研究和开发成本的增加。它们的工作特点如下所述。

LXDs[18] 在内核中开发了一个异步运行时来运行轻量级协作线程，可以高效地实现跨域调用的批处理和流水线化。memif[19] 是一种用于内存移动的操作系统服务，它基于异步和硬件加速实现了低延迟和低开销的接口。Lee[20] 等人通过为内核提出的异步 I/O 堆栈（AIOS）降低了 I/O 延迟，从而显着提高了应用程序的性能。还有许多其他工作也专注于异步 I/O[21-24]。Sundaresan[25] 等人使用异步机制减少内核中的同步瓶颈以获得更高的并行度。

1.3 本文的研究内容

本文致力于研究一种比主流的线程调度开销更低的系统级调度单位及相应的调度框架，以提升系统的整体运行性能。本文的研究动机与相应的研究内容来自以下三个子课题。

问题一：高并发技术是互联网行业提供平稳服务的重要基础，而调度单位所携带的内存资源在高并发环境下会为系统引入相应的调度与切换开销。针对这类问题，本文调研了操作系统中常见的调度单位并从它们的应用特点中发现，协程由于具有比线程更低的资源占用与切换开销，被广泛应用于高并发环境。但由于协程是一种用户态管理的调度单位，无法被内核感知与控制，所以不能作为操作系统的基本调度单位。

从中得到本文的子课题一：使内核可以感知用户态的协程的存在，从而可以在一个全局的视角下干预它们的调度，将协程升级为系统级的调度单位，并且保持协程作为轻量级调度单位的特点，不为协程引入重量级的系统资源。

问题二：根据 1.2 节中的相关工作，异步机制对于内核的性能有着强大的增益作用。但在使用异步机制优化内核时，研究人员通常需要在内核从头开始构造一套专属的异步运行时，缺乏软件复用的设计增加了工程上的开发成本，没有充分利用协程的异步运行特征。

从中得到本文的子课题二：扩展协程的应用范围，使得内核可以创建运行协程。用户态的协程用于执行用户提交的任务，而内核的协程作为异步机制的载体，向内核提供通用的异步运行接口，提高软件复用程度，优化内核应对读写事件时的等待与切换开销。

子课题三研究如何统筹子课题一和二：如果将调度模型实现为内核的子模块，可以为内核引入协程，并在其中管理系统中的所有协程的调度，但这会削弱调度模型的可移植性，并且可能导致协程牺牲轻量级的特点。所以本文提出“函数库 + 少量内核修改”的研究思路，由函数库提供主要功能，在满足子课题一和二的同时尽量减少对于内核开发的依赖。

本文最终基于 Rust 语言在开源操作系统 rCore OS 上完整实现了此调度模型，并通过实验验证了升级为系统级基本调度单位的协程依然保持了低开销与高性能的特征，当达到所测试的最大并发量 4000 时，协程的总执行时间约为线程的 1/3 至 1/2。

1.4 论文组织结构

本文将在第二章介绍与本文相关的技术背景，后续的内容将会涉及这些概念和技术，包括用户态与内核态，函数库与系统调用，进程、线程和协程的区别与联系，从中可以了解到协程作为进程内的局部调度单位的常规实现的局限与原因。在第二章结尾简述了一种协程的实现原理，该示例不依赖于任何高级语言的特殊语法实现，几乎可以应用于所有的高级语言中。

第三章描述本课题提出的调度模型。包括优先级队列的组织形式，协程控制块的存储与获取策略，调度器的工作流程，展现了一个能独立运行的完备的协程运行时。第三章中把调度模型分为局部调度与整体调度两个部分描述，局部调度是指单个进程之内的调度行为，整体调度则是指操作系统内核感知到协程的存在之后进行的系统级调度，两个部分通过优先级位图进行桥接。协程管理器负责协调协程进出调度队列时引起的各个部件的变化。

第四章介绍实现模型的技术细节。本文基于 Rust 语言提供了一个调度模型的参考实现。会以 Rust 语言为例介绍模型实现时的重点，包括数据结构在内存中移动以及线程间传递时必须注意的问题，以及对应的解决方案。第四章中还会具体介绍改造操作系统内核的细节，包括优先级位图的创建以及后续的查找过程。

第五章通过三个实验验证所实现的协程模型的性能。第一个实验对比了调度单位竞争进入临界区的总耗时随着并发量增加的变化趋势；第二个实验测试了调度单位读写内核缓冲区的性能；第三个实验设置对照组测试了高并发环境下优先级调度优化系统负载的能力。

第六章总结本文工作，以及本领域尚待突破的课题。

第 2 章 相关技术与概念

本章介绍本课题所涉及到的技术背景与相关概念。本文的主要工作是设计并实现一种轻量级的系统级调度单位，内容涉及到提出一种全新的调度模型，把原本工作在用户态的协程升级为内核可见、可管理的对象，所以本章会在第一节中介绍用户态与内核态的区别。随后在第二节中介绍目前操作系统中主要的调度单位，以阐明协程在操作系统中的工作与实现方式，以及与其他调度单位之间的区别与联系。

2.1 用户态与内核态

用户态和内核态是指计算机中央处理器（CPU）的两种不同的执行模式，操作系统会控制 CPU 在任何时间都处于一个特定的模式，不同的模式下具有不同的指令执行范围。

指令集（Instruction Set）是一组硬件架构可以直接理解和执行的指令（若干个无二义性比特位），也就是计算机硬件所支持的命令集合。汇编语言可以看作为是指令集的一种表达形式，它是一种人类可读的语言，可以通过汇编器将其转换为机器码。

同时 C P U 指令集有权限分级。CPU 指令集可以直接操作硬件的，要是因为指令操作的不规范或滥用，造成的错误会影响整个计算机系统，甚至在具体业务中造成不可挽回的后果。而对于硬件的操作又是非常复杂的，参数众多，出问题的几率相当大，必须谨慎的进行操作，对开发人员来说是个艰巨的任务，还会增加负担，同时开发人员在这方面也不被信任，所以操作系统内核直接屏蔽了开发人员直接对硬件操作的可能性。如图2.1所示，CPU 在执行指令时会设置一个确定的状态限制当前能执行的指令集，内核一般拥有最高的权限，可以直接发出硬件操作指令。

用户态是大多数应用程序和进程运行的模式，人们在日常生活中所使用的各类软件就是工作用户态。它是一个受限的执行环境，在这种保护模式下，即时程序发生崩溃也是可以恢复的。在该环境中，应用程序无法直接访问硬件设备或内存等系统资源。相反，他们必须通过操作系统的系统调用请求访问这些资源。在用户态下，进程只能访问自己的内存和资源。如果进程试图访问其分配空间之外的资源，则会生成异常并终止进程。在用户模式下，具有用户（U）保护级别，代码没有对硬件的直接控制权限，也不能直接访问地址的内存，程序通过调用系统接口 (System Call APIs) 到达内核之后再访问硬件和内存。

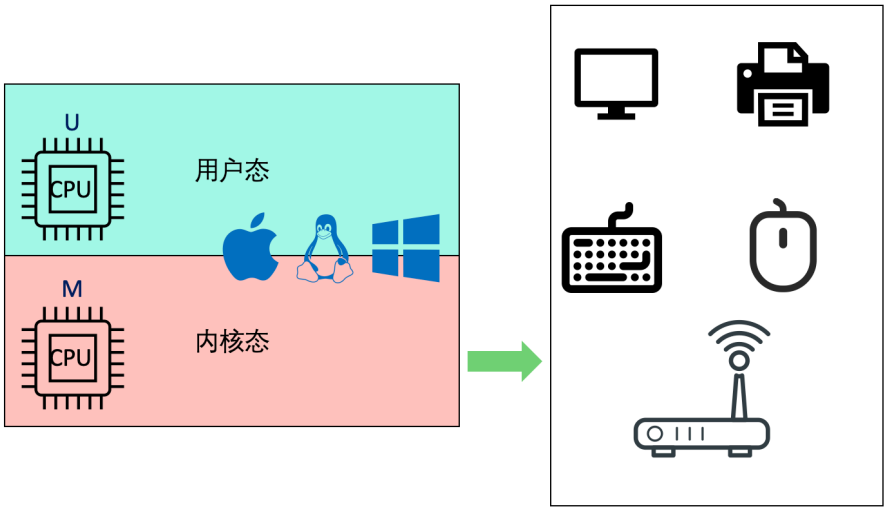

图 2.1 用户态与内核态

Fig. 2.1 User-mode and Kernel-mode

内核态是一种特权执行模式，在这种模式下，操作系统可以直接访问系统上的所有硬件和资源。在内核模式下，操作系统可以不受限制地执行任何操作或访问任何资源。这是操作系统提供服务和管理系统资源所必需的。执行内核空间的代码，具有机器（M）保护级别，有对硬件的所有操作权限，可以执行所有的指令集，访问任意地址的内存，在内核模式下的任何异常都是灾难性的，将会导致整台机器停机。

用户态和内核态之间的主要区别在于进程拥有的特权级别。在用户模式下，应用程序和进程在它们可以访问和执行的操作方面受到限制。它们必须依靠操作系统来提供对资源和服务的访问。在内核模式下，操作系统对系统拥有完全的控制权，可以执行任何操作或访问任何资源。当从用户态进行系统调用时，CPU 会暂时切换到内核态以允许操作系统处理请求。请求完成后，CPU 返回用户态并恢复进行系统调用的进程。

由于进程和线程直接被内核所管理，所以它们的创建、调度与运行需要进入内核。操作系统提供了系统调用[26-29]的方式使得应用程序可以通过状态切换进入到内核使用相应的服务，并得到一个计算结果（返回值）。但是协程直接由进程所管理，所以它的创建与调度不涉及特权级的切换，没有上下文的保存与恢复，其切换开销接近于函数切换，这是它高性能特征的来源。

用户态和内核态的使用对于计算机系统的安全性和稳定性至关重要。通过限制用户模式进程的权限，系统可以防止对资源的未授权访问，并防止错误进程导致的崩溃或其他故障。通过使用内核模式来管理资源和服务，操作系统可以为应用程序运行提供一致和可靠的环境。在必须满足系统安全的强约束下，如果通过系统调用的方式使得协程可以被内核管理，会对协程性能造成极大影响，使其失

去“轻量级调度单位”的优势。本文在第三章中提出的调度模型通过操作系统允许的共享信息的方式，规避了系统调用带来的特权级切换的负面影响，并在第四章实现中详细阐述了相应的内核修改方案。

2.2 操作系统中的调度单位

进程、线程和协程之间的相似之处在于，它们都提供了一种同时执行多个任务的方法，允许程序内的并发或并行，从而可以更有效地利用系统资源并提高性能。它们还都提供了一种方法来将不同的任务或执行上下文相互分离，从而提高程序灵活性。

进程、线程和协程之间的主要区别在于它们之间的隔离级别和资源共享。进程之间是完全隔离的，拥有独立的内存空间、文件描述符和执行状态。线程共享相同的内存空间和文件描述符，但有自己的执行状态，记录在栈中并由内核管理。协程共享内存空间和执行状态，并允许对并发进行细粒度控制。正因如此，进程和线程的调度拥有相应的“重量级资源占用”从而可以被内核直接控制，而协程却不被内核所感知，无法被操作系统统一分配计算资源。

在性能方面，由于上下文切换和进程间通信的开销，进程通常是最慢的，而线程和协程由于共享内存空间和减少的上下文切换开销而更快。然而，由于协程的协作性质和缺乏系统级上下文切换，协程的开销是所有三者中最低的。这些特征与区别来自于下述各自的设计差异。

2.2.1 进程

进程是在独立的内存空间中执行的程序实例。每个进程都有自己的地址空间、文件描述符和执行状态等。进程之间通过操作系统提供的通信机制（例如管道、套接字或共享内存）相互通信。它通常用于运行单独的、独立的任务或隔离的不受信任代码的执行。

根据冯诺依曼体系结构，程序的执行需要从内存上读取指令和数据，所以进程的运行必然有之依赖的内存。虚拟内存技术随着计算机科学的发展被提出，页表的引入使得每个进程都可以拥有自己独立的内存地址空间，并可以在此基础上编程。如图2.3所示是 Linux 进程的模型。BSS 段（block started by symbol segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域。所以它的数据属于静态内存分配的内容。数据段（data segment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。代码段（code segment/text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读，某些架构也允许代码段为可写，即允许修改程序。

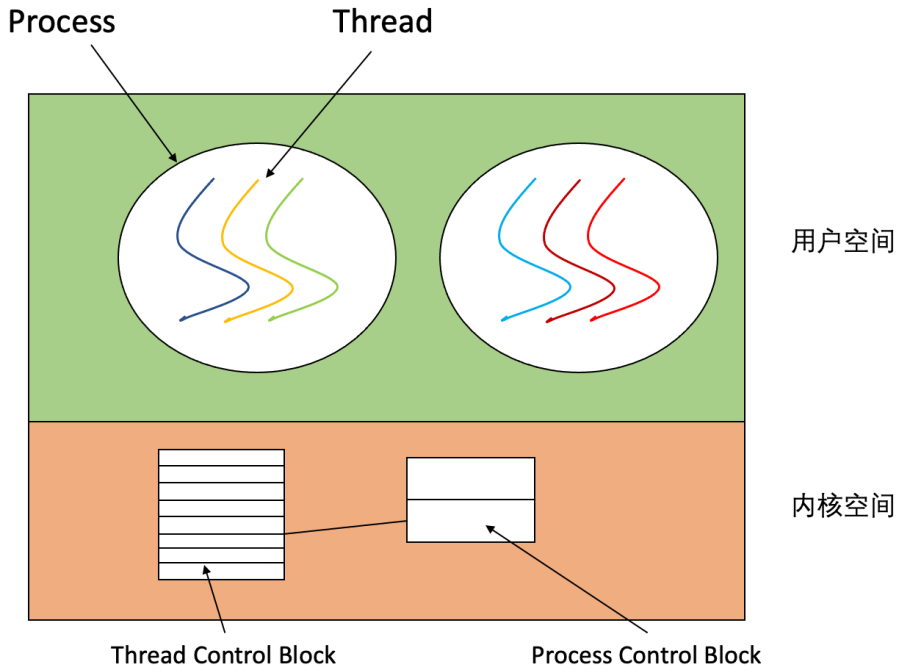

图 2.2 进程和线程

Fig. 2.2 Process and Thread

在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。堆（heap）是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用 malloc 等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用 free 等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）。栈（stack）又称堆栈，是用户存放程序临时创建的局部变量，也就是说函数体中括弧中定义的局部变量（不包括 static 声明的变量，static 意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的线程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进后出特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上讲，可以把堆栈看成一个寄存、交换临时数据的内存区[30]。

2.2.2 线程

线程是进程内的一个执行单元，它与同一进程内的其他线程共享同一个的地址空间和文件描述符。线程可以被认为是轻量级进程，因为它们与进程共享许多相同的属性，但没有自己独立的内存空间。线程通常用于在单个进程中同时执行多个任务，例如处理多个网络连接或执行并行计算。

线程是对函数（一般称为线程主函数）的封装，线程的执行是多个嵌套函数在栈上进出并执行相应的指令。因此，线程切换需要保存栈和栈的状态（CPU 的寄存器值，通常称为上下文）。每个线程都有属于自己的唯一的栈，分配与回收都

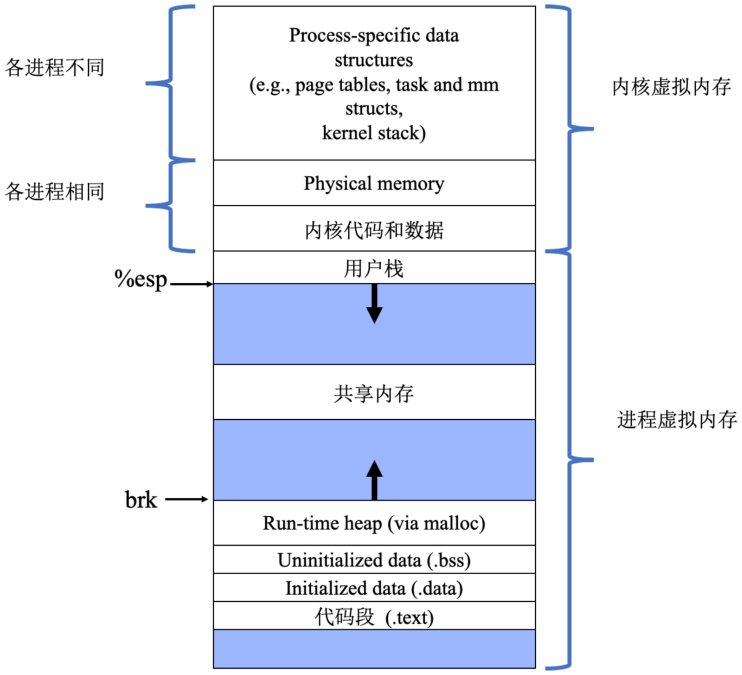

图 2.3 进程的内存模型

Fig. 2.3 Process Memory Model

由操作系统内核管理。线程是执行代码的调度实体，通常来说，它的创建必须要传入一个函数，用以完成相应的功能，这个函数被称之为线程主函数，会被插入到栈底。如图2.4所示，每个函数都使用一个叫做栈帧的结构描述。多个函数的嵌套执行过程，表现在栈上就是多个栈帧不断进栈（函数调用）、出栈（函数返回）的过程。

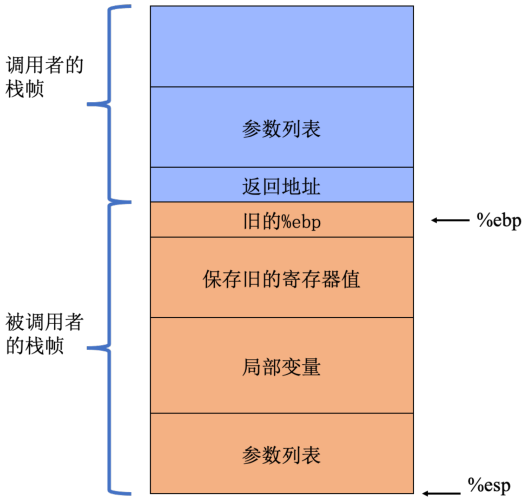

图 2.4 栈帧模型

Fig. 2.4 Stack Frame Model

如图2.4所示，被调用者栈帧从栈顶到栈底如下构成：

(1) 创建的参数表为要调用的函数建立的参数；

(2) 局部变量: 即在函数内部声明的变量 (如果有)；

(3) 保存的寄存器的上下文 (如果有)，当被调用函数返回之前，为调用者函数恢复原来的寄存器中的数据，如果当前架构有足够的空闲物理寄存器可供使用，一些寄存器信息可能不需要压入栈；

(4) 旧的帧指针，即前一栈帧的 ebp 指针，它指向前一帧的栈底。

调用者函数的栈帧：

(1) 参数列表；

(2) 本地变量（如果调用者函数内部有声明局部变量）；

(3) 返回地址：调用者函数内部的执行到 call 指令时会将 call 指令所在行的下一条指令的内存地址压入栈，当被调用者函数内部执行到 ret 指令后，从栈中弹出返回地址，以便调用者函数回到该地址对应的指令继续执行余下的指令。

2.2.3 协程

协程，有时也称为协作式多任务处理或用户级线程，是 continuation[31-34]的继承和发展，是异步概念的起源之一，可以使得单个线程具有并发能力。协程和 continuation 都是可以被暂停并在稍后恢复执行的执行单元抽象，不同特性的 continuation 可以实现不同的协程[35]，反之亦然[36-37]。最早对协程的描述是由 Melvin Conway[38] 于 1958 年提出的，是经过多年发展的计算机科学中流行的抽象概念。一般来说，协程是子例程的泛化[9,39]，具有暂停自身执行的能力，稍后由程序的另一部分恢复，此时协程会继续从之前主动让出时被暂停的地方开始执行，直到下一次暂停或协程的终止。

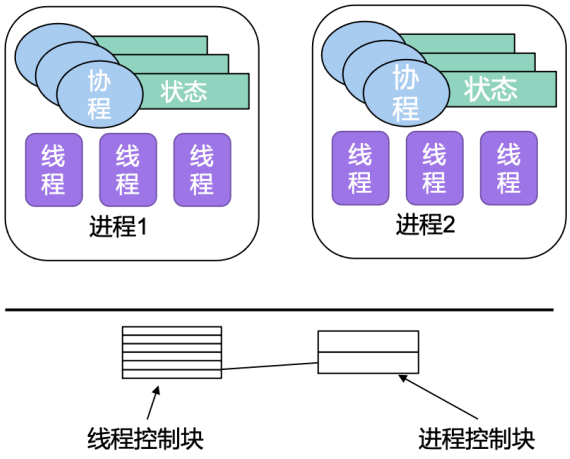

图 2.5 进程，线程与协程的区别与联系

Fig. 2.5 Process, Thread and Coroutine

图中2.5描述的是目前操作系统中常见的三种调度单位之间的关系。进程和线程携带重量级的系统级资源作为操作系统的基本调度单位被保存在内核之中，创建、调度与释放由内核管理。协程作为用户态的程序优化手段用于提升进程之内的任务调度性能，这是因为协程具有异步执行的特征。

在早期的编程语言中就有对异步机制的支持[40-42]，但后来随着多核处理器的出现，它的应用领域被多线程技术广泛替代。直到近年来现实世界各种不断膨胀的业务对网络通信提出了越来越严峻的要求，服务器上的线程常需要处理大量的等待事件，迫使人们将目光再次投向异步机制[43-44]。

协程可以作为异步函数的上一层封装[6,45]。通常，它需要执行的指令数量相对较少，并作为函数调用在线程栈上运行。堆栈必须与线程一一对应，因此可以认为协程运行在线程中。异步函数允许程序在等待外部事件时直接返回并执行程序的其他部分，而不会阻塞整个堆栈的执行流程。运行在同一个线程中的多个协程在不同的时间重用同一个栈来执行各自的异步函数。因此，它们之间的切换没有堆栈切换开销。当异步函数在主动让出之前被中断或抢占时，由于协程是以函数调用的形式在线程中执行的，此时可以使用线程的中断机制，这样不仅保存了线程的中断位置和上下文，它还保存了协程的中断位置和上下文。当线程恢复时协程可以继续执行。可以看出，它以异步方式执行活动。这与线程类似，因为线程允许在单个进程即单个地址空间内进行并发。但是，与线程不同的是，协程不是由操作系统调度程序调度的，而是由程序本身调度的。换句话说，协程是协作的，它们在指定点主动将执行权让给其他协程，从而允许程序并发执行多个任务，而无需线程上下文切换的开销。协程通常用于需要对并发进行细粒度控制的应用程序，例如对性能要求严格且负载极高的网络服务器或视频游戏。

协程根据其占有资源方式的不同一般可分为两类，有栈协程和无栈协程。

有栈协程[46] 类似于将调用信息保存在堆栈上的线程。当一个协程被创建时，栈内存被分配来保存协程的上下文，如果一个子协程被创建，它会继续被压入栈中。有栈协程的特点是使用者不用担心调度问题。缺点是难以分配合理的堆栈大小。使用有栈协程的最先进语言是 Lua[47] 和 Golang[8]。

无栈协程像函数一样使用，通常存储在堆内存中。它不会分配自己的堆栈，而是使用线程的堆栈。无栈协程的特点是协程切换开销小。缺点是协程调度不平衡，需要用户干预调度。C++[10]、Rust[11]、Kotlin[48] 和 Python[49] 等流行语言使用无栈协程。本文的协程模型也是无栈协程。

协程是异步函数的上层抽象，区别于线程内部封装的函数及其运行上下文，协程的内部封装的是异步函数及其运行状态。有栈协程的其实现原理其实就是在用户态应用程序的地址空间内独立构造了一个类似于线程的结构，包含一个将要

运行的主函数和记录寄存机值的上下文数据结构。无栈协程的实现则更为巧妙和复杂，包括多个模块，以便实现一个函数状态机，它的运行将表现出异步的行为。

异步函数的调用可以通过堆解决，同样的技术应用到协程，将每个协程的上下文（如程序计数器）保存在其它地方而不是堆栈上，协程之间相互调用时，被调用的协程只需要从堆栈以外的地方恢复上次主动让出点的上下文，就可以做到类似于线程的上下文切换。但它们都有一个明显的弊端：只能作为一种程序设计手段来使用，无法被内核统一管理，所以也无法成为一种系统级的调度单位在全局视角下进行计算资源的分配。

2.3 本章小结

本章介绍了本课题设计并实现的轻量级调度单位中涉及到的相关概念与技术，阐述了进程，线程和协程这些操作系统的调度单位体现在用户态与内核态之间的区别与联系，明确了进程和线程的资源占用问题的来源，以及协程的低切换开销的原理。下一章将提出一种全新的调度模型，在避免协程参与过多特权级切换的情况下，以系统允许的相对安全的信息共享方式传递协程的调度信息，从而在内核中完成全局调度。

第 3 章 调度模型的设计

根据 1.2 节中的相关工作，协程拥有较低的内存资源占用和切换开销，但作为任务调度单位来说，它们的信息无法被内核所获取，其调度行为亦无法被内核所管理，所以无法作为系统级的调度单位使用，这是现有主流的协程设计的不足。通过调研以往研究工作的特点与不足，本文在继承无栈协程的优势的同时，解决了 1.3 节本文研究内容中提出的子课题一：在不为协程引入与线程同级别的资源占用的的情况下，保持协程的低开销特征并将协程升级为轻量级的操作系统基本调度单位，使得操作系统可以在一个全局的视角下对其分配计算资源。

本章首先在第一节中介绍模型的总体结构设计，总览各个模块的主要职责和设计优势，包括协程控制块，优先级队列，位图，调度器等。在第二节中详细介绍调度器内部和优先级位图的设计思想，以及协程在进程内的局部调度与系统内的全局调度过程。在第四节中介绍本课题设计的协程异步唤醒机制，以及优先级协程在其生命周期中的的状态转换过程。

3.1 总体结构设计

如图3.1所示，Rust 语言提供异步函数，协程库提供优先级和协程 ID 字段，并将它们封装到称为协程控制块的数据结构中。每个协程控制块代表一个协程。而协程可以看作是一个调度对象，与一个全局的异步函数一一对应。模型使用协程 ID 作为索引，将所有协程存储在进程堆内存上的哈希表中。可以通过协程 ID 获取或删除其对应的协程控制块。考虑到协程在其生命周期中会因为等待外部事件而多次进入和退出调度器，相比调度器直接管理协程控制块，它可以减少调度器工作时的内存开销。

协程调度器提供了两个必要的操作：push 和 pop，用于插入和删除协程，这两个接口需要实现协程的调度算法。对于优先级调度，模型会设置多个队列，每个队列具有不同的优先级，并将对应优先级的协程 ID 存储在其中。从高优先级到低优先级查询这些队列，即可完成优先级调度。从调度器获取协程 ID 后，如果需要协程的完整信息，可以通过协程 ID 和哈希表获取。

模型使用优先级位图来描述协程在每个进程中的优先级布局。本文还在内核中使用一个位图来描述所有进程的协程的优先级布局。内核可以通过这些位图感知协程的存在和它们的优先级，并制定进程的调度策略，从而间接干预用户态下的协程调度。这使得协程调度不再是每个进程内部的独立行为，而是可以从全局的角度统一考虑它们。其设计将在 3.2 节中介绍。

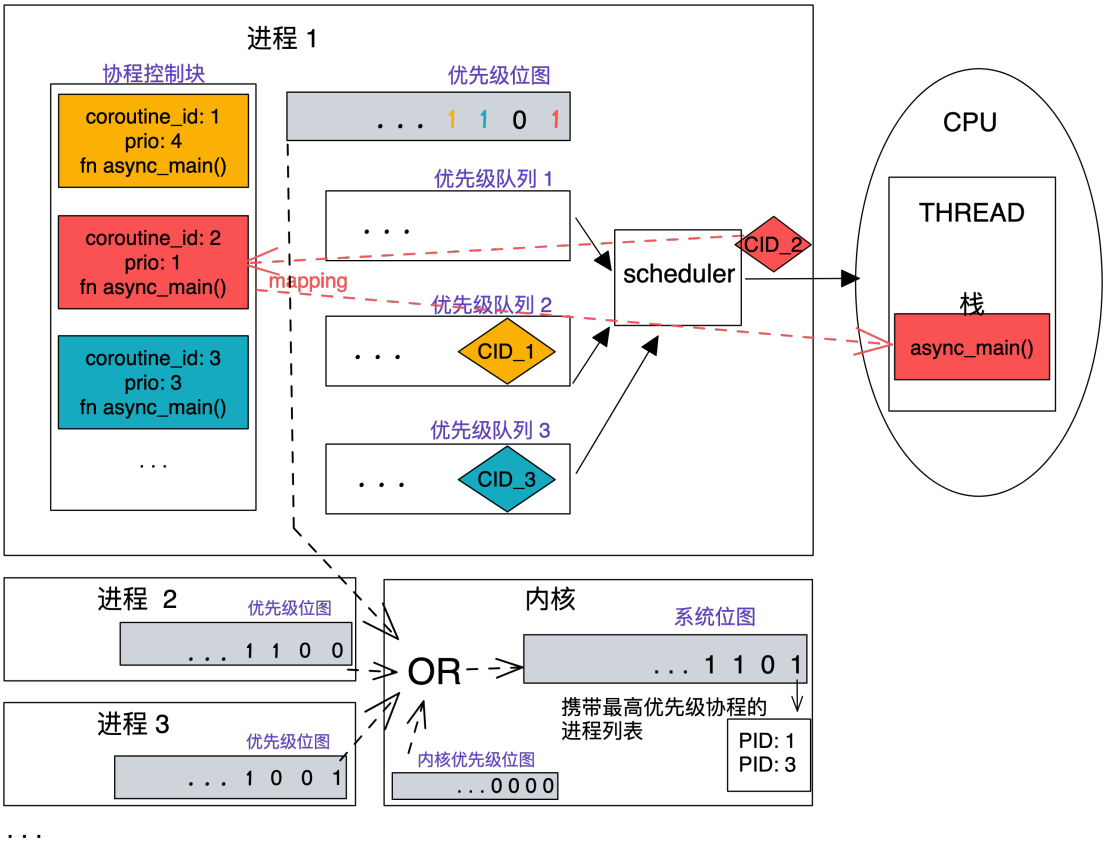

图 3.1 模型总体结构

Fig. 3.1 Architecture of The Model

函数的执行必须依赖栈，异步函数同样如此。为此，本文有如下设计：当一个协程需要执行时，启动协程执行器，它本质上是一个函数。协程执行器会取出所有处于就绪状态的协程，循环运行。协程将使用该线程的堆栈来执行其异步功能。当协程完成或因等待外部事件而挂起时，会通过返回函数退出被占用的线程栈。此时运行协程执行器的线程是空闲的。准确地说，它没有执行任何协程，这意味着下一个协程可以使用它的堆栈。如果没有就绪协程，则协程执行线程跳出循环结束。

在此引入动态绑定的概念来描述协程和栈的关系，即：协程（异步函数）的执行需要一个栈，但是协程在主动让出时可能会在不同的栈上执行和恢复，显然同一个栈也可以执行不同的协程。这个协程模型表现出这种现象，因为每个进程只有一个唯一的协程调度程序。

协程在主动让出后需要被唤醒，因为它等待外部事件。唤醒是协程的一个关键机制。这部分在第 3.3 节中有详细描述。

3.2 局部调度与全局调度

根据优先级处理协程的调度顺序。在一个进程中，协程根据优先级从高到低执行；进程之间，协程优先级最高的进程会被最先调度。调度器只对协程 ID 进行操作，以简化内存操作。当需要协程的完整信息时，可以通过映射协程 ID 来获取。

3.2.1 进程内的协程调度

每个协程都有一个优先级。相同优先级的协程 ID 由同一个队列管理。多个不同的优先级会对应多个不同的队列，它们组成一个队列组。每个进程只有这样一个组，存放在进程的堆内存中。优先级队列中的协程都处于就绪状态，即可以立即调度执行，只不过用优先级区分了其调度顺序。显然，新创建的协程将直接插入到队列中。对于未就绪的协程，在被唤醒后会重新插入到优先级队列中。

在任意线程的任意位置，都可以调用协程运行接口来执行协程。它会在一个循环中不断访问调度程序以获取协程并执行它们。此时，调度器会按照优先级从高到低的顺序访问队列。当第一个非空队列出现时，从队列中弹出一个协程 ID，这个协程是进程中优先级最高的可以立即执行的协程之一。

如果所有队列都为空，则进程中不能执行任何协程，调度器将返回一个空值。此时会进一步判断协程数是否为 0，即是否有处于等待状态的协程。如果没有协程，则说明所有协程都执行完了，可以退出协程执行器。如果有则根据传入的参数选择退出协程运行接口，或者调用系统调用让当前线程切换直到重新调度并且至少有一个就绪协程存在才继续执行。

3.2.2 优先级位图

模型使用位图来描述协程的优先级布局。

如图3.2所示，每个进程都会维护一个位图，长度为协程的优先级数，每个位的值为 0 或 1，表示是否有该优先级的协程。当访问调度器插入或请求协程时，优先级对应的队列会 push 或 pop 一个协程 ID。如果由空变为非空或非空变为空，则进程位图中对应的位写入 1 或 0。

由于内核也有自己的地址空间，它可以创建和运行协程。因此，内核也有一个优先级位图来表示它所管理的协程的优先级信息，称为内核位图。

另外，模型在内核中设置了一个额外的位图来表示整个系统协程的优先级信息，称为系统位图。这个位图的长度也是协程的优先级数。但是，此位图中的每一位的 0 或 1 代表整个系统的所有进程中是否存在该优先级的协程。要计算这个位图的值，需要使内核能够访问所有进程的位图。本文的做法是指定一个虚拟地

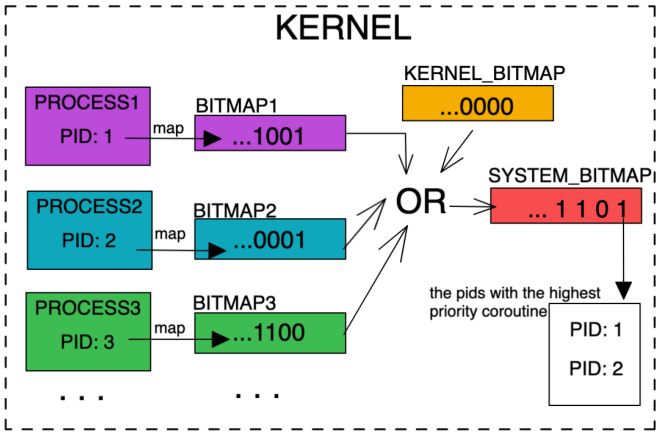

图 3.2 优先级位图逻辑结构

Fig. 3.2 Priority Bitmap Structure

址作为用户态访问位图的进程地址。然后在进程初始化时，向内核申请一个空闲的物理页，将指定的虚拟地址映射到物理页的起始地址，并授予进程对该地址的读写权限。这样进程就可以在用户态读写位图，内核也可以直接访问进程的位图。

将每个进程 ID 与其进程位图的物理地址进行映射，通过枚举进程 ID 就可以得到内核中所有的进程位图。

系统位图的值基于所有进程位图。当时钟中断到来时，内核负责遍历所有进程的位图并更新系统位图。但很明显，这个计算过程是一个比较简单的过程，即只需要对所有进程位图进行 OR 运算——只要至少一个进程的至少一个协程属于这个优先级，对应的位在系统位图中为 1，只有当所有进程都没有该优先级的协程时，系统位图中对应的位才会为 0。通过将所有进程位图的或运算结果赋值给系统完成更新位图。

3.2.3 全局的协程调度

基于之前的设计，可以在进程内部实现协程的优先级调度。现在，将其扩展到整个系统，对所有进程的协程进行优先级调度。这需要获取内核中所有进程的协程的优先级信息，并记录优先级最高的协程的进程。那么当从内核态切换到用户态时，对进程进行调度，可以调度优先级最高的协程在进程中运行。

当前系统具有进程位图和系统位图，可以在每次时钟中断后调度具有最高优先级协程的进程。在时钟中断中，进程的位图将随着协程的创建和退出而改变。尽管如此，在下一次时钟中断之前，每个进程的位图变化都不会同步到系统位图。也就是说当所有最高优先级的协程都返回（exit or yield）时，在下一个时钟中断到来之前，整个系统还没有满足协程的优先级调度。如果希望整个系统的所有协程在运行时都严格遵守优先级调度，本文进一步提供如下设计。

如前文所述，协程将通过函数返回完成执行或主动让出，使得其此前占用的

线程堆栈空闲。不管是哪种情况，之后都会进入下一个循环，再次访问调度器获取协程并执行。进入下一个循环之前，可以读取进程位图和系统位图，比较两者记录的最高优先级，相当于查询这个进程中是否还有未执行的最高优先级协程。如果进程中协程的最高优先级低于系统位图中记录的所有进程中协程的最高优先级，则说明该进程中优先级最高的协程已经全部执行完毕。这时应该调用系统调用使得线程让出当前处理器，进入内核更新系统位图，并根据结果进行调度；否则继续执行。

显然，如果追求严格的优先级调度，在一次时钟中断时会因为多次更新系统位图而引入大量的用户态和内核态的切换开销。所以这种方式应该作为一种选择来提供。由于这是协程执行器的行为（它决定是否检查系统位图），可以通过一个参数来控制它是否开启。默认情况下，执行宽松优先级调度，即只在每次时钟中断后才调度优先级最高的协程进程，系统位图在时钟中断内不更新。

3.3 协程管理器与异步机制

执行器是本模型中的协程管理组织，负责管理协程数据结构、优先级队列以及本节描述的协程唤醒机制。

下面解析协程让出线程栈的两种情况：

1. 协程执行完毕后，执行并返回协程的主函数。此时会根据协程 ID 找到协程的数据结构，并释放其占用的堆内存；

2. 协程还没有执行完，因为在等待外部事件而无法继续执行，所以选择返回函数放弃。这个时候提倡不把协程 ID 插回优先级队列，而是直接进入下一轮循环，访问调度器取出下一个协程执行。当挂起协程的等待事件到来时，执行唤醒操作，即将唤醒的协程 ID 插回优先级队列。这就是这个模型中协程的异步机制。可以方便的完成协程的唤醒操作，保证调度器的工作效率。这意味着优先级队列中的每个协程都处于可以立即执行的就绪状态。访问优先级队列时，不需要检查协程的状态，也不需要访问处于等待状态的协程。

至此，唤醒协程的操作就是将协程的 ID 插入到对应的优先级队列中，这就需要协程的 ID，协程的优先级，优先级队列的地址。可以将这三个变量组合成一个名为 askWaker 的三元组，并在协程 ID 和对应的 TaskWaker 之间建立映射关系。当一个协程需要被唤醒时，只需要它的协程 ID。对于 TaskWaker 的创建时机，可以查看 TaskWaker 是否在协程被选中执行之前创建，还没有开始执行。如果没有，则创建并交给 Executor 管理。显然，当一个协程退出时，它对应的 Taskwaker 会被立即删除。

所以现在的关键问题是如何为执行等待事件和唤醒操作的协程获取等待唤醒

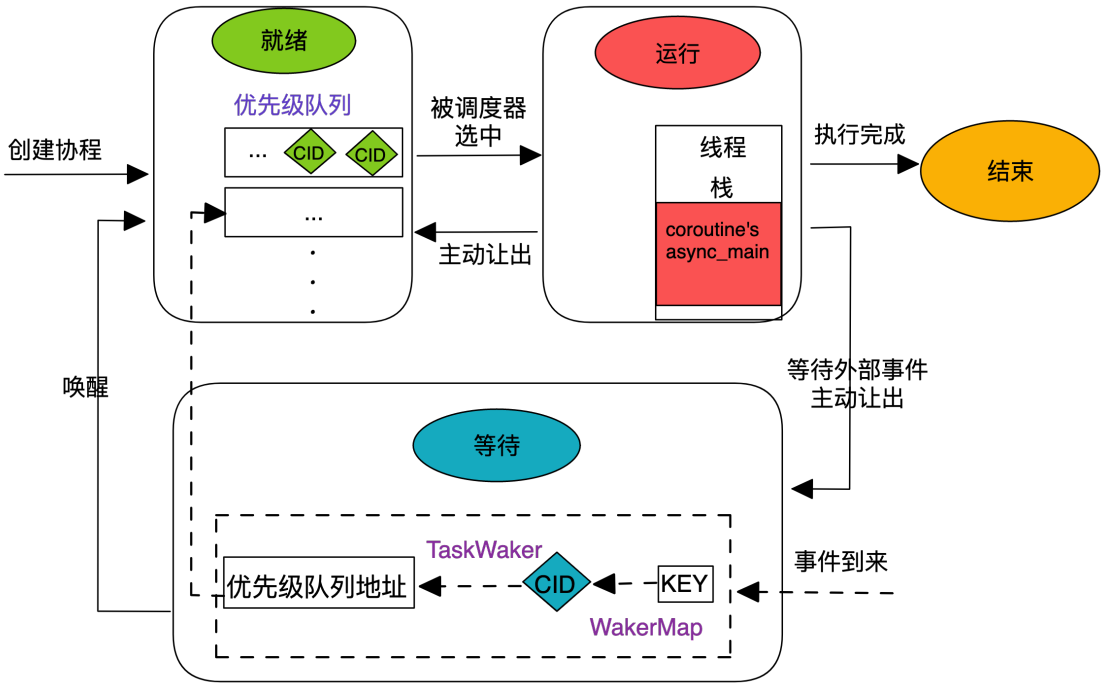

图 3.3 协程的生命周期中的状态转换

Fig. 3.3 State Transitions of A Coroutine

的协程 ID。本文在这里引入了一个名为 WakerMap 的数据结构，它维护整数键值到协程 ID 的映射。对于等待协程和执行唤醒操作的协程，需要在创建时传入相同的 key 值来建立一个联系。当协程等待外部事件时，将参数传入的整数和协程 ID 组成的键值对插入到 WakerMap 中，然后当另一个协程完成外部事件时，可以直接通过 Key 值找到对应的的那个需要被唤醒的协程；如果执行外部事件和唤醒操作的协程被安排先执行，它在查询 WakerMap 唤醒另一个协程时会发现空项而不进行额外操作，这并不影响程序的后续执行。之后，等待外部事件的协程在执行时，会发现等待的事件已经到来（比如 buffer 已经写入足够的数据），从而不会进行函数返回和让出线程栈。

基于以上的设计思路，可以将这个模型的协程调度概括为考虑整个系统（包括用户态和内核）中处于就绪状态的协程，拥有最高优先级协程的进程将在每次时钟中断后由内核调度。然后在该进程内执行协程的优先级调度。如果有协程在用户态的优先级高于内核，则用户态的程序会被优先执行。

3.4 本章小结

本章介绍了协程模型的设计细节，包括优先级队列的组织形式，协程控制块的存储与获取策略，调度器的工作流程，展现了一个能独立运行的完备的协程运行时。该模型具有两个阶段。一是作为函数库，向进程提供一种可用于异步执行

的任务调度模块，完全由进程本身控制。二是修改操作系统内核，通过共享内存上的优先级位图共享用户态的协程信息到内核，使得操作系统可以感知到用户态的调度任务的存在，并进行系统级的调度，把系统中各个独立的地址空间中的调度任务整合到一个统一的调度框架之中。最后描述了本课题提出的协程异步唤醒的工作方式，以及协程运行的生命周期中的各个阶段与状态转换过程。

本章设计的调度模型成功将协程升级为系统级的基本调度单位，并且没有为协程引入与线程同级别的内存资源，解决了子课题一。但在协程的调度过程中增加了访问位图的开销，需要在实验中进行量化。所以本文将在下一章中介绍本模型的实现过程，然后在第五章中通过实验验证协程作为系统级基本调度单位时与线程的性能差异。

第 4 章 调度模型的实现

本章介绍基于 Rust 语言的模型实现。首先，本章利用 Rust 语言提供的异步编程模型封装出能在内核和用户态运行的协程数据结构，解决了子课题二：利用协程向内核提供通用的异步任务接口。对应本章第一、二节。其次，在统筹课题一和二时遵循子课题三中提出的“函数库 + 少量的内核开发依赖”的研究思路，本章将运行器与协程管理器实现为函数库，这两个模块包括了调度模型的主要内容。对应本章第三、四节。在内核中仅实现位图，降低协程的创建与运行过程中对于内核的依赖，尽量保持其低开销的轻量级调度任务特征，并使得此调度模型更具可移植性。内核的修改过程对应本章第五节。

4.1 轻量级调度任务的封装

本文使用协程作为轻量级任务的载体，对应的数据结构称之为协程控制块。协程控制块是系统中占据实际内存，用于记录和描述协程的运行状态的数据结构。cor_id 是协程的唯一标识，可以用于索引和与其他协程区分。prio 是协程的优先级，本文提供的实现仅允许在创建协程时初始化协程的优先级。当然也可以提供一个修改优先级的接口，这需要记录有可能需要变更优先级的协程 id 以便后续查找。

定义协程控制块数据结构为表4.1所示。

future 里面封装了一个异步程序结构，它的数据类型是 Future。在代码块或函数声明前用 async 关键字修饰，即可得到一个 future。

’static 是 Rust 独有的生命周期语法，目的是用来关联函数的不同参数及返回值之间的生命周期，一旦他们取得了某种联系，Rust 就会获得足够的信息来支持保证内存安全的操作，并且阻止那些可能会导致悬垂指针或者其他违反内存安全的行为。简而言之生命周期就是引用的有效作用域，之所以引入这个概念主要是应对复杂类型系统中资源管理的问题。引用是对待复杂类型时必不可少的机

表 4.1 协程控制块数据结构

Tab. 4.1 Data Structure of Coroutine Control Block

访问权限	成员变量名	数据类型
pub	cor_id	CorID
pub	prio	CorPrio
pub	future	Mutex<Pin<Box<dyn Future<Output=()> + ’ static + Send + Sync> > >

制，毕竟复杂类型的数据不能被处理器轻易地复制和计算。系统将异步程序结构 future 保存在堆上以降低协程上下文切换与恢复的设计难度和实现的复杂程度。所以 future 总是以一个全局变量的形式存在，每次访问必然是以通过引用获取其数据，将其生命周期声明为 static 使其常驻系统中，等待手动释放是必要的。

Mutex 代表互斥锁，用于处理多线程之间的变量读写。协程最终会被存放在堆内存中通过调度器被多个线程访问，但是同一个协程同一时刻只能出现在一个线程之中执行。所以对于 future 必须是互斥的。

Sync 和 Send 是 rust 为安全并发提供的两个至关重要的支持，但绝大多数的文献和 Rust 官方文档每当谈到它们就只是直接抛出它们的语义：

（1）实现了 Send 的类型，可以安全地在线程间传递所有权。也就是说可以跨线程移动。

（2）实现了 Sync 的类型，可以安全地在线程间传递不可变借用。也就是说可以跨线程共享。

但是这简单的描述并不能成为实现中必须使用它们的理由。需要透过更深层次的概念来阐述原因。

并发编程中必须使用锁或是类似的机制，Rust 中同样提供了并发编程所用的锁：RwLock<T>，Mutex<T> 等。但是这些锁在实现时都被 Sync，Send 限制了使用场景：只有当类型 T 实现了 Sync/Send，RwLock<T> 或是 Mutex<T> 才会实现 Sync/Send。也就是说如果 T 不 Sync，就不能让多个线程同时拿到 T 类型对象的不可变引用。这个看似不合理的强制要求其实基于一个理论问题：并行只读不会导致内存不安全，而且 Rust 中的 Cell、RefCell 就是拿不可变引用改变内部数据的典型用例。问题的本质其实在于：Rust 的不可变引用并没有对内部可变性做过强的约束，Rust 允许通过 unsafe 关键字直接操作地址，不能假定不可变引用一定是“内部不可变”的。

再回到 Sync 的定义，符合这个要求的类型有两种：

（1）第一种类型不能通过它的不可变引用改变它的内部，它所有的公共域（public field）都是 Sync 的，然后所有的以 &self 作为接收者（receiver）的公有方法也都不改变自身（或返回内部可变引用）；

（2）第二种类型当然所有的公共域都得是 Sync 的，但它可能存在以 &self 作为接收者的公有方法能改变自身（或返回内部可变引用），只不过这些方法本身得自己保证多线程访问的安全性，比如使用锁或者原子；

通过例子来具体说明：

（1）自然实现 Sync 的类型，即所有域（field）全部 Sync，那它所有的公共域显然都是 Sync 的。其以 &self 为接收者的公有方法也只能使用域内的公有方法，

显然都是不可变的。

（2）Mutex<T>，多个线程不能同时通过其不可变引用持有 T 的“可变引用”，也不可能同时持有“可变引用”和“不可变引用”，但可以同时持有“不可变引用”。所以：如果 T 实现 Sync，则 Mutex<T>: Sync；否则，T 将实现!Sync，则 Mutex<T>: !Sync。

关于 Send 和 Sync 的联系，大多数文献都会说“只要实现了 Sync 的类型，其不可变借用就可以安全地在线程间共享”。当然通过上述的分析此问题已经很清晰了，为什么 &T: Send 的要求是 T: Sync；而 &mut T: Send 的要求却更宽松，只需要 T: Send —因为 &mut T 的 Send 意味着 move，而 &T 的 Send 意味着 share。要想多线程共享 &T，T 就必须 Sync。

所以，Rust 的两条简单的定义容易让使用者误解 Sync 和 Send 的语义—它们是使用锁的要求，而非提供功能的某种语言机制。rust 的可变引用要求过于严苛导致很多时候必须使用不可变引用来改变自身，所以 Sync 是用来标记不可变借用可线程安全地访问的。至于可变引用，因为永远只同时存在一个可变引用，且其不与不可变引用共存，所以以可变引用为接收者（receiver）的方法永远是线程安全的，无需其它的约束。对于上文提到的本实现中必须使用的 Mutex<T>，应该保证 T 是 Sync 的，否则它就和 RefCell<T> 没有区别，只不过原本因为 RefCell 会 panic 的逻辑现在会因为 Mutex 死锁。

Pin 是使用 Rust 的 Future 实现协程时必不可少的一个特性，它可以防止一个其修饰的数据类型在内存中被移动。在此业务下必须使用 Pin 并不是一件特别明显能被马上察觉到的事，接下来说明使用它的必要性。

在 Rust 中所有的数据类型根据其在内存中移动的性质可以分为两大类：

(1) 类型的值可以在内存中安全且自由地被移动：如整形，字符串，布尔值等几乎所有类型；

(2) 自引用类型。

举个例子来说明自引用类型。假设创建了一个结构体 Test，其内部包含一个类型为字符串的成员变量 value，和一个类型为字符串引用的 ptr_to_value。现在初始化这个结构体，先对 value 赋值一段字符串常量，然后将 ptr_to_value 赋值为 value 的指针。此时就得到了一个自引用结构。如图4.1所示。

在 Test 结构体中，ptr_to_value 是一个指向自己所属的结构体内的一个变量的指针。当移动它的时候，结构体中的字段就会改变他们的内存地址，而不是他们的值。所以 ptr_to_value 指针仍然指向之前的地址 A，但地址 A 现在没有一个合法的字符串值，之前地址 A 的数据已经被移动到地址 B 了，该地址可能被其他的值写入了。所以，现在 ptr_to_value 是非法指针。这在程序运行中是无法忍受的，

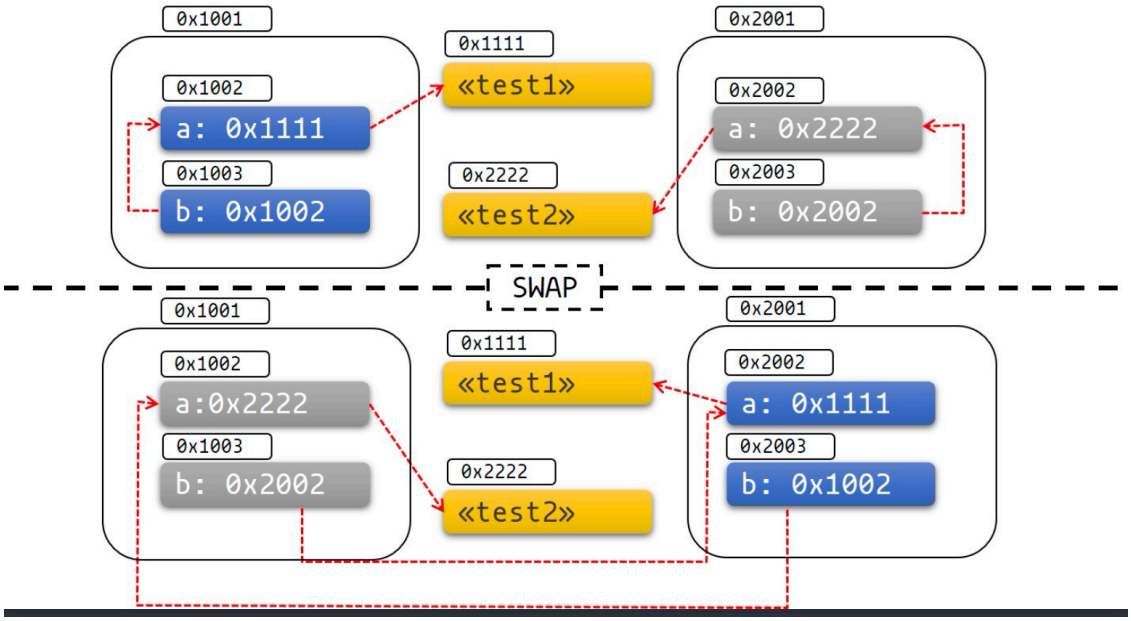

图 4.1 自引用结构示意图

Fig. 4.1 Self-referencing Structure

一般情况下非法指针会造成程序崩溃，最坏的情况下，他是一个可以被恶意利用而对整个系统发起攻击的漏洞。

这就是自引用机构的局限，而且 Rust 的编译器在编译阶段就会通过报错禁止这种结构运行。Pin 就是 Rust 提供的一种支持自引用结构的工具。它可以防止一个类型在内存中被移动。而 async 会将代码块或是函数转换为 Future，Future 的底层又依靠 generator 实现，即状态机。如果 async 代码中存在跨 await 借用，那就会在最终生成的状态机中出现跨 yield 点的变量引用，生成 Rust 禁止使用的容易造成悬空指针的自引用结构。

所以，为了保证实现的协程能够满足各种业务场景之下的合理且自由的变量访问需求，在使用 Rust 的 Future 实现协程时，必须将类型限制为 Pin。

4.2 协程的异步运行原理

本课题基于 Rust 语言提供了一个参考实现。首先介绍 Rust 语言提供的异步支持，这是后续工作的前提。Rust 语言本身没有直接可用的协程机制，但是提供了一套基于关键字 async/await 的异步运行框架，开发人员可以以此定制出满足各种需求的协程库。async 关键字可以用来修饰普通函数，并将其转换为一个叫做 Future 的程序结构。表4.2为 Future 原型。

Future 代表一个可在未来某个时间获取返回值的异步计算任务，每个异步任务分成三个阶段:

1. 轮询阶段 (The Poll Phase)。一个 Future 被轮询后，会开始执行, 直到被阻

表 4.2 Future 原型

Tab. 4.2 Future Prototype

```
trait Future {
  type Output;
  fn poll(self: Pin<&mut self>, cx: &mut Context<_>) -> Poll<Self::Output>;
}
```

塞。通常把轮询一个 Future 这部分称之为执行器 (executor)；

2. 等待阶段. 事件源 (通常称为 reactor) 注册等待一个事件发生，并确保当该事件准备好时唤醒相应的 Future；

3. 唤醒阶段. 事件发生, 相应的 Future 被唤醒。现在轮到执行器 (executor), 就是第一步中的那个执行器，调度 Future 再次被轮询，并向前走一步，直到它完成或达到一个阻塞点，不能再向前走, 如此往复, 直到最终完成. 例如我可以创建一个与某个 IP 建立 TCP 连接的 struct，在构建时完成建立连接的工作，然后实现 Future trait 时检查连接是否已经建立完成。根据建立情况返回 enum Poll 中的两个元素之一。

为了获取这个异步计算任务的执行状况，Future 提供了一个 poll 函数用于判断该它是否执行返回。

1. Poll::Pending: task 还在等待；

2. Poll::Ready(result): task 携带 result 返回。

实际上，基于 async 定义的函数和代码块会直接被编译器编译为 Future。但是 async 函数或代码块无法显式地返回 Pending，因此一般只能完成一些简单的调用其他 Future 的工作。复杂的异步过程通常还是交由实现了 Future trait 的类型完成。

在很长一段时间内，Future 的唯一参数是 &Waker 类型，在标准化 Future 的版本中，考虑到将来可能的扩展需求，Rust 的作者们将 Waker 包裹在 Context 类型内，这样将来需要拓展 Future 时可以避免影响使用旧版本 Rust 编写的软件出错。换句话说，在目前的阶段内，std::task::Context 类型基本等价于 std::task::Waker 类型，Context 目前的作用只用于获取 Waker，而 Waker 的作用是用于提醒 Future 执行器（一般来说，Future 执行器常称为 executor）该 Future 已经准备好运行了。同样以上面的建立 TCP 连接的例子来说，在网络卡顿时，进行一次 poll 可能都没有建立连接，如果没有设置 timeout 之类的东西的话，就需要进行多次 poll。这样的 Future 多了以后，常见的做法是将所有的 Future 都存储在一起，然后另起一个线程用于循环遍历所有的 Future 是否已经 ready，如果 ready 则返回结果。这就是一个非常简单的单线程 executor 的雏形。也就是说，executor 是一个托管运行 task

的工具，类似于多线程，多线程要成功运行需要一个调度器进行调度。但是多线程至少需要语言层面甚至操作系统层面的支持，而在 Rust 的官方文档没有任何关于 executor 的实现。实际上，Rust 选择将 executor 的设计和实现交给开发使用者，语言本身只保留相关的交互接口（类似于 C++，并没有一个官方的 executor 实现，唯一熟知的在语言层面提供支持的只有 Golang 的 goroutine）。

上面讲述的基于轮询所有的 Future 是否已经完成的 excutor 是最低效的一种做法，当 Future 多了以后会带来相当多的 CPU 损耗。考虑到这点，Rust 还提供了一种机制可以用于通知 executor 某个 Future 是否应该被轮询，当然这只是其中的一种解决方式，实际上 Waker 的 wake 函数可以被实现为任何逻辑，取决于 executor。Waker 包含以下几个内容：

（1）Context：上下文，功能是包装 Waker 的入口；

（2）RawWaker：生成 Waker；

（3）RawWakerVTable ： 一个虚函数指针表 (Virtual Function Pointer Table, vtable)，指向 RawWaker 对象的指针的列表；

（4）Waker：通知任务准备启动的入口；

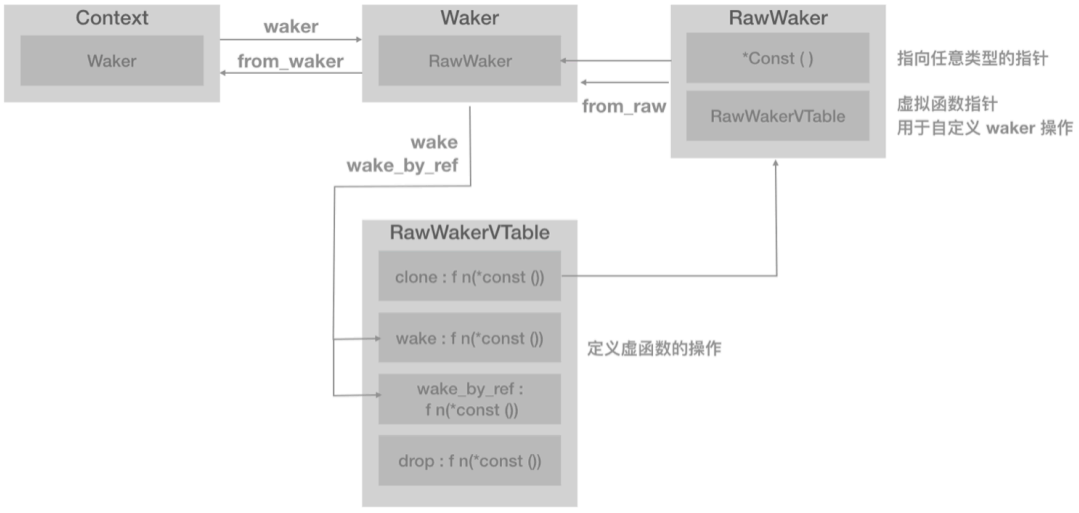

图 4.2 Waker 模型

Fig. 4.2 Waker Model

Context 结构里面定义一个 Waker，实现了和 Waker 的相互转换。RawWaker 定义了一个指向任何数据类型的指针，和虚函数表，用来实现和虚函数的绑定。RawWakerVTable 定义了虚函数的几种操作，其中 wake 和 wake_by_ref 和 Waker 的实例对应。

Waker 里面有两个重要的操作：wake() 和 clone()。wake() 的作用是让 task 再次启动，并且获得 pending 之前的信息 (堆栈和函数操作)。这个代码里面的 vtable

和 data 就是 RawWaker 里面的数据。中间有一个操作 mem::forget(self)，forget 的作用是让 self 实例进入一个不能操作的状态，但是实例还在，可以下次调用。forget 是调用了 mem crate 里面 ManuallyDrop::new() 。ManuallyDrop 能够控制编译器，让其不要自动去调用实例的解构函数 (destructor)，这样就能保存之前函数运行的信息。forget 调用 ManuallDrop::new() ，就是把 self 实例创建成一个 ManuallyDrop，获得之前 Waker 的信息，这样在切换到另外一个 runtime 或者 thread 的时候，之前的 Waker 信息就同步过去，不会随着进程结束而解构。clone() 的作用是 clone 这个 waker ，这个用在 task 在要进入 Pending 之前，把获得数据 clone 一份到一个全局的数据结构里面，下一次调用的时候，不会丢失。

Waker 之所以没有设计为 trait 形式，主要是 clone 函数，受限于 Rust 的 trait object safety，trait 中的任何函数的参数或返回值如果包含 Self 且有 type bound Sized，则不符合 trait object safe 规范，这样的 trait 可以被定义，可以被实现，但是无法与 dyn 一起进行动态绑定。而 clones 函数又是必须的，因为 future 可能还会接着调用 future 的 poll 方法，就需要再 clone 一个 context 传入。或许可以用 Box<dyn Waker> 或者 Arc<dyn Waker> 之类的，但是这些都不比 raw pointer 灵活，所以最终 Rust 还是选择定义一个包含函数指针的 struct。

考虑一种直观的设计：在 task 进入 pending 的时候用一个全局的方法来保存状态，下次启动时再调用。这种工作方式对比于 Waker 存在两个缺陷。一个主要原因是出于性能的考虑，访问记录状态的全局变量的数据结构是一个互斥操作；另一个原因，task 很多时候是以闭包的形式出现，闭包可以获取所在进程的环境变量，但是自己没有 heap allocation 。这样的话，其实就很难通过全局或者参数传递的方法来实现。

所以可以发现，async/await 可以说是异步编程领域的标志，但在 Rust 中这两个关键字只是起到语法糖的作用，其异步机制的核心在于 Future。async 用于快速创建 Future，不管是函数还是代码块或者 lambda 表达式，无论其中的逻辑编写的多么复杂，或是简单到直接返回一个常量，都可以在前面加上 async 关键字快速变成 Future。这个转化分为两步：

（1）把 async 块转化成一个由 from_generator 方法包裹的闭包；

（2）把 await 部分转化成一个循环，调用其 poll 方法获取 Future 的运行结果；

这里的大部分操作还是比较符合逻辑容易理解的的：因为遇到了需要 await 完成的操作，所以运行一个循环去不停的获取结果，完成后再继续。注意到这里，当 x 所代表的 Future 还没有就绪时（即便在本例中并不会存在这种情况），loop 的运行会来到一个 yield 语句，而非 return。因为 break 和 return 会带来以下问题。

break 容易理解，loop 循环直接结束，如果函数后续还有其它操作那么就会被

表 4.3 generator 原型

Tab. 4.3 Generator Prototype

```
enum GeneratorState<Y, R> {
  Yielded(Y),
  Complete(R),
}
trait Generator {
  type Yield;
  type Return;
  fn resume(&mut self) ->
    GeneratorState<Self::Yield, Self::Return>;
}
```

执行。但这显然不符合 await 的语义，因为 await 需要 block 在当前的 Future 上，而不是忽略其结果继续运行后续代码。

对于 return，如果这个 Future 暂时不能 await 出结果，为了应该尽快完成上层函数的 poll 操作，不 block 当前 Executor 对其他 Future 的执行，直接返回一个 Poll::Pending——到目前为止都没什么问题，但问题的关键在于，如果这个 Future 被 Waker 唤醒后，再次被 poll 的时候会把 await 之前的所有代码都再运行一遍，这显然也不是预期的结果。不论是操作系统的线程还是 Future 这种用户态的 Task，任务调度切换显然是需要有一个 “断点续传” 的基本能力。对于系统线程来说，操作系统进行线程调度时，会将上下文信息保存好，以遍后续线程再次被运行时可以通过上下文切换再次恢复运行时的状态。Rust 的异步就是通过 generator 实现该功能。

generator 是 Rust 早期对于异步的尝试的产物，现在用作了 Future 的底层。在 Python，JavaScript 等语言均可以见到 generator 的身影，它并不是 Rust 首创的程序概念，而是计算机科学中的一项经典技术。generator 负责生成执行体 (或者执行函数)，一个执行体由可执行代码和数条 yield 语句组成，每一个 yield 把执行体切分成了不同的执行阶段。对于执行体的执行，会从上一次中断位置执行，直到遇到下一个 yield(又或者称为中断点/保存点) 位置，此时会保存执行体的执行状态，包括更新之后的局部变量等。表4.3是 generator 的原型。

generator 拥有自己的状态，当通过调用 resume 函数来推进其执行状态时，它不会从头来过，而是从上一次 yield 的地方继续向后执行，直到 return。调用 Generator 的 resume 函数会恢复 Generator 的运行，如果还没有启动 Generator 的话则会启动 Generator。在执行 Generator 的过程中，如果遇到 yield 表达式，那么 Generator 就会在这个 yield 点挂起，并产出 yield 表达式的值：GeneratorState::Yielded(Y)。当

再次调用 resume 方法时 Generator 就会在挂起的 yield 点恢复运行。在运行过程中，如果遇到的是 return 语句或者 Generator 末尾的最后一个表达式，那么生成器执行完毕，并返回 GeneratorState::Complete(R)，R 就是 return 语句或者末尾表达式的值。如果 Generator 已经执行完毕，返回了 GeneratorState::Complete，那么当再次调用 Generator 的 resume 方法时将会导致 panic。

对上文内容进行总结就是，通过 rust 提供的 async 关键字直接修饰普通函数，代码块或闭包可以直接得到一个 Future 结构，每一个 Future 的本质其实都是一个 Generator，两者可以互相转换。

Generator 的整体设计和实现思路其实就是一个状态机，每次 yield 就是一次对 enum 实现的状态进行推进，直到最终状态被完成。过程中与状态相关的数据还会被存储到对应的枚举类型里，以遍下一次被推进时使用。容易注意到注意到一个 generator 的 resume 函数和 Future 的 poll 函数似乎有几分相似——都要求方法的调用对象是 Pin 住的，且都会返回一个表示当前状态的枚举类型。对应的 generator 代码在接下来的 Rust 编译过程中，也正是会被变成一个状态机，来表示 Future 的推进状态。

4.3 协程管理器

4.3.1 概述

协程管理器是协程的上一层机构，负责管理所有协程的状态及其转移过程。它系统启动后会占据实际物理内存的一个数据结构，存储在堆中，方便其在进程中的任意位置都可以被获取。tasks 保存协程 ID 到协程内存指针的映射。前文提到，Future 被创建为协程之后有可能生成自引用结构，所以通常选择固定在内存中直到被释放。但由于 Pin 的存在消除了自引用结构的影响，系统出于空间优化的目的对其进行内存移动，以及应用程序主动触发其内存移动都是允许的。比如随着协程的创建和释放，在调度队列中的协程会进队和出队。当规模达到一定程度时，频繁移动大量的协程会造成性能损耗。为了追求极致的性能，在本实现中不会主动触发协程的内存移动，而是针对协程 ID 进行操作，并建立起协程 ID 到协程指针的映射，当需要协程的完整信息时，可以直接通过协程管理器获取。

管理此映射关系的数据结构本实现选择 BTreeMap，它的底层通过 B-Trees 实现，代表了缓存效率和实际最小化搜索中执行的工作量之间的基本折衷。理论上，二叉搜索树 (BST) 是搜索数据结构的最佳选择，因为完美平衡的 BST 执行查找元素所需的理论最小比较次数在对数级别。然而，在实践中，这样做的方式对于现代计算机体系结构来说是非常低效的。特别是，每个元素都存储在自己单独的堆节点中。这意味着每次插入都会触发堆调整（重新分配节点并排序），并且每次比

表 4.4 Excutor 数据结构

Tab. 4.4 Data Structure of Excutor

访问权限	成员变量名	数据类型
pub	tasks	BTreeMap<TaskId, Arc<UserTask> >
pub	task_queue	Arc<Mutex<Box<TaskQueue> > >
pub	waker_cache	BTreeMap<TaskId, Arc<Waker> >
pub	task_num	usize

较都应该是缓存未命中。由于在实践中这些都是成本很高的操作，所以需要重新考虑 BST 策略。

BTreeMap（同 B-Tree）反而使每个节点在一个连续数组中包含 B - 1 到 2 * B - 1 个元素。这样做使得分配次数减少了 B 倍，并提高了搜索中的缓存效率。但是，这确实意味着搜索将不得不平均进行更多的比较，但这只是在对数的数量级上进行常数的增加。是一种在实践中效率很高的折中方案。

task_queue 通过协程 ID 和 tasks 提供的映射关系保存协程的调度顺序，并实现调度算法。

Future 通过接口封装被创建为协程之后，其控制权会交给协程管理器，由协程管理器将协程放置在其内部的数据结构中，并实现提供给调度器的接口完成优先级调度。另外一个特殊的数据结构叫做 waker_cache，它用于耦合协程与其唤醒所需的数据结构，使得只需获取到协程的特征信息（协程本身或是协程 ID）就可能启动它的唤醒操作。task_num 用于记录当前进程内尚存的协程数量，包括正在执行的，就绪的和等待被唤醒的。

4.3.2 优先级队列

协程管理器中的优先级队列将实现调度算法并提供 add_coroutine 和 pop_coroutine 这两个关键接口来为协程管理器插入或弹出协程。为了减少与优先级队列即调度算法的耦合，协程管理器中仅保存优先级队列指针，方便系统灵活选择已实现的调度算法。

定义优先级队列数据结构的唯一成员变量为：queue: Vec<VecDeque<TaskId> >。

优先级队列本质上是一个二维的数据结构，将协程 ID 存储于其中的网格中。第一维的长度固定为优先级数。因为优先级数是内核中规定的常数，所以第一维的长度不需要更改，出于这个特点，系统选取 Vec 作为第一维的容器。Rust 中的 Vec 可以被看作是一个可以动态调整大小的数组。它是一种可以按地址顺序存放一串元素的数据结构，在顺序遍历时拥有所有数据结构中最高的访问效率，只需

保证同一 vector 中的元素类型相同。当试图获取某一确定优先级（索引）的协程时，Vec 可以达到 O(1) 的时间复杂度。它由三部分组成：

（1）指向堆上的 vector 数据的指针；

（2）长度；

（3）容量；

Rust 提供了为其扩容并通过内存移动重新分配元素的接口，但这个功能对本实现的需求来说是一个超集，当然它的存在并不会影响顺序访问的使用效率。

第二维用于存储相同优先级的协程 ID，即通过第一维的优先级索引到所有的拥有同一优先级的协程。显然，这需要一个可以动态扩容的容器，因为不能限制协程管理器所能管理的协程数量，也不应该预分配大量的空闲内存空间给容器。而且，为了保证调度时效的公平性，同一优先级的访问顺序应当遵守先进先出（FIFO）。针对这些需求，第二维需要使用双端队列。VecDeque 是 Rust 中提供的一个使用可增长的循环缓冲区（ring buffer ）实现的线性容器，可以在两端都进行时间复杂度为 O(1) 的插入和删除。

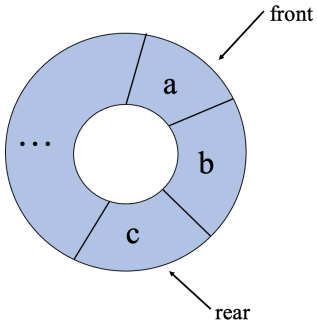

图 4.3　协程在优先级队列中的排列方式

Fig. 4.3　The Arrangement of Coroutines in The Queue

透过上层的逻辑结构，协程 ID（可以直接索引到协程控制块）在内存上的分布就如图所示。由于协程的执行特点和应用场景，通常每个协程携带的任务量较小，但数量较多。所以协程会频繁地插入到队列或从中被删除，或是在等待列表中等待被唤醒，并不会大量地堆积在队列中。所以环形队列适用于此场景。即使环形队列的容量无法满足需求，VecDeque 也可以按 2 的幂进行扩容，因此其扩容次数被控制在对数的数量级上，是理想的性能较高的可扩容数据结构。

数据结构的设计与选取是软件实现中及其关键的一环。顶层设计是软件质量的根基，但是数据结构的好坏（是否满足业务需求、具有良好性能）可以直接影响这个根基或是发挥出它的潜能。本实现所选取的数据结构是在满足需求的前提下相对性能最佳的选择。下表是 Rust 官方提供的性能参考。

表 4.5 Rust 数据结构性能参考

Tab. 4.5 Performance Reference of Rust Data Structures

数据结构	操作		
	get(i)	insert(i)	remove(i)
Vec	*O*(1)	*O*(n)	*O*(n)
VecDeque	*O*(1)	*O*(min(i, n - i))	*O*(min(i, n - i))
LinkedList	*O*(min(i, n - i))	*O*(min(i, n - i))	*O*(min(i, n - i))
HashMap	*O*(1)	*O*(1)	*O*(1)
BTreeMap	*O*(log(n))	*O*(log(n))	*O*(log(n))

协程调度器初始化时需要完成优先级队列的创建。从索引 0 开始创建长度为优先级数的 Vec，并将每个元素初始化为空的 VecDeque。这个操作建议使用 Rust 的 collect 函数来完成，它基于迭代器实现，可以把迭代的所有元素组合成一个新的集合。

不使用传统的 for 循环进行迭代创建而要使用迭代器的原因在于，在 Rust 中，如果没有编译器优化，这两种写法其实是不等价的。由于 Rust 的 Vec 类型的下标访问每次都会做边界检查，越界会直接 panic 使程序退出，而循环本身也会做一次边界检查（0 加到 5 每次循环都要检查），这样就出现了多余的边界检查，造成额外开销，虽然编译器可能会进行优化，但是这种写法依然不值得提倡。如果迭代一些复杂的类型，可能依然会出现问题。

Future（异步函数）被封装为协程控制块之后，通过 add_coroutine 函数将控制权转移给协程管理器。此时协程以及具备了协程 ID（由协程管理器统一生成）和优先级（由使用者指定或采用默认值），所以可以根据优先级直接索引到对应的 VecDeque 并向其中插入协程 ID。插入操作调用 VecDeque 的 push_back 函数完成，它会获取 VecDeque 底层的环形队列的尾部节点，修改其 next 指针为将要插入的新节点，然后更新尾节点的值为新节点，完成插入操作。

插入协程将引起优先级位图的变化，这个实现将在 4.5 节中介绍。

协程执行器通过协程管理器获取协程执行，协程管理器通过优先级队列提供的接口得到协程。优先级队列从最高优先级 0 开始遍历所有第一维元素，依次询问 VecDeque 是否为空。当遇到第一个非空 VecDeque 时，通过 pop_front 函数弹出头部协程，协程执行器就可以获取到当前进程中处于就绪状态的拥有最高优先级且最早被插入的协程。

表 4.6 Waker 数据结构

Tab. 4.6 Data Structure of Waker

访问权限	成员变量名	数据类型
pub	tid	TaskId
pub	prio	usize
pub	queue	Arc<Mutex<Box<TaskQueue> > >

4.3.3 异步唤醒机制

当协程封装的异步任务没有在中间设置主动让出点时，它将退化为普通函数，被调度器选中并被执行器一次执行完成，随即其占据的内存会被释放。否则，协程执行器会在每个协程的主动让出点收到函数返回信息，然后协程将离开优先级队列进入等待状态。在被唤醒之前，协程都不会出现在优先级队列中，所以优先级队列在执行调度算法时只会访问到所有处于就绪状态的协程，此设计和实现是为了追求更高的性能。但是，此时协程依然被协程管理器控制。waker_cache 保存协程 ID 到 Waker 的映射，如前文所述，Waker 是一个框架同时也是一个占据物理内存的数据结构，其内部的虚函数表中包含了实施唤醒行为的函数 wake。本实现将以下数据结构实现为 Waker。

先来解释“将数据结构实现为 Waker”的含义。Wake 是一个 trait，实现了 Wake 这个 trait 的数据结构可以被称之为 Waker。任何 struct 都可以实现 trait，它是 Rust 中的一个极具代表性的程序概念，比较接近于 Go 和 Java 中的 Interface，但 trait 的职责于用处远比 interface 多。所有的 trait 中都有一个隐藏的类型 Self(S 必须大写)，代表当前实现了此 trait 的 struct 的具体类型。trait 中定义的函数，也可以称作关联函数 (associated function)。函数的第一个参数如果是 Self 相关的类型，且命名为 self(s 必须小写)，这个参数可以被称为“receiver”(接收者)。具有 receiver 参数的函数，称为“方法”(method，来自于面向对象编程的概念)，可以通过变量实例使用小数点来调用。没有 receiver 参数的函数，称为“静态函数”(static function，依然是面向对象编程的概念，可以不依赖于具体的实例对象而被调用)，可以通过类型加双冒号:: 的方式来调用。在 Rust 中，函数和方法在实现上没有本质区别，只是因为 Self 和 self 的存在造成了使用上的差异。

本实现中使用到的是 trait 的泛型特性，它是动态类型编程的一种风格：一个对象的有效语义不是由继承特定的类或者实现特定的接口，而是由“当前方法和属性的集合”决定。也就是说针对于系统的设计，不需要继承实现语言提供的标准接口和类，而是可以根据实际的场景自由定制恰当的成员变量，实现相应的行为。协程的唤醒所依赖的外界信息越少越好，这样可以降低唤醒模块与其它模块的耦合程度和维护成本，并有可能提升潜在性能—唤醒需要查询的信息更少了。

在协程执行器每次获取到协程之后，执行协程之前，都会判断该协程是否已经具有 Waker。显然，每个协程第一次被调度时必然会由协程管理器为其创建 Waker，实现为 TaskWaker。它包含协程 ID，优先级，以及优先级队列的指针。优先级这个成员变量是冗余的，因为它可以由协程 ID 映射获取到。但考虑到查询操作的开销，建议牺牲一个 8 字节的整形内存空间作为缓存，以提高运行效率。基于这三个成员变量，可以对唤醒操作做出如下实现。

(1) 创建 TaskWaker 的构造函数 new。要求传入参数 id（协程 ID，从协程 ID 生成器处获取，以保证进程内唯一），prio（协程优先级，被指定或是设为默认值），以及优先级队列的指针或可变引用。值得注意的是，变量的赋值有两种语义，move 和 copy，C++ 和 Rust 作为系统级开发常用的编程语言，都实现了这两种语义，但在形式上有显著区别。对于 C++ 的开发者来说，函数传参时默认使用的是 copy 语义，也就是把值在内存中拷贝一份。所以在实现此函数时要特别注意传入的是地址还是值，否则即使可以通过编译器的检查，也会在执行阶段把协程 ID 插入到一个立即会被释放的优先级队列中。

(2) 创建执行具体唤醒操作的函数 wake_task。此函数不需要传递参数，依靠 TaskWaker 自身缓存的优先级队列指针和协程 ID 就可以把协程重新放回调度器中。

(3) 为 TaskWaker 实现 Wake trait。任何试图实现 Wake trait 的类都需要实现 wake 函数，它不需要任何参数，任何实现了 Waker trait 的类都可以调用 wake 函数，并将其本身作为 Wake 类型在各模块之间传递。在 wake 函数内部直接调用（2）中的 wake_task 函数即可完成唤醒操作。

4.3.4 管理器接口

软件工程中的依赖倒置原则（Dependence Inversion Principle）建议程序要依赖于抽象接口，不要依赖于具体实现。简单的说就是要求对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合。本章节中的将要介绍的接口虽然已经是具体的实现，但依然愿意遵循科学的指导法则。所以在定义接口功能时不依赖于 Rust 语言本身的特性进行设计，而是定义普遍适用于高级语言的程序功能进行实现。

从需求出发，考虑到线程的创建过程：通过接口传入普通函数，此为函数申请相应的内存，初始化成员变量，包括状态，ID 等，然后将创建完成的线程控制块插入到系统的调度器中准备调度运行。此外，还可以通过额外的函数调用，或者增加线程创建的参数，以控制线程优先级。对于协程，同样的思路，系统希望继承这一类似的流程，当软件需要使用协程时，传入一个异步函数和优先级，然后此异步函数被封装为协程。根据此协程模型的设计，本课题提出如下的接口设

计。

fn add_coroutine_with_prio (async_main: Future, prio = DEFAULT: CorPrio);

此接口要求两个输入参数：一个 Future 和一个协程优先级。前者是 Rust 提供的用于描述异步函数的程序结构，可以使用 async 关键字修饰普通函数而直接获得，在其内部可以通过 await 关键字设置多个主动让出点，当需要等待外部事件时可以立即返回，将执行机会让出其他协程。协程优先级是系统预制的宏，本质上是正整数，具体的优先级数取决于具体的需求，没有固定的标准，本文预制了 8 个优先级，0 为最高优先级，7 为最低优先级，中间逐级降低。当没有传入参数指定优先级时，它将被赋予默认值。

该接口功能的完成建议分发给两个函数辅助实现，可以使的各个模块的功能划分更清晰，也是得整个工程更易于维护。

（1）create_cortoutine 函数。此函数需要以将要执行的异步函数 async_main 和优先级 prio 作为参数传入，然后调用协程控制块模块提供的协程构造接口初始化协程的结构，最后申请获取协程管理器的锁访问优先级队列，将协程的控制权转移给协程管理器。

（2）coroutine_run 函数。此函数需要实现为协程执行器，主要负责轮训协程管理器以获取协程执行，但还需要完成一起的工作以保证整个协程系统正确工作，具体的实现放在 4.6 节中介绍。

如果需要执行一批协程，这个接口应该嵌套在一个异步函数中来批量创建协程。第一个协程创建时，接口需要调用 run 函数启动协程执行器；当协程数大于一个时，表示执行器已经启动，该接口只需要完成协程的创建即可。

4.4 运行器

上文介绍的协程创建接口中调用了一个 run 函数启动协程的执行，它的职责是不断请求协程管理器获取当前进程中处于就绪状态（位于优先级队列）的最高优先级协程，并执行。被执行的协程会以函数返回值的形式告知执行器是否已经执行完毕，如果是中断点的主动让出（返回 Pending），执行器不会采取任何动作，所以该协程被弹出之后到被唤醒前，不会出现在优先级队列中，但依然在协程执行的的管辖范围之内（依靠 waker_cache）；如果是协程执行完成（返回 Ready 并携带指定类型的返回值），会请求协程管理器将其删除。

注意到请求协程管理器获取协程的过程需要使用“”单独封装为一个代码块，这是 Rust 的变量生命周期特性的体现。它的核心概念是，每个使用“”包裹的代码块区域被称之为一个”作用域“，变量的生命周期只存在于自身所处的作用域，离开之后变量将被释放。在此处使用此特性可以简化协程管理器的锁的释放，因

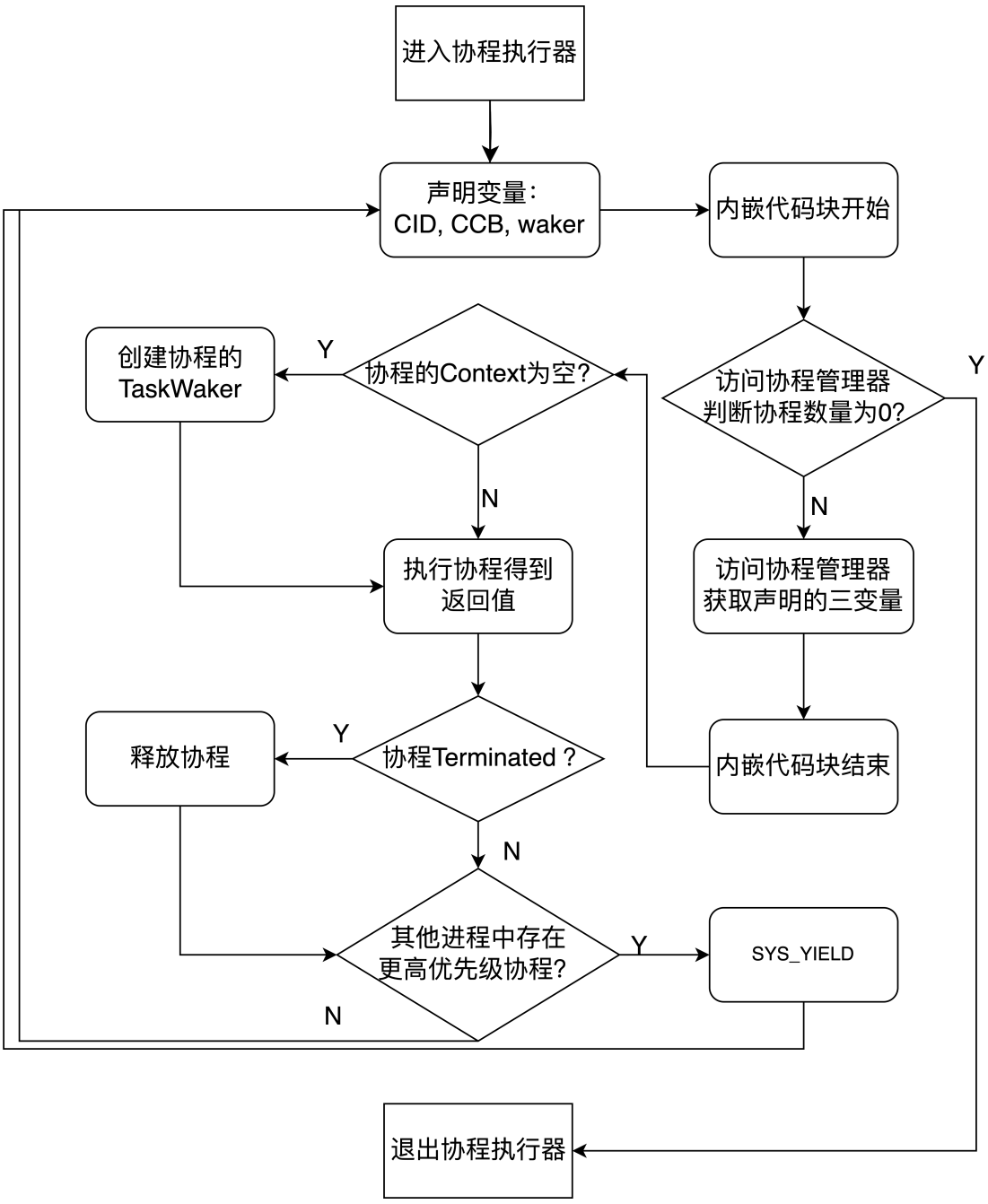

图 4.4 协程运行器流程图

Fig. 4.4 Coroutine Runner Flowchart

为它是要求互斥访问的，所以任何不及时的释放都会造成系统运行效率的降低甚至死锁。在使用不支持此特性的语言实现协程模型时，要注意及时显式释放协程管理器的锁。

执行器在获取到协程的返回值之后，进入到下一轮循环之前，需要查询系统位图以判断其他进程中是否存在更高优先级协程。依靠 4.5 节中介绍的通过系统位图指针直接强制类型转换为 BitMap，就可以得到整个系统中处于就绪状态的协程根据优先级的分布情况，然后与进程的优先级位图比较各自记录的最高优先级，如果其他进程中存在更高优先级协程，则通过 sys_yield 系统调用进入到内核并切换到对应的地址空间，最终调度运行该协程。

4.5 优先级位图

优先级位图用于描述每个进程的协程的优先级布局并将信息提供给内核，使其可以感知到系统内的所有协程，然后根据这些信息做出更灵活的协程调度，使得协程可以作为一种可以被系统感知的调度单位存在，而不是仅作为单个进程之内的软件工程技术使用。

内核会为每个进程创建一个私有的位图，大小为一个物理页（通常为 4KB），初始为全 0，内核可以通过页面起始地址直接读取。内核还会为各个进程开放对应位图的读写权限，使得各个进程可以在插入/释放协程时对位图及时修改。

实现此功能需要严格按照以下步骤：

（1）定义应用程序访问本进程优先级位图的指针（地址）：BITMAP_VADDR。

（2）定义位图所用物理页面的起始分配地址: BITMAP_BASE_PADDR。此地址会将会占用一片连续的物理页组成一个 “物理页面池”，使用进程控制块的编号（PI）来控制偏移（offset）进行分配。

（3）内核需要提供获取当前进程控制块编号的接口，或在地址映射的过程中进程控制块的相关信息通过参数被持续传递，如果没有则需要额外开发。

（4）内核需要提供单一地址映射的接口 page_table_map，如果没有则需要额外开发。利用此接口将（1）中的 BITMAP_VADDR 映射到分配给该进程的物理页面，作为优先级位图结构的存储地址，分配方式为（2）中的 BITMAP_BASE_PADDR 加上进程控制块编号乘以单个页面大小的乘积。

内核从 BITMAP_BASE_PADDR 开始为进程分配物理页面，使用 PID（进程 ID）作为映射计算得到属于自己的位图物理页。由于 PID 在系统中是唯一的，所以线性散列之后得到的物理页也会是唯一的，不会发生冲突。

每个进程在创建时都会由内核为其在 BITMAP_VADDR 和分配到的位图物理页之间建立映射。在用户态为应用程序进行编程时，就可以通过这个指针得到位

图。

pub struct BitMap(usize);

位图的使用方式并不是直接读写大小为 4KB 的页面，而是强制类型转换为 BitMap 结构体，如此就可以为其实现方法，使其作为一个创建好的类对象实例被使用。BitMap 只包含一个 8 字节大小的整形成员变量，一共 64 比特，每一个比特代表系统中是否存在 (1 为存在) 该优先级的协程，从最右端 (小端) 到最左端 (大端) 优先级递减，所以最右边的第一个比特代表是否存在最高优先级协程。以下讨论 BitMap 必须提供的功能。

（1）BitMap 需要提供的第一个函数为查询进程中是否存在某一优先级的协程。依靠位图可以直接将其实现为判断对应的比特位是否为 1，这要求传入一个待查询的优先级 prio，然后把这个优先级左移 piro 位，最后与位图 BitMap 进行与运算，判断结果是否为 0。如果不为 0，则说明该优先级存在协程，即该比特位为 1。注意此处不能判断是否为 1，比如存在次高优先级优先级协程，那么在优先级位图长度为 4 的情景下位图的值可能为 1010，现查询是否存在次高优先级协程，在与 0010 运算之后值为 0010，当判断是否为 1 时就会得出不存在该优先级协程的错误结果。当然这个操作也可以依靠 Rust 提供给整型 usize 的 get_bit 函数完成。

（2）当插入某一优先级的协程到调度器使得该优先级队列的状态从空变为非空时，时需要及时更新位图对应优先级的比特位为 1；当某一优先级队列的协程全部被取出，该优先级队列的状态从非空变为空时，需要及时更行对应优先级的比特位为 0 或 1。通过为 BitMap 实现函数 set 提供此功能，要求传入一个待更新的优先级 prio 和要设置的值（0 或 1）。然后把优先级左移 prio 位，如果需要赋值为 0，就先把移位后的 prio 取反，然后与位图进行与运算，即可保留其他所有位的原值，并置该优先级位为 0；如果需要赋值为 1，则直接将移位后的 prio 与位图进行或运算，即可在不影响其他位的情况下更新该优先级位为 1.

此外，还可以利用 BitMap 提供的 get 功能优化优先级队列的实现：直接从 BitMap 的最右端 (最高优先级) 开始向左扫描第一个 1 比特，利用位扫描替代上文的 Vec 遍历，以提高理论上的搜索效率。

4.6 本章小结

本章所实现的协程可以向内核提供通用的异步任务接口，解决了子课题二。并在子课题三的研究思路的指导下，将调度模型的大部分组件实现为函数库，降低了协程与内核的耦合程度以及协程对于内核的资源依赖。本章从构建轻量级调度单位的数据结构开始，介绍了封装出能在内核和用户态运行的协程需要满足的约束条件，包括内存固定和读写锁等。随后介绍了协程管理器和运行器的函数库

实现，使用协程管理器控制数据结构的创建，插入和删除，并通过运行器执行协程的工作流程。前四节的内容可以单独作为一个函数库向应用程序提供进程内的协程支持。第五节中涉及对操作系统内核的修改，需要在内核中分配物理页面，进行地址映射，创建进程到内核的共享内存作为优先级位图，以分享协程的优先级信息，供操作系统内核作为调度的依据，统一整个系统的协程调度。

第 5 章 性能测试

基于第三、四章的设计与实现，本文提出的调度模型没有为协程引入上下文，栈等重量级的系统资源，在把协程升级为系统级的基本调度单位的同时，尽力保留了协程的低开销特征，实现了一种轻量级的调度单位与相应的调度框架。但对比于现有主流的进程内局部调度的协程，还是增加了一些管理上的额外开销。为了量化它们对于系统的具体影响，本文根据操作系统进行任务调度时常见的三个场景设计实验，验证了本调度框架的协程对比于系统的线程依然能保持协程的高性能特性。

5.1 实验内容与环境

针对本文实现的协程库，设计了实验以验证软件质量。实验设置了三个不同的场景进行性能对比测试，在每一种场景中测试程序的总体执行时间随着并发量的变化关系。

第一个实验场景为并发编程的数据竞争。协程/线程互斥读写堆上的全局变量，测试并发环境下存在剧烈的数据竞争与调度单位的切换时，系统性能的稳定性，即总执行时间是是剧烈增加还是平稳上升。

第二个实验场景为读写内核缓冲区。协程/线程通过管道读写内核缓冲区，测试协程和线程频繁往返用户态与内核的切换与数据传递性能。

第三个实验场景为优先级优化系统性能。协程/线程两两之间多次传递数据，使得调度单位长时间存在于系统中，通过设置赋予可以理解执行完成并释放的协程更高的优先级，使得系统中的协程数量快速下降，优化系统的负载，并设置不开启优先级优化的对照组进行性能对比。

本实验在 qume[50] 模拟的 RISC-V 平台上完成。QEMU（Quick Emulator）是一个免费的开源硬件虚拟化软件，允许使用者在主机操作系统上运行一个或多个操作系统。具体来说，QEMU 是一种 type-2 hypervisors，它可以在多个主机操作系统上运行，包括 Linux、macOS、Windows 和其他类 UNIX 系统，它其他类型 type-2 hypervisors 的不同之处在于它可以模拟各种体系结构，包括 x86、ARM、SPARC、PowerPC 等，使其成为一个通用的工具。QEMU 模拟 guest OS 的硬件，并为运行应用程序提供虚拟化环境。

Hypervisor 是一种系统软件，它充当计算机硬件和虚拟机之间的中介，负责有效地分配和利用由各个虚拟机使用的硬件资源，这些虚拟机在物理主机上单独工作，因此，Hypervisor 也称为虚拟机管理器。可以本地安装并直接在物理主机上

运行的 Hypervisor 称为 Type 1 Hypervisor。Type 2 hypervisor 不能直接安装在裸机系统或物理主机上，它需要首先安装或可用的操作系统，以便部署并通过 OS 层间接访问 CPU、内存、网络、物理存储。

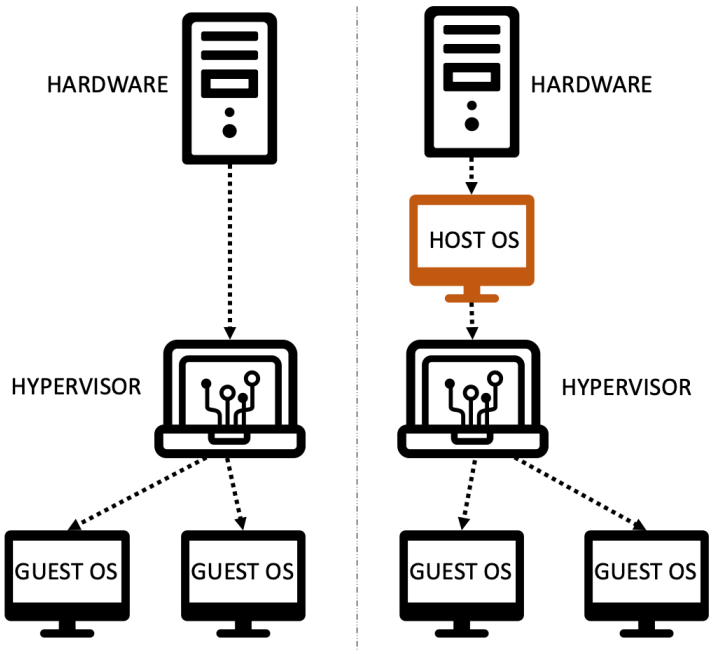

图 5.1 Hypervisor 模型

Fig. 5.1 Hypervisor Model

QEMU 的优点在于其实纯软件实现的虚拟化模拟器，几乎可以模拟任何硬件设备。本实验需要进行功能测试和调度单位之间的相对性能测试，而非测试某一实际场景下的绝对数值，所以 QEMU 是一个能够满足需求的合适的选择。

进行本实验的硬件参数为：

(1) CPU：2.6GHz 6 核 Intel Core i7 处理器；

(2) 内存：16GB 2666MHz DDR4；

进行本实验的软件参数为：

(1) HOST OS：Mac OS 12.6.3；

(2) Hypervisor：Qemu 7.0；

(3) Guest OS：rCore-Tutorial version 3.6；

5.2 性能测试一：读写全局变量

协程与线程的第一项性能对比测试在用户态完成，不使用系统调用接口。以下是性能对比测试第一项实验的实验步骤：

表 5.1 实验一关键节点数值

Tab. 5.1 Key Value of Test 1

并发量	耗时（MS）	
	线程	协程
200	22	49
800	122	124
1000	193	164
4000	2290	616

(1) 实验被创建为进程执行，每个进程内创建 N 个协程或线程执行任务，这 N 个协程或线程编号从 1 到 N，编号的值通过参数传入。在进程开始时获取一个起始时间，在进程退出前获取一个结束时间，以二者时间差作为任务的总执行时间；

(2) 每个协程或线程都会互斥读写一个全局变量，先执行读，读成功后执行写。全局变量初始值设置为 0，当协程或线程读到的全局变量的值等于自己的编号时，认为读取成功，然后执行写操作，即把全局变量的值加 1。当所有协程或线程创建完成之后，主线程把全局变量值加 1，使得编号为 1 的协程或线程可以开始读写；

(3) 当读取不成功时，协程使用异步机制主动让出，等待被唤醒；线程则调用系统调用进行线程切换。

(4) 实验中协程和线程的并发量从 200 递增到 4000，步长为 200；

测得每个并发量下的协程和线程的运行时间的递增趋势如图5.2所示。

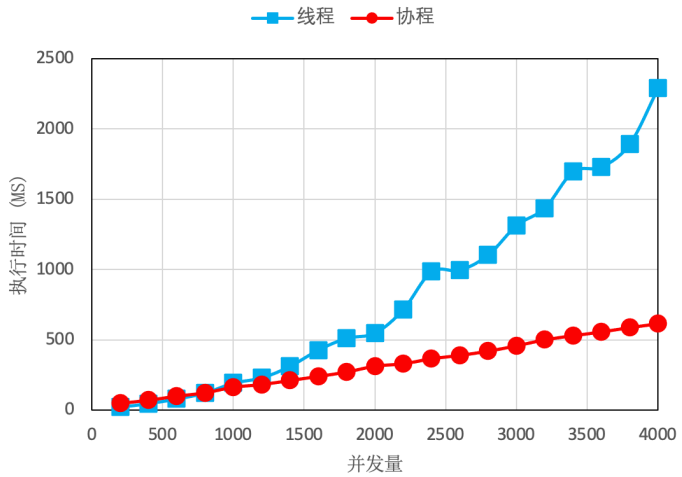

图 5.2 调度单位互斥读写全局变量的执行时间

Fig. 5.2 Execution Time of Units r&w Variables

表5.1是测得数据的一些关键节点。从中可以看出，当并发量较小时，协程的

总执行时间大于线程。随着并发量的增加，协程的执行总时间稳定增加，而线程的执行总时间快速增加并超过协程。从表中可以看到，在本实验中，当并发量为200 时，协程的性能弱于线程；当并发量大于等于 800 时，协程的性能开始超过线程；当达到测试的最大并发量 4000 时，协程的总执行时间约为线程的 26%。

5.3 性能测试二：读写内核缓冲区

协程与线程的第二项性能对比测试选择测试线程与协程通过内核传递、流通信息的能力，这需要使用系统调用频繁进入内核，协程与线程所遭遇的情况会比第一项性能对比测试更为复杂，也更能考验二者的能力并凸显它们的差异。

以下是性能对比测试第二项实验的实验步骤：

（1）实验被创建为进程执行，每个进程内创建 N 个协程或线程执行任务，这 N 个协程或线程编号从 1 到 N。编号的值通过参数传入。在进程开始时获取一个起始时间，在进程退出前获取一个结束时间，以二者时间差作为任务的总执行时间；

（2）创建 N + 1 个管道，编号从 1 到 N + 1，管道可以被读写。所使用的管道的工作方式为：只有当写端关闭时，对此管道的读操作才可以读取到数据然后完成读操作；当写端未关闭时，读操作会在内核引发线程切换；

（3）每个协程或线程执行管道的读写操作，具体来说，编号为 i（1<= i <= N）的协程或线程会从编号为 i 的管道读取数据，待写入端关闭，读取成功后，向编号为 i + 1 的管道中写入数据，然后关闭编号为 i 的管道的写端，使得编号为 i +1 的协程或线程可以完成读操作。相当于使用管道把所有协程或线程串联起来，对同一个管道的读写操作分别位于编号相邻的两个协程或线程中。

（4）所有的协程或线程创建完成之后，测试进程的主线程会直接向管道 1 写入数据然后关闭它的写端，此后随着 task 的调度和执行，数据就会被编号为 1 的 task 从编号为 1 的管道中读出，然后写入管道 2，编号为 2 的 task 再从管道 2 读取数据，然后写入管道 3，以此类推；

（5）每次实验读取与写入的数据量相同，测试了 1B，256B，4096B 这三个数据量。

（6）实验中协程和线程的并发量从 200 递增到 4000，步长为 200；

测得每个并发量下的协程和线程的运行时间的递增趋势如图5.3，5.4，5.5所示。

从图中可以看出，当并发量较小时，协程与线程的总执行时间的差距较小。随着并发量的增加，线程的执行总时间快速增加并超过协程。从下表中可以看到，在本实验中，当并发量为 200 时，协程的性能弱于线程；当并发量大于等于 400 时，

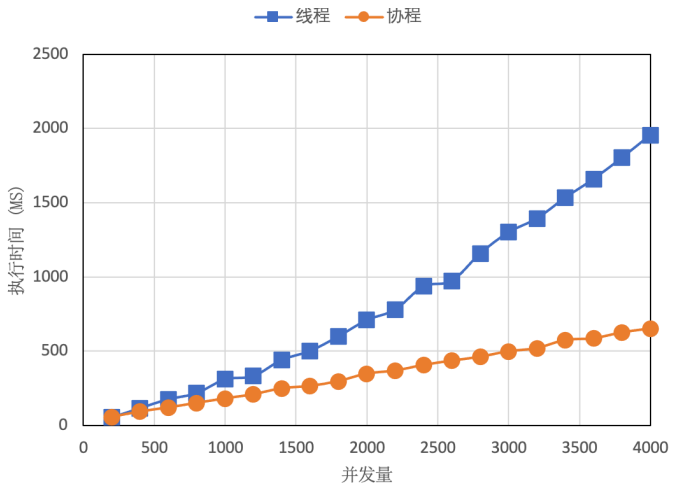

图 5.3 调度单位读写内核缓冲区的执行时间（数据量：1B）

Fig. 5.3 Execution Time of Unit r&w Kernel Buffer (volume: 1B)

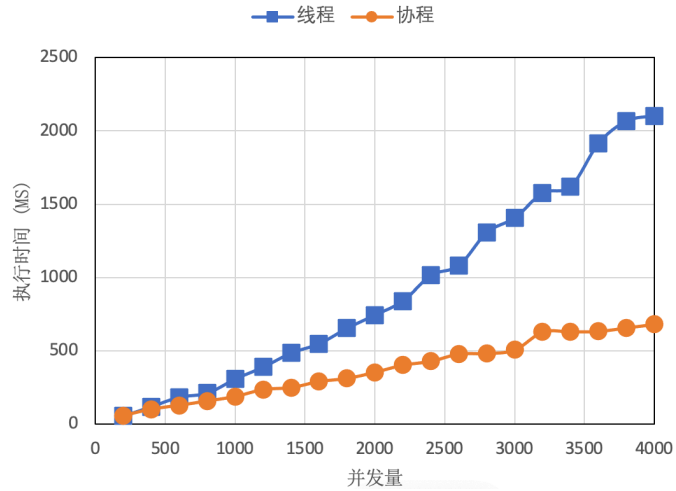

图 5.4 调度单位读写内核缓冲区的执行时间（数据量：256B）

Fig. 5.4 Execution Time of Unit r&w Kernel Buffer (volume: 256B)

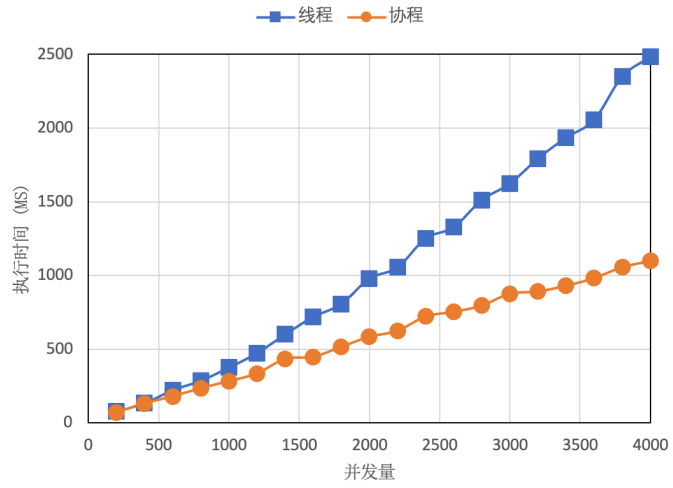

图 5.5 调度单位读写内核缓冲区的执行时间（数据量：4096B）

Fig. 5.5 Execution Time of Unit r&w Kernel Buffer (volume: 4096B)

协程的性能开始超过线程；当达到测试的最大并发量 4000 时，协程的总执行时间约为线程的 32% - 44%。

5.4 性能测试三：使用优先级优化系统负载

并发编程中常涉及通过共享内存进行并发单位间的通信，也就是指多个线程或协程在执行并发任务时常需要进行不通过内核的通信。基于此场景，在协程与线程的第三项性能对比测试中借鉴 TCP 的三次握手的思想测试线程间或协程间的通信能力，它们需要访问存储在堆上的数据结构进行双向通信。需要注意的是，本实验与第二项性能对比测试有着显著差异，本实验是协程间或线程间两两成组，组内通信三次，进程内同时运行多个组；而第二项性能对比测试是所有协程或线程组成一个组传递数据。

与之前的实验不同，本实验中的每个调度单位需要收发多次数据，系统中会长时间存在大量的调度单位，通过赋予可以立即执行并可以马上结束的协程更高的优先级，使得系统内的协程数量快速下降，减轻了系统的负载压力从而提升系统的性能。具体来说，每当协程发送完一条信息之后，会进入等待状态。当对方收到信息并返回时，协程会被异步唤醒。系统会在协程进入等待状态之前，升级协程的优先级，即加 1，使其被异步唤醒之后，有更大的机会被优先执行。而发送的信息越多，剩余需要发送的信息就越少，更有希望尽快执行完成然后从系统中删除，从而使得系统负载快速下降。

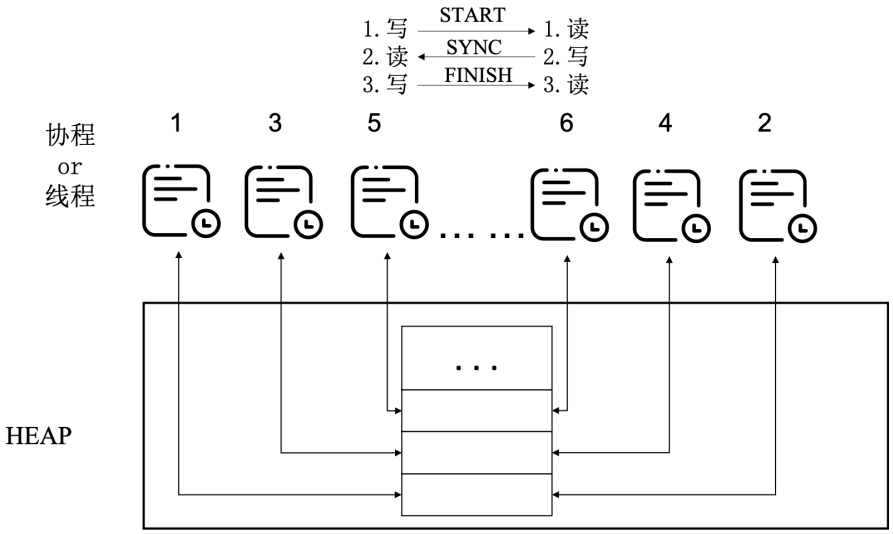

图 5.6 性能测试三：协程或线程两两传递数据

Fig. 5.6 Test 3: Coroutines or Threads Transfer Data in Pairs

以下是性能对比测试第三项实验的实验步骤：

(1) 实验被创建为进程执行，每个进程内创建 N（偶数）个协程或线程执行任务，这 N 个协程或线程编号从 1 到 N。编号的值通过参数传入。在进程开始时获取一个起始时间，在进程退出前获取一个结束时间，以二者时间差作为任务的总执行时间；

(2) 相邻的奇数编号和偶数编号的协程和线程组成一组相互通信（如 1，2 为一组，9，10 为一组等），即通过堆上管道执行 3 次读写操作。每组首先由奇数编号的协程或线程通过管道向对方写入一条消息 START，然后进入等待状态；接着偶数编号的协程或线程读取到 “START” 信息，并通过管道向对方回复"SYNC" 信息，随后进入等待状态准备接收最后的信息；最后由奇数编号的协程或线程接收到 SYNC 信息之后，向对方发送 FIN 信息，待对方接收之后，通信完成。

(3) 创建 N / 2 个管道，编号从 1 到 N / 2，管道可以被双向读写，但同一时间仅运行一端写另一端读。待管道被读取且信息被清空之后可以重置读写两端。

(4) 本实验中不存在先把所有的协程或线程创建完成之后在启动执行，而是创建完即执行。但是这不会影响实验数据的精度，因为测试的时间区间是从进程开始到进程结束前一刻。

(5) 实验中协程和线程的并发量从 200 递增到 4000，步长为 200；

测得每个并发量下的协程和线程的运行时间的递增趋势如图5.7所示。

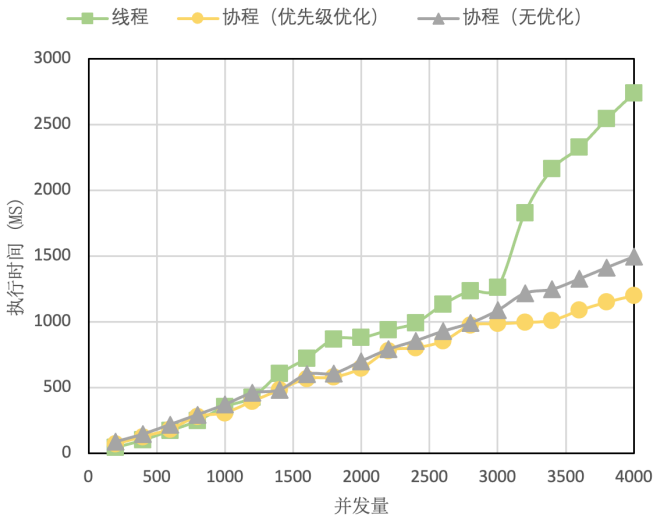

图 5.7 调度单位两两传递数据的执行时间

Fig. 5.7 Execution Time of Units to Transfer Data

5.5 实验结果与分析

从实验中可以看出，协程和线程虽然也是调度单元，但是在不同的并发量下，它们各有特点。当并发较小时，支持协程运行的基本开销，包括协程调度器和唤醒机制，影响显着，导致此时协程的性能接近于线程。然而，正如实验数据所示，协程的使用开销随着并发度的增加呈线性增加，这意味着并发度的增加对协程的平均执行时间和平均切换时间没有显着影响。这是协程区别于线程的一个特点，也是协程对于并发编程的意义。相比之下，线程的上下文切换成本随着并发量的增加而增加，这就导致协程的性能在并发量增加时逐渐超过线程，两者的差距越来越大。

协程和线程是编程世界中的两个重要概念。它们常用于多线程编程以实现并发和并行。尽管它们服务于相同的目的，但协程和线程在设计原理与底层实现之间存在一些显着差异，这些差异成为了上述实验结果的潜在原因。

(1) 上下文切换与内存占用

协程和线程之间性能差异的主要原因是上下文切换。上下文切换涉及保存当前线程或协程的状态并恢复另一个线程或协程的状态。此操作可能会花费一些时间，尤其是在线程或协程占用大量资源的情况下。

协程通常都会表现出比线程更快的运行特征，这就是因为它们不需要完整的上下文切换。相反，他们可以简单地保存当前状态并切换到下一个协程，该协程也运行在相同的内存空间中。而且，协程的内存占用比线程小。由于同一进程中的协程共享相同的堆内存空间并基于状态机实现（本文主要讨论无栈协程），因此它们不需要太多的上下文切换开销。上下文切换是调度单位涉及保存和恢复线程状态以便以后可以恢复的必要操作。

另一方面，线程具有更大的内存占用，因为每个线程都需要自己的内存堆栈，内存堆栈需要占据实际的内存空间，所以线程的上下文开销无法规避。

(2) 操作系统开销

线程依靠操作系统来管理它们的执行。这意味着操作系统必须管理线程的调度、分配内存和其他资源。这种开销会降低程序的性能，尤其是在并发量不断增加的情况下负面效果会更加明显。从实验结果中可以清晰地看出随着并发量的增加，线程的总执行时间呈现出一条下凸曲线的变化趋势，这代表随着 X（并发量）的增加，Y（总执行时间）的变化速度越来越快，类似于加速度增加的加速运动。

另一方面，协程的创建与释放可以在应用程序之内独立完成，使用的是进程的堆空间，省去了创建和回收时陷入内核的开销。在实验中体现为：协程的总执行时间的变化趋势近似于一条直线。这代表这代表随着 X（并发量）的增加，Y（总执行时间）的变化速度基本不变，类似于加速度恒定的加速运动。

5.6 本章小结

本章通过三个实验验证了在引入位图与内核管理的情况下，协程升级为系统级的调度单位依然可以保持低开销高性能的特征。三个性能对比实验分别对应操作系统进行任务调度时常见的三个场景：互斥读取堆上的同一个数据，导致大量的调度单位会在用户态争夺锁并频繁等待、切换；通过管道在调度单位间传递数据，测试调度单位读写内核缓冲区的性能；通过堆/共享内存在调度单位之间两两传递数据，每一对调度单位访问同一个数据，不同对的调度单位访问不同的数据，每一对调度单位之间往返传递 3 次数据，使得系统中会长时间存在大量的调度单位，通过赋予可以立即执行并可以马上结束的协程更高的优先级，使得系统内的协程数量快速下降，减轻了系统的负载压力从而提高系统的运行性能，验证了协程使用优先级调度带来的收益。最后讨论了实验数据和结果，肯定了协程在高并发环境中的性能及良好的开销控制能力。

第 6 章 总结与展望

协程作为轻量级的调度单位常用作并发编程的有效工具使用，但是局限于进程内的特点限制了协程作为调度单位的应用范围。本文在保持协程低开销特征的前提下，提出了一个把协程升级为系统级基本调度单位的调度模型。模型涉及两个部分的内容：进程内的协程调度模块和操作系统内核改造。前者可以作为独立的模块向进程提供协程支持，但此时的协程调度只是各个进程内的独立行为。通过本文提出的改造方案，可以在进程及操作系统内核设置优先级位图，并使用共享内存映射，使得操作系统内核能够通过优先级位图感知进程内协程的存在，从而根据优先级考量整个系统的协程调度策略，使得整个系统内，对于执行时间敏感的协程可以被优先调度，而一些对于执行时间不敏感的协程则可以设置较低优先级，减少与高优先级任务的争抢，使处理器可以更灵活、更科学地被分配。本文在开源操作系统 rCore 上完整实现了此调度模型，并在互斥争夺锁、读写内核缓冲区和优先级优化的不同的场景中测试了协程与线程的性能差距。结果显示，在高并发环境下，协程的性能明显优于线程。

近年来越来越多的研究人员开始关注协程，但是目前对于协程还有若干课题尚待突破：（1）协程的并行机制与并行性能。（2）调试。与传统代码调试顺序发生不同，协程有多个执行路径，这使得调试问题变得困难。错误还可能导致带故障的堆栈跟踪，从而难以遵循执行路径。（3）开发人员经常使用协程来实现异步操作（这也是本文的一个主要驱动），尽管这种方法存在隐藏的缺陷。如果时间上前后执行的两个在相同堆栈上执行的协程逻辑存在任何意外的设计缺陷，可能会导致死锁。如何建立起一套协程的并发原则避免协程的死锁是一个尚待突破的复杂问题，目前仅能依靠开发者的经验。

参考文献

[1] MOHSIN M. 10 google search statistics in 2023[M]. Ottawa: Oberlo, 2023.

[2] 中国互联网信息中心. 中国互联网络发展状况统计报告[J]. China: CNNIC, 2017, 115 (17): 1-1.

[3] 丁燎原. 基于协程的 web 服务器原型的设计与实现[D]. 大连: 大连理工大学, 2017.

[4] LIU T, CURTSINGER C, BERGER E D. Dthreads: efficient deterministic multithreading[C]// Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. New York, NY, USA, 2011: 327-336.

[5] DIAZ J, MUNOZ-CARO C, NINO A. A survey of parallel programming models and tools in the multi and many-core era[J]. IEEE Transactions on parallel and distributed systems, 2012, 23(8): 1369-1386.

[6] BELSON B, HOLDSWORTH J, XIANG W, et al. A survey of asynchronous programming using coroutines in the internet of things and embedded systems[J]. ACM Transactions on Embedded Computing Systems (TECS), 2019, 18(3): 1-21.

[7] KRAGL B, QADEER S, HENZINGER T A. Synchronizing the asynchronous[C]//29th International Conference on Concurrency Theory (CONCUR 2018). Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018: 21-21.

[8] PRABHAKAR R, KUMAR R. Concurrent programming with go[R]. Los Angeles: Citeseer, 2011.

[9] MOURA A L D, IERUSALIMSCHY R. Revisiting coroutines[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2009, 31(2): 31-31.

[10] BELSON B, XIANG W, HOLDSWORTH J, et al. C++20 coroutines on microcontrollers—what we learned[J]. IEEE Embedded Systems Letters, 2021, 13(1): 9-12.

[11] WEBER D, FISCHER J. Process-based simulation with stackless coroutines[C]//Proceedings of the 12th System Analysis and Modelling Conference. New York, NY, USA, 2020: 84-93.

[12] HE Y, LU J, WANG T. Corobase: Coroutine-oriented main-memory database engine[J]. Proc. VLDB Endow., 2020, 14(3): 431–444.

[13] ZHU T, WANG D, HU H, et al. Interactive transaction processing for in-memory database system[C]//International Conference on Database Systems for Advanced Applications. Gold Coast, QLD, Australia: Springer, 2018: 228-246.

[14] KALIA A, KAMINSKY M, ANDERSEN D G. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs.[C]//OSDI. Savannah, GA, USA, 2016: 185-201.

[15] GAO P X, NARAYAN A, KARANDIKAR S, et al. Network requirements for resource disaggregation.[C]//OSDI. Savannah, GA, USA, 2016: 249-264.

[16] NELSON J, MYERS B, HUNTER A H, et al. Crunching large graphs with commodity processors[C]//3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11). Berkeley, CA, USA, 2011: 10-10.

[17] KNOCHE H. Improving batch performance when migrating to microservices with chunking and coroutines[J]. Softwaretechnik-Trends, 2019, 39(4): 20-22.

[18] NARAYANAN V, BALASUBRAMANIAN A, JACOBSEN C, et al. Lxds: Towards isolation of kernel subsystems[C]//2019 USENIX Annual Technical Conference (USENIX ATC 19). Renton, WA, USA, 2019: 269-284.

[19] LIN F X, LIU X. Memif: Towards programming heterogeneous memory asynchronously[J]. ACM SIGPLAN Notices, 2016, 51(4): 369-383.

[20] LEE G, SHIN S, SONG W, et al. Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds[C]//2019 USENIX Annual Technical Conference (USENIX ATC 19). Renton, WA, USA, 2019: 603-616.

[21] LIAO X, LU Y, XU E, et al. Write dependency disentanglement with horae[C]//OSDI. online, 2020: 549-565.

[22] LI D, ZHANG N, DONG M, et al. Pm-aio: An effective asynchronous i/o system for persistent memory[J]. IEEE Transactions on Emerging Topics in Computing, 2021, 10(3): 1558-1574.

[23] MAGOUTIS K. Design and implementation of a direct access file system (dafs) kernel server for freebsd[C]//BSDCon 2002 (BSDCon 2002). San Francisco, CA, USA, 2002: 65-76.

[24] XU Z, HUANG J. Detecting 10,000 cells in one second[C]//International conference on medical image computing and computer-assisted intervention. Athens, Greece, 2016: 676-684.

[25] VENKATASUBRAMANIAN S, VUDUC R W. Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems[C]//Proceedings of the 23rd international conference on Supercomputing. New York, NY, USA, 2009: 244-255.

[26] BAGHERZADEH M, KAHANI N, BEZEMER C P, et al. Analyzing a decade of linux system calls[J]. Empirical Software Engineering, 2018, 23(18): 1519-1551.

[27] SOARES L, STUMM M. Flexsc: Flexible system call scheduling with exception-less system calls.[C]//Osdi: Vol. 10. Vancouver, BC, Canada, 2010: 33-46.

[28] KUZNETSOV D, MORRISON A. Privbox: Faster system calls through sandboxed privileged execution[C]//2022 USENIX Annual Technical Conference (USENIX ATC 22). Carlsbad, CA, USA, 2022: 233-247.

[29] TCHAKOUNTÉ F, DAYANG P. System calls analysis of malwares on android[J]. International Journal of Science and Technology, 2013, 2(9): 669-674.

[30] 蒋晓峰. 面向开源程序的特征码免杀与主动防御突破研究[D]. 上海: 上海交通大学, 2011.

[31] APPEL A W. Compiling with continuations[M]. New Jersey: Princeton University, 2007.

[32] ROMPF T, MAIER I, ODERSKY M. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform[C]//Proceedings of the 14th ACM SIGPLAN international conference on Functional programming. New York, NY, USA, 2009: 317-328.

[33] FARVARDIN K, REPPY J. From folklore to fact: comparing implementations of stacks and continuations[C]//Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, 2020: 75-90.

[34] PROKOPEC A, LIU F. Theory and practice of coroutines with snapshots[C]//32nd European Conference on Object-Oriented Programming (ECOOP 2018). Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018: 3-3.

[35] KOMOLOV S, ASKARBEKULY N, MAZZARA M. An empirical study of multi-threading paradigms reactive programming vs continuation-passing style[C]//2020 the 3rd International Conference on Computing and Big Data. New York, NY, USA, 2020: 37-41.

[36] PETRICEK T, ORCHARD D, MYCROFT A. Coeffects: a calculus of context-dependent computation[J]. ACM SIGPLAN Notices, 2014, 49(9): 123-135.

[37] JAMES R P, SABRY A. Yield: Mainstream delimited continuations[C]//First International Workshop on the Theory and Practice of Delimited Continuations (TPDC 2011). Novi Sad, Serbia, 2011: 96-96.

[38] CONWAY M E. Design of a separable transition-diagram compiler[J]. Communications of the ACM, 1963, 6(7): 396-408.

[39] WAERN L. Coroutines for simics device modeling language[D]. Sweden: Uppsala University, 2021.

[40] DE NICOLA R, DUONG T, INVERSO O, et al. Aerlang: empowering erlang with attribute-based communication[C]//Coordination Models and Languages: 19th IFIP WG 6.1 International Conference. Neuchâtel, Switzerland, 2017: 21-39.

[41] RAY B, POSNETT D, FILKOV V, et al. A large scale study of programming languages and code quality in github[C]//Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. New York, NY, USA, 2014: 155-165.

[42] OKUR S, HARTVELD D L, DIG D, et al. A study and toolkit for asynchronous programming in c#[C]//Proceedings of the 36th International Conference on Software Engineering. New York, NY, USA, 2014: 1117-1127.

[43] LIM M. Directly and indirectly synchronous communication mechanisms for client-server systems using event-based asynchronous communication framework[J]. IEEE access, 2019, 7(2): 81969-81982.

[44] XIE C, CHEN R, GUAN H, et al. Sync or async: Time to fuse for distributed graph-parallel computation[J]. ACM SIGPLAN Notices, 2015, 50(8): 194-204.

[45] IKEBUCHI M, ERBSEN A, CHLIPALA A. Certifying derivation of state machines from coroutines[J]. Proceedings of the ACM on Programming Languages, 2022, 6(POPL): 1-31.

[46] WANG X, HUANG H. Sgpm: A coroutine framework for transaction processing[J]. Parallel Computing, 2022, 114(22): 102980-102980.

[47] GUALANDI H M, IERUSALIMSCHY R. Pallene: A companion language for lua[J]. Science of Computer Programming, 2020, 189(20): 102393-102393.

[48] ELIZAROV R, BELYAEV M, AKHIN M, et al. Kotlin coroutines: design and implementation [C]//Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. New York, NY, USA, 2021: 68-84.

[49] KOCHUROV M, CARROLL C, WIECKI T, et al. Pymc4: Exploiting coroutines for implementing a probabilistic programming framework[C]//Program Transformations for ML Workshop at NeurIPS 2019. NeurIPS 2019, 2019: 1-4.

[50] BALDONI R, COPPA E, D’ ELIA D C, et al. A survey of symbolic execution techniques[J]. ACM Computing Surveys (CSUR), 2018, 51(3): 1-39.

研究生期间的科研情况

在校期间作者参加科研项目与学术活动

实验室项目：开源操作系统 rCore 的异步化工作

致 谢

由衷感谢孙卫真与向勇二位导师，他们在学术、交流合作与解决问题方面给我的指导让我一生受益。

首先，孙老师和向老师是我学术道路上的引路人，他们的悉心指导和无私帮助带领我在研究领域中不断成长。二位尊师教会了我如何进行科学研究、如何解决问题和如何做出正确的决策。他们耐心的指导和不懈的鼓励让我在研究中探索出了更加深入的方向。他们的严谨态度、专业知识和敬业精神让我深受启迪，我将一直珍视他们的教诲和指导，将这些经验运用到我的未来的挑战中。

向老师严谨治学，研究领域广泛且具有深入的见解，对我的研究方向给予了充分的支持，为我提供了严谨的研究方法，让我在学术道路上不断成长。在论文撰写过程中，向老师认真负责，多次深夜阅读我的论文为我提供修改建议。向老师的指导让我有幸一窥学术的奥秘。

孙老师对我有知遇之恩。我带着较少的研究经历来攻读硕士学位，但孙老师愿意收我为徒并传授我许多做学问的经验，在过去三年中的每一个关键时间节点，孙老师都会提前告知我应该做出的准备。在疫情期间也时常关心我的学习和生活状态，鼓励我向前看。可以说，我能顺利找到导师开启研究生生涯是因为孙老师的慷慨，我能顺利毕业也是因为孙老师和向老师的指导与关心。

此外，我要感谢母校对我的培养。我非常幸运能够在首师接受高质量的教育，这为我往后更长远的发展打下了坚实的基础。在学校，我遇到了很多优秀的老师和同学，他们的学习氛围和学术氛围让我受益匪浅。我还要感谢学校为我提供的各种讲座，些经历让我更加深入地了解行业的前沿进展。我深信，首师的培养使我能够为未来的学术研究和职业生涯打下坚实的基础。