
北京理工大学

BEIJING INSTITUTE OF TECHNOLOGY



支持优先级的 Rust 协程调度

队伍名称	你的太阳落山了
赛 题	poj146
指导老师	陆慧梅
项目成员	胡嘉晨、肖嘉骏

二〇二五年 八月

目录

目录.....	II
1 项目简述.....	1
1.1 项目背景	1
1.2 项目创新点.....	1
1.3 项目开发历程	1
2 项目整体设计	2
2.1 整体架构图.....	2
2.2 内核态调度设计.....	3
2.3 用户态调度设计.....	4
2.4 优先级机制设计.....	4
3 详细设计与实现.....	5
3.1 协程模块设计与实现	6
3.1.1 协程模块的设计	6
3.1.2 协程模块的实现	6
3.2 共享区与用户态线程调度设计与实现	9
3.2.1 共享区与用户态线程调度的设计	9
3.2.2 共享区与用户态线程调度的实现	11
4 项目测试.....	19
4.1 协程模块测试	19
4.1.1 协程模块测试方法	20
4.1.2 协程模块测试结果.....	22
4.2 用户态线程调度测试	26
4.2.1 用户态线程调度测试方法	26
4.2.2 用户态线程调度测试结果	26
4.3 综合测试	27
4.3.1 综合测试方法	27

4.3.2	用户态线程调度测试结果	28
5	开发过程中的困难和解决方法	30
5.1	协程调度器开发中的死锁问题	30
5.2	共享区报错 LoadPageError 问题.....	31
5.3	用户态调度报错 Unsupported syscall 问题	31
5.4	vs code 的 rust-analyzer 崩溃的问题.....	32
6	参考资料.....	32

1 项目简述

1.1 项目背景

在现有的操作系统设计中，内核级线程调度虽能有效管理进程和线程资源，但对用户态异步编程模型的支持存在显著不足。随着异步编程范式的普及，传统线程模型因上下文切换开销大、资源占用高等问题，难以高效支持大规模并发任务。

主流异步方案，如 Tokio 运行时，需在用户态实现协程调度器，但这类实现与内核调度器缺乏整合，导致如下问题：

1. 用户态协程无法参与系统级优先级调度，高优先级任务可能因为所属的线程原因被低优先级任务抢占，而造成优先级高的任务无法及时运行完成。
2. 协程与线程、进程的优先级分割，造成优先级出现冲突现象，即各个优先级任务都无法得到及时的响应，造成任务出现饿死的情况。

1.2 项目创新点

本项目基于 rCore 操作系统，创新性地设计了一套统一优先级调度架构，基于带有优先级的协程调度机制，借助内核用户共享区实现的用户态线程调度，实现进程、线程、协程的三级优先级统一管理，提升高并发场景下任务响应效率与资源利用率，同时减少低并发场景下任务切换的开销。

1.3 项目开发历程

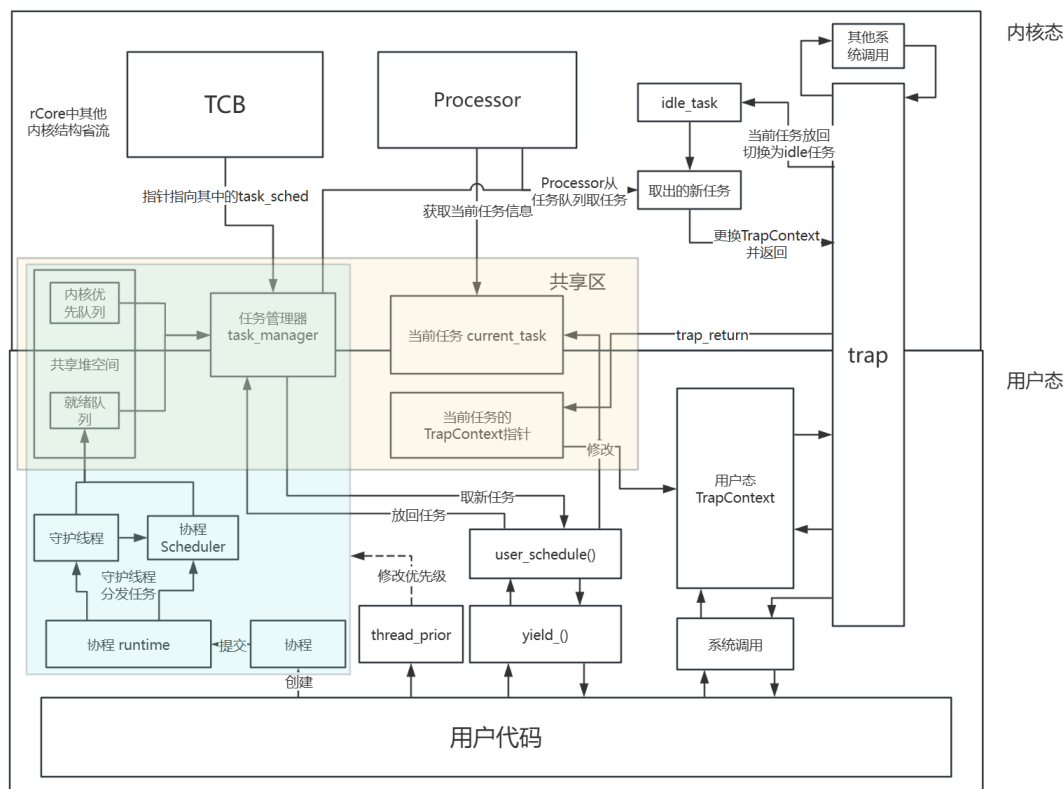
时间	完成内容
第一阶段	
6 月 2 日至 6 月 9 日	初步实现用户态协程机制
6 月 9 日至 6 月 16 日	改进 rCore 线程调度策略，修改为基于大根堆的优先级轮转
6 月 16 日至 6 月 23 日	新增系统调用：修改线程优先级

6 月 23 日至 6 月 30 日	编写测试代码，开始第一阶段测试
第二阶段	
7 月 14 日至 7 月 20 日	优化协程等待队列逻辑，增加守护线程，优化任务分配逻辑
7 月 20 日至 7 月 27 日	优化协程调度器执行逻辑，实现 Waker 基本功能
7 月 27 日至 8 月 3 日	实现内核用户共享区，并简单测试
8 月 3 日至 8 月 10 日	将调度器移入共享区，修改相关调度代码。但是遇到很多问题
8 月 10 日至 8 月 17 日	解决上周问题，撰写报告，整理总结

2 项目整体设计

本节介绍项目的整体设计思路：以优先级为核心，内核态调度进程和线程，用户态调度线程和协程。

2.1 整体架构图



2.2 内核态调度设计

在此基础上，我们为线程加入了优先级，并将线程按照大根堆组织起来，以便于快速找到优先级最高的线程。当 `processor` 取线程时，队首的线程即为优先级最高的线程，并在取线程后和添加新线程时重新维护大根堆的顺序。

内核态的 TCB 被重新组织了，提取出其中涉及到线程调度的部分，并放入 TaskSched 结构体中，以便于在用户态进行调度。

2.3 用户态调度设计

用户态的调度分为两部分：线程调度和协程调度。

线程的调度与内核相似，只不过在切换的过程中不用再考虑陷入等与内核相关的事情。为了实现内核态与用户态的线程切换相统一，我们参考了 `vdso` 技术的思路，设置了内核态和用户态的共享区。我们将就绪线程队列放在了共享区中，使得用户态可以直接访问就绪队列，与内核态的线程队列和调度同步。

在进行线程调时，沿用 `TrapContext` 保存线程上下文，这样可以实现在不切换地址空间的前提下，保存完整的线程上下文，同时不会和内核态的调度出现冲突。此外，在共享区维护一个指针指向了当前任务的 `TrapContext`，在用调度时，通过修改该指针，可以实现线程的切换。

协程调度模块中，我们先基于 Rust 的 `Future trait` 实现了基本的协程定义，包括 `Future` 函数、优先级、协程编号 `CID` 等内容。然后实现了协程调度器 `scheduler`，用来调度协程并通过大根堆维护就绪协程队列。之后，实现了协程运行时 `runtime`，用来维护各个调度器线程，同时包括了公共了就绪队列和一个守护线程。

在调度协程时，我们参考 `tokio` 的调度思路，为每个进程通过 `lazy_static` 的方式创建一个协程运行时，每个运行时中维护多个调度器线程，每个调度器维护一个协程队列，选择最高优先级的协程进行执行。

在使用了协程后，运行进程中会创建出若干各协程调度器线程和一个守护线程。当前进程中所有协程都提交到同一个运行时中，然后分发到调度器线程中，不同调度器线程中的协程通过线程调度进行调度。

2.4 优先级机制设计

优先级机制是本项目的核心。我们在协程结构体的定义中，加入了优先级这一字段，并修改了线程的定义，加入了优先级字段。

在用户提交协程时，需要设置协程的优先级，该优先级会被记录在协程的结构体中，并在协程放入调度器的任务队列时修改该调度器线程的优先级。这里我们规定优先级数值越大，优先级越高。

在协程调度中，优先级是静态的，在协程创建之后不再改变。这是因为我们考虑到协程在运行的过程中，处于运行的时间是很短的，更多的时间是处于 Pending 状态，此时各协程基本上都能够保证得到及时的处理，并且在开发与测试中没有观察到出现饿死的情况。因此，我们认为协程调度中采用静态优先级能够保证调度策略的合理性，同时避免优先级改变带来的大根堆维护等开销。

在线程调度中，优先级是动态的，具体体现在两方面：

- 第一，线程优先级定义为该线程当前运行的协程中优先级的最大值，我们称之为“固有优先级”。每当当前线程在运行中出现协程调度时，都进行一次优先级更新，取协程队列中最大的优先级为当前线程的优先级。
- 第二，线程调度中考虑了线程等待时间，我们称之为“年龄”。无论是内核态调度还是用户态调度，当每次线程调度时，如果线程没有被调度到开始运行，则将该线程的“年龄”加一；如果线程被调度到开始运行，则将“年龄”重置。

在线程队列中，我们也使用大根堆进行排序。与协程调度器的大根堆不同的是，排序时依据“固有优先级”与“年龄”相加的和。这样的好处是可以避免某个线程因为“固有优先级”低而导致始终得不到调度，并且由于无法得到调度使得“固有优先级”无法改变，最终出现饿死的情况。

在进程调度中，由于进程的调度是由线程调度而间接实现的，因此进程中的优先级最终会反映到进程上，间接实现进程的优先级调度。

至此，我们的设计借由协程的优先级实现了基于优先级的协程、线程、进程的统一调度。

3 详细设计与实现

本节介绍项目的详细设计与主要实现细节，包括协程模块、共享区和用户态线程调度两部分。

3.1 协程模块设计与实现

3.1.1 协程模块的设计

协程模块的参考 Tokio 的设计思路，分为协程定义、协程运行时、协程调度器等部分。每个使用协程的进程都有一个唯一的运行时，在运行时中管理若干个协程调度器，以便在多核环境下提高运行效率，同时还保存一个公共任务队列，方便统一分配任务实现负载均衡。协程对象在创建时应该加入优先级，并在后续调度过程中按照优先级组织协程任务队列。协程调度器线程把协程队列的最高优先级确定为调度器线程的优先级。

在这里，协程在调度器中执行 `poll` 后如果返回的是 `Pending`，调度器则不对其进行处理，而是由协程在准备就绪后，由其 `Future` 的上下文中的 `Waker` 重新提交到运行时。这样的好处是：只调度能立即执行的任务，减少无意义的 `poll`，提高高优先级任务的响应速度，同时，在多处理器的情况下，能够降低调度器的锁竞争和队列操作开销。

协程模块总体运行流程为：

- 1) 用户通过 `async` 和 `await` 定义异步函数，并提交函数入口和优先级到运行时
- 2) 运行时创建协程对象，放到公共队列中
- 3) 运行时的守护线程从公共队列取出协程，分配到调度器线程的等待队列上，队列按照优先级排序
- 4) 调度器线程从等待队列的队首取出任务，调用协程的 `poll`，并维护其计数和最大优先级
- 5) 如果结果为 `Ready`，则更新计数和最大优先级，否则不做处理，不放回任务队列
- 6) 其他协程在 `await` 处被唤醒时，在 `wake` 中被提交到公共队列，等待守护线程分配
- 7) 重复步骤 3-6，直到协程执行完毕
- 8) 退出协程调度器，退出守护线程，退出运行时

3.1.2 协程模块的实现

在本项目中，我们自己定义了协程模块。其中的结构体包括：

- 协程类 **Coroutine**：基于 Rust 的 **Future trait** 实现，对每个协程分配 **CID**，在创建时传入优先级，支持 1-20 级优先级，其中 1 级为最低优先级，20 为最高优先级。

同时对协程实现比较的接口，方便在调度器中依据优先级进行协程的排序。

除比较之外，对外暴露接口为初始化函数 `new(future, priority, scheduler)`。
- 协程编号分配器 **CidAllocator**，用于分配线程的 **CID**，提供 `allocate()` 和 `deallocate()` 接口，实现原理与 rCore 的 **TidAllocator** 相同，只在协程内部使用，不在下表中描述。
- 协程调度器 **Scheduler**：基于 **Future** 和 **Wait** 机制实现，内部有一个协程就绪队列，使用大根堆实现对其调度的每个协程进行管理与排序。大根堆在每次队列变化时进行维护。

对外暴露接口为 `new()`、`run()`、`submit_coroutine(Coroutine)` 和 `quit()`，以及一些简单的 `get/set`。

其中，`run()` 为调度器的入口函数，调度器进入 `loop` 循环，取出协程任务后，执行改协程的 `poll` 方法，如果执行结果为 **Ready**，则更新当前的协程计数和最大优先级，并通过 `thread_prio()` 函数向上更新线程优先级。如果执行结果为 **Pending**，暂时不把它放回任务队列，等待协程的 **Future** 上下文的 **Waker** 调用 `wake()` 后再将其放回。

这样做的好处是：只调度能立即执行的任务，减少无意义的 `poll`，提高高优先级任务的响应速度，同时有多处理器的情况下，能够降低调度器的锁竞争和队列操作开销。
- 协程运行时 **CoroutineRuntime**：作为所有协程调度器线程的父线程，同时有一个守护线程，管理所有调度器正常运行。内部有一个协程公共等

待队列，收集所有提交的协程。

守护线程用于将所有的协程公平的分配到各个调度器中，实现负载均衡。这里采取的分配方式是调度器线程任务数量最少的优先分配。

后续，守护线程中可以加入定时器、信号处理等异步功能，目前没有实现。

对外提供接口用于提交协程、移除协程计数、等待所有协程完成、退出运行时。

- **协程事件 Event:** 实现协程同步。具体而言，Event 中维护一个 Waker 对象，协程将自己运行代码中在该 Event 处通过 `await` 等待，当其他协程调用改 Event 的 `set` 时，Waker 对象将通知该协程并继续运行。本质上，这种实现将 Event 也变为了一个协程，并通过调度器调度，在其他协程调用 `set` 时，Event 的 Waker 对象将通知该协程并在调度器中直接运行，之后 Future 机制实现了再次 `wake` 前面的协程，从而实现协程同步。对外暴露接口有创建事件、设置事件就绪、清除事件、等待事件

结构体	函数名	参数	返回值	作用
协程 Coroutine	<code>new</code>	<code>future</code> : Future 实现、 <code>prior</code> : 优先级、 <code>scheduler</code> : 协程调度器	<code>Self</code> : 协程对象	新建协程对象
	<code>new</code>	<code>id</code> : 调度器编号	<code>Self</code> : 协程调度器对象	新建协程调度器对象
协程调度器 Scheduler	<code>run</code>	<code>scheduler</code> : 运行的调度器	无	调度器入口函数，调度主循环

	submit_coroutine	self,	无	外界想调
		coroutine: 要提交的		度器提交
		协程		协程的接
				口
	submit_distribute_	future: Future 实	无	提交协程
	coroutine	现、即协程对象		到运行时
		priority: 优先级		的接口
协程运行时	remove_task	无	无	运行时计
Coroutine-				数减一
Runtime	wait_all_coroutines	无	无	等待所有
				协程完成
	quit_coroutine_runtime	无	无	退出运行
				时
协程事件	create_event	无	事件对象	创建事件
	set	self	无	设置就绪
	clear	self	无	清除事件
	wait	self	事件的	等待事件
Event				
			Future	

此外，我们还编写了简单的库函数：`coro_yield()`和`coro_sleep()`。通过实现 `Future trait`，在 `poll()` 函数中分别不做额外处理和检查时间，实现库函数。具体实现比较简单，这里不再赘述。

3.2 共享区与用户态线程调度设计与实现

3.2.1 共享区与用户态线程调度的设计

由于我们的协程都在用户态执行，同时协程运行时的正常运行中涉及到多个线程的切换，因此实现用户态线程切换能够减少线程陷入内核的次数，提高运行效率。在 `rCore` 中，线程被设计为内核级线程，用户态是无法参与线程调度的，因此我们需要想办法在用户态对内核级线程进行调度。

3.2.1.1 共享区设计

为了实现用户态线程调度，以及与内核态的线程切换相统一，我们参考了 VDSO 技术的设计思路，设置了内核态和用户态的共享区，在内核态与用户态共用一个线程调度器，从而在保证内核态调度不受影响的前提下，引入用户态调度的功能。

我们知道，在 RISC-V 64 架构中，启用 SV39 分页模式下，只有低 39 位是真正有意义的，高 25 位需要与第 38 位相同。在 rCore 的设计中，高 256G 的地址中页面是从高到低占用的，任务几乎不可能占用全部的 256G，较低地址位置一定是空闲的。

因此，我们选择在内核态和用户态地址空间高 256GB 的最低地址，即 $2^{64} - 256G + 1$ 处当作共享区的首地址，共享区大小为 18 个页面：

- 第一页：存放线程调度器，包括其中的线程队列、线程计数和用户调度标志位。
- 之后的 16 页：作为共享区堆空间，用于储存线程对象。
- 最后一页：用于储存当前线程的 Trap 上下文的引用。

3.2.1.2 用户态线程调度设计

首先，将线程调度涉及到的部分提取出来，组织成一个对象类 TaskSched，在此，我们称之为线程调度对象。此对象将在内核中初始化，并存储在共享区堆空间中。调度器的线程队列存储的即为 TaskSched 的引用。

为了方便保存线程调度的全部上下文，我们选择复用 rCore 中存储在用户地址空间的 Trap 上下文。这样做的目的有两个：

- 实现在不切合地址空间的条件下完成线程全部上下文的储存，并且有较为完善的结构体实现。
- 防止出现这种情况：在内核态由线程 A 切换到线程 B。B 上一次调度是由用户态完成的，其在用户态的 sepc 和 ra 等关键寄存器是任务切换函数中的地址。这时切换到 B，会在 Trap 返回时覆盖之前的寄存器，从而发生跳转地址错误、段错误等异常。使用 Trap 上下文后，可以借助 Trap 返回恢复寄存器，能够完全避免这种问题发生

在用户态调度过程中，需要防止被时钟中断打断，否则会出现寄存器保存或恢复不完整，致使出现异常。对此，我们在共享区数据中加上用户调度标志位，在开始用户态调度开始时标记为 `True`，在结束时标记为 `False`。当标记为 `True` 时，如果出现时钟中断进入内核，则直接返回而不切换任务。

线程队列我们也设计为大根堆，这样能够与协程保持一致，从而让优先级能够延伸到线程中。

线程优先级由两部分组成，分别是：

- “固有优先级”：定义为该线程当前运行的协程中优先级的最大值，反映当前线程总体的急迫性。当使用协程机制时，每次进行协程调度时都对其进行更新。具体位置为协程调度器的 `run()` 函数中进行任务队列的队首任务 `poll` 返回 `ready` 后和提交新协程到调度器内时。
- “年龄”：定义为线程在调度器中的等待次数，每次调度时如果没有被调度到，则“年龄”加一，反映线程的等待时间。如果线程被调度到开始运行，则将“年龄”重置。固有优先级的作用是防止线程被饿死

无论是用户态和内核态，在每次调度时，都会对当前线程进行排序，维护大根堆结构，排序依据的优先级是“固有优先级”与“年龄”相加之和得到的动态优先级。这样的好处是可以避免某个线程因为“固有优先级”低而导致始终得不到调度，并且由于无法得到调度使得“固有优先级”无法改变，最终出现饿死的情况。

3.2.2 共享区与用户态线程调度的实现

由于这部分的代码没有严格的按照面向对象的方式模块化编写，因此这部分不再提供接口说明，而是直接将部分源代码展示出来进行论述。

3.2.2.1 共享区实现

我们硬编码地址空间虚拟地址为 $2^{64} - 256G + 1$ 处当作共享区的首地址，共享区大小为 18 个页面，包括以下内容：

- **VdsoData**：存储在第一页的对象的数据结构，定义为：

```
1. pub struct VdsoData {
2.     pub block_sched: AtomicBool, // 阻塞内核抢占
```

```

3.     pub task_manager: TaskManager,
4.     pub current_task: [Option<Arc<TaskSched, LockedHeapAllocator>>; P
    ROCESSOR_NUM],
5. }

```

其中，`block_sched` 即为用户调度标记，`TaskManager` 为调度器对象，`current_task` 数组用来在内核中标记当前 CPU 运行的任务。在内核态和用户态，通过 `lazy_static` 的方式创建静态 `VDSO_DATA`，以进行共享区读写。

- **TaskManager:** 调度器对象，用来管理线程的就绪队列，定义为：

```

1. pub struct TaskManager {
2.     pub ready_heap: Vec<Arc<TaskSched, LockedHeapAllocator>, LockedHe
    apAllocator>, // 就绪任务堆, 大根堆
3. }

```

其中，`TaskSched` 为线程调度对象，`LockedHeapAllocator` 为共享区堆空间的分配器。对外提供接口 `fetch()` 和 `add()`，分别用于取出和存入线程任务，并在其中对线程就绪队列按照动态优先级进行排序，由于不能够使用库函数，我们自己进行了简单实现。

- **LockedHeapAllocator:** 共享区堆空间分配器，用于在堆空间动态分配空间，使得 `Arc` 不会引用共享区外的内容，从而防止出现 `LoadPageError` 等问题。定义为：

```

1. pub struct LockedHeapAllocator(&'static LockedHeap);

```

这里的 `LockedHeap` 为 `buddy_system_allocator` 库中的结构体，一般使用 2 的整数次方个页面进行分配。`LockedHeapAllocator` 向外提供接口为 `allocate()` 和 `deallocate()`，用于在共享区堆空间分配和释放空间。

此外，在内核态也需要做响应的适配，包括：

- 在地址空间建立时初始化共享区：

```

1. fn map_kernel_vdso(&mut self) {
2.     for i in 0..VDSO_PAGES {
3.         self.page_table.map(
4.             VirtAddr::from(KERNEL_VDSO_BASE + i * PAGE_SIZE).into(),

```

```

5.         VDSO_PAGE[i].ppn,
6.         PTEFlags::R | PTEFlags::W,
7.     );
8. }
9. let vpn: VirtPageNum = VirtAddr::from(KERNEL_VDSO_BASE).into();
10. let ppn = VDSO_PAGE[0].ppn;
11. info!(
12.     "VDSO area: vpn: {:?}, ppn: {:?}, PageNum: {:?}",
13.     vpn, ppn, VDSO_PAGES
14. );
15. }
16. fn map_user_vdso(&mut self) {
17.     for i in 0..VDSO_PAGES {
18.         self.page_table.map(
19.             VirtAddr::from(USER_VDSO_BASE + i * PAGE_SIZE).into(),
20.             VDSO_PAGE[i].ppn,
21.             PTEFlags::R | PTEFlags::W | PTEFlags::U,
22.         );
23.     }
24. }

```

并在 `new_kernel()`、`from_elf()`、`from_existed_user()` 中调用。其具体原理是通过 `frame_alloc_more()` 分配连续的 18 个页面，并将其物理地址和我们设计的虚拟地址映射起来，从而实现在用户态和内核态访问共享区的内容。

- 共享区堆的建立和初始化：

```

1. lazy_static! {
2.     static ref VDSO_HEAP: &'static LockedHeap = {
3.         let va: usize = KERNEL_VDSO_BASE + core::mem::size_of::<VdsoData>();
4.         let heap = LockedHeap::empty(); // 创建一个空的堆

```



```

5.         unsafe {
6.             // 写入到 va 地址上
7.             core::ptr::write(va as *mut LockedHeap, heap);
8.             &*(va as *const LockedHeap) // 返回指向堆的引用
9.         }
10.    };
11.    // 用于分配 Arc<TaskSched>所需的内存
12.    pub static ref VDSO_HEAP_ALLOCATOR: LockedHeapAllocator = {
13.        info!("[kernel] VDSO heap allocator initialized");
14.        let alloc = LockedHeapAllocator(*VDSO_HEAP);
15.        unsafe {
16.            alloc.0.lock().init(KERNEL_VDSO_BASE +
17.                                PAGE_SIZE * VDSO_DATA_PAGES,
18.                                VDSO_HEAP_PAGES * PAGE_SIZE);
19.        }
20.        alloc
21.    };
22. }

```

在这里，我们将共享区堆空间的首地址设定为共享区开始位置向后偏移调度器大小的位置，以及后面连续的 16 页。这样，我们在内核态初始化时建立了共享区堆空间和所需的分配器，实现了线程调度对象放在共享区，内核态和用户态都可以正常读写。

3.2.2.2 用户态线程调度实现

- 线程调度对象：在共享区中，调度器调度的内容为线程调度对象 TaskSched，其中记录着线程调度相关的信息。其定义为：

```

1. pub struct TaskSched {
2.     pub id: (usize, usize), // 任务 ID(同时是线程 id)
3.     pub can_user_sched: AtomicBool, // 用户是否可以调度
4.     pub inner: TicketLock<TaskSchedInner>,

```

```

5. }
6. pub struct TaskSchedInner {
7.     pub prio: usize, // 静态优先级
8.     pub dynamic: usize, // 老化机制, 用于防止饥饿
9.     pub task_cx: TaskContext,
10.    pub task_status: TaskStatus,
11. }

```

这里的 id, 定义为(PID, TID)用来区分进程和线程。Inner 通过我们自己定义的内部可变的线程安全的锁包裹, 类似于 rCore 的 UPIntrSafeCell 的作用。

can_user_sched 字段用来表示该任务是否可以有用户态调度, 这是因为会出现线程通过 read()等系统调用进入了内核态, 此时任务的 Trap 上下文保持的是系统调用之后的下一条指令的地址, 如果这时候进行用户态调度, 则会跳过系统调用直接进行后面的任务, 导致出错。

Inner 中, prior 是动态优先级的“固有优先级”的字段, dynamic 是动态优先级计算后的值, task_cx 和 task_status 分别是内核态线程调度所需的上下文和线程状态字段。

TaskSched 除获取内部可变引用的接口外, 还实现了获取、增加、重置动态优先级的接口:

```

1. pub fn get_dynamic_prio(&self) -> usize {
2.     let inner = self.inner_exclusive_access();
3.     inner.dynamic
4. }
5. #[inline]
6. pub fn add_dynamic_priority(&self, delta: usize) {
7.     let mut inner = self.inner_exclusive_access();
8.     inner.dynamic += delta;
9. }
10. #[inline]
11. pub fn clear_dynamic_priority(&self) {

```

```

12.     let mut inner = self.inner_exclusive_access();
13.     inner.dynamic = inner.prio;
14. }

```

在这里，`get_dynamic_prio()`用在实现比较的接口上，`add_dynamic_priority()`和`clear_dynamic_priority()`在 `TaskManager` 的 `fetch()`中用到：

```

1. pub fn fetch(&mut self) -> Option<Arc<TaskSched, LockedHeapAllocator>
    > {
2.     // 取出堆顶任务
3.     ...
4.     task.clear_dynamic_priority();
5.     // 老化
6.     for task in self.ready_heap.iter() {
7.         task.add_dynamic_priority(1);
8.     }
9.     // 下滤操作
10.    ...
11.    Some(task)
12. }

```

可以看到，在 `fetch()`中，将取出的任务的动态优先级重置，并对所有任务的动态优先级加一，实现老化机制，防止饿死发生。这里，动态优先级中“年龄”部分取值为 1 到当前线程数。

- 用户态线程调度函数：这是用户态调度的主函数，其实现的部分代码为：

```

1. pub fn user_schedule() {
2.     ...
3.     let mut vdsso_inner = VDSO_DATA.lock();
4.     vdsso_inner.block_sched(); // 阻塞内核抢占
5.     let current_task = vdsso_inner.current_task[current_processor].clone().unwrap();

```

```

6.     if let Some(next_task_ref) = vdso_inner.task_manager.peek() {
7.         if next_task_ref.id.0 == current_task.id.0 {
8.             // 取出下一个任务
9.             let next_task = vdso_inner.task_manager.fetch().unwrap();
10.            // 修改两个任务的状态
11.            ...
12.            // 获取 2 个 TrapContext
13.            let current_trap_cx = get_trap_cx_ptr(current_task.id.1);
14.            let next_trap_cx = get_trap_cx_ptr(next_task.id.1);
15.            // 修改当前任务
16.            vdso_inner.current_task[current_processor].replace(next_t
ask);
17.            vdso_inner.task_manager.add(current_task);
18.            drop(vdso_inner); // 释放锁
19.            unsafe {
20.                __switch_user(current_trap_cx, next_trap_cx);
21.                // 更新 TRAP_CONTEXT_PTR 指向当前任务的 trap context
22.                (VDSO_TRAP_CONTEXT_PTR_BASE as *mut usize).write(next
_trap_cx as usize);
23.            }
24.            let vdso_inner = VDSO_DATA.lock();
25.            vdso_inner.unblock_sched();
26.        } else {
27.            vdso_inner.unblock_sched();
28.            drop(vdso_inner); // 释放锁
29.            sys_yield();
30.        }
31.    } else {
32.        vdso_inner.unblock_sched();
33.        drop(vdso_inner); // 释放锁

```

```

34.         sys_yield();
35.     }
36. }

```

在这里，我们修改用户调度标志位后先判断当前线程队列中是否有任务，再判断队首任务和当前任务是否处于同一个进程，如果不满足则释放标志位并调用系统调用，在内核态切换

如果满足条件，则先修改两个线程的状态，再调用 `__switch_user` 汇编代码完成线程上下文的切换，将当前任务的上下文存入当前的 `TrapContext` 中，再从下一个任务的 `TrapContext` 中恢复上下文，最后再更新共享区中 `Trap` 上下文的指针并释放标志位，完成线程切换。

用户态线程切换的重点是 `__switch_user` 汇编代码的实现。这个函数的两个参数的确定由 `get_trap_cx_ptr` 函数实现，这个函数的逻辑是从地址空间最高位向下，根据 TID 寻找对应的页面，并得到其地址。在 `__switch_user` 内，我们保持并更换线程的 `ra`、`sepc`、`x2~x9`、`x11~x31` 寄存器，而不更新 `TrapContext` 中 `kernel_satp`、`kernel_sp`、`trap_handler` 字段，这样能够在任务陷入时保持进入内核顺利。

整个用户切换函数的逻辑为：

1. 用户态线程调用 `yield_()` 进入 `user_schedule()` 函数，开始用户态调度。
2. `user_schedule()` 函数取共享区数据，然后设置 `block_sched` 为 `true`。
3. 检查调度条件：就绪队列有线程且和当前线程属于同一进程。如果失败则设置 `block_sched` 为 `false`，并调用 `sys_yield()` 进入内核调度。
4. 如果成功，则调用 `fetch()`，取出优先级最高的线程，修改当前线程状态为 `Ready`、下一线程的状态为 `Running`，修改当前运行任务数组，并把当前线程加入调度器就绪队列。
5. 通过汇编指令保存和加载线程上下文。
6. 修改 `block_sched` 为 `false`，用户态线程调度结束。

- 用户态修改线程优先级：我们实现了在用户态进行优先级改变，避免通过系统调用修改线程优先级时的额外开销。实现逻辑是：取共享区的当前任务，修改其中的优先级字段。在修改前后都要对用户调度标志位进行修改，放在修改过程被时钟中断打断。具体实现比较简单，不在这里展示具体代码。
- 内核优先调度队列：在内核中，加入了内核优先调度队列，用来存放 `can_user_sched` 字段为 `false` 的线程。这个队列是用户态不可见的线程队列。线程调度器在内核进行 `fetch()` 时，要先取出内核优先调度队列的任务，将其动态优先级+5 与共享区的就绪队列进行比较，选择优先级高的线程执行。
这样做的优点是：在能解决用户态不可调度的任务的同时，还考虑当前动态优先级最大的任务的紧迫性，防止出现线程饿死。
- 内核态 `trap` 的修改：由于线程可以在用户态进行调度了，因此 `rCore` 之前的 `trap` 策略需要进行一定的修改。当前线程的 `TrapContext` 不再存放在 `sscratch` 中，而是在 `trap_return` 前读取共享区最后一个页面，得到 `TrapContext` 的指针，并在内核态的 `trap.S` 中进行解析和恢复。

此外，还有一些适配工作，用于将新的基于优先级的调度机制与 `rCore` 之前的代码结合，由于没有产生代码逻辑上的改变，这里不再赘述。

4 项目测试

本节进行项目的测试，从三个角度测试代码的正确性和有效性：

- 协程模块测试
- 用户态线程调度测试
- 整体测试

4.1 协程模块测试

这部分测试主要验证协程模块能够正常工作，包括：

- 测试协程基础功能：验证协程运行时、协程调度器能够正常运行，并且用户编写的异步函数能够正常创建协程并运行。
- 测试协程的同步功能：验证协程的跨线程提交和协程事件机制能够正常运行。
- 测试协程的优先级功能：验证协程的优先级机制能够在协程模块内正常运行。

4.1.1 协程模块测试方法

我们分三次测试对协程模块进行验证。

(1) 测试协程基础功能

为了测试协程的运行时、调度器、协程对象正常运行，我们编写两个简单的异步函数，在异步函数内调用我们定义的库函数协程放弃运行函数 `coro_yield()`，提交到运行时并观察输出。预期输出是看到两个异步函数交替输出。

编写测试代码 `coroutine_test` 如下：

```

1. pub fn main() -> i32 {
2.     async fn another_task() {
3.         println!("another task");
4.         coro_yield_once().await;
5.         println!("another task end");
6.     }
7.     async fn task() {
8.         println!("step 1");
9.         coro_yield_once().await;
10.        println!("step 2");
11.        coro_yield(3).await;
12.        println!("step 3");
13.    }
14.    submit_distribute_coroutine(task(), 1);
15.    submit_distribute_coroutine(another_task(), 1);

```

```

16.     wait_all_coroutines();
17.     quit_coroutine_runtime();
18.     println!("quit coroutine test");
19.     0
20. }

```

(2) 测试协程的同步功能

测试我们定义的协程事件 `Event`，同时测试协程在提交时跨越线程进行提交。我们编写测试代码，创建 3 个线程，每个线程提交 2 个协程。每个线程中，第一个协程等待一个事件、另一个协程调用我们的库函数线程睡眠 `coro_sleep()` 后唤醒事件，之后主动 `coro_yield()`。预期输出是协程 2 先运行，睡眠一段时间后唤醒协程 1。

编写测试代码 `coroutine_test_all`。测试代码主函数不再展示，创建的线程的入口函数代码如下：

```

1. fn cor_task(thread_num: usize) {
2.     let c_event = create_event();
3.     let c_event_clone = c_event.clone();
4.     let coroutine_1 = async move {
5.         println!("THREAD {} coroutine 1 start", thread_num)
6.         ;
7.         c_event_clone.wait().await;
8.         println!("THREAD {} coroutine 1 end", thread_num);
9.     };
10.    let coroutine_2 = async move {
11.        println!("THREAD {} coroutine 2 start", thread_num)
12.        ;
13.        coro_sleep(3000).await;
14.        c_event.set();
15.        coro_yield_once().await;
16.        println!("THREAD {} coroutine 2 end", thread_num);
17.    };

```



```

16.     submit_distribute_coroutine(coroutine_1, thread_num);
17.     submit_distribute_coroutine(coroutine_2, thread_num + 1
    );
18.     exit(0);
19. }

```

(3) 测试协程的优先级功能

测试协程模块内的优先级机制，保证高优先级协程先执行。编写测试代码，连续提交 20 个协程，其优先级分别为 1 到 20。预期输出为高优先级的先调度，低优先级后调度

编写测试代码 `coroutine_prior_test`:

```

1. pub fn main() -> i32 {
2.     let coro_num = 20;
3.     for i in 0..coro_num {
4.         let future = async move {
5.             println!("coroutine {} before sleep", i);
6.             coro_sleep(1000).await;
7.             println!("coroutine {} after sleep", i);
8.         };
9.         submit_distribute_coroutine(future, i + 1);
10.    }
11.    wait_all_coroutines();
12.    quit_coroutine_runtime();
13.    0
14. }

```

4.1.2 协程模块测试结果

(1) 运行 `coroutine_test`，结果如下：

```

root@LAPTOP-RMUH26AB: /h  ×  +  ▾
adder_peterson_spin
*****/
[ INFO] [kernel] VDSO heap allocator initialized
[ INFO] [kernel] VDSO data initialized
Rust user shell
>> coroutine_test
step 1
another task
step 2
another task end
step 3
quit coroutine test
>> |

```

可以看到，两个协程交替输出，并且运行结果正确。协程运行时、协程调度器、协程对象均能正常运行。说明协程的基本机制是可以正常工作，

(2) 运行 `coroutine_test`，结果如下：

```

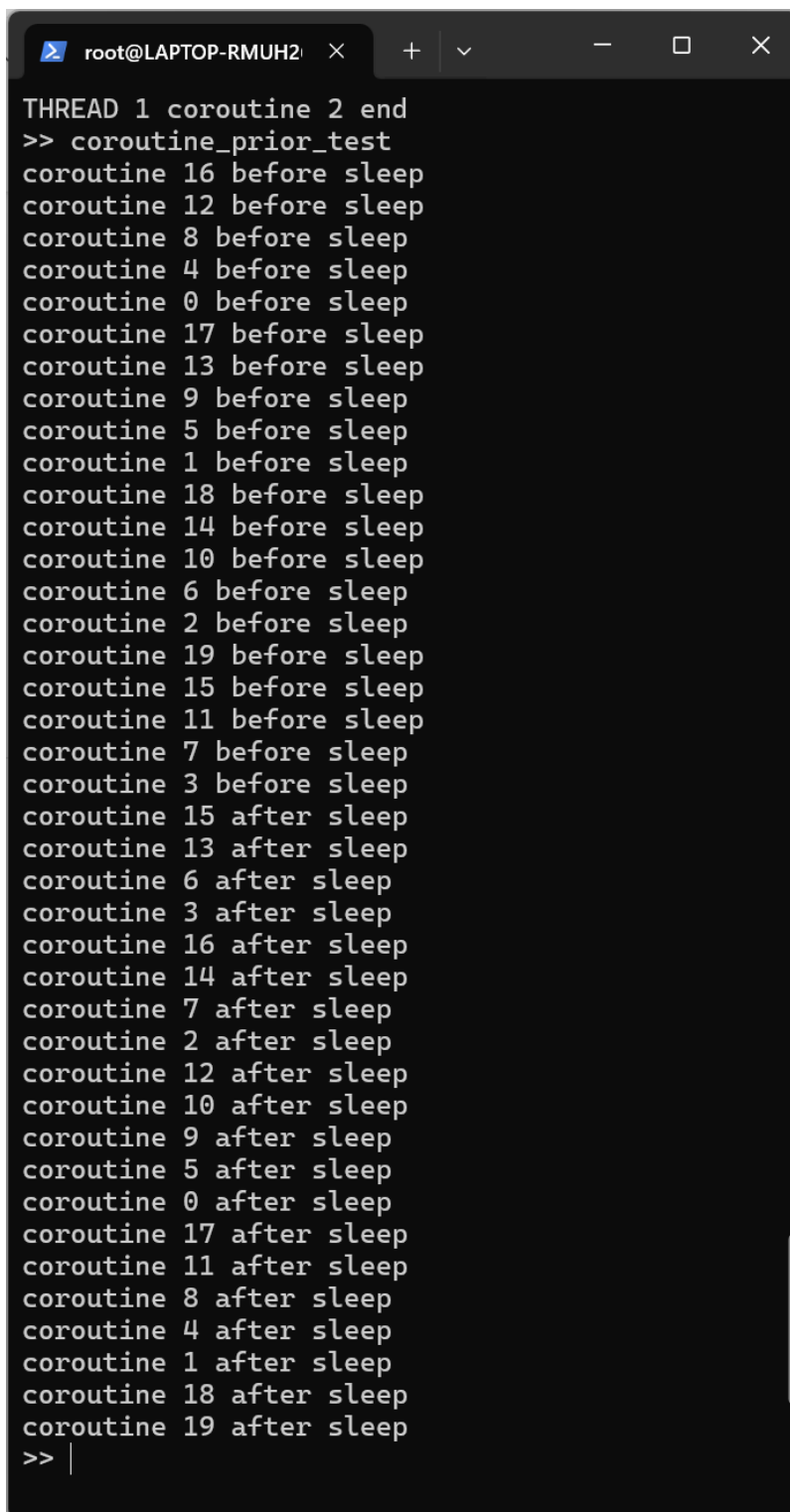
root@LAPTOP-RMUH26AB: /h  ×  +  ▾
step 3
quit coroutine test
>> coroutine_test_all
THREAD 0 created
THREAD 1 created
THREAD 2 created
All threads created
Priority must be between 1 and 20, use default priority.
THREAD 0 coroutine 1 start
THREAD 2 coroutine 1 start
THREAD 2 coroutine 2 start
THREAD 0 coroutine 2 start
THREAD 1 coroutine 1 start
THREAD 1 coroutine 2 start
THREAD 0 coroutine 1 end
THREAD 0 coroutine 2 end
THREAD 2 coroutine 1 end
THREAD 2 coroutine 2 end
THREAD 1 coroutine 1 end
THREAD 1 coroutine 2 end
>> |

```

可以看到，能够正常运行，且每个线程的协程 1 的结束早于协程 2，说明在协程的 `coro_yield()` 能够正常工作。

由于在图片中不易展示协程的休眠和事件等待，详细情况请参见演示视频。

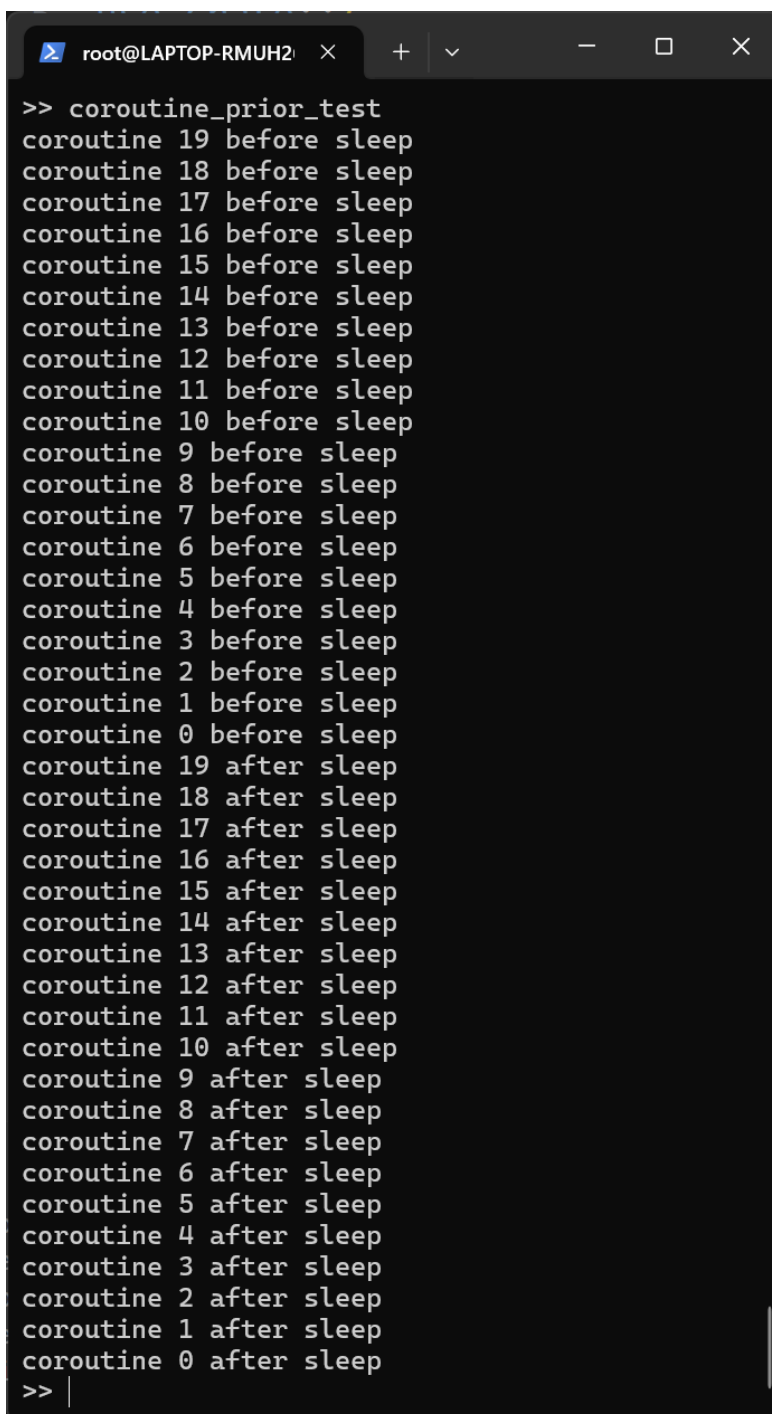
(3) 运行 `coroutine_prior_test`，结果如下：



```
root@LAPTOP-RMUH2 x + v - □ ×
THREAD 1 coroutine 2 end
>> coroutine_prior_test
coroutine 16 before sleep
coroutine 12 before sleep
coroutine 8 before sleep
coroutine 4 before sleep
coroutine 0 before sleep
coroutine 17 before sleep
coroutine 13 before sleep
coroutine 9 before sleep
coroutine 5 before sleep
coroutine 1 before sleep
coroutine 18 before sleep
coroutine 14 before sleep
coroutine 10 before sleep
coroutine 6 before sleep
coroutine 2 before sleep
coroutine 19 before sleep
coroutine 15 before sleep
coroutine 11 before sleep
coroutine 7 before sleep
coroutine 3 before sleep
coroutine 15 after sleep
coroutine 13 after sleep
coroutine 6 after sleep
coroutine 3 after sleep
coroutine 16 after sleep
coroutine 14 after sleep
coroutine 7 after sleep
coroutine 2 after sleep
coroutine 12 after sleep
coroutine 10 after sleep
coroutine 9 after sleep
coroutine 5 after sleep
coroutine 0 after sleep
coroutine 17 after sleep
coroutine 11 after sleep
coroutine 8 after sleep
coroutine 4 after sleep
coroutine 1 after sleep
coroutine 18 after sleep
coroutine 19 after sleep
>> |
```

可以看到，这样的运行情况显得很奇怪，似乎没有满足优先级调度的顺序。但详细分析可以发现：在 `before sleep` 中，协程的 CID 是有规律的，每次相隔为 4。这是由于我们设置了调度器线程数量为 4，但是缺运行在单核 CPU 上，导致协程虽然按照优先级进行分配，但调度器线程因为协程队列中还有任务而不主动放弃运行，并且在我们设计中不支持用户态线程的抢断。才会出现这种情况。

修改协程调度器线程数量为 1，再次测试：



```
>> coroutine_prior_test
coroutine 19 before sleep
coroutine 18 before sleep
coroutine 17 before sleep
coroutine 16 before sleep
coroutine 15 before sleep
coroutine 14 before sleep
coroutine 13 before sleep
coroutine 12 before sleep
coroutine 11 before sleep
coroutine 10 before sleep
coroutine 9 before sleep
coroutine 8 before sleep
coroutine 7 before sleep
coroutine 6 before sleep
coroutine 5 before sleep
coroutine 4 before sleep
coroutine 3 before sleep
coroutine 2 before sleep
coroutine 1 before sleep
coroutine 0 before sleep
coroutine 19 after sleep
coroutine 18 after sleep
coroutine 17 after sleep
coroutine 16 after sleep
coroutine 15 after sleep
coroutine 14 after sleep
coroutine 13 after sleep
coroutine 12 after sleep
coroutine 11 after sleep
coroutine 10 after sleep
coroutine 9 after sleep
coroutine 8 after sleep
coroutine 7 after sleep
coroutine 6 after sleep
coroutine 5 after sleep
coroutine 4 after sleep
coroutine 3 after sleep
coroutine 2 after sleep
coroutine 1 after sleep
coroutine 0 after sleep
>> |
```


可以看到，运行结果正确。注意到有时会连续出现两次 ‘-’，这是因为调度到了该用户进程的主线程，主线程中一直在 `waittid` 而没有输出，因此会多打印 ‘-’。总体而言，运行结果正常。

4.3 综合测试

这一部分进行综合测试，验证协程的优先级调度带动线程改变优先级，从而进行线程的动态优先级调度。

4.3.1 综合测试方法

修改协程调度器线程数量为 4，并在每次进行线程切换时都打印所有线程的动态优先级，观察线程的优先级变化，同时观察线程调度顺序。

这样修改时，可能由于单核运行多线程程序导致协程运行顺序不正确，但是能够直接的反映出协程优先级对线程优先级的影响，同时能直观的反映老化机制，展示出动态优先级的变化。因此用这种方式测试是可行的。

此外，为了扩大优先级变化的区间，我们把所有线程在初始时的优先级定为 10。

编写测试代码 `coroutine_thread_test`:

```
1. pub fn main() -> i32 {
2.     for i in 0..10 {
3.         let task = async move {
4.             println!("task {}", i);
5.             coro_yield_once().await;
6.             println!("task {} end", i);
7.         };
8.         submit_distribute_coroutine(task, i / 2);
9.     }
10.    wait_all_coroutines();
11.    quit_coroutine_runtime();
12.    println!("quit coroutine test");
```

```

13.     0
14. }

```

4.3.2 用户态线程调度测试结果

测试结果为：

结果的前半部分：

```

root@LAPTOP-RMUH26AB: /h  ×  +  ▾
task 2-0, dynamic prio: 10
task 2-0, dynamic prio: 11
task 2-0, dynamic prio: 12
task 2-0, dynamic prio: 10
task 2-0, dynamic prio: 11
task 2-0, dynamic prio: 12
Priority must be between 1 and 20, use default priority.
Priority must be between 1 and 20, use default priority.
task 2-1, dynamic prio: 10
task 2-2, dynamic prio: 10
task 2-3, dynamic prio: 10
task 2-4, dynamic prio: 10
task 2-5, dynamic prio: 10
task 2-0, dynamic prio: 10
task 2-0, dynamic prio: 11
task 2-1, dynamic prio: 11
task 2-2, dynamic prio: 11
task 2-3, dynamic prio: 11
task 2-4, dynamic prio: 11
task 2-5, dynamic prio: 11
task 2-0, dynamic prio: 12
task 2-2, dynamic prio: 12
task 2-1, dynamic prio: 12
task 2-3, dynamic prio: 12
task 2-4, dynamic prio: 12
task 2-5, dynamic prio: 12
task 2-2, dynamic prio: 13
task 2-3, dynamic prio: 13
task 2-1, dynamic prio: 13
task 2-4, dynamic prio: 13
task 2-5, dynamic prio: 13
task 2-0, dynamic prio: 3
task 1
task 9
task 5
task 2-3, dynamic prio: 14
task 2-1, dynamic prio: 14
task 2-2, dynamic prio: 10
task 2-4, dynamic prio: 14
task 2-5, dynamic prio: 14
task 2-0, dynamic prio: 4
task 6
task 2
task 2-1, dynamic prio: 15

```

结果的后半部分:

```
root@LAPTOP-RMUH26AB: /h × + v
task 2-4, dynamic prio: 9
task 2-5, dynamic prio: 9
task 2-3, dynamic prio: 11
task 2-0, dynamic prio: 6
task 2-3, dynamic prio: 12
task 2-4, dynamic prio: 10
task 2-5, dynamic prio: 10
task 2-0, dynamic prio: 7
task 2-3, dynamic prio: 13
task 2-4, dynamic prio: 11
task 2-5, dynamic prio: 11
task 2-0, dynamic prio: 8
task 2-5, dynamic prio: 12
task 2-4, dynamic prio: 12
task 2-0, dynamic prio: 9
task 2-5, dynamic prio: 13
task 2-4, dynamic prio: 13
task 2-0, dynamic prio: 10
task 2-4, dynamic prio: 14
task 2-0, dynamic prio: 11
task 2-5, dynamic prio: 4
task 2-0, dynamic prio: 12
task 2-5, dynamic prio: 5
task 2-0, dynamic prio: 13
task 2-5, dynamic prio: 6
task 2-5, dynamic prio: 7
task 2-0, dynamic prio: 3
task 2-5, dynamic prio: 8
task 2-0, dynamic prio: 4
task 2-5, dynamic prio: 9
task 2-0, dynamic prio: 5
task 2-5, dynamic prio: 10
task 2-0, dynamic prio: 6
task 2-5, dynamic prio: 11
task 2-0, dynamic prio: 7
task 2-5, dynamic prio: 12
task 2-0, dynamic prio: 8
task 2-0, dynamic prio: 9
task 2-0, dynamic prio: 10
task 2-0, dynamic prio: 11
task 2-0, dynamic prio: 12
quit coroutine test
>> qemu-system-riscv64: terminating on si
```


可以看到，在最开始的部分，协程的多个线程还没有创建出来时，主线程的优先级不断从 10 增加到 12，再回到 10。这是因为主线程、初始线程 `Initproc` 和用户 `Shell` 三个线程轮流调度。

之后，在所有线程不断创建出来后，运行线程从主线程切换为 2 号线程，这时主线程的动态优先级变为 3：初始是 1，创建线程过程中老化了 2。下一个运行的线程为队首线程 2 号线程，可以在下一轮打印中发现 2 号线程优先级变为了 10，其他线程优先级都增加了 1。

在代码运行结果的后半部分，1 号线程和 2 号线程已经运行结束，可以看到，在后半部分最开始的部分 0 号、3 号、4 号、5 号线程的优先级都在增加。这是因为时钟中断导致初始线程和用户 `Shell` 得到了调度。之后，3 号线程和 4 号线程相继得到调度并运行结束，最终结束 5 号线程和 0 号线程，运行结束。

可以看到，运行结果具有可解释性，符合预期的输出。因此，可以认为测试通过，可以验证协程的优先级调度带动线程改变优先级，从而进行线程的动态优先级调度。

5 开发过程中的困难和解决方法

本小节介绍我们在开发过程中遇到的困难，并简单说明其解决方法。

5.1 协程调度器开发中的死锁问题

在开发协程模块时，因为协程被设计成多个线程都可以向调度器中提交任务，这需要保证调度器线程安全，因此使用了 `Mutex` 锁。但因为协程结构中包括了当前调度器的弱引用，因此需要使用 `Arc` 包装，这就需要使用 `Arc` 锁。此时，对协程的操作很容易产生死锁，例如在向指定调度器提交任务时，提交者获得了调度器的锁，然后创建协程时又试图获取调度器的锁，这时就会产生死锁。

为了解决这个问题，我们把提交过程中，提交任务的队列变成了公共队列，即所有协程都提交到一个队列中，然后再由守护线程分发任务到各个调度器内。这样通过运行时管理所有协程队列更加简单，避免了死锁。

5.2 共享区报错 LoadPageError 问题

在开发共享区时经常出现报错 LoadPageError，原因是其次我们的共享区中使用的是 Arc 指针，而这个指针可能会指向共享区外的数据，导致在内核态可以正常运行，而在用户态尝试调度时会报错。

为了解决这个问题，我们扩大了共享区的页数，并分配处 16 页用来当作共享堆空间，把所有线程的 TaskSched 结构体都放在共享堆空间中，这样就可以保证在用户态调度时不会出现跨地址空间的问题。

5.3 用户态调度报错 Unsupported syscall 问题

在初步实现了用户态调度时，进行调度测试发现报错 Unsupported syscall，但是我们的测试代码并没有发起系统调用，并且也不应该进入 syscall 部分。初步判断原因是我们错误的把 TaskContext 中的 ra 当作是用户态返回的地址，而它实际上是内核态返回地址，例如 Trap_return。

对此，我们额外的增加了 user_ra 字段，在进入内核前保存用户态返回地址，在 Trap_return 中恢复用户态返回地址。但测试发现，仍然不对，并且多了报错 StorePageError 的问题。进一步调试，我们发现问题出在了进入用户态调度代码 user_schedule 后，返回地址 ra 没有保存正确。

之后，我们将调用汇编函数 __switch_user 由 call 的方式转换成 jr 的方式，来防止 ra 被覆盖。但这样还是没有解决问题。

最后，我们发现调度的所有上下文应该全部保存，包括所有寄存器、用户栈等信息，而内核栈也应该同时转换，否则就会出现内核中地址不能正确返回。对此，我们选择直接使用 TrapContext 作为保存上下文的结构体，并修改 Trap_return 中恢复所有寄存器、用户栈等信息，然后在 user_schedule 中直接保存和加载上下文。最终解决了这个问题。

5.4 vs code 的 rust-analyzer 崩溃的问题

rust-analyzer 的问题很多，经常出现加载项目时加载失败，导致代码编写时无法正常提供提示、跳转等功能。

对此，我们搜索了很多资料，最后在 rust-analyzer 的 Github 仓库中发现了相关 Issue (<https://github.com/rust-lang/rust-analyzer/issues/10910>)，按照社区中各位的建议，在清空 Cargo 的缓存后解决了这个问题。

6 参考资料

1. Chen, Y., & Wu, Y. (2020-2022). rCore-Tutorial-Book-v3. rCore-Tutorial-Book 第三版. <https://rcore-os.cn/rCore-Tutorial-Book-v3/index.html>
2. rosy233333. (n.d.). vdso_crate_template. rosy233333/vdso_crate_template. https://github.com/rosy233333/vdso_crate_template