哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

COS

——快速、灵活、抗干扰的用户态调度框架

队伍名称：COS

成员：	段子豪（组长）
	谢岸峰
	黄雯萱

校内指导老师：夏文、李诗逸、仇洁婷

校外指导老师：吴云、贾浩（字节跳动）

2023 年 08 月

摘　要

COS 是我们队从零开发的用户态调度框架，由内核态的 COS 调度类、用户态的 Lord 线程、负责内核态到用户态通信的消息队列、负责用户态到内核态通信的 shoot 系统和 cgroup 兼容这五个部分组成。

COS 实现了消息队列、shoot area、中断加速调度策略执行和自适应调度优先级调整这四个高性能优化点，具有**快速、灵活、抗干扰**的特性。

我们的赛题完成度如下：

	目标	说明
基本目标	task delegation	完成，初赛 EXT 时延在 **100ms-1s**；决赛 COS 通过 shoot 与中断将线程以 **10 - 100us** 的时延调度到目标 CPU 上并且运行，提升 **10000** 倍
基本目标	task preemption	完成，初赛 EXT 抢占机制通过插队实现，时延较高；决赛 COS 通过 IPI 中断与消息队列支持对线程快速抢占与抢占消息传递
基本目标	fairness	完成，初赛 EXT 优先级低于 CFS，只具备公平性，性能差；决赛 COS 调度类优先级自适应变化，兼顾公平与性能
基本目标	CPU cgroup	初赛 EXT 未完成；决赛已完成，通过 COS 提供的四个系统调用对 cgroup 进行控制，限制 COS 线程 CPU 使用率
基本目标	性能测试报告	完成，初赛较为简陋；决赛 COS 对 task delegation 时延和 cgroup 均进行了相关测试
扩展目标	挖掘实际应用场景	初赛未完成；决赛已完成，选取 IO 密集型场景 RocksDB，测试吞吐量与时延
扩展题	针对场景将调度注入内核达到优化	初赛未完成；决赛已完成，决赛对 COS、CFS、EXT 在 RocksDB 场景，测试吞吐量与时延关系，COS 尾延迟平均为 CFS 的 **1.23%**
额外目标	与 ghOSt 进行对比	初赛未完成；决赛已完成，COS 的 task delegation 时延为 ghOSt 的 **0.1%**，IO 密集型场景 rocksdbCOS 尾延迟平均为 ghOSt 的 **2.26%**，并且支持 cgroup
	完成度 100%	

目 录

1 项目背景与相关工作

1.1 用户态调度框架背景

线程调度是操作系统最核心的功能。Linux 支持多种调度实现，通过支持多种调度类[1]来实现多种调度策略。值得注意的是，Linux 的设计初衷是尽可能地兼容多种使用场景，不会针对特定场景进行优化。

为了优化特定应用程序的性能，如果 Linux 内核中现有的调度策略不能满足当前负载需要，程序员可以选择创建一个全新的内核调度类，以针对特定负载场景进行优化。然而，对 Linux 内核进行修改会有如下缺点：首先，Linux 内核的复杂性，在 Linux 最核心的调度模块进行修改对程序员的能力是一种极大的考验；其次，这种针对特定负载定制优化的调度类不具备通用性，在其他负载场景性能往往远远差于 Linux 的默认调度器完全公平调度器[2]（下用 CFS 指代）；而且，根据云服务厂商经验，例如在 2015-2021 年期间，Google 部署新的内核需要 O（月）时间复杂度；最后，这种大规模重新部署内核调度策略极易出错，例如，在 2020 年 6 月，Google 的磁盘服务器上出现了调度器错误，导致数百万美元的收入损失。以上的缺陷使得许多开发人员更倾向于使用现有的通用调度策略 CFS。

用户态调度框架则是一种很合理的解决方案，用户态调度框架分为内核实现和用户实现：内核通过为用户态提供接口（系统调用，eBPF[3]钩子），将策略决策委托给用户空间；用户则通过内核提供的接口在用户态搭建部署自己的调度策略。用户态调度框架将内核调度机制与策略定义分开：实现机制驻留在内核中，很少发生变化；调度策略定义驻留在用户空间中，变化频繁。这使得基于用户态调度框架部署调度算法和直接修改内核相比：开发难度、通用性、部署时间、稳定性等方面均远胜于直接修改 Linux 内核。

[1] Linux 调度类（*scheduling classes*）：可以看作一个基类，内核调度策略如完全公平调度器、实时调度器等继承了该基类，实现了相关方法。详情可见下文。

[2] 完全公平调度器（*Completely Fair Scheduler*）：简称 CFS，为 Linux 内核默认调度器。

[3] eBPF（*Extended Berkeley Packet Filter*）：一个能够在内核运行沙箱程序的技术，提供了一种在内核事件和用户程序事件发生时安全注入代码的机制，使得非内核开发人员也可以对内核进行控制。

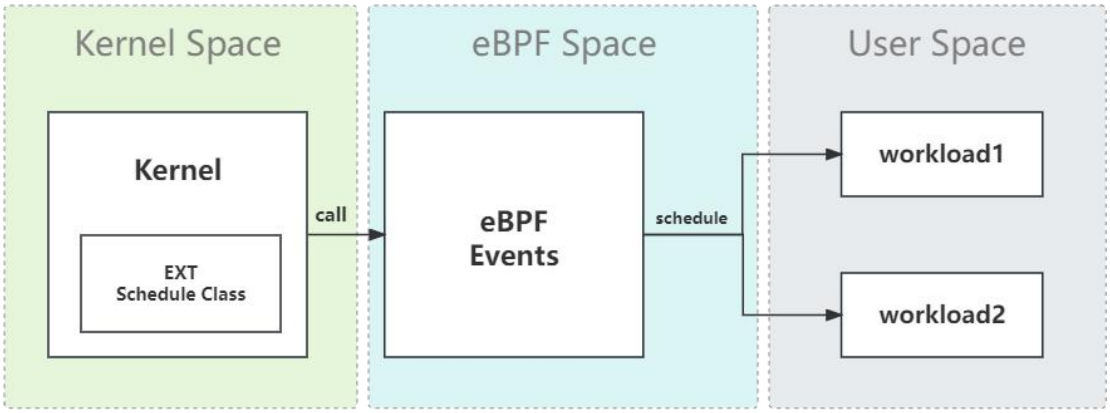
回顾赛题要求：实现基于 CFS 的用户态调度框架，基于 CFS 即兼容 CPU cgroup[4]，也就是说需要实现的用户态框架不仅需要具备用户态框架的基本特性——用户态控制调度行为，还需要兼容 cgroup，支持限制线程的 CPU 使用率。

1.2 国内外研究调研

1.2.1 EXT

EXT（*Extensible Scheduler Class*）是由 Meta 开发的一个基于 eBPF 实现的用户态调度框架。具体而言，EXT 在内核中新增一个调度类，就类似常用的 CFS 调度类。该调度类定义了一系列最终由用户在用户态实现的 eBPF 钩子，并且将其挂载到 Linux 内核调度策略起作用的关键节点，从而达到用户态调度策略动态注入的效果。同时，该调度类兼容 cgroup，可以对 EXT 线程的 CPU 使用率进行限制。

在初赛阶段和决赛初期，我们基于 EXT 内核拓展开发了一个用户态调度框架。



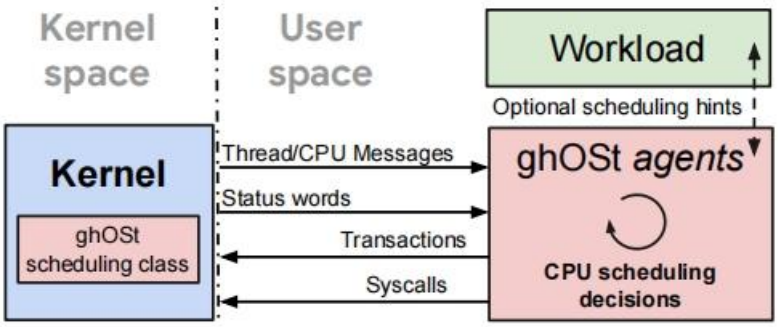
1.2.2 ghOSt

ghOSt 是由 Google 和 Stanford 联合开发的用户态调度框架，其相关成果发表在 2021 年操作系统两大顶会之一的 SOSP[5]中，在 Google 内部已经广泛部署。

ghOSt 的内核部分被实现为一个调度类。这个调度类向用户态提供一系列的系统

[4] cgroup（*Control Group*）：用于限制进程对系统各种资源的使用，比如 CPU、内存、I/O 等。
[5] SOSP（*Symposium on Operating Systems Principles*）：计算机系统最顶级的两个会议之一，和 OSDI 并称为系统界奥斯卡。

调用来定义任意调度策略。为了帮助用户态的 Agent 做出调度决定，内核态与用户态间通过共享内存传递线程状态。Agent 通过系统调用将调度策略部传入内核。但是 ghOSt 在云原生场景下用途十分有限，因为其不支持 cgroup。

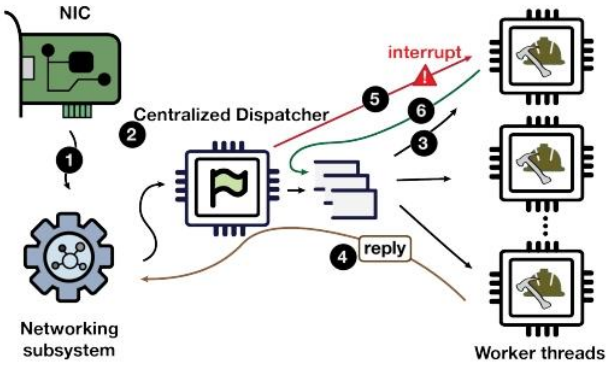


1.2.3 Shinjuku

Shinjuku 是由 Stanford 开发的一个高度专用网络调度场景的操作系统，相关成果发表在 2019 年的网络顶会 NSDI[6]中。

它通过硬件支持以及对 Linux 内核模块的修改，使得基于其上运行的 Shinjuku 调度算法在微秒粒度上实现了抢占机制。在微秒级应用程序（如网络搜索、社交网络、机器翻译等云应用程序）的请求处理上表现出高吞吐量、低尾延迟的优良特性。

Shinjuku 调度算法是基于 centrallized model[7]实现的用户态抢占式多优先级 FIFO[8]调度策略。也就是说，Shinjuku 调度算法会将线程以多级 FIFO 调度队列的形式管理，并且根据优先级的不同为每个线程分配一定的时间片，并且在用户态实现了微秒粒度的抢占机制。



[6] NSDI（*Networked Systems Design & Implementation*）：计算机系统的一个顶级会议，方向偏网络。
[7] centrallized model：该模型创建了一个线程作为专门的调度线程来执行调度逻辑、管理其他的工作线程并将它们调度到特定 CPU 上。
[8] FIFO（*First in, First out*）：即先来先服务调度算法。

1.3 成果概述

在初赛，我们基于 Meta 开发的 EXT 内核暴露的 eBPF 事件拓展开发了用户态调度框架：

1. 开发了 EXT 的用户态框架 lib 库，增加了 Agent 端与 Client 端的通信；
2. 新增了 SJF[9]、CFS、MFQ[10]以及 Shinjuku 这四种调度算法实现。

在决赛初，我们深入 EXT 内核进行了调研和开发：

1. 完善了 EXT 内核处理机制，为其新增了两个 helper 函数；
2. 另外开发了一版基于 eBPF 事件实现的 Shinjuku 调度算法；
3. 扩展了初赛的用户态框架，新增了基于共享内存的消息队列实现。

与此同时，我们在决赛期间对 EXT 内核进行了建设性地分析，发现 EXT 存在很多痛点，这些缺陷阻碍我们性能的进一步提升和更加便捷地部署用户态调度策略：

- EXT内核性能痛点

1. 用户态调度器不能持续运行在 CPU 之上；
2. 用户态做出的调度决策，具有毫秒（ms）甚至秒（s）级的时延；
3. eBPF 程序与内核 C 语言相比天然的性能劣势（基于虚拟机的语言，资源限制）；
4. EXT 调度类优先级位于 CFS 之下，在目前 CFS 作为所有线程默认调度器的情况下，EXT 线程运行的时间很有限；
5. eBPF 程序是同步运行的，这意味着它们必须对调度事件快速做出反应，并在完成之前阻塞 CPU，这无法实现内核消息发送与消费的异步性。

故我们决定从零开发新的用户态调度框架——COS，该用户态调度框架摒弃之前 EXT 的 eBPF 注入策略，采用系统调用部署用户态调度策略。

- COS优点

1. 用户态调度器可以持续运行在 CPU 之上；
2. 通过硬件支持的 IPI[11]，用户态做出的调度决策时延可以达到次微秒（μs）级；

[9] SJF（*Shortest Job First*）：最短时间优先调度算法。
[10] MFQ（*Multilevel Feedback Queue*）：多级反馈队列调度算法。
[11] IPI（*Inter-Processor Interrupt*）：一种特殊类型的中断，即在多处理器系统中，一个 CPU 向系统中的目标 CPU 发送中断信号，以使目标 CPU 执行特定的操作。

3. C 语言相比 eBPF 解释性语言性能更强，无资源限制；

4. COS 采取了优先级动态变动策略，在保证性能的同时兼顾系统稳定性；

5. 通过消息队列实现了内核与用户间的异步通信。

所以，决赛我们基于 Linux 6.4.0+版本，新增了 COS 调度器，成功解决了以上痛点，系统能够稳定运行，该调度器同时支持 cgroup，在很多场景中性能远远优于 CFS 与 EXT。

2 系统设计

2.1 EXT

在初赛和决赛中，我们基于 EXT 内核，对其用户态框架进行了拓展开发。

2.1.1 背景介绍：EXT 内核设计

在本部分中，我们将针对 Meta 开发的 EXT 内核做一个简要的原理简介。

- eBPF钩子概览

EXT 提供了一系列 eBPF 钩子。在具体实现中，我们可以在 eBPF 事件实现中将相关消息传递给用户态（userland 思路），也可以直接在 eBPF 程序中保存任务调度队列，并在相关 eBPF 事件中实现处理逻辑。主要的 eBPF 事件及其含义如下表所示：

eBPF 事件	触发时机	一般实现逻辑
enqueue()	在任务需要进入调度类调度队列时触发。	将一个任务入队到 BPF Scheduler 中存储任务的数据结构，或者直接 dispatch 该任务到对应 CPU 上。
dispatch()	在一个 CPU 的调度队列为空时触发。	催促 BPF 程序做出调度决策，并将需要调度的线程入队到某个 DSQ（*Dispatch Queue*）中。

runnable()、 running()、 stopping()、 quiescent()	当任务分别变为可运行、在 CPU 上开始运行、离开 CPU 或变为不可运行时被触发。 当 stopping 事件的 runnable 参数为 true 时，表明任务被抢占；否则则为被阻塞。	

- eBPF钩子与内核调度类回调函数

EXT 在内核中添加了 SCHED_EXT 调度类，实现了调度类相关回调函数，这些回调函数代表着事件的触发，如 enqueue_task 代表线程入队。在这些回调函数的具体实现中放置了 eBPF 的钩子，从而做到了 eBPF 事件的精准触发。

```
static void enqueue_task_scx(struct rq *rq, struct task_struct *p, int enq_flags)
{
  if (SCX_HAS_OP(enqueue)) { // 如果用户实现了enqueue eBPF事件
        SCX_CALL_OP_TASK(SCX_KF_ENQUEUE, enqueue, p, enq_flags); // 触发eBPF事件
    }
}

DEFINE_SCHED_CLASS(ext) = {
    .enqueue_task       = enqueue_task_scx
}
```

主要的 eBPF 钩子与调度类回调函数的映射关系如下表所示：

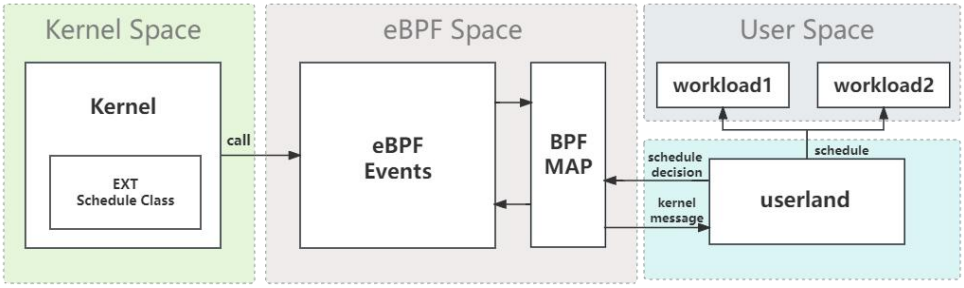
调度类回调函数	eBPF 钩子	备注
enqueue_task	runnable	当 task 被分派到非 local CPU 上时，只会触发 runnable 事件；当 task 耗尽时间片时，只会触发 enqueue 事件。
	enqueue	
dequeue_task	stopping(false)	当 task 触发 dequeue_task 事件，表明它即将因阻塞/死亡停止运行，故而在此触发 stopping 事件（runnable 为 false 表明非抢占）和 quiescent 事件。
	quiescent	

pick_next_task	running	当 task 被挑选为下一个需要运行的对象时，它就可以被认为处于“正在运行”状态。

2.1.2 用户态框架改进与算法简化

- 框架改进

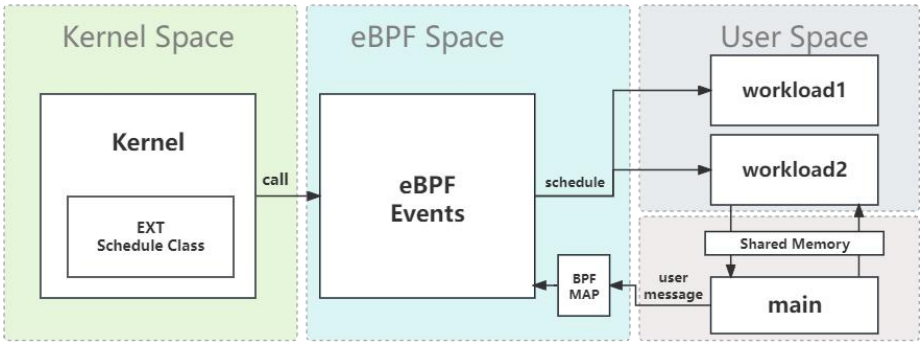
在初赛中，EXT 用户态调度框架使用的是 userland 模型：



然而这个模型在性能的方面不尽人意。一个 task 从进入运行队列到被实际调度到 CPU 上运行要经过以下几个阶段：

1. 传入 eBPF MAP1 被用户态获取
2. 用户态执行入队以及决策操作、
3. 将结果传回 eBPF MAP2

若替换为 centralized 模型，也即将在用户态只负责加载和打开 eBPF 程序以及在必要场景里和负载线程进行通信，具体决策在 eBPF 中进行，则可以避免这种繁杂过程。



然而，将决策部分放置于 eBPF 程序中实际上可能受到一些限制：

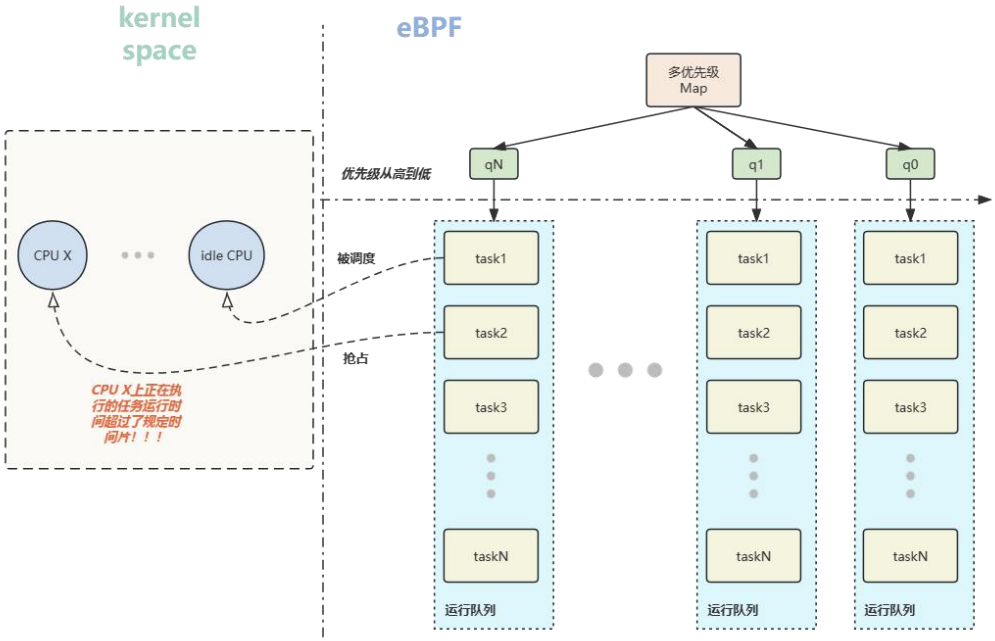
1. eBPF 程序无法完成一些操作，如申请动态内存；

2. eBPF 程序无法直接感知负载状态，如是否需要运行。

对于第一个困难，使用全局静态变量和 EXT 内核提供的 DSQ 队列；对于第二个困难，使用中间代理人 agent 线程，它通过共享内存接收负载线程的状态改变消息，然后写回 eBPF MAP 中让 eBPF 程序感知。

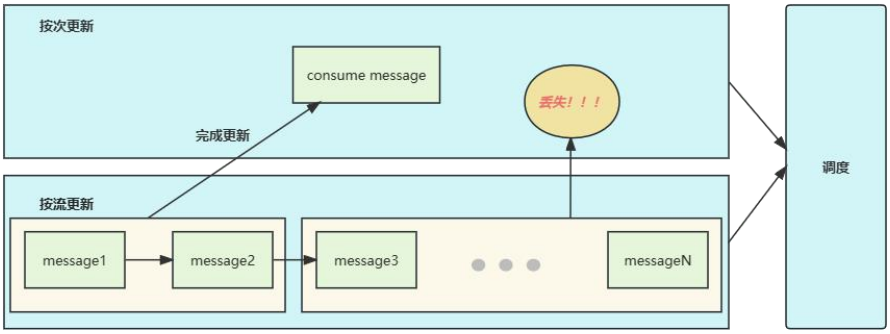
- 算法简化

Shinjuku 调度算法实质上就是一个以 FIFO 为基础实现的、增加进程间通信的抢占式调度算法。因而，实现 Shinjuku 调度算法的**核心数据结构**是多级双端队列（“级”代表优先级），**最根本要点**是用户态抢占和进程间通信。



我们主要针对“进程间通信”这一部分进行了算法简化。

在初赛中，我们参考了 ghOSt 的 Shinjuku 算法实现，在进程间通信部分引入了较为复杂的状态机，通过共享内存来进行**按调用次数**刷新负载线程信息；而在决赛中，我们使用基于共享内存的消息队列来进行**按流**进行负载线程信息的刷新，虽然牺牲了一定稳定性，但这减少了 **50%**的代码量，极大简化了 Shinjuku 算法的实现。

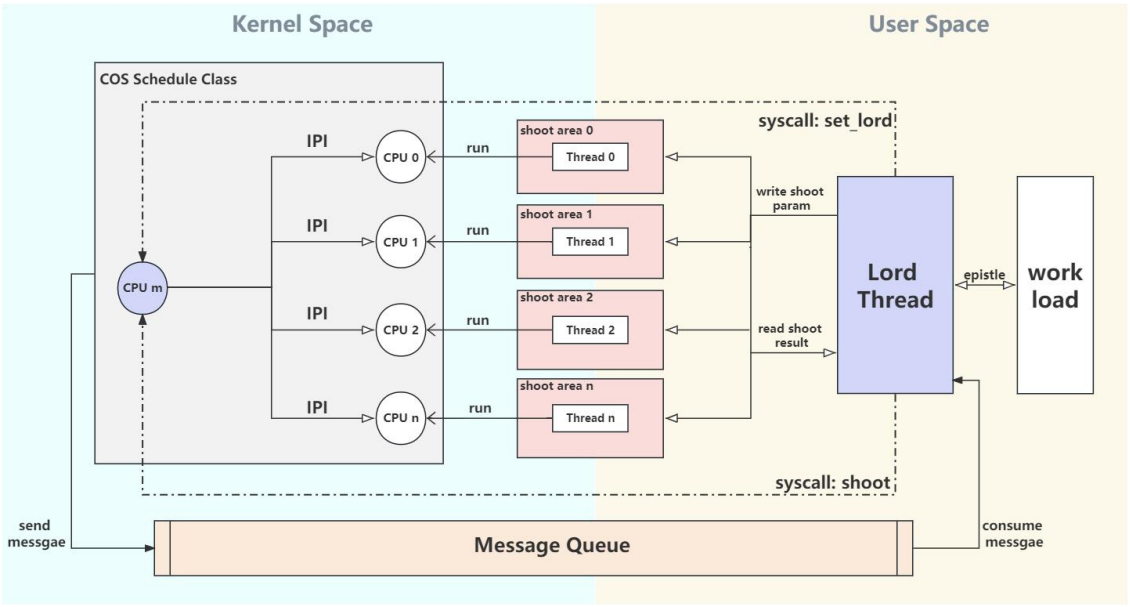


2.2 COS

2.2.1 COS 设计

COS是我们在Linux内核新增的调度类，它具备用户态控制调度策略的能力，用户态通过系统调用去开启COS调度，并且将用户态的调度策略部署到内核。同时，用户态与内核态可以通过基于共享内存的消息队列来进行高效的异步通信。

COS 的整体架构如下图所示：



COS中设置了4个系统调用来支持用户态部署调度策略：

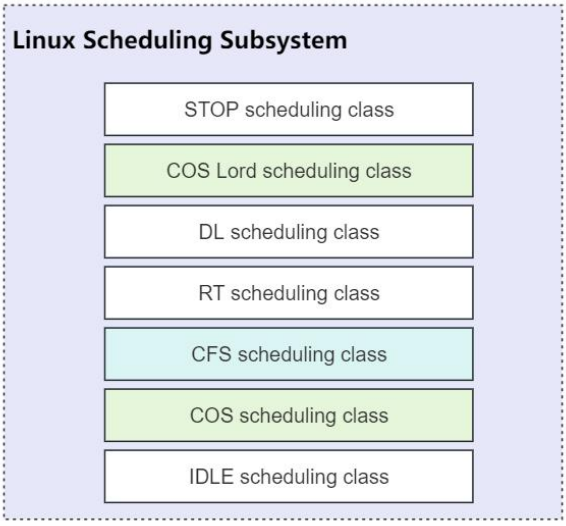
系统调用	含义
set_lord()	设置调用者线程为 Lord 线程

create_mq()	创建消息队列
init_shoot()	初始化 shoot area
shoot_task()	将用户态调度策略部署到内核

COS 用户态调度框架主要分为以下几个模块：

- 内核态：COS调度类

COS在内核中新增一个调度类，类似常用的CFS调度类。这个调度类向用户态提供一组系统调用来供用户态定义任意调度策略，同时负责执行用户态传入的调度策略。



COS调度类优先级位于CFS之下，目的是不影响CFS的稳定运行，若在CFS之上，可能会造成CFS线程饿死，值得注意的是负责维持着内核的稳定性内核线程[12]都是CFS调度类，如果这些线程得不到及时运行，会导致内核崩溃。

Lord线程调度类优先级则高于CFS等调度类，仅次于STOP调度类，这确保它能够一直运行在它绑定的CPU上，实时做出调度决策，避免因为被其他线程抢占而导致暂时得不到运行而导致调度时延。

- 用户态：Lord线程

Lord线程位于用户态，负责部署调度算法传入内核。

[12] 内核线程（*kernel-level threads*）：是工作在内核空间的，不属于任何一个进程，如软中断处理线程 ksoftirq、OOM 处理线程、磁盘缓存写回线程。

用户程序可以通过set_lord()系统调用将自己绑定在指定CPU上，开启内核COS调度，成为Lord线程。只有Lord线程才有调度其他COS调度类线程（下称COS线程）的特权。同一个主机上只能有一个Lord线程，若已经有其他线程设置自己为Lord线程，则该系统调用会返回错误。

- 内核态到用户态信息传递：消息队列

内核会将重要的消息通过消息队列传递到Lord线程，下表是COS消息的类型：

消息类型	含义
MSG_TASK_NEW	一个新的线程以可运行状态进入 COS 调度类
MSG_TASK_NEW_BLOCKED	一个新的线程以不可运行状态进入 COS 调度类
MSG_TASK_RUNNABLE	COS 调度类线程变为可运行状态
MSG_TASK_BLOCKED	COS 调度类线程变为阻塞状态
MSG_TASK_DEAD	COS 调度类线程死亡
MSG_TASK_PREEMPT	COS 调度类线程被其他非 COS 调度类线程抢占
MSG_TASK_COS_PREEMPT	COS 调度类线程被其他 COS 调度类线程抢占

消息的作用是防止Lord做出错误的调度策略。例如，一个COS线程状态变为阻塞，但是Lord线程做出了调度该线程到CPU1号上的决策，这是不被允许的。

同时，内核必须保证用户态Lord线程消费消息的实时性。每条消息都有一个seq num，来检测Lord每次做出调度策略的时候，是否已经消费完最新的消息。内核通过维护下一个要分配的seq num：a，与Lord每次做出调度决策时传入的已消费消息的最大seq num：b对比，若b不等于a + 1，意味着Lord还有消息没有消费，内核会拒绝本次调度决策。

只有Lord线程才有权力调用create_mq()系统调用创建消息队列，该系统调用只能被单个Lord线程调用一次。

- 用户态到内核态消息传递：shoot系统

shoot系统负责将用户态的调度策略传递到内核态，分为shoot area区域和

shoot_task()系统调用两部分。

shoot area负责存储一次部署中用户态调度策略传入内核的参数，例如将指定线程ID调度到指定CPU、Lord线程消费到的最大seq num等。只有Lord线程才有权力调用init_shoot()系统调用创建shoot area区域，该系统调用只能被单个Lord线程调用一次。

shoot_task()系统调用将一次调度策略传入内核，是整个COS中最核心的系统调用接口，正是这个系统调用，赋予了用户态做出调度决策并且部署到内核的权力，实现了用户态决定调度策略。

例如，Lord线程想将COS线程1调度到CPU1号上，COS线程2调度到CPU3号上，Lord线程将相关参数写入地址shoot area，该地址是专门存放shoot参数的区域，然后再将要shoot的CPU号通过shoot系统调用参数传入到内核。内核在检查通过之后（线程id具备运行条件、seq num最新、CPU号合法等），进行线程调度。

线程调度分为两种：

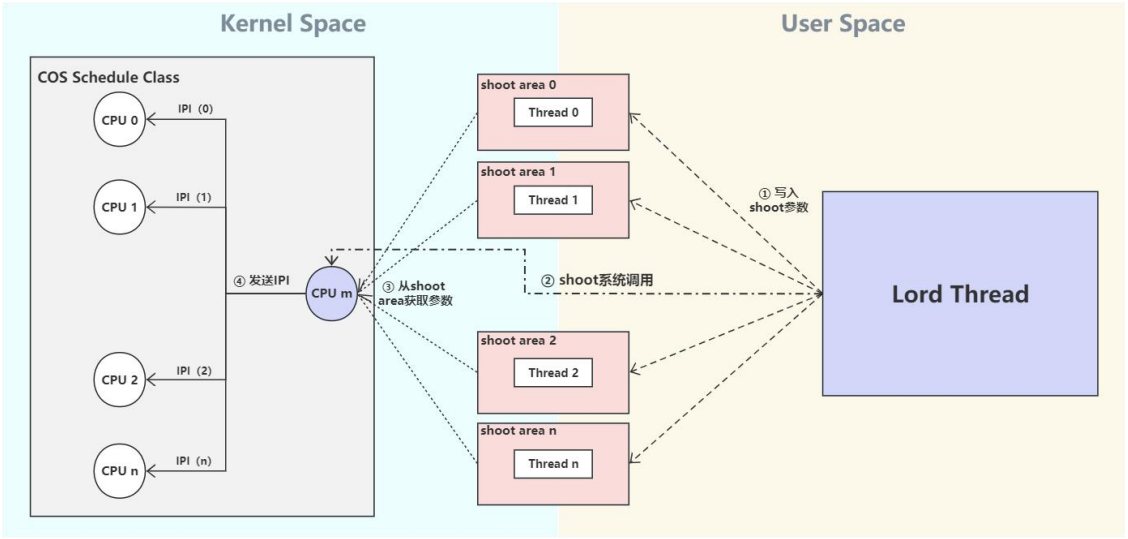
1. 本地调度

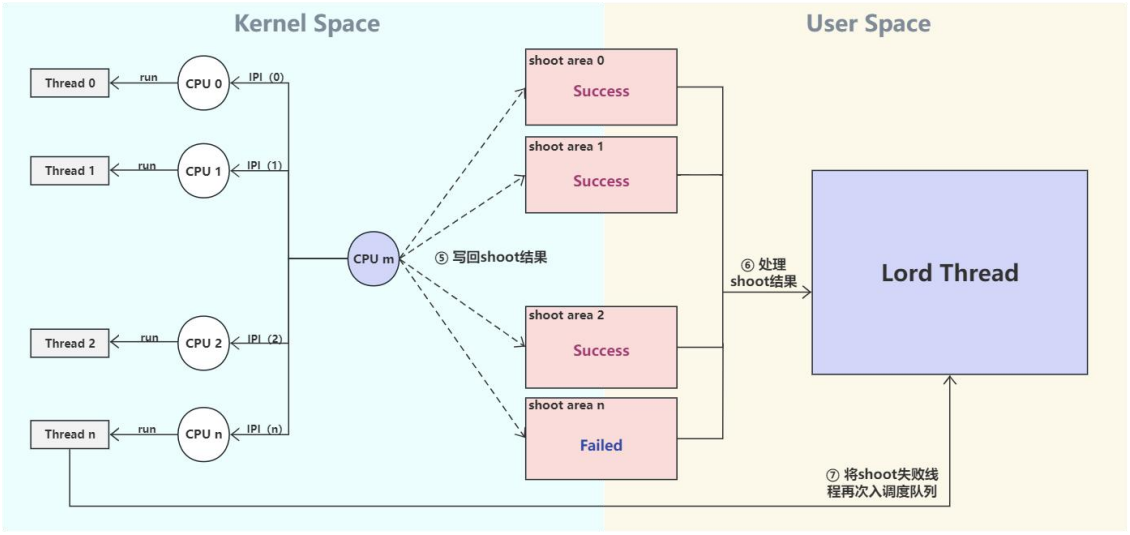
要调度线程需要运行的CPU和Lord在同一个CPU上，此时，Lord会暂时让出自己的CPU给此COS线程运行；

2. 远程调度

要调度线程需要运行的CPU和Lord不在同一个CPU上，此时则需要硬件的支持，这是软硬件协同的一个场景，通过IPI来发送中断给对应的CPU，让其执行调度函数，调度本次部署的线程。在IPI中断处理函数中，目标CPU检测到本次的中断类型为resched，则执行调度函数调度目标线程。

Lord线程部署一次调度策略流程图：



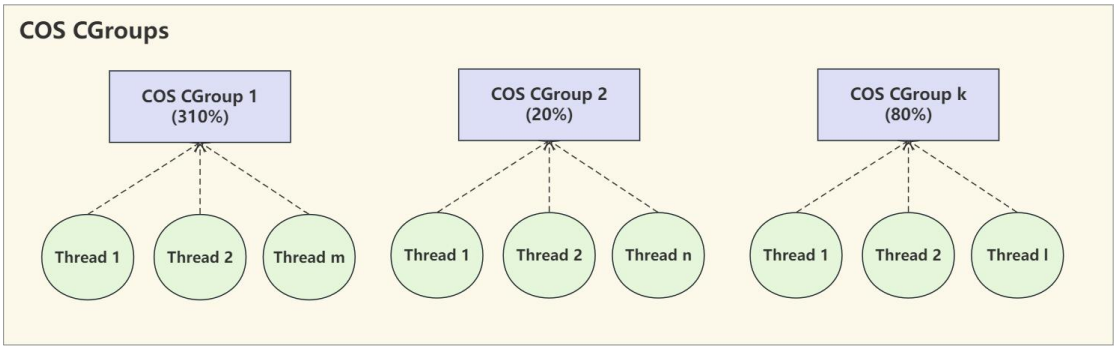


- CPU cgroup兼容

为了支持云原生场景，契合题意（赛题要求支持cgroup），COS支持cgroup对CPU使用率的限制。COS额外提供了四个系统调用。

系统调用	含义
coscg_create()	创建一个 cgroup
coscg_ctl()	将线程加入或移除 cgroup
coscg_rate()	设置 cgroup 的 CPU 使用率
coscg_delete()	删除一个 cgroup

COS负载线程可以随意将自己添加入一个cgroup，或者删除。同时也可以调整cgroup的CPU使用率，如80%，意味着当前cgroup中的所有线程对CPU的使用率最大值为80%，例如在1ms内最多只能获取0.8ms的运行时间。这意味着COS能够支持云原生场景中的容器。



2.2.2 高性能

前面我们阐述了 EXT 内核的痛点，我们在 COS 内核中完美地解决了这些问题，

- 消息队列高性能通信

用户态与内核态之间通信有很多种方式：信号、eBPF注入、系统调用、共享内存。在其中，性能最高效的，就是内核与用户态的共享内存。共享内存能够使得内核消息传递无需阻塞，同时用户态的Lord线程也能够异步处理消息，内核只需要维护seq num来保证Lord消费消息的实时性即可。

内核态用户态共享内存使得用户读取消息的时候无需陷入内核，陷入内核是一个开销十分大的操作，陷入内核的上下文切换，L1 L2 Cache刷新失效，TLB刷新失效，会极大地影响整体性能。

- shoot area高性能

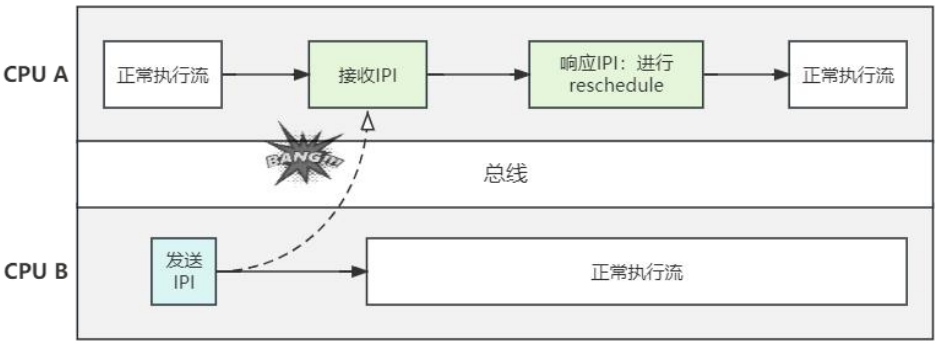
每次执行shoot部署Lord的调度决策时候，需要传递本次要调度的线程和目标CPU，以及其他的辅助参数。这些内存在用户态和内核态两个地址空间来回拷贝，开销大。

同时，shoot的返回值也很复杂，可能因为cgroup，或者线程状态检测不通过，并且还可能会出现部分线程调度成功，部分失败。

为了解决这种痛点，我们选择为每一个CPU核维护一个shoot area，来存储shoot需要传递的参数，和本次shoot的返回值情况。这减少了shoot时延，同时，也简化了shoot系统调用的定义。

- 使用中断加速调度策略执行

在Lord做出调度决策调度到目标CPU的时候，EXT内核是等待目标CPU自己主动去调度，这在io密集场景，线程从阻塞态恢复需要立马得到运行的场景，时延会很高。而COS内核改进了这一缺陷，使用硬件支持的IPI，给目标CPU核心发送中断，强制其立刻执行调度。这能够做到次微秒（μs）级的时延，而对于EXT和CFS，这往往需要毫秒（ms）甚至秒（s）级的时延。

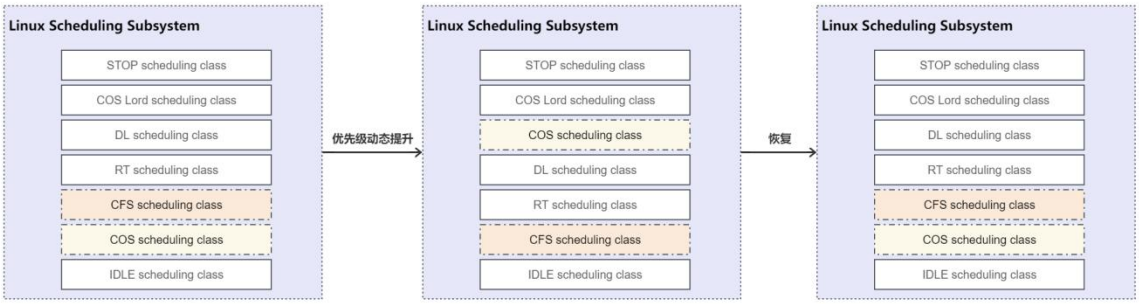


- 自适应调度优先级调整

上文提到过，为了确保系统稳定，COS调度类优先级低于CFS调度类，但是这样的话，会大大增加调度时延，因为在Linux内核中，我们调用schedule调度函数的时候，它会按优先级从高到低挑选各个调度类的就绪线程。那么就算之前IPI发送速度再快，它在调度schedule函数调度的时候，由于内核中CFS线程占绝大多数，COS优先级低于CFS，那么很大概率会导致本次触发schedule调度上CPU的线程不是我们本次shoot的COS线程，而是CFS线程，等待CFS线程运行结束后，才会调度COS线程，这会大大增加调度时延。

为了避免这种情况，COS实现了优先级动态提升的机制，在一个COS线程被调度到目标CPU上时候，此时我们将其优先级短暂地提升到最高，之后当目标CPU调用schedule函数挑选其上CPU执行的时候，在将其优先级下降到CFS之下，这样既兼

顾了COS线程调度到上CPU的时延保持很低的值，也维护了系统的稳定性，让CFS线程不饿死。



3 系统实现

3.1 EXT

下文将以 Shinjuku 调度算法实现为例，介绍我们基于 EXT 开发的用户态 lib 框架。

3.1.1 初赛基于 EXT-userland 的 Shinjuku 实现

本部分将以基于 EXT-userland、参照 ghOSt 的 Shinjuku 实现为例，简要介绍我们初赛基于 EXT 内核搭建起的 EXT-userland 用户态调度框架。

- 调度流程

由于 Shinjuku 算法基于 centralized model，故而会有一个专门的 Agent 线程不断循环执行调度流程。具体调度流程如下：

```
// 核心调度流程
static void sched_main_loop(void) {
  while (!exit_req) {
    update_shm(); // 集中刷新共享内存
    consume_message(); // 消耗消息队列
    shinjuku_rq.PickNextShinjukuTask(); // 进行调度决策
    drain_enqueued_map(); // 从eBPF端获取需要调度的线程
  }
}
```

- 消息队列

在 userland 模型中，为了实现 Shinjuku 算法，我们需要通过消息队列从 eBPF 程序获取一些诸如线程开始运行、线程停止运行的调度消息。每当特定事件发生，eBPF

程序就会向消息队列发送一个指定类型的消息。用户态会定期消费消息队列，根据消息的类型来做出不同的处理。

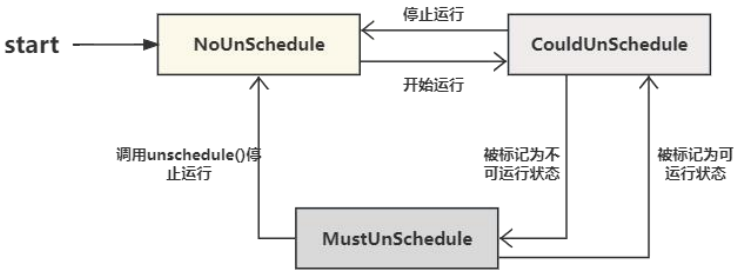
- 抢占机制

我们在 enqueued_task 结构体中记录线程上次被调度的时间戳 last_ran。在每次进行 PickNextTask 调度决策中，我们通过从 eBPF 程序中获取的 CPU 信息来轮询正在运行的线程，检测其运行时间。当线程的已运行时间超过时间片长度，并且调度队列中有其他线程等待被调度执行，该线程就会通过 unschedule_task 函数被放入抢占队列 preempted，然后在 eBPF 程序中被真正抢占。最后，在用户态中接收其真正被抢占的消息，进行一定的后处理。

- 进程通信

负载程序可以随时通过共享内存将线程的运行状态标记为可运行（*Runnable*）或者不可运行（*Idle*）。初始时，线程都为 Idle 状态。只有当线程被标记为 Runnable 状态时，线程才能被调度运行；而若一个 Runnable 状态的线程被标记为 Idle 状态，线程就必须立刻停止运行或被从调度队列中移除。

在具体实现中，我们使用以哈希表形式管理的共享内存来实现进程通信，参照了 ghOSt 的 Shinjuku 实现引入了状态机，选择以集中刷新的方式定时轮询共享内存，根据状态转移进行相应的处理。具体的状态转移图如下：



3.1.2 决赛基于 EXT-central 的 Shinjuku 实现

本部分将以基于 EXT-eBPF 的 Shinjuku 实现为例，介绍我们决赛基于 EXT 内核进行的进一步开发。

- eBPF端

  eBPF端负责调度算法的实现，通过eBPF map与用户态的Agent端通信。

  - 调度队列

  我们使用 DSQ 组成我们的多级双端队列数据结构。

  同时我们修改了 EXT 内核，添加了两个 helper 函数：

函数名称	含义
scx_bpf_dsq_peek_pid	获取 DSQ 队头线程的 pid
scx_bpf_dsq_put_prev_peek	将 DSQ 队头线程出队，并且再次放入 DSQ 队尾

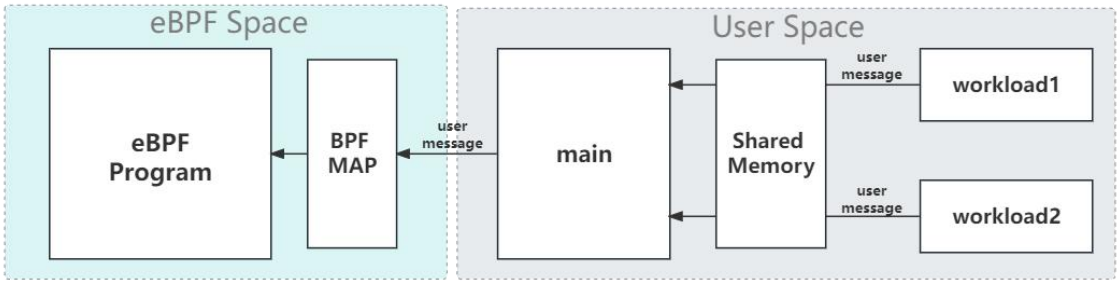
  - 抢占机制

我们在 bpf_task_ctx 结构体中记录线程上次被调度的时间戳 last_ran，每次线程触发 running 事件时更新时间戳。在每次触发 dispatch 事件进行调度决策之前，通过维护的 CPU 信息来遍历正在运行的线程，若时间片耗尽，则可以抢占。

```
// 简化版代码实现
bpf_repeat(16) {  // 遍历所有CPU
    struct cpu_stat* cs = bpf_map_lookup_elem(&cpulist, &cpu);
    if (!cs || cs->idle || !cs->availible)  continue; // 如果CPU不能被抢占（CPU空闲或运行CFS线程）则跳过本轮循环

    int curr_task_pid = cs->pid; // 获取CPU当前正在运行任务的pid及其上下文
    struct bpf_task_ctx* bpf_sw = bpf_map_lookup_elem(&bpf_task_sw, &curr_task_pid);

    if ((long)(bpf_sw->last_ran + preemption_time_slice) - (long)(now) < 0) { // 如果时间片用完，则进行抢占
        scx_bpf_kick_cpu(cpu, SCX_KICK_PREEMPT);
        cs->pid = 0;
    }
}
```

- Agent端

为了使得负载编写更为简洁，我们在 eBPF 端和负载端之间插入了一层用户态 main 进程作为二者通信的媒介。负载将用户态信息通过共享内存传递给 main 进程，main 进程通过 eBPF MAP 将用户态信息传递给 eBPF 程序，如下图所示：



- 代理端与负载线程的通信

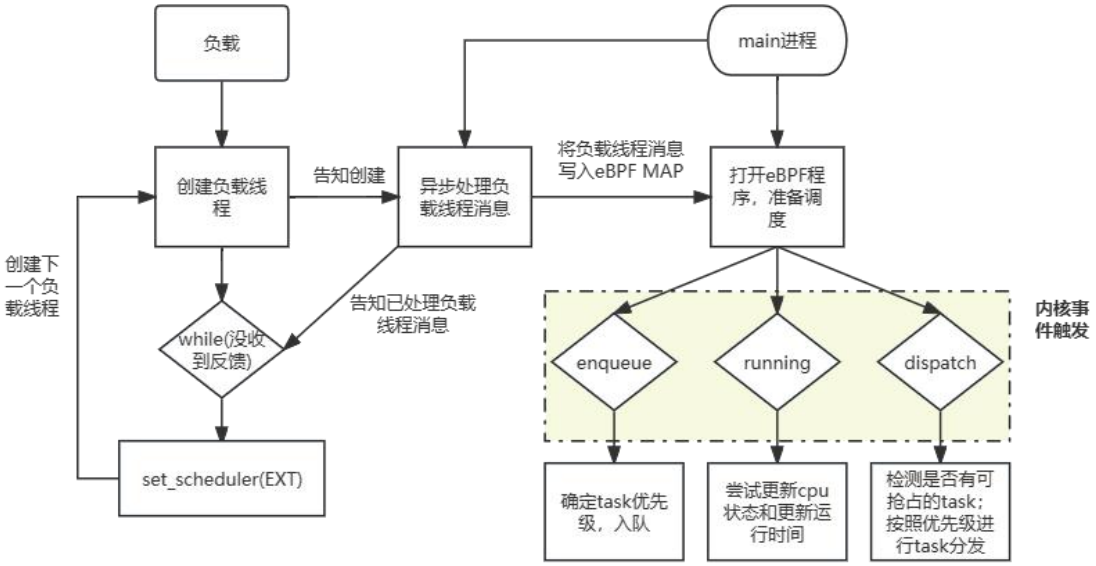
在 EXT-eBPF 中，我们摒弃了初赛的复杂实现，采用消息队列的方法来实现与用户态进程的通信

```
MessageQueue<RPC> writer("/ext/reader", 4096, true); // 创建写回消息队列
MessageQueue<ThreadInfo> reader("/ext/writer", 4096, false); // 创建读取消息队列

while (!exit) {
  ThreadInfo new_thread_info = reader.Pop(); // 从消息队列中读取最新的一条消息
  int pid = new_thread_info.pid; // 获取发送消息的负载线程的pid
  int e = bpf_map_update_elem(task_sw_fd, &pid, &new_thread_info.task_sw,
                              BPF_ANY); // 将负载线程消息同步更新到eBPF MAP
  RPC rpc = {
      .over = true, // 用于写回告知负载线程已经处理了消息，让负载线程将自身调度类设置成EXT
  };
  writer.Push(&rpc); // 将rpc消息压入写回消息队列中
}
```

- 调度流程

以一个线程的生命周期为线索，我们实现的 Shinjuku 调度算法基本流程如下：

调度流程图：



3.2 COS 系统实现

在这一节，我们介绍 COS 在 Linux 内核以及用户态的具体实现。

3.2.1 COS 调度类

- 回调函数

在 Linux 中，每一个调度类都需要实现调度类的回调函数，以下是 Linux 调度类的部分定义：

```
struct sched_class {
    // 线程入队操作
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    // 线程出队操作
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    // 挑选要运行的线程
    struct task_struct *(*pick_next_task)(struct rq *rq);
    // 为线程挑选就绪队列
    int  (*select_task_rq)(struct task_struct *p, int task_cpu, int flags);
    // 线程被唤醒
    void (*task_woken)(struct rq *this_rq, struct task_struct *task);
    // 线程调用fork创建子线程
    void (*task_fork)(struct task_struct *p);
    // 线程死亡
    void (*task_dead)(struct task_struct *p);
};
```

在 COS 中，我们新增了两个调度类 COS 与 COS Lord。COS 负载线程和 Lord 线程的调度类都是 COS 调度类。

- pick_next_task

内核在执行调度函数 schedule 的时候，会按调度类优先级从高到低调用各个调度类的 pick_next_task 函数，若其返回的线程不为空，则调度该线程。

COS 的实现会直接返回 next_task 字段代表的线程。在 Lord 部署调度策略的时候，会将目标 CPU 的调度队列的 next_task 字段设置为目标线程，从而达到在下一次调用调度函数 schedule 的时候，直接返回该线程去运行。

- dequeue_task

内核在将线程出队的时候会调用该函数。

COS 会检测本次出队原因是否是因为阻塞，若是则会拉高 is_blocked 标记，在下文中会讲述到这最终会导致线程阻塞的消息发送。

- task_woken

内核在线程被唤醒时会调用该函数。

COS 线程被唤醒，即状态变为可运行，此时需要发送线程可运行消息。

- task_dead

内核在线程死亡会调用该回调函数。

COS 直接发送线程死亡消息。

- Lord 线程的调度类

上文我们说到，Lord 线程调度类同样也是 COS，在系统设计一章中我们又提到说 Lord 调度类为 COS Lord。这里其实 Lord 是假借 COS Lord 调度类的优先级，实际 Lord 调度类为 COS。COS Lord 调度类仅仅实现了 pick_next_task 函数，该函数是 COS Lord 调度类实现的唯一一个回调函数。该回调函数实现机理就是直接挑选 Lord 线程作为要运行的线程。这样虽然 Lord 线程调度类为 COS，但是其实际被挑选运行的优先级

则是 COS Lord 的优先级。

之所以不将 Lord 线程调度类设置为 COS Lord 调度类，是因为 Lord 线程所属的调度类同样需要实现 COS 调度类的其他回调函数，那么在两个调度类中实现几乎一样的回调函数，这不太符合代码设计理念，同时 COS Lord 调度类还负责下文动态优先级变化的实现，所以我们采用了现在的设计策略。

- 自适应调度优先级

实现同样是靠 COS Lord 调度类。在 Lord 线程将调度策略传入内核时，将变量 is_shoot_first 拉高，设为高电平，当内核去按优先级遍历调度类时，调用 COS Lord 调度类的 pick_next_task 函数。若 Lord 不存在且 is_shoot_first 为高电平，则返回 COS 调度类 next_task 字段代表的线程，这是为优先级提升，同时将该电平拉低，是为优先级下降。

3.2.2 Lord

Lord 线程具备部署调度策略、创建消息队列，创建 shoot area 的权力。线程可以调用 set_lord(int cpu_id)系统调用将当前线程设置为 Lord 线程。同一台主机上，只能有一个 Lord 线程。

下面是 set_lord 系统调用的底层实现伪代码：

```
int cos_do_set_lord(int cpu_id)
{
    cos_move2target_rq(cpu_id); // 将当前线程迁移到指定CPU

    if (cos_on) return -EINVAL; // 如果当前已经存在Lord线程，返回错误

    spin_lock_irqsave(&cos_global_lock, flags); // 加锁

    sched_setscheduler(p, SCHED_COS, &param); // 设置当前调度类为COS

    open_cos(rq, p, cpu_id); // 设置当前线程为lord

    spin_unlock_irqrestore(&cos_global_lock, flags); // 解锁
}

```

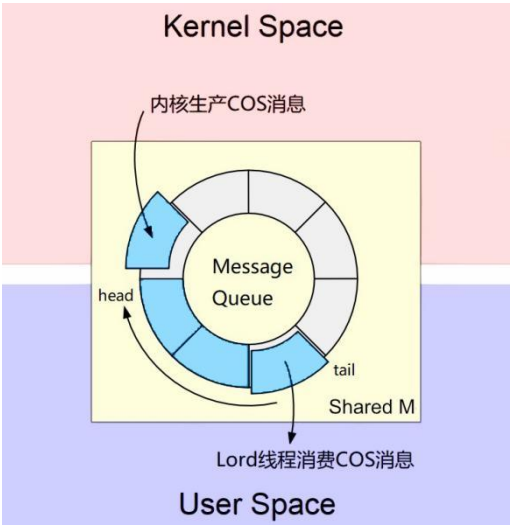
而在用户态库中，Lord 被抽象出为一个基类：

```
class Lord {
    // Lord基类构造函数
    Lord(int lord_cpu) {
        // 设置当前线程为Lord线程，绑定在lord_cpu中运行
        set_lord(lord_cpu);
        // 调用系统调用新建消息队列
        mq_ = new MessageQueue;
        // 调用系统调用初始化shoot area
        sa_ = new ShootArea;
    }

    // 调度器函数，供特定调度算法继承实现，如fifo
    virtual void schedule() = 0;

    // 消费不同类型消息的函数，供特定调度算法继承实现
    virtual void consume_msg_task_runnable(cos_msg msg) = 0;
    virtual void consume_msg_task_blocked(cos_msg msg) = 0;
    virtual void consume_msg_task_new(cos_msg msg) = 0;
    virtual void consume_msg_task_new_blocked(cos_msg msg) = 0;
    virtual void consume_msg_task_dead(cos_msg msg) = 0;
    virtual void consume_msg_task_preempt(cos_msg msg) = 0;
    virtual void consume_msg_task_preempt_cos(cos_msg msg) = 0;
};
```

程序员可以根据自己想要实现的调度算法去继承此调度类，重写消费各类消息函数和调度函数来实现自己的调度算法。

3.2.3 消息队列

消息队列由 Lord 线程创建，生命周期随 Lord 线程消亡而消亡。消息队列是基于内核态用户态共享内存实现的 ring buffer，类似于内核的 io_uring 缓冲区。



消息和消息队列具体定义如下：

```c
struct cos_msg {
    u_int16_t pid; //线程id
    u_int16_t type; //消息类型
    u_int32_t seq; //消息seq num
};

struct cos_message_queue {
    u_int32_t head; //ring头部
    u_int32_t tail; // ring尾部
    struct cos_msg data[_MQ_SIZE]; //消息队列
};
```

- 消息队列创建

对于用户态内核态共享内存（消息队列，shoot area），我们重写其File结构体的mmap回调函数，将其mmap映射和返回地址为private_data。来达到用户内核共享内存。

```c
static int queue_mmap(struct file *file, struct vm_area_struct *vma)
{
    // 取出文件private_data作为映射内存
    struct cos_message_queue *mq = file->private_data;
    // 对mmap参数进行校验
    _cos_mmap_common(mq, sizeof(mq));
    // 对私有地址进行mmap映射，作为mmap返回地址
    remap_vmalloc_range(vma, mq, 0);
}
```

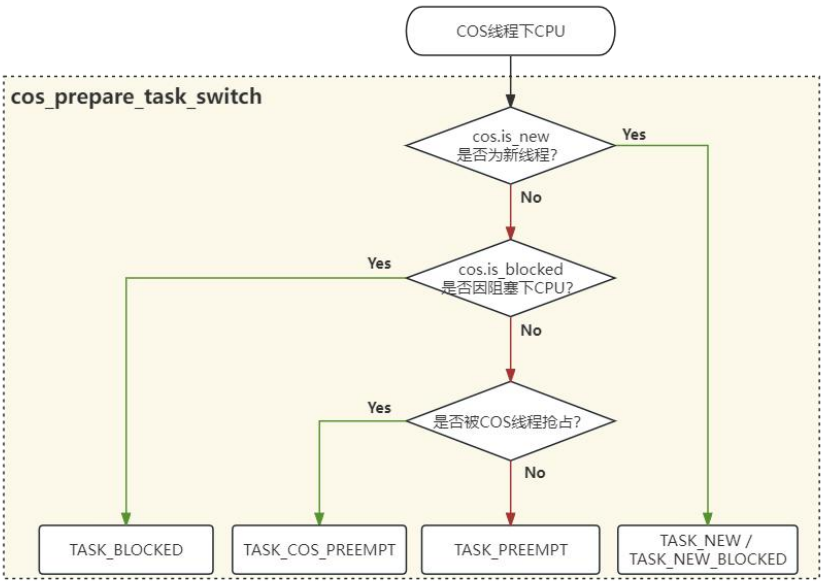
消息队列创建使用系统调用create_mq()，该系统调用创建成功后，返回一个代表该消息队列的文件描述符。

```c
int cos_create_queue(void)
{
    // 分配一块内核态和用户态都能使用的内存
    global_mq = vmalloc_user(sizeof(struct cos_message_queue));
    // 新建文件，将该地址传入作为private_data
    int fd = anon_inode_getfd("[cos_queue]", &queue_fops, global_mq, O_RDWR | O_CLOEXEC);
    return fd;
}
```

用户态库对消息队列进行了封装。将其抽象出一个类。

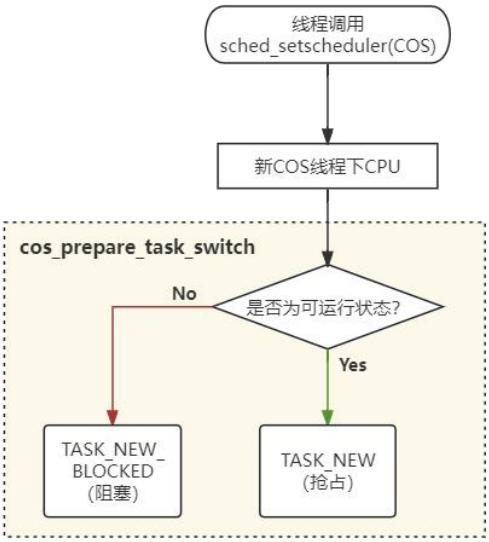
```
class MessageQueue {
public:
    MessageQueue() {
        // 创建消息队列
        mq_fd_ = create_mq();
        // mmap映射出共享内存
        mq_ = static_cast<cos_message_queue*>
          (mmap(nullptr, _MQ_MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, mq_fd_, 0));
    }
    int mq_fd_;
    cos_message_queue * mq_;
};
```

- 内核消息发送时机



  - 新建COS线程

COS调度类不支持调度类遗传，即一个线程新建时候它的调度类不可能是COS。线程只能调用sched_setscheduler系统调用将自己调度类设置为COS。此时，COS线程还运行在CPU上，如果立马发送TASK_NEW消息，Lord线程在消费到这条消息后做出调度该线程到指定CPU上的决策，会导致该线程同时在两个CPU核上运行。所以等到新建的COS线程下CPU的时候，才能发送消息。



若新建的线程被抢占，状态仍然为可运行的时候，发送TASK_NEW消息，代表线程被创建，并且状态为可运行；若新建的线程被阻塞，状态为不可运行的时候，发送TASK_NEW_BLOCKED消息，代表线程被新建，但是状态为阻塞。

具体而言，在线程设置自己调度类为COS的时候，设置标志位task_new为1。在Linux中的prepare_context_switch函数中，比较当前被替换下CPU的线程prev，若其调度类为COS，task_new为1，并且其状态为可运行，发送TASK_NEW消息，状态为不可运行，则发送TASK_NEW_BLOCKED消息。

- COS线程状态变为可运行

当线程状态变为可运行的时候，如从睡眠中被唤醒、阻塞等待的资源被分配等情况，发送TASK_RUNNABLE消息。

- COS线程状态变为不可运行

当线程状态变为不可运行的时候，如睡眠、获取某些暂时得不到分配的资源的时候，需要阻塞线程，同时标记task_blocked为1，发送TASK_BLOCKED消息。

- COS线程被抢占

由于Linux中，线程被抢占的场景有很多，并且很复杂。所以在这里我们采取了排除法来判断线程是否因被抢占而下CPU。

在Linux中的prepare_context_switch函数中，比较当前被替换下CPU的线程prev和上CPU的线程next，若prev调度类为COS，并且其task_blocked和task_new字段均为0，则代表其被抢占。检查next的调度类，若其调度类为非COS，则代表其不是被COS线程抢占，发送TASK_PREEMPT消息。否则是被COS线程抢占，发送TASK_COS_PREEMPT消息。

- COS线程死亡

  发送TASK_DEAD消息，代表线程死亡。

- 消息队列生产与消费

消息队列读写是一个多生产者单消费者的场景，内核中的所有CPU核心是生产者，内核的多个CPU核心上的COS线程状态发生变化（新建，阻塞，可运行，被抢占，死亡等）时，该核心均会通过消息队列发送消息至用户态，此时，需要互斥锁与内存屏障来保障多线程安全。

```
1 int produce(struct cos_msg *msg)
2 {
3     spin_lock_irqsave(&cos_mq_lock, flags);
4     defer spin_unlock_irqrestore(&cos_mq_lock, flags);
5 
6     if (global_mq->head - global_mq->tail >= _MQ_SIZE) return -EINVAL;
7 
8     // 分配seq num
9     msg->seq = global_seq;
10    global_seq++;
11    if (global_seq >= _MAX_SEQ_NUM) return -EINVAL;
12 
13    global_mq->data[global_mq->head % _MQ_SIZE] = *msg;
14    // 内存屏障，防止第13行代码先于第10行代码执行。
15    smp_wmb();
16    global_mq->head++;
17 }
```

而消费者方则是 Lord 线程，通过调用上面 MessageQueue 类的 consume 方法，采用内存屏障来保证读取的多线程安全性，但是无需加锁。

```
cos_msg consume() {
    if (empty())   return {}; // queue empty
    // 消费
    cos_msg ret = mq_->data[mq_->tail % _MQ_DATA_SIZE];
    // 内存屏障，防止第7行代码先于第4行执行
    smp_mb();
    mq_->tail++;
    return ret;
}

```

- 消息序列号 seq num

每一条消息都有其序列号，内核通过该序列号来保障Lord消费消息的实时性，每次Lord部署调度策略shoot入内核时，内核均会检测其消费的最新seq num和内核已经生产的最新seq num是否一致，若不一致，则会返回错误。

3.2.4 shoot 系统

shoot_task(size_t cpusetsize, cpu_set_t *mask)系统调用是整个 COS 系统最核心的系统调用，正是该系统调用，赋予了用户态决定调度策略的权力。

下面是 Lord 线程一次 shoot 流程伪代码：

```
// 初始化shoot area区域
sys_init_shoot_area();
while (true) {
    // 消费完所有消息
    consume_all_message();
    // 将shoot参数填入shoot area
    fill_args_to_shoot_area();
    // 调用shoot系统调用部署调度策略
    sys_shoot_task(cpumask, sizeof(cpu_mask));
    // 处理shoot结果
    check_and_solve_shoot_error();
}
```

- shoot area

在shoot之前，首先需要通过init_shoot()系统调用来初始化shoot area区域，shoot area原理与消息队列一样，基于内核态用户态共享内存，在此展示shoot area结构体：

```c
struct cos_shoot_arg {
    u_int32_t pid; // 本次shoot的线程id
    u_int32_t info; // shoot结果
};

struct cos_shoot_area {
    u_int64_t seq; //Lord消费的最大seq num
    struct cos_shoot_arg area[_SHOOT_AREA_SIZE]; // shoot area，下标为对应CPU ID
};
```

在执行shoot前，通过cpuid确定下标后填入目标线程id和最大seq num。

- shoot_task系统调用

  标记目标CPU于cpumask中，调用shoot系统调用，下面是shoot系统调用底层实现：

```c
int cos_do_shoot_task(cpumask_var_t shoot_mask)
{
    // 若消费消息不实时，返回
    if (global_seq - 1 != task_shoot_area->seq) return -EINVAL;
    // 遍历目标CPU
    for_each_cpu(cpu_id, shoot_mask) {
        // 设置被调度的线程为目标线程
        set_pick_next(p);
        // 若目标CPU为Lord线程所在的cpu
        if (cpu_id == lord_cpu) need_local_shoot = true;
        else __cpumask_set_cpu(cpu_id, ipimask);
    }
    // 对目标CPU发送IPI
    cos_remote_shoot(ipimask);
    // Lord让出CPU，给目标线程运行
    if (need_local_shoot) cos_local_shoot();
}
```

在IPI中断发送至指定的CPU核后，该CPU保存当前上下文，陷入中断处理函数，检测本次IPI传递的信息为resched，即触发调度函数，便调用schedule函数进行线程调度。按优先级从高到低依次调用各个调度类的pick_next_task()函数选择，若返回的线程结构体不为空，则调度之。

```
void handle_irq() {
    // 若触发中断的理由是ipi resched
    if (ipi_flag && resched_flag) {
        // 按优先级遍历所有调度类
        for each sched_class {
            thread = sched_class->pick_next_task();
            // 调度线程
            if (thread) schedule(thread);
        }
    }
}
```

shoot_task系统调用结束返回后，Lord检测shoot area对应的返回结果，如果有错误，则进行相关的处理（如将失败的线程重新放回就绪队列）。

3.2.5 CPU cgroup 兼容

除上述功能外，COS 还兼容 cgroup。本质上，cgroup 是通过控制一组 cgroup 中线程对 cpu 的使用率。

- COS cgroup数据结构

  在Linux中，我们使用以下结构体表示一个cgroup：

```
struct cos_cgroup {
    u_int64_t coscg_id;
    // cpu使用率
    int rate;
    // 当前周期剩余运行时间
    int64_t salary;
    // 当前cgroup中所有线程
    struct list_head task_list;
};
```

- 定时器补充运行时间

  我们在Linux内核中注册了一个定时器，该定时器每隔周期T便会遍历所有的cgroup，根据其rate补充其salary。

  具体而言，是将其salary重置。salary的计算公式如下：

$$\text{salary} = \frac{\text{T} \times \text{rate}}{100}, \quad 0 < rate < core\_num \times 100$$

其中core_num为核心数。

定时器注册：

```
static int coscg_timer_init(void)
{
    // 定时器初始化
    hrtimer_init(&coscg_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
    // 定时器开启
    hrtimer_start(&coscg_timer, kt, HRTIMER_MODE_REL);
    // 设置定时器回调函数
    coscg_timer.function = htimer_handler;
}
```

- COS线程运行时间统计

当COS线程换上CPU时，会开始运行时间的统计，当换下CPU时，会停止计时，并将其时间从其对应的cgroup的salary字段中减去。

当一个cgroup的salary小于0的时候，其中的线程便不能被调度到CPU上运行。用户通过系统调用设置rate值便达到了控制CPU使用率的目的。

4 性能测试

4.1 硬件配置与环境搭建

- 硬件配置

我们在开发流程中使用的是Linux物理机，具体硬件配置如下：

CPU: Intel(R) Core(TM) i7-8700 CPU @ 3 with hyper-threading

Memory: 14GB

OS: Ubuntu 22.04.3 LTS

with Linux kernel 6.4.0+(COS) && 6.4.0-rc3+(EXT) && 5.11.0+(ghOSt)

- 环境搭建

环境配置教程
性能测试教程

4.2 Task Delegation 时延

4.2.1 测试方案

- 测试对象

  我们选取 COS、ghOSt、EXT 进行测试。

- 测试机理

  task delegation latency，即调度算法部署时延，准确的说，是用户态做出将线程 T 运行在目标 CPU 的决定开始，到线程 T 运行上目标 CPU 而结束。具体而言分为四个时间点：

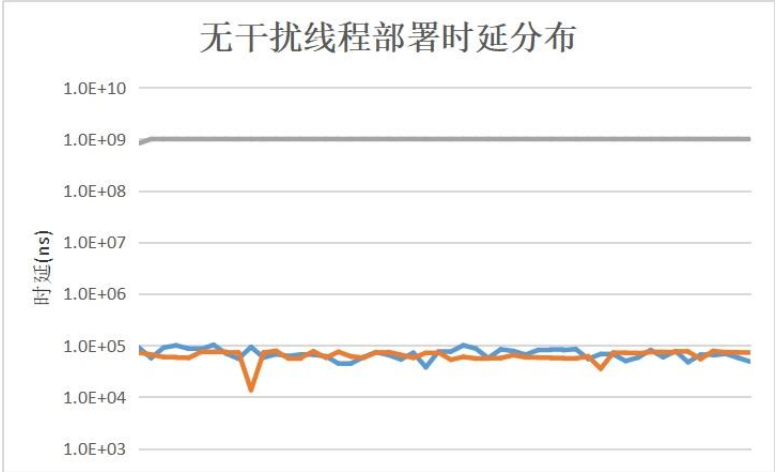
  1. 用户态做出调度策略的决定 a
  2. 陷入内核开始执行具体策略实施 b
  3. 策略实施结束，目标线程 T 准备上 CPU 运行 c
  4. 目标线程 T 恢复到用户态开始执行 d

- 负载设置

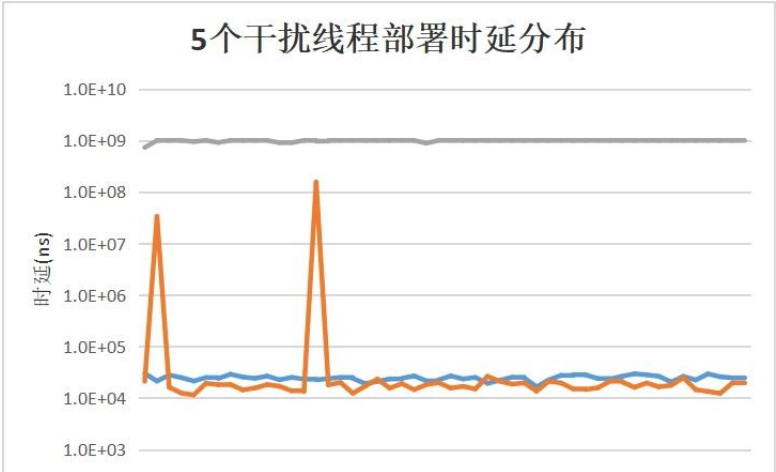
  分为 0、5、6、7、8、9 个 CFS 干扰线程 6 种情况，分别测试时延 d - a 的值 50 次。

4.2.2 数据处理

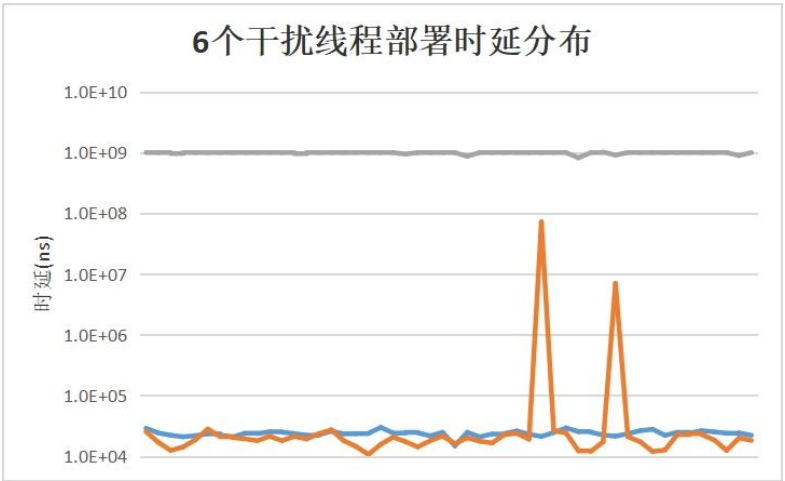
- 0负载线程


无干扰线程部署时延分布
1.0E+10
1.0E+09
1.0E+08
1.0E+07
1.0E+06
1.0E+05
1.0E+04
1.0E+03
时延(ns)

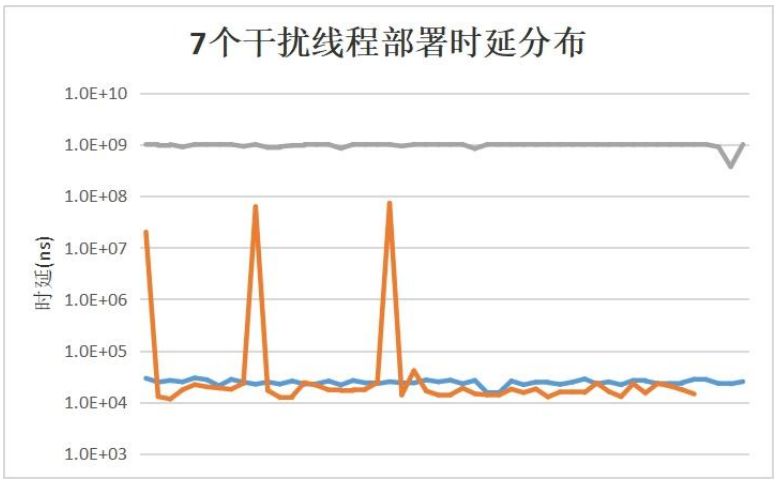
- 5负载线程


5个干扰线程部署时延分布
1.0E+10
1.0E+09
1.0E+08
1.0E+07
1.0E+06
1.0E+05
1.0E+04
1.0E+03
时延(ns)

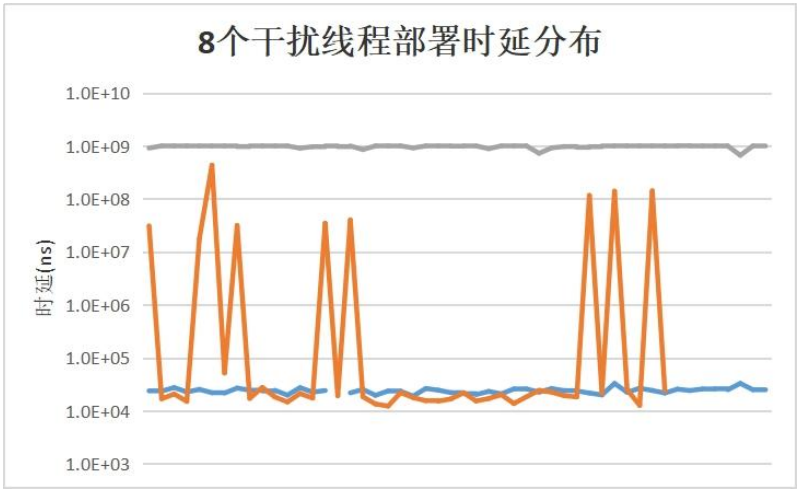
- 6负载线程


6个干扰线程部署时延分布
1.0E+10
1.0E+09
1.0E+08
1.0E+07
1.0E+06
1.0E+05
1.0E+04
时延(ns)

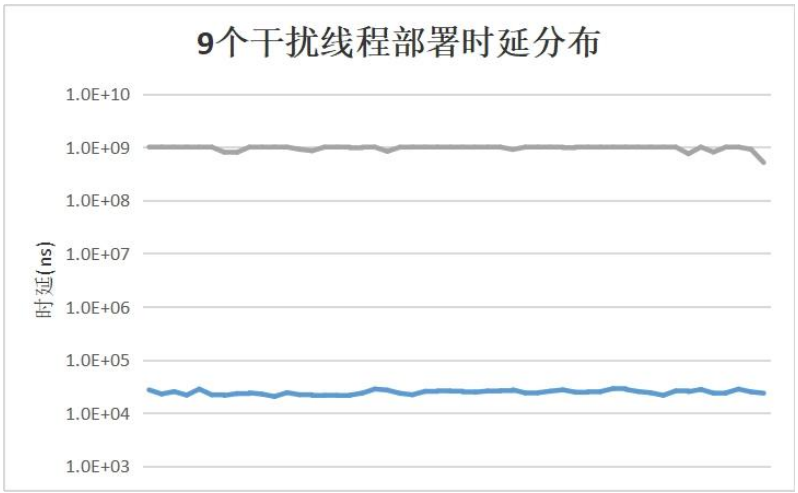
- 7负载线程



- 8负载线程

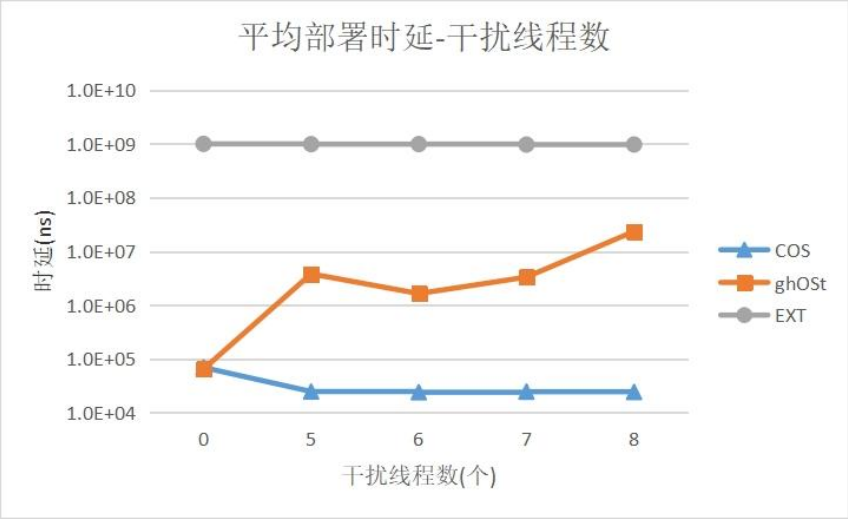


- 9负载线程

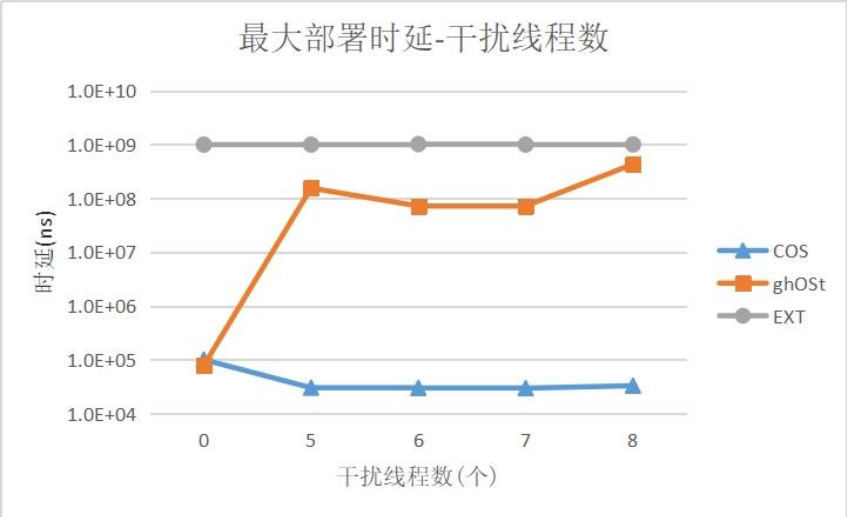


ghOSt在运行9个CFS干扰线程的情况下直接崩溃，故此处没有记录结果。

- 平均时延与干扰线程数关系



- 最大时延与干扰线程数关系



4.2.3 结果分析

- 数据描述

在无干扰线程的情况下，ghOSt 和 COS 的部署时延均较为稳定地维持在 10us~100us 区间内，EXT 的部署时延维持在 100ms~1s 区间内。

在有干扰线程的情况下，随着干扰线程数量增大，ghOSt 出现部署时延突增（大于 1ms）的情况发生频率越大，当干扰线程数量为 9 时甚至直接崩溃；而 COS 的部署时延仍然稳定维持在 10us~100us 区间，不受干扰线程影响；EXT 的部署时延仍然

维持在 100ms~1s 区间内。

- 结果分析

无论是在无负载和有负载的情况下，ghOSt 和 COS 性能优于 EXT。这是因为前二者是基于 IPI 硬件支持的核间中断立刻调度目标线程，而 EXT 则是等待目标 CPU 主动去调度，导致耗时增大。

当存在干扰线程时，ghOSt 的部署时延的不稳定性增强；但 COS 却不会受到这种影响，这是因为 ghOSt 没有 COS 类似的优先级动态调整机制。ghOSt 线程优先级低于 CFS，在 CFS 干扰线程运行的情况下，ghOSt 调度的线程很可能会被 CFS 直接拦截，造成时延急剧升高，甚至可能出现系统崩溃；但是 COS 优先级动态提升使得其不会被 CFS 拦截，大大提升性能，同时优先级恢复又维护了系统稳定性，二者兼得。

- 总结

综上所述，task delegation 时延在无 CFS 线程干扰情况下，ghOSt 和 COS 性能差距不大，EXT 性能远差于 ghOSt 和 COS。在有 CFS 线程干扰情况下，COS 的部署时延仍然稳定维持在 us 级；ghOSt 的平均时延和稳定性都差于 COS，在 CFS 线程数多的情况下甚至不能启动；EXT 性能仍然差于 ghOSt 和 COS。

4.3 cgroup 有效性测试

4.3.1 测试方案

- 测试对象

为了体现 COS 支持 cgroup 而 ghOSt 不支持 cgroup，我们选取 COS、ghOSt 作为比较对象。

- 测试机理

CPU cgroup 支持对线程的 CPU 使用率进行限制，如 50%代表该 cgroup 中的线程

只能使用 50%的 CPU 带宽。cgroup 是容器技术（docker）的基石，往往有以下两种场景：支持 docker 容器，在操作系统层级进行虚拟化；在程序评测系统等需要执行远端传入的代码场景，将远端代码放入容器沙盒中执行防止其对 CPU 等资源的破坏与大肆消耗。

由于支持 docker 运行工作量很大（一个大厂相关团队耗费数月），在这里我们仅仅对第二种场景进行模拟，验证 COS 支持的 cgroup 确实能起到沙盒作用，限制其他线程的 CPU 使用率，使其不会妨碍工作线程的正常执行。以下是测试场景：

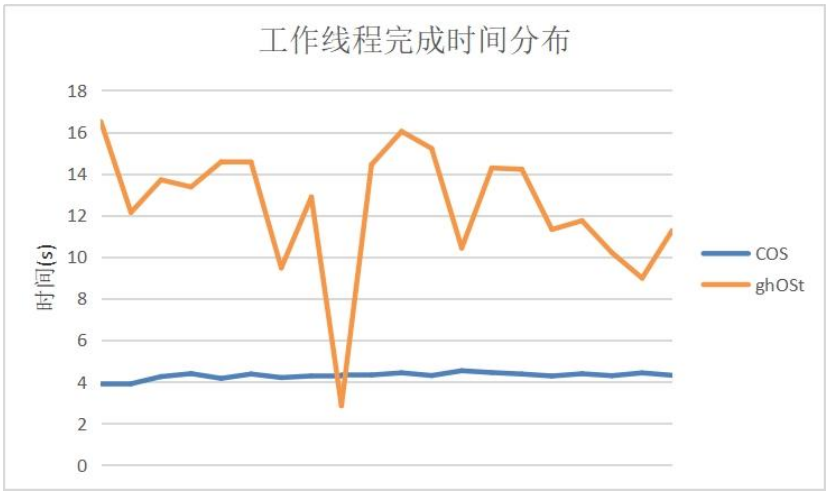
COS 与 ghOSt 均运行代码评测程序，评测程序由一个服务器工作线程和 40 个远端传入的待评测程序组成。COS 将 40 个远端传入的待评测程序放入 cgroup，限制 CPU 使用率 600%。

- 负载设置

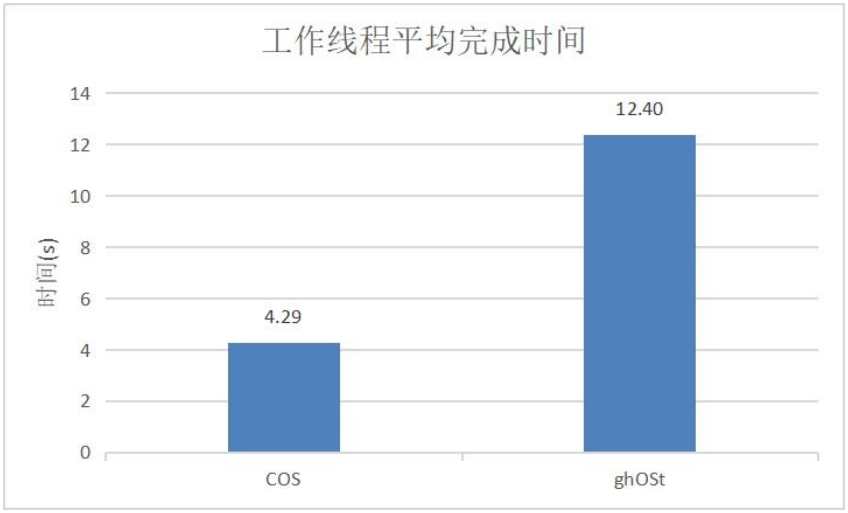
运行代码评测程序，评测程序由一个服务器工作线程和 40 个远端传入的待评测程序组成。测试 20 次。COS 将干扰线程放入 cgroup 中，对 CPU 带宽限制为 600%，ghOSt 不支持 cgroup，不做设置。最终以工作线程运行完成所需时间作为性能评估标准。

4.3.2 数据处理

- 工作线程完成时间



- 工作线程平均完成时间



4.3.3 结果分析

- 数据描述

整体看，20 次测试中，除了第 9 次，其余 COS 的工作线程运行时间均低于 ghOSt。COS 数值很稳定地维持在 4s 出头，而 ghOSt 则在 8~18s 区间波动。

对于平均完成时间，COS 是 4.29s，而 ghOSt 是 12.40s，COS 大约是 ghOSt 时间的三分之一。

- 结果分析

我们可以看到，COS 工作线程完成时间大约是 ghOSt 三分之一，这是因为 COS 使用 cgroup 将 COS 干扰线程放入沙盒中，限制其 CPU 使用率为 600%，这使得工作线程能够较为充分地得到 CPU 带宽去运行而不被干扰；而对于 ghOSt，由于其不支持 cgroup，所以其工作线程必须和干扰线程一起竞争 CPU 带宽，导致完成时间高于 COS。

- 总结

综上所述，COS 支持的 cgroup 能够起到沙盒作用，成功限制 cgroup 中线程的

CPU 带宽，而 ghOSt 不支持该功能，这也是 COS 的优势。

4.4 io 密集型场景 RocksDB

4.4.1 测试方案

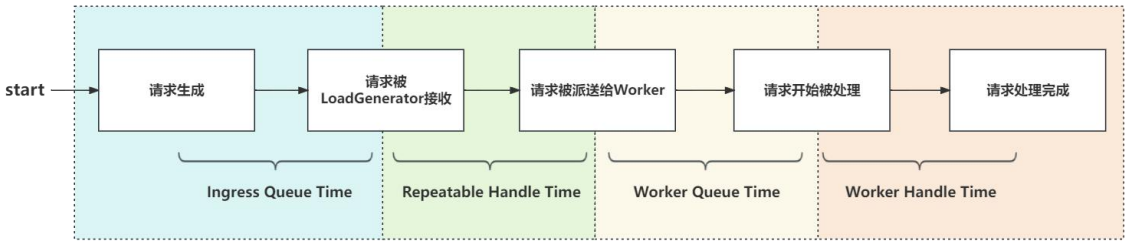
为了更好地与 ghOSt 调度框架进行性能对比，我们将 ghOSt 提供的 Shinjuku 调度算法实现以及 RocksDB 负载移植到了 EXT 框架和我们自己实现的 COS 用户态调度框架上。

- 测试对象

  我们选取 COS、ghOSt、EXT、CFS 四个作为比较对象

- 测试机理

  我们测试 RocksDB 查询请求生成到处理结束的时间，该时间可以分位四个部分：



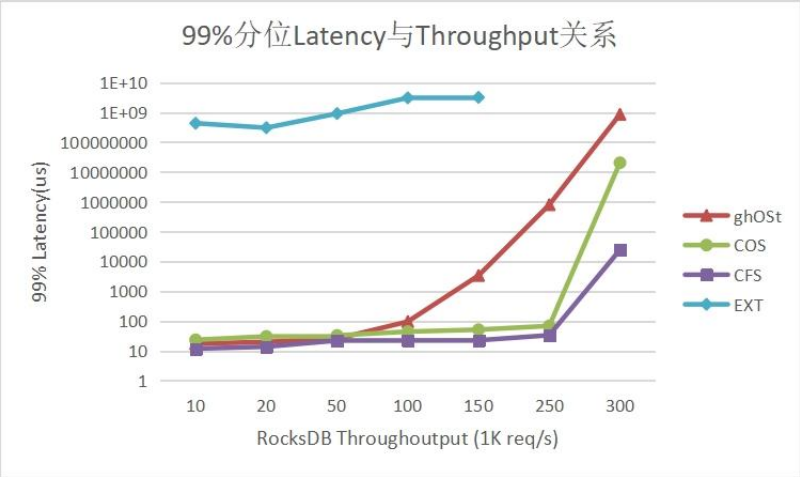
其中请求生成线程负责生成 RocksDB 查询请求，该生成采用泊松分布模拟现实生产场景。请求生成线程绑定在一个核上运行。COS、ghOSt、EXT 运行 Shinjuku 调度算法，该调度算法在提升尾时延方面有很好效果。COS、ghOSt、EXT 运行调度 100 个 worker 线程处理 RocksDB 请求。对于 CFS，则固定运行 9 个 worker 线程，并且绑定在 9 个核上运行，这是为了最大化利用其性能，若 worker 线程过多，会造成很多上下文切换。性能不如 9 个线程绑核。同时为了模拟真实生产场景，我们运行了 7 个无关的 CFS 线程。

- 负载设置

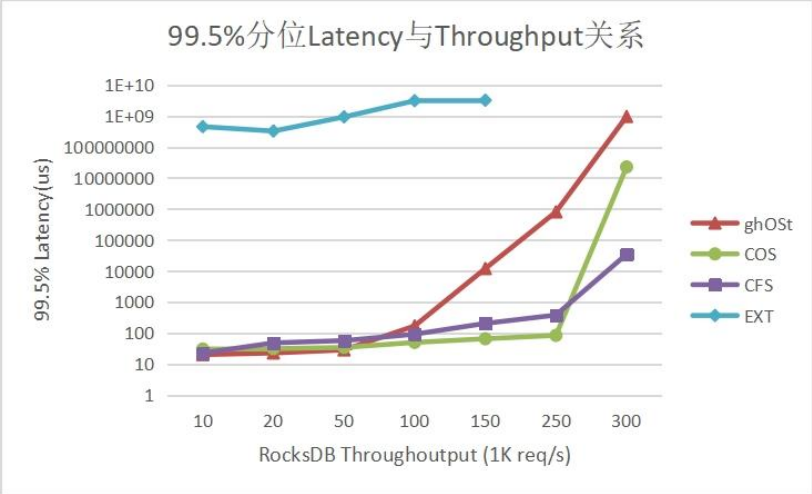
COS、ghOSt、EXT 均使用 9 个处理器核心运行 100 个 RocksDB worker 线程，一个处理器运行用户态调度器线程（Lord、agent），一个处理器运行 RocksDB 请求生成线程，绑核运行。而 CFS 则 9 个处理器运行 9 个 RocksDB worker 线程，绑核运行。一个处理器运行 RocksDB 请求生成线程，绑核运行。测试其 10k、20k、50k、100k、150k、250k（qps）吞吐量与 50%、99%、99.5%、99.9%、100%分位时延（latency），其中重点是测试尾时延（tail latency）。测试时间为 5s。

4.4.2 数据处理

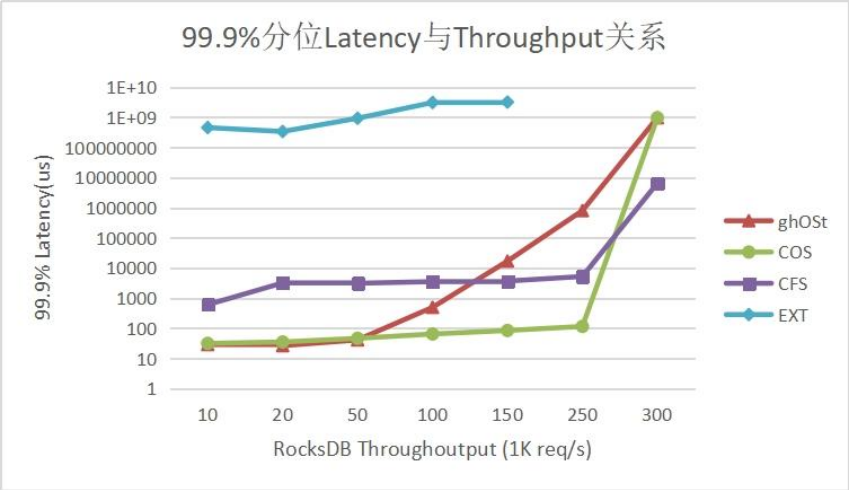
- 99%时延



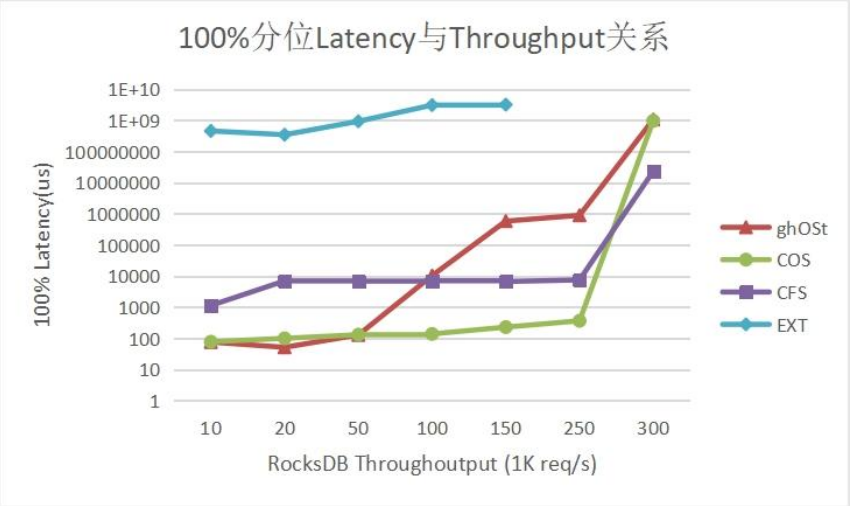
- 99.5%时延



- 99.9%时延



- 最大时延



4.4.3 结果分析

- 数据描述

99%分位情况下，随着吞吐量增大，ghOSt 时延也开始增大，以 100k 为转折点。前面稳定在 10-100us，后面急剧上升。而 COS 和 CFS 仍然维持在 10-100us，但是 COS 略高于 CFS。250k 后，二者均开始急剧上升。EXT 则完全和前三者不在一个层次，时延稳定 100ms - 10s。

99.5%分位情况下和 99%有点类似，唯一区别是 COS 略低于 CFS。

99.9%分位和 100%情况相似，CFS 开始明显高于 COS 与 ghOSt，最后稳定在 1-10ms，而 ghOSt 以 50k 为转折点，前期低于 100us，50k 之后开始上升，在 100k-150k 间超过 CFS。而 COS 最大一直稳定在 100-1000us。很稳定。超过 250k 后，COS 与

CFS 均急剧上升。

- 结果分析

  综合看来，EXT 和后面三者完全不能相提并论。最后甚至崩溃。

  99%分位下，ghOSt 性能变差开始，这是因为 ghOSt 没有优先级动态提升，导致可以利用的 CPU 可能被其他 CFS 占用导致尾时延上升。而越到后面的百分位，CFS 表现随吞吐量增大也开始急剧下滑，而 COS 则一直稳如泰山，这是 Shinjuku 调度算法的优势，在 IO 密集型场景极大地提升尾时延，同时 COS 还有上面提到的四大高性能，使得其表现很优异。

- 总结

  综上所述，在 250k 吞吐量内，随着吞吐量的增大，COS 的尾时延远远低于 ghOSt 和 CFS，稳定在 10-100us 间。250k 后均急剧上升。

5 未来展望

接下来可以在以下方面进一步完善 COS 项目：

- COS 内核目前还不太稳定，如在 vmvare 虚拟机上会导致禁用 CPU，未来我们会修复，提升其稳定性
- COS cgroup 目前使用系统调用实现，未来会将其使用 Linux 中文件系统替换
- 在 Lord 遍历调度的时候配合 eBPF，实现性能优化
- 基于 COS 搭建更多的负载，如基于 COS 的突发实例实现
- 尝试将 COS 部署在工业界

6 参考文献

[1] Humphries J T, Natu N, Chaugule A, et al. ghost: Fast & flexible user-space delegation of linux scheduling[C]//Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021: 588-604.

[2] sched_ext 项目 github 地址

[3] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for second-scale tail latency. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 345–360, Boston, MA, February 2019. USENIX Association.