华中科技大学

全国大学生计算机系统能力大赛
操作系统功能赛道全国赛报告

面向轻量级虚拟化的优化技术

队长姓名 郑云鲲

队员姓名 郑云鲲、刘佳璇、贾竟一

指导老师 黄卓、樊浩

2024 年 7 月 30 日

摘　要

容器技术作为一种操作系统虚拟化技术，因其轻量化的特点，在云计算领域得到广泛应用。然而，容器技术的轻量化是基于对宿主机内核的共享实现，这也限制了容器对内存管理策略的个性化定制，而且其对主机 CPU 调度系统的复用会导致严重的应用性能损失。共享的宿主机统一内存管理策略无法满足不同容器应用在不同场景下对透明大页和虚拟内存管理策略的定制化需求，从而严重了影响容器应用性能。调度系统中单一默认的策略和参数配置无法满足应用的多样化调度需求，也会对容器内应用造成严重的性能损失。

针对容器共享内核的根本问题，我们首先提出了面向容器的虚拟内核框架。普通容器所有操作直接与宿主内核交互，而在引入虚拟内核后，容器直接与自身对应的虚拟内核交互，消除部分容器对宿主机内核的依赖，从而实现容器自身的定制化策略。它在操作系统中引入一个新的抽象层——虚拟内核层，将传统操作系统内核虚拟为多个个性化虚拟内核。虚拟内核可以根据不同的应用需求、环境特征等进行灵活定制，满足云计算环境对内核隔离性和多样性的需求。

同时我们提出了基于虚拟内核的面向容器的 CPU 调度虚拟化方案和容器内存管理策略定制化方法，在保证容器轻量化特性的前提下，支持容器 CPU 调度的虚拟化和 CPU 调度的虚拟化内存管理策略的定制化。在面向容器的 CPU 调度虚拟化方案中，我们在宿主机 CPU 调度器上为每个容器虚拟化一个独立的、可供配置的 CPU 调度器，其中用全局调度器和局部调度器来区分描述宿主机上和容器内的 CPU 调度环境。针对不同场景下容器应用内存管理策略定制化需求，我们在虚拟内核框架中基于完全隔离原则和投票机制设计了容器透明大页和虚拟内存策略定制化方法。

在容器 CPU 调度系统和内存管理策略的虚拟化的基础上，我们还设计了容器的系统调用的虚拟化，基于 inode 虚拟化的文件访问控制模块等，旨在真正意义上地实现容器**轻量级**、**可定制**、**安全**和**高效**的目标。

关键词：虚拟化、云计算、容器、CPU 调度、内存管理

目　录

CHAPTER 1

项目背景及意义

1.1 容器技术概述

容器技术作为一种操作系统级虚拟化技术，在云计算中带来了重大的变革。相比传统的虚拟机技术，容器通过共享主机操作系统的方式实现了轻量化和快速启动的特性，从而可以快速部署容器化应用程序，与硬件虚拟化相比，容器的内核虚拟化是一种更加轻量级的虚拟化方法，具有明显的性能优势。可以看到，与传统操作系统虚拟化相比，内核虚拟化提出和设计了新颖的虚拟内核技术体系，具有隔离性强、灵活性高、适应性广的特点，可对操作系统内核架构产生深远影响。

同时，容器技术提高了应用的可移植性和可扩展性，为云环境中的应用提供了一种标准化的交付模式，使得应用开发、测试、部署和运行更加便捷。在当前的云计算背景下，容器技术已经成为开发者和运维人员必不可少的工具之一。

然而，只依赖内核提供的机制支持容器化应用间的物理资源限制，无法实现完全的隔离，容易产生 CPU 资源争用，内存资源泄露等问题。2019 年发现的“Dirty Cow”漏洞（CVE-2016-5195）利用了 Linux 内核的一个竞争条件漏洞，允许本地用户提升权限。攻击者可以从一个容器中逃逸并获得宿主机的控制权，从而影响所有容器的安全。与此同时，为了保证运行过程中的安全性和隔离性，通常情况下仅允许其内应用使用单一策略且不可配置参数，无法满足应用的多样化调度需求，这会对容器内应用造成严重的性能损失。可以说，共享内核一方面为容器带来高效性，但是另外一方面也给容器带来了诸多的安全隐患和性能损耗。这些问题成为制约容器发展和应用的主要技术瓶颈。

为了应对这些挑战，并进一步提升容器的性能和灵活性，我们采用了加载内核模块的技术，将容器与主机内核解耦，让容器在不损失性能的情况下，摆脱对部分内核的依赖。

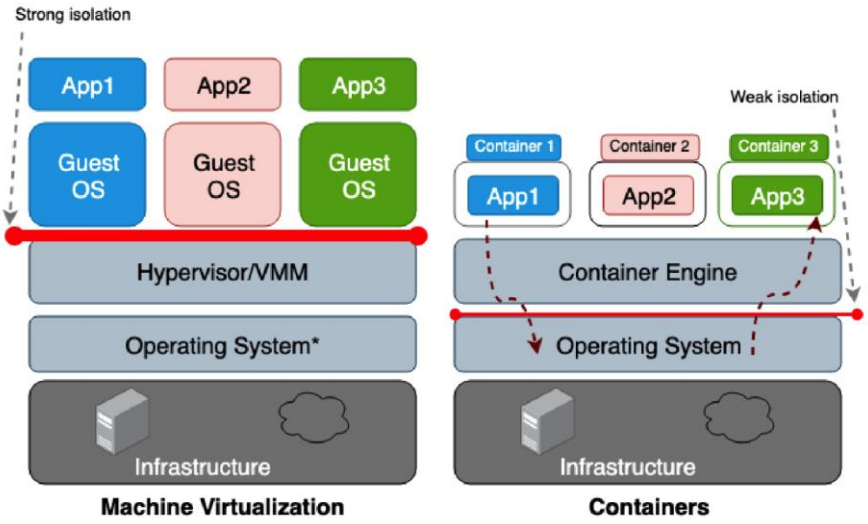

图 1.1　虚拟机与容器架构图

本项目针对性地对于 CPU 调度和内存管理实现了对容器内存管理策略的个性化定制。定制化 CPU 调度方案通过为每个容器提供独立的、可配置的 CPU 调度器，解决了因共享主机调度器而导致的性能瓶颈问题。在这个方案中，容器的调度系统被分为全局调度器和局部调度器两层。全局调度器负责宿主机上的调度，使用默认的完全公平调度器（CFS）来保证容器间的安全性和隔离性；局部调度器则为容器内的任务提供独立的调度方式和参数视图，从而满足容器内应用的个性化需求 。

定制化 CPU 调度方案通过为每个容器提供独立的、可配置的 CPU 调度器，解决了因共享主机调度器而导致的性能瓶颈问题。在这个方案中，容器的调度系统被分为全局调度器和局部调度器两层。全局调度器负责宿主机上的调度，使用默认的完全公平调度器（CFS）来保证容器间的安全性和隔离性；局部调度器则为容器内的任务提供独立的调度方式和参数视图，从而满足容器内应用的个性化需求 。

为每个容器提供独立的可配置调度器和内存管理策略，使得容器可以根据自身特点，选择最合适的策略，解决了共享内核带来的性能损耗和安全隐患问题。从而在保障容器的轻量化特性的同时，也满足了不同应用场景下的个性化需求。

1.2 存在的问题

容器轻量级的本质在于其共享主机内核，而传统虚拟机则在硬件级别进行虚拟化。无疑会造成很大的性能开销。而容器通过利用操作系统内核提供的命名空间（Namespace）和控制组（cgroup）技术，实现了进程的隔离和资源的管理。控制组机制通过将进程分层组来达到限制和监控各类资源的目的，而命名空间技术通过控制资源的可见性，为不同用户提供独立的资源视图，从而实现资源的隔离。

1.2.1 CPU 调度系统

而对于 CPU 调度系统，应用可以通过配置调度系统中的策略和参数进行个性化配置，而容器虚拟化环境中应用的个性化配置受到了限制，这是因为容器通常仅使用 CFS 调度器运行并禁止了对参数的配置。

CFS 调度器单一的策略无法满足应用多样化的调度需求。应用可以根据其特征或运行环境对策略进行配置来获得更好的执行效率。对容器内应用进行策略配置通常需要提升容器的权限，无论是特权容器还是具有 CAP_SYS_NICE 能力的容器都将带来严重的安全威胁。同时策略配置将影响主机操作系统上其它任务的正常运行，破坏了容器的隔离性。而目前还没有解决单一策略下造成性能损失和提升权限带来的安全性和隔离性问题的良好解决方案。

不可配置且共享的参数无法满足应用多样化的调度需求。应用可以根据具体运行过程中的需求对内核调度参数进行个性化配置，以获得更好的执行效率或达到特殊的调度需求。然而，在容器虚拟化环境中，容器对参数的配置同样需要提升容器的权限，而权限提升将带来安全威胁。此外，容器对参数的配置将作用到整个系统，而不同应用对参数的需求不同，这种全局性的参数配置可能会对其他容器造成性能损失，这破坏了容器的隔离性。而目前并没有针对容器环境的参数配置研究，因此也没有良好的方案来解决容器上参数的配置将破坏安全性和隔离性的问题。

1.2.2 容器内存管理

但是，memory cgroup 机制只能支持对物理内存使用量的限制，容器应用分配的物理内存、虚拟内存对应的管理策略都依赖宿主机内核的策略，容器只能使用内存资源，无法定制化自身的内存管理策略，从而严重限制容器应用性能的发挥。详细来说，共享宿主内核的容器内存管理存在以下两个问题：

（1）无法适应不同容器应用对物理内存的不同使用模式。由于不同的容器应用具有不同的物理内存使用需求，例如访存密集型应用可以通过透明大页机制显著提高性能，而延迟敏感型应用则倾向于禁用透明大页机制，避免引起高延迟峰值。然而，容器技术的底层隔离实现只提供物理内存使用量的审计和限制机制，物理内存管理策略则依赖于宿主内核的配置策略，无法支持应用个性化定制的物理内存透明大页策略。

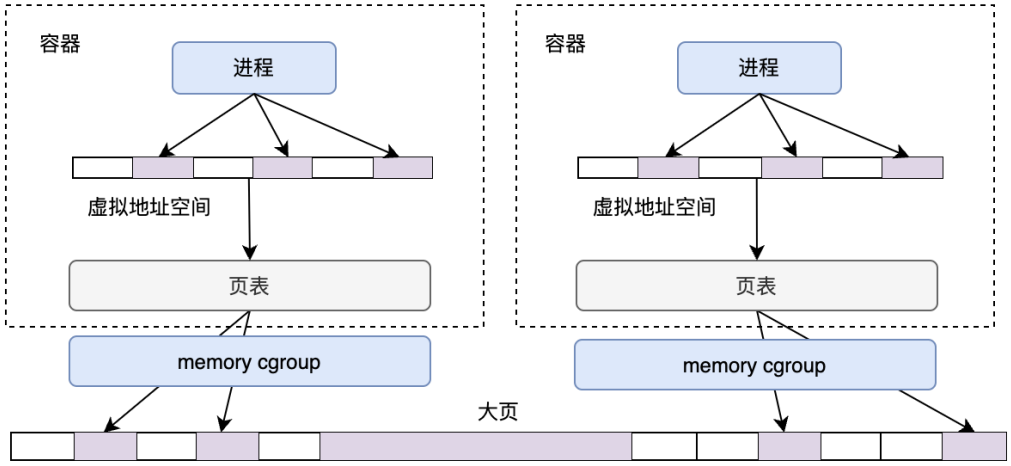

图 1.2 操作系统内存管理架构

（2）无法实现虚拟内存的配置和资源隔离。每个进程拥有独立的虚拟内存空间，但操作系统虚拟内存审计机制是针对全局的所有进程。在多租户容器应用场景下，全局虚拟内存超配策略与容器私有需求之间可能存在不一致，不同应用对虚拟内存的个性化配置策略需求也无法得到满足，也无法与 memory cgroup 的物理内存限制协同管理容器自身内存。例如，Redis 与 Postgresql 的官方手册分别建议设置不同的虚拟内存超配策略。传统的容器技术由于共享宿主内核的限制，无法支持容器个性化定制虚拟内存超配策略，从而严重影响容器应用的性能和稳定性。

1.3 国内外研究现状

容器隔离最近引起了工业界和学术界的广泛关注，不仅因为安全性，还因为容器之间的性能干扰日益受到关注。针对容器共享宿主内核导致的隔离性不足、无法个性化定制内核机制的问题，近年来出现的解决方案总的来说有如下方向：

（1）用户级内核隔离：用户级别的内核隔离将对主机内核的请求重定向到在用户级别实现的特定于应用程序的内核。重定向用户级请求的关键和开销的主要来源是拦截对主机内核的请求。例如 gVisor，拦截应用程序系统调用以创建系统接口，类似于主机内核，无需硬件虚拟化。然而，请求拦截不可避免地会导致过多的上下文切换，从而带来巨大的开销。

（2）基于 VM 的内核隔离：比较典型的是基于虚拟机的安全容器方案 Kata，Kata 容器为每个容器运行私有的轻量内核，并通过硬件虚拟化提供容器隔离。但它需要虚拟机监视器（VMM）将虚拟化硬件公开给 guest kernel，与原生容器相比仍然会产生不可忽略的开销。

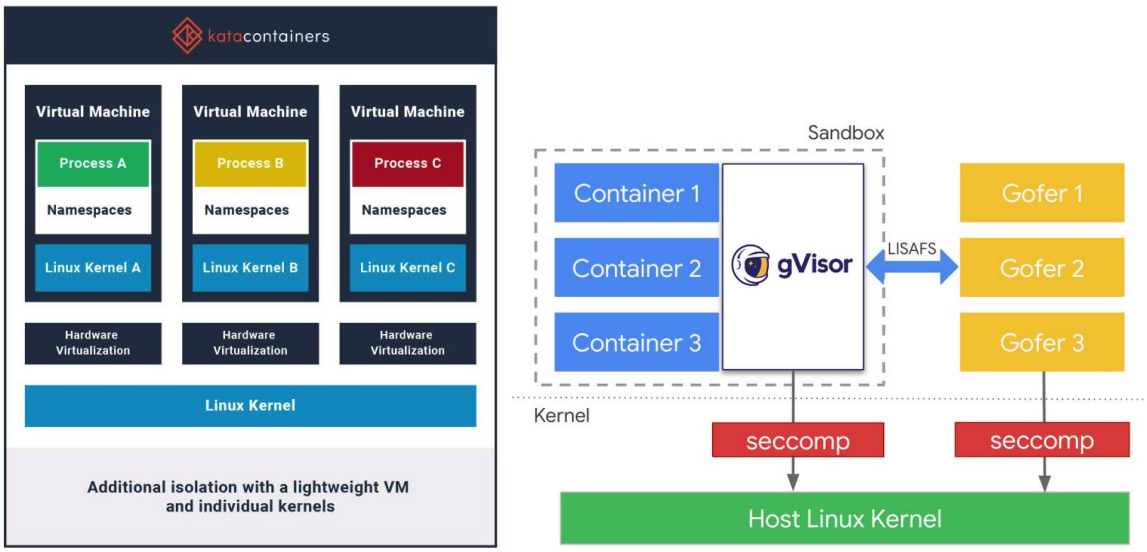

图 1.3 基于 VM 的内核隔离和用户级内核隔离方案（以 Kata 和 gVisor 为例）

（3）基于 linux 内核现有机制隔离：利用操作系统内核中现有的资源管理和安全机制，例如 cgroup、namespace、capabilities、seccomp 和 apparmor ，为容器提供系统资源的隔离视图，限制它们对系统调用、特权函数和敏感文件的访问。虽然这种方法由于与主机内核紧密耦合而实现了接近本机的性能，但它无法提供足够的隔离或允许应用程序自定义内核配置或策略。

以上三种隔离方式有一些共同的缺陷：

- 基于白名单和黑名单的内核级隔离不如为容器维护单独内核的方法那么灵活。最重要的是，除了安全检查，容器之间没有物理隔离，可能导致逃避权限检查。
- 现有的安全机制无法支持容器特定的内核定制。
- 缺乏容器间的数据隔离。许多内核数据结构在内核初始化时分配并在内核空间中全局共享，对共享数据的并发更新可能会导致严重的锁定争用，从而导致性能急剧下降。并且，用户可能会无意或有意耗尽共享数据所需的固定内存块，从而导致拒绝服务或内存不足错误。

因此，本项目拟进行的工作是实现一个可定制的全面隔离内核框架，同时拥有近似于原生 Docker 容器性能。

1.4 预期目标

本项目旨在对目前的轻量级虚拟化技术进行优化，改进目前容器技术存在的安全性和灵活性问题，在容器的内存管理和 CPU 调度系统上的预期目标如下：

（1）轻量级：vkernel 依赖于内核 ftrace 机制拦截发送到主机内核的请求并将其重定向到 vkernel 实例（vKI），vKI 可以作为内核模块动态加载和更新，并且独立于主机内核。通过提供仅包含应用程序需要的最小内核，来减少启动时间和运行时开销，从而提高隔离性和性能。通过精简操作系统内核相关的部分，这种方法可以在提高隔离性的同时，保持低延迟启动，接近于原生 Docker 容器的启动时间。

（2）可定制：用户可以自定义 vkernel，在特定容器中提高常用系统调用中的数据隔离，启用共享内核参数的不同配置，支持自定义调度策略。

（3）安全：内核隔离可以通过减少攻击面、权限控制来防止攻击者利用这个容器来攻击其他容器。与安全增强机制结合使用，提供基于角色的访问控制，进一步加固容器和系统的安全。同时，隔离机制还能更轻松地跟踪容器的行为，确保审计日志的清晰和准确。

（4）高效：基于内核可加载模块为容器生成私有运行环境，支持容器个性化定制 CPU 调度策略、虚拟内存超配策略等，从而提升容器应用的性能和稳定性。

CHAPTER 2

面向容器的虚拟内核框架设计

针对目前容器环境下容器应用共享宿主内核，无法个性化定制内存管理策略和进行 CPU 调度虚拟化的问题，我们采用了一种基于虚拟内核的虚拟化框架，通过为容器在内核态引入一个虚拟内核，解耦容器对宿主机内核的共享。我们通过可加载内核模块(Loadable Kernel Module, LKM)实现了虚拟内核的设想。相较于修改内核，LKM 的优势在于可以在运行时动态加载和卸载模块，不需要重新编译内核，因此更加灵活，更加容器部署。并且可加载模块同样位于内核态，与修改内核的方式有一样的性能

2.1 虚拟内核框架总体思想

容器应用无法做到个性化定制内存管理策略和 CPU 调度虚拟化的根本原因在于对宿主内核的共享，不同于安全容器基于硬件虚拟化并为容器提供独立内核的机制，虚拟内核基于原生容器消除硬件虚拟化层且共享宿主机内核的基础，为容器提供一个额外的虚拟内核，在虚拟内核中通过对容器依赖的最小内核代码与数据实现私有化。如图 2.1 所示，普通容器所有内存操作直接与宿主内核交互，而在引入虚拟内核后，容器直接与自身对应的虚拟内核交互，消除部分容器对宿主机内存管理机制的依赖，从而实现容器可以定制自身内存管理策略。

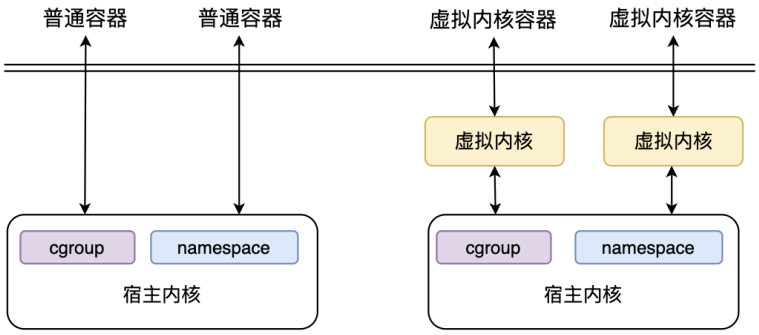

图 2.1 面向容器的虚拟化内核总体思想

在总体思想的指导下，虚拟内核框架在设计面向容器的内存管理策略隔离时需要考虑以下几点原则：

（1）轻量高效。容器的轻量特性是其重要的优势，因此虚拟内核框架必须以最小的性能开销来实现容器内存管理策略的个性化定制，以确保在保证高性能的同时，不会影响容器应用的正常运行。

（2）安全。在实现虚拟内核框架支持内存管理策略隔离时，必须确保不会为容器系统引入新的代码漏洞。为了避免安全隐患，虚拟内核框架的设计必须遵循最佳实践，使用安全的编程语言和代码库，并经过充分的测试和验证。

（3）易部署。用户使用虚拟内核应可以快速部署，无需重新编译内核，或服务停机。同时，为了不破坏现有的容器应用和环境，虚拟内核框架应该实现与容器系统的无缝集成，不需要对容器应用进行修改，让用户可以无感知地使用。

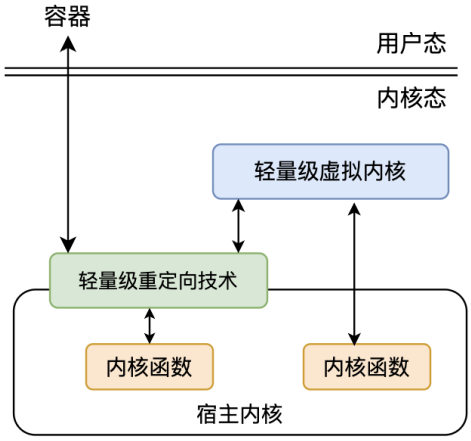

图 2.2 虚拟内核设计框架

为了保证容器的轻量高效特性，虚拟内核框架与原生容器一样消除了硬件虚拟化层，为容器提供一个虚拟内核实例以解耦容器依赖的部分内核机制。基于前文对现有研究的分析，gVisor 容器技术与本文类似为容器实现一个单独内核，并通过重定向技术拦截容器系统调用访问。然而，用户态内核方案与开销较高的重定向技术使其性能较低。为了使虚拟内核框架达到轻量级、可定制、安全和高效的目标，我们从虚拟内核实现机制与重定向机制两方面入手，设计了内核态的方案与轻量级的重定向技术，保证了虚拟内核框架的安全高效。

2.2 轻量级重定向技术

基于以上虚拟内核框架的总体思想，我们设计了轻量级的重定向技术。在实现轻量级重定向技术时，需要在设计轻量级重定向技术时需要考虑性能问题。由于内存系统和 CPU 调度是内核中频繁使用的子系统，需要将内核函数访问在虚拟内核和宿主机内核之间进行频繁重定向，因此在实现时需要尽量减少对性能的影响。

在传统内核中，存在两种重定向技术，一种是基于陷阱（trap）的方法，另一种是基于跳转的方法。基于 trap 的技术需要操作系统的支持，通过在被跟踪点插入断点指令，当该指令被执行时，中断处理程序将控制转移到指令代码区域。这种技术的缺点是开销较高。而基于跳转的技术则提供了更低的开销，它通过简单的跳转指令（例如 call 指令）跳转到指定区域，而不需要触发 trap 指令。

函数跟踪器（Function Tracer, Ftrace）是一种高效的内核执行流跟踪与重定向技术。Ftrace 包括一整套内核跟踪与分析工具，本文这里只关注其底层内核执行流重定向机制。该技术基于 gcc 编译的“-pg”选项，在编译时为每个函数的入口插入桩函数，并利用 call 指令对内核函数进行重定向来实现对内核执行流的跟踪和控制。在系统初始化时，这些桩函数被替换为无操作（No Operation, NOP）指令，以确保对系统性能的最小影响。但是，在启用 Ftrace 时，这些 NOP 指令会被替换为 call 指令，进而完成内核执行流的跟踪和重定向。尽管基于跳转的方式比基于 trap 的方式性能更加出色，但是在实际应用中仍然存在一些不可避免的开销。

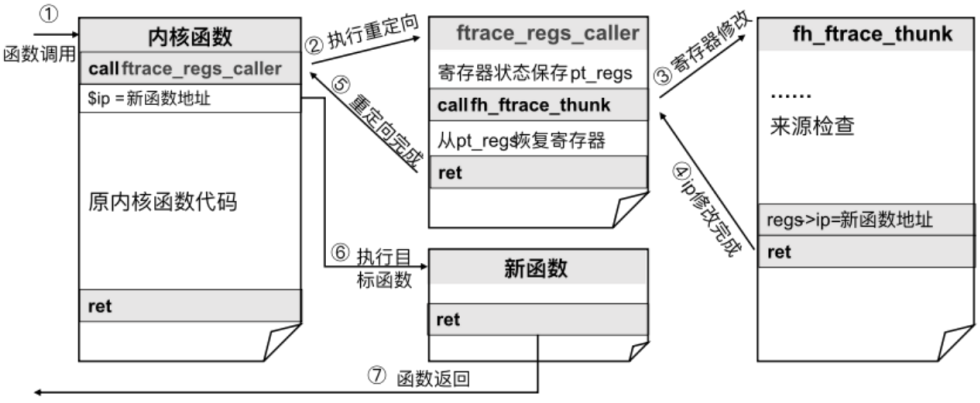

图 2.3 ftrace 重定向执行流

具体而言，如图 2.3 所示，当 Ftrace 重定向一个内核函数执行流时，内核会首先访问被重定向的原内核函数，接着在该函数的第一条指令处由于被替换为 call 指令，

将跳转至 ftrace_regs_caller 函数完成寄存器的保存操作。然后，再次调用 call 指令跳转到重定向关键函数 fh_ftrace_thunk，在该函数中完成保存的寄存器 ip 修改。随后，原内核函数会返回至 fh_ftrace_thunk，由于保存的寄存器 ip 值已被修改，因此返回后原内核函数不会继续执行，而是会跳转至新函数执行，最后再次返回调用点。在整个过程中，仍然需要经过两次 call 指令和多次复杂的栈操作，这些操作不可避免地会对系统性能产生一定的影响。

2.3　vkernel 内核模块架构

2.3.1 容器的 vkernel 实例

在 vkernel 内核模块中，vKI 是在启动容器时加载的内核模块，根据容器的 PID namespace 绑定到每一个容器上。在容器运行的过程中，vKI 负责执行特定于容器的安全检查、数据隔离以及用户定义的自定义。

vKI 本质上充当了共享主机内核上的最小虚拟化接口，用户可以根据自己的需求定义配置文件并生成 vKI。配置文件指定了容器可以访问的系统调用、特权函数和文件，以及用户定义的数据隔离和其他资源管理策略。vKM 在哈希表中存储容器的 PID namespace 与其对应的 vKI 的引用对。vKI 通过 seccomp、 capability 和 AppArmor 等方式为系统调用、权限检查和文件访问提供更有效的隔离。

```
struct vkernel {
    char name[50];
    /* syscall virtualization */
    sys_call_ptr_t sys_call_table[__NR_syscall_max+1];
    long (*do_futex)(u32 __user *uaddr, int op, u32 val, ktime_t *timeout,
                u32 __user *uaddr2, u32 val2, u32 val3);
    /*task_struct->cred->cap_effective*/
    kernel_cap_t    cap_effective;
    /* hashmaps which contain access rights */
    struct Vkernel_hashmap *vknhash_reg;
    struct Vkernel_hashmap *vknhash_dir;
};
```

1. 系统调用隔离

Linux 内核在一个全局系统调用表(sys_call_table)中保存所有系统调用的地址，并根据其 ID 定位系统调用的实现。由于系统调用表在主机上的所有容器之间共享，因此现有的用于系统调用权限检查的安全机制（如 seccomp）使用权限过滤器来针对

表中的所有调用检查请求调用，直到找到匹配项。与这种设计相反，vKI 为每个容器维护一个私有的系统调用表 vki_sys_call_table，其中仅包含容器允许访问的系统调用。私有系统调用表中其他系统调用的所有条目都标记为 NULL，容器发起的这些系统调用会被 KILL 信号拒绝。对每一个容器保存系统调用表保证了权限检查在很短的时间内就可以完成。

2. 特权函数隔离

对于 capability 来说，如果一个进程通过内核漏洞获得了提权，它就可以逃避权限检查，降低了内核的安全性。对于特权函数，我们在 VKI 中额外添加一个权限检查。实现上，每个 VKI 对每个容器保存一个只读的 capability。容器级的 capability 优先级比进程的 capability 要高,当两者冲突时，前者可以覆盖后者。对于截取的函数，vKI 首先检查容器的 capability，如果通过了，继续检查每个进程的 capability。否则，就检测到了进程的越权。由于容器级的 capability 是只读的，且对容器内的进程不可见，所以 vKI 比 capability 机制更加安全。设置容器初始化进程的 capability 如下

```
int task_set_capability(struct task_struct *target, unsigned long *caps_data,
        unsigned long *caps_bounding, unsigned long *caps_ambient)
{
    kernel_cap_t effective, inheritable, permitted, _effective, _inheritable, _permitted;
    struct cred *new;
    int ret, i, j, action;

    cap_capget(target, &_effective, &_inheritable, &_permitted);

    // bounding
    if ((1 << CAP_SETPCAP) & _effective.cap[0]) {
        for (i = 0; i <= CAP_LAST_CAP; i++) {
            j = i;
            if (j > 31)
                j %= 32;
            if ((1 << (unsigned int)j) & caps_bounding[i / 32])
                continue;
            ret = cap_task_prctl(PR_CAPBSET_DROP, i, 0, 0, 0);
            if (ret != 0)
                return ret;
        }
    }

    // effective, permitted, inheritable
    for (i = 0; i < _KERNEL_CAPABILITY_U32S; i++) {
        effective.cap[i] = caps_data[i];
```

```
        permitted.cap[i] = caps_data[2 + i];
        inheritable.cap[i] = caps_data[4 + i];
    }

    new = prepare_creds();
    if (!new)
        return -ENOMEM;

    ret = cap_capset(new, current_cred(), &effective, &inheritable, &permitted);
    if (ret != 0)
        return ret;
    commit_creds(new);

    // ambient
    for (i = 0; i <= CAP_LAST_CAP; i++) {
        action = PR_CAP_AMBIENT_LOWER;
        j = i;
        if (j > 31)
            j %= 32;
        if ((1 << (unsigned int)j) & caps_ambient[i / 32])
            action = PR_CAP_AMBIENT_RAISE;
        ret = cap_task_prctl(PR_CAP_AMBIENT, action, i, 0, 0);
        if (ret != 0)
            return ret;
    }
    return 0;
}
```

3. 文件隔离

apparomor 机制经常会造成不必要的开销，因为它会检查所有的文件权限，但是大多数文件都不是敏感文件。vKI 用两步实现文件权限检查。第一步，vKI 利用文件 inode 的 i_opflags 中未使用的比特位来指示一个文件是否敏感。如果任何容器的 AppArmor 配置在黑名单中包含了它的路径，则敏感比特位设置为 1。

第二步，在初始化时，vKI 扫描容器的 apparmor 配置以识别应该在文件访问时进行检查的敏感文件，并构建一个哈希表，从敏感文件的 inode 号映射到配置中指定的相应访问权限。在文件访问时，vKI 拦截通用权限函数并检查请求的 inode 的 i_opflags 中的敏感比特位。如果比特位为空，则立即返回 ALLOW。否则，它会在哈希表中查找该 inode 的访问权限。如果黑名单中存在匹配项，则拒绝文件访问。vKI 可以帮助移除对非敏感文件的权限检查。inode 结构体部分代码如下

```
struct inode {
    umode_t         i_mode;
    unsigned short      i_opflags;
    kuid_t          i_uid;
    kgid_t          i_gid;
    unsigned int        i_flags;
...
}
```

2.2.2 系统级 vkernel 管理模块

vKM 是一个可加载的内核模块，负责将容器发出的内核请求，根据哈希表重定向到相应的 vKI。vKM 依赖 inline hooks 来拦截系统调用和特权函数调用。启动时，主机将 vKM 作为可加载的内核模块加载，vKM 为默认的安全检查和用户自定义的隔离注册 inline hooks。需要强调的是，考虑到 ftrace 带来的高开销，vKM 并不使用已有的内核跟踪机制 ftrace，而是使用 inline hooks。它的工作机制是：首先找到需要重定位的函数地址，然后重定向到 vKM 中的新功能实现。

Inline hook 技术的设计方式是通过在函数入口处插入一个 jmp 指令，并直接指向自定义函数的入口地址。当内核调用该函数时，会首先跳转到自定义函数的入口处执行自定义的代码，完成操作后再跳转回原函数继续执行。由于 Inline hook 技术采用的是基于 jmp 指令的重定向技术，不需要保存和恢复寄存器状态，也不需要进行额外的栈操作，因此可以极大地减少重定向操作的开销。

容器对物理内核的部分访问会重定向到虚拟内核，使个性化策略作用到内核实例中。在 vKM 中首先实现了用于跟踪的 ftrace_hook 以及安装和卸载方法，如下方代码所示。

```
struct ftrace_hook {
    const char *name;
    void *new_func;
    void *orig_func;

    unsigned long address;
    struct ftrace_ops ops;
};

int fh_install_hooks(struct ftrace_hook *hooks, size_t count);
void fh_remove_hooks(struct ftrace_hook *hooks, size_t count);
```

2.2.3 vkernel 容器运行时

vKernel 容器运行时在原生 runc 运行时的基础上，实现一个可加载内核模块的 vKernel 容器运行时。它负责加载、更新和卸载容器的 vKI，并在 vKM 中注册容器对应的 vKI。vKernel 容器运行时符合 OCI 规范，保证了与现有容器开发工具的兼容性。

容器创建：vKernel 容器运行时首先读取容器的配置信息，然后设置 namespace 和 cgroup，构建 rootfs 并挂载文件系统。原生的 runc 是根据 cgroup 和 namespace 来实现容器的隔离和资源限制,vKernel 容器运行时则绑定了容器 id 和对应的内核模块，确保进程在指定的隔离和资源限制环境下运行。

容器启动：vKernel 容器运行时会创建并启动容器进程。可以通过指定自定义的 vKernel 运行时，初始化容器的 vKernel 模块。vKernel 运行时仍然和原生的运行时性能接近，不会带来严重的额外开销。

隔离与资源限制：vKernel 容器运行时结合 seccomp 限制进程可以使用的系统调用，降低容器逃逸的风险，提高容器间的安全隔离。runc 和 AppArmor 结合则可以为容器进程设置更严格的访问控制策略，可以限制容器进程访问宿主机文件系统、网络资源等，进一步增强容器安全性。

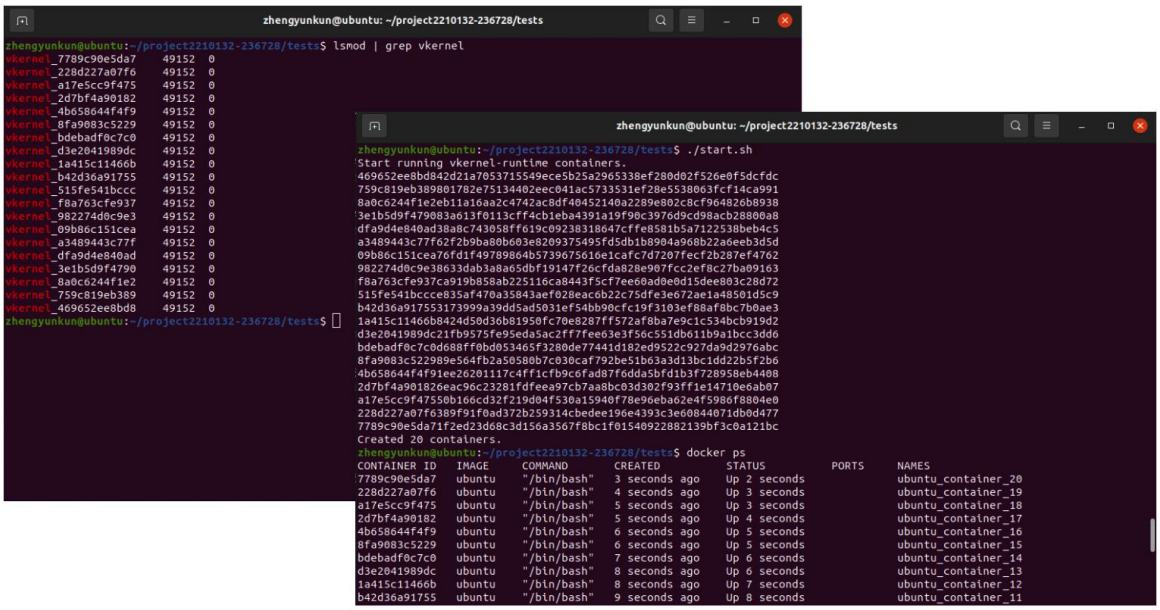

图 2.4 容器创建和 vkernel 内核模块初始化

CHAPTER 3

面向容器的 CPU 调度虚拟化方案设计

3.1 总体设计概述

CPU 调度方面面临的局限性主要源于其与宿主机调度系统的紧密耦合，CPU 调度虚拟化方案的主要目的是为每个容器提供多样化、可配置的调度解决方案，同时保证宿主机上全局调度器和容器内局部调度器之间、不同容器的局部调度器之间都相互隔离，互不影响。

为了保证容器的安全性和隔离性，CPU 调度虚拟化方案种选择 CFS 调度器作为全局调度器，这是由于 CFS 调度器具有公平性且不需要依赖于权限等优势。而在局部调度器上，CPU 调度虚拟化方案能够提供多样化的策略和参数配置方案，使得容器内部的任务调度可以根据具体的应用场景进行个性化配置，从而达到更高的性能和效率。

CPU 调度虚拟化主要基于容器的两层调度模式实现。cgroup 机制指出容器对资源的控制都是以分层的形式来进行的，同时 namespace 机制也是以层级结构进行资源视图的隔离。这使得在容器调度过程中，全局调度器和局部调度器上的运行相互透明，全局调度器上不区分局部调度器和普通线程，而局部调度器上资源的分配和控制也不影响全局调度器上的调度结构。此外全局调度器上 CFS 公平调度的特点特能能够保证容器间的公平性和隔离性。以上分析这意味着 CPU 调度虚拟化具有可行性。

面向容器的 CPU 调度系统虚拟化工作将从策略和参数两方面进行，基于 Linux6.1 版本，在容器内部局部调度器上实现独立任务调度和参数视图隔离，以实现应用的个性化配置。具体而言，它包括以下两个方面：

（1）独立任务调度：局部调度器上的 CPU 资源的分配和控制与全局调度器相互独立。这意味着当全局调度器将 CPU 资源分配给局部调度器后，局部调度器上的 CPU 资源的分配应当不受到全局调度器上配置的影响。这满足了容器内部根据自身需求对

任务调度方式进行控制的要求。

（2）参数视图隔离：局部调度器上应当有单独的参数视图。这意味着全局调度器上对参数的配置应当仅作用于全局调度器，局部调度器不再复用全局调度器上的参数配置，而具有其独立的参数列表。这满足了容器内部根据调度需求对参数进行隔离化配置的要求。

3.2 CPU 调度虚拟化中的独立任务调度

3.2.1 独立任务调度概述

对于调度系统来说，任务调度方式受限于调度策略，而 CPU 调度虚拟化方案旨在为容器内局部调度器提供可供选择的任务调度方式。因此，需要为局部调度器提供不同的策略可供选择配置，实现任务调度的独立化设计。这一设计思路用图 4.1 所示的过程来概括。

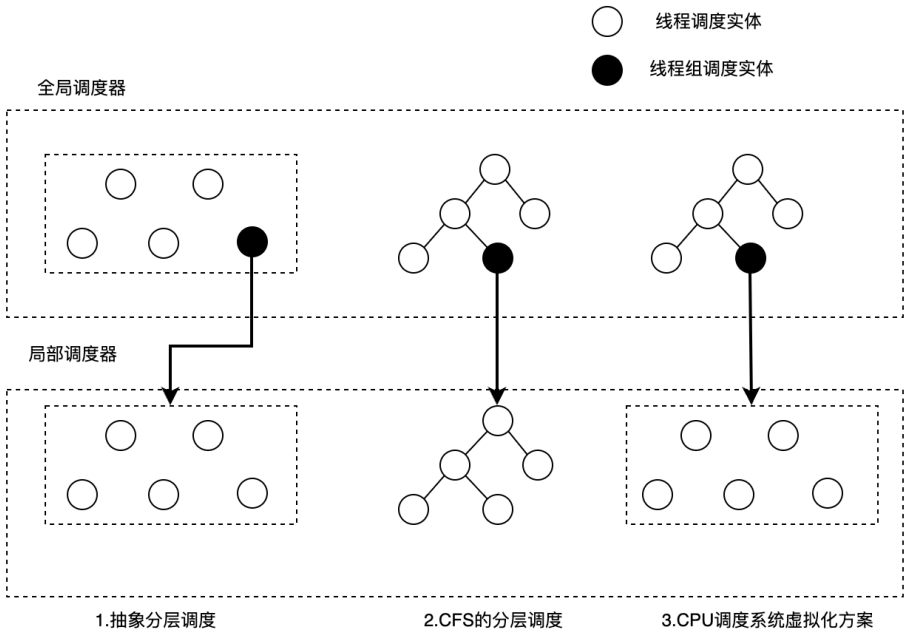

图 3.1 任务调度方式的演变

图 4.1 中的 1 描述了容器的抽象两层调度模式，其中全局调度器和局部调度器分别控制宿主机上和容器内的调度行为。由于安全性和隔离性的要求，容器通常运行在 CFS 调度器上，如图 4.1 中 2 所示，容器将实际运行在 CFS 调度器上的层级红黑树结构中，无法使用 CFS 以外的调度方案。CPU 调度虚拟化方案，提出了在局部调度器上实现独立的任务调度方式，如图 4.1 中的 3 所示，局部调度器不受全局调度器的影响。为了实现在局部调度器上的独立任务调度方式，需要首先对在局部调度器上引入新任

务调度方式，同时，对这一实现方式进行可行性和兼容性分析。然后，在具体设计如何引入新的任务调度方式时，则需要根据具体策略的调度需求具体设计。

3.2.2 独立任务调度方案设计

本方案将以先进先出（First-in-First-out，FIFO）调度策略作为第一个样例来验证独立任务调度的可行性。FIFO 策略是最简单的调度策略之一，其特点在于保持运行队列上调度实体的先进先出，即先到达运行队列上的任务先获得服务资源。此外 FIFO 策略理想情况下要求调度实体一直运行到自愿放弃 CPU 资源为止，但实际情况下考虑到系统安全性和稳定性，为避免其他任务被严重阻塞或饿死，需要一定程度上放宽该限制。

为了在局部调度器上实现 FIFO 模式的任务调度方式，首先需要考虑如何设计接口来实现对局部调度器上任务调度方式即策略的配置，然后再具体设计实现新策略，其中对新策略的设计将主要包含四个部分。

（1）运行队列的定义。即为新策略设计运行队列，难点主要在于如何选择合适的位置添加，从而不影响 CFS 调度器的正常运行，保证兼容性。

（2）运行队列的维护。即运行队列上任务的入队、出队和选择等调度行为，需要 Linux 对调度器良好的封装，尽可能复用内核已提供的接口，避免系统漏洞。

（3）调度时机的判断。即新策略在周期性调度中和特殊情况的抢占中的逻辑实现，主要取决于新策略的调度理念，需要同时考虑对对全局调度器的影响。

（4）附加机制补充指对新策略附加机制的补充，而 FIFO 策略作为一种简单的调度策略，并没有如 CFS 调度器中复杂均衡等复杂的机制，因此在 FIFO 策略的实现中并不需要考虑该内容。

下文将从前三个方面对独立任务调度方案进行详细设计，同时在最后简单展示了与 FIFO 策略调度逻辑类似的轮转（Round-Robin，RR）策略的实现方式。

1. 策略的配置接口

开始为局部调度器添加新策略前，需要首先考虑策略的配置是作用于单个线程、单个运行队列还是单个控制组。

考虑策略作用于单个线程，该方案看似是最合理的方案，但设计实现过于复杂，而且在实际情景中，多策略同时存在可能会导致优先级较低的策略任务很可能被阻塞，

倘如想通过资源限制的方案来解决阻塞问题，又必然需要在局部调度器中引入大量的判定过程（参考 RT 调度器），这可能会为调度结果带来明显的负面影响。并且容器常被用于单进程多线程环境，运行环境并不复杂，引入复杂的多策略并存画蛇添足。

考虑策略作用于单个运行队列，这意味着同一个控制组在不同 CPU 上的运行队列使用的策略不同，这将有利于多策略任务同时存在且保证任务之间的隔离性。但为了避免任务的调度策略不可控制的随意改变，可能需要将任务固定在同一个 CPU 上，这必然导致任务可能获得的资源受限，不仅影响任务的运行，还浪费了硬件资源。

考虑策略作用于整个控制组。一方面容器本身依赖控制组来实现资源的控制，控制组与局部调度器将具有良好的适配性。另一方面控制组机制已相对完善，其中提供大量的接口可以降低对策略配置实现的复杂性，便于实现。

综上，本项目采用策略作用于整个控制组的方案，前两个方案分别存在设计复杂，不利于充分利用 CPU 资源的问题。

在 CPU 控制组的参数中包括 cpu.shares、cpu.cfs_period_us、cpu.cfs_quota_us 等。我们在局部调度器在控制组的参数列表中添加 cpu.real_policy 用来指示局部调度器上实际正在使用的 CPU 资源分配和控制方式，即策略类型。当该值为 0 时表示局部调度器正在使用 CFS 调度器提供的 SCHED_NORMAL 策略，当该值为 1 时表示正在使用 FIFO 策略，后续新策略的引入可以继续补充。与此同时，在调度组的数据结构中增加"real_policy"变量，该变量用于在调度过程中进行判定和区分，在对参数文件“cpu.real_policy”进行读写时，需要同时更新所属调度组的对应变量。

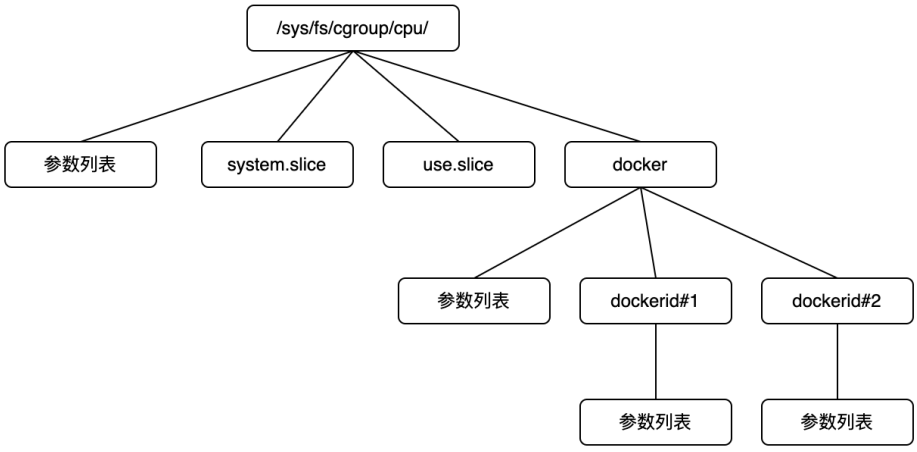

图 3.2 Linux 控制组目录下的文件结构

虽然由于直接复用控制组接口将在容器以外的系统层级也出现配置参数

“cpu.real_policy”，但可以通过引入对层级的特判，在实际读写时限制无法对根调度组层级的参数进行配置，同时在调度过程中也避免对根调度组进行处理，从而避免造成系统错误并减少局部调度器对全局调度器的影响。

在读写权限检查的关键路径上添加判断来赋予容器对策略配置文件的写入权限。

此外，还有一个至关重要的问题需要考虑，容器内部不具备对控制组参数进行配置的权限。为此，我们可以在写权限检查的关键路径上添加判断，赋予容器对策略配置文件的写入权限。在__mnt_is_readonly 函数中加入对如下函数的判断即可。

```
int vkernel_security_check_file(struct file* file){ //检查给定文件路径中是否包含特定的内核参数
    struct pid_namespace *pid_ns = task_active_pid_ns(current);
    if(!pid_ns || pid_ns->level == 0)
    return 0;
    char path, buf;
    if(file){
        buf = __getname();
        if(!buf)
           return 0;
        path=dentry_path_raw(file->f_path.dentry,buf,PATH_MAX); //使用 dentry_path_raw 函数获取文件的路径
        if(IS_ERR(path)){
            __putname(buf);
            return 0;
        }
        if(strstr(path,"cpu.real_policy") != NULL ||
                strstr(path,"sched_wakeup_granularity_ns") != NULL  ){
            __putname(buf);
            return 1;
        }
        __putname(buf);
    }else{
        if(current && current->nameidata && current->nameidata->name){
            if(strstr(current->nameidata->name->name,"cpu.real_policy") != NULL ||
                strstr(current->nameidata->name->name,"sched_wakeup_granularity_ns") != NULL){
                return 1;
            }
        }
    }
    return 0;
}
```

2.运行队列的定义

FIFO 的数据结构采用简单的链式结构，同样的链式结构和调度理念可以在运行队列的设计和维护方面参考 RT 调度器中对 SCHED_FIFO 的实现。

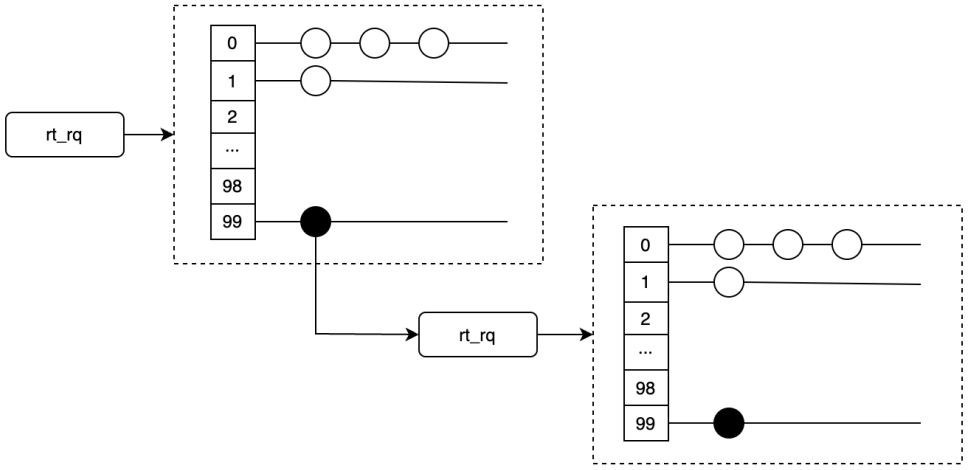

图 3.3 RT_调度器的多优先级链表结构

SCHED_FIFO 策略在具体实现时由于需要考虑到复杂调度环境中任务将会具有不同的优先级，因此 RT 调度器实际维护的是图 4.3 所示的一组链式队列。其中每一个链式队列对应一个优先级，高优先级任务结束运行之前低优先级任务无法得到运行机会。同时在每一个链式队列上保持任务的先进先出逻辑，当任务因为非自愿原因需要让出 CPU 资源时，将重新插入到队列的尾部等待下一次的调度机会。此外 RT 调度器同样支持组调度机制，因此与 CFS 调度器类似，RT 调度器上同时存在线程和线程组的调度实体，如图 4.3 所示线程组调度实体将具有自己的子层级运行队列，其中同样包含多个优先级的链式队列，最终整体上形成层级可嵌套的链式结构。

而 FIFO 策略的实现中将简化 RT 调度器上的多优先级链表结构，取而代之的是使用单链表方式实现。一方面是因为容器的实际运行环境通常不如宿主机上复杂，从而并没有实现多优先级的必要性，另一方面策略配置文件是作用于整个控制组的，如果实现为容器内不同任务配置不同的优先级，同样需要考虑如何落实对单一线程的配置设计。维护多优先级队列所需要投入的成本和引入的更复杂判定流程都与最终得到性能效益不甚匹配。因此最终 FIFO 策略选择在局部调度器中实现单一优先级的先进先出调度。

为了实现单链式、局部调度器内部统一的 FIFO 策略，需要首先考虑新策略将如何

良好地兼容全局调度器的 CFS，即如何选择合适的位置来为 FIFO 策略添加新运行队列而不对全局调度器造成过多影响。控制组机制中每一层都有独立的运行队列，不同层级之间的运行队列靠调度实体中的成员相连接，为了解决该问题，首先需要对全局调度器使用的 CFS 进行数据结构分析。最终 CFS 调度器上的数据结构关系图如图 4.4 所示，其中简化了结构体内部信息。

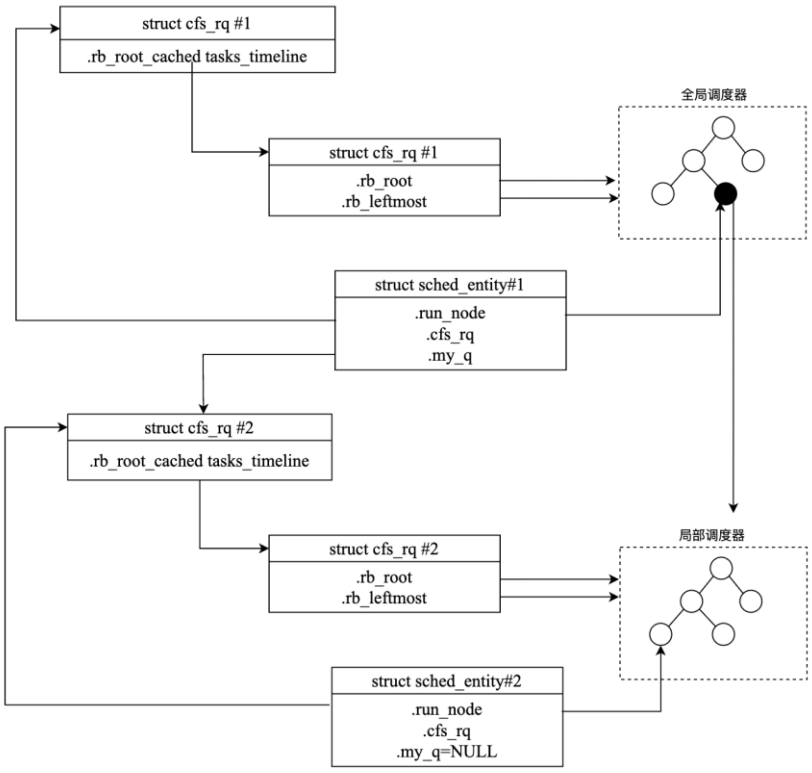

图 3.4 CFS 调度器中数据结构的关系

图中编号 1 表示父层级，编号 2 表示子层级。调度实体 sched_entity 表示一个线程或一个线程组，其中成员 run_node 是红黑树单一节点 rb_node 的类型，cfs_rq 指向运行队列，my_q 指向子层级的运行队列。CFS 调度器中，cfs_rq 维护了红黑树上的根节点和最左侧结点。在调度器选择调度实体时，使用成员 my_q 作为循环中的判断条件来逐步自上而下进行选择，当 my_q 成员不指向空指针时，就进入子层级的 cfs_rq 上继续进行选择。最后从 task_timeline 中选择出 rb_leftmost 结点。可以说，真正的线程放置和选择逻辑都在 task_timeline 中进行。

```
struct sched_entity {
    struct load_weight      load;
    struct rb_node          run_node;
    struct list_head        group_node;
```

```
    struct list_head        list_node;
    unsigned int            on_rq;
    unsigned int            on_list;
    ......
}
/*
 * Perform scheduler related setup for a newly forked process p.
 * p is forked by current.
 *
 * __sched_fork() is basic setup used by init_idle() too:
 *
 * 在进程复制（fork）时被调用，用于初始化新进程的调度相关的数据结构。
 * 由于增加了 FIFO ，因此需要初始化一个链表头。
 *
 */
static void __sched_fork(unsigned long clone_flags, struct task_struct *p)
{
    ......
    p->se.nr_migrations     = 0;
    p->se.vruntime          = 0;
    INIT_LIST_HEAD(&p->se.group_node);
    INIT_LIST_HEAD(&p->se.list_node);
    ......
}

void __init sched_init(void)
{
    unsigned long ptr = 0;
    int i;
    ......
    root_task_group.shares = ROOT_TASK_GROUP_LOAD;
    root_task_group.real_policy = SCHED_NORMAL; // 全局调度器要采用 CFS 策略
    ......
}
```

因此，在涉及 FIFO 局部调度器时，可以考虑整体复用 CFS 调度器的结构，只在最关键的选择和放置逻辑处，即 task_timeline 处增加一个链式运行队列，只修改了调度实体 sched_entity 在 cfs_rq 上的放置方式，而不修改调度器对运行队列 cfs_rq 和调度实体 sched_entity 的调度行为，从而减少对原生 CFS 调度器的影响。

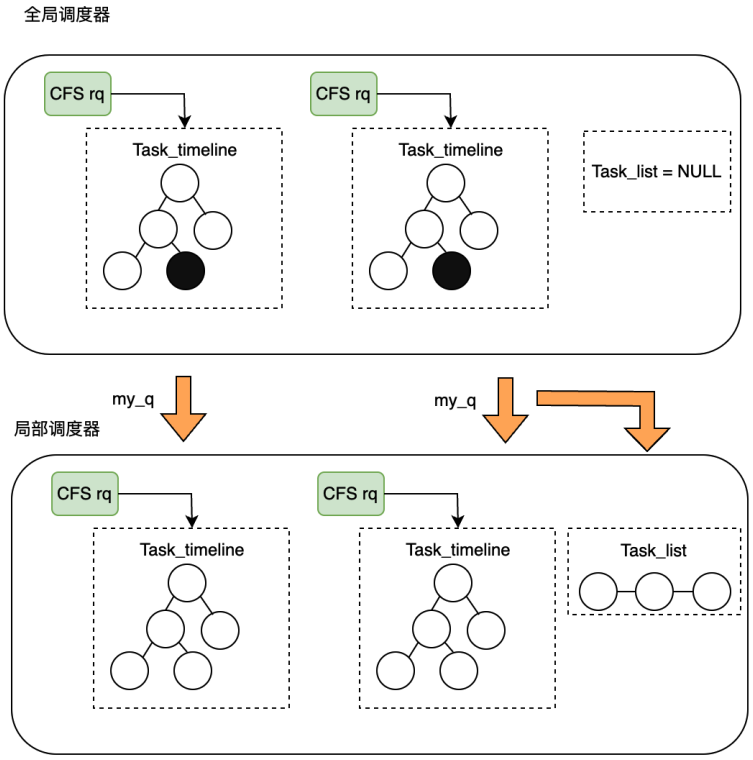

图 3.5 运行队列数据结构调整

3. 运行队列的维护

运行队列的维护包括任务的入队、出队和选择逻辑。在 FIFO 策略下，由于先进先出的调度逻辑，因此在进行任务选择时将直接从队列头部进行选择，同时入队时将任务放置在队列尾部。

FIFO 策略的入队逻辑比较简单，在线程停止占用 CPU 资源或新创建的线程时，将线程放置到运行队列尾部。在全局调度器中任务默认放置在 cfs_rq 的 tasks_timeline 中并以 vruntime 为键值放置到合适的位置上。而在局部调度器中，首先对任务所在调度组正在使用的调度策略进行判断。在调度策略的判断中如果是 CFS 调度器提供的 SCHED_OTHER 调度策略，则维持原操作行为将调度实体放置在 tasks_timeline 上，如果是 FIFO 调度策略，则需要将调度实体放置到 cfs_rq 的 tasks_list 运行队列上。

```
static void __enqueue_entity(struct cfs_rq cfs_rq, struct sched_entity se)
{
    if(policy_of_cfs(cfs_rq) == SCHED_FIFO){
        list_add_tail(&se->list_node, &cfs_rq->tasks_list);
        se->on_list = 1;
        cfs_rq->list_nr_running++;
        return;
    }
```

```
    rb_add_cached(&se->run_node, &cfs_rq->tasks_timeline, __entity_less);
}
```

FIFO 策略的出队逻辑可以直接从队列上将任务移除即可，问题在于局部调度器需要判断从哪一运行队列上执行任务出队操作。为简化运行队列上的出队行为，在调度实体 sched_entity 中添加了标识位 on_list 用来标记是否位于 tasks_list 上。标识位 on_list 在调度实体被放置到运行队列 tasks_list 上时设置为 1。因此在出队时可以通过调度实体的 on_list 判断是从 tasks_timeline 还是 tasks_list 上移除调度实体，当从 tasks_list 上移除调度实体时需要将 on_list 复位为 0。

```
static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity se)
{
    if(se->on_list){
        list_del_init(&se->list_node);
        se->on_list = 0;
        cfs_rq->list_nr_running--;
    }else
        rb_erase_cached(&se->run_node, &cfs_rq->tasks_timeline);
}
```

FIFO 策略中的选择逻辑可以直接通过从运行队列 tasks_list 的队列头部选择任务实现，在全局调度器上，将从 CFS 调度器的运行队列 tasks_timeline 上选择红黑树中的最左节点，即 vruntime 最小的节点对应的任务来运行。而在局部调度器中，需要首先对任务所在调度组正在使用的调度策略进行判断。在调度策略的判断中如果是 CFS 调度器提供的 SCHED_OTHER 调度策略，则维持原操作行为从 tasks_timeline 上选择，如果是 FIFO 调度策略，则需要从 cfs_rq 的 tasks_list 运行队列上进行选择。

静态的考虑上述策略似乎没有什么问题，但当策略配置已经改变，实际调度实体仍旧处于原来的运行队列中时，此时会出现严重的问题。调度实体无法按照配置的策略进行调度，会出现幻影现象。

当然，我们可以在策略动态改变的时候专门将所有任务从旧运行队列迁移到新运行队列上，保证调度策略和运行队列数据结构的一致性。但这必然会引入包括上下文切换在内的各种开销。为此，我们采用了一种更为轻盈巧妙的算法，即同时结合策略和当前运行状态来选择运行队列。

首先我们需要为运行队列增加一些辅助参数来帮助我们判断此时运行队列的状态。定义 list_nr_running，用来指示放置在当前 cfs_rq 的 tasks_list 上的任务数量，仅包含当

前层级而不包含子层级。定义 nr_running 表示 cfs_rq 当前层级的调度实体数量，同时包含单一任务实体和调度组实体。

```
struct cfs_rq { // cfs_rq 用于表示 CFS 运行队列的结构体
    struct load_weight  load;
    unsigned int        nr_running;
    unsigned int        h_nr_running;      / SCHED_{NORMAL,BATCH,IDLE} /
    unsigned int        idle_nr_running;   / SCHED_IDLE /
    unsigned int        idle_h_nr_running; / SCHED_IDLE /
    unsigned int        list_nr_running;
    // cfs 运行队列也需要为CFS 增加 list_nr_running 属性
    // 用来指示放置在当前 cfs_rq 的 tasks_list 上的任务数量
    u64         exec_clock;
    u64         min_vruntime;
```

最终的选择算法如下：

a）当局部调度器策略为 SCHED_OTHER 且 list_nr_running 不为 0 时，从 tasks_list 上进行任务选择；

b）当局部调度器策略为 FIFO 且 list_nr_running 等于 nr_running 时，从 tasks_list 上进行任务选择；

c）其余情况下从 tasks_timeline 上进行任务选择。

下面对上述算法做进一步解释。当调度策略由 SCHED_OTHER 变为 FIFO 时，若 tasks_timeline 不为空，则仍从原来的 tasks_timeline 上进行选择；若 tasks_timeline 为空，说明运行队列已经可以与调度策略匹配，任务全部迁移完成，即 list_nr_running 等于 nr_running 从 tasks_list 上进行任务选择。当调度策略由 FIFO 变为 SCHED_OTHER 时，若 tasks_list 不为空，即策略为 SCHED_OTHER 且 list_nr_running 不为 0，则仍从原来的 tasks_list 上进行选择；若 tasks_list 为空，说明运行队列已经可以与调度策略匹配，任务全部迁移完成，从 tasks_timeline 上进行选择。

```
struct sched_entity *__pick_next_entity(struct cfs_rq *cfs_rq)
{
    struct rb_node *left = rb_first_cached(&cfs_rq->tasks_timeline);

    if(cfs_rq->list_nr_running > 0){
        struct list_head *queue = &cfs_rq->tasks_list;
        if(policy_of_cfs(cfs_rq) == SCHED_NORMAL){
            if(queue->next != queue){
                return list_entry(queue->next, struct sched_entity, list_node);
            }
```

```
        }else if(cfs_rq->nr_running == cfs_rq->list_nr_running){
            if(queue->next != queue){
                return list_entry(queue->next, struct sched_entity, list_node);
            }
        }else{
            printk("policy_of_cfs(cfs_rq) != SCHED_NORMAL, nr_running = %d, list_nr_running
= %d\n ",cfs_rq->nr_running,cfs_rq->list_nr_running);
        }
    }
    if (!left)
        return NULL;

    return __node_2_se(left);
}
```

4. 调度时机的判定

调度器的调度时机可以分为两大类，以系统时间粒度 tick 更新运行信息带来的周期性调度和特殊时间节点通过已经判断信息带来的主动调度。

FIFO 策略除了链式结构下先进先出的特性以外，还有一个重要特性是任务将一直运行到自愿放弃 CPU 资源位置，而在两层抽象结构下，无法实现全局性的一直占据 CPU 资源，这也是为了避免容器内调度影响其它容器或任务的运行。因此在实现局部调度器上的 FIFO 策略时，只需要保证在容器范围内正在运行的任务不应当被抢占，而如果有容器外线程试图抢占当前运行的线程，则意味着当前容器获得的 CPU 资源可能已接近耗尽，这部分判断直接复用 CFS 调度器的判断部分即可。

综上，在周期性调度中，FIFO 策略不需要考虑运行队列上的公平性，因此可以仅更新任务运行信息，而不对是否发生抢占进行判断。

在主动调度中，两个调度实体属于同一个容器即控制组时，可直接跳过抢占检查，而其它情况下按照原生 CFS 调度器的判断方式来进行。

```
// 在更新调度实体的运行时间和虚拟运行时间的函数中，对 FIFO 策略进行配置
// 由于 FIFO 特性，直接返回即可
static void
entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued)
{
    /*
     * Update run-time statistics of the 'current'.
     */
    update_curr(cfs_rq);
    ......
#endif
    if(policy_of_cfs(cfs_rq) == SCHED_FIFO)
```

```
        return;
    if (cfs_rq->nr_running > 1)
        check_preempt_tick(cfs_rq, curr);
}
```

3.2.3 独立任务调度进阶设计—自适应策略

在实际容器应用场景中，容器所执行的任务的特点可能会动态变化。如何选择合适的调度策略，让容器发挥最大性能，始终是一个重要的话题。尽管我们可以通过对容器根据任务特点指定特定的调度策略，但对容器应用单一的调度策略，细粒度较大，灵活性不高，实际性能有限。

我们可以进一步细化策略调整的粒度，将调度策略与容器本身进一步解耦，让容器实际执行任务的情况，灵活的选择调度策略。具体来说，可以通过机器学习或引入新的工作负载指标的方式，对过往运行状态进行收集和分析，从而为任务自适应地选择调度策略。

由于机器学习方法的数据较难获取，本文就第二种方法进行了探索和实现。

具体实现可以分为两部分，首先我们可以选取一些与可以影响调度策略的指标，监测容器运行过程中这些指标的动态变化。然后通过对这些监测指标进行判断，在独立任务调度的基础上，修改 real_policy 参数，进行调度策略的动态调整。

我们在之前实现的 FIFO 策略上，和 CFS 功能进行动态选择。首先进行检测指标的选取。之后进行动态调整策略的设计。

1. 检测指标的选取

选取合适的检测指标是策略调整的基础，我们需要选择一些可以反映当前任务的运行特性和资源需求的指标，具体来说有一下几类。

我们知道，，FIFO 策略更适合并行度低的长任务环境，而 SCHED_NORMAL（通常指的是 CFS）策略更适合并行度高的短任务环境。这是因为 FIFO 可以保证任务按照其到达的顺序执行，上下文切换较少，适合那些需要连续、不可中断执行的长任务。而 CFS 能够动态分配时间片，确保高并行度环境中每个任务都有机会得到执行。

表 3.1 监测指标

参数名称	简单描述
CPU 使用率	反映当前容器或任务的计算密集程度。
上下文切换频率	反映任务的切换频率，频繁的上下文切换表明任务细粒度低且调度频繁。
负载平均值	反映系统的整体负载情况，较高的负载平均值意味着系统资源紧张。
任务的睡眠时间	判断任务在资源获取中的等待时间，任务的睡眠时间较长表明资源竞争激烈。

如表 3.1 所示，我们可以自定义一些监测指标来对容器任务的执行情况进行衡量。本项目中，我们以 CPU 使用率和上下文切换频率为例，展示了如何设计监测指标。我们在 core.c 中定义了 get_context_switches，通过访问 task_struct 结构体中的 nvcsw 和 nivcsw 字段，分别获取任务的自愿和非自愿上下文切换次数，并将它们相加，以计算总的上下文切换频率。定义了 get_cpu_usage，通过 rq->idle_stamp 获取 CPU 的空闲时间戳，通过 rq->clock_task 获取 CPU 的总任务时间。通过这两个时间戳的差值，我们可以计算出 CPU 使用率。

```c
//  获取当前任务的上下文切换次数
static inline int get_context_switches(struct task_struct *task) {
    return task->nvcsw + task->nivcsw;
}
//  估算当前 CPU 的使用率
static inline int get_cpu_usage(void) {
    struct rq *rq = this_rq();  // 获取当前 CPU 的运行队列
    u64 idle_time = rq->idle_stamp;  // 获取 CPU 空闲时间戳
    u64 total_time = rq->clock_task;  // 获取 CPU 任务时间戳
    if (total_time == 0)
        return 0;

    // 计算 CPU 使用率百分比
    return 100 * (total_time - idle_time) / total_time;
}

//  自适应策略调整函数，根据 CPU 使用率和上下文切换次数来动态调整调度策略。
void monitor_and_adjust_policy(struct cfs_rq *cfs_rq) {
    int cpu_usage = get_cpu_usage();  // 获取 CPU 使用率
    int ctx_switches = get_context_switches(current); // 获取当前任务的上下文切换次数

    // 获取当前任务组
```

```
    struct task_group *tg = cfs_rq->tg;

    if (cpu_usage > 70 && ctx_switches > 1000) {
        tg->real_policy = SCHED_NORMAL; // 将调度策略设置为CFS
    } else if (cpu_usage < 50 && ctx_switches < 500) {
        tg->real_policy = SCHED_FIFO; // 将调度策略设置为FIFO
    }
}
```

2. 周期性策略调整的设计

在容器调度策略调整过程中，如果在每次任务调度时都判断是否需要调整策略，可能会引入较大的性能开销。原因在于每次调度都要执行复杂的逻辑和计算，而在实际应用中，调度策略的调整并不需要如此高的细粒度。因此，我们引入了周期性策略调整机制，以在降低性能损耗的同时，保证调度策略的适应性和系统的整体效率。

具体实现为，利用内核自有的 sched_count 参数，在每次执行__schedule 函数后都进行自增，定义 MONITOR_INTERVAL，仅在 sched_count 达到这个值时，才触发调度策略的检查和调整，并将 sched_count 置为 0。

```
static void __sched notrace __schedule(bool preempt)
{
    ......
    // 调度计数器增加
    rq->sched_count++;

    // 检测周期内只调用一次 monitor_and_adjust_policy
    if (rq->sched_count >= MONITOR_INTERVAL) {
        monitor_and_adjust_policy(&rq->cfs);
        rq->sched_count = 0;  // 重置计数器
    }
    ......
}
```

项目中，诸如 MONITOR_INTERVAL 等超参数可以在实际使用过程中不断调优，确保在性能和调度灵活性之间找到最佳平衡点。monitor_and_adjust_policy 函数中的判断条件也作为一个经验值，后续可以进一步利用机器学习等方法进行进一步优化。

3.3 CPU 调度虚拟化中的参数视图隔离

此部分需要为每个容器的局部调度器实现自身的调度参数列表，与宿主机上全局调度器的调度参数列表区分开。当线程在运行过程中需要对调度参数进行读写时，将根据线程所属于的环境来判断读写操作将要作用于哪一组调度参数列表上。

为实现这种方式的调度参数隔离，首先需要确定哪些参数需要被隔离，然后实现对

参数的隔离，最后需要保证容器具有读写这些参数的能力。本小节后续内容将主要从这三个方面出发进行设计和实现。

3.3.1 参数选择

在 ProcFS 下有大量的参数与调度系统相关，但并不是所有参数都有必要进行隔离。因此可以在设计之初对参数隔离必要性进行分析。如 sched_schedstats 之类的参数，对宿主机上和容器内进行区别配置并没有太大意义。仅仅对 sched_latency_ns、sched_min_granularity_ns 和 sched_wakeup_granularity_ns 等明确影响相关调度器运行结果的参数进行隔离。本项目基于命名空间机制以 sched_wakeup_granularity_ns 为例实现了参数视图隔离。

表 3.2 与调度系统相关的参数

参数名称	简单描述
sched_latency_ns	一个运行队列所有进程运行一次的周期
sched_min_granularity_ns	SCHED_OTHER 策略下表示进程最少运行时间
sched_wakeup_granularity_ns	用来判断新唤醒的进程是否应该抢占当前进程
sched_migration_cost_ns	用来判断某个进程是否要在 CPU 之间发生迁移
sched_nr_migrate	负载均衡时一次最多可以移动多少个进程到另一个 CPU 上
sched_rr_timeslice_ms	SCHED_RR 策略情况中时间片轮转的单位时间
sched_schedstats	启用或禁用调度程序统计信息
sched_tunable_scaling	当内核调整 sched_min_granularity_ns 等参数时所使用的更新方法

3.3.2 参数配置能力

容器对 ProcFS 文件的权限是只读的，提高权限的方法不安全，因此可以从内核接口中进行。Linux 内核提供__mnt_is_readonly 接口检查一个虚拟文件系统装载点结构体是否是只读的，容器在对 ProcFS 进行配置时，将会通过该函数检查容器是否是只读的。因此为了赋予容器对指定参数的配置能力，可以在该接口中对所检查的文件进行判断，如果当前检查的问题是规定范围内的参数文件，则通过返回错误来指示容器对该文件同时具有读写权限，从而实现容器对参数的配置能力。其中 vkernel_security_check_file 函数在前文已经定义。

```
bool __mnt_is_readonly(struct vfsmount *mnt)
```

```
{
    if(vkernel_security_check_file(NULL))
        return 0;
    return (mnt->mnt_flags & MNT_READONLY) || sb_rdonly(mnt->mnt_sb);
}
```

3.3.3 实现参数视图隔离

参数视图隔离包括容器内参数读写的隔离，以及容器内线程运行时使用参数的隔离。参数读写的隔离主要需要在通过系统调用进行内核参数读写时指向容器内的调度参数列表来实现。而线程运行时使用参数的隔离需要首先定位参数的使用方式，在参数的使用中合适地、正确地将最终进行处理的参数设置为容器内调度参数列表上。

参数视图隔离的设计首先考虑考虑新增命名空间，但由于 PidNS 和参数的隔离需求都是在每一个容器中存在且唯一，因此参数视图隔离页可以直接依赖 PidNS 实现。在 PidNS 的结构体中添加 kernel_para 用来表示调度参数列表。

对 pid_namespace.h 修改如下

```
struct vkernel_kernel_para{
    unsigned int sysctl_sched_wakeup_granularity; // 参数列表目前为系统调度唤醒粒度,后续可扩展
};

struct pid_namespace {  // 在 pid_namespace 结构体中添加新成员： kernel_para 表示调度参数列表
  ......
    struct user_namespace *user_ns;
    struct ucounts *ucounts;
    int reboot; /* group exit code if this pidns was rebooted */
    struct ns_common ns;
    struct vkernel_kernel_para kernel_para;
} __randomize_layout;
```

在对进程进行初始化的时候，需要对新增的参数 kenel_para 进行初始化，此处初始化为 3ms。

```
struct pid_namespace init_pid_ns = {
    .ns.count = REFCOUNT_INIT(2),
    ......
    .kernel_para = {
        .sysctl_sched_wakeup_granularity = 3000000,  // 3ms
    },
};
```

由于 PidNS 维护了树状的层级结构，由父层级的 PidNS 来创建新的子层级的 PidNS，因此需要对根 PidNS 中的 kernel_para 进行初始化。同时在新创建 PidNS 时使子级

PidNS 继承父级 PidNS 中的调度参数列表 kernel_para。

```
static struct pid_namespace *create_pid_namespace(struct user_namespace *user_ns,
    struct pid_namespace *parent_pid_ns)
{
    struct pid_namespace *ns;
   ......
    ns->kernel_para = parent_pid_ns->kernel_para; // 继承内核参数配置
    return ns;
}
```

这里需要注意的是虽然对根 PidNS 同样添加了调度参数列表，但为了避免可能对宿主机造成的影响，在 PidNS 的层级为 0 即根 PidNS 层级时将直接使用默认的调度参数。这同时保证了参数视图隔离的可行性，并避免对内核进行过多修改。

从用户态通过 ProcFS 对参数进行读写将会通过一系列注册函数来处理，注册函数中包括了参数的文件名、变量名、参数范围、对应的读写处理函数等。其中参数 sched_wakeup_granularity_ns 的变量类型是无符号整数型，且由于参数在上下限要求，因此该参数的读写首先将通过 proc_dointvec_minmax 函数初始化上下限，然后在 do_proc_dointvec 函数获得最终进行操作的变量，最后调用__do_proc_dointvec 来进行最终的读写行为。

参数视图隔离实现的重点在于修改最终读写作用范围。当通过 ProcFS 对参数进行配置时，会进入到函数函数 do_proc_dointvec 中。当函数开始处理参数 sched_wakeup_granularity_ns，需要首先通过 PidNS 提供的接口 tasks_active_pid_ns 获得当前正在运行进程 current 的命名空间，并将最终被处理的参数指向对应 PidNS 内的参数，使最终__do_proc_dointvec 中的读写作用到 current 所属 PidNS 内的调度参数列表上。而如果无法获得 PidNS 或获得的 PidNS 时根 PidNS。从而实现从不同 PidNS 中对参数进行读写都能作用到该 PidNS 内部的参数列表上，即不同容器中对参数的读写都将作用到容器内的参数列表上。

```
/* 当通过 ProcFS 对参数进行配置时，会进入到函数函数 do_proc_dointvec 中。
 * 当函数开始处理参数 sched_wakeup_granularity_ns，
 * 需要首先通过 PidNS 提供的接口 tasks_active_pid_ns 获得当前正在运行进程 current 的命名空间
 * 并将最终被处理的参数指向对应 PidNS 内的参数
 * 使最终__do_proc_dointvec 中的读写作用到 current 所属 PidNS 内的调度参数列表上。
 * 而如果无法获得 PidNS 或获得的 PidNS 为根 PidNS，不做特殊处理，直接调用__do_proc_dointvec
 * 从而实现从不同 PidNS 中对参数进行读写都能作用到该 PidNS 内部的参数列表上
 */

// do_proc_dointvec 函数获得最终进行操作的变量
```

```c
// 最后调用__do_proc_dointvec 来进行最终的读写行为。
static int do_proc_dointvec(struct ctl_table *table, int write,
          void *buffer, size_t *lenp, loff_t *ppos,
          int (*conv)(bool *negp, unsigned long *lvalp, int *valp,
                  int write, void *data),
          void *data)
{
    void * tbl_data = table->data;
    struct pid_namespace *pid_ns = task_active_pid_ns(current);
    // 如果table->procname 指向一个名为 "sched_wakeup_granularity_ns" 的 /proc 文件。
    // 将 tbl_data 设置为指向 PID 命名空间中的 sysctl_sched_wakeup_granularity 字段的地址。
    if(pid_ns && pid_ns->level > 0){
        if(table->procname && strcmp(table->procname,"sched_wakeup_granularity_ns")==0)
            tbl_data = &(pid_ns->kernel_para.sysctl_sched_wakeup_granularity);
    }
    return __do_proc_dointvec(table->data, table, write,
            buffer, lenp, ppos, conv, data);
   }
```

我们所选择的参数主要作用于 wakep_gran 函数，该函数使用参数 sched_wakeup_granularity_ns 计算发生抢占所需要的时间差，wakeup_gran 的返回值将与 vruntime 之间的差值进行对比，若 vruntime 的差值更大则发生抢占。因此该参数仅针对单一线程有效，因此参数的作用范围可以被限制在控制组内，即参数可能被限制在容器范围内而不对其它容器造成影响。

```c
// 对唤醒粒度进行定制
// 如果当前调度任务是一个实体，获取当前任务的pid_ns，
// 如果pid_ns 不为全局，即当前调度器不是全局调度器，就采用容器内自定义的内核参数
static unsigned long wakeup_gran(struct sched_entity *se)
{
    unsigned long gran = sysctl_sched_wakeup_granularity;
    if(entity_is_task(se)){
        struct pid_namespace *pid_ns = task_active_pid_ns(task_of(se));
        if(pid_ns && pid_ns->level > 0){
            gran = pid_ns->kernel_para.sysctl_sched_wakeup_granularity;
        }
    }
    /*
     * Since its curr running now, convert the gran from real-time
     * to virtual-time in his units.
     *
     * By using 'se' instead of 'curr' we penalize light tasks, so
     * they get preempted easier. That is, if 'se' < 'curr' then
```

```
     * the resulting gran will be larger, therefore penalizing the
     * lighter, if otoh 'se' > 'curr' then the resulting gran will
     * be smaller, again penalizing the lighter task.
     *
     * This is especially important for buddies when the leftmost
     * task is higher priority than the buddy.
     */
    return calc_delta_fair(gran, se);
}
```

在对参数 sched_wakeup_granularity_ns 进行访问前，首先需要通过 PidNS 提供的接口 tasks_active_pid_ns 获得当前所需判断的调度实体 se 所在的命名空间，然后将计算时使用的参数 gran 更换为命名空间内参数列表中的参数值，从而实现不同命名空间内的线程使用不同的参数列表，即不同容器中线程运行过程中对参数的使用都将依赖于容器内的参数列表。

CHAPTER 4

基于虚拟内核的容器内存管理策略的设计

4.1 虚拟内存问题分析

虚拟内存(Virtual Memory)是计算机系统内存管理的一种技术。 它使得应用程序认为自己拥有一个连续完整的地址空间(虚拟地址空间)，而实际上物理内存通常被分隔成多个内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。它为每个进程提供一个一致的、私有的地址空间,让进程产生独享主存的错觉,降低了程序员对内存管理的复杂性，同时可以保护每个进程的地址空间不会被其他进程破坏,提高了系统的安全性。

在多容器环境下，由于不同容器应用共享宿主内核机制，容器缺少独立的虚拟内存管理机制，使得容器应用无法个性化定制自身虚拟内存管理策略。更为严重的是，在宿主机限制系统虚拟内存总量时，所有容器共享这些虚拟内存限制，如果存在恶意容器过度占据虚拟内存，会严重影响应用的性能与稳定。另一个问题是，不同容器应用在虚拟内存超配策略上表现出不尽相同的需求，而当前容器架构难以支持虚拟内存超配策略定制化。

4.2 虚拟内存管理策略总体设计

由于在虚拟内核框架下大部分内核机制仍然在容器间共享，容器定制化的内存管理策略可能会与宿主机策略相互影响，所以在实现容器内存管理定制化时需要考虑以下几个问题：

（1）容器策略配置冲突问题。当容器的内存管理策略配置不一致时会产生冲突。

（2）容器共享内存决策问题。由于多个容器共存于同一宿主机之上，容器定制化自身内存管理策略时，容器之间共享的内存同样会面临配置冲突的问题。

（3）容器内存管理策略自适应问题。由于不同容器具有不同的内存使用模式，在不同场景下，需要支持容器自适应地调整内存管理策略，以最大限度地发挥应用性能。

为了解决以上问题，我们在虚拟内核架构的基础上设计了基于完全隔离的容器内存管理机制、基于投票的容器内存共享冲突决策方法以解决配置冲突问题。此外，为了适应不同容器应用负载，我们在透明大页策略和虚拟内存超配策略的可定制化基础上，实现了用户态策略自定义组件支持用户设计应用特定的内存管理策略自适应方案。

4.2.1 基于完全隔离的容器内存管理机制

容器作为一种操作系统虚拟化工具，本质上是宿主操作系统上的进程。在面向容器的内存管理策略个性化定制中，容器既需要被宿主机内核策略管理，又需要有自身的内存管理策略以最大化应用性能。然而，当容器内存管理策略与宿主机策略不一致时，就会出现冲突问题。如图 4.1 所示，如果宿主禁用虚拟内存超配机制，而容器开启虚拟内存超配机制，由于宿主机机制作为全局策略，容器应用的虚拟内存同样会被全局策略所限制，而容器自定义策略则不能有效发挥作用。此外，当宿主全局禁用大页机制时，相应的缺页中断分配大页机制与 khugepaged 线程也会被关闭，而容器大页机制却依赖于这两种内核机制。因此，如何解决配置冲突是第一个需要考虑的问题。

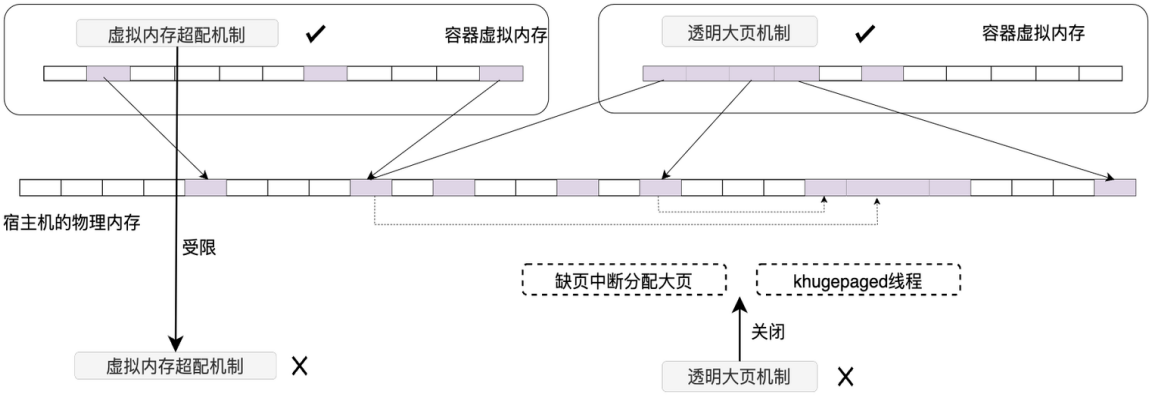

图 4.1 内存管理策略冲突示意图

基于完全隔离思想的容器内存管理机制为每个容器提供一套完全独立的内存管理机制，包括独立的透明大页管理和独立的虚拟内存管理机制。针对独立的透明大页管理机制，虚拟内核需要有独立的透明大页管理参数、缺页中断和 khugepaged 线程逻辑支持。但是，为了降低独立管理机制的复杂度，虚拟内核的缺页中断和 khugepaged 透明大页支持逻辑可以复用宿主物理内核逻辑。而仅需要对容器的缺页中断进行重定向，缺页中断处理大页时处理决策基于虚拟内核管理参数，处理逻辑基于宿主内核逻辑。

此外，khugepaged 线程逻辑同样需要重定向，并且被设置为一个内核常驻线程，以支持容器透明大页机制。需要注意的是，该常驻线程对于关闭透明大页机制的容器不起作用，当容器或宿主内核关闭透明大页机制时，该常驻线程不会增大内存访问延迟。

4.2.2 基于投票的容器共享内存冲突决策

容器作为宿主机上的进程，拥有独立的虚拟地址空间，但它们使用的物理内存是全局共享的。容器支持透明大页策略时会出现页面共享的情况，一方面，目前透明大页机制只作用于匿名页，匿名页在进程间是存在共享状态的，如 fork 系统调用产生的子进程与父进程间匿名内存是共享的，另一方面，对文件缓存的透明大页支持内核社区目前在持续推进，在文件缓存透明大页机制下，容器间页面共享更为频繁。而在容器间存在页面共享时，支持容器个性化定制透明大页管理策略就需要解决共享内存的策略冲突问题。如图 3.1 所示，在 khugepaged 线程探测到右侧容器进程存在一个连续的虚拟地址空间时，会将其对应的物理页转换为连续的透明大页，但是由于部分页面与左侧容器进程共享，而左侧容器如果禁用透明大页机制，则该转换过程涉及的页面则在两种不同且互斥的策略下，导致转换过程存在严重问题。

为了解决容器内存共享冲突问题，我们设计了一种基于投票的容器内存共享冲突决策机制。在容器环境下，每个进程所使用的物理页面会被物理内存审计机制统计到容器对应的 memory cgroup 中，以实现容器物理内存使用量的资源限制。对于容器间共享的页面，内核基于先分配者统计机制，即首次分配引入共享物理页面的 cgroup 统计该物理页面。所以，为了公平起见，我们基于“统计者优先”的原则解决容器内存共享带来的透明大页定制问题。

当 khugepaged 线程探测到开启透明大页机制的容器进程包含 512 个连续 4KB 页面的虚拟地址区域时，首先遍历这块虚拟地址区域的每一个页面，根据页表获取虚拟地址对应的物理 4KB 页面，如果物理页面存在，则获取对该物理页面统计计费的所有者，并根据所有者的 memory cgroup 查找对应容器，根据容器 pid_namespace 获取对应虚拟内核的个性化配置策略，并为对应策略投票。遍历结束后，根据票数最多的策略决定是否执行透明大页合并机制。

4.3 容器内存管理策略定制化系统实现

在以上设计的基础上，为了支持容器应用个性化定制内存管理策略，本节介绍针对容器内存管理策略冲突问题的独立内存管理策略、投票机制实现，以及针对策略自适应问题的用户态策略自定义组件的实现。

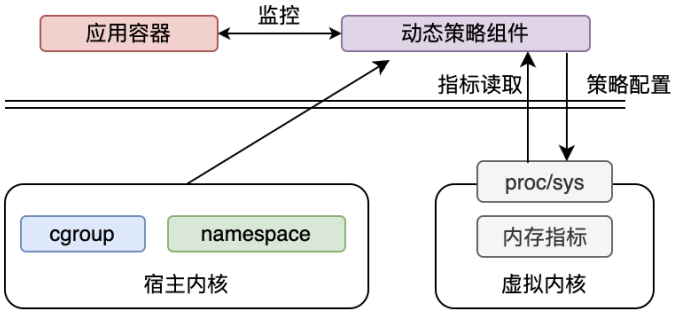

图 4.2 容器内存管理策略组件设计

4.3.1 容器独立内存管理策略的实现

cgroup 中的内存子系统结构复杂，如果要针对容器实现完全独立的内存管理机制难度较大，所以我们结合虚拟内核框架，提出基于参数的最小化内核实现方法，极大地简化了容器独立内存管理机制和投票机制的实现。

对于虚拟内核为用户提供的内存指标，在虚拟内核中完成采集并通过 proc/sys 文件系统提供给用户访问。容器的内存管理策略个性化定制从实现上来讲，包括两个方面：个性化管理参数与策略处理逻辑。以虚拟内存管理策略定制为例，面向用户是内核可调参数，如 overcommit_memory、overcommit_ratio、overcommit_kbytes，基于可调参数，内核一方面将这些可调参数暴漏到用户态，一方面在虚拟内存分配过程中的虚拟内存审计逻辑真正使得参数起作用。

内核面向用户提供了很多可调参数用于控制内核状态，其实现于 proc/sys 虚拟文件系统。除了部分被 namespace 与 cgroup 机制重构外，其它参数均为全局唯一，被所有容器共享，对共享参数的修改将对所有容器生效。但是对其中一些参数，不同容器可能会有不同需求，从而需要不同设定值。比如控制虚拟内存是否可以超配的 overcommit_memory 系列参数。vkernel 为每个容器实现独立的虚拟内存管理，具体包括私有化的 overcommit_memory 系列虚拟内存管理参数，以及针对 vkernel 的

vm_memory_committed 等通用内存管理函数。vkernel 中虚拟内存管理将执行独立于宿主的管理逻辑，从而实现内核参数的重构私有化。

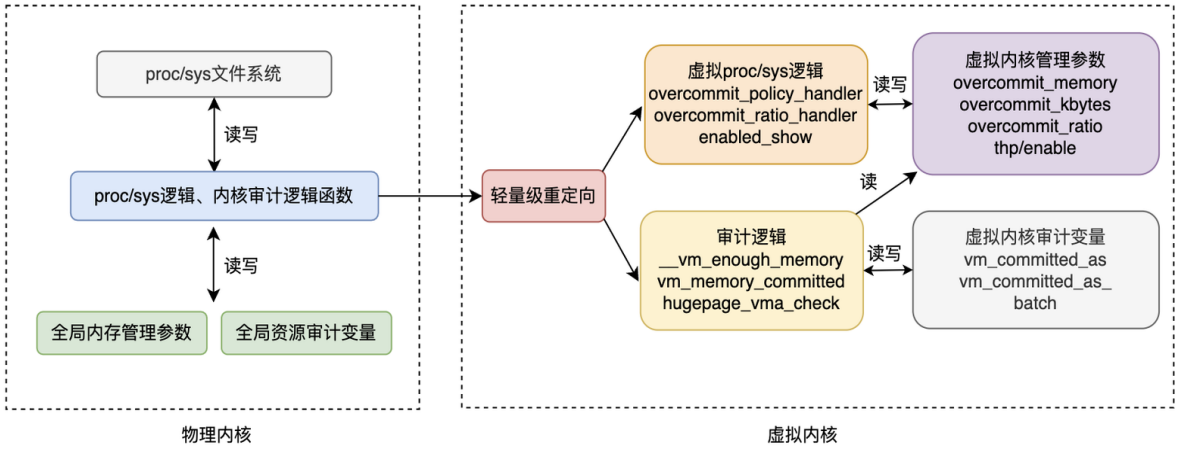

图 4.3 基于参数的容器内存管理策略定制化实现原理

由上可知只需重定向内存管理策略紧相关函数，复用宿主其它内存管理逻辑，以低复杂度实现容器独立内存管理策略和投票机制，就可以解决容器内存管理配置冲突问题，可以满足不同容器场景下的内存管理需求。

CHAPTER 5

面向容器的 Linux 系统调用虚拟化

5.1 背景介绍

随着云计算技术的发展，容器由于其轻量性和高效率等特点，受到越来越广泛的重视，但是由于所有的容器共享主机系统的内核，容器在安全性方面的防护仍旧较弱，具有许多的安全隐患。

在诸多安全隐患中，系统调用的共享对容器安全性影响尤其明显。Linux 内核只提供了统一的系统调用入口以及实现了一致的系统调用。一方面，由于系统调用是共享的，当多个容器竞争调用相同系统调用的时候，产生的剧烈竞争比较容易降低该系统调用的性能。另外一方面，恶意的容器可以通过篡改系统调用入口进行恶意操作，或者通过触发系统调用漏洞产生提权甚至引发宕机。

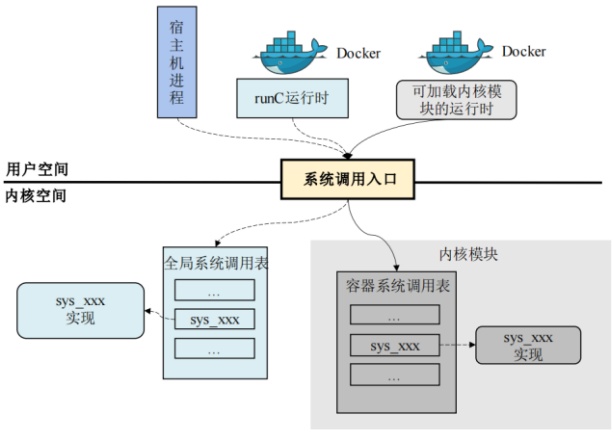

图 5.1 面向容器环境的 Linux 系统调度虚拟化实现架构图

现有的容器虚拟化技术，如 gVisor 和 Kata Containers 等，虽然通过使用用户态内核或者 KVM 虚拟化的方式提高容器的安全性和隔离性，但它们都引入了不容忽视的开销，无法满足现有的容器轻量性和隔离性的需求。

5.2 功能实现

5.2.1 可加载内核模块的容器运行时实现

面向容器环境的 Linux 系统调用虚拟化系统以内核模块的方式实现。为了加载内核模块，需要在容器开始启动后、容器内部应用进程真正运行前，调用内核模块，完成相应功能的初始化以及模块与容器进程的绑定。同时，内核模块要与容器一一对应，伴随着容器的启动停止而动态加载与销毁。

5.2.2 双重 Capabilities 保护

容器创建后，只会使用到少量的 capabilities（默认 14 个）。内核内部有一套自己的 capabilities 安全机制。通常情况下，这些有限的 capabilities 集合在内核的 capabilities 机制下可以保证大部分容器的正常运行，同时也能提高容器环境的安全性。但目前公开的一些漏洞显示，在容器环境下，这些漏洞可以通过利用部分内核 bug 进行提权和逃逸，从而摆脱容器的安全限制，进而可能危害宿主机。

vkernel 运行时在容器应用进程创建前会记录容器初始的 capabilities。之后，容器的所有需要进行权限验证的行为会在内核 capabilities 验证的基础上进行 vkernel 的二次验证，保证容器的所有行为都在初始时的 capabilities 约束集合内，避免越权的行为发生。

5.2.3 容器系统调用表虚拟化实现

容器的系统调用表通过内核虚拟化不同的系统调用入口，移除了容器对全系统调用表的访问权限，代之以每个容器拥有的位于内核空间不同区域的独立系统调用表。内核模块在初始化后，会在内核空间中开辟一块用于存放容器的系统调用表。

容器的系统调用表与全局的系统调用表结构相同，但部分内容有所不同。对于需要进行参数检测的系统调用，会修改系统调用表的表项，对系统调用函数进行二次封装；对于需要屏蔽的系统调用函数，直接返回-1；对于涉及到隔离的子系统资源，会修改重定向该系统调用函数，将其指向内核模块内部的子系统；其余的则默认使用原来的全 系统调用函数。

CHAPTER 6

基于 inode 虚拟化的文件访问控制模块

6.1 背景介绍

inode 虚拟化是 vkernel 项目的核心组成部分，属于 vkernel 的安全防护模块，主要实现的是基于 inode 虚拟化的文件保护。该模块的背景是目前 Linux 的内核强制访问机制如 AppArmor、SELinux 等在为文件访问提供审核管控时，所有文件在被访问时都需要进行相应规则的权限检测，会产生较大的性能开销，资源利用率低。另外，这些文件访问控制也有较为复杂的一套匹配规则，在面临大量的文件访问操作时会带来较大的性能开销，而且在容器内部用户默认拥有 root 权限，可以通过更改 profile 文件来更改 AppArmor 等的相应规则，具有安全隐患。

为了解决上述问题，该模块基于 vkernel 高效高安全性的设计思想，提出了一种基于 inode 虚拟化的文件保护机制，利用内核中已有的权限检测机制，针对 inode 进行虚拟化，只对需要保护的文件进行访问的审核控制，在保证内核安全性与强制访问控制能力的同时，大大地提升了内核的运行效率，提升了系统整体的文件访问速度。

6.2 功能实现

6.2.1 整体结构

为尽量保证 Linux 内部的数据结构不变，选取 inode 中的 i mode 字段中未被使用的比特位作为标识位，用于标识该文件或目录是否在 vkernel 中有相应的权限限制，而每个容器的 vkernel 在初始化时都会初始化一个哈希表，key 为文件 inode 号，value 为相应文件的访问权限，并根据用户自定义的配置文件，将权限信息存储在该哈希表内。当文件被访问时，先检查相应标志位，当标志位为 0，文件在 vkernel 中并未被限制访问，直接通过，当标志位为 1，文件在 vkernel 中有访问限制，则查询哈希表，进行权限检测。

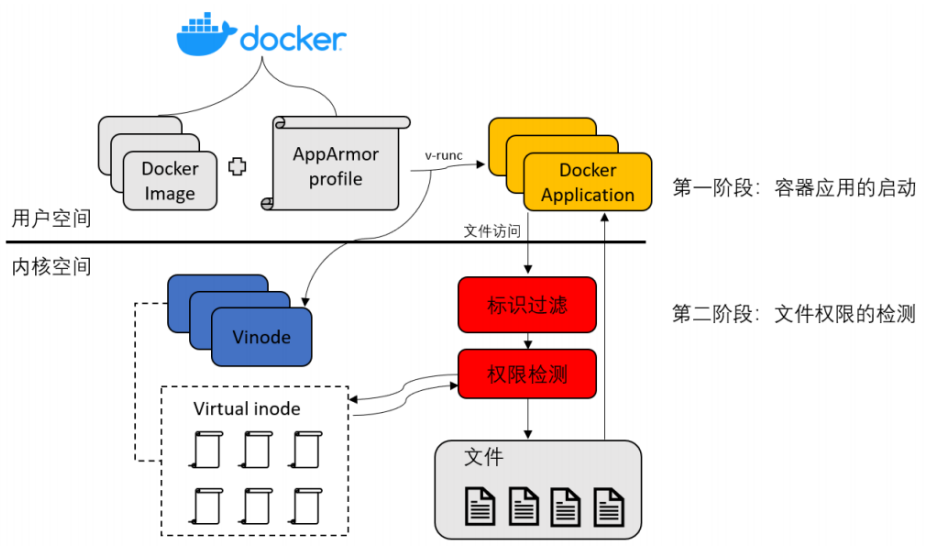

图 6.1 inode 虚拟化架构

6.2.2 对单个文件的权限检测

取一位 i mode 未被使用的比特位，1 表示 vkernel 内有该文件的权限限制，0 表示 vkernel 内无权限限制，在 Linux 内核中的通用权限检测函数 generic permission 中，添加基于 inode 虚拟化的 vkernel 权限检测，将文件访问申请的权限与 vkernel 内哈希表中的信息进行匹配检测。

6.2.3 对目录的权限检测

在 generic permission 中对目录的 inode 的权限检测只是针对该目录的权限进行限制，而并非对目录下的文件进行相应的权限限制，结合实际应用中针对目录下文件的权限限制多是针对 procfs、sysfs、cgroupfs，于是取另一位 i mode 未被使用的比特位用于表示目录在 vkernel 内是否有权限限制，然后在初始化一张新的哈希表用于存储目录的权限控制信息，在 generic permission 中，针对 procfs、sysfs、cgroupfs 文件系统的文件，会逐级向上查看目录是否有相应的权限进行检测。

CHAPTER 7

基于容器镜像的最小化内核定制工具

7.1 背景介绍

与运行自己的操作系统的虚拟机相比，用户可以在主机的同一操作系统内核之上启动多个容器，这使得容器更加轻巧，有着更好的资源利用率和更好的性能。但是，容器性能的提升是以隔离度较弱为代价的。由于主机上的容器都共享同一内核，因此容器彼此之间的隔离是进程级别的，只能由底层操作系统内核通过软件机制来保证。因此，如果有攻击者有权通过某个容器来访问主机的内核并对内核的漏洞加以利用，就会对主机和其他容器的安全带来极大的威胁。

尽管操作系统提供了严格的软件隔离机制试图解决这一问题，例如 Linux 中用以进行细粒度进程权限控制的 Capability 和用于资源视图隔离的 namespace 等等，但是，目前的 namespace 仍不支持对系统调用进行视图隔离，因此，恶意的租户仍然可以通过系统调用访问共享内核，并利用内核漏洞绕过这些隔离机制。例如，“waitid”系统调用中的漏洞（CVE-2017-5123）允许恶意用户运行特权升级攻击，并逃脱容器以获得对主机的访问权限。

vkernel 本质上是一个内核模块，旨在替代 Linux 内核中 seccomp 等的安全机制，减少性能开销，增强隔离性。每个容器都有一个自己专用的 vkernel 为自己服务。容器只能通过 vkernel 来访问主机资源，因此可以在 vkernel 中对容器的行为进行一些限制，提高容器的隔离性。

但是，vkernel 是专用的，不同镜像，甚至同一镜像的不同容器，他们所需要的内核功能也是不一样的。如果每次启动容器时都要手动修改代码，编译模块，那用户的使用体验就会大大降低，也不利于实际应用和自身的推广。所以需要一个 vkernel 的自动构建工具，用户只需要提供容器镜像信息及其对应的配置文件，就能分析容器所需要的最小内核功能，并根据分析结果，自动生成容器专属的 vkernel 模块的代码，构

建出目标容器专用的 vkernel 模块。

7.2 功能实现

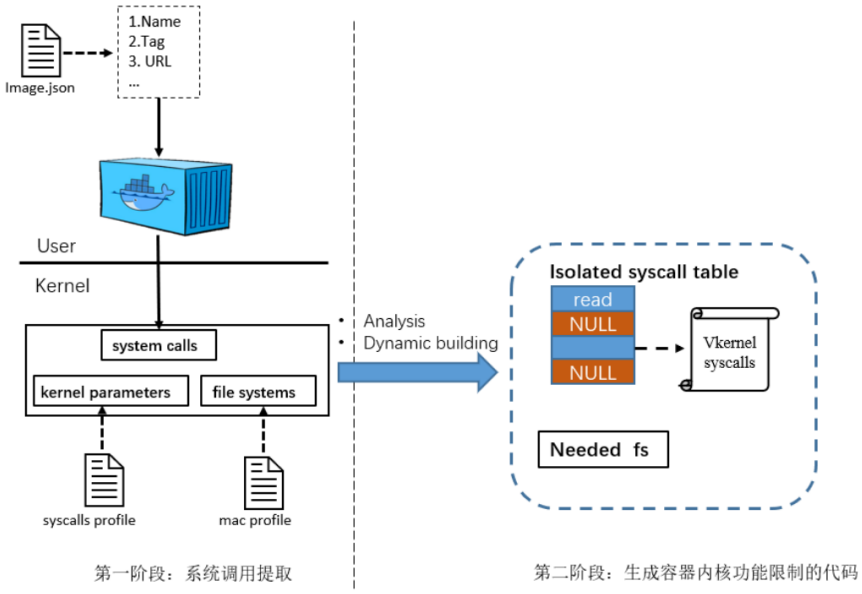

图 7.1 基于容器镜像的最小化内核定制工具架构图

7.2.1 容器镜像系统调用提取

该模块用以对用户提供的容器镜像进行动态分析，以提取其所需要用到的系统调用，供后面生成限制性策略的模块使用。具体实现上，首先需要对用户提供的包含容器镜像信息的 json 文件进行解析，获取容器镜像的名称，版本号，URL 等信息，随后将主机上现存的容器全部停止并删除，再启动目标容器，保证主机上只有目标容器在运行。然后启动 sysdig 监控程序，设定运行时间，并将这段时间内监控的结果以文本的形式保存在指定目录下。最后对监控结果进行分析，提取与目标容器相关的条目，截取系统调用相关的字段并保存。

7.2.2 自动构建 vkernel 模块

工具会对第一阶段的容器镜像分析结果进行解析，结合用户提供的容器的系统调用和文件访存的配置文件，并生成对应的限制性策略的代码。

具体来说，针对系统调用配置文件，由于文件定义了容器允许访问哪些系统调用，因此只需要对配置文件进行解析，并结合第一阶段提取的系统调用，两者取并集，就

能得到允许容器访问的系统调用的结合。其次，配置文件还定义了哪些系统调用在访问时需要进行参数检测，因此系统还会对这些系统调用进行重新封装，插入参数检测的代码，用重新封装好的系统调用去替换系统调用表上的默认的系统调用，实现限制系统调用的策略。而针对文件访存的配置文件，文件中的条目定义了容器对于某个路径下的文件是否具有各类权限（读、写、链接权限等），因此工具会对配置文件进行解析。由于文件路径是 glob 正则表达式，因此首先需要对路径进行解析，得到 Linux 下的标准路径，并且根据路径去读取出该路径下的所有文件或文件夹，随后使用位图来表示这些文件对应的权限。在得到这些信息之后，将文件及对应的权限以键值对的形式存入哈希表中，并自动生成哈希表初始化的代码，容器在访问这些文件时，需要先去哈希表中进行查询，确认自己持有相关权限才可访问，进而达到限制容器对文件进行访存的目的。当系统调用限制和文件访存限制两部分代码生成之后，工具会自动执行编译指令，对 vkernel 源码进行编译，生成.ko 模块文件，供容器运行使用。

CHAPTER 8
项目测试

8.1 面向容器的 CPU 调度虚拟化方案性能评估

8.1.1 测试方法概述

本章以执行效率为性能指标将首先对原生环境下测试结果进行分析，然后从独立任务调度方面，对面向容器的 CPU 调度虚拟化方案进行性能评估。

Hackbench 是 Linux 测试项目中的一项测试工具，通常被用于调度器性能测试工作，这里使用 Hackbench 对 CPU 调度虚拟化方案进行性能评估。其工作原理类似于聊天应用程序，其中通过参数可以设定有多少组实体进行相互通信，以及每个实体通信的循环次数即每个实体发送信息的数量。

测试中，将实体组数分别配置为 50、100、150、200，同时将每个实体发送信息的数量分别配置为 500、1000、3000、5000、7000、9000。此外每项测试将重复 5 次，去除一个最大值和一个最小值后取平均值作为最终结果。通过编译 hackbench.c 得到可执行文件，编写脚本 run_tests.sh 对项目进行测试。chmod +x run_tests.sh 确保脚本有执行权限:。

8.1.2 原生环境中的运行结果分析

原生环境中配置策略和运行环境得到的测试结果如图？？所示，其中 FIFO 和 NORMAL 分别表示 SCHED_FIFO 和 SCHED_NORMAL 策略。此外用 HOST 表示在宿主机上运行，PRI 表示在添加了运行参数“—privileged=true”的特权容器上运行，CAP 表示在添加了运行参数“—cap-add=CAP_SYS_NICE”的具有 CAP_SYS_NICE 权限的容器上运行，DEFAULT 表示不进行额外配置的默认容器上运行。

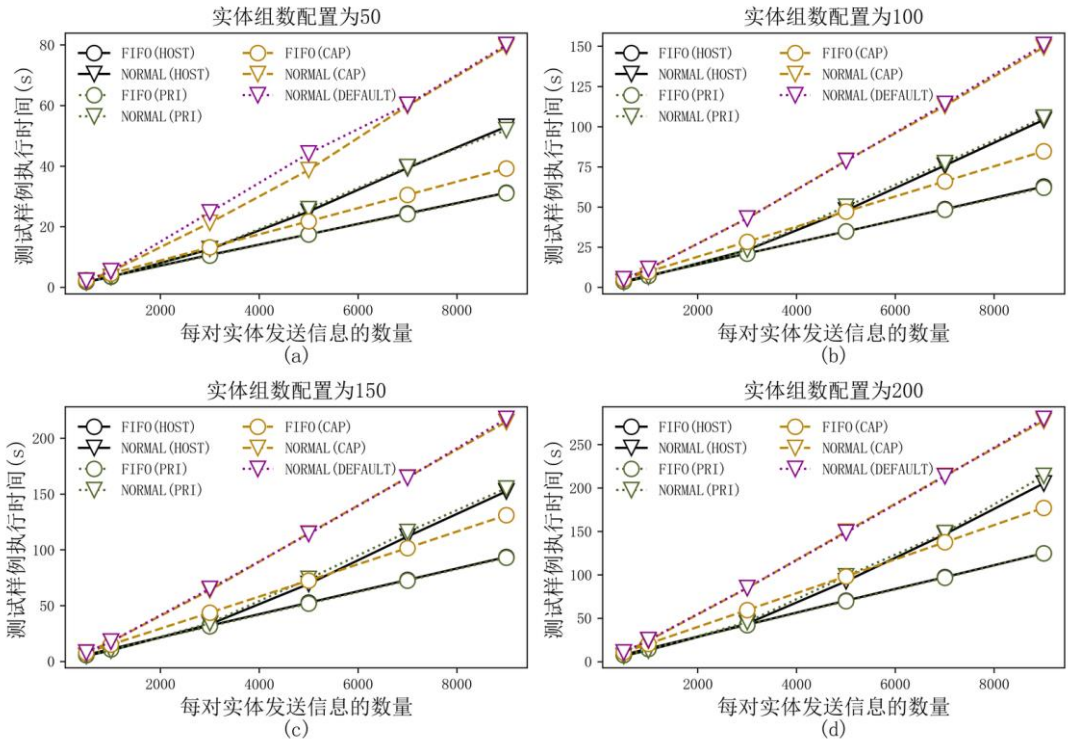

图 8.1 原生环境中运行结果

从图中得到如下结论：

1. 不同实体组数配置下应用受策略配置的影响趋势基本一致。不同配置下随着实体发送信息数量的增加，运行时间展现出的趋势基本一致，hackbench 的结果由实体组数和发送信息的数量来决定，随着发送信息数量的增加，会放大不同策略之间的性能差距。

2. 在每对实体发送信息数量较少时，两者的运行结果接近，SCHED_NORMAL 可能优于 SCHED_FIFO。而随着实体发送信息数量的增加，SCHED_FIFO 优势将越来越明显。这可能是因为随着单一线程的生命周期变长，SCHED_NORMAL 策略为了保证任务间公平而进行的频繁上下文切换将会来带明显的性能开销。

3. 容器内权限的配置可以影响应用的执行效率。当容器配置为特权模式时，容器内的运行结果与宿主机上基本一致，普通容器与仅具有 CAP_SYS_NICE 权限的容器内的运行结果接近，但都明显要差于特权容器和宿主机上的运行结果。这大概是因为在容器没有被配置为特权模式时，将需要在运行过程中进行大量的权限检查，而特权容器将直接禁用安全机制。

8.1.3 调度虚拟化的性能评估

这部分的主要流程是在 CPU 调度虚拟化方案实施的新内核中运行一个普通容器，分别通过将参数“cpu.real_policy”设置为 0 和 1 来表示局部调度器上的调度策略配置为 SCHED_NORMAL 和 FIFO，最终得到的运行结果如图 ？？所示。其中 SCHED_NORMAL 和 FIFO 分别表示容器内局部调度器选择默认的 SCHED_NORMAL 策略和 FIFO 策略。

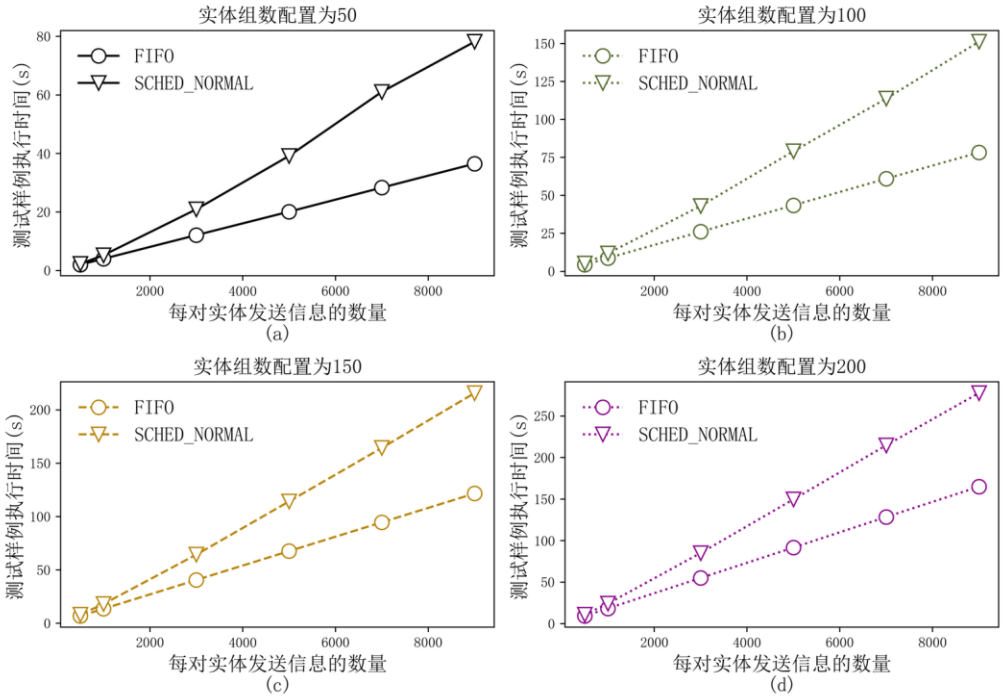

图 8.2 CPU 调度虚拟化方案中配置不同策略的运行结果对比图

从图中得到如下结论：

1. CPU 调度系统虚拟化提供的 FIFO 策略能够明显提高应用的执行效率。在实体发送的信息数量较少时，局部调度器中使用 SCHED_NORMAL 与 FIFO 策略运行结果近似，而随着实体发送的信息数量的增加，FIFO 策略带来的优势逐渐明显。

2. 随着实体组数的增加，FIFO 策略的优化效果将减少。这可能是因为当实体数量增大时，整个系统中可运行任务的数量也将显著增加，此时 FIFO 策略中尽可能久地占用 CPU 资源的调度特性对使得排队靠后的任务等待时间过长，更适合使用保持任务间公平调度的 SCHED_NORMAL 策略。

将以上分析进行整理可以得知，FIFO 策略更适合并行度低的长任务环境，而 SCHED_NORMAL 策略更适合并行度高的短任务环境。

8.2 容器日志隔离测试

容器的日志隔离测试基于不同版本内核进行内核日志读取，分析在不同环境下内核日志读取的执行情况。测试通过在原版本 linux 内核下和我们的 vkernel 内核下执行 dmesg 查看内核日志进行对比，此处创建的均为 linux 容器，在 linux 容器中查看内核日志。

首先在原版本内核上执行容器日志的读写，记录执行情况，可以看到在没有 vkernel 框架下容器能够访问到所有的内核日志，执行结果与主机下直接执行 dmesg 结果相同。说明在原生容器环境下并未实现内核日志在容器中的隔离，原生容器环境中仍然存在隔离性不足的问题。

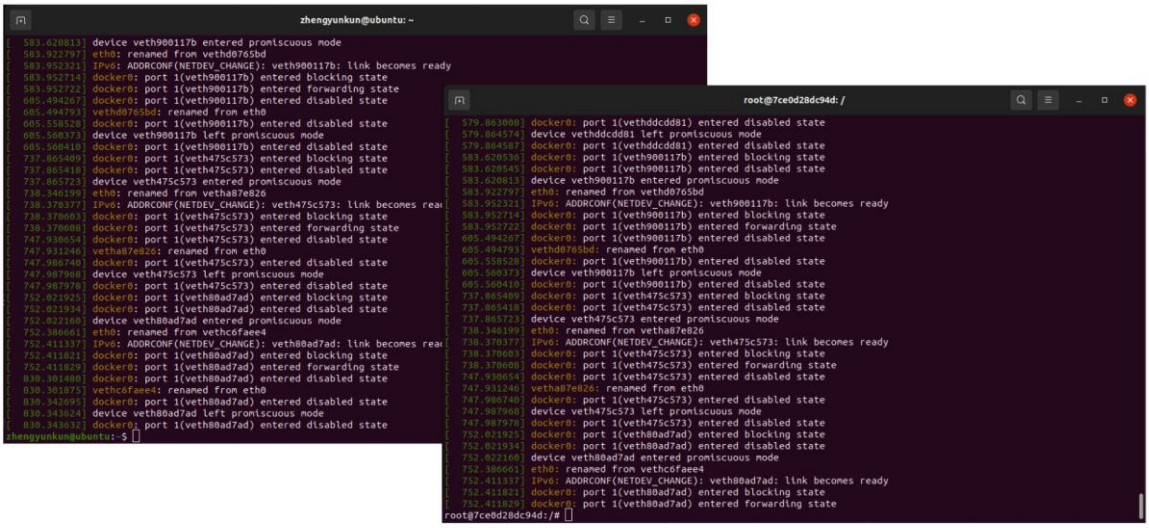

图 8.2.1 原生容器中读取内核日志对比

接下来在实现了内核日志隔离的 vkernel 内核下执行相同的操作，分别在容器中和主机中执行 dmesg 查看内核日志，结果存在明显差异。可以看到在容器中只能看到和 vkernel 相关的内核日志，其他的全局内容在容器中均不可见，实现了容器的日志隔离。

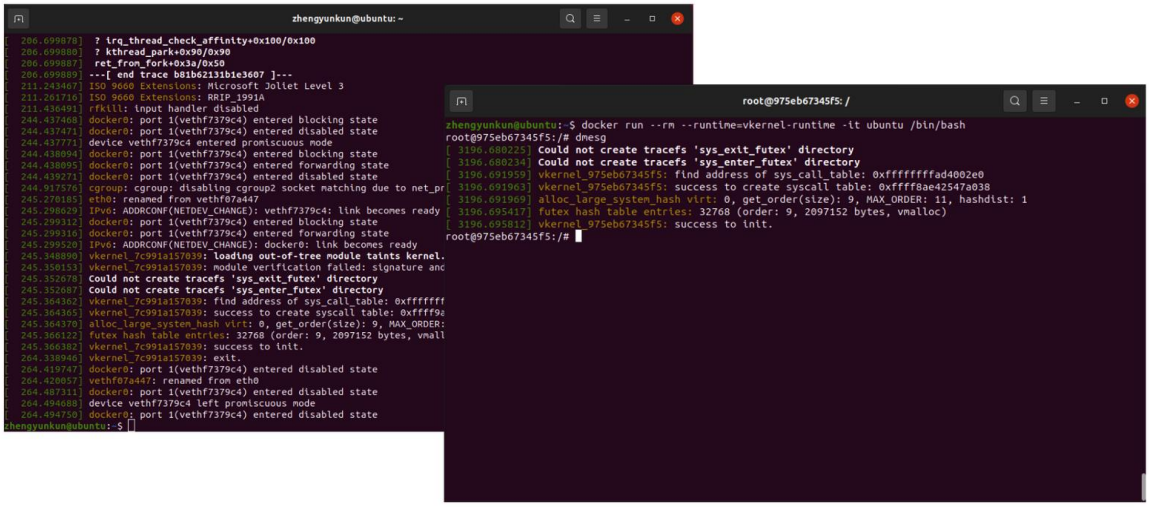

图 8.2.2 vkernel 容器中读取内核日志对比

总的来说，内核日志原本是内核中的全局资源，任何具有 root 用户权限的主体，都能够读取所有的内核日志。但是出于容器安全性和隔离性考虑，我们需要让容器处于一个相对主机隔离的环境中，这就要求我们对主机全局资源对容器进行隔离，加强现有容器环境的隔离性，我们的 vkernel 很好的做到了这一点。

8.3 容器启动效率测试

应用运行测对于容器技术，启动效率一直是一个重要的考虑因素。然而，使用虚拟内核框架可能会对容器的启动时间产生负面影响。为了测试不同的容器在启动时间方面的表现，我们对 ubuntu 容器进行了测试，并将启动时间为原生 docker 的启动时间作为基准进行计算。

图 8.3 容器启动时间测试结果

容器启动时间相对值测试结果如图 8.3 所示，结果表明，基于虚拟内核框架的容器相比原生容器不会对容器的启动时间造成较多的增加。相比之下，gVisor 容器的启动时间增加了大约 52%，主要是由于其用户级跟踪进程的初始化带来的影响。通过对不同隔离方法的容器进行测试，我们可以看到相比其他容器策略，vkernel 在容器启动效率上存在一定的优势。

8.4 真实应用性能测试

此外，为了展示 vkernel 下容器在真实应用运行中的轻量化特性，本文在与宿主机内核相同情况下对 Nginx、Pwgen 两个常用容器进行了运行性能测试。Nginx 是最常用的轻量级 web 服务器以及代理服务器，本节使用阿帕奇测试工具（Apache Benchmark，ab）作为压测工具，在 20 并发数下进行 30K 个请求，以测试 Nginx 容器吞吐率。Pwgen 是无服务器计算中广泛使用的密码生成器，本节测试其生成 1037 个随机密码消耗的时间。

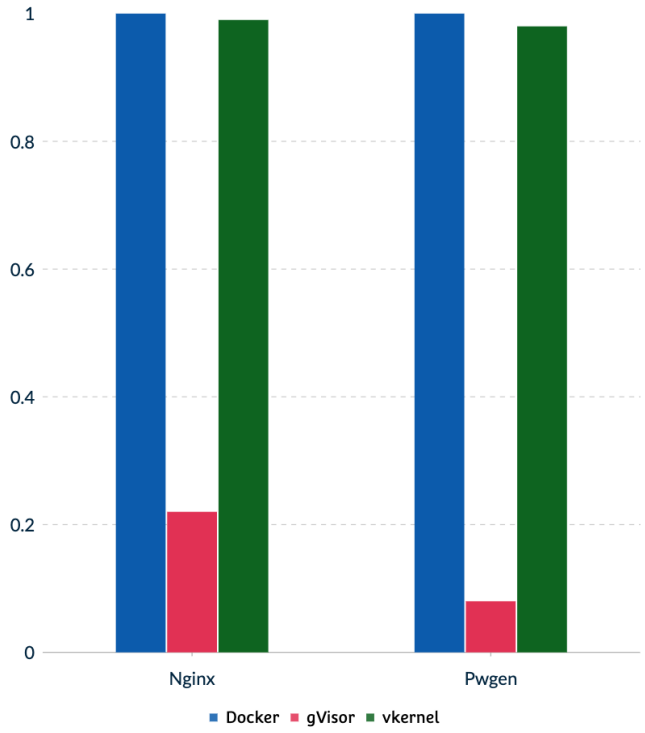

图 8.3 虚拟内核框架对 Nginx 和 Pwgen 应用性能的影响

应用运行测试结果如图 8.3 所示，gVisor 容器由于其性能开销较大的拦截重定向机制在每个应用上都造成了超过较大的性能开销，相对于 gVisor 技术，虚拟内核框架的轻量级实现对应用运行性能的影响相对于原生 Docker 容器，在每个应用容器上影响都较小，这也说明了 vkernel 容器在真实应用上的轻量化表现。