华中科技大学
Huazhong University of Science and Technology

vkernel：虚拟内核场景下面向轻量级虚拟化的优化技术

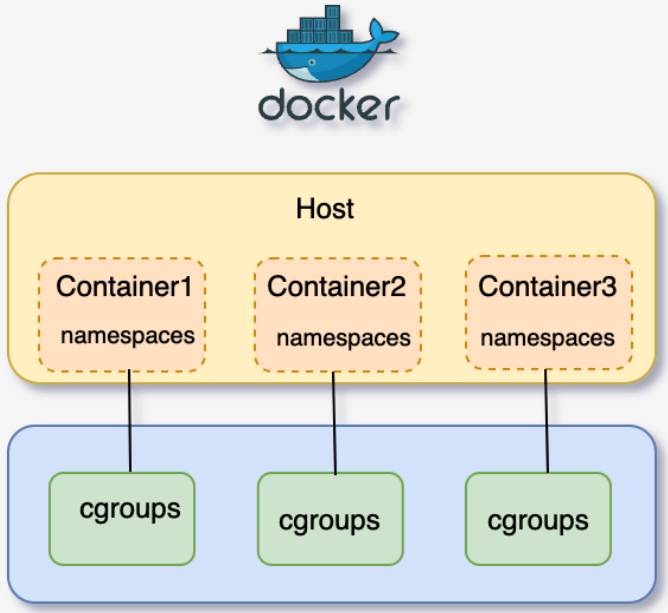
- 华中科技大学
- 队伍成员：郑云鲲、刘佳璇、贾竟一

答辩人 郑云鲲　　指导老师 黄卓、樊浩

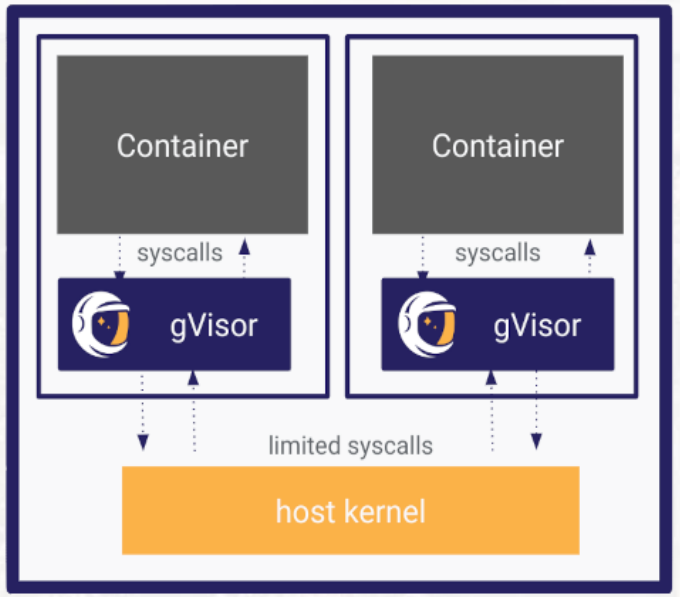
CONTENTS

背景介绍-容器技术的发展

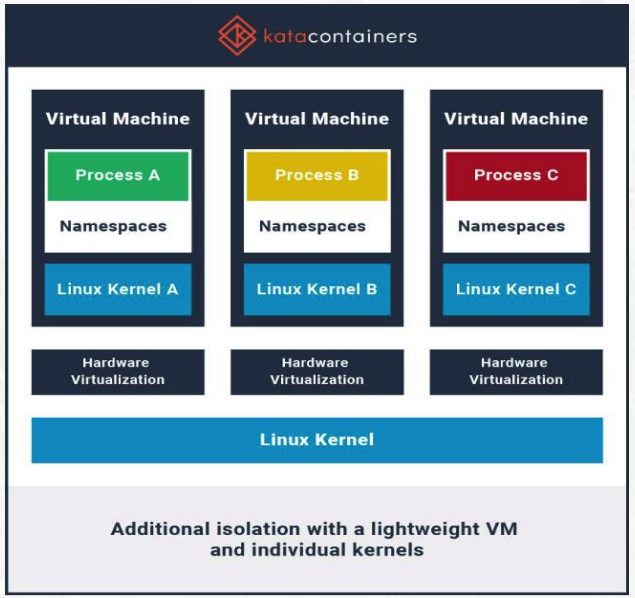
- 基于主机内核的隔离方案
  - 利用linux内核的namspace和cgroup
  - 通过capability, seccomp等机制进行安全隔离



- 基于gVisor的用户态内核
  - 为容器实现用户态内核host kernel
  - 使用进程虚拟化的方法隔离限制对host kernel的访问



- 基于轻量级虚拟机的guest内核（Kata）
  - 与传统虚拟机类似
  - 基于Hypervisor技术提供轻量级的内核



背景介绍-容器技术的发展

- 基于主机内核的隔离方案

本质上还是**多个容器共享主机操作系统内核**，虽然效率高但是存在明显隔离性不足和安全性问题

- 基于gVisor的用户态内核

通过用户态内核虚拟化方法提供强隔离的执行环境，在**用户空间中重新实现了内核服务**，但是这种在用户态实现内核的方案会带来严重的性能开销

- 基于轻量级虚拟机的guest内核（Kata）

基于虚拟机的安全容器方案，为每个容器**提供私有的轻量内核**，解决了传统容器共享内核的安全个隔离性问题，但是以牺牲容器性能为代价

现存问题-隔离性与性能不足

- 共享主机内核模式：
  - ✔□轻量化
  - ✔□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

容器隔离性不足

- 容器技术进一步发展的挑战：
  - ✔单个主机上的容器数量和复杂度不断增加
  - ✔不同应用对内核机制的个性化需求不同

严重限制应用性能发挥

现存问题-容器安全性

- 系统调用的共享
  - 多个容器竞争调用相同的系统调用时产生的**剧烈竞争**会降低该系统调用的性能
  - 恶意容器可以通过**篡改系统系统调用入口**进行恶意操作，通过漏洞占用系统调用资源

定制系统调用虚拟化系统，减少容器间系统调用的竞争

现存问题-CPU调度系统

- 现有的容器策略

容器本身是一组受控制组和命名空间限制的进程，默认情况下容器并不具有策略和参数的配置权限。

不可配置且共享的参数无法满足应用多样化的调度需求

当根据调度需求，要为应用配置CFS以外的策略时，需要提升容器具有的权限，这时可以通过添加运行参数来实现。

- —privileged=true

  提升为具有root权限的特权容器

- —cap-add=CAP_SYS_NICE

  提升为具有CAP_SYS_NICE能力的容器

  CAP_SYS_NICE可以为进程提供配置CPU调度器上的调度策略和优先级、CPU亲和力等权限。

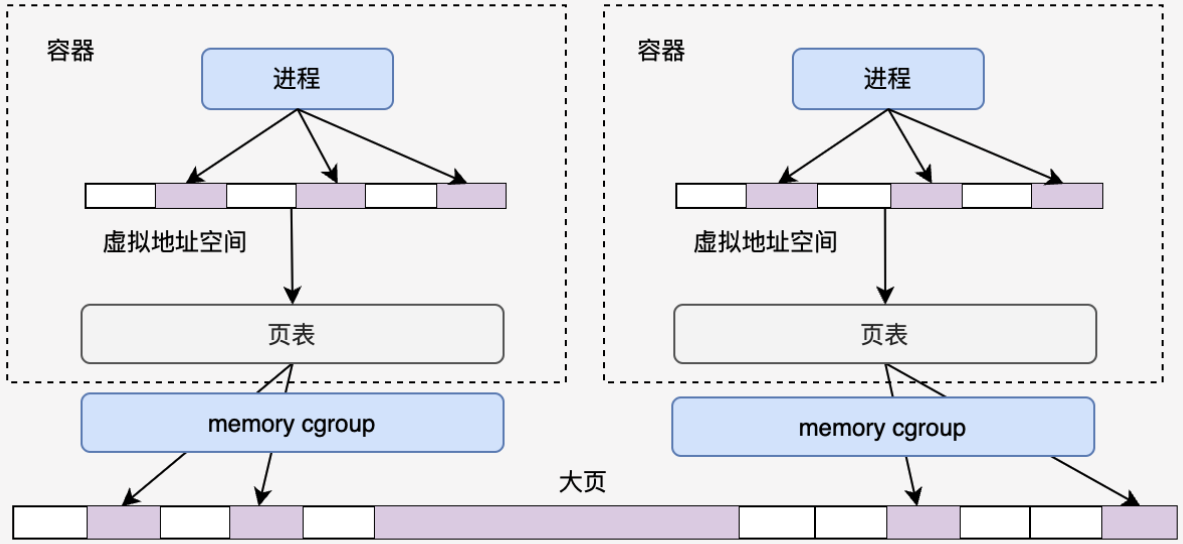
引入安全性威胁和隔离性被破坏的问题

现存问题-容器内存管理策略

- 虚拟内存问题分析

  多容器环境下由于**共享主机内核机制**导致

  - 容器共享主机虚拟内存限制，过度占据虚拟内存影响容器的稳定性
  - 容器架构难以支持超配策略定制



背景介绍-内核功能扩展

- 通过修改内核扩展cgroup: 高效实现定制化策略，但需要重新编译内核，对系统的稳定性和可维护性会造成影响

- 可加载内核模块 (Loadable Kernel Module, LKM):

  优势在于可以在运行时**动态加载**，无需重新编译内核，易于部署，且可加载模块同样位于内核态

- eBPF（扩展伯克利包过滤器）：

  在内核空间中运行可编程的虚拟机，依赖于kprobes机制，性能开销较大，安全性更强

	修改内核	LKM	eBPF
功能	✓	✓	受限
性能	✓	✓	受限
安全	X	X	✓
易部署	X	✓	✓

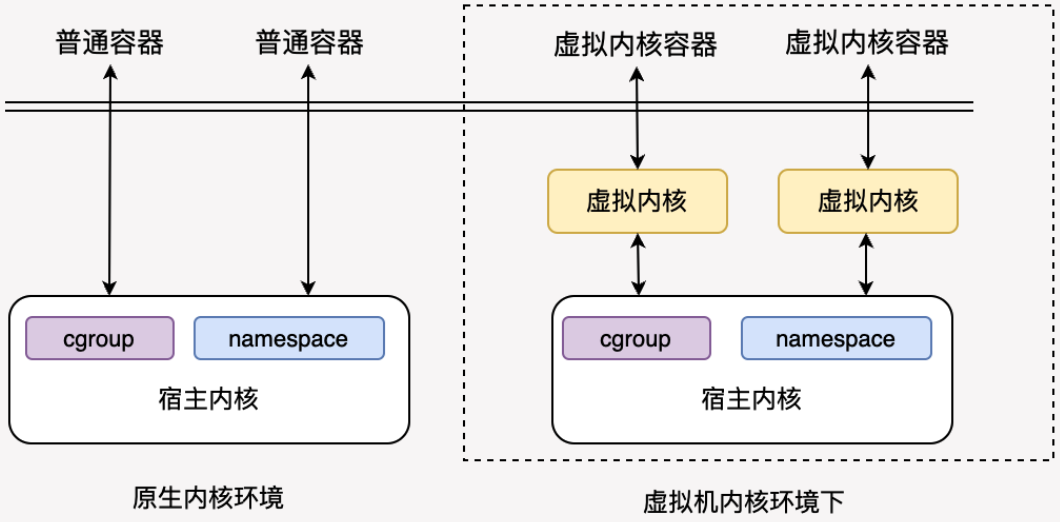
系统设计-虚拟内核架构

- 基于原生容器

  ✔□消除硬件虚拟化层且共享主机内核

  ✔□消除容器对主机内核机制的部分依赖

容器与自身对应的**虚拟内核**直接进行交互，虚拟内核中通过对容器依赖的最小化内核代码与数据实现容器私有化

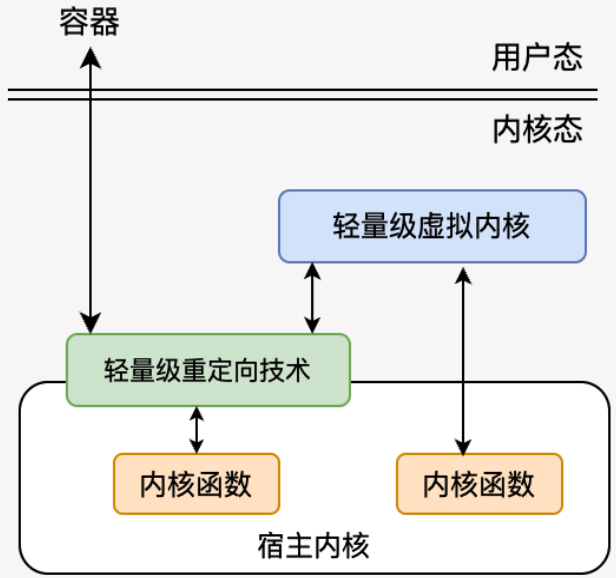


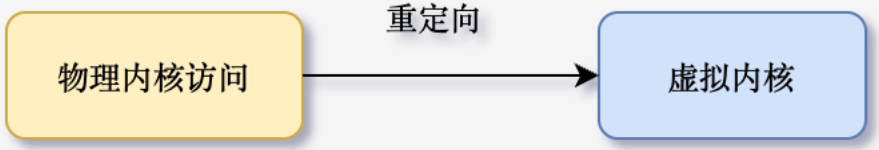
系统设计-虚拟内核架构

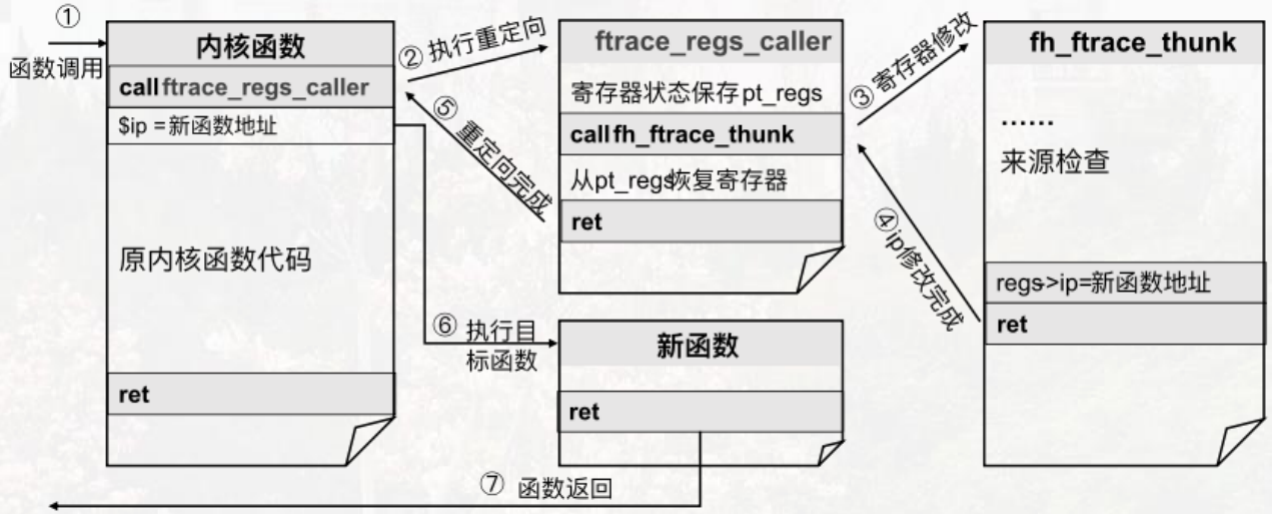
- 轻量级重定向技术

Ftrace: 内核执行流跟踪与重定向技术　　高开销

Inline Hook: 基于jmp指令的重定向技术，无需保存和回复寄存器状态　　极大减少重定向开销







系统设计-虚拟内核架构

- 预期目标

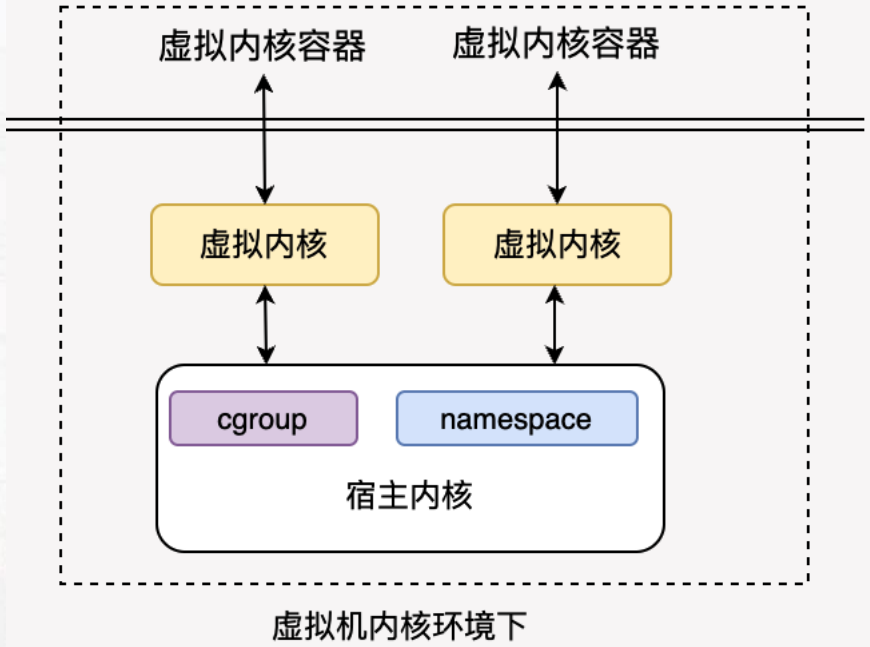
通过上述分析，本项目旨在**对现有的轻量级虚拟化技术进行优化**，实现下列目标：

✔□**高性能**：不同于gVisor和Kata等方案以牺牲性能为代价换取隔离性，本项目希望能够实现接近原生Docker容器的负载性能

✔□**高安全性**：vkernel系统部署在虚拟机环境中，只需要对虚拟机内核进行修改，不会对主机内核的安全性造成影响

✔□**可定制**：用户可以自定义vkernel，在特定容器中支持自定义调度策略

✔□**轻量级**：在实现较强的隔离性和安全性的前提下，能够实现接近Docker容器的启动延迟，不会带来额外的开销

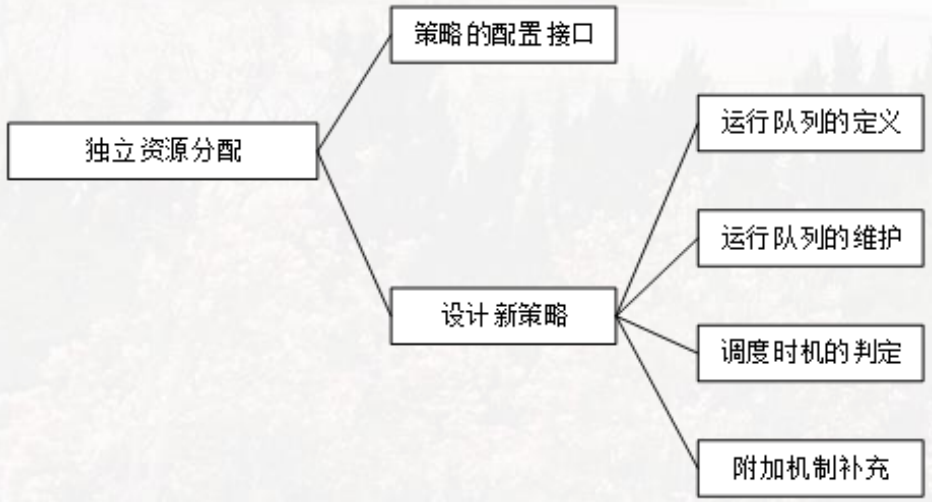


系统设计-面向容器的CPU调度虚拟化

对CPU调度的虚拟化从两个方面进行设计：

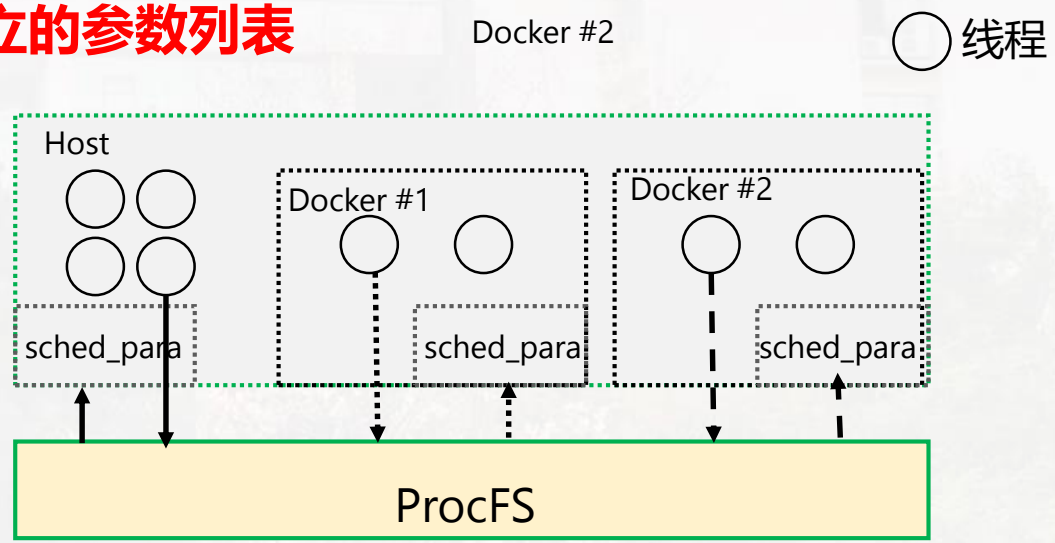
- **独立任务调度**

  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，自定义调度策略



- **参数视图隔离**

  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



系统设计–调度虚拟化之独立任务调度

- 策略的配置接口

考虑策略配置的作用范围？

从线程，运行队列，控制组角度考虑

策略作用于单个线程	设计实现过于复杂 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 容器常被用于单进程多线程环境，运行环境并不复杂
策略作用于单个运行队列	同一个控制组在不同 CPU 上的运行队列使用的策略不同 将任务固定在同一个 CPU 上，影响任务运行，浪费硬件资源。
策略作用于整个控制组	容器本身依赖控制组实现资源控制，与局部调度器具有良好的适配性 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

同时增加"real_policy" 变 量，
指示局部调度器实际策略类型

系统设计–调度虚拟化之独立任务调度

- cpu.real_policy的适配

在调度组的数据结构中增加"real_policy" 变 量
在 对 参 数 文 件“cpu.real_policy”进行读写时，同时更新所属调度组的对应变量。

约定参数定义

当该值为 0 时，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，后续可扩充。.

加入层级特判

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，
可以通过引入对层级的特判

增加写入权限

容器内部不具备对控制组参数进行配置的权限，可以在写权限检查的关键路径上添加判断。
定义vkernel_security_check_file函数，在__mnt_is_readonly 函数中加入对该函数的判断

系统设计–调度虚拟化之独立任务调度

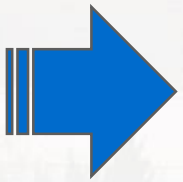
- 独立任务调度方案设计：First-in-First-out（FIFO）

运行队列的**定义**

运行队列的**维护**

调度时机的**判断**

附加机制**补充**



即为新策略**设计运行队列**

即运行队列上任务的**入队、出队和选择**等调度行为
尽可能**复用内核已提供的接口**，避免系统漏洞。

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

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

系统设计–调度虚拟化之独立任务调度

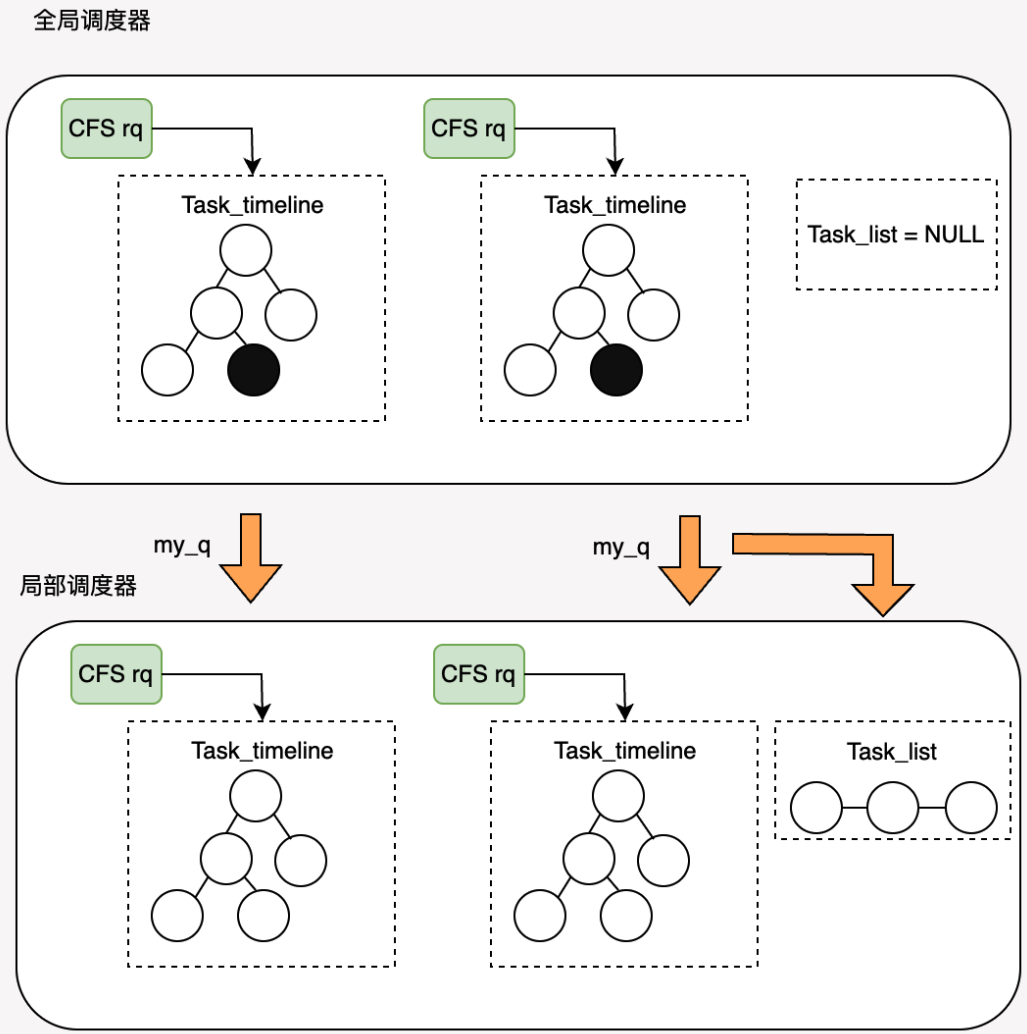
- 运行队列的定义
  - 轻量性

    参考RT调度器中对SCHED_FIFO的实现，同时进一步简化，用单链表方式实现

  - 兼容性

    - CFS调度器上真正的线程放置和选择逻辑都在task_timeline中进行

    - 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tasks_list，只修改了调度实体sched_entity在cfs_rq上的放置方式



系统设计-调度虚拟化之独立任务调度

- 运行队列的维护

- 入队

判断所在调度组使用的调度策略

SCHED_OTHER放置在tasks_timeline上

FIFO调度策略，放置到tasks_list上

- 出队

标识位on_list指示任务所在位置，调度实体在tasks_list上时设置为1，以此确定从哪里出队

- 选择

引入list_nr_running：当前层级tasks_list上任务数量

引入nr_running：当前层级的总任务数量

SCHED_OTHER 且 list_nr_running 不为 0

FIFO 且 list_nr_running 等 于 nr_running

其余情况下从 tasks_timeline 上进行任务选择。

- 调度时机的判定

- 在周期性调度中，仅更新任务运行信息

- 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。

系统设计-调度虚拟化之自适应策略

- 监测指标选取
  - CPU 使用率：任务的自愿和非自愿上下文切换次数相加
  - 上下文切换频率：rq->idle_stamp 获取CPU 的空闲时间戳，rq->clock_task 获取 CPU 的总任务时间
- 周期性策略调整

FIFO	低并行度，长任务	cpu_usage > α_1 && ctx_switches > β_1
CFS	高并行度，每个任务都有机会得到执行	cpu_usage < α_2 && ctx_switches < β_2

定义 MONITOR_INTERVAL，以其为周期进行调度策略的调整

系统设计-调度虚拟化之参数视图隔离

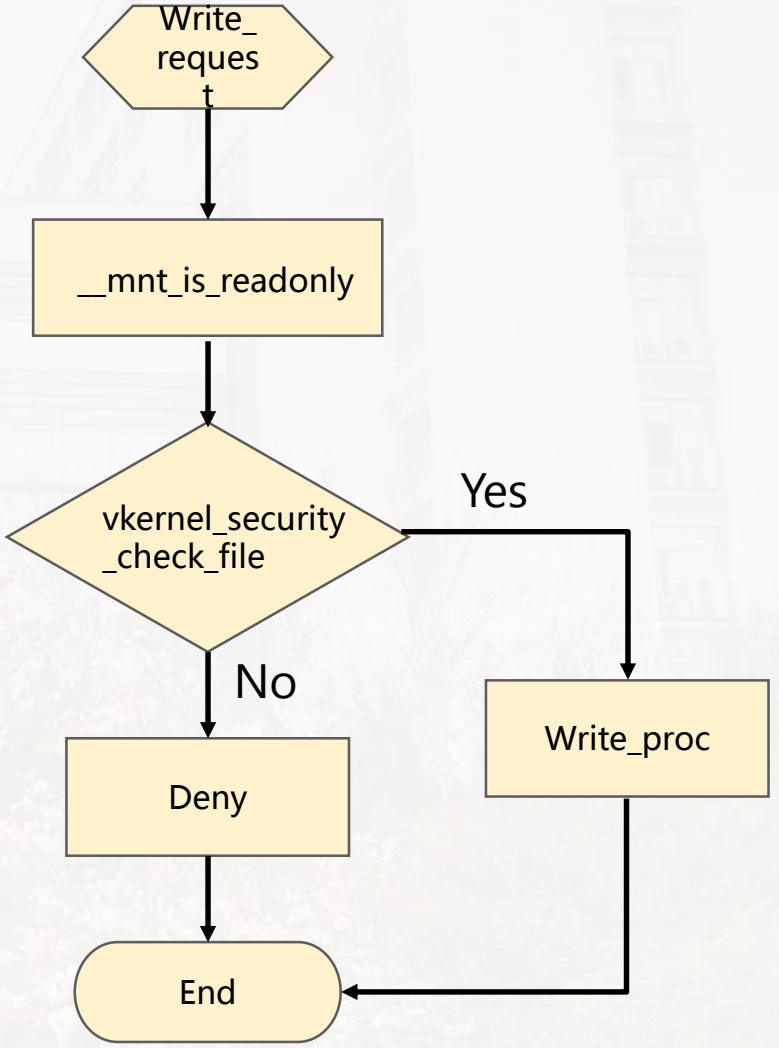
- 参数选择

ProcFS下有大量的参数与调度系统相关，可以选择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

- 参数配置能力

考虑容器对ProcFS文件的权限是只读的，提高权限的方法不安全利用Linux内核提供__mnt_is_readonly接口，通过vkernel_security_check_file函数检查当前文件是否是规定范围内的参数文件，若是则返回错误，实现对参数的配置能力。



系统设计–调度虚拟化之参数视图隔离

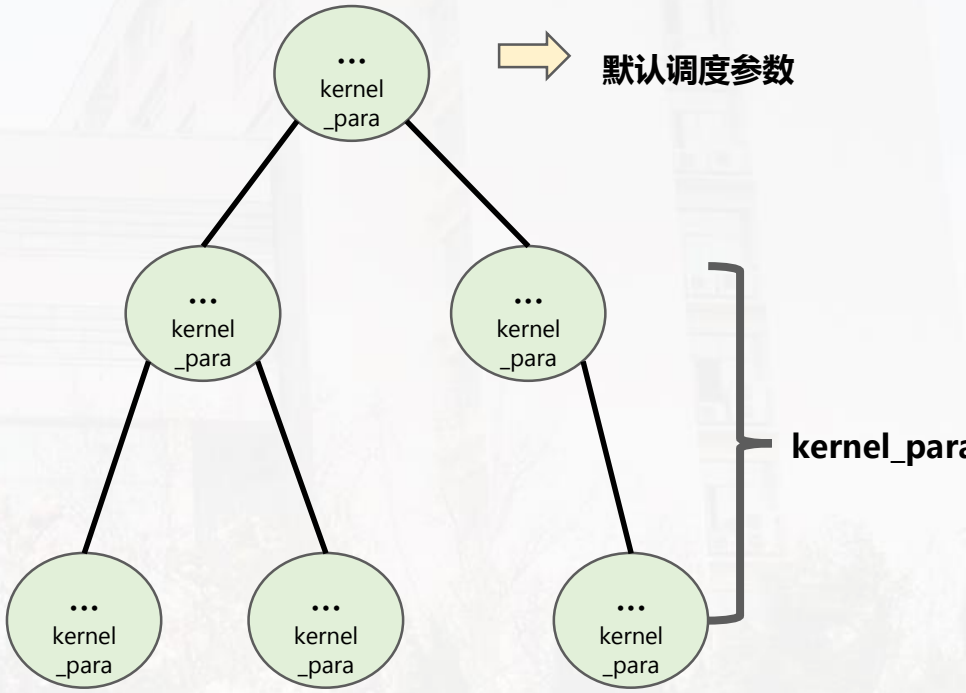
- 参数定义和初始化

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。

- 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。先对根PidNS中的kernel_para进行初始化，子级PidNS继承父中的调度参数列表kernel_para。

- 加入对PidNS层级为0的特判，当层级为0时，直接使用默认调度参数。



系统设计–调度虚拟化之参数视图隔离

- 参数读写
  - 注册函数

    - sched_wakeup_granularity_ns的变量类型是无符号整数型
    - 由于参数存在上下限范围，使用proc_dointvec_minmax函数来初始化参数范围。

  - 读写范围

    - 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，最终被处理的参数指向对应PidNS内的参数，将__do_proc_dointvec中的读写作用到current所属PidNS内的调度参数列表上

    - 当通过 ProcFS对参数进行配置时，会进入到函数do_proc_dointvec中
      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。

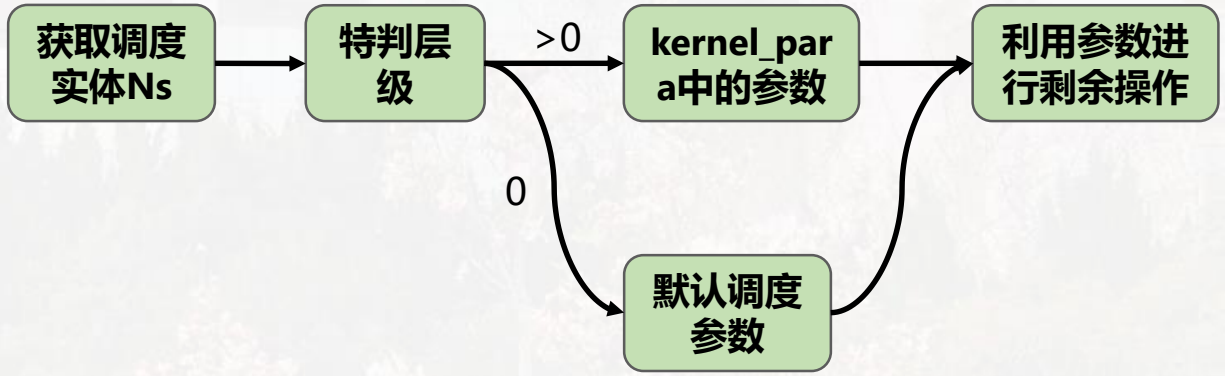
系统设计–调度虚拟化之参数视图隔离

- 参数适配

  - sched_wakeup_granularity_ns参数主要作用于wakep_gran函数，以其为例实现了参数隔离的适配

  - 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，



系统设计-内存管理策略

- 基于完全隔离的容器内存管理机制

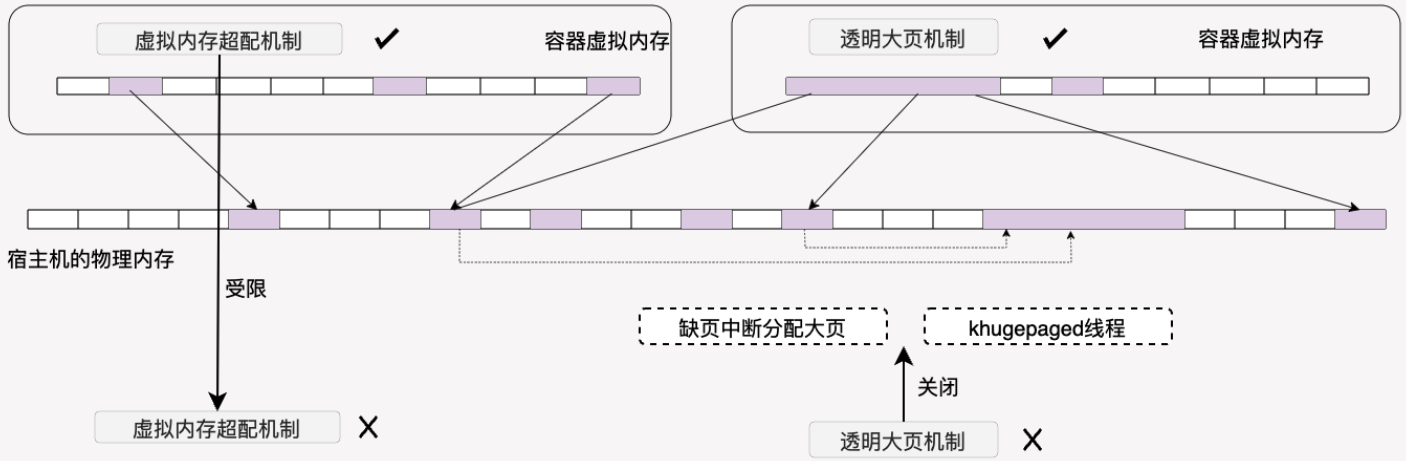
独立的透明大页机制

- 透明大页管理参数、缺页中断和khugepaged透明大页支持逻辑

独立的虚拟内存管理机制

- 虚拟内存超配策略、参数配置等

容器内存管理策略与主机不一致时出现冲突



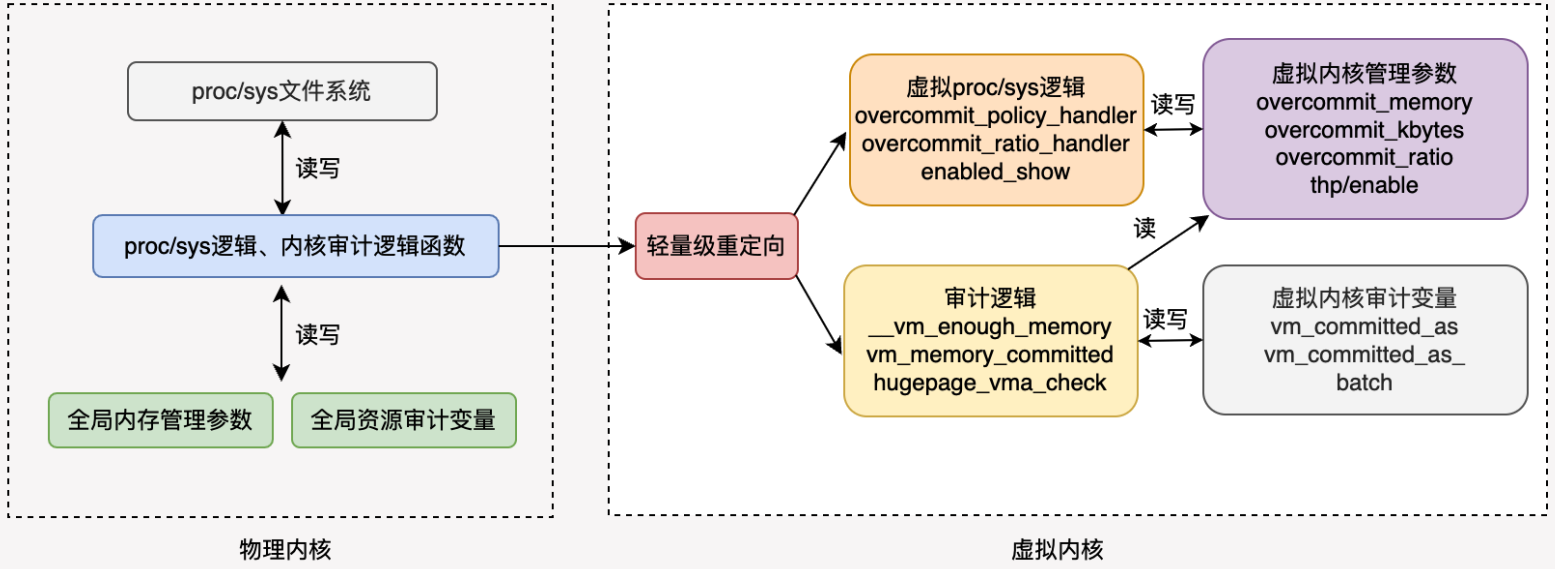
系统设计-内存管理策略

- 个性化管理参数

- 内核可调参数，overcommit_memory、overcommit_ratio等

- 策略处理逻辑

- 重定向内存管理策略相关函数、复用宿主管理逻辑



系统设计-系统调用虚拟化

- 容器系统调用表虚拟化实现

  - 移除了容器对全局系统调用表的访问权限，每个容器拥有位于内核空间不同区域的独立系统调用表

在内核模块初始化后，会在内核空间中开辟一块独立的内存区域用于存放容器的系统调用表

```
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;
};
```

- 系统调用资源虚拟化以futex为例

系统设计-容器日志隔离

- 内核日志结构的改进

  - 使内核日志具备能够区分不同pid_namespace的身份标识

  - 修改内核日志读取函数，增加vkernel进程识别，根据权限计算日志大小并复制到用户空间

```
/* 判断是不是vkernel进程 */
#ifdef CONFIG_VKERNEL
  if(unlikely(task_vkernel_enabled(current))){
    /* 是vkernel进程，隔离读取 */
    msg = log_from_idx2(user->idx);
  }else{
    /* 不是vkernel进程，原始读取 */
    msg = log_from_idx(user->idx);
  }
#else
  msg=log_from_idx(user->idx);
#endif
```

```
static struct printk_log *log_from_idx2(u32 idx)
{
  struct printk_log *msg = (struct printk_log *)(log_buf + idx);


  /* log_namespace与当前进程的pid_namespace进行比对，判断是否具有读取权限 */

  struct pid_namespace *cur_pid_ns;    //pid_namespace struct of current task
  unsigned int cur_ns;
  /*get current namespace*/
  cur_pid_ns=task_active_pid_ns(current);
  cur_ns=cur_pid_ns->ns.inum;

  if(cur_ns==4026531836)          //主机pid_namespace，拥有所有权限
  {
    /* A length == 0 record is the end of buffer marker. Wrap around and
     * read the message at the start of the buffer. */
    if (!msg->len)
      return (struct printk_log *)log_buf;
    return msg;
  }
  if(msg->log_ns==cur_ns)           //如果namespace相匹配，拥有读取权限
  {
    /* A length == 0 record is the end of buffer marker. Wrap around and
     * read the message at the start of the buffer. */
    if (!msg->len)
      return (struct printk_log *)log_buf;
    return msg;
  }
  else
  {
    return NULL;
  }
}
```

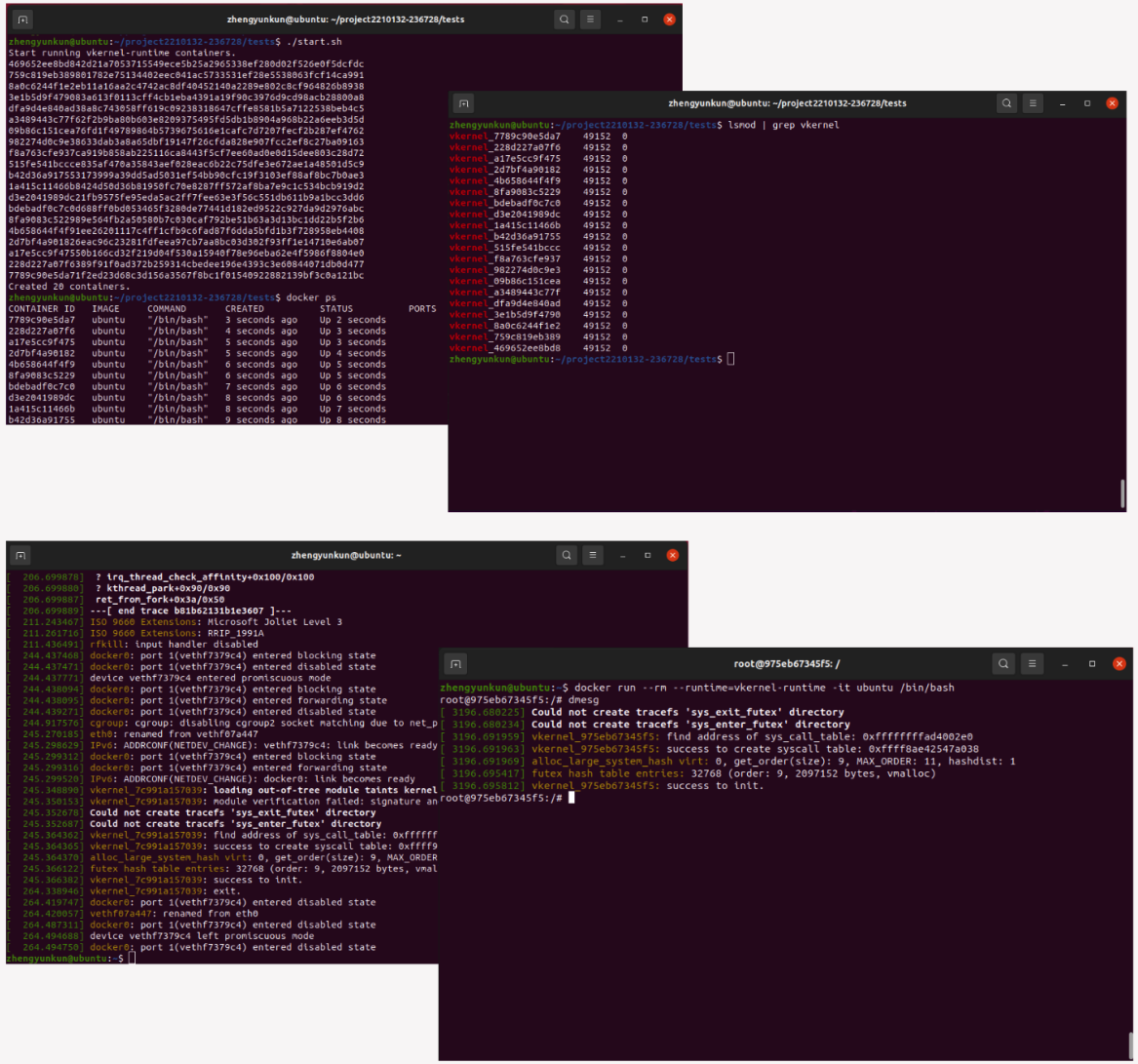
系统评估-隔离性

✔□容器日志隔离：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

- 限制了容器对内核日志的访问，成功实现主机内核日志的隔离化

✔□vkernel内核模块：定制化虚拟内核，为容器提供隔离性的内核环境

- 为容器定制vkernel内核模块，每个模块为容器独享，容器间互相隔离



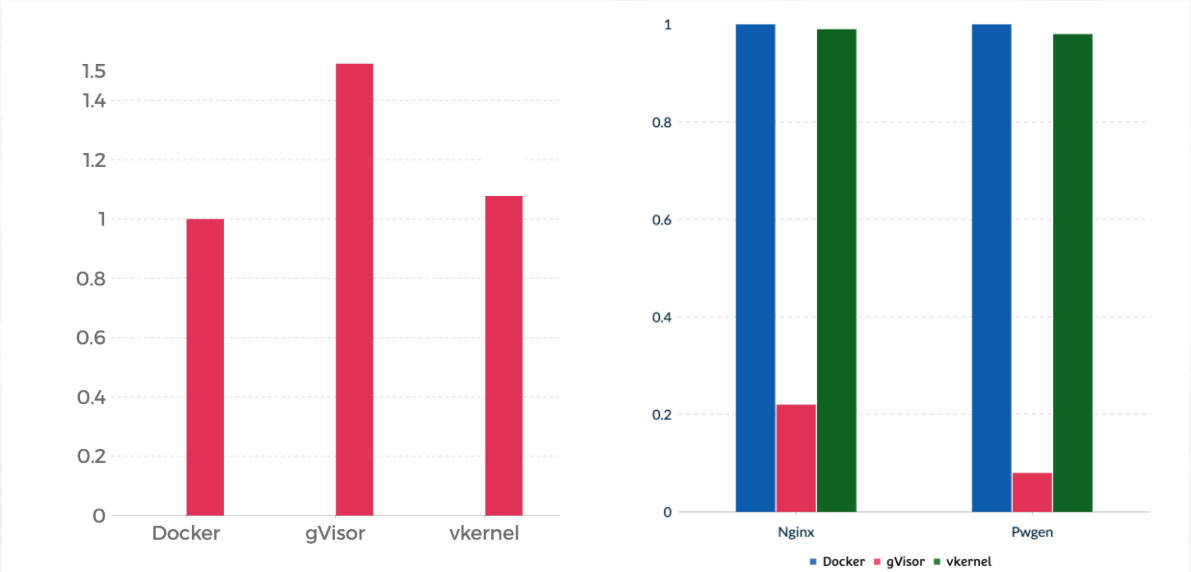
系统评估-轻量性

✔容器启动效率：以相对比例启动时间为测量指标，发现基于vkernel容器**不会对启动时间造成明显增加**

- 相比之下，gVisor的容器启动时间增加了**接近52%，实现了隔离性基础上接近原生Docker容器性能的目标**

✔真实应用性能测试：以原生Docker容器为基准计算相对值，可以看到vkernel容器在Nginx和Pwgen测试中**影响较小**

- 可以说明vkernel容器在真实应用上的轻量性表现



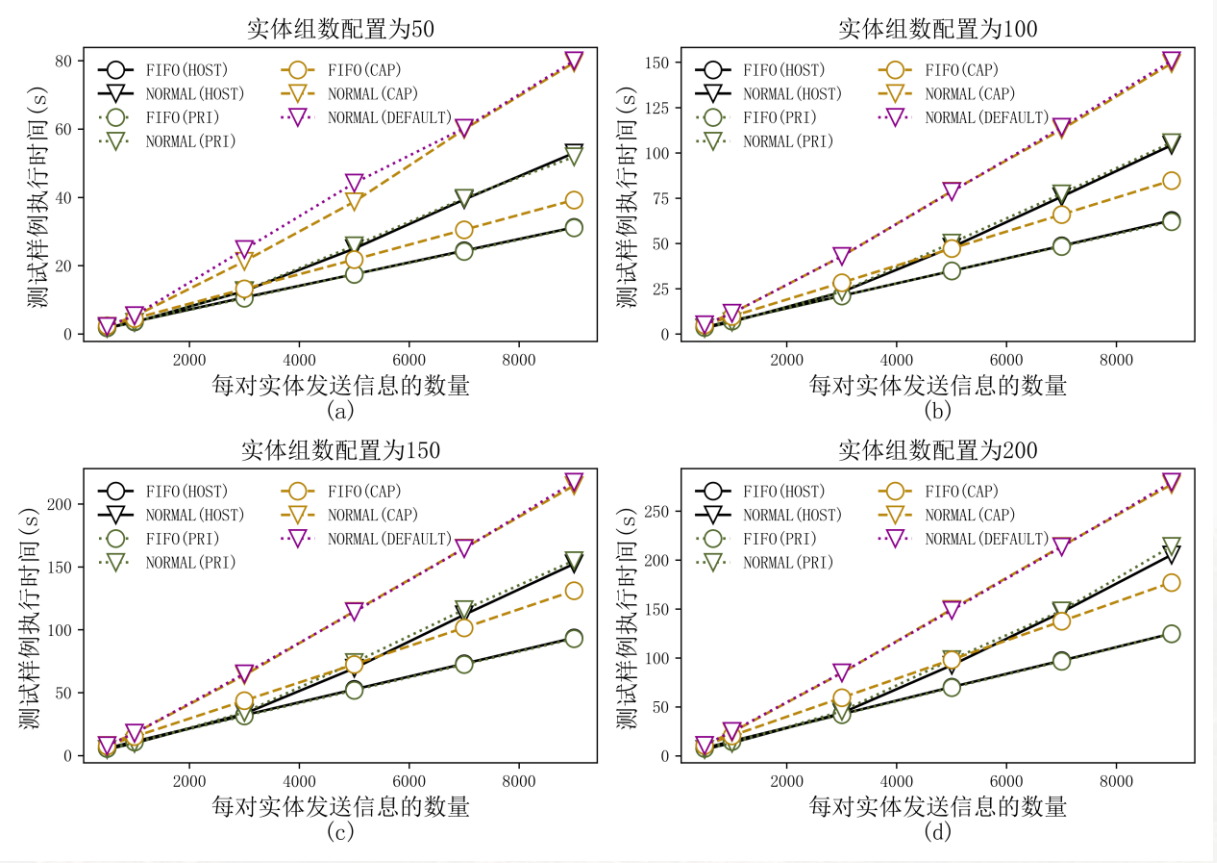
系统评估-CPU调度虚拟化方案性能评估

- 原生环境中的运行结果分析

  - 在每对实体发送信息数量较少时，两者的运行结果接近，而随着实体发送信息数量的增加，SCHED_FIFO 优势将越来越明显

  - 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

- 调度虚拟化的性能评估

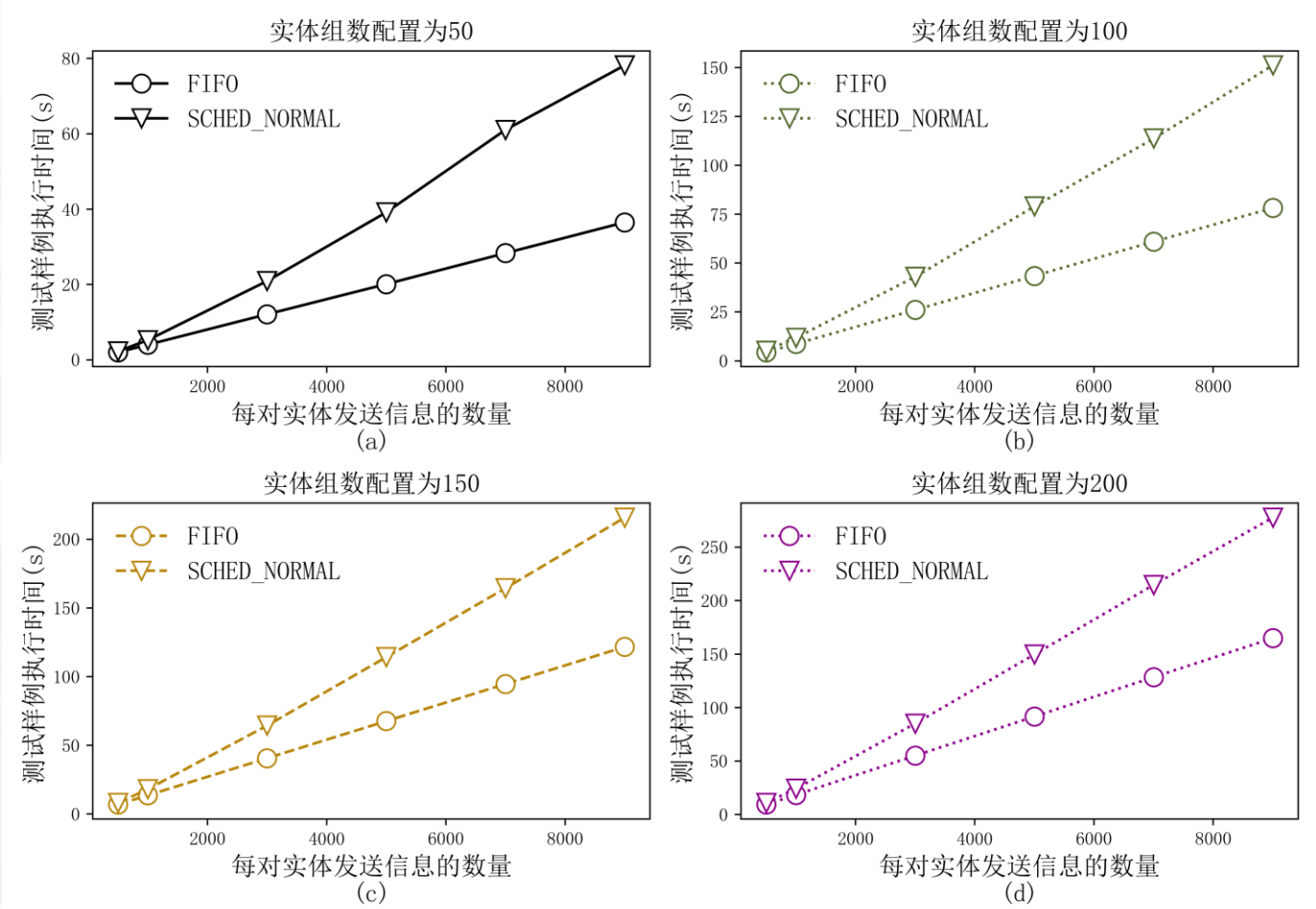


系统评估-CPU调度虚拟化方案性能评估

- 原生环境中的运行结果分析

- 调度虚拟化的性能评估

  - 整体运行测试结果与在宿主机中基本类似，与原本使用单一的CFS调度策略相比，在实体发送信息量较大的长任务场景中，性能有了显著提升



总结—创新性分析

✔□虚拟内核框架设计：实现了内核模块的具体内容，在容器运行时中实现了对vkernel内核模块的支持

✔□CPU调度虚拟化和自适应策略：对容器的CPU调度策略进行较为详细的分析、设计和测试，实现了容器实际运行中调度策略的动态化调整

✔全局资源内核日志的隔离：实现了主机内核日志的隔离，为容器提供了隔离性更强的运行环境

✔容器内存管理策略：对内存管理策略进行了较为深入的分析，完成了方案设计

感谢老师们的聆听！