学习札记

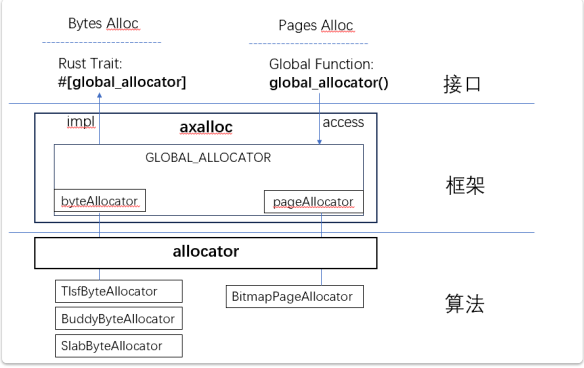
1 基础调用流程

调用组件的流程 :app-->ulib:axstd-->arceos_api-->axhal-->axruntime-->app
axhal组件：_start()(初始化页表)-->rust_entry()-->
axruntime组件: rust_main()(打印logo+基本信息，初始化日志，显示kernel各各段范围，初始化内存分配器，platform初始化，初始化thread调度器，初始化设备和驱动，初始化文件系统网络系统，初始化中断，等待所有cpu启动)-->
apphello_world组件:main()-->
axstd组件: println(macros.rs)-->print_impl-->stdout::Write(io.rs)-->
arceos_api组件:ax_console_write_bytes-->
axhal组件：console write_bytes-->riscv64 put_char-->
sbi: putchar

使用features可以自由选择组件

print_with_color实验
直接在字符串两端加入颜色字符即可

2 动态内存分配支持

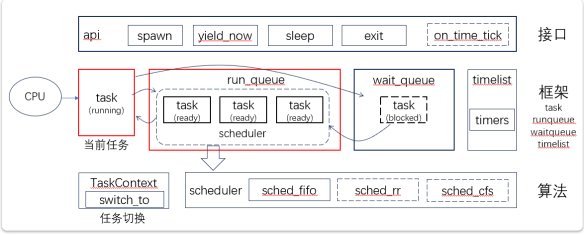


1. 使用rust trait #[global_allocator]支持rust library中内存分配
2. 支持hashmap思路：使用vec实现，根据key的hash值%hashmap容量作为位置存下value即可
3. bump_allocator实现思路：[bytes-used | avail-area | pages-used] 比较简单记录下三个区域分开的位置即可

3 ReadPFlash 引入页表

1. PFlash的作用：Qemu的PFlash模拟闪存磁盘，启动时自动从文件加载内容到固定的MMIO区域，而且对读操作不需要驱动，可以直接访问。
2. 为何不指定"paging"时导致读PFlash失败？ ArceOS Unikernel包括两阶段地址空间映射，Boot阶段默认开启1G空间的恒等映射；如果需要支持设备MMIO区间，通过指定一个feature - "paging"来实现重映射。
3. axhal提供体系结构无关的接口方法set_kernel_page_table写入根页表地址并启用分页

4 启动子任务



接口公开的是runqueue的对应方法

1. spawn&spawn_raw：产生一个新任务，加入runqueue，处于Ready
2. yield_now (协作式调度的关键)主动让出CPU执行权
3. sleep&sleep_until睡眠固定的时间后醒来在timers定时器列表中注册，等待唤醒
4. exit当前任务退出，标记状态，等待GC回收

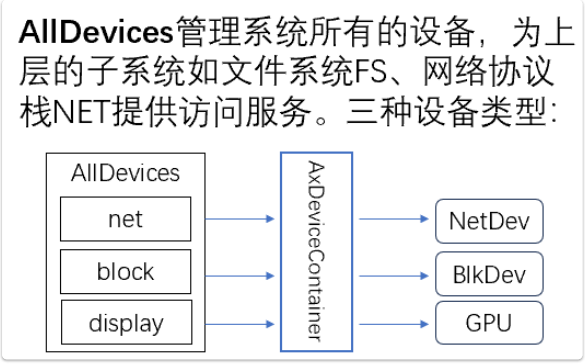
5 协作式调度算法

1. context_switch
2. 协作式调度：任务之间通过“友好”协作方式分享CPU资源。具体的，当前任务是否让出和何时让出CPU控制权完全由当前任务自己决定。

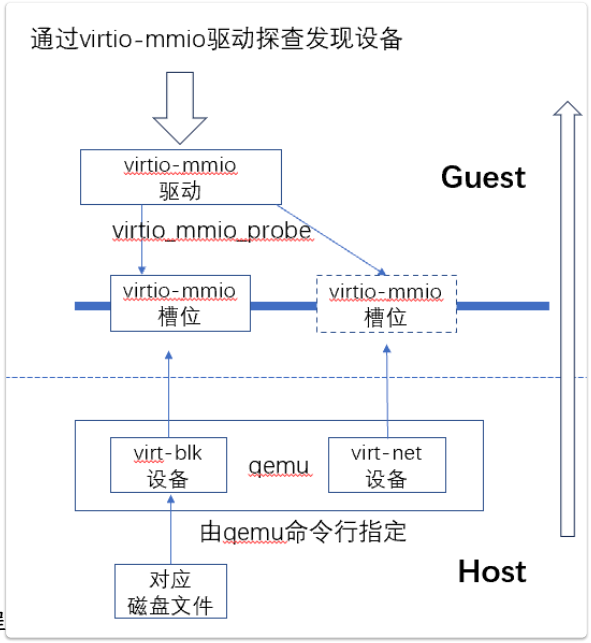
6 抢占式调度

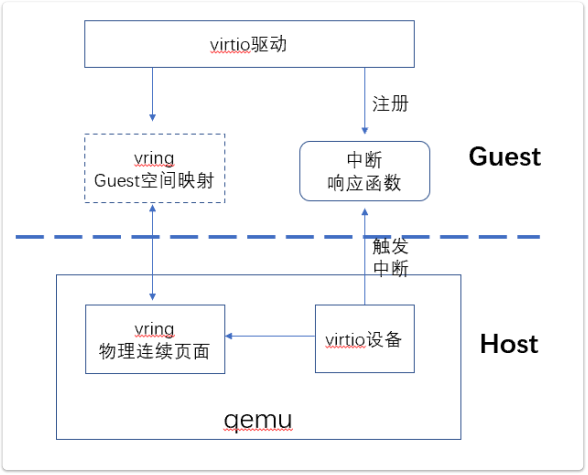
1. 只有内外条件都满足时，才发生抢占；内部条件举例任务时间片耗尽，外部条件类似定义某种临界区，控制什么时候不能抢占，本质上它基于当前任务的preempt_disable_count。
2. 只在 禁用->启用 切换的下边沿触发；下边沿通常在自旋锁解锁时产生，此时是切换时机。
3. 推动内部条件变化(例：任务时间片消耗)和边沿触发产生(例：自旋锁加解锁)的根本源是时钟中断。
4. CFS算法
    1. vruntime最小的任务就是优先权最高任务，即当前任务。计算公式：vruntime = init_vruntime + (delta / weight(nice))系统初始化时，init_vruntime，delta，nice三者都是0
    2. 新增任务：新任务的init_vruntime等于min_vruntime即默认情况下新任务能够尽快投入运行
    3. 设置优先级set_priority：只有CFS支持设置优先级，即nice值，会影响init_vruntime以及运行中vruntime值，该任务会比默认情况获得更多或更少的运行机会。
    4. 任务队列基于BtreeMap，即有序队列，排序基于vruntime

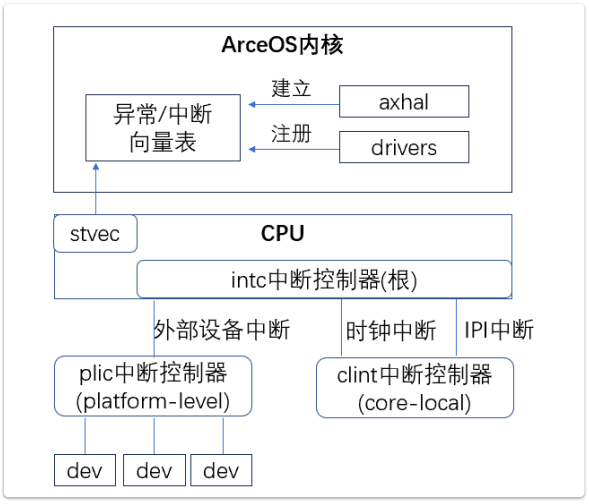
7 ReadBlock



1. virtio设备的probe过程

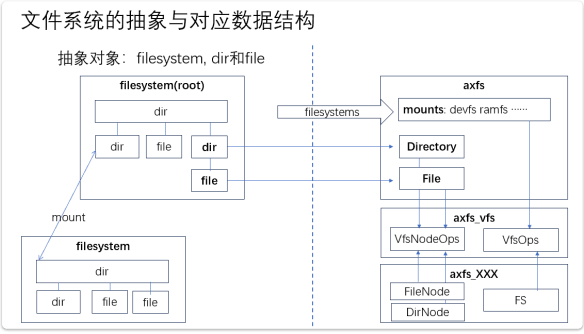




2. virtio驱动和virtio设备交互的两条路：
    1. 主要基于vring环形队列:本质上是连续的Page页面，在Guest和Host都可见可写
    2. 中断响应的通道主要对等待读取大块数据时是有用。
3. 

4. 块设备驱动Trait - BlockDriverOps

8 加入文件系统



1. 文件系统节点的操作流程：第一步：获得Root 目录节点 第二步：解析路径，逐级通过lookup方法找到对应节点，直至目标节点 第三步：对目标节点执行操作
2. rename_ramfs实验
    1. 实验踩坑1：没有添加patch的部分 `axfs_ramfs={path="axfs_ramfs"}`
    2. 实验踩坑2： axfs中rename函数有问题，没有考虑dst的挂载

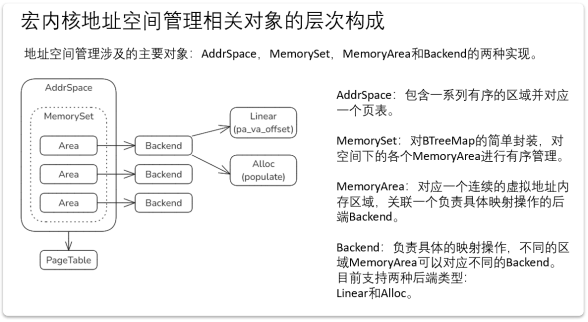
9 引入特权级

1. 分析从Unikernel基础到目标最小化宏内核需要完成的增量工作：
    1. 用户地址空间的创建和区域映射
    2. 在异常中断响应的基础上增加系统调用
    3. 复用Unikernel原来的调度机制，针对宏内核扩展Task属性
    4. 在内核与用户两个特权级之间的切换机制
2. 实例
    1. 为应用创建独立的用户地址空间 涉及组件：axmm

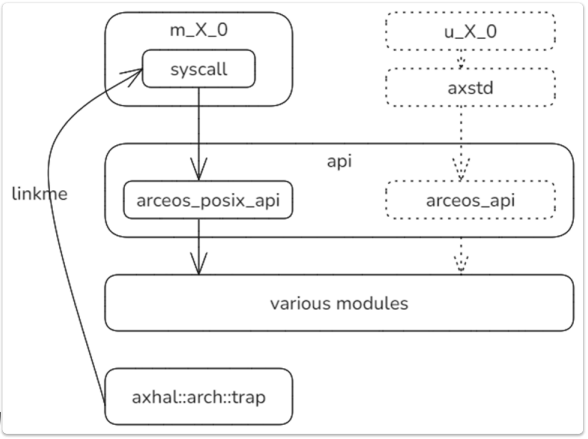
2. 加载应用程序代码到地址空间 涉及组件：axfs，axmm
3. . 初始化用户栈 涉及组件：axmm
4. 创建用户任务 涉及组件：axtask (with taskext)
5. 让出CPU，使得用户任务运行 涉及组件：axtask，axhal

3. 切使用系统调用时使用异常

10 缺页异常与内存映射



1. 地址空间区域映射后端Backend，负责针对空间中特定区域的具体的映射操作， Backend从实现角度是一个Trait
2. 如何让Linux的原始应用（二进制）直接在我们的宏内核上直接运行？
   在应用和内核交互界面上实现兼容。兼容界面包含三类：
   1. syscall
   2. procfs & sysfs等伪文件系统
   3. 应用、编译器和libc对地址空间的假定，涉及某些参数定义或某些特殊地址的引用
3. elf格式加载
   1. 需要注意文件内偏移和预定的虚拟内存空间内偏移可能不一致，特别是数据段部分
4. 初始化应用的栈
5. 系统调用层次结构



6. sys_mmap实现：先读到buf，在用户态页表找一片物理地址，转换为内核态地址，然后把buf里的东西复制过去。

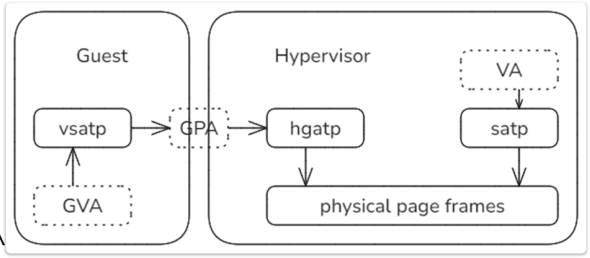
11 Hypervisor

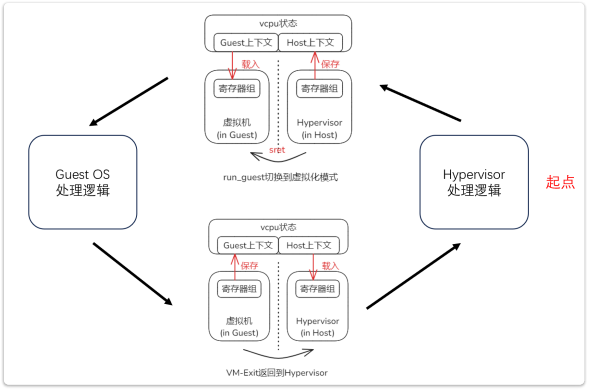
1. I型：直接在硬件平台上运行 II型：在宿主OS上运行
2. Hypervisor支撑的资源对象层次：
   1. VM：管理地址空间；同一整合下级各类资源
   2. vCPU：计算资源虚拟化，VM中执行流
   3. vMem：内存虚拟化，按照VM的物理空间布局
   4. vDevice：设备虚拟化：包括直接映射和模拟
   5. vUtilities：中断虚拟化、总线发现设备等
3. 最简Hypervisor执行流程：
   1. 加载Guest OS内核Image到新建地址空间。
   2. 准备虚拟机环境，设置特殊上下文。
   3. 结合特殊上下文和指令sret切换到V模式，即VM-ENTRY。
   4. OS内核只有一条指令，调用sbi-call的关机操作。
   5. 在虚拟机中，sbi-call超出V模式权限，导致VM-EXIT退出虚拟机，切换回Hypervisor。
   6. Hypervisor响应VM-EXIT的函数检查退出原因和参数，进行处理，由于是请求关机，清理虚拟机后，退出。

4. Riscv64:M/HS/U形成Host域，用来运行I型Hypervisor或者II型的HostOS，三个特权级的作用不变。VS/VU形成Guest域，用来运行GuestOS，这两个特权级分别对应内核态和用户态。HS是关键，作为联通真实世界和虚拟世界的通道。体系结构设计了双向变迁机制。
   H扩展后，S模式发送明显变化：原有s[xxx]寄存器组作用不变，新增hs[xxx]和vs[xxx]
   hs[xxx]寄存器组的作用：面向Guest进行路径控制，例如异常/中断委托等
   vs[xxx]寄存器组的作用：直接操纵Guest域中的VS，为其准备或设置状态
5. 为进入虚拟化模式准备的条件
   1. ISA寄存器misa第7位代表Hypervisor扩展的启用/禁止。对这一位写入0，可以禁止H扩展
   2. 进入V模式路径的控制：hstatus第7位SPV记录上一次进入HS特权级前的模式，1代表来自虚拟化模式。执行sret时，根据SPV决定是返回用户态，还是返回虚拟化模式。
   3. Hypervisor首次启动Guest的内核之前，伪造上下文作准备：设置Guest的sstatus，让其初始特权级为Supervisor；设置Guest的sepc为OS启动入口地址VM_ENTRY，VM_ENTRY值就是内核启动的入口地址，对于Riscv64，其地址是0x8020_0000。
6. 从Host到Guest的切换run_guest每个vCPU维护一组上下文状态，分别对应Host和Guest。从Hypervisor切断到虚拟机时，暂存Host中部分可能被破坏的状态；加载Guest状态；然后执行sret完成切换。封装该过程的专门函数run_guest。
7. VM-Exit原因
   1. 执行特权操作
   2. Guest环境的物理地址区间尚未映射，导致二次缺页，切换进入Host环境
8. simple_hv实验：只需改变sepc寄存器，并将对应值存进对应寄存器

12 Hypervisor 两阶段地址映射

1. 有两种处理方式：
   1. 模拟模式 - 为虚拟机模拟一个pflash，以file1为后备文件。当Guest读该设备时，提供file1文件的内容。
   2. 透传模式 - 直接把宿主物理机(即qemu)的pflash透传给虚拟机。
      优劣势：模拟模式可为不同虚拟机提供不同的pflash内容，但效率低；透传模式效率高，但是捆绑了设备。
2. Hypervisor负责基于HPA面向Guest映射GPA，基本寄存器是hgatp；Guest认为看到的GPA是“实际”的物理空间，它基于satp映射内部的GVA虚拟空间。 GVA–> (vsatp)->GPA–> (hgatp) ->HPA



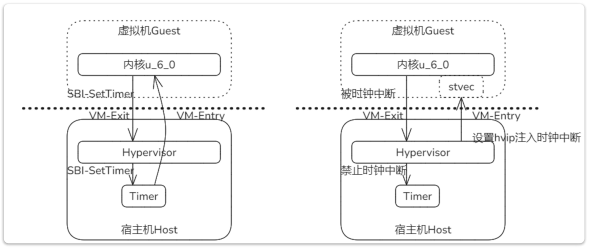
3. Hypervisor的主逻辑包含三个部分：
   1. 准备VM的资源：VM地址空间和单个vCPU
   2. 切换进入Guest的代码
   3. 响应VMExit各种原因的代码
4. 

5. 相比于宏内核多了vm-entry和vm-exit

13 虚拟时钟中断支持；虚拟机外设的支持

1. 物理环境或者qemu模拟器中，时钟中断触发时，能够正常通过stvec寄存器找到异常中断向量表，然后进入事先注册的响应函数。但是在虚拟机环境下，宿主环境下的原始路径失效了。有两种解决方案：
   1. 启用Riscv AIA机制，把特定的中断委托到虚拟机Guest环境下。要求平台支持，且比较复杂。
   2. 通过中断注入的方式来实现。即本实验采取的方式。注入机制的关键是寄存器hvip，指示在Guest环境中，哪些中断处于Pending状态。

2. 支持虚拟机时钟中断需要实现两部分的内容：



    1. 响应虚拟机发出的SBI-Call功能调用SetTimer
    2. 响应宿主机时钟中断导致的VM退出，注入到虚拟机内部
3. 具体实现modules/riscv_vcpu/src/vcpu.rs
4. 管理上的层次结构：虚拟机（VM），设备组VmDevGroup以及设备VmDev。Riscv架构下，虚拟机包含的各种外设通常是通过MMIO方式访问，因此主要用地址范围标记它们的资源。