浙江大学
ZHEJIANG UNIVERSITY

开发文档

赛题名称	Page table using hash table
成　　员	周宇鑫
	赵晨希
	李政
学　　院	竺可桢学院
专　　业	混合班
指导老师	寿黎但
赛题导师	万波　高金皓

摘要

影响虚拟机访存性能的因素较多，从页表的角度来看，随着系统内存容量的快速增长，目前普遍采用的 4KB 页面导致了 TLB 条目数量过多，TLB 换入换出开销较大，无法满足云计算性能要求，迫切需要更大粒度的页面（8KB 以上）支持。而现有的 RISC-V 只支持 4KB 页大小，对内存需求较大的应用不友好。

我们为 Linux Kernel 中搭建了一个 16KB 粒度的虚拟内存系统，实现了 16KB 粒度的 two stage 的内核虚拟化流程，设计并完成了在 stage-2 阶段的哈希寻址模式。这些工作有效提升了地址翻译的效率，从而提升了基于 RISC-V 处理器的 Linux 系统在内存敏感应用场景下的性能，为解决服务器的高性能计算需求提出了一个可行的解决方案，也为扩展 RISC-V 生态做出了贡献。

为了进一步提升系统访存性能，我们希望通过优化 stage-2 页表的页粒度和页表机制来降低 TLB 压力，提升访存性能，间接地降低进程切换开销。所以做出了以下两点优化：①在 RISC-V 的仿真器和 Hypervisor 中实现 16KB 粒度的页表，从而降低 TLB 压力，提升 TLB 命中率，最终提升访存性能。②设计并使用 16KB 的哈希地址转换模式，减少寻址开销，从而进一步提升访存性能。

我们在 LMbench、Redis 等测试集上进行了大量的测试实验，验证了所实现的 16KB 粒度多级页表与哈希页表的功能和性能。实验表明：本项目实现的 16KB 粒度页表能够正常支撑 RISC-V KVM 虚拟机的运行；与原有的 4KB 粒度的多级页表相比，内存访问次数降低 74.14%，响应时间降低 71.99%。此外，PTE 紧致、PTE 压缩等技术应用在哈希页表中，在 16KB 粒度页面的基础上，将内存访问次数进一步降低 50.95%，响应时间降低 32.35%。

关键词：RISC-V；Hypervisor；KVM；16KB；哈希；虚拟化；仿真器；页表

目录

1 概述

1.1 项目背景与意义

在“互联网+”时代背景下，云计算已然成为数字经济时代下的基础设施。云计算也催化出大数据在应用领域的“井喷”，而支撑云计算服务的操作系统也是“东数西算”数字化基建中不可或缺的环节。随着近年来国家发布“十四五规划纲要”，把科技自立自强作为国家发展的战略支撑，国内学术界和产业界开始掀起以国产软硬件支撑云计算平台的研究热潮。

云计算的核心是资源的网络化共享、应用，这种透明的资源映射基础就是虚拟化技术。因此，在国产的软硬件平台实现虚拟化和有关的操作系统技术具有重要的科技和应用价值。

RISC-V 作为国家大力发展的处理器体系结构，很有希望成为国产云计算的主要载体。首先 RISC-V 架构开放、灵活、模块化，特别适合满足 AIoT 时代场景碎片化、差异化的市场需求；其次，在全球 RISC-V 生态中，中国地位凸显。一方面，中国企业对技术自研极其重视，RISC-V 成为摆脱“卡脖子”的一个最佳选择；另一方面，中国有着繁盛的 IoT 行业以及丰富的应用场景，驱动 RISC-V 技术和生态进化。因此，在 RISC-V 体系架构上研究虚拟化技术，对云计算的国产化具有特别重要的意义。

虚拟内存作为虚拟化的核心之一，是云计算操作系统中的关键技术。RISC-V 架构和其他国外的主流架构一样提供了虚拟内存的支持，默认采用了 4KB 大小的页面来实现虚拟内存。尽管 4KB 的页面可以产生较少碎片，从而节省内存空间，但多年来，对页面大小有影响的因素已经发生了变化。标准计算机的普通内存大小已经从若干 KB 增加到了若干 GB。因此，标准的 TLB（translation lookaside buffers）只覆盖了当代计算机的普通内存的一小部分。此外，当代标准磁盘的访问速度没有跟上吞吐量的增加。在过去的几年中，云计算导致应用的吞吐量提高了 100 倍，而访问速度只提高了 3 倍[1]。因此，使用一般页面实现虚拟内存已经无法满足需求，迫切需要使用更大的页面（即超过 4KB 的内存页面）进行存储。

目前的国外主流处理器平台(如 X86、ARM 等)都已经能很好地支持 16KB、64KB 乃至更大的页面来满足这种大页的需求，但 RISC-V 作为一种较新的体系结构还无法提供这类大页的支持。此外，在 RISC-V 架构中，页表是分层的，将翻译组织在 4 级（或 3 级）基数树中。在这个层次结构中找到一个缺失的翻译项将会产生四次（或三次）内存引用的开销。尽管 TLB 通常会避免这种开销，但它未命中时仍会显著降低性能，并且可能占应用程序运行时间的 50%。[2]

本项目基于以上的现状，将在 RISC-V 虚拟机上实现 16KB 页表，并在此基础上实现哈希地址转换逻辑，进一步降低访存开销，优化虚拟机在内存敏感场景下的性能，从而提高 RISC-V 架构在虚拟化服务上的表现，这也是本项目的意义所在。

本项目要在硬件模拟、体系结构和操作系统三个层面进行综合的设计和实验，在研究过程中，我们攻克了以下难点：首先是 RISC-V 虚拟仿真平台的搭建，由于现行服务器架构并不包括 RISC-V，我们需要在 ARM 架构上搭建一个 RISC-V 仿真平台，自行交叉编译系列测试程序及其依赖库，并且要学习 QEMU 仿真器的源码，在此基础上实现我们设计的寻址逻辑的仿真程序，并观测记录一些系统层面程序不能直接得到的测试数据。其次是设计支持 16KB 页表的虚拟内存系统，这需要我们对于虚拟化内存子系统有着深入了解的同时要阅读 Linux 源码并掌握虚拟内存系统的相关技术，从而在最新的 Linux Kernel 发行版本（比赛期间为 Linux Kernel 5.17）中实现。最后是哈希页表的设计，除了要熟练掌握对于 Kernel 中 KVM 模块的运作流程外，我们还需要从算法优化和硬件设计的角度综合考虑，应用开放寻址、表项集簇与表项紧致等技术对哈希表进行优化。

项目是推动学习的最好方法。我们团队通过本项目，深入理解了软件和硬件在操作系统领域的工作原理。在采用哈希页表实现虚拟机的 stage-2 页表的过程中，我们深入理解了虚拟化领域中最复杂的也是最重要的内存子系统，掌握了虚拟化技术、操作系统原理、以及硬件规范等内容。

1.2 项目主要工作

我们项目是赛题驱动项目，主要按照赛题要求进行，依据完成顺序可分为三个部分，下面我们将逐一介绍各个部分。

1.2.1 环境搭建

如图 1.1 所示，我们在 aarch64 系统上搭建了 RISC-V 虚拟机环境——用 QEMU 仿真平台运行带有 RISC-V Hypervisor[1]的 Host OS(Linux kernel 版本为 5.17)，并用 KVMTools 拉起运行在 RISC-V Hypervisor 之上的 Guest OS(Linux kernel 版本为 5.17)。

1.2.2 RISC-V 16K 粒度内存子系统

我们在 QEMU 中实现了支持 16KB 粒度页表的仿真逻辑，并在以此搭建了一个支持 16KB 粒度页表项的 RISC-V 仿真器，并预先设计了一个采用 16KB 粒度四级页表的 Test OS 验证该仿真器仿真逻辑的正确性。

之后，我们设计了一套 RISC-V 架构下的 16KB 粒度的内存子系统，并将其应用到 Linux Kernel 5.17 版本上，作为我们的 Host OS 成功地在上述的仿真器中运行。根据我们应用到 Host OS 和 Guest OS 的 16K 内存系统，我们在 Host OS 的 KVM 模块的缺页流程中，也配套设计和建立了 16KB 粒度的多级页表，以此来支持 16KB 粒度的 Guest OS 运行。

最终，我们在修改后的 RISC-V QEMU 与 Hypervisor 上成功运行起了 16KB 粒度的 Host OS 与 Guest OS。

1.2.3 基于哈希页表 KVM 模块

在实现 16KB 粒度页表的基础上，我们进一步设计了 RISC-V KVM 模块的 16KB 粒度的哈希地址转换逻辑，并配套地设计了哈希页表项格式。我们还利用了开放寻址，表项压缩，索引聚簇等技术优化我们的哈希页表设计（地址转

[1] 虚拟机监视器，用来建立与执行虚拟机的软件、固件或硬件。

换逻辑详见第 3.6 节），可以达到虚拟机 2G 内存访存无哈希冲突，从而将地址转换的访存开销压缩至一次。

我们同样也尝试了 4K 粒度的哈希页表的设计，但是效果并不如 16K 粒度的哈希页表降低 TLB 压力的效果显著，具体数据我们将在第四章中给出。

同样地，为验证设计的正确性，我们在 QEMU 中实现了哈希页表的地址转换逻辑，借此搭建了一个支持哈希页表项的 RISCV 仿真平台，并在该仿真平台上成功拉起运行了 16K 粒度的基于哈希页表的虚拟机。

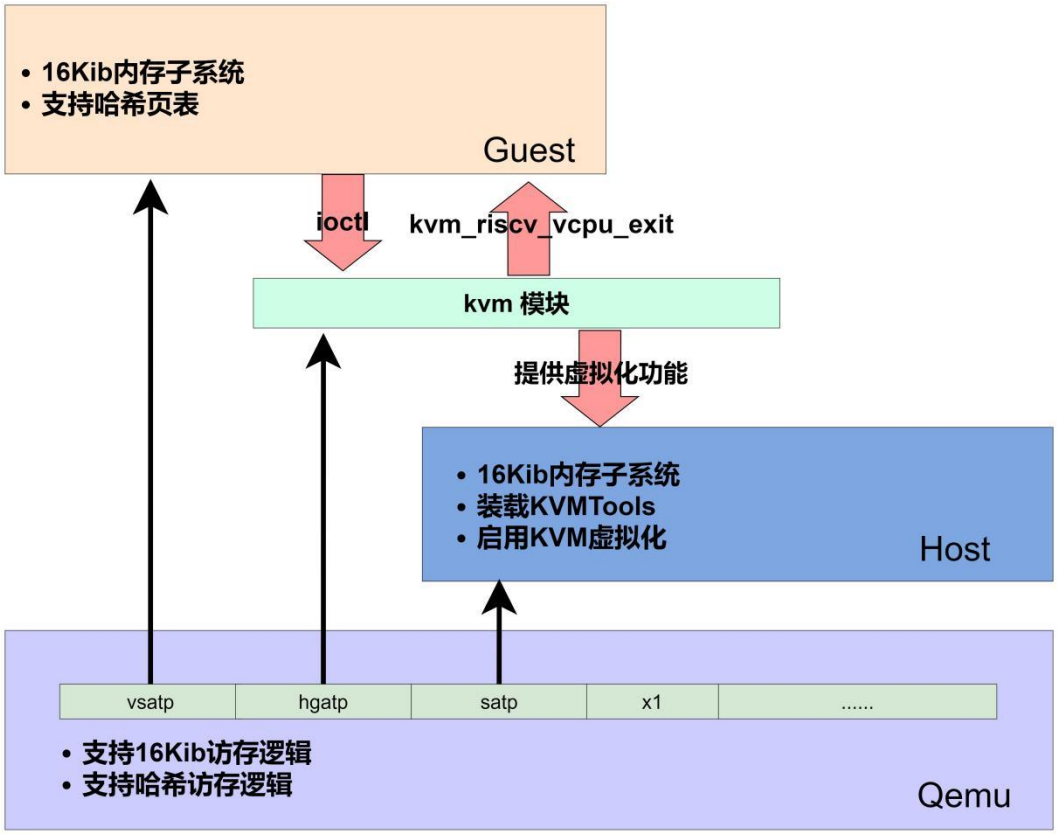


图 1.1 设计总览图

1.3 项目创新设计

本项目以 RISC-V 的虚拟化入手，为 RISC-V 服务器化探究道路，设计和实现均为首次完成，具有较强的创新性，主要体现为以下几点：

1) 首次在 RISC-V 架构中实现 16KB 粒度的内存子系统，并利用 KVM 模块拉起运行 16K 粒度的 Guest OS。

2) 首次在 RISC-V 虚拟化过程中使用哈希页表技术，来降低 TLB 压力。

3) 在哈希页表中加入开放寻址，表项压缩，索引聚簇等技术优化。

4) 在 QEMU 中实现了 RISC-V 架构的 16KB 粒度多级页表的仿真逻辑和哈希页表的仿真逻辑。

5) 完成题目规定内容后创新性地将 16KB 粒度的内存子系统和哈希页表结合，获得更好的访存性能。

1.4 操作系统教学启发

本项目对虽然难度较大，但内容新颖，设计知识广泛，对操作系统的教学有着启发式的意义。

首先，本项目可以帮助学生了解新版本 Linux 的相关知识。国内的操作系统教学大多较为关注系统的基础知识，课程实践中采用的版本也较为原始，对于最新的操作系统加载的新型技术往往欠缺了解，而本项目的基础需要完成者阅读 Linux Kernel 5.16 以上版本源码的内核启动和内存系统的相关部分，可以帮助完成者充分的了解这项相关的最新技术，并在实践中与课本的基础知识相互印证，加强理解。

第二，本项目可以帮助完成者了解硬件架构和模拟器的相关工作原理。本项目基于RISC-V 架构实现，有助于国内操作系统教学实践向RISC-V 架构过渡。同时，本项目是基于仿真器实现，完成者需要进行交叉编译、阅读和修改 QEMU 源码等步骤，可以帮助完成者了解 QEMU 地址翻译的流程，对于仿真器原理和硬件逻辑的进一步学习。

第三，本项目可以帮助完成者进一步掌握虚拟化相关技术，现在的系统教学一般聚焦与内存，中断，IO，调度几大基本方向，而随着云计算的日益发展，作为云计算基础的虚拟化领域的研究得也到了越来越多的重视。本项目求采用哈希页表实现虚拟机的 stage-2 页表，并以此入手，串联内存子系统和虚拟化两大板块，加深完成者对于两者的相关知识的掌握。

第四，本项目作为一个较大的综合性实践项目，内容集源码阅读、硬件仿真、系统模块的实现于一体。通过本项目，可帮助完成者深入理解软件和硬件在操作系统领域的工作原理，探究虚拟化领域中最复杂的也是最重要内存子系统，掌握虚拟化技术、操作系统原理、以及硬件规范等内容，对从硬件到体系结构再到操作系统的计算机运行完整流程有一个立体的认识。

在图 1.2 中，我们整理了本项目涉及的操作系统教学中的部分知识点。

图 1.2 操作系统教学知识点

2 相关工作与原理

2.1 应用场景及需求

本项目通过改变页表粒度和地址转换机制，来提高虚拟机的访存性能，对几乎所有的虚拟机上运行的内存敏感应用均有显著的效果，特别适用如云计算服务、游戏渲染、科学矩阵计算、流式媒体、数字图书馆、数据中心虚拟化等数据大规模高性能计算的场景。

基于以上需求，需要我们的项目中实现：

1) 支持 16KB 粒度的 Guest OS 能够无修改地运行在 Type-1, Type-2 和混杂模式的 Hypervisor 上。
2) 降低内存访问次数，减小 TLB 更新的压力。
3) 提高内存访问带宽，以提升 Guest OS 在高性能计算和内存敏感应用场景下的表现。
4) 在高并发下的大型应用端性能中有良好的表现。

2.2 相关工作现状

2.2.1 Kernel Virtualization

近年来，以 KVM（Kernel-based Virtual Machine）为代表的内核级虚拟化技术以高性能与高稳定性得到了广泛应用。在内核级虚拟化下，Host OS 在经过特殊修改的内核上运行，该内核包含旨在管理和控制多个虚拟机的扩展，每个虚拟机都包含一个 Guest OS。

KVM 作为内核级虚拟化技术的典例，是一种内置于 Linux 中的开源虚拟化技术。具体来说，KVM 允许用户将一个 Linux 标准内核转换成为一个虚拟机管理程序，它允许 Host OS 运行多个隔离的虚拟环境，即虚拟机。

为了减小 TLB 压力，提升 RISC-V 在虚拟化服务上表现，KVM 正是本项目着力进行优化的关键一环。

2.2.2 RISC-V with H-extension

随着 RISC-V 社区的发展，以及开源工作的不断努力，目前 RISC-V 指令已经正式支持虚拟化功能，即 H-extension。RISC-V 的实现的虚拟化主要包括以下几个方面：RISC-V CPU 本身的虚拟化、内存虚拟化、IO 虚拟化、Timer 虚拟化、中断虚拟化。自 Linux Kernel 5.16 版本，RSIC-V 架构也正式支持 KVM 模块。

当前，能够运行虚拟化软件的仿真器以及操作系统都在社区可公开使用。这也为我们项目的开发提供了体系架构和系统层面上的支持。

2.2.3 Larger Page Size

P. Weisber 和 Y. Wiseman[1]在《Using 4KB page size for Virtual Memory is obsolete》一文中指出了为虚拟内存选择最佳的页面大小需要考虑若干因素。一方面，较小的页面大小可以减少内部碎片化的数量。另一方面，较大的页面将增加 TLB 的覆盖率，从而消除访问内存常驻页表的需要。他们基于 SPEC2000 套件中的应用程序模拟了从 4KB 到 256KB 页面大小的 TLB 缺失和内存使用情况，最终表明了使用 16KB 大小的页面是最优的选择，这也为本项目的页面大小的选择提供了理论支撑。

2.2.4 Hash Page Table

Hash Page Table 使用一张哈希表存储虚拟页到物理页的映射信息。在不产生哈希冲突的情形下，哈希页表仅需要一次内存访问，就能完成虚拟地址到物理地址的转换。

然而，Thomas W. Barr, Alan L. Cox, Scott Rixner 在《Translation Caching: Skip, Don’t Walk (the Page Table)》[3]一文中指出，在 SPEC 2006 应用程序集上，哈希页表产生了 5 倍于多级页表的内存访问；即使在虚拟地址局部性特征不明显的应用场景下，哈希页表仍然表现不佳。基于此，文中指出了哈希页表的三个缺陷：（1）哈希页表将原本连续的页映射到了不同的 cache line 中，这使得哈希页表在许多 workload 上无法很好地利用局部性。（2）哈希页表的 PTE 要比多级页

表的 PTE 大，这使得内存访问的开销增大了。（3）哈希冲突出现的频率较高，这也导致在页查询中产生了更多的内存访问。

随后的《Hash, Don't Cache (the Page Table)》[2]一文指出，《Translation Caching: Skip, Don't Walk (the Page Table)》中提及的哈希页表缺陷仅仅是在 Itanium 设计架构下的一个特例。文中进而提出了针对哈希页表的三个优化方法：开放寻址、PTE 聚簇、PTE 紧致。通过这些优化，哈希页表在 SPEC mcf、SPEC xalancbmk、graph500、GUPS 这些 benchmark 的虚拟化和非虚拟化情景下，取得了和多级页表相近的运行时间优化效果；而在 SPEC cactusADM 这一 benchmark 的虚拟化和非虚拟化情景下，哈希页表对运行时间的优化效果，要远远大于多级页表。

因此，为了缓解 TLB 的压力，提高对内存的访存速度，我们，选择了在 KVM 中使用哈希页表来替代多节页表的地址转换模式。

2.3 相关技术原理

2.3.1 RISC-V Virtual-Memory System

虚拟内存是现代操作系统使用的一种管理和分配内存的机制，由相应的寄存器，地址转换模式，虚拟地址格式，虚拟地址空间布局组成了特定的虚拟内存系统。

Satp 是一个 SXLEN-bit 读写寄存器，控制 S-MODE 下地址转换和地址保护。其格式如图 2.1 和图 2.2 所示（图 2.1 用于 SXLEN=32，图 2.2 用于 SXLEN=64）。它控制着 S-MODE 下的地址转换和保护。这个寄存器保存根页表的物理页号(PPN)；地址空间标识符(ASID)，表示当前进程的 ASID 号；以及 MODE 字段，它选择当前的地址转换方案。具体介绍请阅读 RISC-V Privileged Spec 4.1.10。

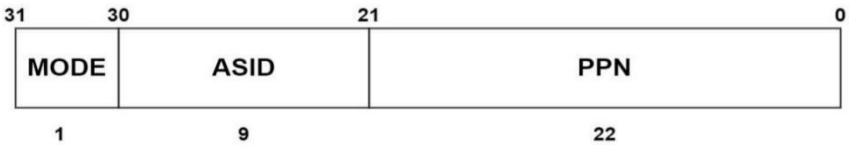


图 2.1 RV32 satp

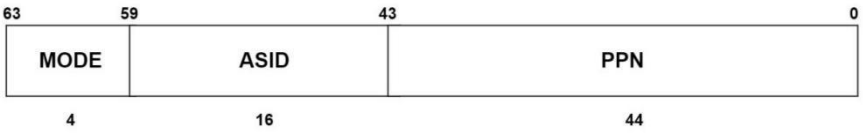


图 2.2 RV64 satp

表 2.1 和表 2.2 显示了 RV32 和 RV64 的 MODE 字段的编码和不同模式对应的地址转换和物理内存保护方案。具体介绍请阅读 RISC-V Privileged Spec 4.1.10。

表 2.1 RV32

Value	Name	Description
0	Bare	No translation or protection
1	Sv32	Page-based 32-bit virtual addressing

表 2.2 RV64

Value	Name	Description
0	Bare	No translation or protection
1-7	-----	Reserved for standard use
8	Sv39	Page-based 39-bits virtual addressing
9	Sv48	Page-based 48-bits virtual addressing
10	Sv57	Page-based 57-bits virtual addressing
11	Sv64	Page-based 64-bits virtual addressing
12-13	----	Reserved for standard use
14-15	----	Designated for custom use

此处，我们以 Sv39 和 Sv48 为例介绍两种虚拟地址和物理地址的转换过程，而本项目的 16KB 的转换也将基于这两种模式完成。

1. Sv39

Sv39 实现支持 39 位虚拟地址空间，它被划分为若干个 4KB 大小的页面。一个 Sv39 虚拟地址被划分为虚拟页码(VPN)和页面偏移量，如图 2.3 所示。当在 satp 寄存器的 mode 字段中选择 Sv39 虚拟内存模式时，虚拟地址通过一个三级页表被转换为物理地址。如图 2.4，27 位的 VPN 被转换为 44 位的物理页面号(PPN)，之后以 PPN 所在的物理页内加上页内偏移量，便可以直接转换为物理地址。

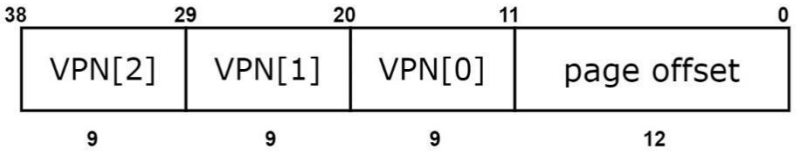


图 2.3 Sv39 virtual address

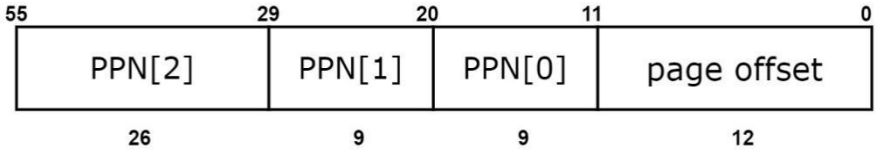


图 2.4 Sv39 physical address

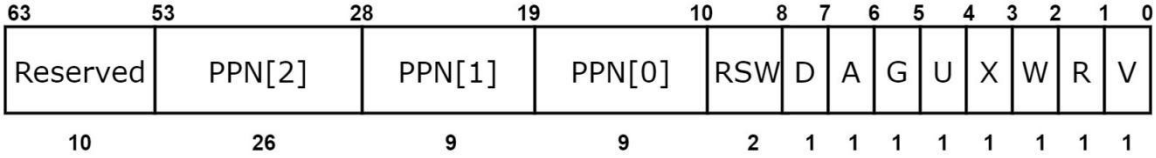


图 2.5 Sv39 page table entry

Sv39 页表由 512 个页表项(PTEs)组成，每个 PTE 为 4 个字节。页表正好是一个页面的大小，并且必须始终与页面边界对齐。根页表的物理页号存储在 satp 寄存器中。

Sv39 的 PTE 格式如图 2.5 所示。V 位表示 PTE 是否有效；如果它是 0，PTE 中的所有其他位都是无效的，可以由用户自由使用。权限位 R、W 和 X 分别表示该页面是可读、可写和可执行的。当这三个值都为 0 时，PTE 是指向页表下一层的指针；否则，它是一个叶 PTE。可写页面也必须标记为可读；相反的组合留作将来使用。表 2.3 总结了权限位的编码。

表 2.3 权限位汇总

X	W	R	Meaning
0	0	0	Pointer to next level of level page table
0	0	1	Read-only page
0	1	0	Reserved for future use
0	1	1	Read-write page
1	0	0	Execute-only page
1	0	1	Read-execute page
1	1	0	Reserved for future use
1	1	1	Read-write-execute page

三级页表的寻址逻辑如图 2.6 所示，从 satp 寄存器中我们得到 PGD 的物理页号，根据 VPN[2]作为 index 找到对应的 PTE，若其权限位不为 0，则 PTE 中取出 PPN 为物理页号并以 page offset 页内偏移找到物理地址；否则，取出 PPN 为 PMD 的物理页号，根据 VPN[1]作为 index 找到对应的 PTE。若其权限位不为 0，则 PTE 中取出 PPN 为物理页号并以 page offset 页内偏移找到物理地址；否则，取出 PPN 为 PGT 的物理页号，以 page offset 页内偏移得到物理地址。

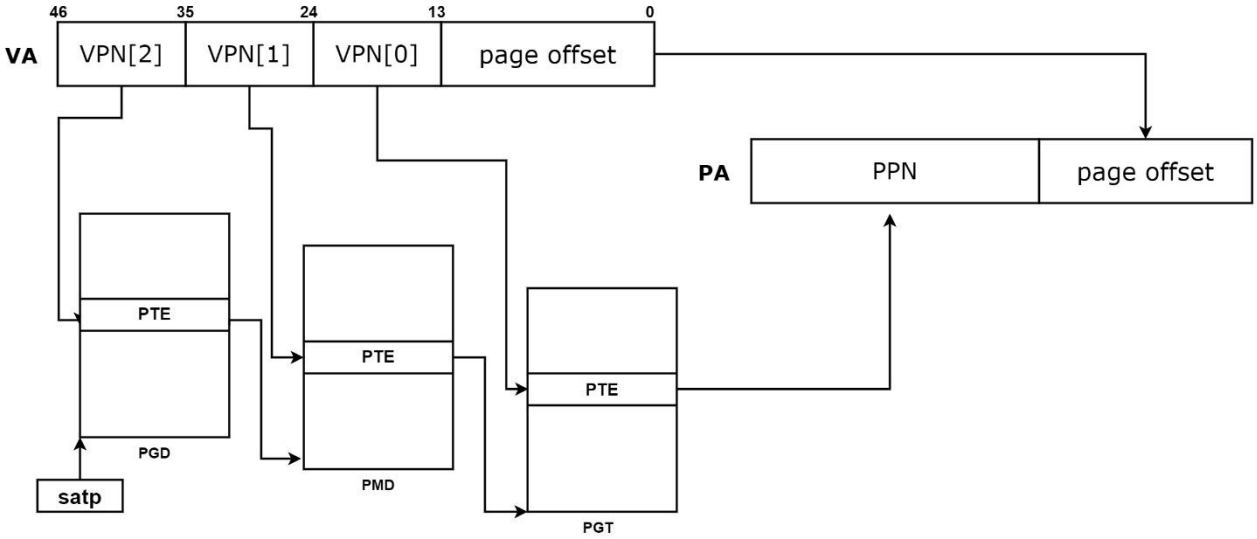


图 2.6 Sv39 三级页表转换过程

2. Sv48

Sv48 的地址转换过程和 Sv39 基本一致，区别为 Sv48 用四级页表进行地址转化，此处仅介绍 Sv48 的虚拟地址格式（图 2.7）、物理地址格式（图 2.8）、页表项（PTE）格式（图 2.9），和四级页表转化的流程图（图 2.10），详细的请参考 Sv39 和 RISC-V Privileged Spec 4.3.2

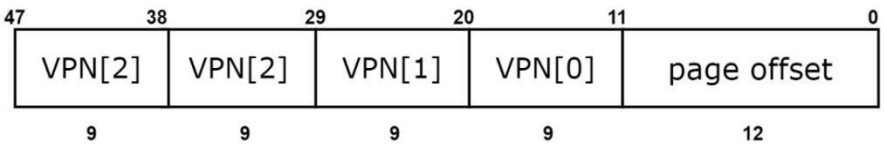


图 2.7 Sv48 virtual address

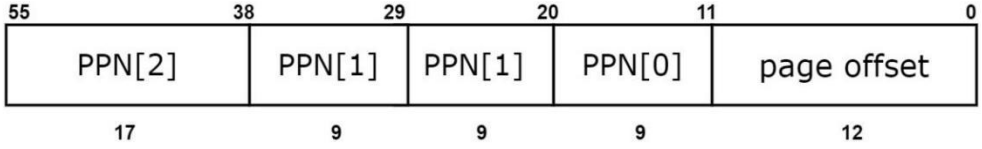


图 2.8 Sv48 physical address

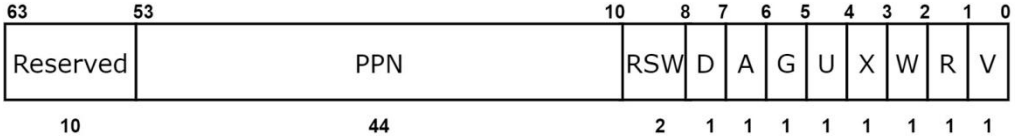


图 2.9 Sv48 page table entry

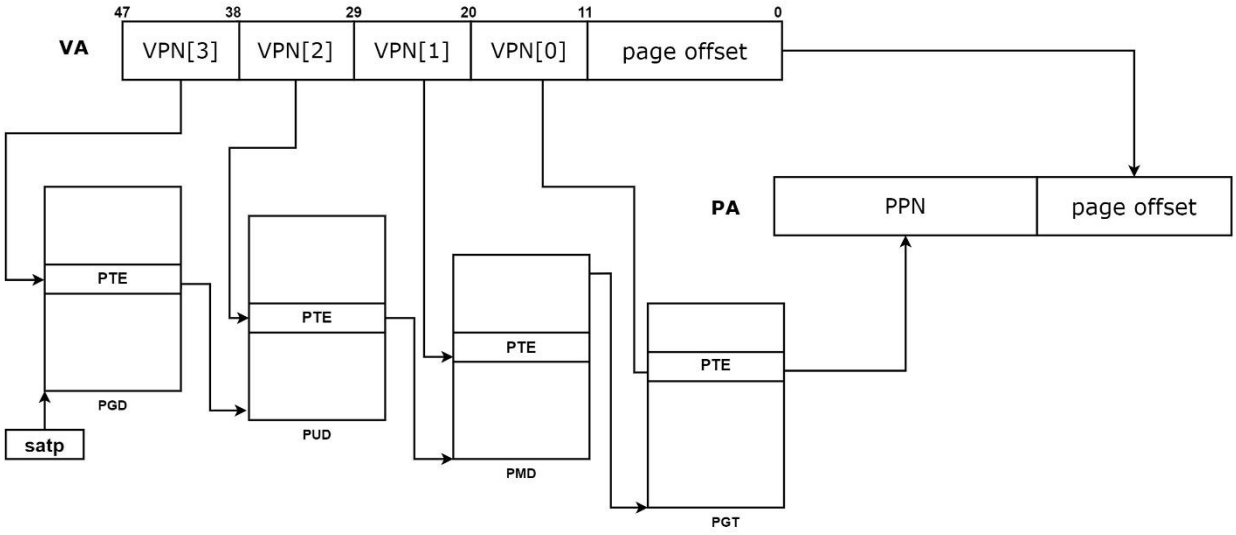


图 2.10 Sv48 四级页表转换过程

2.3.2 Virtual Memory Layout on RISC-V Linux

不同架构的内核和不同的虚拟内存系统都有不同的虚拟地址空间布局，下以 RISC-V 架构下的 Sv39 为例进行说明。

64 位地址“必须使第 63-48 位都等于第 47 位”：这将虚拟地址空间分成两半，形成了一个巨大的空洞，下半部分是用户空间，上半部分是 RISC-V Linux Kernel。

Linux Kernel 分为 direct mapping，kasan，modules 和 kernel 四个部分。其中 direct mapping 是线性映射区域，长度是内核虚拟地址空间的一半。Kasan 是 Kernel Address Sanitizer 的缩写，它是一个动态检测内存错误的工具，主要功能

是检查内存越界访问和使用已释放的内存等问题。Modules 区域长度 128M，是内核模块使用的虚拟地址空间。Kernel 区域即为内核与 OpenSBI (Open Supervisor Binary Interface) 在运行过程中所映射到的虚拟内存空间。具体布局见表 2.4 和图 2.11。

表 2.4 Sv39 虚拟空间布局

Start addr	Offset	End addr	Size	Description
0x0000000000000000	0	0x0000003FFFFFFFFF	256 GB	user-space virtual memory
0x0000004000000000	256GB	0xFFFFFFBFFFFFFFFF	16M TB	huge, almost 64 bits wide hole
Kernel-space virtual memory:				
0xFFFFFFC6FEE00000	-228GB	0xFFFFFFC6FEFFFFFF	2 MB	Fixmap
0xFFFFFFC6FF000000	-228GB	0xFFFFFFC6FFFFFFFF	16 MB	PCI io
0xFFFFFFC700000000	-228GB	0xFFFFFFC7FFFFFFFF	4 GB	vmemmap
0xFFFFFFC800000000	-224GB	0xFFFFFFD7FFFFFFFF	64 GB	vmalloc/ioremap space
0xFFFFFFD800000000	-160GB	0xFFFFFFF6FFFFFFFF	124 GB	direct mapping of all physical memory
0xFFFFFFF700000000	-36 GB	0xFFFFFFFEFFFFFFFF	32 GB	kasan
------	------	------	------	Reserved
0xFFFFFFFF00000000	-4 GB	0xFFFFFFFF7FFFFFFF	2 GB	modules, BPF
0xFFFFFFFF80000000	-2 GB	0xFFFFFFFFFFFFFFFF	2 GB	kernel

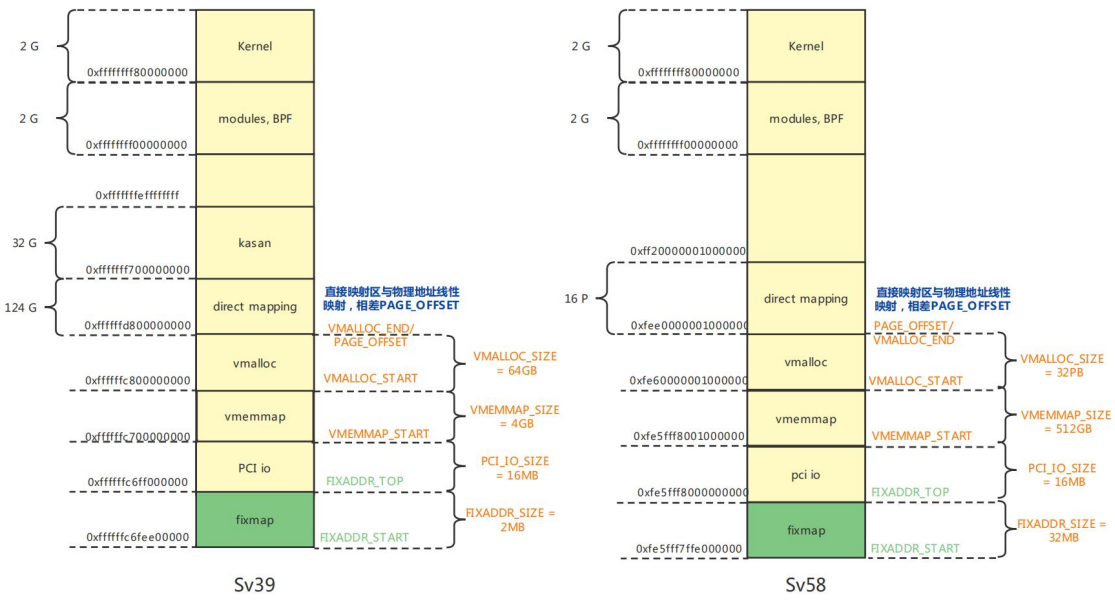


图 2.11 虚拟内存空间布局

左图为 4KB，右图为 16KB，为对比两者区别特放在一处，右图具体设计会在 3.3.2 节给出。

2.3.3 Two Stage Translation

非虚拟化情况下，CPU 访问物理内存前，需要先建立页表映射（虚拟地址到物理地址的映射），最终通过查表的方式来完成访问。在 RISC-V 中，根页表基地址存放在 satp 寄存器中。

而在虚拟化情况下，内存的访问会分为两个 Stage：Host OS 控制 Stage 1 的转换，将虚拟地址转换为中间物理地址；而 Hypervisor 控制 Stage 2 的转换，将中间物理地址转换为物理地址，地址转换详见图 2.12。在 RISC-V 中，Stage 1 的根页表基地址存放在 vsatp 寄存器中，而 Stage 2 的根页表基地址存放在 hgatp 寄存器中。

Hypervisor 能通过 Stage 2 来控制虚拟机的内存视图，控制虚拟机是否可以访问某块物理内存，进而达到隔离的目的。

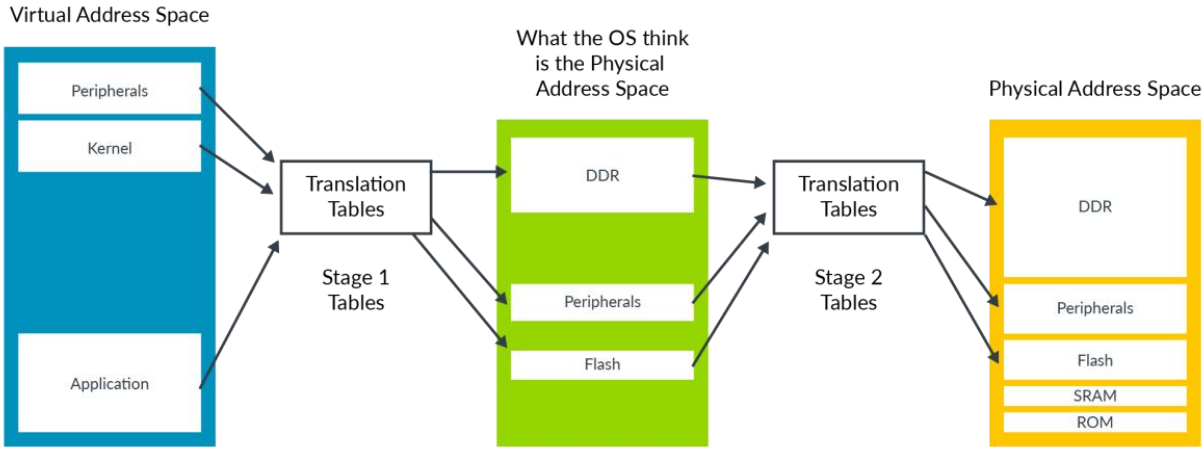


图 2.12 两阶段地址转换[5]

2.3.4 Hash Page Table

哈希页表是一种处理大于 32 位地址空间的常用映射方法。哈希页表采用虚拟页码作为哈希值，采用一张哈希表存储虚拟页到物理页的映射信息。传统的哈希页表采用链地址法解决哈希冲突，其每个条目都包括一个链表，该链表的元素哈希到同一位置，其中的每个元素由三个字段构成：虚拟页码、映射的帧码、指向链表内下一元素的指针。

哈希页表的工作流程为：首先将虚拟地址的虚拟页号通过哈希函数进行哈

希，获得其在哈希页表中的索引；接着将地址的虚拟页码与该索引对应链表中每一个元素的虚拟页码字段进行比较，直到找到匹配的元素；最后取出该元素中的帧码字段与自身偏移地址一起，形成物理地址。详细流程可见图 2.13。

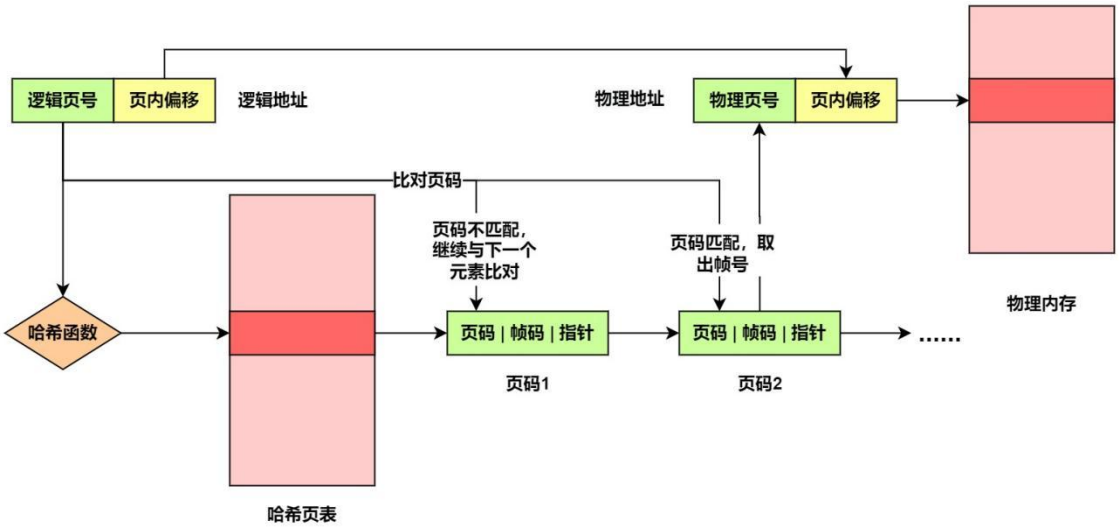


图 2.13 哈希页表寻址流程

3 设计思路与实现重点

3.1 总体思路与系统框架

如图 3.1 所示，我们项目的系统框架是在服务器上搭建了 RISC-V 虚拟机环境——以 QEMU（我们在其中实现了 16KB 粒度多级页表和哈希页表的仿真逻辑）作为仿真器运行带有 RISC-V Hypervisor 的 16KB 粒度的 Host OS（具体过程可见图中的页表 Host 部分），并用 KVM-Tools 拉起运行在 RISC-V Hypervisor 之上的 16KB 粒度的 Guest OS，且支持在 KVM 模块中 stage-2 阶段的 MMU 中使用哈希页表进行 HPA 和 GPA 的地址转换（具体过程可见图中的页表 Guest 部分）。

为了尽可能提升虚拟机的性能，减少由于系统本身出现的漏洞，我们选用了开题当时最先进的 RISC-V 架构下的 Linux Kernel 5.17 版本作为 Host OS 和 Guest OS。但是由于 Kernel5.17 版本加载了较多了新型技术，直接修改验证的难度较大，我们实现了一个具有虚拟内存和进程切换的 Test OS 来验证仿真逻辑的正确性，确保仿真逻辑正确后，再于仿真器上运行 Kernel 5.17 版本。

同时我们根据 Sv39 和 Sv48 实现了基于 16KB 页面的 Sv47 和 Sv58，为此我们还重新设计了相应的 Sv58 的虚拟内存系统，重新规划了虚拟地址空间布局。我们还修改了 KVM 模块，使其可以配适 16KB 粒度的虚拟机。

最后在实现 16KB 粒度页表的基础上，我们进一步设计了 RISC-V stage-2 的 16KB 粒度的哈希地址转换逻辑，并配套地设计了哈希页表项格式。我们还利用了开放寻址，表项压缩，索引聚簇等技术优化我们的哈希页表设计（地址转换逻辑详见图），可以达到虚拟机 2G 内存映射无哈希冲突发生，从而将 stage-2 地址转换的访存开销可压缩至一次。

同样地，为验证设计的正确性，我们 QEMU 中实现了一个支持哈希页表地址转换的仿真逻辑，并基于此搭建了一个支持哈希页表项的 RISCV 仿真平台，使用 Test OS 测试验证哈希页表寻址的仿真逻辑正确后，在该仿真平台上拉起运行 16KB 粒度的 Guest OS。

总的来说，我们分成了如下几个步骤进行实现：

1. 用 RISC-V 仿真平台运行带有 RISC-V Hypervisor 的 Host OS。
2. 用 RISC-V QEMU 拉起运行在 RISC-V Hypervisor 之上的 Guest OS。
3. 设计 RISC-V stage-2 的 16KB 粒度的页表项格式。
4. 在 RISC-V Hypervior 的 stage-2 缺页流程中，建立以 16KB 粒度的页表。
5. 在 RISC-V 仿真器中，在 stage-2 的 MMU 中实现 16KB 粒度的页表寻址的仿真逻辑。
6. 设计 RISC-V stage-2 的哈希页表。
7. 在 RISC-V Hypervior 的 stage-2 缺页流程中，实现哈希页表建立过程。
8. 在 RISC-V 仿真器中，在 stage-2 的 MMU 中实现哈希页表寻址的仿真逻辑。

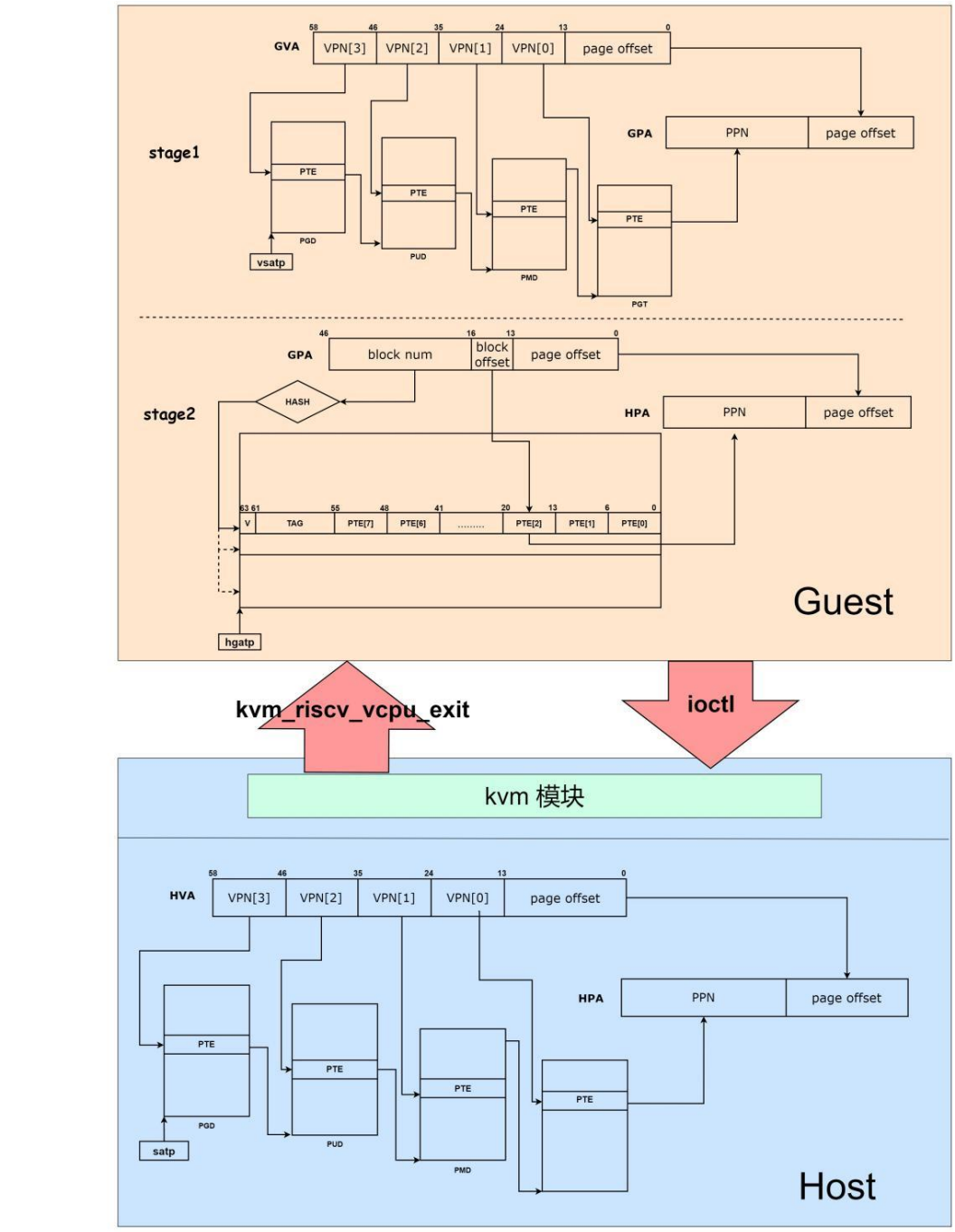

GVA
VPN[3]
VPN[2]
VPN[1]
VPN[0]
page offset
stage1
PTE
PGD
PUD
PMD
PGT
vsatp
GPA
PPN
page offset
GPA
block num
block offset
page offset
stage2
HASH
HPA
PPN
page offset
V
TAG
PTE[7]
PTE[6]
PTE[2]
PTE[1]
PTE[0]
hgatp
Guest
kvm_riscv_vcpu_exit
ioctl
kvm 模块
HVA
VPN[3]
VPN[2]
VPN[1]
VPN[0]
page offset
HPA
PPN
page offset
satp
Host
QEMU

图 3.1 项目系统框架

3.2 RISC-V 仿真平台搭建

由于现行服务器架构中并无 RISC-V 架构，我们需要在 ARM 架构上搭建 RISC-V 仿真平台，具体工作如下：

首先，我们在现有的基于 ARM 架构的服务器上安装了 RISC-V 交叉编译工具（riscv-gnu-toolchain），这为我们之后的环境搭建做好了铺垫。

之后，我们安装了 QEMU，使用 RISC-V 交叉编译工具完成了 OpenSBI 的构建，并获取与编译了 Linux 5.17 版本内核。这一步完成后，我们测试了 QEMU 启动 Host OS 并获得了成功，运行界面如图 3.2 所示。

图 3.2 Host OS 成功运行

为了编译 KVM-Tools（轻量级 QEMU），我们需要使用交叉编译工具链中的 libfdt 库。而 libfdt 库在交叉编译工具链中一般不可用，因此我们从 DTC 项目中显式编译了 libfdt 并将其添加到 CROSS_COMPILE SYSROOT 目录中。

最后，我们完成了 KVM-Tools 的构建，并使用 busybox[2]构建了包含 Guest OS，KVM 模块与 KVM-Tools 的 Host OS 的根文件系统。这一步完成后，我们测试了在 Host OS 上使用 KVM-Tools 拉起 Guest OS 并获得了成功，运行界面如图 3.3 所示。

[2] busybox 集成压缩了 Linux 的许多工具和命令，可用来引导内核

```
[   11.218615] VFS: Mounted root (9p filesystem) on device 0:14.
[   11.318261] devtmpfs: mounted
[   11.727022] Freeing unused kernel image (initmem) memory: 2152K
[   11.822145] Run /virt/init as init process
Mounting...
/ # ls
bin     etc     host    lib64   root    sys     usr     virt
dev     home    lib     proc    sbin    tmp     var
/ # cd bi[   58.107699] random: fast init done
n
/host/bin # ls
arch            dmesg           iostat          mv              sleep
ash             dnsdomainname   kbd_mode        netstat         stat
base64          dumpkmap        kill            nice            stty
bash            echo            link            nuke            su
busybox         ed              linux32         pidof           sync
cat             egrep           linux64         ping            tar
chattr          false           ln              printenv        touch
chgrp           fatattr         login           ps              true
chmod           fgrep           ls              pwd             umount
chown           fsync           lsattr          resume          uname
cp              getopt          lzop            rm              uncompress
cpio            grep            mkdir           rmdir           usleep
cttyhack        gunzip          mknod           sed             vi
date            gzip            mktemp          setpriv         watch
dd              hostname        mount           setserial       zcat
df              hush            mpstat          sh
```

图 3.3 Guest OS 成功运行

3.3 Sv58 虚拟内存系统设计

由于 Linux Kernel 5.17 版本会优先使用 4 级页表进行地址转换，所以我们也使用四级页表作为默认选项，为此我们设计了基于 16KB 粒度页表的 Sv58 虚拟内存系统，供 Host OS 和 Guest OS 使用。

3.3.1 16KB 粒度页表表项格式

由于物理页大小设置为了 16KB，页内偏移量变为了 14 位，为了充分使用页面，容纳更多的表项，相应的我们需要增加 VPN 的位数，于是重新设计页表格式，16KB 的一页可以容纳 2048 个表项，所以需要把作为索引的 VPN 设置为 11 位，根据 3 级页表和 4 级页表，得到了虚拟地址位数为 47 和 58 的虚拟地址格式，如图 3.4 和图 3.5 所示，我们不妨命名为 Sv47 和 Sv48。

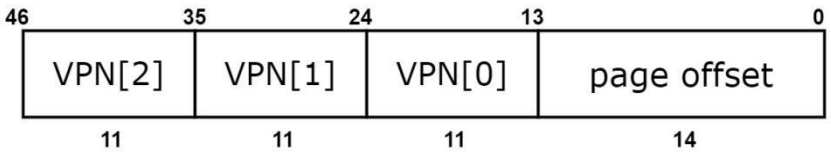


图 3.4 Sv47 virtual address

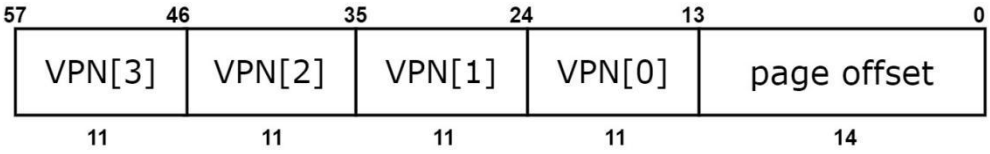


图 3.5 Sv58 virtual address

去掉 14 位的页内偏移量后，我们得到了 42 位的物理页号 PPN，相应的也就得到了 Sv47 和 Sv48 的页表表项的格式，如图 3.6 和图 3.7 所示：

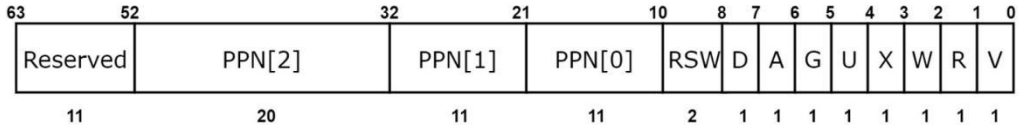


图 3.6 Sv47 page table entry

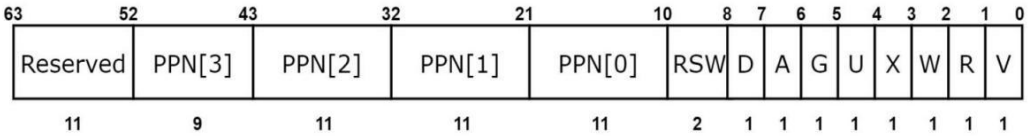


图 3.7 Sv58 page table entry

鉴于 RV64 中 14 和 15 两个字段是为用户保留的(可见表 2.2)，我们在 RV64 记录 MODE 字段的编码表中为 Sv47 和 Sv58 增加了两项，见表 3.1。

表 3.1 RV64

Value	Name	Description
0	Bare	No translation or protection
1-7	-----	Reserved for standard use
8	Sv39	Page-based 39-bits virtual addressing
9	Sv48	Page-based 48-bits virtual addressing
10	Sv57	Page-based 57-bits virtual addressing
11	Sv64	Page-based 64-bits virtual addressing
12-13	----	Reserved for standard use
14	Sv47	Page-based 47-bits virtual addressing
15	Sv58	Page-based 58-bits virtual addressing

3.3.2 Sv58 虚拟地址空间布局

为了保证内核启动过程中在 setup_vm[3]函数中对于内核进行的 1PMD 映射可以正常进行，我们需要调物理地址映射的起始位置，从 0x80200000 调整至 0x80000000，这 2M 的内存空间是储存的是加载的 OpenSBI，但这额外映射的部分不会对内核的正常运行产生任何影响，因为 OpenSBI 运行在 M 态，直接使用物理地址。

同时，fixmap 技术也要求内核空间的虚拟地址的起始地址可以被 pmd_size[4]整除，才能正确的用一块固定的虚拟地址来动态映射多段物理地址，从而页表建立之前用于完成对 IO 设备的访问、设备树的解析以及 paging_init[5]中的页表切换等任务。

此外，direct mapping、vmalloc 等区域均的大小与排布依赖与内核虚拟地址空间和页表的大小。

基于上述几点考虑我们从新设计了如图 3.8 右侧的虚拟地址空间布局（对比 Sv39），详细信息可见表 3.2。

表 3.2 Sv58 虚拟空间布局

Start addr	Offset	End addr	Size	Description
0xFE5FFF7FFE000000	-104PB	0xFE5FFF7FFFFFFFFF	32 MB	fixmap
0xFE5FFF8000000000	-104PB	0xFE5FFF8000FFFFFF	16 MB	PCI io
0xFE5FFF8001000000	-104PB	0xFE60000000FFFFFF	512 GB	vmemmap
0xFE60000001000000	-104PB	0xFEE0000000FFFFFF	32 PB	vmalloc/ioremap space
0xFEE0000001000000	-72 PB	0xFF20000000FFFFFF	16 PB	direct mapping of all physical memory
0xFF20000001000000	------	0xFFFFFFFEFFFFFFFF	------	Reserved
0xFFFFFFFF00000000	-4 GB	0xFFFFFFFF7FFFFFFF	2 GB	modules, BPF
0xFFFFFFFF80000000	-2 GB	0xFFFFFFFFFFFFFFFF	2 GB	kernel

[3] Kernel 启动过程中初步开启虚拟地址的重要函数。

[4] pmd_size 指 page middle dictionary 中一条表项可以映射的地址的大小。

[5] paging_init 是 Kernel 启动过程中初始化页表的重要函数。

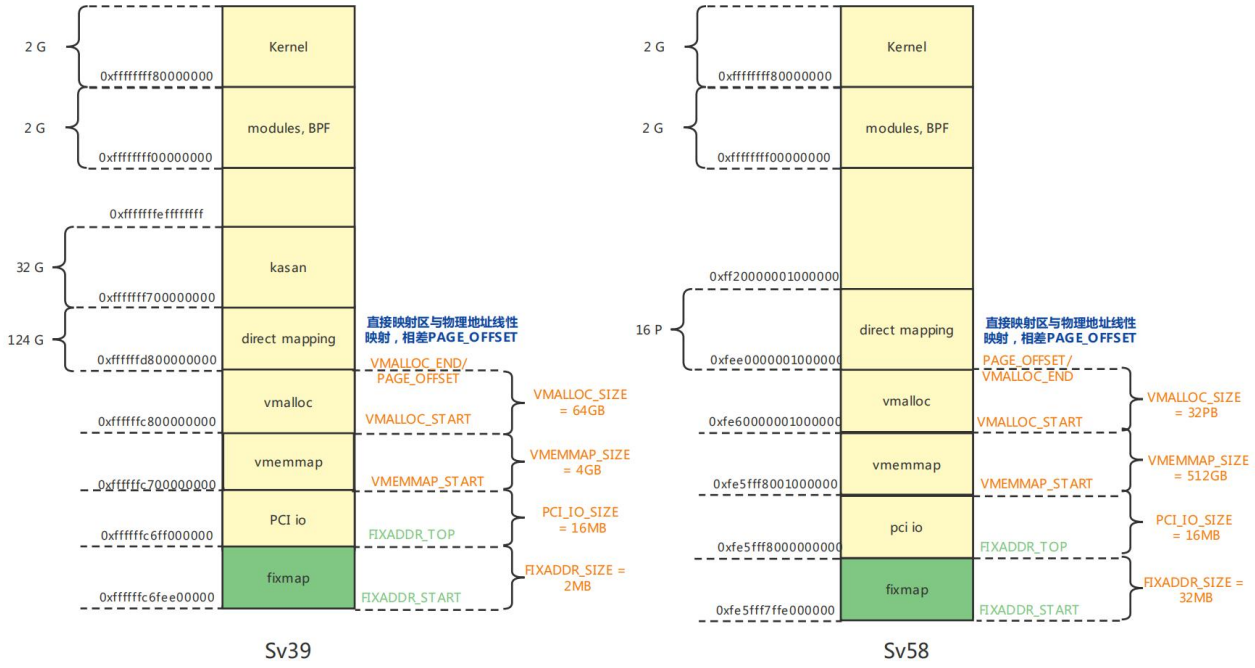


图 3.8 虚拟内存空间布局

左图为 4KB，右图为 16KB

3.4 16KB 粒度 KVM 模块的实现

根据 3.3.1 节中我们设计的页表表项格式以及图 3.9 的 KVM 模块异常处理流程，我们在 Linux Kernel 5.17 版本中的 KVM 模块实现了基于 16KB 粒度的缺页异常处理的流程。

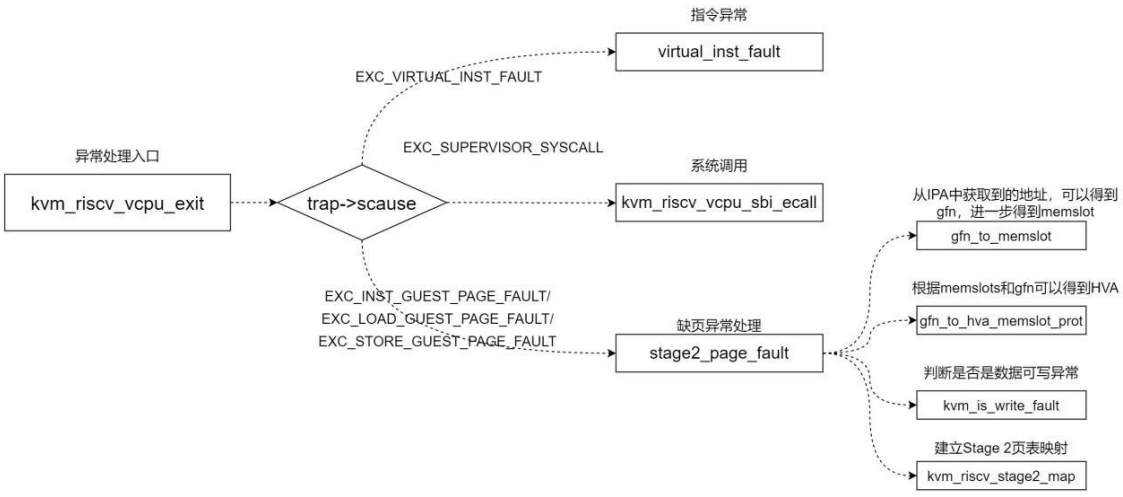


图 3.9 KVM 模块异常处理流程图

kvm_riscv_vcpu_exit 函数是 KVM 模块异常处理的入口，需要根据 scause 寄存器的值判定异常类型，对于缺页异常会进入 stage2_page_fault 函数进行处理。在建立页表映射前，我们需要从 IPA 中获取地址。kvm_riscv_stage2_map 函数会接收 IPA 与获取的 HVA 建立对应的页表映射，详见图 3.10。

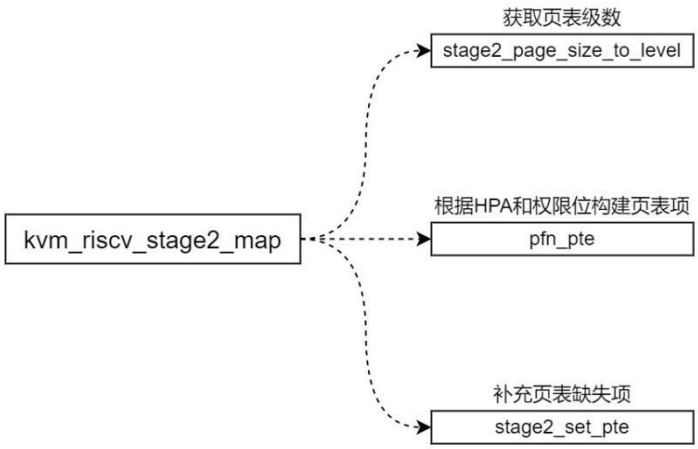


图 3.10 KVM 模块异常处理流程图

3.5 基于 16KB 粒度页表的仿真器的实现

和 Sv48 一样，Sv58 也采用四级页表进行地址转换，详细过程可见 2.3.1 节，只是由于页表粒度不同造成虚拟页号、物理页号和页内偏移量的位数不同，根据 3.3.1 节中我们设计的页表表项格式和虚拟地址格式，我们在 QEMU 仿真器中实现如图 3.11 的仿真逻辑。

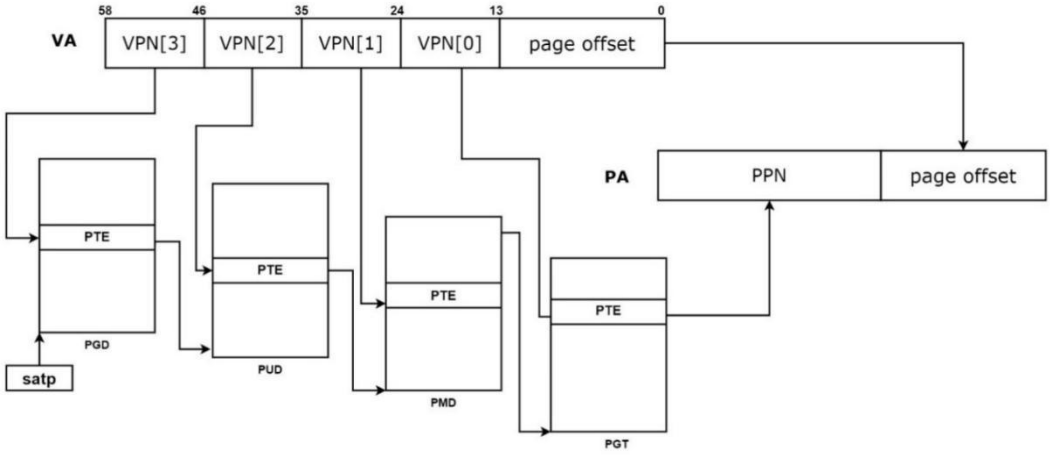


图 3.11 Sv48 四级页表转换过程

如图 3.12 所示，我们在 Test OS 上验证了其仿真逻辑的正确性，进程可以根据最短任务调度算法进行切换进程。

```
Boot HART ID              : 0
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsuh
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
PC changed!
The value of scause is:000000000000000c
...mm_init done!
...process init done!
16K test OS is running!
SET [ PID = 1 COUNTER = 2]
SET [ PID = 2 COUNTER = 2]
SET [ PID = 3 COUNTER = 7]
SET [ PID = 4 COUNTER = 5]
SET [ PID = 5 COUNTER = 5]
```

图 3.12 Sv58 Test OS 在仿真器中成功运行

随即将 3.3 节中的 Sv58 虚拟内存系统，应用到 Linux Kernel 5.17 版本上，我们也成功的在刚刚验证过的仿真器中成功运行了 Host OS，可见图 3.13。

```
[    1.665623] debug_vm_pgtable: [debug_vm_pgtable         ]: Validating architecture page table helpers
[    1.936785] Freeing unused kernel image (initmem) memory: 16560K
[    1.939690] Run /init as init process
          _  _
         | ||_|
         | | _ ____  _   _  _  _
         | || |  _ \| | | |\ \/ /
         | || | | | | |_| |/    \
         |_||_|_| |_|\____|\_/\_/

              Busybox Rootfs

Please press Enter to activate this console.
/ # ls
apps     dev      init     proc     sbin     usr
bin      etc      linuxrc  root     sys
/ # cd apps/
/apps # ls
Image         guest.img     kvm.ko        lkvm-static   stream_c.exe
/apps #
```

图 3.13 Sv58 Host OS 在仿真器中成功运行

具体地为 QEMU 增加 16KB 粒度仿真逻辑的代码和将 Sv58 虚拟内存系统应用到 Kernel 的 5.17 版本中有一些诸如字段对齐，更改内核栈的位置，修改烧录的内核大小等值得注意的细节上的操作，我们都会在附录中详细地说明，由于与原理无关，此处就不再赘述。

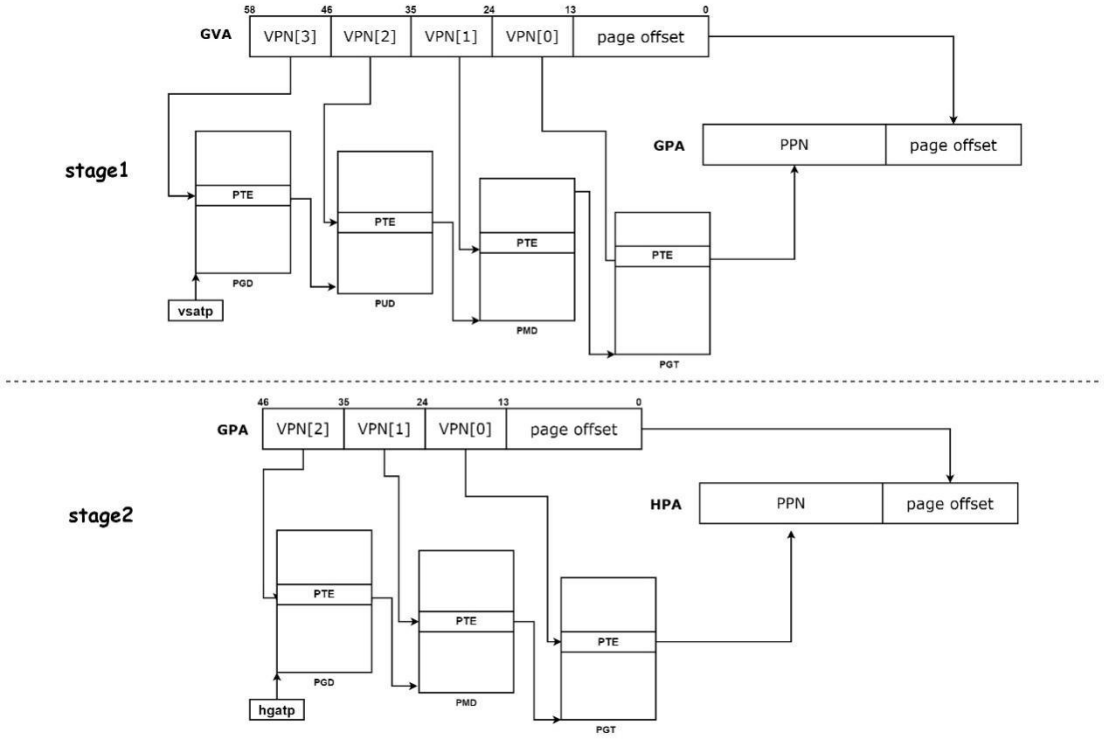


图 3.14 16K two-stage 地址转换过程

同样的，我们根据 3.3 节和 3.4 节的内容也在 Linux Kernel 5.17 版本中实现了 16KB 粒度的 KVM 模块，和 Sv58 相似，16KB 粒度和 4KB 粒度的 two-stage 地址转换也是由于页表粒度不同造成虚拟页号、物理页号和页内偏移量的位数不同，具体流程可以参照 2.3.3 节中的内容，根据我们设计的 Sv58 和 Sv47 页表表项格式和虚拟地址格式，我们在 QEMU 实现了图 3.14 中的 two-stage 地址转换的仿真逻辑。

我们也同样在 Host OS 之上拉起运行了 16KB 粒度的 Guest OS，见图 3.15。

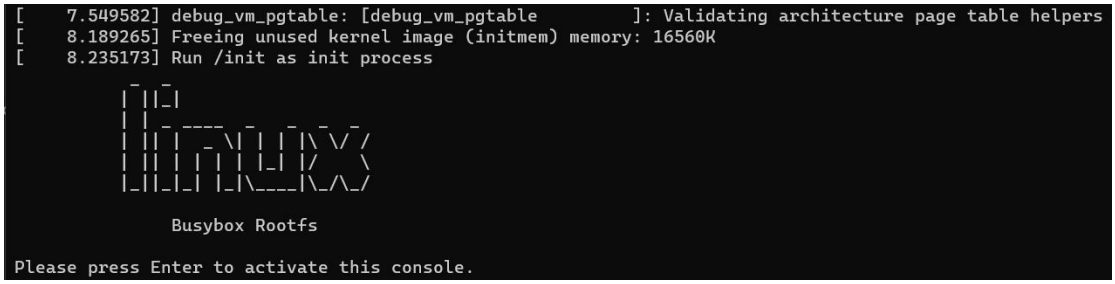


图 3.15 16K 粒度 Guest OS 成功运行

3.6 哈希页表及寻址逻辑的设计

3.6.1 表项设计

根据 2.2.2 节所陈述的哈希页表的现存问题，我们采用了以下三种技术针对哈希页表进行优化：

1) 开放寻址

即采用开放寻址的方式解决哈希页表中的哈希冲突。据我们观察，《Translation Caching: Skip, Don't Walk (the Page Table)》一文中 Itanium 设计架构下的哈希页表采用链地址法解决哈希冲突，这导致了额外的内存开销，极大地影响了性能。因此，我们采用开放寻址的方式，将所有哈希表项直接存储在哈希页表，并采用线性探测，最大程度的利用整个哈希页表的空间。

2) PTE 集簇

将 8 个页号相邻页的 PTE 封装成一个块，存放在一个哈希表项中，使用虚拟页号的最低 12 位在块内进行索引。这样，单个表项中将能存放 8 个 PTE 信息，有助于更好地利用空间局部性。

3) PTE 紧致

PTE 中的最高几位为保留位。这些保留位中没有存放与 PTE 有关的任何信息。因此，将这些保留位移除，可以减小 PTE 的大小，从而在不改变哈希表项大小的情况下，在单个哈希表项中存放更多 PTE。在同等装载密度的前提下，这种设计能进一步节省内存上的开销。

此处我们以 16K 粒度的物理地址为例，根据图 3.7 Sv58 模式的页表表项格式可以看到信息仅存在与前 52 位中，即 7 个 bytes 就可表示 Sv58 页表表项所携带的所有信息，所以我们把 Sv58 的页表表项压缩为 7 个 bytes 来紧致哈希页表，我们就得到如图 3.16 的表项。

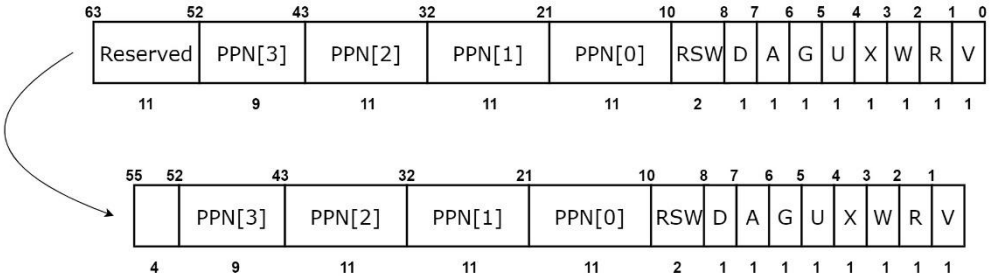


图 3.16 PTE 紧致

基于本项目需要设计实现 stage-2 哈希页表，即 GPA->HPA 的过程。因此，基于上述优化方案，我们设计了存储 GPA->HPA 映射信息的哈希页表。其表项如图 3.17 所示：

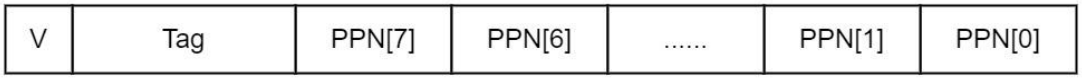


图 3.17 hash page table entry

Valid 位用于指示该表项是否有效，Tag 位用于识别该表项是否为所要查找的表项，余下的 8 个 PTE 则记录了 8 个地址相邻页的入口信息。

相应地，对于一个 GPA 地址，我们将其拆分为如图 3.18 所示的几个部分：

图 3.18 hash address

block number 位用于指示不同的页集簇，也用作与哈希表项的 Tag 位进行比较；block offset 表示块内偏移地址，用于在集簇内定位对应的页；page offset 即页内偏移地址。

3.6.2 寻址逻辑

哈希页表参与的寻址发生在 GPA 的 stage-2 页索引流程中，具体流程如下：

（1）对 GPA 的 PPN 进行哈希，获得表项索引值，并从 hgatp 找到对应的表项。

（2）检查 Valid 位并比对 block_num 与 tag 位，检查是否为同一个页。若不是，则引发缺页失效。

（3）根据 block_offset 在 8 个 PTE 中找到对应的 PTE，并通过 PTE 查出 HPA 的 PPN 部分，结合 page_offset 生成最终的 HPA。

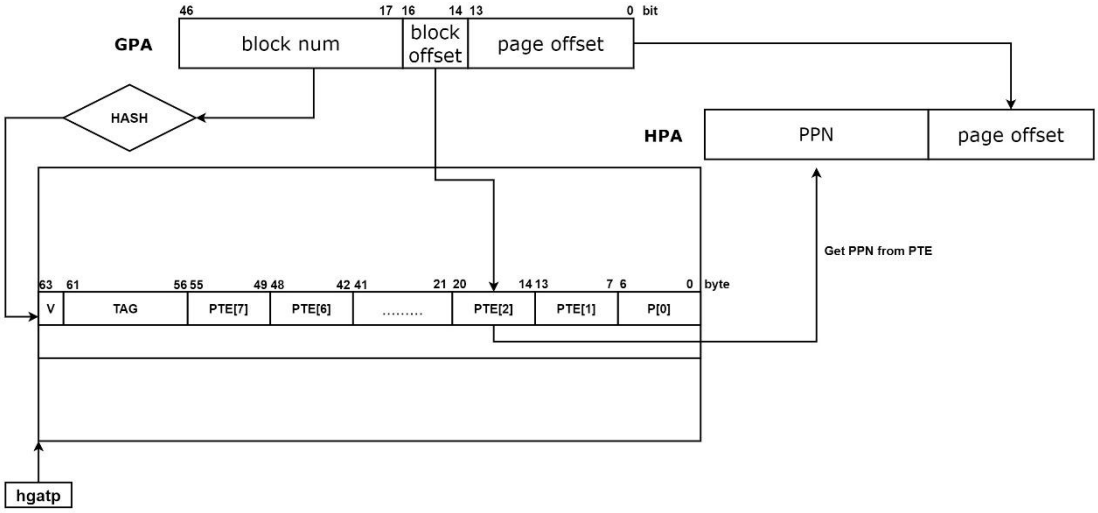


图 3.19 two-stage hash 流程

基于以上的设计原理我们分别得到了 4KB 和 16KB 粒度的哈希表项，分别如图 3.20 和图 3.21 所示。

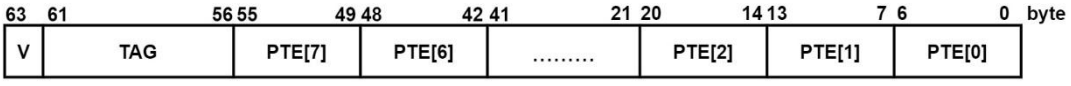


图 3.20 4KB hash page table entry

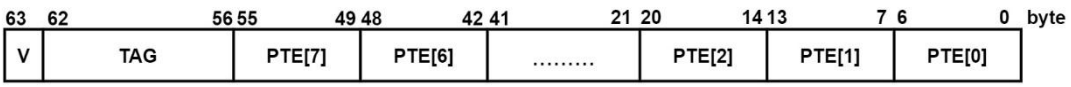


图 3.21 16KB hash page table entry

3.7 基于哈希页表的 KVM 模块的设计

根据 3.6 节中我们设计的哈希页表表项格式以及 3.4 节中完成的基于 16KB 粒度的缺页异常处理，我们在 Linux Kernel 5.17 版本中的 KVM 模块设计并实现了基于哈希页表的缺页异常处理流程。

哈希页表与传统的多级页表不同，无需获取缺失页表项级数（页表扁平化），另外因采取了 PTE 紧致技术，在补充缺失页表项时，需要逐字节赋值。在 hash 冲突处理中，我们选择了开放寻址——线性探测技术，即从发生冲突的位置开始，依次向后探测，直到寻找到下一个空位置为止。缺页处理流程详见图 3.22。

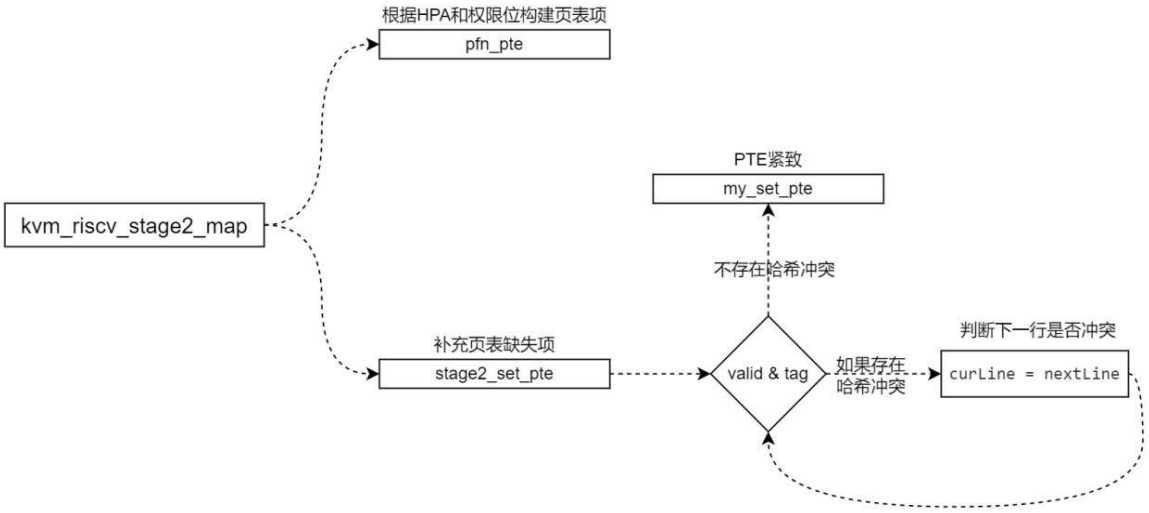


图 3.22 基于哈希页表的 KVM 缺页异常处理流程

虚拟机每释放一个页面，在将自身页表表项清空后，执行 INVLPG 指令，失效 TLB 中的对应的映射，与此同时，虚拟机在执行 INVLPG 时会产生 VM-Exit，告知 KVM 要失效的虚拟机 GPA，通过该 GPA 和哈希页表，可以找到对应的哈希页表表项，释放该项对应的 Host 物理页，并将该项清空。

当退出虚拟化时，KVM 从低到高遍历每一个 Guest 物理页，逐个根据哈希页表找到对应的 Host 物理页，进行释放，同时执行 INVLPG 指令，将 TLB 中的该条映射失效。

3.8 基于哈希页表的仿真器的实现

基于3.6节的哈希页表寻址逻辑，我们在QEMU中实现了基于哈希页表的two-stage地转换逻辑。stage-1阶段的寻址逻辑与3.5节中所展示的设计相同，在此不再赘述；stage-2阶段，我们将GPA拆分为block num、block offset、page offset，并通过前文所述的哈希页表寻址逻辑进行寻址。具体仿真逻辑实现如图3.23所示：

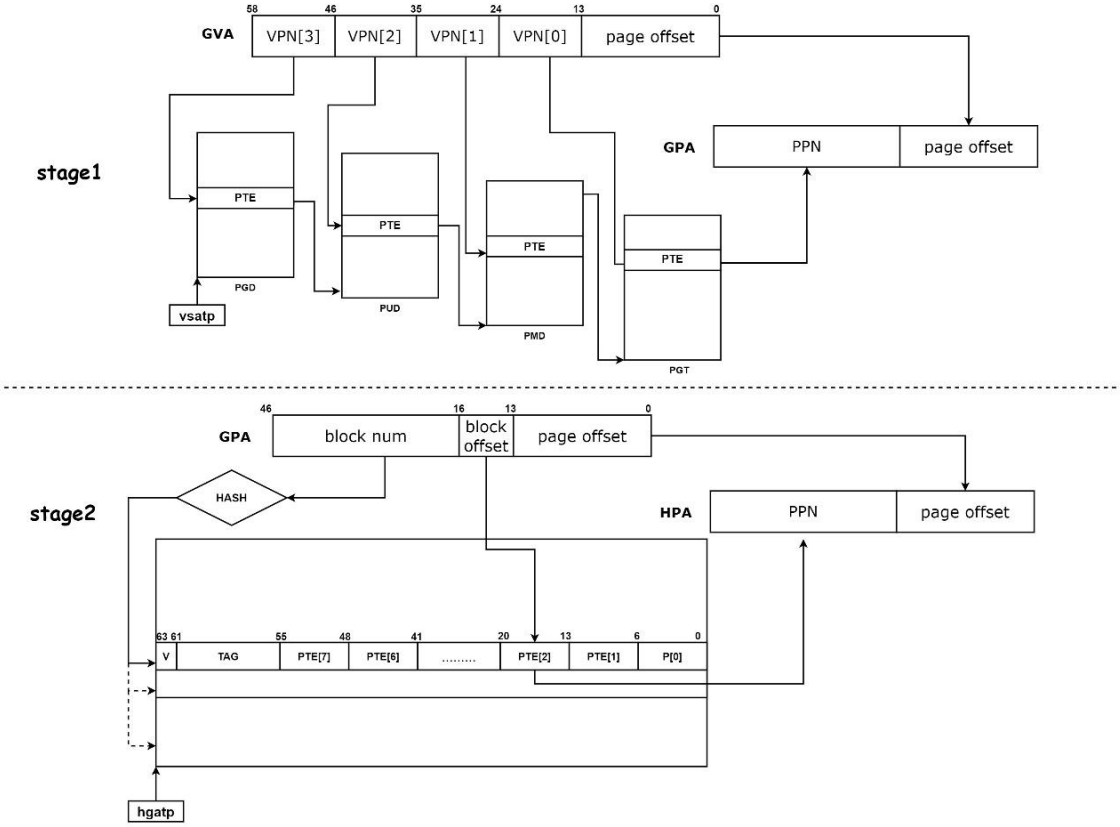


图 3.23 基于哈希页表的 KVM 缺页异常处理流程

基于哈希页表的4KB和16KB粒度的Guest，我们均在仿真平台上成功拉起，分别见图3.24和图3.25。

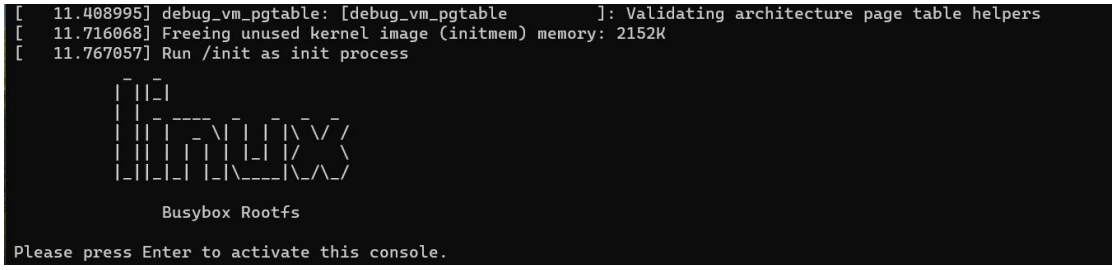
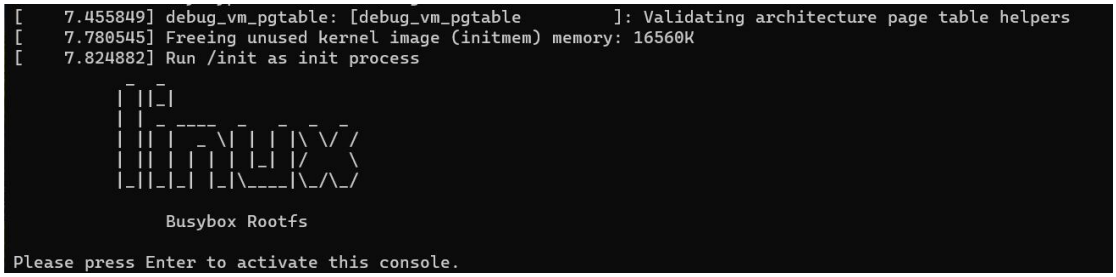


图 3.24 4k-hash Guest OS 成功运行



3.25 16-hash Guest OS 成功运行

4 测试结果及分析

为验证第三章所述的 16K 与哈希页表设计在访存性能上的提升，我们在 Guest OS 上对四个综合性能测试程序集分别进行了响应时间和访存次数的测试（访存次数的精确统计方法在附录 C 中详述）。其中，Guest OS 为第 3.2 节 RISC-V 仿真平台搭建中介绍的使用 KVMTools 在 Host OS 上拉起的操作系统。除此之外，为验证该设计的通用性与可扩展性，我们还通过设置不同的运行场景进行了多次重复测试。

4.1 测试集介绍

首先，对于我们选择与使用的综合性能测试程序集——STREAM，MBW，LMbench 和 Redis-benchmark 介绍如下：

4.1.1 MBW

MBW（内存带宽基准）是用 C 语言编写的开源命令行软件，旨在确定可用于位于用户空间中的程序的“复制”内存带宽。

MBW 通过在内存中复制大量数据来确定可用的内存带宽，该测试是实际应用程序的简单渐进模型，并不适合于极端条件并且不关心硬件结构。它提供

memcpy、dumb、mcblock 等三种速度测试。对于第三种速度测试可以使用 -b 由用户指定固定块的大小。

dumb 为最简单的操作，即逐字节复制数据。

memcpy 是调用 C 和 C++使用的内存拷贝函数，从源内存地址的起始位置开始，拷贝需要大小的字节数到目标内存地址中。

mcblock 同样是调用 C 和 C++使用的内存拷贝函数，不同的是采用固定大小的块进行若干次拷贝。

4.1.2 STREAM

STREAM 是一套由 Virginia University 提供的综合性能测试程序集，通过 Fortran 和 C 两种高级且高效的语言编写完成，由于这两种语言在数学计算方面的高效率，使得 STREAM 测试例程可以充分发挥出内存的能力。STREAM 通过生成以下四种不同模式下的内存读写操作，用于测试高性能计算机的可持续运行的内存带宽最大值。

Copy 为最简单的操作，即从一个内存单元中读取一个数，并复制到另一个内存单元，有 2 次访存操作。

Scale 是乘法操作，从一个内存单元中读取一个数，与常数 scale 相乘，得到的结果写入另一个内存单元，有 2 次访存。

Add 是加法操作，从两个内存单元中分别读取两个数，将其进行加法操作，得到的结果写入另一个内存单元中，有 2 次读和 1 次写共 3 次访存。

Triad 是前面三种的结合，先从内存中读取一个数，与 scale 相乘得到一个乘积，然后从另一个内存单元中读取一个数与之前的乘积相加，得到的结果再写入内存。所以 Triad 操作共有 2 次读和 1 次写共 3 次访存操作。

4.1.3 LMbench

LMbench 是用于评价系统综合性能的多平台开源 benchmark，能够测试包括文档读写、内存操作、进程创建销毁开销、网络等性能。它由 C 语言编写，具有较好的可移植性。

其主要功能包括带宽评测工具和反应时间测评工具。其中带宽测评工具包含读取缓存文件，拷贝内存，读写内存，管道，TCP 等带宽测试程序。而反应

时间测评工具则包含上下文切换，文件系统的建立和删除，进程创建，信号处理，上层的系统调用以及内存读入反应时间的测试程序。

4.1.4 Redis-benchmark

Remote Dictionary Server(Redis) 是一个由 Salvatore Sanfilippo 写的，遵守 BSD 协议、支持网络、可基于内存、分布式的键值对(Key-Value)存储数据库，是跨平台的非关系型数据库。它有以下三个特点：支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用；不仅支持 key-value 类型的数据，还提供 list，set，zset，hash 等数据结构的存储；支持数据的备份。

Redis-Benchmark 可以通过对 Redis 数据库服务进行测试从而反映大型应用端的性能，尤其是并发负载的最终性能表现，其中包括 PING，SET/GET，INCR，LPUSH/RPUSH，LPOP/RPOP，SADD，SPOP 和 LRANGE 等数据库基础操作测评。

4.2 实验结果

我们将分八小节进行实验结果的展示，前三小节围绕访存次数进行了各方面的测评，分别为总访存次数测评，单次操作访存次数测评和平均访存开销测评；之后三小节围绕响应时间进行了综合测评，包括总响应时间测评，单次操作响应时间测评和平均响应时间测评；第七小节通过 Redis-benchmark 进行大型应用端的并发负载性能测试；最后一小节通过在不同运行场景下进行测试证明我们设计的通用性。

4.2.1 总访存次数测评

根据理论分析，在 16K 页表下，由于页大小增加，相同情况下 TLB 覆盖范围会相应增加，TLB 命中率随之提高，最终会减少访问内存次数。另外，在不产生哈希冲突的情形下，哈希页表仅需要一次内存访问，就能完成虚拟地址到物理地址的转换，这与原有的多级页表相比，大大减少了内存访问的次数。

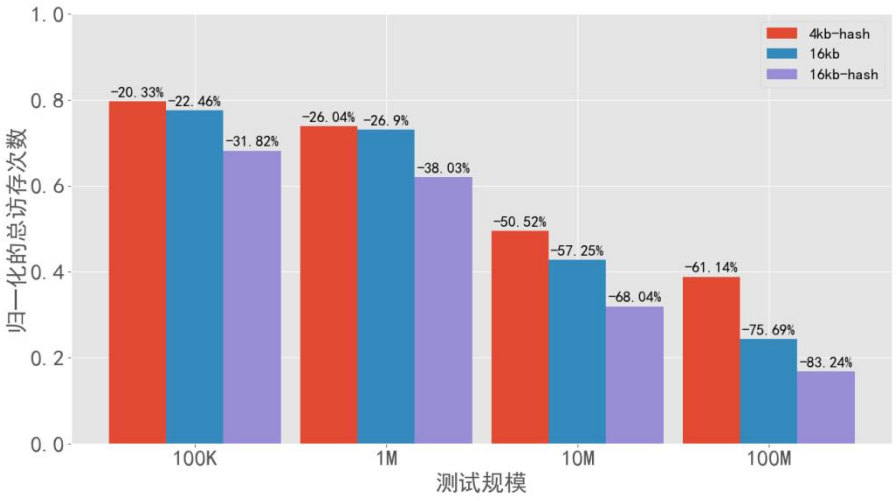


图 4.1 Guest OS 在 MBW 上的测试结果

（归一化总访存次数含义：将 4K 多级页表的总访存次数作为单位 1）

首先以 Guest OS 在 MBW 测试集上的运行结果为例，从页粒度的角度来看，16KB 页粒度的多级页表在三种测试类型上的访存次数均少于 4KB 页粒度的多级页表。从页表实现形式的角度来看，相同页粒度下的哈希页表访存次数均少于多级页表的访存次数。

而随着测试规模的增大，总访存次数的优化效果也越来越明显，在 100M 测试规模下，16KB 多级页表的总访存次数是 4KB 多级页表的 **40%**左右，4KB 哈希页表能将总访存次数降低至 **25%**，而 16KB 哈希页表则仅需不到 **20%**的总访存次数。

另外通过 4KB 多级页表-16KB 多级页表-4KB 哈希页表三者对比来看，增大页粒度与哈希页表均能有效减少访存次数，而增大页表粒度的效果更为明显。

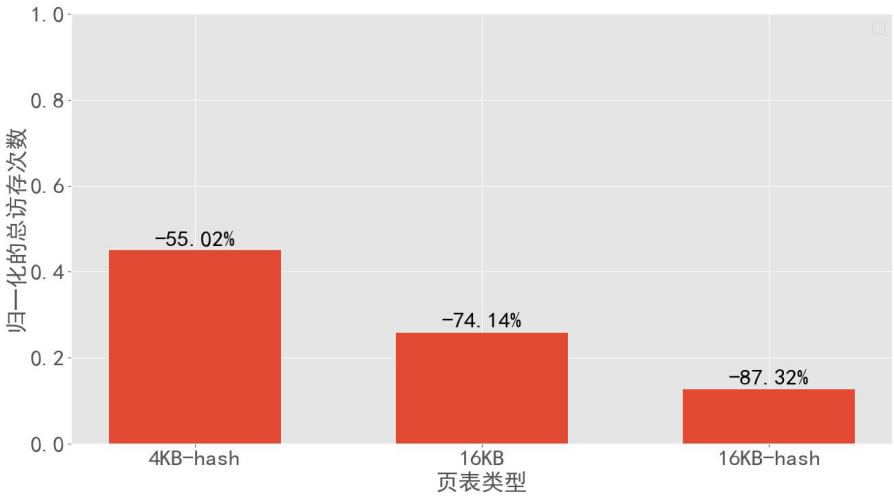


图 4.2 Guest OS 在 STREAM 上的测试结果（测试规模:10M）

Guest OS 在 STREAM 测试集上的运行结果与 MBW 类似。在 10M 测试规模下，16KB 多级页表的总访存次数相比 4KB 多级页表降低了 **55.02%**，4KB 哈希页表能将总访存次数降低 **74.14%**，而 16KB 哈希页表则降低了 **87.32%**的总访存次数。

4.2.2 单次操作访存次数测评

由于总访存次数并不能直观反映不同测试规模下优化效果的变化，我们考察了单次操作对应的访存次数，具体操作为：先在测试集中设定好具体操作的次数，之后将统计的总访存次数除以预先设定的操作次数即可。

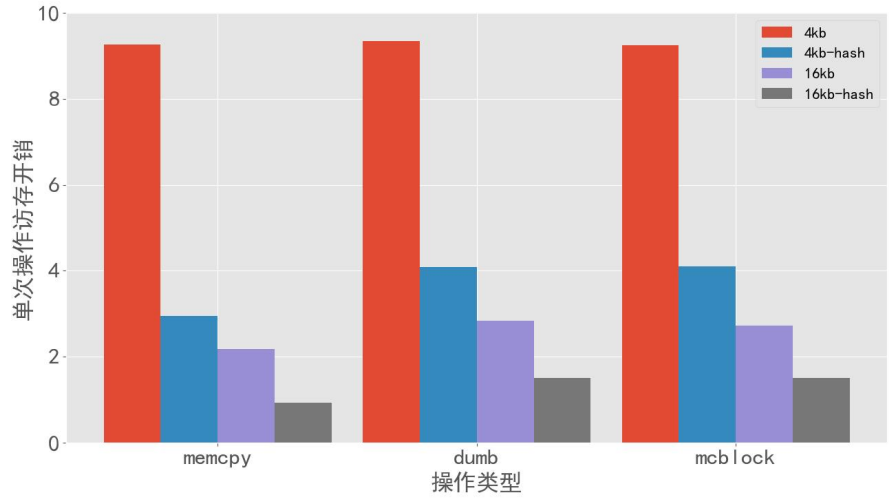


图 4.3 Guest OS 在 MBW 上的单次操作访存开销（测试规模:10M）

首先以 Guest OS 在 MBW 测试集上的运行结果为例，从页粒度的角度来看，16KB 页粒度的多级页表在三种操作类型上的单次操作访存开销均少于 **4.5** 次，而 4KB 页粒度的多级页表的单次操作访存开销均多于 **9** 次，优化效果超过两倍。

另外，通过 4KB 多级页表-16KB 多级页表-4KB 哈希页表三者对比来看，增大页粒度与哈希页表均能有效减少单次操作访存开销，而增大页表粒度的效果更为明显。

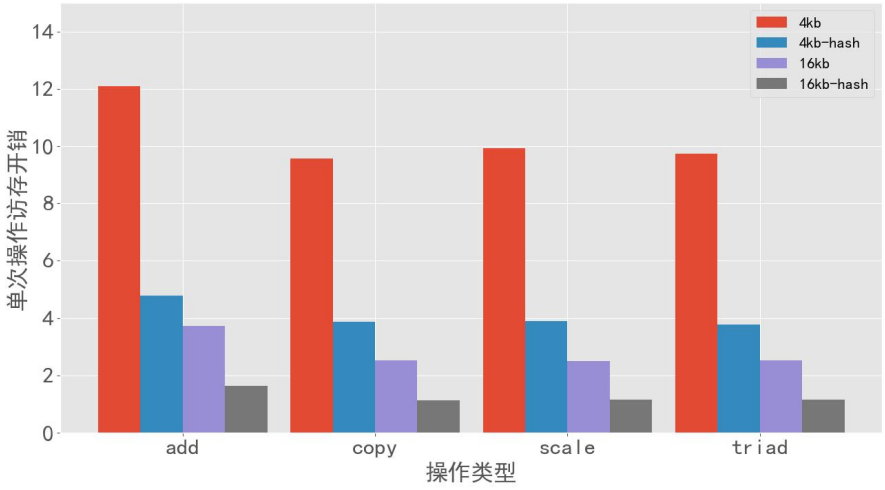


图 4.4　Guest OS 在 STREAM 上的单次操作访存开销（测试规模:10M）

Guest OS 在 STREAM 测试集上的运行结果与 MBW 类似，从页粒度的角度来看，16KB 页粒度的多级页表在四种测试类型上的单次操作访存开销均少于 **4** 次，而 4KB 页粒度的多级页表的单次操作访存开销均多于 **8** 次，优化效果也超过两倍。

综合两个测试集的七种操作，我们可以发现 16KB 哈希页表单次操作的访存开销均小于两次，相比 4KB 多级页表的单次操作访存开销有了明显的优化。

4.2.3 平均访存开销测评

由于不同测试集的操作并不相同，单条操作对应的指令数也不尽相同，我们进一步考察了平均访存开销，具体操作为：修改 QEMU 仿真器，使其记录对物理地址进行内存访问的次数（有效访存次数），之后将统计的总访存次数除以有效访存次数即可。

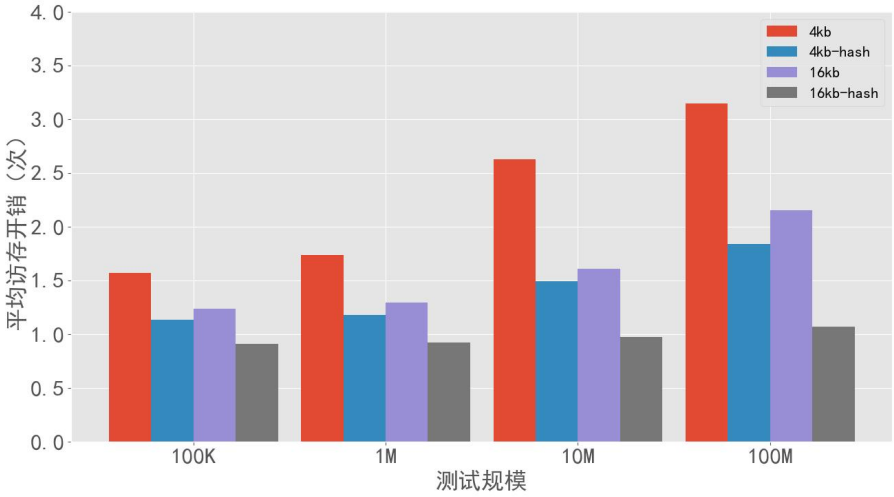


图 4.5 不同测试规模下的 MBW 平均访存开销测试结果

首先以 Guest OS 在 MBW 测试集上的运行结果为例，我们可以发现，增大测试规模，各种页表的平均访存开销也会随之增加。这是因为 TLB 覆盖范围不变的情况下，增大测试规模会减小 TLB 命中的概率，从而增大平均访存开销。而随着测试规模的增大，我们可以发现增大页粒度和使用哈希页表的优化效果变得更明显。

另外，通过 4KB 多级页表-16KB 多级页表-4KB 哈希页表三者对比来看，增大页粒度与哈希页表均能有效减少平均访存开销，而使用哈希页表的效果更为明显。

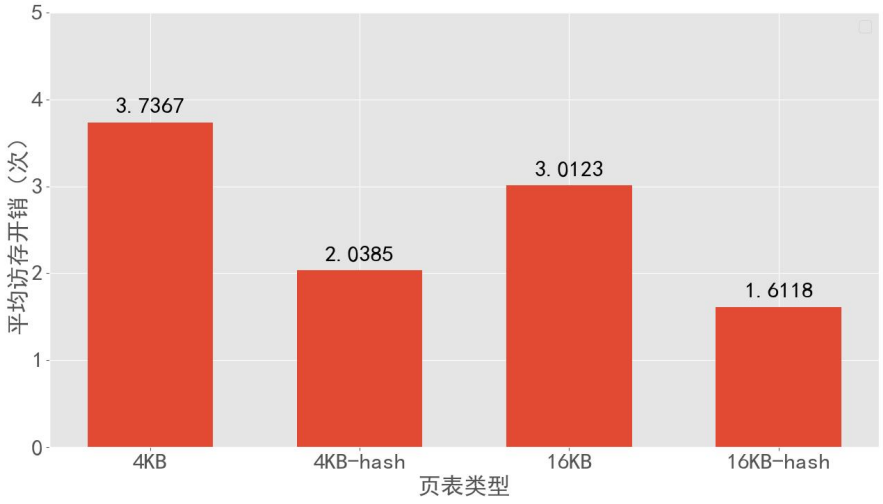


图 4.6 STREAM 平均访存开销测试结果（测试规模:10M）

Guest OS 在 STREAM 测试集上的运行结果与 MBW 类似，从页粒度的角度来看，16KB 页粒度的多级页表在的单次操作访存开销少于 4KB 页粒度的多级页表。从页表实现形式的角度来看，相同页粒度下的哈希页表平均访存开销少于多级页表的平均访存开销。

4.2.4 总响应时间测评

程序响应时间包括访存时间与 CPU 计算时间，我们在第 4.2.1 节总访存次数测评时通过理论分析与实验证明了增大页粒度与使用哈希页表替换多级页表在减少访存次数上均有正向影响，因此理论上，我们采用的这两种技术在减少响应时间上也会有对应的优化效果。

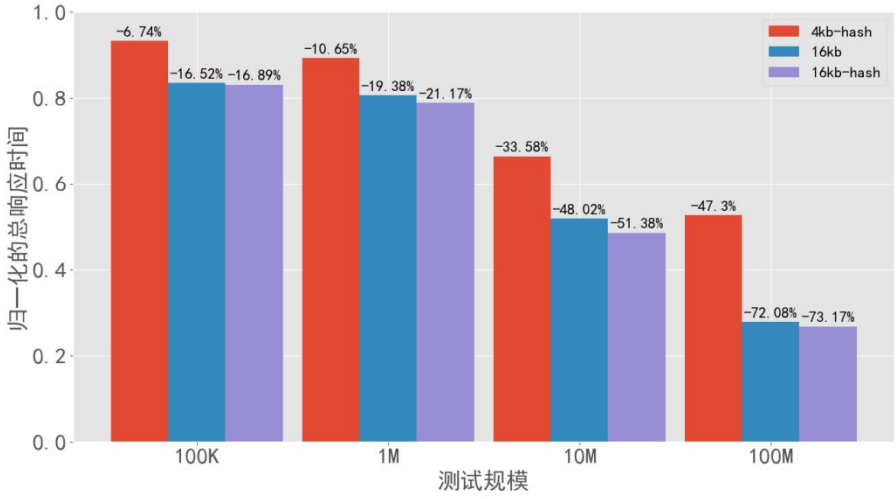


图 4.7 Guest OS 在 MBW 上的测试结果

（归一化总响应时间含义：将 4K 多级页表的总响应时间作为单位 1）

首先以 Guest OS 在 MBW 测试集上的运行结果为例，从页粒度的角度来看，16KB 页粒度的多级页表在三种测试类型上的响应时间均少于 4KB 页粒度的多级页表。从页表实现形式的角度来看，相同页粒度下的哈希页表响应时间均少于多级页表的响应时间。而随着测试规模的增大，优化效果也越来越明显，在 100M 测试规模下，16KB 多级页表的总响应时间是 4KB 多级页表的 **52.7%**，4KB 哈希页表能将总响应时间降低至 **27.92%**，而 16KB 哈希页表则仅需不到 **26.83%**的总响应时间。

另外，通过 4KB 多级页表-16KB 多级页表-4KB 哈希页表三者对比来看，增大页粒度与哈希页表均能有效减少响应时间，而增大页表粒度的效果更为明显。

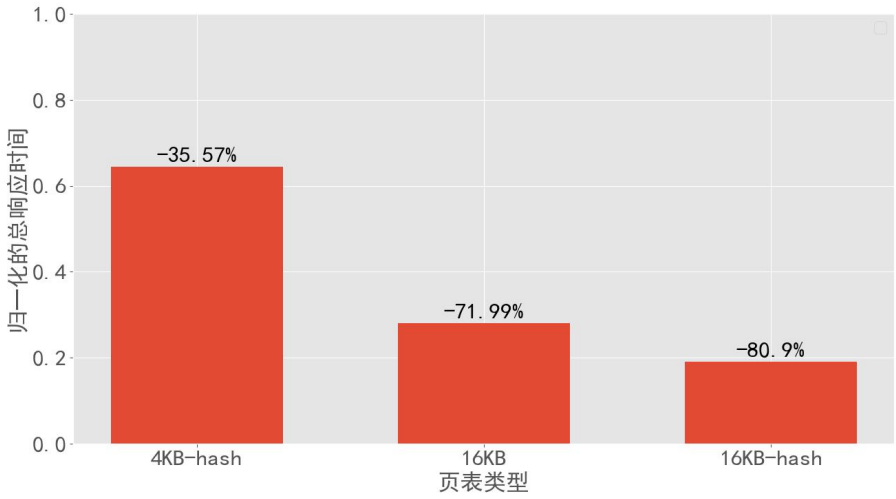


图 4.8 Guest OS 在 STREAM 上的测试结果（测试规模:10M）

同样地，我们从 Guest OS 在 STREAM 测试集上的运行结果上也能得到类似的结论：增大页粒度与哈希页表均能有效减少响应时间，而增大页表粒度的效果更为明显。

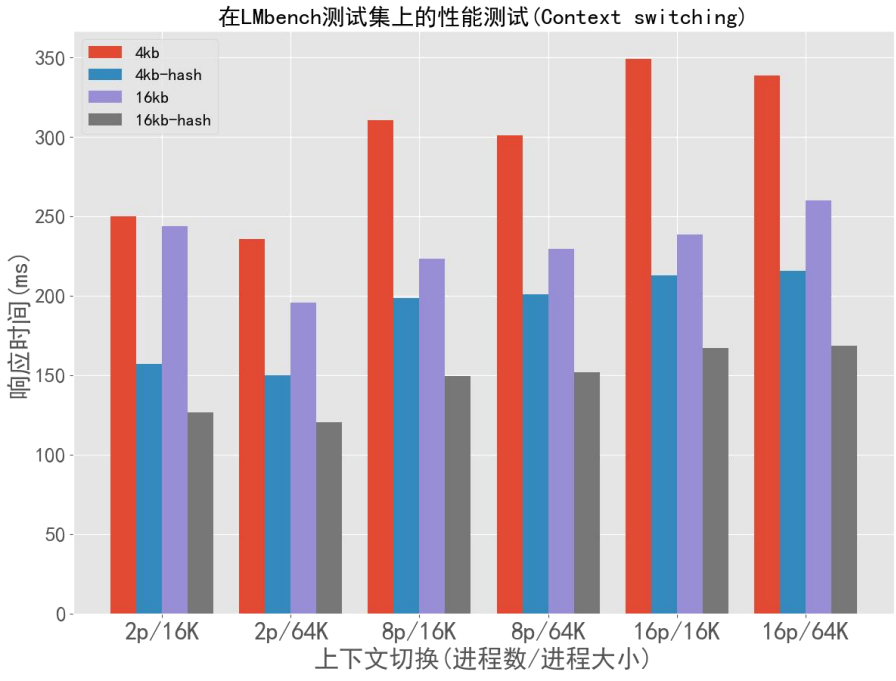


图 4.9 Guest OS 在 LMbench 中的上下文切换测试结果

在使用 LMbench 进行上下文切换测评时，我们通过设置不同的进程数与不同的进程大小进行了多组测试，结果一致表明增大页粒度与使用哈希页表替换多级页表在上下文切换场景下均能有效减少响应时间，而通过 4KB 多级页表-16KB 多级页表-4KB 哈希页表三者对比来看使用哈希页表的效果更为明显。

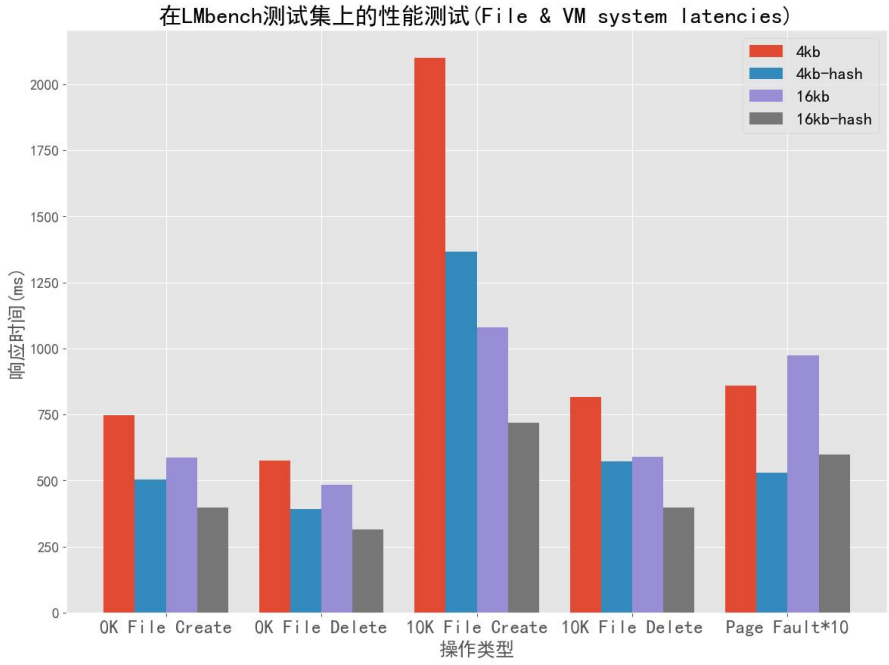


图 4.10 Guest OS 在 LMbench 中的文件与虚拟内存系统测试结果

我们通过使用 LMbench 进行文件系统与虚拟内存系统测评分别得到了创建与删除空文件,创建与删除 10KB 大小文件与处理 10 次缺页异常的响应时间。从创建与删除文件的结果中，我们发现增大页粒度与使用哈希页表替换多级页表均能有效减少响应时间，而通过 4KB 多级页表-16KB 多级页表-4KB 哈希页表三者对比来看使用哈希页表的效果更为明显。

从处理缺页异常的响应时间上看，使用哈希页表替换多级页表能有效减少响应时间，而增大页粒度则会增加响应时间，但使用 16KB 哈希页表仍然优于 4KB 多级页表。

4.2.5 单次操作响应时间测评

由于总响应时间并不能直观反映不同测试规模下优化效果的变化，我们考察了单次操作对应的响应时间，具体操作为：先在测试集中设定好具体操作的次数，之后将统计的总响应时间除以预先设定的操作次数即可。

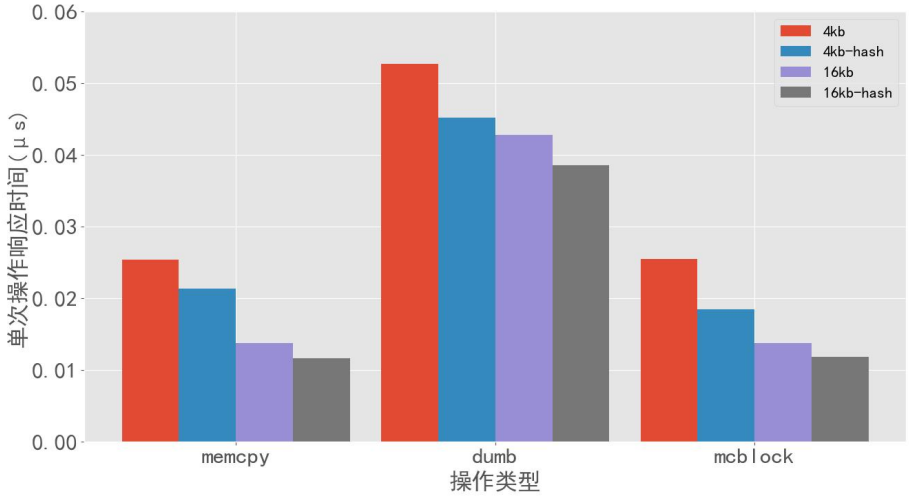


图 4.11 MBW 单次操作响应时间（测试规模:10M）

首先以 Guest OS 在 MBW 测试集上以 10MB 作为测试规模的运行结果为例，从页粒度的角度来看，16KB 页粒度的多级页表在三种测试类型上的单次操作响应时间均少于 4KB 页粒度的多级页表。从页表实现形式的角度来看，相同页粒度下的哈希页表单次操作响应时间均少于多级页表的单次操作响应时间。而通过 4KB 多级页表-16KB 多级页表-4KB 哈希页表三者对比来看，增大页粒度与哈希页表均能有效减少响应时间，而增大页表粒度的效果更为明显。

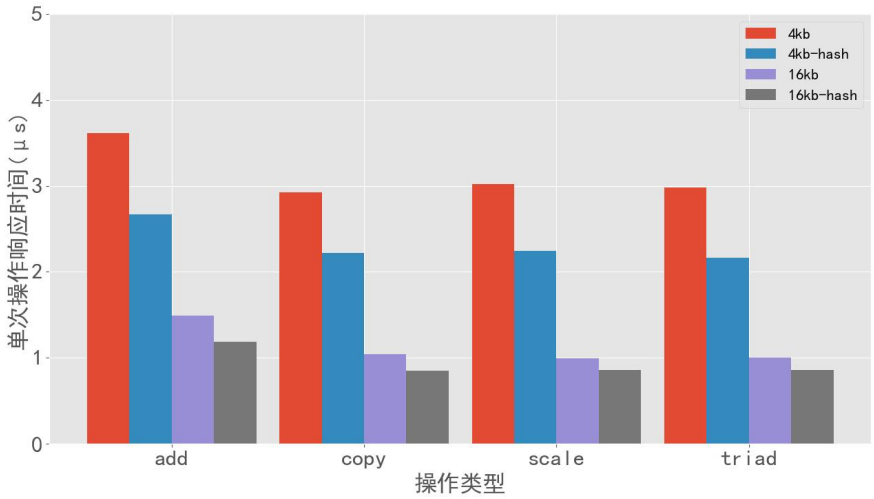


图 4.12 STREAM 单次操作响应时间（测试规模:10M）

Guest OS 在 STREAM 测试集上的运行结果与 MBW 类似，不过 16KB 多级页表的单次操作响应时间仅为 4KB 多级页表的一半，优化效果更明显。

4.2.6 平均响应时间测评

由于不同测试集的操作并不相同，单条操作对应的指令数也不尽相同，我们进一步考察了平均响应时间，具体操作为：修改 QEMU 仿真器，使其记录对物理地址进行内存访问的次数（有效访存次数），之后将统计的总响应时间除以有效访存次数即可。

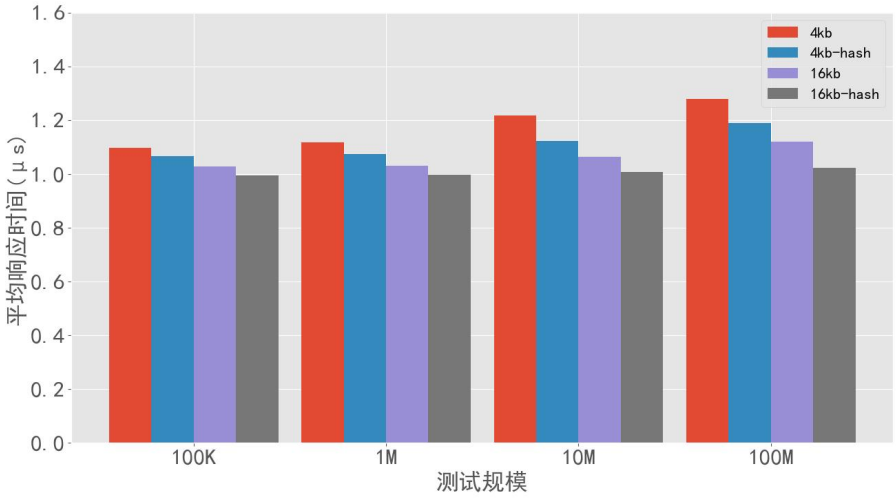


图 4.13 不同测试规模下的 MBW 平均响应时间

首先以 Guest OS 在 MBW 测试集上的运行结果为例，从测试规模上看，在各种页表实现中增大测试规模，平均响应时间会随之增大。这与平均访存开销增大（详见第 4.2.3 节）有着直接联系。

从页粒度的角度来看，16KB 页粒度的多级页表在三个测试规模上的平均响应时间均少于 4KB 页粒度的多级页表。从页表实现形式的角度来看，相同页粒度下的哈希页表平均响应时间均少于多级页表的平均响应时间。

而通过 4KB 多级页表-16KB 多级页表-4KB 哈希页表三者对比来看，增大页粒度与哈希页表均能有效减少响应时间，而增大页表粒度的效果更为明显。这个结果与第 4.2.3 节中有出入，原因是哈希页表在采用 PTE 紧致技术后，牺牲了时间来换空间，增大了单次访存的时间开销。

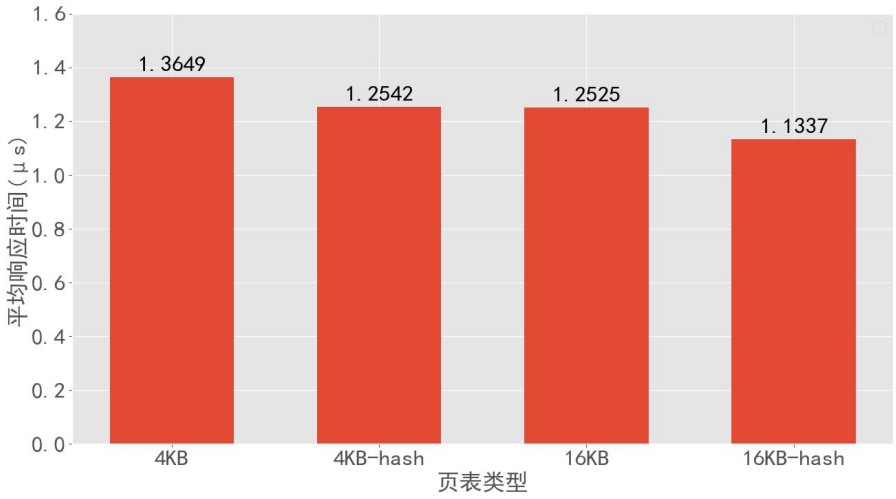


图 4.14 STREAM 平均响应时间

Guest OS 在 STREAM 测试集上的运行结果与 MBW 类似，通过 4KB 多级页表-16KB 多级页表-4KB 哈希页表三者对比来看，增大页粒度与哈希页表均能有效减少响应时间，两者的效果基本相当。

4.2.7 大型应用端的性能测试

在使用不同页表大小与页表实现机制的 Guest OS 上，我们通过搭建 Redis 本地服务器，运行 Redis-benchmark 程序快捷有效地计算出性能参数。其中，测试时使用的参数为：20 个 Redis 客户端同时请求 PING、GET、SET、INCR、LPUSH/RPUSH、LPOP/RPOP 等操作，一共请求一万次。

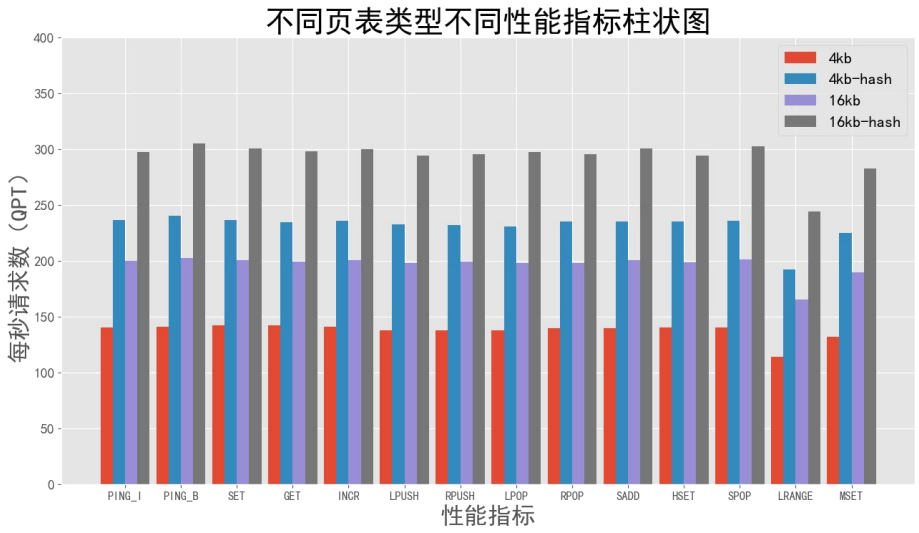


图 4.15 Guest OS 在 Redis-benchmark 上的测试结果

在一系列 Redis 数据库基本操作测评中，我们可以发现：16KB 多级页表的每秒请求数基本是 4KB 多级页表的 **1.4** 倍，4KB 哈希页表的每秒请求数基本是 4KB 多级页表的 **1.7** 倍，而 16KB 哈希页表的每秒请求数基本在 4KB 多级页表的 **2** 倍以上，优化效果明显。

另外，通过 4KB 多级页表-16KB 多级页表-4KB 哈希页表三者对比来看，增大页粒度与哈希页表均能有效增加每秒请求数，而使用哈希页表的效果更为明显。

4.2.8 运行场景测评

在云服务器应用场景中，存在各用户对内存需求均匀与需求不均的两种情况。为体现我们页表设计的通用性，我们模拟了这两种运行场景并进行了性能测试。

现取一组具有代表性的测试进行展示，分为同质与异质的负载两个测试场景，其中同质的负载场景为运行 15 个 10M 计算进程，异质的负载场景为运行 1 个 100M 计算进程与 5 个 10M 计算进程。

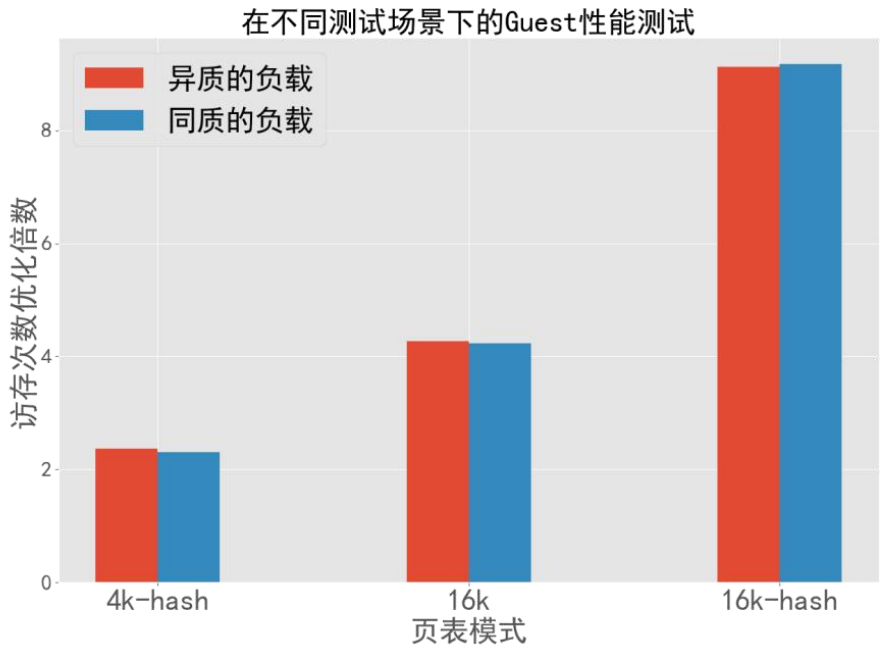


图 4.16 Guest OS 在不同运行场景下的测试结果（访存次数）

表 4.1 不同运行场景下的性能优化比例（访存次数）

运行场景 页表格式	4k-hash (与 4k 比较)	16k (与 4k 比较)	16k-hash (与 4k 比较)
异质的负载	2.366253345	4.273567013	9.129428605
同质的负载	2.307236034	4.235403911	9.180770291

如图 4.16 与表 4.1 所示，在同质与异质的负载两种场景下，我们设计的不同页表格式与策略于访存次数上具有一致的优化效果：采用 4K 大小与哈希页表寻址策略优化比例为 2.3-2.4 倍，采用 16K 大小与多级页表寻址策略优化比例为 4.2-4.3 倍，而采用 16K 大小与哈希页表寻址策略优化比例为 9.1-9.2 倍。

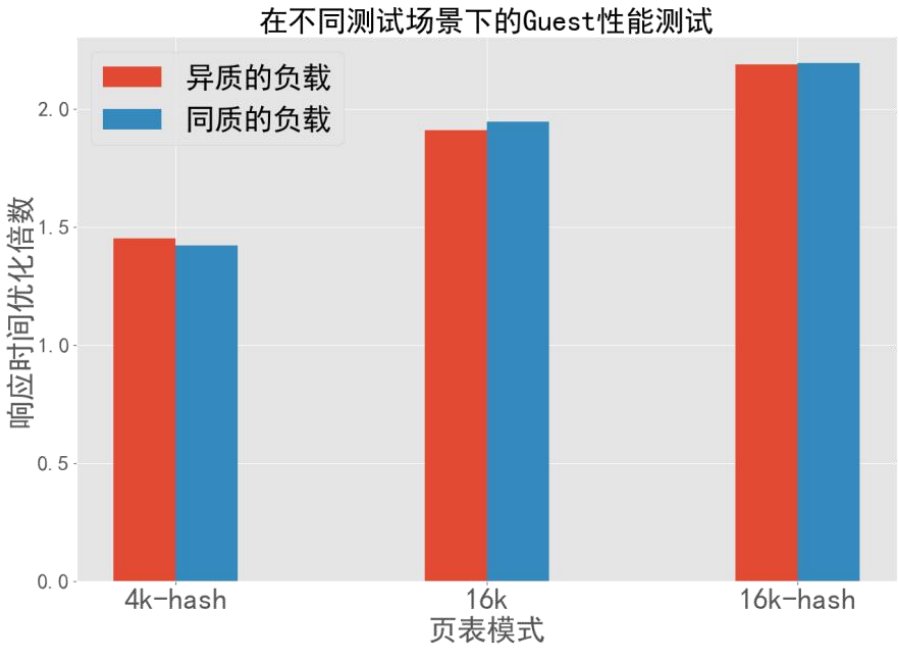


图 4.17 Guest OS 在不同运行场景下的测试结果（响应时间）

表 4.2 不同运行场景下的性能优化比例（响应时间）

运行场景 页表格式	4k-hash (与 4k 比较)	16k (与 4k 比较)	16k-hash (与 4k 比较)
异质的负载	1.453526649	1.911064087	2.189873555
同质的负载	1.423722563	1.947086033	2.196716285

如图 4.17 与表 4.2 所示，在同质与异质的负载两种场景下，我们设计的不同页表格式与策略于响应时间上也具有一致的优化效果：采用 4K 大小与哈希页表寻址策略优化比例为 **1.4-1.5** 倍，采用 16K 大小与多级页表寻址策略优化比例为 **1.90-1.95** 倍，而采用 16K 大小与哈希页表寻址策略优化比例为 **2.19-2.20** 倍。

综合上述实验结果，可以得出：在同质与异质的负载两种场景下，增大页粒度与使用哈希页表替换多级页表在性能优化上具有一致的效果。这也充分体现了我们页表设计的通用性。

4.3 实验结论

上述实验证实了增大页粒度以及使用哈希页表替换多级页表在所给定 STREAM 测试集，MBW 测试集，LMbench 测试场景下对于总访存次数、单次操作访存次数、平均访存次数以及总响应时间、单次操作响应时间、平均响应时间的优化效果，其中使用 16K 哈希页表在绝大多数场景下能够取得最优异的运行结果。

结合第 4.2.7 节大型应用端的性能测试结果，我们可以判断在大型应用请求上，我们的三种页表设计也有一定的优化效果。另外，结合第 4.2.8 节运行场景测评结果，我们可以判断在云计算场景面对不同用户需求时，我们的三种页表设计也能优于 4K 多级页表。

5 总结与展望

在本项目中，我们从 QEMU 仿真器出发，自底向上搭建起 QEMU->Host OS->KVM->Guest OS 的运行仿真环境链。在此基础上，设计并实现了 16KB 页粒度页表与哈希页表的 two-stage 页查找完整仿真运行逻辑。与此同时，我们使用了 STREAM、MBW、LMbench 和 Redis 等 benchmark，测试了系统在多种内存访问情景与不同内存访问规模下的总体表现，通过实验数据证实了 16KB 页与哈希页表在一定应用场景下的优化潜力，达成了预期目标。

当然，该项目依然存在较大的探索空间。本项目在实现 16KB 页粒度哈希页表并达到优化目标后，也一并实现了 4KB 哈希页表作为补充与对比，实验结果表明哈希页表在 4KB 与 16KB 页粒度上均有优化效果。此外，本项目实现了 stage-2 哈希页表查询流程，属于一维页表查询，在未来，可以向二维乃至多维页表查询性能的方向进行探索与深挖。最后，本项目的实验结果表明，增大页粒度能在一定应用场景下稳定提升系统性能，因此，未来也可在更大页粒度，如 32KB、64KB 的情景下进行探索，充分挖掘系统在更大页粒度应用场景下的潜力。

6 参考文献

[1] P. Weisberg and Y. Wiseman, "Using 4KB page size for Virtual Memory is obsolete," 2009 IEEE International Conference on Information Reuse & Integration, 2009, pp. 262-265, doi: 10.1109/IRI.2009.5211562.

[2] Idan Yaniv and Dan Tsafrir. 2016. Hash, Don't Cache (the Page Table). SIGMETRICS Perform. Eval. Rev.44, 1 (June 2016), 337–350. https://doi.org/10.1145/2964791.2901456

[3] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation caching: skip, don't walk (the page table). SIGARCH Comput. Archit. News 38, 3 (June 2010), 48–59. https://doi.org/10.1145/1816038.1815970

[4] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2022. Every walk's a hit: making page walks single-access cache hits. Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Association for Computing Machinery, New York, NY, USA, 128–141. https://doi.org/10.1145/3503222.3507718

[5] ARM® Learn the architecture - AArch64 Virtualization .

附录 A

附录 A 中记录了我们实验中遇到的困难和对应的解决方案，以及我们对后续工作的一些思路。

问题一：在 QEMU 中实现了 16KB 粒度多级页表仿真逻辑后，无法验证其正确性。

解决方法：我们实现了一个简易的 Test OS，它具有 16KB 粒度的多级页表映射虚拟内存，可以根据时间中断进行最短时间调度进程切换输出信息。利用这个 Test OS 在 16KB 粒度下的 Bare 模式、Sv47、Sv58 均成功的运行，我们验证了我们在 QEMU 中实现的 16KB 粒度的多级页表仿真逻辑的正确性。

问题二：由于我们对 QEMU 进行了修改，Test OS 和加载了 16KB 粒度的内存子系统的 Linux Kernel 5.17 也是第一次运行，我们很难去确定 BUG 处在模拟器还是内核，此外在启动虚拟内存前 GDB 和 printk 也无法提供很好的帮助。

解决方法：我们先是通读了一遍 QEMU 的源码，掌握了 QEMU 模拟的工作原理，并在 QEMU 中做了一些调整，将一些关于地址的关键信息输出出来，并结合物理地址，虚拟地址和符号表中的关键地址，利用 GDB 设置断点，来进行综合的调试和判断。

问题三：出现了 BUG 后，由于没有开启虚拟内存，无法输出信息，符号表也只记载了函数入口的地址，无法具体定位到内核运行错误的位置。

解决方法：我们首先利用 OpenSBI 的提供终端和中断服务自己实现了一个 printk 函数，用来输出信息。之后，我们定义一些测试函数来确定程序运行的位置，利用 GDB 不断设置断点来确定程序运行的位置，此处尤其需要注意为这些函数设置不可被优化的标识，或者直接取消掉-O3 优化，否则一些上下文关联性较差的函数会被编译器优化掉。

问题四：在 setup_vm_final()函数中寻找分配空间失败。

解决方法：原因是 memset 函数失败，无法初始化一个页面，对这种情况的发生原因，我们对整个流程进行了跟踪，发现是 fixmap 映射页面失败了，我们根据 fixmap 的原理从新设计的虚拟地址空间布局，调整了 page_offset，就可以顺利的分配页面了。

问题五：在 setup_vm_final()函数中开启虚拟内存后，函数执行失败，内核崩溃。

解决方法：这个问题我们通过 QEMU 中的访问地址错误类型来判断是缺少写权限，所以导致函数参数无法压入栈中，导致内核崩溃，我们经过判断是由于更改为 16KB 粒度，更改后导致两块区域赋权限位的时候出现了重叠，将内核栈又赋成只读权限了，所以我们重新赋给内核栈读写权限后，问题解决。

问题六：烧录的内核镜像和根文件系统位置重叠

解决方法：由于修改了段对齐，导致内核空间过大，所以和根文件系统重叠了，我们只能修改了段对齐方式，压缩了空间，同时又修改了地址空间的布局，自此载有 16KB 粒度的 Linux Kernel 5.17 版本就能顺利运行了。

问题七：在进行性能测试时如何准确统计访存次数

解决方法：在虚拟空间中固定一个访问点，连续三次访问该固定访问点，统计开始，在 QEMU 中维护一个计数器记录访存次数，测试结束再连续三次访问该固定访问点，结束统计并输出计数器的值。

问题八：LMbench 测试程序编译时报错：对 llseek 未定义的引用

解决方法：因为大数据重新定位读/写文件偏移量时，llseek 接口函数在 GCC（>7.3.0）编译器中被 lseek64 替代，我们将测试工具的 disk.c 源码文件中的 llseek 接口函数全部替换成 lseek64，重新编译解决了该问题。

问题九：测试程序中的时间包括运算的时间，需要在进行性能测试时准确统计访存的时间。

解决方法：通过阅读 QEMU 对于内存管理的代码，理解 QEMU 内存管理的流程，在其模拟内存访问的若干函数中加入计时的代码，并输出到文件中进行记录，起始和结束的信号和记录次数的开始和结束信号相同，详见问题七。

附录 B

附录 B 中我们将介绍项目实现过程中的部分重要代码。

1. 在 Linux Kernel 5.17 RISC-V 架构中实现 16KB 内存子系统

   ■ 修改宏定义

   按照我们的 Sv58 页表格式，我们首先需要修改 PAGE_SHIFT, PGDIR_SHIFT_L4, PGDIR_SHIFT_L3, PUD_SHIFT, PMD_SHIFT 宏的定义。

```
-- linux/arch/riscv/include/asm/page.h

# define PAGE_SHIFT (14)
-- linux/arch/riscv/include/asm/pgtable-64.h

#define PGDIR_SHIFT_L3     36
#define PGDIR_SHIFT_L4     47
#define PUD_SHIFT          36
#define PMD_SHIFT          25
```

   ■ relocate

   当我们修改页面大小为 16KB 时，原有的一个 PMD 映射：虚拟地址 0xffffffff80000000 映射到 0x80200000 不再成立。

   所以我们修改以下内容，使得 relocate 依然能够正常运行。

```
-- linux/arch/riscv/include/asm/pgtable.h

#define KERNEL_LINK_ADDR (ADDRESS_SPACE_END - SZ_2G + 1 + 0x200000)
linux/arch/riscv/kernel/head.S

relocate:
 /* Relocate return address */
 la a1, kernel_map
 XIP_FIXUP_OFFSET a1
 REG_L a1, KERNEL_MAP_VIRT_ADDR(a1) # a1 为虚拟地址,需要映射到 80200000
 li t1,0x200000
 add a1,a1,t1
 la a2, _start
 sub a1, a1, a2
 add ra, ra, a1
-- linux/arch/riscv/mm/init.c

```

```
kernel_map.virt_addr = KERNEL_LINK_ADDR - 0x200000;
kernel_map.phys_addr = (uintptr_t)(&_start) - 0x200000;
kernel_map.va_pa_offset = PAGE_OFFSET - (kernel_map.phys_addr + 0x200000);
```

- fixmap

在进行 fixmap 映射时，注意到程序要求 fixmap 的起始地址能整除 PMD_SIZE，为了满足这一要求，我们需要重新设计虚拟内核空间布局。因此，这里我们通过修改 PAGE_OFFSET 的值来调整虚拟内核空间布局。

```
config PAGE_OFFSET
default 0xfee0000001000000 if 64BIT
```

- KVM

按照我们的 Sv58 页表格式，我们首先需要修改 stage2_index_bits 宏的定义。

```
#define stage2_index_bits   11
```

为了页表翻译过程中能准确得到对应的 VPN 我们需要修改 stage2_page_size_to_level 以及 stage2_level_to_page_size 这一对函数。

```
static int stage2_page_size_to_level(unsigned long page_size, u32 *out_level)
{
    u32 i;
    unsigned long psz = 1UL << HGATP_PAGE_SHIFT;

    for (i = 0; i < stage2_pgd_levels; i++) {
        if (page_size == (psz << (i * stage2_index_bits))) {
            *out_level = i;
            return 0;
        }
    }

    return -EINVAL;
}

static int stage2_level_to_page_size(u32 level, unsigned long *out_pgsize)

{
```

```
    if (stage2_pgd_levels < level)
        return -EINVAL;

    *out_pgsize = 1UL << (HGATP_PAGE_SHIFT + (level * stage2_index_bits));

    return 0;
}
```

- 段对齐

  按照我们的 Sv58 页表格式，需要修改段对齐的大小，而段对齐大小由宏 SECTION_ALIGN 控制，故修改该宏。

```
#define SECTION_ALIGN (1 << 24)
```

2. 实现 stage-2 缺页流程中建立哈希页表，并采用哈希页表进行地址转换
   - 建立页表预置要求

     修改两个工具函数：

     - my_set_pte 修改自 stage2_set_pte，目的是为了将 value 值填入 pte 表项中。

```
void my_set_pte(pte_t * ptep, unsigned long value){
    //将 ptep 的前 7 个 bytes 置为 value 的值
    unsigned char* charValue = (unsigned char*)&value;
    unsigned char* charPte = (unsigned char*)ptep;
    int i=0;
    for(i=0;i<7;i++){
        charPte[i] = charValue[i];
   }
}
```

     - my_pte_val 修改自 pte_val，目的是为了获取 pte 表项的值。

```
unsigned long my_pte_val(pte_t pte){
     //取出 pte 前 7 个 bytes 的值
     unsigned long  tmp = 0;
     unsigned char* charPte = (unsigned char*)&pte;
     unsigned char* charValue = (unsigned char*)&tmp;
     int i=0;
     for(i=0;i<7;i++){
```

```
            charValue[i] = charPte[i];
        }
        return tmp;
    }
```

■ 建立页表预置要求

实现缺页填写页表操作：实现 hash 函数对于 addr 的映射以及开放寻址。

```
static int stage2_set_pte(struct kvm *kvm, u32 level,
                          struct kvm_mmu_memory_cache *pcache, gpa_t addr,
                          const pte_t *new_pte)
{
    //获取 PGD 的起始地址
    unsigned char *tmpPgd = (unsigned char *)(kvm->arch.pgd);
    //获取哈希的 idx 和组号
    unsigned long blockNum = (unsigned long)addr >> 17;
    unsigned long hashIdx = blockNum & 0xfff; //取后 12 位作为 idx
    unsigned long blockOff = (unsigned long)addr << 47 >> 61;
    //获取该行的 tag，需要进行比对是否一致
    unsigned char *curLine = tmpPgd + (hashIdx * 64);

    //前 8 位(再去掉最开始两个)为 tag 位
    unsigned long tag, valid;
    unsigned long i = 0, j = 0;

    //startTable
    unsigned char *startTable;
    //PTE 开始的地方
    unsigned char *startPTE = (unsigned char *)&(new_pte->pte);
    while (1) {
        tag = *((unsigned long *)curLine);
        startTable = curLine + blockOff * 7 + 8;
        valid = tag >> 56;
        if (valid) {
            if ((tag & 0x0000ffffffffffff) == blockNum) {
                //write PTE
                my_set_pte((pte_t *)startTable,startPTE);
                break;
            } else {
                //跳到下一行
                curLine = (curLine+64) % (16384*32);
```

```
            }
        } else {
            //write PTE
            my_set_pte((pte_t *)startTable,startPTE);
            //write TAG
            *((unsigned long *)curLine) =
                0x0100000000000000 | blockNum;
            break;
        }
    }
    return 0;
}
```

- 实现建表相关函数
  - stage2_op_pte 是负责初始化页表项的函数，需要将其修改为支持哈希页表版本。

```
static void stage2_op_pte(struct kvm *kvm, gpa_t addr, pte_t *ptep,
                          u32 ptep_level, enum stage2_op op)
{
    int i, ret;
    pte_t *next_ptep;
    u32 next_ptep_level;
    unsigned long next_page_size, page_size;

    page_size = PAGE_SIZE;

    BUG_ON(addr & (page_size - 1));

    //判断这个 pte 为 0
    if (!my_pte_val(*ptep))
        return;

    if (op == STAGE2_OP_CLEAR)
        my_set_pte(ptep, 0);//定义函数:
    else if (op == STAGE2_OP_WP)
        my_set_pte(ptep, my_pte_val(*ptep) & ~_PAGE_WRITE);
}
```

  - stage2_get_leaf_entry 是根据获取 addr 对应页表项的函数，在多级页表中该函数会迭代查找，直到找到叶节点（存储最终的物理地址）。

```
static bool stage2_get_leaf_entry(struct kvm *kvm, gpa_t addr, pte_t **ptep,
                                  u32 *ptep_level)
 {
     //addr ---hash---> 找到对应的位置
     //获取 PGD 的起始地址
     unsigned char *tmpPgd = (unsigned char *)(kvm->arch.pgd);
     //获取哈希的 idx 和组号
     unsigned long blockNum = (unsigned long)addr >> 17;
     unsigned long hashIdx = blockNum & 0xff; //取后 8 位作为 idx
     unsigned long blockOff = (unsigned long)addr << 47 >> 61;
     //获取该行的 tag，需要进行比对是否一致
     unsigned char *curLine = tmpPgd + (hashIdx * 64);

     //前 8 位(再去掉最开始两个)为 tag 位
     unsigned long tag, valid;
     unsigned long i = 0, j = 0;

     //startTable
     unsigned char *startTable;

     while (1) {
         tag = *((unsigned long *)curLine);
         startTable = curLine + blockOff * 7 + 8;
         valid = tag >> 56;
         if (valid) {
             if ((tag & 0x0000ffffffffffff) == blockNum) {
                 //找到正确的位置
                 *ptep_level = 1;
                 *ptepp = (pte_t *)startTable;
                 break;
             } else {
                 //跳到下一行
                 curLine = (curLine+64) % (16384*32);
             }
         } else {
             return false;
         }
     }
     return true;
 }
```

3. QEMU

   - 实现 16KB 粒度多级页表

为实现 stage-2 的 16KB 粒度的地址转换仿真逻辑，我们在 get_physical_address 函数中进行了修改。

```
switch (vm)
    {
    case VM_1_10_SV32:
        levels = 2;
        ptidxbits = 10;
        ptesize = 4;
        break;
    case VM_1_10_SV39:
        levels = 3;
        ptidxbits = 11;
        ptesize = 8;
        break;
    case VM_1_10_SV48:
        levels = 4;
        ptidxbits = 11;
        ptesize = 8;
        break;
    case VM_1_10_SV57:
        levels = 5;
        ptidxbits = 11;
        ptesize = 8;
        break;
    case VM_1_10_MBARE:
        *physical = addr;
        *prot = PAGE_READ | PAGE_WRITE | PAGE_EXEC;
        return TRANSLATE_SUCCESS;
    default:
        g_assert_not_reached();
    }
```

- 实现哈希页表仿真逻辑

为实现 stage-2 的哈希地址转换仿真逻辑，我们实现了 get_physical_address_hash 函数，在哈希页表中进行查询从而完成虚拟地址到物理地址的转换。

```
static int get_physical_address_hash(CPURISCVState *env, hwaddr *physical,

                                     int *prot, target_ulong addr,
                                     target_ulong *fault_pte_addr,
                                     int access_type, int mmu_idx,
```

```
                                        bool first_stage, bool two_stage,
                                        bool is_debug)
{
    //env, &vbase, &vbase_prot, base, NULL, MMU_DATA_LOAD,
    //mmu_idx, false, true, is_debug
    CPUState *cs = env_cpu(env);
    MemTxResult res;
    MemTxAttrs attrs = MEMTXATTRS_UNSPECIFIED;
    /* check that physical address of PTE is legal */
    hwaddr pte_addr;
    int mxr = get_field(env->vsstatus, MSTATUS_MXR);

    unsigned long blockNum = (unsigned long)addr >> 17;
    unsigned long idx = blockNum & 0xfff; //取后 12 位作为 idx
    unsigned long block_offset = addr << 47 >> 61;
    unsigned long tag;
    unsigned long valid;
    hwaddr base = (hwaddr)get_field(env->hgatp, SATP64_PPN) << PGSHIFT;
    hwaddr block_addr = base + idx * 64;
    while (1)
    {
        tag = address_space_ldq(cs->as, block_addr, attrs, &res);
        valid = tag >> 56;
        if (!valid)
            return TRANSLATE_G_STAGE_FAIL;
        if ((tag & 0x0000ffffffffffff) == addr >> 17)
            break;
        block_addr = (block_addr-base+64)%(16384*32)+base;
    }

    pte_addr = block_addr + block_offset * 7 + 8;
    unsigned long pte = address_space_ldq(cs->as, pte_addr, attrs, &res);
    pte = pte & 0x00ffffffffffffff;
    *physical = (pte >> 10 << 14) | (addr & 0x3fff);
    if (!(pte & PTE_V))
    {
        /* Invalid PTE */
        return TRANSLATE_FAIL;
    }
    else if (access_type == MMU_DATA_STORE && !(pte & PTE_W))
    {
        /* Write access check failed */
        return TRANSLATE_FAIL;
    }
```

```
49.
50.     if ((pte & PTE_R) || ((pte & PTE_X) && mxr))
51.     {
52.         *prot |= PAGE_READ;
53.     }
54.     if ((pte & PTE_X))
55.     {
56.         *prot |= PAGE_EXEC;
57.     }
58.     if ((pte & PTE_W) && (access_type == MMU_DATA_STORE || (pte & PTE_D)))
59.     {
60.         *prot |= PAGE_WRITE;
61.     }
62.     return TRANSLATE_SUCCESS;
63. }
```

附录 C

附录 C 中我们将介绍测试过程中部分测试指标的统计方法。

（一）总访存次数

总访存次数包含查询 stage-1 与 stage-2 页表，进行虚拟地址到物理地址转换的次数。

首先我们需要修改 QEMU 仿真器，使其记录总访存次数，这一部分在 `qemu/target/riscv/cpu_helper.c` 中的 `get_physical_address` 函数中：

```
1.  if (riscv_cpu_mxl(env) == MXL_RV32) {
2.      pte = address_space_ldl(cs->as, pte_addr, attrs, &res);
3.  } else {
4.      pte = address_space_ldq(cs->as, pte_addr, attrs, &res);
5.  }
6.  // address_space_ldq 即是 riscv64 下的访存
7.  if(is_test == true) all_cnt++;
```

之后编写测试脚本，控制在运行测试集时开始统计访存次数，运行完毕后停止统计访存次数，这一部分的原理详见附录 A 的问题七。

（二）有效访存次数

有效访存次数是对得到的物理地址进行内存访问的次数。相比总访存次数而言，有效访存次数一般要小得多，这是因为开启内核虚拟化后，虚拟地址需要经过两个阶段的页表翻译才能得到对应的物理地址，也就是一次有效访存对应着若干次的总访存。

我们需要修改 QEMU 仿真器，使其记录有效访存次数，这一部分在 `qemu/accel/tcg/cputlb.c` 中。`get_page_addr_code`，`load_helper` 和 `store_helper` 分别对应着三种类型的访存，将这三类涉及访存的函数的执行次数进行汇总即可得到有效访存次数。

（三）单次操作访存次数

首先，我们需要在测试集程序中设定操作(如 Add，Triad 等)的次数，这在 MBW 和 STREAM 的说明文档中有介绍，如 STREAM 可以在编译时添加编译参数`-DSTREAM_ARRAY_SIZE=10000000` 来设置操作次数，MBW 则是在运行时添加命令行参数来设置。

之后，将统计的总访存次数除以预先设定的操作次数即可。

（四）平均访存开销

结合总访存次数和有效访存次数的统计，我们可以测量平均访存开销：将统计的总访存次数除以有效访存次数即可。

附录 D

附录 D 为我们根据操作系统课程设计的 Test OS 实现重点。

一、代码架构

Test OS 的完整代码架构如下：

```
.
├── arch
│   └── riscv
│       ├── include
│       │   ├── defs.h
│       │   ├── mm.h
│       │   ├── proc.h
│       │   └── sbi.h
│       ├── kernel
│       │   ├── clock.c
│       │   ├── entry.S
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── mm.c
│       │   ├── proc.c
│       │   ├── sbi.c
│       │   ├── trap.c
│       │   ├── vm.c
│       │   └── vmlinux.lds.S
│       └── Makefile
├── include
│   ├── printk.h
│   ├── rand.h
│   ├── stddef.h
│   ├── string.h
│   └── types.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   ├── printk.c
│   ├── rand.c
│   └── string.c
├── Makefile
└── rootfs.ext4
```

其中 mm.h\mm.c 提供了简单的物理内存管理接口，rand.h\rand.c 提供了 rand() 接口用以提供伪随机数序列，string.h/string.c 提供了 memset 接口用以初始化一段内存空间。defs.h 中包含了一些自定义的宏，其中 PAGE_SIZE 表示页大小，这里设置为 16KB。proc.h\proc.c 则实现了两种不同的调度算法。

二、线程相关数据结构与线程切换函数的定义

```
/* 用于记录 `线程` 的 `内核栈与用户栈指针` */
struct thread_info {
    uint64 kernel_sp;
    uint64 user_sp;
};

/* 线程状态段数据结构 */
struct thread_struct {
    uint64 ra;
    uint64 sp;
    uint64 s[12];
};

/* 线程数据结构 */
struct task_struct {
    struct thread_info* thread_info;
    uint64 state;    // 线程状态
    uint64 counter;  // 运行剩余时间
    uint64 priority; // 运行优先级 1 最低 10 最高
    uint64 pid;      // 线程 id

    struct thread_struct thread;
};

/* 线程初始化 创建 NR_TASKS 个线程 */
void task_init();

/* 在时钟中断处理中被调用 用于判断是否需要进行调度 */
void do_timer();

/* 调度程序 选择出下一个运行的线程 */
void schedule();

/* 线程切换入口函数*/
void switch_to(struct task_struct* next);
```

三、线程调度功能实现

1．线程初始化

在初始化线程的时候，我们参考 Linux v0.11 中的实现为每个线程分配一个 16KB 的物理页，我们将 task_struct 存放在该页的低地址部分，将线程的栈指针 sp 指向该页的高地址。具体内存布局如下图所示：

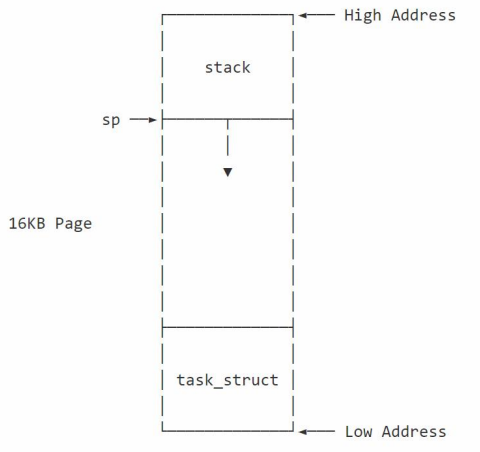


图 D.1 线程物理页内存布局

当我们的 Test OS 运行起来时，其本身就是一个线程 （idle 线程），但是我们并没有为它设计好 task_struct。所以第一步我们要为 idle 设置 task_struct。并将 current, task[0] 都指向 idle。

为了方便起见，我们将 task[1] ~ task[NR_TASKS - 1]，全部初始化， 这里和 idle 设置的区别在于要为这些线程设置 thread_struct 中的 ra 和 sp。

```
1.  void task_init() {
2.      //初始化 idle 线程
3.      task[0]=(struct task_struct *)kalloc();
4.      task[0]->thread_info=0;
5.      task[0]->state=TASK_RUNNING;
6.      task[0]->counter=0;
7.      task[0]->priority=0;
8.      task[0]->pid=0;
9.
10.     idle=task[0];
11.     current=task[0];
12.
13.     // 1. 为 task[1] ~ task[NR_TASKS - 1] 进行初始化
14.     // 2. 其中每个线程的 state 为 TASK_RUNNING, counter 为 0, priority 使用 rand() 来设置, pid 为该线程在线程数组中的下标。
15.     // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 `thread_struct` 中的 `ra` 和 `sp`,
16.     for(int i=1;i<NR_TASKS;i++)
```

```
    {
        task[i]=(struct task_struct *)kalloc();
        task[i]->thread_info=0;
        task[i]->state=TASK_RUNNING;
        task[i]->counter=0;
        task[i]->priority=rand();
        task[i]->pid=i;

        task[i]->thread.ra=(uint64)__dummy;
        task[i]->thread.sp=(uint64)task[i]+PGSIZE;
    }

    printk("...process init done!\n");
}
```

2．实现线程切换

判断下一个执行的线程（next）与当前的线程（current）是否为同一个线程，如果是同一个线程，则无需做任何处理，否则调用 __switch_to 进行线程切换。

```
extern void __switch_to(struct task_struct* prev, struct task_struct* next);
void switch_to(struct task_struct* next) {
    struct task_struct* prev=current;
    current=next;
    if(next->pid == prev->pid)
        return;
    else{
        __switch_to(prev,next);
    }
    return;
}
```

在 entry.S 中实现线程上下文切换 __switch_to：接受两个 task_struct 指针作为参数，保存当前线程的 ra, sp, s0~s11 到当前线程的 thread_struct 中并将下一个线程的 thread_struct 中的相关数据载入到 ra, sp, s0~s11 中。

```
__switch_to:        #a0 current a1 next
    sd ra,40(a0)
    sd sp,48(a0)
    sd s0,56(a0)
    sd s1,64(a0)
    sd s2,72(a0)
    sd s3,80(a0)
    sd s4,88(a0)
    sd s5,96(a0)
```

```
    sd s6,104(a0)
    sd s7,112(a0)
    sd s8,120(a0)
    sd s9,128(a0)
    sd s10,136(a0)
    sd s11,144(a0)

    ld ra,40(a1)
    ld sp,48(a1)
    ld s0,56(a1)
    ld s1,64(a1)
    ld s2,72(a1)
    ld s3,80(a1)
    ld s4,88(a1)
    ld s5,96(a1)
    ld s6,104(a1)
    ld s7,112(a1)
    ld s8,120(a1)
    ld s9,128(a1)
    ld s10,136(a1)
    ld s11,144(a1)

    ret
```

3．实现调度入口函数

实现 do_timer(), 并在时钟中断处理函数中调用。

```
void do_timer(void) {
    /* 1. 如果当前线程是 idle 线程 直接进行调度 */
    /* 2. 如果当前线程不是 idle 对当前线程的运行剩余时间减 1
          若剩余时间任然大于 0 则直接返回 否则进行调度 */
    if(current->pid==idle->pid){
        printk("16K test OS is running!\n");
        schedule();
    }
    else{
        current->counter--;
        if(current->counter > 0)
            return;
        else
            schedule();
    }
}
```

4．实现线程调度

我们实现了短作业优先调度算法（SJF），调度规则如下：

遍历线程指针数组 task(不包括 idle)，在所有运行状态 (TASK_RUNNING) 下的线程运行剩余时间最少的线程作为下一个执行的线程。如果所有运行状态下的线程运行剩余时间都为 0，则对 task[1] ~ task[NR_TASKS-1] 的运行剩余时间重新赋值，之后再重新进行调度。

```
1.  void schedule(void) {
2.      uint64 min_rem=0xffffffffffffffff;
3.      int flag=0;
4.      while(flag==0)
5.      {
6.          for(int i=NR_TASKS-1;i>=1;i--)
7.          {
8.              if(task[i]->counter < min_rem && task[i]->counter)
9.              {
10.                 min_rem=task[i]->counter;
11.                 flag=i;
12.             }
13.         }
14.         if(flag==0)
15.         {
16.             for(int i=1;i<NR_TASKS;i++)
17.             {
18.                 task[i]->counter=rand();
19.                  printk("SET [ PID = %lu COUNTER = %lu]\n",task[i]->pid,task[i]->counte
    r);
20.             }
21.         }
22.     }
23.     struct task_struct* next;
24.     next=task[flag];
25.
26.     switch_to(next);
27. }
```

四、开启虚拟内存映射

在 RISC-V 中开启虚拟地址被分为了两步：setup_vm 以及 setup_vm_final，下面将介绍相关的具体实现，此处我们使用了 16KB 粒度的三级页面来进行虚拟地址的映射。

1．setup_vm 的实现

将 0x80000000 开始的 1GB 区域进行两次映射，其中一次是等值映射（PA == VA），另一次是将其映射至高地址（PA + PV2VA_OFFSET == VA）。如下图所示：

图 D.2 物理地址的两次映射

```
unsigned long early_pgtbl[2048] __attribute__((__aligned__(PGSIZE)));
unsigned long pud[2048] __attribute__((__aligned__(PGSIZE)));
unsigned long pmd[2048] __attribute__((__aligned__(PGSIZE)));
void setup_vm(void)
{
    /**
     *   直接映射:   0x80000000     0xffffffe000000000 各自 1GB
     * 57        47 46         36 35         25 24         14  13           0
     * 00000000000 _ 00000000000 _ 00001000000 _ 00000000000 _ 00000000000000
     *                 (0)           (64)          (0)           (0)
     * 11111111111 _ 11111111110 _ 00000000000 _ 00000000000 _ 00000000000000
     *    (2047)        (2046)        (0)           (0)
     */

    early_pgtbl[2047] = ((unsigned long)pud >> PGSHIFT << 10) | 0x1;
    pud[2046] = ((unsigned long)pmd >> PGSHIFT << 10) | 0x1;
    for(int i=0;i<32;i++){
        pmd[i] = (   (i*2048 + (0x80000000 >> PGSHIFT))    << 10) | 0xf;
    }
}
```

完成上述映射之后，通过 relocate 函数，完成对 satp 的设置，以及跳转到对应的虚拟地址。至此我们已经完成了虚拟地址的开启，之后我们运行的代码也都将在虚拟地址上运行。

```
relocate:
    li t0,0xffffffe000000000
    li t1,0x0000000080000000
    sub t0,t0,t1                #虚拟地址与物理地址的差值

    add sp,sp,t0                #sp 重定位
    add ra,ra,t0                #ra 重定位

    #set stvec to change pc, while
    la t1,change_pc
    add t1,t1,t0
    csrw stvec,t1

    la t1,early_pgtbl           #将一级映射页表装入 satp
    srli t1,t1,14
    addi t2,zero,9
    slli t2,t2,60
    or t1,t1,t2

    sfence.vma zero,zero
    csrw satp,t1
```

2．setup_vm_final 的实现

由于 setup_vm_final 中需要申请页面的接口，应该在其之前完成内存管理初始化，mm.c 中初始化的函数接收的起始结束地址需要设置为虚拟地址。

对所有物理内存 (128M) 进行映射，并设置正确的权限。

```
/* swapper_pg_dir: kernel pagetable 根目录， 在 setup_vm_final 进行映射。*/
unsigned long swapper_pg_dir[512] __attribute__((__aligned__(PGSIZE)));
void setup_vm_final(void)
{
    memset(swapper_pg_dir, 0x0, PGSIZE); //清除 swapper_pg_dir
    // No OpenSBI mapping required
    // mapping kernel text X|-|R|V
    create_mapping(swapper_pg_dir, (uint64)_stext, ((uint64)_stext - PA2VA_OFFSET), ((uint64)_etext - (uint64)_stext), 11);
    // mapping kernel rodata -|-|R|V
    create_mapping(swapper_pg_dir, (uint64)_srodata, ((uint64)_srodata - PA2VA_OFFSET), ((uint64)_erodata - (uint64)_srodata), 3);
    // mapping other memory -|W|R|V
    create_mapping(swapper_pg_dir, PGROUNDUP((uint64)_erodata), (PGROUNDUP((uint64)_erodata) - PA2VA_OFFSET), (PHY_END + PA2VA_OFFSET - PGROUNDUP((uint64)_erodata)), 7);

```

```
    // set satp with swapper_pg_dir
    uint64 swp_tlb_pa = (uint64)swapper_pg_dir - PA2VA_OFFSET; //将虚拟地址转换为物理地址
    uint64 set_satp = (swp_tlb_pa >> 12) | ((uint64)8 << 60);  //设置 satp
    csr_write(satp, set_satp);

    asm volatile("sfence.vma zero, zero");
    return;
}
```

```
void create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, int perm)
{
    /*
    pgtbl 为根页表的基地址
    va, pa 为需要映射的虚拟地址、物理地址
    sz 为映射的大小
    perm 为映射的读写权限，可设置不同 section 所在页的属性，完成对不同 section 的保护
    创建多级页表的时候可以使用 kalloc() 来获取一页作为页表目录
    可以使用 V bit 来判断页表项是否存在
    */
    uint64 perm64 = (uint64)perm; //先做类型转换
    uint64 sva = PGROUNDDOWN(va);       //向下取整（事实上起始地址都是 16KB 对齐的）
    uint64 eva = PGROUNDUP(va + sz);    //向上取整
    uint64 spa = PGROUNDDOWN(pa);       //同理
    uint64 epa = PGROUNDUP(pa + sz);

    uint64 tmpa;
    uint64 cva = sva;
    uint64 cpa = spa;

    uint64 VPN0, VPN1, VPN2; //对应虚拟地址中的虚拟页号
    uint64 PPN0, PPN1, PPN2; //对应物理地址中的物理页号

    uint64 *p;

    static int cnt = 0;

    while (cpa < epa)
    {                                                   //未映射完，继续映射
        VPN0 = cva << 17 >> 17 >> (11 + 11 + PGSHIFT);          //提取 11 位一级页表索引
        VPN1 = cva << (17 + 11) >> (17 + 11) >> (11 + PGSHIFT); //提取 11 位二级页表索引
        VPN2 = cva << (17 + 11 + 11) >> (17 + 11 + 11) >> PGSHIFT;//提取 11 位三级页表索引

```

```
34.         if (!(pgtbl[VPN0] << 63 >> 63))
35.         {                                                   //若当前页无效
36.             tmpa = kalloc();                                //申请一块 16KB 的内存
37.             cnt++;                                          //用于统计申请的总页数
38.             pgtbl[VPN0] = (tmpa - PA2VA_OFFSET) >> PGSHIFT << 10; //写入其物理页号
39.             pgtbl[VPN0] |= 1;                                       // Valid 位置 1
40.         }
41.         p = (uint64 *)((pgtbl[VPN0] >> 10 << PGSHIFT) + PA2VA_OFFSET); //继续搜索三级页
    表
42.
43.         if (!(p[VPN1] << 63 >> 63))
44.         {                       //若无效，申请一块新的页
45.             tmpa = kalloc(); //申请页
46.             cnt++;
47.             p[VPN1] = (tmpa - PA2VA_OFFSET) >> PGSHIFT << 10;
48.             p[VPN1] |= 1; // Valid 置位
49.         }
50.
51.         p = (uint64 *)((p[VPN1] >> 10 << PGSHIFT) + PA2VA_OFFSET);
52.
53.         p[VPN2] = cpa >> PGSHIFT << 10; //三级页出写入最终物理地址的物理页号
54.         p[VPN2] |= perm64;              //改写权限
55.
56.         cpa += PGSIZE; //继续映射下一块内存
57.         cva += PGSIZE;
58.     }
59. }
```