实验报告：Lab3 —— 内存管理子系统实现

一、实验目的

1. 通过阅读 xv6 和实验框架代码，理解 **RISC-V Sv39 虚拟内存机制**。
2. 在此基础上，**独立实现物理内存分配器**，支持内核区 / 用户区的页级分配与释放。
3. 实现 **多级页表管理系统**，包括：
   - 从虚拟地址逐级查找页表项；
   - 建立 / 解除虚实映射；
   - 打印页表以便调试；
   - 释放整棵页表树。
4. 学会在 RISC-V 上 **启用分页并保持内核正常运行**，理解 satp、sfence.vma 等关键指令的作用。
5. 在基本功能之上，加入：
   - 多页分配接口 `allocated_pages(int n)`；
   - double free 防护；
   - 当前空闲页统计接口；
   - `destroy_pagetable` 等资源回收功能。

完成情况

- **物理内存管理**：`kernel/mem/pmem.c` 将可用物理内存划分为内核/用户两个 `alloc_region_t`，实现 `pmem_alloc/pmem_free`、`allocated_pages()`、`pmem_free_pages_count()`，并在 `pmem_free()` 中加入 double-free 检测以保护链表。
- **虚拟内存与页表**：`kernel/mem/vmem.c` 完成 `vm_getpte/vm_mappages/vm_unmappages/vm_destroy_pagetable/vm_print`，支持 Sv39 多级页表遍历、映射与整棵释放；`kernel/mem/kvm.c` 中的 `kvm_init/kvm_inithart` 已能构造并启用内核页表。
- **自测与调试**：`kernel/boot/main.c` 编写 `test_pmem_*`、`test_vmem_*` 等用例，验证单页/多页分配、页表映射/回收及异常（对齐、越界、double free），对应截图保存在 `picture/test*.png`。

二、实验环境与代码框架

1. 实验环境

- 开发平台：Ubuntu 24.04（x86_64）
- 交叉编译工具链：`riscv64-linux-gnu-gcc`
- 模拟器：`qemu-system-riscv64 -machine virt -nographic`
- 调试工具：`gdb-multiarch`（可选）

2. 代码框架（与本实验相关部分）

whu-oslab-lab3 ├── include │ ├── uart.h // (串口头文件) │ ├── lib │ │ ├── print.h // (打印库头文件) │ │ └── lock.h // (锁/自旋锁头文件) │ ├── proc │ │ ├── cpu.h // (CPU/Hart 相关) │ │ └── proc.h // (进程/线程相关) │ ├── mem │ │ ├── pmem.h // (新增: 物理内存管理接口) │ │ └── vmem.h // (新增: 虚拟内存/页表管理接口) │ ├── common.h // (基本类型定义) │ ├── memlayout.h // (物理内存布局) │ └── riscv.h // (RISC-V 寄存器和页宏) ├── kernel │ ├── boot │ │ ├── main.c // (内核入口，包含 pmem/vmem 测试) │ │ ├── start.c // (启动代码 C 部分) │ │ ├── entry.S // (启动代码 汇编部分) │ │

└── Makefile │ ├── dev │ │ ├── uart.c // (串口驱动实现) │ │ └── Makefile │ ├── lib │ │ ├── print.c // (打印库实现) │ │ ├── spinlock.c // (自旋锁实现) │ │ └── Makefile │ ├── proc │ │ ├── pro.c // (进程/线程实现) │ │ └── Makefile │ ├── mem │ │ ├── pmem.c // (新增: 物理内存分配器实现) │ │ ├── vmem.c // (新增: 页表管理与虚拟内存初始化) │ │ └── Makefile // (mem 目录下的 Makefile) │ ├── Makefile │ └── kernel.ld // (链接脚本) ├── picture │ └── *.png // (图片资源) ├── Makefile // (顶层 Makefile) ├── common.mk // (通用配置) ├── README.md // (项目说明) └── Report.md // (实验报告)

三、系统设计部分

3.1 架构设计说明

整个内存管理子系统分为三层：

1. **物理内存层（pmem）**
   - 负责"砖头"的管理：哪些物理页可用、如何分配/回收。
   - 将可用物理内存划分为**内核区域**和**用户区域**，分别维护空闲页链表。
   - 对外提供：
     - `pmem_init()`：初始化两个区域的可分配页；
     - `pmem_alloc(in_kernel)` / `pmem_free(page, in_kernel)`；
     - `allocated_pages(int n, bool in_kernel)`：多页分配接口；
     - `pmem_free_pages_count(in_kernel)`：统计当前空闲页数。
2. **虚拟内存 / 页表层（vmem）**
   - 负责"砖头怎么摆"：实现 Sv39 多级页表遍历和映射建立。
   - 提供接口：
     - `vm_getpte(pgtbl, va, alloc)`：多级遍历，必要时分配中间页表；
     - `vm_mappages(pgtbl, va, pa, len, perm)`：一段虚拟地址到物理地址的映射；
     - `vm_unmappages(pgtbl, va, len, freeit)`：解除映射，选项决定是否回收物理页；
     - `vm_destroy_pagetable(pgtbl, free_leaf)`：释放整棵页表树；
     - `vm_print(pgtbl)`：打印当前页表映射用于调试。
3. **内核启动和测试层（boot/main.c）**
   - 在主核中依次调用：
     1. `pmem_init()`：建立物理空闲页链表；
     2. `kvm_init()`：创建并填充内核页表；
     3. `kvm_inithart()`：写 satp & sfence.vma，启用分页；
     4. 构造一张测试用页表 `test_pgtbl`，调用 `vm_mappages` / `vm_unmappages` / `vm_print` 验证功能；
     5. 调用扩展接口测试多页分配、double free 防护等。

三层之间只通过清晰的 C 接口交互，层次关系如下：

> **硬件内存 & 页表结构**
> ↑（riscv.h + memlayout.h 宏）
> **物理内存管理 pmem.c**
> ↑
> **虚拟内存管理 vmem.c**

> ↑
> **启动流程 & 测试 main.c**

3.2 关键数据结构

3.2.1 物理内存区域 alloc_region_t

```
typedef struct page_node {
    struct page_node* next;
} page_node_t;

// 许多物理页构成一个可分配的区域
typedef struct alloc_region {
    uint64 begin;           // 起始物理地址（页对齐）
    uint64 end;             // 终止物理地址（开区间）
    spinlock_t lk;          // 自旋锁，保护链表与计数
    uint32 allocable;       // 当前空闲页数量
    page_node_t list_head; // 单向链表虚假头节点
} alloc_region_t;

static alloc_region_t kern_region, user_region;
```

3.2.2 设计要点：

可分配物理内存是一个连续区间，区间内部每个 4 KiB 页都可以直接强转为 page_node_t*；

不需要额外元数据数组，页本身作为链表节点，类似 xv6 的 struct run 设计；

kern_region 和 user_region 分别保存内核 / 用户页，后续通过地址范围判断属于哪个区域。

3.2.3 页表项与页表类型

vmem.h中定义：

```
typedef uint64 pte_t;    // 页表项
typedef uint64* pgtbl_t; // 页表指针（512 项）

// 从虚拟地址取 VPN[2..0]
#define VA_SHIFT(level)       (12 + 9 * (level))
#define VA_TO_VPN(va, level)  ((((uint64)(va)) >> VA_SHIFT(level)) & 0x1FF)

// PTE 与PA的转换
#define PA_TO_PTE(pa) ((((uint64)(pa)) >> 12) << 10)
#define PTE_TO_PA(pte) (((pte) >> 10) << 12)

// 权限位
#define PTE_V (1 << 0)
#define PTE_R (1 << 1)
#define PTE_W (1 << 2)
```

```
#define PTE_X (1 << 3)
#define PTE_U (1 << 4)

// 判断是否为“中间页表”（R/W/X 全 0）
#define PTE_CHECK(pte) (((pte) & (PTE_R | PTE_W | PTE_X)) == 0)

// VA 上限
#define VA_MAX (1ul << 38)
```

这些宏是整个页表管理逻辑的基础，直接对应 RISC-V Sv39 的 PTE 格式。

3.3 与 xv6 对比分析

3.3.1 物理内存管理

xv6：使用 struct run { struct run *next; }; 单向链表管理空闲页；使用 freerange() 按照地址区间将所有空闲页链接起来；不区分内核/用户区域（所有物理页共用一条链表）。 本实验：在 xv6 设计基础上增加了一层抽象 alloc_region_t，将物理页分为内核区域和用户区域；每个区域都有自己的锁和链表，更便于未来做权限控制、统计以及 per-CPU 优化；采用相同的“页作为节点”的思想，不额外引入 bitmap/buddy 等复杂结构。

3.3.2 页表管理

xv6 的 walk() / mappages() 与本实验的 vm_getpte() / vm_mappages() 本质一致：按层次从 VPN[2] → VPN[1] → VPN[0] 逐级索引；中间级 PTE 的 R/W/X 必须为 0，仅设置 V 位作为“页表指针”；alloc 参数控制是否自动创建中间页表。

3.3.3 本实验额外实现了：

vm_unmappages()：支持按段解除映射并可选择释放物理页； vm_destroy_pagetable()：递归释放整棵页表树； vm_print()：递归打印页表并标注区域，如 UART、PLIC、KERNEL_TEXT、KERNEL_DATA。

3.3.4 内核页表初始化

xv6 的 kvminit()恒等映射 UART、PLIC、内核代码和数据。本实验的 kvm_init() 采用相同思路： [KERNEL_BASE, etext] → 只读 + 可执行（RX）；[etext, PHYSTOP] → 读写（RW）；UART/PLIC 等设备等值映射为 RW。总体上，本实验在保持 xv6 简洁性的同时，对区域划分和调试工具做了扩展。

3.4 设计决策理由

（1）为何采用“链表 + 页即节点”的物理分配器？ 优点：实现简单、内存开销极小、每次分配/释放都是 O(1)，适合教学和小规模系统；在实验中，我们只需要页粒度分配，虚拟内存层可以通过“非连续物理页 + 连续虚拟地址”实现大块内存，因此该设计足够。 （2）为什么将物理页分为内核区 / 用户区？ 内核页表、内核栈等必须使用“内核区域”的物理页，避免被用户错误释放或覆盖；用户地址空间的物理页单独计数，更方便实现统计、限制和 future work（如 cgroup 内存限制）。

（3）为什么实现 vm_getpte 而不是每次手写遍历？ 将“硬件页表行走算法”封装为一个函数，方便在 vm_mappages、vm_unmappages、vm_destroy_pagetable 中统一使用；逻辑更接近 RISC-V 手册对 hardware page walk 的描述，便于理解和调试。 （4）double free 防护为什么放在 pmem 层？ 页表层、进程管理层都有

可能调用 pmem_free；将 double free 检查集中到 pmem 层，可以在发生逻辑错误时尽早 panic，避免默默破坏链表结构。

四、实验过程部分

4.1 实现步骤记录

（1）阅读材料与现有框架——阅读 PPT 中有关 Sv39 页表结构、PTE 格式和实验要求；阅读 xv6 的 kalloc.c、vm.c，理解其物理分配器和页表遍历算法；熟悉本实验框架中的 riscv.h、memlayout.h 和链接脚本 kernel.ld。（2）实现物理内存分配器 pmem.c——定义 page_node_t 与 alloc_region_t；在 pmem_init() 中：使用链接脚本符号 ALLOC_BEGIN 作为可分配内存起点，向上按页对齐；将 [alloc_begin, alloc_begin + KERNEL_PAGES * PGSIZE]作为内核区域；将剩余到 PHYSTOP 的部分作为用户区域；通过循环调用 pmem_free() 将每一页加入各自的空闲链表。实现 pmem_alloc(in_kernel) / pmem_free(page, in_kernel)，并增加统计 allocable。在此基础上增加 allocated_pages(int n)：循环调用 pmem_alloc 获取 n 个物理页，返回数组或首地址。 （3）增加 double free 防护与统计接口 在 pmem_free() 中，在持锁状态下遍历当前区域空闲链表，若发现待释放页已经存在则 panic；提供 pmem_free_pages_count(bool in_kernel)，返回当前区域 allocable 的值，便于打印和调试。（4）实现虚拟内存管理 vmem.c—— 编写 vm_getpte(pgtbl, va, alloc)：首先检查 va < VA_MAX；从 level=2 逐级查找，若中间 PTE 无效且 alloc==true 则分配新的页表页；中间页表 PTE 仅设置 V 位，R/W/X 为 0；最终返回最低级 PTE 的指针。实现 vm_mappages()：检查 va 和 len 是否按页对齐；以页为单位循环，调用 vm_getpte 获取叶子 PTE；若 PTE 已经有效则 panic 防止重映射； 填入 PPN + 权限 + V。实现 vm_unmappages()：同样按页对齐检查；使用 vm_getpte(..., alloc=false) 获取叶子 PTE；如果存在映射且 freeit==true，根据物理地址所在区间选择 pmem_free(pa, true/false)；最后将 PTE 清零。 （5）内核页表初始化与启用分页—— 在 kvm_init() 中：为根页表 kernel_pgtbl 分配一个内核物理页，清零；使用 vm_mappages 分别映射 UART、PLIC、内核 text 段与 data 段 / 剩余物理内存；在 kvm_inithart() 中：调用 w_satp(MAKE_SATP(kernel_pgtbl)) 设置 satp；使用 sfence_vma() 刷新 TLB。（6）页表打印与销毁接口—— 参考 vm_print_recursive_new 的实现，完成 vm_print()：递归遍历所有有效 PTE；对叶子 PTE 计算 VA/PA，打印权限和区域名称。实现 vm_destroy_pagetable(pgtbl, free_leaf)：递归遍历多级页表；对中间页表释放其子页表页；free_leaf==true 时释放叶子对应的物理页；最后释放根页表本身。 （7）在 main.c 中编写测试代码 调用 pmem_alloc、pmem_free 和 allocated_pages 测试物理分配器；构造测试页表，建立多种映射组合（只读、可写、可执行），打印页表；调用 vm_unmappages、vm_destroy_pagetable 测试解除映射与资源回收。

4.2 问题与解决方案

（1）pmem_init 死循环 / 崩溃问题 起因：初版使用 for (uint64 p = end - PGSIZE; p >= begin; p -= PGSIZE) 逆序初始化空闲链表，由于 uint64 无符号下溢，导致 p 从 begin 再减 4K 后变成极大值，引起死循环和非法访问。 解决：改为升序循环 for (p = begin; p < end; p += PGSIZE)，确保循环结束条件正确。 （2）vm_unmappages 释放物理页归属判断 问题：如何知道某个物理页属于内核区域还是用户区域？ 方案：复用 pmem_init 时的区域划分，使用 kbegin = PG_ROUND_UP((uint64)ALLOC_BEGIN); ksplit = kbegin + KERNEL_PAGES * PGSIZE; [kbegin, ksplit] → 内核区域； [ksplit, PHYSTOP] → 用户区域； 若不在这两个区间中则 panic。（3）double free 检测导致的锁问题 问题：double free 检测需要遍历链表，必须在持锁状态下进行，否则存在竞态条件； 解决：将检查过程全部放在 spinlock_acquire / spinlock_release 之间，并在检测到 double free 时先释放锁再 panic，避免死锁。 （4）页表销毁中的递归边界 问题：递归释放页表时，必须区分“中间页表 PTE”和“叶子 PTE”，否则会错误地把叶子页当作子页表去递归； 解决：使用与 vm_getpte 相同的 PTE_CHECK 宏，只在 PTE_CHECK(pte) && level > 0 时把它当作子页表递归，否则视为叶子进行处理。

五、测试验证部分

5.1 功能测试以及结果截图

（1）物理分配器基本功能——在 main.c 中：连续调用 pmem_alloc(true/false) 获取多个页。检查：返回地址按 4 KiB 对齐；不同调用返回的页地址互不相同；将数据写入某一页，再次读出值一致。 释放后重新分配：回收已释放的页，pmem_free_pages_count 统计值恢复。

（2）多页分配接口测试——调用 allocated_pages(5, false) 分配 5 个用户物理页，记录返回的数组；对这 5 个页逐一写入不同的 magic 值，确保互不覆盖；释放这 5 个页后，再次调用 allocated_pages(5, false)，观察是否能重新获得这些或其他页。(*** 测试结果详见test1&&test2 ***)

（3）页表映射与打印——构造 test_pgtbl，对以下 VA 建立映射：0x0 → mem[0] 只读；0xA*PGSIZE → mem[1] 读写；0x200*PGSIZE → mem[2] 可执行；使用 vm_print(test_pgtbl) 检查：VA/PA 映射是否正确；权限位 r/w/x/u 是否与期望一致；Region 字段是否正确标注 KERNEL_TEXT / KERNEL_DATA / UNKNOWN 等。

```
test-1


=== KERNEL PAGE TABLE MAPPINGS ===
Root Page Table: 80054000

  Virtual Address    ->  Physical Address   | Perm | Region
  ------------------------------------------------------------
  VA: 0x0000000000000000 -> PA: 0x000000008040e000 | r--- | UNKNOWN
  VA: 0x000000000000a000 -> PA: 0x000000008040f000 | rw-- | UNKNOWN
  VA: 0x0000000000200000 -> PA: 0x0000000080410000 | r-x- | UNKNOWN
  VA: 0x0000000040000000 -> PA: 0x0000000080410000 | r-x- | UNKNOWN
  VA: 0x0000003fffffff000 -> PA: 0x0000000080412000 | -w-- | UNKNOWN
  ------------------------------------------------------------
  Legend: r=read, w=write, x=execute, u=user
=== END PAGE TABLE ===
```

```
test-2


=== KERNEL PAGE TABLE MAPPINGS ===
Root Page Table: 80054000

  Virtual Address    ->  Physical Address   | Perm | Region
  ------------------------------------------------------------
  VA: 0x0000000000000000 -> PA: 0x000000008040e000 | r--- | UNKNOWN
  VA: 0x0000000040000000 -> PA: 0x0000000080410000 | r-x- | UNKNOWN
  VA: 0x0000003fffffff000 -> PA: 0x0000000080412000 | -w-- | UNKNOWN
  ------------------------------------------------------------
  Legend: r=read, w=write, x=execute, u=user
=== END PAGE TABLE ===
```

（4）内核页表与分页启用——在 kvm_init() 前后打印关键地址值（如 KERNEL_BASE、etext）；调用 kvm_inithart() 后继续执行 printf 验证：内核代码仍正常执行；访问静态全局变量未崩溃；UART 输出仍然正常。

（5）页表销毁——为测试页表调用 vm_destroy_pagetable(test_pgtbl, true)；使用 pmem_free_pages_count 观察空闲页数是否增加到销毁前的水平；再次访问已经销毁的页表时系统会 panic，表明资源已经被释放。
alt text

5.2 异常测试以及结果截图

（1）地址未对齐——调用 vm_mappages(pgtbl, va=0x1234, ...)，期望触发 panic("mappages: va not aligned")；调用 vm_unmappages 时传入非页对齐长度，同样触发 panic。

（2）越界虚拟地址——调用 vm_getpte(pgtbl, va >= VA_MAX, true)，触发 panic("vm_getpte: virtual address out of bound")。

（3）double free——对同一个物理页连续两次 pmem_free(page, in_kernel)，触发 panic("pmem_free: double free detected")，验证防护逻辑生效。

（4）物理地址不在任何区域——人为构造错误的 PTE，使其 PPN 对应的 PA 不落在内核/用户管理区域之内，再调用 vm_unmappages，触发 panic("vm_unmappages: pa out of any known region")。

```
--- test-3: Batch Allocation/Free ---
User free pages before batch alloc: 31725
Allocated 7 pages out of 7 requested.
User free pages after batch alloc: 31718 (Expected: 31718)
Releasing 7 allocated pages...
User free pages after batch free: 31725 (Expected: 31725)
Test 3 PASSED: Free page count restored correctly.


--- test-4: Alignment Check ---
Attempting vm_mappages with unaligned VA (Should PANIC if implemented correctly)...
panic: mappages: va not aligned
```

这些异常测试表明：在误用 API 时，系统能以“可诊断的方式”失败，而不是静默破坏内存。

六、思考与扩展分析

1.性能优化 使用位图为每个物理页维护“已分配/空闲”状态，将 double free 检查降为 O(1)；

2.本实验的物理内存分配器与 xv6 有何不同？

共同点：都使用“页本身作为链表节点”的简单 free-list 方案，空间开销极小。 不同点：本实验显式区分内核 / 用户区域，并为两者维护独立的 alloc_region_t；增加了统计信息和 double free 检测接口，方便调试和后续扩展。

3.当前实现的主要性能瓶颈在哪里？ pmem 层 double free 检测需要遍历空闲链表，复杂度 O(空闲页数)；所有分配/释放操作都需要持有区域自旋锁，多核下可能产生锁竞争；页表操作逐页映射，在处理大区间时开销较大。

4.可能的优化方向（未来工作） 为内核物理页引入 per-CPU 小缓存，减少频繁访问全局 freelist 时的锁竞争；对内核大块内存采用 2MB/1GB 大页映射，减少页表项数量和 TLB miss；在 vm_destroy_pagetable 中维护“非空子树计数”，对完全空子树剪枝，减少遍历量。

七、实验总结

通过本次内存管理实验，我完成了从“读 xv6 源码”到“在自己内核中动手实现一整套内存管理子系统”的过程，主要收获如下： 理解了 Sv39 的地址分解方式和多级页表结构，可以从任意一个虚拟地址手工推断出硬件是如何逐级查表最终得到物理地址的。掌握了页粒度物理分配器的设计方法：利用“页本身作为链表节点”节约元数据；通过区域抽象区分内核/用户页；在分配/释放接口上加入 double free 防护和统计信息。通过 vm_getpte、vm_mappages、vm_unmappages 和 vm_destroy_pagetable 的实现，真正理解了页表不只是“查表”，更是一棵需要被创建、维护和销毁的树结构。在调试过程中，体会到 panic + 自解释错误信息 的重要性：很多 bug（例如 unsigned 下溢、错误区域释放）如果没有断言，很难排查。虽然只做了有限的性能分析，但已经能理性地讨论当前设计的时间复杂度和未来优化方向，为后续可能实现更复杂的分配算法（bitmap、buddy、SLAB）打下基础。 总体来说，本实验让“虚拟内存”从课本里的概念变成了手边能跑、能调试的代码，对操作系统内核如何管理内存有了更直观和系统的认识。

八、实验思考题解答

1. 设计对比与权衡物理内存分配器设计对比与选择理由 物理内存分配器与 xv6 的分配器在核心机制上是一致的，都采用了**页本身作为链表节点的 Free-List 方案**这种选择是为了追求实现简单和极小的内存开销，因为每次分配和释放都只需对链表头进行 $O(1)$ 操作，非常适合教学和小规模内核。然而，这种设计也存在权衡，即它缺乏对大块连续内存的有效管理，更容易产生外部碎片。 本实验的设计在 xv6 基础上进行了抽象和扩展：区域隔离： 我显式区分了内核区域和用户区域，并为它们各自维护了独立的 alloc_region_t 结构和自旋锁。选择这样做是为了实现更好的权限控制和隔离，防止用户程序意外破坏或释放内核关键页（如页表页）。代价是增加了初始化和判断逻辑的复杂性，且两个区域的空闲页不能相互借用。此外，我还增加了 double free 检测接口和空闲页统计功能。double free 检测极大地提高了系统的健壮性，能够在逻辑错误发生时及时 panic。但这种检测需要遍历空闲链表，在空闲页很多时，其 $O(\text{空闲页数})$ 的复杂度会成为性能上的瓶颈。
2. 内存安全 （1）防止分配器被恶意利用 为了防止内存分配器被恶意程序利用，需要从多个层面构建防护：Double Free 防护：最直接的方式是在 pmem_free() 中检查待释放的物理页是否已存在于空闲链表中。一旦发现重复释放，应立即触发 panic。这能阻止攻击者通过重复释放来破坏 Free-List 结构，从而构造恶意指针。区域和边界检查：严格隔离内核页和用户页，确保用户地址空间的逻辑错误不会影响到内核的物理内存。同时，在释放操作中，必须检查物理地址是否落在分配器管理的合法区域内，防止释放任意地址。地址对齐：分配和释放的地址都必须严格检查是否按页对齐，以确保指针的合法性。 （3）页表权限设置的安全考虑 页表权限的设置是虚拟内存安全的核心：用户模式隔离（PTE_U）：内核必须确保清除内核代码、数据和设备映射页的 PTE_U 位。这使得用户进程在用户模式下无法通过任何虚拟地址访问到内核空间。权限最小化：遵循最小权限原则。代码段应该只设置 RX（Read + Execute），清除 Write，以防止运行时代码被篡改（代码注入）。数据段应该设置 RW（Read + Write），清除 Execute，以防止将数据当作代码执行（防止缓冲区溢出后的代码执行攻击）。页表页的保护：所有作为中间节点的页表页，其 PTE 必须清除 R、W、X 权限位，仅设置 V 位。这保证了中间页表只能被 RISC-V 硬件的页表行走机制使用，防止它们被误当作普通的数据页读写或执行。
3. 性能分析 （1）当前实现的性能瓶颈： 当前分配器的主要性能瓶颈集中在以下几个方面：Double Free 检测的线性开销：如前所述，为了健壮性，pmem_free() 必须遍历空闲链表。在空闲页数量很大时，这会导致释放操作的时间复杂度高达 $O(\text{空闲页数})$，显著拖慢系统性能。全局锁竞争：尽管您将物理内存分为了内核区和用户区并设置了独立的锁，但在多核 CPU 下，任何对任一区域的频繁分配/释放操作仍会导致锁竞争，从而串行化操作，降低并行性。逐页映射开销：页表操作（如 vm_mappages）是逐页进行的。如果需要映射大块连续的内存区域，将导致多次页表遍历和 PTE 写入，并可能引起大量 TLB（转换后备缓冲器）失效。 （2）测量和优化内存访问性能 测量方法：可以利用 RISC-V 架构提供的性能计数器，如 cycle 寄存器。通过在关键函数的入口和出口（如 pmem_alloc、pmem_free 或 vm_mappages）记录 cycle 值，可以统计出操作的精确耗时。同时，统计锁的获取失败次数和等待时间，可以量化锁竞争的激烈程度。 优化方向： 优化 Double Free 检测：将 Free-List 替换为基于**位图（Bitmap）**的分配器，或者维护一个独立的已分配页集合，可以将页状态查询和 double free 检查的复杂度降低到 $O(1)$。优化锁竞争：可以为内核物理页引入 per-CPU 小缓存，允许每个 CPU 在不需要竞争全局锁的情况下，快速分配和回收少量页，只有缓存用尽时才去访问全局 Freelist。优化页表操作：引入 大页（Huge Page）映射机制。对大块连续的内存区域（如内核代码、某些用户堆），可以使用 2MB 甚至 1GB 的页来映射，大大减少了页表项的数量，提升了 TLB 命中率，降低了 Page Walk 开销。
4. 扩展性 （1）支持用户进程所需的修改——要支持用户进程，关键在于实现地址空间的隔离和切换： 进程页表创建与继承：为每个新进程分配独立的根页表，并将其映射到进程控制块（PCB）。新页表必须继承内核的映射（如设备和内核代码），同时将用户程序代码、数据、堆、栈等映射到用户地址空间，并设置 PTE_U 权限位。 地址空间切换：在进程切换时，内核需将目标进程页表的物理地址写入 satp 寄存器，完成 CPU 虚拟地址空间的切换。 物理页分配源：为用户进程分配内存时，需确保调用 pmem_alloc(false) 使用用户区域的物理页。 （2）内存共享和写时复制（COW）的实现 内存共享：通过

让多个进程页表中的 PTE 指向同一个物理页实现。权限通常设置为 Read-Only (R) 或 Read/Execute (RX)，以限制写入，确保隔离。 写时复制 (COW)：用于优化 fork() 性能。子进程创建时，父子进程共享内存，但权限均设为 Read-Only。任一进程尝试写入时，触发 Page Fault。内核捕获异常，分配新物理页，拷贝数据，修改当前进程的 PTE 指向新页并设置 Read/Write 权限，从而实现只有在发生写入时才进行复制。

5. 错误恢复 （1）页表创建失败时的资源清理 页表创建必须具备事务性——数据页回滚：如果映射过程中分配了数据物理页但操作失败，需显式调用 pmem_free() 回收这些页。页表树回滚：如果中间页表分配失败，应调用 vm_destroy_pagetable(pgtbl, false)，指示它只释放页表页本身，不对叶子 PTE 指向的数据页进行操作，防止错误的资源回收。 （2）检测和处理内存泄漏 运行时统计检测：在程序流程的关键点（如进程创建/销毁前后），检查 pmem_free_pages_count() 的值，如果空闲页数持续或异常减少，则表明存在内存泄漏。 页引用计数：为每个物理页添加引用计数器。每当一个 PTE 指向该页时计数加一，解除映射时减一。只有当计数归零时，才真正释放物理页，从而防止过早释放和检测逻辑错误。 分配追踪：记录分配操作的调用栈信息（文件/行号），用于在系统运行结束后，定位所有未被释放资源的分配来源。