# 武汉大学计算机学院
# 本科生课程设计报告

# 实验三：页表与内存管理


专 业 名 称　：计算机科学与技术

课 程 名 称　：操作系统实践 A

指 导 教 师　：李祖超　副教授

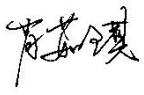
学 生 学 号　：2023302111416

学 生 姓 名　：肖茹琪


二〇二五年十月

# 郑 重 声 明

本人呈交的设计报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本设计报告不包含他人享有著作权的内容。对本设计报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本设计报告的知识产权归属于培养单位。

本人签名：  　　日期：2025.10.13

# 摘　要

本次实验旨在深入理解现代操作系统中虚拟内存管理的核心机制。通过分析xv6-riscv的内存管理源码，并基于RISC-V Sv39分页规范，独立设计并实现了一个简化的内核级内存管理系统。实验内容覆盖了从底层硬件模式切换、物理内存管理到虚拟内存映射的全过程。首先，通过编写汇编和C代码，实现了从机器模式（M-mode）到监督模式（S-mode）的正确切换，并配置了物理内存保护（PMP）为内核运行提供安全环境。接着，实现了一个基于空闲链表的物理页分配器，用于管理内核结束地址到物理内存顶端之间的所有可用内存。最后，基于Sv39三级页表规范，实现了页表的创建、遍历（walk）、映射（mappages）等核心功能，并为内核构建了完整的地址空间，最终成功启用分页。实验通过分层测试验证了物理内存分配、页表功能和虚拟内存激活的正确性，所有测试均成功通过，标志着内核已在虚拟地址空间上稳定运行，为后续实现多进程、系统调用等高级功能奠定了坚实的内存管理基础。

**关键词：**RISC-V；操作系统；内存管理；Sv39页表；物理页分配器

# 目录

# 1 实验目的和意义

## 1.1 实验目的

本实验旨在深入理解现代操作系统中虚拟内存管理的核心机制。通过分析 xv6-riscv 的内存管理源码，并基于 RISC-V Sv39 分页规范，独立设计并实现一个简化的内核级内存管理系统。具体目标包括：

1.理解物理内存管理：掌握物理内存的布局、发现与组织方式，实现一个基于空闲链表的物理页分配器。

2.掌握 Sv39 页表机制：理解 RISC-V 三级页表的地址转换原理、页表项（PTE）的结构与权限位作用。

3.实现页表管理系统：编码实现页表的创建、虚拟地址到物理地址的映射、以及页表的激活。

4.构建内核地址空间：为内核创建页表，完成内核代码、数据、设备等关键区域的映射，并成功启用分页。

## 1.2 实验意义

内存管理是操作系统的三大核心功能之一，而虚拟内存是现代操作系统的基石。通过亲手实现从物理内存分配到虚拟地址映射的全过程，能够将课本上抽象的理论知识与底层硬件细节紧密结合，深刻理解内核如何在最早期阶段为自身建立起内存管理，并完成从物理地址到虚拟地址的“自举”过程。

这次实践是后续学习进程管理、文件系统和系统调用的基础，是构建一个完整操作系统的关键一步。

# 2 实验准备

在正式编码前，首先完成对 xv6-riscv 源码和 RISC-V 规范的学习。

## 2.1 任务 1：深入理解 Sv39 页表机制

### 2．1．1　39 位虚拟地址的分解：

```
 38        30 29        21 20        12 11         0
 |  VPN[2]  |   VPN[1]   |   VPN[0]   |   offset   |
```

图 1 39 位虚拟地址

- 每个 VPN 段的作用：RISC-V Sv39 模式下的虚拟地址被分为四个部分：VPN[2]、VPN[1]、VPN[0] 各占 9 位，分别作为三、二、一级页表的索引；offset 占 12 位，用于页内寻址。
- 9 位而非其他位数的原因：9 位索引的设计是因为一个 4KB 的页表页刚好可以存放 4096 / 8 = 512 个 PTE，而 2^9 = 512。

### 2.1.2 页表项（PTE）格式：

PTE 是一个 64 位结构，包含 V（有效位）、R/W/X（权限位）、U（用户态访问位）和 PPN（物理页号）等关键字段，共同决定了映射的有效性和访问权限。

- 选择三级页表的原因： 这是空间效率与转换效率的权衡。一级页表太大（需要 GB 级别连续内存），多级页表则可以用时间（多次访存）换空间，只为实际使用的地址分配页表页，大大节省了内存。三级页表在 39 位地址空间下提供了合适的粒度。
- 对"页表也存储在物理内存中"的理解：页表本身不是特殊的硬件结构，它就是存储在 RAM 中的普通数据。CPU 的 MMU 硬件根据 satp 寄存器找到根页表的物理地址，然后像访问普通内存一样逐级读取 PTE，最终计算出目标物理地址。

正如以上代码，代码中统一将有符号数转为无符号数处理，避免符号位干扰，转换完成后再统一添加负号。此外，支持 sign 参数控制是否进行符号处理（%u 不需要）。

## 2.2 任务 2：分析 xv6 的物理内存分配器

### 2.2.1 研读 kalloc.c 的核心数据结构

```
struct run {
  struct run *next;
};
```

图 2 struct run 数据结构

核心数据结构 struct run 的设计巧妙之处在于“就地取材”的核心。它将空闲的物理页本身当作链表节点来使用。当一个物理页被释放时，kfree 会把它强制转换为 struct run*类型，并将其 next 指针指向当前的空闲链表头。这样

不需要为链表节点额外分配任何元数据空间，极大地简化了设计并节约了内存。

#### 2.2.2 kinit()、kalloc()和kfree()的实现：

1. kinit(): 初始化函数。它从链接脚本提供的_end 符号（内核静态部分的结束地址）开始，到 PHYSTOP（物理内存顶端）结束，逐个物理页调用 kfree()，将所有可用的物理内存都加入到空闲链表中。
2. kalloc(): 分配函数。它从空闲链表（kmem.freelist）的头部取下一个节点（一个空闲页），更新链表头，然后返回这个页的地址。这是一个非常高效的 O(1)操作。
3. kfree(): 释放函数。它接收一个物理页地址，将其转换为 struct run*，然后将其插入到空闲链表的头部。这也是一个 O(1)操作。

# 3 实验原理与设计

## 3.1 系统架构设计

本次实验选择构建一个精简但完整的内存管理系统，由以下模块组成：

1. 启动模块 (entry.S, start.c): 负责从 M-mode 切换到 S-mode，并为内核设置初始栈和 PMP，为后续内存操作提供安全的环境。

2. 物理内存分配器 (kalloc.c): 管理从 end 到 PHYSTOP 的物理内存，提供 kalloc()和 kfree()接口。

3. 虚拟内存管理器 (vm.c): 实现 RISC-V Sv39 页表操作，提供 mappages 接口用于建立映射，并提供 kvminit 和 kvminithart 来创建和激活内核页表。

4. 主控模块 (main.c): 协调调用以上模块的初始化函数，并在初始化完成后执行分层测试，验证系统的正确性。

## 3.2 核心数据结构定义

1. 页表项格式 (riscv.h)

```
#define PTE_V (1L << 0)    // 有效位
#define PTE_R (1L << 1)    // 可读
#define PTE_W (1L << 2)    // 可写
#define PTE_X (1L << 3)    // 可执行
#define PTE_U (1L << 4)    // 用户可访问
```

图 3 页表项格式定义

2. 物理内存管理结构（kalloc.c）

```
struct run {
  struct run *next;          // 空闲链表
};

struct {
  struct run *freelist;      // 空闲页链表头
} kmem;
```

图 4 物理内存管理结构定义

## 3.3 与 xv6 设计的异同分析

- 相同点：
  1. 核心算法一致：完全采用了 xv6 的空闲链表物理内存分配算法和三级页表遍历映射算法。
  2. 启动流程相似：遵循 entry.S -> start.c -> main.c 的启动顺序，并完成了 M-mode 到 S-mode 的切换。
  3. 内存布局一致：遵循 QEMU virt 机器的内存布局，将内核加载在 0x80000000。
- 不同点：

表 1 本实验与 xv6 的不同点

| 特性 | xv6 设计 | 本实验设计 | 设计理由 |
| --- | --- | --- | --- |
| 内存初始化 | 复杂范围检测 | 简化固定范围 | 降低实现复杂度 |
| 错误处理 | 完整 panic 机制 | 基础错误检查 | 聚焦核心功能 |
| 多核支持 | 自旋锁保护 | 单核假设 | 教学环境简化 |

# 4 实验步骤与实现

## 4.1 实验步骤记录

### 4.1.1 阶段一：修正启动流程

1. 创建 start.c 文件，实现从 M-mode 到 S-mode 的切换逻辑，并正确配置 PMP。

2. 重写 entry.S，使其使用 start.c 中定义的静态栈，并调用 start()函数。

3. 更新 Makefile 和 riscv.h，添加新文件和必需的 CSR 操作函数。

#### 4.1.2 阶段二：核心功能实现

1. 创建 kalloc.c，实现基于 struct run 的空闲链表。

2. 实现 kinit()，将 end 到 PHYSTOP 之间的所有物理内存页加入空闲链表。

3. 实现 kalloc()和 kfree()，用于分配和释放物理页。

#### 4.1.3 阶段三：实现虚拟内存管理

1. 创建 vm.c，实现 walk()函数，用于遍历三级页表并找到指定虚拟地址的 PTE。

2. 实现 mappages()函数，用于将一段虚拟地址连续地映射到一段物理地址。

3. 实现 kvminit()，分配根页表，并调用 mappages 完成内核代码段、数据段和 UART 的映射。

4. 实现 kvminithart()，将根页表的物理地址写入 satp 寄存器并刷新 TLB，正式启用分页。

#### 4.1.4 阶段四：集成与测试

1. 在 main.c 中按顺序调用 kinit, kvminit, kvminithart。

2. 编写并加入 test_physical_memory, test_pagetable, test_virtual_memory 三个分层测试函数。

3. 编译并运行，观察测试结果是否全部通过。

### 4.2 核心关键代码理解总结

- **关键代码 1：物理内存分配器核心算法。** 物理内存分配器采用空闲链表管理策略，这是操作系统内存管理的基础设施。其核心实现包含三个关键函数：

  ① kinit() - 内存初始化

  - 内存范围确定：从_end 符号（内核结束地址）开始，到 PHYSTOP（物理内存顶部）结束，这是内核未使用的可用内存区域。
  - 地址对齐处理：PGROUNDUP 确保起始地址按 4KB 页面对齐，这是 RISC-V 架构的硬件要求。
  - 链表构建：通过循环调用 kfree 将每个物理页插入空闲链表，构建

初始可用内存池。

```
void kinit() {
  char *p = (char*)PGROUNDUP((uint64)_end);
  for(; p + PGSIZE <= (char*)PHYSTOP; p += PGSIZE) {
    kfree(p);
  }
}
```

图 5 内存初始化函数

② kalloc() - 物理页分配

- LIFO 策略：从链表头部分配，实现 O(1)时间复杂度。
- 内存清理：分配后使用 memset 填充特定值(5)，便于调试和检测未初始化内存使用。
- 无锁设计：基于单核假设，简化并发控制。

```
void* kalloc(void) {
  struct run *r;
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;

  if(r)
    memset((char*)r, 5, PGSIZE);
  return (void*)r;
}
```

图 6 物理页分配函数

③ kfree() - 物理页释放

- 安全擦除：释放时填充值 1，防止敏感数据泄漏。
- 头插法：将释放的页插入链表头部，保持分配局部性。
- 内存复用：释放的页可立即被重新分配，提高内存利用率。

```
void kfree(void *pa) {
  struct run *r;
  memset(pa, 1, PGSIZE);
  r = (struct run*)pa;
  r->next = kmem.freelist;
  kmem.freelist = r;
}
```

图 7 物理页释放函数

- **关键代码 2：页表遍历算法深度解析。** walk 函数是虚拟内存系统的核心，负责将虚拟地址转换为物理地址的页表项。三级页表遍历机制如下：

① **地址解析阶段：**

- PX(level, va)宏从 39 位虚拟地址中提取各级页表索引。
- Level 2: VPN[2] (bits 38-30)。
- Level 1: VPN[1] (bits 29-21)。
- Level 0: VPN[0] (bits 20-12)。

① **页表项有效性检查：**

- 检查 PTE_V 位，判断当前页表项是否有效。
- 有效时通过 PTE2PA 提取下一级页表物理地址。
- 无效时根据 alloc 参数决定是否创建新页表。

② **动态页表创建：**

- 调用 kalloc()分配 4KB 物理页作为新页表。
- memset 清零确保所有 PTE 初始为无效状态。
- 设置父 PTE 指向新页表，并标记为有效。

③ **返回值设计：**

- 成功时返回最后一级 PTE 的指针。
- 失败时返回 0，支持错误处理。

```
pte_t* walk(pagetable_t pagetable, uint64 va, int alloc) {
  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pagetable[PX(level, va)];
    if(*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      if(!alloc || (pagetable = (pagetable_t)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)];
}
```

图 8 页表遍历算法

- **关键代码 3：内存映射建立机制。**mappages 函数负责在虚拟地址和物理地址

之间建立映射关系。映射建立流程如下：

① **地址对齐处理：**

- PGROUNDDOWN 确保虚拟和物理地址按页面对齐。
- 计算映射范围的起始和结束页面。

② **逐页映射循环：**

- 对每个虚拟页调用 walk 获取对应的 PTE。
- 检查是否发生重映射，输出警告信息。
- 设置 PTE 的物理页号、权限位和有效位。

③ **权限位组合：**

- PTE_R | PTE_W | PTE_X | PTE_U 的不同组合。
- 内核代码段：PTE_R | PTE_X（可读可执行）。
- 内核数据段：PTE_R | PTE_W（可读可写）。
- 设备内存：PTE_R | PTE_W（可读可写）。

```
int mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm) {
  uint64 a, last;
  pte_t *pte;

  a = PGROUNDDOWN(va);
  last = PGROUNDDOWN(va + size - 1);
  for(;;){
    if((pte = walk(pagetable, a, 1)) == 0)
      return -1;
    if(*pte & PTE_V)
      printf("mappages: remap\n");
    *pte = PA2PTE(pa) | perm | PTE_V;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

图 9 内存映射建立

- **关键代码 4：内核页表初始化与激活**

① **kvminit() - 内核页表构建：**包含 UART 设备映射，内核代码段映射和内核数据段映射。

```
void kvminit(void) {
  kernel_pagetable = (pagetable_t) kalloc();
  memset(kernel_pagetable, 0, PGSIZE);

  mappages(kernel_pagetable, UART0, PGSIZE, UART0, PTE_R | PTE_W);
  mappages(kernel_pagetable, KERNBASE, (uint64)etext-KERNBASE, KERNBASE, PTE_R | PTE_
X);
  mappages(kernel_pagetable, (uint64)etext, PHYSTOP-(uint64)etext, (uint64)etext, PTE_R
| PTE_W);
}
```

图 10 内核页表构建函数

② **kvminithart() - 虚拟内存激活：**包含 SATP 寄存器设置和 TLB 一致性维护。

```
void kvminithart() {
  w_satp(MAKE_SATP(kernel_pagetable));
  sfence_vma();
}
```

图 11 虚拟内存激活函数

# 5 实验测试与结果

## 5.1 实验测试

在完成了代码的编写和集成后，执行 make run 指令，系统启动并自动执行分层测试。终端输出如下：

```
=== 内核编译完成 ===
xiaoruqi@xiaoruqi-VMware-Virtual-Platform:~/Desktop/OS_experiment/lab3$ make run
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -nographic
xv6-riscv-simplified by @X XX

--- Test 1: Physical Memory Allocator ---
Physical memory test PASSED

--- Test 2: Page Table Functionality ---
Page table test PASSED

--- Test 3: Virtual Memory Activation ---
Kernel code is executable after enabling paging.
Kernel data is accessible.
Device (UART) access is working.
Virtual memory test PASSED

====== All Tests Passed! ======
Booting complete!
QEMU: Terminated
```

图 12 运行测试截图

### 5.2 结果分析

- 物理内存测试通过：表明 kalloc 能成功分配不同页，kfree 后内存能被正确回收复用。
- 页表功能测试通过：表明 walk 和 mappages 能正确创建映射，PTE 的权限位设置正确，并且通过映射后的虚拟地址可以成功读写物理内存。
- 虚拟内存激活测试通过：表明启用分页后，内核页表正确工作。printf 依然能输出（UART 映射成功），内核代码能继续执行（.text 段映射成功），内核数据能被修改（.data 段映射成功）。

# 6 遇到的问题及解决

## 6.1 问题 1：沿袭第四节的启动结构，导致链接错误与启动流程不兼容

- **现象**：在实验二的代码基础上添加内存管理代码后，make 时链接器报错 undefined reference to '_end'。
- **分析**：实验二的启动汇编 entry.S 非常简单，它依赖链接脚本提供的 _end 符号来动态计算栈顶地址。然而，实验三引入了 xv6 的内存模型，需要一个标准化的启动流程来完成从 M-mode 到 S-mode 的切换，并使用静态定义的栈。旧的 entry.S 无法满足这些新需求，导致了链接错误。
- **解决**：放弃了实验二的启动方式，完全重构为与 xv6 一致的启动流程。具体步骤是：创建 start.c 负责 M-mode 初始化和模式切换；重写 entry.S 以设置静态栈并调用 start()；最后更新 Makefile 将 start.c 加入编译。

## 6.2 问题 2：编译 start.c 时出现大量 implicit declaration 和 undefined reference 错误

- **现象**：在实验二的代码基础上添加内存管理代码后，make 时链接器报错 undefined reference to '_end'。start.c 文件本身编译时就报出一连串关于 r_mstatus 等函数未声明的警告，链接时则彻底报错。

- **分析：**start.c 中使用了大量读写 CPU 控制寄存器（CSR）的内联汇编函数，这些函数的定义都位于 riscv.h 中。经检查，发现 start.c 忘记了 #include "riscv.h"，同时 riscv.h 文件本身也不完整，缺少了大部分 CSR 操作函数的定义。
- **解决：**首先，为 start.c 文件添加#include "riscv.h"。然后，用一个包含了所有必需的 CSR 操作函数（r_mstatus, w_mepc, w_pmpcfg0 等）的完整版本替换了 kernel/riscv.h 文件。

## 6.3 问题 3：启用分页后，程序在 kinit 函数中“卡死”

- **现象：**在实验二的代码基础上添加内存管理代码后，make 时链接器报错 undefined reference to '_end'。
- **分析：**memset 试图写入内核代码段之外的物理内存。在 RISC-V 架构中，S-mode 对物理内存的访问权限受到 M-mode 配置的 PMP（物理内存保护）寄存器的限制。如果未明确配置，S-mode 默认没有权限访问所有内存。访问一个没有权限的地址会触发一个精确的访问异常（trap）。由于此时我们还没有实现异常处理程序，CPU 遇到异常后就进入了一个无声的崩溃-重启循环，表现为“卡死”。
- **解决：**在 start.c 的 mret 指令前，添加配置 pmpaddr0 和 pmpcfg0 的代码，为 S-mode 授予对全部物理内存（0x3fffffffffffull）的读、写、执行权限（0xf）。

# 7 思考题

## 7.1 架构设计

### 7.1.1 你的物理内存分配器与 xv6 有什么不同？

主要不同点在于并发安全。我的实现是一个简化的版本，没有包含任何锁机制。而 xv6 的 kalloc.c 中，kalloc() 和 kfree() 函数在修改全局空闲链表 kmem.freelist 之前，都会通过 acquire(&kmem.lock) 获取一个自旋锁，操作完成后再通过 release(&kmem.lock) 释放锁。

### 7．1．2　为什么选择这种设计？有什么权衡？

选择简化设计的原因是为了聚焦本次实验的核心目标——理解内存分配算法和页表机制本身。在单核、无进程调度的早期内核启动阶段，不存在并发访问空闲链表的可能，因此可以安全地省略锁。

权衡之处在于通用性 vs 复杂度。

优点：代码更简单，逻辑更清晰，更容易理解物理页分配的核心思想。

缺点：这个分配器不是线程安全的。一旦未来内核引入多核支持或中断驱动的并发任务，就必须像 xv6 一样加上锁，否则会导致链表损坏和内存分配错误。

## 7.2 内存安全

### 7．2．1　如何防止内存分配器被恶意利用？

1. 严格的指针检查：在 kfree() 中，必须严格检查传入的地址 pa 是否合法。例如，检查它是否页对齐，是否在可管理的物理内存范围内（end 到 PHYSTOP 之间）。这可以防止内核的其他部分错误地释放一个无效地址，从而破坏空闲链表。

2. 隔离内核与用户空间：这是最重要的安全机制。内核内存分配器分配的物理页，绝不能直接暴露给用户进程。当用户进程需要内存时，内核应分配物理页，然后通过页表将其映射到用户进程的虚拟地址空间，并设置好权限（如 PTE_U 标志，清除 PTE_W 实现只读等）。

3. 栈保护: 虽然与分配器不直接相关，但在函数栈上放置哨兵值可以检测栈溢出，防止恶意代码通过覆盖返回地址来执行攻击。

### 7．2．1　页表权限设置的安全考虑有哪些？

1. 最小权限原则：内核代码段（.text）应映射为只读、可执行（R-X），防止被意外或恶意修改。内核数据段（.data, .bss）应映射为可读、可写（R-W），但不可执行，这可以防止代码注入攻击。

2. 用户/监督模式隔离：所有内核空间的映射（代码、数据、页表自身）都不能设置 PTE_U 位，确保用户进程无法读取或修改内核内存。

3. 栈保护页：在用户栈或内核栈的底部设置一个无效的页表项（PTE 的 V 位为 0）。当发生栈溢出时，访问这个地址会立即触发缺页异常，而不是悄无声

息地破坏下方内存。

4. 写时复制：在 fork() 时，父子进程可以共享只读的物理页面。当任何一方尝试写入时，触发缺页异常，内核再为其分配一个新的可写页面副本。这既高效又安全。

## 7.3 性能分析

### 7.3.1 当前实现的性能瓶颈在哪里？

1. 物理内存分配：虽然 kalloc/kfree 是 O(1) 操作，但如果未来系统有大量、频繁的小块内存分配需求（而不是整页分配），这个简单的页分配器会非常低效，并产生大量内部碎片。此时需要更复杂的 slab 或 buddy 分配器。

2. TLB Miss：walk 函数每次进行地址转换都需要三次内存访问（三级页表）。如果程序的内存访问局部性差，会导致大量的 TLB（Translation Lookaside Buffer）未命中，每次都需要 MMU 硬件或软件重新遍历页表，这将是主要的性能瓶颈。

### 7.3.2 如何测量和优化内存访问性能？

- **测量：**可以通过硬件性能计数器（Performance Counters）来测量 TLB Miss 的频率、缺页中断的次数等指标。
- **优化：**
  1. 使用大页：对于大块的连续内存区域（如内核自身、帧缓冲），可以使用 2MB 或 1GB 的大页进行映射。一个大页 PTE 可以直接映射一大块物理内存，减少了页表级数和 TLB 条目数，从而降低 TLB Miss 率。
  2. 优化数据结构布局：在编程时，有意识地将频繁一起访问的数据放在同一个内存页中，以提高空间局部性。
  3. 预取：在访问一个数据前，使用预取指令将可能很快会用到的数据提前加载到缓存中。

# 8 实验总结

通过本次实验，我成功地从零开始构建了一个简化的操作系统内核内存管理系统。这不仅是一次编码练习，更是一次对操作系统底层原理的深刻探索。

从最初面对 undefined reference 链接错误的困惑，到通过分析 xv6 启动流程找到解决方案；从程序在 kinit 中神秘“卡死”，到最终定位到 RISC-V PMP 这一硬件特性，整个过程极大地锻炼了我底层调试和问题分析的能力。亲手实现 walk 和 mappages 函数让我对虚拟地址如何一步步“翻译”成物理地址有了直观的认识，而配置 satp 寄存器并启用分页的那一刻，则让我真切感受到了操作系统是如何接管硬件内存管理单元的。

本次实验让我将《操作系统》课本上的抽象概念——物理页、页表、地址转换——转化为了看得见、摸得着、能运行的代码，为后续学习更复杂的内核功能（如进程管理和缺页中断）打下了坚实的基础。

# 教师评语评分

评语：_____

_____

_____

_____

_____

_____

_____

_____

评分：_____

评阅人：

年　　月　　日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）