实验3：页表与内存管理

一、实验概述

1. 实验目标

实现 **RISC-V 架构** 下的：

- **物理内存分配器**
- **三级页表管理系统**
- **虚拟内存激活机制**

并验证：

- 内存分配的**正确性**
- 页表映射的**准确性**
- 虚拟内存的**正常工作**

---

2. 完成情况

- ☑ **实现物理内存管理器**（`alloc_page / free_page`）
- ☑ **实现三级页表核心操作**（创建、映射、查找、销毁）
- ☑ **实现内核虚拟内存初始化与激活**
- ☑ **完成物理内存、页表、虚拟内存的测试用例**
- ☑ **验证页面对齐、读写有效性、地址映射正确性**

---

3. 开发环境

- **OS**：:contentReference[oaicite:0]{index=0}
- **Toolchain**：`riscv64-unknown-elf-gcc 12.2.0`
- **QEMU**：:contentReference[oaicite:1]{index=1} `qemu-system-riscv64 7.2.0`
- **参考源码**：:contentReference[oaicite:2]{index=2}（commit: `8b96797`）

二、技术设计

1. 系统架构

本次实验实现的内存管理系统整体架构如下：

```
[物理内存层] → 空闲页链表管理物理内存，提供 alloc/free 接口
↓
[页表管理层] → 三级页表实现虚拟地址到物理地址的映射
↓
[虚拟内存层] → 初始化内核页表，激活 Sv39 分页模式
```

与 xv6 的对比

- **相同点**
  - 使用空闲链表管理物理内存
  - 采用 **Sv39 三级页表机制**
- **简化点**
  - 仅支持 **内核态内存管理**
  - 使用固定物理内存区间，简化内存探测逻辑
- **增强点**
  - 页面分配时 **默认清零**
  - 页表操作增加 **详细调试输出**

---

2. 关键数据结构

2.1 物理内存管理结构

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

static struct {
    struct run *freelist;
    uint64 start;
    uint64 end;
} pmm;
```

设计理由：

- 使用 **单链表** 管理空闲页，结构简单、效率高
- 全局状态结构体便于统一维护内存范围和分配状态

2.2 页表相关定义

```
typedef uint64 pte_t;
typedef pte_t* pagetable_t;

#define PTE_V 0x001
#define PTE_R 0x002
#define PTE_W 0x004
#define PTE_X 0x008
```

```
#define PPN_SHIFT 10
#define PGSIZE 4096
```

设计理由:

- 严格遵循 **Sv39 页表格式**
- 通过宏定义隔离硬件细节，提高可维护性

---

3. 核心流程

3.1 物理内存分配流程

```
void* alloc_page(void) {
    if (pmm.freelist == NULL)
        return NULL;

    struct run *r = pmm.freelist;
    pmm.freelist = r->next;

    memset((void*)r, 0x00, PGSIZE);
    return (void*)r;
}

void free_page(void *page) {
    if (((uint64)page & (PGSIZE - 1)) != 0 ||
        (uint64)page < pmm.start ||
        (uint64)page >= pmm.end) {
        panic("invalid free page");
    }

    memset(page, 0x00, PGSIZE);

    struct run *r = (struct run*)page;
    r->next = pmm.freelist;
    pmm.freelist = r;
}
```

3.2 页表映射流程

```
int map_page(pagetable_t pt, uint64 va, uint64 pa, int perm) {
    if ((va & (PGSIZE - 1)) || (pa & (PGSIZE - 1)))
        return -1;

    pte_t *pte = walk_create(pt, va);
    if (pte == NULL)
        return -1;
```

```
    *pte = PA2PTE(pa) | perm | PTE_V;
    return 0;
}
```

```
pte_t* walk_create(pagetable_t pt, uint64 va) {
    for (int level = 2; level >= 0; level--) {
        pte_t *pte = &pt[VPN(va, level)];
        if (*pte & PTE_V) {
            pt = (pagetable_t)PTE2PA(*pte);
        } else {
            pt = (pagetable_t)alloc_page();
            if (pt == NULL) return NULL;
            memset(pt, 0, PGSIZE);
            *pte = PA2PTE((uint64)pt) | PTE_V;
        }
    }
    return &pt[VPN(va, 0)];
}
```

三、实现细节

1. 物理内存初始化（pmm_init）

```
void pmm_init(uint64 start, uint64 end) {
    pmm.start = (start == 0) ? PHYS_MEM_START : start;
    pmm.end   = (end   == 0) ? PHYS_MEM_END   : end;

    pmm.freelist = NULL;
    uint64 page_addr = pmm.start;

    while (page_addr + PGSIZE <= pmm.end) {
        struct run *r = (struct run*)page_addr;
        r->next = pmm.freelist;
        pmm.freelist = r;
        page_addr += PGSIZE;
    }

    printf("PMM initialized: start=0x%p, end=0x%p, total pages=%d\n",
           (void*)pmm.start, (void*)pmm.end,
           (pmm.end - pmm.start) / PGSIZE);
}
```

---

2. 页表查找（walk_lookup）

```
pte_t* walk_lookup(pagetable_t pt, uint64 va) {
    for (int level = 2; level >= 0; level--) {
        pte_t *pte = &pt[VPN(va, level)];
        if (*pte & PTE_V)
            pt = (pagetable_t)PTE2PA(*pte);
        else
            return NULL;
    }
    return &pt[VPN(va, 0)];
}
```

```
#define VPN(va, level) (((va) >> (12 + 9 * level)) & 0x1FF)
#define PA2PTE(pa) ((pa) >> 12)
#define PTE2PA(pte) ((pte) << 12)
```

---

3. 虚拟内存激活（kvminit / kvminithart）

```
pagetable_t kernel_pagetable;

void kvminit(void) {
    kernel_pagetable = create_pagetable();
    if (!kernel_pagetable)
        panic("kvminit failed");

    map_page(kernel_pagetable, (uint64)_text,
             (uint64)_text, PTE_R | PTE_X);

    map_page(kernel_pagetable, (uint64)_etext,
             (uint64)_etext,
             (uint64)_end - (uint64)_etext,
             PTE_R | PTE_W);

    map_page(kernel_pagetable, UART0, UART0,
             PGSIZE, PTE_R | PTE_W);
}
```

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

```
#define MAKE_SATP(pt) ((0x8UL << 60) | ((uint64)(pt) >> 12))
```

四、测试与验证

1. 功能测试

测试1：物理内存分配测试

测试内容：

- 页面分配与释放
- 地址对齐检查
- 数据读写验证

运行结果：

```
--- Testing Physical Memory Allocator (kalloc/kfree) ---
pmm: alloc_page -> 0x80043000 (remain=63)
pmm: alloc_page -> 0x80042000 (remain=62)
page1=0x80043000, page2=0x80042000
page1 aligned OK
write/read OK: 0x12345678
pmm: free_page <- 0x80043000 (remain=63)
pmm: alloc_page -> 0x80043000 (remain=62)
page3=0x80043000 (may equal page1)
pmm: free_page <- 0x80042000 (remain=63)
pmm: free_page <- 0x80043000 (remain=64)
=== Physical Memory Test End ===
```

测试2：页表映射测试

```
level 2: vpn=0x1, pte=0x80004001
level 1: vpn=0x0, pte=0x80005001
level 0: vpn=0x0, pte=0x80006007
```

```
[Test 2] Page Table Management

--- 2. Testing Page Table Functions ---
pmm: alloc_page -> 0x80043000 (remain=63)
pmm: alloc_page -> 0x80042000 (remain=62)
pmm: alloc_page -> 0x80041000 (remain=61)
pmm: alloc_page -> 0x80040000 (remain=60)
Dump pagetable:
MAP: va=0x40000000 -> pa=0x80042000 perm=0x6
lookup: va=0x40000000 -> pa=0x80042000
pmm: free_page <- 0x80042000 (remain=61)
pmm: free_page <- 0x80040000 (remain=62)
pmm: free_page <- 0x80041000 (remain=63)
pmm: free_page <- 0x80043000 (remain=64)
```

测试3：虚拟内存激活测试

```
=== Virtual Memory Test Start ===
Before enabling paging...
Current mode: direct memory access
pmm: alloc_page -> 0x80043000 (remain=63)
pmm: alloc_page -> 0x80041000 (remain=62)
pmm: alloc_page -> 0x80040000 (remain=61)
pmm: alloc_page -> 0x80042000 (remain=60)
pmm: alloc_page -> 0x8003f000 (remain=59)
kvminit: kernel_pagetable created and regions mapped
Kernel pagetable created
kvminithart: satp set 0x8000000000080043
Paging enabled, satp register set
After enabling paging...
Current mode: virtual memory with paging
Testing kernel code execution...
Kernel text: 0x80000000 - 0x800017ca
Testing kernel data access...
Test variable value: 0xdeadbeef
Kernel data access OK
Testing device access...
Device access test OK (UART mapped)
Testing pagetable lookup...
KERNBASE lookup OK: va=0x80000000 -> pa=0x80000000
=== Virtual Memory Test End ===
```

五、问题与总结

1. 遇到的问题

问题1：分页后访问崩溃

- **原因**：映射范围不完整
- **解决**：严格使用 `_text / _etext / _end`

问题2：物理页未对齐

- **原因**：初始化未按 `PGSIZE` 对齐
- **解决**：使用 `PGROUNDUP`

问题3：UART 输出乱码

- **原因**：设备未映射
- **解决**：加入 UART MMIO 映射

2. 实验收获

- 深入理解 **Sv39 地址转换机制**
- 掌握 **内核级内存管理实现**
- 提升 **内核调试与问题定位能力**

3. 改进方向

- 引入 **伙伴系统 / slab 分配器**
- 支持 **大页映射**
- 完善 **错误回滚与一致性保障**