常见问题诊断

输出不完整：

- 检查UART发送是否等待完成
- 验证字符串是否正确终止
- 确认缓冲区大小是否足够

数字输出错误：

- 验证进制转换算法
- 检查负数处理逻辑
- 测试INT_MIN等边界值

格式解析错误：

- 打印解析过程的中间状态
- 验证va_arg的参数类型匹配
- 检查未知格式符的处理

思考题

1. **架构设计：**
   - 为什么需要分层？每层的职责如何划分？
   - 如果要支持多个输出设备（串口+显示器），架构如何调整？
2. **算法选择：**
   - 数字转字符串为什么不用递归？
   - 如何在不使用除法的情况下实现进制转换？
3. **性能优化：**
   - 当前实现的性能瓶颈在哪里？
   - 如何设计一个高效的缓冲机制？
4. **错误处理：**
   - printf遇到NULL指针应该如何处理？
   - 格式字符串错误时的恢复策略是什么？

实验3：页表与内存管理

实验目标

通过深入分析xv6的内存管理系统，理解虚拟内存的工作原理，独立实现物理内存分配器和页表管理系统。

核心学习资料

RISC-V内存管理机制

- **RISC-V特权级规范 第12章： Supervisor-Level ISA**
  - 12.4 Sv39: Page-Based 39-bit Virtual-Memory System
- **在线文档：**https://github.com/riscv/riscv-isa-manual

xv6内存管理源码分析

- `kernel/kalloc.c` - 物理内存分配器
  - 重点函数：`kinit()`，`kalloc()`，`kfree()`
  - 学习要点：空闲页链表管理、简单分配算法
- `kernel/vm.c` - 虚拟内存管理
  - 重点函数：`walk()`，`mappages()`，`uvmcreate()`
  - 学习要点：页表遍历、映射建立、地址转换
- `kernel/riscv.h` - RISC-V相关定义
  - 重点内容：页表项格式、权限位定义、地址操作宏

内存管理理论基础

- **操作系统概念 第9-10章：**内存管理和虚拟内存
  - 在线图书：https://www.os-book.com/OS10/index.html
- **深入理解计算机系统 第9章：**虚拟内存

任务列表

任务1：深入理解Sv39页表机制

学习重点：

1. 分析39位虚拟地址的分解：

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

- 每个VPN段的作用是什么？
- 为什么是9位而不是其他位数？

2. 理解页表项（PTE）格式：
   - V位：有效性标志
   - R/W/X位：读/写/执行权限
   - U位：用户态访问权限
   - 物理页号（PPN）的提取方式

深入思考：

- 为什么选择三级页表而不是二级或四级？

- 中间级页表项的R/W/X位应该如何设置？
- 如何理解"页表也存储在物理内存中"？

任务2：分析xv6的物理内存分配器

代码阅读指导：

1. 研读 `kalloc.c` 的核心数据结构：

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

- 这个设计有什么巧妙之处？
- 为什么不需要额外的元数据存储？

2. 分析 `kinit()` 的初始化过程：
    - 如何确定可分配的内存范围？
    - 空闲页链表是如何构建的？
    - 为什么要按页对齐？
3. 理解 `kalloc()` 和 `kfree()` 的实现：
    - 分配算法的时间复杂度是多少？
    - 如何防止double-free？
    - 这种设计的优缺点是什么？

设计思考：

- 如果要实现内存统计功能，应该如何扩展？
- 如何检测内存泄漏？
- 更高效的分配算法有哪些？

任务3：设计你的物理内存管理器

设计要求：

1. 确定内存布局方案
2. 选择合适的数据结构
3. 实现分配和释放接口

关键设计决策：

```
// 你需要决定的接口设计
void pmm_init(void);              // 初始化内存管理器
void* alloc_page(void);           // 分配一个物理页
void free_page(void* page);       // 释放一个物理页
void* alloc_pages(int n);         // 分配连续的n个页面（可选）

// 你需要考虑的问题：
// 1. 如何确定可用内存范围？
// 2. 如何处理内存碎片？
```

```
// 3. 是否需要支持不同大小的分配？
```

实现策略：

1. 首先实现最简单的链表方案
2. 添加基本的错误检查
3. 考虑性能优化（如适用）

任务4：理解xv6的页表管理

代码阅读重点：

1. 分析 `walk()` 函数的递归遍历：
   - 如何从虚拟地址提取各级索引？
   - 遇到无效页表项时如何处理？
   - 为什么需要 `alloc` 参数？
2. 研究 `mappages()` 的映射建立：
   - 如何处理地址对齐？
   - 权限位是如何设置的？
   - 映射失败时的清理工作
3. 理解地址转换宏定义：

```
#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
#define PTE_PA(pte)    (((pte) >> 10) << 12)
```

实现挑战：

- 如何避免页表遍历中的无限递归？
- 映射过程中的内存分配失败应该如何恢复？
- 如何确保页表的一致性？

任务5：实现你的页表管理系统

核心接口设计：

```
// 页表类型定义
typedef uint64* pagetable_t;

// 基本操作接口
pagetable_t create_pagetable(void);
int map_page(pagetable_t pt, uint64 va, uint64 pa, int perm);
void destroy_pagetable(pagetable_t pt);

// 辅助函数（内部使用）
pte_t* walk_create(pagetable_t pt, uint64 va);
pte_t* walk_lookup(pagetable_t pt, uint64 va);
```

实现步骤指导：

1. **地址解析实现：**

```
// 从虚拟地址提取页表索引
#define VPN_SHIFT(level) (12 + 9 * (level))
#define VPN_MASK(va, level) (((va) >> VPN_SHIFT(level)) & 0x1FF)
```

2. **页表遍历实现：**
   - 从根页表开始逐级查找
   - 检查每一级页表项的有效性
   - 必要时创建中间级页表
3. **映射建立实现：**
   - 确保地址按页对齐
   - 正确设置权限位
   - 处理映射冲突

调试检查点：

```
// 实现页表打印功能用于调试
void dump_pagetable(pagetable_t pt, int level) {
    // 递归打印页表内容
    // 显示虚拟地址到物理地址的映射关系
    // 标明权限位设置
}
```

任务6：启用虚拟内存

参考xv6的内核初始化：

1. 研读 `kvminit()` 的内核页表创建：
   - 哪些内存区域需要映射？
   - 为什么采用恒等映射？
   - 设备内存的权限设置
2. 分析 `kvminithart()` 的页表激活：
   - satp寄存器的格式和设置
   - `sfence.vma` 指令的作用
   - 激活前后的注意事项

实现策略：

```
void kvminit(void) {
    // 1. 创建内核页表
    kernel_pagetable = create_pagetable();

    // 2. 映射内核代码段（R+X权限）
    map_region(kernel_pagetable, KERNBASE, KERNBASE,
               (uint64)etext - KERNBASE, PTE_R | PTE_X);
```

```
    // 3. 映射内核数据段（R+W权限）
    map_region(kernel_pagetable, (uint64)etext, (uint64)etext,
               PHYSTOP - (uint64)etext, PTE_R | PTE_W);

    // 4. 映射设备（UART等）
    map_region(kernel_pagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W);
}

void kvminithart(void) {
    // 激活内核页表
    w_satp(MAKE_SATP(kernel_pagetable));
    sfence_vma();
}
```

关键技术细节：

- SATP寄存器格式：`MODE[63:60] | ASID[59:44] | PPN[43:0]`
- MODE=8表示Sv39模式
- `sfence.vma` 用于刷新TLB

测试与调试策略

分层测试方法

1. **物理内存分配器测试：**

```
void test_physical_memory(void) {
    // 测试基本分配和释放
    void *page1 = alloc_page();
    void *page2 = alloc_page();
    assert(page1 != page2);
    assert(((uint64)page1 & 0xFFF) == 0);  // 页对齐检查

    // 测试数据写入
    *(int*)page1 = 0x12345678;
    assert(*(int*)page1 == 0x12345678);

    // 测试释放和重新分配
    free_page(page1);
    void *page3 = alloc_page();
    // page3可能等于page1（取决于分配策略）

    free_page(page2);
    free_page(page3);
}
```

2. **页表功能测试：**

```
void test_pagetable(void) {
```

```
    pagetable_t pt = create_pagetable();

    // 测试基本映射
    uint64 va = 0x1000000;
    uint64 pa = (uint64)alloc_page();
    assert(map_page(pt, va, pa, PTE_R | PTE_W) == 0);

    // 测试地址转换
    pte_t *pte = walk_lookup(pt, va);
    assert(pte != 0 && (*pte & PTE_V));
    assert(PTE_PA(*pte) == pa);

    // 测试权限位
    assert(*pte & PTE_R);
    assert(*pte & PTE_W);
    assert(!(*pte & PTE_X));
}
```

3. **虚拟内存激活测试：**

```
void test_virtual_memory(void) {
    printf("Before enabling paging...\n");

    // 启用分页
    kvminit();
    kvminithart();

    printf("After enabling paging...\n");

    // 测试内核代码仍然可执行
    // 测试内核数据仍然可访问
    // 测试设备访问仍然正常
}
```

常见问题诊断

问题：启用分页后系统崩溃

- 检查点1：内核代码是否正确映射？
- 检查点2：栈空间是否映射？
- 检查点3：设备地址是否映射？
- 调试方法：在启用前后打印关键地址的映射状态

问题：页表映射失败

- 检查地址对齐：虚拟地址和物理地址都必须页对齐
- 检查内存不足：中间页表创建可能失败
- 检查权限冲突：重复映射可能导致权限不一致

问题：地址转换错误

- 验证VPN提取算法是否正确
- 检查PTE格式是否符合RISC-V规范
- 确认物理地址计算是否正确

GDB调试技巧

```
# 查看页表内容
(gdb) x/64gx $satp_register_content
# 查看特定虚拟地址的映射
(gdb) monitor info mem
# 检查页表遍历过程
(gdb) b walk_create
(gdb) watch $a0  # 监视页表指针变化
```

性能优化考虑

内存分配优化

1. **批量分配**：一次性分配多个连续页面
2. **分级分配**：针对不同大小需求使用不同分配器
3. **缓存优化**：保持少量预分配页面池

页表优化

1. **TLB友好**：合理安排虚拟地址布局
2. **大页支持**：对于大块内存使用大页映射
3. **延迟映射**：按需创建页表项

思考题

1. **设计对比**：
   - 你的物理内存分配器与xv6有什么不同？
   - 为什么选择这种设计？有什么权衡？
2. **内存安全**：
   - 如何防止内存分配器被恶意利用？
   - 页表权限设置的安全考虑有哪些？
3. **性能分析**：
   - 当前实现的性能瓶颈在哪里？
   - 如何测量和优化内存访问性能？
4. **扩展性**：
   - 如果要支持用户进程，需要什么修改？
   - 如何实现内存共享和写时复制？
5. **错误恢复**：

- 页表创建失败时如何清理已分配的资源？
- 如何检测和处理内存泄漏？

实验4：中断处理与时钟管理

实验目标

通过分析xv6的中断处理机制，理解操作系统如何响应硬件事件，实现完整的中断处理框架和时钟中断驱动的任务调度。

核心学习资料

RISC-V中断机制

- **RISC-V特权级规范 第3章**：Machine-Level ISA
  - 3.1.9节：Machine Interrupt Registers
  - 3.2.1节：Machine Timer Registers
- **RISC-V特权级规范 第12章**：Supervisor-Level ISA
  - 12.1.3节：Supervisor Interrupt Registers
  - 重点理解：mie、mip、sie、sip寄存器的作用

xv6中断处理源码分析

- `kernel/trap.c` - 中断和异常处理
  - 重点函数：`usertrap()`，`kerneltrap()`，`devintr()`
  - 学习要点：中断分发、异常处理、系统调用入口
- `kernel/kernelvec.S` - 内核态中断向量
  - 重点：上下文保存和恢复机制
- `kernel/start.c` - 机器模式初始化
  - 重点函数：timer中断的设置和代理

时钟管理理论

- **SBI规范**：https://github.com/riscv-non-isa/riscv-sbi-doc
  - 第4.6节：Timer Extension
- **操作系统概念 第5章**：CPU调度

任务列表

任务1：理解RISC-V中断架构

学习重点：