操作系统内核实验报告规范

总体说明

报告目标

实验报告的目的是**简洁明了**地记录你的实验过程、关键技术决策和实验结果，而不是追求篇幅长度。一份好的实验报告应该让读者快速理解：

- 你做了什么
- 你是怎么做的
- 遇到了什么问题以及如何解决
- 实验结果如何

提交格式

- 支持Markdown格式（`.md`文件）也可以是PDF
- 无字体、字号、行距等格式要求
- 建议使用标准Markdown语法，保持文档结构清晰
- 代码块使用语法高亮（如```c）
- 可嵌入图片、表格、流程图等辅助说明

报告结构

每个实验的报告应包含以下五个核心部分：

第一部分：实验概述（5–10%篇幅）

内容要求

简要说明本次实验的目标和完成情况。

必须包含

1. **实验目标**：用1–2句话概括本次实验要实现什么功能
2. **完成情况**：哪些任务已完成，哪些未完成（如有）
3. **开发环境**：操作系统、工具链版本、QEMU版本等关键信息

示例

```
## 实验概述

### 实验目标
实现RISC-V裸机启动，通过UART输出"Hello OS"，理解操作系统的引导过程。

```

```
### 完成情况
- ✅ 实现启动汇编代码（entry.S）
- ✅ 编写链接脚本（kernel.ld）
- ✅ 实现UART驱动
- ✅ 成功在QEMU中输出"Hello OS"

### 开发环境
- 操作系统：Ubuntu 22.04 LTS
- 工具链：riscv64-unknown-elf-gcc 12.2.0
- QEMU：qemu-system-riscv64 7.2.0
```

第二部分：技术设计（30–40%篇幅）

内容要求

说明你的实现方案和关键技术决策，这是报告的核心部分。

必须包含

1. 系统架构设计

- 绘制系统架构图或模块关系图
- 说明各模块的职责
- 与xv6对比，说明你的设计有何异同

2. 关键数据结构

- 列出核心数据结构的定义
- 解释为什么这样设计
- 如有改进xv6的地方，说明理由

3. 核心算法与流程

- 用流程图或伪代码说明关键算法
- 解释重要的技术选择
- 说明边界情况的处理方法

写作建议

- **不要**简单复制代码，应提取关键逻辑
- **使用图表**：架构图、流程图、状态图等比文字更清晰
- **突出亮点**：如果有创新设计或优化，重点说明

示例

```
## 技术设计

```

```
### 系统架构
我的启动流程分为三个阶段：

\```
[QEMU加载] → [entry.S汇编初始化] → [C语言main函数]
     ↓               ↓                        ↓
  加载到0x80000000  设置栈、清BSS         UART输出
\```

与xv6的区别：
- **简化**：xv6支持多核，我只实现单核启动
- **简化**：xv6有复杂的内存布局，我使用固定的8KB栈

### 关键数据结构

#### 页表项结构（实验3）
\```c
typedef struct {
    uint64 ppn  : 44;  // 物理页号
    uint64 rsw  : 2;   // 保留
    uint64 d    : 1;   // Dirty bit
    uint64 a    : 1;   // Access bit
    uint64 g    : 1;   // Global
    uint64 u    : 1;   // User可访问
    uint64 x    : 1;   // 可执行
    uint64 w    : 1;   // 可写
    uint64 r    : 1;   // 可读
    uint64 v    : 1;   // 有效位
} pte_t;
\```

**设计理由**：采用位域结构，清晰表达RISC-V Sv39页表项格式。

### 核心流程：物理内存分配（实验3）

\```c
void* kalloc() {
    if (freelist == NULL)
        return NULL;

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

    memset((void*)r, 0x00, PGSIZE);  // 清零页面
    return (void*)r;
}
\```
```

```
与xv6的区别：
- **相同**：采用空闲链表管理
- **增加**：分配时清零页面，防止信息泄漏
```

第三部分：实现细节与关键代码（20–30%篇幅）

内容要求

展示关键代码片段，并解释重点实现。

必须包含

1. 关键函数实现

- 选择2–3个最核心的函数
- 给出完整代码或关键片段
- 添加注释说明重点逻辑

2. 难点突破

- 说明实现过程中的技术难点
- 如何解决这些难点
- 踩过哪些坑，如何避免

3. 源码理解

- 对照xv6源码，说明你的理解
- 解答实验手册中的思考题（如适用）

写作建议

- **代码要精简**：不要粘贴整个文件，选择关键部分
- **注释要到位**：代码中的关键行添加注释
- **对比说明**：与xv6对比，说明异同

示例

```
## 实现细节

### 关键函数：walk_create（实验3）

\```c
// 页表遍历函数：给定虚拟地址，返回对应的PTE指针
// 如果中间级页表不存在，自动创建
pte_t* walk_create(pagetable_t pagetable, uint64 va) {
    // 从L2到L0三级遍历
```

```
    for (int level = 2; level \u003e= 0; level--) {
        pte_t *pte = \u0026pagetable[VPN(va, level)];  // 获取当前级PTE

        if (*pte \u0026 PTE_V) {
            // 有效，继续下一级
            pagetable = (pagetable_t)PTE2PA(*pte);
        } else {
            // 无效，分配新页表
            pagetable = (pagetable_t)kalloc();
            if (pagetable == NULL)
                return NULL;  // 内存不足

            *pte = PA2PTE(pagetable) | PTE_V;  // 设置PTE
        }
    }
    return \u0026pagetable[VPN(va, 0)];  // 返回叶子PTE
}
\```

**实现难点**：
1. **地址提取**：如何从64位虚拟地址正确提取3个9位VPN？
   - 解决：定义宏`#define VPN(va, level) (((va) \u003e\u003e (12 + 9*level

2. **内存分配失败处理**：中间级分配失败如何清理已分配的页表？
   - 当前实现：简单返回NULL，调用者负责检查
   - 改进方向：实现回滚机制

**与xv6对比**：
- xv6的walk函数有alloc参数控制是否创建页表，我直接创建
- xv6使用panic处理错误，我返回NULL让调用者处理
```

第四部分：测试与验证（20–30%篇幅）

内容要求

证明你的实现是正确的。

必须包含

1. 功能测试

- 基本功能测试用例及结果
- 边界情况测试
- 异常处理测试

2. 运行截图/录屏

- QEMU运行截图，显示关键输出

- 如有必要，提供运行视频链接
- 截图应清晰可读，不要截取整个屏幕

3. 性能数据（如适用）

- 如实验涉及性能优化，给出测试数据
- 对比优化前后的差异

写作建议

- **截图要有说明**：每张截图下方注明测试内容
- **测试要全面**：不仅测试正常情况，也要测试边界和异常
- **数据要真实**：不要编造测试结果

示例

```
## 测试与验证

### 功能测试

#### 测试1：基本启动（实验1）
**测试内容**：验证系统能否正确启动并输出

**运行结果**：
\```
$ make qemu
Hello OS!
Initialization complete.
\```

![启动成功截图](images/lab1-boot-success.png)

#### 测试2：printf格式化（实验2）
**测试代码**：
\```c
printf("Integer: %d\\n", 42);
printf("Hex: 0x%x\\n", 0xABCD);
printf("String: %s\\n", "Hello");
printf("Negative: %d\\n", -123);
printf("INT_MIN: %d\\n", -2147483648);
\```

**预期输出**：
\```
Integer: 42
Hex: 0xABCD
String: Hello
Negative: -123
```

```
INT_MIN: -2147483648
\```

**实际输出**：✅ 与预期一致

#### 测试3：页表映射（实验3）
**测试内容**：验证虚拟地址到物理地址的正确映射

\```c
void test_pagetable() {
    uint64 va = 0x1000000;
    uint64 pa = (uint64)kalloc();

    map_page(kernel_pt, va, pa, PTE_R | PTE_W);

    // 写入虚拟地址
    *(int*)va = 0x12345678;

    // 从物理地址读取
    printf("Read from PA: 0x%x\\n", *(int*)pa);
}
\```

**输出**：
\```
Read from PA: 0x12345678
\```
✅ 映射正确

### 边界测试

#### 测试：INT_MIN的正确处理
**测试代码**：`printf("%d", -2147483648)`
**结果**：✅ 正确输出（未使用取负导致溢出）

#### 测试：内存分配耗尽
**测试代码**：连续分配直到失败
**结果**：✅ kalloc正确返回NULL，未崩溃
```

第五部分：问题与总结（10–20%篇幅）

内容要求

记录实验过程中的问题、解决方案和收获。

必须包含

1. 遇到的问题与解决

- 列出3–5个典型问题
- 说明问题现象、原因分析、解决方法
- 给出预防建议

2. 实验收获

- 通过本次实验理解了哪些概念
- 对操作系统的哪些机制有了更深入的认识
- 代码能力或调试能力有何提升

3. 改进方向（可选）

- 当前实现的不足之处
- 如何进一步优化或扩展
- 对后续实验的启示

写作建议

- **问题要具体**：不要笼统地说”遇到了bug”
- **分析要深入**：尽量找到根本原因
- **总结要真实**：写出真实的收获，不要空话套话

示例

```
## 问题与总结

### 遇到的问题

#### 问题1：启用分页后系统崩溃

**现象**：
调用`kvminithart()`启用分页后，系统立即触发异常，QEMU显示：
\```
scause: 0xd (Load page fault)
sepc: 0x80001234
\```

**原因分析**：
启用分页后，程序计数器（PC）指向的代码地址未正确映射。检查页表发现，我只映射了数据段，

**解决方法**：
在kvminit中添加代码段映射：
\```c
map_region(kernel_pt, KERNBASE, KERNBASE,
           (uint64)etext - KERNBASE, PTE_R | PTE_X);
```

\```

预防建议：
启用分页前，用`dump_pagetable()`打印页表内容，确保所有必要的内存区域都已映射。

问题2：printf输出INT_MIN为正数

现象：
`printf("%d", -2147483648)` 输出 `2147483648`

原因分析：
我的printint函数对负数先取负再转换：
\```c
if (num \u003c 0) {
    console_putc('-');
    num = -num; // 错误！INT_MIN取负溢出
}
\```

INT_MIN取负后仍是INT_MIN（补码表示的特性），导致当作正数处理。

解决方法：
改为将负数转为unsigned后处理：
\```c
if (num \u003c 0) {
    console_putc('-');
    num = -(unsigned)num;
}
\```

预防建议：
处理整数边界值时，考虑补码表示的特殊性。

实验收获

1. **深入理解了启动过程**
   - 理解了为什么需要设置栈：C语言函数调用依赖栈保存局部变量和返回地址
   - 理解了BSS段清零的必要性：C语言规范要求未初始化全局变量为0
   - 认识到链接脚本的重要性：它决定了程序在内存中的布局

2. **掌握了虚拟内存机制**
   - 理解了三级页表的地址转换过程
   - 认识到页表本身也存储在物理内存中
   - 理解了TLB的作用和刷新时机

3. **提升了调试能力**
   - 学会了使用GDB调试内核代码
   - 掌握了通过插入调试输出定位问题

```
    - 学会了阅读RISC-V特权级规范查找细节

### 改进方向

1. **错误处理**：当前很多函数遇到错误直接panic，可以改为返回错误码
2. **内存管理**：空闲链表效率较低，可以考虑伙伴系统
3. **调试支持**：可以实现更完善的页表打印、内存统计等调试工具
```

附录：报告写作检查清单

提交前，请检查你的报告是否满足以下要求：

内容完整性

- [ ] 五个部分都已包含
- [ ] 关键代码有注释说明
- [ ] 测试结果有截图或输出
- [ ] 问题分析有根本原因

表达清晰性

- [ ] 技术术语使用准确
- [ ] 图表清晰易懂
- [ ] 逻辑连贯，无跳跃
- [ ] 代码格式正确（语法高亮）

内容充实性

- [ ] 不是简单复制代码
- [ ] 有自己的分析和理解
- [ ] 与xv6有明确对比
- [ ] 思考题有认真回答

篇幅合理性

- [ ] 报总页数不超过60页A4纸（Markdown渲染后）
- [ ] 各部分比例合理
- [ ] 无大段无关内容
- [ ] 无冗余重复

常见问题 FAQ

Q1: 报告应该写多长?

A: 没有固定长度要求，但建议尽量控制在60页A4纸（Markdown渲染为PDF后）。重点是内容的质量，不是篇幅的长度。保持每个实验的内容完整性和清晰性。

Q2: 是否需要粘贴所有代码?

A: 不需要。完整代码应该通过Git仓库提交。报告中只需要展示关键代码片段（一般每个实验2-3个核心函数即可），并加上必要的注释说明。

Q3: 截图应该包含什么?

A: 截图应该清晰地显示测试结果。建议截取终端窗口，而不是整个桌面。每张截图下方应有文字说明，解释这是在测试什么。

Q4: 如何体现"理解"而不是"抄袭"?

A: 关键在于：

- 用自己的话解释原理
- 对比xv6说明异同
- 分析设计的优缺点
- 回答"为什么这样设计"
- 记录自己踩过的坑

Q5: 实验没有完全完成怎么办?

A: 在"实验概述"中如实说明完成情况。在"问题与总结"中分析未完成的原因，以及尝试了哪些解决方法。诚实面对问题比回避更重要。

示例报告模板

为方便同学们上手，提供一个Markdown模板：

```
\```markdown
# 实验X：[实验名称]

**姓名**：张三
**学号**：2021xxxxx
**日期**：2024-xx-xx

## 一、实验概述

### 实验目标
[用1-2句话说明目标]

### 完成情况
- ✅ 任务1
- ✅ 任务2
- ⚠️ 任务3（部分完成，原因...）

### 开发环境
- OS: Ubuntu 22.04
- Toolchain: riscv64-unknown-elf-gcc 12.2.0
- QEMU: 7.2.0

## 二、技术设计

### 系统架构
```

```
[架构图 + 说明]

### 关键数据结构
[代码 + 解释]

### 核心流程
[流程图 + 说明]

## 三、实现细节

### 关键函数1
[代码 + 注释 + 说明]

### 难点突破
[问题 + 解决方案]

## 四、测试与验证

### 功能测试
[测试用例 + 结果 + 截图]

### 边界测试
[特殊情况测试]

## 五、问题与总结

### 问题1
**现象**：[具体描述]
**原因**：[分析]
**解决**：[方法]

### 实验收获
[真实的理解和收获]

### 改进方向
[不足之处和优化思路]
\```
```