实验 1：RISC-V 引导与裸机启动

实验目标

通过参考 xv6 的启动机制，理解并实现最小操作系统的引导过程，最终在 QEMU 中输出"Hello OS"。

核心学习资料

xv6 关键文件分析

- kernel/entry.S - 启动汇编代码
  - 重点理解：栈设置、BSS 清零、跳转到 C 代码
  - 思考：为什么需要设置栈？栈应该多大？
- kernel/kernel.ld - 链接脚本
  - 重点理解：入口点设置、段的组织、符号定义
  - 思考：各个段的作用和排列顺序
- kernel/uart.c - 串口驱动
  - 重点理解：UART 寄存器操作、字符输出实现
  - 思考：如何简化为最小实现？

RISC-V 启动机制

- 特权级规范第 3.1 节：机器模式启动状态
- QEMU virt 平台：内存布局和设备地址

bash

```
# 查看 QEMU 设备树
qemu-system-riscv64 -machine virt,dumpdtb=virt.dtb -nographic
dtc -I dtb -O dts virt.dtb | grep -A5 -B5 "uart\|memory"
```

任务列表

任务 1：理解 xv6 启动流程

学习方法：

1. 阅读 kernel/entry.S，回答：
   - 为什么第一条指令是设置栈指针？
   - la sp, stack0 中的 stack0 在哪里定义？
   - 为什么要清零 BSS 段？
   - 如何从汇编跳转到 C 函数？
2. 分析 kernel/kernel.ld，思考：
   - ENTRY(_entry) 的作用是什么？
   - 为什么代码段要放在 0x80000000？
   - etext、edata、end 符号有什么用途？

深入思考：

- xv6 支持多核，你的单核系统可以如何简化？
- xv6 的内存管理很复杂，最小系统需要哪些部分？

任务 2：设计最小启动流程

设计要求：

1. 绘制你的启动流程图
2. 确定内存布局方案
3. 列出必需的硬件初始化步骤

关键问题：

- 栈应该放在内存的哪个位置？需要多大？
- 是否需要清零 BSS 段？为什么？
- 最简单串口输出需要配置哪些寄存器？

任务 3：实现启动汇编代码

参考 xv6 实现思路，但要大幅简化：

实现步骤：

1. 创建 kernel/entry.S
2. 设置入口点和栈指针
3. 清零 BSS 段（如果需要）
4. 跳转到 C 主函数

调试检查点：

```
asm
# 在关键位置插入调试代码
_start:
    li t0, 0x10000000    # UART 基地址
    li t1, 'S'    # 启动标记
    sb t1, 0(t0)    # 输出字符 S 表示启动

# 设置栈后再输出一个字符验证
la sp, stack_top
li t1, 'p'    # 栈设置完成标记
sb t1, 0(t0)
```

任务 4：编写链接脚本

参考 xv6 的基本结构，简化复杂部分：

设计考虑：

1. 确定起始地址（通常是 0x80000000）
2. 组织代码段、数据段、BSS 段
3. 定义必要的符号供 C 代码使用

验证方法：

```
bash
# 编译后检查内存布局
riscv64-unknown-elf-objdump -h kernel.elf
riscv64-unknown-elf-nm kernel.elf | grep -E "(start|end|text)"
```

任务 5：实现串口驱动

参考 xv6 的 uart.c，实现最小功能：

学习要点：

1. UART 16550 的基本寄存器：
    - THR (Transmit Holding Register): 0x10000000
    - LSR (Line Status Register): 0x10000005
2. 输出一个字符的完整流程
3. 为什么需要检查 LSR 的 THRE 位？

实现策略：

```
c
```

```
// 先实现最基本的字符输出
void uart_putc(char c);

// 成功后实现字符串输出
void uart_puts(char *s);
```

调试建议：

- 先在汇编中直接写 UART 验证硬件工作
- 再在 C 函数中实现相同功能
- 最后实现完整的字符串输出

任务 6：完成 C 主函数

设计考虑：

- 函数名可以不是 main，与链接脚本保持一致
- 程序结束后应该做什么？死循环还是关机？
- 如何防止程序意外退出导致系统重启？

调试策略

分阶段调试法

1. 硬件验证阶段：在汇编中直接写 UART
2. 启动验证阶段：验证能跳转到 C 函数
3. 功能验证阶段：实现完整的 Hello 输出

常见问题诊断

- **问题：QEMU 启动后无任何输出**
  - 检查链接脚本的起始地址
  - 验证 UART 基地址是否正确
  - 确认程序是否被正确加载
- **问题：输出乱码或不完整**
  - 检查 UART 初始化是否充分
  - 验证字符发送间隔是否太快
  - 确认字符串是否正确终止

GDB 调试技巧

```
bash
# 启动调试环境
# 终端 1:
make qemu-gdb
# 终端 2:
gdb-multifarch kernel/kernel.elf
(gdb) target remote :1234
(gdb) b _start
(gdb) c
(gdb) layout asm
(gdb) si  # 单步执行汇编
```

思考题

1. 启动栈的设计：
   - 你如何确定栈的大小？考虑哪些因素？
   - 如果栈溢出会发生什么？如何检测栈溢出？

2. BSS 段清零：
    - 写一个全局变量，不清零 BSS 会有什么现象？
    - 哪些情况下可以省略 BSS 清零？
3. 与 xv6 的对比：
    - 你的实现比 xv6 简化了哪些部分？
    - 这些简化在什么情况下会成为问题？
4. 错误处理：
    - 如果 UART 初始化失败，系统应该如何处理？
    - 如何设计一个最小的错误显示机制？