

# 武汉大学计算机学院

## 本科生课程设计报告

### 实验一：RISC-V 引导与裸机启动

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

二〇二五年九月

# 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. 9. 29

## 摘 要

本次实验的核心目标是深入理解并亲手实现一个最小化操作系统的引导过程。实验参考 MIT 的 `xv6-riscv` 项目并进行大幅简化, 聚焦于最核心的启动步骤。实验设计遵循从底层硬件初始化到上层应用逻辑的构建原则, 内容主要包括: 分析 `xv6` 启动流程、设计链接脚本与内存布局、编写汇编启动代码以设置运行环境、实现串口驱动作为输出通道, 并在 C 语言环境中完成 BSS 段清零及主循环逻辑。最终, 在 QEMU 模拟的 RISC-V 硬件平台上, 通过自行编写的汇编和 C 代码, 成功引导系统并由串口打印出启动信息, 并实现了交互式回显功能。

实验结论为成功构建了一个能够正确引导、具备基本输入输出能力的最小化 RISC-V 操作系统内核, 为后续操作系统核心功能的开发奠定了坚实基础。

**关键词:** RISC-V; 操作系统; 引导启动; 裸机编程; QEMU

# 目录

|                                      |    |
|--------------------------------------|----|
| 1 实验目的和意义 .....                      | 5  |
| 1.1 实验目的 .....                       | 5  |
| 1.2 实验意义 .....                       | 5  |
| 2 实验准备 .....                         | 5  |
| 2.1 阅读 kernel/entry.S, 回答以下问题: ..... | 5  |
| 2.2 分析 kernel/kernel.ld, 思考: .....   | 6  |
| 3 实验原理与设计 .....                      | 7  |
| 3.1 RISC-V 启动流程图 .....               | 7  |
| 3.2 内存布局设计 .....                     | 8  |
| 3.3 关键硬件初始化 .....                    | 9  |
| 4 实验步骤与实现 .....                      | 9  |
| 4.1 项目结构设计与创建 .....                  | 9  |
| 4.2 链接脚本实现 (kernel.ld) .....         | 9  |
| 4.3 汇编启动代码实现 (entry.S) .....         | 10 |
| 4.4 串口驱动实现 (uart.h & uart.c) .....   | 11 |
| 4.5 主启动逻辑实现 (start.c) .....          | 12 |
| 4.6 构建系统实现 (Makefile) .....          | 13 |
| 5 实验测试与结果 .....                      | 15 |
| 5.1 编译及运行结果 .....                    | 15 |
| 5.2 功能验证 .....                       | 15 |
| 6 思考题 .....                          | 15 |
| 6.1 启动栈的设计 .....                     | 15 |
| 6.2 BSS 段清零 .....                    | 16 |
| 6.3 与 xv6 的对比 .....                  | 16 |
| 6.4 错误处理 .....                       | 17 |
| 7 实验总结 .....                         | 18 |

# 1 实验目的和意义

## 1.1 实验目的

本实验旨在通过深入分析 xv6 操作系统的 printf 实现机制，独立设计并实现一个功能完整的内核级格式化输出系统。具体目标包括理解内核输出的特殊性，掌握内核为何不能依赖用户态库函数；实现支持%d、%x、%s、%c、%%等基本格式符的格式化输出功能；基于 ANSI 转义序列实现终端清屏、清行与光标精确定位功能；扩展支持多颜色文本输出；构建清晰的硬件驱动层与格式化层的分层架构。通过从底层硬件操作到上层增强功能的完整实现过程，深入理解操作系统内核输出系统的设计原理与实现方法。

## 1.2 实验意义

通过实现内核级 printf、清屏、清行、光标定位和颜色输出等完整功能，能够深入理解操作系统内核输出机制的设计原理，掌握从硬件寄存器操作到上层增强功能的完整开发流程。实验不仅培养了底层编程能力和系统调试能力，更让人体会到内核开发与用户态编程的本质差异——内核必须自给自足，所有功能都需要从最底层的硬件操作开始构建。这种从零搭建包含颜色控制、光标定位等增强功能的系统组件实践经验，为后续学习进程管理、内存管理等更复杂的操作系统机制奠定了坚实基础，培养了从系统全局视角分析问题与解决问题的综合能力。

# 2 实验准备

在正式开始实验之前，首先完成“任务 1：理解 xv6 启动流程”，即阅读理解 xv6 的相关代码，理解其设计的核心思想。

## 2.1 阅读 kernel/entry.S，回答以下问题：

### 2.1.1 为什么第一条指令是设置栈指针？

因为栈是 C 语言函数调用能够正常工作的前提。entry.S 的最后一条指令是 call start，这将跳转到 C 语言编写的 start() 函数。在执行这条指令之前，必须已经设置好一个有效的栈空间，否则 start 函数及其内部调用的所有其他函数都将无法正确执行（无法保存返回地址、参数和局部变量），导致立即崩溃

或行为不可预测。

### 2. 1. 2 `la sp, stack0` 中的 `stack0` 在哪里定义？

在 `kernel/start.c` 中的如下位置定义，这表示为每个 CPU 核心分配了 4KB（4096 字节）的栈空间：

```
// entry.S needs one stack per CPU.  
__attribute__((aligned (16))) char stack0[4096 * NCPU];
```

图 1 `stack0` 定义

### 2. 1. 3 为什么要清零 BSS 段？

为了确保所有未初始化的全局变量和静态变量具有确定的初始值（零）。根据 C 语言标准，BSS 段的变量在程序开始时必须被初始化为零。然而，为了节省磁盘空间，编译器不会在内核镜像文件中为这些零值分配空间。因此，内核自己必须在跳转到 C 代码之前，显式地将这一整块 BSS 内存清零。如果不做这一步，这些全局变量的值将是随机的，导致程序行为不可预测。

### 2. 1. 4 如何从汇编跳转到 C 函数？

通过 RISC-V 的 `call` 指令。汇编代码中的 `call start` 首先会将下一条指令的地址存入返回地址寄存器 `ra`，接着将 `pc` 设置为函数 `start` 的地址。

## 2.2 分析 `kernel/kernel.ld`，思考：

### 2. 2. 1 `ENTRY(entry)` 的作用是什么？

指定程序执行的入口点。当 QEMU 使用 `-kernel` 选项加载内核时，CPU 会直接从 `_entry` 标签处的指令开始执行。

### 2. 2. 2 为什么代码段要放在 `0x80000000`？

1. 硬件约定：`0x80000000` 是 RISC-V 架构中操作系统内核的标准加载地址，QEMU 的 `-kernel` 选项默认将内核镜像加载到这个地址。
2. 技术原因：
  - ① 地址空间布局：RISC-V 将低地址空间（如 `0x0-0x80000000`）保留给用户程序或特定用途，内核放在高地址。
  - ② 内存映射：`0x80000000` 通常是 DRAM 的起始地址或内核镜像的固定

加载点。

③ 符号扩展：这个地址在符号扩展后仍然是正数，避免地址计算问题。

### 2. 2. 3 `etext`、`edata`、`end` 符号有什么用途？

1. `etext`：代码段结束地址。
2. `edata`：已初始化数据段结束地址。
3. `end`：所有已分配内存的结束地址。

## 3 实验原理与设计

### 3.1 RISC-V 启动流程图

参照 xv6 的启动设计，绘制的简化启动流程图如下：

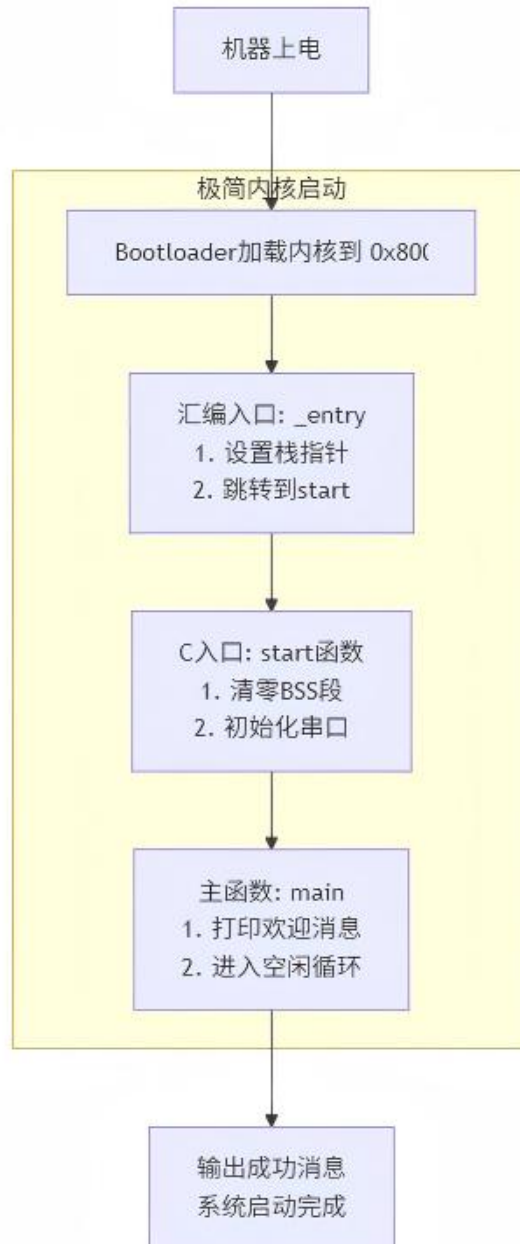


图 2 RISC-V 启动流程图

## 3.2 内存布局设计

如下图:





图 3 内存布局示意图

### 3.3 关键硬件初始化

1. 栈指针设置：为 C 代码执行准备环境。
2. BSS 段清零：确保未初始化变量值为 0。
3. 串口初始化：建立调试输出通道。

## 4 实验步骤与实现

### 4.1 项目结构设计与创建

参考 xv6 的项目结构并进行简化，本次实验的项目结构设计如下：

```
lab1/  
├─ Makefile  
└─ kernel/  
    ├─ entry.S  
    ├─ start.c  
    ├─ kernel.ld  
    ├─ uart.c  
    └─ uart.h
```

图 4 lab1 项目结构

### 4.2 链接脚本实现（kernel.ld）

链接脚本是内核启动的基础，它定义了内核在内存中的精确布局。考虑到

RISC-V 架构的特定要求，将入口地址设置为 0x80000000，这是 QEMU 加载内核的标准地址。实现的关键设计决策：

1. 入口点明确：通过 ENTRY(\_entry) 指定汇编入口函数，确保 CPU 从正确位置开始执行。
2. 段顺序固定：严格按照 .text->.rodata->.data->.bss 的顺序排列，符合程序执行的逻辑流程。
3. 符号导出：定义 \_bss\_start 和 \_end 符号，为 BSS 段清零操作提供准确的地址范围。

```
1  /* kernel.ld */
2  OUTPUT_ARCH("riscv")
3  ENTRY(_entry)
4  SECTIONS {
5      . = 0x80000000;
6      .text : {
7          *(.text .text.*)
8      }
9      .rodata : {
10         *(.rodata .rodata.*)
11     }
12     .data : {
13         *(.data .data.*)
14     }
15     .bss : {
16         _bss_start = .;
17         *(.bss .bss.*)
18         _end = .;
19     }
20 }
```

图 5 kernel.ld 源代码

### 4.3 汇编启动代码实现 (entry.S)

汇编启动代码是内核执行的第一段代码，承担着搭建 C 语言运行环境的关键任务。实现过程分为两个主要阶段：

1. 栈指针设置阶段：
  - 计算栈的起始位置：sp = \_end + 4096。
  - 为每个 CPU 核心预留独立的栈空间，避免多核环境下的竞争条件。

- 确保栈指针按页面对齐，满足后续分页机制的要求。

## 2. 环境切换阶段：

- 使用 `call` 指令跳转到 C 函数，自动保存返回地址。
- 设置安全兜底机制：在 `start` 函数返回后进入无限循环，防止执行未知内存代码。

```

1  # kernel/entry.S
2  .section .text
3  .global _entry
4  _entry:
5      # 设置栈指针（在_end后分配4KB空间）
6      la sp, _end
7      li t0, 4096
8      add sp, sp, t0
9      # 跳转到C入口函数
10     call start
11     spin:
12     j spin

```

图 6 entry.S 源代码

## 4.4 串口驱动实现（uart.h & uart.c）

串口驱动是内核与外界通信的唯一通道，其实现遵循了硬件编程的最佳实践。

### 1. 寄存器映射设计：

- 使用内存映射 IO 方式访问硬件寄存器。
- 通过基地址+偏移量的方式定义寄存器常量，提高代码可读性。
- 使用 `volatile` 关键字确保编译器不优化硬件访问操作。

### 2. 功能模块划分：

- 初始化函数：配置串口工作模式（8N1 格式），启用 FIFO 缓冲。
- 发送功能：实现阻塞式字符发送，确保数据传输的可靠性。
- 接收功能：提供轮询式字符接收，支持实时输入处理。
- 状态检测：通过 LSR 寄存器判断设备状态，实现非阻塞检测。

```

1 // kernel/uart.c
2 #include "uart.h"
3
4 // UART寄存器基地址 (QEMU virt机器的默认地址)
5 #define UART_BASE 0x10000000
6
7 // 寄存器偏移量
8 #define UART_RHR 0 // 接收保持寄存器 (读)
9 #define UART_THR 0 // 发送保持寄存器 (写)
10 #define UART_IER 1 // 中断使能寄存器
11 #define UART_FCR 2 // FIFO控制寄存器
12 #define UART_LCR 3 // 线路控制寄存器
13 #define UART_LSR 5 // 线路状态寄存器
14
15 // 寄存器访问宏
16 #define REG(r) (*(volatile unsigned char *)(UART_BASE + r))
17
18 // LSR寄存器位定义
19 #define LSR_RX_READY (1 << 0) // 接收数据就绪
20 #define LSR_TX_READY (1 << 5) // 发送保持寄存器空
21
22 void uart_init(void) {
23     // 禁用中断
24     REG(UART_IER) = 0x00;
25
26     // 设置线路控制: 8位数据, 无奇偶校验, 1位停止位
27     REG(UART_LCR) = 0x03;
28
29     // 启用FIFO, 清空FIFO
30     REG(UART_FCR) = 0x01;
31 }
32
33 void uart_putc(char c) {
34     // 等待发送缓冲区为空
35     while ((REG(UART_LSR) & LSR_TX_READY) == 0);
36
37     // 发送字符
38     REG(UART_THR) = c;
39
40     // 如果发送的是换行符, 额外发送回车符 (\r)
41     if (c == '\n') {
42         uart_putc('\r');
43     }
44 }
45
46 void uart_puts(const char *s) {
47     while (*s) {
48         uart_putc(*s++);
49     }
50 }
51
52 char uart_getc(void) {
53     // 等待直到有数据可读
54     while ((REG(UART_LSR) & LSR_RX_READY) == 0);
55
56     // 读取字符
57     return REG(UART_RHR);
58 }
59
60 int uart_has_input(void) {
61     // 检查是否有数据可读
62     return (REG(UART_LSR) & LSR_RX_READY) != 0;
63 }

```

图 7 uart.c 源代码

## 4.5 主启动逻辑实现 (start.c)

主启动逻辑是整个内核的控制中心, 按照严格的顺序执行初始化任务。

### 1. BSS 段清零阶段:

- 获取 BSS 段的准确边界地址 (由链接脚本提供)。
- 使用循环逐字节清零, 确保所有未初始化变量具有确定值。
- 此操作必须在任何代码访问全局变量之前完成。

### 2. 硬件初始化阶段:

- 首先初始化串口, 建立调试输出能力。
- 按照依赖关系顺序初始化各硬件模块。
- 每个初始化步骤都包含状态输出, 便于调试。

### 3. 主循环设计:

- 实现简单的回显功能, 验证系统基本功能。
- 采用非阻塞式输入检测, 提高系统响应性。
- 为后续的功能扩展预留了接口。

其中, start.c 中还额外实现了交互式回显功能, 实时进行输入检测并进行字符级回显。

```

1 // kernel/start.c
2 #include "uart.h"
3
4 extern char _bss_start[], _end[];
5
6 void clear_bss(void) {
7     for (char *p = _bss_start; p < _end; p++) {
8         *p = 0;
9     }
10 }
11
12 void start(void) {
13     // 清零BSS段
14     clear_bss();
15
16     // 初始化串口
17     uart_init();
18
19     // 打印启动信息
20     uart_puts("\n=====\\n");
21
22     uart_puts("This is lab1!\\n");
23     uart_puts("Hello!\\n");
24     uart_puts("=====\\n\\n");
25
26     // 简单的回显测试
27     uart_puts("Type something (echo test): ");
28
29     // 主循环：回显用户输入
30     while (1) {
31         if (uart_has_input()) {
32             char c = uart_getc();
33             uart_putc(c); // 回显
34
35             // 如果收到回车，换行
36             if (c == '\\r') {
37                 uart_putc('\\n');
38             }
39         }
40     }

```

图 8 start.c 源代码

## 4.6 构建系统实现 (Makefile)

参照 xv6 的 makefile 文件，删去了冗余部分，仅保留实验一相关的核心内容。关键部分包括工具链自动检测、编译选项设置（如-ffreestanding）、链接脚本的使用以及 QEMU 运行配置。

```

1 K=kernel
2
3 # 最小内核所需的目标文件
4 OBJS = \
5     $K/entry.o \
6     $K/start.o \
7     $K/uart.o
8
9 # 工具链前缀（自动检测）
10 ifndef TOOLPREFIX
11 TOOLPREFIX := $(shell if riscv64-unknown-elf-objdump -i 2>&1 | grep 'elf64-big' >/dev/null 2>&1; \
12     then echo 'riscv64-unknown-elf-'; \
13     elif riscv64-linux-gnu-objdump -i 2>&1 | grep 'elf64-big' >/dev/null 2>&1; \
14     then echo 'riscv64-linux-gnu-'; \
15     else echo ""; fi)
16 endif
17
18 # QEMU命令
19 QEMU = qemu-system-riscv64
20
21 # 编译工具
22 CC = $(TOOLPREFIX)gcc
23 LD = $(TOOLPREFIX)ld
24 OBJCOPY = $(TOOLPREFIX)objcopy
25 OBJDUMP = $(TOOLPREFIX)objdump
26
27 # 编译选项
28 CFLAGS = -Wall -O -fno-omit-frame-pointer -ggdb
29 CFLAGS += -MD
30 CFLAGS += -mcmodel=medany
31 CFLAGS += -ffreestanding -fno-common -nostdlib
32 CFLAGS += -I.
33 CFLAGS += -fno-stack-protector
34
35 # 链接选项
36 LDFLAGS = -z max-page-size=4096
37
38 # 主要目标：生成内核可执行文件
39 $K/kernel: $(OBJS) $K/kernel.ld
40     $(LD) $(LDFLAGS) -T $K/kernel.ld -o $K/kernel $(OBJS)
41     $(OBJDUMP) -S $K/kernel > $K/kernel.asm
42     $(OBJDUMP) -t $K/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $K/kernel.sym
43     @echo "=== 内核编译完成 ==="
44
45 # 汇编文件编译规则
46 $K/%.o: $K/%.S
47     $(CC) -g -c -o $@ $<
48
49 # C文件编译规则
50 $K/%.o: $K/%.c
51     $(CC) $(CFLAGS) -c -o $@ $<
52
53 # QEMU运行配置
54 QEMUOPTS = -machine virt -bios none -kernel $K/kernel -m 128M -nographic
55
56 # 运行内核
57 run: $K/kernel
58     $(QEMU) $(QEMUOPTS)
59
60 # 清理生成文件
61 clean:
62     rm -f $K/*.o $K/*.d $K/*.asm $K/*.sym $K/kernel
63
64 # 包含依赖文件
65 -include $K/*.d
66
67 .PHONY: all run clean

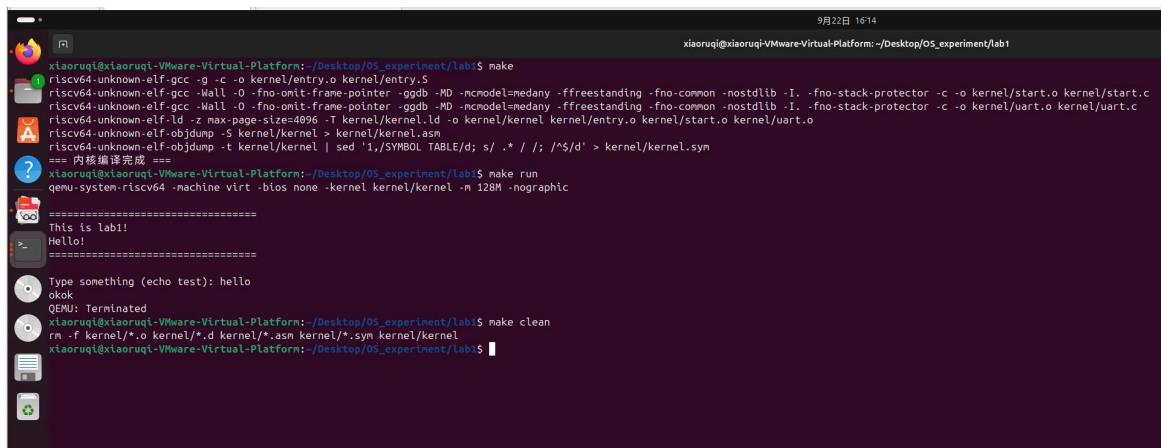
```

图 9 Makefile 源代码

## 5 实验测试与结果

### 5.1 编译及运行结果

终端输入 make 指令进行编译，之后输入 make run 运行，屏幕成功显示：This is lab1! Hello! 并提示 Type something 允许用户输入字符实时回显。运行截图如下：



```
xiaoruqi@xiaoruqi-Virtual-Platform:~/Desktop/OS_experiment/lab1$ make
riscv64-unknown-elf-gcc -g -c -o kernel/entry.o kernel/entry.S
riscv64-unknown-elf-gcc -Wall -O -fno-omit-frame-pointer -ggdb -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -I. -fno-stack-protector -c -o kernel/start.o kernel/start.c
riscv64-unknown-elf-gcc -Wall -O -fno-omit-frame-pointer -ggdb -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -I. -fno-stack-protector -c -o kernel/uart.o kernel/uart.c
riscv64-unknown-elf-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel kernel/entry.o kernel/start.o kernel/uart.o
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /$/d' > kernel/kernel.sym
=== 内核编译完成 ===
xiaoruqi@xiaoruqi-Virtual-Platform:~/Desktop/OS_experiment/lab1$ make run
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -nographic

=====
This is lab1!
Hello!
=====
Type something (echo test): hello
okok
QEMU: Terminated
xiaoruqi@xiaoruqi-Virtual-Platform:~/Desktop/OS_experiment/lab1$ make clean
rm -f kernel/*.o kernel/*.d kernel/*.asm kernel/*.sym kernel/kernel
xiaoruqi@xiaoruqi-Virtual-Platform:~/Desktop/OS_experiment/lab1$
```

图 10 运行测试截图

### 5.2 功能验证

- 内核成功从 0x80000000 启动。
- BSS 段正确清零。
- 栈指针正确设置。
- 串口输入输出功能正常。
- 回显功能正常工作。

## 6 思考题

### 6.1 启动栈的设计

#### 6.1.1 你如何确定栈的大小？考虑哪些因素？

本次实验选用 4KB 的典型选择。具体考量了：

1. 函数调用深度：启动阶段调用链较浅。
2. 局部变量大小：避免大数组在栈上分配。
3. 中断上下文：保存寄存器需要~256 字节。

4. 对齐要求：RISC-V 需要 16 字节对齐。
5. 安全边际：预留空间应对意外情况。

### 6. 1. 2 如果栈太小会发生什么？如何检测栈溢出？

如果栈太小可能导致栈溢出，最终致使数据损坏、代码崩溃等。  
可以利用栈边界检查检测栈溢出。如以下代码：

```
1  uint64 stack_top = STACK_BASE;
2  uint64 stack_bottom = STACK_BASE - STACK_SIZE;
3
4  void stack_check(void) {
5      uint64 current_sp;
6      asm volatile("mv %0, sp" : "=r"(current_sp));
7
8      if (current_sp < stack_bottom) {
9          uart_puts("STACK OVERFLOW!\n");
10         while(1); // 停机
11     }
12 }
```

图 11 检测栈溢出示例代码

## 6.2 BSS 段清零

### 6. 2. 1 写一个全局变量，不清零 BSS 会有什么现象？

可能会有以下现象：

1. 变量值取决于内存之前的残留数据。
2. 程序行为不可预测，难以调试。
3. 经典“今天正常，明天崩溃”。

### 6. 2. 1 哪些情况下可以省略 BSS 清零？

1. 引导加载器已清零：某些 bootloader 会清理内存。
2. 硬件保证：某些嵌入式芯片上电后内存自动清零。
3. 安全环境：内存之前未被使用（如模拟器冷启动）。
4. 不使用 BSS 变量：所有变量都显式初始化。

## 6.3 与 xv6 的对比



### 6. 3. 1 你的实现比 xv6 简化了哪些部分？

表 1 本项目与 xv6 的对比

| 模块   | xv6 实现       | 我的简化实现    |
|------|--------------|-----------|
| 进程管理 | 完整 PCB、调度器   | 单任务，无进程   |
| 内存管理 | 多级页表、kmalloc | 物理页分配器    |
| 文件系统 | inode、日志、缓存  | 无文件系统     |
| 设备驱动 | 控制台、磁盘、时钟    | 仅 UART 驱动 |
| 系统调用 | 陷入机制、权限检查    | 直接函数调用    |
| 并发控制 | 自旋锁、睡眠锁      | 禁用中断      |

### 6. 3. 2 这些简化在什么情况下会成为问题？

比如在需要运行用户程序时，由于我的系统只能运行内核代码，因此无法加载和执行外部程序；此外，需要调试时，我的系统只能简单提示，没有 xv6 那样完整的 panic 和 backtrace。等等。

## 6.4 错误处理

### 6. 4. 1 如果 UART 初始化失败，系统应该如何处理？

系统的错误处理参见以下代码：

```
1  int uart_init(void) {
2      // 尝试访问UART寄存器
3      uint8_t test_value = 0xAA;
4      REG(UART_THR) = test_value;
5      // 检查是否可写
6      if (REG(UART_THR) != test_value) {
7          return -1; // 初始化失败
8      }
9      // 配置寄存器
10     REG(UART_LCR) = 0x03;
11     // 验证配置
12     if (REG(UART_LCR) != 0x03) {
13         return -2; // 配置失败
14     }
15     return 0; // 成功
16 }
17 void start(void) {
18     if (uart_init() != 0) {
19         // UART不可用，进入最小错误模式
20         emergency_halt();
21     }
22     // 正常启动流程...
23 }
```

图 12 uart 初始化代码（含错误处理）

#### 6. 4. 2 如何设计一个最小的错误显示机制？

可以使用内存标记法，在固定内存地址记录错误信息，如下：

```
1  #define ERROR_ADDR 0x8000FF00
2
3  void record_error(int code, const char *msg) {
4      volatile uint32_t *error_code = (uint32_t*)ERROR_ADDR;
5      volatile char *error_msg = (char*)(ERROR_ADDR + 4);
6
7      *error_code = code;
8      for (int i = 0; msg[i] && i < 60; i++) {
9          error_msg[i] = msg[i];
10     }
11     error_msg[60] = '\0';
12 }
```

图 13 内存标记法记录错误信息示例代码

## 7 实验总结

本次实验成功实现了一个基于 RISC-V 架构的最小化操作系统内核启动流程。通过精心设计的代码结构和模块化实现，完成了从硬件上电到交互式命令行环境的完整引导过程。实验不仅达到了所有基本要求，还额外实现了交互式回显功能，为后续操作系统开发奠定了坚实基础。

## 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。）