

武汉大学计算机学院

本科生课程设计报告

实验二：内核 **printf** 与清屏功能实现

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

本次实验旨在深入理解操作系统内核输出系统的设计原理，通过分析 xv6 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。实验内容包括理解内核输出特殊性、实现格式化输出、支持清屏清行功能、光标精确定位以及多颜色文本输出，并构建合理的分层架构。通过实现硬件驱动层与格式化层的清晰接口，成功在 QEMU 模拟的 RISC-V 平台上实现了支持 `%d`、`%x`、`%s`、`%c`、`%%` 等格式符的 `printf` 函数，基于 ANSI 转义序列实现了终端清屏、清行、光标控制以及 8 种颜色的文本输出功能。实验结果表明，系统能够稳定运行，输出效果准确美观，具备良好的可扩展性与可维护性，为后续操作系统内核开发提供了重要的输出基础。

关键词：RISC-V；操作系统；内核输出；`printf`；清屏；光标定位；颜色输出

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 理解分层设计:	6
2.3 深入思考:	7
3 实验原理与设计	8
3.1 系统架构图设计	8
3.2 各层接口定义	8
3.3 与 xv6 设计的异同分析	9
4 实验步骤与实现	10
4.1 实验步骤记录	10
4.2 核心关键代码理解总结	10
5 实验测试与结果	12
5.1 基础功能测试	13
5.2 边界情况测试	14
5.3 颜色输出测试	15
5.4 光标控制测试	16
5.5 清屏清行测试	17
6 思考题	19
6.1 架构设计	19
6.2 算法选择	19
6.3 性能优化	20
6.4 错误处理	20
7 实验总结	22

1 实验目的和意义

1.1 实验目的

本实验旨在通过动手实现一个 RISC-V 最小内核的引导过程，深入理解操作系统从硬件上电到内核初始化的完整流程。具体目标是分析 xv6 系统的启动设计，亲自编写汇编与 C 代码完成栈设置、BSS 段清零和串口初始化等核心步骤，最终在 QEMU 模拟器上成功引导内核并实现串口输出，验证裸机启动的正确性。

1.2 实验意义

本实验是理解计算机系统底层工作原理的关键实践。通过裸机编程，能够打通软硬件界限，揭示硬件上电后协同工作直至软件获得控制权的完整过程。这一过程为后续学习进程、内存等复杂操作系统机制奠定了坚实基础，同时培养了从系统全局视角分析问题与实现解决方案的综合能力。

2 实验准备

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

2.1 研读 printf.c 中的核心函数，回答下列问题

2.1.1 printf() 如何解析格式字符串？

xv6 使用状态机方式解析格式字符串，具体解析策略如下：

1. 逐字符扫描，遇到 % 进入格式解析模式
2. 预读后续 2 个字符（c0, c1, c2）进行最长匹配
3. 使用 va_arg(ap, 类型) 提取可变参数

2.1.2 printint() 如何处理不同进制转换？

```
1 | do {  
2 |     buf[i++] = digits[x % base]; // 关键: digits数组索引  
3 | } while((x /= base) != 0);
```

图 1 printf.c 文件部分代码 1

正如以上代码，预定义 `digits[] = "0123456789abcdef"` 包含所有进制字符，再利用 `x % base` 得到当前最低位数字、利用 `x /= base` 移除已处理的最低位，循环直到 `x == 0` 即完成转换。

2. 1. 3 负数处理有什么特殊考虑？

```
1  if(sign && (sign = (xx < 0))) // 检查是否需要符号处理
2      x = -xx;                  // 转为正数处理
3  else
4      x = xx;
5
6  // ... 转换完成后
7  if(sign)
8      buf[i++] = '-';           // 最后添加负号
```

图 2 printf.c 文件部分代码 2

正如以上代码，代码中统一将有符号数转为无符号数处理，避免符号位干扰，转换完成后再统一添加负号。此外，支持 `sign` 参数控制是否进行符号处理（`%u` 不需要）。

2.2 理解分层设计：



图 3 xv6 的分层设计示意图

2. 2. 1 每一层的职责是什么？

1. **格式化层**：负责解析 `%d`、`%s` 等格式符，将数字转换为字符串，处理可变参数。它完成数据到文本的转换，但不关心输出目标。
2. **控制台层**：作为中间抽象层，提供统一的输出接口，可以管理多个输出设备（串口、显示器等），决定字符路由和设备选择。
3. **驱动层**：直接操作硬件寄存器，负责设备初始化、状态监控和数据传输，是软件与硬件的桥梁。
4. **终端/硬件层**：执行最终的字符显示和 ANSI 转义序列解释，如清屏

(\033[2J)、光标控制等物理操作。

2. 2. 2 这种设计有什么优势？

1. 模块化清晰：每层职责单一，修改格式化逻辑不影响硬件驱动，更换硬件只需重写驱动层，大幅提升可维护性。
2. 可移植性强：通过硬件抽象层，同一套 printf 代码可在不同平台(RISC-V、ARM 等)和设备(串口、显示器)上运行。
3. 扩展灵活：添加新输出设备只需在控制台层注册，无需修改上层代码，支持输出重定向和多设备并行输出。
4. 健壮性高：错误被隔离在单层内，驱动故障不会导致系统崩溃，可实现优雅降级(如串口故障改用网络输出)。
5. 便于测试：可逐层测试，用模拟驱动替代真实硬件进行自动化测试，调试时能快速定位问题层级。

2.3 深入思考：

2. 3. 1 xv6 为什么不使用递归进行数字转换？

递归可能存在以下问题：

1. 栈溢出风险：数字转换深度有限(最多 20 位)，但内核栈很小
2. 性能开销：函数调用、参数传递、栈帧分配

2. 3. 2 printint() 中处理 INT_MIN 的技巧是什么？

INT_MIN = -2147483648，-INT_MIN 会溢出。因此，xv6 选择使用 unsigned long long，可以安全容纳 32 位 INT_MIN 的绝对值，避免溢出。

2. 3. 3 如何实现线程安全的 printf？

xv6 通过自旋锁实现线程安全，具体的线程安全机制包含如下内容：

1. 互斥访问：防止多个 CPU 同时调用 printf 导致输出交错
2. panic 特殊处理：panic 时不加锁，避免死锁
3. 轻量级：自旋锁比互斥锁更适合内核短临界区

3 实验原理与设计

3.1 系统架构图设计

本实验设计的简化系统架构图如下：

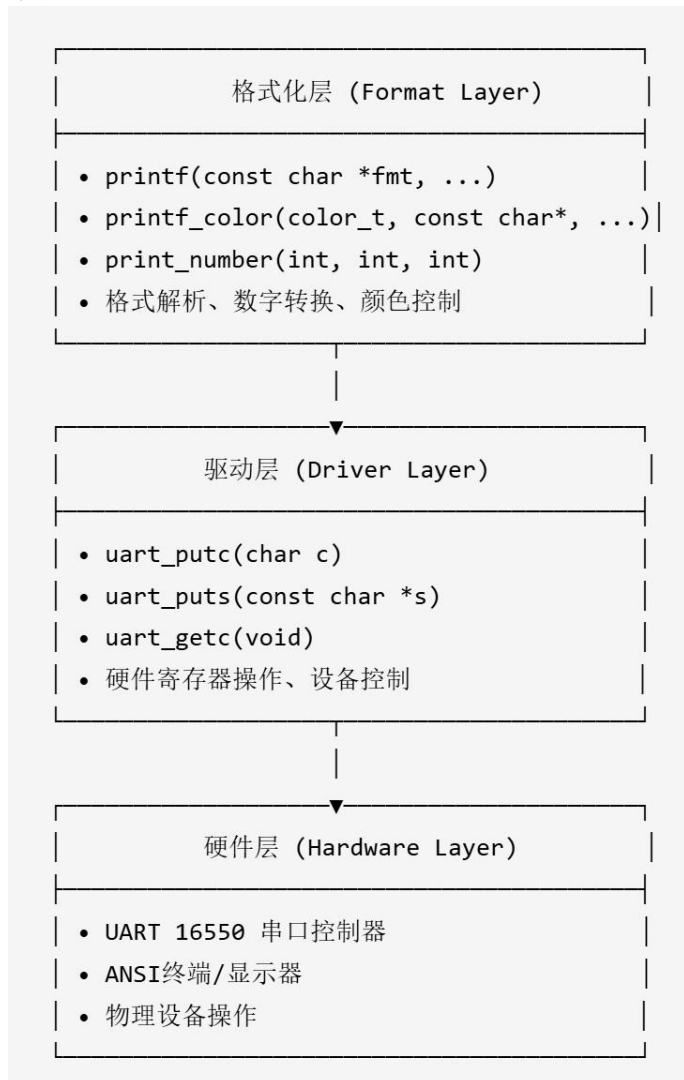


图 4 本实验的系统架构图

3.2 各层接口定义

1. 格式化层接口 (`printf.h`):


```

1 // 基础输出功能
2 int printf(const char *fmt, ...);
3 void print_number(int num, int base, int sign);
4
5 // 增强显示功能
6 void clear_screen(void);
7 void clear_line(void);
8 void goto_xy(int x, int y);
9 int printf_color(color_t color, const char *fmt, ...);
10
11 // 测试控制
12 void wait_for_enter(int x);

```

图 5 格式化层接口

2. 驱动层接口 (uart.h):

```

1 // 设备初始化
2 void uart_init(void);
3
4 // 字符级操作
5 void uart_putc(char c);
6 char uart_getc(void);
7
8 // 字符串级操作
9 void uart_puts(const char *s);
10 int uart_has_input(void);

```

图 6 驱动层接口

3.3 与 xv6 设计的异同分析

● 相同点:

1. 都采用分层架构，分离格式化逻辑与硬件操作
2. 都使用状态机解析格式字符串
3. 都支持基本的格式符(%d, %x, %s, %c, %%)
4. 都采用迭代算法进行数字转换，避免递归栈溢出

● 不同点:

表 1 本实验与 xv6 的不同点

特性	xv6 设计	本实验设计	设计理由
架构层次	三层（格式化+控	两层（格式化+驱	简化设计，减少函

	制台+驱动)	动)	数调用开销
并发安全	自旋锁保护多核 并发	无锁，单核假设	教学环境简化
功能扩展	基础格式化功能	增强显示（颜色+ 光标）	提升用户体验
错误处理	复杂 panic 机制	简单空指针保护	聚焦核心功能
缓冲机制	有输出缓冲	直接字符输出	降低实现复杂度

4 实验步骤与实现

4.1 实验步骤记录

4.1.1 阶段一：基础框架搭建

1. 分析 xv6 源码：深入研究 printf.c 的状态机解析逻辑和 printint 的数字转换算法
2. 设计接口规范：定义清晰的层间接口，确保模块化设计
3. 实现 UART 驱动：完成最基本的字符输出功能，验证硬件通路正常

4.1.2 阶段二：核心功能实现

1. 可变参数处理：使用 stdarg.h 实现 va_list 参数提取，支持不定参数
2. 格式解析状态机：实现 %d、%x、%s 等格式符的识别和处理
3. 数字转换算法：采用迭代逆序法，重点解决 INT_MIN 边界情况

4.1.3 阶段三：增强功能开发

1. ANSI 转义序列：实现 \033[2J 清屏、\033[x;yH 光标定位
2. 颜色输出支持：扩展 printf_color 函数，支持 8 种基本颜色
3. 交互功能完善：添加 wait_for_enter 实现测试流程控制

4.2 核心关键代码理解总结

- 关键代码 1：数字转换算法（解决 INT_MIN 边界）。用无符号运算解决边界溢出，迭代算法保证栈安全。

```

1 void print_number(int num, int base, int sign) {
2     char buf[16];
3     int i = 0;
4     unsigned int unum; // 关键：使用无符号数避免溢出
5     int neg = 0;
6
7     // INT_MIN特殊处理：-(-2147483648)会溢出，用无符号运算
8     if (sign && num < 0) {
9         neg = 1;
10        unum = (unsigned int)(-num); // 无符号转换避免溢出
11    } else {
12        unum = (unsigned int)num;
13    }
14
15    // 迭代算法替代递归，避免内核栈溢出
16    do {
17        buf[i++] = digits[unum % base]; // 逆序存储
18        unum /= base;
19    } while (unum > 0);
20
21    // 逆序输出得到正确顺序
22    while (--i >= 0) {
23        uart_putc(buf[i]);
24    }
25 }

```

图 7 数字转换算法

- **关键代码 2：格式解析状态机。** 用状态机模式实现格式解析，结构清晰，容易扩展。

```

1 int printf(const char *fmt, ...) {
2     va_list ap;
3     va_start(ap, fmt);
4
5     for (int i = 0; fmt[i] != '\0'; i++) {
6         if (fmt[i] != '%') {
7             uart_putc(fmt[i]); // 普通字符状态
8             continue;
9         }
10
11        i++; // 进入格式解析状态
12        switch (fmt[i]) {
13            case 'd':
14                print_number(va_arg(ap, int), 10, 1); // 十进制
15                break;
16            case 'x':
17                print_number(va_arg(ap, int), 16, 0); // 十六进制
18                break;
19            // 其他格式符处理...
20        }
21    }
22    va_end(ap);
23    return 0;
24 }

```

图 8 格式解析状态机

- **关键代码 3：ANSI 清屏与光标控制。** 遵循 ANSI 标准化实现，确保终端具备兼容性。

```

1 void clear_screen(void) {
2     uart_puts("\033[2J"); // ED命令: 清除整个屏幕
3     uart_puts("\033[H");  // CUP命令: 光标归位
4 }
5
6 void goto_xy(int x, int y) {
7     printf("\033[%d;%dH", y + 1, x + 1); // 行列定位(1-based)
8 }
9
10 int printf_color(color_t color, const char *fmt, ...) {
11     printf("\033[3%dm", color); // 设置前景色
12     // 格式化输出...
13     printf("\033[0m");          // 重置属性
14 }

```

图 9 ANSI 清屏与光标控制

- **关键代码 4: 硬件寄存器精确操作。** 状态检查确保传输可靠性, 换行符转换提升兼容性。

```

1 void uart_putc(char c) {
2     while ((REG(UART_LSR) & LSR_TX_READY) == 0); // 等待就绪
3     REG(UART_THR) = c; // 写入发送寄存器
4
5     if (c == '\n') {
6         uart_putc('\r'); // 换行转换: \n -> \r\n
7     }
8 }

```

图 10 硬件寄存器精确操作

5 实验测试与结果

终端输入 make 指令进行编译, 之后输入 make run 运行, 进入如下的开始测试界面:

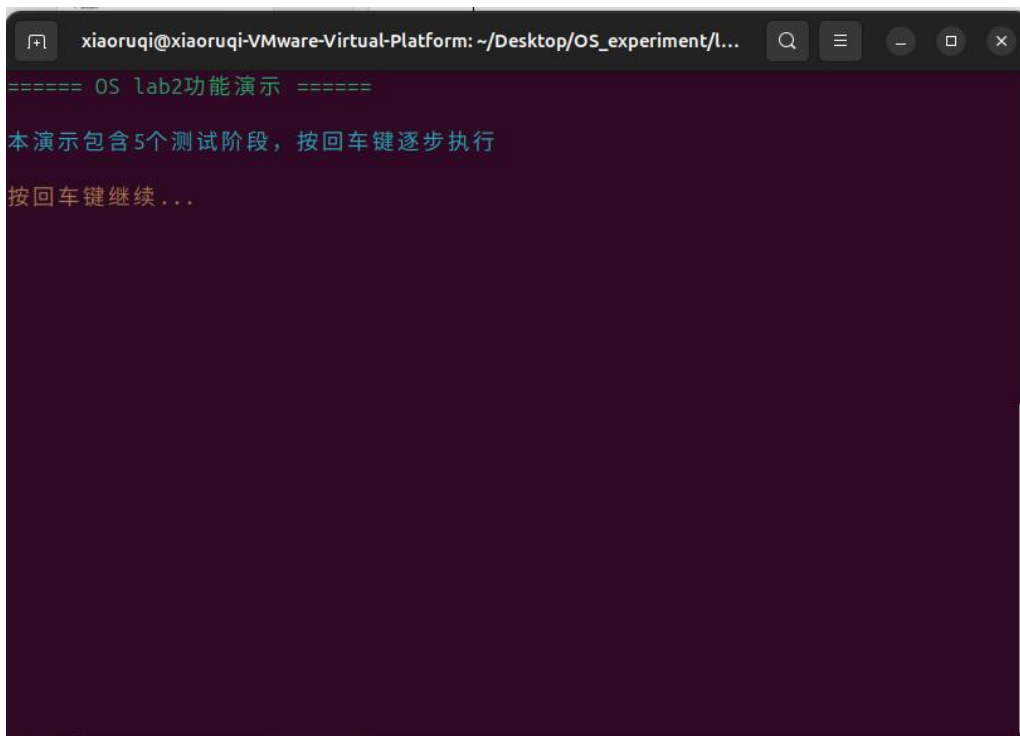
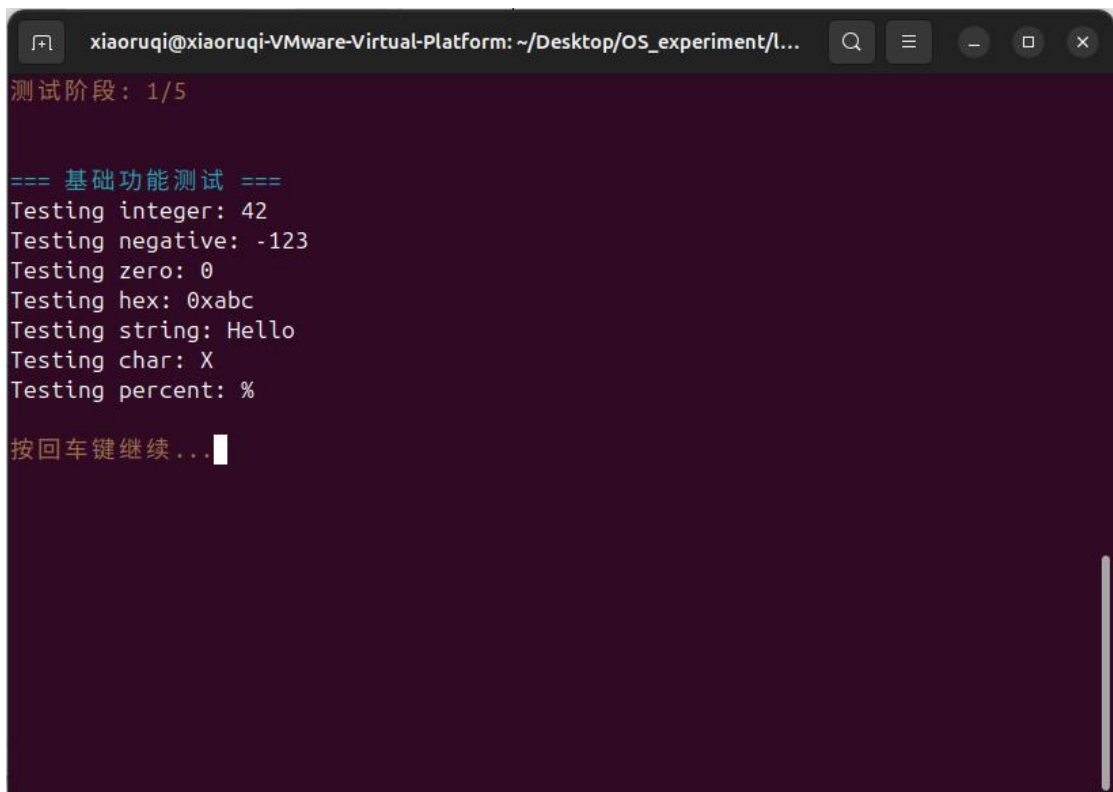


图 11 运行测试初始界面

5.1 基础功能测试

验证 printf 基本格式化功能，包括整数、字符串、字符、十六进制等格式符的正确解析和输出。测试涵盖正数、负数、零值等常规情况。

输出结果如下图：

A terminal window with a dark purple background. The title bar shows the user 'xiaoruqi' and the path '~/Desktop/OS_experiment/l...'. The terminal output is in Chinese and English, showing a series of tests for integer, negative, zero, hex, string, char, and percent. It ends with a prompt to press the enter key to continue.

```
xiaoruqi@xiaoruqi-VMware-Virtual-Platform: ~/Desktop/OS_experiment/l...
测试阶段: 1/5

=== 基础功能测试 ===
Testing integer: 42
Testing negative: -123
Testing zero: 0
Testing hex: 0xabc
Testing string: Hello
Testing char: X
Testing percent: %

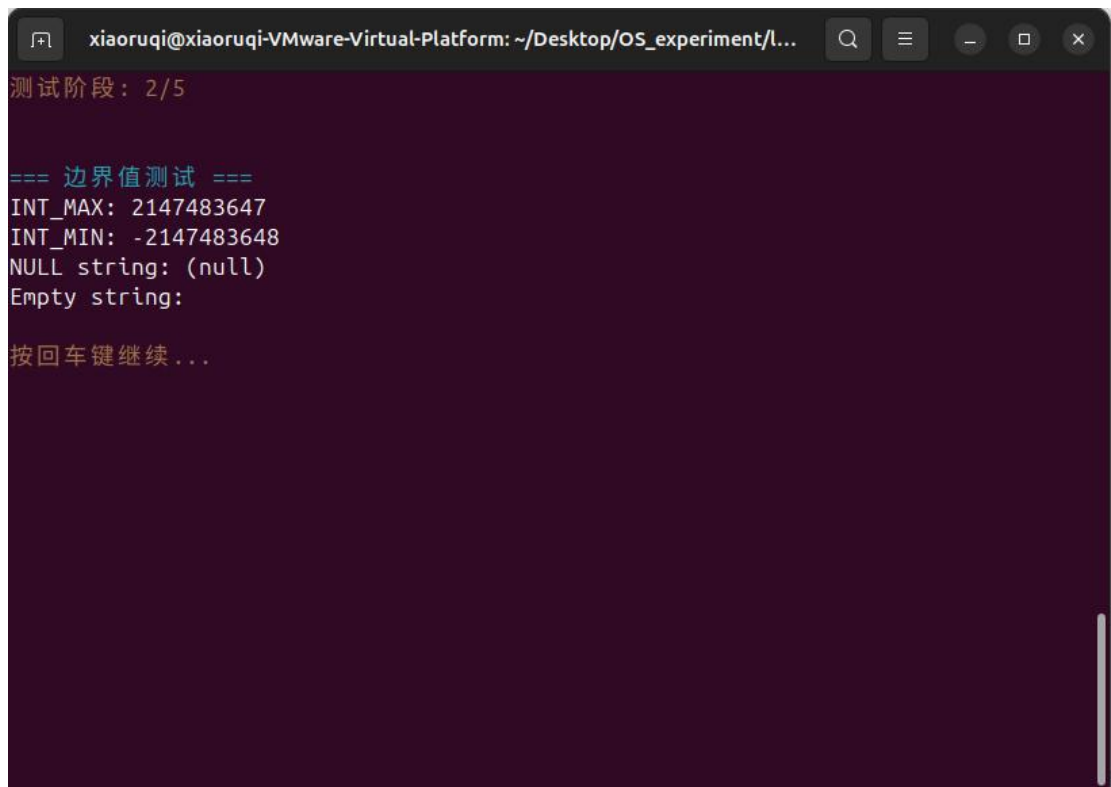
按回车键继续...
```

图 12 测试 1-基础功能测试

5.2 边界情况测试

重点测试系统在极端输入下的稳定性，包括 INT_MAX、INT_MIN 等数值边界，空指针、空字符串等异常输入，以及未知格式符的错误恢复能力。

输出结果如下图：

A terminal window with a dark purple background. The title bar shows the user 'xiaoruqi' and the path '~/Desktop/OS_experiment/l...'. The terminal content includes a status message '测试阶段: 2/5', a section header '=== 边界值测试 ===', and several test results: 'INT_MAX: 2147483647', 'INT_MIN: -2147483648', 'NULL string: (null)', and 'Empty string:'. It ends with a prompt '按回车键继续...'.

```
xiaoruqi@xiaoruqi-VMware-Virtual-Platform: ~/Desktop/OS_experiment/l...
测试阶段: 2/5

=== 边界值测试 ===
INT_MAX: 2147483647
INT_MIN: -2147483648
NULL string: (null)
Empty string:

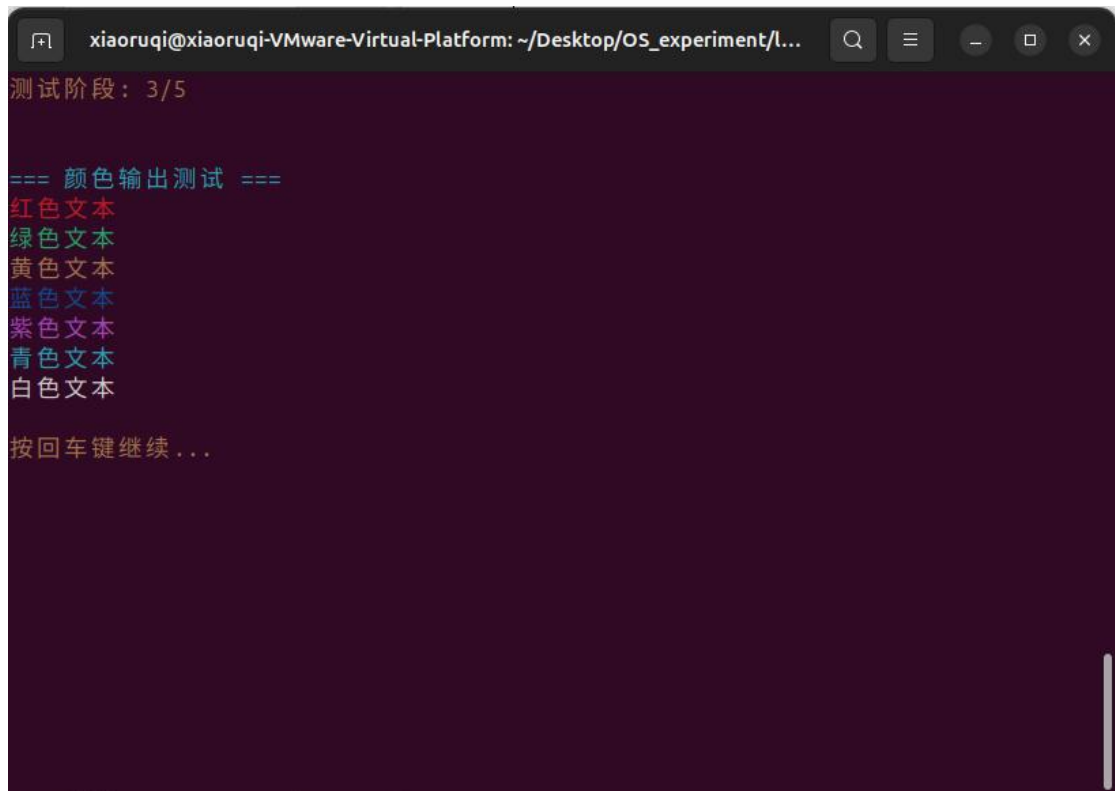
按回车键继续...
```

图 13 测试 2-边界情况测试

5.3 颜色输出测试

验证 ANSI 颜色转义序列的实现效果，测试 8 种基本颜色在终端中的实际显示效果，验证颜色设置和重置功能是否正常。

输出结果如下图，所有颜色正确显示：

A terminal window with a dark purple background. The title bar shows the user 'xiaoruqi' and the path '~/Desktop/OS_experiment/l...'. The terminal content includes the text '测试阶段: 3/5' at the top. Below it is a separator '=== 颜色输出测试 ==='. This is followed by seven lines of text, each with a color name and '文本' (text): '红色文本' (red), '绿色文本' (green), '黄色文本' (yellow), '蓝色文本' (blue), '紫色文本' (purple), '青色文本' (cyan), and '白色文本' (white). The last line is '按回车键继续...' (Press the Enter key to continue...).

```
xiaoruqi@xiaoruqi-VMware-Virtual-Platform: ~/Desktop/OS_experiment/l...
测试阶段: 3/5

=== 颜色输出测试 ===
红色文本
绿色文本
黄色文本
蓝色文本
紫色文本
青色文本
白色文本

按回车键继续...
```

图 14 测试 3-颜色输出测试

5.4 光标控制测试

测试光标控制功能的准确性，验证 `goto_xy` 函数能否精确定位到指定行列位置，检查坐标计算的正确性。

输出结果如下图，光标精确定位到第 7 行开头显示绿色标记线，随后定位到第 9 行第 10 列显示蓝色标记，定位准确无偏差。

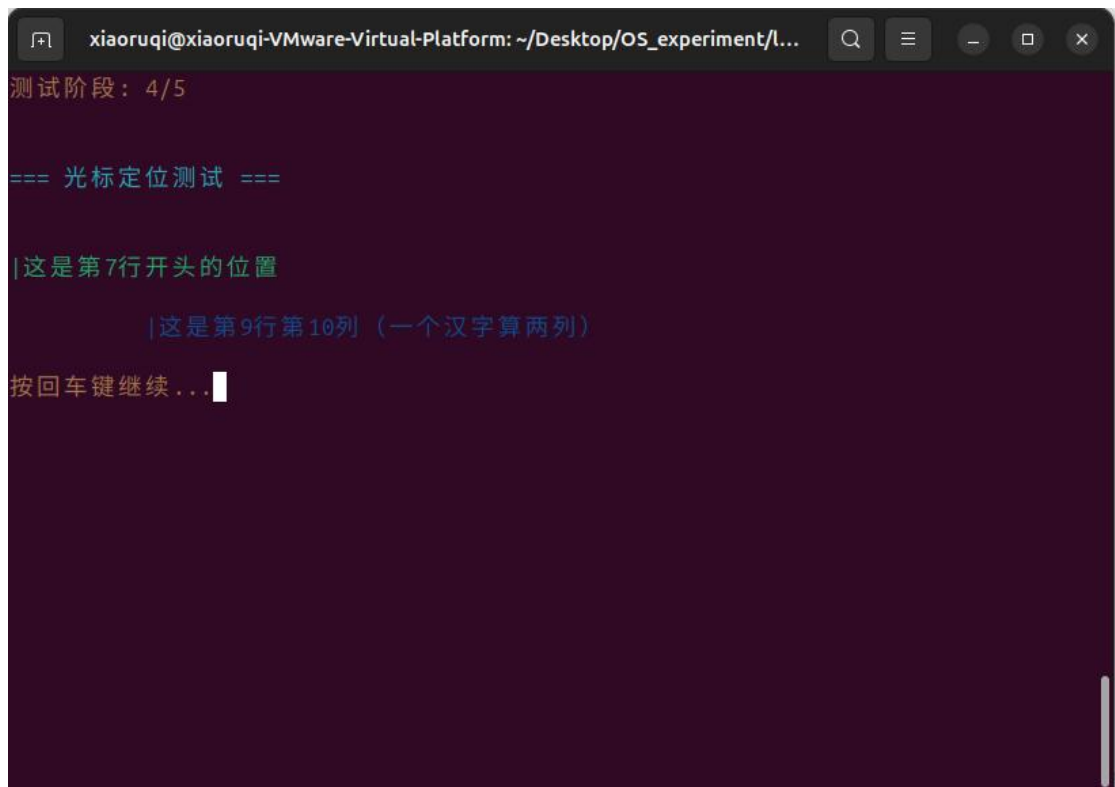


图 15 光标控制测试

5.5 清屏清行测试

由于每一轮测试展示时，都调用了 `clean_screen()` 函数，进行一次清屏，因此这里不再对清屏功能单独测试，这里重点测试清行功能。

1. 首先显示测试内容后提示用户“按回车键清除”，如下图：

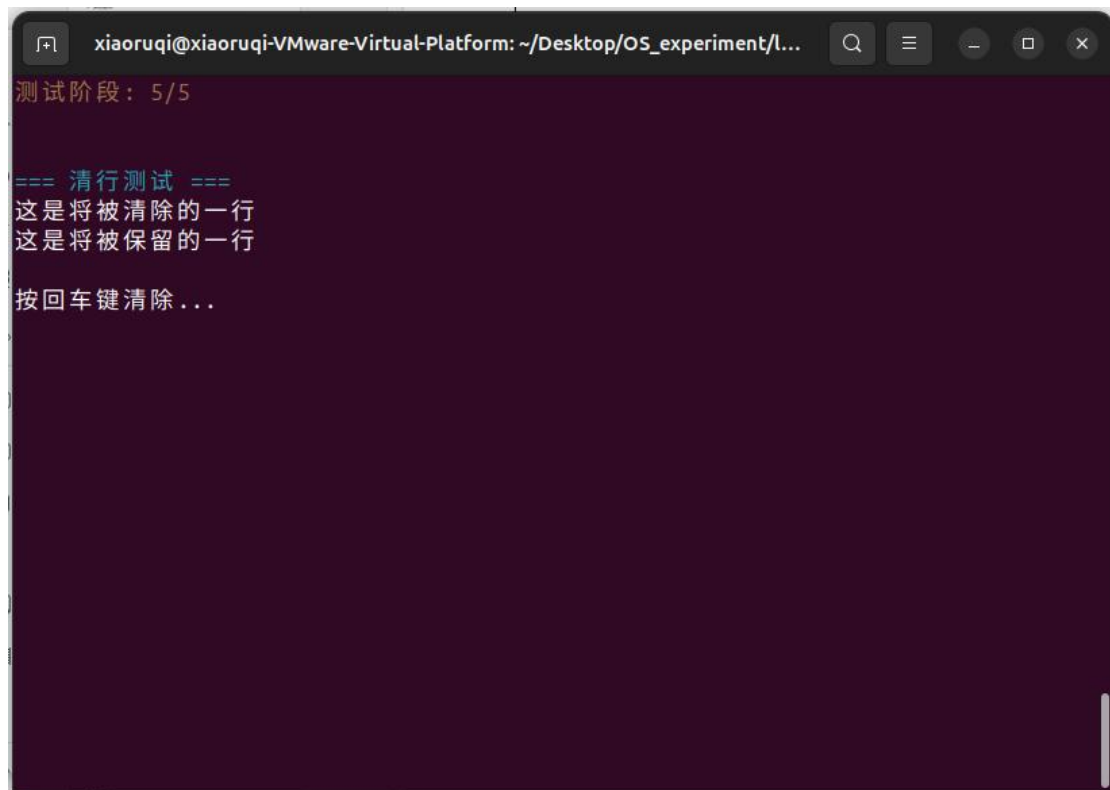


图 16 测试 5-清屏清行测试 1

2. 待用户按键后特定行内容被精确清除而其他行保留，演示动态界面更新效果。

效果如下：

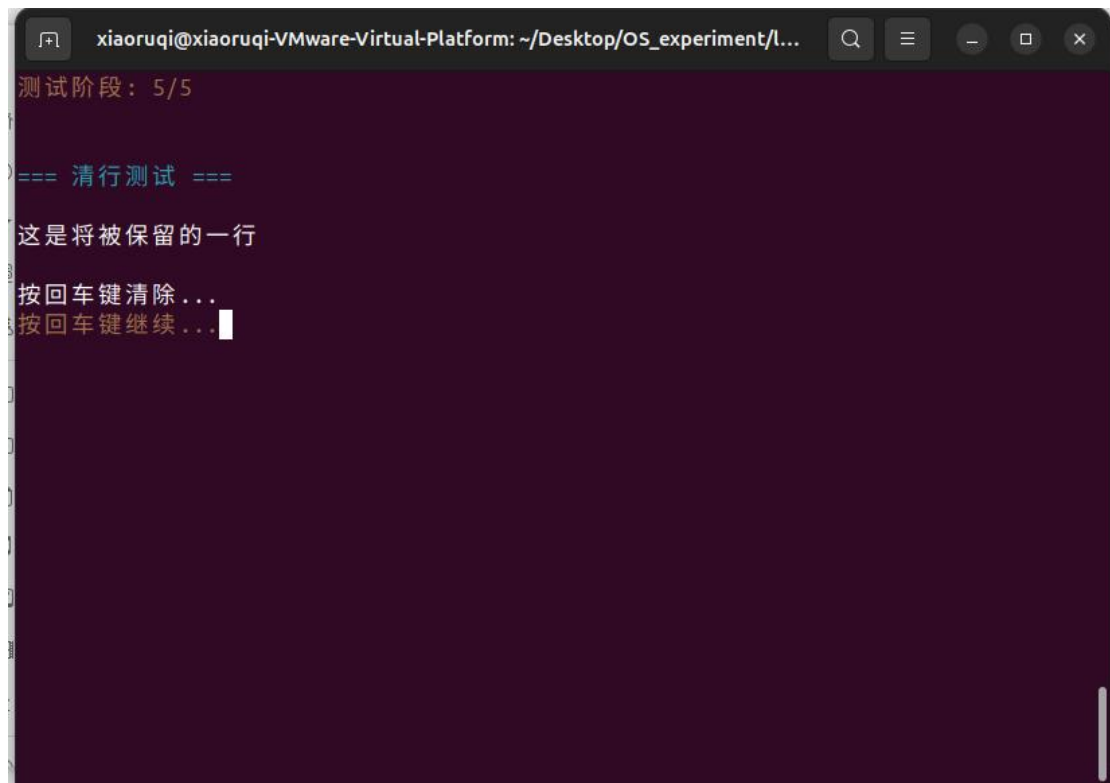


图 17 测试 5-清屏清行测试 2

6 思考题

6.1 架构设计

6.1.1 为什么需要分层？每层的职责如何划分？

本次内核输出系统分层主要解决三个核心问题——硬件差异抽象、功能逻辑分离和代码维护性。通过分层，将易变的硬件操作与稳定的业务逻辑解耦，提高系统可移植性和可维护性。

6.1.2 如果要支持多个输出设备（串口+显示器），架构如何调整？

如需支持串口+显示器双输出，架构可以进行如下调整：

1. 增加设备抽象接口，统一 uart 和 vga 的操作方法
2. 控制台层实现输出路由策略（同步/异步、主备切换等）
3. 添加设备管理模块，动态检测设备可用状态

6.2 算法选择

6.2.1 数字转字符串为什么不用递归？

在 2.1.3 节已讨论过该问题。

6.2.1 如何在不使用除法的情况下实现进制转换？

可以使用查表法——预计算各进制的幂次表，通过比较和减法实现转换：

```
1 // 例如十进制转十六进制
2 const uint32_t hex_powers[] = {0x10000000, 0x1000000, ..., 0x10, 0x1};
3
4 void print_hex_no_divide(uint32_t n) {
5     for (int i = 0; i < 8; i++) {
6         int digit = 0;
7         while (n >= hex_powers[i]) {
8             n -= hex_powers[i];
9             digit++;
10        }
11        uart_putc(digits[digit]);
12    }
13 }
```

图 18 查表法实现进制转换

6.3 性能优化

6.3.1 当前实现的性能瓶颈在哪里？

1. 字符级传输：每个字符都要单独检查 UART 状态，大量 CPU 时间浪费在等待循环
2. 无缓冲机制：频繁的小数据量传输，无法利用硬件批量传输优势
3. 格式解析开销：每次 printf 都要重新解析格式字符串

6.3.2 如何设计一个高效的缓冲机制？

可以采用双缓冲+批量传输方案。通过这种方案，可以减少 90%以上的状态检查开销；同时支持 DMA 批量传输，能够释放 CPU 资源并提升吞吐量。代码如下：

```
1  #define BUF_SIZE 256
2  struct output_buffer {
3      char data[BUF_SIZE];
4      int count;
5      int device_id; // 支持多设备
6  };
7
8  // 缓冲输出函数
9  void buffered_printf(const char *fmt, ...) {
10     if (buffer.count + estimated_len > BUF_SIZE) {
11         flush_buffer(); // 缓冲区满时触发传输
12     }
13     // 格式解析结果直接写入缓冲区
14     append_to_buffer(formatted_data);
15 }
16
17 // 条件刷新机制
18 void flush_buffer(void) {
19     if (buffer.count > 0) {
20         dma_transfer(buffer.data, buffer.count); // DMA批量传输
21         buffer.count = 0;
22     }
```

图 19 双缓冲+批量传输方案

6.4 错误处理

6.4.1 printf 遇到 NULL 指针应该如何处理？

采用分级处理方案：

```
1      case 's': {
2          char *s = va_arg(ap, char*);
3          if (s == NULL) {
4              // 方案1: 安全输出
5              uart_puts("(null)");
6              // 方案2: 调试信息
7              uart_puts("[NULL_PTR]");
8              // 方案3: 容错恢复
9              log_error("NULL string pointer");
10             break;
11         }
12         uart_puts(s);
13     }
```

图 20 NULL 指针的分级处理方案

6. 4. 2 格式字符串错误时的恢复策略是什么？

采用状态机容错机制：

```

1   for (int i = 0; fmt[i] != '\0'; i++) {
2       if (fmt[i] != '%') {
3           uart_putc(fmt[i]);
4           continue;
5       }
6
7       i++; // 跳过%
8       if (fmt[i] == '\0') {
9           uart_putc('%'); // 字符串以%结束
10          break;
11      }
12
13      switch (fmt[i]) {
14          // 已知格式符处理...
15          default:
16              // 未知格式符：原样输出%和字符
17              uart_putc('%');
18              uart_putc(fmt[i]);
19              log_warning("Unknown format: %%%c", fmt[i]);
20              break;
21      }
22  }

```

图 21 格式字符串状态机容错机制

7 实验总结

通过本次实验，我深入掌握了内核级输出系统的设计与实现，从 xv6 的复杂架构分析到自主设计简化的两层模型，成功实现了支持格式化输出、颜色显示和清屏功能的完整系统。实验过程中遇到的 INT_MIN 边界处理、ANSI 转义序列兼容性、状态机解析等实际问题，锻炼了我底层编程和调试能力。更重要的是，这次实践让我深刻理解了内核开发与用户态编程的本质差异——内核必须自给自足，不能依赖外部库，所有功能都需要从硬件寄存器操作开始逐层构建。这种从零搭建系统组件的经验，不仅巩固了操作系统理论知识，更为后续更深层次的内核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。）