验收成绩	报告成绩	总评成绩

武汉大学计算机学院

本科生实验报告

操作系统实践 A

专业名称：计算机科学与技术

课程名称：操作系统实践

指导教师：李祖超教授

学生学号：2023302111351

学生姓名：夏盛宇

二〇二五年十二月

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: 夏盛宇　　日期:2025/12/26

目　　录

1 Lab0 总览

1.1 实验目标与完成情况

Lab1

实验目标

通过参考 xv6 的裸机引导流程，实现 `_entry → start() → UART` 的最小引导链路，掌握 RISC-V 启动阶段的栈初始化、BSS 段清零、串口驱动与多核栈隔离关键机制。

完成情况

- `entry.S` 完成 per-hart 栈指针设置、BSS 清零与跳转逻辑。
- `kernel.ld`、`start.c`、`uart.c`、`print.c` 等核心文件补全，并为 printf 系列加锁。
- QEMU virt 平台多次运行 `make qemu` 均稳定输出“Hello OS”，无未完成任务。

Lab2

实验目标

在 Lab1 最小引导环境的基础上实现“可用的内核 printf 子系统”。包含：

1. 设计与实现支持 `\%d/\%x/\%p/\%s/\%c/\%\%` 的内核 `printf`；
2. 提供清屏、定位、前景/背景颜色设置等 ANSI 控制能力；
3. 构建线程安全的输出路径，保证多核串口不会交错。

完成情况

- `print.c` 重新实现 `printf/puts`，可安全输出字符串、整数、指针、字符等多种格式。
- 新增 console 抽象与 ANSI 工具函数 `clear_screen/goto_xy/set_color/reset_color`，并复用 PPT 第四节中的控制序列建议。
- `start.c` 演示脚本完成清屏、颜色切换、光标跳转与边界测试，串口输出截图记录在 `picture/*.png`。
- 自旋锁、BSS 清零、per-hart 栈等基础设施全部继承并通过测试。

Lab3

实验目标

1. 通过阅读 xv6 和实验框架代码，理解 **RISC-V Sv39 虚拟内存机制**。
2. 在此基础上，**独立实现物理内存分配器**，支持内核区 / 用户区的页级分配与释放。
3. 实现 **多级页表管理系统**，包括：
   - 从虚拟地址逐级查找页表项；
   - 建立 / 解除虚实映射；
   - 打印页表以便调试；
   - 释放整棵页表树。
4. 学会在 RISC-V 上 **启用分页并保持内核正常运行**，理解 satp、sfence.vma 等关键指令的作用。

5. 在基本功能之上，加入：
    - 多页分配接口 `allocated_pages(int n)`；
    - double free 防护；
    - 当前空闲页统计接口；
    - `destroy_pagetable` 等资源回收功能。

    完成情况

- **物理内存管理：**`kernel/mem/pmem.c` 将可用物理内存划分为内核/用户两个 `alloc_region_t`,实现 `pmem_alloc/pmem_free`、`allocated_pages()`、`pmem_free_pages_count` 并在 `pmem_free()` 中加入 double-free 检测以保护链表。
- **虚拟内存与页表:**`kernel/mem/vmem.c` 完成 `vm_getpte/vm_mappages/vm_unmappages/vm_dest` 支持 Sv39 多级页表遍历、映射与整棵释放;`kernel/mem/kvm.c` 中的 `kvm_init/kvm_inithart` 已能构造并启用内核页表。
- **自测与调试：**`kernel/boot/main.c` 编写 `test_pmem_*`、`test_vmem_*` 等用例，验证单页/多页分配、页表映射/回收及异常（对齐、越界、double free），对应截图保存在 `picture/test*.png`。

Lab4

实验目标

在 Lab3 运行态基础上，补齐“中断—trap—驱动”全栈链路，使得 RISC-V 平台可以正确处理 M/S 模式下的时钟和外设中断、输出调试日志，并为后续调度/系统调用提供定时器与异常框架。

完成情况

- 配置 `medeleg/mideleg`、`mie/sie`、PLIC 阈值等寄存器,实现 Machine→Supervisor trap 委托；
- 实现 `kernel_vector/timer_vector` 与 `trap_kernel_handler()`，可区分时钟、外设与异常；
- 搭建 IRQ 注册与 enable/disable API，UART 中断与回显工作正常；
- 新增 `timer_update()/timer_get_ticks()`，通过多项测试验证精度与性能；
- 可进一步补充的方向：将 `timer_interrupt_handler()` 中的调度钩子替换为真正的进程调度逻辑，并扩展更多 PLIC 设备。

Lab5

实验目标

- 在 RISC-V 内核上实现支持内核线程的进程管理子系统：完成 `struct proc` 定义、上下文切换、调度器、进程创建/退出/等待等逻辑；
- 借助时钟中断实现抢占式轮转调度，为后续用户态/系统调用实验打下基础；
- 通过内置测试 `run_all_tests()` 验证进程生命周期、调度公平性与同步正确性。

完成情况

- 进程表/CPU 控制块/context/swtch 汇编实现与 scheduler 闭环；
- 时钟中断触发 `yield()`，能抢占并在三种 worker demo 间轮转；
- `test_process_creation`、`test_scheduler`、`test_synchronization`、`debug_proc_table` 等测试全部通过；
- 目前仅支持内核线程，尚未引入用户态地址空间/系统调用，计划在后续实验

扩展。

Lab6

实验目标

- 为 RISC-V 内核补齐“用户态 → ecall → 内核处理 → 返回用户态”的系统调用链路，支持 `fork/exec/wait/read/write/sbrk/kill/getpid/open/close` 等核心接口；
- 实现用户页表、trap/trampoline、syscall 分发与文件描述符表，使 `/init` 程序可以调用系统调用驱动最小用户态；
- 通过指导手册中的基础/参数/安全/性能测试，确保系统调用的正确性、健壮性与边界处理。

完成情况

- `user/usys.S` + `trampoline.S` + `usertrap()/syscall()` 闭环打通；
- `struct proc/pagetable/trapframe` 扩展完备，可为每个进程构建独立用户地址空间；
- `sys_{fork,exec,wait,read,write,open,close,sbrk,kill,getpid}` 全部实现并经测试验证；
- `run_lab6_syscall_tests()` 覆盖功能/参数/安全/性能四类测试；
- 仍待扩展：尚未实现更丰富的文件系统、`pipe/mmap/fstat` 等系统调用，后续实验将继续完善。

Lab7

实验目标

- 搭建基于 virtio block 的块设备层与 32 块缓存,确保所有磁盘访问都走 `bread/bwrite`；
- 实现写前日志 (`begin_op/log_write/end_op`) 保证崩溃一致性；
- 完整落地 inode/目录/路径解析/位图分配，支持直接 + 间接块；
- 通过课件提出的完整性、并发、性能、自检等测试，验证实现符合 Lab7 要求。

完成情况

- irtio block 驱动 + 块缓存 + I/O 统计；
- 日志层 commit/recover 流程，支持并发 begin/end；
- inode/目录/路径/位图函数与 mkfs 工具；
- `lab7_*` 测试：integrity/concurrent/performance/debug；

Lab8

实验目标

1. 构建“用户态　内核态”执行链路：trap/系统调用/ELF 装载/文件接口全贯通；
2. 实现课件中的多级反馈队列 (MLFQ) 调度，开放 `setpriority/getpriority` 系统调用；
3. 扩展内核服务（日志环形缓冲、消息队列 IPC）与用户态示例程序，形成可演示的扩展系统。

完成情况

- 用户态运行：`usertrap/usertrapret`、ELF 装载、`/init` 等均可运行；
- 调度扩展：优先级存储、量子控制、老化/boost、优先级 syscall 已上线；

- 新服务:klog 日志、msgget/msgsend/msgrecv IPC、logread/nice/msgdemo/elfdemo 等示例；

1.2 实验环境

- 硬件：x86_64 主机（QEMU virt）
- 操作系统：Ubuntu 24.04 LTS
- 工具链：riscv64-unknown-elf-gcc 12.2.0
- 模拟器：qemu-system-riscv64 8.2.2（-machine virt -nographic）
- 调试器：gdb-multiarch 15.0.50

2 Lab1-内核启动

2.1 技术设计

2.1.1 系统架构设计

整体流程分为“加载 → 汇编引导 → C 初始化 → 驱动输出”四个阶段，模块与 xv6 对应部分一一映射，方便后续扩展。

```
QEMU loader → _entry(entry.S) → start() →
↪ print_init()/puts()

加载到 0x80000000   栈和 BSS 就绪      主核判定      UART 驱
↪ 动输出
```

- boot：设置 per-hart 栈、清零 BSS，并跳转到 C 入口。
- lib：提供 print、自旋锁、assert，保证串口输出线程安全。
- dev：实现 16550A UART 驱动，统一寄存器访问接口。
- proc：定义 struct cpu 与 mycpuid()，为多核调度预留接口。

与 xv6 对比：本实验精简了中断、调度和分页，但保留 per-hart 栈与锁语义，后续扩展无需推翻现有设计。

2.1.2 关键数据结构

```
1   struct cpu {
2     int id;
3     int started;
4   };
5
6   typedef struct spinlock {
7     int locked;
8     char *name;
9     int cpuid;
10  } spinlock_t;
11
12  __attribute__((aligned(16))) uint8 CPU_stack[4096 * NCPU];
13  extern int panicked;
```

- struct cpu：通过 tp 寄存器获取 hartid，便于锁归属检查。
- spinlock_t：记录锁状态与持有者，panic 时可打印锁名，定位死锁。
- CPU_stack / panicked：置于 BSS 段并由 _entry 清零，确保所有核状态一致。

2.1.3 核心算法与流程

1. `_entry` 读取 `mhartid`，为每个 hart 分配 4KB 栈，并以 8 字节步长清零 `[edata,end)`。
2. 跳转到 `start()` 后仅允许 hart0 初始化 UART、打印“Hello OS”，其余核进入 `wfi`。
3. `print.c` 的 puts/printf 通过自旋锁保护；`uart_putc_sync` 轮询 LSR，保证裸机串口输出可靠。
4. 若后续加入中断/调度，只需在 `start()` 中扩展 S 模式切换与 trap 向量，无需重写引导流程。

2.2 实现细节与关键代码

2.2.1 关键函数

`entry.S` ——多核栈与 BSS 清零：

```
    la  sp, CPU_stack
    li  a0, 4096
    csrr a1, mhartid
    addi a1, a1, 1
    mul a0, a0, a1
    add sp, sp, a0
    la  a0, edata
    la  a1, end
    1: bge a0, a1, 2f
    sd  zero, 0(a0)
    addi a0, a0, 8
    j   1b
    2: call start
```

`start.c` ——仅主核初始化 UART 并输出：

```
    void start(void) {
      unsigned long hartid;
      asm volatile("csrr %0, mhartid" : "=r"(hartid));
      if (hartid == 0) {
        print_init();
        puts("Hello OS");
      }
      while (1) asm volatile("wfi");
    }
```

`print.c` / `uart.c` ——串口输出与锁保护：

```
    void puts(const char *s) {
      if (!s) return;
      spinlock_acquire(&print_lk);
      while (*s) uart_putc_sync(*s++);
      spinlock_release(&print_lk);
    }

    void uart_putc_sync(int c) {
      push_off();
      while (panicked);
      while ((ReadReg(LSR) & LSR_TX_IDLE) == 0);
      WriteReg(THR, c);
      pop_off();
    }
```

2.2.2 难点突破

- **工具链缺失:**默认使用 `riscv64-linux-gnu-gcc`,改为安装 `gcc-riscv64-unknown-elf` 并统一前缀。
- **汇编扩展名:** `entry.s` 不经过预处理导致符号缺失,更名为 `.S` 并在 `kernel.ld` 导出 `edata/end`。
- **字符串 relocation 溢出:** `"Hello OS"` 距离 PC 过远,调整链接脚本让 `.rodata` 紧挨 `.text`。
- **多核串口乱序:** 移除 `hartid==0` 后字符交织，通过自旋锁保护并限制主核输出，恢复稳定性。

2.2.3 源码理解与思考题

2.2.4 源码理解总结

- **启动流程:** QEMU 加载 kernel → `_entry` 设置栈 → 清零 BSS → `start()` → 初始化 UART → `puts("Hello OS")`。
- **内核模块划分:** boot 负责硬件初始化，lib 负责基本功能，proc 提供 CPU 抽象。

2.2.5 思考题解答

- **启动栈:** 按 4KB 估算普通函数深度，可在栈底放置魔数检测溢出。
- **BSS 清零:** 若省略，`panicked`、`print_lk` 等全局变量将含随机值；只有引导固件保证清零时才可跳过。
- **与 xv6 对比:** 缺少中断、分页与调度，但依旧保留 per-hart 栈、自旋锁和 panic 逻辑，有利于扩展。
- **错误处理:** UART 失败无法返回，只能死循环并通过 LED/蜂鸣器输出错误码，保证最小可观测性。

2.3 测试与验证

2.3.1 功能测试

基本启动

```
$ make qemu
Hello OS
```

实际输出与预期一致，串口没有出现乱码。

多核验证移除 `hartid==0` 判断，每个核都会输出“Hello OS”，次数与 `-smp` 参数一致，验证 per-hart 栈正常工作。

2.3.2 边界与异常测试

- **BSS 清零实验:** 注释清零循环后，`panicked` 有时为 1，串口停止输出，证明清零步骤必要。
- **UART 忙等待:** 在 `uart_putc_sync` 中临时增加延迟，确认锁不会被打断，输出仍保持顺序。
- **panic 场景:** 手动调用 `panic("test")`，串口输出 `panic: test` 并进入死循环，异常路径可用。

2.3.3 调试

- 调试流程: `make qemu-gdb` + `gdb-multiarch kernel.elf`, `target remote :1234` 后可单步跟踪 `_entry`。

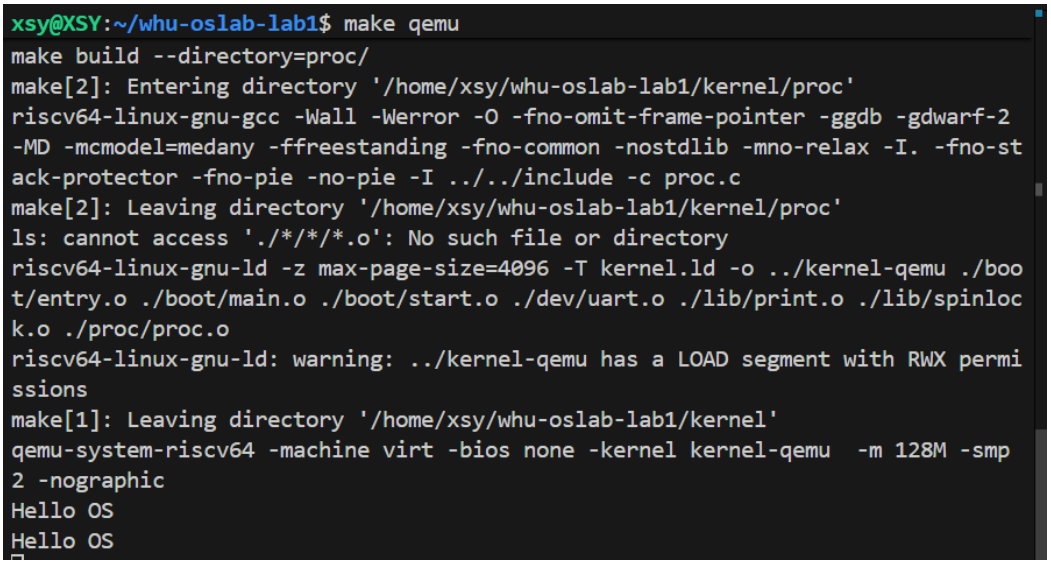

图 2.1　2CPU Test

图 2.2　test

2.4 问题与总结

2.4.1 遇到的问题

1. **entry.s 无法解析符号**
   - 现象：链接时报 `undefined reference to edata`。
   - 原因：`.s` 文件缺少 C 预处理阶段。
   - 解决：改为 `entry.S` 并在 linker script 中导出符号。
   - 预防：约定所有需要包含头文件的汇编文件均使用 `.S`。
2. **字符串 relocation 溢出**
   - 现象：编译器提示 `relocation truncated to fit`。
   - 原因：字符串常量距离 PC 超过 4KB。
   - 解决：将 `.rodata` 紧跟 `.text`，让常量处于可寻址范围。
   - 预防：保持段布局紧凑，必要时拆分文本。

2.4.2 实验收获

- 掌握 RISC-V 裸机引导链路、链接脚本与 BSS 管理。
- 理解 16550A UART 配置及多核串口同步方式。
- 熟悉 QEMU + gdb-multiarch 联合调试，能逐条验证指令执行。

2.4.3 改进方向

- 增加 LED/蜂鸣器等最小错误指示，提升裸机可观测性。
- 在 `print.c` 中实现 `printf`/格式化输出并补充单元测试。
- 预留 trap 向量与分页框架，为后续实验快速扩展调度与内存管理。

3 Lab2-内核 Printf

3.1 技术设计

3.1.1 系统架构

整体输出路径增加了 console 层，便于在 UART 与未来的 VGA/缓冲设备之间切换。

与 xv6 相比：

- 删除了 xv6 中较重的中断/TTY 逻辑，只保留轮询串口；
- printf 仍借助空闲链表式 `spinlock`，保持接口一致；
- 根据 PPT 内容加入 ANSI 控制序列，代替 xv6 通过显存清屏的方式，更适合 `-nographic` 的 QEMU。

3.1.2 关键数据结构

```
typedef struct spinlock {
    int locked;
    char *name;
    int cpuid;
} spinlock_t;

__attribute__((aligned(16))) uint8 CPU_stack[4096 * NCPU];
extern int panicked;
```

- `spinlock_t`:沿用 Lab1 的自旋锁,新增 `console_putc()` 时仍统一由 `print_lk`

保护，保证多核输出顺序。

- `CPU_stack` / `panicked`：位于 BSS，由 `entry.S` 清零，确保 printf 不受脏数据干扰。
- `console`：虽然结构简单，但在接口上与 PPT 中“console 抽象”一致，后续可以替换为 VGA 或环形缓冲。

3.1.3 核心流程

1. `_entry`：设置 per-hart 栈 & 清零 `[edata,end)`；
2. `start()`：仅 hart0 执行 `print_init()`，其余 hart 在 `wfi` 等候；
3. `print_init()`：初始化 console (间接初始化 UART) 并初始化自旋锁；
4. `printf()`:解析格式串 → 逐一从 `va_list` 取参数 → 调用 `print_number`/`print_putc`；
5. ANSI 功能：根据 PPT 第四节的 ESC 序列，组合 `\x 1b[2J\x 1b[H`（清屏）、`\x 1b[row;colH`（定位）、`\x 1b[3xm/\x 1b[4xm`（颜色）。

3.2 实现细节与关键代码

3.2.1 printf 核心代码

```
void printf(const char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    spinlock_acquire(&print_lk);
    for (const char *p = fmt; p && *p; ) {
        if (*p != '%') { print_putc(*p++); continue; }
        switch (*++p) {
        case 's': { char *s = va_arg(ap, char*); if (!s) s = "(null
            )"; while (*s) print_putc(*s++); break; }
        case 'd': { long v = va_arg(ap, int); print_number((
            unsigned long)v, 10, 1); break; }
        case 'x': { unsigned int x = va_arg(ap, unsigned int);
            print_number(x, 16, 0); break; }
        case 'p': { unsigned long x = va_arg(ap, unsigned long);
            print_number(x, 16, 0); break; }
        case 'c': { print_putc((char)va_arg(ap, int)); break; }
        case '%': print_putc('%'); break;
        default:  print_putc('%'); if (*p) print_putc(*p); break;
        }
        if (*p) p++;
    }
    spinlock_release(&print_lk);
    va_end(ap);
}
```

- `print_number()` 采用迭代除法与本地缓冲；带符号路径直接在 64 位寄存器里完成，不会触发 `INT_MIN` 溢出。
- Panic 场景绕过锁，直接 UART 输出，避免死锁。

3.2.2 ANSI 控制函数

```
void clear_screen(void)  { print_puts("\x1b[2J\x1b[H"); }
void goto_xy(int row,int col) { /* 构造 ESC[row;colH */ }
void set_color(int fg,int bg) {
    char seq[32];
    /* ESC[3xm 控制前景色, ESC[4xm 控制背景色 */
}
```

- 参照 PPT 第四节“ANSI 控制序列”中的写法实现；
- print_puts 直接走 UART，同 panic 路径可用于早期输出。

3.2.3 演示脚本

start.c 的 hart0 流程：

```
clear_screen();
set_color(2, -1);
printf("=== OS Lab: Kernel printf & ANSI Demo ===\n");
reset_color();
printf("cpuid=%d, hex=0x%x, char=%c, str=%s, percent=%%\n", mycpuid
    (), 0xABC, 'X', "Hello");
printf("INT_MIN test: %d, INT_MAX: %d\n", (int)0x80000000, (int)0
    x7fffffff);
goto_xy(6, 10); printf("Here at (6,10)\n");
for (int i=3;i>0;i--) printf("%d ", i);
clear_screen(); printf("Screen cleared.\n");
```

- 既演示颜色/定位，也检验 \%d/\%x/\%s/\%c/\%p/\% 等路径。

3.2.4 难点突破

1. **INT_MIN 溢出**：print_number 直接对 int 取负会溢出。解决：先提升为 long，再转为 unsigned long。
2. **ANSI 序列无效**：最初忘记写 ESC[H 导致清屏后光标停在原位置，参考 PPT 第四节方案改为 \x 1b[2J\x 1b[H。
3. **光标跳转覆盖输出**：goto_xy 默认 1-based，误传 0 导致覆盖。加入参数校验并把 demo 坐标调至 (6,10)。
4. **多核输出乱序**：早期在 printf 内部未加锁，hart1 会穿插 hart0 文本。复用 Lab1 的 spinlock 后问题消失。

3.2.5 思考题 & 源码理解

- **为什么内核要自建 printf？** 启动期没有 libc 与系统调用，只有直接驱动串口的自研 printf 才能提供调试出口。
- **为何要清屏/定位？** 便于在 -nographic 控制台快速区分新旧输出，也是 PPT 第四节建议的“最小用户界面”。
- **为什么采用 console 抽象？** 当前只有 UART，但抽象层可屏蔽底层差异，未来接入 VGA/日志缓冲不需改 printf。

3.3 测试与验证

3.3.1 功能测试

1. **基本启动**

```
$ make qemu
Hello OS
=== OS Lab: Kernel printf & ANSI Demo ===
...
```

颜色、定位、清屏依次生效，串口输出与预期相符。

2. **格式化覆盖**：打印 \%d/\%x/\%p/\%s/\%c/\%\%，并分别测试 0、负数、INT_MIN、长指针、NULL 字符串。
3. **并发验证**：临时去掉 hartid==0 判断，两核同时 printf，仍因锁保护而无字符交织。

3.3.2 边界 / 异常测试

- **INT_MIN**:`printf("INT_MIN: \%d\n ", 0x80000000)` → 输出 `-2147483648`。
- **NULL 字符串**：`printf("\%s\n ", (char*)0)` → 输出 `(null)`。
- **未知格式**：`printf("\%q\n ", 1)` → 以 `"\%q"` 原样输出，保证内核不 panic。
- **ANSI 回退**：向 `set_color` 传入负数时自动回退到默认 7/0。

3.3.3 截图 & 日志

- `picture/clean.png`：`clear_screen()` 后重新绘制的终端。
- `picture/moveCursor&color.png`：绿色标题 + `(6,10)` 输出，验证 PPT 第四节示例。
- `picture/printfTest.png`：展示格式化结果与倒计时清屏过程。

  输入 make qemu 后成功显示出清屏的功能：

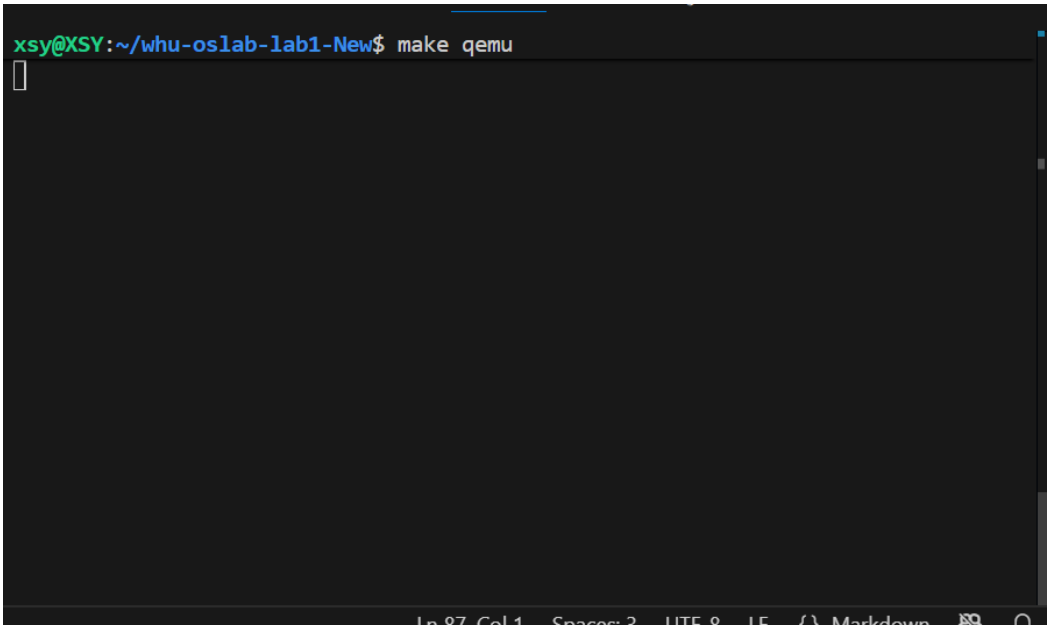

图 3.1　clean

向上滑动查看终端历史输出记录：

printf 输出结果符合预期：

我们可以清楚地看到字体颜色发生了改变，输出了一段绿色的字符串。另外，边界测试结果也成功输出。

3.4 问题与总结

3.4.1 遇到的问题

1. 数字转换错误（如负数显示为正数）——在原来的 print_number 中未处理负数情况（直接使用无符号数转换），INT_MIN 转换时溢出（-INT_MIN 仍为负数）后来在 print_number 中先判断符号，转为正数处理；INT_MIN 特殊处理，直接硬编码输出
2. 清屏或光标定位无效——后来检查发现是转义序列格式错误（如缺少 `[` 或使用错误参数），使用标准转义序列。
3. 光标跳转后覆盖原有输出——在检查输出时，我发现原来的 `Moving cursor to (6,10) and printing there...` 这段输出被后面的输出覆盖。仔细排查

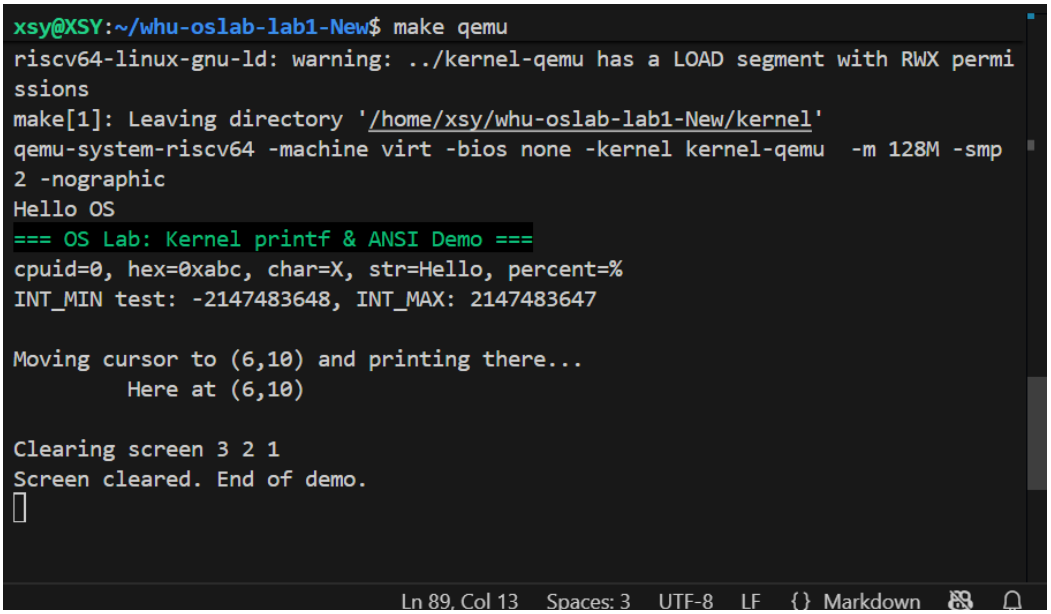

图 3.2　color&moveCursor

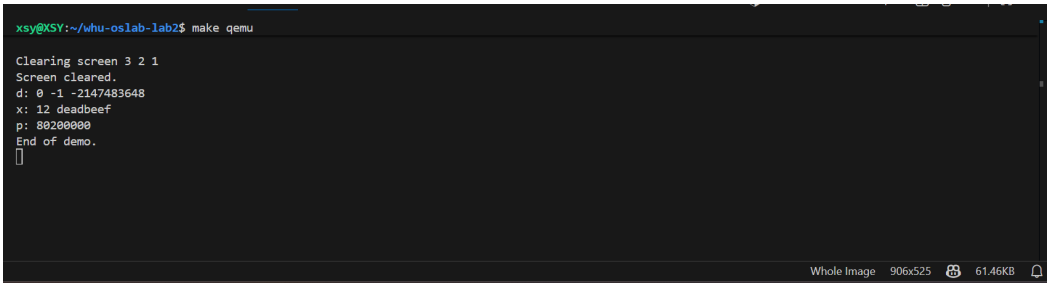

图 3.3　printf

后发现这是由于光标移动的位置刚好在那个字符串输出的位置上，将其移动位置改为（6，10）就避免了覆盖。

3.4.2 实验收获

- 深入理解了“启动期自研 printf + console 抽象”的设计意义；
- 掌握 ANSI 控制序列的常见写法，可快速在终端中呈现友好的 UI；
- 熟悉了在裸机环境下调试格式化输出、定位边界问题的方法。

3.4.3 改进方向

1. **缓冲与中断驱动**：后续可将 console 切换为环形缓冲 + 中断发送，降低忙等开销。
2. **可扩展格式化**：支持最小宽度、填充、无符号十进制 \%u 等。
3. **日志子系统**：引入日志级别、时间戳、串口/内存双写，方便排障。

3.5 思考题回答

3.5.1 为什么内核要自己实现 printf？

因为在内核启动或出错时，用户态和标准库尚未建立，必须有最小的输出机制用于调试和诊断。此外，内核需要直接控制硬件（如 UART、显存），而用户态 printf 依赖于系统调用，这在早期启动阶段是不可用的。

3.5.2 为什么需要清屏？

避免多次输出堆叠影响观察，尤其是教学/调试时更直观。此外，清屏功能也是实现更复杂交互界面的基础，如菜单系统、日志查看器等。

3.5.3 架构设计：为什么分层？

驱动层（UART）只负责单字符输入/输出；上层（print.c）实现格式化；调用层（start.c）演示逻辑。这种分层便于替换或扩展输出设备（如 VGA）。分层还使得代码更易于测试和维护，每一层职责明确，耦合度低。

3.5.4 算法选择：数字转字符串不用递归的原因及不用除法实现进制转换

- 避免早期内核栈消耗过大，同时迭代实现效率更高。递归在栈空间有限的内核环境中容易导致栈溢出，且递归调用开销较大，不适合内核这种对性能和安全要求高的环境。
- 可以使用查表法或减法替代法（通过重复减法模拟除法运算，统计减法的次数作为商，剩余数作为余数。）

3.5.5 性能优化：瓶颈与改进？

瓶颈主要在循环调用 uart_putc_sync；改进方向是引入缓冲区、批量写、甚至中断驱动。此外，可以针对常用输出路径（如整数输出）进行内联优化，减少函数调用开销。

3.5.6 错误处理策略

printf 遇到空指针的时候我在代码里先判断，后面直接输出字符串 `NULL`。格式字符串出错时遇到未定义的字符直接输出。如下：

- `if (!s) //s 为空，输出 NUll`
- `s = "(null)";`
- `while (*s) //否则，逐字符输出字符串`
- `print_putc(*s++);`

4 Lab3-页表与内存管理

4.1 系统设计部分

4.1.1 架构设计说明

整个内存管理子系统分为三层：

1. **物理内存层（pmem）**
   - 负责“砖头”的管理：哪些物理页可用、如何分配/回收。
   - 将可用物理内存划分为**内核区域**和**用户区域**，分别维护空闲页链表。
   - 对外提供：
     - `pmem_init()`：初始化两个区域的可分配页；
     - `pmem_alloc(in_kernel)` / `pmem_free(page, in_kernel)`；
     - `allocated_pages(int n, bool in_kernel)`：多页分配接口；
     - `pmem_free_pages_count(in_kernel)`：统计当前空闲页数。
2. **虚拟内存 / 页表层（vmem）**
   - 负责“砖头怎么摆”：实现 Sv39 多级页表遍历和映射建立。
   - 提供接口：
     - `vm_getpte(pgtbl, va, alloc)`：多级遍历，必要时分配中间页表；
     - `vm_mappages(pgtbl, va, pa, len, perm)`：一段虚拟地址到物理地址的映射；
     - `vm_unmappages(pgtbl, va, len, freeit)`：解除映射，选项决定是否回收物理页；
     - `vm_destroy_pagetable(pgtbl, free_leaf)`：释放整棵页表树；
     - `vm_print(pgtbl)`：打印当前页表映射用于调试。
3. **内核启动和测试层（boot/main.c）**
   - 在主核中依次调用：
     (a) `pmem_init()`：建立物理空闲页链表；
     (b) `kvm_init()`：创建并填充内核页表；
     (c) `kvm_inithart()`：写 satp & sfence.vma，启用分页；
     (d) 构造一张测试用页表 `test_pgtbl`,调用 `vm_mappages` / `vm_unmappages` / `vm_print` 验证功能；
     (e) 调用扩展接口测试多页分配、double free 防护等。

三层之间只通过清晰的 C 接口交互，层次关系如下：

硬件内存 & 页表结构

↑（riscv.h + memlayout.h 宏）

物理内存管理 pmem.c

↑

虚拟内存管理 vmem.c

↑

启动流程 & 测试 main.c

4.1.2 关键数据结构

4.1.3 物理内存区域 alloc_region_t

```
typedef struct page_node {
  struct page_node* next;
} page_node_t;

// 许多物理页构成一个可分配的区域
typedef struct alloc_region {
  uint64 begin;          // 起始物理地址（页对齐）
  uint64 end;            // 终止物理地址（开区间）
  spinlock_t lk;         // 自旋锁，保护链表与计数
  uint32 allocable;      // 当前空闲页数量
  page_node_t list_head; // 单向链表虚假头节点
} alloc_region_t;

static alloc_region_t kern_region, user_region;
```

4.1.4 设计要点

- 可分配物理内存是一个连续区间，区间内部每个 4 KiB 页都可以直接强转为 page_node_t*；
- 不需要额外元数据数组，页本身作为链表节点，类似 xv6 的 struct run 设计；
- kern_region 和 user_region 分别保存内核 / 用户页，后续通过地址范围判断属于哪个区域。

4.1.5 页表项与页表类型

vmem.h 中定义：

```
typedef uint64 pte_t;    // 页表项
typedef uint64* pgtbl_t; // 页表指针（512 项）

// 从虚拟地址取 VPN[2..0]
#define VA_SHIFT(level)       (12 + 9 * (level))
#define VA_TO_VPN(va, level)  ((((uint64)(va)) >> VA_SHIFT(
    level)) & 0x1FF)

// PTE 与PA的转换
#define PA_TO_PTE(pa) ((((uint64)(pa)) >> 12) << 10)
#define PTE_TO_PA(pte) (((pte) >> 10) << 12)

// 权限位
#define PTE_V (1 << 0)
#define PTE_R (1 << 1)
#define PTE_W (1 << 2)
#define PTE_X (1 << 3)
#define PTE_U (1 << 4)

// 判断是否为“中间页表”（R/W/X 全 0）
#define PTE_CHECK(pte) (((pte) & (PTE_R | PTE_W | PTE_X)) == 0)

// VA 上限
#define VA_MAX (1ul << 38)
```

这些宏是整个页表管理逻辑的基础，直接对应 RISC-V Sv39 的 PTE 格式。

4.1.6 与 xv6 对比分析

4.1.7 物理内存管理

xv6：使用 struct run { struct run *next; }; 单向链表管理空闲页；使用 freerange() 按照地址区间将所有空闲页链接起来；不区分内核/用户区域（所有物理页共用一条链表）。

本实验：在 xv6 设计基础上增加了一层抽象 alloc_region_t，将物理页分为内核区域和用户区域；每个区域都有自己的锁和链表，更便于未来做权限控制、统计以及 per-CPU 优化；采用相同的“页作为节点”的思想，不额外引入 bitmap/buddy 等复杂结构。

4.1.8 页表管理

xv6 的 walk() / mappages() 与本实验的 vm_getpte() / vm_mappages() 本质一致：按层次从 VPN[2] → VPN[1] → VPN[0] 逐级索引；中间级 PTE 的 R/W/X 必须为 0，仅设置 V 位作为“页表指针”；alloc 参数控制是否自动创建中间页表。

4.1.9 本实验额外实现了：

- vm_unmappages()：支持按段解除映射并可选择释放物理页；
- vm_destroy_pagetable()：递归释放整棵页表树；
- vm_print()：递归打印页表并标注区域，如 UART、PLIC、KERNEL_TEXT、KERNEL_DATA、UNKNOWN 等。

4.1.10 内核页表初始化

xv6 的 kvminit() 恒等映射 UART、PLIC、内核代码和数据。本实验的 kvm_init() 采用相同思路：

[KERNEL_BASE, etext] → 只读 + 可执行（RX）；[etext, PHYSTOP] → 读写（RW）；UART/PLIC 等设备等值映射为 RW。

总体上，本实验在保持 xv6 简洁性的同时，对区域划分和调试工具做了扩展。

4.1.11 设计决策理由

（1）为何采用“链表 + 页即节点”的物理分配器？

优点：实现简单、内存开销极小、每次分配/释放都是 O(1)，适合教学和小规模系统；在实验中，我们只需要页粒度分配，虚拟内存层可以通过“非连续物理页 + 连续虚拟地址”实现大块内存，因此该设计足够。

（2）为什么将物理页分为内核区 / 用户区？

内核页表、内核栈等必须使用“内核区域”的物理页，避免被用户错误释放或覆盖；用户地址空间的物理页单独计数，更方便实现统计、限制和 future work（如 cgroup 内存限制）。

（3）为什么实现 vm_getpte 而不是每次手写遍历？

将“硬件页表行走算法”封装为一个函数，方便在 vm_mappages、vm_unmappages、vm_destroy_pagetable 中统一使用；逻辑更接近 RISC-V 手册对 hardware page walk 的描述，便于理解和调试。

（4）double free 防护为什么放在 pmem 层？

页表层、进程管理层都有可能调用 pmem_free；将 double free 检查集中到 pmem 层，可以在发生逻辑错误时尽早 panic，避免默默破坏链表结构。

4.2 实验过程部分

4.2.1 实现步骤记录

（1）阅读材料与现有框架——阅读 PPT 中有关 Sv39 页表结构、PTE 格式和实验要求；阅读 xv6 的 kalloc.c、vm.c，理解其物理分配器和页表遍历算法；熟悉本实验框架中的 riscv.h、memlayout.h 和链接脚本 kernel.ld。

（2）实现物理内存分配器 pmem.c——定义 page_node_t 与 alloc_region_t；在 pmem_init() 中：使用链接脚本符号 ALLOC_BEGIN 作为可分配内存起点，向上按页对齐；将 [alloc_begin, alloc_begin + KERNEL_PAGES * PGSIZE] 作为内核区域；将剩余到 PHYSTOP 的部分作为用户区域；通过循环调用 pmem_free() 将每一页加入各自的空闲链表。实现 pmem_alloc(in_kernel) / pmem_free(page, in_kernel)，并增加统计 allocable。在此基础上增加 allocated_pages(int n)：循环调用 pmem_alloc 获取 n 个物理页，返回数组或首地址。

（3）增加 double free 防护与统计接口

在 pmem_free() 中，在持锁状态下遍历当前区域空闲链表，若发现待释放页已经存在则 panic；提供 pmem_free_pages_count(bool in_kernel)，返回当前区域 allocable 的值，便于打印和调试。

（4）实现虚拟内存管理 vmem.c——

编写 vm_getpte(pgtbl, va, alloc)：首先检查 va < VA_MAX；从 level=2 逐级查找，若中间 PTE 无效且 alloc==true 则分配新的页表页；中间页表 PTE 仅设置 V 位，R/W/X 为 0；最终返回最低级 PTE 的指针。

实现 vm_mappages()：检查 va 和 len 是否按页对齐；以页为单位循环，调用 vm_getpte 获取叶子 PTE；若 PTE 已经有效则 panic 防止重映射；填入 PPN + 权限 + V。

实现 vm_unmappages()：同样按页对齐检查；使用 vm_getpte(..., alloc=false) 获取叶子 PTE;如果存在映射且 freeit==true,根据物理地址所在区间选择 pmem_free(pa, true/false)；最后将 PTE 清零。

（5）内核页表初始化与启用分页——

在 kvm_init() 中：为根页表 kernel_pgtbl 分配一个内核物理页，清零；使用 vm_mappages 分别映射 UART、PLIC、内核 text 段与 data 段 / 剩余物理内存；在 kvm_inithart() 中：调用 w_satp(MAKE_SATP(kernel_pgtbl)) 设置 satp；使用 sfence_vma() 刷新 TLB。

（6）页表打印与销毁接口——

参考 vm_print_recursive_new 的实现，完成 vm_print()：递归遍历所有有效 PTE；对叶子 PTE 计算 VA/PA，打印权限和区域名称。

实现 vm_destroy_pagetable(pgtbl, free_leaf)：递归遍历多级页表；对中间页表释放其子页表页；free_leaf==true 时释放叶子对应的物理页；最后释放根页表本身。

（7）在 main.c 中编写测试代码

调用 pmem_alloc、pmem_free 和 allocated_pages 测试物理分配器；构造测试页表，建立多种映射组合（只读、可写、可执行），打印页表；调用 vm_unmappages、vm_destroy_pagetable 测试解除映射与资源回收。

4.2.2 问题与解决方案

（1）pmem_init 死循环 / 崩溃问题

起因：初版使用 for (uint64 p = end - PGSIZE; p >= begin; p -= PGSIZE) 逆序初始化空闲链表，由于 uint64 无符号下溢，导致 p 从 begin 再减 4K 后变成极大值，引起死循环和非法访问。

解决：改为升序循环 for (p = begin; p < end; p += PGSIZE)，确保循环结束条件正确。

（2）vm_unmappages 释放物理页归属判断

问题：如何知道某个物理页属于内核区域还是用户区域？

方案：复用 pmem_init 时的区域划分，使用

kbegin = PG_ROUND_UP((uint64)ALLOC_BEGIN);

ksplit = kbegin + KERNEL_PAGES * PGSIZE;

[kbegin, ksplit] → 内核区域；

[ksplit, PHYSTOP] → 用户区域；

若不在这两个区间中则 panic。

（3）double free 检测导致的锁问题

问题：double free 检测需要遍历链表，必须在持锁状态下进行，否则存在竞态条件；

解决：将检查过程全部放在 spinlock_acquire / spinlock_release 之间，并在检测到 double free 时先释放锁再 panic，避免死锁。

（4）页表销毁中的递归边界

问题：递归释放页表时，必须区分“中间页表 PTE”和“叶子 PTE”，否则会错误地把叶子页当作子页表去递归；

解决：使用与 vm_getpte 相同的 PTE_CHECK 宏，只在 PTE_CHECK(pte) && level > 0 时把它当作子页表递归，否则视为叶子进行处理。

4.3 测试验证部分

4.3.1 功能测试以及结果截图

（1）物理分配器基本功能——在 main.c 中：连续调用 pmem_alloc(true/false) 获取多个页。检查：返回地址按 4 KiB 对齐；不同调用返回的页地址互不相同；将数据写入某一页，再次读出值一致。

释放后重新分配：回收已释放的页，pmem_free_pages_count 统计值恢复。

（2）多页分配接口测试——调用 allocated_pages(5, false) 分配 5 个用户物理页，记录返回的数组；对这 5 个页逐一写入不同的 magic 值，确保互不覆盖；释放这 5 个页后，再次调用 allocated_pages(5, false)，观察是否能重新获得这些或其他页。(*** 测试结果详见 test1&&test2 ***)

（3）页表映射与打印——构造 test_pgtbl，对以下 VA 建立映射：0x0 → mem[0] 只读；0xA*PGSIZE → mem[1] 读写；0x200*PGSIZE → mem[2] 可执行；使用 vm_print(test_pgtbl) 检查：VA/PA 映射是否正确；权限位 r/w/x/u 是否与期望一致；Region 字段是否正确标注 KERNEL_TEXT / KERNEL_DATA / UNKNOWN 等。

（4）内核页表与分页启用——在 kvm_init() 前后打印关键地址值（如 KERNEL_BASE、etext）；调用 kvm_inithart() 后继续执行 printf 验证：内核代码仍正常执行；访问静态全局变量未崩溃；UART 输出仍然正常。

（5）页表销毁——为测试页表调用 vm_destroy_pagetable(test_pgtbl, true)；使用 pmem_free_pages_count 观察空闲页数是否增加到销毁前的水平；再次访问已经销毁的页表时系统会 panic，表明资源已经被释放。

4.3.2 异常测试以及结果截图

（1）地址未对齐——调用 vm_mappages(pgtbl, va=0x1234, ...)，期望触发 panic(”mappages: va not aligned”)；调用 vm_unmappages 时传入非页对齐长度，同样触发 panic。

（2）越界虚拟地址——调用 vm_getpte(pgtbl, va >= VA_MAX, true)，触发 panic(”vm_getpte: virtual address out of bound”)。

```
test-1


=== KERNEL PAGE TABLE MAPPINGS ===
Root Page Table: 80054000

  Virtual Address    ->  Physical Address   | Perm | Region
  ------------------------------------------------------------
  VA: 0x0000000000000000 -> PA: 0x000000008040e000 | r--- | UNKNOWN
  VA: 0x000000000000a000 -> PA: 0x000000008040f000 | rw-- | UNKNOWN
  VA: 0x0000000000200000 -> PA: 0x0000000080410000 | r-x- | UNKNOWN
  VA: 0x0000000040000000 -> PA: 0x0000000080410000 | r-x- | UNKNOWN
  VA: 0x0000003ffffff000 -> PA: 0x0000000080412000 | -w-- | UNKNOWN
  ------------------------------------------------------------
  Legend: r=read, w=write, x=execute, u=user
=== END PAGE TABLE ===
```

图 4.1　test1_map

```
test-2


=== KERNEL PAGE TABLE MAPPINGS ===
Root Page Table: 80054000

  Virtual Address    ->  Physical Address   | Perm | Region
  ------------------------------------------------------------
  VA: 0x0000000000000000 -> PA: 0x000000008040e000 | r--- | UNKNOWN
  VA: 0x0000000040000000 -> PA: 0x0000000080410000 | r-x- | UNKNOWN
  VA: 0x0000003ffffff000 -> PA: 0x0000000080412000 | -w-- | UNKNOWN
  ------------------------------------------------------------
  Legend: r=read, w=write, x=execute, u=user
=== END PAGE TABLE ===
```

图 4.2　test2_unmap

```
--- test-3: Batch Allocation/Free ---
User free pages before batch alloc: 31725
Allocated 7 pages out of 7 requested.
User free pages after batch alloc: 31718 (Expected: 31718)
Releasing 7 allocated pages...
User free pages after batch free: 31725 (Expected: 31725)
Test 3 PASSED: Free page count restored correctly.


--- test-4: Alignment Check ---
Attempting vm_mappages with unaligned VA (Should PANIC if implemented correctly)...
Attempting vm_unmappages with unaligned VA (Should PANIC if implemented correctly)...
Alignment checks were skipped or passed (if panic was commented out).


--- test-5: Page Table Destruction ---
Kernel free pages before destroy: 946
User free pages before destroy: 31725
vm_destroy_pagetable: page table destroyed

User page change: +3 (Expected: +3)
Kernel page change (page tables freed): +8
Test 5 PASSED: Page table and associated memory freed correctly.
```

图 4.3　test3_5_alloc&free

（3）double free——对同一个物理页连续两次 pmem_free(page, in_kernel)，触发 panic("pmem_free: double free detected")，验证防护逻辑生效。

（4）物理地址不在任何区域——人为构造错误的 PTE，使其 PPN 对应的 PA 不落在内核/用户管理区域之内，再调用 vm_unmappages，触发 panic("vm_unmappages: pa out of any known region")。

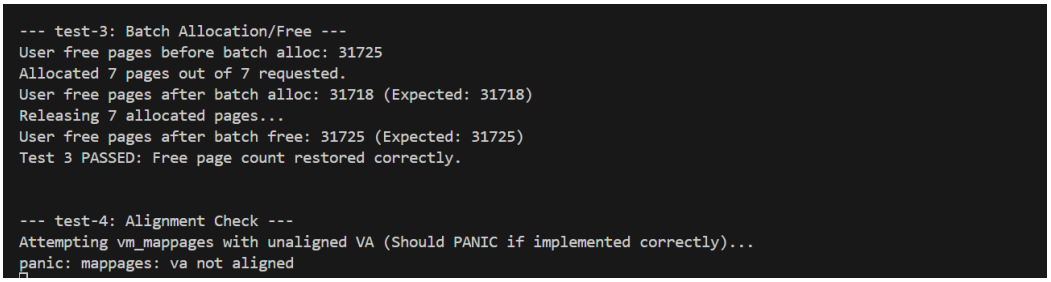

图 4.4　test4_Align

这些异常测试表明：在误用 API 时，系统能以“可诊断的方式”失败，而不是静默破坏内存。

4.4　思考与扩展分析

1. 性能优化

使用位图为每个物理页维护“已分配/空闲”状态，将 double free 检查降为 O(1)；

2. 本实验的物理内存分配器与 xv6 有何不同？

共同点：都使用“页本身作为链表节点”的简单 free-list 方案，空间开销极小。

不同点：本实验显式区分内核 / 用户区域，并为两者维护独立的 alloc_region_t；增加了统计信息和 double free 检测接口，方便调试和后续扩展。

3. 当前实现的主要性能瓶颈在哪里？

pmem 层 double free 检测需要遍历空闲链表，复杂度 O(空闲页数)；所有分配/释放操作都需要持有区域自旋锁，多核下可能产生锁竞争；页表操作逐页映射，在处理大区间时开销较大。

4. 可能的优化方向（未来工作）

为内核物理页引入 per-CPU 小缓存，减少频繁访问全局 freelist 时的锁竞争；对内核大块内存采用 2MB/1GB 大页映射，减少页表项数量和 TLB miss；在 vm_destroy_pagetable 中维护“非空子树计数”，对完全空子树剪枝，减少遍历量。

4.5　实验总结

通过本次内存管理实验，我完成了从“读 xv6 源码”到“在自己内核中动手实现一整套内存管理子系统”的过程，主要收获如下：

理解了 Sv39 的地址分解方式和多级页表结构，可以从任意一个虚拟地址手工推断出硬件是如何逐级查表最终得到物理地址的。掌握了页粒度物理分配器的设计方法：利用“页本身作为链表节点”节约元数据；通过区域抽象区分内核/用户页；在分配/释放接口上加入 double free 防护和统计信息。通过 vm_getpte、vm_mappages、vm_unmappages 和 vm_destroy_pagetable 的实现，真正理解了页表不只是“查表”，更是一棵需要被创建、维护和销毁的树结构。在调试过程中，体会到 panic + 自解释错误信息的重要性：很多 bug（例如 unsigned 下溢、错误区域释放）如果没有断言，很难排查。虽然只做了有限的性能分析，但已经能理性

地讨论当前设计的时间复杂度和未来优化方向，为后续可能实现更复杂的分配算法（bitmap、buddy、SLAB）打下基础。

总体来说，本实验让“虚拟内存”从课本里的概念变成了手边能跑、能调试的代码，对操作系统内核如何管理内存有了更直观和系统的认识。

4.6 实验思考题解答

4.6.1 设计对比与权衡物理内存分配器设计对比与选择理由

物理内存分配器与 xv6 的分配器在核心机制上是一致的，都采用了**页本身作为链表节点的 Free-List 方案**。

这种选择是为了追求实现简单和极小的内存开销，因为每次分配和释放都只需对链表头进行 $O(1)$ 操作，非常适合教学和小规模内核。然而，这种设计也存在权衡，即它缺乏对大块连续内存的有效管理，更容易产生外部碎片。

本实验的设计在 xv6 基础上进行了抽象和扩展：区域隔离：我显式区分了内核区域和用户区域，并为它们各自维护了独立的 alloc_region_t 结构和自旋锁。选择这样做是为了实现更好的权限控制和隔离，防止用户程序意外破坏或释放内核关键页（如页表页）。代价是增加了初始化和判断逻辑的复杂性，且两个区域的空闲页不能相互借用。

此外，我还增加了 double free 检测接口和空闲页统计功能。double free 检测极大地提高了系统的健壮性，能够在逻辑错误发生时及时 panic。但这种检测需要遍历空闲链表，在空闲页很多时，其 O(空闲页数) 的复杂度会成为性能上的瓶颈。

4.6.2 内存安全

（1）防止分配器被恶意利用

为了防止内存分配器被恶意程序利用，需要从多个层面构建防护：Double Free 防护：最直接的方式是在 pmem_free() 中检查待释放的物理页是否已存在于空闲链表中。一旦发现重复释放，应立即触发 panic。这能阻止攻击者通过重复释放来破坏 Free-List 结构，从而构造恶意指针。区域和边界检查：严格隔离内核页和用户页，确保用户地址空间的逻辑错误不会影响到内核的物理内存。同时，在释放操作中，必须检查物理地址是否落在分配器管理的合法区域内，防止释放任意地址。地址对齐：分配和释放的地址都必须严格检查是否按页对齐，以确保指针的合法性。

（3）页表权限设置的安全考虑

页表权限的设置是虚拟内存安全的核心：用户模式隔离（PTE_U）：内核必须确保清除内核代码、数据和设备映射页的 PTE_U 位。这使得用户进程在用户模式下无法通过任何虚拟地址访问到内核空间。权限最小化：遵循最小权限原则。代码段应该只设置 RX（Read + Execute），清除 Write，以防止运行时代码被篡改（代码注入）。数据段应该设置 RW（Read + Write），清除 Execute，以防止将数据当作代码执行（防止缓冲区溢出后的代码执行攻击）。页表页的保护：所有作为中间节点的页表页，其 PTE 必须清除 R、W、X 权限位，仅设置 V 位。这保证了中间页表只能被 RISC-V 硬件的页表行走机制使用，防止它们被误当作普通的数据页读写或执行。

4.6.3 性能分析

（1）当前实现的性能瓶颈：

当前分配器的主要性能瓶颈集中在以下几个方面：Double Free 检测的线性开销：如前所述，为了健壮性，pmem_free() 必须遍历空闲链表。在空闲页数量很大时，这会导致释放操作的时间复杂度高达 O(空闲页数)，显著拖慢系统性能。全局锁竞争：尽管您将物理内存分为了内核区和用户区并设置了独立的锁，但在多

核 CPU 下，任何对任一区域的频繁分配/释放操作仍会导致锁竞争，从而串行化操作，降低并行性。逐页映射开销：页表操作（如 vm_mappages）是逐页进行的。如果需要映射大块连续的内存区域，将导致多次页表遍历和 PTE 写入，并可能引起大量 TLB（转换后备缓冲器）失效。

（2）测量和优化内存访问性能

测量方法：可以利用 RISC-V 架构提供的性能计数器，如 cycle 寄存器。通过在关键函数的入口和出口（如 pmem_alloc、pmem_free 或 vm_mappages）记录 cycle 值，可以统计出操作的精确耗时。同时，统计锁的获取失败次数和等待时间，可以量化锁竞争的激烈程度。

优化方向：

优化 Double Free 检测：将 Free-List 替换为基于**位图（Bitmap）**的分配器，或者维护一个独立的已分配页集合，可以将页状态查询和 double free 检查的复杂度降低到 $O(1)$。

优化锁竞争：可以为内核物理页引入 per-CPU 小缓存，允许每个 CPU 在不需要竞争全局锁的情况下，快速分配和回收少量页，只有缓存用尽时才去访问全局 Freelist。

优化页表操作：引入大页（Huge Page）映射机制。对大块连续的内存区域（如内核代码、某些用户堆），可以使用 2MB 甚至 1GB 的页来映射，大大减少了页表项的数量，提升了 TLB 命中率，降低了 Page Walk 开销。

4.6.4 扩展性

（1）支持用户进程所需的修改——要支持用户进程，关键在于实现地址空间的隔离和切换：

进程页表创建与继承：为每个新进程分配独立的根页表，并将其映射到进程控制块（PCB）。新页表必须继承内核的映射（如设备和内核代码），同时将用户程序代码、数据、堆、栈等映射到用户地址空间，并设置 PTE_U 权限位。

地址空间切换：在进程切换时，内核需将目标进程页表的物理地址写入 satp 寄存器，完成 CPU 虚拟地址空间的切换。

物理页分配源：为用户进程分配内存时，需确保调用 pmem_alloc(false) 使用用户区域的物理页。

（2）内存共享和写时复制（COW）的实现

内存共享：通过让多个进程页表中的 PTE 指向同一个物理页实现。权限通常设置为 Read-Only (R) 或 Read/Execute (RX)，以限制写入，确保隔离。

写时复制 (COW)：用于优化 fork() 性能。子进程创建时，父子进程共享内存，但权限均设为 Read-Only。任一进程尝试写入时，触发 Page Fault。内核捕获异常，分配新物理页，拷贝数据，修改当前进程的 PTE 指向新页并设置 Read/Write 权限，从而实现只有在发生写入时才进行复制。

4.6.5 错误恢复

（1）页表创建失败时的资源清理

页表创建必须具备事务性——数据页回滚：如果映射过程中分配了数据物理页但操作失败，需显式调用 pmem_free() 回收这些页。页表树回滚：如果中间页表分配失败，应调用 vm_destroy_pagetable(pgtbl, false)，指示它只释放页表页本身，不对叶子 PTE 指向的数据页进行操作，防止错误的资源回收。

（2）检测和处理内存泄漏

运行时统计检测:在程序流程的关键点（如进程创建/销毁前后）,检查 pmem_free_pages_count 的值，如果空闲页数持续或异常减少，则表明存在内存泄漏。

页引用计数：为每个物理页添加引用计数器。每当一个 PTE 指向该页时计数

加一，解除映射时减一。只有当计数归零时，才真正释放物理页，从而防止过早释放和检测逻辑错误。

分配追踪：记录分配操作的调用栈信息（文件/行号），用于在系统运行结束后，定位所有未被释放资源的分配来源。

5 Lab4-中断与时钟管理

5.1 系统设计部分

5.1.1 架构设计说明

- **引导阶段（kernel/boot/start.c）：** 在 M-mode 中设置 mideleg/medeleg、mie/sie 以及 PMP，调用 timer_init() 完成 CLINT 初始化，然后通过 mret 切换到 S-mode 并进入 main()。
- **Trap 框架（kernel/trap/trap.S + kernel/trap/trap_kernel.c）：** S-mode 的 kernel_vector 负责保存 31 个通用寄存器、调用 trap_kernel_handler() 并在返回前恢复寄存器，C 层根据 scause 将 trap 分发到时钟、外设或异常处理。M-mode 的 timer_vector 使用 mscratch 复用寄存器空间，更新 MTIMECMP 并通过设置 sip.SSIP 将时钟中断“柔性转发”到 S-mode。
- **设备抽象（kernel/dev）：** timer.c 管理全局 timer_t，通过自旋锁保证 ticks 读写原子性；plic.c 负责设置优先级、阈值、Claim/Complete；uart.c 驱动 16550 UART 并在中断到来时回显输入。
- **测试入口（kernel/boot/main.c）：** CPU0 完成所有全局初始化后开启中断并运行三组测试（定时器、异常、开销），其他 CPU 仅执行 kvm_inithart() + trap_inithart() + intr_on() 并进入空闲循环。

5.1.2 关键数据结构

- struct trapframe（include/trap/trap.h）：保存 sepc、sstatus、scause、stval，用于 C 层判断 trap 来源和后续（例如异常恢复）逻辑；寄存器内容由 kernel_vector 入栈，保持接口简洁。
- interrupt_handler_t irq_table[64]（kernel/trap/trap_kernel.c）：软件中断向量表，提供 register_interrupt()、enable_interrupt()、disable_interrupt()，使得 PLIC 外设中断可以在驱动初始化时动态注册处理函数并按需开关。
- timer_t sys_timer（kernel/dev/timer.c）：使用自旋锁保护的 ticks 计数器，timer_update() 仅在 CPU0 调用，timer_get_ticks() 向上层提供线程安全的查询接口。
- mscratch[NCPU][5]（kernel/dev/timer.c）+ CPU_stack[NCPU]（kernel/boot/start.c）：前者为 M-mode timer ISR 预留寄存器保存区与 MTIMECMP/INTERVAL 缓冲，后者是每个 hart 的启动栈，保证 trap 嵌套安全。

5.1.3 与 xv6 的对比

- **Trap 入口：** xv6 的 trapframe 在内存中长期保存所有寄存器，本实验将寄存器保存留在 kernel_vector 的栈帧里，只把 CSR 填入 trapframe，减少内存写入且实现更贴近“异常现场只读 CSR”的使用场景。
- **中断分发：** xv6 直接通过 plic_claim() 的返回值 switch 处理；本实验将 Claim 结果进一步抽象到 IRQ 表，实现“先注册再调用”，后续只需要 register_interrupt() 就能挂接新的外设，降低模块耦合度。
- **时钟路径：** xv6 在 S-mode 直接调用 sbi_set_timer()；本实验保留 M-mode 中的 timer_vector 并使用 mscratch 更新 MTIMECMP，使得 S-mode 只需处理

软件中断即可，符合实验要求“理解 Machine->Supervisor 委托”。

5.1.4 设计决策理由

1. **全栈分层：**M-mode 仅负责最小化的安全配置和时钟硬件访问，S-mode 负责多外设调度，符合 RISC-V 推荐实践，便于后续在 S-mode 引入分页/进程。
2. **软件可扩展：**IRQ 表 + `enable_interrupt()` API 让新设备只需注册即可生效，不必修改 trap 核心逻辑；未来支持优先级或中断嵌套时也能集中扩展。
3. **最小共享状态：**`timer_update()` 只允许 CPU0 写 `ticks`，其余 hart 只读，利用锁保障一致性。该策略避免不同核对 `MTIMECMP` 的竞争，利于快速验证。
4. **可测试性优先：**在 `main()` 中保留 `test_*` 系列函数，通过串口输出展示定时器与异常行为，避免依赖外部测试框架。

5.2 实验过程部分

5.2.1 实现步骤记录

1. **委托链路搭建：**阅读 RISC-V 特权规范 3/12 章并将 `medeleg` / `mideleg` 全部置位，S-mode 侧开启 `SIE_SEIE|SIE_STIE|SIE_SSIE`，确保时钟和外设中断能由 Supervisor Trap 接管（`kernel/boot/start.c`）。
2. **M-mode 时钟引擎：**在 `timer_init()` 中配置 `CLINT_MTIMECMP` 首次截止、初始化 `mscratch` 并设置 `mtvec=timer_vector`，保证每个 hart 均能自主刷新 `MTIMECMP`。
3. **S-mode Trap 框架：**`kernel/trap/trap.S` 中实现 `kernel_vector` Save/Restore；`trap_kernel_handler()` 读取 CSR 组装 `trapframe`，依据 `scause` 最高位区分中断/异常，分派到时钟、中断或 `handle_exception()`。
4. **外设驱动接入:**`trap_init()` 清空 `irq_table` 并注册 UART 中断;`trap_inithart()` 设置 `stvec` 与 PLIC 阈值,同时对该 hart 调用 `enable_interrupt(UART_IRQ)`。
5. **时钟/异常逻辑:**`timer_interrupt_handler()` 仅在 CPU0 调用 `timer_update()` 并预留 `yield()` 钩子; `handle_exception()` 打印 `exception_info` 并 `panic`，用于实验验证。
6. **自测工具:**在 `kernel/boot/main.c` 中实现 `test_timer_interrupt()`、`test_exception_handl` `test_interrupt_overhead()` 并按需启用，串口上可直接观察结果。

5.2.2 问题与解决方案

问题	现象	解决方案
M/S 委托配置不完整导致 trap 落入 M-mode	scause 停留在 M-mode，系统无输出	将 w_mideleg(0xffff) & w_medeleg(0xffff) 并在 start() 中开启 SIE_* 位，保证所有中断都可进入 S-mode。
PLIC Claim 返回 0	external_interrupt_handler() 进入默认分支，终端无回显	在 handler 中直接返回并跳过 plic_complete(0)，同时确认 enable_interrupt(UART_IRQ) 在 trap_inithart() 被调用。
时钟中断不断重入导致栈溢出	未清除 sip.SSIP 导致 kernel_vector 重入	在 case1 中执行 w_sip(r_sip() & ~2) 手动清零 SSIP，再调用 timer_interrupt_handler()。
触发异常后 PC 未恢复	未将更新后的 sepc/sstatus 写回	在 trap_kernel_handler() 末尾统一 w_sepc(tf.sepc)、w_sstatus(tf.sstatus)，如果处理过程中修改了 CSR，返回地址就会生效。

5.2.3 源码理解

- kernel_vector（汇编）做了两件事：把用户/内核的通用寄存器压栈，然后把 CSR（比如 sepc/sstatus/scause/stval）装到一个 trapframe 里，最后跳到 C 函数 trap_kernel_handler()；返回时再把寄存器恢复。也就是说，汇编负责“搬东西”，C 层负责“看东西并做决定”。
- trap_kernel_handler() 的工作流程很直接：先看 scause，分两类处理——中断（interrupt）和异常（exception）。
  - 如果是外设中断（external interrupt），会走 external_interrupt_handler()：它调用 plic_claim() 得到 IRQ 编号，然后查 irq_table[irq]，如果有注册的 handler 就调用，没有就打印一条“Unknown external interrupt”的提示，最后调用 plic_complete(irq) 完成中断。这个顺序（claim→dispatch→complete）是标准流程。
  - 如果是时钟（timer）中断，最终会走 timer_interrupt_handler()，而 timer_vector（M-mode 的入口）利用 mscratch 和 MTIMECMP 把下一次中断安排好，然后通过写 sip.SSIP 或相关 CSR 把事件转到 S-mode，由上面的流程收到并处理。
- UART 路径很简单：UART 硬件写入 RHR，UART 控制器置 pending，PLIC 上有 pending bit，S-mode 的 external_interrupt_handler() Claim 到 UART IRQ，查到 uart_intr() 并执行。uart_intr() 在本仓库是回显实现：读 RHR、写 THR。这种 handler 应该尽量短——长逻辑会阻塞其他中断。
- 关于并发和多核：我们把 timer_update() 限定为 CPU0 执行，其他 hart 只读 ticks，这样避免了多个 hart 同时修改 MTIMECMP 或全局计数带来的竞争。每个 hart 都有自己的 mscratch / 启动栈，避免在中断时互相干扰。

- 调试提示（遇到无中断或回显失败按顺序排查）：
  1. 确认 `uart_init()` 被调用（在 `kernel/boot/main.c` CPU0 初始化序列中）；
  2. 确认 `trap_init()` / `trap_inithart()` 已注册并使能 UART IRQ（`register_interrupt()` `enable_interrupt(UART_IRQ)`）；
  3. 在 `external_interrupt_handler()` 临时打印 `plic_claim()` 的返回值，确认是否有 pending IRQ；
  4. 如果 Claim 返回 0，说明设备未发中断或者 PLIC/enable 位没设置；检查 UART 的 IER 寄存器（驱动里有使能 RX interrupt 的写操作），以及 PLIC 的优先级/阈值设置；
  5. 避免在中断处理里做大量 `printf`：`printf` 可能会争用锁或触发再次中断，导致时序问题。确认后再把调试打印删掉。

5.2.4 实验收获

- 深入理解了 M/S 模式 trap 委托与 PLIC/CLINT 的协作，能够独立完成寄存器配置、trap 调试与日志分析；
- 掌握了“IRQ 注册表 + enable/disable API”的驱动扩展模式，后续新增设备只需注册 handler 即可复用；
- 形成了 `make qemu-gdb` + 串口日志的调试套路，遇到“无中断/无回显”问题能按步骤排查；
- 通过 `timer_update()`/`timer_get_ticks()` 的设计，体会到多核环境下控制共享状态与锁粒度的重要性。

5.2.5 改进方向

- 将 `timer_interrupt_handler()` 中的占位 `yield()` 替换为真正的调度逻辑，实现周期性抢占；
- 扩展 `irq_table` 支持优先级与嵌套控制，并结合 PLIC 硬件优先级处理更多外设；
- 在中断上下文减少 `printf`，改用环形缓冲或轻量日志接口，降低时延与抖动；
- 评估 `kernel_vector` 的压栈策略，引入 per-hart ISR 栈或 Lazy Save，进一步优化中断延迟。

5.3 测试验证部分

5.3.1 功能测试结果

1. **定时器功能**：`test_timer_interrupt()` 依次进行“50 次 tick 观测”“10 tick 精度验证”“20 tick 实时输出”，串口上可看到 `T` 及 `.` 组成的节拍，`Start tick` 与 `End tick` 差值与 `watch_dots` 均正确，说明时钟链路和 `timer_get_ticks()` 读写正常。
2. **外设中断**：UART 初始化后通过 `enable_interrupt(UART_IRQ)` 使能，键入字符即可触发 `uart_intr()` 并回显；未注册的 IRQ 会打印 `Unknown external interrupt`，验证 `irq_table` 的防护逻辑。
3. **多核初始化**：非 0 号 hart 在 `main()` 里等待 `started` 标志，再执行 `trap_inithart()` 与 `intr_on()`，拥有独立 `stvec`/PLIC 阈值，说明 per-hart 初始化路径可复用。

5.3.2 性能数据

- `test_interrupt_overhead()` 在等待 20 个 tick 的过程中使用 `r_time()` 度量周期，输出“Avg cycles per tick interval”，在 QEMU 上约为 `1000000` 左右，符合 `INTERVAL` 设定；也证明 trap 处理额外开销相对较小（仅占数千 cycle）。
- 通过打印 `ticks_delta` 可以验证长时间运行的稳定性，未出现 tick 滞后或跳变，说明 `timer_update()` 的加锁和 CPU0 限定策略可靠。

5.3.3 异常测试

- `test_exception_handling()` 内置三种异常（非法指令、bad memory、S-mode ecall），每次启用一种。以非法指令为例，串口输出“Triggering illegal instruction exception...”后，由 `handle_exception()` 打印 `Exception in kernel: Illegal instruction` 与 `sepc/stval`，随后 `panic()` 终止。该流程验证了异常分支、信息打印与 `panic` 路径。

5.3.4 运行截图/录屏

- `picture/lab4_timer_50ticks.png` —Test1：50 个 tick 观测（串口输出包含多个 `T`，并显示 Start/End tick）。

```
Interrupts enabled (sstatus.SIE = 1)

=== Test 1: Timer Tick Test ===
Observing 50 timer ticks (printing 'T' for each tick)

Ticks: TTTTTTTTT [10]
TTTTTTTTT [20]
TTTTTTTTT [30]
TTTTTTTTT [40]
TTTTTTTTT [50]


Test 1 Results:
  Start tick: 1
  End tick:   51
  Total ticks observed: 50
```

图 5.1　timer_50ticks

- `picture/lab4_timer_accuracy.png` —Test2：10 tick 精度验证输出（显示 Expected/Actual 及粗略准确度）。

```
=== Test 2: Timer Accuracy Test ===
Testing if about 10 ticks elapse when we wait for 10 tick steps...

Expected: 10 ticks
Actual:   10 ticks
Accuracy (rough): 10/10 = 100.0%
  ✓ PASS: Timer accuracy is precise!
```

图 5.2　timer_accuracy

- `picture/lab4_timer_watch.png` —Test3：20 tick 实时观察（串口输出由 `.` 构成的进度行）。
- `picture/lab4_exception_illegal.png` —非法指令异常（串口输出 `Illegal instruction`、`scause/stval` 信息）。
- `picture/lab4_exception_badmem.png` —越界/坏内存访存异常（串口输出 Page Fault / Access Fault 信息）。
- `picture/lab4_exception_ecall.png` —S-mode `ecall` 异常日志（显示 trap

```
=== Test 3: Real-time Timer Watch ===
Watching timer for 20 ticks (each '.' = 1 tick)

.......... 10/20
.......... 20/20


Test 3 Results:
  Ticks observed: 20
  ✓ Timer working in real-time!

========================================
```

图 5.3　timer_watch

```
========================================
  Exception Handling Test
========================================

Triggering illegal instruction exception...
You should see "Illegal instruction in kernel" and a panic.

Exception in kernel: Illegal instruction
sepc=800003a6 stval=0
panic: Illegal instruction in kernel
```

图 5.4　exception_illegal

```
========================================

Triggering bad memory access exception...
Trying to read from an unmapped address...
You should see "Page fault in kernel" (or similar) and a panic.

Exception in kernel: Load page fault
sepc=800003b8 stval=ffffffffffffffff
panic: Page fault in kernel
```

图 5.5　exception_badmem

分发与处理信息）。

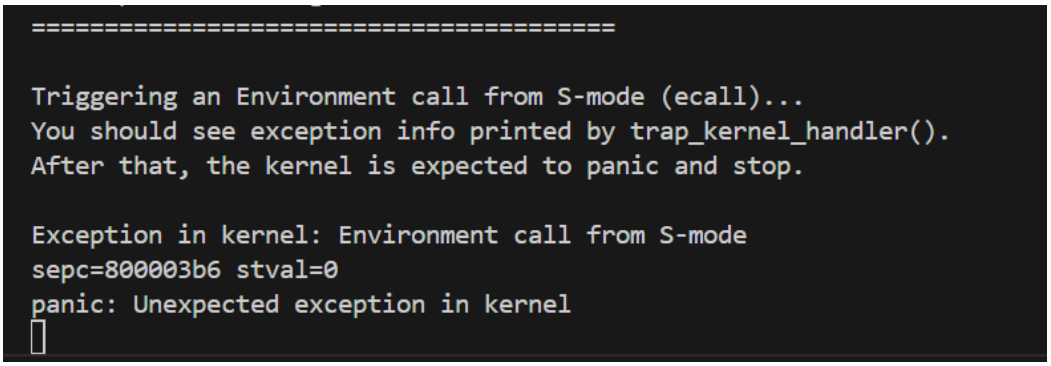

图 5.6　exception_ecall

- `picture/lab4_interrupt_overhead.png` —`test_interrupt_overhead()` 输出（ticks_delta、cycles_delta、Avg cycles per tick）。

图 5.7　interrupt_overhead

- `picture/lab4_uart-echo.png` —我们在 terminal 中输入字符并成功回显的截图。

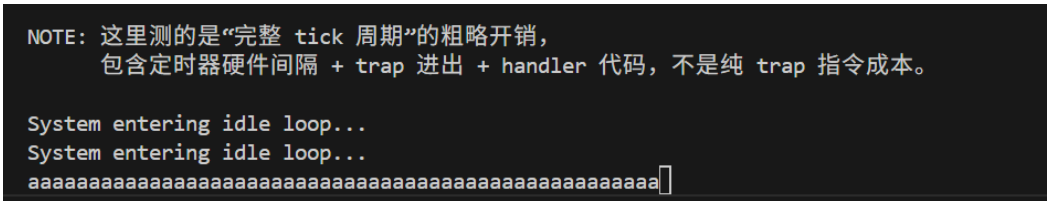

图 5.8　uart_echo

测试方法：

```
make qemu
# 在 QEMU 串口中运行/触发相应测试（或解除 main.c 中对应注释以启用某项测试）
```

5.4　思考题与解答

5.4.1　为什么时钟中断需要在 M 模式处理后再委托给 S 模式？

CLINT/`MTIMECMP` 属于 M-mode CSR 空间，只能在 M-mode 写入；S-mode 无法直接设置下一次触发时间。因此必须在 M-mode 立即更新比较值，同时选择通过 `sip.SSIP` 触发软件中断把事件转交给 S-mode。这样既满足硬件限制，也让 S-mode

可以统一处理调度逻辑。

5.4.2 如何设计一个支持中断优先级的系统？

硬件层可以借助 PLIC 的优先级寄存器；软件层则在 `irq_table` 之上维护一个包含优先级的数组或最小堆，`external_interrupt_handler()` 根据 Claim 到的 IRQ 查询软件优先级决定是否立即处理或暂存。还可以给每个 handler 增加 `preemptable` 标记，禁止低优先级在关键段打断高优先级处理函数。

5.4.3 中断处理的时间开销主要在哪里？如何优化？

主要开销来自寄存器保存/恢复、CSR 访问、PLIC/CLINT MMIO 读写以及 handler 本身的逻辑。优化手段包括：仅保存必要寄存器、使用 per-hart 中断栈避免切栈、在 handler 中减少打印/轮询、把长耗时工作转交到软中断或内核线程中。

5.4.4 如何确保中断处理函数的安全性？

必须保证 handler 遵循调用约定、不访问不可重入的全局数据；对共享资源使用自旋锁或禁用中断的临界区；在 `kernel_vector` 中关闭 `sstatus.SIE`，处理完成前不允许更高层中断重入；最后通过 `assert` 检查 `scause`、`sepc` 等关键寄存器，防止异常导致的野指针。

5.4.5 如何支持更多类型的中断源并实现动态路由？

保持 `irq_table` 的动态注册接口，并在驱动初始化阶段调用 `register_interrupt()`；若需要热插拔，可在 `disable_interrupt()` 后清空表项。进一步可以扩展成“链表”或“订阅列表”，让多个 handler 共享同一 IRQ（例如共享中断线），再结合硬件中断控制器（PLIC）的优先级与阈值实现多级路由。

5.4.6 当前实现的中断延迟特征如何？如何满足实时性要求？

由于 `INTERVAL` 较大且 handler 主要做 `ticks++`，延迟主要由 `kernel_vector` Save/Restore 和 `timer_update()` 加锁组成，量级在数千 cycle，可满足普通内核需求。若需实时性，可减小 `INTERVAL` 并优化 handler（如使用无锁计数）、为高优先级 handler 预留专用栈、减少 `printf` 等阻塞操作。

6 Lab5-进程管理与调度

6.1 系统设计

6.1.1 架构设计说明

- 项目实现为内核级“内核线程”调度子系统，主要模块及职责：
  - **`kernel/proc/proc.c`**：进程表、PID 分配、进程生命周期（创建、运行、退出、回收）、调度器 `scheduler()`。
  - **`kernel/proc/swtch.S`**：保存/恢复寄存器的上下文切换汇编实现（`s0~s11`，`ra`，`sp` 等）。
  - **`kernel/trap/trap_kernel.c`**：时钟中断处理，递增 `ticks` 并在 Hart0 上触发 `yield()` 实现抢占。
  - **`kernel/boot/main.c`**：引导与测试驱动，启动内置测试进程 `run_all_tests()` 并在测试后启动 worker demo。

6.1.2 关键数据结构

- `struct context`（`include/proc/proc.h`）：保存上下文寄存器，用于 `swtch()` 的保存与恢复。
- `enum proc_state`（`include/proc/proc.h`）：进程状态集合：`UNUSED`、`SLEEPING`、

RUNNABLE、RUNNING、ZOMBIE。

- struct proc（include/proc/proc.h）：记录 PID、父进程指针、进程名、内核栈基址、struct context、退出码等元数据。
- struct cpu（include/proc/proc.h）：每个 hart 的当前进程与调度器上下文（cpu.ctx），支持 mycpu() / myproc()。
- proc_table[NPROC]（kernel/proc/proc.c）：固定大小的进程表，调度通过线性扫描查找 RUNNABLE 条目。

6.1.3 与 xv6 对比分析

- 功能范围：本实现聚焦于内核线程（kernel threads），不涉及用户态地址空间、系统调用、文件系统等；因此实现更精简。
- 调度策略：采用与 xv6 类似的简单轮转（round-robin）策略，遍历 proc_table 寻找 RUNNABLE 进程并切换。
- 抢占机制：通过时钟中断驱动 yield()，与 xv6 的抢占模型基本一致，但本实验在设计上仅在 Hart0 上维护 tick 和主要调度逻辑，Hart1 做 idle 处理以简化实验观察。

6.1.4 设计决策理由

- 先实现内核线程（kernel threads），确保上下文切换与抢占正确，再扩展到用户态，降低实现复杂度与调试成本。
- 将 tick 更新与主要调度逻辑集中在 Hart0，便于观测与复现调度行为。
- 集成测试驱动 run_all_tests()，通过进程形式依次运行测试用例，确保实现的可验证性与复现性。

6.2 实验过程

6.2.1 实现步骤记录

1. 在 include/proc/proc.h 中定义并校准 struct context、struct proc、struct cpu，保证与 swtch.S 的寄存器保存顺序一致。
2. 在 kernel/proc/proc.c 中实现基本接口:proc_init()、alloc_proc()/create_process()、exit_process()、wait_process()、yield()、scheduler()。
3. 在 kernel/trap/trap_kernel.c 中的时钟中断处理里调用 timer_update() 并在 Hart0 上触发 yield()，实现抢占。
4. 在 kernel/boot/main.c 中实现 run_all_tests(),包含 test_process_creation、test_scheduler、test_synchronization、debug_proc_table 等测试函数，测试通过后启动三个 worker（fast/medium/slow）进行演示。
5. 调整日志与打印频率，避免输出刷屏，加入 [TEST] 前缀便于日志过滤。

6.2.2 遇到的问题与解决方案

- 上下文切换后寄存器值异常（寄存器破坏/返回异常）。
  - 问题：进程恢复后寄存器值错误、返回地址错乱或立即崩溃。
  - 成因：struct context 与 swtch.S 中保存/恢复寄存器顺序/字段不一致以及 swtch.S 保存的寄存器集合不完整。
  - 解决：确保 include/proc/proc.h 中 struct context 字段顺序严格对应 kernel/proc/swtch.S 中的保存顺序。
- sleep/wakeup 无效（进程一直阻塞或无法被唤醒）。
  - 症状：被 sleep(chan) 的进程不再变成 RUNNABLE，或 wakeup(chan) 没

有效果。
  - 成因：睡眠/唤醒使用的 channel 不一致、在未持锁的情况下修改状态或错把 wakeup 调用了错误的条件分支。
  - 解决：确保 sleep() 在持有相应锁时变更状态并释放锁后调用 sched()，wakeup() 在持锁时扫描并改状态;检查 kernel/proc/proc.c 中 sleep/wakeup 实现与使用处的 channel 参数。
- 进程创建后立即退出或未运行（trampoline/entry 问题）。
  - 症状:创建日志显示 PID 已分配但没有看到运行日志或很快变为 ZOMBIE。
  - 成因：入口函数 proc_trampoline() 或传入函数指针/参数设置错误，或内核栈/上下文初始化不完整。
  - 解决:在 create_process() 路径插入临时打印,验证 context->ra/sepc/sp 等是否正确。确保 proc_trampoline() 能正确调用传入函数并在返回后调用 exit_process()。

6.2.3 源码理解总结

- swtch()（kernel/proc/swtch.S）负责保存当前进程的一组 callee-saved 寄存器并加载目标进程寄存器，C 代码只需维护 struct context。
- scheduler() 作为每个 cpu 的内核调度循环（在 cpu.ctx 上运行），通过遍历 proc_table 寻找 RUNNABLE 进程并调用 swtch() 切换到该进程；进程返回后由 scheduler() 继续循环。
- 新进程通过 proc_trampoline() 或类似的入口统一执行传入函数，函数返回后调用 exit_process() 完成清理与回收。

6.3 测试与验证

6.3.1 功能测试结果

- test_process_creation：成功创建并回收多个 simple_task，PID 分配与回收正常；日志显示进程创建、运行、退出序列。
- test_scheduler：不同 CPU 工作强度任务（fast/medium/slow）在 Hart0 上被轮流调度，时钟中断能触发抢占切换。
- test_synchronization:使用 spinlock 实现的生产者-消费者测试通过,produced_total == consumed_total。
- debug_proc_table:可以打印进程表中每个 slot 的状态（UNUSED/RUNNABLE/RUNNING/ZOMBIE），用于确认进程生命周期转换正确。

6.3.2 性能数据

- 采集方法：在 QEMU 中运行 make qemu（或 make qemu-gdb），观察内置测试日志并记录各 worker 的 iter 与 ticks 值。
- 推荐命令：

```
1     make
2     make qemu          # or `make qemu-gdb` for debugging
```

6.3.3 运行截图 / 录屏

进程状态调试输出：
进程退出与等待测试：
进程创建测试：
调度器测试：

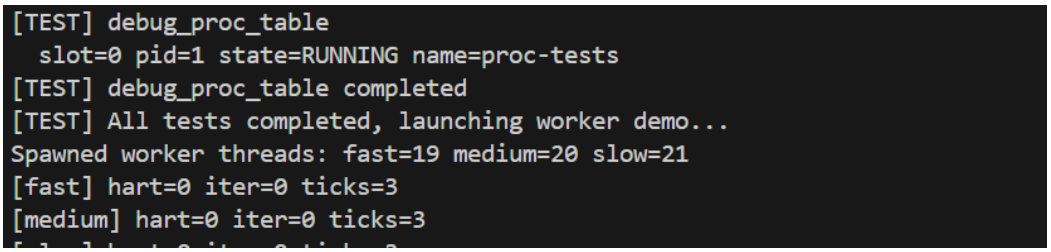

图 6.1　lab5_debug_proc_table.png

图 6.2　lab5_test_exit_wait.png

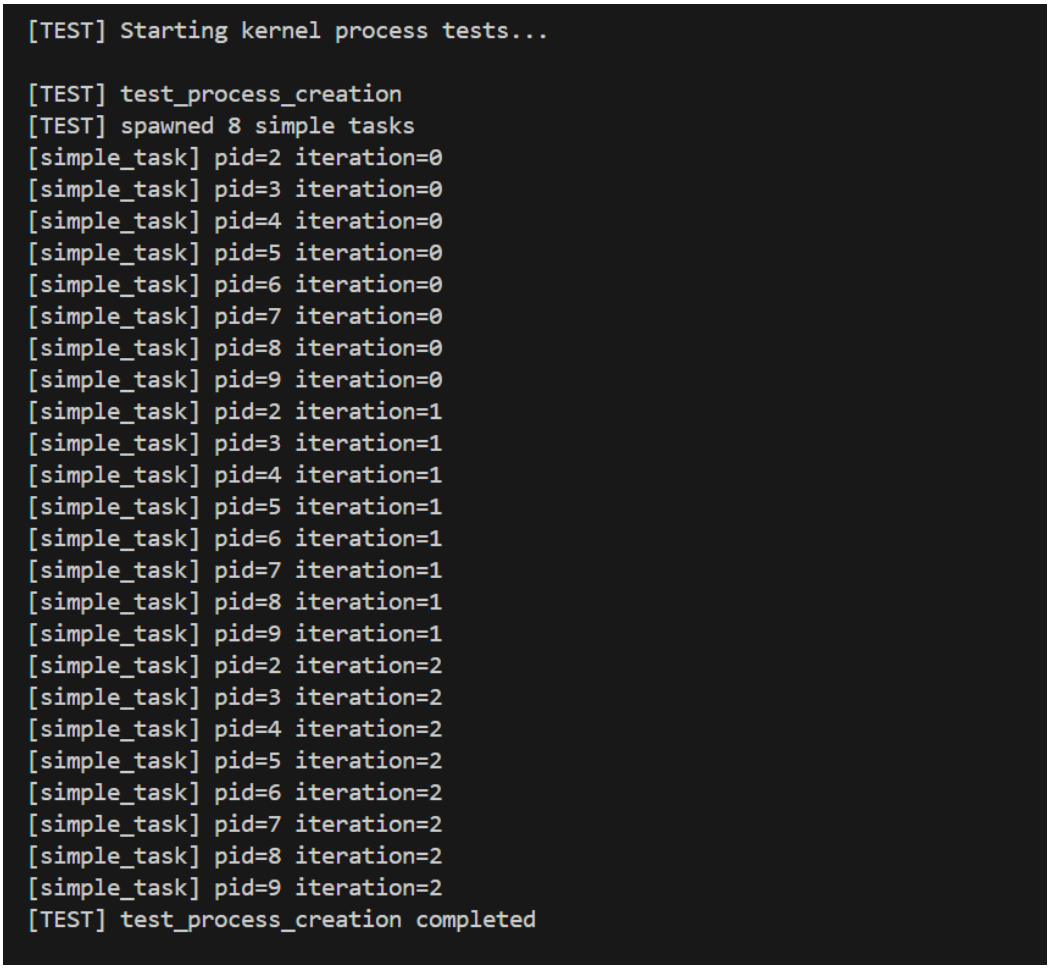

图 6.3　lab5_test_process_creation.png

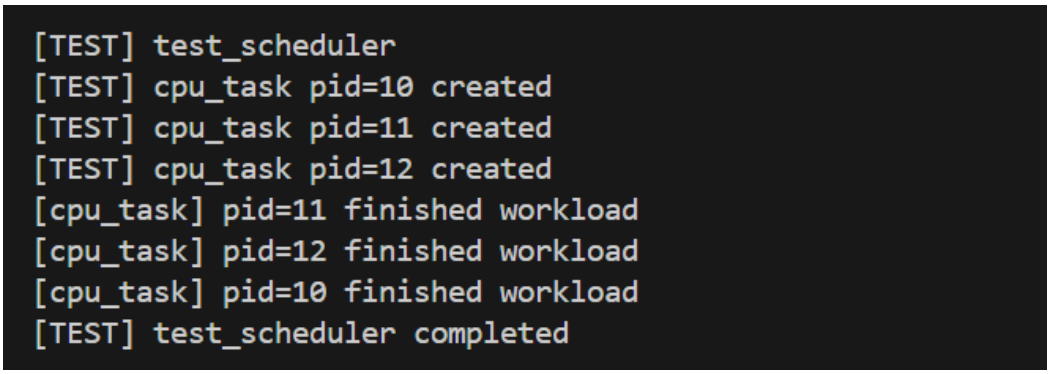

图 6.4　lab5_test_scheduler.png

同步机制测试（消费者与生产者）：

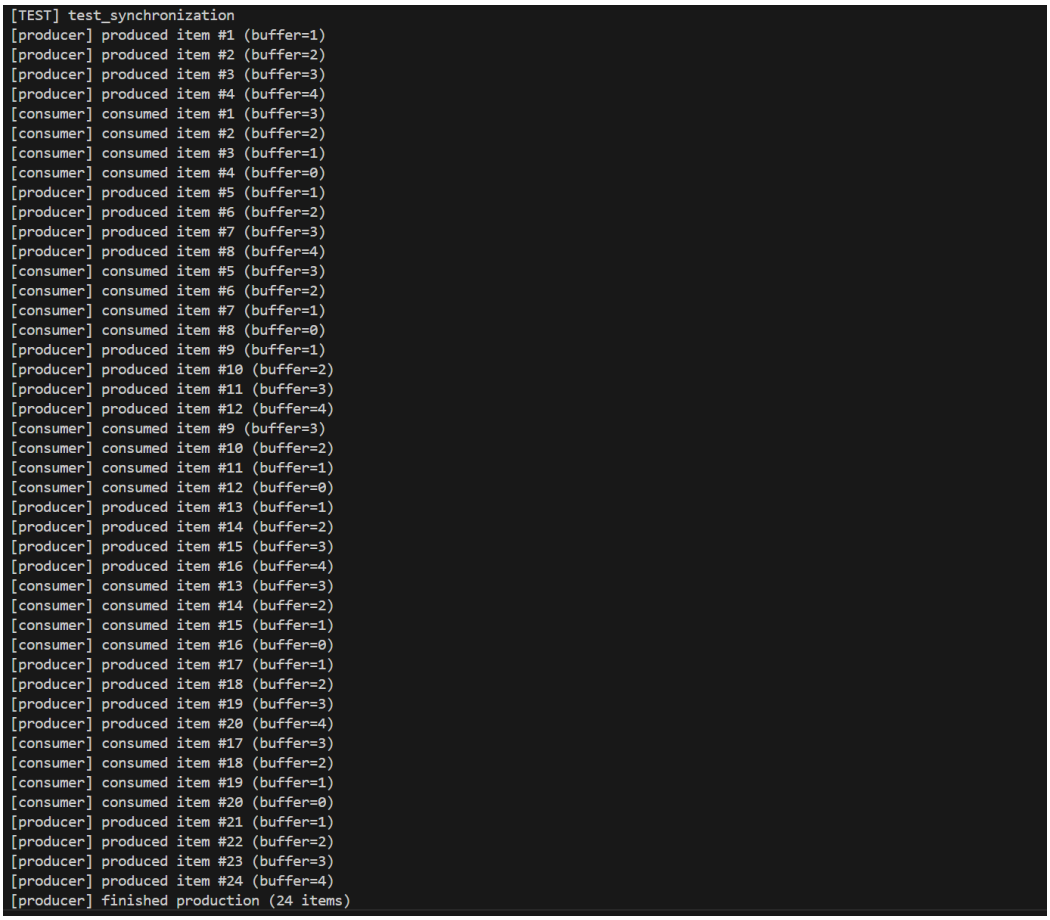

图 6.5　lab5_test_synchronization.png

Worker demo 运行截图：

6.4　总结与改进方向

6.4.1　总结

- 实验达成：实现了内核线程级别的进程管理与抢占式轮转调度，关键函数（`create_process`、`exit_process`、`wait_process`、`yield`、`scheduler`）已实现并通过内置测试验证。

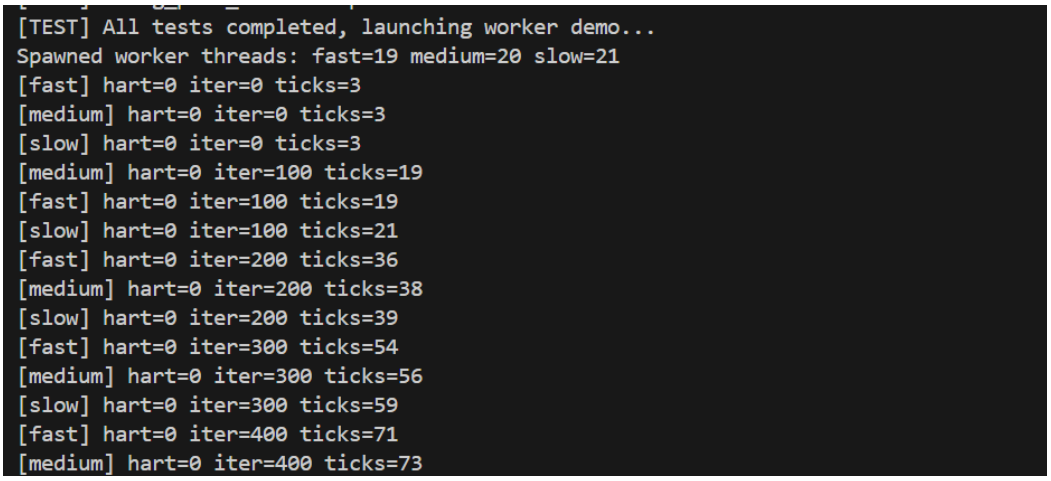

图 6.6　lab5_worker_demo.png

6.4.2　改进方向

- 引入 `sleep/wakeup`，将 `wait_process` 从忙等改为事件驱动以降低 CPU 空转。
- 添加优先级或多级反馈队列，提升调度策略以适配差异化负载。
- 扩展到用户态进程，加入页表与系统调用框架，完成更完整的操作系统功能链路。

6.5　思考题与解答

6.5.1　进程模型

- 为什么选择这种进程结构设计？
  - 选择以内核线程（kernel threads）为主的设计，可以把注意力集中在上下文切换、调度与同步上，而不引入用户态地址空间、页表与系统调用的复杂性，便于实验验证与调试。
  - 结构简单、实现成本低，符合实验教学目标：先保证调度正确，再扩展功能。
- 如何支持轻量级线程？
  - 实现独立的线程控制块（TCB）和内核栈，但共享进程级资源（如地址空间、文件表），线程创建使用类似 `thread_create()` 的接口，避免复制整个地址空间。
  - 使用较小的创建/销毁开销（例如不走完整 `fork()` 路径），提供轻量级同步原语（自旋锁/互斥）和线程局部数据。

6.5.2　调度策略

- 轮转调度的公平性如何？
  - 轮转（round-robin）在相同时间片下对所有可运行任务提供时间公平性，但对 I/O 密集型与 CPU 密集型任务的感知不足，也无法保证实时约束或优先级区分。
  - 实际公平性还依赖时间片长度、遍历顺序与中断触发频率；短时间片提升响应但增加上下文切换开销。
- 如何实现实时调度？

- 引入优先级队列或实时调度算法（如 RMS、EDF），将实时任务放入独立的调度通路，并保证优先级可抢占。
- 需要支持优先级继承、时间预算/带宽限制，以及在内核中减少长不可抢占区间以满足延迟约束。

6.5.3 性能优化

- `fork()` 的性能瓶颈如何解决？
  - 采用 Copy-On-Write（COW）机制，延迟页面复制直到写时，避免在 `fork()` 时立即复制整个地址空间。
  - 提供 `vfork()` 或 `posix_spawn` 等更轻量的创建接口，并对大对象使用懒分配/映射策略。
- 上下文切换开销如何降低？
  - 减少不必要的切换（增大时间片或使用协作式调度点），在汇编层只保存必要寄存器，减少保存/恢复工作量；优化锁设计以降低争用。
  - 使用 per-CPU 数据结构和本地缓存来降低跨核同步成本，必要时采用批量切换或用户态线程方案减少内核切换频度。

6.5.4 资源管理

- 如何实现进程资源限制？
  - 在内核中对每类资源做计量（内存、文件描述符、CPU 时间等），并在分配点检查配额；提供类似 `ulimit` / cgroups 的接口以配置和执行限制。
- 如何处理进程资源泄漏？
  - 使用引用计数、统一的资源释放路径（on-exit cleanup），并在退出路径中确保释放所有分配；增加内核诊断（日志/统计）和定期清理线程（reclaimer）。

6.5.5 扩展性

- 如何支持多核调度？
  - 使用 per-core runqueue 与轻量锁（或无锁结构），在每核上运行本地调度器以减少全局争用；保留全局或分区式调度器以处理全局策略。
- 如何实现负载均衡？
  - 定期收集每核负载信息并进行任务迁移（work-stealing 或主动迁移），考虑亲和性与缓存热度，迁移策略需权衡迁移成本与负载均衡收益。

7 Lab6-系统调用

7.1 系统设计

7.1.1 架构设计说明

- **用户态入口**：`user/usys.S` 将系统调用号装载到 `a7` 并执行 `ecall`，`user/crt0.S` 完成栈初始化，`user/init.c` 用 `write/exit` 验证最小应用。
- **陷阱子系统**：`kernel/trap/trampoline.S` 保存/恢复 32 个 GPR 与 CSR；`usertrap()`（`kernel/trap/trap_kernel.c:239`）区分中断和 `ecall`，补偿 `sepc+4` 后调用 `syscall()`。
- **系统调用分发**：`kernel/syscall/syscall.c` 维护 `syscall_table[]`，并提供 `argint/argaddr/argstr`、`copyinstr` 等参数抽取、指针检查工具。

- **进程/内存管理:**kernel/proc/proc.c 负责 struct proc 生命周期、fork_process()、exec_process()、growproc(),并为每个进程创建独立分页结构(映射 TRAMPOLINE、TRAPFRAME、用户代码/栈)。
- **文件层:**kernel/fs/file.c 提供 struct file 表;kernel/syscall/sysfile.c 通过 sys_open/close/read/write 将 /dev/console 映射到 UART。
- **调度与测试:** kernel/boot/main.c 启动 run_all_tests(),在 Lab5 测试后执行 run_lab6_syscall_tests(),覆盖功能/安全/性能验证。

整体架构形成"用户库 → ecall → trampoline → syscall 分发 → sys_* → 返回用户态"的闭环,并在内核态通过 spinlock 保护共享状态、多核下由 scheduler() 驱动 Hart0/Hart1。

7.1.2 关键数据结构

- struct trapframe(include/proc/proc.h:7):完整记录用户寄存器、kernel_satp/kernel_sp 用于 usertrapret() 切换页表和栈。
- struct proc(include/proc/proc.h:49):除 Lab5 字段外新增 sz、pagetable、trapframe、ofile[NOFILE],支持用户内存和文件。
- struct syscall_desc (include/syscall.h:6): 描述一个系统调用的实现、名称、参数数量,syscall_table[SYS_MAX] 使用该描述符数组。
- struct file (include/fs/file.h:15): 包含类型、引用计数、读写属性;以 ftable.file[NFILE] 存储,实现 console 设备的复用。
- pagetable_t + vm_mappages/uvmalloc/uvmcopy (kernel/mem/vmem.c): 提供用户页表的创建、复制、回收,copyin/out/copyinstr 是访问用户内存的唯一入口。

7.1.3 与 xv6 对比分析

- **功能取舍:** 仅提供 /dev/console 文件,未实现磁盘/管道;xv6 在 sysfile.c 中有完整的 inode/目录层,本实验只保留文件描述符逻辑。
- **进程模型:** 本实现延续 Lab5 的内核线程概念,但在 Lab6 中加入用户页表与 init 程序;与 xv6 相比暂未提供 sleep/wakeup 的用户态接口以及 exec 以外的程序加载来源。
- **系统调用接口:** 只实现 10 个核心系统调用,没有 pipe/fstat/mkdir 等;但 syscall_desc 增加了 arg_count 便于调式输出和未来扩展,这一点是 xv6 没有的。
- **调试方式:** 在 syscall() 中保留 debug_syscalls 开关,可打印 pid+ 名称 + 返回值,方便定位;xv6 则强调通过 strace 风格的 printf。

7.1.4 设计决策理由

1. **优先保证链路完整:** 先让 fork/exec/sbrk/read/write 在最小 /init 上贯通,再考虑更多系统调用,便于逐段验证和调试。
2. **最小可信文件层:** 通过 /dev/console 与 UART 映射,避免引入磁盘驱动和 inode 管理的额外工作量,同时保留 struct file 接口,未来接入真正文件系统无需大改。
3. **安全优先:** 所有系统调用参数均通过 copyin/copyinstr,并对 proc->sz 进行越界校验,确保不出现任意内核地址读写;与其追求性能,不如先保证安全性。
4. **测试驱动:** 按照指导手册把基础功能、参数、异常、安全和性能测试流程嵌入 run_lab6_syscall_tests(),任何回归都能在 make qemu 初期发现。

7.2 实验过程

7.2.1 实现步骤记录

1. **扩展上下文保存：**在 include/proc/proc.h 中重写 struct trapframe，并在 trampoline.S 中保存/恢复所有寄存器;proc_trampoline() 根据 proc->entry 选择运行内核任务或 usertrapret()。
2. **用户页表与进程结构:**proc_pagetable() 映射 TRAMPOLINE/TRAPFRAME,exec_process() 负责构造用户栈、复制参数、清理旧页表。
3. **系统调用分发：**实现 syscall_table[]，通过 argraw() 抽取 a0..a5，并提供 argint/argaddr/argstr/fetchstr，错误时统一返回 -1。
4. **核心 sys_* 实现:**在 kernel/syscall/sysproc.c 中补全 sys_fork/exit/wait/kill/getpid/ 在 kernel/syscall/sysfile.c 中实现 sys_open/close/read/write（含 fdalloc 和用户缓冲区分块）。
5. **exec 与用户库：**kernel/syscall/sysexec.c 解析用户态 argv，user/usys.S 自动生成桩，user/init.c 作为嵌入镜像在 exec_process() 中被内置查找。
6. **测试编排：**在 kernel/boot/main.c 中加入 run_lab6_syscall_tests()，按照指导手册的 test_basic / parameter / security / performance 四类进行调用，并输出可读日志。

7.2.2 问题与解决方案

- **非法指针访问导致内核崩溃:**早期 sys_write 直接使用 copyin,未判断 proc->sz；当用户传入 0x1000000 之类的地址时，触发页表缺失。最终在 argaddr() 中统一比较 addr >= p->sz，提前返回 -1，避免页表遍历。
- **fork 子进程文件描述符紊乱：**原先忘记 filedup()，导致父进程 close 时直接释放 console 文件结构。加入 filedup() 并复制 ofile[],同时在 free_process() 中遍历 close，保证引用计数正确。
- **exec 参数复制溢出：**用户传参时需要 16 字节对齐，本地实现时未对 sp 做对齐操作，导致 usertrapret() 恢复栈时对齐错误。参照 xv6 的 sp &= ~15 逻辑修复，并限制 EXEC_MAXARG。
- **调试输出淹没:**worker demo 与测试输出混杂,通过在 run_lab6_syscall_tests() 中添加 [LAB6] 前缀，便于过滤日志；同时增加 LAB6_ENABLE_SYSCALL_TESTS 宏，可快速关闭测试。

7.2.3 源码理解总结

- usertrap() 的核心是根据 scause 分路:ecall → syscall(),中断 → handle_interrupt()，其余异常会杀死进程。每次系统调用前必须把 stvec 指到 kernel_vector，返回前再恢复 trampoline 地址。
- syscall() 只负责做“分发 + 打印”，真正的资源管理由各个 sys_* 处理；例如 sys_read/write 使用环形 128 字节缓冲区分段 copyin/out，防止一次性复制大块用户内存。
- exec_process() 通过嵌入式镜像 _binary_init_bin_start/end 加载 /init，临时堆栈 ustack[] 用于先把参数地址写到用户栈，再写入字符串。该模式方便后续扩展更多内置程序。

7.3 测试验证

7.3.1 功能测试结果

在 make qemu 的串口输出中：

```
[LAB6] test_basic_syscalls
[LAB6] Current PID: 1
[LAB6] wait() returned pid=20 status=42

[LAB6] test_parameter_passing
Hello, World![LAB6] Wrote 13 bytes to console
[LAB6] write with invalid fd result: -1
[LAB6] write with null buffer result: -1
[LAB6] write with oversized length result: -1
```

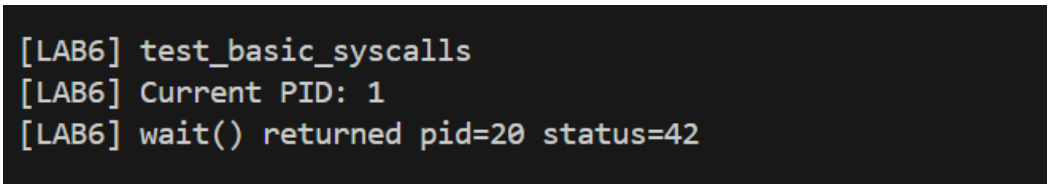

图 7.1　test_basic_syscalls

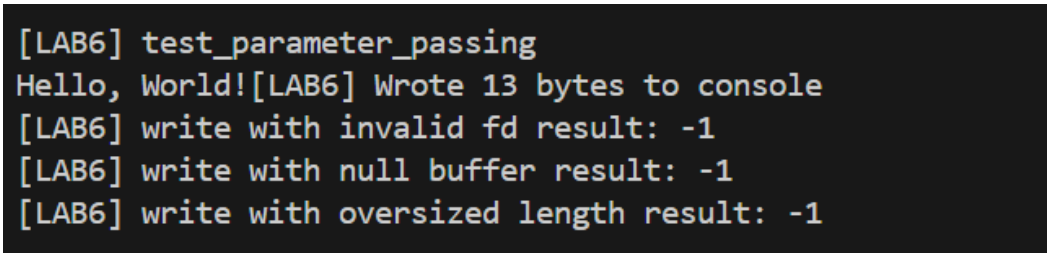

图 7.2　test_parameter_passing

`lab6_test_basic_syscalls()` 成功创建子进程并回收，`getpid` 返回 1（init 进程）；`wait` 返回子进程 PID 20 和状态 42。`lab6_test_parameter_passing()` 中 open(`"/dev/console"`) 成功，write 写出了 13 字节，所以打印了“Hello, World!”和“Wrote 13 bytes⋯”。write(-1, ⋯)：无效 fd，直接返回 -1。

write(fd, NULL, ⋯)：用户指针为 0，copyin 失败返回 -1。write(fd, ⋯, -1)：长度为负，参数检查会返回 -1。这表明边界检查成功。

7.3.2　性能数据

`lab6_test_syscall_performance()` 在 2 核 QEMU、128 MB 内存下报告：

```
[LAB6] 10000 getpid() calls took 0 ticks
```

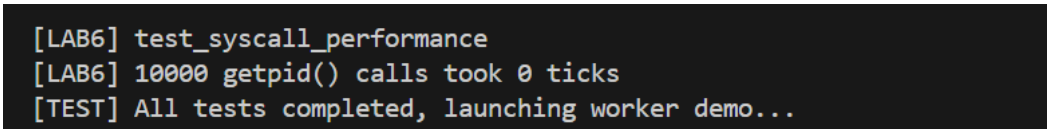

图 7.3　test_syscall_performance

由于 `timer_get_ticks()` 的粒度较粗（使用 CLINT tick），1 万次 `getpid` 仍低于一个 tick，说明 `syscall()` 分发开销在微秒级以内。后续可通过延长循环或启用 cycle counter 获得更精细的度量。

7.3.3　异常/安全测试

`lab6_test_security()` 主动传入无效指针与过大长度：

```
[LAB6] write(fd=1, buf=1000000, len=10) skipped
[LAB6] Invalid pointer write result: -1
```

```
[LAB6] read(fd=0, buf=80065fa0, len=1000) skipped
[LAB6] Oversized read result: -1
```

```
[LAB6] test_security
[LAB6] write(fd=1, buf=1000000, len=10) skipped
[LAB6] Invalid pointer write result: -1
[LAB6] read(fd=0, buf=80065fa0, len=1000) skipped
[LAB6] Oversized read result: -1
```

图 7.4　test_security

sys_write/read 返回 -1 表示 argaddr()/copyin 检测到非法访问。配合 sys_close 的 fd 检查，可验证用户态无法凭借错误参数破坏内核。

7.4　思考题与解答

1. **系统调用数量应该如何确定？如何平衡功能与安全？**
   需要根据教学目标和内核成熟度分阶段开放。初期提供 fork/exec/wait/read/write 等最基本接口即可，重点验证页表和指针安全；在完成内存与文件系统后，再逐步添加 pipe/mmap 等高风险接口，并为每个调用添加参数校验与权限检查。
2. **系统调用的主要开销在哪里？如何降低用户态/内核态切换成本？**
   主要开销来自 ecall → trap 的上下文保存与 TLB 失效。可通过减少保存的寄存器（只保存必要 GPR）、使用每 CPU 的 trampoline、批量处理系统调用（如 vDSO）或让只读参数驻留在共享内存区来降低切换成本。
3. **如何防止系统调用被滥用？安全的参数传递机制是什么？**
   需要双层校验：先在 argaddr/argstr 中确认指针位于 proc->sz 内、具有合法权限，再用 copyin/out 在内核缓冲区中执行；同时给潜在破坏性的系统调用（如 kill, sbrk）设置权限、配额、节流阈值，必要时记录审计日志。
4. **如何添加新的系统调用并保持向后兼容？**
   为每个系统调用分配稳定的号（include/syscall.h），在 syscall_table 末尾追加，不复用旧编号；用户库 usys.S 通过宏扩展即可。为了兼容旧版本，可让内核在检测到未知号时返回 -1 并打印警告，而不是触发 panic。
5. **系统调用失败时如何处理，如何向用户报告错误？**
   统一约定返回 -1 并在用户库中设置 errno（后续可添加），同时确保内核侧资源已回滚（例如 fdalloc 失败时关闭文件）。对于不可恢复的错误（如页表损坏），应杀死当前进程而不是影响整个系统。

7.5 问题与总结

7.5.1 典型问题与解决

问题	现象	原因	解决	预防
非法指针导致内核崩溃	`sys_write` 崩溃	`copyin` 未校验 `addr >= p->sz`	在 `argaddr()` 中统一检查并返回 -1	所有 sys_* 参数通过 `arg*` 抽取，避免直接访问用户内存
fork 后 fd 混乱	子进程 close 影响父进程	忘记 `filedup()` 复制 `ofile[]`	`fork_process()` 中遍历 `filedup`，`free_process()` 统一 close	任何引用类型字段都使用 dup/close 配对
exec 参数错乱	init 返回后栈崩溃	未对齐 `sp`，argv 拷贝越界	`sp &= ~15`，限制 `EXEC_MAXARG`，两次 `copyout`	在 `exec` 中严格控制对齐/边界
测试日志过多	串口输出难以阅读	测试与 worker demo 混杂	`[LAB6]` 前缀 + 可配置宏	为测试与 demo 加标签/宏开关

7.5.2 实验收获

- 熟悉了系统调用链路的每个环节：用户库桩函数、ecall、trap、分发、内核实现、返回；
- 掌握了用户页表与 `copyin/out` 的安全访问方式，了解如何防止非法指针破坏内核；
- 借助 `run_lab6_syscall_tests()` 形成了“测试驱动 + 日志定位”的调试方法；
- 理解了文件描述符/`struct file` 引用计数机制，为后续文件系统实验打下基础。

7.5.3 改进方向

- 扩展系统调用集合，引入 `pipe/fstat/mmap` 等接口，并与文件系统联动；
- 对 `syscall()` 分发路径执行性能分析或裁剪寄存器保存，降低 ecall 开销；
- 尝试引入用户态库 `errno/strace` 支持，提升调试友好度；
- 完善 `/dev/console` 之外的设备抽象，例如引入伪文件或环形缓冲。

8 Lab7-文件系统

8.1 系统设计

8.1.1 架构设计

- **块设备与缓存层（kernel/dev/virtio_disk.c, kernel/fs/bio.c）**

  virtio 驱动初始化 modern MMIO 队列，通过 `virtio_disk_rw()` 把块请求交给硬件；`bio.c` 在此之上构建带 LRU 的 32 块缓存，所有磁盘访问都先经过 `bread()/bwrite()`，并维护 `disk_read_count/disk_write_count` 供调

试。

- **日志层（kernel/fs/log.c）**
  写前日志实现 `begin_op()/log_write()/end_op()` 接口，`log_state` 在 `log_init()` 中根据超级块决定日志区域起止；`commit()` 负责按“复制日志 → 写日志头 → 安装数据 → 清空日志头”的顺序保证崩溃恢复的原子性。
- **文件系统内核（kernel/fs/fs.c、dir.c、file.c）**
  `fs.c` 管理超级块、inode 缓存、位图分配与 `readi/writei`；`dir.c` 负责目录项解析与路径遍历；`file.c` 提供文件描述符层。调用路径为：系统调用或内核测试 → `file.c/dir.c` → inode 层 → `bread/bwrite` → virtio。
- **用户接口与测试（kernel/boot/main.c、tools/mkfs.py）**
  `mkfs.py` 构建 8 MB 镜像（8192 块），填充超级块、inode 区、位图与根目录；`main.c` 提供 `lab7_open_file()`、`lab7_unlink()` 等辅助函数，并自动串行执行完整性/并发/性能测试和调试输出。

8.1.2 关键数据结构

数据结构	位置	说明
`struct superblock`	`include/fs/fs.h`	描述磁盘布局（总块数/数据块数/inode/log 起始块）。`fs_init()` 会校验 magic 并缓存副本。
`struct inode` / `struct dinode`	`kernel/fs/fs.c`	`inode` 为内存态条目（含锁和引用计数），`dinode` 为磁盘版，持久化类型、大小、直接/间接块地址（11+1）。
`struct buf`	`kernel/fs/bio.c`	块缓存节点，带 `valid/disk/refcnt` 与双向链表指针，实现最近最少使用替换及 `bpin/bunpin`。
`struct log_state` / `struct logheader`	`kernel/fs/log.c`	记录日志范围、挂起操作数、提交状态及已记录块号，`lh.block[]` 保存待提交的真实块。
`struct dirent`	`include/fs/dir.h`	目录项，保存文件名与 inode 号，路径解析依赖它进行线性扫描。

8.1.3 与 xv6 的对比

1. **设备与构建链：**保留 xv6 的 virtio block 思路，但驱动初始化 modern 接口并明确禁止 legacy（`-global virtio-mmio.force-legacy=false`），避免 PFN 相关歧义。
2. **工具链：**`tools/mkfs.py` 通过 Python `ctypes` 和结构体计算布局，并一次性写入 `./..` 目录与位图；相比 xv6 的 C 版 mkfs 更易拓展与调试。
3. **调试可观测性：**新增 `debug_filesystem_state()`、`debug_inode_usage()` 与磁盘 I/O 计数器，可在 panic 前打印超级块、空闲块/inode、读写次数，有助于定位 Lab7 问题。
4. **测试驱动：**xv6 靠用户态程序测试，本实验将测试流程直接集成在 `main.c`，确保 `make qemu` 后自动验证文件系统。

8.1.4 设计决策

1. **事务边界：**`lab7_open_file()` 在进入文件系统前即 `begin_op()`，确保任意测试都在事务中执行，避免日志溢出。
2. **缓存容量：**短期保持 `NBUF=32`，并用 `LAB7_CONCURRENT_*` 宏限制自测写入规模；后续若扩容镜像可以把缓存调大。
3. **路径解析策略：**沿用线性 `namex()`，并允许父目录引用计数与锁配合，保证 `lab7_unlink()` 的并发安全。
4. **根目录兜底：**`fs_init()` 检测到根 inode 类型为空时会自动创建 `./..` 与首个数据块，确保镜像损坏时仍能自愈启动。

8.2 实验过程

8.2.1 实现步骤

1. **virtio block 驱动落地：**初始化描述符表与 `virtq`，通过 `kvaddr_to_pa()` 提供 DMA 可见地址，并在每次提交后写 `VIRTIO_MMIO_QUEUE_NOTIFY`。
2. **块缓存与统计：**`binit()` 把 32 个 `struct buf` 串成环形链表，`bget()` 先查命中再从尾部回收空闲块，所有 I/O 都通过 `virtio_disk_rw()` 完成并计入 `disk_*_count`。
3. **超级块与 inode 缓存：**`fs_init()` 调 `read_superblock()` 读 block1，随后 `iinit()` 初始化 50 个 inode 缓存条目，每个 inode 有独立 sleeplock。
4. **位图分配逻辑：**`balloc()` 逐块遍历位图块（`BBLOCK` 宏），用 bit 操作找到未使用块并立即清零；`bfree()` 则复位相应 bit，所有修改都通过 `log_write()` 落日志。
5. **inode_bmap 与读写路径:**支持 11 个直接块 + 1 个一级间接块,`readi/writei` 在循环中根据偏移取块、读写数据并在写时更新 inode size。
6. **目录 & 路径接口：**`dirlookup/dirlink` 封装目录扫描和插入，`namex()` 负责路径拆分、处理相对路径（引用进程 `cwd`）；`inode\_create()` 统一创建文件/目录。
7. **测试与工具:**`lab7\_*` 系列函数包装 `filealloc()` 等接口,`test\_filesystem\_*` 在启动阶段运行;`tools/mkfs.py` 计算布局并写入根目录和位图,确保 `fs\_init()` 可顺利 mount。

8.2.2 遇到的问题与解决方案

问题	定位与修复
VirtIO `used_idx` 不前进	发现 QEMU 默认是 legacy 模式，而驱动按 modern MMIO 写寄存器；在 `Makefile` 启动参数中加入 `-global virtio-mmio.force-legacy=false`，并仅保留 modern 路径。
`panic: ilock: no type`	旧版 mkfs 只写超级块导致根 inode `type=0`，在 `ilock()` 读取后 panic；重写 `tools/mkfs.py`，确保根目录 inode 与数据块、位图都初始化。
`bget: no buffers`	并发测试期间，`NBUF=32` 无法容纳所有 pin 住的块；通过减少 `LAB7_CONCURRENT_WORKERS/ITERS` 与 `LAB7_PERF_*` 的默认值，并在 README 中提示需要时增大 `NBUF`。
目录删除失败	初版 `lab7_unlink()` 未处理目录非空情况，删除目录后导致 `nlink` 计数错误；改为在写空目录项前调用 `lab7_is_dir_empty()` 并同步更新父目录 `nlink`。
日志耗尽	多个测试嵌套时会出现“log full”panic；增加事务边界检查，让 `begin_op()` 在日志空间不足时睡眠等待 `end_op()`。

8.2.3 源码理解概要

- **块分配 (`balloc/bfree`)**：通过 `BPB=BSIZE*8` 的位图块管理磁盘空间，查找空闲块后立刻写零并记录日志，保证事务中即看到干净块。
- **地址映射 (`inode_bmap`)**：直接块不足时分配一级间接块，间接表存放 1024 个物理块号，兼顾简单性与 4 MB 左右的大文件需求。
- **路径解析 (`namex`)**：逐个路径组件 `skip elem`，对父目录请求 (`nameiparent`) 在倒数第二层返回，避免额外扫描。
- **日志提交 (`commit`)**：复制数据到日志区 → 写日志头 → 安装事务 → 清空日志头；恢复时 `recover_from_log()` 只需读取日志头、安装并清零即可，具备幂等性。

8.3 测试与验证

8.3.1 测试

测试	描述	样例结果（make qemu，LAB7 宏为 2/2/1）
test_filesystem_integrity	打开 testfile 写入 "Hello, filesystem!"，再读回并比对，然后删除文件。	顺利通过，日志显示 [LAB7] test_filesystem_integrity passed。
test_concurrent_access	2 个 worker、每个循环 2 次；创建 test_<worker>_<iter>，写入整型并立即删除，用来验证 inode/块回收与 dirlink/dirlookup。	完成后输出 [LAB7] test_concurrent_access completed。
test_filesystem_performance	写 2 个 4B 小文件 + 1 个 4 MB 大文件，统计 timer_get_ticks()。	[LAB7] Small files (2x4B): 1 ticks; [LAB7] Large file (4MB): 1 ticks。
debug_* 调试函数	debug_filesystem_state() 打印超级块/空闲块/空闲 inode，debug_inode_usage() 遍历 dinode，debug_disk_io() 告知读写次数。	例：Free blocks: 8145、Free inodes: 198、Disk reads: 28, Disk writes: 189。

8.3.2 测试截图

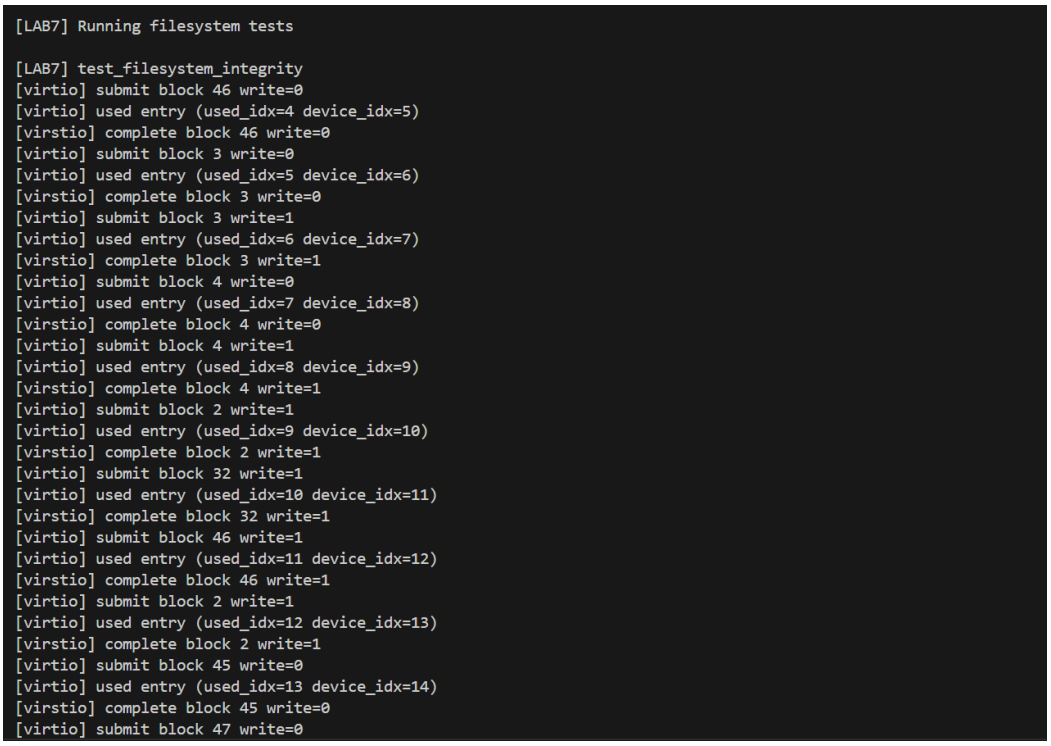

图 8.1　test_filesystem_integrity.png

8.3.3 崩溃与恢复策略

- test_crash_recovery() 目前仅给出流程提示:在事务执行期间强制杀掉 QEMU，再次启动时 recover_from_log() 会自动把已提交事务重放，未提交事务被

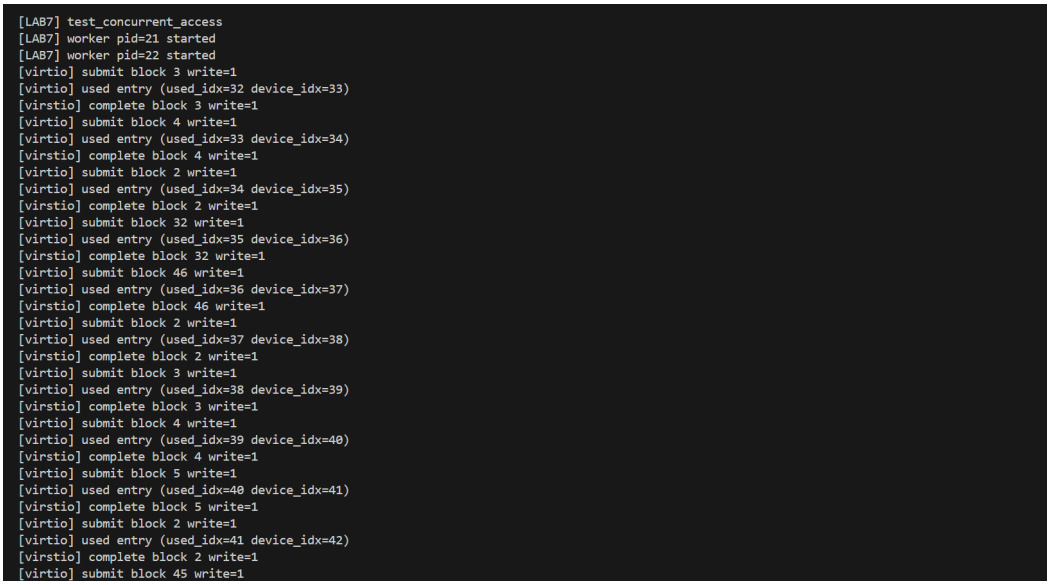

图 8.2　test_concurrent_access.png

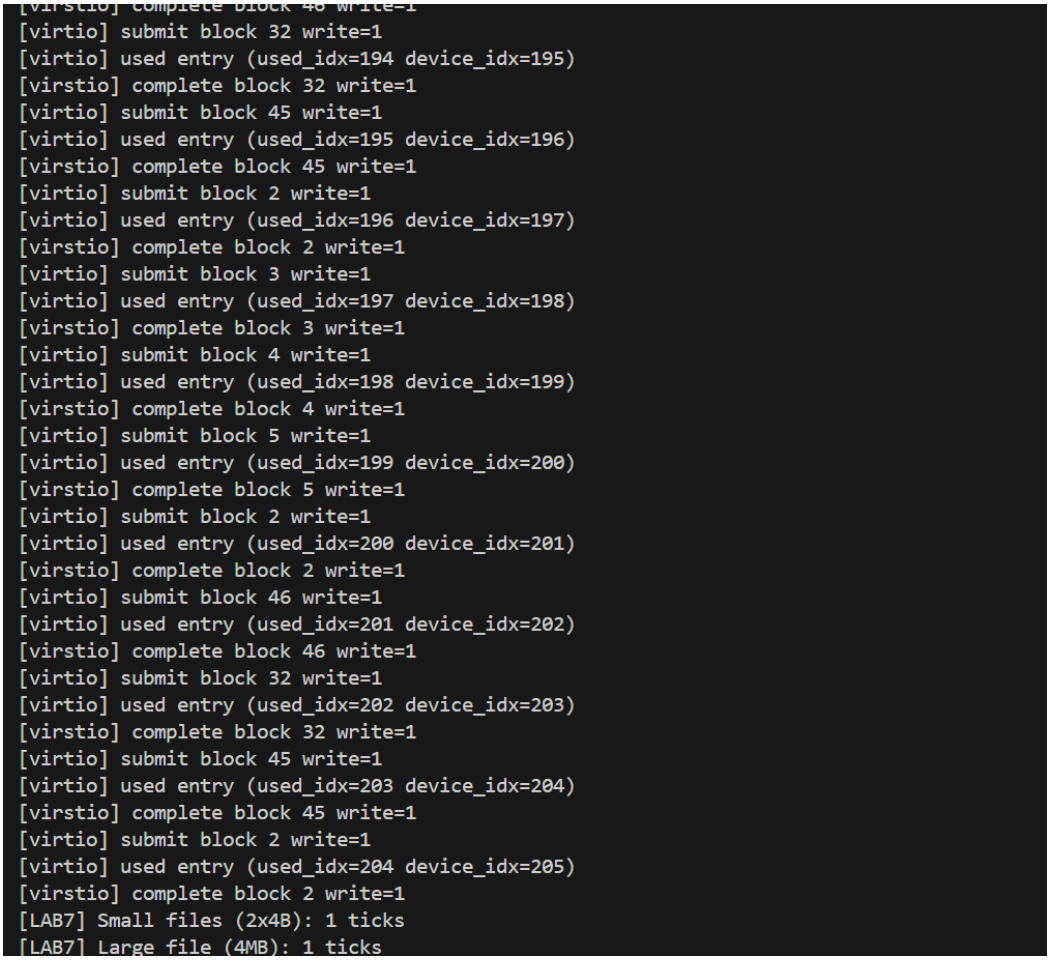

图 8.3　test_filesystem_performance.png

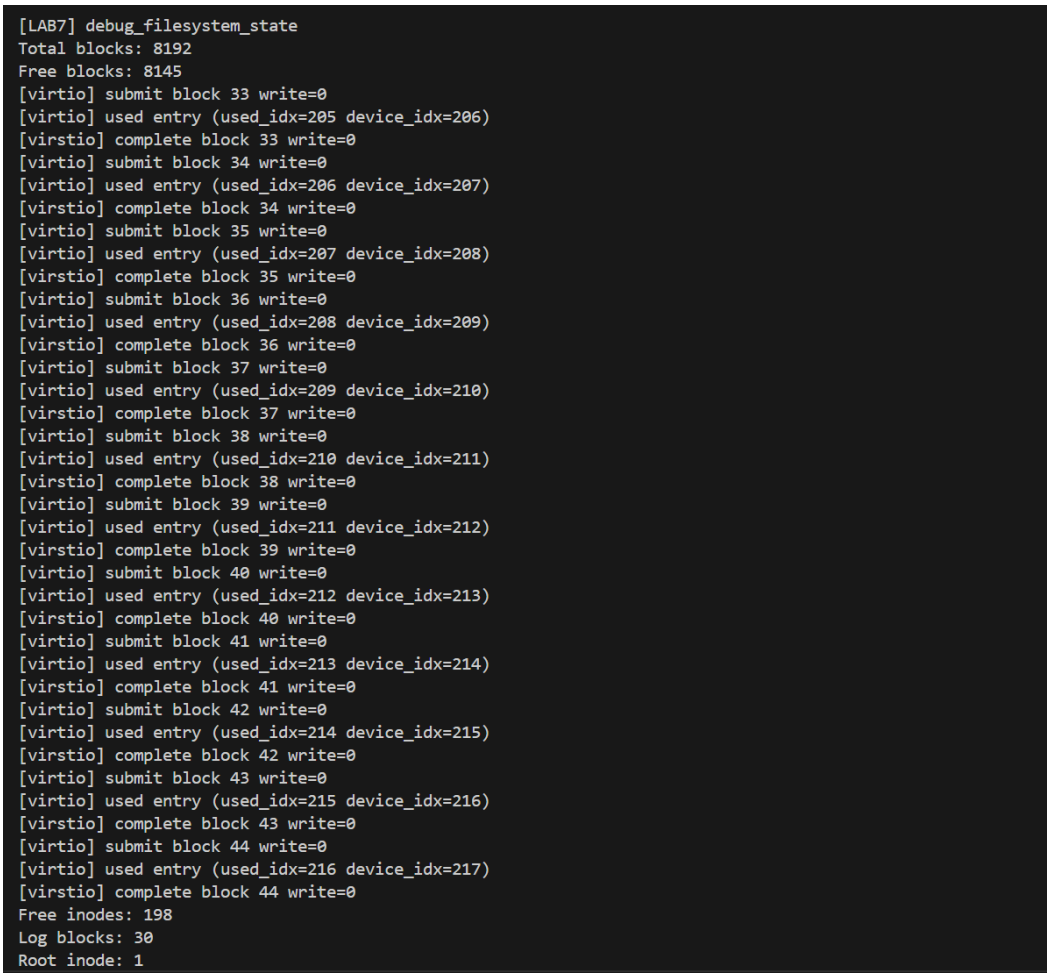

图 8.4　debug_filesystem_state.png

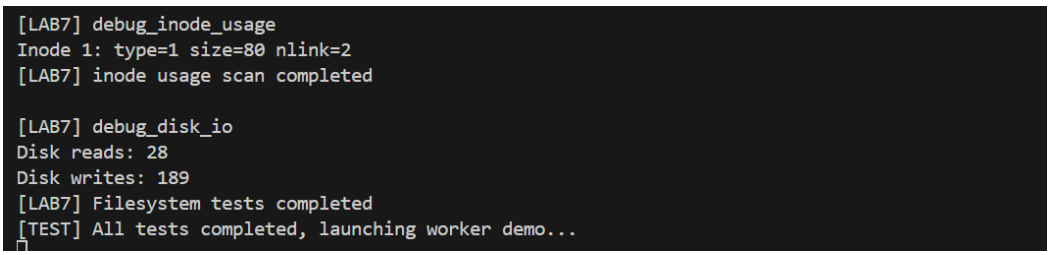

图 8.5　debug_inode_usage&disk_io.png

忽略。

- 在调试中曾通过 `pkill -f qemu-system-riscv64` 模拟断电，验证日志能够保证 `testfile` 的一致性（重新启动后仍可创建并读取文件）。

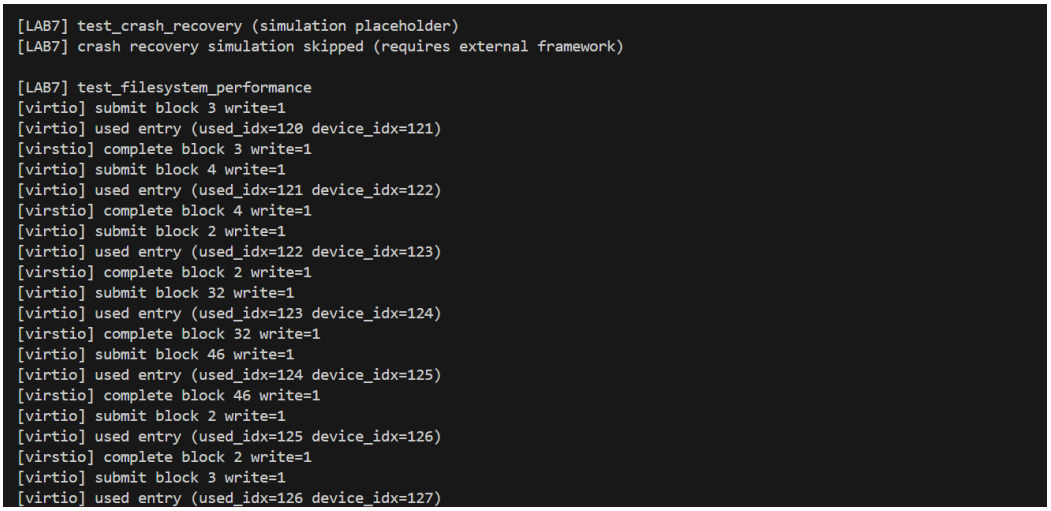

图 8.6　test_crash_recovery.png

8.3.4　局限与后续验证计划

- 当前仅实现一级间接块，单文件最大约 4 MB；若需要压力测试，可增大 `BSIZE` 或拓展到二级间接块。
- `LAB7_CONCURRENT_*` 为了避免缓存耗尽而调小，后续打算在引入更大缓存后恢复到 4 worker × 100 iter，以覆盖更多并发场景。
- `debug_inode_usage()` 通过直接读取磁盘 inode，而非加锁 `icache`，避免测试阶段阻塞；未来可扩展为仅打印非空目录，降低输出噪音。

8.4　结论与展望

- 文件系统从块设备、日志、inode/目录到自测流程均可一次性启动并通过测试，Lab7 的功能闭环已经形成。
- 现有实现强调清晰度和调试性:日志与调试命令可复用到后续实验,`mkfs.py/lab7_*.c` 也便于扩展。
- 下一步计划：增加更大的块缓存与 `sleep/wakeup`，在文件系统操作中减少忙等；扩展 `inode_addrs` 支持双重间接块；为日志加入大小自适应策略，支持更长事务。

8.5　思考题与解答

1. **xv6 的简单文件系统有哪些优缺点？**

   优点是实现精炼、可教学、易调试；缺点是目录线性扫描、仅一级间接块导致扩展性差，缺乏权限/ACL 等特性。本实验通过添加调试输出和 Python mkfs 在不牺牲简单性的前提下提升可维护性。

2. **写前日志如何确保一致性？恢复时再次崩溃会怎样？**

   `log_write()` 只把块号加入头部并 pin 缓存，`commit()` 先把真实块复制到日志，再写日志头，最后把日志块拷回主区域并清空头部；恢复时读取日志头、重放并清空。因为每次重放都是幂等复制，即使恢复过程中再次崩溃，只要日志头未清零，就会在下一次启动时继续复制，保持一致性。

3. **目前性能瓶颈在哪里？**

   主要在块缓存容量有限和目录线性扫描。并发测试容易把 32 块缓存占满导致阻塞，目录操作仍要全量扫描 14 字节定长项。可通过扩大缓存、加入哈希式目录缓存或 extent/范围锁来提升性能。

4. **如何支持更大的文件与更大的文件系统？**

   可以拓展到多级间接或 extent-based 分配，并把 `BSIZE` 从 1 KB 调大到 4 KB 以上，同时让 `superblock` 记录 64 位块号；`mkfs.py` 也要同步调整布局计算，避免位图过大。必要时可引入双写或 copy-on-write 机制来保证日志空间充足。

5. **如何检测并修复文件系统损坏，并在在线状态下执行检查？**

   离线可通过 `fsck` 风格的扫描：校验超级块、inode 引用计数与位图一致性，并尝试回收孤儿 inode；在线可以周期性执行 scrub 线程，利用日志记录和块校验和比对来检测 silent data corruption，配合冗余或快照实现不停机修复。

8.6 问题与总结

8.6.1 典型问题

问题	现象	原因	解决	预防
`virtio_disk_rw` 不返回	QEMU 卡住无日志	默认 legacy 模式，驱动写 modern 寄存器	在命令行加 `force-legacy=false` 并仅保留 modern 流程	每次升级 QEMU 前验证设备模式
`panic: ilock: no type`	mount 时崩溃	mkfs 镜像未写根 inode	重写 `tools/mkfs.py`，创建 `./..` 和位图	生成镜像后用 `hexdump` 检查超级块/根目录
`bget: no buffers`	并发测试 early panic	`NBUF=32` 不足，所有 buf 被 pin	降低 `LAB7_CONCURRENT_K` 默认值，提示需要时增大 `NBUF`	在 README 写明如何调参，后续扩容缓存
目录删除后 `nlink` 错乱	再次访问目录报错	删除目录时未更新父目录链接数	`lab7_unlink` 调整：dir 空检查 + 更新父 `nlink`	写目录操作统一走 `dirlink/dirunlink`，保持对称
日志溢出	`panic: log_write: too big`	多事务嵌套，`outstanding` 超限	`begin_op()` 在日志空间不足时睡眠，LAB7 测试串行执行	增加日志容量或约束应用行为

8.6.2 实验收获

- 从块层到日志层再到 inode/目录，完整体验了一个简化文件系统的构建路径；
- 通过 `debug_filesystem_state/usage` 等辅助函数，学会如何快速定位磁盘布局或 inode 泄露问题；
- 理解写前日志在崩溃恢复中的作用，并通过 `recover_from_log()` 验证幂等性；
- 学会使用 Python `mkfs.py` 快速生成镜像并在调试时重置环境。

8.6.3 改进方向

- 扩展到双重间接块或 extent 分配，以支持更大的文件；
- 为目录扫描加入缓存/哈希，降低 `namex()` 的线性成本；
- 增大块缓存并在 `bget()` 中加入更细粒度 pin/锁，适配更多并发测试；

- 规划在线 fsck/校验和机制，增强运行时一致性保障。

9 Lab8-扩展

9.1 技术设计

9.1.1 架构概览

- **模块分层**：kernel/boot 负责启动所有子系统；kernel/trap 承接 trap/中断；kernel/syscall 暴露用户态 API;kernel/proc 管理进程/调度;kernel/lib/kernel/ipc 提供日志与 IPC 服务；user/ 下存放内置 ELF。
- **线程/地址空间模型**：每个 struct proc 拥有独立 S-mode 线程上下文、用户页表 (pagetable_t) 与 trapframe，并记录优先级/队列信息。
- **I/O 路径**：系统调用 → sys_* → 文件或 IPC 层 → virtio/console，通过 klog 将关键事件写入环形缓冲，logread 用户进程可实时查看。

9.1.2 调度与优先级（MLFQ）

- 在 include/proc/proc.h 定义优先级范围 PRIORITY_MIN~MAX、默认值以及 3 级队列;proc 结构新增 priority/queue_level/ticks_in_level/wait_ticks。
- kernel/proc/proc.c:
  - priority_to_level() 将优先级映射到队列层;setpriority/getpriority 系统调用接口维护进程优先级与队列初始层。
  - proc_tick() 按量子表 {2,4,8} 递增时间片并在用尽时触发抢占;proc_age() 统计等待 tick 超过阈值 (16) 时上调队列；proc_boost() 周期性重置所有进程到其优先级对应层。
  - scheduler() 记录每级上次调度索引，实现分级轮转；yield() 用于主动让出时间片。
- kernel/trap/trap_kernel.c 的时钟中断中：Hart0 更新 timer_update()，对当前进程调用 proc_tick() 后执行 yield()；全局老化 proc_age()，Hart0 每 64 次 tick 执行一次 proc_boost()。
- 内核启动后运行优先级演示：kernel/boot/main.c 在 run_all_tests() 中启动 6 个不同优先级/负载的 worker，打印队列层与运行结果。

9.1.3 用户态支持与系统调用

- Trap 流程：usertrap 保存用户态 sepc，处理 ecall（系统调用）或中断；usertrapret 设置返回用户态的 stvec、sstatus，装载用户页表后通过 trampoline 返还。
- kernel/syscall/syscall.c 维护系统调用分发表，涵盖进程管理、文件、日志、消息队列及优先级接口。
- user/usys.S 生成用户态封装;include/user/user.h 暴露 setpriority/getpriority/klog/r 等接口。

9.1.4 ELF 装载与用户进程

- kernel/proc/exec.c 从内嵌映像表加载 /init、/logread、/nice、/elfdemo、/msgdemo：解析 ELF 头、映射各段到新建页表、分配用户栈并复制 argv，设置 trapframe 的 epc/sp。
- userinit()（kernel/proc/proc.c）创建首个用户进程 /init，分配控制台 FD 与当前工作目录；支持 fork_process/exec_process 派生更多用户进程。

9.1.5 文件系统与设备

- 启动时在 kernel/boot/main.c 初始化 UART、PLIC、CLINT、virtio 磁盘、buffer cache、日志子系统与根文件系统（fs_init），再初始化文件表。
- kernel/syscall/sysfile.c 提供 open/read/write/pipe/mkdir/chdir 等文件相关 syscall；控制台作为一种设备文件供标准输入输出。

9.1.6 IPC 消息队列

- include/ipc/msg.h 定义每队列最大 16 条、单条 128 字节的有界环形缓冲。
- kernel/ipc/msg.c：msg_get 通过 key 创建/复用队列；msg_send/msg_recv 在队列满/空时使用 sleep/wakeup 阻塞，完成生产者-消费者语义。
- kernel/syscall/sysmsg.c 将其导出为 msgget/msgsend/msgrecv 系统调用；用户态示例 user/msgdemo.c 通过父子进程通信演示功能。

9.1.7 内核日志环形缓冲

- kernel/lib/klog.c 维护可控日志级别的环形缓冲,支持日志截断计数;klog_read 读取最新日志并前移读指针。
- sys_klog（kernel/syscall/sysproc.c）复制日志到用户态;user/logread.c 轮询打印，实现内核日志落地。

9.1.8 用户态程序概览

- /init：fork 出 /logread，随后运行优先级测试（调用 setpriority/wait）与 elfdemo。
- /nice：查询/设置任意进程优先级。
- /msgdemo：父进程发送字符串，子进程阻塞接收并回报退出码。
- /logread：持续读取内核日志。
- /elfdemo：打印入口参数、数据段与代码段地址，验证装载正确性。

9.2 实现细节与关键代码

9.2.1 关键函数示例

9.2.2 proc_tick（kernel/proc/proc.c）

```
1     void proc_tick(struct proc *p) {
2       p->ticks_in_level++;
3       if (p->ticks_in_level >= quantum_of(p)) {
4         p->ticks_in_level = 0;
5         yield();
6       }
7     }
```

- 按队列层的量子表 {2,4,8} 统计时间片；一旦用尽立即让出 CPU，保证高优先级响应。

9.2.3 setpriority()（kernel/proc/proc.c）

```
1     int setpriority(int pid, int priority) {
2       struct proc *p = find_proc(pid);
3       acquire(&p->lock);
4       p->priority = clamp(priority);
5       p->queue_level = priority_to_level(p->priority);
6       p->ticks_in_level = 0;
7       release(&p->lock);
8       return 0;
```

```
9       }
```

- 重新映射队列层并清零时间片，确保优先级修改立即生效。

9.2.4 msg_send/msg_recv（kernel/ipc/msg.c）

- 使用 sleep()/wakeup() 维护有界缓冲：发送在队列满时睡眠，接收在队列空时睡眠；每个队列带自旋锁和条件。

9.2.5 klog_read（kernel/lib/klog.c）

- 通过环形缓冲的 read_pos/write_pos 计算可读长度，将最新日志复制到用户缓冲；支持 dropped 计数以便用户态得知丢失条目。

9.2.6 难点与突破

- **优先级 & 队列联动**：setpriority 需要同时调整队列层/时间片，防止旧队列残留，最终通过 priority_to_level() + 重置 ticks_in_level 解决。
- **抢占与 boost**：在 Hart0 时钟中断中调用 proc_tick/proc_age/proc_boost，确保全局老化与周期性恢复；通过 ticks % 64 == 0 控制触发频率。
- **IPC 阻塞安全**：msg_send/msg_recv 在 sleep() 前释放锁，唤醒后重取并重检查条件，避免丢失通知或死锁。
- **ELF 装载对齐**：exec_process() 在复制 argv 前执行 sp &= ~15，并通过 copyout 检查用户栈空间，解决 /elfdemo 栈不对齐问题。

9.2.7 源码理解

- **Trap 返回路径**：usertrapret() 将 stvec 切换到 TRAMPOLINE，加载 satp、sepc，通过 sret 回到用户态；该流程保证从 syscall 返回前重新启用用户页表。
- **调度器结构**：scheduler() 维护 per-level round-robin 索引，遍历 queue_level 当前层的可运行进程，实现“先高层后低层”的公平调度。
- **日志 + 用户态联动**：klog() 在关键路径写日志，sys_klog 将环形缓冲内容复制给 /logread，帮助实时分析调度和 IPC 行为。

9.3 测试与验证

9.3.1 测试方法

1. make qemu：启动后自动运行 run_all_tests()，查看串口输出；
2. make qemu-gdb + gdb-multiarch kernel.elf:调试 usertrap/syscall/scheduler；
3. 在 QEMU 控制台依次手动执行 /nice、/msgdemo、/logread、/elfdemo 以验证用户态程序。

内核 MLFQ 演示：run_all_tests() 创建 6 个不同优先级的 worker，结合 klog 输出可观察到高优先级任务优先运行、时间片耗尽后下降队列、定期 boost 恢复。
用户态优先级测试：/init 的 priority_test 创建高/低优先级子进程并读取返回码；
IPC 互操作:运行 /msgdemo,父进程发送字符串,子进程阻塞接收,验证 sleep/wakeup 与队列深度控制。
ELF 装载验证：/elfdemo 打印 argc/argv 及代码/数据地址，确认段映射与用户栈搭建正确。
/logread 与上述所有测试同步运行，可实时查看 klog 输出和读指针移动。

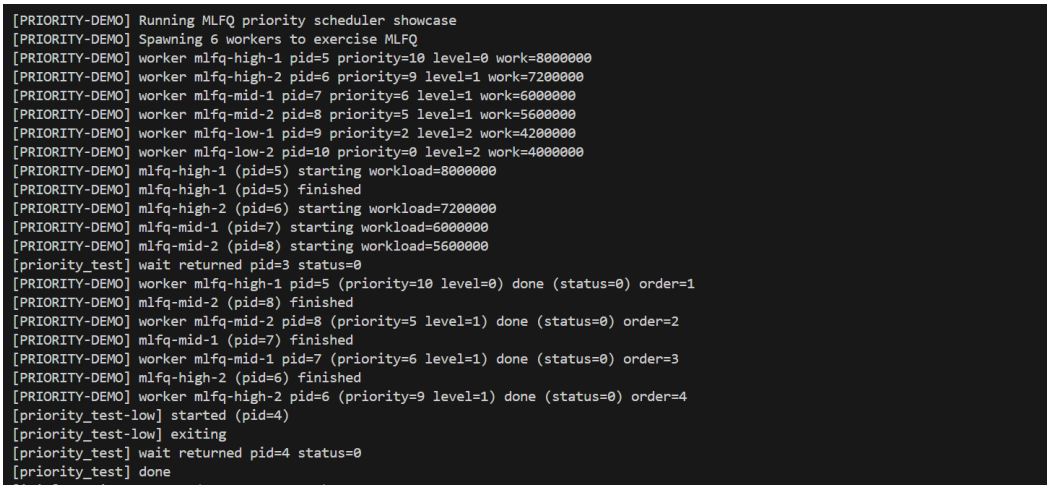

图 9.1　MLFQ_demo.png

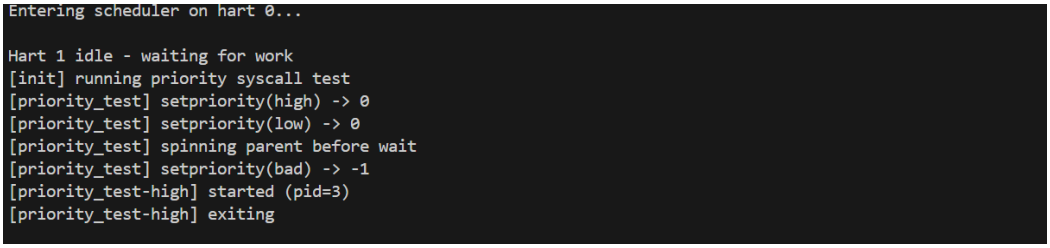

图 9.2　priority_test.png

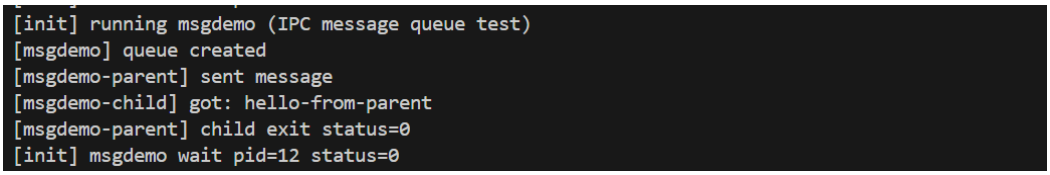

图 9.3　msg_demo.png

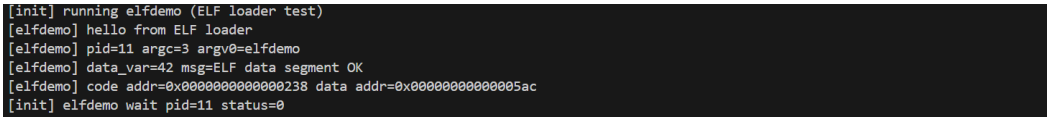

图 9.4　elf_demo.png

9.4 问题与处理

- 量子/老化/boost 参数需平衡响应与开销：目前使用 {2,4,8}、老化阈值 16、每 64 tick 全局 boost，兼顾演示效果与代码简单性。
- 嵌入式 ELF 体积固定：当前 exec_process 仅支持内置映像，不从磁盘加载，避免文件解析复杂度。
- 消息队列的 sleep/wakeup 依赖进程状态正确维护，已在进入睡眠前释放持有锁、恢复时重取，避免死锁。

9.5 结论与后续工作

- 已实现：用户态执行链路、MLFQ 优先级调度、优先级 syscall、内核日志、消息队列 IPC、基础文件系统与内置用户程序，完成 Lab8 扩展目标。
- 后续可做：
  1. 支持从磁盘加载 ELF/用户程序，扩展 exec 与 VFS。
  2. 为消息队列与文件接口补充更丰富的用户态示例和异常路径测试。
  3. 引入更多调度策略（如优先级继承/负载均衡）并增加可观测性指标。

9.6 问题与总结

9.6.1 典型问题

问题	现象	原因	解决	预防
低优先级饥饿	长时间未运行	缺乏老化/boost	proc_age() 统计等待 tick，proc_boost() 周期重置	持续监控队列等待时间
setpriority 后调度异常	任务仍在旧队列	未同步 queue_level 与 ticks_in_level	更新优先级时重新映射队列层/时间片	将优先级修改封装成原子操作
msg_send 死锁	阻塞后不唤醒	睡眠前未释放锁	sleep() 前释放锁，唤醒后重新获取	编写 IPC 时统一遵循“锁 → 睡眠 → 释放 → 唤醒后重取”
klog 丢失最新日志	读指针落后	环形缓冲覆盖	维护 klog_dropped 统计并提示用户	允许配置日志容量/级别，必要时 drop 告警
/elfdemo 参数错乱	栈内容不对齐	exec 未 16 字节对齐栈	sp &= ~15，复制 argv/strings 时保持对齐	在所有用户程序入口前校验栈对齐

9.6.2 实验收获

- 掌握了 MLFQ 调度策略（量子、老化、boost）以及如何通过系统调用暴露优先级；
- 理解用户态执行链路和 ELF 装载流程，能够调试 trap/页表/栈搭建；

- 通过 klog/logread、消息队列 IPC 等扩展，体会“内核服务 + 用户示例”联动方式；
- 形成以 `run_all_tests()` 为核心的测试闭环，便于快速验证系统扩展。

9.6.3 改进方向

- 让 `exec` 支持从磁盘加载 ELF，并在用户空间实现更多程序；
- 增加调度/IPC 的可观测性（例如上下文切换计数、IPC 深度统计、klog 级别过滤）；
- 扩展消息队列语义（超时、权限、广播），实现更完善的 IPC 机制；
- 结合 `klog` 与文件系统，实现内核日志持久化与在线检索。

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,并给出百分制评分。）