操作系统内核编写项目报告

目录

[一、项目目的 2](#_Toc28906)

[二、实验环境 2](#_Toc4778)

[三、项目结构 2](#_Toc12096)

[四、关键代码与原理 3](#_Toc7272)

[4.1 内核主函数（main.c） 3](#_Toc29411)

[4.2 链接脚本（linker.ld） 3](#_Toc8589)

[4.3 Bochs 配置（bochsrc） 4](#_Toc7026)

[4.4 Xmake 构建脚本（xmake.lua） 4](#_Toc1928)

[五、 运行步骤和结果 4](#_Toc13226)

[5.1 内核的编译与运行 4](#_Toc9478)

[5.2 时钟中断验证 5](#_Toc23261)

[5.3 中断与异常处理测试 6](#_Toc6657)

[5.4 验证进程调度 7](#_Toc32587)

[5.5 多线程支持验证 9](#_Toc22850)

[5.6 内存管理测试 10](#_Toc6158)

[5.7 系统调用机制测试 11](#_Toc30467)

[六、 项目总结 13](#_Toc17348)

## 一、项目目的

本项目的目标是从零编写一个简化的操作系统内核。通过构建交叉编译环境、使用 GRUB 完成引导加载、利用链接脚本布置内核内存布局，并在内核中实现若干核心功能，使内核能够在 QEMU/Bochs 模拟器中成功运行。

主要功能包括：

1. 内核入口与启动流程：通过 GRUB 加载内核，利用 boot.S 建立执行环境，并跳转至内核主函数 main。
2. 内存管理：基于 GRUB 提供的内存信息，完成内存初始化与动态分配接口，实现内核级内存管理。
3. 中断与异常处理：构建全局描述符表（GDT）和中断描述符表（IDT），并完成时钟中断的初始化与处理中断机制。
4. 系统调用机制：基于 syscall 指令，建立用户态与内核态的交互通道，支持基础系统调用功能。
5. 多线程支持：设计并实现线程管理模块，能够创建并调度多个线程，实现内核级并发执行。

## 二、实验环境

操作系统：Ubuntu 20.04/22.04

编译工具链：x86\_64-elf-gcc、ld、nasm

构建工具：xmake

引导加载器：GRUB2 (Multiboot2)

模拟器：QEMU、Bochs

## 三、项目结构

实验工程的主要文件和目录如下：

src/：核心源码目录

boot/boot.S：启动汇编代码，提供 \_start 入口，调用 main

kernel/\*.c：内核核心模块（内存管理、进程/线程、系统调用等）

user/\*.c：用户态测试程序

lib/\*.c：辅助函数（如 printf、string 库）

build/：编译生成目录，保存 kernel.elf 和 kernel.iso

grub/grub.cfg：GRUB 配置文件，指定加载内核

main.c：内核主函数，负责初始化各子系统并创建测试线程

linker.ld：链接脚本，定义内核加载地址和内存布局

bochsrc：Bochs 模拟器配置文件

xmake.lua：构建脚本，自动完成编译、链接和 ISO 镜像生成

## 四、关键代码与原理

### 4.1 内核主函数（main.c）

内核的入口函数 main 完成以下工作：

1. 内存初始化：调用 memory\_init(magic, addr)，接收 GRUB 提供的启动信息，建立内存管理机制。
2. 终端与串口初始化：terminal\_init() 与 serial\_init() 用于屏幕输出和串口调试。
3. 中断与 GDT 初始化：设置中断描述符表和全局描述符表，保证内核能处理异常和中断。
4. 堆内存与时钟初始化：kheap\_init() 提供动态内存分配；timer\_init() 设置时钟中断。
5. 系统调用与线程初始化：注册系统调用表，建立线程管理机制。
6. 测试线程执行：通过 process\_execute(thread\_a, "proc\_a") 和 process\_execute(thread\_b, "proc\_b") 创建并调度两个线程，测试系统调用机制。

测试机制： thread\_a 调用 syscall，将 RAX=1； thread\_b 调用 syscall，将 RAX=0。 说明内核已正确支持系统调用号传递与调度。

### 4.2 链接脚本（linker.ld）

该文件定义了内核的段布局：

GRUB Multiboot2 Header：放置在低地址 1M 处，保证 GRUB 能识别并加载内核。

.text / .rodata / .data / .bss：分别存放代码、只读数据、已初始化数据、未初始化数据。

高地址映射：通过 0xFFFFFFFF80000000 将内核映射到内核空间，隔离用户态地址。

符号导出：如 \_kernel\_phys\_end，用于确定内核大小和内存管理边界。

### 4.3 Bochs 配置（bochsrc）

设置内存 megs: 32

选择 boot: cdrom，从 build/kernel.iso 启动

开启 magic\_break，支持 BOCHS\_BREAK 指令触发断点

串口 com1 输出到文件 com1\_log，便于调试

### 4.4 Xmake 构建脚本（xmake.lua）

定义 x86\_64-elf 交叉编译工具链

添加 src/boot、src/kernel、src/user 源文件

设置编译参数：-ffreestanding -nostdlib -mcmodel=kernel，保证无标准库依赖

链接时使用 linker.ld，生成 system.map

构建完成后：

1. 拷贝 kernel.elf 到 build/iso/boot/
2. 添加 grub.cfg，生成 kernel.iso

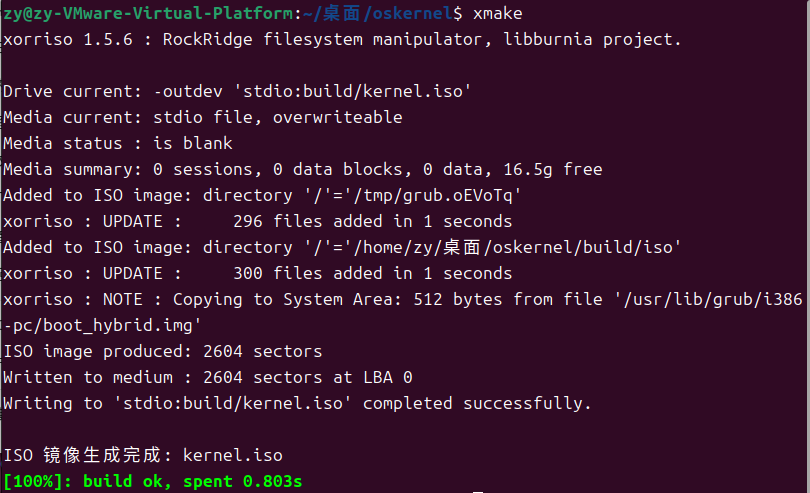
提供 on\_run 命令，可直接调用 qemu-system-x86\_64 -cdrom build/kernel.iso 启动内核

## 运行步骤和结果

### 5.1 内核的编译与运行

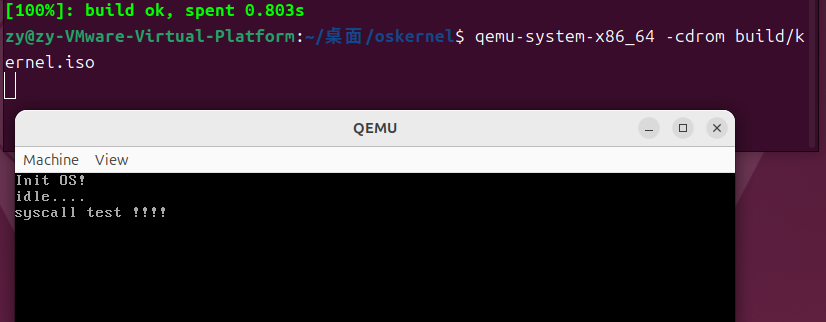
在 Ubuntu 环境下，进入工程根目录执行以下命令：

Xmake



启动内核

执行：qemu-system-x86\_64 -cdrom build/kernel.iso



GRUB 正确加载了内核，启动完成，并进入执行流程

### 5.2 时钟中断验证

修改 time.c，在 timer\_interrupt\_handler 里加入了一个打印输出语句：

void timer\_interrupt\_handler(uint8\_t vec\_nr){

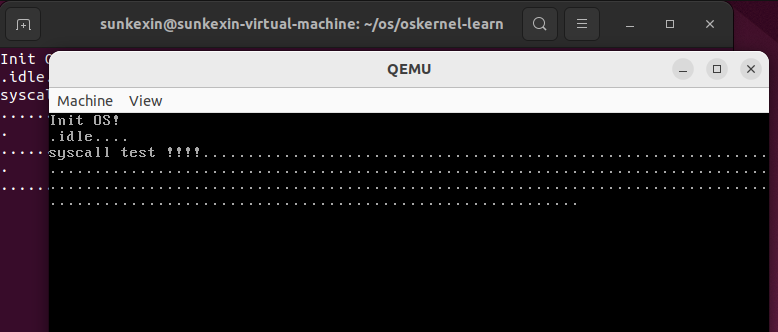
    ASSERT(vec\_nr == 0x20);

    pic\_send\_eoi(vec\_nr);

    putchar('.');   *// 每次时钟中断打印一个点*

    thread\_handler();

}

在每次时钟中断发生时，屏幕上打印一个 .  
运行内核时，看到屏幕不断出现 ....，说明时钟中断在周期性触发并输出点

### 5.3 中断与异常处理测试

在完成时钟中断测试后，为了进一步验证 CPU 异常 (#DE, #UD, #PF) 能否被内核捕获与处理，在 main.c 中增加了三个测试线程：

*// 除零异常测试 (#DE)*

void thread\_div\_zero(void\* arg) {

    printk("[TEST] div\_zero thread running\n");

    int a = 1, b = 0;

    int c = a / b;   *// 触发 #DE*

    (void)c;

    while(1);

}

*// 非法指令异常测试 (#UD)*

void thread\_ud2(void\* arg) {

    printk("[TEST] ud2 thread running\n");

    \_\_asm\_\_ volatile("ud2");   *// 触发 #UD*

    while(1);

}

*// 缺页异常测试 (#PF)*

void thread\_page\_fault(void\* arg) {

    printk("[TEST] page\_fault thread running\n");

    int\* p = (int\*)0x123;

    \*p = 42;   *// 触发 #PF*

    while(1);

}

在 main() 中，注册对应的异常处理函数：

set\_interrupt\_handler(0,  general\_interrupt\_handler);  *// Divide Error*

set\_interrupt\_handler(6,  general\_interrupt\_handler);  *// Invalid Opcode*

set\_interrupt\_handler(14, general\_interrupt\_handler);  *// Page Fault*

*// 启动测试线程*

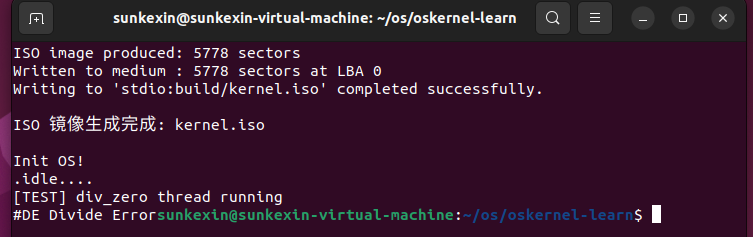
thread\_start("div\_zero", 5, thread\_div\_zero, NULL);

thread\_start("ud2",      5, thread\_ud2, NULL);

thread\_start("page\_fault", 5, thread\_page\_fault, NULL);

并在 interrupt.h 中补充了 general\_interrupt\_handler() 的声明

运行内核后，输出结果如下：

、

系统首先完成了内核初始化，并进入 idle 线程，说明调度器工作正常。

测试线程 div\_zero 成功启动，触发除零异常。

CPU 抛出的 Divide Error (#DE) 被内核的 general\_interrupt\_handler() 捕获，并在屏幕/串口输出异常信息。

实验结论

1. IDT 设置正确：异常号 (#DE、#UD、#PF) 已能正确分发到指定的异常处理函数。
2. 调度与异常联动正常：idle 线程和测试线程能正确切换，线程执行中发生的异常能被捕获。
3. 异常处理机制可见性：内核能够在屏幕或串口上输出对应异常信息，便于后续调试和扩展。

### 5.4 验证进程调度

为了验证内核线程调度机制，在 main.c 中编写两个内核态线程：

*// 内核态线程：不断打印 'A'*

void thread\_a(void\* arg){

    while (1){

        putchar('A');

        for (volatile int i = 0; i < 100000; ++i) { }

    }

}

*// 内核态线程：不断打印 'B'*

void thread\_b(void\* arg){

    while (1){

        putchar('B');

        for (volatile int i = 0; i < 100000; ++i) { }

    }

}

在 main() 函数初始化完成后，调用 thread\_start() 创建两个内核线程：

thread\_start("thread\_a", 31, thread\_a, NULL);

thread\_start("thread\_b", 31, thread\_b, NULL);

然后开启中断：

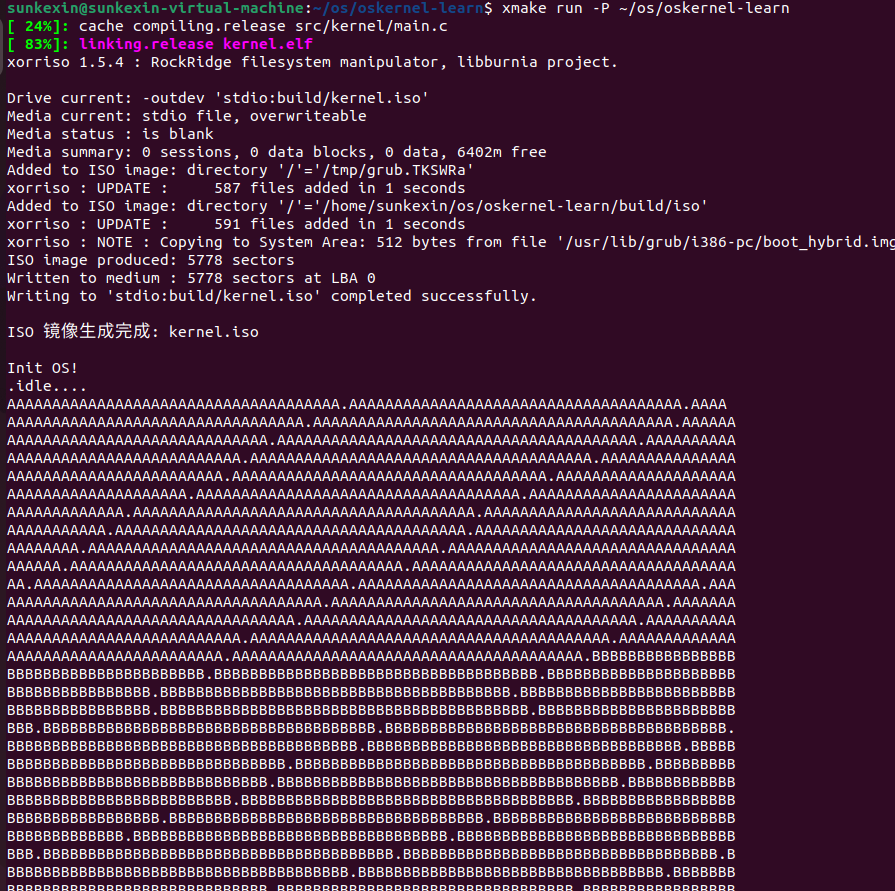
intr\_set\_status(true);   *// 允许中断*

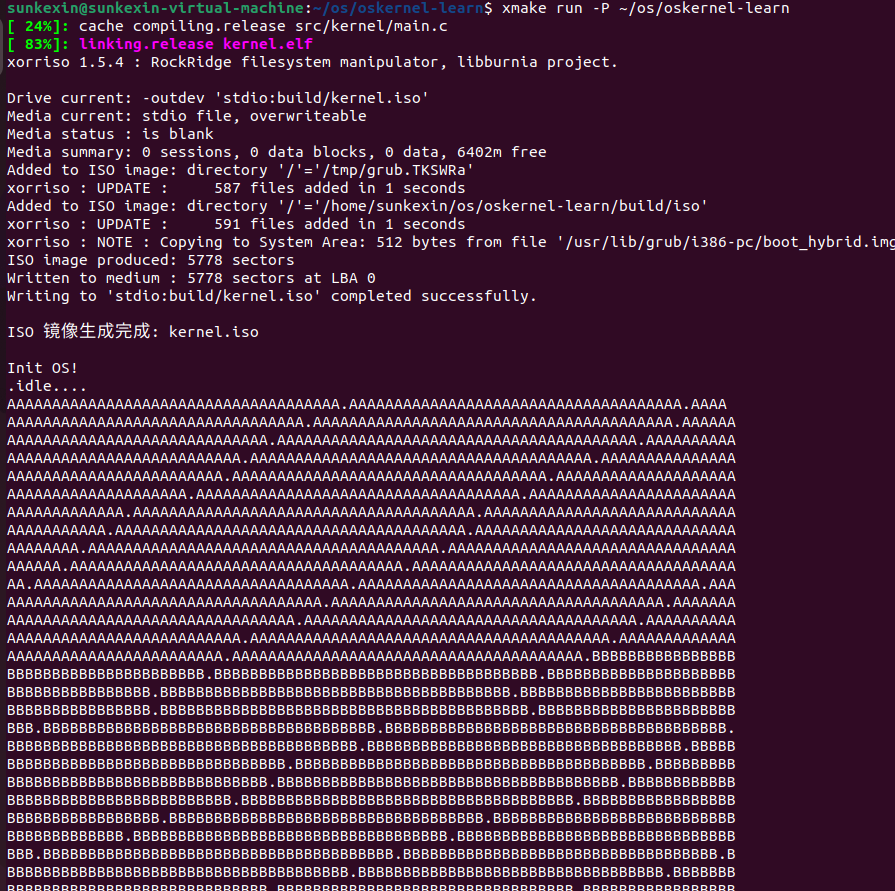
while (1) {

    hlt();               *// 进入低功耗等待中断*

}

xmake运行后得到





创建了两个普通线程，每个线程循环打印不同的字符， A 和 B，能够交替运行；

验证时钟中断触发后，调度器能够切换线程。运行结果说明时钟中断与线程调度都能够正常运行。内核支持多线程并发执行的基本功能。

### 5.5 多线程支持验证

在验证了基本的进程调度后，进一步测试内核是否能够 支持多个线程，并验证调度的公平性。

1. 创建三个普通线程 thread\_a、thread\_b、thread\_c，分别通过循环累加各自的计数器 count\_a、count\_b、count\_c。
2. 创建一个监控线程 monitor，周期性打印三个线程的计数值。

*// 全局计数器*

static volatile uint64\_t count\_a = 0;

static volatile uint64\_t count\_b = 0;

static volatile uint64\_t count\_c = 0;

*// 线程 A*

void thread\_a(void\* arg){

    while (1){ count\_a++; for (volatile int i = 0; i < 100000; ++i) {} }

}

*// 线程 B*

void thread\_b(void\* arg){

    while (1){ count\_b++; for (volatile int i = 0; i < 100000; ++i) {} }

}

*// 线程 C*

void thread\_c(void\* arg){

    while (1){ count\_c++; for (volatile int i = 0; i < 100000; ++i) {} }

}

*// 监控线程*

void monitor(void\* arg){

    while (1){

        printk("[MONITOR] A=%d B=%d C=%d\n", (int)count\_a, (int)count\_b, (int)count\_c);

        for (volatile int i = 0; i < 5000000; ++i) {}

    }

}

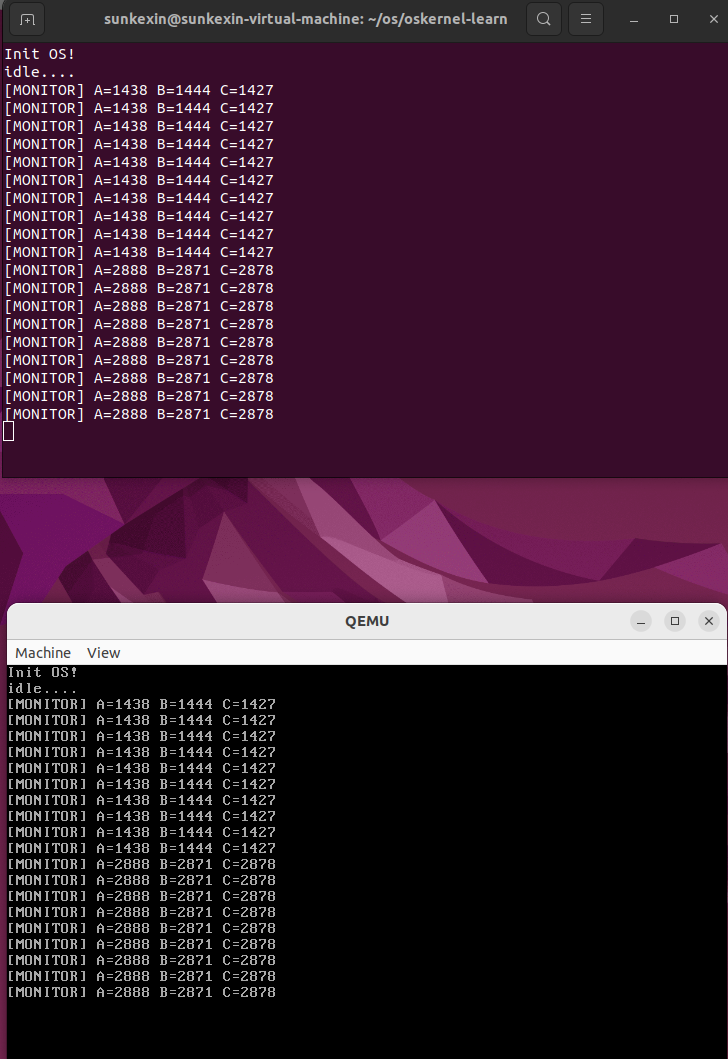
在 main() 中，依次启动四个线程：

thread\_start("thread\_a", 31, thread\_a, NULL);

thread\_start("thread\_b", 31, thread\_b, NULL);

thread\_start("thread\_c", 31, thread\_c, NULL);

thread\_start("monitor",  10, monitor,  NULL);



三个计数器的数值不断增长，并且数值相差不大。

实验结论

1. 线程并发正常：三个工作线程能够同时运行并不断更新计数值。
2. 调度公平：计数器差值较小，说明调度器采用的时间片轮转（Round-Robin）策略能保证各线程获得大致相等的执行机会。
3. 监控机制有效：通过 monitor 线程周期性输出，可以清晰观测到线程调度的公平性，而不是单纯依赖屏幕输出的视觉效果。

### 5.6 内存管理测试

为了验证内核堆内存分配与回收机制，在 main.c 中编写了 mem\_test() 函数：

在 main() 初始化阶段调用 mem\_test()，完成对 kmalloc/kfree 的测试

*// 内核内存管理测试函数*

void mem\_test() {

    printk("==== Kernel Memory Management Test Start ====\n");

    void\* p1 = kmalloc(128);

    void\* p2 = kmalloc(256);

    void\* p3 = kmalloc(512);

    printk("Allocated kernel memory addresses:\n");

    printk("p1=0x%x\n", (uint64\_t)p1);

    printk("p2=0x%x\n", (uint64\_t)p2);

    printk("p3=0x%x\n", (uint64\_t)p3);

    kfree(p1);

    kfree(p2);

    kfree(p3);

    printk("Memory freed!\n");

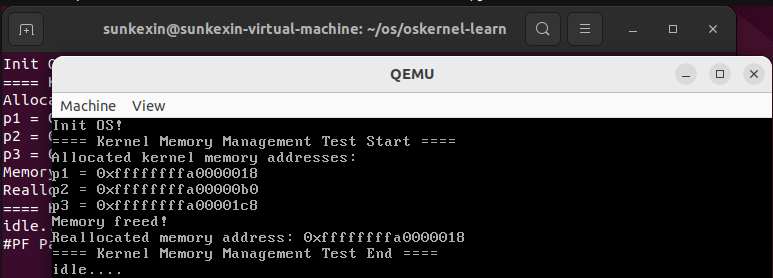
    void\* p4 = kmalloc(128);

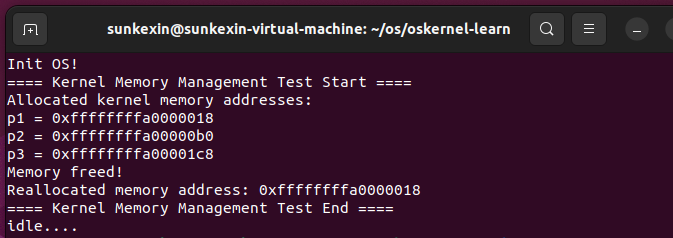
    printk("Reallocated memory address: 0x%x\n", (uint64\_t)p4);

    printk("==== Kernel Memory Management Test End ====\n");

}

QEMU 运行时输出：





kmalloc/kfree 在虚拟地址上工作正常，地址落在内核堆区（高端内核空间）；

释放后再次分配的地址可能与之前释放的地址相同，说明复用正常；

内核正常访问映射的虚拟地址，页表映射正确。

### 5.7 系统调用机制测试

为了验证内核的用户态与内核态切换及系统调用机制，首先在 main.c 中创建两个测试线程：

void thread\_a(void\* arg){

    \_\_asm\_\_ \_\_volatile\_\_(

        "movq $(1), %rax \n"

        "syscall         \n"

    );

    while (1) { }

}

void thread\_b(void\* arg){

    \_\_asm\_\_ \_\_volatile\_\_(

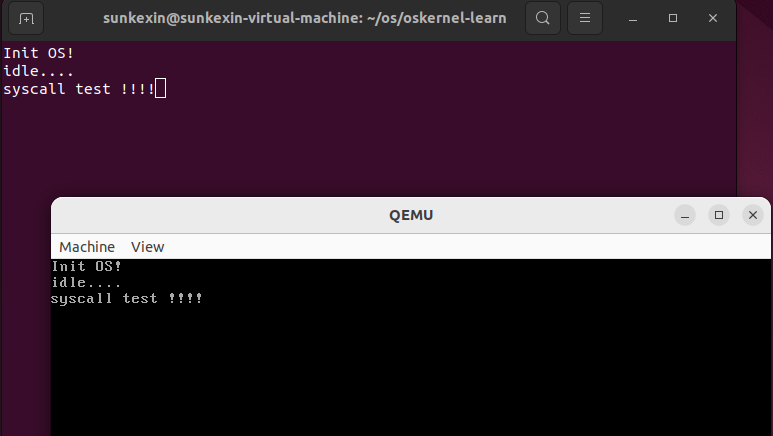
        "movq $(0), %rax \n"

        "syscall         \n"

    );

    while (1) { }

}



运行结果显示：

1. 内核启动成功 (Init OS!)；2. idle 线程正常运行；3. 成功触发系统调用 (syscall test !!!!)

说明内核态 syscall\_handler 能够被调用。

随后，为了进一步测试 用户态 → 内核态 → 用户态 的完整切换，在 src/user/user.c 中编写 user\_test()：

void user\_test(void\* arg){

    printk(">>> User thread start (user mode)\n");

*// 调用 SYS\_TEST*

    \_\_asm\_\_ \_\_volatile\_\_(

        "movq $0, %%rax\n"

        "syscall\n"

        :

        :

        : "rax", "rcx", "r11"

    );

    printk(">>> User thread returned from SYS\_TEST (back to user mode)\n");

*// 调用 SYS\_YIELD*

    \_\_asm\_\_ \_\_volatile\_\_(

        "movq $1, %%rax\n"

        "syscall\n"

        :

        :

        : "rax", "rcx", "r11"

    );

    printk(">>> User thread returned from SYS\_YIELD (back to user mode)\n");

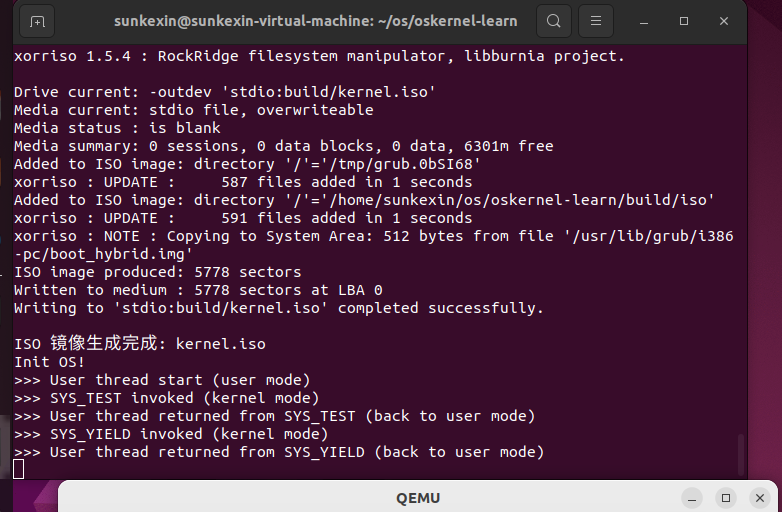
    while(1) { hlt(); }

}

在 main() 中加入：

process\_execute(user\_test, "user\_test");

并重新编译运行。



实验结果

1. 内核初始化成功：内存、GDT/IDT、线程、堆等模块正常工作。
2. 用户态线程能够启动：CPU 从内核态切换至用户态，执行 user\_test()。
3. 系统调用机制正常：用户态触发 syscall 指令后，CPU 能进入内核态并调用对应处理函数。
4. 上下文切换正确：系统调用返回后，CPU 恢复用户态上下文，继续执行用户线程。
5. 线程调度机制可用：SYS\_YIELD 系统调用能够让线程主动放弃 CPU，调度器完成线程切换。

## 项目总结

本项目实现了一个可在 QEMU/Bochs 稳定运行的操作系统内核：采用 GRUB2（Multiboot2）引导与自定义链接脚本完成高地址映射与段布局，初始化 GDT/IDT 与可编程中断控制器，时钟中断按期触发；实现基础物理/虚拟内存初始化与内核堆 kmalloc/kfree，地址位于高端内核空间且释放可复用；提供内核线程与时间片轮转调度，二线程交替输出与三线程计数监控显示切换稳定且基本公平；打通用户态 syscall→内核态处理→返回用户态的完整路径，验证了参数传递、特权级切换与上下文恢复。

同时对各项功能进行了验证，用点阵打印确认时钟中断、专用线程精准触发 #DE/#UD/#PF、监控线程量化调度公平性、内存地址与复用性输出核对页表与堆管理，结合 xmake 一体化构建与串口/屏幕日志。

目前系统仍以基础 RR 调度与内核堆为主，用户态页表隔离、按需缺页/回收、驱动与文件系统尚待完善。后续将进一步优化，逐步提升系统的可用性与可扩展性。