华中科技大学

本 科 生 课 程 设 计 报 告

课　　程：操作系统课程设计

题　　目：任务切换机制与设备阻塞工作机制研究

院　　系：网络空间安全学院

专业班级：

学　　号：

姓　　名：

2022 年　03 月 19 日

目　录

1　实验一 任务切换机制 1

1.1　实验概述 1
1.2　实验设计思路 1
1.3　实验程序的难点或核心技术分析 7
1.4　运行和测试过程 9
1.5　实验心得和建议 10
1.6　学习和编程实现参考网址 10

2　实验二 设备阻塞工作机制 12

2.1　实验概述 12
2.2　实验设计思路 12
2.3　实验程序的难点或核心技术分析 15
2.4　运行和测试过程 16
2.5　实验心得和建议 18
2.6　学习和编程实现参考网址 19

1 实验一 任务切换机制

1.1 实验概述

实验目的：

- 理解保护模式的概念；
- 掌握保护模式程序的编写；
- 理解 CPU 对段机制/页机制的支持；
- 理解段机制/页机制的原理和简单应用；
- 理解任务的概念和任务切换的过程。

实验任务：

- 启动保护模式，建立两个任务，分别循环输出“HUST”和“IS19”字符串，每个任务各自建立页目录和页表，初始化 8253 时钟和 8259 中断，实现两个任务在时钟驱动下的切换。

1.2 实验设计思路

任务整体可以分为任务环境配置、启动保护模式、保护模式下的段/页机制、保护模式下的任务切换以及中断的使用五部分。

（一）任务环境配置[1]

该任务的实现环境为 Ubuntu 20.04.1，内核版本为 5.13.0，使用到的工具有：Bochs、NASM、FreeDOS 等，任务环境配置过程如下（工作目录为 Task1）：

- 使用 apt-get 安装 bochs、bochs-x、nasm。
- 在 Bochs 官方网站下载 FreeDos，解压后将其中的 a.img 复制到工作目录下，并重命名为 freedos.img。
- 使用 bximage 在工作目录下生成一个软盘映像 pm.img。
- 修改 bochsrc.txt，具体内容见附件。
- 启动 Bochs，待 FreeDos 启动完成后使用“format b:”命令格式化 B:盘。
- 将 asm 代码指定的起始位置 07c00h 修改为 0100h，并使用 nasm 编译生成.com 可执行文件，之后将.com 可执行文件复制到虚拟软盘 pm.img 上，即可在 Bochs 内运行该程序。编译与复制操作集成到了位于工作目录下的脚本文件 mount.sh 中。（注：若提示“mount: /mnt/floppy: mount point does not exist.”，则需先在/mnt 下创建挂载点 floppy：“sudo mkdir /mnt/floppy”；若提示“NTFS signature is missing. ...”，则需先完成上一

步的格式化。）

至此，即可在 Bochs 内运行编写的.com 程序，任务环境搭建完成。

（二）启动保护模式

该部分涉及到描述符（Descriptor）、全局描述符表（GDT）、选择子（Selector）、保护模式下的段式寻址、实模式与保护模式间的切换等原理与概念。实模式与保护模式的切换流程如图 1-1 所示（含后续获得内存信息和中断相关操作）。由于该实验任务的执行结果为两个死循环间相互切换，因此程序将持续运行在保护模式，未执行返回实模式并退出的代码。

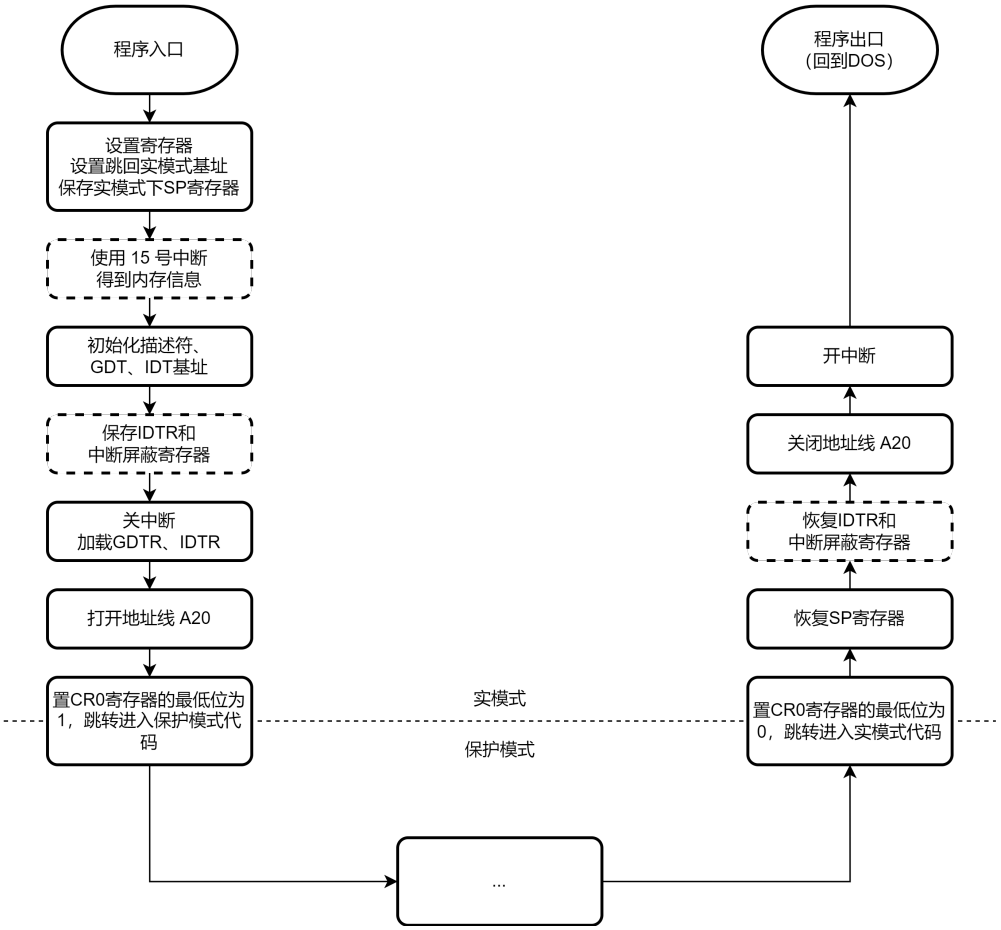

图 1-1 实模式与保护模式的切换流程

（三）保护模式下的段/页机制

该部分涉及到实模式下的寻址方式：物理地址（Physcial Address）=段值（Segment）×16+偏移（Offset）；保护模式下的寻址方式：线性地址（Linear Address）=段选择子中的基址+偏移；分页机制（保护模式下线性地址与物理地址的转换）；内存信息的获取；页表的构建等原理与概念。

在该实验任务中，由于需要构建页目录和页表，所以首先使用 15 号中断获得内存的相关信息，可得内存大小为 31MB 。又每页页表对应 $4096 \times 1024\text{B} = 4\text{MB}$ 内存，因此需要构建的页目录项个数即为 8 个；而对于每个页目录项，均对

应 1024 个页表项。在本设计中，认为两个页表的所有线性地址对应相等的物理地址、且不考虑内存空洞，所以可以连续地初始化所有页表，且两个任务对应的页表内容一致，仅为基地址有所不同。以上页目录和页表的构建在 SetupPaging0 和 SetupPaging1 中实现，分页机制示意图如图 1-2 所示。

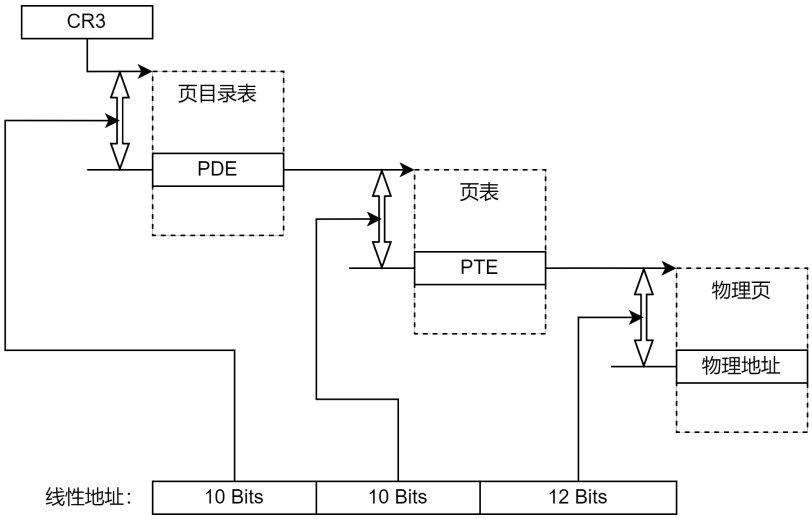

图 1-2 分页机制示意

分页机制的启动与关闭由 CR0 寄存器的 PG 位控制，而页目录的物理基地址由 CR3 寄存器控制。因此，若要打开分页机制，只需将已经初始化好的页目录的基址加载到 CR3 寄存器，再将 CR0 寄存器的 PG 位置为 1 即可；若要关闭分页机制，只需将 CR0 寄存器的 PG 位置为 0 即可。在任务切换中，第一次任务切换手动加载了 CR3，后续的任务切换中的页表切换由 TSS 自动完成。

在参考程序中，为了展示分页机制的功能，特意将两段程序通过 MemCpy 函数复制到了内存中的固定地址，并相应地修改页表，使得使用不同页表将相同的线性地址转换到了不同的物理地址。但在该实验任务的实现中，不采用这一方式，而是两个页表均为简单的线性地址向物理地址的直接映射，两个任务的代码段具有不同的线性地址和物理地址。

（四）保护模式下的任务切换

该部分涉及到局部描述符表（LDT）、任务状态段（TSS）、任务切换等原理与概念。

参考《LINUX 内核完全剖析：基于 0.12 内核》[2]，任务切换有以下四种方式：

（1）当前任务对 GDT 中的 TSS 描述符执行 JMP 或 CALL 指令；

（2）当前任务对 GDT 或 LDT 中的任务门描述符执行 JMP 或 CALL 指令；

（3）中断或异常向量指向 IDT 表中的任务门描述符；

（4）当 EFLAGS 中的 NT 标志置位时当前任务执行 IRET 指令。

在该实验设计的实现中，两个任务分别输出“HUST”和“IS19”，分别称其

为任务 0 和任务 1。首次进入任务 0 时，采用 IRETD 进行切换；后续在时钟中断的驱动下进行的切换采用 JMP TSS 进行。对于每个任务，都需要对应的数据段、堆栈段、代码段以及存储这些仅属于对应任务的描述符的局部描述符表 LDT 和描述任务状态信息的任务状态段 TSS。在代码段中，使用直接写入显存的方式直接输出信息，在一次输出完毕后，延迟一段时间后再次进行输出，直到切换至另一任务。

参考程序的代码中对使用 RETF 进行任务切换做出了示例，经过查阅资料[3][4]，IRETD 与 RETF 的不同仅为除 SS、ESP、CS、ESP 外，IRETD 还需在中间额外将 EFLAGS 入栈，因此只需在 RETF 进行任务切换的基础上增加 EFLAGS 的入栈指令，即可使用 IRETD 完成任务切换，执行 IRETD 前的栈状态如图 1-3 所示（入栈顺序由上到下）。另外，需要注意的是，首次切换任务时需要手动加载 TSS、LDT、CR3。

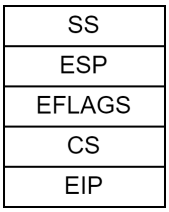

图 1-3 IRETD 前的栈状态（入栈顺序由上到下）

在后续的时钟中断的驱动下进行的切换，由于直接使用 JMP TSS 进行切换，故无需手动加载 LDT、CR3 等，但需要将其和一些寄存器信息填入两个任务分别对应的 TSS 的对应位置，如任务对应的 Ring 0 堆栈选择子、Ring 0 栈顶指针、页目录物理基地址 CR3、开启中断以完成后续任务切换的 EFLAGS（0x200）、栈顶指针 ESP、代码段选择子 CS、堆栈段选择子 SS、数据段选择子 DS、显存段选择子 GS、局部描述符表选择子 LDT 等，其中一些未使用到的寄存器（如 DS 等）可以不设置，但为了程序的完善，还是将其设定为对应的选择子。由于实现代码中每个任务的代码段单独成段，且在程序开始时已经将其段的基地址初始化到对应选择子，故初始时只需将 TSS 中的 CS 设置为对应选择子、EIP 保持为 0 即可。

至此，若要切换两个任务，只需 JMP 任务对应的 TSS 即可，两个任务间的切换见（五）中断的使用。

（五）中断的使用

该部分涉及到 8253A 时钟芯片的初始化、8259A 芯片的初始化、中断描述符表（IDT）、中断处理等原理与概念。

8253 定时器芯片[5]具有三个独立通道。每个通道包含以下部分：一个带有 CLOCK 输入的 16 位递减计数器、一个用于开启/触发计数的 GATE 输入、一个计数器输出 OUT、一个保存计数值的 16 位寄存器 COUNT 和一个用于操控计数

器的行为和 COUNT 寄存器的加载/读取的 CONTROL 寄存器。每个通道都可以以六种模式之一计数（interrupt on terminal count、 hardware retriggerable one-shot、rate generator、 square wave generator、 software triggered strobe、 and hardware triggered strobe）并且可以以 BCD 码或二进制计数。通过将控制字节写入 I/O 端口 43h 来加载通道的 CONTROL 寄存器，控制字节见表 1-1。

表 1-1 8253 芯片的控制字节组成

Bits 7,6	Channel ID (11 is illegal)
Bits 5,4	Read/load mode for two-byte count value: 00 -- latch count for reading 01 -- read/load high byte only 10 -- read/load low byte only 11 -- read/load low byte then high byte
Bits 3,2,1	Count mode selection (000 to 101)
Bits 0	0/1: Count in binary/BCD

通道 0，1，2 的 16 位 COUNT 寄存器分别对应 I/O 端口 40h，41h 和 43h。在本实验中需要产生频率为 50Hz 的时钟信号，由于 8253 芯片的默认频率为 1.1931817 MHz，因此需要向芯片写入的 COUNT 值为 1.1931817MHz/50Hz，即大约为 23863（0x5D37）。使用通道 0、模式 3、二进制计数，且需要分两次写入 COUNT，因此首先向端口 43h 发送控制字节 00110110b，然后向该通道的 COUNT 寄存器对应的 I/O 端口 40h 分别写入 23863（0x5D37）的低 8 位和高 8 位。自此，8253 时钟芯片设置完成，以上即为函数 Init8253A 的内容。

对 8259A 芯片，其负责将可屏蔽中断与 CPU 相关联，可以通过对其寄存器的设置来屏蔽或打开相应的中断、并且根据优先级在同时发生中断的设备中选择应该处理的请求。与 CPU 相连的是两篇级联的 8259A 芯片，总共可以挂接 15 个不同的外部设备，对 8259A 芯片的设置是通过向相应的端口写入特定的 ICW 来实现的。其初始化和中断设置过程为（函数 Init8259A）：①向端口 20h（主片）或 A0h（从片）写入 ICW1；②向端口 21h（主片）或 A1h（从片）写入 ICW2；③向端口 21h（主片）或 A1h（从片）写入 ICW3；④向端口 21h（主片）或 A1h（从片）写入 ICW4；⑤向端口 21h（主片）或 A1h（从片）写入 OCW1。四个 ICW 用于初始化 8259A 芯片，OCW1 用于屏蔽或打开外部中断。其中较为重要的为 ICW2 和 OCW1，其格式如图 1-4 和图 1-5 所示。

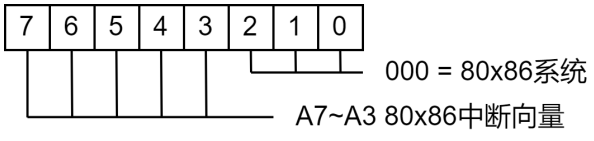

图 1-4 ICW2 格式

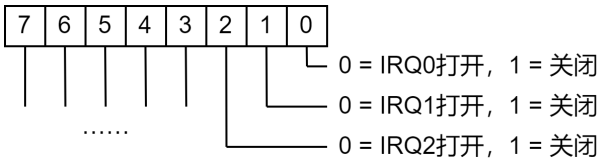

图 1-5 OCW1 格式

设计实现中，向主 8259A 写入 ICW2 时，IRQ0 对应中断向量号 20h，因此 IRQ0~IRQ7 对应中断向量 20h~27h；类似地，IRQ8~IRQ15 对应中断向量 28h~2Fh。8259A 芯片与 CPU、外部设备的关系如图 1-6 所示，在该设计实现中，使用到了由 8253 时钟芯片产生的时钟信号，而在上述过程中将 IRQ0 时钟对应的中断向量设置为 20h，因此需要在 IDT 中中断向量为 20h 的对应位置为时钟中断添加中断门。同时，在 8259A 芯片对应的中断中，仅仅使用到了 IRQ0 时钟中断，因此在初始化函数的第⑤步（即 OCW1 的写入）中，仅需将主 8259A 的最低位设置为 0，主 8259A 的其他位和从 8259A 的所有位均设置为 1，从而只开启使用到的定时器中断。

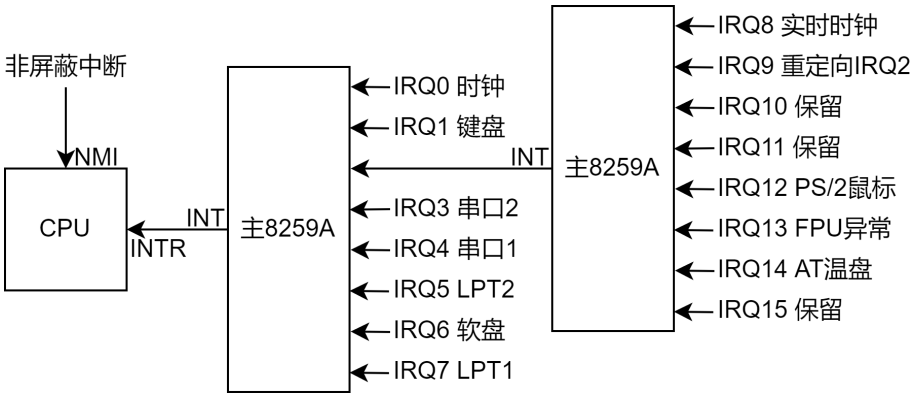

图 1-6 8259A 与 CPU、外部设备的关系

通过对 8253 时钟芯片、8259A 中断芯片和 IDT 的设置，已经将时钟中断对应的中断向量设置为 20h，只需在 20h 对应的中断处理函数中完成中断响应和两个任务的切换即可（即标号_ClockHandler 处）。首先，为了完成任务切换，需要一个变量标识当前任务，因此在全局数据段添加变量 dwCurrentTask。该变量为 0 时，即当前正在执行任务 0，若发生时钟中断，则将该变量设置为 1、通过 JMP TSS1 跳转到任务 1 继续执行；该变量为 1 时，即当前正在执行任务 1，若发生时钟中断，则将该变量设置为 0、通过 JMP TSS0 跳转到任务 0 继续执行。由于该变量存储在全局任务段，但中断触发时（即任务 0 或任务 1 执行过程中）寄存器 DS 指向任务各自的数据段，因此应在以上跳转处理前保存 DS 的原值并使其指向全局数据段，并在中断处理的最后将其恢复。同时，由于每个任务都是一个死循环，若将发送 EOI 放到任务跳转后，则永远无法发送 EOI，将不能再继续处理后续的中断，因此应将 EOI 放在任务跳转前，使得中断能够重入，从而完成任务的切换。

1.3 实验程序的难点或核心技术分析

（1）内存分布

设计实现程序的内存分布如图 1-7 所示。

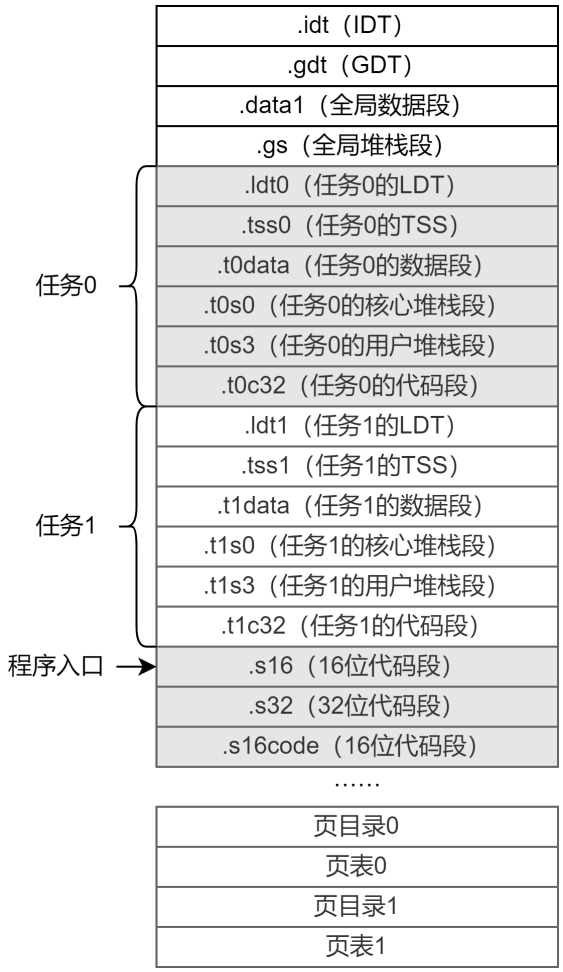

图 1-7 内存分布

（2）描述符基址初始化

在进入保护模式前的准备过程中，即程序刚刚开始后，需要初始化各个描述符的基址，此时未进行其他操作，并且 DS=ES=SS=CS，而初始化各个描述符基址的代码过于重复，即通过实模式下的寻址方式计算出各个段的基址并将其填入描述符的对应部分，因此将其编写为 NASM 的宏[6]InitDescriptorBase（定义在include/pm.inc），代码如下。

```
; 初始化段描述符基址
; usage :
; InitDescriptorBase LABEL_SEGMENT_START, LABEL_SEGMENT_DESCRIPTOR
%macro InitDescriptorBase 2
  xor   eax, eax
  mov   ax, ds
```

```
  shl   eax, 4
  add   eax, %1
  mov   word [%2 + 2], ax
  shr   eax, 16
  mov   byte [%2 + 4], al
  mov   byte [%2 + 7], ah
%endmacro
```

（三）清屏操作

为了避免程序运行前的显示信息对程序运行结果的影响，在输出信息前首先进行清屏操作（函数 ClearScreen），该操作将屏幕上所有区域设置为黑底黑字，即不显示任何信息，从而避免了信息的干扰。该函数位于.s32 段内，代码如下。

```
ClearScreen:
    push   eax
    push   ebx
    push   ecx

    mov    ah, 00000000b        ; 0000: 黑底    0000: 黑字
    mov    al, 0
    mov    ebx, 0
    mov    ecx, 4000
.1:
    mov    [gs:ebx], ax
    add    ebx, 2
    loop   .1

    pop    ecx
    pop    ebx
    pop    eax

    ret
```

（四）任务跳转所需要的信息（IRETD 和 JMP TSS）

尽管“1.2 实验设计思路”中的“（四）保护模式下的任务切换”对使用 IRETD 进行任务切换过程中需要入栈的寄存器和使用 JMP TSS 进行任务切换过程中 TSS 需要填入的值做出了解释，但在设计过程中，由于缺少参考资料，所需填入的值在一开始尚不清晰。为了获得这些信息，首先参考参考代码给出的使用 RETF 进行任务切换的方式，对比 RETF 指令和 IRETD 指令的区别并结合参考资料[3][4]，得到了使用 IRETD 的任务切换方式。再根据使用 RETF 或 IRETD 时所需要提供的寄存器信息类比到 TSS 中，获得一些必须填写的关键字段，再根据任务执行过程中对其他寄存器信息的使用填充其他字段，从而完成 TSS 的填写。

1.4 运行和测试过程

- 安装 nasm、bochs、bochs-x。
- 在工作目录 Task1 下，运行./mount.sh（或 make）完成.asm 文件的编译和.com 可执行程序的复制。
- 编译和复制完成后，可在工作目录下直接运行 bochs，bochs 会自动寻找当前目录下的配置文件并运行；也可以使用 make bochs 指定配置文件运行 bochs。
- 开启 bochs 后，输入 c 继续运行，以上步骤操作及结果如图 1-8 所示。

```
xubiang@ubuntu:~/Desktop/OS/Task1$ ls
bochsrc.txt  final_project_task1.asm  freedos.img  include  Makefile  mount.sh  pm.img
xubiang@ubuntu:~/Desktop/OS/Task1$ ./mount.sh
xubiang@ubuntu:~/Desktop/OS/Task1$ ls
bochsrc.txt  final_project_task1.asm  final_project_task1.com  freedos.img  include  Makefile  mount.sh  pm.img
xubiang@ubuntu:~/Desktop/OS/Task1$ bochs
========================================================================
                    Bochs x86 Emulator 2.6.11
            Built from SVN snapshot on January 5, 2020
              Timestamp: Sun Jan  5 08:36:00 CET 2020
========================================================================
00000000000i[      ] LTDL_LIBRARY_PATH not set. using compile time default '/usr/lib/bochs/plugins'
00000000000i[      ] BXSHARE not set. using compile time default '/usr/share/bochs'
00000000000i[      ] lt_dlhandle is 0x55a169337df0
00000000000i[PLUGIN] loaded plugin libbx_unmapped.so
00000000000i[      ] lt_dlhandle is 0x55a169339410
00000000000i[PLUGIN] loaded plugin libbx_biosdev.so
00000000000i[      ] lt_dlhandle is 0x55a16933ac10
00000000000i[PLUGIN] loaded plugin libbx_speaker.so
00000000000i[      ] lt_dlhandle is 0x55a16933c990
00000000000i[PLUGIN] loaded plugin libbx_extfpuirq.so
00000000000i[      ] lt_dlhandle is 0x55a16933d160
00000000000i[PLUGIN] loaded plugin libbx_parallel.so
00000000000i[      ] lt_dlhandle is 0x55a16933edc0
00000000000i[PLUGIN] loaded plugin libbx_serial.so
00000000000i[      ] lt_dlhandle is 0x55a1693431c0
00000000000i[PLUGIN] loaded plugin libbx_gameport.so
00000000000i[      ] lt_dlhandle is 0x55a1693439f0
00000000000i[PLUGIN] loaded plugin libbx_iodebug.so
00000000000i[      ] reading configuration from bochsrc.txt
00000000000i[      ] lt_dlhandle is 0x55a169344480
00000000000i[PLUGIN] loaded plugin libbx_x.so
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
Next at t=0
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b          ; ea5be000f0
<bochs:1> c
```

图 1-8 编译与运行 Bochs

- 在 Bochs 窗口中查看该程序，如图 1-9 所示。

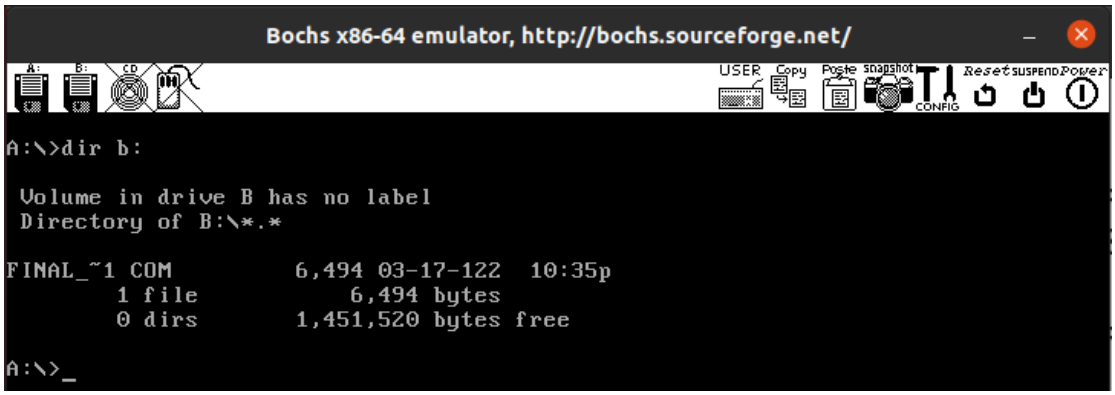

图 1-9 查看该程序

- 运行该程序，运行结果的两个时刻如图 1-10 所示。

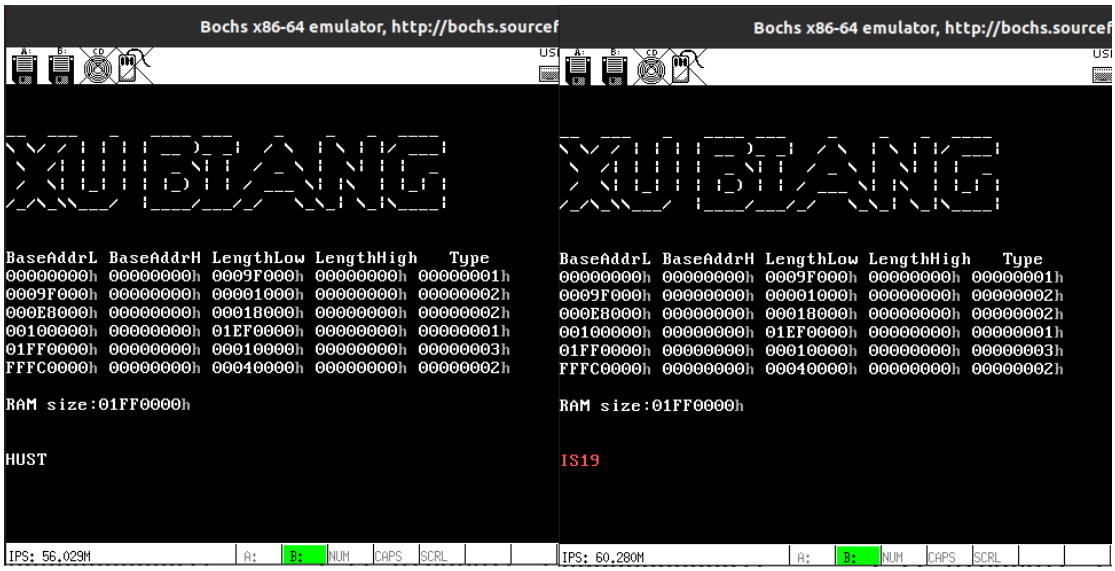

图 1-10 程序运行结果

1.5 实验心得和建议

该任务涉及到保护模式、保护模式下的段/页机制、保护模式下的任务切换以及中断的使用等很多知识，但只要认真阅读和学习任务讲解、《自己动手写操作系统》、《LINUX 内核完全剖析：基于 0.12 内核》以及相关示例代码，就能以示例代码为基础，将多部分知识结合起来，完成任务设计和实现。设计实现中的很多问题，比如由于 TSS 中字段缺失而导致的 JMP TSS 失败、中断重入中 EOI 的返回和寄存器的保存及恢复等，都可以通过大胆尝试、逻辑推理与资料分析相结合的方式解决，从而既学习了理论知识、又提高了动手能力和分析问题的能力。

在我的实现中，程序在进入保护模式后一直处于 ring0 特权级，没有发生特权级的转换，尽管在《自己动手写操作系统》及其示例代码中都有对特权级转换的介绍，但是在学习了相关内容并研读了代码后，我对特权级转换的诸多情况和每种情况下应该满足的条件仍然有些迷茫，希望老师在今后能够对特权级及其转换做出更易于理解的讲解。

1.6 学习和编程实现参考网址

[1] [Orange'S:一个操作系统的实现]bochs 下安装 freedos
https://blog.csdn.net/Lord_sh/article/details/94570386

[2] 《LINUX 内核完全剖析：基于 0.12 内核》
赵炯，P129-131

[3] Switching to User-mode using iret

https://stackoverflow.com/questions/6892421/switching-to-user-mode-using-iret

[4] 10. User mode (and syscalls)
https://web.archive.org/web/20160326062442/http://jamesmolloy.co.uk/tutorial_html/10.-User%20Mode.html

[5] 8253 Timer Chip
https://www.ic.unicamp.br/~celio/mc404s2-03/8253timer.html

[6] Nasm 宏定义
http://www.bytekits.com/nasm/maco.html

2 实验二 设备阻塞工作机制

2.1 实验概述

实验目的：

- 理解和应用“设备就是文件”的概念；
- 熟悉 Linux 设备驱动程序开发过程；
- 理解设备的阻塞和非阻塞工作机制；
- 理解和应用内核同步机制（等待队列）。

实验任务：

- 编写设备驱动程序，对内存缓冲区进行读写；
- 熟悉 Linux 设备驱动程序开发过程；
- 实现设备的阻塞和非阻塞两种工作方式；
- 理解和应用内核等待队列同步机制。

2.2 实验设计思路

任务整体可以分为字符设备的初始化和退出、kfifo 的使用、等待队列的使用、read 和 write 函数四部分。该任务的实现环境为 Ubuntu 20.04.1，内核版本为 5.13.0，工作目录为 Task2。

（一）字符设备的初始化和退出[1]

在“《操作系统原理》课程设计-网安 19 级-2022 春季-04-第一堂课集中讲授.pdf”中，示例程序片段使用的是 miscdevice 杂项设备[2]，因此在初始化阶段和退出阶段只需简单地使用 misc_register()函数和 misc_deregister()函数即可。为了理解设备初始化和退出的更具体流程，以下设计实现中仍然使用字符设备 cdev，因此初始化和退出过程会较为复杂。初始化（chr_init()函数）流程为：

- 使用 alloc_chrdev_region()函数动态申请设备号；
- 使用 cdev_init()函数初始化字符设备，使用 cdev_add()函数将该字符设备添加至系统；
- 使用 class_create()函数创建类；
- 使用 device_create()函数创建设备节点；
- 使用 kfifo_alloc()函数在内核申请缓冲区；
- 使用 init_waitqueue_head()初始化等待队列，使用 mutex_init()初始化互斥锁。

与之对应的退出（chr_exit()函数）流程为：

- 使用 kfifo_free()函数释放缓冲区；
- 使用 device_destroy()函数移除设备节点；
- 使用 class_destroy()函数删除类；
- 使用 cdev_del()函数从系统中移除字符设备；
- 使用 unregister_chrdev_region()函数注销设备号。

（2）kfifo 的使用[3]

在设计实现中，使用到的 kfifo 相关类及函数（宏）有：

- struct kfifo：
  kfifo 类；
- kfifo_alloc()：
  动态分配一个 fifo 缓冲区；
- kfifo_free()：
  释放 fifo 缓冲区；
- kfifo_len()：
  返回 fifo 中使用了的元素个数；
- kfifo_avail()：
  返回 fifo 中未使用的元素个数；
- kfifo_out_peek()：
  从 fifo 中读取数据，但不移除相应数据；
- kfifo_from_user()：
  将数据从用户空间复制到 fifo；
- kfifo_to_user()：
  将数据从 fifo 复制到用户空间，并将相应数据从 fifo 中移除。

同时，kfifo.h 中指出，在仅有一个读取者和一个写入者且不调用 kfifo_reset()函数时，不需要对 fifo 进行锁操作；若存在多个读取者或多个写入者，则需要对读取者或写入者加锁，以防止并行操作导致的错误。因此，在设计实现中，需要使用互斥锁 mutex 对读取和写入操作做出限制，具体见“（4）read 和 write 函数”。

（3）等待队列的使用[4]

在设备的阻塞工作机制中，为了完成进程的阻塞和唤醒，需要使用等待队列，其定义在<linux/wait.h>中[5]，使用到的相关结构及函数（宏）有：

- wait_queue_head_t：
  等待队列类；
- init_waitqueue_head()：
  初始化等待队列；

- wait_event_interruptible()：

  若不满足条件，则睡眠，直到满足条件或接收到信号（仅在等待队列被唤醒时才检查条件），若被信号唤醒则返回-ERESTARTSYS，若因为满足条件而被唤醒则返回 0；

- wake_up_interruptible()：

  尝试唤醒等待队列中的一个处于 INTERRUPTIBLE 状态的进程。

（4）read 和 write 函数

在代码 final_project_task2.c 中，为了在内核消息缓冲区中输出相应的运行跟踪信息及 fifo 缓冲区的状态，添加了较多输出逻辑，导致程序执行路径不够清晰，因此 read 和 write 函数的核心流程参考图 2-1，代码实现仅为在此基础上增加输出信息相关语句。

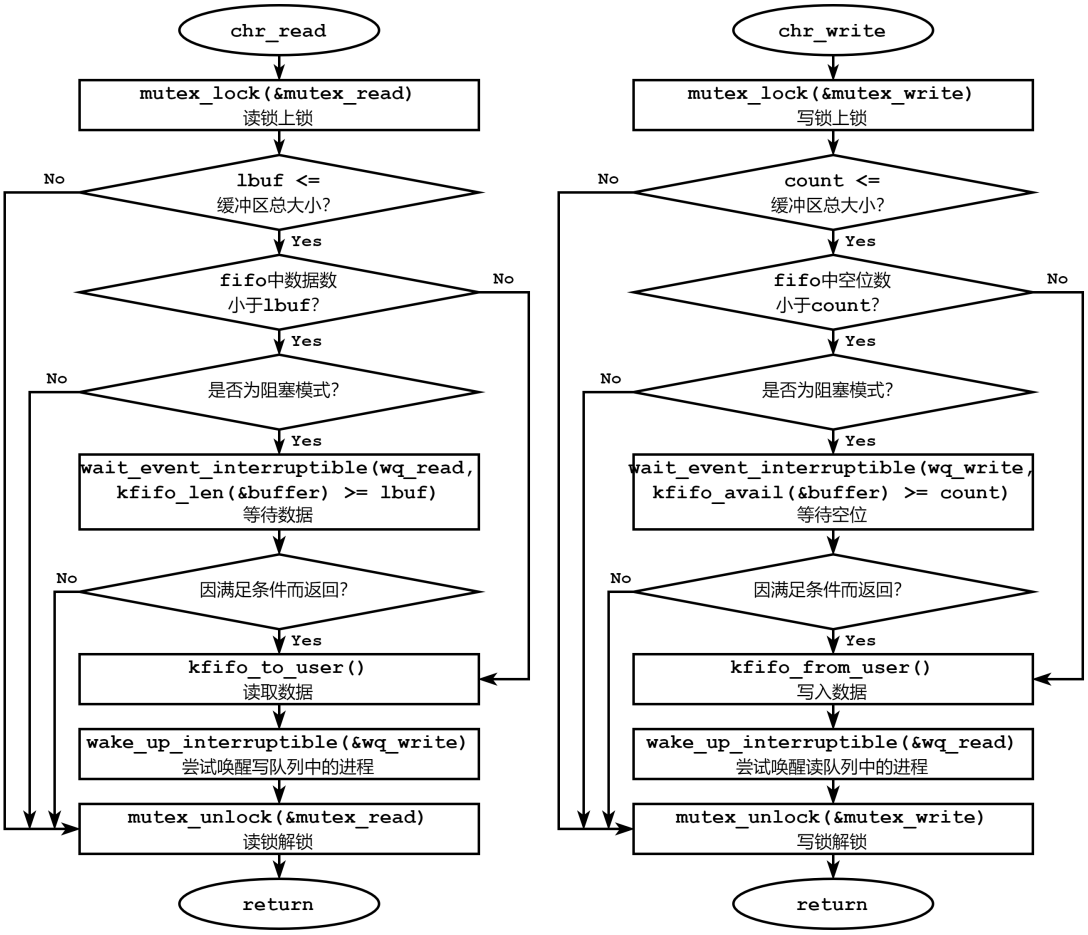

图 2-1 read 和 write 函数的执行流程

图 2-1 中，mutex_read 和 mutex_write 分别为控制读取和写入的互斥锁；wq_read 和 wq_write 分别为等待读取和等待写入的等待队列；lbuf 和 count 均为形参，分别表示希望读取的数据长度和希望写入的数据长度。由“（3）等待队列的使用”中对 wait_event_interruptible()函数的介绍可知，其可能由满足条件后的 wake_up_interruptible()唤醒，也可能由其他信号唤醒，因此在唤醒后应测试返回

值，若因满足条件而返回，则应进行相应的读取或写入操作，否则应返回错误。

2.3 实验程序的难点或核心技术分析

（1）open()函数的 flag 参数[6]

int open(const char *pathname, int oflag, …)函数的第二个参数可以接受 O_RDONLY（只读）、O_WRONLY（只写）、O_RDWR（读写）中的一个与其他 flag 相 OR 的结果，默认情况下 open()函数以可阻塞的方式打开文件（设备），若要指定不可阻塞，则需指定 O_NONBLOCK 或 O_NDELAY 标志。因此，以下分别为使用可阻塞方式打开设备和使用不可阻塞方式打开设备的代码：

```
int fd = open("/dev/ChrDev_XBA", O_RDWR);
int fd = open("/dev/ChrDev_XBA", O_RDWR | O_NONBLOCK);
```

在驱动程序中，相关标志存储在 struct file* file 的 f_flags 属性中，据此可以判断是否以不可阻塞的方式打开设备，用于该功能的宏定义如下：

```
#define IS_NONBLOCK(file) (file->f_flags & O_NONBLOCK)
```

（2）一些特殊情况的处理

- 对于期望长度大于缓冲区总长度的读/写请求，直接返回失败，不使用分次读取/写入；
- 由于使用了互斥锁限制了读进程和写进程的数量，因此同一时刻最多存在一个读进程与一个写进程同时操作 fifo 缓冲区，从而保证了并发安全；
- 由于以上使用的锁策略，可能会使程序陷入死锁，如 fifo 缓冲区总长度为 32 字节，此时有 18 字节数据，读进程请求读取 20 字节数据，而写进程请求写入 20 字节数据，这两个进程便进入了死锁状态，其他进程也无法对缓冲区做出读取与写入操作。为了解决这个问题，可以采取等待一定时间后若仍未满足要求则放弃该请求并重新尝试的策略；
- 对于读程序和写程序，可以采取若请求失败则放弃数据，重新生成数据并请求的策略，也可以采取请求失败后重新尝试的策略，其取决于用户程序的具体需求，在给出的测试程序 test.c 中采取的是放弃数据的策略。

（3）关于进程的状态

对 ps 指令输出的状态信息，通过 man ps 可得如图 2-2 的解释。可知 R 表示正在运行或可运行（在运行队列中）状态，S 表示可中断的睡眠（等待一事件的完成）状态，+表示位于前台进程组（拥有控制终端）。

```
PROCESS STATE CODES
       Here are the different values that the s, stat and state output specifiers (header
       "STAT" or "S") will display to describe the state of a process:

               D    uninterruptible sleep (usually IO)
               I    Idle kernel thread
               R    running or runnable (on run queue)
               S    interruptible sleep (waiting for an event to complete)
               T    stopped by job control signal
               t    stopped by debugger during the tracing
               W    paging (not valid since the 2.6.xx kernel)
               X    dead (should never be seen)
               Z    defunct ("zombie") process, terminated but not reaped by its parent

       For BSD formats and when the stat keyword is used, additional characters may be
       displayed:

               <    high-priority (not nice to other users)
               N    low-priority (nice to other users)
               L    has pages locked into memory (for real-time and custom IO)
               s    is a session leader
               l    is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
               +    is in the foreground process group
```

图 2-2 进程状态码

2.4 运行和测试过程

- 在工作目录 Task1 下，运行 make 完成驱动程序和测试程序的编译，得到 final_project_task2.ko 和 test。
- 运行 make insmod 完成驱动程序的安装，可通过 cat /proc/devices | grep ChrDev_XBA 查看设备，如图 2-3 所示，对应的主设备号为 236（若使用 cat /proc/devices 查看所有设备可知该设备位于 Character devices 字符设备中，由于篇幅限制此处不做展示）。

```
xubiang@ubuntu:~/Desktop/OS/Task2$ cat /proc/devices | grep ChrDev_XBA
236 ChrDev_XBA
```

图 2-3 查看设备

- 非阻塞状态测试：安装设备驱动程序后，运行 sudo ./test 打开测试程序，依次输入 2（不阻塞）和 1（生产者），如图 2-4 所示，当设备缓冲区充足时，写入成功，否则写入失败。使用 dmesg -T --follow 指令[7]查看内核信息如图 2-5 所示。（注：测试程序的输出前缀为[时间@当前进程 PID]；内核信息前缀为[内核时间][设备名::进程 PID::是否阻塞](缓冲区元素数/缓冲区总大小)。）消费者测试过程大致相同，不再赘述。
- 阻塞状态测试-进程阻塞效果：继续使用以上测试环境，运行 sudo ./test 打开测试程序，依次输入 1（阻塞）和 2（消费者），如图 2-6 所示，当设备缓冲区数据充足时，读取成功，否则阻塞等待。内核信息如图 2-7 所示（已使用 dmesg -c 清空缓冲区）。在未阻塞时和阻塞时分别查询测试程序的进程信息如图 2-8 所示，可见程序在正常运行时处于 R+状态（位于前台进程组且处于

正在运行或可运行状态），阻塞时处于 S+状态（位于前台进程组且处于可中断的睡眠状态）。

```
xubiang@ubuntu:~/Desktop/OS/Task2$ sudo ./test
[sudo] password for xubiang:
[16:02:24@3375]: 请选择打开模式(1-阻塞/2-不阻塞/3-结束): 2
[16:02:26@3375]: 请选择角色(1-生产者/2-消费者/3-结束): 1
[16:02:27@3375]: 尝试写入 9 个字符: Cx80shxj1 ...
[16:02:27@3375]: 写入了 9 个字符.
[16:02:28@3375]: 尝试写入 9 个字符: 4aD8AD06U ...
[16:02:28@3375]: 写入了 9 个字符.
[16:02:30@3375]: 尝试写入 7 个字符: EMTT9Fb ...
[16:02:30@3375]: 写入了 7 个字符.
[16:02:31@3375]: 尝试写入 15 个字符: HXLHGTm1R6Aa5kr ...
[16:02:31@3375]: 写入失败, 错误码: -1.
^C
```

图 2-4 非阻塞状态测试：测试程序输出信息

```
xubiang@ubuntu:~/Desktop/OS/Task2$ dmesg -T --follow
[Sat Mar 19 16:02:07 2022] [ChrDev_XBA]: 设备初始化 成功: Major 236, Minor 0, BufferSize 32.
[Sat Mar 19 16:02:25 2022] [ChrDev_XBA::3375::BLK-]: 进程 3375 打开设备.
[Sat Mar 19 16:02:27 2022] [ChrDev_XBA::3375::BLK-]: ( 0/32) 写入  9 字节 ...          []
[Sat Mar 19 16:02:27 2022] [ChrDev_XBA::3375::BLK-]: ( 9/32) 写入  9 字节 成功.       +[Cx80shxj1]
[Sat Mar 19 16:02:28 2022] [ChrDev_XBA::3375::BLK-]: ( 9/32) 写入  9 字节 ...          [Cx80shxj1]
[Sat Mar 19 16:02:28 2022] [ChrDev_XBA::3375::BLK-]: (18/32) 写入  9 字节 成功.       +[4aD8AD06U]
[Sat Mar 19 16:02:29 2022] [ChrDev_XBA::3375::BLK-]: (18/32) 写入  7 字节 ...          [Cx80shxj14aD8AD06U]
[Sat Mar 19 16:02:29 2022] [ChrDev_XBA::3375::BLK-]: (25/32) 写入  7 字节 成功.       +[EMTT9Fb]
[Sat Mar 19 16:02:31 2022] [ChrDev_XBA::3375::BLK-]: (25/32) 写入 15 字节 ...          [Cx80shxj14aD8AD06UEMTT9Fb]
[Sat Mar 19 16:02:31 2022] [ChrDev_XBA::3375::BLK-]: (25/32) 写入 15 字节 失败: 请求过长.
[Sat Mar 19 16:02:32 2022] [ChrDev_XBA::3375::BLK-]: 进程 3375 释放设备.
```

图 2-5 非阻塞状态测试：内核信息

```
xubiang@ubuntu:~/Desktop/OS/Task2$ sudo ./test
[16:16:41@3639]: 请选择打开模式(1-阻塞/2-不阻塞/3-结束): 1
[16:16:47@3639]: 请选择角色(1-生产者/2-消费者/3-结束): 2
[16:16:47@3639]: 尝试读取 1 个字符 ...
[16:16:47@3639]: 读取了 1 个字符: C.
[16:16:49@3639]: 尝试读取 2 个字符 ...
[16:16:49@3639]: 读取了 2 个字符: x8.
[16:16:50@3639]: 尝试读取 3 个字符 ...
[16:16:50@3639]: 读取了 3 个字符: 0sh.
[16:16:51@3639]: 尝试读取 10 个字符 ...
[16:16:51@3639]: 读取了 10 个字符: xj14aD8AD0.
[16:16:52@3639]: 尝试读取 6 个字符 ...
[16:16:52@3639]: 读取了 6 个字符: 6UEMTT.
[16:16:54@3639]: 尝试读取 15 个字符 ...
```

图 2-6 阻塞状态测试：测试程序输出信息

```
xubiang@ubuntu:~/Desktop/OS/Task2$ dmesg -T --follow
[Sat Mar 19 16:16:46 2022] [ChrDev_XBA::3639::BLK+]: 进程 3639 打开设备.
[Sat Mar 19 16:16:47 2022] [ChrDev_XBA::3639::BLK+]: (25/32) 读取  1 字节 ...          [Cx80shxj14aD8AD06UEMTT9Fb]
[Sat Mar 19 16:16:47 2022] [ChrDev_XBA::3639::BLK+]: (24/32) 读取  1 字节 成功.       -[C]
[Sat Mar 19 16:16:48 2022] [ChrDev_XBA::3639::BLK+]: (24/32) 读取  2 字节 ...          [x80shxj14aD8AD06UEMTT9Fb]
[Sat Mar 19 16:16:48 2022] [ChrDev_XBA::3639::BLK+]: (22/32) 读取  2 字节 成功.       -[x8]
[Sat Mar 19 16:16:50 2022] [ChrDev_XBA::3639::BLK+]: (22/32) 读取  3 字节 ...          [0shxj14aD8AD06UEMTT9Fb]
[Sat Mar 19 16:16:50 2022] [ChrDev_XBA::3639::BLK+]: (19/32) 读取  3 字节 成功.       -[0sh]
[Sat Mar 19 16:16:51 2022] [ChrDev_XBA::3639::BLK+]: (19/32) 读取 10 字节 ...          [xj14aD8AD06UEMTT9Fb]
[Sat Mar 19 16:16:51 2022] [ChrDev_XBA::3639::BLK+]: ( 9/32) 读取 10 字节 成功.       -[xj14aD8AD0]
[Sat Mar 19 16:16:52 2022] [ChrDev_XBA::3639::BLK+]: ( 9/32) 读取  6 字节 ...          [6UEMTT9Fb]
[Sat Mar 19 16:16:52 2022] [ChrDev_XBA::3639::BLK+]: ( 3/32) 读取  6 字节 成功.       -[6UEMTT]
[Sat Mar 19 16:16:53 2022] [ChrDev_XBA::3639::BLK+]: ( 3/32) 读取 15 字节 ...          [9Fb]
[Sat Mar 19 16:16:53 2022] [ChrDev_XBA::3639::BLK+]: ( 3/32) 读取 15 字节 等待数据 ...
```

图 2-7 阻塞状态测试：内核信息

```
xubiang@ubuntu:~$ ps -au | egrep '(USER|root)'
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root        3638  0.0  0.1  11928  4572 pts/1    S+   16:16   0:00 sudo ./test
root        3639 19.7  0.0   2496   700 pts/1    R+   16:16   0:01 ./test
xubiang     3641  0.0  0.0   8908   724 pts/2    S+   16:16   0:00 grep -E --color=auto (USER|root)
xubiang@ubuntu:~$ ps -au | egrep '(USER|root)'
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root        3638  0.0  0.1  11928  4572 pts/1    S+   16:16   0:00 sudo ./test
root        3639 44.3  0.0   2496   700 pts/1    S+   16:16   0:06 ./test
xubiang     3658  0.0  0.0   8908   656 pts/2    S+   16:16   0:00 grep -E --color=auto (USER|root)
```

图 2-8 阻塞状态测试：测试程序的运行状态变化

- 阻塞状态测试-进程唤醒效果：继续使用以上测试环境，此时读进程 3639 处于阻塞状态，另外打开一个测试程序向内核缓冲区写入数据，待缓冲区有足够数据后，读进程 3639 将被唤醒。写进程的输出信息如图 2-9 所示，读进程的输出信息如图 2-10 所示，内核信息如图 2-11 所示。

```
xubiang@ubuntu:~/Desktop/OS/Task2$ sudo ./test
[sudo] password for xubiang:
[16:26:04@3899]: 请选择打开模式(1-阻塞/2-不阻塞/3-结束): 1
[16:26:08@3899]: 请选择角色(1-生产者/2-消费者/3-结束): 1
[16:26:11@3899]: 尝试写入 14 个字符: 8VgP48XcPuBf4C ...
[16:26:11@3899]: 写入了 14 个字符.
[16:26:12@3899]: 尝试写入 9 个字符: c5fDaAi7E ...
[16:26:12@3899]: 写入了 9 个字符.
[16:26:13@3899]: 尝试写入 12 个字符: iLe3RwIa4im0 ...
[16:26:13@3899]: 写入了 12 个字符.
^C
```

图 2-9 阻塞状态测试：新开启的写进程（3899）的输出信息

```
[16:16:54@3639]: 尝试读取 15 个字符 ...
[16:26:11@3639]: 读取了 15 个字符: 9Fb8VgP48XcPuBf.
[16:26:12@3639]: 尝试读取 12 个字符 ...
[16:26:13@3639]: 读取了 12 个字符: 4Cc5fDaAi7Ei.
^C
```

图 2-10 阻塞状态测试：被阻塞的进程（3639）被写进程（3899）唤醒的输出信息

```
xubiang@ubuntu:~/Desktop/OS/Task2$ dmesg -T --follow
[Sat Mar 19 16:26:08 2022] [ChrDev_XBA::3899::BLK+]: 进程 3899 打开设备.
[Sat Mar 19 16:26:11 2022] [ChrDev_XBA::3899::BLK+]: ( 3/32) 写入 14 字节 ...          [9Fb]
[Sat Mar 19 16:26:11 2022] [ChrDev_XBA::3899::BLK+]: (17/32) 写入 14 字节 成功.       +[8VgP48XcPuBf4C]
[Sat Mar 19 16:26:11 2022] [ChrDev_XBA::3639::BLK+]: ( 2/32) 读取 15 字节 成功.       -[9Fb8VgP48XcPuBf]
[Sat Mar 19 16:26:12 2022] [ChrDev_XBA::3639::BLK+]: ( 2/32) 读取 12 字节 ...          [4C]
[Sat Mar 19 16:26:12 2022] [ChrDev_XBA::3639::BLK+]: ( 2/32) 读取 12 字节 等待数据 ...
[Sat Mar 19 16:26:12 2022] [ChrDev_XBA::3899::BLK+]: ( 2/32) 写入  9 字节 ...          [4C]
[Sat Mar 19 16:26:12 2022] [ChrDev_XBA::3899::BLK+]: (11/32) 写入  9 字节 成功.       +[c5fDaAi7E]
[Sat Mar 19 16:26:13 2022] [ChrDev_XBA::3899::BLK+]: (11/32) 写入 12 字节 ...          [4Cc5fDaAi7E]
[Sat Mar 19 16:26:13 2022] [ChrDev_XBA::3899::BLK+]: (23/32) 写入 12 字节 成功.       +[iLe3RwIa4im0]
[Sat Mar 19 16:26:13 2022] [ChrDev_XBA::3639::BLK+]: (11/32) 读取 12 字节 成功.       -[4Cc5fDaAi7Ei]
[Sat Mar 19 16:26:14 2022] [ChrDev_XBA::3899::BLK+]: 进程 3899 释放设备.
[Sat Mar 19 16:26:14 2022] [ChrDev_XBA::3639::BLK+]: 进程 3639 释放设备.
```

图 2-11 阻塞状态测试：写进程（3899）唤醒被阻塞的进程（3639）的内核信息

- 运行 make rmmod 卸载设备驱动，运行 make clean 清理生成文件。

2.5 实验心得和建议

略

2.6 学习和编程实现参考网址

[1] Linux 设备驱动之字符设备驱动
https://blog.csdn.net/andylauren/article/details/51803331

[2] misc_register、register_chrdev 的区别总结
https://blog.csdn.net/nanhangfengshuai/article/details/50533230

[3] kfifo.h - include/linux/kfifo.h - Linux source code (v5.13) - Bootlin
https://elixir.bootlin.com/linux/v5.13/source/include/linux/kfifo.h

[4] Character device drivers - The Linux Kernel documentation
https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html#waiting-queues

[5] wait.h - include/linux/wait.h - Linux source code (v5.13) - Bootlin
https://elixir.bootlin.com/linux/v5.13/source/include/linux/wait.h

[6] fcntl.h - tools/include/uapi/asm-generic/fcntl.h - Linux source code (v5.13) - Bootlin
https://elixir.bootlin.com/linux/v5.13/source/tools/include/uapi/asm-generic/fcntl.h

[7] How to Use the dmesg Command on Linux
https://www.howtogeek.com/449335/how-to-use-the-dmesg-command-on-linux/