本 科 生 课 程 设 计 报 告

课　　程：操作系统原理课程设计

题　　目：任务切换与设备阻塞机制研究

院　　系：软件学院

专业班级：软件工程 2003 班

学　　号：U202010851

姓　　名：侯皓斐

2022 年　06 月 01 日

目　录

1 任务一 任务切换机制

1.1 实验任务概述

实验目的如下:

1.理解保护模式的概念

2.掌握保护模式程序的编写

3.理解 CPU 对段机制/页机制的支持

4.理解段机制/页机制的原理和简单应用

5.理解任务的概念和任务切换的过程

6.阅读和理解 X86 保护模式初始化程序（pmtest1~pmtest5）

7.阅读和理解 X86 段和页工作机制程序（pmtest6~pmtest7）

8.编程实现一套页目录和页表，两个任务，并实现任务切换。

启动保护模式，建立两个任务（两个任务分别循环输出“HUST”和“MRSU”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 中断芯片，实现两个任务在时钟驱动下进行切换。

其步骤需要如下：CPU 进入保护模式；初始化 GDT，LDT，IDT，TSS 等数据结构；对内存中建立页表和页目录，两个任务每个任务使用各自对应的页表，每个任务简单地输出 A 或 B；初始化 8253 时钟模块和 8259 中断模块，在时钟驱动下支持 2 个任务切换。

1.2 实验设计思路

本实验基于银河麒麟操作系统 V10 桌面版（Linux 内核版本为 5.17.5）上的

bochs 虚拟机与 freedos.img 实现。

一、实现 bochs 下 x86 虚拟机的搭建

```
moonlight@moonlight-VMware:~/design1$ bximage
========================================================================
                                bximage
  Disk Image Creation / Conversion / Resize and Commit Tool for Bochs
         $Id: bximage.cc 13481 2018-03-30 21:04:04Z vruppert $
========================================================================

1. Create new floppy or hard disk image
2. Convert hard disk image to other format (mode)
3. Resize hard disk image
4. Commit 'undoable' redolog to base image
5. Disk image info

0. Quit

Please choose one [0] 1

Create image

Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd] fd

Choose the size of floppy disk image to create.
Please type 160k, 180k, 320k, 360k, 720k, 1.2M, 1.44M, 1.68M, 1.72M, or 2.88M.
 [1.44M]

What should be the name of the image?
[a.img] pmtest.img

Creating floppy image 'pmtest.img' with 2880 sectors

The following line should appear in your bochsrc:
  floppya: image="pmtest.img", status=inserted
moonlight@moonlight-VMware:~/design1$ ls
bochsrc.txt  freedos-img  freedos.img  freedos-img.tar.gz  pmtest.img
```

图 1 任务软盘环境的创建

图 2：bochs 虚拟机的配置文件

首先是在线安装 nasm；然后在线安装虚拟机 bochs（内含 bximage 工具），期间应注意检查 bochs 安装目录，为修改 bochrc 文件备用；在 bochs 官网下载

freedos 软盘映像文件；解压 freedos 后内含 4 个文件仅保留 a.img 并更名为 freedos.img 备用。软盘 A：启动盘，系统盘，而使用 bximage 工具创建用户虚拟软盘 pmtest.img,如图 1;配置 bochs 的运行控制文件 bochsrc.txt 如图 2;使用 bochs 命令打开虚拟机，终端中输入 c，进入 freedos。

二、实现 hustmrsu.asm，完成段页式变换加任务切换。

运行的程序主要由 16 位程序段，32 位程序段，两个运行的任务（LDT 与 TSS）和我们的中断处理程序组成。

16 位程序段的任务是得到内存数和相关信息并输出，初始化 GDT 与 LDT 表内的描述符信息，然后经过一系列措施（保存中断屏蔽寄存器值，打开地址线 A20，加载 GDTR，加载 IDTR），最终进入保护模式，同时跳入 32 位程序段。

```
; 准备切换到保护模式
mov eax, cr0
or  eax, 1
mov cr0, eax

; 真正进入保护模式
jmp dword SelectorCode32:0  ; 扌
```

图 3：程序进入保护模式的历史性时刻

32 位程序段经过了一系列预备措施，首先是 call Init8259A，完成对 8259A 中断控制器的初始化，call PagingDemo 后进入函数实现分页机制的打开，初始化两个页目录和对应的页表。然后加载 LDTR，进入我们的任务 1。（以页目录 1）

```
513     call    Init8259A
514
515     call    DispMemSize     ; 显示内存信息
516
517     call    PagingDemo      ; 演示改变页目录的效果
518
519     mov     ax, SelectorLDT1
520     lldt    ax
521     jmp     SelectorLDT1Task:0
522
523     call    SetRealmode8259A
524
525     jmp SelectorCode16:0
```

图 4：32 位段做的主要事情

两个任务段较为简单，依照 TSS 的格式填好其任务状态段，两个进程均死循环跳入一个函数，这个函数再次跳入输出任务，由于页表映射不用，跳入的地址不同，两个任务运行结果不同。

当时钟中断到来时，时钟中断处理函数查看当前进程，并利用 jmp tss 语句跳入另一个进程。完成简单的进程调度。

```
620 ClockHandler    equ _ClockHandler - $$
621     mov al, 20h
622     out 20h, al             ; 输出 EOI
623     ;iretd
624
625     mov ax, SelectorData
626     mov ds, ax
627
628     mov ax, word [inwhatcode]
629     cmp ax, 1
630     je  jumptotss2
631     mov ax, 1
632     mov word [inwhatcode], ax
633     jmp SelectorTSS1:0
634     jmp endclockhandle
635 jumptotss2:
636     mov ax, 2
637     mov word [inwhatcode], ax
638     jmp SelectorTSS2:0
639 endclockhandle:
640     iretd
```

图 5：时钟中断处理函数的编写，完成简单的进程调度

运行的程序综上依赖于如下的数据结构，GDT，IDT，LDT，TSS，还有页目录与页表。而且为了实现进程切换应当在内核数据段内保存一个变量 NowProcess。

GDT 中应该保存有 16 位程序段，32 位程序段，各个进程的内核堆栈段，LDT 段等的描述符，并且后续利用宏产生这些描述符的选择子。

每个任务有一个 LDT 和 TSS，而每个进程要有用户程序段，用户堆栈段，用户数据段，故 LDT 中应该有这三者的描述描述符，并通过宏产生三者的选择子。TSS 中存放着其重要的寄存器，堆栈，运行资源相关信息。

```
23  [SECTION .gdt]
24  ; GDT
25  ;                                       段基址,       段界限     , 属性
26  LABEL_GDT:          Descriptor          0,                0, 0           ; 空描述符
27  LABEL_DESC_NORMAL:  Descriptor          0,            0ffffh, DA_DRW         ; Normal 描述符
28  LABEL_DESC_FLAT_C:  Descriptor          0,           0fffffh, DA_CR | DA_32 | DA_LIMIT_4K; 0 ~ 4G
29  LABEL_DESC_FLAT_RW: Descriptor          0,           0fffffh, DA_DRW | DA_LIMIT_4K   ; 0 ~ 4G
30  LABEL_DESC_CODE32:  Descriptor          0,  SegCode32Len - 1, DA_CR | DA_32  ; 非一致代码段, 32
31  LABEL_DESC_CODE16:  Descriptor          0,            0ffffh, DA_C           ; 非一致代码段, 16
32  LABEL_DESC_DATA:    Descriptor          0,       DataLen - 1, DA_DRW            ; Data
33  LABEL_DESC_STACK:   Descriptor          0,        TopOfStack, DA_DRWA | DA_32; Stack, 32 位
34  LABEL_DESC_STACK1:  Descriptor          0,       TopOfStack1, DA_DRWA | DA_32; Stack1, 32 位
35  LABEL_DESC_LDT1:    Descriptor          0,       LDT1Len - 1, DA_LDT         ; LDT1
36  LABEL_DESC_TSS1:    Descriptor          0,       TSSLen1 - 1, DA_386TSS      ; TSS1
37  LABEL_DESC_STACK2:  Descriptor          0,       TopOfStack2, DA_DRWA | DA_32; Stack2, 32 位
38  LABEL_DESC_LDT2:    Descriptor          0,       LDT2Len - 1, DA_LDT         ; LDT2
39  LABEL_DESC_TSS2:    Descriptor          0,       TSSLen2 - 1, DA_386TSS      ; TSS2
40  LABEL_DESC_VIDEO:   Descriptor    0B8000h,            0ffffh, DA_DRW         ; 显存首地址
41  ; GDT 结束
42
43  GdtLen      equ $ - LABEL_GDT   ; GDT长度
44  GdtPtr      dw  GdtLen - 1      ; GDT界限
45              dd  0               ; GDT基地址
```

图 6：程序运行依赖的 GDT

```
221  ; IDT
222  [SECTION .idt]
223  ALIGN   32
224  [BITS   32]
225  LABEL_IDT:
226  ; 门                                   目标选择子,            偏移, DCount, 属性
227  %rep 32
228              Gate    SelectorCode32, SpuriousHandler,      0, DA_386IGate
229  %endrep
230  .020h:      Gate    SelectorCode32,    ClockHandler,      0, DA_386IGate
231  %rep 95
232              Gate    SelectorCode32, SpuriousHandler,      0, DA_386IGate
233  %endrep
234  .080h:      Gate    SelectorCode32,  UserIntHandler,      0, DA_386IGate
235
236  IdtLen      equ $ - LABEL_IDT
237  IdtPtr      dw  IdtLen - 1  ; 段界限
238              dd  0       ; 基地址
239  ; END of [SECTION .idt]
```

图 7：程序运行以来的 IDT

```
916  ; LDT1
917  [SECTION .ldt1]
918  ALIGN   32
919  LABEL_LDT1:
920  ;                                       段基址       段界限     ,   属性
921  LABEL_LDT1_DESC_TASK:   Descriptor        0,     Task1Len - 1,   DA_C + DA_32  ; Code, 32 位
922  LABEL_LDT1_DESC_DATA:   Descriptor        0,     Data1Len - 1,   DA_DRW        ; Data, 32 位
923  LABEL_LDT1_DESC_STACK:  Descriptor        0,        Stack1Len,   DA_DRW + DA_32; Stack, 32 位
924  LDT1Len     equ $ - LABEL_LDT1
925
926  ; LDT 选择子
927  SelectorLDT1Task    equ LABEL_LDT1_DESC_TASK - LABEL_LDT1 + SA_TIL
928  SelectorLDT1Data    equ LABEL_LDT1_DESC_DATA - LABEL_LDT1 + SA_TIL
929  SelectorLDT1Stack   equ LABEL_LDT1_DESC_STACK - LABEL_LDT1 + SA_TIL
930  ; END of [SECTION .ldt]
```

图 8：任务 1 运行依赖的 LDT 与 LDT 选择子

```
151 LABEL_TSS1:             ;定义LABEL_TSS
152         DD  0           ; Back
153         DD  TopOfStack1 ; 0 级堆栈   //内层ring0级堆栈放入TSS中
154         DD  SelectorStack1;
155         DD  0           ; 1 级堆栈
156         DD  0           ;
157         DD  0           ; 2 级堆栈
158         DD  0           ;               //TSS中最高只能放入Ring2级堆栈，
159         DD  PageDirBase0; CR3
160         DD  0           ; EIP
161         DD  0           ; EFLAGS
162         DD  0           ; EAX
163         DD  0           ; ECX
164         DD  0           ; EDX
165         DD  0           ; EBX
166         DD  Stack1Len   ; ESP
167         DD  0           ; EBP
168         DD  0           ; ESI
169         DD  0           ; EDI
170         DD  0           ; ES
171         DD  SelectorLDT1Task            ; CS
172         DD  SelectorLDT1Stack           ; SS
173         DD  0           ; DS
174         DD  0           ; FS
175         DD  SelectorVideo           ; GS
176         DD  SelectorLDT1; LDT
```

图 9：任务 1 运行和跳转依赖的 TSS（部分）

正如图九，页目录和页表的起始位置被固定，在 PagingDemo 中页目录和页表 1 按照线性地址直接填写，如图 11；页目录和页表 2 仅修改 LinearAddrDemo 的线性地址对应的页表物理地址位 ProcTask2。

```
 9  PageDirBase0        equ 200000h ; 页目录开始地址:   2M
10  PageTblBase0        equ 201000h ; 页表开始地址:     2M +  4K
11  PageDirBase1        equ 210000h ; 页目录开始地址:   2M + 64K
12  PageTblBase1        equ 211000h ; 页表开始地址:     2M + 64K + 4K
13
14  LinearAddrDemo  equ 00401000h
15  ProcTask1       equ 00401000h
16  ProcTask2       equ 00501000h
17
18  ProcPagingDemo  equ 00301000h
```

图 10：页目录与页表的起始地主

```
707     ; 初始化页目录
708     mov ax, SelectorFlatRW
709     mov es, ax
710     mov edi, PageDirBase1   ; 此段首地址为 PageDirBase1
711     xor eax, eax
712     mov eax, PageTblBase1 | PG_P  | PG_USU | PG_RWW
713     mov ecx, [PageTableNumber]
714 .4:
715     stosd
716     add eax, 4096       ; 为了简化，所有页表在内存中是连续的.
717     loop    .4
718
719     ; 再初始化所有页表
720     mov eax, [PageTableNumber]  ; 页表个数
721     mov ebx, 1024       ; 每个页表 1024 个 PTE
722     mul ebx
723     mov ecx, eax        ; PTE个数 = 页表个数 * 1024
724     mov edi, PageTblBase1   ; 此段首地址为 PageTblBase1
725     xor eax, eax
726     mov eax, PG_P  | PG_USU | PG_RWW
727 .5:
728     stosd
729     add eax, 4096       ; 每一页指向 4K 的空间
730     loop    .5
```

图 11：页目录和页表的初始化

因此当我们经过 nasm hustmrsu.asm -o hustmrsu.com，sudo mount pmtest.img /mnt/floppy，sudo cp hustmrsu.com /mnt/floppy 等一系列步骤后打开 bochs 虚拟机，运行 b 软盘上的 hustmrsu.com 文件，其便可以出现 HUST 和 MRSU 交替出现的景象。而当我们按下回车，程序进入了保护模式，开启了页式地址映射，初始化了各类中断函数。其运行流程如下：

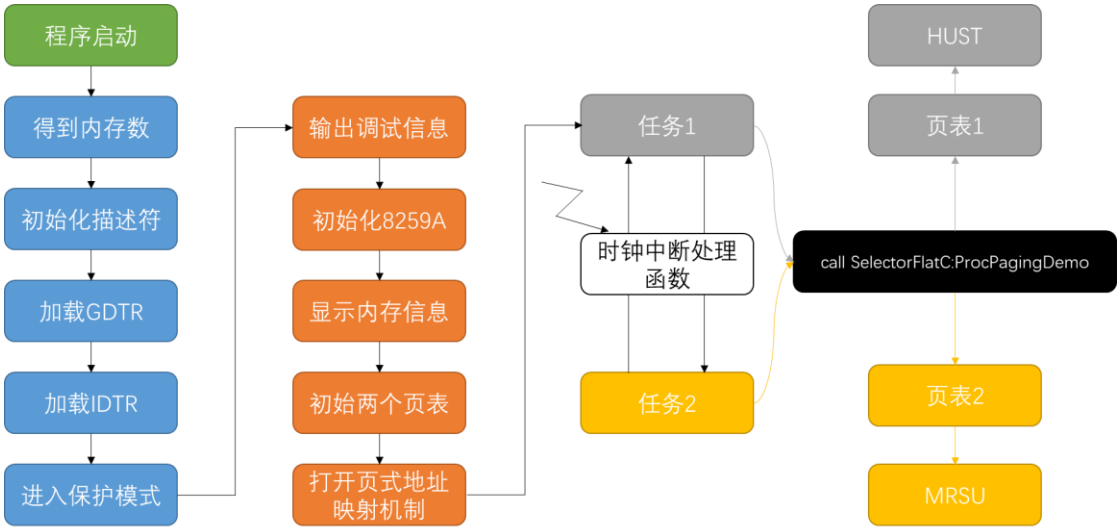

图 12：hustmrsu.asm 实现的流程与进程调度方案

1.3 程序的重点/难点/核心技术分析

一、设计过程中的最重难点是保护模式下汇编语言程序设计

之前的汇编语言程序设计均基于实模式，这次在深入了解保护模式的基础上进行编程仍然很有难度，实验主要借鉴于渊的代码，学习了不少知识。

这个过程中也出现了不少的错误。调 Bug 的过程十分痛苦。其中经常性的出现类似图 13 莫名其妙的错误。

首先是程序无法进入任务，我无法找到错误的地点，于是在可能出错的地方进行输出调试，如图 14。若不能成功运行到这里屏幕上便不会有@。

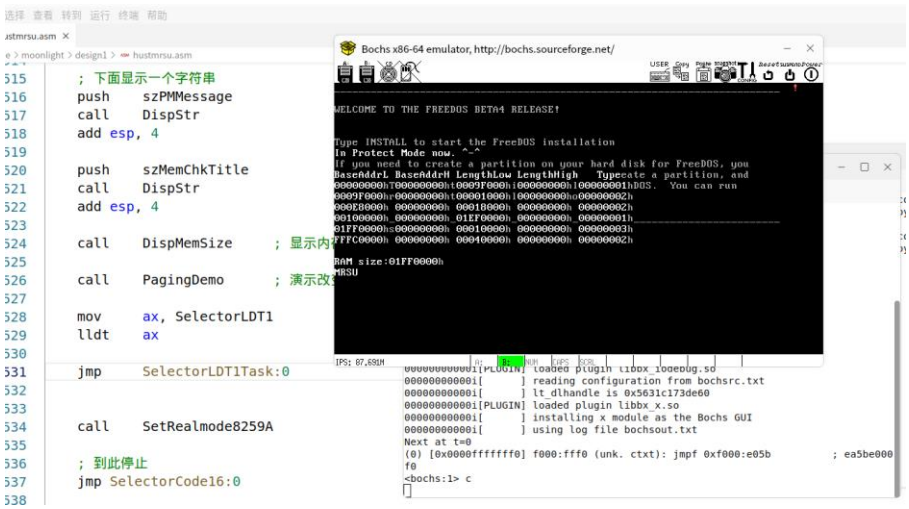

图 13：由于某种原因程序出现错误进入中断

```
985 LABEL_Task2:
986     mov ah , 0Fh
987     mov al , '@'
988     mov [gs:((80 * 14 + 2) * 2)], ax     ; 屏幕第 17 行, 第 2 列。
989     sti
990
991     mov  ax, SelectorVideo
992     mov  gs, ax
993     call SelectorFlatC:ProcPagingDemo
994     jmp LABEL_Task2
```

图 14：如何进行输出调试

首先我找到了程序逻辑上的错误，跳转过程不够清晰。

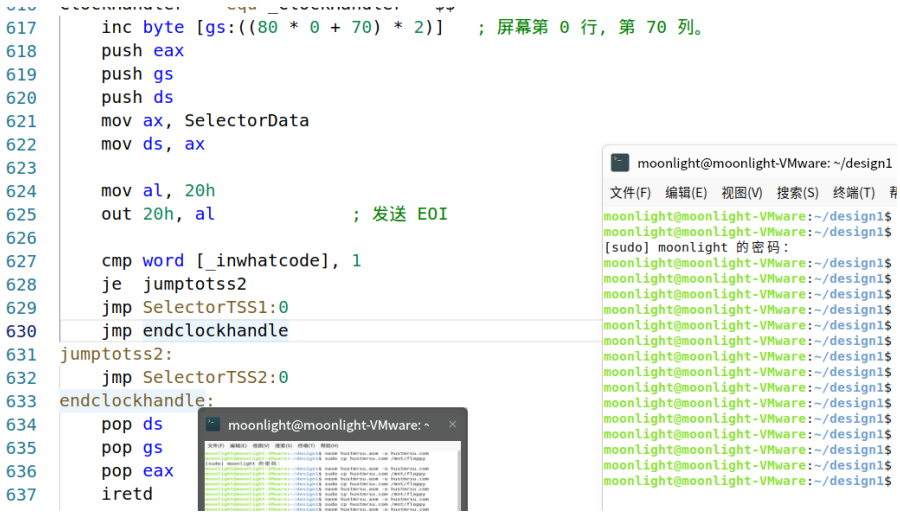

图 15：程序出错集中于时钟中断处理函数

而且保护模式下地址的访问也出了乱子，在实模式下变量应该加_使用，而保护模式下完全不需要提醒 nasm。

```
629     cmp word [_inwhatcode], 1
630     je  jumptotss2
631     mov word [_inwhatcode], 1
632     jmp SelectorTSS1:0
633     jmp endclockhandle
634 jumptotss2:
635     mov word [_inwhatcode], 2
636     jmp SelectorTSS2:0
637 endclockhandle:
638     iretd
```

图 16：众多错误的来源

二、重难点也在与理顺逻辑关系，实现清晰恰当的代码

汇编语言本身十分艰涩难懂，写代码时更应该注意逻辑，注意变量名的合理化，其中各种函数，各种表，各种跳转，各种指令（尤其是宏汇编指令和机器指令）要区分好。

1.4 运行和测试过程

因此当我们经过 nasm hustmrsu.asm -o hustmrsu.com，sudo mount pmtest.img /mnt/floppy，sudo cp hustmrsu.com /mnt/floppy 等一系列步骤后打开 bochs 虚拟机，终端上输入 c，运行 b 软盘上的 hustmrsu.com 文件，其便可以出现 HUST 和 MRSU 交替出现的景象。而当我们按下回车，程序进入了保护模式，开启了页式地址映射，初始化了各类中断函数。最终得以运行，其出现 HUST 和 MRSU 交相出现的场景，如图 17。

图 17：输出 HUST

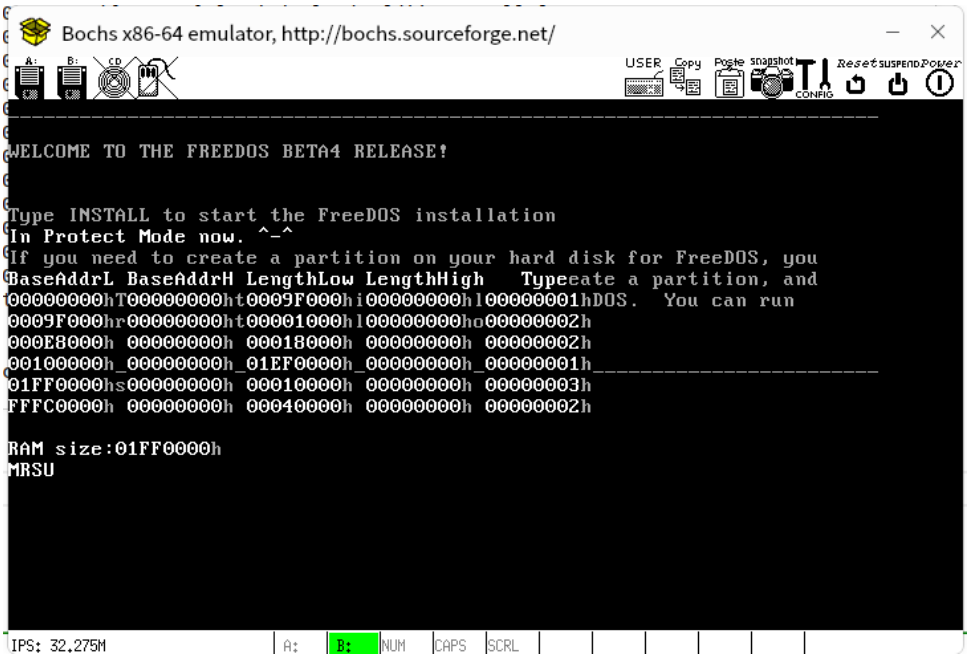

图 18：输出 MRSU

1.5 实验心得和建议

任务一结束后，我结下了老师的附加任务，完成附加任务后实现了 supertask。总之，使用汇编语言实现了两个到多个的基于优先数的进程调度。通过这次试验加深了我对保护模式，进程，中断处理函数等概念的理解，深入学习了保护模式

下汇编程序的编写，特别是基于选择子的段间跳转和页表机制，也锻炼了自己的分析问题，寻找资料，解决问题的能力，编程能力与逻辑思维能力。而我实现的只是一个操作系统的皮毛的皮毛，足见操作系统的复杂。

1.6 学习和编程实现参考网址

[1]于渊. ORANGE’S 一个操作系统的实现[M]. 北京：电子工业出版社，2008.

[2]苏曙光. 操作系统原理（慕课版）[M]. 北京：人民邮电出版社，2022.

[3]NASM-The Netwide Assembler(version 2.15.05)

[4]http://yanyahua.com/kernel/tss/

[5]https://www.cnblogs.com/chenchenluo/archive/2012/11/20/2779763.html

[6]https://blog.csdn.net/judyge/article/details/52346963

[7]https://blog.csdn.net/u012323667/article/details/79266623

[8]https://blog.csdn.net/qq_45740212/article/details/113469718

[9]https://bbs.csdn.net/topics/380098131

[10]https://www.jianshu.com/p/19041feed87b

[11]https://bbs.csdn.net/topics/380098131

[12]https://blog.csdn.net/cherisegege/article/details/80370186

[13]https://blog.csdn.net/djrblog/article/details/20841219

[14]https://blog.csdn.net/qq_26858401/article/details/116212611

[15]https://blog.csdn.net/m0_46125480/article/details/120385476

[16]http://c.biancheng.net/cpp/html/66.html

[17]http://c.biancheng.net/view/3752.html

[18]https://blog.csdn.net/qq_30624591/article/details/105114419

[19]https://blog.csdn.net/Bruno_Mars/article/details/100061793

[20]https://blog.csdn.net/poptar/article/details/111686050

[21]https://blog.csdn.net/syflyhua/article/details/9096031

[22]https://blog.csdn.net/deniece1/article/details/102934003

[23]https://blog.csdn.net/deniece1/article/details/103522866

[24]https://blog.csdn.net/weixin_33691817/article/details/94206295

[25]https://blog.csdn.net/weixin_30781775/article/details/99163390

2 任务二 设备阻塞机制

2.1 实验任务概述

实验目的如下:

1.理解和应用“设备就是文件”的概念

2.熟悉 Linux 设备驱动程序开发过程

3.理解和实现设备的阻塞和非阻塞工作机制

4.理解和应用内核同步机制（等待队列）

5.编写设备驱动程序，对内存缓冲区进行读写

6.熟悉 Linux 设备驱动程序开发过程

编写驱动程序，支持应用程序对内核缓冲区的读写。其中我们需要实现:

1.设定内核缓冲区大小（例如 32 字节）

2.缓冲区是环形缓冲区，驱动程序维护两个读写指针

3.缓冲区按序读写，每个数据的读写不重复，不遗漏

4.编写若干个应用程序，循环读或写缓冲区的若干字节

5.当缓冲区有足够的数据读就读，否则就阻塞进程，直到有足够数据

6.当缓冲区有足够的空位写就写，否者就阻塞进程，直到有足够空位

7.驱动程序内部维护缓冲区的读写，并适时阻塞或唤醒相应进程

观察缓冲区变化与读/写进程的阻塞/被唤醒的同步情况。

2.2 实验设计思路

我们具体的实现应当参考实验四的一、三。其基本上也是编写内核模块并且注册设备文件,建立设备文件结构体。实现设备时我们要注意实现内核的缓冲区,支持阻塞和非阻塞两种工作方式,而且实现内核同步机制。将任务分解,实现便容易了起来。

```
7  #define DEV_NAME "BlockFIFODevV3"
8  #define BUFFER_SIZE 32
9  MODULE_LICENSE("GPL");
10
11 DEFINE_KFIFO(FIFOBuffer, char, BUFFER_SIZE);
12 struct mutex Mutex;
13 static struct device *my_miscdevice;
14 wait_queue_head_t read_queue;
15 wait_queue_head_t write_queue;
```

图 19：实现中主要依赖的对象与方法

实现内核缓冲区，我们使用内核现成的 KFIFO 对象，其保证了循环的的读写。而我们也利用 wait_queue_head_t 队列和 wait_event_interruptible，wake_up_interruptible 方法实现任务的阻塞与唤醒。使用内核 mutex 锁完成临界区的互斥访问。

于是我们可如下编写内核缓冲区的 Read 函数。

```
25 static ssize_t
26 DevRead(struct file *file,char __user *buf, size_t count,loff_t *ppos) {
27     int actual_readed = 0, ret = 0;
28     if (kfifo_is_empty(&FIFOBuffer)) {
29         if(file->f_flags & O_NONBLOCK)
30             return 0;
31         wait_event_interruptible(read_queue, !kfifo_is_empty(&FIFOBuffer));
32     }
33     mutex_lock(&Mutex);
34     ret = kfifo_to_user(&FIFOBuffer, buf, count, &actual_readed);
35     mutex_unlock(&Mutex);
36     if (!kfifo_is_full(&FIFOBuffer))
37         wake_up_interruptible(&write_queue);
38     return actual_readed;
39 }
```

图表 20：BlockFIFODevV3 设备的 Read 函数，即读出缓冲区

我们也可同理编写设备的 Write 函数。

模块注册时，我们将内核中各类所需对象初始化，并完成杂项设备的设备文件的注册。

```
72  static int __init DevInit(void){
73      int ret = misc_register(&miscDeviceFIFOBlock);
74      mutex_init(&Mutex);
75      my_miscdevice = miscDeviceFIFOBlock.this_device;
76      init_waitqueue_head(&read_queue);
77      init_waitqueue_head(&write_queue);
78      printk("succeeded register char device: %s\n", DEV_NAME);
79      return ret;
80  }
```

图 21：模块的注册函数

综上，当我们编写完内核模块后，再编写好 Makefile 文件，执行 make，sudo insmod BlockFIFODevV3.ko，如图 22 我们测试一下设备可否正常运行。即可用我们提前编写好的以阻塞模式和非阻塞模式打开设备的程序运行。

```
moonlight@moonlight-VMware:~/design2$ sudo ./test.out
1:520 10It is fate?
2:1206 314 521I can not believe!
3:实现中华民族伟大复兴的中国梦！
```

图 22：测试设备正常运行

2.3 程序的重点/难点/核心技术分析

鉴于之前已有过编写内核模块与设备驱动的相关经验，此时的难度便下降了不少。我们可以认为此处难度在于各种对象的综合学习与使用。

核心技术在于如何保证支持阻塞和非阻塞两种工作方式，而且实现内核同步机制两者同时实现，其关键在于 PV 操作。综合分析后，我们可以写出如图 20 一般的 Read 函数，保证以非阻塞模式打开设备的任务在可读字节数为 0 时直接返回 0，其可一直判断循环尝试读入。而以阻塞模式打开设备的任务将被直接阻塞。使用 mutex 保证读写缓冲区不会同时多个进程进行操作。最后尝试唤醒被阻塞的写进程。

其中比较难受的是，老师在提示的时候让我们建立一个结构体，声明这个结

构体的指针，但并未为其分配空间，其一开始导致了我内核模块多次注册失败，如图 23。老师在 PPT 中提示推荐的使用的函数和结构中根本没有提及 mutex 这个事情，导致我的程序一开始程序全部进入死锁，如图 24。后来分析这些问题都得到了解决。

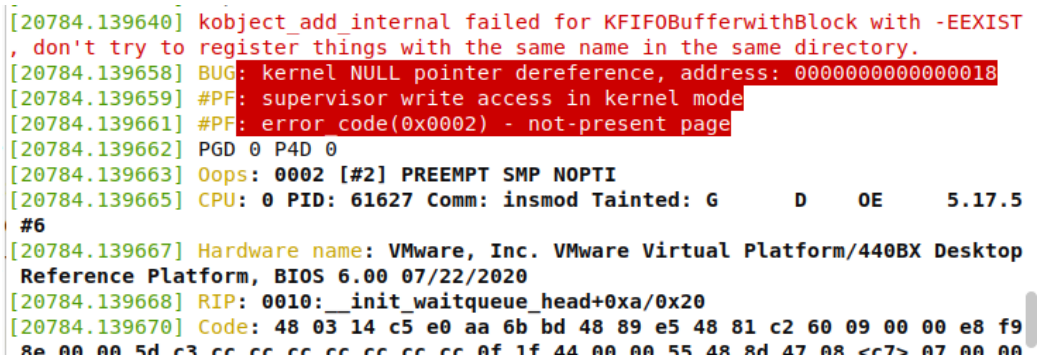

图 23：内核模块注册失败

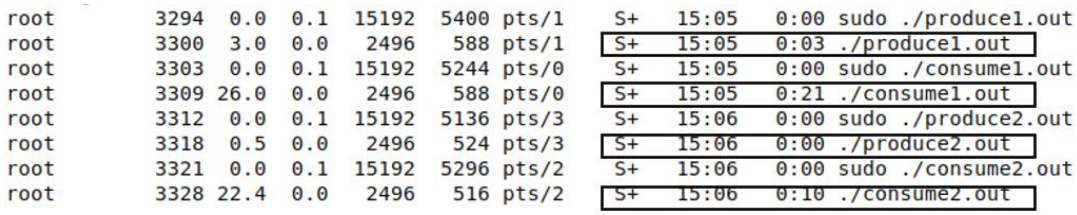

图 24：程序进入死锁的问题

2.4 运行和测试过程

我们实现了 1，2，3 三个生产者，其中 3 时以非阻塞模式打开的。也实现了 1，2，3 三个消费者，其中 3 时以非阻塞模式打开的。

当我们如图 25 成功注册设备后，打开六个终端，分别打开这六个进程，最后运行效果如图 26。当我们输出 ps -aux 时，我们看到最多只有一个进程读写缓冲区而且没有出错，其完成了我们的目标。

```
moonlight@moonlight-VMware:~/design2$ make
make -C /lib/modules/5.17.5/build M=/home/moonlight/design2 modules
make[1]: 进入目录“/home/moonlight/linux-5.17.5”
  CC [M]  /home/moonlight/design2/FIFODeviceV3.o
  MODPOST /home/moonlight/design2/Module.symvers
  CC [M]  /home/moonlight/design2/FIFODeviceV3.mod.o
  LD [M]  /home/moonlight/design2/FIFODeviceV3.ko
make[1]: 离开目录“/home/moonlight/linux-5.17.5”
moonlight@moonlight-VMware:~/design2$ sudo insmod FIFODeviceV3.
insmod: ERROR: could not load module FIFODeviceV3.: No such file or directory
moonlight@moonlight-VMware:~/design2$ sudo insmod FIFODeviceV3.ko
moonlight@moonlight-VMware:~/design2$ dmesg
[  956.576137] succeeded register char device: BlockFIFODevV3
moonlight@moonlight-VMware:~/design2$ sudo rmmod FIFODeviceV3
moonlight@moonlight-VMware:~/design2$ dmesg
[  956.576137] succeeded register char device: BlockFIFODevV3
[ 1206.575410] removing device: BlockFIFODevV3
moonlight@moonlight-VMware:~/design2$
```

图 25：成功注册设备

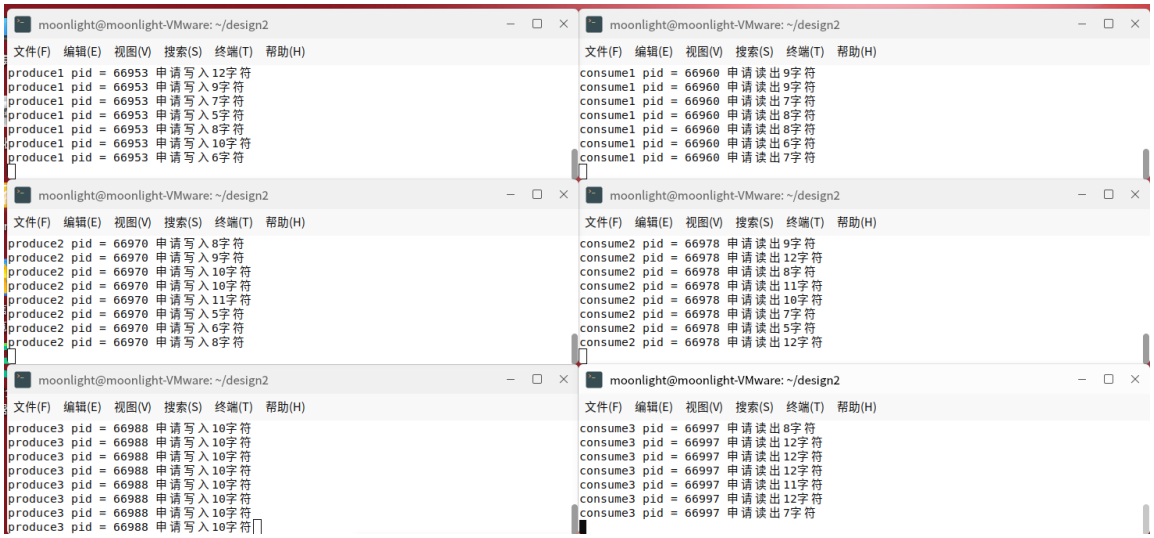

图 26：3 个消费者与生产者同时运行

```
root        5024  0.0  0.1  15192  5396 pts/2    S+   15:19   0:00 sudo ./consume1.out
root        5031 12.4  0.0   2496   588 pts/2    S+   15:19   0:18 ./consume1.out
root        5036  0.0  0.1  15192  5204 pts/3    S+   15:19   0:00 sudo ./produce1.out
root        5042 16.6  0.0   2496   588 pts/3    S+   15:19   0:21 ./produce1.out
moonlig+    5060  0.0  0.1  11412  5068 pts/1    Ss   15:20   0:00 bash
moonlig+    5093  0.0  0.1  11412  5316 pts/4    Ss   15:20   0:00 bash
moonlig+    5125  0.0  0.1  11544  5540 pts/5    Ss   15:20   0:00 bash
moonlig+    5156  0.0  0.1  11412  5284 pts/6    Ss   15:20   0:00 bash
root        5175  0.0  0.1  15192  5396 pts/1    S+   15:20   0:00 sudo ./consume2.out
root        5181 11.6  0.0   2496   588 pts/1    S+   15:20   0:11 ./consume2.out
root        5190  0.0  0.1  14908  5396 pts/4    S+   15:20   0:00 sudo ./produce2.out
root        5195 10.8  0.0   2496   528 pts/4    S+   15:20   0:08 ./produce2.out
root        5198  0.0  0.1  15192  5372 pts/5    S+   15:20   0:00 sudo ./consume3.out
root        5205 16.7  0.0   2496   588 pts/5    R+   15:20   0:11 ./consume3.out
root        5212  0.0  0.1  14908  5256 pts/6    S+   15:21   0:00 sudo ./produce3.out
root        5217 13.7  0.0   2496   516 pts/6    S+   15:21   0:07 ./produce3.out
```

图表 27：使用 ps -aux，此时仅有 consume3 在运行

2.5 实验心得和建议

简单的编写了内核缓冲区的驱动函数，这是一个开始，我们虽然是软件工程专业，但不能丢掉了汇编，C 语言，驱动程序，操作系统这些基础性的，底层性

的东西,这是中国软件事业发展的重要方向之一,也是中国软件事业的短板方向。

下学期我们仍要和 MRSU 再会一起学习微机接口一课，我们十分期待!

2.6 学习和编程实现参考网址

[1]苏曙光. 操作系统原理（慕课版）[M]. 北京：人民邮电出版社，2022.

[2]课程相关课件：感谢华中科技大学软件学院苏曙光老师

[3]https://www.cnblogs.com/big-xuyue/p/3905597.html

[4]https://www.cnblogs.com/sky-heaven/p/8257288.html

[5]https://blog.51cto.com/21cnbao/120099

[6]https://blog.csdn.net/xiaofei0859/article/details/73605593

[7]https://www.cnblogs.com/mrlayfolk/p/15857477.html

[8]https://blog.csdn.net/fangye945a/article/details/86617551

[9]https://www.cnblogs.com/big-xuyue/p/3905597.html

[10]https://www.cnblogs.com/audioZane/p/11083091.html

[11]https://zhuanlan.zhihu.com/p/304298927

[12]https://blog.csdn.net/qq_33643100/article/details/99336698

[13]https://blog.csdn.net/yafeixi/article/details/72832396

[14]https://blog.csdn.net/allen6268198/article/details/8112551

[15]https://blog.csdn.net/scarecrow_byr/article/details/28491195

[16]https://www.cnblogs.com/huangchaosong/p/7127532.html