

# 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

院 系： 网络安全学院

专业班级： 信安 1902 班

学 号： U201910210

姓 名： 徐铭睿

2022 年 03 月 19 日

## 目录

<b>本科生课程设计报告.....</b>	<b>1</b>
<b>1 实验一 任务切换机制 .....</b>	<b>3</b>
1.1 实验概述.....	3
1.2 实验设计思路.....	4
1.3 实验程序的难点或核心技术分析.....	8
1.4 运行和测试过程.....	10
1.5 实验心得和建议.....	12
1.6 学习和编程实现参考网址.....	13
<b>2 实验二 设备阻塞工作机制 .....</b>	<b>14</b>
2.1 实验概述.....	14
2.2 实验设计思路.....	15
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.....	19
2.6 学习和编程实现参考网址.....	21

# 1 实验一 任务切换机制

## 1.1 实验概述

### (1) 实验目的

- 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

### (2) 实验任务

启动保护模式，建立两个任务（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进行切换。

### (3) 实验内容

- 1) 阅读和理解 X86 保护模式初始化程序（pmtest1~pmtest5）
- 2) 阅读和理解 X86 段和页工作机制程序（pmtest6~pmtest7）
- 3) 编程实现一套页目录和页表，两个任务，并实现任务切换步骤：

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，编写两个任务

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

本实验的实现思路在 1.1 的实验内容部分，已经被较为明确描述。即编程实现 CPU 进入保护模式，初始化 GDT, LDT, IDT, TSS 等数据结构，以及建立页表与页目录。并建立两个分别使用对应页表的任务，用于在屏幕打印字符。初始化时钟模块，设定中断时间间隔；初始化中断模块，用于实现两个任务的切换，以实现在屏幕上循环打印字符串 ‘HUST’ 和 ‘MRX’。下面进行详细介绍。

### (1) 由实模式进入保护模式以及段描述符

在保护模式中，段值不再为地址的一部分，而是一个索引，这个索引指向一个数据结构表项，即 GDT。GDT 中的表项成为描述符（图 1-1）。

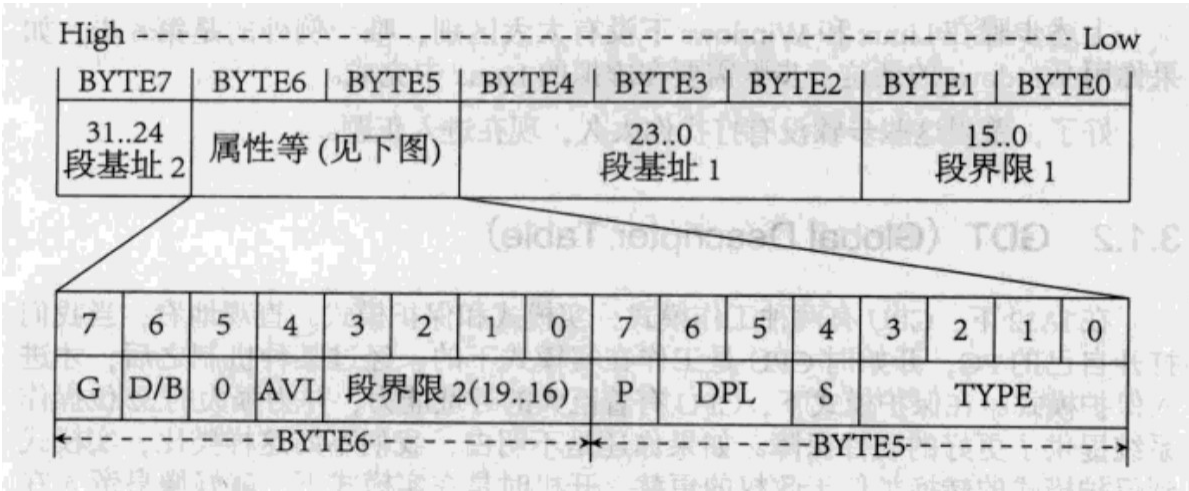
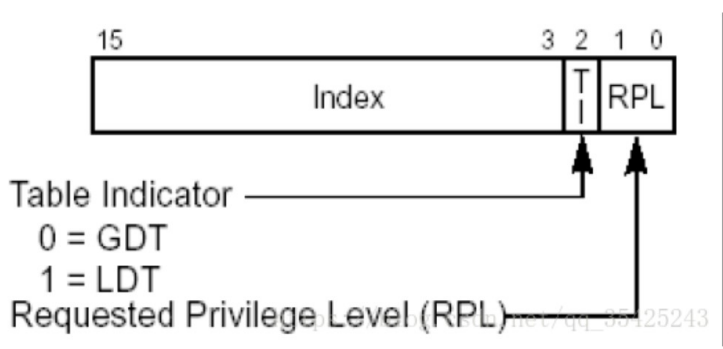


图 1-1

由上图可知，GDT 中每一个描述符定义一个段，而选择子确定描述符，描述符确定段基地址，段基地址与偏移之和就是线性地址。选择子结构如图 2-2 所示。

段选择子结构



- RPL:请求特权级别
- TI:
  - TI=0查GDT表
  - TI=1查LDT表 (Windows没有使用)

图 1-2

从实模式进入保护模式的主要步骤,即准备 GDT 及相应选择子,再使用 lgdt 指令加载 gdt, 打开 A20 地址线, 置 cr0 的 pe 位为 0, 跳转, 进入保护模式。之后, 就在保护模式下完成两任务在时钟中断控制下循环切换打印字符。

(2) 特权级转移

在本任务中, 主要用到门的调用以实现从 ring0 向 ring3 的跳转。门描述符如图 1-3 所示。

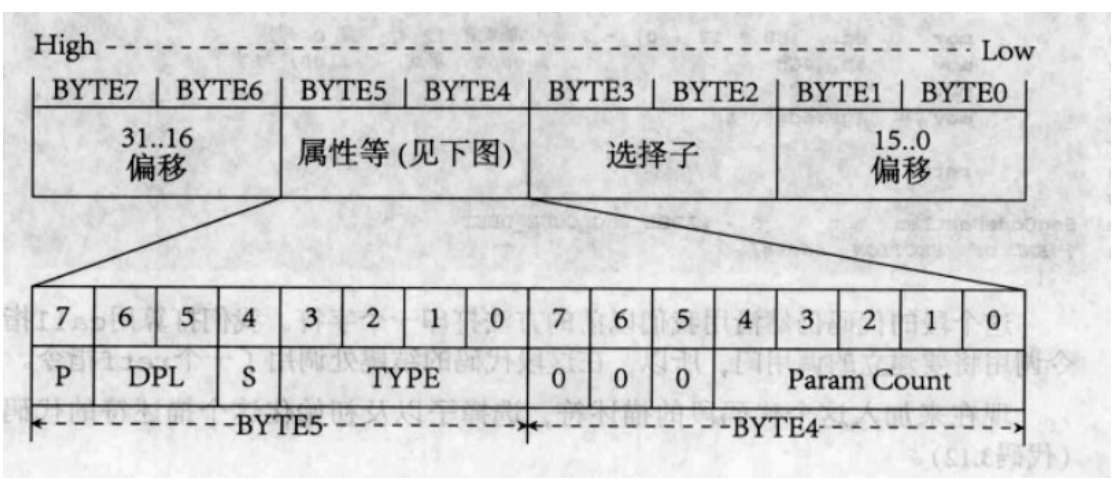


图 1-3

在中断调用的过程中, 需要注意现场的保护与恢复, 因而需要使用堆栈 (图 1-4) 与 TSS (图 1-5)。

具体调用流程为, 根据目标代码段的 DPL, 从 TSS 中选择应该切换到哪个 ss 和 esp, 从 TSS 中读取新的 ss 和 esp。如果 ss、esp 或 TSS 界限错误会导致 TSS

异常。无异常则暂时保存 ss 和 esp 的值。之后加载新的 ss 和 esp 的值，并将保存的 ss 和 esp 值压入新的堆栈。从调用者堆栈将参数复制到被调用者堆栈中，复制参数的数目由调用门中 param count 决定，并将当前的 cs 和 eip 压栈（如图 1-4）。最后加载调用门中指定的新 cs, eip, 执行被调用者过程。恢复过程不在赘述。

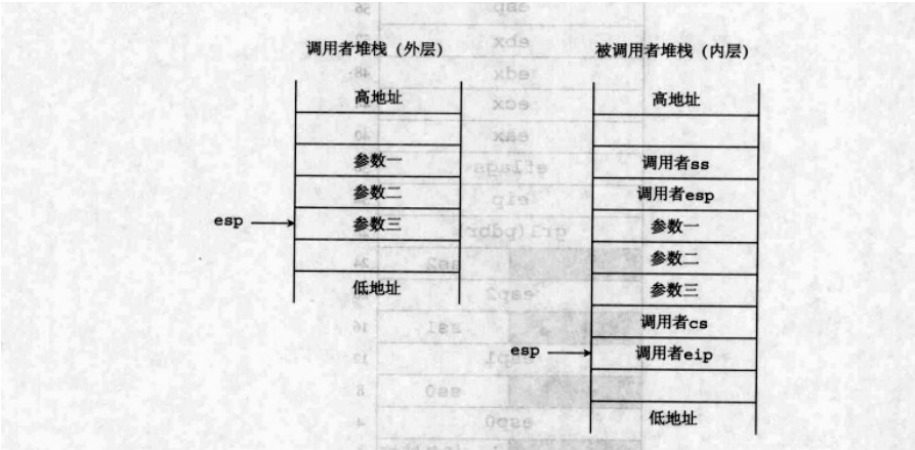


图 1-4

I/O 位图基址		T
	LDT 选择子	96
	gs	92
	fs	88
	ds	84
	ss	80
	cs	76
	es	72
	edi	68
	esi	64
	ebp	60
	esp	56
	ebx	52
	edx	48
	ecx	44
	eax	40
	eflags	36
	eip	32
	gr3 (pdbr)	28
	ss2	24
	esp2	20
	ss1	16
	esp1	12
	ss0	8
	esp0	4
	上一任务链接	0

保留位，被设为0。

图 1-5

(3) 页式存储

在本实验中，采用分页机制对内存进行管理，进行转换时，先是由寄存器 cr3 指定的页目录中根据线性地址的高 10 位得到页表地址，然后在页表中根据线性地址的第 12 到 21 位得到物理页首地址，将这个首地址加上线性地址低 12 位便得到了物理地址（图 1-6）。

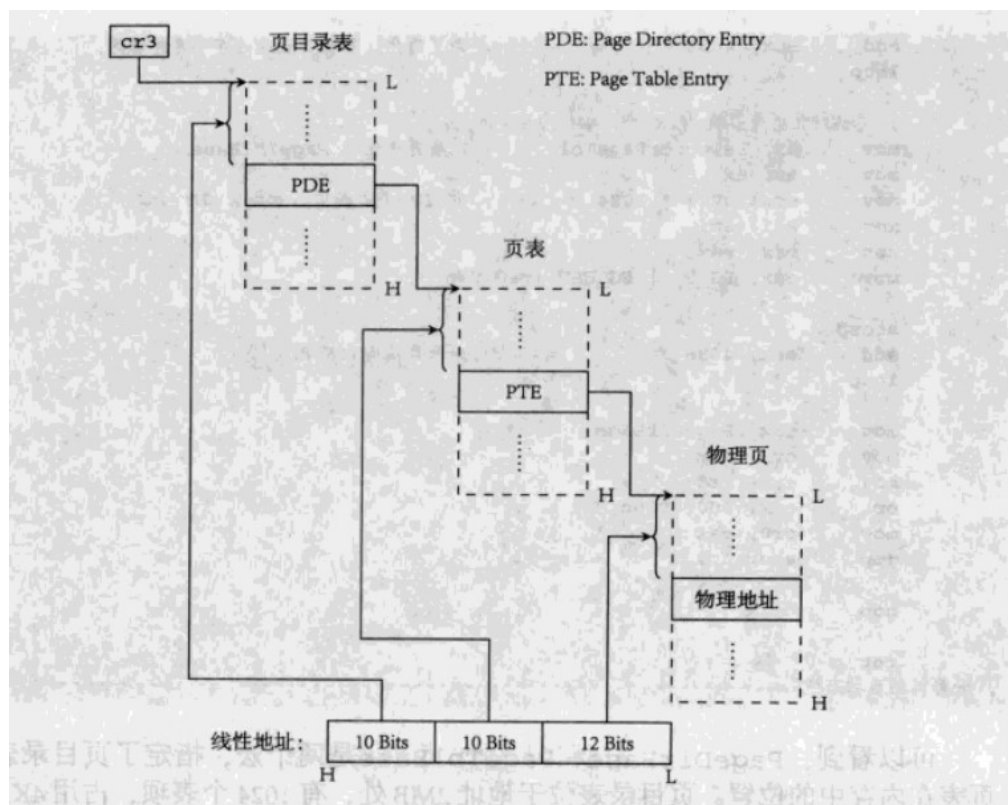


图 1-6

#### (4) 中断模块

在本实验中，使用了 8259A（图 1-7）外部中断来实现对任务切换的控制。在使用时需要对其进行初始化，主要通过向相应端口写入特定 ICW 实现，主 8259A 对应端口地址为 20h, 21h，从 8259A 对应端口地址为 a0h,a1h，向相应端口写入 ICW，即可完成对 8259A 的初始化。并指定所需中断类型。

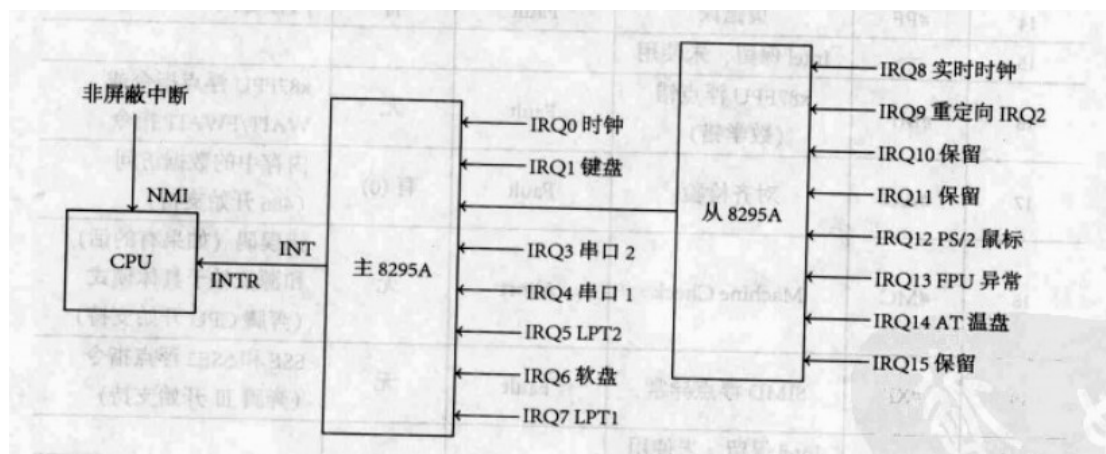


图 1-7

## 1.3 实验程序的难点或核心技术分析

### (1) mount 命令的使用

在配置好实验环境后，运行 bochs 发现 B 盘中相应的 task\_a.com 程序并未挂接成功。后发现是对挂接命令的使用方法不了解。

mount 可以将分区挂接到 Linux 的一个文件夹下，从而将分区和该目录联系起来，因此我们只要访问这个文件夹，就相当于访问该分区了。

实验中：

首先建立一个新目录作为挂接点（mount point）

```
sudo mkdir /mnt/floppyB
```

进行挂接

```
sudo mount -o loop ./pctest.img /mnt/floppyB
```

在将所需文件复制到该目录下

```
sudo cp ../task_a.com /mnt/floppyB/
```

解除挂接

```
sudo umount /mnt/floppy
```

按照上述命令操作即可解决该问题。

在实验中，由于参考程序已完成基本框架的搭建，因而程序完成过程出现问题较少。

### (2) 8253A 时钟模块的初始化

由于书上并没有关于 8253A 时钟模块初始化的介绍，因此这里着重描述一下。

与 8259A 初始化相同，需要向 8253A 的相应端口写入 ICW 进行初始化，用



00110110b 设置工作模式为技术。并根据图 1-8 计算出计数器 C 的值在中断间隔 50ms 时计数值应为 59650，将该值写入 040h 端口（如图 1-9）。

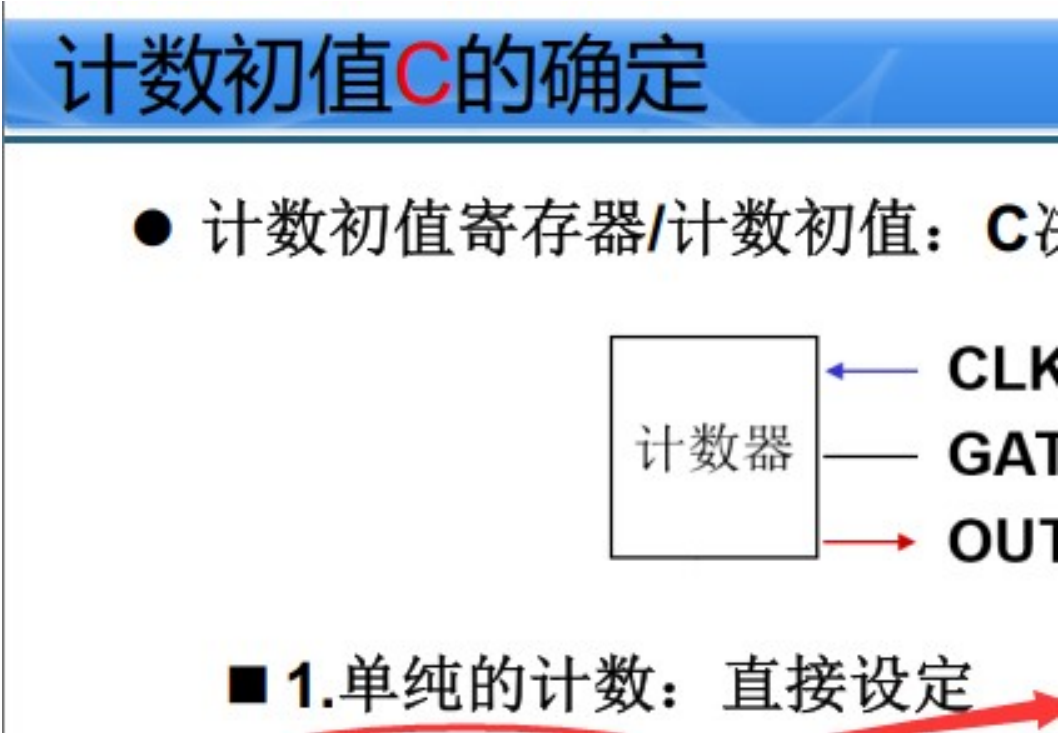


图 1-8

```
-----  
; Init8253A -----  
-----  
Init8253A:  
    mov     al, 00110110b  
    out     043h, al  
    call    io_delay  
    mov     ax, 59650  
    out     040h, al  
    call    io_delay  
    mov     al, ah  
    out     040h, al  
    call    io_delay  
  
    ret  
; Init8253A -----  
-----
```

图 1-9

(3) 两个任务间的切换

在进行两个任务间的切换时，需要注意现场的保护，否则会导致程序运行出错。代码如图 1-10 所示。

```

_ClockHandler:
ClockHandler    equ    _ClockHandler - $$
                inc     cx

                push    ds
                push    edx
                push    ecx
                push    ebx
                push    eax

                mov     ax, SelectorData
                mov     ds, ax                ; 数据段选择子
                mov     ah, 0Ch              ; 0000: 黑底    1100: 红字

                mov     bx, cx
                and     bx, 1
                mov     ax, 0
                cmp     bx, ax
                je      HUST
                jmp     MRX

```

图 1-10

## 1.4 运行和测试过程

图 1-11,1-12,为 task\_a 的 Makefile 文件，以及 bochs 的相应配置文件。

```

#####
#makefile of task a
#####
src = task_a.asm
src2 = task_a.com

.PHONY : everything clean

everything : $(src2)
    sudo mount -o loop pmtest.img /mnt/floppyB/
    sudo cp $(src2) /mnt/floppyB/ -v
    sudo umount /mnt/floppyB/

$(src2) : $(src)
    nasm $(src) -o $(src2)

bochs:
    bochs -f bochsrc.txt

clean:
    rm -rf $(src2)

```

图 1-11

```
mrx@ubuntu: ~/Desktop/os-coursedesign
megs: 32
romimage: file=/usr/local/share/bochs/BIOS-bochs-latest
vgaromimage: file=/usr/local/share/bochs/VGABIOS-lgpl-latest
floppya: 1_44=freedos.img, status=inserted
floppyb: 1_44=pmtest.img, status=inserted
boot: a
log: bochsout.txt
mouse: enabled=0
```

图 1-12

运行结果如下图所示，即交替循环打印字符‘MRX’和‘HUST’。

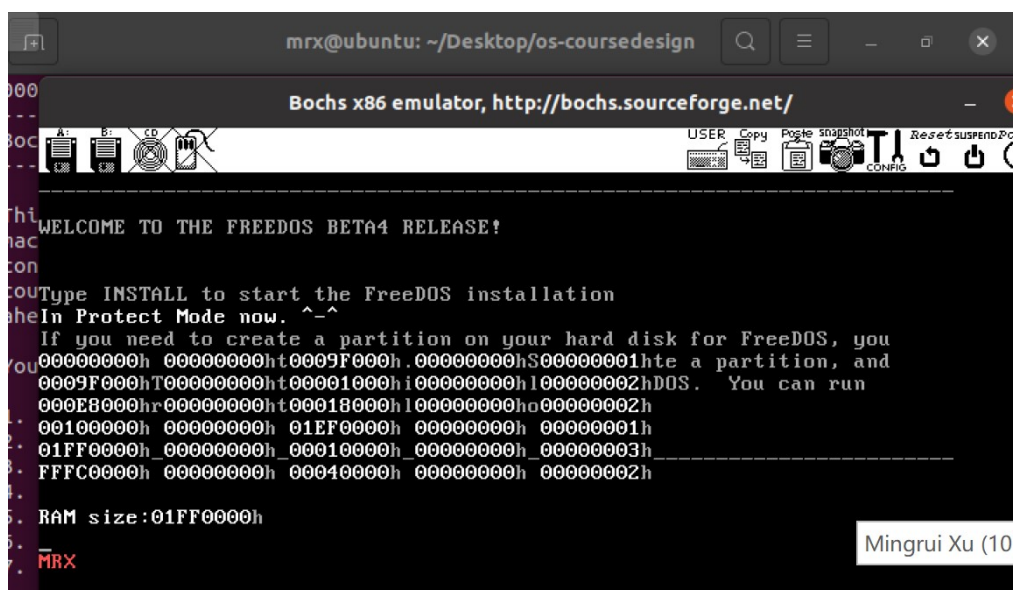


图 1-13

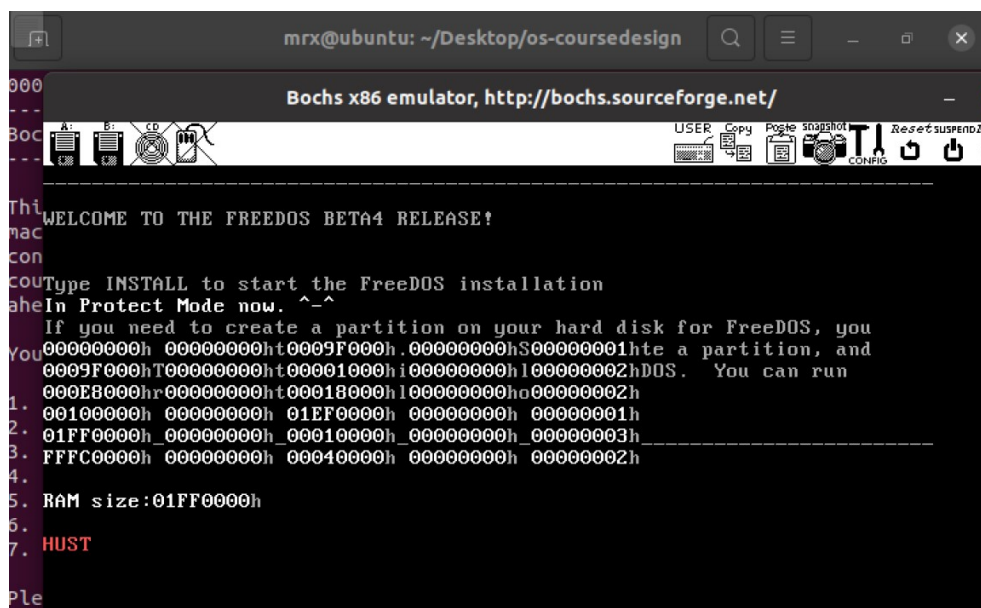


图 1-14

## 1.5 实验心得和建议

总体上看，本次实验的代码量并不是很大，因为在参考书《一个操作系统的实现》随书参考代码，已经把课设的框架搭建完成了。所以需要做的只是添加一些代码，满足相应要求。

主要添加的部分即两个新任务，初始化相应的数据结构，并新定义两个堆栈端用于两个任务跳转时的维护。在新建任务的代码段部分编写打印输出的相应内容即可。因此本实验的难点主要在对参考书上内容的理解以及实际使用。

但也由于框架已经建立好了，所以更多的关注于如何完成任务内容，在很多参考代码已经实现了的部分，就没有那么重视它的原理，只是模仿提供的代码完成功能，比如在特权级转移实现这一部分了解的不太清楚，仅仅是套用了书上的样例代码。

在实验过程中，通过对《一个操作系统的实现》的阅读以及相应代码的编写，我对操作系统理论课上学习的内容有了更深刻、更具体的感受。比如操作系统在中断调用前后进行的动作，以及现场的保护，在这次课设中都需要通过自己编写代码实现，认识更加深刻。并且通过这次课设，对操作系统在底层如何实现有了一个直观的认识。可以说，这次实验收获很多。

## 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,2009.

[2] <https://www.cnblogs.com/chengmf/p/12526821.html>

[3] <https://blog.csdn.net/jiaruitao777/article/details/103492344>

[4] [https://blog.csdn.net/weixin\\_42845306/article/details/109771924](https://blog.csdn.net/weixin_42845306/article/details/109771924)

[5] <https://max.book118.com/html/2018/0304/155683549.shtm>

## 2 实验二 设备阻塞工作机制

### 2.1 实验概述

#### (1) 实验目的

- 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）

#### (2) 实验任务

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

#### (3) 实验内容

- 1) 编写驱动程序，支持应用程序对内核缓冲区的读写
- 2) 设定内核缓冲区大小（例如 32 字节）
- 3) 缓冲区是环形缓冲区，驱动程序维护两个读写指针
- 4) 缓冲区按序读写，每个数据的读写不重复，不遗漏
- 5) 编写若干个应用程序，循环读或写缓冲区的若干字节


当缓冲区有足够的`数据`读就读，否则就阻塞进程，直到有足够数据可供读时才被唤醒；

当缓冲区有足够的`空位`写就写，否者就阻塞进程，直到有足够空位可供写时才被唤醒；

- 6) 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
- 7) 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

相较实验一本实验要简单很多，主要思路为使用 `miscdevice` 创建驱动设备，使用 `MISC_DYNAMIC_MINOR` 设置次设备号，使用老师推荐的函数以及驱动结构完成程序编写（图 2-1），并设计一个写设备程序，一个非阻塞和一个阻塞读设备程序进行测试。



- **Linux驱动程序开发**
- **Linux内核同步机制：等待队列，互斥锁，异步事件**
- **设备阻塞/非阻塞工作方式 | 阻塞/非阻塞方式打开设备**
- **推荐的函数**

```
DEFINE_KFIFO(FIFO_BUFFER, char, 64);  
wait_queue_head_t WriteQueue/ReadQueue;  
wait_event_interruptible( );  
wake_up_interruptible( );
```

图 2-1

## 2.3 实验程序的难点或核心技术分析

本实验代码量较小，并且核心结构参考资料和网上资源都比较丰富，没有特别的难点，下面进行核心技术分析。

### （1）读设备

使用 `kfifo_is_empty` 函数，判断缓冲区是否为空。如果缓冲区为空，则继续判断是阻塞还是非阻塞状态下使用设备。阻塞状态下将当前进程挂起到 `read` 等待队列中，非阻塞状态下则终止运行。

如果缓冲区非空，则程序从 `fifo` 缓冲区中读取数据复制到 `user` 缓冲区中，并传递读取的字节数。

完成读取后，使用 `kfifo_is_full` 函数，检查 `fifo` 缓冲区是否已满，如果不满，则可从 `write` 等待队列中唤醒一进程。函数返回实际读取字节数。

```

// read device
static ssize_t mydevice_read(struct file *file, char __user *buf, size_t count,
loff_t *offset){
    if(kfifo_is_empty(&FIFOBuffer)){
        if(file->f_flags & O_NONBLOCK)
            return -EAGAIN;
        ret = wait_event_interruptible(BlockDevice->read_queue, !kfifo_is_empty(&FIFOBuffer));
    }
    ret = kfifo_to_user(&FIFOBuffer, buf, count, &actual_readed);
    if(!kfifo_is_full(&FIFOBuffer)){
        wake_up_interruptible(&BlockDevice->write_queue);
    }
    return actual_readed;
}

```

图 2-2

## (2) 写设备

写函数与读函数结构十分相似，使用 `kfifo_is_full` 函数，判断缓冲区是否已满。如果缓冲区已满，阻塞状态下将当前写进程挂起，非阻塞状态则终止运行。

缓冲区未满，从 `user` 缓冲区中复制数据到 `fifo` 缓冲区中，写入完成后，判断 `fifo` 是否为空，不为空则在读队列中唤醒一个等待进程。函数返回实际写入字节数。

```

// write device
static ssize_t mydevice_write(struct file * file, const char __user *buf, size_t
count, loff_t *offset){
    if(kfifo_is_full(&FIFOBuffer)){
        if(file->f_flags & O_NONBLOCK)
            return -EAGAIN;

        ret = wait_event_interruptible(BlockDevice->write_queue, !kfifo_is_full(&FIFOBuffer));
    }
    ret = kfifo_from_user(&FIFOBuffer, buf, count, &actul_write);

    if(!kfifo_is_empty(&FIFOBuffer))
        wake_up_interruptible(&BlockDevice->read_queue);
    return actul_write;
}

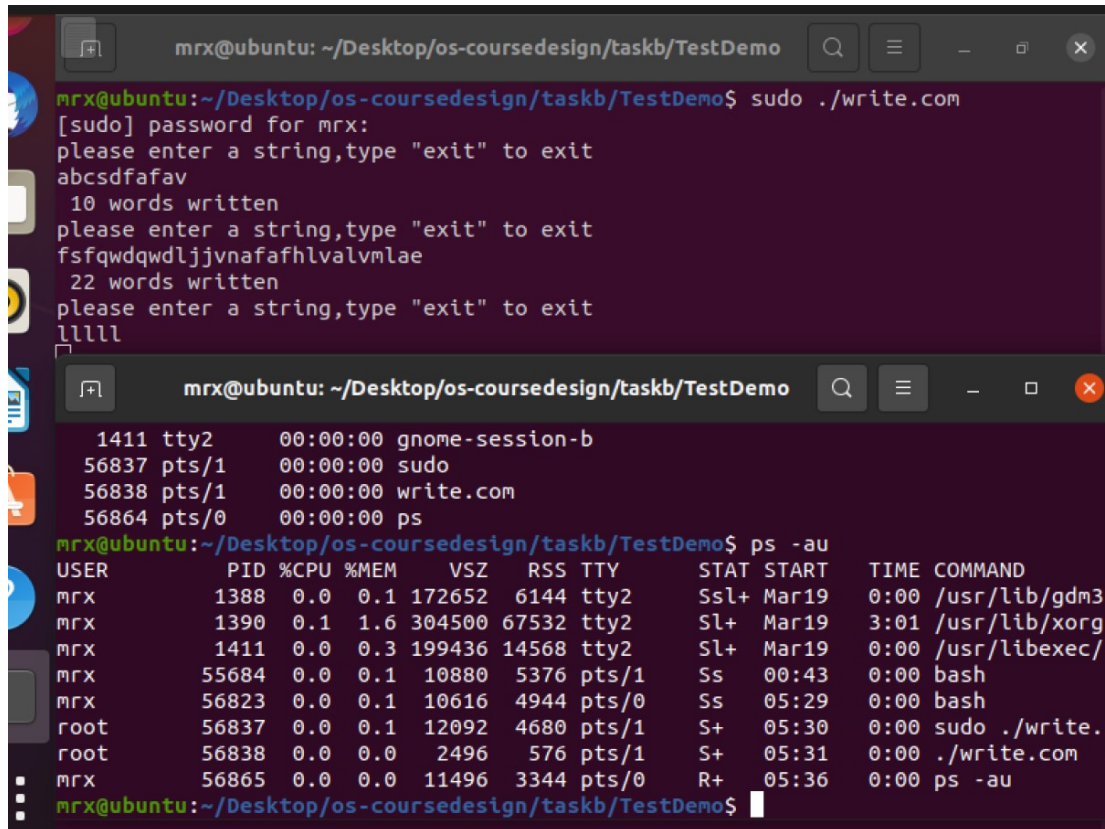
```

图 2-3

## 2.4 运行和测试过程

(1) 先将字符串写入缓冲区，再进行阻塞、非阻塞测试。可以用 `ps -au` 指令查看运行相关是否处于休眠状态。





The image shows two terminal windows from a user 'mrX' on an Ubuntu system. The top window shows the execution of a program named 'write.com' using 'sudo'. The program prompts for a password, then asks for a string to write. The user enters 'abcsdfafav', and the program reports '10 words written'. The user then enters 'fsfqwdqwdljvnaafahlvmlae', and the program reports '22 words written'. The user enters 'lllll' and the program exits. The bottom window shows the output of the 'ps -au' command, listing the current processes. The processes include 'gnome-session-b', 'sudo', 'write.com', and 'ps'. The 'write.com' process is shown as running with PID 56838.

```
mrX@ubuntu: ~/Desktop/os-coursedesign/taskb/TestDemo
[sudo] password for mrX:
please enter a string,type "exit" to exit
abcsdfafav
10 words written
please enter a string,type "exit" to exit
fsfqwdqwdljvnaafahlvmlae
22 words written
please enter a string,type "exit" to exit
lllll
mrX@ubuntu: ~/Desktop/os-coursedesign/taskb/TestDemo$ ps -au
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
mrX           1388  0.0  0.1 172652  6144 tty2    Ssl+  Mar19   0:00 /usr/lib/gdm3
mrX           1390  0.1  1.6 304500 67532 tty2    Sl+   Mar19   3:01 /usr/lib/xorg
mrX           1411  0.0  0.3 199436 14568 tty2    Sl+   Mar19   0:00 /usr/libexec/
mrX           55684  0.0  0.1 10880  5376 pts/1    Ss    00:43   0:00 bash
mrX           56823  0.0  0.1 10616  4944 pts/0    Ss    05:29   0:00 bash
root          56837  0.0  0.1 12092  4680 pts/1    S+    05:30   0:00 sudo ./write.
root          56838  0.0  0.0  2496    576 pts/1    S+    05:31   0:00 ./write.com
mrX           56865  0.0  0.0 11496  3344 pts/0    R+    05:36   0:00 ps -au
mrX@ubuntu: ~/Desktop/os-coursedesign/taskb/TestDemo$
```

图 2-4

(2) 阻塞状态下, 当没有字符时, 进程阻塞, 测试代码如图。

```
char inputBuf[32], outputBuf[32];
int main(){
    int fd, m, n;
    fd = open("/dev/mydevice", O_RDWR);
    if(fd < 0){
        printf("open /dev/mydevice failed\n");
        exit(-1);
    }
    while(1){
        sleep(1);
        m = read(fd, outputBuf, 1 * sizeof(char));
        if(m < 0 || outputBuf[0]<0){
            printf("no character to read!\n");
            continue;
        }
        else{
            printf("read char = %c \n", outputBuf[0], outputBuf[0]);
        }
    }
    close(fd);
    return 0;
}
```

图 2-5

```
mrx@ubuntu: ~/Desktop/os-coursedesign/taskb/TestDemo
mrx      56865  0.0  0.0  11496  3344 pts/0    R+   05:36   0:00 ps -au
mrx@ubuntu:~/Desktop/os-coursedesign/taskb/TestDemo$ sudo ./readBlock.com
[sudo] password for mrx:
read char = a
read char = b
read char = c
read char = s
read char = d
read char = f
read char = a
read char = f
read char = a
read char = v
read char = f
read char = s
```

图 2-6

```
mrx@ubuntu: ~/Desktop/os-coursedesign/taskb/TestDemo
read char = l
read char = j
read char = j
read char = v
read char = n
read char = a
read char = f
read char = a
read char = f
read char = h
read char = l
read char = v
read char = a

mrx@ubuntu: ~/Desktop/os-coursedesign/taskb/TestDemo
mrx      1390  0.1  1.6 307028 66908 tty2      Sl+  Mar19   3:04 /usr/lib/x
mrx      1411  0.0  0.3 199436 14568 tty2      Sl+  Mar19   0:00 /usr/libex
mrx      55684 0.0  0.1  10880  5376 pts/1    Ss   00:43   0:00 bash
mrx      56823 0.0  0.1  10748  5216 pts/0    Ss   05:29   0:00 bash
root     56837 0.0  0.1  12092  4680 pts/1    S+   05:30   0:00 sudo ./wri
root     56838 0.0  0.0   2496   576 pts/1    S+   05:31   0:00 ./write.co
root     56872 0.0  0.1  12092  4692 pts/0    S+   05:40   0:00 sudo ./rea
root     56873 0.0  0.0   2496   576 pts/0    S+   05:40   0:00 ./readBloc
mrx      56883 0.0  0.1  10616  5020 pts/2    Ss   05:41   0:00 bash
mrx      56893 0.0  0.0  11496  3268 pts/2    R+   05:42   0:00 ps -au
mrx@ubuntu:~/Desktop/os-coursedesign/taskb/TestDemo$
```

图 2-7

(3) 非阻塞状态，使用 `O_NONBLOCK` 参数打开设备，可以看到，进程在缓冲区无字符时不阻塞，继续执行，输出“no character!”。测试代码如图。

```

int main(){
    int fd, m, n;
    fd = open("/dev/mydevice", O_RDWR|O_NONBLOCK);
    if(fd < 0){
        printf("open /dev/mydevice failed\n");
        exit(-1);
    }
    while(1){
        sleep(1);
        m = read(fd, outputBuf, 1 * sizeof(char));
        if(m < 0 || outputBuf[0]<0){
            printf("no character!\n");
            continue;
        }
        else{
            printf("read char = %c \n", outputBuf[0], outputBuf[0]);
        }
    }
    close(fd);
    return 0;
}

```

图 2-8

```

Mrx@ubuntu:~/Desktop/os-coursedesign/taskb/TestDemo$ sudo ./readNonBlock.com
read -> char = e (ASCII : 101)
read -> char = x (ASCII : 120)
read -> char = i (ASCII : 105)
read -> char = t (ASCII : 116)
read -> char =   (ASCII : 32)
read -> char = i (ASCII : 105)
read -> char =   (ASCII : 32)
read -> char = l (ASCII : 108)
read -> char = o (ASCII : 111)
read -> char = v (ASCII : 118)
read -> char = e (ASCII : 101)
read -> char =   (ASCII : 32)
read -> char = y (ASCII : 121)
read -> char = o (ASCII : 111)
read -> char = u (ASCII : 117)
read -> char =   (ASCII : 32)
no character!
no character!
no character!
no character!
no character!
no character!
no character!
no character!
no character!

```

图 2-9

## 2.5 实验心得和建议

本实验相当于上学期操作系统实验课程中一个小实验的进阶版，需要设置设备的阻塞与非阻塞工作方式，并且使用了 misc 设备，即杂项设备。设备主设备号确定，无需创建设备节点，即无需进行 mknod 操作。

本实验相较实验一简单很多，也相当于对上一学期 linux 驱动设备创建的一

个复习,在实验过程中,我更加深刻的了解到阻塞方式与非阻塞方式读写的区别,以及相应的实现方法。

总的来说,还是有不少收获。

## 2.6 学习和编程实现参考网址

[1] [https://blog.csdn.net/qq\\_36172505/article/details/80372029](https://blog.csdn.net/qq_36172505/article/details/80372029)

[2] <https://www.cnblogs.com/wanghuaijun/p/7703737.html>

[3] <https://xknote.com/blog/130126.html>

[4] <https://elixir.bootlin.com/linux/v5.17-rc5/source/include/linux/kfifo.h#L30>

[5] <https://blog.csdn.net/yikai2009/article/details/8653697>