

实验四：具有中断处理的内核

实验目的

1. PC系统的中断机制和原理
2. 理解操作系统内核对异步事件的处理方法
3. 掌握中断处理编程的方法
4. 掌握内核中断处理代码组织的设计方法
5. 了解查询式I/O控制方式的编程方法

实验要求

1. 知道PC系统的中断硬件系统的原理
2. 掌握x86汇编语言对时钟中断的响应处理编程方法
3. 重写和扩展实验三的的内核程序，增加时钟中断的响应处理和键盘中断响应。
4. 编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性

实验方案

实验环境

硬件：个人计算机

操作系统：Windows 10

虚拟机软件：Bochs

实验的模式：x86_64长模式

实验开发工具

开发环境：Windows Subsystem for Linux

语言工具：x86汇编语言、C语言

编译器：gcc

汇编器: as (gcc里的汇编器)

汇编调试工具: bochsdbg

链接器: ld

反汇编工具: objdump

磁盘映像文件浏览编辑工具: WinHex

代码编辑器: Visual Studio Code

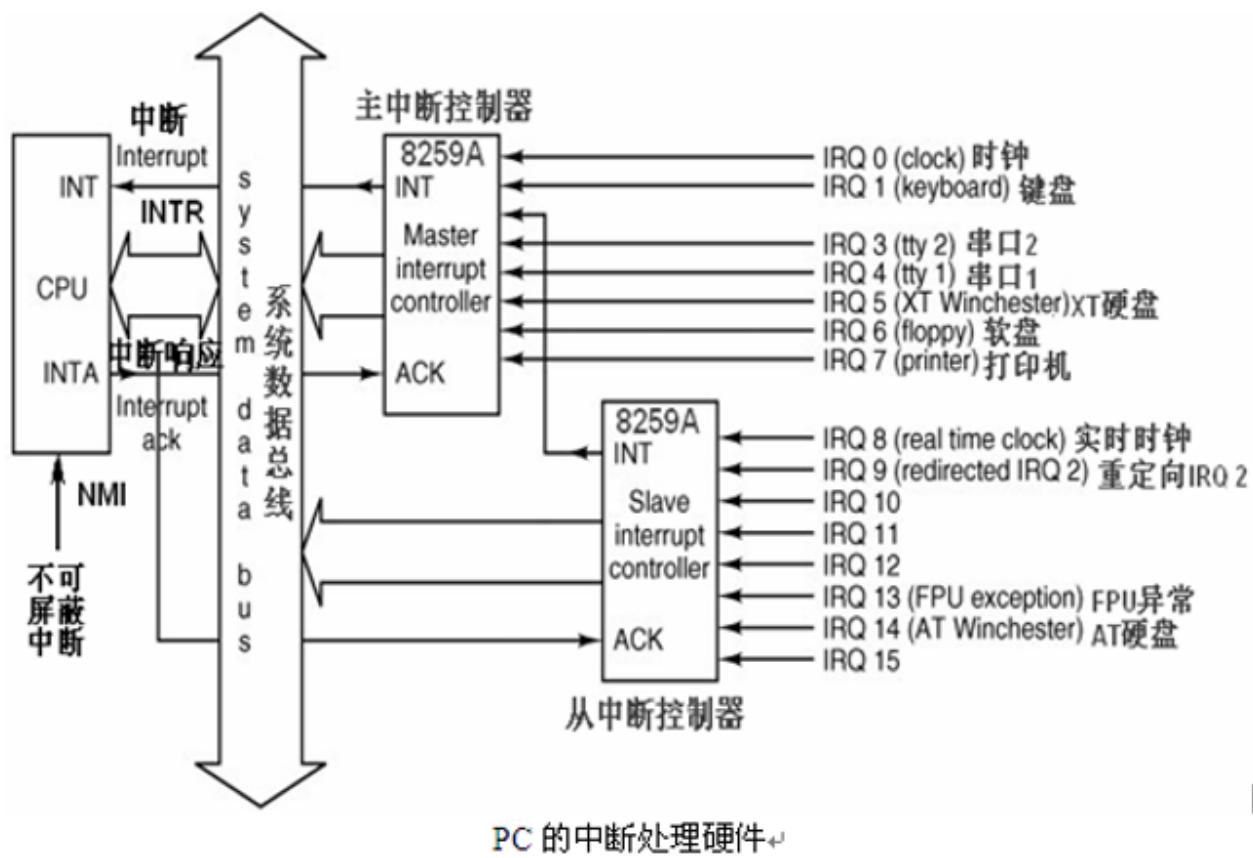
程序设计

在上一次实验中，我已经成功进入64位长模式，并完成了基础的设置。在这个实验里，我给时钟中断和键盘中断分配中断号 **0x20** 和 **0x21**

中断技术

中断是CPU实现异步事件的一种途径。

x86处理器用两个级联的 **8259A** 芯片作为外设向CPU申请中断的代理接口，使一条 **INTR** 线扩展成15条中断请求线。



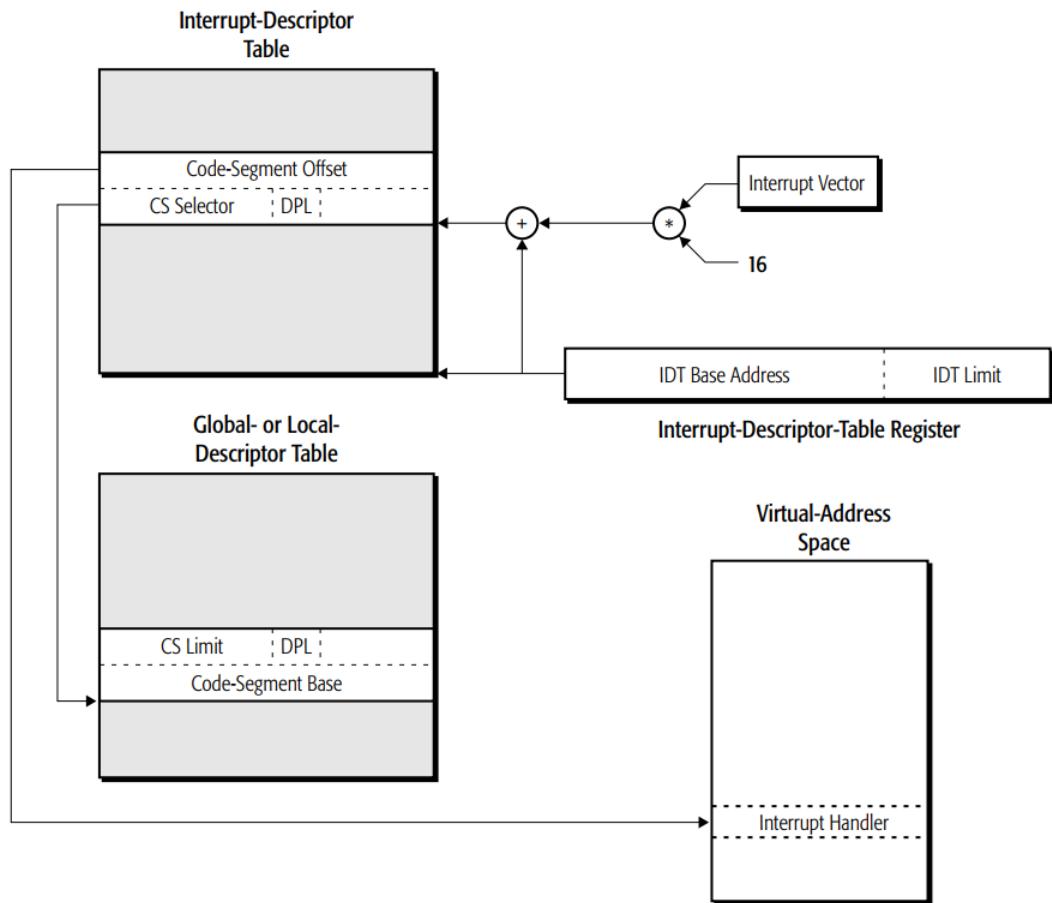
由上图可见，时钟和键盘都已经连在了 **8259A** 的IRQ0和IRQ1上。因此，要在长模式下实现时钟中断和键盘中断，要以下几个步骤：

- 建立中断向量表同时设置好idtr寄存器
- 正确初始化 **8259A** 芯片
- 将时钟和键盘的中断服务程序入口地址放到中断向量表对应位置

- 使用 `sti` 命令，置 `IF` 为1. 打开中断

长模式的中断机制

与实模式不太一样的是，长模式要实现中断需要初始化IDT（中断描述表），再利用长模式的调用方式调用中断。



上图是长模式的中断处理方式：利用中断描述符表IDTR寄存器找到中断描述符表的入口地址，在加上中断向量*16即可获得对应的中断调用描述符。

中断调用门格式如下：

31	16 15 14 13 12 11 10 9 8 7	0
Reserved, IGN	0 0 0 0 0	Reserved, IGN
Target Offset[63:32]		+12
Target Offset[31:16]	P DPL 0 Type	+8
Target Selector	Target Offset[15:0]	+4
		+0

Figure 4-23. Call-Gate Descriptor—Long Mode

Target Offset 就是中断服务例程入口地址的虚拟地址，P表示该中断描述符是否有效，**DPL** 是特权级，**Target Selector** 是段地址描述符。不过在长模式，该项仅用于判断调用方是否有相应权限调用中断。

建立中断向量表

由于长模式用的是虚拟地址，建立中断向量表有两步：

- 分配一个空闲的物理地址专门给中断向量表
- 给idtr中断向量表的入口地址

下图是我实际的物理内存地址分配情况：

0	1MB	2MB	3MB	4MB	5MB	6MB	7MB
按照原有 的，没有 更改 <i><- 级别></i>	PML-4 <i><- 级别></i>	PML-4 中分配给 内核的地 址指向的 PDT-1 <i><= 级别></i>	PML-4 中给用户 的 PDPT-2 <i><= 级别></i>	PDPT-1 指向的 PDT-1 <i><三級></i>	PDPT-2 指向的 PDT-2 <i><三級></i>	PDT-1 指向的 PT-kern <i><四級></i>	PDT-2 指向的 PT-user <i><四級></i>

实际内存地址表 - 1

接上表

8MB	9MB	10MB	11MB	12MB	
虚拟地址 0xfffff8000 0000 0000 用于存放 IDT	虚拟地址 0x0000 0000 0010 0000 用于存放	虚拟地址 0xffff ffff fff 00000 → 0xffff ffff ffffffff +栈	PT_USER_2 <四级> 用于存放 被加载 的用户 程序	虚拟地址 0x0000 0000 00C0 0000 未分配	. . .

实际内存地址表 - 2

按照这个图，可以初始化中断向量表：

```
// 把0x900000这个物理地址分配给虚拟地址0x100000
unsigned long long idt = 0x9000027;
for (int i = 0; i < 1 << 8; ++i, idt += 0x1000) {
    pt_user_1[i + 256] = idt;
}
```

再用此初始化IDTR寄存器

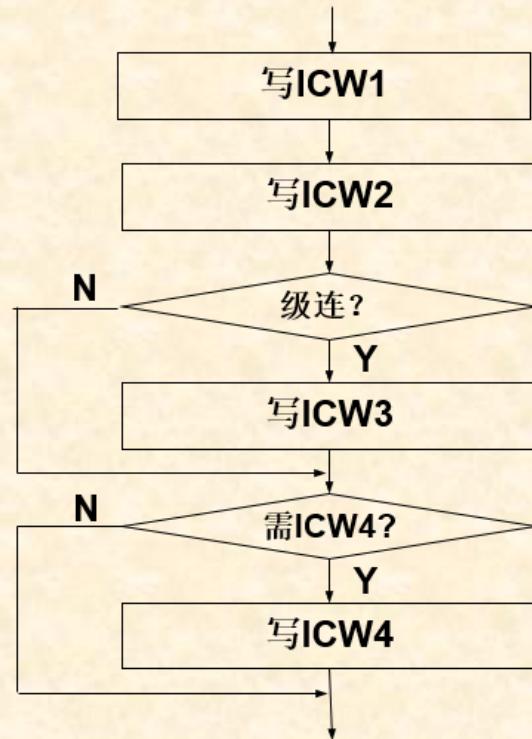
```
lidt idt64desc

idt64desc:
.word 0x8000
.quad 0x100000
```

初始化8259芯片

初始化流程图

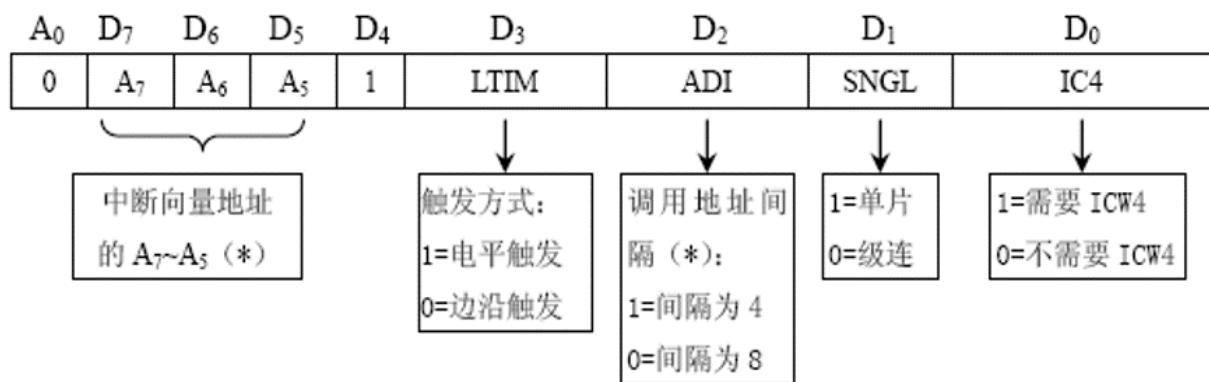
3、8259A的初始化顺序



• 8259的初始化流程图

我根据上面这个图，一步步初始化 8259A

写ICW1



注: (*) 表示只用于 MCS-80/85 系统

图 8-21 ICW1 命令字的格式

前4位都是无效的, D₃设置为边沿触发, D₁设置级连方式, D₀设置需要ICW4。

写好后分别送到主 8259A 和从 8259A , 对应端口: 0x20和0xa0

```

movb $0x11, %al      # ICW1
out %al, $0x20
out %al, $0xa0

```

写ICW2

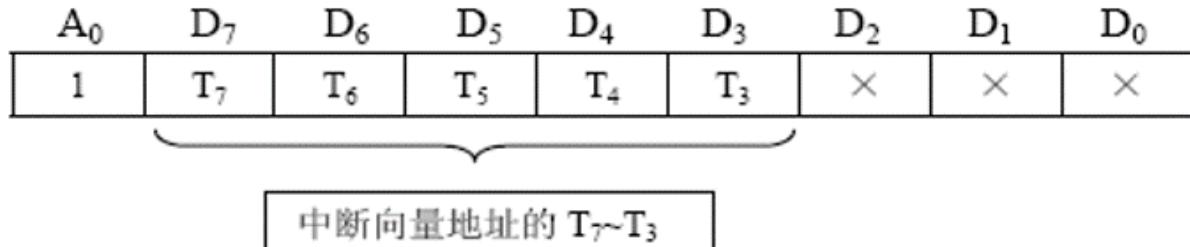


图 8-22 ICW2 命令字的格式

主8259A芯片分配0x20-0x27，从8259A芯片分配0x28-0x2f。

```

movb $0x20, %al      # master ICW2
out %al, $0x21

movb $0x28, %al      # slave ICW2
out %al, $0xa1

```

写ICW3

ICW3——级连控制字

- 主片的级联控制字
 - S_i=1 对应IR_i线上连接了从片
 - S_i=0 对应IR_i线上没连从片

A ₀	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	S ₇	S ₆	S ₅	S ₄	S ₃	S ₂	S ₁	S ₀

- 从片的级联控制字
 - ID₂~ID₀ 标识码，说明本从片连接到主片的哪个IR引脚上。000~111分别对应IR₀~IR₇。

A ₀	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	0	0	0	0	0	ID ₂	ID ₁	ID ₀

按照上面的主8259A芯片图，主8259A的IRQ2线连接了从8259A，所以初始化代码如下：

```
movb $0x04, %al      # master ICW3  
out %al, $0x21  
  
movb $0x02, %al      # slave ICW3  
out %al, $0xa1
```

写ICW4

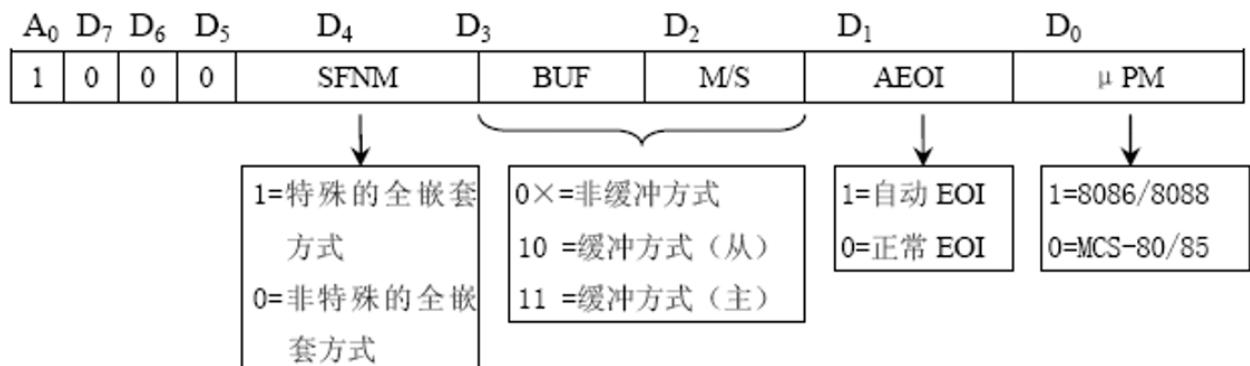


图 8-25 ICW4 命令字的格式

这里我设置的是非特殊的全嵌套方式、非缓存方式、正常EOI

```
movb $0x01, %al      # ICW4  
out %al, $0x21  
out %al, $0xa1
```

屏蔽其他中断

为了避免其它中断对系统造成不必要的影响，我还写了OCW1控制字，只打开IRQ0和IRQ1（即时钟中断和键盘中断）

OCW1——中断屏蔽字

A ₀	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	M7	M6	M5	M4	M3	M2	M1	M0

M_i=1 中断请求线IR_i被屏蔽(不允许中断)

M_i=0 允许该IR_i中断

OCW1将写入IMR寄存器。

A0=1时读OCW1可读出设置的IMR内容。

```
movb $0xfc, %al
out %al, $0x21      # 屏蔽中断

movb $0xff, %al      # 屏蔽从8259的所有中断
out %al, $0xa1
```

设置中断向量入口地址

这部分我主要是写了一个可以重复调用的设置中断向量的函数 `SetInterrupt` 函数，函数的流程：

- 关闭中断cli
- 设置好相应格式
- 打开中断sti

首先我利用中断调用门的格式，定义了一个对应的数据变量和一些常量方便设置：

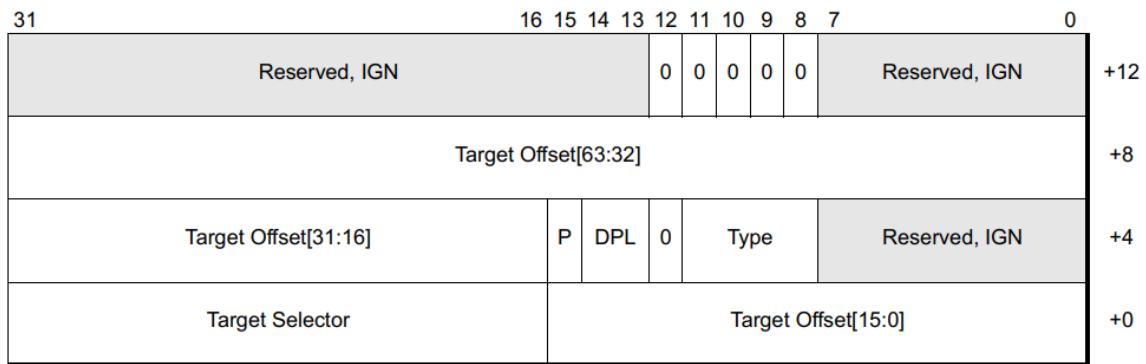


Figure 4-23. Call-Gate Descriptor—Long Mode

```
#define IDT_CS_SELECTOR 0x8;
#define IDT_x86_64_TYPE 0x0e00;
#define IDT_SYSTEM_DPL 0x0
#define IDT_PRESENT 0x8000

typedef struct InterruptGate {
    uint16_t offset_low;
    uint16_t selector;
    uint16_t attr;
    uint16_t offset_mid;
    uint32_t offset_high;
    uint32_t reserved;
} InterruptGate;
```

设置函数：

```
/* 输入设置的中断向量号及对应的中断向量例程 */
void SetInterrupt(unsigned short int_num, void *func) {
    __asm__ ("cli\n\t"); /* 关闭中断 */
    InterruptGate *idt_addr = 0x100000; /* IDT起始位置 */
    unsigned long long func_addr = (unsigned long long)func;
    idt_addr[int_num].reserved = 0x0;
    idt_addr[int_num].offset_high = func_addr >> 32;
    idt_addr[int_num].offset_mid = (func_addr >> 16) & 0xffff;
    idt_addr[int_num].offset_low = func_addr & 0xffff;
    idt_addr[int_num].selector = IDT_CS_SELECTOR;
    idt_addr[int_num].attr = IDT_PRESENT | IDT_SYSTEM_DPL | IDT_x86_64_TYPE;
    idt_addr[int_num].reserved = 0;
    __asm__ ("sti\n\t");
}
```

刚开始的关闭中断标志位，是避免在设置中断的时候被打断，这样可能会有一些潜在的bug。

后面的赋值操作就是按照格式设置调用门。

完成所有操作后，再打开中断标志位，确保中断工作正常。

风火轮中断服务程序

高级语言开始的设计就是不涉及底下对于硬件的操作，但是中断服务例程与普通函数的不同之处在于，中断服务例程的返回指令是 `iRet`。显然，用普通函数没法做到这一点，我根据gcc的编译选项，利用 `__attribute__` 的预编译指令，告诉编译器这个是一个中断函数。

根据GCC提供的格式，该中断函数还要带一个表示中断帧的结构体 `struct interrupt_frame`。

```
struct interrupt_frame{
    uint64_t rip;
    uint64_t cs;
    uint64_t rflags;
    uint64_t ss;
};

__attribute__((interrupt))
void CcHandler(struct interrupt_frame* frame) {
    char *hot_wheels_shape = "|/\\\";
    static int index = 0;
    unsigned char *vga_addr = 0xb8000;

    /* 显示风火轮 */
    vga_addr[((24 * 80) + 79) * 2] = hot_wheels_shape[index];
    vga_addr[((24 * 80) + 79) * 2 + 1] = 0x30;

    /* 更新风火轮状态 */
    index = (index + 1) % 3;

    EOI();
}

/* 时钟中断初始化 */
void InitClock() {
    SetInterrupt(CLOCK_IRQ, CcHandler);
}
```

风火轮逻辑很简单，每次调用的时候，切换一下形态，再把利用VGA显存地址显示风火轮当前形状。

下面的EOI命令，是中断服务例程用来告诉CPU该中断已经处理完毕了。

最后，再利用初始化程序，把对应例程装到中断向量0x20处。

键盘OUCH! 中断服务程序

键盘中断处理稍微有点麻烦，因为每次按键实际上可以分解成两个动作——按下键和释放键。

另外，我们可以通过读取0x60端口，获取当前的输入扫描码。

```

__attribute__ ((interrupt))
void Ouch(struct interrupt_frame* frame) {
    unsigned char scan_code = inp(0x60);
    if (scan_code < 0x80) {
        static CurPosition cur = {55, 17};
        cur.y = (cur.y + 2) % 25;
        cur.x = (cur.x - 2) % 80;
        SetCurPos(cur);
        puts("Ouch! Ouch!");
    }
    EOI();
}

```

为了程序演示效果，我们只处理按下的动作，而忽略抬起的操作。根据扫描码，按下的扫描码比抬起的扫描码恰好相差0x80，可以据此进行判断。

同时，每次按键显示位置都会发生变化，可以看的更加清楚（演示效果在下一部分和录屏中）。

键盘输入中断服务程序

在进入了长模式以后，原有的BIOS中断实际上是不能继续使用的，所以需要重新编写键盘输入程序。

恰好我们的原理课已经学到了同步问题，我利用了生产者-消费者问题的模型进行了这个程序的编写。

在我的原型操作系统里，我提供了一个库函数 `getchar`，即从键盘里一次读取一个字符，并完成回显功能。此外，我还写了一个处理键盘中断的中断服务例程 `KbHandler`。

键盘输入模型

生产者：键盘中断服务例程 `KbHandler`

消费者：输入库函数 `getchar`

信号量的变量：`output`

`semWait`：while语句

`semSignal`：while语句里的判断值发生变化

由于整个操作系统还不是很完善，我的进程阻塞就是通过一个while语句的死循环实现。同时，因为整个操作系统还没有实现进程切换，对于共享变量 `output` 访问不会产生互斥问题，因此不需要锁的实现。

```

// 键盘中断服务例程（生产者）
__attribute__ ((interrupt))
void KbHandler(struct interrupt_frame* frame) {
    unsigned char scan_code = inp(0x60);
    /* 暂时不支持大写输入 */
    if (scan_code < 0x80) {
        char ch = normalmap[scan_code];
        output = ch;
    }
}

```

```

EOI();
}

// 读取输入库函数 (消费者)
char getchar() {
    output = 0;
    // output相当于一个Condition Variable, KbHandler改变它的值之前阻塞原有的系统内核进程
    while (output <= 0) {
        /* do nothing */
    }
    putc(output); // 回显功能
    return output;
}

```

上面的程序，利用我们在原理课上对于生产者-消费者问题的学习，可以证明其正确性——即不涉及数据竞争和死锁的问题。

键盘输入功能实现

而要实现读取相应字符，需要利用从0x60端口读取键盘缓冲区里当前字符的扫描码，再通过下面的表格，找到对应的字符：

5、主键盘 scan code 表

key	mark	break	key	mark	break	key	mark	break
~/.	29	a9		0f	8f		3a	ba
!/1	02	82	q	10	90	a	1e	9e
@/2	03	83	w	11	91	s	1f	9f
#/3	04	84	e	12	12	d	20	a0
\$/4	05	85	r	13	93	f	21	a1
%/5	06	86	t	14	94	g	22	a2
~/6	07	87	y	15	95	h	23	a3
&/7	08	88	u	16	96	j	24	a4
*/8	09	89	i	17	97	k	25	a5
(/9	0a	8a	o	18	98	l	26	a6
)/0	0b	8b	p	19	99	:;	27	a7
_/-	0c	8c	{/[1a	9a	"/	28	a8
+/=	0d	8d	}/]	1b	9b		1c	9c
/\	2b	ab						
	0e	8e						

由于时间问题，我只实现了小写ascii码输入的映射，还没有实现大写的映射。

操作系统内核流程

引导程序：

- 切换模式进入保护模式
- 设置虚拟地址
- 进入长模式
- 加载系统内核到内存
- 跳转到内核主程序

内核：

- 初始化时钟中断（无敌风火轮）
- 初始化键盘中断（用于处理输入）
- 打印提示符
- 等待用户输入
- 加载用户程序
- 切换键盘的中断服务例程（OUCH!）
- 把控制权交给用户程序

用户程序：

- ...
- `ret` 把控制权交回给内核

内核：

- 切换键盘中断（由于处理输入）

除了引导程序使用汇编语言写的，剩下部分均已用C语言完成重写，并且已经在前面的实验报告中阐述过。

截至目前的操作系统项目目录树：

```
# ouyhan @ DESKTOP-3TMC71 in ~/Code Workplace/os/os_backup on git:master ✘ [23:59:12]
└── tree
    ├── Makefile
    └── boot
        ├── SetPageTable.c
        ├── boot.S
        └── puts.c
    └── img
        └── os1.img
    └── include
        ├── 1.h
        └── asm.h
    └── kern
        ├── driver.c
        ├── main.S
        └── shell.c
    └── user_program
        ├── user1.c
        ├── user2.c
        ├── user3.c
        └── user4.c

5 directories, 14 files
```

Makefile

这次实验中，我利用Makefile进行交叉构建实现全自动化构建项目，具体流程：

- 用gcc编译源文件，生成相应的.o文件
- 用ld进行链接，分别生成引导部分boot.bin、内核部分kernal.bin、4个用户程序的bin文件这些可执行文件
- 用dd命令将这些二进制文件全部复制到一个磁盘映像文件里
- 使用bochs运行和调试

本makefile的功能如下

- 编译并链接各部分文件
- 一键调用objdump反汇编二进制文件，方便调试
- 一键生成对应的磁盘映像文件
- 一键运行

具体效果在下一部分进行演示。

makefile的源码：

```
CC = gcc
LD = ld
OBJDUMP = objdump
CFLAGS = -ffreestanding -I ./include -mgeneral-regs-only
LINKFLAGS = --oformat binary
BOOTFLAGS = -Ttext 0x7c00
KERNELFLAGS = -Ttext 0xffff800000000000 -Tdata 0xffff80000001000
```

```

USERFLAGS = -Ttext 0x400000 -Tdata 0x4005b0

output = img/os1.img
boot_obj = boot.o puts.o
kernal_obj = main.o shell.o driver.o
user_obj = user1.o user2.o user3.o user4.o
user_bin = user1.bin user2.bin user3.bin user4.bin

# all the file object
obj = boot
obj += kernal
obj += user

# file offset
boot_offset = 0
kernal_offset = 5
user1_offset = 16
user2_offset = 19
user3_offset = 22
user4_offset = 25

# File Search
vpath %.h ./include
vpath %.c ./boot../../kern../../user_program
vpath %.S ./boot../../kern
vpath %.img ./img

run:all
    bochs -f bochsrc.bxrc

# make disk file
all:${obj}
    dd if=boot.bin of=$(output) bs=512 seek=$(boot_offset) conv=notrunc; \
    dd if=kernal.bin of=$(output) bs=512 seek=$(kernal_offset) conv=notrunc; \
    dd if=user1.bin of=$(output) bs=512 seek=$(user1_offset) conv=notrunc; \
    dd if=user2.bin of=$(output) bs=512 seek=$(user2_offset) conv=notrunc; \
    dd if=user3.bin of=$(output) bs=512 seek=$(user3_offset) conv=notrunc; \
    dd if=user4.bin of=$(output) bs=512 seek=$(user4_offset) conv=notrunc; \

backup:${obj}
    cp -r ./ ../os_backup; \
    cd ../os_backup; \
    rm -f */*.o; \
    rm -f *.o; \
    rm -f *.bin; \
    rm -f */*.out; \

# Link boot program
boot:${boot_obj}
    $(LD) $(LINKFLAGS) $(BOOTFLAGS) $(boot_obj) -o boot.bin

# Link kernal program

```

```

kernal:$(kernal_obj)
    $(LD) $(LINKFLAGS) $(KERNELFLAGS) $(kernal_obj) -o kernal.bin

# Link user program
user:$(user_bin)

# Debug
kernal_debug:$(kernal_obj)
    $(LD) $(KERNELFLAGS) $(kernel_obj) -o kernel_debug.bin; \
    $(OBJDUMP) -d kernel_debug.bin

$(user_bin):%.bin:%.o
    $(LD) $(LINKFLAGS) $(USERFLAGS) $< -o $@

# Dependency
driver.o: driver.c 1.h
main.o: main.S 1.h
boot.o: boot.S asm.h

# ASM program compile rule
%.o: %.S
    $(CC) -c $(CFLAGS) $< -o $@

# C program compile rule
%.o: %.c
    $(CC) -c $(CFLAGS) $< -o $@

# Clean all the obj file
.PHONY:clean
clean:
    -rm -f */*.o
    -rm -f *.o
    -rm -f *.bin
    -rm -f */*.out

```

实验过程和结果

(1) dos下实现时钟中断

为了观察效果，我修改了实验一里用的反弹程序，为这个程序加上了时钟中断。整个程序的效果就是一边反弹球，右下一边无线风火轮。

整个程序的流程：

- 段、栈的初始化
- 保存旧的中断向量
- 设置时钟中断向量

- 开始反弹程序
- 设置回原来的时钟中断向量
- int 21h, 把控制权交回给dos系统

对应代码的编写

在dos系统里，系统时钟中断号为08h，所以中断向量地址就在20h（存放IP值）和22h（存放CS的值）。因此，我们先读取20h和22h的内容，并做压栈处理：

```

xor ax,ax           ; AX = 0
mov es,ax
mov ax,word[es:20h]
push ax             ; 压入IP值
mov ax,word[es:22h]
push ax             ; 压入cs值

```

此后，装载我们的时钟中断地址。

```

mov word[es:20h],Timer
mov ax,cs
mov word [es:22h],ax

```

而时钟中断服务例程的主代码：

- 先保存之后用到的寄存器的值，防止影响程序的运行
- 变化时钟计数器的值，确定此时风火轮的状态
- 向B8000h写入内容，完成显示

```

; 时钟中断例程
Timer:
    push ax           ; 压栈保护
    push bx           ; 压栈保护
    dec byte[tcount] ; 变化计数器的值实现风火轮状态的变化
    jnz tend

; 重置计数变量=初值tdelay
    mov byte[tcount],tdelay
; bx = ti
    mov bl,[ti]
    mov bh,0
; 风火轮进行变化后，显示最新的状态
    mov al,[tchar + bx]
    mov byte[gs:((80*24+79)*2)],al
    mov al, 0x30
    mov byte[gs:((80*24+79)*2+1)],al
    dec byte[ti]
    jnz tend
; 重置风火轮偏移量
Resetti:
    mov al, 3

```

```
    mov byte[ti],al
; 中断的结束, 发送EOI, 同时恢复寄存器的值
tend:
    mov al,0x20
    out 0x20,al          ; 发送EOI到主8259A
    out 0xa0,al          ; 发送EOI到从8259A
    pop bx
    pop ax
    iret
```

程序结束时, 恢复原来的时钟中断向量, 并把控制权还给dos系统

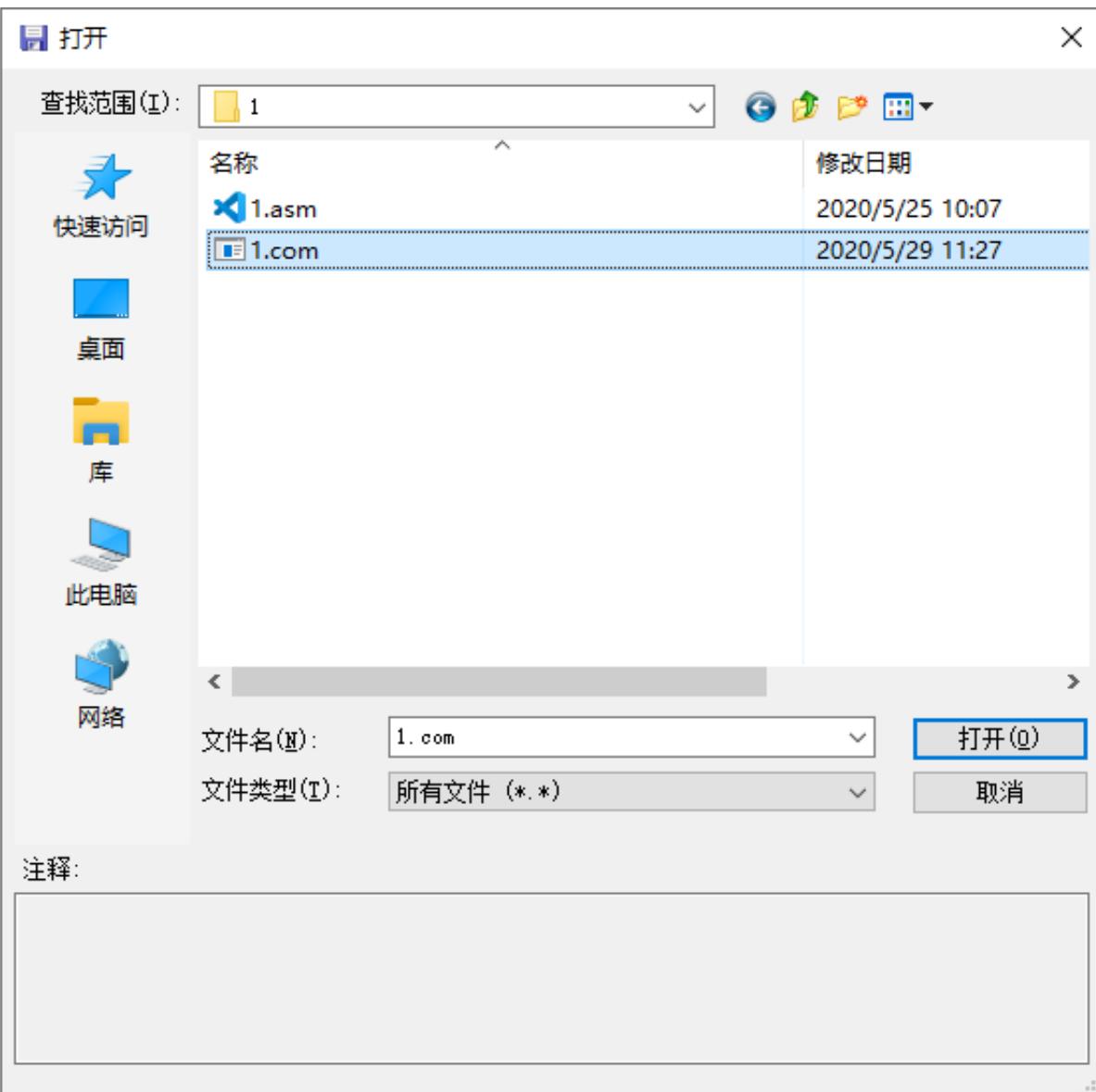
```
end:
; 恢复原有时钟中断
pop ax
mov word[es:22h],ax
pop ax
mov word[es:20h],ax
; 把控制权还给dos系统
mov ax, 4c00h
int 21h
```

实验结果

代码的编译:

```
PS D:\Files\Os\Lab4\1> nasm -f bin 1.asm -o 1.com
```

利用winimage软件把1.com用户程序装入dos的磁盘映像文件



WinImage - D:\Files\Os\MSDOS\MSDOS_1.img

文件(F) 映像(I) 磁盘(D) 选项(O) 帮助(H)

卷标: STARTUP

	名称	大小	类型	修改时间
	1.com	547	MS-DOS 应用程序	2020/5/25 10:07:32
	attrib.exe	11,208	应用程序	1994/5/31 6:22:00
	chkdsk.exe	12,241	应用程序	1994/5/31 6:22:00
	command.com	54,645	MS-DOS 应用程序	1994/5/31 6:22:00
	debug.exe	15,718	应用程序	1994/5/31 6:22:00
	drvspace.bin	66,294	BIN 文件	1994/5/31 6:22:00
	drvspace.exe	181,840	应用程序	1994/5/31 6:22:00
	edit.com	413	MS-DOS 应用程序	1994/5/31 6:22:00
	expand.exe	16,129	应用程序	1994/5/31 6:22:00
	fdisk.exe	29,336	应用程序	1994/5/31 6:22:00
	format.com	22,974	MS-DOS 应用程序	1994/5/31 6:22:00
	io.sys	40,774	系统文件	1994/5/31 6:22:00
	mem.exe	32,502	应用程序	1994/5/31 6:22:00
	msd.exe	165,864	应用程序	1994/5/31 6:22:00
	msdos.sys	38,138	系统文件	1994/5/31 6:22:00
	qbasic.exe	194,309	应用程序	1994/5/31 6:22:00
	scandisk.exe	124,262	应用程序	1994/5/31 6:22:00
	sys.com	9,432	MS-DOS 应用程序	1994/5/31 6:22:00
	undelete.exe	26,416	应用程序	1994/5/31 6:22:00
	xcopy.exe	16,930	应用程序	1994/5/31 6:22:00

1440 KB, 393,216 字节可用 | 20 个文件 (1,059,972 字节) | \

使用virtual box进行运行（完整过程详见视频文件）

MSDOS [正在运行] - Oracle VM VirtualBox

管理 控制 视图 热键 设备 帮助

你已打开了 **自动独占键盘** 的选项。现在当该虚拟电脑窗口处于活动状态时就将 **完全独占** 键盘，这时处于该虚拟电脑外的其它程序将无  

Starting MS-DOS...

Current date is Fri 05-29-2020

Enter new date (mm-dd-yy):

Current time is 11:30:56.88a

Enter new time:

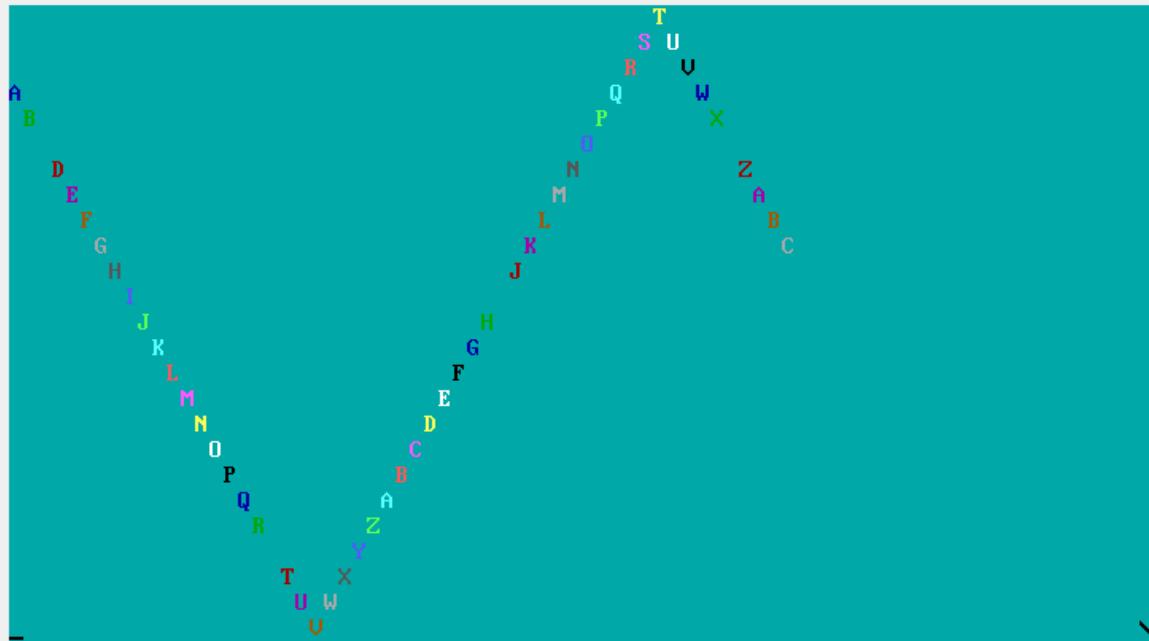
Microsoft(R) MS-DOS(R) Version 6.22
(C)Copyright Microsoft Corp 1981-1994.

A:\>1.com

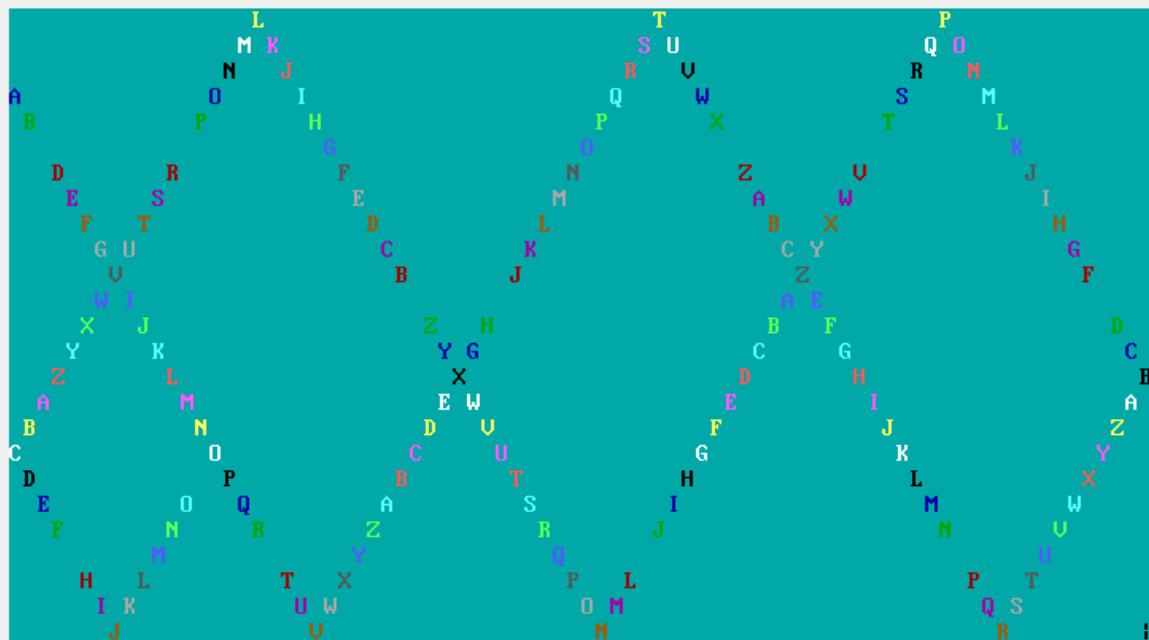
         Right Ctrl ...

管理 控制 视图 热键 设备 帮助

你已打开了 **自动独占键盘** 的选项。现在当该虚拟电脑窗口处于活动状态时就将 **完全独占** 键盘，这时处于该虚拟电脑外的其它程序将无  

 Right Ctrl

你已打开了 **自动独占键盘** 的选项。现在当该虚拟电脑窗口处于活动状态时就将 **完全独占** 键盘，这时处于该虚拟电脑外的其它程序将无  



看右下角可以知道，在程序运行过程中，风火轮的状态是在变化的。

(2) 重写和扩展实验三的内核程序

这部分主要是增加了系统内核键盘输入和无敌风火轮（时钟中断）的功能。

代码的编写

这里只需要修改系统内核的代码即可。

我在内核主程序入口部分增加了两条语句

```
call InitClock  
call InitKeyboard
```

相应的初始化部分

```
void InitKeyboard() {  
    SetInterrupt(KEYBOARD_IRQ, KbHandler);  
}  
  
/* 时钟中断初始化 */  
void InitClock() {  
    SetInterrupt(CLOCK_IRQ, CcHandler);  
}
```

代码的编译和链接

由于已经写好了Makefile，这里可以直接进行编译和链接

```
# ouyhan @ DESKTOP-3TMC71 in ~/Code Workplace/os/Lab4 on git:master x [13:40:00]  
$ make all  
gcc -c -ffreestanding -I ./include -mgeneral-reg-only ./boot/boot.S -o boot.o  
.boot/boot.S: Assembler messages:  
.boot/boot.S:152: Warning: indirect jmp without '*'  
gcc -c -ffreestanding -I ./include -mgeneral-reg-only ./boot/puts.c -o puts.o  
.boot/puts.c: In function 'putc':  
.boot/puts.c:63:31: warning: initialization of 'unsigned char *' from 'int' makes pointer from integer without a cast [-Wint-conversion]  
63 |     unsigned char *vga_addr = 0xb8000;  
|~~~~~  
ld --ofORMAT binary -Ttext 0x7c00 boot.o puts.o -o boot.bin  
gcc -c -ffreestanding -I ./include -mgeneral-reg-only ./kern/main.S -o main.o  
gcc -c -ffreestanding -I ./include -mgeneral-reg-only ./kern/shell.c -o shell.o  
.kern/shell.c: In function 'LoadUserProgram':  
.kern/shell.c:28:28: warning: initialization of 'void *' from 'int' makes pointer from integer without a cast [-Wint-conversion]  
28 |     void *user_code_addr = 0x400000;  
|~~~~~  
.kern/shell.c: In function 'Welcoming':  
.kern/shell.c:43:31: warning: initialization of 'unsigned char *' from 'int' makes pointer from integer without a cast [-Wint-conversion]  
43 |     unsigned char *vga_addr = 0xb8000;  
|~~~~~  
.kern/shell.c: Assembler messages:  
.kern/shell.c:30: Warning: indirect call without '*'  
gcc -c -ffreestanding -I ./include -mgeneral-reg-only ./kern/driver.c -o driver.o  
.kern/driver.c: In function 'SetInterrupt':  
.kern/driver.c:70:31: warning: initialization of 'InterruptGate *' {aka 'struct InterruptGate *} from 'int' makes pointer from integer without a cast [-Wint-conversion]  
70 |     InterruptGate *idt_addr = 0x100000;  
|~~~~~  
.kern/driver.c: In function 'CcHandler':  
.kern/driver.c:179:31: warning: initialization of 'unsigned char *' from 'int' makes pointer from integer without a cast [-Wint-conversion]  
179 |     unsigned char *vga_addr = 0xb8000;  
|~~~~~  
.kern/driver.c: In function 'ScrollUp':  
.kern/driver.c:222:32: warning: initialization of 'short unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]  
222 |     unsigned short *vga_addr = 0xb8000; /* 两个字节为基本单位 */  
|~~~~~  
.kern/driver.c: In function 'putc':  
.kern/driver.c:235:31: warning: initialization of 'unsigned char *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
```

```

|
~~~~~
ld --format binary -Ttext 0xfffff80000000000 -Tdata 0xfffff800000001000 main.o shell.o driver.o -o kernal.bin
gcc -c -ffreestanding -I ./include -mgeneral-regs-only ./user_program/user1.c -o user1.o
./user_program/user1.c:12:18: warning: initialization of 'char *' from 'int' makes pointer from integer without a cast
-Wint-conversion]
12 | char *vga_addr = 0xb8000;
| ~~~~~
ld --format binary -Ttext 0x400000 -Tdata 0x4005b0 user1.o -o user1.bin
gcc -c -ffreestanding -I ./include -mgeneral-regs-only ./user_program/user2.c -o user2.o
./user_program/user2.c:12:18: warning: initialization of 'char *' from 'int' makes pointer from integer without a cast
-Wint-conversion]
12 | char *vga_addr = 0xb8000;
| ~~~~~
ld --format binary -Ttext 0x400000 -Tdata 0x4005b0 user2.o -o user2.bin
gcc -c -ffreestanding -I ./include -mgeneral-regs-only ./user_program/user3.c -o user3.o
./user_program/user3.c:12:18: warning: initialization of 'char *' from 'int' makes pointer from integer without a cast
-Wint-conversion]
12 | char *vga_addr = 0xb8000;
| ~~~~~
ld --format binary -Ttext 0x400000 -Tdata 0x4005b0 user3.o -o user3.bin
gcc -c -ffreestanding -I ./include -mgeneral-regs-only ./user_program/user4.c -o user4.o
./user_program/user4.c:12:18: warning: initialization of 'char *' from 'int' makes pointer from integer without a cast
-Wint-conversion]
12 | char *vga_addr = 0xb8000;
| ~~~~~
ld --format binary -Ttext 0x400000 -Tdata 0x4005b0 user4.o -o user4.bin
dd if=boot.bin of=img/os1.img bs=512 seek=0 conv=notrunc; \
dd if=kernal.bin of=img/os1.img bs=512 seek=5 conv=notrunc; \
dd if=user1.bin of=img/os1.img bs=512 seek=16 conv=notrunc; \
dd if=user2.bin of=img/os1.img bs=512 seek=19 conv=notrunc; \
dd if=user3.bin of=img/os1.img bs=512 seek=22 conv=notrunc; \
dd if=user4.bin of=img/os1.img bs=512 seek=25 conv=notrunc; \
4+1 records in
4+1 records out
2336 bytes (2.3 kB, 2.3 KiB) copied, 0.002685 s, 870 kB/s
9+1 records in
9+1 records out
4616 bytes (4.6 kB, 4.5 KiB) copied, 0.0025554 s, 1.8 MB/s
2+1 records in
2+1 records out

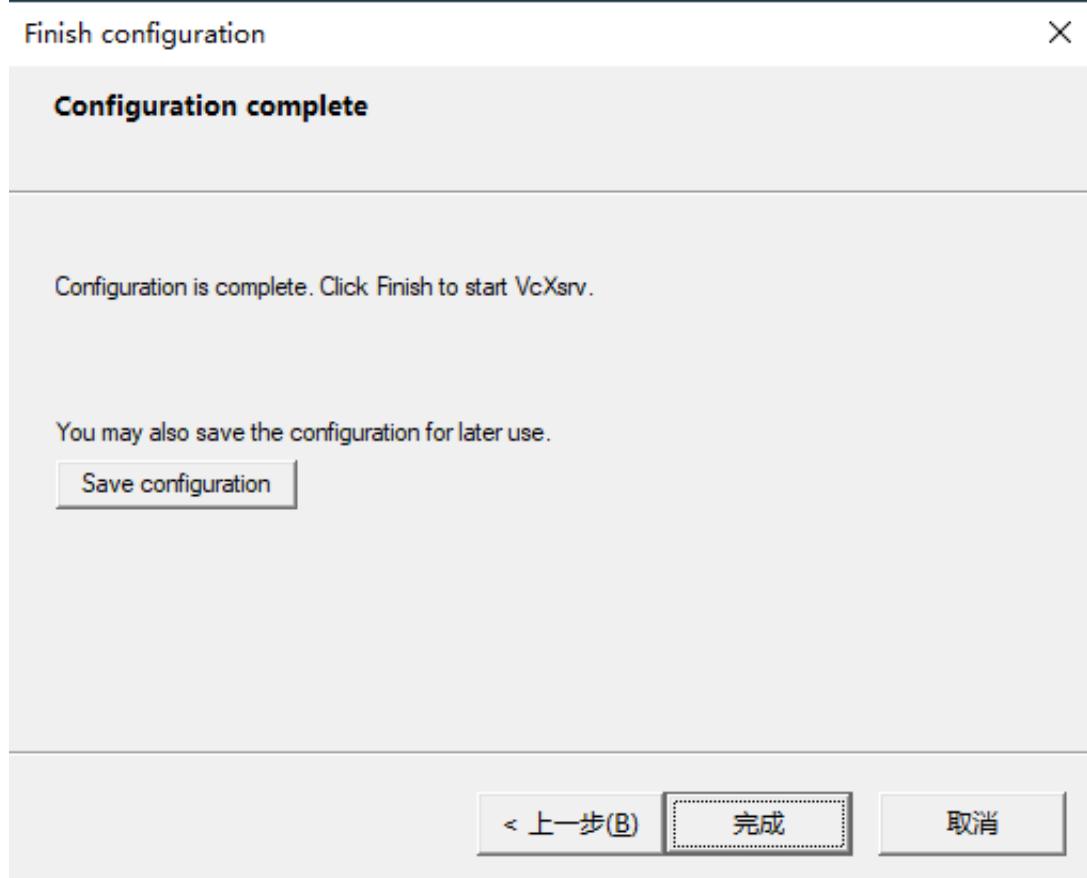
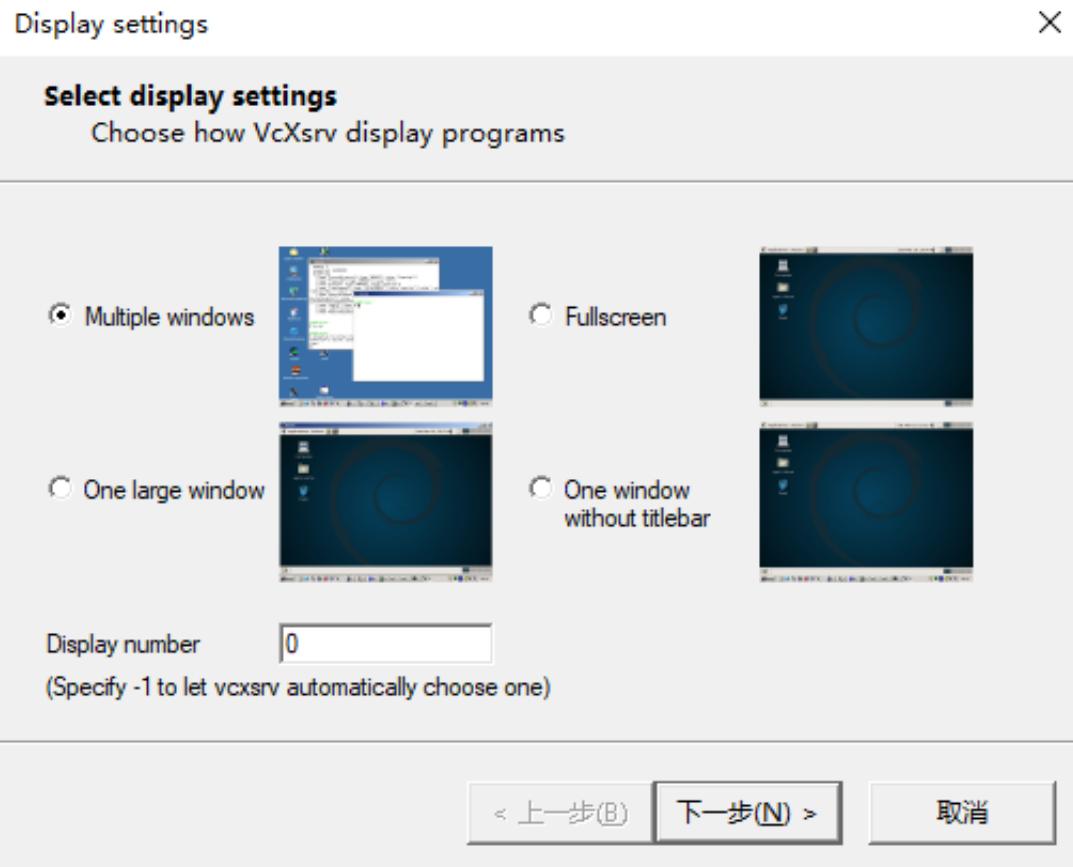
```

这样直接就已经生成好相应的磁盘映像文件了。

代码的运行

由于wsl没有GUI，要想看到运行结果，必须配置X server进行GUI的输出。这里我利用一个叫XLaunch的软件。

初步配置一下



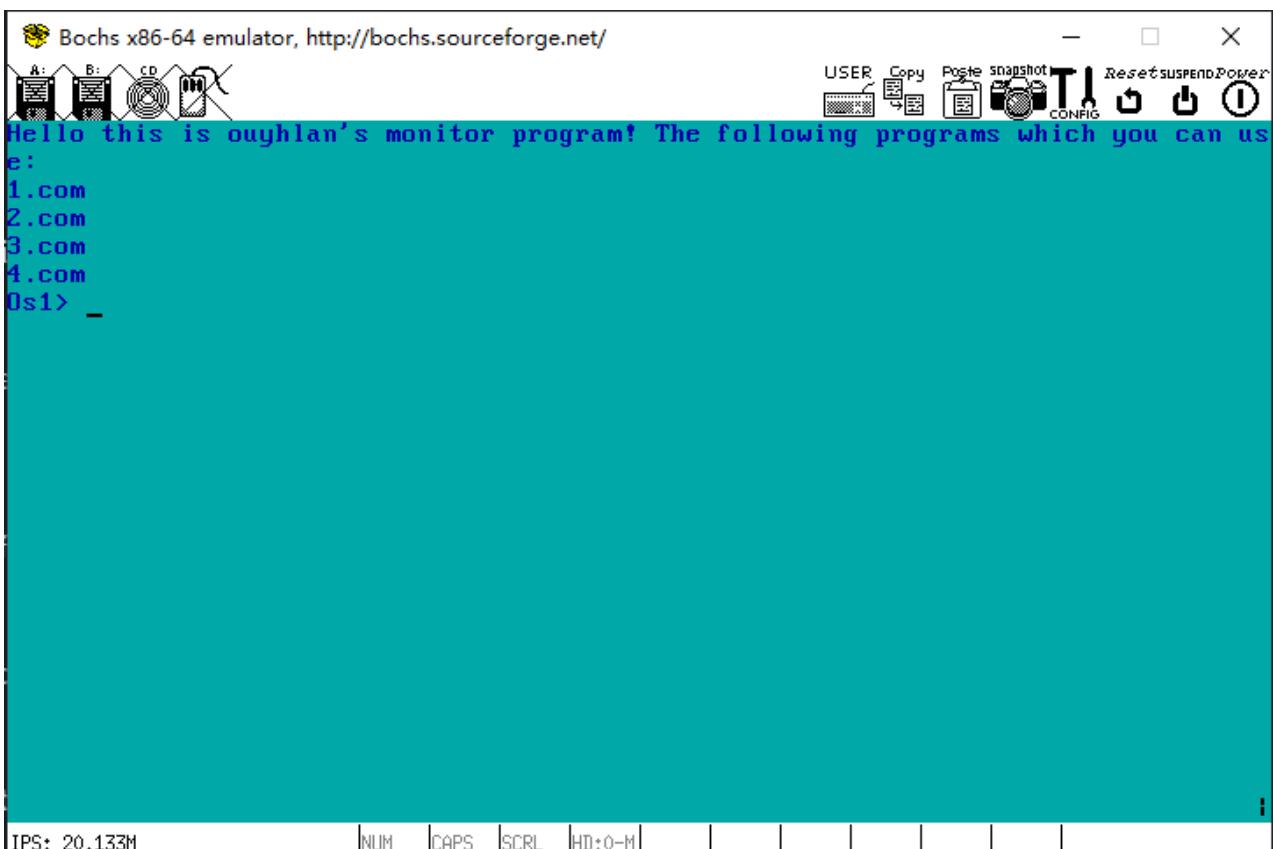
使用make run指令运行

```

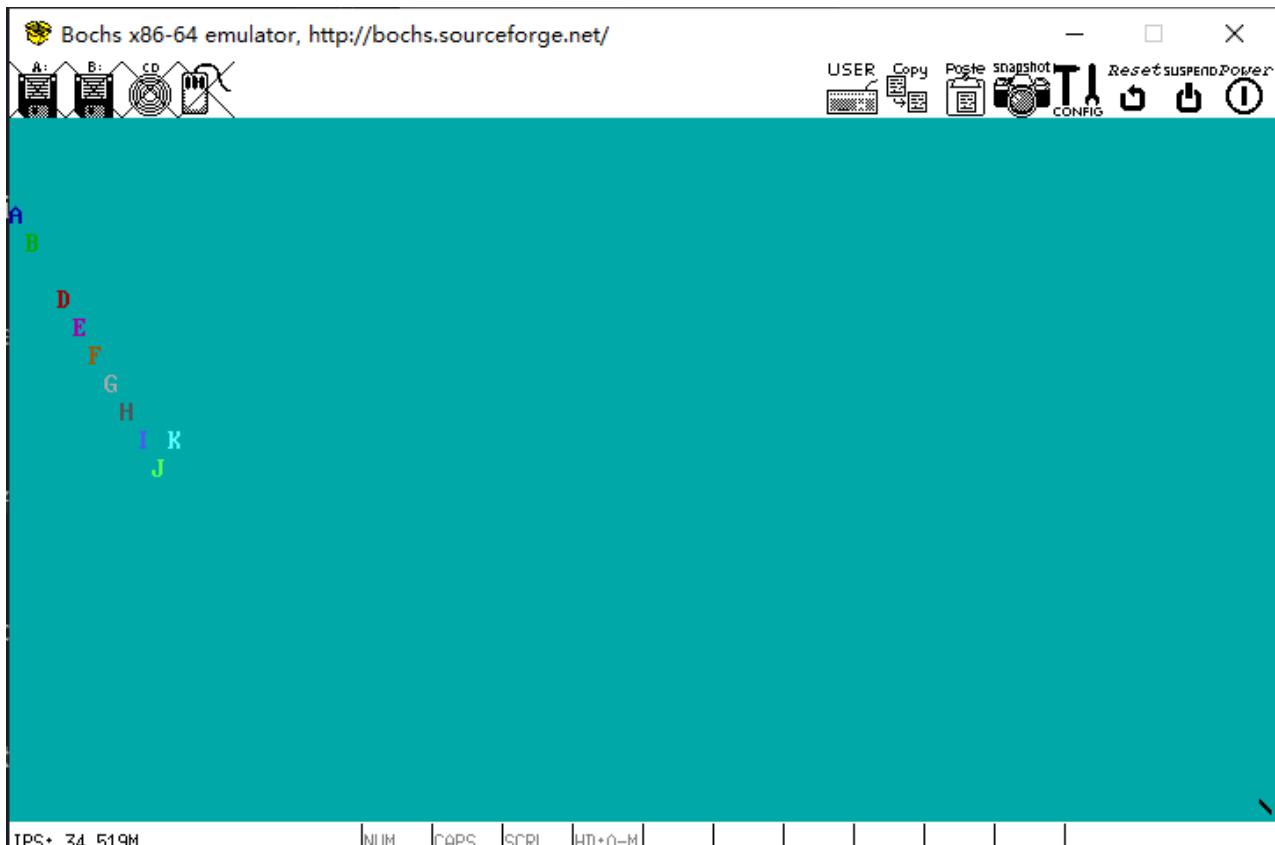
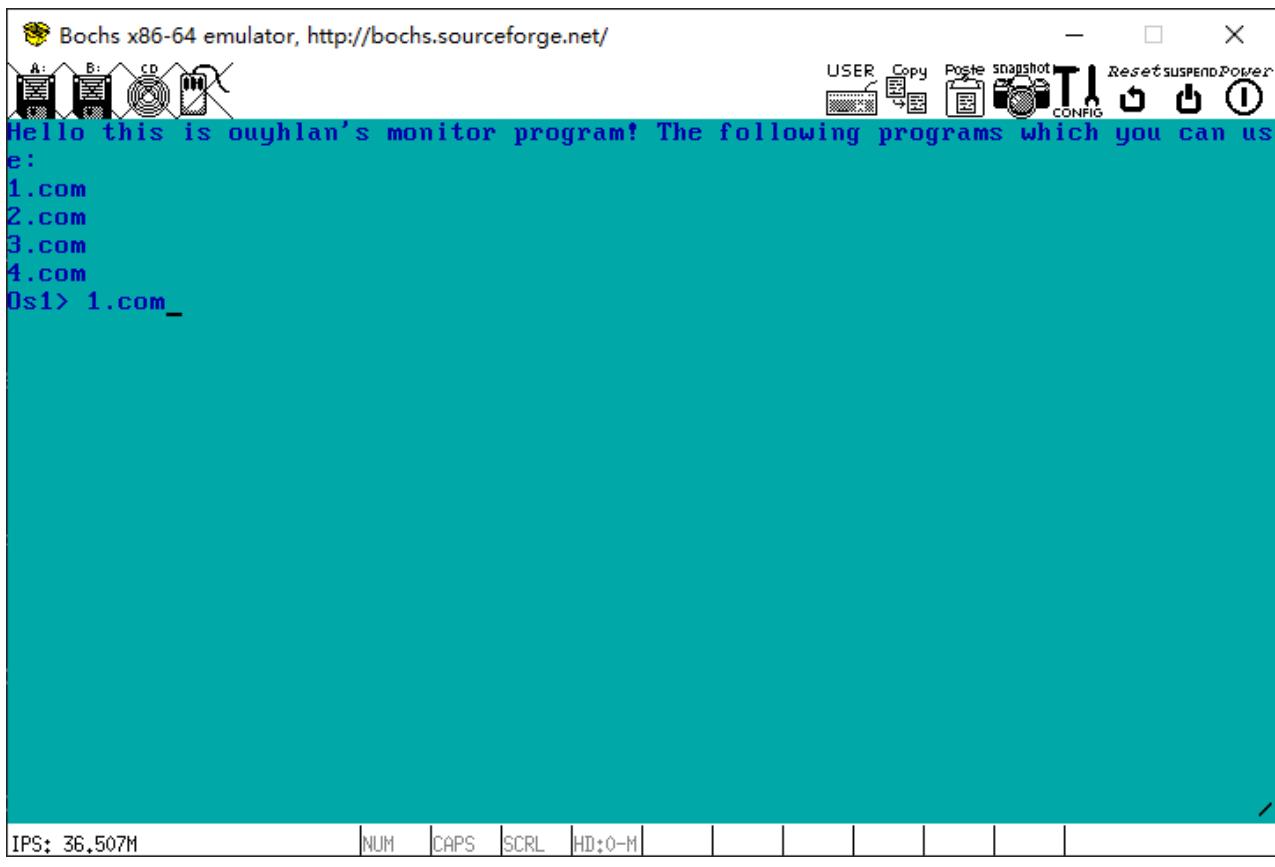
# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab4 on git:master x [13:43:20]
→ make run
ld --oformat binary -Ttext 0x7c00 boot.o puts.o -o boot.bin
ld --oformat binary -Ttext 0xfffff8000000000000 -Tdata 0xfffff8000000001000 main.o shell.o driver.o -o kernel.bin
dd if=boot.bin of=img/os1.img bs=512 seek=0 conv=notrunc; \
dd if=kernal.bin of=img/os1.img bs=512 seek=5 conv=notrunc; \
dd if=user1.bin of=img/os1.img bs=512 seek=16 conv=notrunc; \
dd if=user2.bin of=img/os1.img bs=512 seek=19 conv=notrunc; \
dd if=user3.bin of=img/os1.img bs=512 seek=22 conv=notrunc; \
dd if=user4.bin of=img/os1.img bs=512 seek=25 conv=notrunc; \

4+1 records in
4+1 records out
2336 bytes (2.3 kB, 2.3 KiB) copied, 0.0029982 s, 779 kB/s
9+1 records in
9+1 records out
4616 bytes (4.6 kB, 4.5 KiB) copied, 0.0014397 s, 3.2 MB/s
2+1 records in
2+1 records out
1473 bytes (1.5 kB, 1.4 KiB) copied, 0.0012923 s, 1.1 MB/s
2+1 records in
2+1 records out
1477 bytes (1.5 kB, 1.4 KiB) copied, 0.0010861 s, 1.4 MB/s
2+1 records in
2+1 records out
1473 bytes (1.5 kB, 1.4 KiB) copied, 0.0011265 s, 1.3 MB/s
2+1 records in
2+1 records out
1477 bytes (1.5 kB, 1.4 KiB) copied, 0.0012115 s, 1.2 MB/s
bochs -f bochsrc.bxrc
=====
Bochs x86 Emulator 2.6
Built from SVN snapshot on September 2nd, 2012
=====
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 0x7fffef2a6040

```

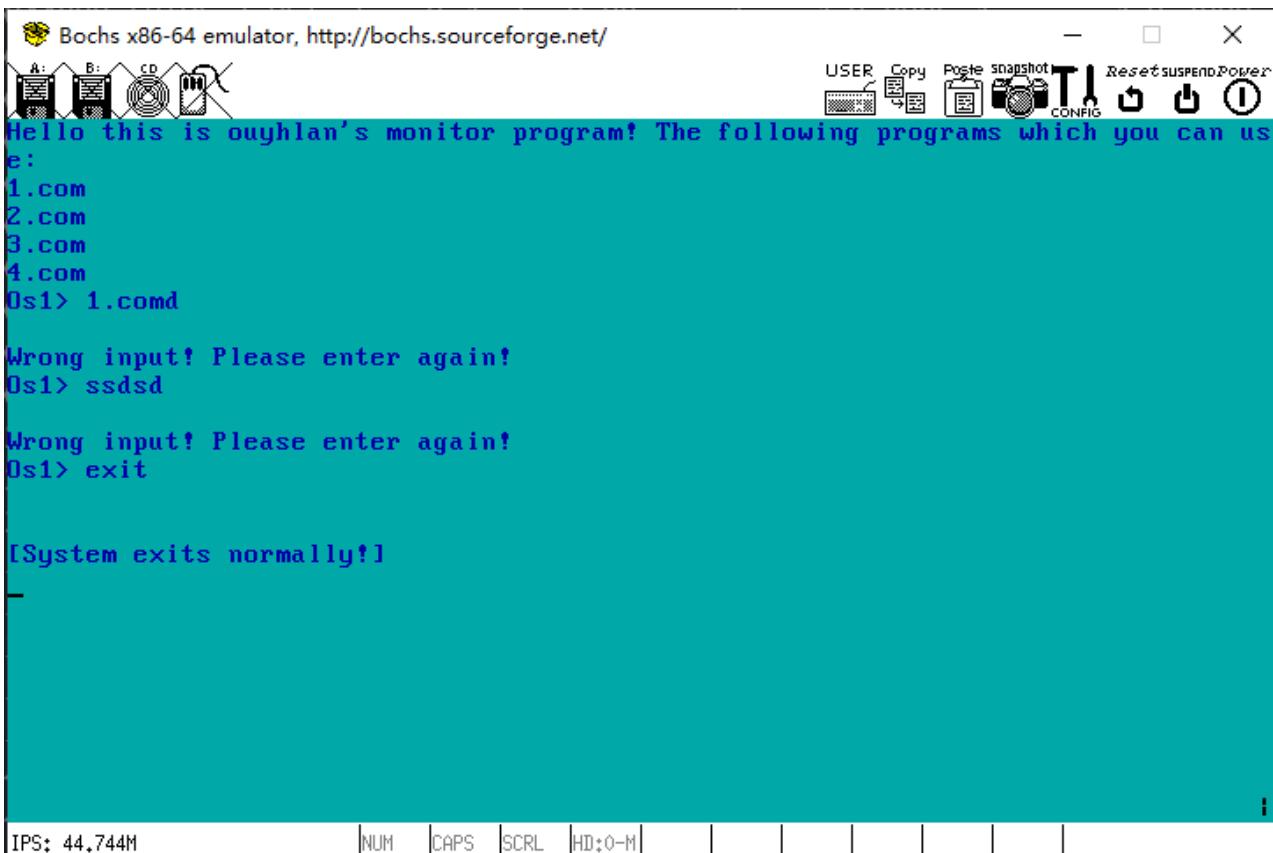


输入一个用户程序名



从上面三张截图的右下角，可以看出风火轮正常运行。

检测非法输入功能：



(3) 用户程序键盘中断OUCH!

这部分主要是内核程序要在跳转到用户程序前要进行键盘中断的切换，完成了用户程序以后，又要切换回来。

代码的编写

```
// 这部分代码位于内核部分
while (1) {
    puts("Os1> ");
    gets(str);
    int program_id = 0;
    if (StrCmp(str, "1.com", 5) == 0 && (StrLen(str) == 5)) {
        program_id = 1;
    }
    else if (StrCmp(str, "2.com", 5) == 0 && (StrLen(str) == 5)) {
        program_id = 2;
    }
    else if (StrCmp(str, "3.com", 5) == 0 && (StrLen(str) == 5)) {
        program_id = 3;
    }
    else if (StrCmp(str, "4.com", 5) == 0 && (StrLen(str) == 5)) {
        program_id = 4;
    }
    else if (StrCmp(str, "exit", 4) == 0 && (StrLen(str) == 4)) {
        puts("\n[System exits normally!]\n");
        break;
    }
}
```

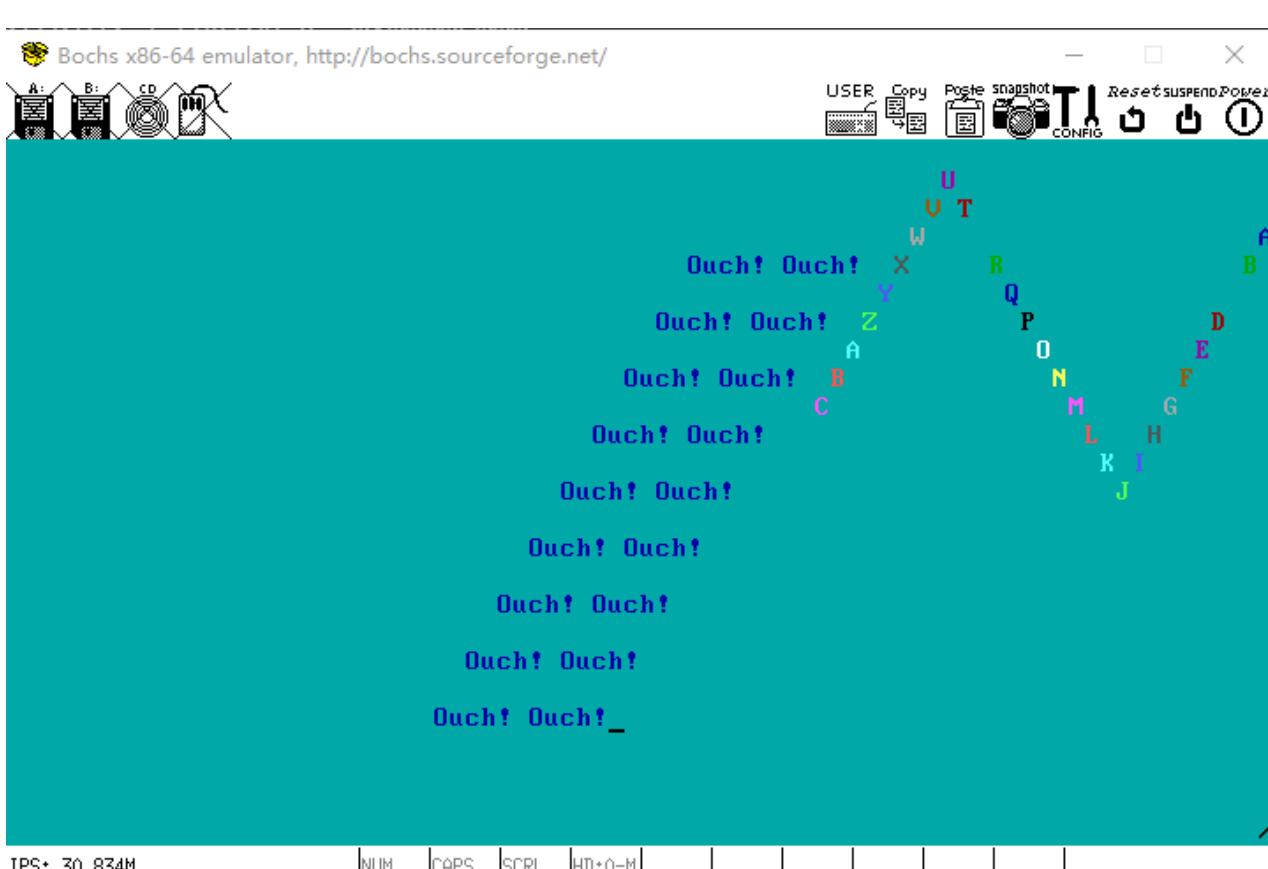
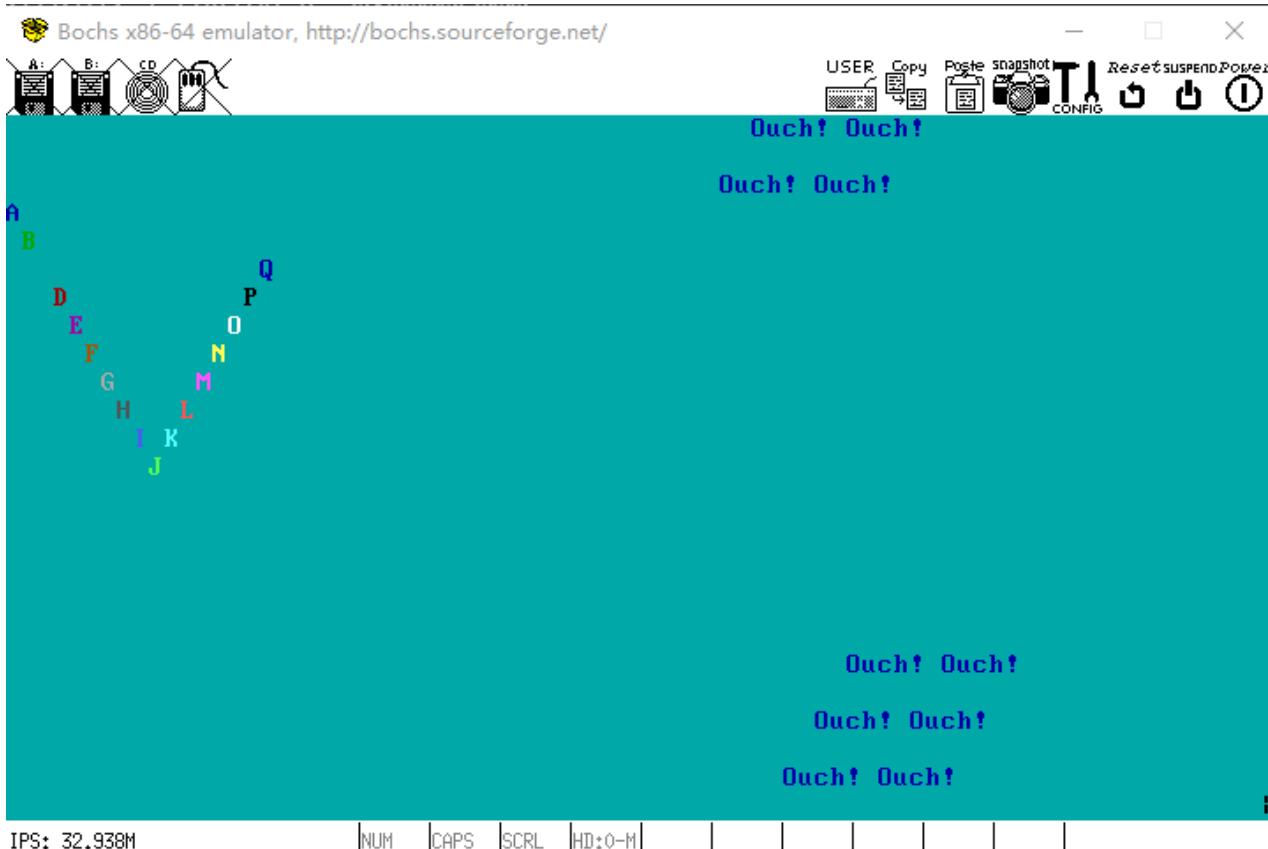
```
    }
    else {
        puts("Wrong input! Please enter again!\n");
        continue;
    }
SetOuch();
switch (program_id) {
case 1:
    LoadUserProgram(USER_1, 3);
    break;
case 2:
    LoadUserProgram(USER_2, 3);
    break;
case 3:
    LoadUserProgram(USER_3, 3);
    break;
case 4:
    LoadUserProgram(USER_4, 3);
    break;
default:
    /* Do nothing */;
}
InitKeyboard();
}
```

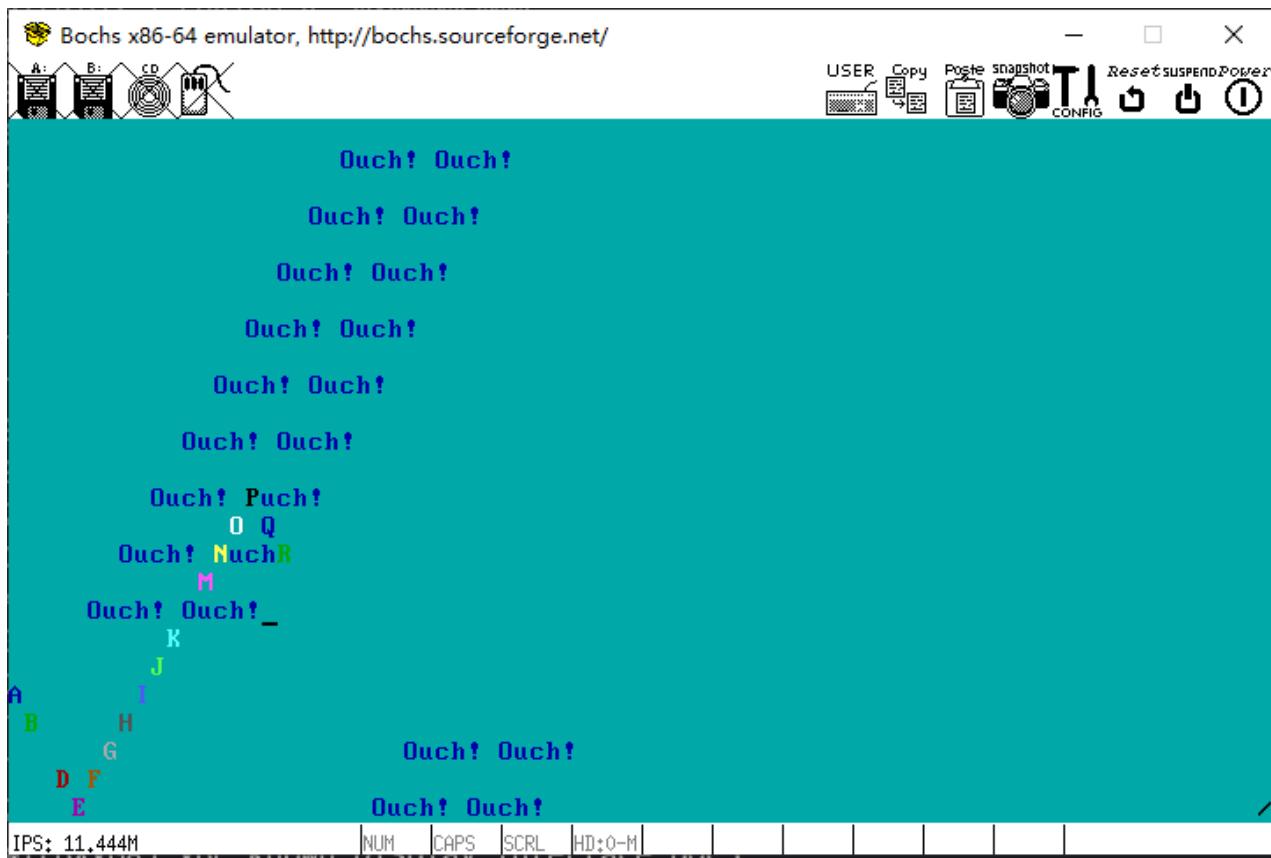
这个用户输入的处理。中间的 `SetOuch()` 正是切换处理，而 `InitKeyboard()` 则是把中断切换回键盘输入。

代码的运行

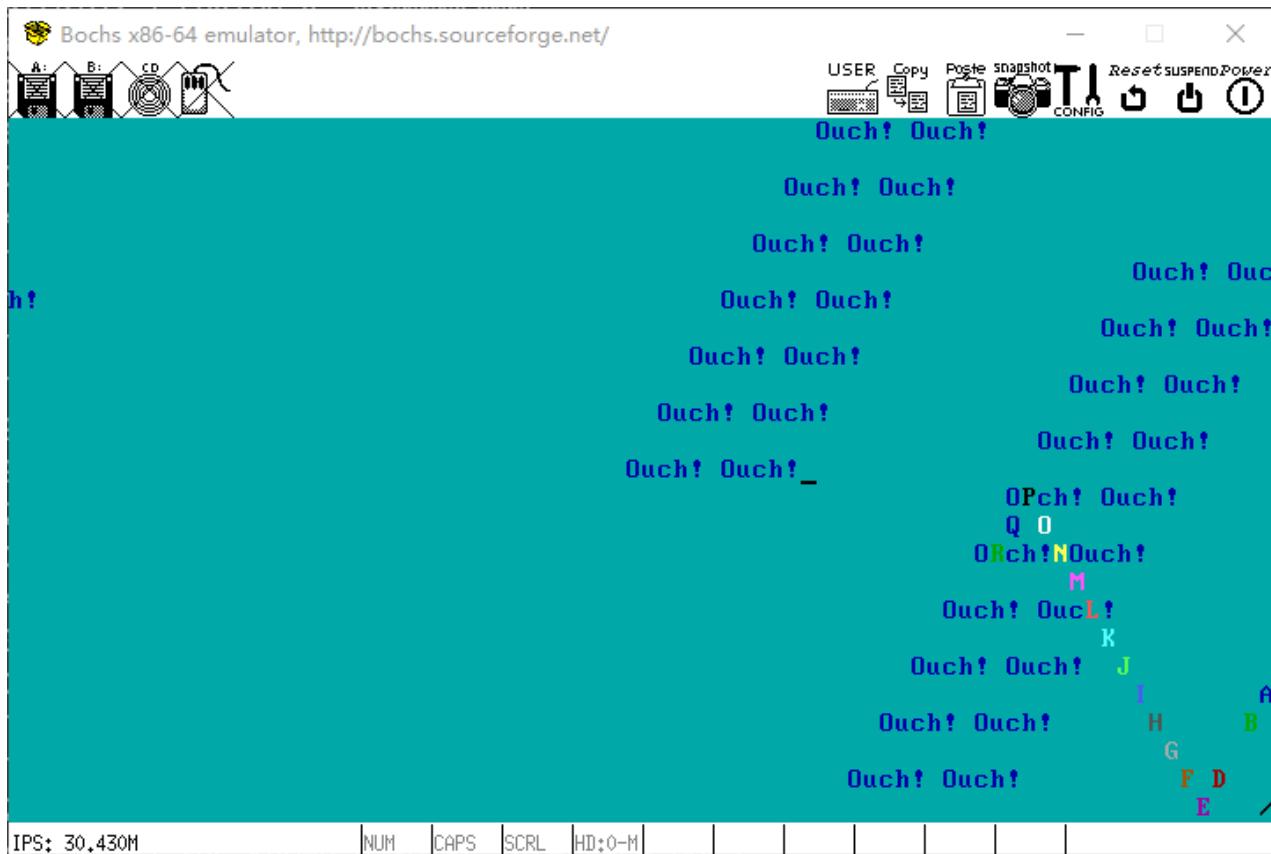
代码的编译同上，这里直接展示Ouch! 中断：

user1:





user4:



实验中遇到的问题

(1) Makefile的问题

这部分主要是一些语法上的不熟悉，一开始没有建立好文件之间的依赖关系，导致修改了头文件而makefile不更新的问题。

(2) gcc编译的问题

这部分是一些C语言的特性和gcc编译的实现方面导致的问题。

```
# ouyilan @ DESKTOP-3TMC71 in ~/Code Workplace/os/Lab4 on git:master x [13:58:37] C:2
+ make kernal_debug
gcc -c -ffreestanding -I ./include -mgeneral-regs-only ./kern/shell.c -o shell.o
./kern/shell.c: In function 'LoadUserProgram':
./kern/shell.c:28:28: warning: initialization of 'void *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
  28 |     void *user_code_addr = 0x400000;
     |             ^
./kern/shell.c: In function 'Welcoming':
./kern/shell.c:43:31: warning: initialization of 'unsigned char *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
  43 |     unsigned char *vga_addr = 0xb8000;
     |             ^
./kern/shell.c: Assembler messages:
./kern/shell.c:30: Warning: indirect call without '*'
ld -Ttext 0xffff800000000000 -Tdata 0xffff800000001000 main.o shell.o driver.o -o kernal_debug.bin; \
objdump -d kernal_debug.bin
ld: failed to convert GOTPCREL relocation; relink with --no-relax
objdump: 'kernal_debug.bin': No such file
Makefile:68: recipe for target 'kernal_debug' failed
make: *** [kernal_debug] Error 1
```

这里出现的问题就是我的shell.c文件使用了另一个文件里的函数地址。当我使用objdump反汇编的时候，我发现它使用了一个叫GOT表的动态链接技术。

```
ffff800000000392:    90          .nop
ffff800000000393:    48 8b 05 5e 0c 20 00  mov    0x200c5e(%rip),%rax      # ffff800000200ff8 <.got>
ffff80000000039a:    48 89 c6          mov    %rax,%rsi
ffff80000000039d:    bf 21 00 00 00  mov    $0x21,%edi
ffff8000000003a2:    e8 64 00 00 00  callq  ffff80000000040b <SetInterrupt>
ffff8000000003a7:    e9 25 fe ff ff  jmpq   ffff8000000001d1 <Welcoming+0xd8>
ffff8000000003ac:    90          nop
ffff8000000003ad:    c9          leaveq
ffff8000000003ae:    c3          retq
```

但很显然，我还没有实现这部分，所以代码出现了bug。

后面我把使用函数地址和函数定义体放在了同一个文件里，解决了这个问题。

(3) 中断导致的问题

我的系统看上去没有明显问题，但是在打开了时钟中断以后，系统会在某些地方莫名的死机。

经过测试，原来是这个地方通用寄存器的值突然发生了变化。在仔细debug后，我发现原来是中断服务例程的问题。

我在上个实验中对于中断服务例程代码的书写格式：

```
void KbHandler() {
    unsigned char scan_code = inp(0x60);
    /* 暂时不支持大写输入 */
    if (scan_code < 0x80) {
        char ch = normalmap[scan_code];
        output = ch;
    }
    __asm__("movb $0x20, %al\n\t");
    __asm__("out %al, $0x20\n\t");
    __asm__("nop\n\tleaveq\n\tiretq\n\t");
}
```

而实际上，中断和系统内核间，可以看作是并发执行的。那么切换到中断的时候，相应上下文应该正确的保存以后。

同时，x86汇编里面有个caller-saved register和callee-saved register的规则，即gcc汇编出来的汇编代码，在函数的入口部分并不会帮你保存所有的寄存器，而只会保存callee-saved register里面规定的寄存器。

Figure 3.4: Register Usage

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of SSE registers used; 1 st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4 th integer argument to functions	No
%rdx	used to pass 3 rd argument to functions; 2 nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2 nd argument to functions	No
%rdi	used to pass 1 st argument to functions	No
%r8	used to pass 5 th argument to functions	No
%r9	used to pass 6 th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12-r15	callee-saved registers	Yes
%xmm0-%xmm1	used to pass and return floating point arguments	No
%xmm2-%xmm7	used to pass floating point arguments	No
%xmm8-%xmm15	temporary registers	No
%mmx0-%mmx7	temporary registers	No
%st0	temporary register; used to return long double arguments	No
%st1	temporary registers; used to return long double arguments	No
%st2-%st7	temporary registers	No
%fs	Reserved for system use (as thread specific data register)	No

所以，我把中断例程的格式改为了下面的写法：

```

__attribute__ ((interrupt))
void KbHandler(struct interrupt_frame* frame) {
    unsigned char scan_code = inp(0x60);
    /* 暂时不支持大写输入 */
    if (scan_code < 0x80) {
        char ch = normalmap[scan_code];
        output = ch;
    }
    EOI();
}

```

上面这种写法是GNU对于x86中断服务例程的固定格式。经过测试，之前的莫名bug解决了。

(4) bochs提示键盘缓冲区已满

经过排查，其实是因为没有使用打开中断标志位指令 `sti`，导致中断服务例程没有被调用，而键盘的输入缓冲区有限。

实验总结

这次是我在操作系统实验课的第四个实验，本次实验，是在前一个实验基础上的一次修修补补。

事实上，长模式的中断我已经在上个实验处理键盘输入问题上解决过了，这次实验我主要是构建一个完整的编译环境——纯Linux环境进行原型操作系统的开发。如同上面说的，我通过利用Makefile，已经实现了前几次实验总结提到的自动化构建项目。目前已经实现的功能有：

- 一键编译和链接
- 一键运行（指的是使用bochs运行操作系统）
- 一键备份源文件（防止因为Makefile的一些书写错误导致实验文件丢失）
- 一键debug（使用objdump查看反汇编后的汇编代码）

后续的实验中，我还将会对Makefile实现以下的功能：

- 自动生成源文件依赖，防止因为项目规模扩大而导致Makefile文件的编写变得困难
- 使用git管理项目版本

另外，这次实验我也解决了一直困扰我的一个bug——键盘输入中断不符合代码逻辑（听着有点玄学）。实际上，问题出在同步问题（即中断和内核进程实际上是并发执行的，其中会涉及一些同步上的问题）。而这个问题，我通过原理课上所学到的信号量知识，再通过网上查阅到的GNU对于x86汇编中断服务例程的写法，终于完美地解决了中断的问题。

在后续，我会继续完善键盘的输入功能，包括：

- 建立系统内核的输入缓冲区
- 增加大写支持
- 增加回退backspace的支持
- 增加Ctrl-Z、Ctrl-C的信号处理

最后，在完整使用了Linux操作系统的工具链以后，我觉得整个开发速度和开发效率都提高了很多，接下来将会专注于对操作系统功能的进一步完善。