



# 本科生实验报告

实验课程：\_\_\_\_\_操作系统\_\_\_\_\_

实验名称：\_\_\_\_\_第三章：中断\_\_\_\_\_

专业名称：\_\_\_\_\_网络空间安全\_\_\_\_\_

学生姓名：\_\_\_\_\_陈浚铭\_\_\_\_\_

学生学号：\_\_\_\_\_20337021\_\_\_\_\_

实验地点：\_\_\_\_\_温暖的家\_\_\_\_\_

实验成绩：\_\_\_\_\_

报告时间：\_\_\_\_\_

## 1. 实验要求

对操作系统的中断有深入的理解

## 2. 实验过程和结果

实验 1:

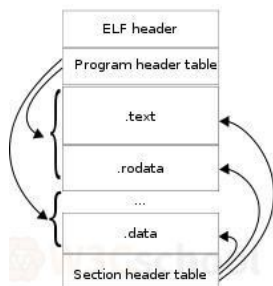
```
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/4$ dir
asm_utils.asm  c_func.c  c_func.o  cpp_func.cpp  main.cpp  main.out  Makefile
```

在文件夹里， 我们有 `c_func.c`, `cpp_func.cpp` 分别为 C 和 C++ 的代码并提供了 `asm_utils.asm` 要使用的函数。之后，我们有 `asm_utils.asm` 它里面有一个函数分别调用 C 和 C++ 的函数。最后 `main.cpp` 调用 `asm_utils.asm` 的函数。

```
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/4$ gcc -o c_func.o -m32 -c c_func.c
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/4$ g++ -o cpp_func.o -m32 -c cpp_func.cpp
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/4$ g++ -o main.o -m32 -c main.cpp
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/4$ nasm -o asm_func.o -f elf32 asm_func.asm
nasm: fatal: unable to open input file 'asm_func.asm'
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/4$ dir
asm_utils.asm  c_func.o  cpp_func.o  main.o  Makefile
c_func.c      cpp_func.cpp  main.cpp  main.out
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/4$ nasm -o asm_func.o -f elf32 asm_utils.asm
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/4$ g++ -o main.out main.o c_func.o cpp_func.o asm_func.o -m32
g++: error: asm_func.o: No such file or directory
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/4$ g++ -o main.out main.o c_func.o cpp_func.o asm_func.o -m32
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/4$ ./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done.
```

这里我们看到把 4 个文件从新定义为可重定位文件 `.o`，然后把这 `.o` 文件进行链接的过程。

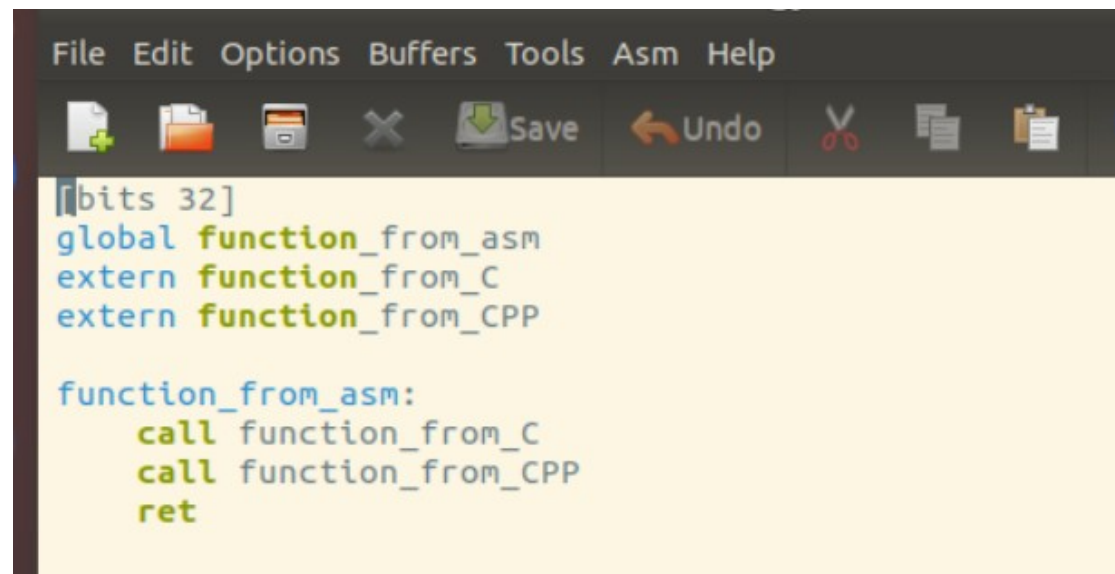
这里我们在使用 `nasm` 编译汇编文件 `asm_utils.asm` 的时候，使用了 `-f elf32` 的 options。ELF 的格式如下：



ELF 文件格式除了 `.o`（可从新定位文件）以外，还有

.out（可执行文件）和.so（共享目标文件）。可执行文件就是可以直接执行的文件，而共享目标文件就是用在以动态链接的文件。

我们看见到程序打印的次序是根据在 asm\_utils.asm 调用 C 函数然后调用 C++函数的次序。从 asm\_utils.asm 可见，



```
[bits 32]
global function_from_asm
extern function_from_C
extern function_from_CPP

function_from_asm:
    call function_from_C
    call function_from_CPP
    ret
```

在 function\_from\_asm 函数之前有 global function\_from\_asm, extern function\_from\_C 和 extern function\_from\_CPP。extern 语句用来声明这些函数来自外部。global 语句用来声明汇编代码能够被外部的 C/C++ 程序代码使用。

```

main.out: main.o c_func.o cpp_func.o asm_func.o
    g++ -o main.out main.o c_func.o cpp_func.o asm_func.o -m32

c_func.o: c_func.c
    gcc -o c_func.o -m32 -c c_func.c

cpp_func.o: cpp_func.cpp
    g++ -o cpp_func.o -m32 -c cpp_func.cpp

main.o: main.cpp
    g++ -o main.o -m32 -c main.cpp

asm_func.o: asm_func.asm
    nasm -o asm_func.o -f elf32 asm_func.asm

clean:
    rm *.o

```

从例子里面的 makefile， 我们看到 最终目标： main.out 可执行文件和中间文件 main.o c\_func.o cpp\_func.o asm\_func.o 的关系。而下一条产生 main.o 的 shell 指令。

使用 make 命令来编译的过程：

```

jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/4$ make
g++ -o main.o -m32 -c main.cpp
gcc -o c_func.o -m32 -c c_func.c
g++ -o cpp_func.o -m32 -c cpp_func.cpp
nasm -o asm_func.o -f elf32 asm_func.asm
g++ -o main.out main.o c_func.o cpp_func.o asm_func.o -m32
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/4$ ./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done.
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/4$ 

```

实验 2:

我们在这个实验透过 C 和汇编的混合， 来实现 kernel 从硬盘加载到内存并执行。

```

jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$ nasm -o mbr.bin -f bin -I../include/ ../src/boot/mbr.asm
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$ nasm -o bootloader.bin -f bin -I../include/ ../src/boot/bootloader.asm

```

首先我们需要把 bootloader.asm 和 mbr.asm 编译到二进制文件， 正如在实验 3 一样， 但这里我们用到 include 文件夹的 boot.inc 里面的参数需要 -I option 来链接。

```

jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$ nasm -o entry.obj -f elf32 ../src/boot/entry.asm
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$ nasm -o asm_utils.o -f elf32 ../src/utlis/asm_utils.asm
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$ g++ -g -Wall -march=i386 -m32 -nostdlib -fno-builtin -ffreestanding -fno-pic -I.
./include -c ../src/kernel/setup.cpp
ccplus: note: valid arguments to '-march=' switch are: i386 i486 i586 pentium lakenont pentium-mmx winchip-c6 winchip2 c3 samuel-2 c3-2 nehemi
ah c7 esther i686 pentiumpro pentium2 pentium3 pentium3m pentium-m pentium4 pentium4m prescott nocona core2 nehalem core17 westmere sandybridge
core17-avx ivybridge core-avx-i haswell core-avx2 broadwell skylake skylake-avx512 bonnell atom silvermont slm knl geode k6 k6-2 k6-3 athlon a
thlon-tbird athlon-4 athlon-xp athlon-mp x86-64 eden-x2 nano nano-1000 nano-2000 nano-3000 nano-x2 eden-x4 nano-x4 k8 k8-sse3 opteron opteron-s
se3 athlon64 athlon64-sse3 athlon-fx amd64 barcelona bdver1 bdver2 bdver3 bdver4 znver1 btver1 btver2; did you mean 'i386'?
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$ g++ -g -Wall -march=i386 -m32 -nostdlib -fno-builtin -ffreestanding -fno-pic -I.
./include -c ../src/kernel/setup.cpp
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$

```

之后，我们需要把链接生成的重新定位文件分为两个文件  
只是包含代码的文件 kernel.bin 和 kernel.o， 当中  
kernel.o 只是用在 gdb 上调试。

```

jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$ ld -o kernel.o -me
lf_i386 -N entry.obj setup.o asm_utils.o -e enter_kernel -Ttext -0x00020000
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$ ld -o kernel.bin -
melf_i386 -N entry.obj setup.o asm_utils.o -e enter_kernel -Ttext -0x00020000
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$ ld -o kernel.bin -
melf_i386 -N entry.obj setup.o asm_utils.o -e enter_kernel -Ttext -0x00020000 --
oforamt binary

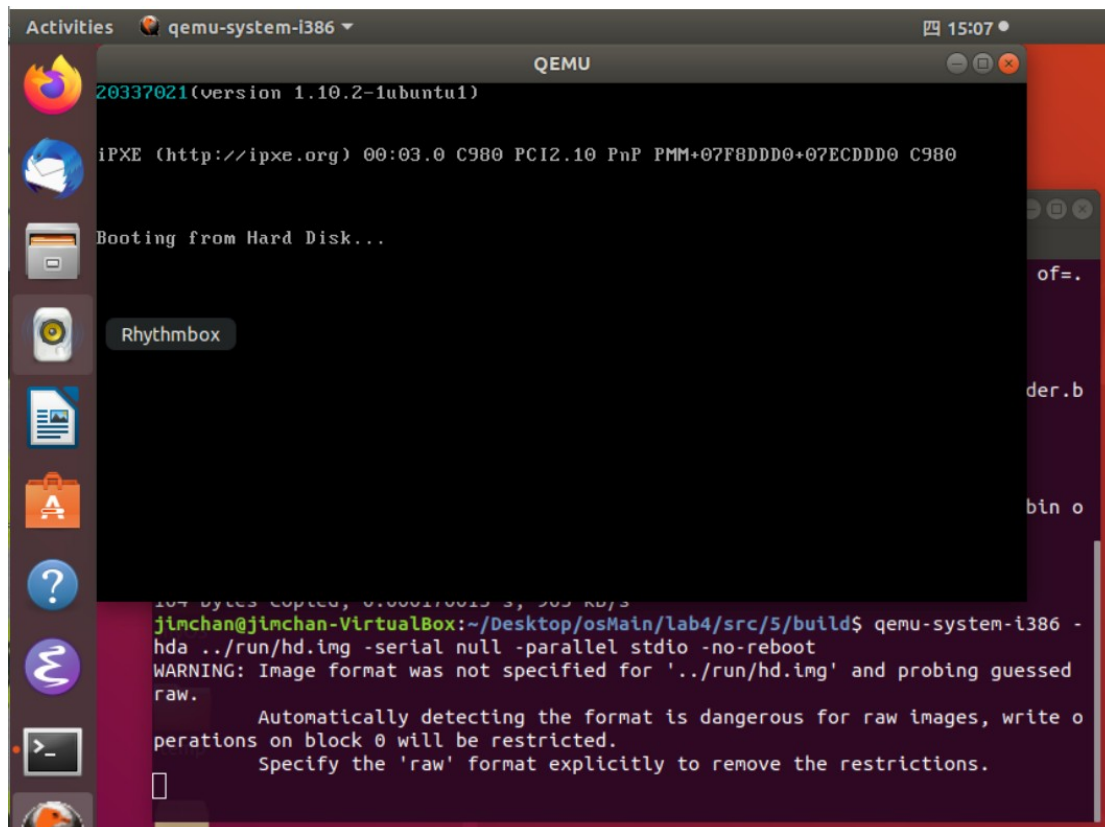
```

之后我们使用 dd 命令来把 mbr.bin, bootloader.bin,  
kernel.bin 写入硬盘， 如下所示：

```
jimchan@jimchan-VirtualBox: ~/Desktop/osMain/lab4/src/5/build
File Edit View Search Terminal Help
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$ dd if=mbr.bin of=
./run/hd.img bs=512 count=1 seek=0 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.00161724 s, 317 kB/s
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$ dd if=bootloader.b
in of=../run/hd.img bs=512 count=5 seek=1 conv=notrunc
0+1 records in
0+1 records out
281 bytes copied, 0.00923636 s, 30.4 kB/s
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$ dd if=kernel.bin o
f=../run/hd.img bs=512 count=200 seek=6 conv=notrunc
0+1 records in
0+1 records out
164 bytes copied, 0.000170013 s, 965 kB/s
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab4/src/5/build$
```

最后执行 `qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot`  
得到结果





### 实验 3

在这个实验我们通过 example 3 深入了解关于操作系统中断的原理。

在保护模式下，通过中断向量号来定义不同类型的中断。  
比如 0 号为除出错的异常（在这里也都是当作中断来处理），  
也就是 程序出现 Division by zero 的情况。

对于确定中断表的位置，我们使用汇编的函数 `lidt` 实现。  
`lidt` 的输入为一个 48 位的变量 `IDTR`，它包括了中断表的  
32 位地址和 16 位的表界限。之后，我们便可以设定 256 个  
中断向量号的中断描述符。透过 `InterruptManager` 类的  
`setInterruptDescriptor(uint32 index, uint32  
address, byte DPL)` 的函数，根据 64 中断描述符的定义：



void

```
InterruptManager::setInterruptDescriptor(uint32
index, uint32 address, byte DPL)
```

```
{
    IDT[index * 2] = (CODE_SELECTOR << 16) |
(address & 0xffff);

    IDT[index * 2 + 1] = (address & 0xffff0000) |
(0x1 << 15) | (DPL << 13) | (0xe << 8);
}
```

从代码可见，我们分别设定第  $i$  中断向量号的中断描述符：

目标代码段描述符选择子 = CODE\_SELECTOR

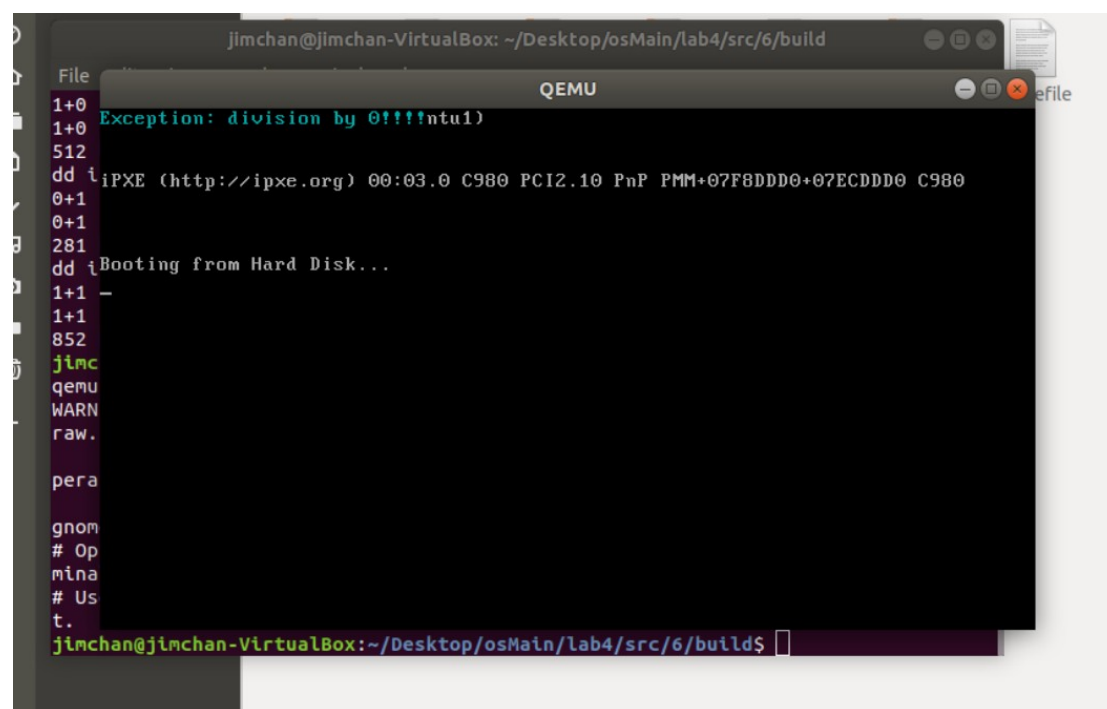
$$P(\text{段存在位}) = 1$$

$D = 1$ , 也就是我们运行 32bit 代码

中断处理过程在目标代码段内的偏移量 = address

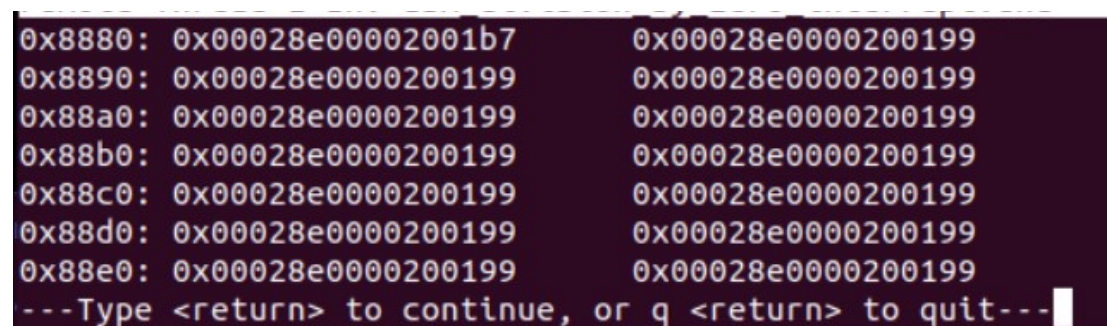


我在代码实现了修改：在中断标号为 0，也就是被 0 除的操作的中断里面的描述符的目标代码段内的偏移量等于我定义的 `asm_division_by_zero_interrupt()` 的代码地址，当中会输出 “Exception: division by 0!!!”



The screenshot shows a QEMU window titled "jimchan@jimchan-VirtualBox: ~/Desktop/osMain/lab4/src/6/build". The window displays the output of a virtual machine boot. The first line of output is "Exception: division by 0!!!!ntu1)". This is followed by the IPXE boot menu, which lists various boot options. The "Boot from Hard Disk..." option is selected, and the boot process continues. The output then shows the QEMU command line and the raw disk image path. The window also shows the QEMU version and the raw disk image path.

而我们在 gdb 看到



The screenshot shows a gdb debugger window displaying memory addresses and values. The addresses range from 0x8880 to 0x88e0, and the values are all 0x00028e0000200199. The output is as follows:

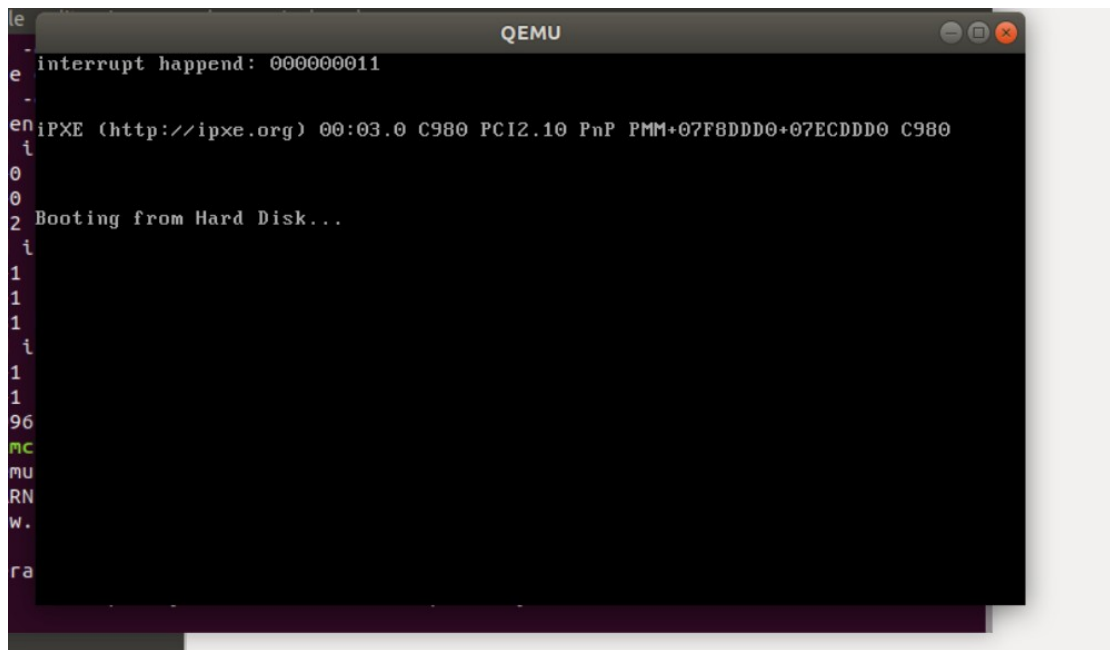
Address	Value
0x8880	0x00028e00002001b7
0x8890	0x00028e0000200199
0x88a0	0x00028e0000200199
0x88b0	0x00028e0000200199
0x88c0	0x00028e0000200199
0x88d0	0x00028e0000200199
0x88e0	0x00028e0000200199

---Type <return> to continue, or q <return> to quit---

第一个中断描述符跟其他的中断描述符的值不同，而其他的都为一样。这就可见我们对中断描述符进行了修改。

## 实验 4

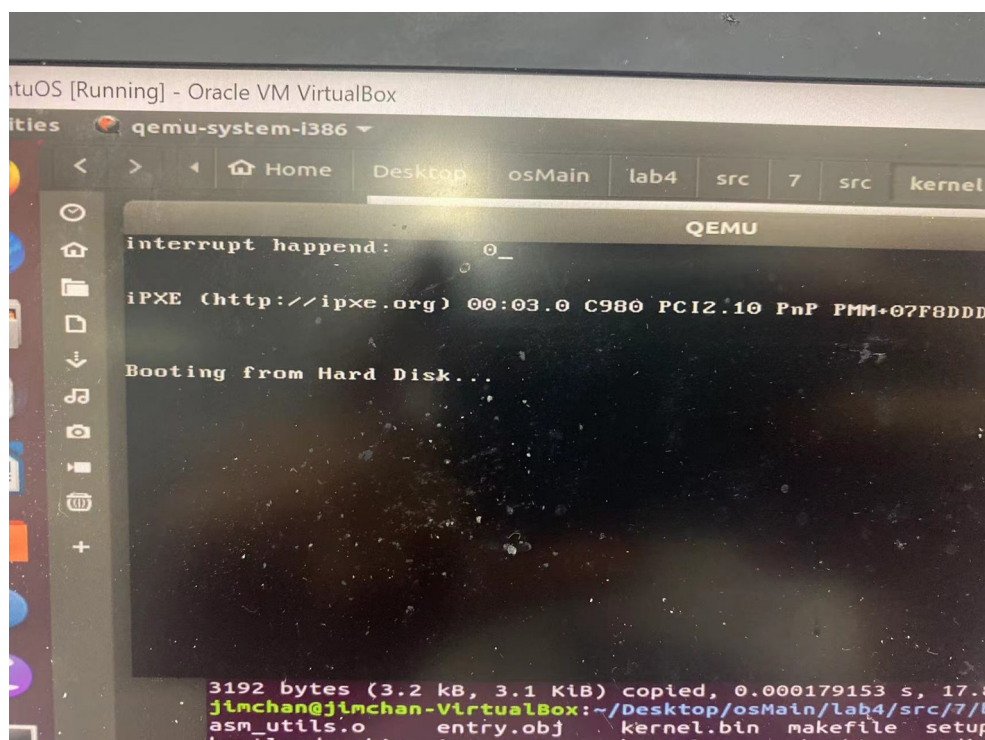
复现实验 4 的结果：

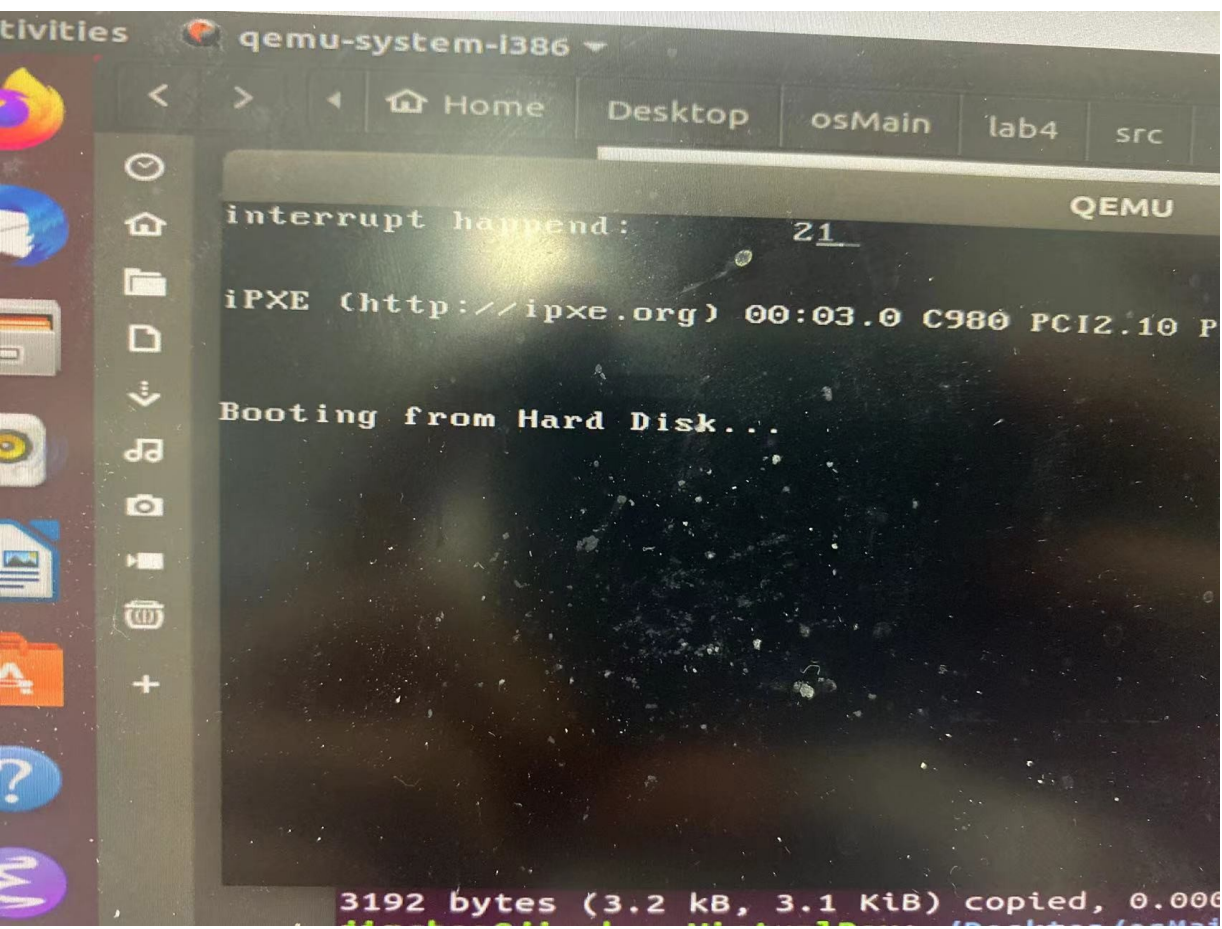
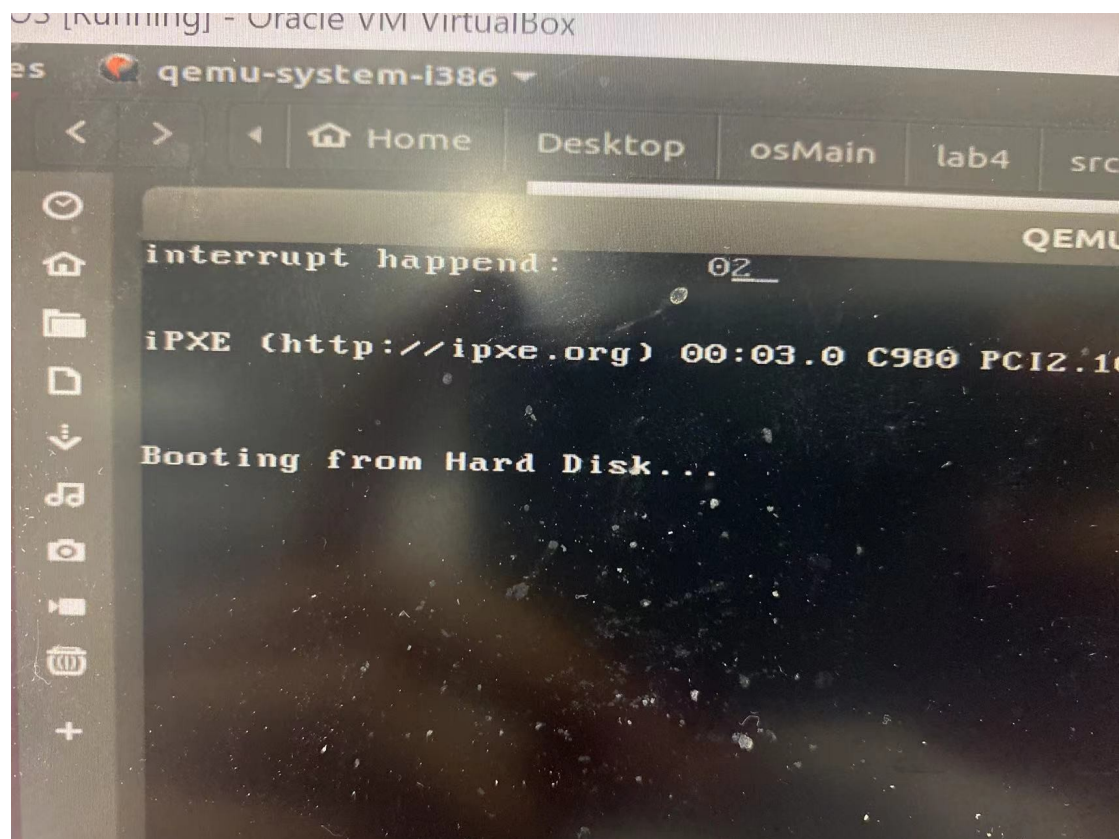


我们从实验可以了解到中断的操作。这里我们是时钟中断，我们设定在每一秒都会有一次中断，并输出发生中断的次数，所以那个数是不断的增加的。

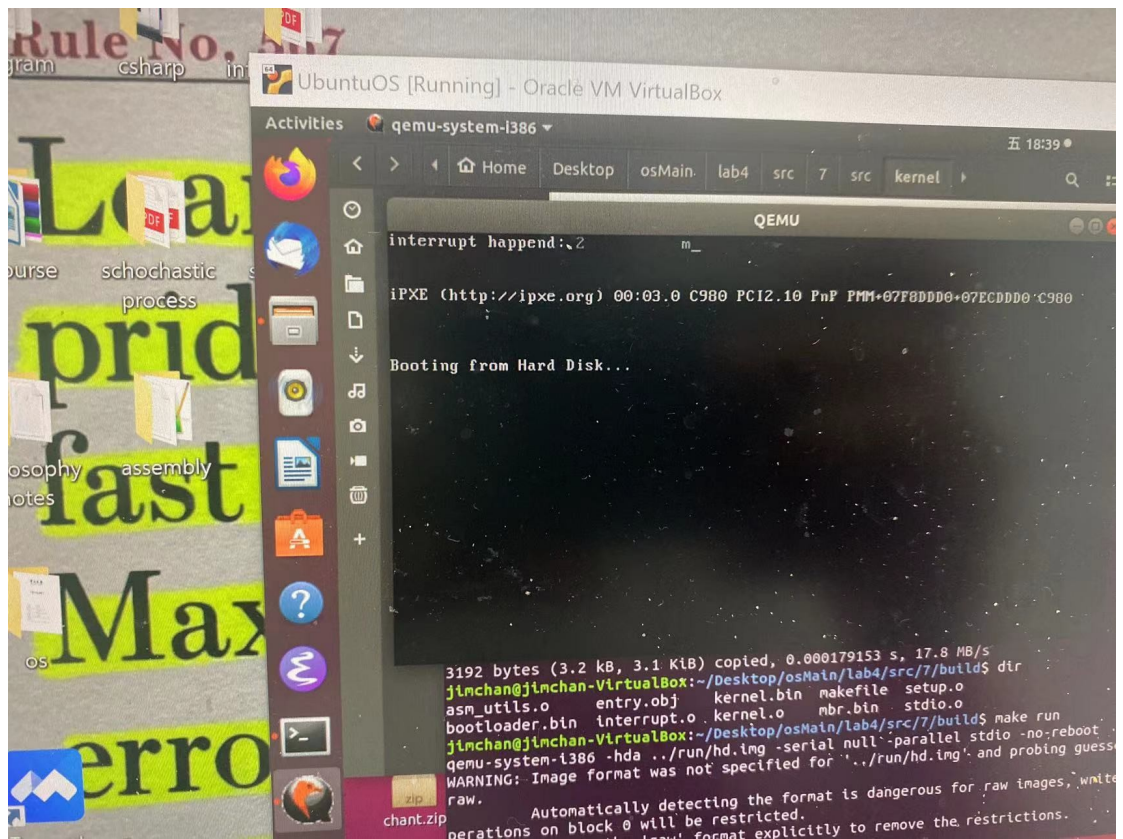
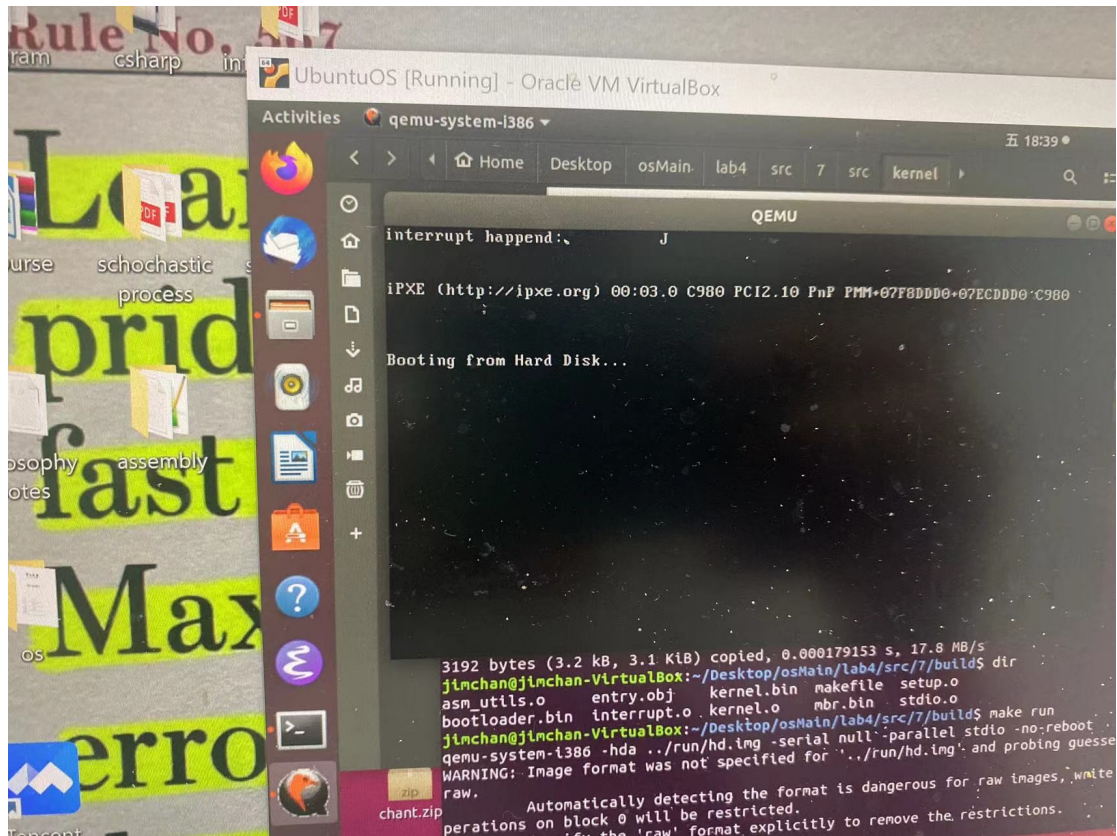
我们透过时间中断来实现一个显示学号和英文姓名的跑马灯。

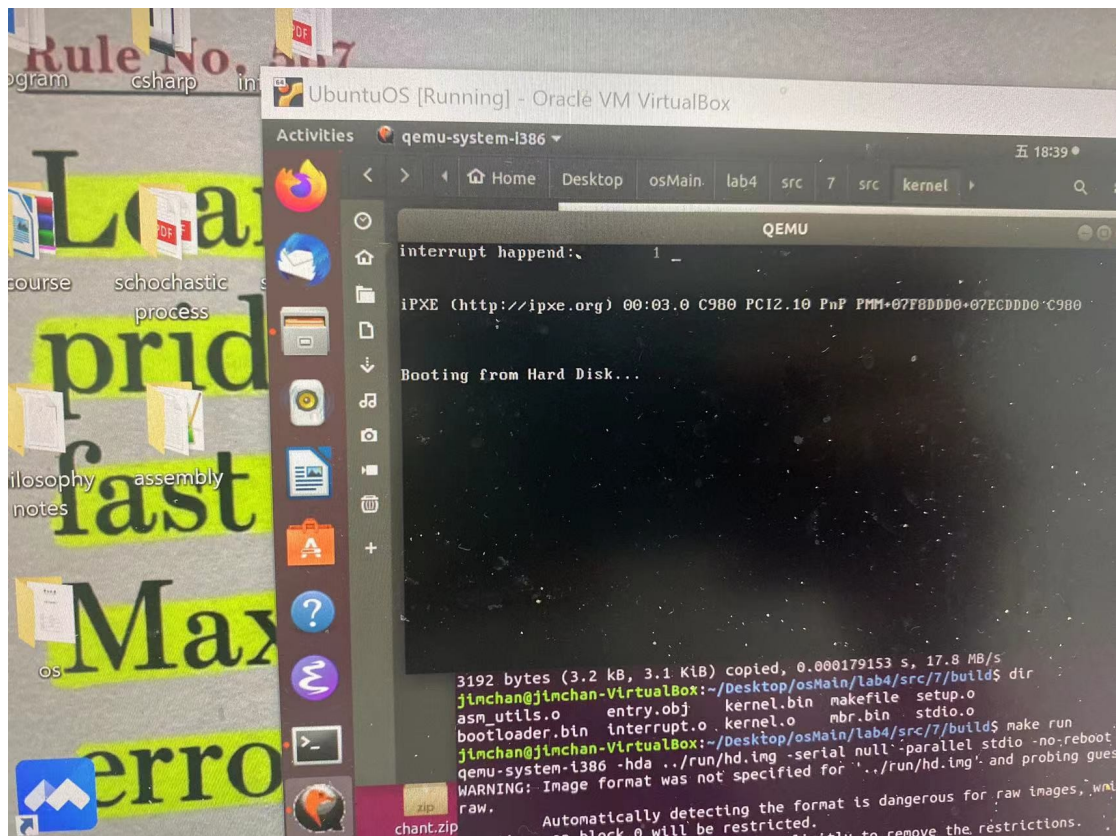
**实验结果：**









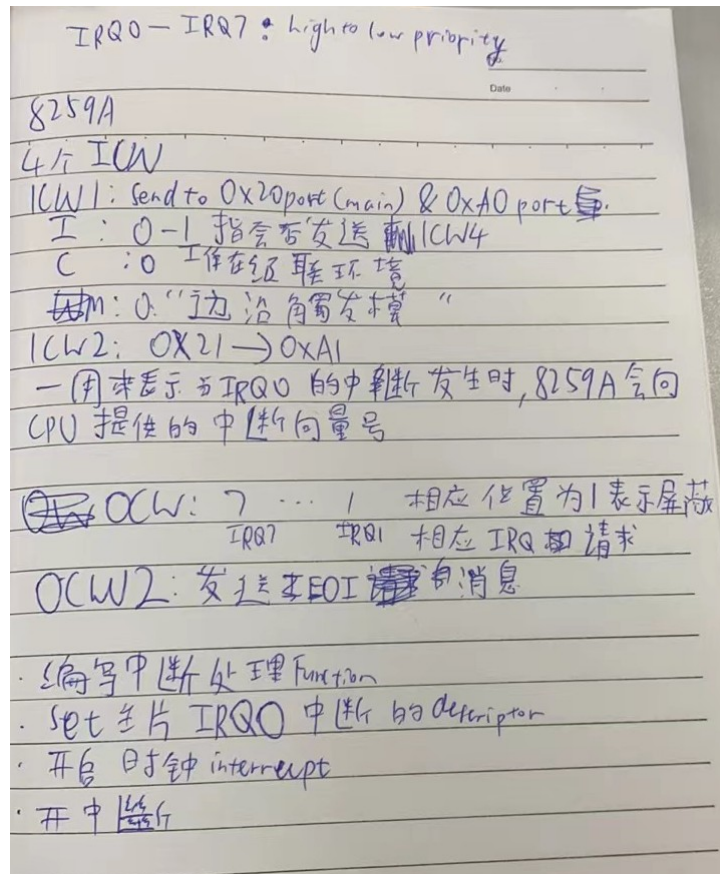


实验原理：

中断向量表是一个把中断处理程序列表与中断向量表中的中断处理请求互相关联的数据结构。 中断向量表的每个条目都是中断处理程序的地址。

我们知道 4 个 ICW1, ICW2, ICW3, ICW4 只是有 ICW2 是可变的，其他都为固定的。对于主片和从片， ICW2 是用来表示当 IRQ0 的中断发生时， 8259a 会向 CPU 提供的中断向量号，因此我们 ICW2 会设定对应时钟中断的中断向量号。





### 3. 关键代码

实验4的跑马灯方式显示学号和姓名:

```
extern "C" void schoolnumber_interrupt()
{
    char studentNumberAndNameArray[] = "20337021 Jim";
    const int lengthOfArray = 12;
    // 清空屏幕
    for (int i = 0; i < 80; ++i)
    {
        stdio.print(0, i, ' ', 0x07);
    }

    ++times;
    char str[] = "interrupt happend: ";
    char number[10];
    int temp = times;

    // 移动光标到(0,0)输出字符
    stdio.moveCursor(0);
    for(int i = 0; str[i]; ++i) {
        stdio.print(str[i]);
    }

    int outputIndex = times % lengthOfArray;
    for(int i = 0; i < outputIndex; i++)
    {
        stdio.print(' ');
    }
    stdio.print(studentNumberAndNameArray[outputIndex]);
}
```



除了写处理中断的函数以外， 我们需要在 asm\_utils 进行改写：

```
11
12 extern schoolnumber_interrupt ←
13 ASM_UNHANDLED_INTERRUPT_INFO db 'Unhandled interrupt happened, halt...'
14                                db 0
15 ASM_IDTR dw 0
16          dd 0
17
18 asm_enable_interrupt:
19     sti
20     ret
21 asm_time_interrupt_handler:
22     pushad
23
24     ; 发送EOI消息，否则下一次中断不发生
25     mov al, 0x20
26     out 0x20, al
27     out 0xa0, al
28
29     call schoolnumber_interrupt ←
30
31     popad
32     iret
33
```

#### 4. 总结

在这次实验深入理解到中断在 OS 里面的作用。

中断的定义来自一个连接电脑的设备或则在电脑里面程序的信号，它要求 OS 停止目前在执行的程序，从而让中断处理程序来执行，执行完后恢复执行状态。

如果没有中断，那么处理器和外部设备的通信时，必须在向该设备发出指令后进行忙等待 (busy waiting)，因此中断使用以提高计算机工作的效率，每当外部设备有需要时就可以发送一个中断请求

大概分为软中断和硬中断，例如 DIVISION\_BY\_ZERO 基本上可看为一种软中断（其实在理论上应该定义

为一种异常)。而硬中断的例子就是打印机的工作已经打印完，并透过中断告诉 OS。