



中山大學
SUN YAT-SEN UNIVERSITY

本科生实验报告

实验课程: 操作系统原理实验

任课教师: 刘宁

实验题目: 保护模式进入内核以及保护模式下的中断

专业名称: 信息与计算科学

学生姓名: 罗弘杰

学生学号: 22336173

实验地点: 实验中心D503

实验时间: 2024/4/27

Section 1 实验概述

在本章中, 我们首先介绍一份C代码是如何通过预编译、编译、汇编和链接生成最终的可执行文件。接着, 为了更加有条理地管理我们操作系统的代码, 我们提出了一种C/C++项目管理方案。在做了上面的准备工作后, 我们开始介绍C和汇编混合编程方法, 即如何在C代码中调用汇编代码编写的函数和如何在汇编代码中调用使用C编写的函数。介绍完混合编程后, 我们来到了本章的主体内容——中断。我们介绍了保护模式下的中断处理机制和可编程中断部件8259A芯片。最后, 我们通过编写实时钟中断处理函数来将本章的所有内容串联起来。

通过本章的学习, 同学们将掌握使用C语言来编写内核的方法, 理解保护模式的中断处理机制和处理时钟中断, 为后面的二级分页机制和多线程/进程打下基础。

预备知识: 保护模式下汇编函数编程;

C语言和汇编函数混合编程;

makefile工具使用和gdb调试;

- 实验环境:
 - 虚拟机版本/处理器型号: ubuntu-18.0.4, 阿里云服务器 通用cpu
 - 代码编辑环境: vim
 - 代码编译工具: gcc, nasm
 - 重要三方库信息: 无

Section 3 实验任务

实验任务1:

复现汇编和C/C++混合编程的例子

实验任务2:

使用C/C++编写内核：复现网址中“内核的加载”部分，在进入 setup_kernel 函数后，将输出 Hello World 改为输出 你的学号+姓名首字母，保存结果截图并说说你是怎么做的。

实验任务3:

复现网址中“初始化IDT”部分，你可以更改默认的中断处理函数为你编写的函数，然后触发之，结果截图并说说你是怎么做的。要求：调用处理函数时输出包含个人学号或姓名信息。

实验任务4:

实现一个字符弹射程序

Section 4 实验步骤与实验结果

该节描述每个实验任务的具体的完成过程，包括思路分析、代码实现与执行、结果展示三个部分，实验任务之间的划分应当清晰明了，实验思路分析做到有逻辑、有条理。

----- 实验任务1-----

任务要求:

复现汇编和C/C++混合编程的例子

思路分析:

C/C++和汇编混合编程包含两个方面。

- 在C/C++代码中使用汇编代码实现的函数。
- 在汇编代码中使用C/C++中的函数。

对于第一点，需要在C/C++中声明使用的汇编函数来自外界extern--，并在汇编文件中，将函数声明为global;

对于第二点，需要在汇编文件中将使用函数声明为extern来自外界，然后对于C++函数要使用extern“C”前缀声明，避免c++的重载机制；对于带有参数的函数，要注意：

- 如果函数有参数，那么参数从右向左依次入栈。
- 如果函数有返回值，返回值放在eax中。
- 放置于栈的参数一般使用ebp来获取

代码分析

```
//main.cpp
#include <iostream>

extern "C" void function_from_asm(); //声明使用的函数来自外界

int main() {
    std::cout << "Call function from assembly." << std::endl;
    function_from_asm();
    std::cout << "Done by 22336173 Luo Hongjie" << std::endl;
}
```

```
gcc -o c_func.o -m32 -c c_func.c
g++ -o cpp_func.o -m32 -c cpp_func.cpp
g++ -o main.o -m32 -c main.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
```

实验结果：

```
root@ubuntu:/home/rogers2/os/lab4/task1# ls
Makefile      asm_func.o   c_func.o     cpp_func.o   main.o
asm_func.asm  c_func.c    cpp_func.cpp  main.cpp     main.out
root@ubuntu:/home/rogers2/os/lab4/task1# main.out
WARNING:root:could not open file '/etc/apt/sources.list'

main.out: command not found
root@ubuntu:/home/rogers2/os/lab4/task1# ./main.out
Call function from assembly.
This is a function from C
This is a function from C++.
Done by 22336173 Luo Hongjie
root@ubuntu:/home/rogers2/os/lab4/task1#
```

----- 实验任务2 -----

任务要求：

使用C/C++编写内核：复现网址中“内核的加载”部分，在进入 setup_kernel 函数后，将输出 Hello World 改为输出 你的学号+姓名首字母，保存结果截图并说说你是怎么做的

思路分析：

前面我们已经实现了进入保护模式的方式，然后在32位下需要设定内核的地址和大小，规定在保护模式下跳转的位置，在编写内核的时候使用C语言会加快效率。

例如本节文件目录如下。

```
├─ build
│   └─ makefile
├─ include
│   ├── asm_utils.h
│   ├── boot.inc
│   ├── os_type.h
│   └─ setup.h
├─ run
│   ├── gdbinit
│   └─ hd.img
└─ src
    ├── boot
    │   ├── bootloader.asm
    │   ├── entry.asm
    │   └─ mbr.asm
    ├── kernel
    │   └─ setup.cpp
    └─ utils
        └─ asm_utils.asm
```

代码分析：

常量的定义放置在 `5/include/boot.inc` 下，新增的内容如下。

```
; _____kernel_____
KERNEL_START_SECTOR equ 6
KERNEL_SECTOR_COUNT equ 200
KERNEL_START_ADDRESS equ 0x20000
```

我们在 `src/boot/entry.asm` 下定义内核进入点。

```
extern setup_kernel
enter_kernel:
    jmp setup_kernel
```

我们会在链接阶段巧妙地将 `entry.asm` 的代码放在内核代码的最开始部份，使得 `bootloader` 在执行跳转到 `0x20000` 后，即内核代码的起始指令，执行的第一条指令是 `jmp setup_kernel`。在 `jmp` 指令执行后，我们便跳转到使用 C++ 编写的函数 `setup_kernel`。此后，我们便可以使用 C++ 来写内核了。

`setup_kernel` 的定义在文件 `src/kernel/setup.cpp` 中，内容如下。

```
#include "asm_utils.h"

extern "C" void setup_kernel()
{
    asm_hello_world();
    while(1) {

    }
}
```

为了方便汇编代码的管理，我们将汇编函数放置在 `src/utils/asm_utils.h` 下，如下所示。

```
[bits 32]

global asm_hello_world

asm_hello_world:
    push eax
    xor eax, eax

    mov ah, 0x03 ;ÇàÉ«
    mov al, 'K'
    mov [gs:2 * 0], ax

    mov al, 'E'
    mov [gs:2 * 1], ax

    mov al, 'R'
    mov [gs:2 * 2], ax

    mov al, 'N'
    mov [gs:2 * 3], ax

    mov al, 'E'
    mov [gs:2 * 4], ax

    mov al, 'L'
    mov [gs:2 * 5], ax

    mov al, ' '
    mov [gs:2 * 6], ax

    mov al, '2'
    mov [gs:2 * 7], ax

    mov al, '2'
    mov [gs:2 * 8], ax

    mov al, '3'
    mov [gs:2 * 9], ax

    mov al, '3'
    mov [gs:2 * 10], ax
```

```

mov al, '6'
mov [gs:2 * 11], ax

mov al, '1'
mov [gs:2 * 12], ax

mov al, '7'
mov [gs:2 * 13], ax

mov al, '3'
mov [gs:2 * 14], ax

mov al, '_'
mov [gs:2 * 15], ax

mov al, 'L'
mov [gs:2 * 16], ax

mov al, 'H'
mov [gs:2 * 17], ax

mov al, 'J'
mov [gs:2 * 18], ax
pop eax
ret

```

然后我们统一在文件 `include/asm_utils.h` 中声明所有的汇编函数，这样我们就不用单独地使用 `extern` 来声明了，只需要 `#include "asm_utils.h"` 即可，如下所示。

```

#ifndef ASM_UTILS_H
#define ASM_UTILS_H

extern "C" void asm_hello_world();

#endif

```

修改 `bootloader.asm` 添加进入内核的地址跳转：

```

mov eax, KERNEL_START_SECTOR
mov ebx, KERNEL_START_ADDRESS
mov ecx, KERNEL_SECTOR_COUNT

load_kernel:
    push eax
    push ebx
    call asm_read_hard_disk ; 读取硬盘
    add esp, 8
    inc eax
    add ebx, 512
    loop load_kernel

jmp dword CODE_SELECTOR:KERNEL_START_ADDRESS ; 跳转到kernel

```

```
jmp $ ; 死循环
```

```
asm_read_hard_disk: ;32位函数
```

```
push ebp
mov ebp, esp
```

```
push eax
push ebx
push ecx
push edx
```

```
mov eax, [ebp + 4 * 3] ; 逻辑扇区低16位
```

```
mov edx, 0x1f3
out dx, al ; LBA地址7~0
```

```
inc edx ; 0x1f4
mov al, ah
out dx, al ; LBA地址15~8
```

```
xor eax, eax
inc edx ; 0x1f5
out dx, al ; LBA地址23~16 = 0
```

```
inc edx ; 0x1f6
mov al, ah
and al, 0x0f
or al, 0xe0 ; LBA地址27~24 = 0
out dx, al
```

```
mov edx, 0x1f2
mov al, 1
out dx, al ; 读取1个扇区
```

```
mov edx, 0x1f7 ; 0x1f7
mov al, 0x20 ; 读命令
out dx, al
```

```
; 等待处理其他操作
```

```
.waits:
in al, dx ; dx = 0x1f7
and al, 0x88
cmp al, 0x08
jnz .waits
```

```
; 读取512字节到地址ds:bx
```

```
mov ebx, [ebp + 4 * 2]
mov ecx, 256 ; 每次读取一个字, 2个字节, 因此读取256次即可
mov edx, 0x1f0
```

```
.readw:
in ax, dx
mov [ebx], eax
add ebx, 2
loop .readw
```

```

    pop edx
    pop ecx
    pop ebx
    pop eax
    pop ebp

    ret

pgdt dw 0
    dd GDT_START_ADDRESS

```

注意在32位的保护模式下相应的寄存器要使用32位才能正常发生内核跳转。

使用makefile来批次编译和链接可执行文件

```

ASM_COMPILER = nasm
C_COMPLIER = gcc
CXX_COMPLIER = g++
CXX_COMPLIER_FLAGS = -g -Wall -march=i386 -m32 -nostdlib -fno-builtin -
ffreestanding -fno-pic
LINKER = ld

SRCDIR = ../src
RUNDIR = ../run
BUILDDIR = build
INCLUDE_PATH = ../include

CXX_SOURCE += $(wildcard $(SRCDIR)/kernel/*.cpp)
CXX_OBJ += $(CXX_SOURCE:$(SRCDIR)/kernel/%.cpp=%.o)

ASM_SOURCE += $(wildcard $(SRCDIR)/utils/*.asm)
ASM_OBJ += $(ASM_SOURCE:$(SRCDIR)/utils/%.asm=%.o)

OBJ += $(CXX_OBJ)
OBJ += $(ASM_OBJ)

build : mbr.bin bootloader.bin kernel.bin kernel.o
    qemu-img create $(RUNDIR)/hd.img 10m
    dd if=mbr.bin of=$(RUNDIR)/hd.img bs=512 count=1 seek=0 conv=notrunc
    dd if=bootloader.bin of=$(RUNDIR)/hd.img bs=512 count=5 seek=1 conv=notrunc
    dd if=kernel.bin of=$(RUNDIR)/hd.img bs=512 count=145 seek=6 conv=notrunc
# nasm的include path有一个尾随/

mbr.bin : $(SRCDIR)/boot/mbr.asm
    $(ASM_COMPILER) -o mbr.bin -f bin -I$(INCLUDE_PATH)/ $(SRCDIR)/boot/mbr.asm

bootloader.bin : $(SRCDIR)/boot/bootloader.asm
    $(ASM_COMPILER) -o bootloader.bin -f bin -I$(INCLUDE_PATH)/
$(SRCDIR)/boot/bootloader.asm

entry.obj : $(SRCDIR)/boot/entry.asm
    $(ASM_COMPILER) -o entry.obj -f elf32 $(SRCDIR)/boot/entry.asm

kernel.bin : kernel.o
    objcopy -O binary kernel.o kernel.bin

```



```

kernel.o : entry.obj $(OBJ)
    $(LINKER) -o kernel.o -melf_i386 -N entry.obj $(OBJ) -e enter_kernel -Ttext
0x00020000

$(CXX_OBJ):
    $(CXX_COMPLIER) $(CXX_COMPLIER_FLAGS) -I$(INCLUDE_PATH) -c $(CXX_SOURCE)

asm_utils.o : $(SRCDIR)/utils/asm_utils.asm
    $(ASM_COMPILER) -o asm_utils.o -f elf32 $(SRCDIR)/utils/asm_utils.asm
clean:
    rm -f *.o* *.bin

run:
    qemu-system-i386 -hda $(RUNDIR)/hd.img -serial null -parallel stdio -no-
reboot

debug:
    qemu-system-i386 -S -s -parallel stdio -hda $(RUNDIR)/hd.img -serial null&
    @sleep 1
    gnome-terminal -- "gdb -q -tui -x $(RUNDIR)/gdbinit"

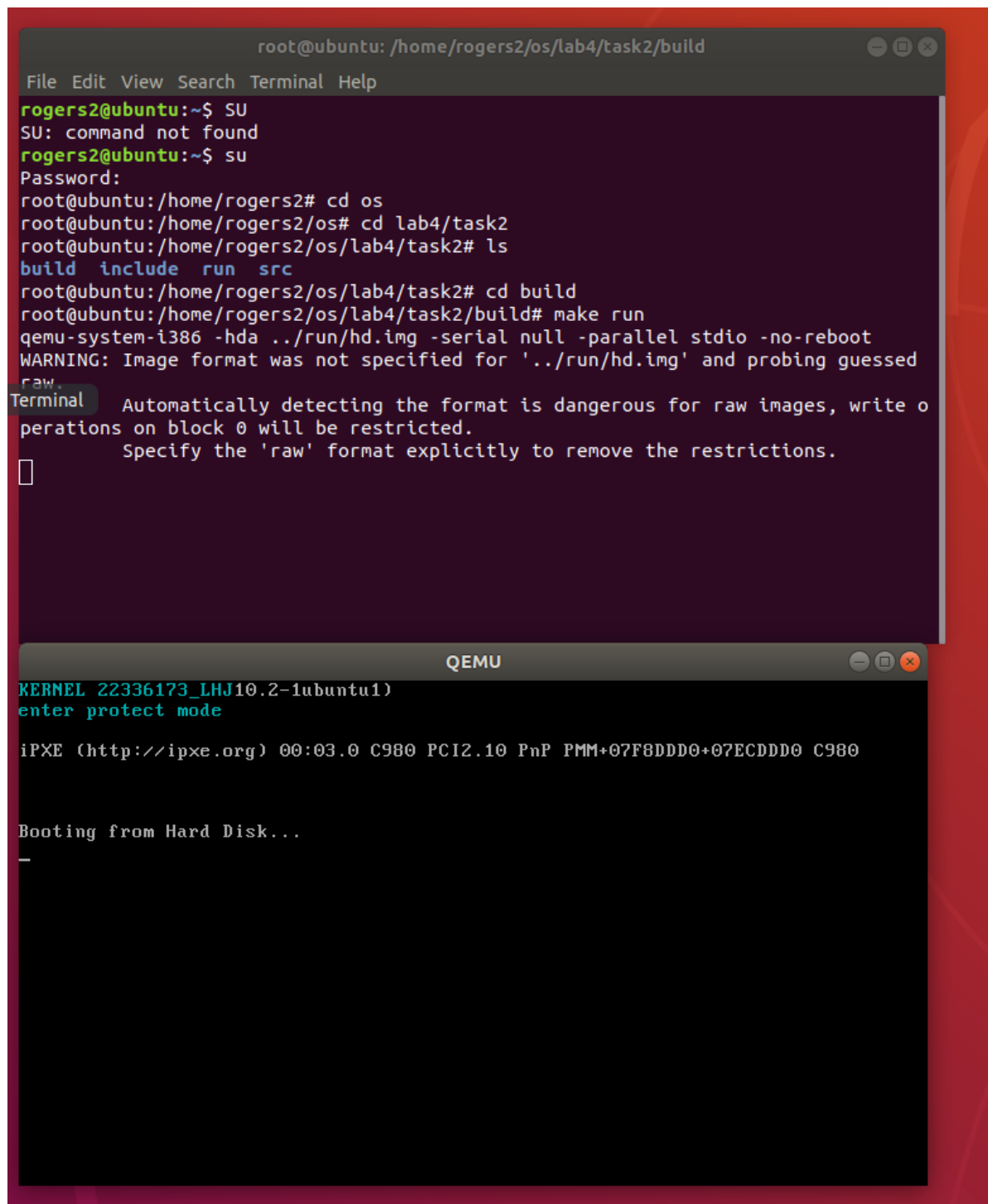
```

注意，如果不想手动在run下生成hd.img的话，**需要在build目录下规定生成hd.img这一步**。这是源文件没有的。

修改汇编代码，输出内核模式下进入的标志：

“KERNEL 22336173_LHJ”

成果展示:



```
root@ubuntu: /home/rogers2/os/lab4/task2/build
File Edit View Search Terminal Help
rogers2@ubuntu:~$ SU
SU: command not found
rogers2@ubuntu:~$ su
Password:
root@ubuntu:/home/rogers2# cd os
root@ubuntu:/home/rogers2/os# cd lab4/task2
root@ubuntu:/home/rogers2/os/lab4/task2# ls
build include run src
root@ubuntu:/home/rogers2/os/lab4/task2# cd build
root@ubuntu:/home/rogers2/os/lab4/task2/build# make run
qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '../run/hd.img' and probing guessed
raw.
Terminal Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
[]

QEMU
KERNEL 22336173_LHJ10.2-1ubuntu1)
enter protect mode

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD0+07ECDDDD0 C980

Booting from Hard Disk...
-
```

#####

----- 实验任务3 -----

任务要求:

实验任务3: 中断的处理: 复现网址中“初始化IDT”部分, 你可以更改默认的中断处理函数为你编写的函数, 然后触发之, 结果截图并说说你是怎么做的。要求: 调用处理函数时输出包含个人学号或姓名信息。

思路分析：

在任务二的基础上，添加了保护模式中断向量的设定和加载，要使用C语言编写函数，但是由于中断向量表寄存器只能用汇编函数lidt指令才能改变，所以要配合汇编函数来混合编程。

中断有两种类型，外部中断和内部中断。外部中断由硬件产生，因此又被称为硬中断。内部中断通过在程序中使用 `int` 指令调用，因此又被称为软中断。为了处理中断，OS需要预先建立中断和中断向量号的对应关系。这里，中断向量号是用来标识不同中断程序的序号。例如，我们可以使用 `int 10h` 来调用10h中断，10h就是中断向量号。

外部中断有屏蔽中断和不可屏蔽中断两种类型，屏蔽中断由INTR引脚产生，通过8259A芯片建立。不可屏蔽中断通过NMI引脚产生，例如除零错误。

内部中断就是程序中通过汇编指令调用的中断，如 `int 10h`，10h（16进制）是中断向量号，调用的10h中断就被称为内部中断。在实模式下，BIOS中集成了一些中断程序，在BIOS加电启动后这些中断程序便被放置在内存中。当我们需要调用某个中断时，我们直接在 `int` 指令中给出中断向量号即可。但是，BIOS内置的中断程序是16位的。所以，在保护模式下这些代码便不再适用。不仅如此，保护模式重新对中断向量号进行编号，也就是说，即使是相同的中断向量号，其在实模式和保护模式中的意义不再相同。

代码分析：

✓ task3

✓ build

M makefile

✓ include

C asm_utils.h

≡ boot C:\Users\rogers\Documents\le

C interrupt.h

C os_constant.h

C os_modules.h

C os_type.h

C setup.h

✓ run

≡ hd.img

✓ src

✓ boot

ASM bootloader.asm

ASM entry.asm

ASM mbr.asm

✓ kernel

C++ interrupt.cpp

C++ setup.cpp

✓ utils

在内核的开始添加中断向量表实现:

asm asm_utils.asm

setup.cpp:

```
#include "asm_utils.h"
#include "interrupt.h"

// 中断管理器
InterruptManager interruptManager;

extern "C" void setup_kernel()
{
    // 声明进入了内核
    asm_hello_world();
    // 中断处理部件
    interruptManager.initialize();

    // 尝试触发除0错误
    int a = 1 / 0;

    // 死循环
    asm_halt();
}
```

interrupt.h

```
#ifndef INTERRUPT_H
#define INTERRUPT_H

#include "os_type.h"

class InterruptManager
{
private:
    // IDT起始地址
    uint32 *IDT;

public:
    InterruptManager();
    // 初始化
    void initialize();
    // 设置中断描述符
    // index 第index个描述符, index=0, 1, ..., 255
    // address 中断处理程序的起始地址
    // DPL 中断描述符的特权级
    void setInterruptDescriptor(uint32 index, uint32 address, byte DPL);
};

#endif
```

interrupt.cpp

```
#include "interrupt.h"
#include "os_type.h"
```

```

#include "os_constant.h"
#include "asm_utils.h"

InterruptManager::InterruptManager()
{
    IDT = nullptr;
}

void InterruptManager::initialize()
{
    IDT = (uint32 *)IDT_START_ADDRESS;
    asm_lidt(IDT_START_ADDRESS, 256 * 8 - 1);

    for (uint i = 0; i < 256; ++i)
    {
        setInterruptDescriptor(i, (uint32)asm_unhandled_interrupt, 0);
    }
}

// 设置中断描述符
// index    第index个描述符, index=0, 1, ..., 255
// address  中断处理程序的起始地址
// DPL      中断描述符的特权级
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);
}

```

汇编函数:

```

[bits 32]

global asm_hello_world
global asm_lidt
global asm_unhandled_interrupt
global asm_halt

ASM_UNHANDLED_INTERRUPT_INFO db '222336173 interrupt happened, enter handle
program'

                                db 0

ASM_IDTR dw 0
          dd 0

; void asm_unhandled_interrupt()
asm_unhandled_interrupt:
    cli
    mov esi, ASM_UNHANDLED_INTERRUPT_INFO
    xor ebx, ebx

```

```

    mov ah, 0x03
.output_information:
    cmp byte[esi], 0
    je .end
    mov al, byte[esi]
    mov word[gs:bx], ax
    inc esi
    add ebx, 2
    jmp .output_information
.end:
    jmp $

; void asm_lidt(uint32 start, uint16 limit)
asm_lidt:
    push ebp
    mov ebp, esp
    push eax

    mov eax, [ebp + 4 * 3]
    mov [ASM_IDTR], ax
    mov eax, [ebp + 4 * 2]
    mov [ASM_IDTR + 2], eax
    lidt [ASM_IDTR]

    pop eax
    pop ebp
    ret

asm_hello_world:
    push eax
    xor eax, eax

    mov ah, 0x03 ;ÇàÉ«
    mov al, 'K'
    mov [gs:2 * 0], ax

    mov al, 'E'
    mov [gs:2 * 1], ax

    mov al, 'R'
    mov [gs:2 * 2], ax

    mov al, 'N'
    mov [gs:2 * 3], ax

    mov al, 'E'
    mov [gs:2 * 4], ax

    mov al, 'L'
    mov [gs:2 * 5], ax

    mov al, ' '
    mov [gs:2 * 6], ax

    mov al, '2'
    mov [gs:2 * 7], ax

```



```

mov al, '2'
mov [gs:2 * 8], ax

mov al, '3'
mov [gs:2 * 9], ax

mov al, '3'
mov [gs:2 * 10], ax

mov al, '6'
mov [gs:2 * 11], ax

mov al, '1'
mov [gs:2 * 12], ax

mov al, '7'
mov [gs:2 * 13], ax

mov al, '3'
mov [gs:2 * 14], ax

mov al, '_'
mov [gs:2 * 15], ax

mov al, 'L'
mov [gs:2 * 16], ax

mov al, 'H'
mov [gs:2 * 17], ax

mov al, 'J'
mov [gs:2 * 18], ax
pop eax
ret

asm_halt:
    jmp $

```

makefile 不需要改变：因为我们之前在makefile中写了可以自动找到项目文件夹下的所有 .cpp 文件的语句，因此在本节中，我们虽然增加了 `src/kernel/interrupt.cpp` 文件，我们也不需要修改makefile也可以编译

结果展示：

由于在内核函数中计算了0作为除数的计算所以发生了除以0的中断，调用了中断处理函数。

```
root@ubuntu: /home/rogers2/os/lab4/task3/build
File Edit View Search Terminal Help
root@ubuntu:/home/rogers2/os/lab4/task2# ls
build include run src
root@ubuntu:/home/rogers2/os/lab4/task2# cd build
root@ubuntu:/home/rogers2/os/lab4/task2/build# make run
qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '../run/hd.img' and probing guessed
raw.
    Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
    Specify the 'raw' format explicitly to remove the restrictions.
root@ubuntu:/home/rogers2/os/lab4/task2/build# cd ..
root@ubuntu:/home/rogers2/os/lab4/task2# cd .
root@ubuntu:/home/rogers2/os/lab4/task2# cd task3
bash: cd: task3: No such file or directory
root@ubuntu:/home/rogers2/os/lab4/task2# cd ..
root@ubuntu:/home/rogers2/os/lab4# cd task3/build
root@ubuntu:/home/rogers2/os/lab4/task3/build# make run
qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '../run/hd.img' and probing guessed
raw.
    Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
    Specify the 'raw' format explicitly to remove the restrictions.

```

```
QEMU
222336173 interrupt happened, enter handle program
enter protect mode

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980

Booting from Hard Disk...
-
```

----- 实验任务4 -----

任务要求:

复现网址中“8259A编程——实时钟中断的处理”部分，要求：仿照该章节中使用C语言来实现时钟中断的例子，利用 C/C++、InterruptManager、STDIO 和你自己封装的类来实现 你的时钟中断处理过程(例如，通过时钟中断，你可以在屏幕的第一行实现一个跑马灯。跑 马灯显示自己学号和英文名，即类似于LED屏幕显示的效果)，保存结果截图并说说你的思路 和做法。

思路分析：

在interrupt.cpp中编写中断处理函数，在每一次中断发生的时候，显示自己的学号名字和中断次数，维护跑马灯的位置和跑马灯颜色的变化；

代码分析：

其他函数和实验资料给出的一致，只需要在中断处理函数中改写就可以

interrupt.cpp

```
int time = 0;
int count = 0; //位置
uint8 color =0; //颜色
extern "C" void c_time_interrupt_handler()
{
    // 清空屏幕
    for (int i = 0; i < 80; ++i)
    {
        stdio.print(0, i, ' ', 0x07);
    }

    // 输出中断发生的次数
    ++times;
    ++count;
    char str[] = "22336173 LHJ maked the interrupt happend: ";
    char number[10];
    int temp = times;

    // 将数字转换为字符串表示
    for(int i = 0; i < 10; ++i ) {
        if(temp) {
            number[i] = temp % 10 + '0';
        } else {
            number[i] = '0';
        }
        temp /= 10;
    }

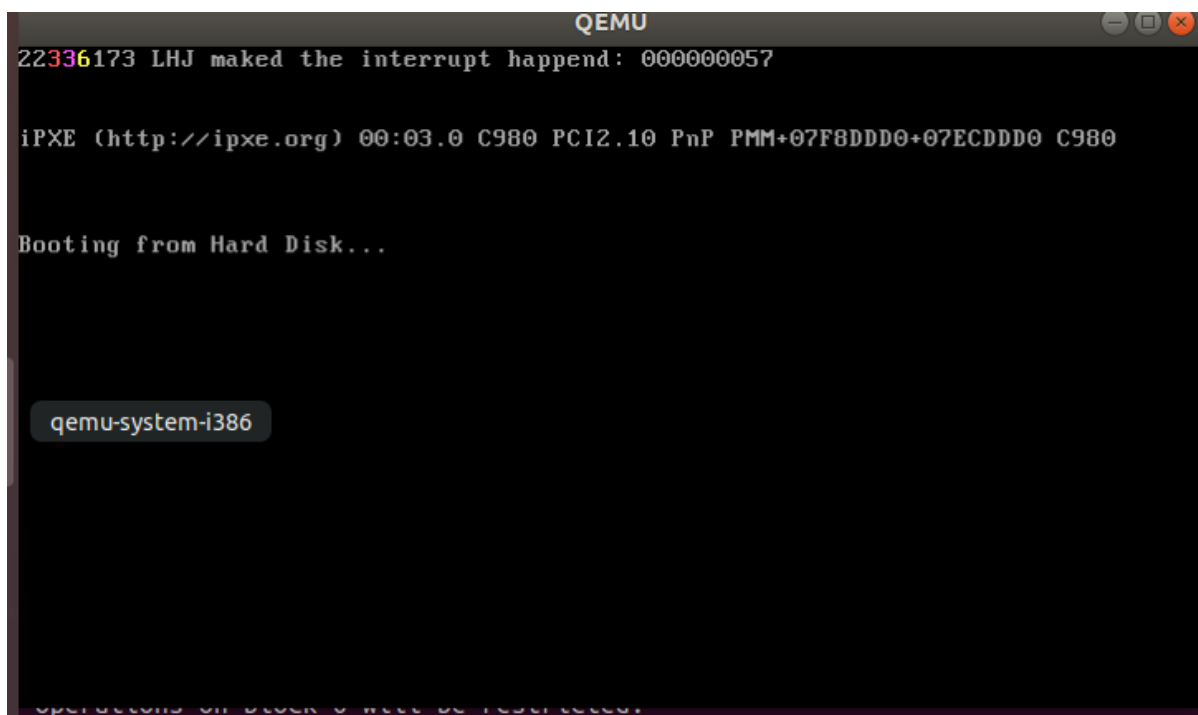
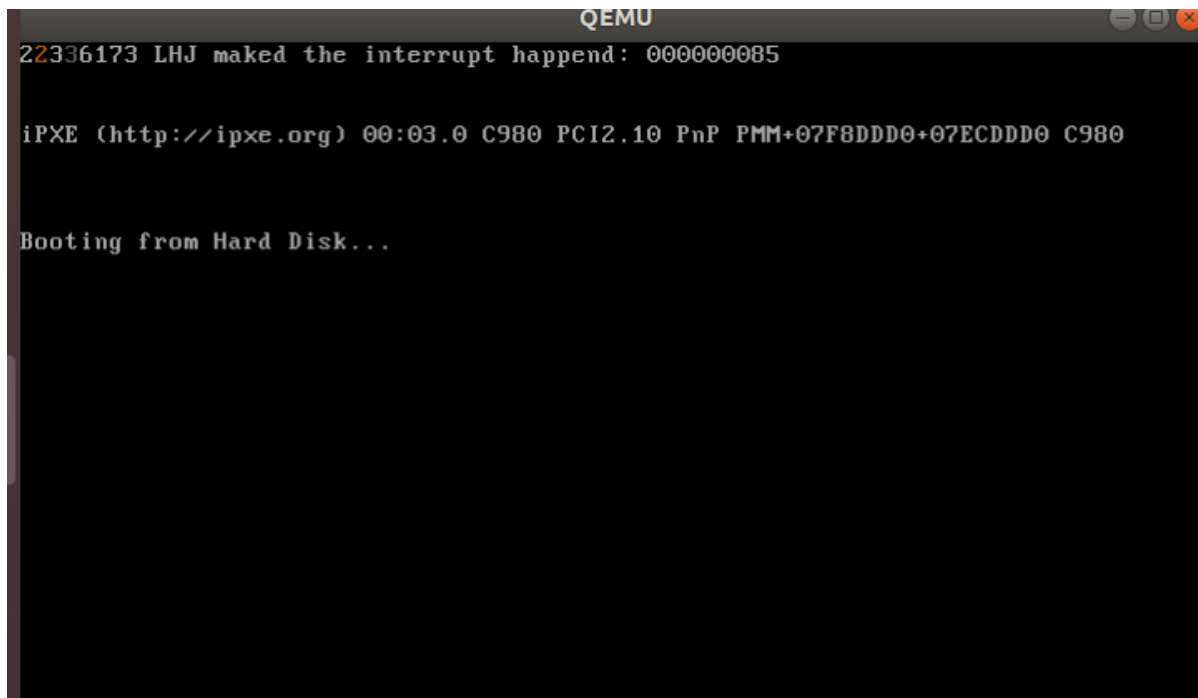
    // 移动光标到(0,0)输出字符
    stdio.moveCursor(0);
    for(int i = 0; str[i]; ++i ) {
        //跑马灯设计
        if (i == count){
            stdio.print(str[i], color); //0x0e为黄色 //一次显示三个字符
            stdio.print(str[i+1], ++color %= 16); //
            stdio.print(str[i+2], ++color %= 16); //color为递增的颜色
            i += 2;
        }
        else{
            stdio.print(str[i]);
        }
    }

    ++count; //跑马灯
```

```
count %= 37;

// 输出中断发生的次数
for( int i = 9; i > 0; --i ) {
    stdio.print(number[i]);
}
}
```

成果展示:



QEMU

22336173 LHJ maked the interrupt happend: 000000034_

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980

Booting from Hard Disk...