

操作系统原理实验报告

**实验名称:** 实验四 中断

**授课教师：** 张青

**学生姓名:** 曾慧蕾

**学生学号:** 21307358

1. **实验要求**

* **Assignment 1 混合编程的基本思路**

复现Example 1，结合具体的代码说明C代码调用汇编函数的语法和汇编代码调用C函数的语法。例如，结合代码说明global、extern关键字的作用，为什么C++的函数前需要加上extern "C"等， 结果截图并说说你是怎么做的。同时，学习make的使用，并用make来构建Example 1，结果截图并说说你是怎么做的。

* **Assignment 2 使用C/C++来编写内核**

复现Example 2，在进入setup\_kernel函数后，将输出 Hello World 改为输出你的学号，结果截图并说说你是怎么做的。

* **Assignment 3 中断的处理**

复现Example 3，你可以更改Example中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。

* **Assignment 4 时钟中断**

复现Example 4，仿照Example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，结果截图并说说你是怎么做的。注意，不可以使用纯汇编的方式来实现。(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。)

1. **实验过程**
2. **混合编程的基本思路：**
   1. **原理**

**global：**

global 为汇编编译器中的关键字，其作用是使得链接器ld能够识别symbol，声明symbol是全局可见的。

在C/C++调用汇编函数之前，我们先需要在汇编代码中将函数声明为global。例如我们需要调用汇编函数function\_from\_asm，那么我们首先需要在汇编代码中将其声明为global。如：

*global function\_from\_asm*

**extern：**

extern放在变量和函数声明之前,表示该变量或者函数在别的文件中已经定义,提示编译器在编译时要从别的文件中寻找。除此之外,extern还可以用来进行链接指定。

当我们需要在汇编代码中使用C的函数function\_from\_C时，我们需要在汇编代码中声明这个函数来自于外部。

*extern function\_from\_C*

声明后便可直接使用，例如。

*call function\_from\_C*

**extern “C”**

如果我们需要在汇编代码中使用来自C++的函数function\_from\_CPP时，我们需要现在C++代码的函数定义前加上extern "C"。因为C++支持函数重载，为了区别同名的重载函数，C++在编译时会进行名字修饰。也就是说，function\_from\_CPP编译后的标号不再是function\_from\_CPP，而是要带上额外的信息。而C代码编译后的标号还是原来的函数名。因此，extern "C"目的是告诉编译器按C代码的规则编译，不进行名字修饰。例如，我们首先需要在C++代码中声明如下。

*extern "C" void functionFromCpp();*

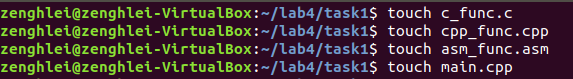
同样地，在汇编代码中声明这个函数即可。

*extern function\_from\_CPP*

如果不加extern "C"，那么我们在汇编代码中声明的标号就不是function\_from\_CPP，而是function\_from\_CPP经过名字修饰后的标号，这将会变得非常麻烦。

**1.2实验过程**

我们创建四个文件：



我们首先在文件c\_func.c中定义C函数function\_from\_C。

#include <stdio.h>

void function\_from\_C() {

   printf("This is a function from C.\n");

}

然后在文件cpp\_func.cpp中定义C++函数function\_from\_CPP。

#include <iostream>

extern "C" void function\_from\_CPP() {

   std::cout << "This is a function from C++." << std::endl;

}

接着在文件asm\_func.asm中定义汇编函数function\_from\_asm，在function\_from\_asm中调用function\_from\_C和function\_from\_CPP。

[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

后在文件main.cpp中调用汇编函数function\_from\_asm。

#include <iostream>

extern "C" void function\_from\_asm();

int main() {

   std::cout << "Call function from assembly." << std::endl;

   function\_from\_asm();

   std::cout << "Done." << std::endl;}

我们首先将这4个文件统一编译成可重定位文件即.o文件，然后将这些.o文件链接成一个可执行文件，编译命令分别如下。

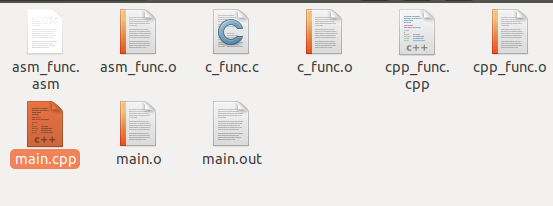
gcc -o c\_func.o -m32 -c c\_func.c

g++ -o cpp\_func.o -m32 -c cpp\_func.cpp

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

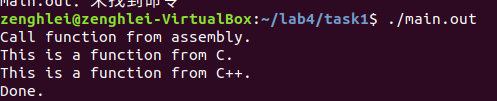
nasm -o asm\_func.o -f elf32 asm\_func.asm

g++ -o main.out main.o c\_func.o cpp\_func.o asm\_func.o -m32

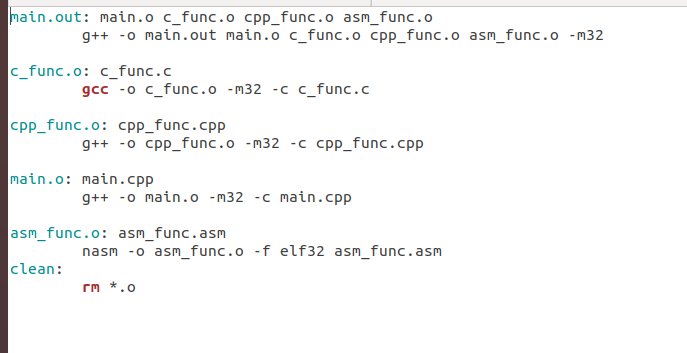


其中，-f elf32指定了nasm编译生成的文件格式是ELF32文件格式，ELF文件格式也就是Linux下的.o文件的文件格式。

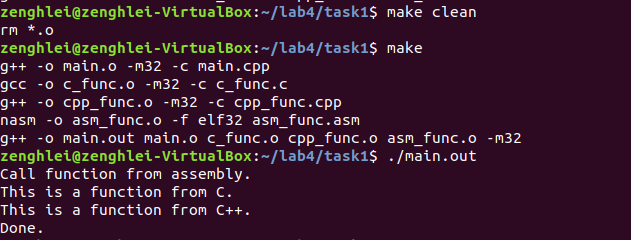
最后我们执行main.out，输出如下结果。



make就是在文件里面补全要用的指令，编写完成后在终端调用一下make就会自动完成make中的命令



Make的使用：



1. **使用C/C++来编写内核：**

**2.1 原理**

我们在bootloader中加载操作系统内核到地址0x20000，然后跳转到0x20000。内核接管控制权后，输出“21307358ZHL”。

假设我们实现的内核很小，因此下面我们约定内核的大小是200个扇区，起始地址是0x20000，内核存放在硬盘的起始位置是第6个扇区。bootloader在进入保护模式后，从硬盘的第6个扇区中加载200个扇区到内存起始地址0x20000处，然后跳转执行。

**2.2 实验过程**

在上次实验中的bootloader的基础上，我们在bootloader的最后加上读取内核的代码，如下所示。

... ; 进入保护模式并初始化的代码

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(memory,block)

; 加载逻辑扇区号为block的扇区到内存地址memory

... ;省略

常量的定义放置boot.inc中，其新增的内容如下。

; \_\_\_\_\_\_\_\_\_\_kernel\_\_\_\_\_\_\_\_\_

KERNEL\_START\_SECTOR equ 6

KERNEL\_SECTOR\_COUNT equ 200

KERNEL\_START\_ADDRESS equ 0x20000

*接下来编写内核。首先在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 ;青色

   mov al, '2'

   mov [gs:2 \* 0], ax

   mov al, '1'

   mov [gs:2 \* 1], ax

   mov al, '3'

   mov [gs:2 \* 2], ax

   mov al, '0'

   mov [gs:2 \* 3], ax

   mov al, '7'

   mov [gs:2 \* 4], ax

   mov al, ' 3'

   mov [gs:2 \* 5], ax

   mov al, '5'

   mov [gs:2 \* 6], ax

   mov al, '8'

   mov [gs:2 \* 7], ax

   mov al, ' '

   mov [gs:2 \* 8], ax

   mov al, 'Z'

   mov [gs:2 \* 9], ax

   mov al, 'H'

   mov [gs:2 \* 10], ax

mov al, 'L'

    mov [gs:2 \* 11], ax

   pop eax

   ret

然后我们统一在文件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

*然后我们在build文件夹下开始编译，我们首先编译MBR、bootloader。*

nasm -o mbr.bin -f bin -I../include/ ../src/boot/mbr.asm

nasm -o bootloader.bin -f bin -I../include/ ../src/boot/bootloader.asm

其中，-I参数指定了头文件路径，-f指定了生成的文件格式是二进制的文件。

接着，我们编译内核的代码。我们的方案是将所有的代码（C/C++，汇编代码）都同一编译成可重定位文件，然后再链接成一个可执行文件。

我们首先编译entry.asm和asm\_utils.asm。

nasm -o entry.obj -f elf32 ../src/boot/entry.asm

nasm -o asm\_utils.o -f elf32 ../src/utils/asm\_utils.asm

回忆以下，在Linux下的可重定位文件的格式是ELF文件格式。加上我们是32位保护模式，-f参数指定的生成文件格式是elf32，而不再是bin。

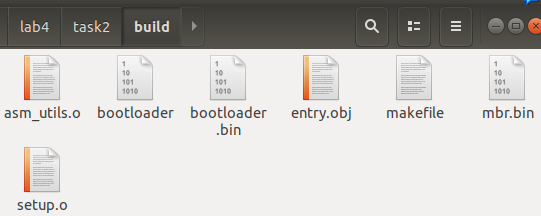
接着，我们编译setup.cpp。

g++ -g -Wall -march=i386 -m32 -nostdlib -fno-builtin -ffreestanding -fno-pic -I../include -c ../src/kernel/setup.cpp

上面的参数有点多，我们逐一来看。

* -O0告诉编译器不开启编译优化。
* -Wall告诉编译器显示所有编译器警告信息
* -march=i386告诉编译器生成i386处理器下的.o文件格式。
* -m32告诉编译器生成32位的二进制文件。
* -nostdlib -fno-builtin -ffreestanding -fno-pic是告诉编译器不要包含C的任何标准库。
* -g表示向生成的文件中加入debug信息供gdb使用。
* -I指定了代码需要的头文件的目录。
* -c表示生成可重定位文件。

Build文件夹下的编译结果：



最后我们链接生成的可重定位文件为两个文件：只包含代码的文件*kernel.bin，可执行文件kernel.o。*

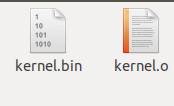
ld -o kernel.o -melf\_i386 -N entry.obj setup.o asm\_utils.o -e enter\_kernel -Ttext 0x00020000

ld -o kernel.bin -melf\_i386 -N entry.obj setup.o asm\_utils.o -e enter\_kernel -Ttext 0x00020000 --oformat binary

这里面同样涉及很多参数，我们逐一来看。

* -m参数指定模拟器为i386。
* -N参数告诉链接器不要进行页对齐。
* -Ttext指定标号的起始地址。
* -e参数指定程序进入点。
* --oformat指定输出文件格式。

链接结果如下所示：



链接后我们使用dd命令将mbr.bin bootloader.bin kernel.bin写入硬盘即可，如下所示。

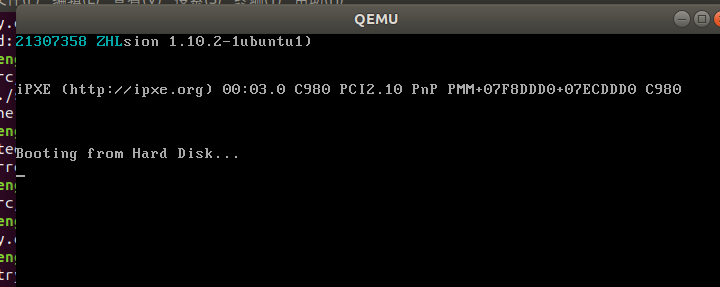
dd if=mbr.bin of=../run/hd.img bs=512 count=1 seek=0 conv=notrunc

dd if=bootloader.bin of=../run/hd.img bs=512 count=5 seek=1 conv=notrunc

dd if=kernel.bin of=../run/hd.img bs=512 count=200 seek=6 conv=notrunc

在run目录下，启动后的效果如下。

qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot



对于上述过程，我们也可以采用makefile来实现。我们可以使用makefile的命令自动帮我们找到.c，.cpp文件，然后编译生成.o文件。然后我们又可以使用makefile找到所有生成的.o文件，使用ld链接生成二进制文件。这样做的好处是当我们新增一个.c或.cpp文件后，我们几乎不需要修改makefile，大大简化了编译过程。makefile如下。

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

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 : entry.obj $(OBJ)

$(LINKER) -o kernel.bin -melf\_i386 -N entry.obj $(OBJ) -e enter\_kernel -Ttext 0x00020000 --oformat binary

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

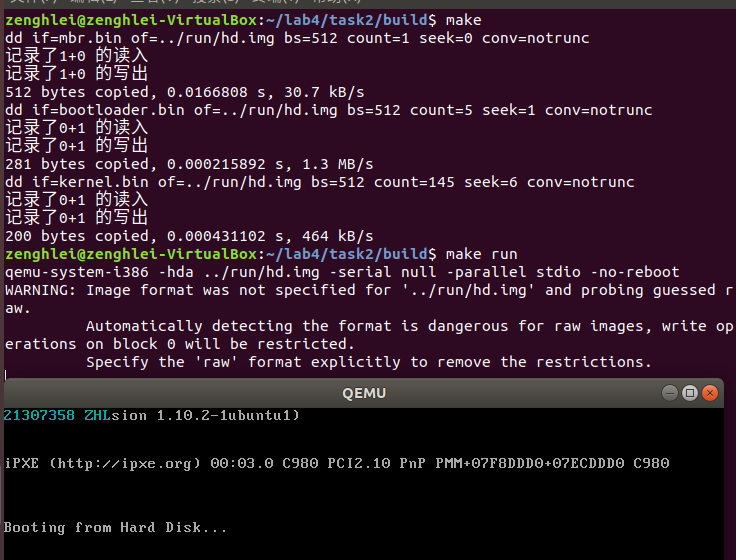
gnome-terminal -e "gdb -q -tui -x $(RUNDIR)/gdbinit"

然后使用如下命令即可编译、运行。

make

make run

运行结果与上一图一致。



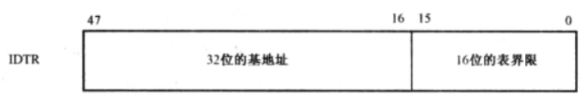
1. **中断的处理**

**3.1 原理**

要初始化内核，我们要做的事情只有三件：

* 确定IDT的地址。
* 定义中断默认处理函数。
* 初始化256个中断描述符

为了使CPU能够找到IDT中的中断处理函数，我们需要将IDT的信息放置到寄存器IDTR中。当中断发生时，CPU会自动到IDTR中找到IDT的地址，然后根据中断向量号在IDT找到对应的中断描述符，最后跳转到中断描述符对应的函数中进行处理，IDTR的结构如下。



由于我们只有256个中断描述符，每个中断描述符的大小均为8字节，因此我们有



此时，IDTR的32位基地址是0x8880，表界限是2047。

确定了IDT的基地址和表界限后，我们就可以初始化IDTR了。IDTR的初始化需要用到指令lidt，其用法如下。

lidt [tag]

将IDT的信息放入到IDTR后，我们就可以插入256个默认的中断处理描述符到IDT中。在实验中，对于中断描述符，有几个值是定值

* P=1表示存在。
* D=1表示32位代码。
* DPL=0表示特权级0.
* 代码段选择子等于bootloader中的代码段选择子，也就是寻址4GB空间的代码段选择子

从目前来看，不同的中断描述符间变化的只是中断处理程序在目标代码段中的偏移。由于我们的程序运行在平坦模式下，也就是段起始地址从内存地址0开始，长度为4GB。此时，函数名就是中断处理程序在目标代码段中的偏移。

**3.2 实验过程**

Assignment3的代码将会在assignment2的代码的基础上进行修改

为了能够抽象地描述中断处理模块，我们不妨定义一个类，称为中断管理器InterruptManager，其定义放置在include/interrupt.h中，如下所示。

#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

我们在include/os\_type.h定义了基本的数据类型的别名，如下所示。

#ifndef OS\_TYPE\_H

#define OS\_TYPE\_H

// 类型定义

typedef unsigned char byte;

typedef unsigned char uint8;

typedef unsigned short uint16;

typedef unsigned short word;

typedef unsigned int uint32;

typedef unsigned int uint;

typedef unsigned int dword;

#endif

在使用中断之前，我们首先需要初始化IDT，承担起这项工作的函数是InterruptManager::initialize，如下所示。

void InterruptManager::initialize()

{

   // 初始化IDT

   IDT = (uint32 \*)IDT\_START\_ADDRESS;

   asm\_lidt(IDT\_START\_ADDRESS, 256 \* 8 - 1);

   for (uint i = 0; i < 256; ++i)

  {

       setInterruptDescriptor(i, (uint32)asm\_interrupt\_empty\_handler, 0);

  }

}

对于初始化IDTR的部分，需要知道，lidt实际上是将以tag为起始地址的48字节放入到寄存器IDTR中。由于我们打算在C代码中初始化IDT，而C语言的语法并未提供lidt语句。因此我们需要在汇编代码中实现能够将IDT的信息放入到IDTR的函数asm\_lidt，代码放置在utils/asm\_utils.asm中，如下所示。

; 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\_IDTR dw 0

     dd 0

我们将段描述符的设置定义在函数InterruptManager::setInterruptDescriptor中，如下所示。

// 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);

}

其中，IDT是中断描述符表的起始地址指针，实际上我们可以认为中断描述符表就是一个数组。在InterruptManager中，我们将变量IDT视作是一个uint32类型的数组。由于每个中断描述符的大小是两个uint32，第index个中断描述符是IDT[2 \* index],IDT[2 \* index + 1]。

接下来，我们定义一个默认的中断处理函数是asm\_interrupt\_empty\_handler，放置在src/utils/asm\_utils.asm中，如下所示。

ASM\_UNHANDLED\_INTERRUPT\_INFO db 'Unhandled interrupt happened, halt...'

                            db 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 $

这一部分主要进行的是：首先关中断，然后输出提示字符串，最后做死循环。

在InterruptManager::initialize最后，我们调用setInterruptDescriptor放入256个默认的中断描述符即可，这256个默认的中断描述符对应的中断处理函数是asm\_unhandled\_interrupt。

for (uint i = 0; i < 256; ++i)

{

setInterruptDescriptor(i, (uint32)asm\_unhandled\_interrupt, 0);

}

最后，我们在函数src/kernel/setup\_kernel.cpp中定义并初始化中断处理器。注意，我们只会定义一个InterruptManager的实例，因为中断管理器有且只有一个。

... // 头文件的包含

// 中断管理器

InterruptManager interruptManager;

extern "C" void setup\_kernel()

{

   // 中断处理部件

   interruptManager.initialize();

   // 死循环

   asm\_halt();

}

然后我们在include/os\_modules.h中声明这个实例，以便在其他cpp文件中使用。

#ifndef OS\_MODULES\_H

#define OS\_MODULES\_H

#include "interrupt.h"

extern InterruptManager interruptManager;

#endif

最后，我们将一些常量统一定义在文件include/os\_constant.h下。

#ifndef OS\_CONSTANT\_H

#define OS\_CONSTANT\_H

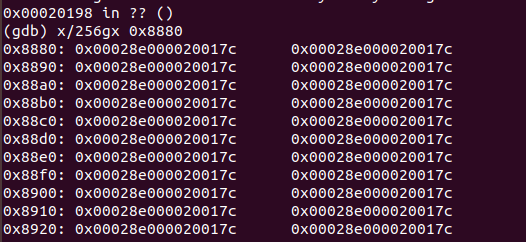
#define IDT\_START\_ADDRESS 0x8880

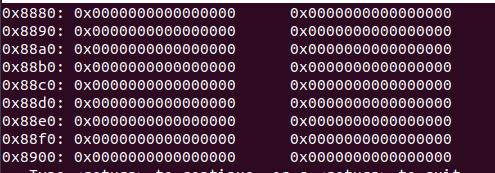
#define CODE\_SELECTOR 0x20

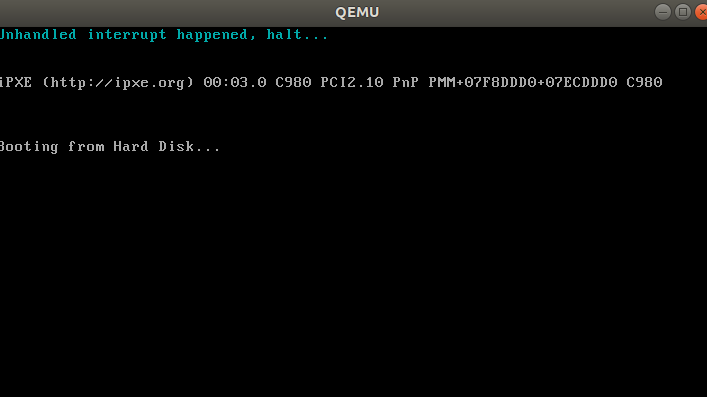
#endif

由于我们之前在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assignment3中，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。从这一点上来说，使用makefile编译的便利性便能很好地体现出来了。直接使用make编译与make debug来检查

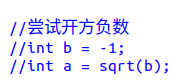
在使用make debug并检查断点的放入的默认中断描述符后发现，输出结果符合预期。



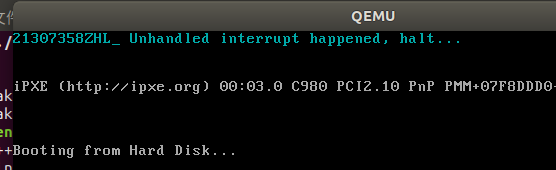




在使用自己的中断函数时，我选择用开方负数这一中断来作为函数，具体如下：



在触发开方负数异常后中断仍可以被正常调用（这里换了自己的输出函数）



1. **时钟中断**

**4.1 实验过程**

Assignment4是对8529A芯片进行编程，并利用8259A芯片对其产生的时钟中断进行处理，目标是用一个函数在第一行显示目前中断发生的次数。

为中断控制器InterruptManager加入如下成员变量和函数。

class InterruptManager

{

private:

   uint32 \*IDT;              // IDT起始地址

   uint32 IRQ0\_8259A\_MASTER; // 主片中断起始向量号

   uint32 IRQ0\_8259A\_SLAVE;  // 从片中断起始向量号

public:

   InterruptManager();

   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);

   // 开启时钟中断

   void enableTimeInterrupt();

   // 禁止时钟中断

   void disableTimeInterrupt();

   // 设置时钟中断处理函数

   void setTimeInterrupt(void \*handler);

private:

   // 初始化8259A芯片

   void initialize8259A();

};

在使用8259A芯片之前，我们首先要对其初始化，初始化的代码放置在成员函数initialize8259A中，如下所示。

void InterruptManager::initialize8259A()

{

   // ICW 1

   asm\_out\_port(0x20, 0x11);

   asm\_out\_port(0xa0, 0x11);

   // ICW 2

   IRQ0\_8259A\_MASTER = 0x20;

   IRQ0\_8259A\_SLAVE = 0x28;

   asm\_out\_port(0x21, IRQ0\_8259A\_MASTER);

   asm\_out\_port(0xa1, IRQ0\_8259A\_SLAVE);

   // ICW 3

   asm\_out\_port(0x21, 4);

   asm\_out\_port(0xa1, 2);

   // ICW 4

   asm\_out\_port(0x21, 1);

   asm\_out\_port(0xa1, 1);

   // OCW 1 屏蔽主片所有中断，但主片的IRQ2需要开启

   asm\_out\_port(0x21, 0xfb);

   // OCW 1 屏蔽从片所有中断

   asm\_out\_port(0xa1, 0xff);

}

初始化8259A芯片的过程是通过设置一系列的ICW字来完成的。由于我们并未建立处理8259A中断的任何函数，因此在初始化的最后，我们需要屏蔽主片和从片的所有中断。

asm\_out\_port是对out指令的封装，放在asm\_utils.asm中，如下所示。

; void asm\_out\_port(uint16 port, uint8 value)

asm\_out\_port:

   push ebp

   mov ebp, esp

   push edx

   push eax

   mov edx, [ebp + 4 \* 2] ; port

   mov eax, [ebp + 4 \* 3] ; value

   out dx, al

   pop eax

   pop edx

   pop ebp

   ret

我们首先编写中断处理的函数。

此时，我们需要对屏幕进行输出，之前我们只是单纯地往显存地址上赋值来显示字符。但是这样做并不太方便，我们希望能够像printf和putchar这样的函数来调用。因此，我们下面简单封装一个能够处理屏幕输出的类STDIO，声明放置在文件include/stdio.h中，如下所示。

#ifndef STDIO\_H

#define STDIO\_H

#include "os\_type.h"

class STDIO

{

private:

   uint8 \*screen;

public:

   STDIO();

   // 初始化函数

   void initialize();

   // 打印字符c，颜色color到位置(x,y)

   void print(uint x, uint y, uint8 c, uint8 color);

   // 打印字符c，颜色color到光标位置

   void print(uint8 c, uint8 color);

   // 打印字符c，颜色默认到光标位置

   void print(uint8 c);

   // 移动光标到一维位置

   void moveCursor(uint position);

   // 移动光标到二维位置

   void moveCursor(uint x, uint y);

   // 获取光标位置

   uint getCursor();

public:

   // 滚屏

   void rollUp();

};

#endif

下面我们看看如何处理光标，光标就是屏幕上一直在闪烁的横杠。屏幕的像素为25\*80，所以光标的位置从上到下，从左到右依次编号为0-1999，用16位表示。与光标读写相关的端口为0x3d4和0x3d5，在对光标读写之前，我们需要向端口0x3d4写入数据，表明我们操作的是光标的低8位还是高8位。写入0x0e，表示操作的是高8位，写入0x0f表示操作的是低8位。如果我们需要需要读取光标，那么我们从0x3d5从读取数据；如果我们需要更改光标的位置，那么我们将光标的位置写入0x3d5。如下所示。

void STDIO::moveCursor(uint position)

{

   if (position >= 80 \* 25)

  {

       return;

  }

   uint8 temp;

   // 处理高8位

   temp = (position >> 8) & 0xff;

   asm\_out\_port(0x3d4, 0x0e);

   asm\_out\_port(0x3d5, temp);

   // 处理低8位

   temp = position & 0xff;

   asm\_out\_port(0x3d4, 0x0f);

   asm\_out\_port(0x3d5, temp);

}

uint STDIO::getCursor()

{

   uint pos;

   uint8 temp;

   pos = 0;

   temp = 0;

   // 处理高8位

   asm\_out\_port(0x3d4, 0x0e);

   asm\_in\_port(0x3d5, &temp);

   pos = ((uint)temp) << 8;

   // 处理低8位

   asm\_out\_port(0x3d4, 0x0f);

   asm\_in\_port(0x3d5, &temp);

   pos = pos | ((uint)temp);

   return pos;

}

其中，asm\_in\_port是对in指令的封装，代码放置在asm\_utils.asm中，如下所示

; void asm\_in\_port(uint16 port, uint8 \*value)

asm\_in\_port:

   push ebp

   mov ebp, esp

   push edx

   push eax

   push ebx

   xor eax, eax

   mov edx, [ebp + 4 \* 2] ; port

   mov ebx, [ebp + 4 \* 3] ; \*value

   in al, dx

   mov [ebx], al

   pop ebx

   pop eax

   pop edx

   pop ebp

   ret

在STDIO::print的实现中，我们向光标处写入了字符并移动光标到下一个位置。特别地，如果过光标超出了屏幕的范围，即字符占满了整个屏幕，我们需要向上滚屏，然后将光标放在(24,0)处。滚屏实际上就是将第2行的字符放到第1行，第3行的字符放到第2行，以此类推，最后第24行的字符放到了第23行，然后第24行清空，光标放在第24行的起始位置。

实现滚屏的函数是STDIO::rollUp，如下所示。

void STDIO::rollUp()

{

   uint length;

   length = 25 \* 80;

   for (uint i = 80; i < length; ++i)

  {

       screen[2 \* (i - 80)] = screen[2 \* i];

       screen[2 \* (i - 80) + 1] = screen[2 \* i + 1];

  }

   for (uint i = 24 \* 80; i < length; ++i)

  {

       screen[2 \* i] = ' ';

       screen[2 \* i + 1] = 0x07;

  }

}

接下来，我们定义中断处理函数c\_time\_interrupt\_handler。由于我们需要显示中断发生的次数，我们需要在src/kernel/interrupt.cpp中定义一个全局变量来充当计数变量，如下所示。

int times = 0;

中断处理函数c\_time\_interrupt\_handler如下所示。

// 中断处理函数

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;

   char str[] = "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 ) {

       stdio.print(str[i]);

  }

   // 输出中断发生的次数

   for( int i = 9; i > 0; --i ) {

       stdio.print(number[i]);

  }

}

一个完整的时钟中断处理函数如下所示，代码保存在asm\_utils.asm中。

asm\_time\_interrupt\_handler:

   pushad

   nop ; 否则断点打不上去

  ; 发送EOI消息，否则下一次中断不发生

   mov al, 0x20

   out 0x20, al

   out 0xa0, al

   call c\_time\_interrupt\_handler

   popad

   iret

其中，pushad指令是将EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI依次入栈，popad则相反。注意，对于8259A芯片产生的中断，我们需要在中断返回前发送EOI消息。否则，8259A不会产生下一次中断。

编写好了中断处理函数后，我们就可以设置时钟中断的中断描述符，也就是主片IRQ0中断对应的描述符，如下所示。

void InterruptManager::setTimeInterrupt(void \*handler)

{

   setInterruptDescriptor(IRQ0\_8259A\_MASTER, (uint32)handler, 0);

}

然后我们封装一下开启和关闭时钟中断的函数。关于8259A上的中断开启情况，我们可以通过读取OCW1来得知；如果要修改8259A上的中断开启情况，我们就需要先读取再写入对应的OCW1

void InterruptManager::enableTimeInterrupt()

{

   uint8 value;

   // 读入主片OCW

   asm\_in\_port(0x21, &value);

   // 开启主片时钟中断，置0开启

   value = value & 0xfe;

   asm\_out\_port(0x21, value);

}

void InterruptManager::disableTimeInterrupt()

{

   uint8 value;

   asm\_in\_port(0x21, &value);

   // 关闭时钟中断，置1关闭

   value = value | 0x01;

   asm\_out\_port(0x21, value);

}

最后，我们在setup\_kernel中定义STDIO的实例stdio，最后初始化内核的组件，然后开启时钟中断和开中断。

extern "C" void setup\_kernel()

{

   // 中断处理部件

   interruptManager.initialize();

   // 屏幕IO处理部件

   stdio.initialize();

   interruptManager.enableTimeInterrupt();

   interruptManager.setTimeInterrupt((void \*)asm\_time\_interrupt\_handler);

   asm\_enable\_interrupt();

   asm\_halt();

}

我们在include/os\_modules.h声明这个实例。

#ifndef OS\_MODULES\_H

#define OS\_MODULES\_H

#include "interrupt.h"

extern InterruptManager interruptManager;

extern STDIO stdio;

#endif

开中断需要使用sti指令，如果不开中断，那么CPU不会响应可屏蔽中断。也就是说，即使8259A芯片发生了时钟中断，CPU也不会处理。开中断指令被封装在函数asm\_enable\_interrupt中，如下所示。

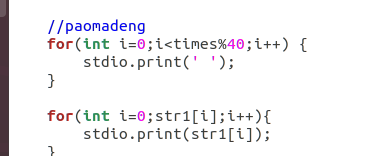
; void asm\_enable\_interrupt()

asm\_enable\_interrupt:

   sti

   ret

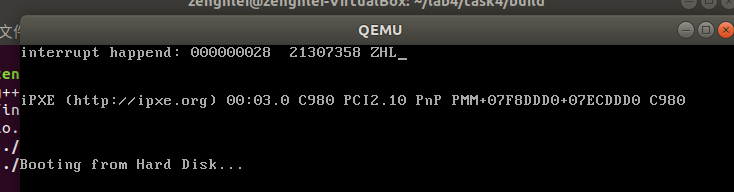
为了输出学号，我们在中断函数中另外添加：

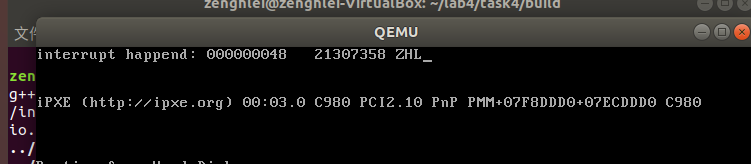


根据输出空格的不同，以及在空格后面输出学号，即可实现跑马灯的效果

在make和make run之后，在qemu的运行结果如下所示：







**3. 总结**

此次实验学习了如何用c/c++来混合编程以及保护模式下的中断处理，使用makefile来简化实验操作，另外了解了IDT的机制及其初始化和8259A芯片的运作原理和编程方法，在通过四个任务的实践过后对本次实验了解到的内容有了更深入的认识。