

**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：中断

专业名称：计算机科学与技术

学生姓名：

学生学号：

实验地点：实验中心B202

实验时间：2025.4.9

**Section 1 实验概述**

本次实验中我们会了解C/C++代码通过预编译、编译、汇编和链接生成最终的可执行文件。同时为了更加有条理地管理操作系统的代码，学习使用了一种C/C++项目管理方案来管理我们的代码。同时学习C/C++和汇编混合编程方法，即如何在C/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/C++编写的函数。最后学习保护模式下的中断处理机制和可编程中断部件8259A芯片的知识后，利用实时钟中断处理函数来编写利用中断实现的混合编程的功能。

完成本次实验后，我们将会学习并掌握使用C/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。

**Section 2 预备知识与实验环境**

* 预备知识：x86汇编语言、IA-32处理器体系结构、字符显示的原理、C/C++程序设计
* 实验环境：
  + 虚拟机版本/处理器型号：Virtualbox7.0.6、Ubuntu18.04
  + 代码编辑环境：gedit文本编辑器、gdb调试器、qemu
  + 代码编译工具：nasm

**Section 3 实验任务**

* 实验任务1：**混合编程的基本思路**，复现网址“⼀个混合编程的例子”部分
* 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，输出“学号+姓名首字母”
* 实验任务3：**中断的处理**，复现网址中“初始化IDT”部分，调用处理函数时输出个人学号或姓名信息
* 实验任务4：**时钟中断的处理**，复现网址中“8259A编程——实时钟中断的处理”部分，通过时钟中断，在屏幕的第一行实现一个跑马灯显示自己学号和英文名，不要使用纯汇编的方式来实现

**Section 4 实验步骤与实验结果**

------------------------- **实验任务1** -------------------------

* 任务要求：复现网址中“⼀个混合编程的例子”部分，要求：
  + 将原例子中的输出"Done"改为"Done by 学号 姓名首字母"
  + 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。结合关键代码说明global、extern关键字的作用，为什么C++的函数前需要加上 extern "C"
  + 学习make的使用，并用make来构建项目

保存结果截图并说说你是怎么做

* 思路分析：为了实现混合编程我们可以分别在C文件中实现并定义一个函数function\_from\_C，在C++文件中实现函数function\_from\_CPP，在asm文件中实现汇编函数function\_from\_asm，在function\_from\_asm的实现中我们调用上面实现的C/C++函数，然后在文件main.cpp中调用汇编函数function\_from\_asm，最后我们在main函数中输出我们的学号和名字“Done By 学号 姓名首字母”
* 实验步骤：

**一、复现并修改代码输出学号和姓名**

Step1：创建c文件c\_func.c中并定义C函数function\_from\_C

#include <stdio.h>

void function\_from\_C() {

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

}

Step2：创建cpp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;

}

Step3：创建汇编文件asm\_func.asm中定义汇编函数function\_from\_asm，在实现中调用前面的C/C++函数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

Step4：创建文件main.cpp中调用汇编函数function\_from\_asm，同时并输出我们的学号和姓名

#include <iostream>

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

int main() {

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

function\_from\_asm();

std::cout << **"Done by 23320104 ltx."** << std::endl;

}

}

Step5：将上述的4个文件统一编译成可重定位文件即.o文件，并将这些.o文件链接成一个可执行文件.exe

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

step6：运行可执行文件查看效果

main.out

**二、混合编程的语法总结**

①**在汇编代码中使用C函数**function\_from\_C:

声明：需要**在汇编代码中**声明这个函数来自于外部

extern function\_from\_C

调用：

对于**无参数的函数的调用**，声明后便可直接使用：

call function\_from\_C

对于**有若干参数的函数的调用**：

当我们需要在汇编代码中调用函数function\_from\_C(1,2)，应该从右到左将参数压入栈中，汇编代码如下。

push 2 ; arg2

push 1 ; arg1

call function\_from\_C

add esp, 8 ; 清除栈上的参数

call指令返回后，函数的返回值被放在了eax中。

**②在汇编代码中使用来自C++的函数function\_from\_CPP：**

定义：**在C++代码中函数定义前加上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，这导致了函数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，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。

在**C++代码中函数定义前**加上extern "C"如下

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

声明：**在汇编代码中声明**这个函数

extern function\_from\_CPP

**③在C/C++调用汇编函数：**

定义：需要**在汇编代码中将函数声明为global**

global function\_from\_asm

声明：

在C中将其声明来自外部

extern void function\_from\_asm();

在C++中需要声明为extern "C"

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

调用：

在C/C++代码中调用函数function\_from\_asm时，使用如下语句即可。

int ret = function\_from\_asm(1, 2);

**三、makefile文件处理混合编程编译、连接的编写**

makefile文件中代码如下：

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

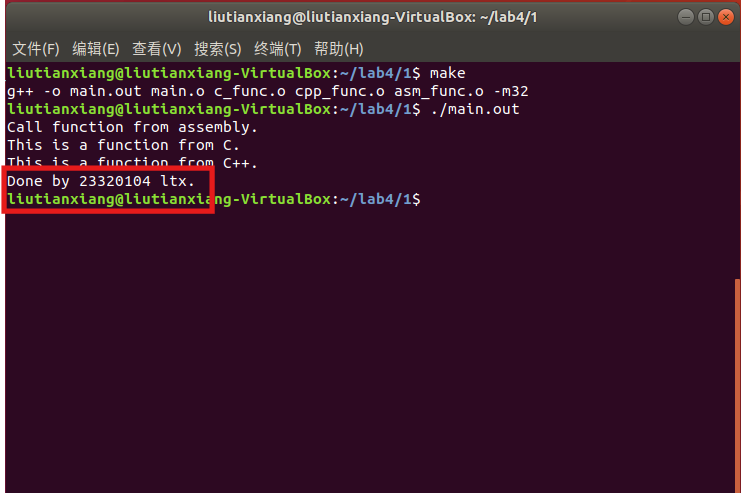
asm\_func.o: asm\_func.asm

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

clean:

rm \*.o

* 实验结果展示：通过执行前述代码，可得下图结果。

****

------------------------- **实验任务2** -------------------------

* 任务要求：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“学号+姓名首字母”，保存结果截图并说明是如何做的
* 思路分析：在前面两个实验中我们编写了mbr将我们的bootloader从我们的磁盘中加载到内存突破了512B的限制，同时编写了bootloader进入我们的保护模式并完成了环境的初始化。在进入保护模式之后我们需要在bootloader中从磁盘中加载我们的内核到内存中，然后我们跳转到内核的初始地址进入内核，我们可以通过混合编程的原理，用C/C++语言编写我们的内核，对于内核中某些需要用到汇编语句的地方，可以通过汇编编写并将其包装成一个函数供我们的内核调用。我们的任务：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“学号+姓名首字母”。

在这之前我们要明确我们操作系统源代码的文件结构，方便后续的完善和学习，具体如下：

├── build

│ └── makefile

├── include

│ └── 我们编写的文件中函数的声明和一些常量所编写的头文件

├── run

│ └── 存放gdb配置文件，硬盘映像.img文件等

└── src 存放函数实现的具体源代码

├── boot 加载mbr、bootloader的代码

├── kernel 内核源代码

└── utils 汇编函数实现代码

* 实验步骤：

在上一次实验的mbr.asm和bootloader.asm的基础上，我们修改以下bootloader.asm在其中加上加载内核程序代码到内存的函数代码，同时我们跳转到内核的起始地址，进入内核状态

Step1：**在boot.inc中加入新的一些常量**

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

KERNEL\_START\_SECTOR equ 6

KERNEL\_SECTOR\_COUNT equ 200

KERNEL\_START\_ADDRESS equ 0x20000

Step2：**在bootloader.asm的最后加上读取内核的代码**，我们在进入保护模式和初始化后，紧接着就加载我们的内核到内存中，我们需要读取200个扇区，从第6个扇区开始

... ; 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

Step3：**定义内核进入点entry.asm**，我们会在链接阶段巧妙地将它放在内核代码的最开始的地方，也就是0x20000处，使得bootloader在执行跳转到0x20000后，执行jmp setup\_kernel调用我们用C/C++所写的内核代码

extern setup\_kernel

enter\_kernel:

jmp setup\_kernel

Step4：**使用C++来编写内核函数setup\_kernel**

#include "asm\_utils.h"

extern "C" void setup\_kernel(){

asm\_hello\_world();

while(1) {

}

}

Step5：**编写汇编函数实现在asm\_utils.h文件中**，为了方便汇编代码的管理，我们将所有的汇编函数实现都防止在放置在asm\_utils.h下，在原来的asm\_hello\_world函数中是是输出hello world，**下面是修改后的代码用于输出我的学号和名字，**下面的程序利用了循环的方式来实现字符串的输出

[bits 32]

global asm\_hello\_world

asm\_hello\_world:

push eax

xor eax, eax

mov ecx, name\_and\_student\_number\_end - name\_and\_student\_number

mov ebx, 0

mov esi, name\_and\_student\_number

mov ah, 0x03 ;青色

loop\_table:

mov al, [esi]

mov word[gs:ebx], ax

add ebx, 2

inc esi

loop loop\_table

pop eax

ret

name\_and\_student\_number db '23320104 ltx' ;学号和名字 23320104ltx

name\_and\_student\_number\_end:

Step6：**创建文件asm\_utils.h统一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

#ifndef ASM\_UTILS\_H

#define ASM\_UTILS\_H

extern "C" void asm\_hello\_world();

#endif

Step7：最后**创建一个makefile文件**，将我们的编译、连接、生成可执行文件、dd写入硬盘、启动qemu都打包成makefile文件，最后输入make即可

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 : 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.asmclean:

rm -f \*.o\* \*.bin

run:

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

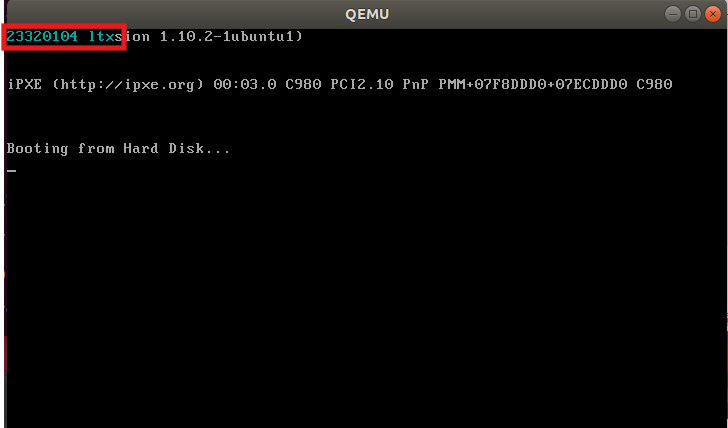
debug:

qemu-system-i386 -S -s -parallel stdio -hda $(RUNDIR)/hd.img -serial null&

@sleep 1

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

* 实验结果展示：通过执行前述代码，可得下图结果。



------------------------- **实验任务3** -------------------------

* 任务要求：复现网址中“初始化IDT”部分，更改默认的中断处理函数为你编写的函数，然后触发之，调用处理函数时输出个人学号或姓名信息，对结果进行截图并说明是如何实现的。
* 思路分析：初始化中断描述符表IDT，需要完成下面的三步：
  + 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

这里我们将会初始化IDT的256个中断，这256个中断的中断处理程序均是向屏幕输出“学号+姓名首字母”后关中断并死循环

* 实验步骤：

Step1：**创建os\_type.h定义了基本的数据类型的别名**，方便后续使用

#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

Step2：**在头文件夹中定义中断处理模块类中断管理器InterruptManager**

#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

voidsetInterruptDescriptor(uint32index,uint32address,byte DPL);

};

#endif

Step3：然后我们就要一一**实现中断管理器的成员函数**

**InterruptManager::initialize函数**：先设置IDTR，然后再初始化256个中断描述符

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);

}

}

观察上述的代码我们可以看到还是需要很多函数需要实现，其中asm\_lidt是使用汇编代码来实现的一个函数：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，因为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

; 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

**InterruptManager::setInterruptDescriptor函数：**设置中断描述符,不同的中断描述符的差别只在于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，函数名地址就是中断处理程序在目标代码段中的偏移。

// 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);

}

Step4：**定义默认的中断处理函数asm\_interrupt\_empty\_handler**，我们使用汇编来实现,这里我们要实现我们的名字学号的输出，所以我们把上面的字符串改成自己的学号和名字,**首先关中断，然后输出提示字符串，最后做死循环**

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

**---23320104 ltx**'

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 $

Step5：**在setup\_kernel.cpp中中定义并初始化中断处理器**，并尝试触发除0异常，来出发我们的中断

... // 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();

// 尝试触发除0错误

**int a = 1 / 0;**

// 死循环

asm\_halt();

}

创建os\_modules.h并声明中断处理器实例，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

创建os\_constant.h将一些常量统一定义

#ifndef OS\_CONSTANT\_H

#define OS\_CONSTANT\_H

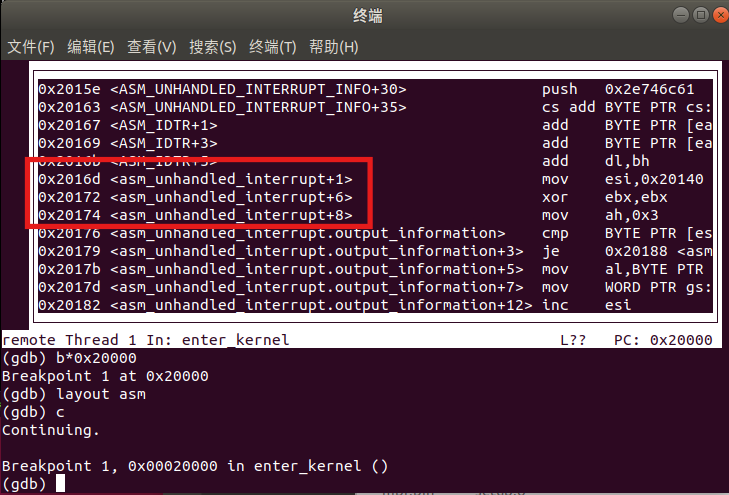
#define IDT\_START\_ADDRESS 0x8880

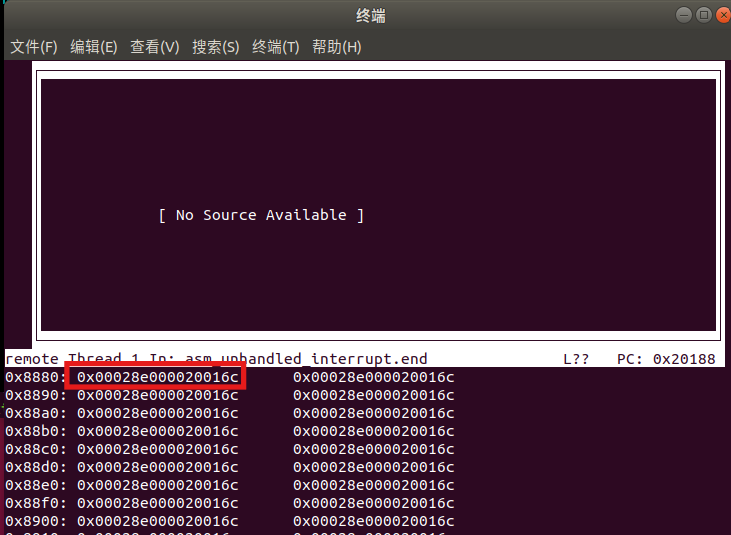
#define CODE\_SELECTOR 0x20

#endif

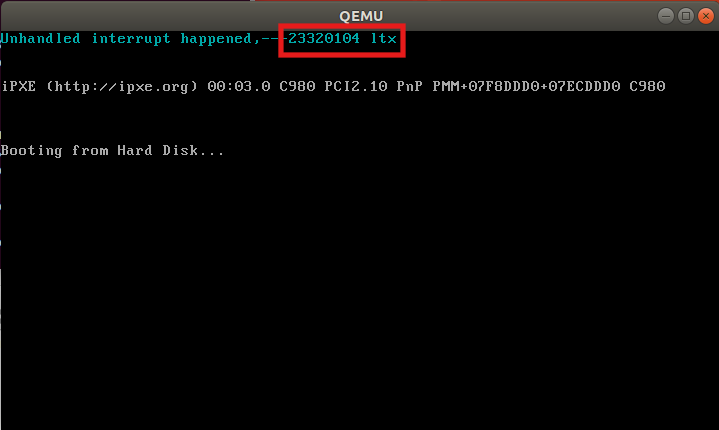
Step6：**代码调试**

我们使用make debug指令对代码进行了调试，可以看到我们的IDT中的后32位就是我们中断处理函数的地址也就是下图所示的0x2016c





* 实验结果展示：通过执行前述代码，可得下图结果。



------------------------- **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，仿照该章节中使用C语言来实现时钟中断的例子，**使用混合编程的方式**来实现利用C/C++、InterruptManager、STDIO和你自己封装的类来实现时钟中断处理过程：通过时钟中断，在屏幕的第一行实现一个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：我们对8529A芯片进行编程，添加处理时钟中断的函数，这个函数是一个跑马灯显示学号的函数，实现学号名字的滚动显示效果，首先我们编写好我们的时钟中断处理函数，然后再读取8529A芯片上的OCW1并修改OCW1来实现时钟中断的开启和结束，时钟中断开启，每次中断都会调用我们的时钟中断处理函数，然后返回，这里可以理解成一个循环，把时钟中断处理函数当作我们的循环体即可。然后我们需要设置开中断，不然CPU可能不会理会我们的可屏蔽中断。
* 实验步骤：

**一、准备工作**

Step1：**为中断控制器InterruptManager加入如下成员变量和函数**

// 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);

// 初始化8259A芯片

void initialize8259A();

Step2：**将汇编中in，out指令封装成函数**，因为初始化8259A芯片的过程是通过设置一系列的ICW字来完成的，其中涉及到对系统I/O端口的读写，为了在C/C++程序中能够使用到in，out汇编功能我们需要对其包装成函数

; 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

; 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

Step3：**实现initialize8259A对8259A芯片进行初始化用于后续使用**

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);

}

Step4：**编写能够处理屏幕输出的类STDIO**，在中断处理函数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，我们希望能够将这些汇编代码包装成函数能够直接调用

**类声明文件**

#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

**成员函数的实现**：

三个重载的print是直接向显存对应的位置写入字符和颜色，比较简单

**moveCursor与getCursor函数**：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

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;

}

**STDIO::rollUp函数**：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

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;

}

}

**二、利用时钟中断编写程序**

在完成所有的准备工作后，我们正式的开始利用时钟中断编写我们的程序

Step1：**编写中断处理函数**，这里我们直接修改了实验手册原来的中断处理函数，**编写了跑马灯展示学号和姓名的中断处理函数**，具体就是**每一次中断的时候都将上一次显示的字符消除，显示当前的字符**

extern "C" void trotting\_horse\_lamp\_interrupt\_handler(){

char str[]="23320104ltx";

// 清空屏幕

for (int i = 0; i < 80; ++i){

stdio.print(0, i, ' ', 0x07);

}

int len=11;

int pre\_index=(times-1) > 0 ? (times - 1)%len : (times- 1 + len)%len;

int cur\_indedx=(times) > 0 ? times%len : (times + len)%len;

if(cur\_indedx == 10){

for(int i=0;str[i];i++){

stdio.print(str[i],i+2);

}

asm\_delay();

}

times++;

stdio.moveCursor(pre\_index);

stdio.print(' ');

stdio.moveCursor(cur\_indedx);

stdio.print(str[cur\_indedx],cur\_indedx+2);

asm\_delay();

}

这里我们为了**体现我们使用了混合编程的编程方法**，我们**用汇编语言编写了一个asm\_delay函数实现延迟效果**，使得结果更加的可视化，具体实现就是利用循环操作使得CPU空转达到延迟效果

asm\_delay:

;双重循环使CPU空转以达到延迟效果

mov cx, 0x500 ;外层循环次数

.out\_delay:

mov si, 0xFFFF ;内层循环次数

.in\_delay:

dec si

jnz .in\_delay

dec cx

jnz .out\_delay

ret

由于C语言不支持iret指令，但是完整的中断处理程序需要保护现场和恢复现场，所以我们需要使用汇编语言来进行完整的中断处理函数的编写

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 trotting\_horse\_lamp\_interrupt\_handler

popad

iret

Step2：**设置主片IRQ0中断对应的中断描述符**

void InterruptManager::setTimeInterrupt(void \*handler)

{

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

}

Step3：**开启和关闭时钟中断**

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);

}

Step4：**开中断**

; void asm\_enable\_interrupt()

asm\_enable\_interrupt:

sti

ret

Step5：**最后在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();

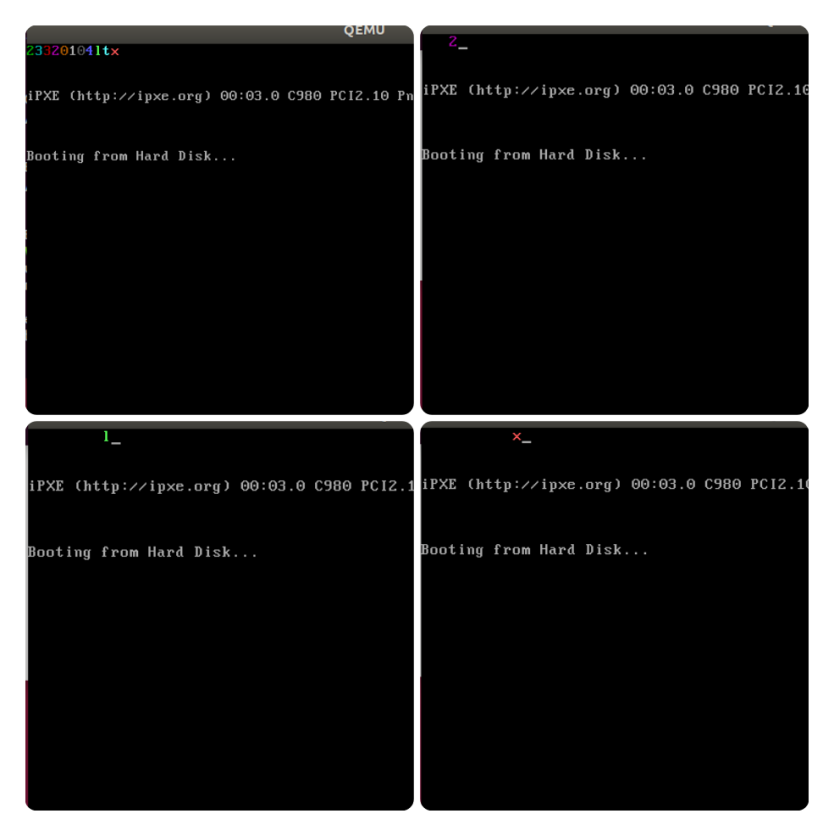
interruptManager.setTimeInterrupt((void \*)asm\_time\_interrupt\_handler);

asm\_enable\_interrupt();

asm\_halt();

}

* 实验结果展示：通过执行前述代码，可得下图结果，在提供的源代码中还有程序执行的动态效果



**Section 5 实验总结与心得体会**

通过本次实验，学了关于将源代码到可执行文件的四个阶段，预处理、编译、汇编、连接，通过这四个阶段将多个源代码生成一整个可执行文件，进而使得我们能够运行。

在了解代码编译的过程之后，我们利用这个过程，进一步学习了混合编程，即通过使用不同的编程语言来完成我们的项目的搭建，最经典的就是混合使用C/C++高级语言和x86汇编语言来进行编程，我们可以结合C/C++语言和汇编语言的优势来进行我们代码的书写，例如使用C/C++语言来对项目进行主要的搭建，其中一些关于磁盘、内存的操作我们使用汇编语言来编写，通过函数的方式来调用。

在混合编程的基础上，我们优化了代码的文件结构，使得我们的各个代码文件更加有条理，在这基础上我们从上一次实验的mbr、boot结合干刚刚学习的混合编程的方式进一步的搭建我们的操作系统内核，并进入我们的内核，所谓的内核起始就是操作系统的代码。

在编写了内核的入口的基础上，我们继续在内核之中完善我们内核要实现的操作系统的功能，其中第一个要完成的功能便是操作系统的中断，我们通过初始化中断描述符IDT以及中断处理函数的编写来实现我们的中断处理。

最后我们学习到了8259A芯片的结构，了解了产生中断的硬件原理，并利用时钟中断来编写我们的一个中断处理程序。

通过本次实验，更加深入的了解到了操作系统内核的结构以及操作系统中断的产生、中断处理实现，更加深刻的体会到了操作系统是通过中断来驱动的这句话的含义。

**Section 6 附录：代码清单**

由于代码文件较多，代码量较大，为了篇幅的精简，具体代码以附件的形式给出