

本科生实验报告

实验课程: 操作系统原理实验

实验名称: 编译内核/利用已有内核创建OS 专业名称: 计算机科学与技术

学生姓名: 张玉瑶

学生学号: 23336316

实验地点: 实验楼B203

实验成绩:

报告时间: 2024年4月10日

**Section 1 实验概述**

* 实验任务1：

1. 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。
2. 学习make的使用，并用make来构建Example 1。

* 实验任务2：完成...
* 实验任务3：

1. 在src/8的基础上，仿照Example 3编写段错误的中断处理函数，正确实现段错误的中断处理并正确地在中断描述符中注册。

2.额外思考：使用尽可能多的方法触发段错误，并在实验报告里总结一下，引发段错误都有哪几种。

* 实验任务4：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，并通过这样的时钟中断，使用C/C++语言来复刻lab2的assignment 4 的字符回旋程序。

**Section 2 实验步骤与实验结果**

---------------------- **实验任务1：混合编程的基本思路**--------------------

* 任务要求：

1.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。

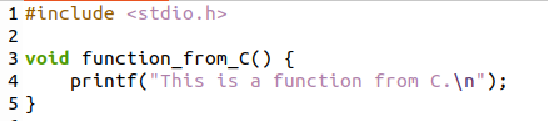
2.学习make的使用，并用make来构建Example 1。

* 思路分析：本实验通过C++调用汇编函数，再由汇编调用C和C++函数，展示混合编程机制。关键点包括：

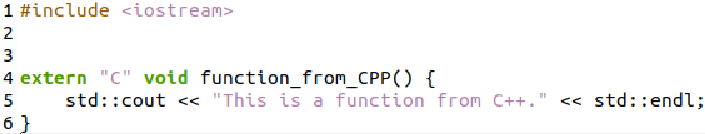
1. extern "C" 确保C++函数名不被修饰，使汇编能正确调用。
2. global 和 extern 在汇编中声明导出/导入函数。
3. Makefile 自动化编译链接，处理不同语言的编译规则。

* 实验步骤：

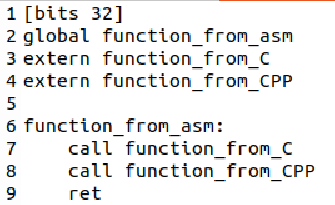
1. 在文件c\_func.c中定义C函数function\_from\_C。



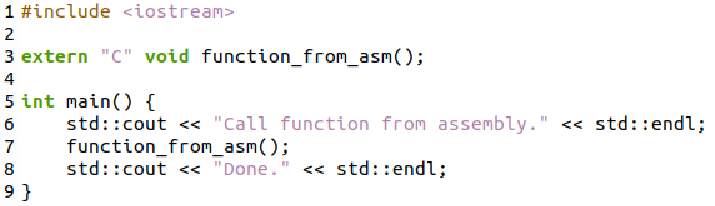
1. 在文件cpp\_func.cpp中定义C++函数function\_from\_CPP。



1. 在文件asm\_utils.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。



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



5.c代码调用汇编函数的语法：

1) main.cpp调用function\_from\_sam();

2) asm\_utils.asm调用function\_from\_C()和function\_from\_CPP()。

3) asm\_utils.asm的语法

global function\_from\_asm ;声明function\_from\_asm为全局符号，可供其他模块调用

extern function\_from\_C ;声明function\_from\_C为外部符号，将在其他模块中定义

extern function\_from\_CPP ;声明function\_from\_CPP为外部符号，将在其他模块中定义

function\_from\_asm:

call function\_from\_C ;调用C函数function\_from\_C()

call function\_from\_CPP ;调用C++函数function\_from\_CPP()

ret ;返回

4）c\_func.c的语法

#include <stdio.h>

void function\_from\_C() {

printf("This is a function from C.\n"); //打印引号中的内容

}

5）cpp\_func.cpp的语法

#include <iostream>

extern "C" void function\_from\_CPP() {//禁用C++的名称修饰，使函数名保持为简单的"function\_from\_CPP"

std::cout << "This is a function from C++." << std::endl; //打印引号中的内容

}

6）main.cpp语法

#include <iostream>

extern "C" void function\_from\_asm(); //extern "C"用于声明汇编函数，确保名称匹配

int main() {

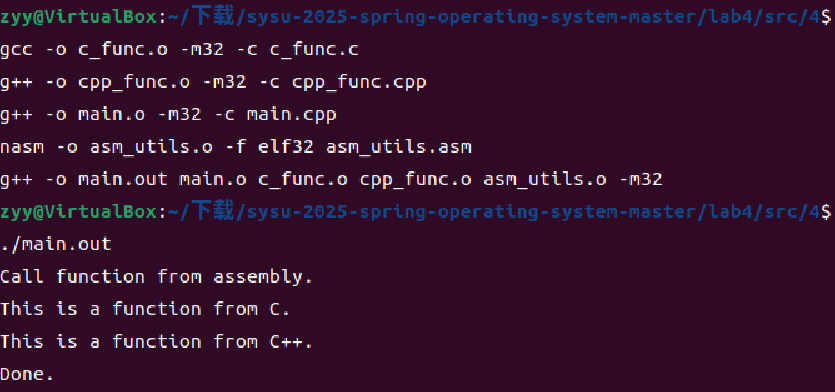
std::cout << "Call function from assembly." << std::endl;

function\_from\_asm();

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

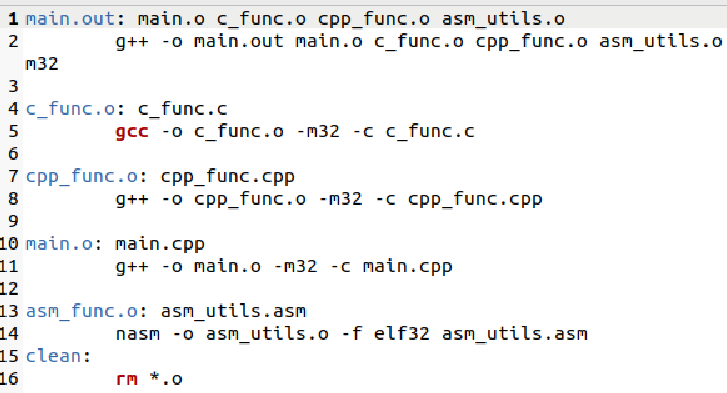
}

6.普通编译

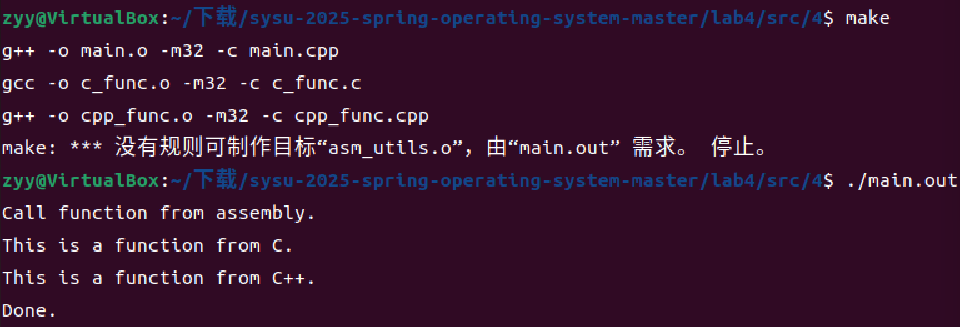


7．使用make来构建。

1）MakeFile文件的编写。



2）编译运行，展示结果。



--------------------- **实验任务2：使用c/c++来编写内核**-------------------

* 任务要求： 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。
* 思路分析：这个实验的核心思路是通过bootloader加载操作系统内核并完成控制权交接。主要分为以下几个步骤：

1．bootloader完成保护模式初始化后，从硬盘第6扇区加载200个扇区的内核到内存0x20000地址处，然后跳转到该地址执行。

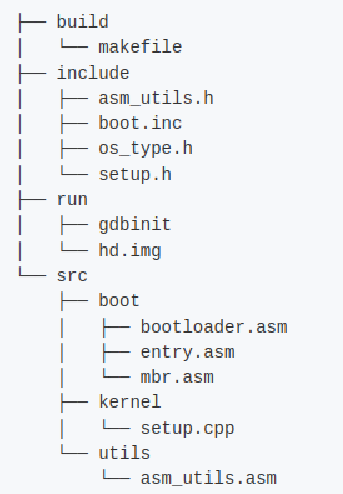
2．内核采用C/C++和汇编混合编程，确保：内核入口代码(entry.asm)被放置在二进制文件开头，使用objcopy生成纯净的机器指令文件(kernel.bin)

3．目录结构采用标准C/C++项目布局，通过Makefile管理编译过程，使用-I参数简化头文件引用。

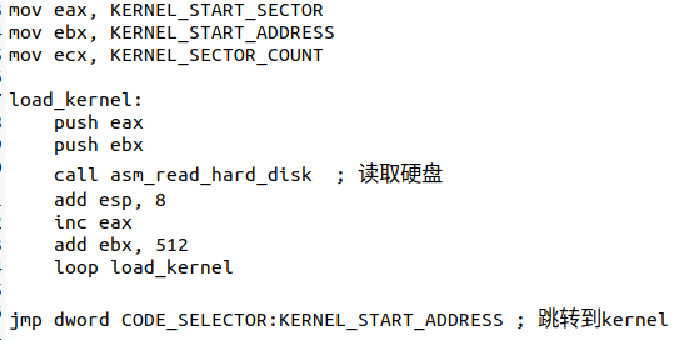
4．最终通过dd命令将MBR、bootloader和内核按顺序写入硬盘映像，实现完整的启动流程。

* 实验步骤：

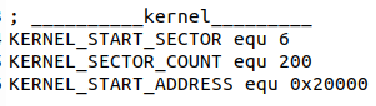
1.文件目录结构。



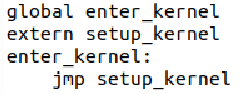
2.把读取内核的代码置于bootloader的最后。



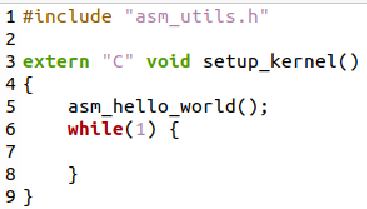
3. 常量的定义放置在5/include/boot.inc下，新增的内容如下。



4. 在src/boot/entry.asm下定义内核进入点。



5.将setup\_kernel的定义在文件src/kernel/setup.cpp中。



6. 将汇编函数放置在src/utils/asm\_utils.h下。

[bits 32]

global asm\_hello\_world

asm\_hello\_world:

push eax

xor eax, eax

mov ah, 0x68 ;

mov al, 'h'

mov [gs:2 \* 0], ax

mov al, 'e'

mov [gs:2 \* 1], ax

mov al, 'l'

mov [gs:2 \* 2], ax

mov al, 'l'

mov [gs:2 \* 3], ax

mov al, 'o'

mov [gs:2 \* 4], ax

mov al, ' '

mov [gs:2 \* 5], ax

mov al, 'w'

mov [gs:2 \* 6], ax

mov al, 'o'

mov [gs:2 \* 7], ax

mov al, 'r'

mov [gs:2 \* 8], ax

mov al, 'l'

mov [gs:2 \* 9], ax

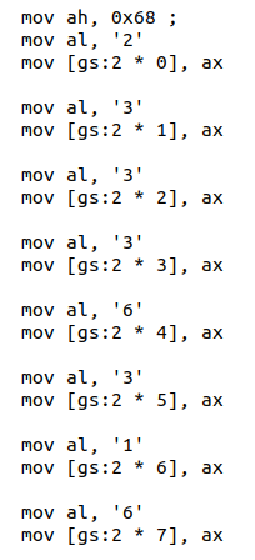
mov al, 'd'

mov [gs:2 \* 10], ax

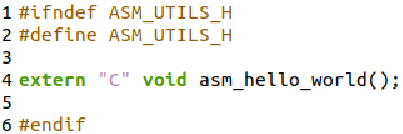
pop eax

ret

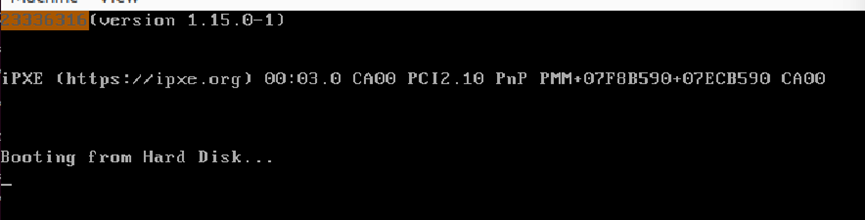
如果要输出23336316，把代码改成如下。

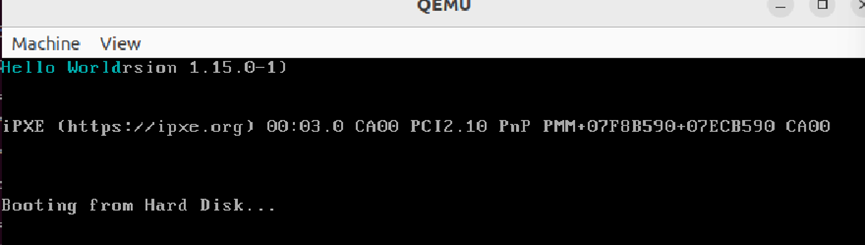


7. 统一在文件include/asm\_utils.h中声明所有的汇编函数，这样我们就不用单独地使用extern来声明了，只需要#include "asm\_utils.h"即可。

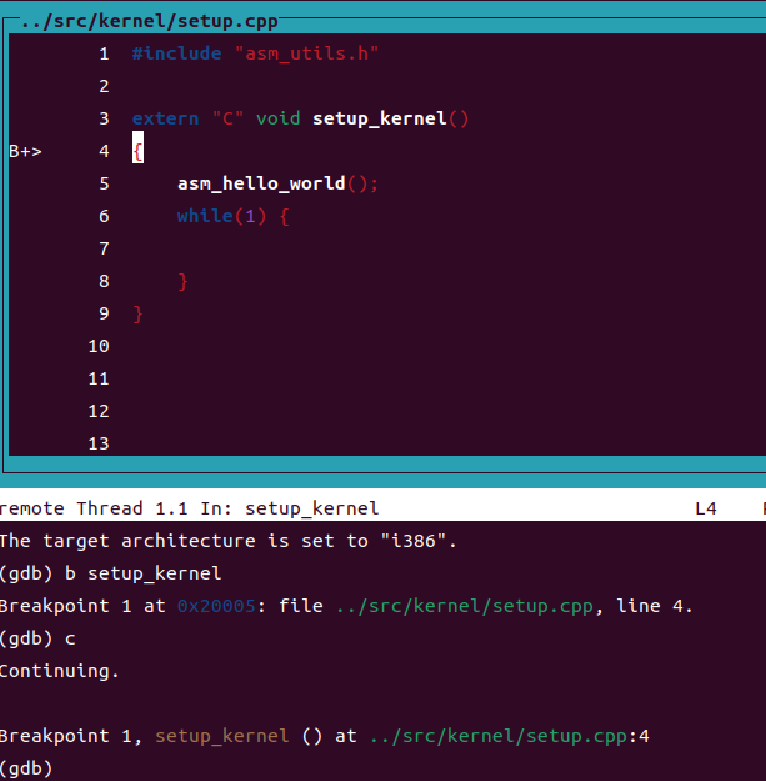


8.利用make进行编译。





9.利用make debug进行调试。



**------------------------- 实验任务3 ：中断的处理-------------------------**

* 任务要求：

1. 在src/8的基础上，仿照Example 3编写段错误的中断处理函数，正确实现段错误的中断处理并正确地在中断描述符中注册。

2.额外思考：使用尽可能多的方法触发段错误，并在实验报告里总结一下，引发段错误都有哪几种。

* 思路分析：

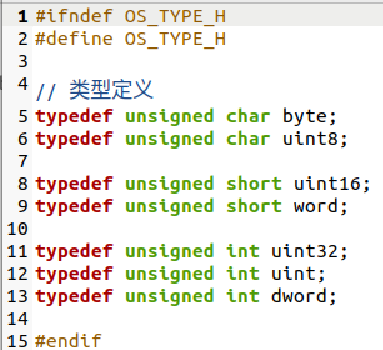
本实验通过实现段错误的中断处理机制，深入理解操作系统的内存保护与中断管理。核心步骤包括：1) 定义中断管理器类（InterruptManager），初始化IDT并设置默认中断描述符；2) 编写段错误处理函数，通过IDT注册；3) 触发段错误（如访问非法内存、空指针等），验证处理流程。实验重点在于掌握IDT结构、中断描述符配置及保护模式下的异常处理机制，同时通过多种方式触发段错误（如越界访问、权限违规），分析其原理与防护措施。

* 实验步骤：

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

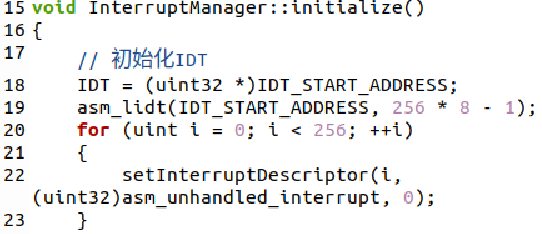


1. 在include/os\_type.h定义了基本的数据类型的别名。



1. 在使用中断之前，我们首先需要初始化IDT。

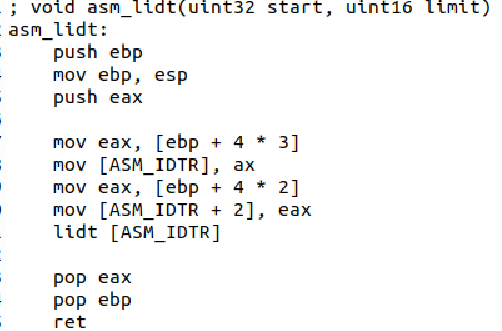
1）初始化IDT的重要函数为InterruptManager::initialize。



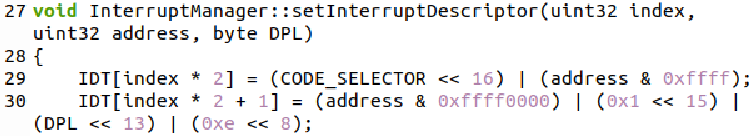
2）将IDT设定在地址0x8880处，即IDT\_START\_ADDRESS=0x8880。



3）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src/utils/asm\_utils.asm中。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。



4.将段描述符的设置定义在函数InterruptManager::setInterruptDescriptor中。



5.编写段错误的中断处理函数。

1）定义默认的中断处理函数asm\_interrupt\_empty\_handler。函数首先关中断，然后输出提示字符串，最后做死循环。

; 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 $

2）初始化页目录寄存器并启用分页机制的函数asm\_init\_page\_reg(int \*directory)，将传入的页目录表地址（directory参数）加载到CR3寄存器，设置CR0寄存器的PG位（第31位）为1，启用分页机制。

; void asm\_init\_page\_reg(int \*directory);

asm\_init\_page\_reg:

push ebp

mov ebp, esp

push eax

mov eax, [ebp + 4 \* 2]

mov cr3, eax ; 放入页目录表地址

mov eax, cr0

or eax, 0x80000000

mov cr0, eax ; 置PG=1，开启分页机制

pop eax

pop ebp

ret

3）检查当前中断状态的函数asm\_interrupt\_status()，可以查询当前CPU是否允许中断。先读取EFLAGS寄存器，然后提取中断标志位（第9位，0x200），最后返回该位的状态（0表示中断禁用，非0表示中断启用）。

; int asm\_interrupt\_status();

asm\_interrupt\_status:

xor eax, eax

pushfd

pop eax

and eax, 0x200

ret

4）禁用处理器中断函数asm\_disable\_interrupt()，执行cli指令清除中断标志，在进入临界区前关闭中断。

; void asm\_disable\_interrupt();

asm\_disable\_interrupt:

cli

ret

; void asm\_init\_page\_reg(int \*directory);

asm\_enable\_interrupt:

sti

ret

5）启用处理器中断函数asm\_enable\_interrupt()，执行sti指令设置中断标志，离开临界区后恢复中断。

asm\_enable\_interrupt:

sti

ret

6）asm\_in\_port(uint16 port, uint8 \*value) 功能为从指定I/O端口读取一个字节。先使用in指令从端口读取数据，再将结果存储到传入的指针指向的内存位置，实现与硬件设备通信。

; 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

7）asm\_out\_port(uint16 port, uint8 value)功能为向指定I/O端口写入一个字节，可以向硬件设备发送命令或数据。

; 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

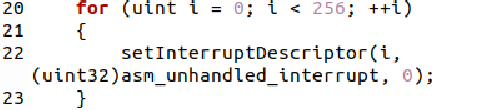
pop eax

pop edx

pop ebp

ret

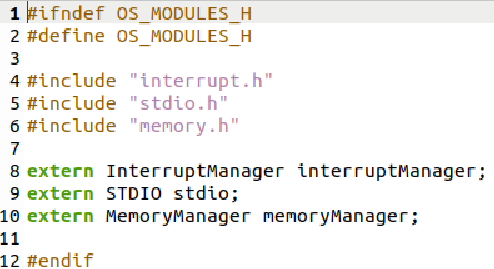
6. 在InterruptManager::initialize最后，调用setInterruptDescriptor放入256个默认的中断描述符，这256个默认的中断描述符对应的中断处理函数是asm\_unhandled\_interrupt。



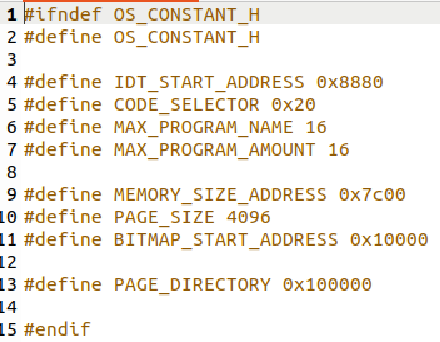
7．最后，我们在函数src/kernel/setup\_kernel.cpp中定义并初始化中断处理器、输出管理器和内存管理器。



8.在include/os\_modules.h声明实例。

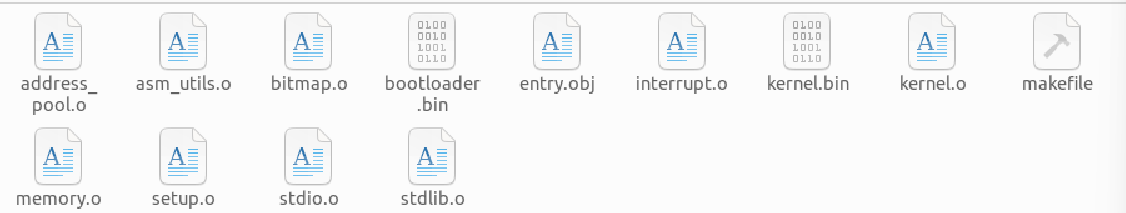


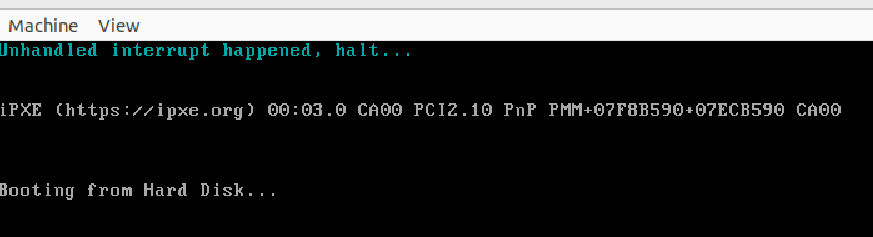
9．将一些常量统一定义在文件include/os\_constant.h下。



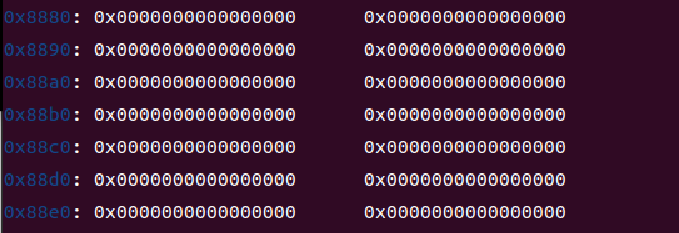
10.编译运行！中断被触发。

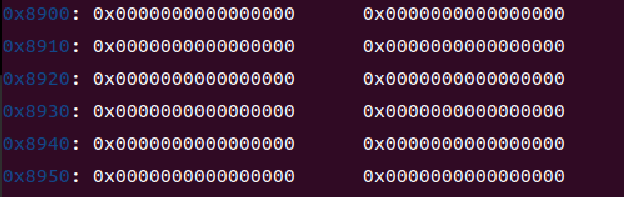




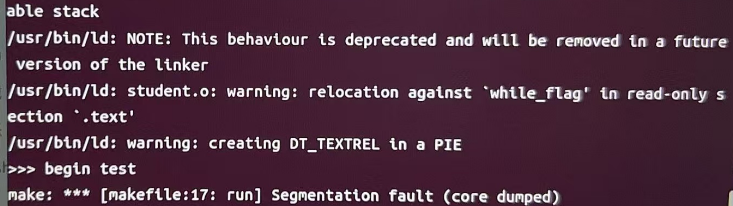


11.debug，在gdb下使用x/256gx 0x8880命令可以查看我们是否已经放入默认的中断描述符。





12.**其他段错误：**之前lab2的assignment3写的代码曾经触发过段错误，说明我的程序试图访问非法内存，导致操作系统强制终止程序并生成 core dump 文件。



your\_while:

mov ebx, [a2]

loop:

cmp ebx, 25

jge end\_while

call my\_random

shl ebx, 1

mov ecx, [while\_flag]

add ebx,ecx

mov [ebx], eax

mov ecx,[a2]

add ecx,1

mov [a2],ecx

mov ebx, [a2]

jmp loop

end\_while:

%include "end.include"

**其他的段错误的触发方式**：访问空指针，访问内核空间，栈溢出等。

**思考题总结：**因为实验内核通常采用平坦内存模型，未严格隔离内存区域，导致越界访问仍落在有效范围内，所以有时候“数组越界”等错误并不会触发中断。但是访问未映射内存、权限违规或执行不可执行代码会触发段错误，因为现代操作系统通过分页等机制来严格的检测错误，内核缺少这些保护，导致这些非法访问未被拦截威胁内核，因而显示报错。

------------------------- **实验任务4** -------------------------

* 任务要求： 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，并通过这样的时钟中断，使用C/C++语言来复刻lab2的assignment 4 的字符回旋程序。
* 思路分析：

本实验通过实现时钟中断处理机制，结合C/C++和汇编完成动态字符显示功能。核心步骤包括：1) 初始化8259A中断控制器，配置时钟中断；2) 封装屏幕输出类（STDIO）管理光标和显示；3) 编写中断处理函数，实现实时时钟显示、固定学号输出及字符回旋动画。实验重点在于理解硬件中断机制、8253/8259A芯片的编程方法，以及混合编程下的中断响应与处理流程，最终通过时钟中断驱动动态界面，复现字符回旋效果。

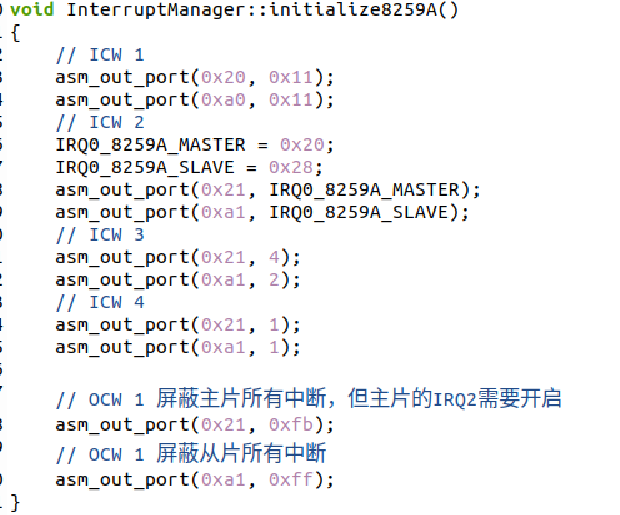
* 实验步骤：

1. 复现example4。按照教程不再赘述。
2. 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，并通过这样的时钟中断，使用C/C++语言来复刻lab2的assignment 4 的字符回旋程序。

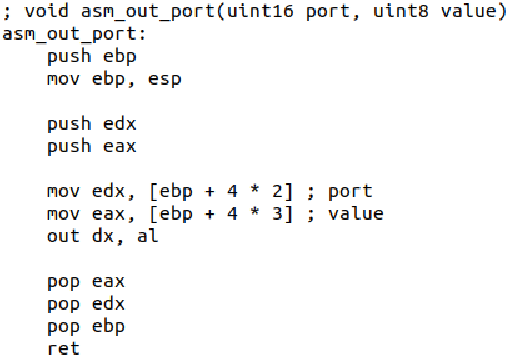
1） 为中断控制器InterruptManager加入如下成员变量和函数。



2）初始化8259A芯片，初始化的代码放置在成员函数initialize8259A中。

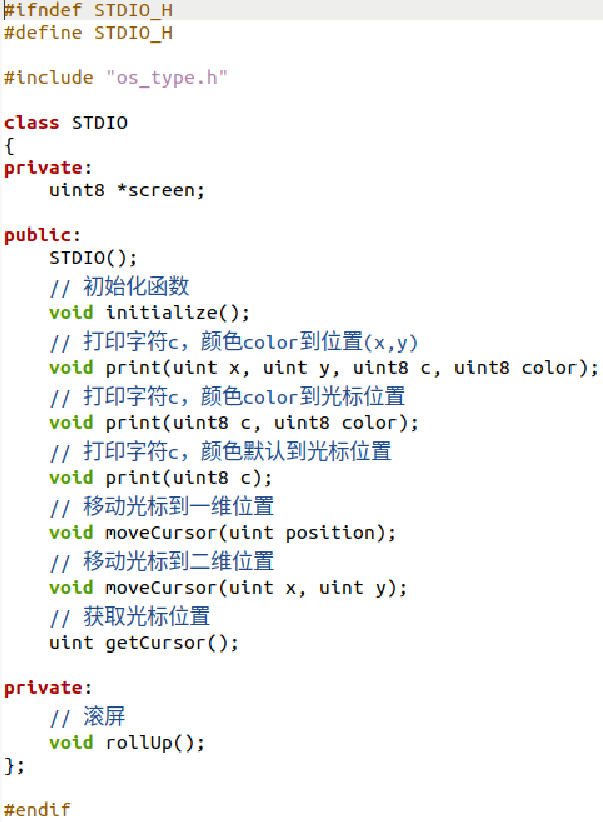


3）初始化8259A芯片的过程是通过设置一系列的ICW字来完成的。由于我们并未建立处理8259A中断的任何函数，因此在初始化的最后，我们需要屏蔽主片和从片的所有中断。其中，asm\_out\_port是对out指令的封装，放在asm\_utils.asm中。



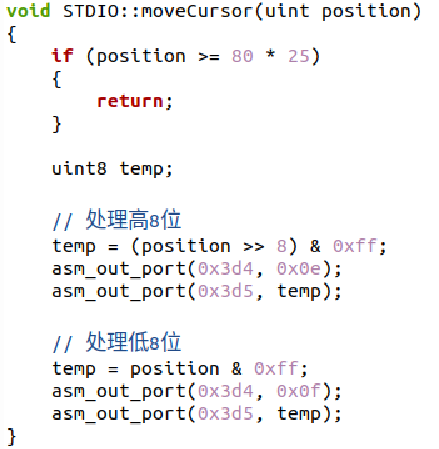
4）接下来处理时钟中断，我们处理的时钟中断是主片的IRQ0中断。在计算机中，有一个称为8253的芯片，其能够以一定的频率来产生时钟中断。当其产生了时钟中断后，信号会被8259A截获，从而产生IRQ0中断。处理时钟中断并不需要了解8253芯片，只需要对8259A芯片产生的时钟中断进行处理即可。

I．编写中断处理函数。我们希望能够像printf和putchar这样的函数来调用，因此，我们简单封装一个能够处理屏幕输出的类STDIO。

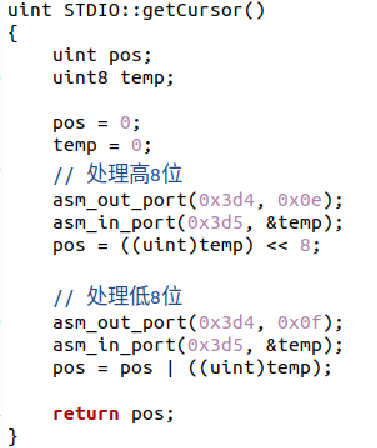


接下来处理光标，屏幕的像素为25\*80，所以光标的位置从上到下，从左到右依次编号为0-1999，用16位表示。与光标读写相关的端口为0x3d4和0x3d5，在对光标读写之前，我们需要向端口0x3d4写入数据，表明我们操作的是光标的低8位还是高8位。写入0x0e，表示操作的是高8位，写入0x0f表示操作的是低8位。如果我们需要需要读取光标，那么我们从0x3d5从读取数据；如果我们需要更改光标的位置，那么我们将光标的位置写入0x3d5。

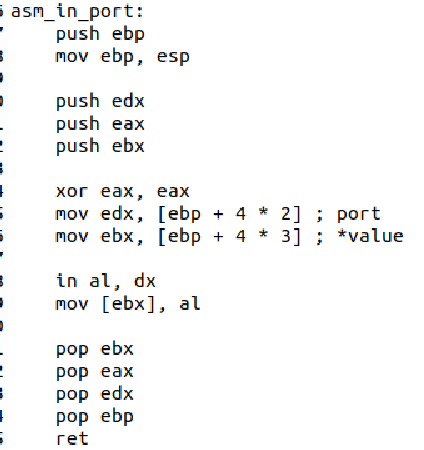
移动光标函数：



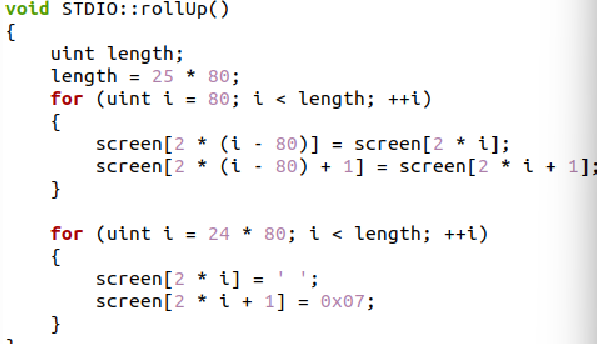
获取光标位置的函数：



封装in指令：

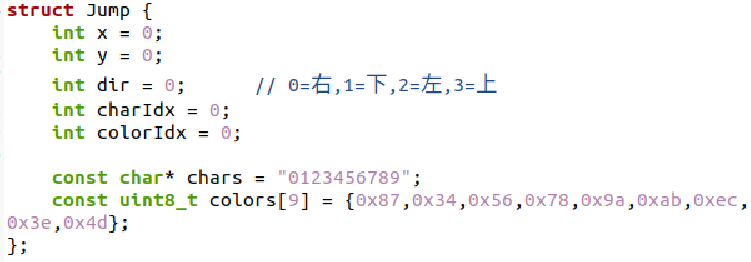


实现滚屏的函数：



**定义中断处理函数：c\_time\_interrupt\_handler，每18次中断记作一秒，按时钟格式显示。实现功能：1.****把计数器改成时钟格式显示；2。固定显示学号；3.实现字符串回旋。**

Jump类：实现移动、变色、变字符



I）．把计数器改成时钟格式显示。

extern "C" void c\_time\_interrupt\_handler()

{

static Jump jump;

++ticks;

// 每18次中断大约1秒

if(ticks% 18 == 0) {

++seconds;

char timeStr[9] = "00:00";

uint32\_t minutes = seconds / 60;

uint32\_t secs = seconds % 60;

// 分钟十位和个位

timeStr[0] = '0' + minutes / 10;

timeStr[1] = '0' + minutes % 10;

// 秒数十位和个位

timeStr[3] = '0' + secs / 10;

timeStr[4] = '0' + secs % 10;

stdio.moveCursor(13,35);

// 打印完整时间字符串

for(int i = 0; i &lt; 5; ++i) {

stdio.print(timeStr[i]);

}

Ii）固定显示学号。

if(!id\_shown){

const int center\_row = 12;

const int center\_col = (80 - 8)/2;

stdio.moveCursor(center\_row, center\_col);

const char\* student\_id = "23336316";

for(int i = 0; i &lt; 8; ++i) {

stdio.print(student\_id[i]);

}

id\_shown = true;

}

}

Iii）每一次中断都更新一次回旋字符串。

// 更新位置

switch(jump.dir) {

case 0:

if(++jump.x &gt;= 24) {

jump.x=23;

jump.dir=1;

} break;

case 1:

if(++jump.y &gt;= 79) {

jump.y=78;

jump.dir=2;

} break;

case 2:

if(--jump.x &lt; 0) {

jump.x=0;

jump.dir=3;

} break;

case 3:

if(--jump.y &lt; 0) {

jump.y=0;

jump.dir=0;

} break;

}

// 更新字符和颜色

jump.charIdx = (jump.charIdx + 1) % 10;

jump.colorIdx = (jump.colorIdx + 1) % 11;

stdio.print(jump.x, jump.y,

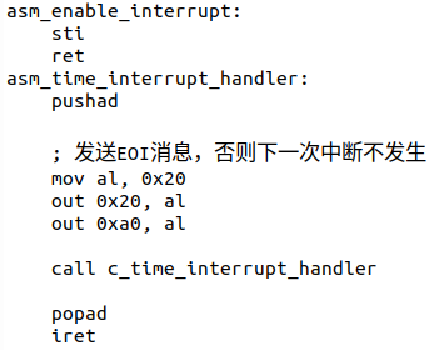
jump.chars[jump.charIdx],

jump.colors[jump.colorIdx]);

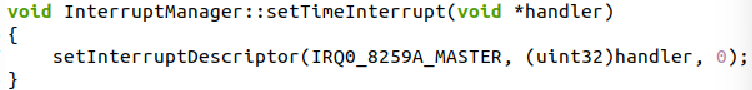
asm\_out\_port(0x20, 0x20); // EOI

}

完整的时钟处理函数。



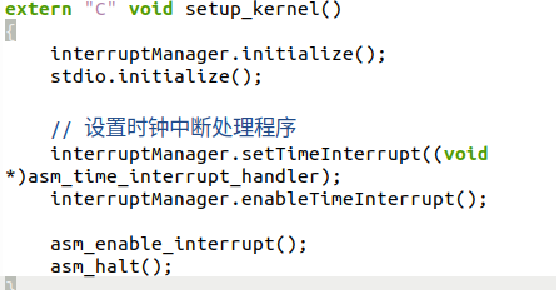
Ii. 设置时钟中断的中断描述符，也就是主片IRQ0中断对应的描述符。



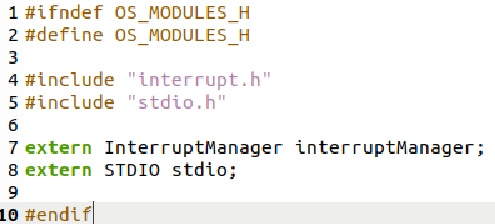
封装一下开启和关闭时钟中断的函数。关于8259A上的中断开启情况，我们可以通过读取OCW1来得知；如果要修改8259A上的中断开启情况，我们就需要先读取再写入对应的OCW1。



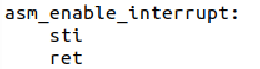
Iii & iv 在setup\_kernel中定义STDIO的实例stdio，最后初始化内核的组件，然后开启时钟中断和开中断。



声明实例。

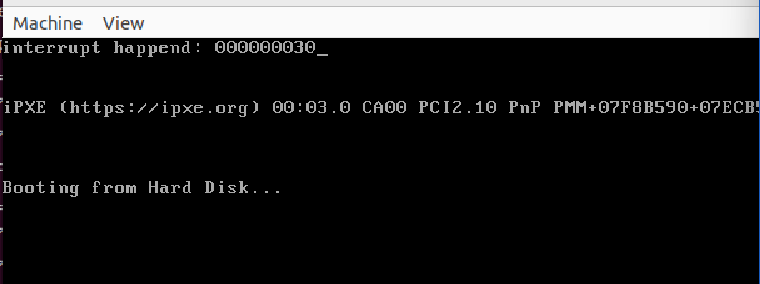


开中断指令。



最后编译运行。

* 实验结果展示：通过执行前述代码，可得下图结果。





**Section 5 实验总结与心得体会**

从这次实验中，我更深入理解了操作系统底层的中断处理机制，学习了混合编程，段错误的触发方式等。本次学习还是很有难度的，我会继续学习加深理解，为后续的实验和理论做好充分的准备。