

本科生实验报告

实验课程:\_\_\_\_\_\_\_\_\_\_\_\_\_操作系统实验 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

实验名称:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**lab4** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

专业名称:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_计科 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

学生姓名:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_丁晓琪\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

学生学号:\_\_\_\_\_\_\_\_\_\_\_\_\_ 22336057\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

实验地点:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

实验成绩:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

报告时间:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Assignment1

1. **实验要求**

**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。**

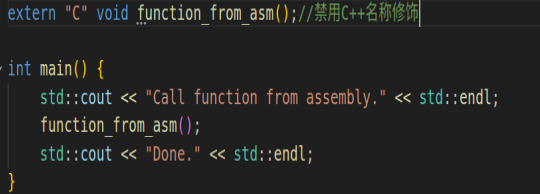
1. **实验过程**
2. main函数在main.cpp中，在main.cpp中调用汇编函数function\_from\_asm()
3. 在function\_from\_asm()中分别调用C语言函数function\_from\_C和C++函数function\_from\_CPP()
4. 在c\_func.c中实现 function\_from\_c的定义，cpp\_func.cpp中实现function\_from\_CPP的定义
5. 在makefile文件中，编写上述文件的编译和依赖规则
6. make编译，生成可执行文件和运行
7. **关键代码**
8. 在main.cpp中调用汇编函数function\_from\_asm

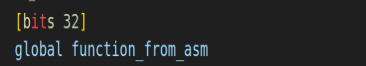
* 要在汇编代码实现中将其声明为 global，以便在链接阶段找到函数实现

global关键字：将符号添加到符号表，程序的任何部分可以引用，其他代码段也可以访问和使用该符号

* 在main.cpp中声明函数来自外部

extern “C”: 告知编译器该函数遵循C语言的名称修饰，禁用C++的名称修饰（C++允许重名，重名时会根据参数的不同使用名称修饰），这里确保汇编函数的名称与C++声明中指定的名称匹配





1. 在汇编函数中调用C函数和C++函数：

* 在asm\_utils.asm中用extern声明函数来自外部模块

extern：声明是外部函数，该函数在其他模块定义，编译器不会生成该函数的代码，需要链接包含函数定义的模块

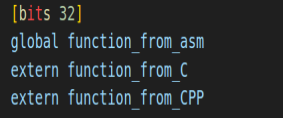
* 在c\_func.c中直接定义function\_from\_c()

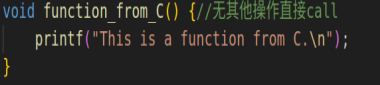
为什么function\_from\_c不需要声明为global：

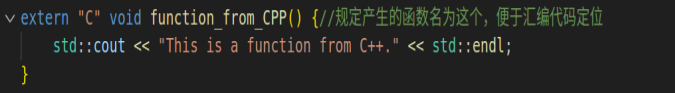
在汇编代码中使用 C 函数时，编译器负责生成必要的汇编代码，因此不需要 global 指令。而在 C 代码中使用汇编函数时，需要使用 global 指令来声明汇编函数，以便编译器可以在链接时找到它。

* 在cpp\_fun.c中要将function\_from\_CPP()用关键字extern “C”声明：

让编译器按C的规则编译，不加名字修饰，便于汇编函数能定位到函数的位置







1. Makefile文件编写：

* 编译目标main.out：

依赖文件（需要链接的文件）：main.o c\_func.o cpp\_func.o asm\_func.o

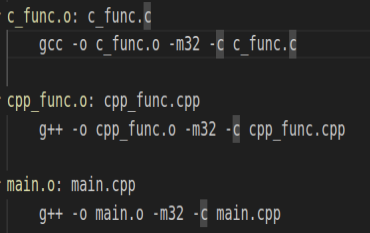
编译规则：将四个文件链接成32位代码可执行文件main.out



* 依赖文件中c文件和cpp文件的编译：

格式：生成的对象文件：依赖文件

编译规则:-c:只编译源代码，但不链接，生成目标文件（.o）而不是可执行文件。



* 依赖文件中汇编代码的编译：

编译规则：-f elf32：指定输出目标文件格式为ELF32（可执行可链接）目标文件

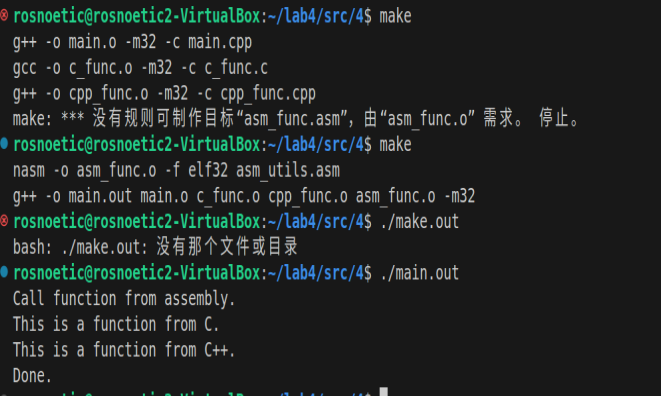


* 清除操作：

Clean目标执行的操作是rm \*.o命令，rm清除操作，\*.o所有.o后缀的文件



1. **实验结果**



1. **总结**

为什么这里makefile编译的时候没有用到ld链接？

因为gcc和g++已经包含了链接器

Assignment2

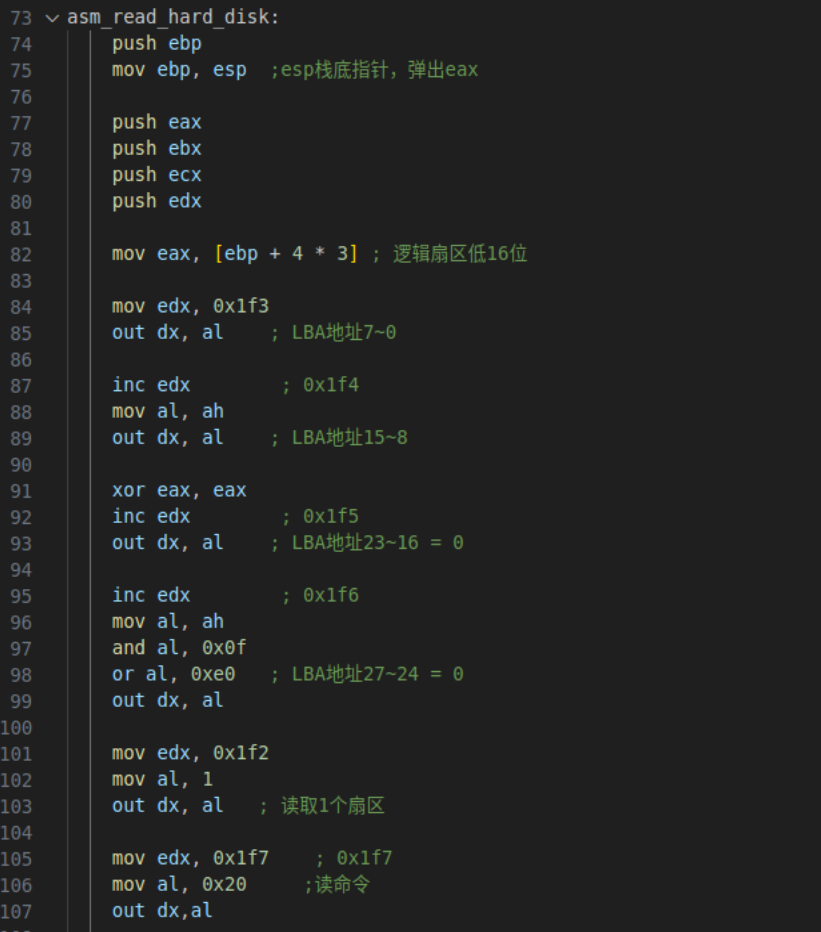
1. **实验要求**

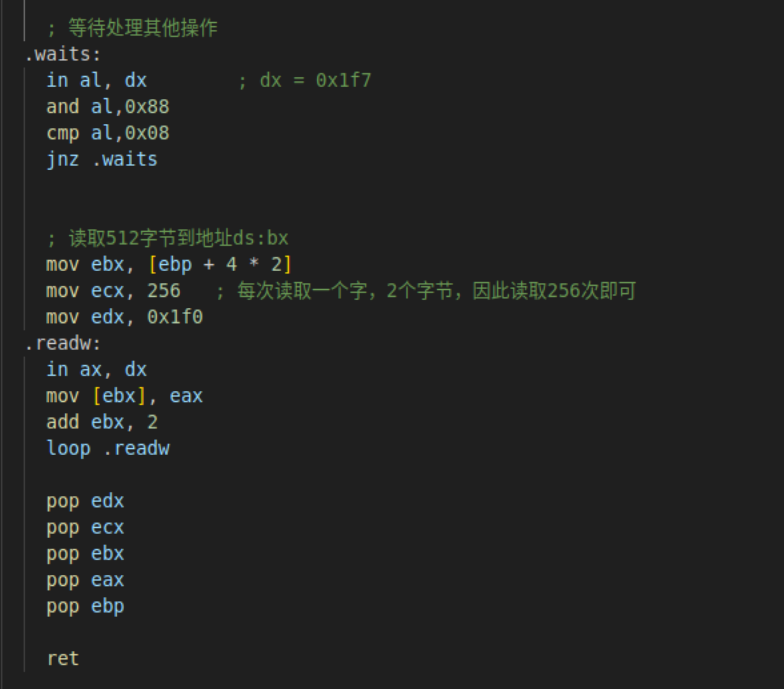
复现章节内核的加载，在进入setup\_kernel函数后，将输出 Hello World 改为输出你的学号

1. **实验过程**
2. mbr.asm的编写：通过LBA方法加载bootloader5个扇区到内存，并且跳转到bootloader的起始位置（同lab3）
3. bootloader.asm的编写：进入保护模式，并且通过LBA方法加载内核的200个扇区（位置：硬盘的第六个扇区起）到内存，跳转到内核的起始位置（指定为0x20000）
4. entry.asm（内核进入）的编写：声明来自setup.cpp

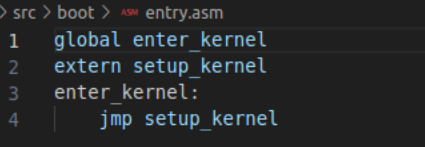
的外部函数setup\_kernel，并且调用setup\_kernel

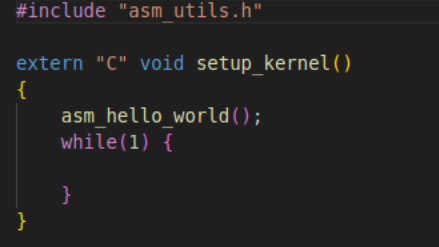
1. set.cpp的编写：声明来自asm\_utils.h的外部汇编函数asm\_hello\_world(打包在asm\_utils.h，使用声明时include该头文件)，定义setup.kernel函数，并且在其中调用外部函数asm\_hello\_world
2. asm\_utils.asm的编写：声明asm\_hello\_world为全局变量，实现asm\_hello\_world的定义
3. 编译链接文件，将bootloader和mbr和entry加载进虚拟硬盘，用qemu启动
4. **关键代码**
5. LBA方法加载硬盘





1. 汇编和C++的混合编程





1. Makefile编译链接

* 变量：（具体看图中注释）

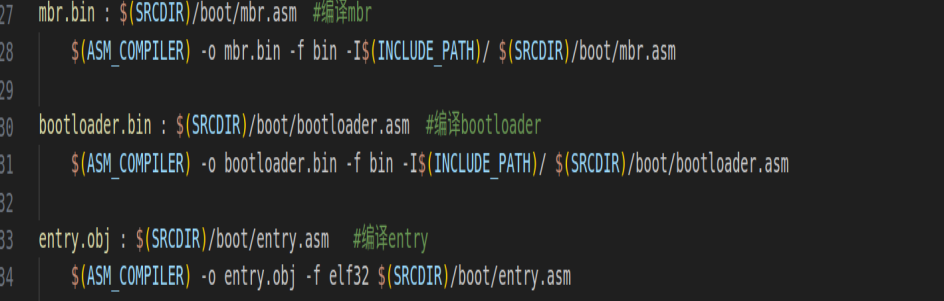
C++编译选项：

|  |  |  |
| --- | --- | --- |
| -g | -Wall | -march=i386 |
| 生成调试信息 | 启动所有警告 | I386架构 |
| -m32 | -nostdlib | -fno-builtin |
| 32位 | 不链接标准库 | 禁内联函数 |
| -ffreestanding | -fno-pic |  |
| 独立程序，不依赖外部库 | 不生成位置无关代码 |  |

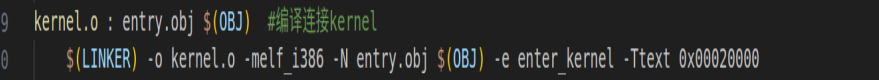


* 编译链接：
* mbr.bin和bootloader.bin直接编译
* kernel.bin：

将kernel.bin依赖的entry.asm和asm\_utils.asm和setup.cpp编译成中间文件



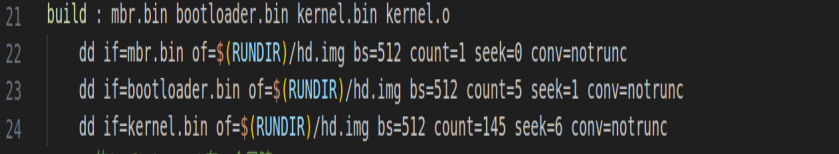
将中间文件链接成kernel.o并指定标号的起始地址（按执行顺序链接），要将内核进入点代码（entry.obj）放在所有文件前面，使二进制文件中起始地址一定是内核进入点



将kernel中取出代码段，放到kernel.bin中，丢弃掉ELF文件中的其他信息，使kernel.bin中只含有可以直接执行的指令

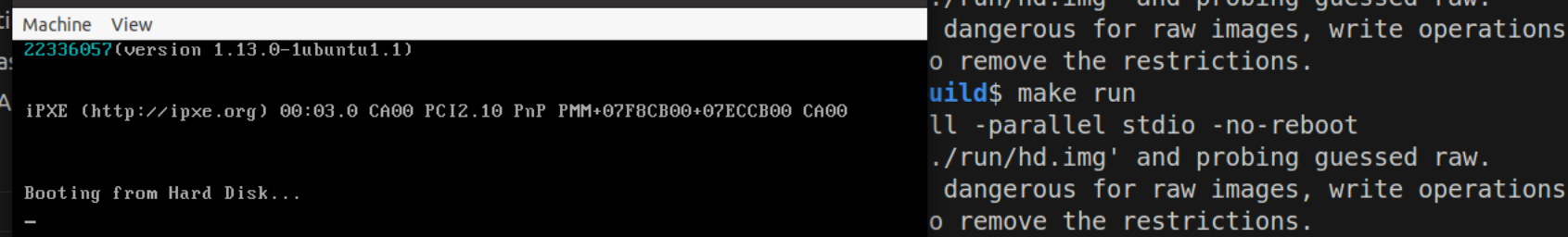


* 将bootloader，mbr，kernel写入硬盘



1. **实验结果**

成功显示学号

****

1. **总结**

.o和.bin和.obj的区别与联系？

|  |  |  |
| --- | --- | --- |
| .o | 目标文件 | 编译后产生的中间文件，未被链接 |
| .bin | 二进制文件 | 包含可执行代码或数据，存储于固件 |
| .obj | 目标文件 | 包含已经编译的代码数据，未被链接 |

Assignmet3

1. **实验要求**

复现章节初始化IDT，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

1. **实验过程**
   * + 1. 定义InterruptManger类中断处理器，管理中断描述表IDT
       2. 在InterruptManger中实现IDT的初始化：（initialize()）
2. 创建IDT，包含256个中断描述符(asm\_lidt())
3. 初始化256个中断描述符（setInterruptDescriptor()）
4. 定义默认中断处理函数：asm\_interrupt\_empty\_handler()
   * + 1. 在内核中定义初始化中断处理器实例
       2. 触发除0错误检验
       3. 自定义除0错误的0号中断描述符
5. **关键代码**
   * + 1. 在asm\_utils.asm中初始化IDTR：

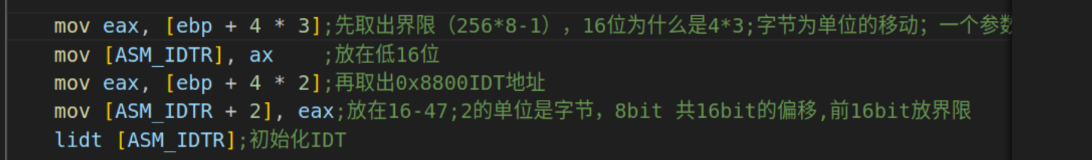
* 初始化IDR：要把中断描述符表的32位基地址和16位界限写入IDTR中
* cpp函数向汇编函数传参问题：

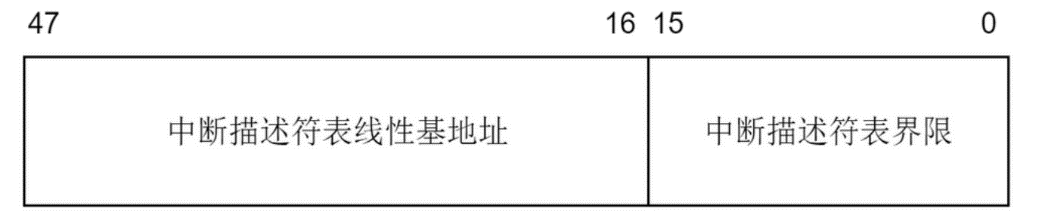
栈的结构如下：

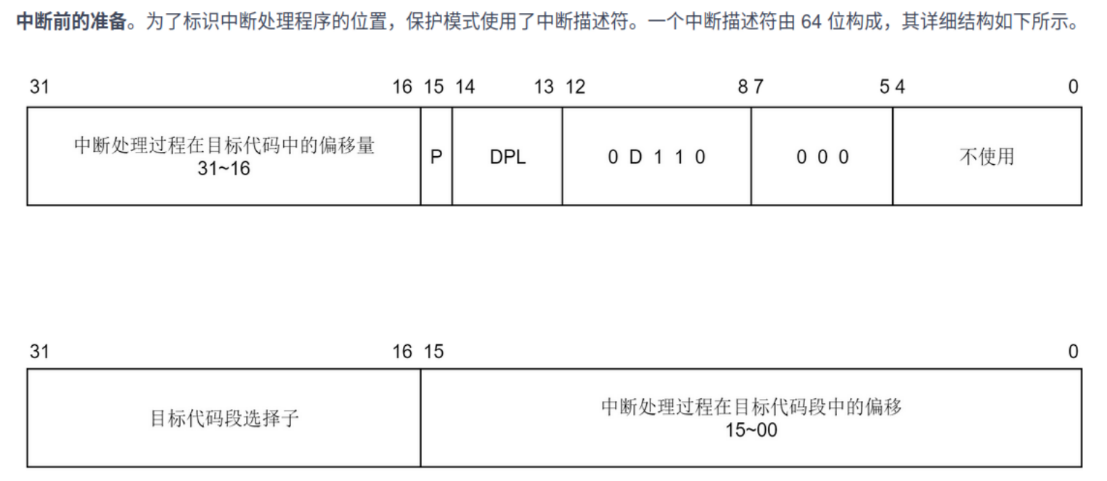
|  |  |
| --- | --- |
| 栈底：传入参数:界限信息  高地址 |  |
| 传入参数：IDT的起始地址 |  |
| 压栈的ebp | Ebp保存的指向位置 |
| 栈顶：eax  低地址 | Esp寄存器指向位置 |

提取界限：提取ebp向高地址增加12个字节后的内容

提取基地址：提取ebp向高地址增加8个字节后的内容





* + - 1. 对中断描述符的初始化：
* 低三十二位:

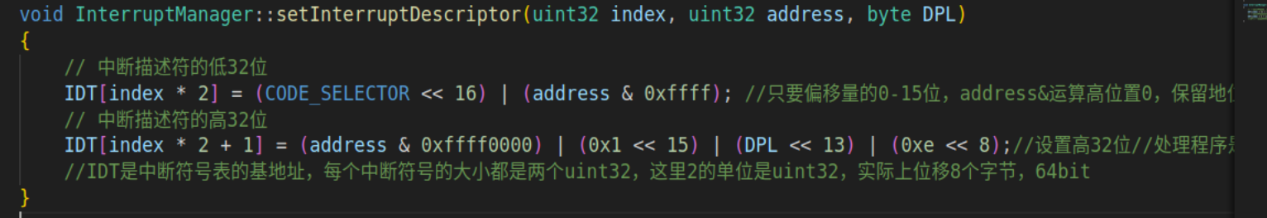
16-31位：目标代码段选择子：CODE\_SELECTOR

0-15位：处理过程在段选择子中的偏移（低15位）：(address & 0xffff)

* 高三十二位：

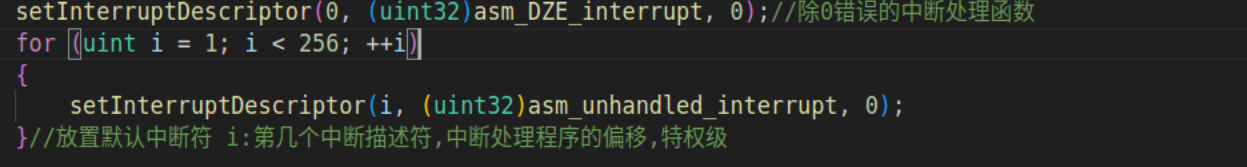
0-16位：固定

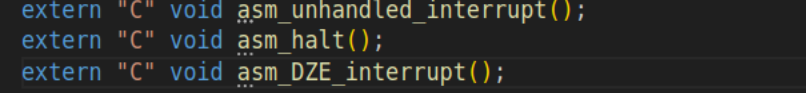
16-32位：偏移量高15位

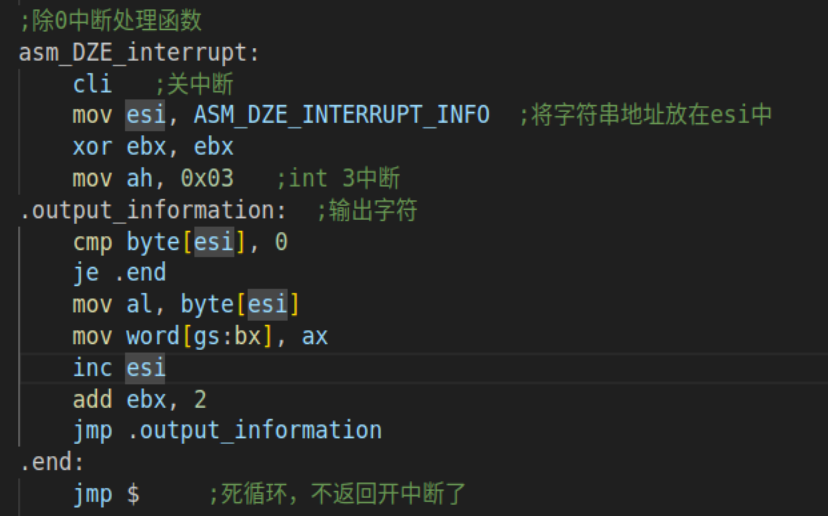


* + - 1. 自定义中断处理函数：

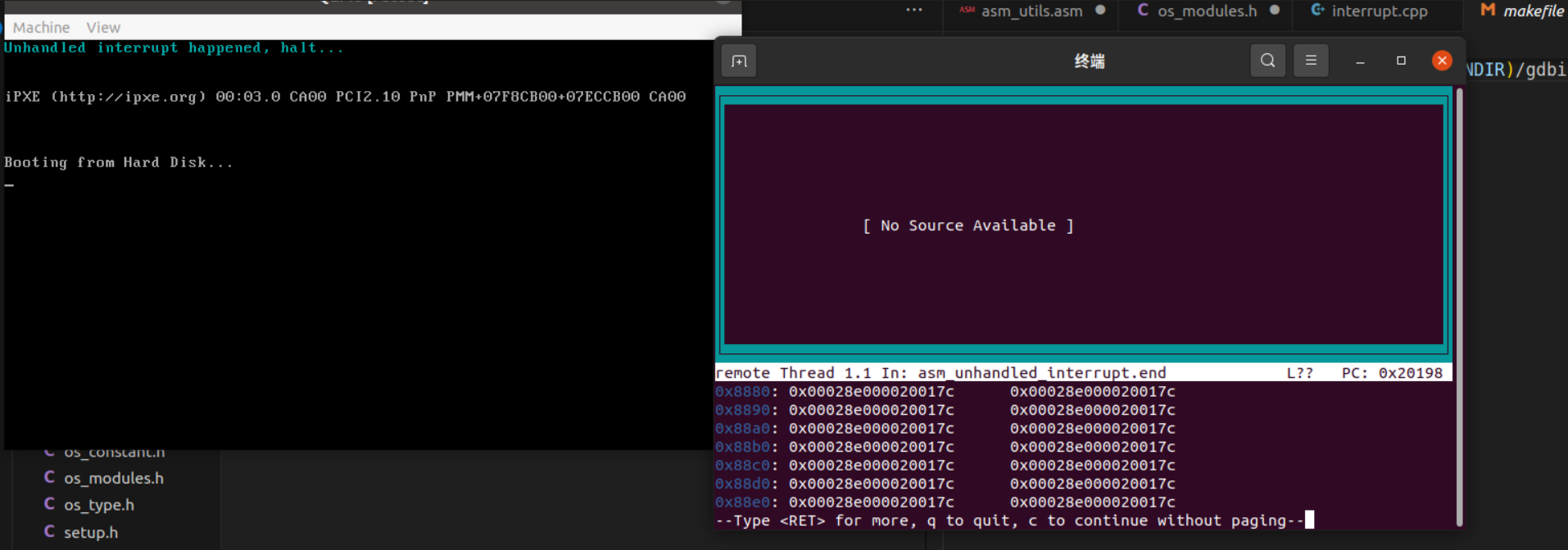
除0错误是自动找到0号中断描述符，然后跳转到其中对应的处理程序，下面自定义0号中断描述符的处理程序，使得触发除0错误时说明除0错误的出现



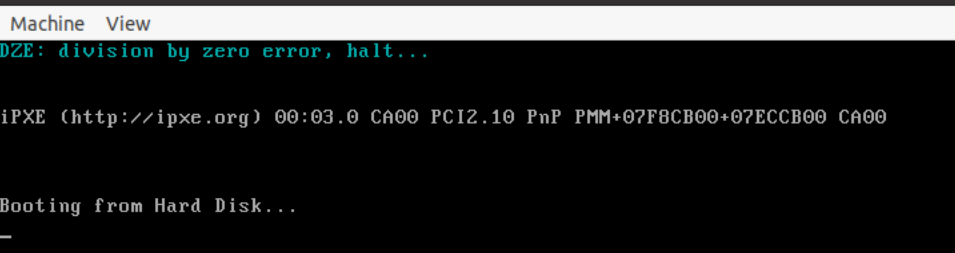




1. **实验结果**
   * + 1. IDT的初始化和触发除0错误中断



* + - 1. 自定义的中断处理函数



1. **总结**

* cpp文件中的extern符号：

声明一个变量或函数，该变量或函数在其他源文件中定义，告诉编译器在该源文件中使用该符号，即使该符号未在该源文件中定义。

* 为什么除0错误会知道触发自定义的中断处理器和特定的中断描述符？

处理器自动检测到除0错误，0-31号错误中断处理器是可以自动检测的，无需自定义函数检测。

* Cpp函数向汇编函数传参问题：

Cpp传入参数在汇编函数里从右到左压栈，且esp是栈顶指针

Assignment4：

1. **实验要求**

复现章节**8259A编程——实时钟中断的处理**， 通过时钟中断，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. 中断处理器初始化

* 初始化中断描述表
* 初始化8259A芯片：ICW1(ICW的相关设置)，ICW2（设置芯片上中断的对应中断向量号），ICW3（设置主从芯片的连接），ICW4（设置芯片所在框架和相关设置EOI），OCW1（设置相关中断屏蔽）

1. 定义打印输出类stdio
2. 定义中断处理函数c\_time\_interrrupt\_handler

asm\_time\_interrupt\_handler

1. 设置时钟中断描述符和相关对时钟中断的控制函数
2. **关键代码**
3. 初始化8259芯片

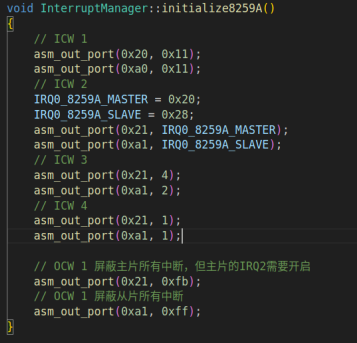
ICW1：I为1，C为0，W为0

ICW2：主片起始中断向量号0X20，从片0x28

ICW3：主片从片连接的端口是主片的IRQ2

ICW4：关EOI，开80x86

OCW1：初始化屏蔽所有中断



1. 定义stdio打印输出类：

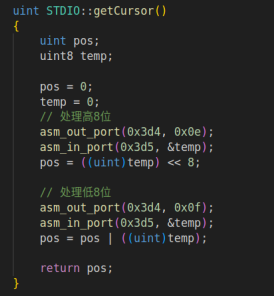
* Print型函数：

参照lab2原理中往0xb800地址写入信息从而操作屏幕显示

* Cursor函数：

操作光标的端口是0x3d5（读取或者写入光标位置信息），0x3d4（定义对0x3d5端口是高位还是低位的操作）

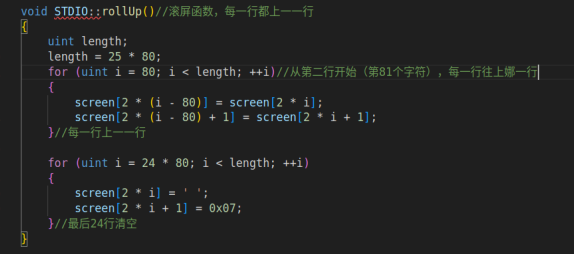
movecursor()：对0x3d5写入要移动到的位置，getcursor():从0x3d5读取当前光标位置





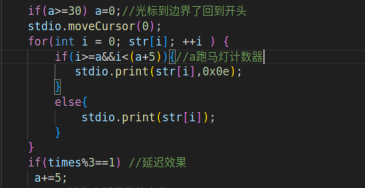
* rollup函数(滚屏函数)：

屏幕的显示数据，从第二行开始每一行往上挪动一行，最后一行全部清空



1. 中断处理函数（实现跑马灯效果）：

每次中断，光标从（0，0）位置打印字符串，其中有5个字符会显示不同颜色，按顺序变换显示颜色的5个字符，实现跑马灯效果

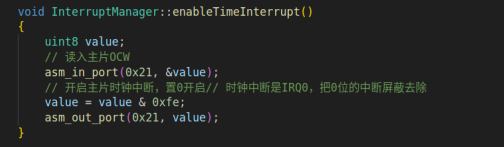


完整的中断处理函数（要保存和恢复现场和利用OCW2端口发送EOI信号）

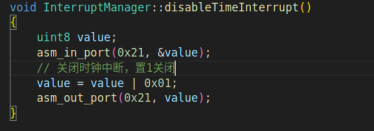


1. 时间中断描述符和时间中断相关的函数：

打开时间中断：

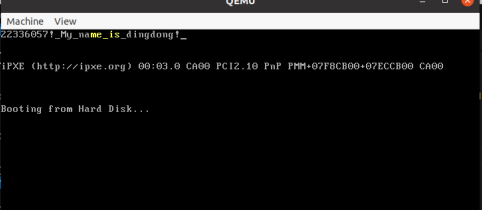
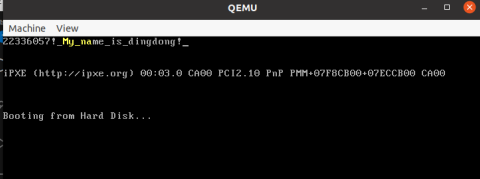


关闭时间中断：



1. **实验结果**

**实现跑马灯效果**



1. **总结**

时间中断触发和处理的流程：

1. 在打开中断和时间中断后，每一次时钟中断发送时间中断请求信号到8259A的IR0，发送IR0对应的中断向量号到CPU处理。
2. 然后根据中断向量号在中断描述表中找到时间中断描述符。根据时间中断描述符中的信息找到时间中断的处理函数，跳转到中断处理函数执行。
3. 在中断处理函数中，保存断点，保存现场，执行中断处理程序，恢复现场，返回。