

操作系统原理实验报告

**实验名称:** 实验四 中断

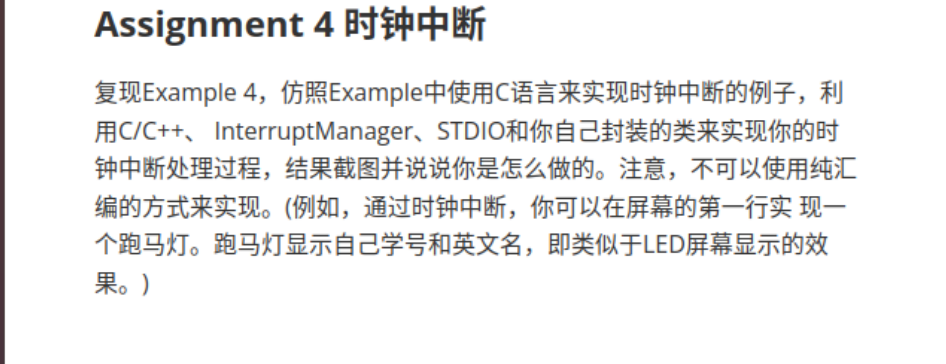
**授课教师：** 张青

**学生姓名:**

**学生学号:** 2233

1. **实验要求**





1. **实验过程**

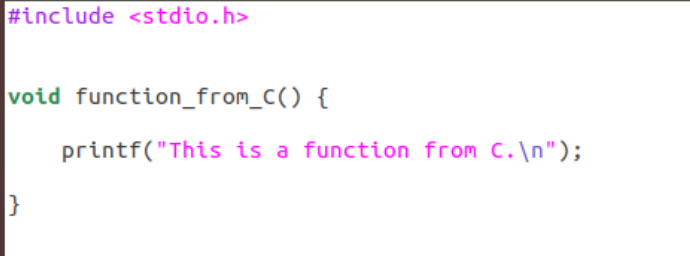
（1）Assignment 1

结合教程中example1示例代码理解关键字：“global”的作用是使汇编符号对链接器可见，也就是说所在代码链接的代码后续都可见到这个名字；“extern”的作用表示应用了文件外部的变量，告诉编译器它所声明的函数和变量已在其他文件中定义，后续作为全局变量使用；extern "C"

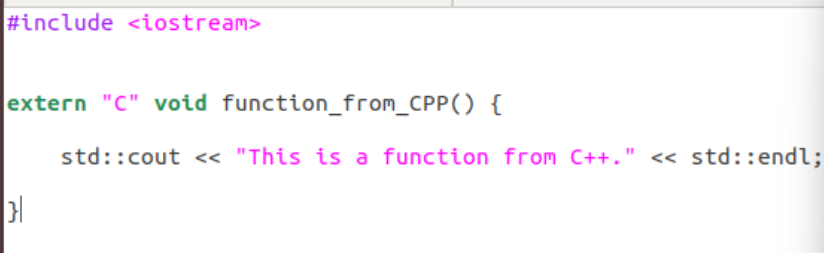
告诉编译器此语句块按 C 的特性编译和链接。因为 C++ 支持重载，因而函数在符号表中的名字与 C 中函数的不一样。为了能让 C 代码调用此函数，需要此语句。在 C++ 中引用 C 语言中的函数和变量时需要换用 C 编译和链接的方法，一般使用这个语句。

1.1） 复现example 1

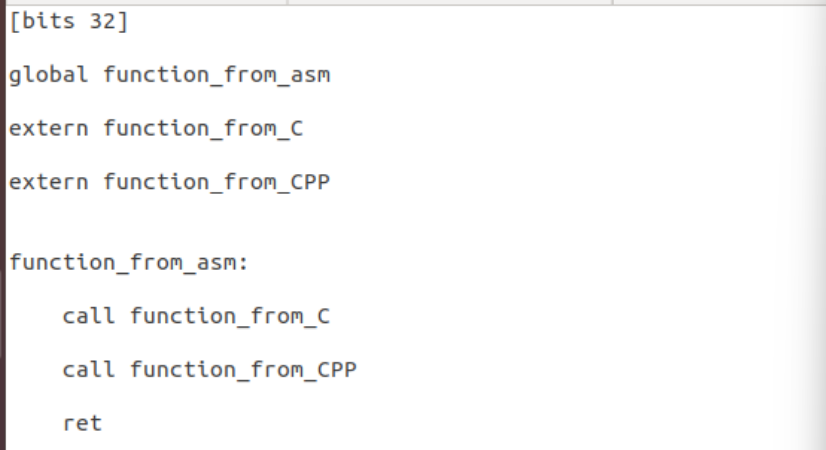
在文件c\_func.c中定义函数function\_from\_C:



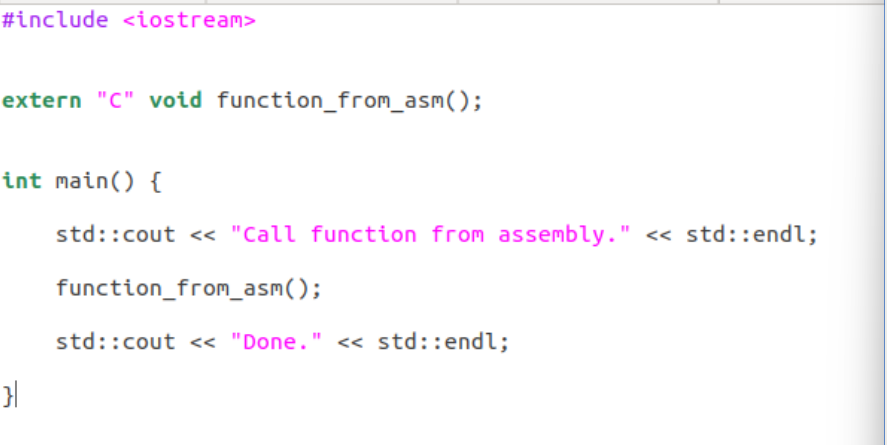
在文件cpp\_func.cpp中定义函数function\_from\_CPP:



在文件asm\_func.asm中定义汇编函数function\_from\_asm,其中调用function\_from\_C和function\_from\_CPP:



在main.cpp中调用汇编函数function\_from\_asm:



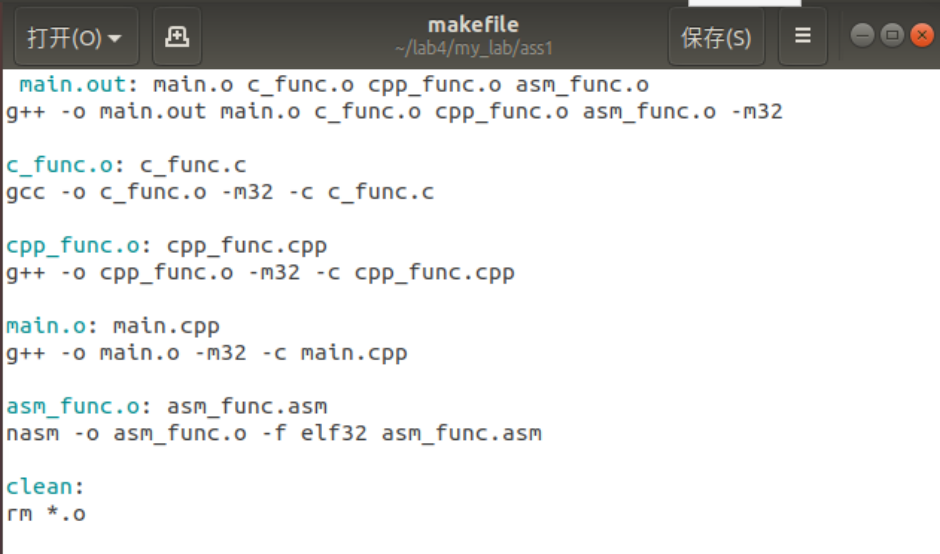
编译文件：



执行后结果见相应实验结果处。

1.2）使用make来构建example 1

编写makefile文件：



使用make构建项目：执行命令行make && make run得到如实验结果所示。

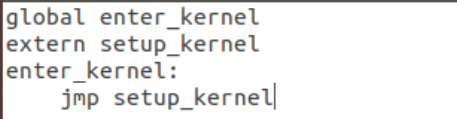
（2）Assignment 2

为了在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，输出相应内容，我们首先需要学会转变我们的工作目录环境为：

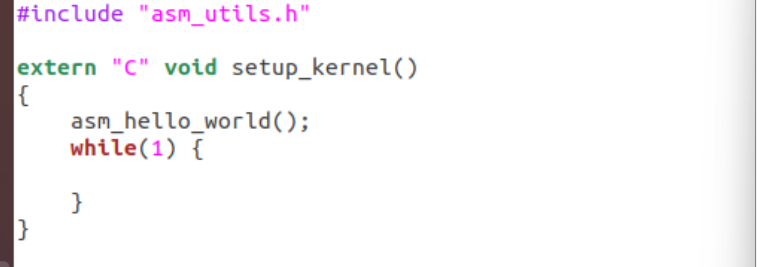


如example 2所示为bootloader中加入读取内核的代码，并将常用的定义放入boot.inc下，根据生成内核二进制文件的基本步骤完善bootloader。

编写内核，首先我们在entry.asm下定义内核进入点：



然后我们编写上述代码即将要跳转到的setup\_kernel函数（放置于setup.cpp中）：



而相应的汇编代码asm\_hello\_world函数被放于asm\_utils.h中此处我将helloworld改为自己的学号显示了，我们在asm\_utils.h中声明所有汇编函数，到此实现了example 2的代码复现，我们利用build文件夹中的makefile文件进行编译执行，结果见实验结果板块。

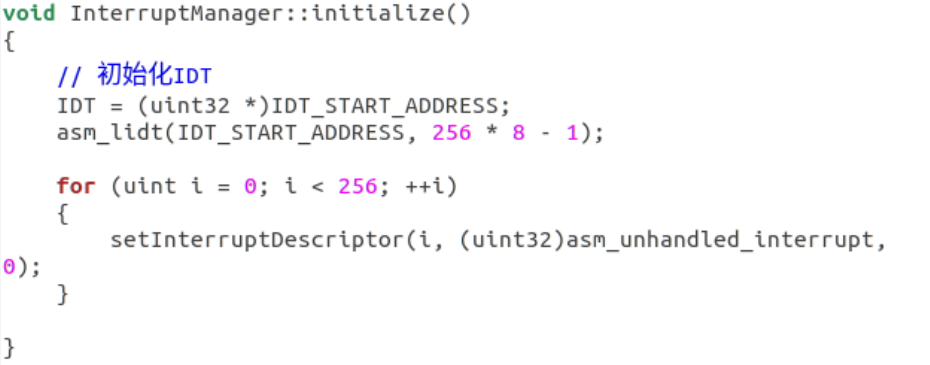
（3）Assignment 3复现example 3初始化IDT

明确我们的目标：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。

我们定义一个称为中断管理器的类InterruptManager，放置在interrupt.h中：



使用中断前我们需要做的第一步是初始化IDT，在interrupt.cpp中添加函数InterruptManager::initialize：



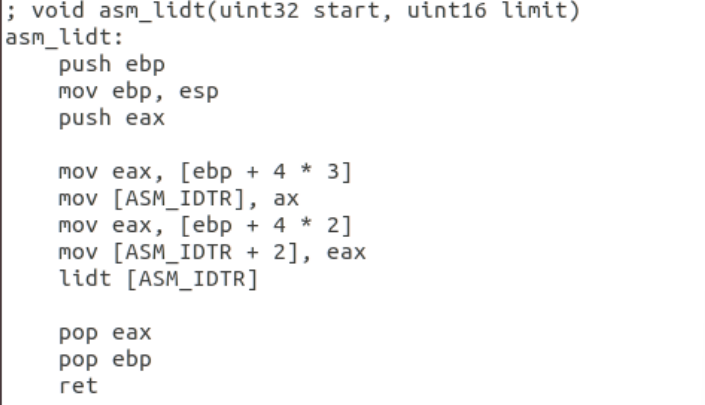
其中我们实现了设置IDTR并初始化了256个中断描述符，设置IDT在地址在地址 0x8880 处，即 IDT\_START\_ADDRESS=0x8880 。为了使CPU能够找到IDT中的中断处理函数，我们需要将IDT的信息放置到寄存器IDTR中。当中断发生时，CPU会自动到IDTR中找到IDT的地址，然后根据中断向量号在IDT找到对应的中断描述符，最后跳转到中断描述符对应的函数中进行处理.

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：

表界限=8\*256-1

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，我们进行初始化IDTR。IDTR的初始化需要用到指令 lidt 。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，代码添加到asm\_utils.asm 中，如下所示:



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。在我们的实验中，对于中断描述符，有几个值是定值：

P=1表示存在。

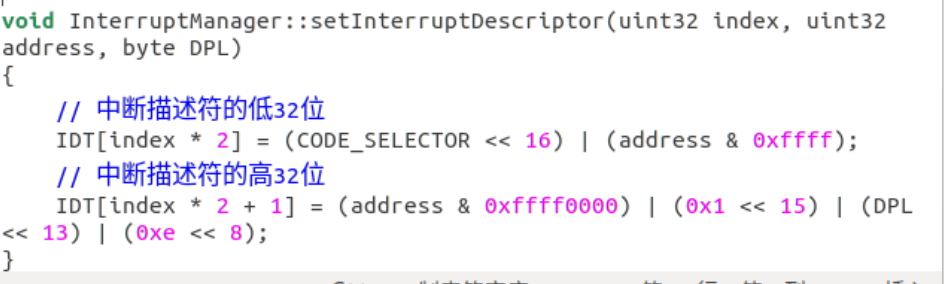
D=1表示32位代码。

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。

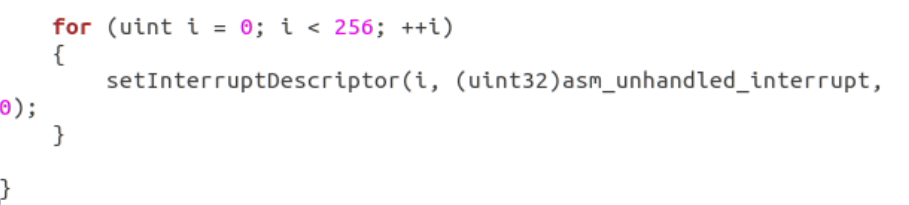
因此，从目前来看，不同的中断描述符间变化的只是中断处理程序在目标代码段中的偏移。由于我们的程序运行在平坦模式下，也就是段起始地址从内存地址0开始，长度为4GB。此时，函数名就是中断处理程序在目标代码段中的偏移。

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



定义一个默认的中断处理函数asm\_interrupt\_empty\_handler，具体代码见关键代码处，实现关中断输出提示字符串然后死循环/

最后我们调用我们调用 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：



相应完善一下其他细节，我们进行debug过程。

进入build文件夹运行make build &&make debug

进入gdb后运行：

**b 10 //设置断点**

**c //往下执行**

**x/256gx 0x8880**

（4）Assignment 4 时钟中断

4.1)复现example 4

在任务3的基础上我们对8529A芯片进行编程，添加处理实时钟中断的函数，函数在第一行显示目前中断发生的次数。

在中断控制器中添加如下成员变量和函数：



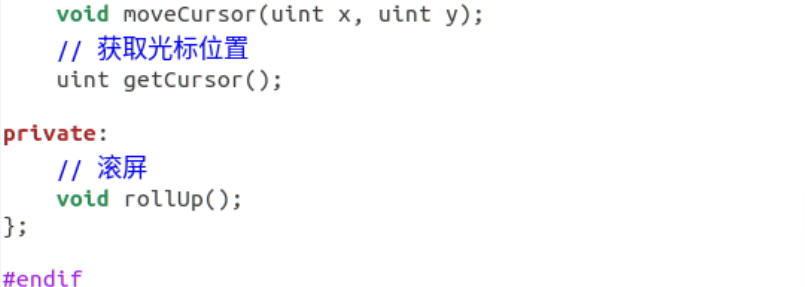


对8259A芯片初始化，即设置一系列ICW字来完成，并且屏蔽主片和从片的所有中断，因为我们实验中没有建立处理8259A中断的任何函数，代码与example 4示例代码一致，此处不展示。

处理时钟中断，步骤如下：编写中断处理函数、设置主片IRQ0中断对应的中断描述符，开启时钟中断，开中断。

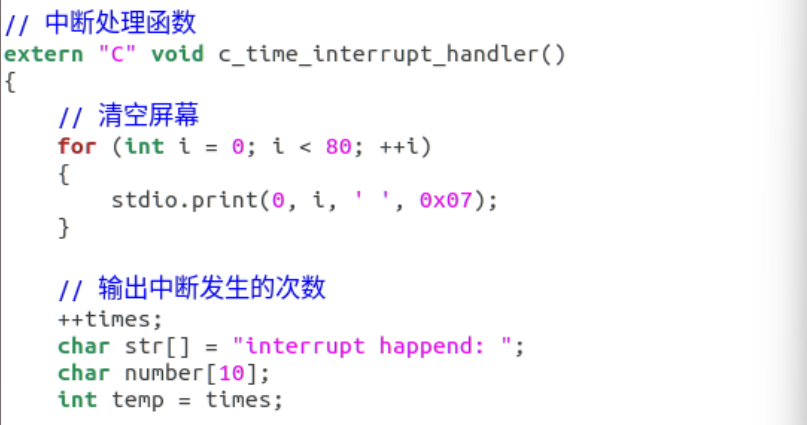
编写中断处理函数：封装一个能够处理屏幕输出的类STDIO：





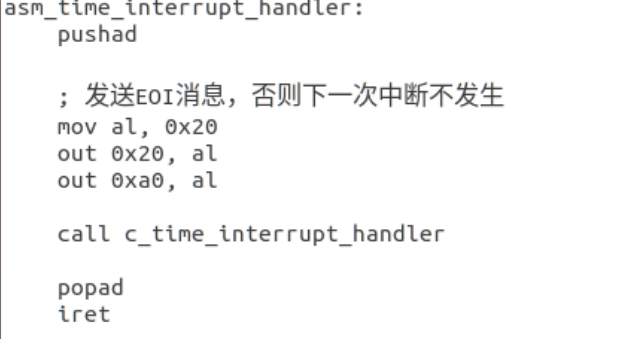
相应去实现类中各个函数，此处代码采用和example 4示例代码一致的实现。

接下来我们定义中断处理函数c\_time\_interrupt\_handler：

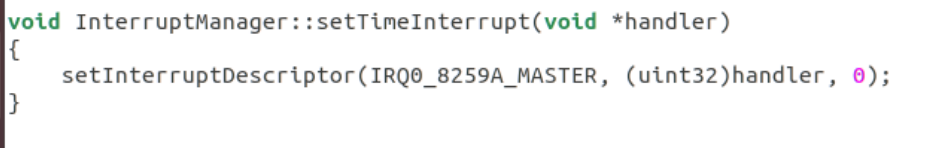




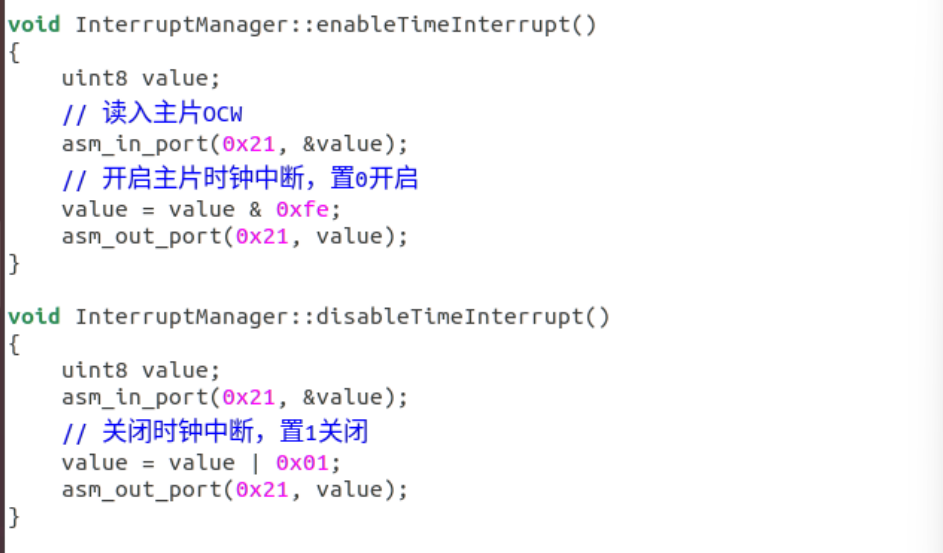
其中times是用于计数的全局变量，由于C语言缺少可以编写一个完整的中断处理函数的语法，因此当中断发生后，CPU首先跳转到汇编实现的代码，然后使用汇编代码保存寄存器的内容。保存现场后，汇编代码调用 call 指令来跳转到C语言编写的中断处理函数主体。于是再用汇编实现完善此函数成为一个完整的时钟中断处理函数如下：



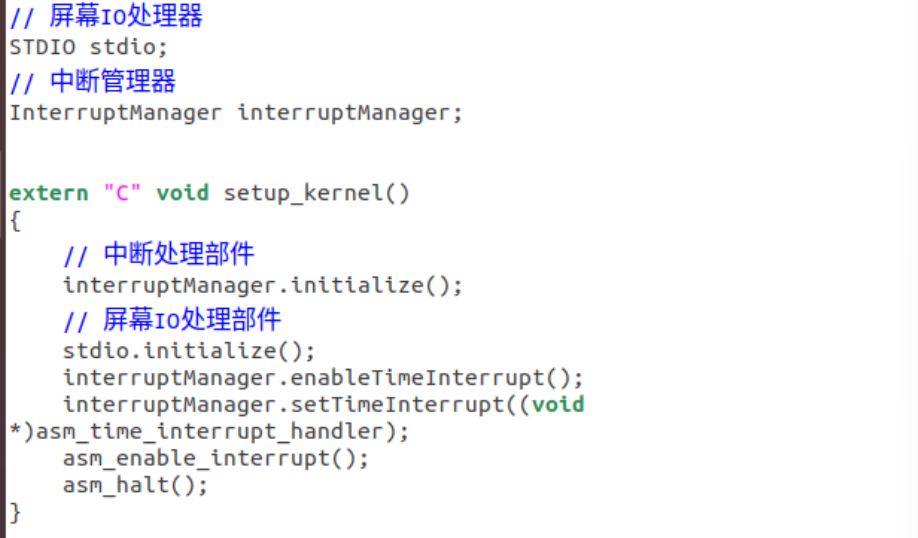
接下来我们进行时钟中断的中断描述符设置：

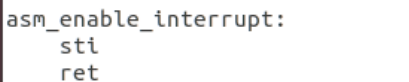


封装开启和关闭时钟中断的函数：



再完善一下相应设置，初始化内核组件，开启时钟中断和开中断：





编译运行代码得到结果。

4.2）时钟中断实现学号跑马灯

更改默认中断函数如关键代码所示以实现学号跑马灯实现，其余代码保持不变，运行后得到相应结果。

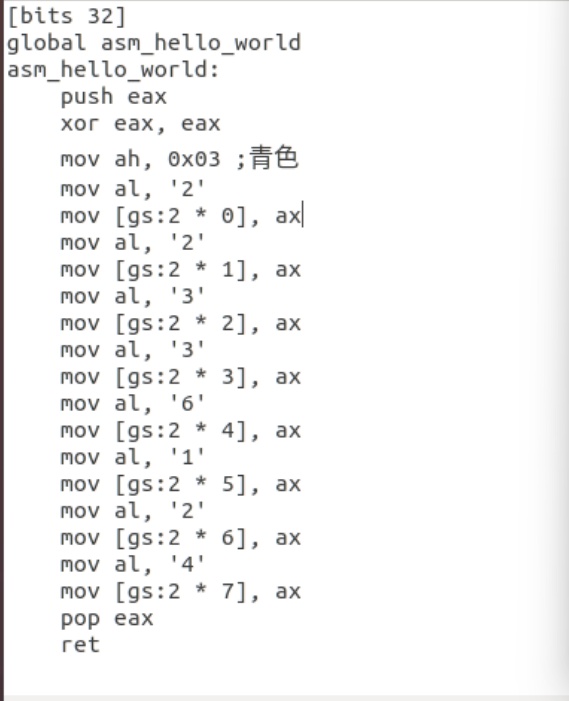
1. **关键代码**

**（1）assignment 1：**

1.1与1.2都详见相应实验过程。

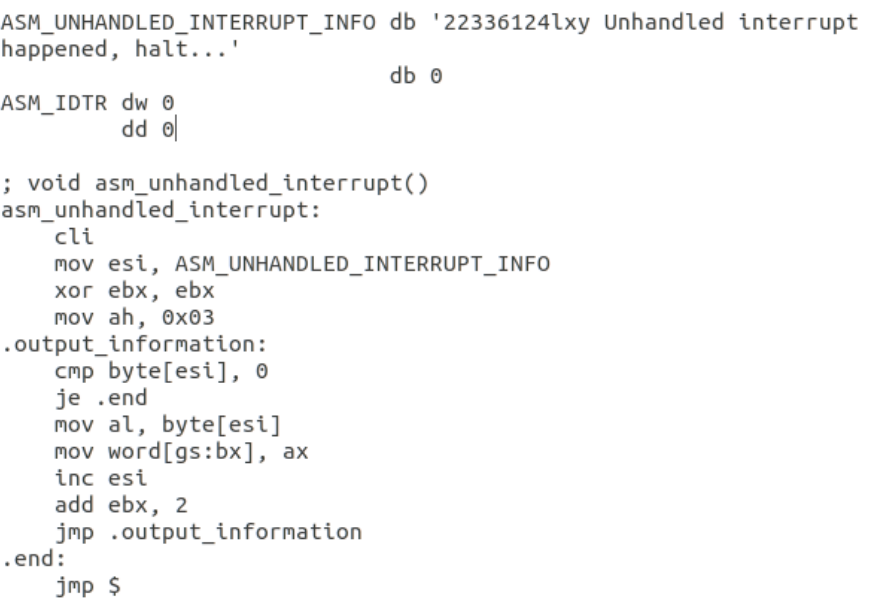
**（2）assignment 2：**

学号显示汇编函数如下：

****

**（3）assignment 3**

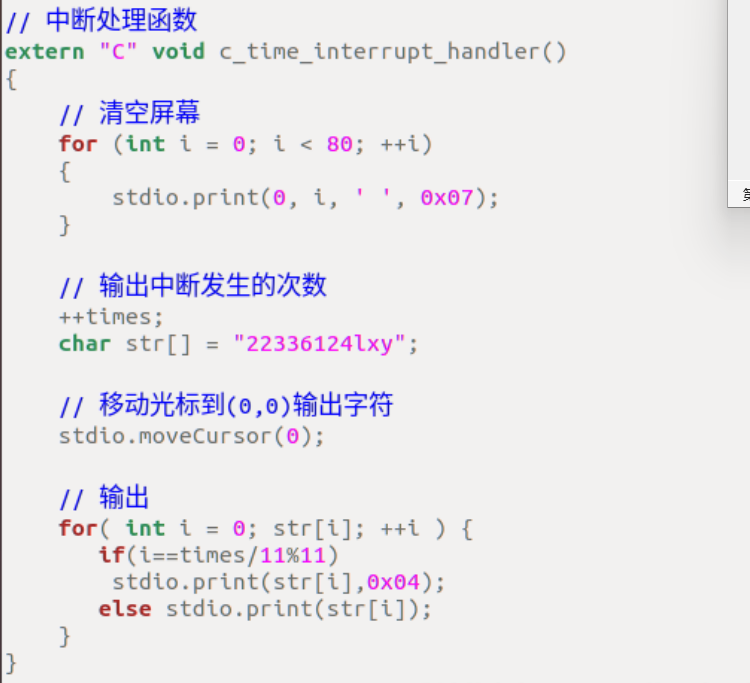
更改后的默认中断函数：

****

**（4）assignment**

4.1的关键代码见实验过程；

4.2的中断处理函数改为：

****

以上代码实现了跑马灯的实现，此处我理解的跑马灯为平时不“亮”（白色），轮转亮灯（红色），每当中断11次滚动亮一个字符，循环中times先除11主要是想使得实验结果更易于观察又不破坏函数功能。

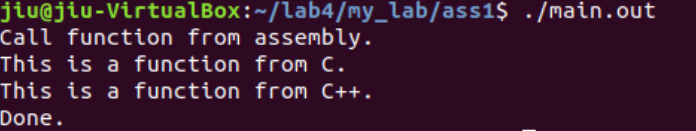
1. **实验结果**

**（1）assignment 1**

1.1运行结果：

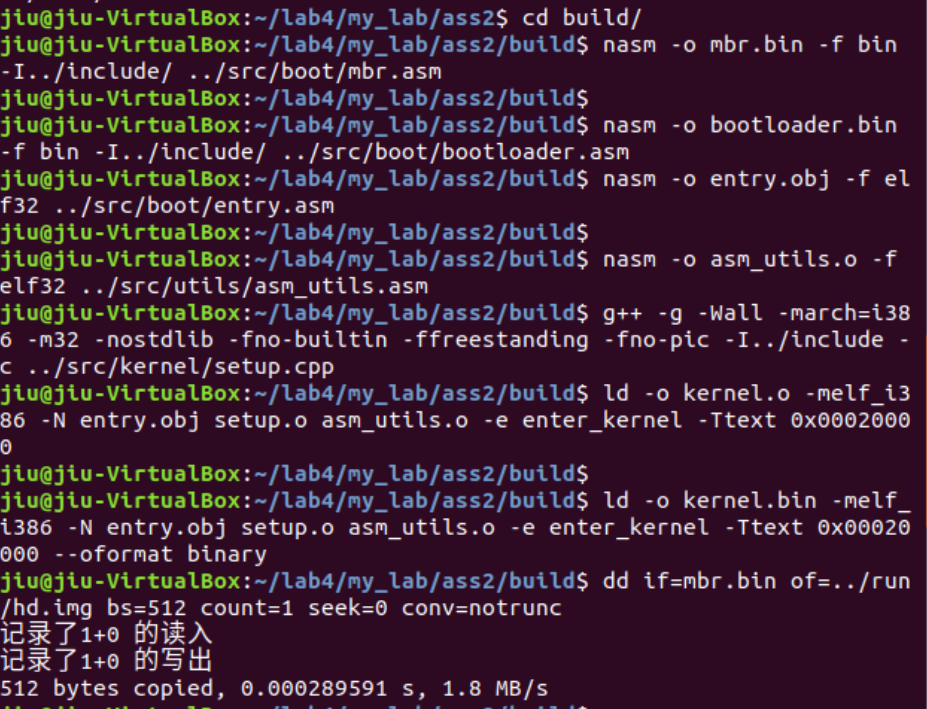
****

1.2运行结果：

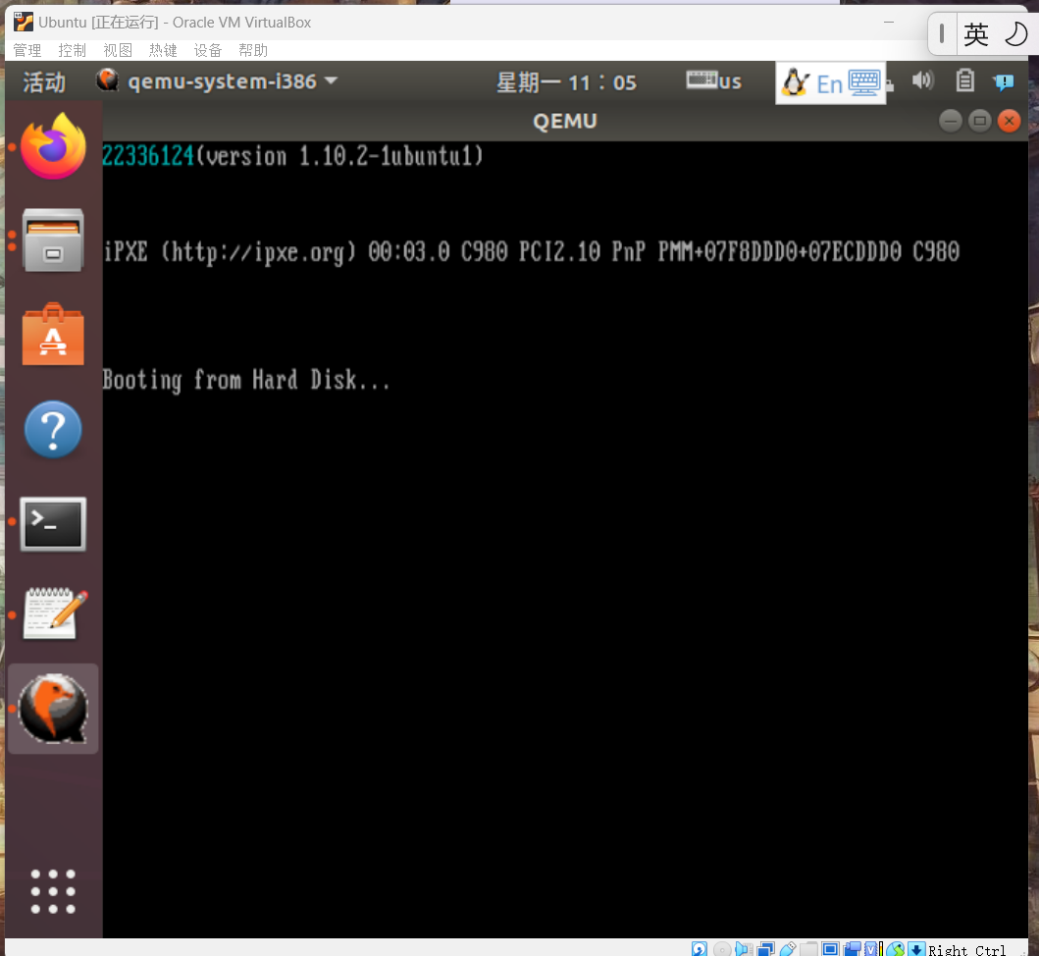
****

**（2）assignment 2**

编译运行结果：

****

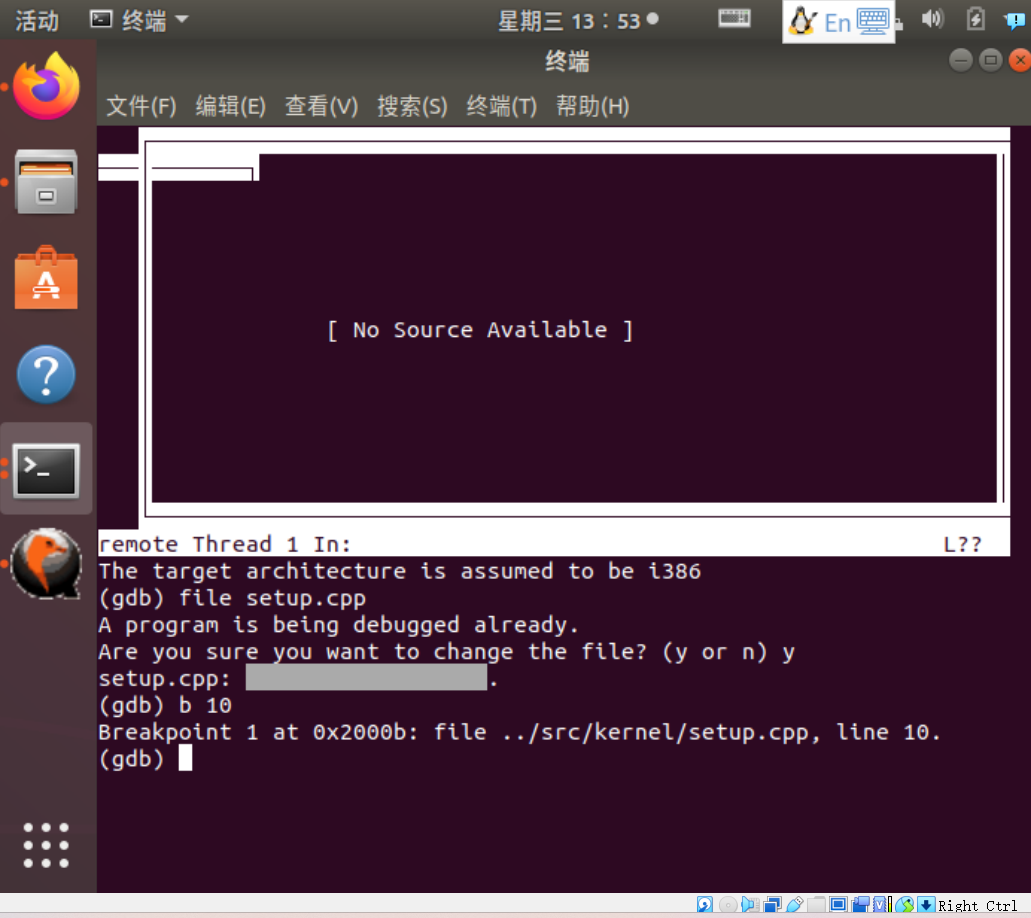
****

****

**（3）assignment 3**

Debug过程截图：

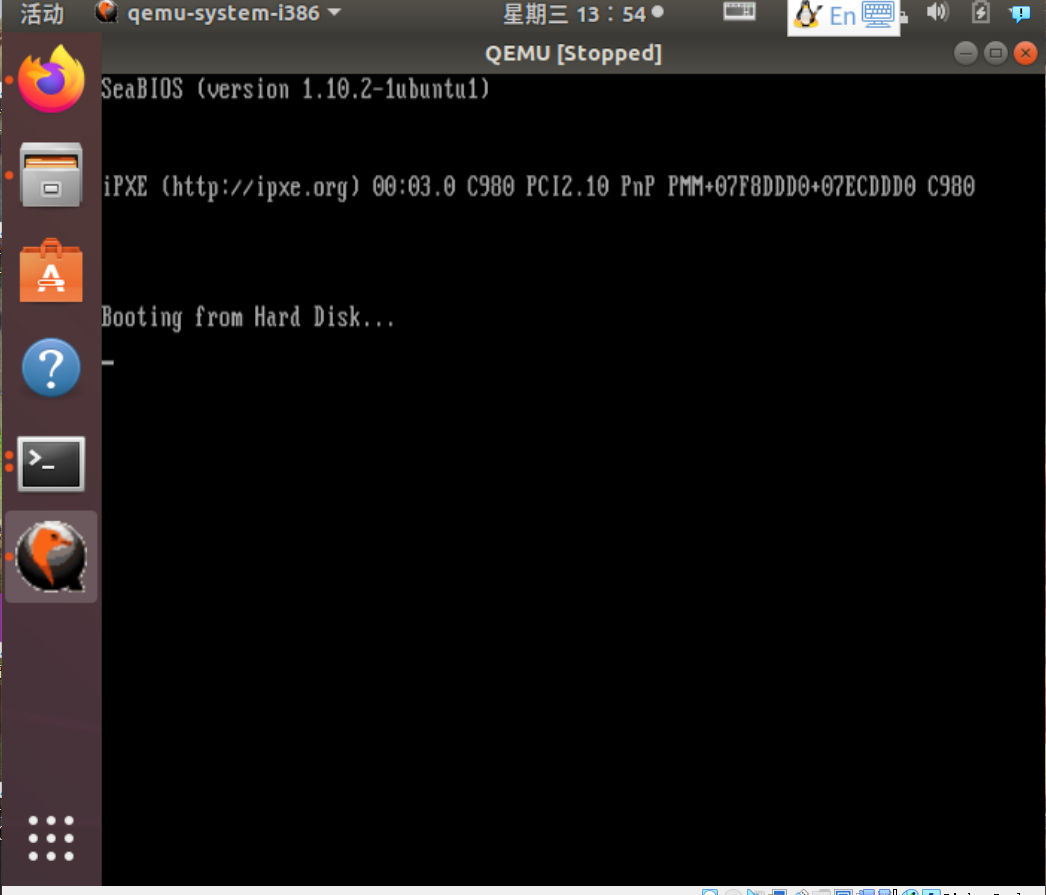
设置断点**：**

****

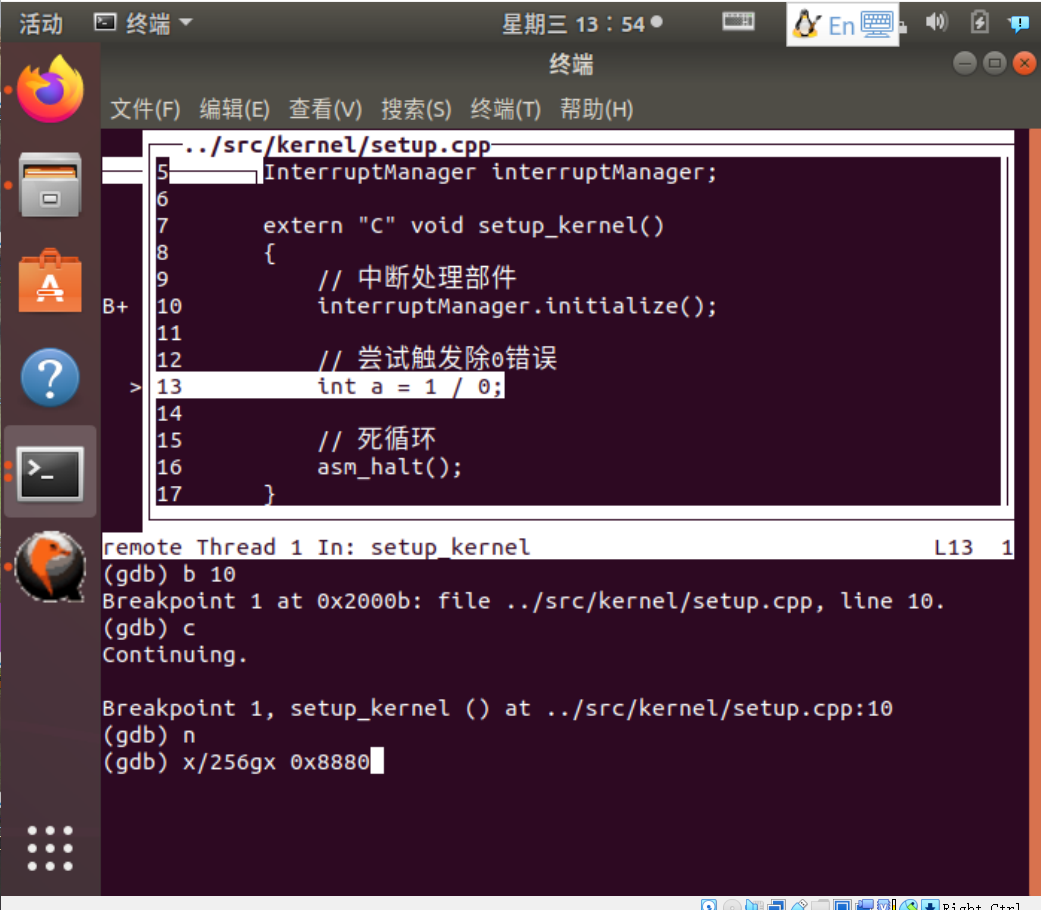
向下执行至断点处：

****

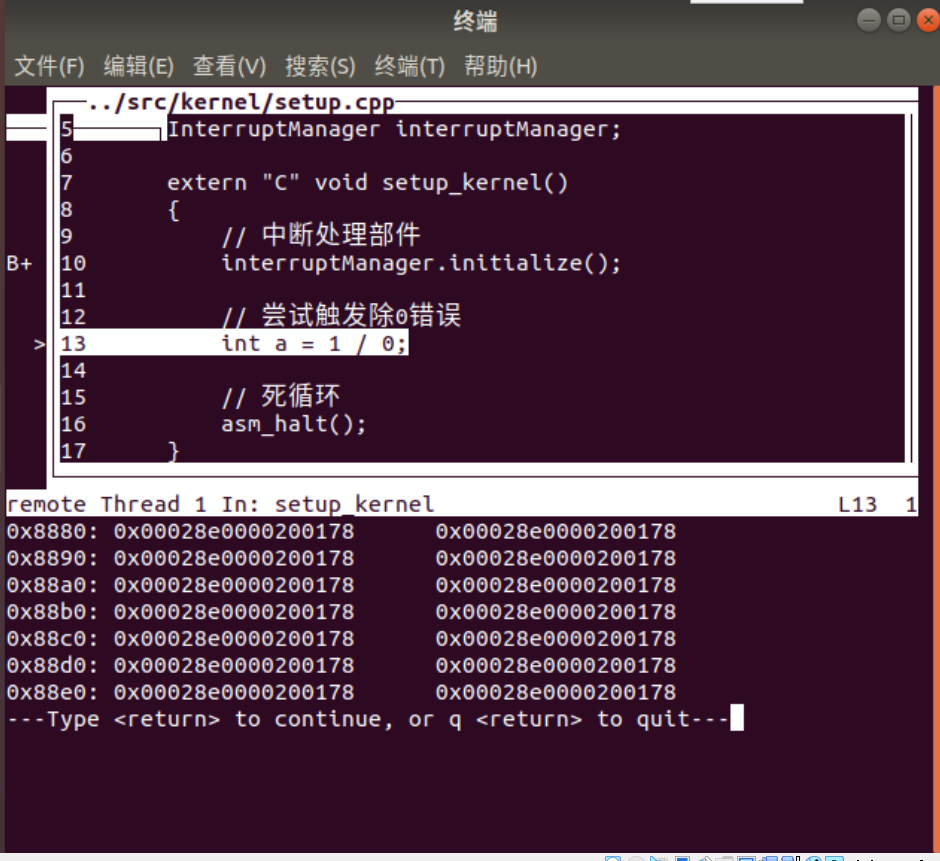
此时的qemu显示：

****

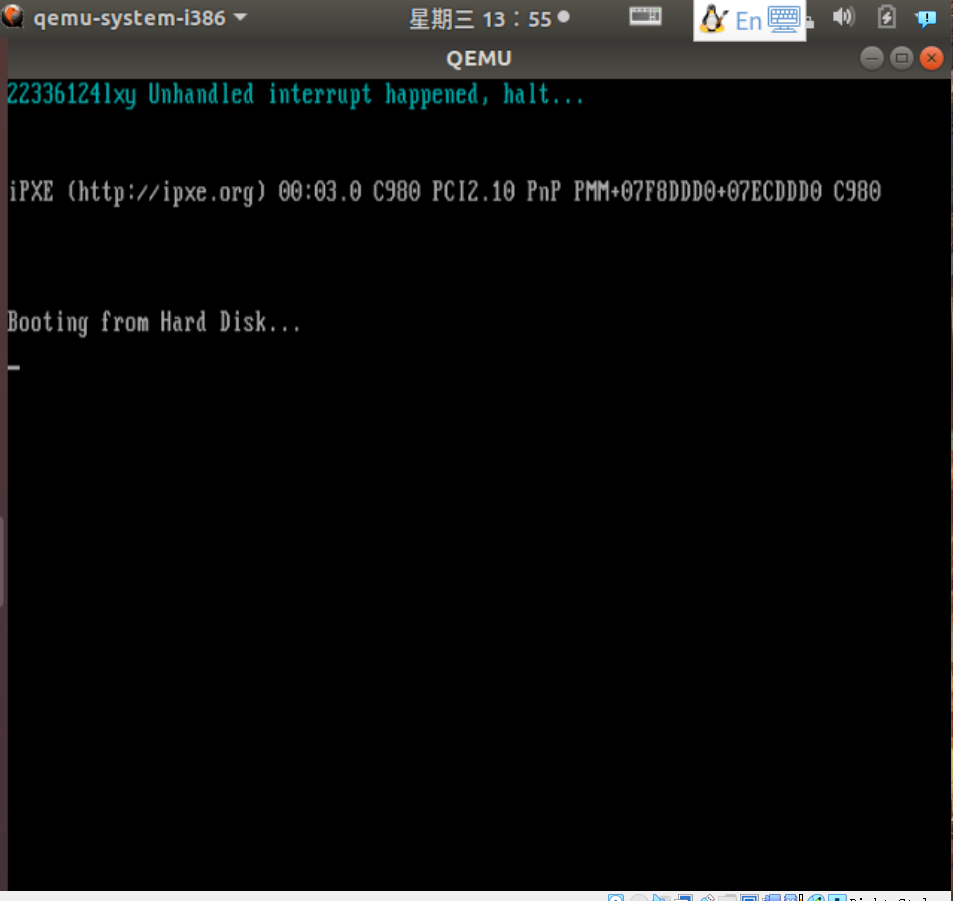
继续执行语句:

****

Debug结果：

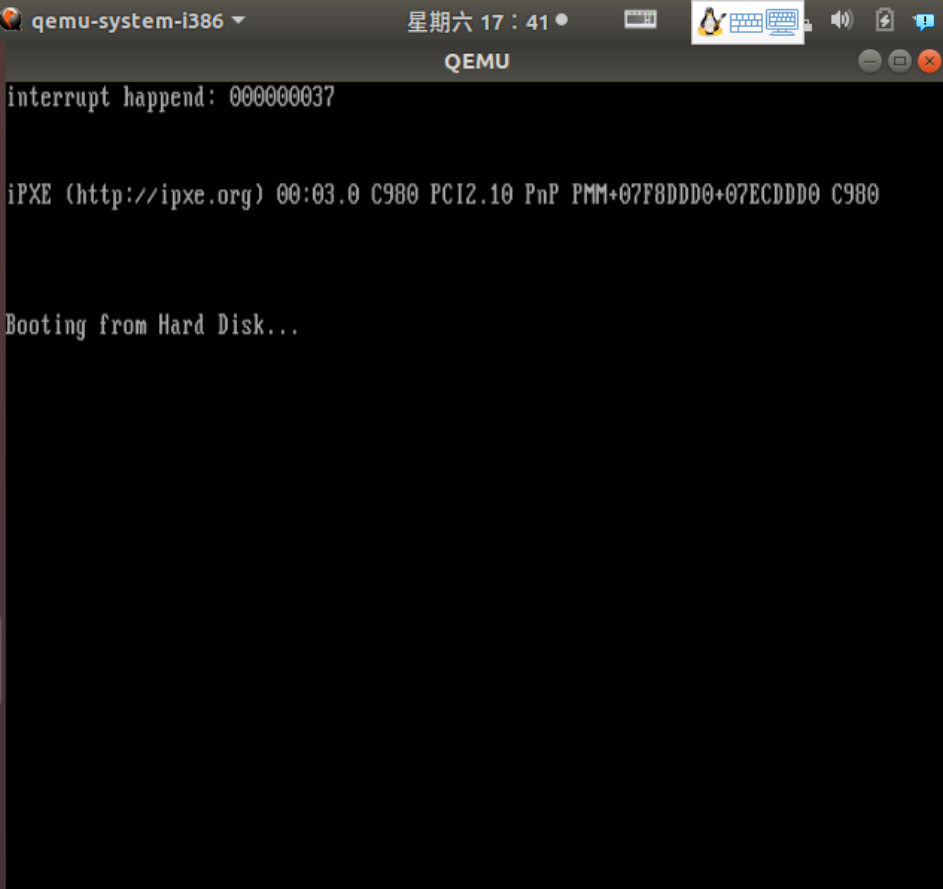
****

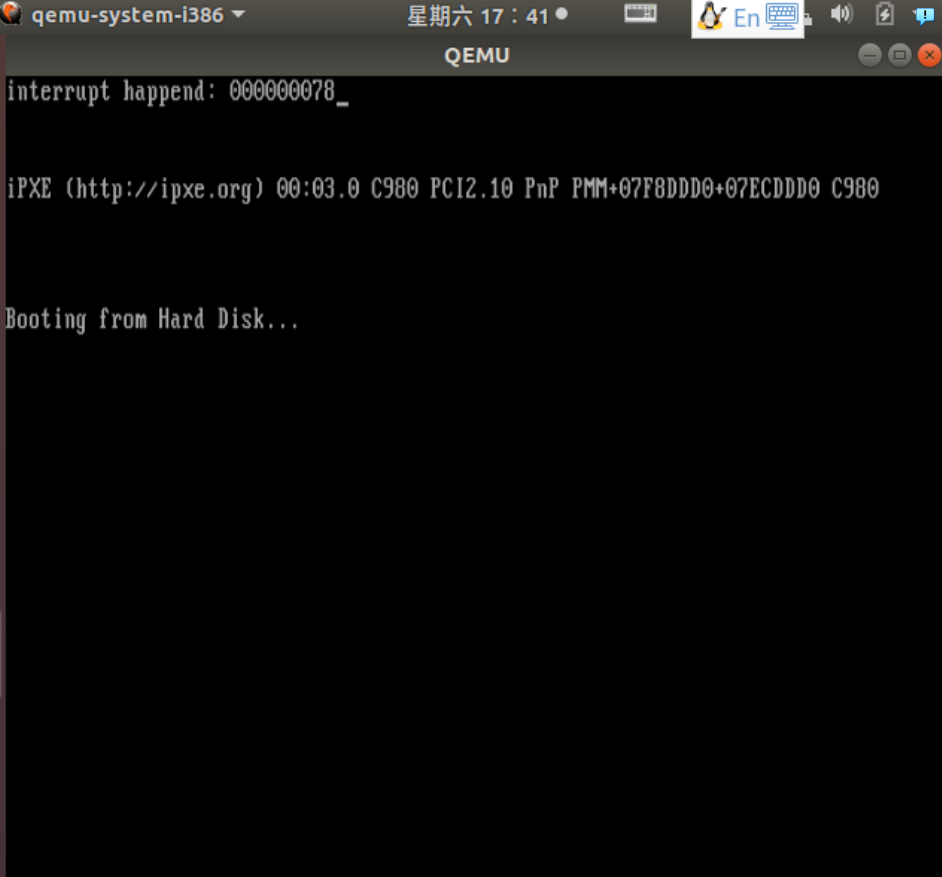
最终实验结果：

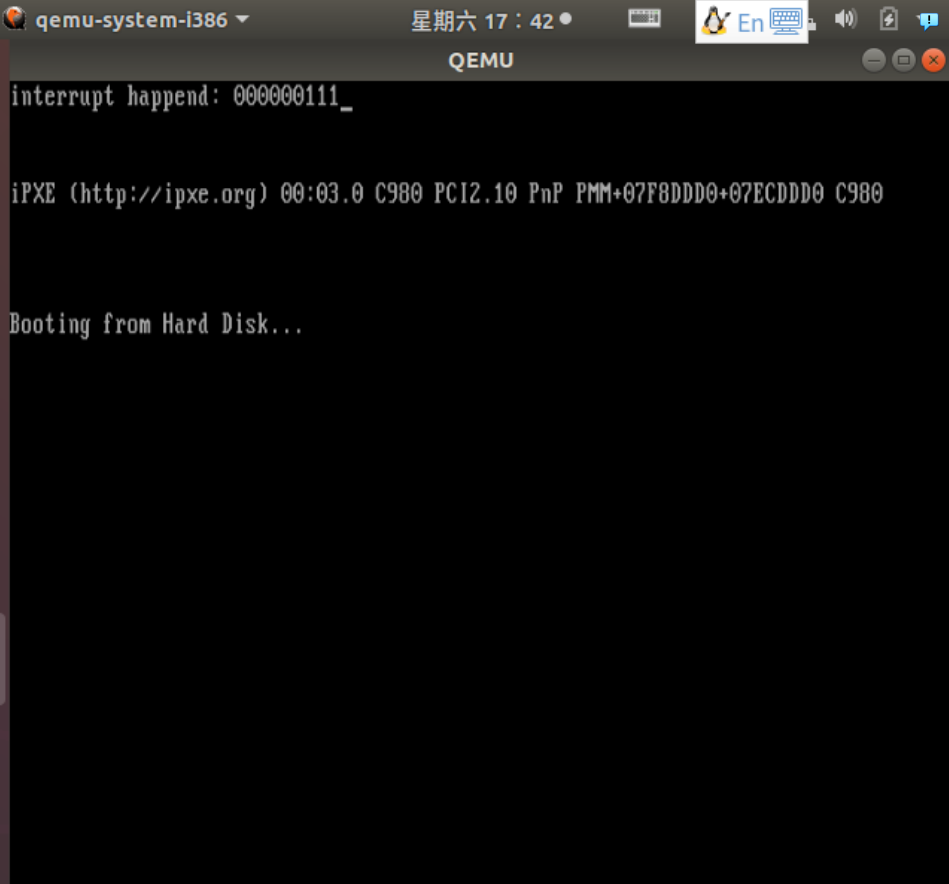
****

**（4）assignment 4**

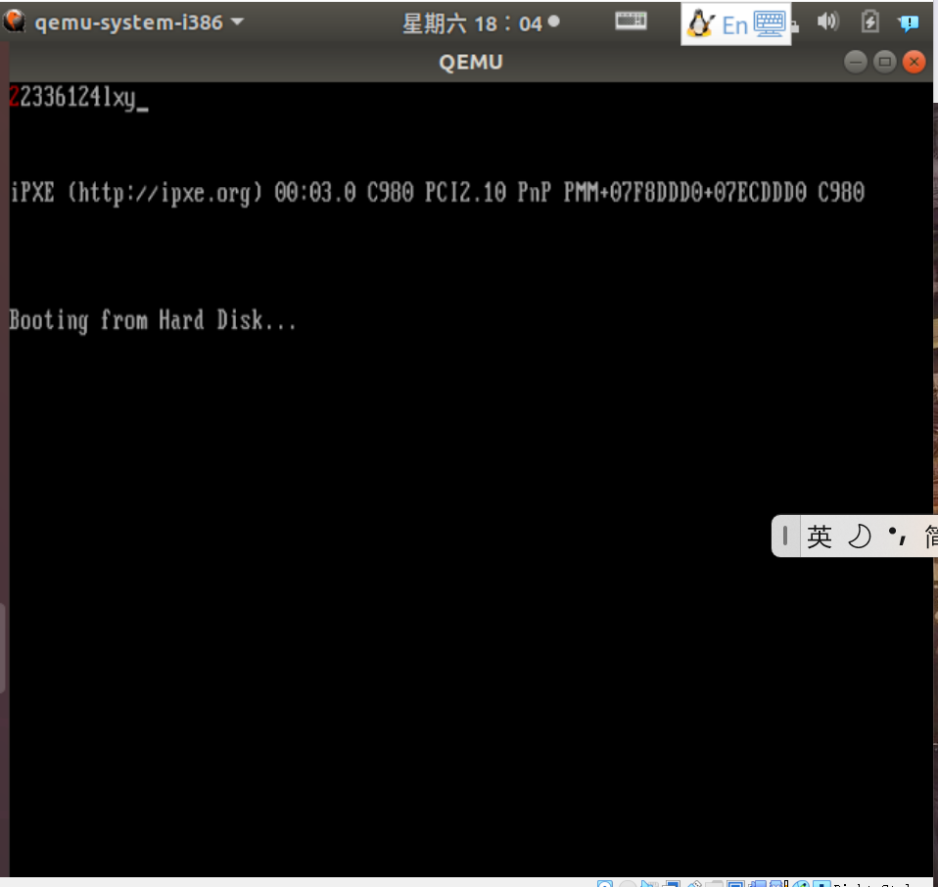
复现example 4：

****

****

****

跑马灯实现：

****

****

****

1. **总结**

实验三debug过程中，gdb定位不到相应文件一直显示空白，输入指令往往因此操作无效，解决方式：按照报错提示手动输入文件名帮助gdb定位文件，成功后再进行相应调试。