



本科生实验报告

实验课程 中山大学 2021 学年春季操作系统课程

实验名称 中断

专业名称 计算机科学与技术（超算）

学生姓名 黄玟瑜

学生学号 19335074

任课教师 陈鹏飞

实验地点

实验成绩

二〇二一年四月十一日

目录

| | | |
|----------|----------------------------------|-----------|
| 1 | 混合编程的基本思路 | 1 |
| 1.1 | 复现 Example 1: 混合编程 | 1 |
| 1.2 | 关键代码解释 | 3 |
| 2 | 使用 C/C++ 来编写内核 | 7 |
| 2.1 | 复现 Example 2: 内核的加载 | 7 |
| 2.1.1 | build | 8 |
| 2.1.2 | include | 10 |
| 2.1.3 | run | 11 |
| 2.1.4 | src | 12 |
| 2.2 | 输出学号 | 14 |
| 3 | 中断的处理 | 16 |
| 3.1 | 复现 Example 3: 初始化 IDT | 16 |
| 3.2 | 修改中断处理函数 | 22 |
| 4 | 时钟中断 | 23 |
| 4.1 | 复现 Example 4: 8259A 编程 | 23 |
| 4.2 | 时钟中断处理程序: 跑马灯显示学号 | 31 |

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。

1.1 复现 Example 1: 混合编程

我们需要做的工作如下：

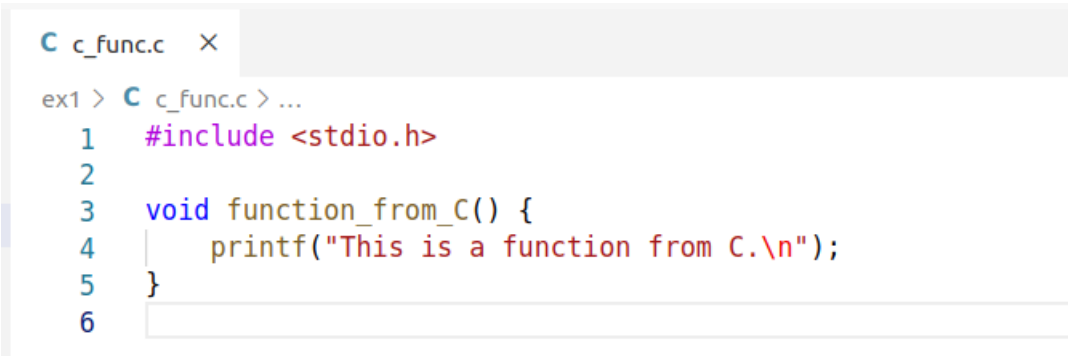
- 在文件 `c_func.c` 中定义 C 函数 `function_from_C`;
- 在文件 `cpp_func.cpp` 中定义 C++ 函数 `function_from_CPP`;
- 在文件 `asm_func.asm` 中定义汇编函数 `function_from_asm`, 在 `function_from_asm` 中调用 `function_from_C` 和 `function_from_CPP`;
- 在文件 `main.cpp` 中调用汇编函数 `function_from_asm`。

打开 VSCode，在主目录下新建文件夹 `lab4`，在 `lab4` 下新建文件夹 `ex1`。

在目录 `ex1` 下依次创建以下文件：

- `c_func.c`
- `cpp_func.cpp`
- `asm_func.asm`
- `main.cpp`
- `Makefile`

我们首先在文件 `c_func.c` 中定义 C 函数 `function_from_C`



```
C c_func.c X
ex1 > C c_func.c > ...
1  #include <stdio.h>
2
3  void function_from_C() {
4      printf("This is a function from C.\n");
5  }
6
```

然后在文件 `cpp_func.cpp` 中定义 C++ 函数 `function_from_CPP`

```
C c_func.c  cpp_func.cpp X
ex1 > G+ cpp_func.cpp > ...
1  #include <iostream>
2
3
4  extern "C" void function_from_CPP() {
5      std::cout << "This is a function from C++." << std::endl;
6  }
```

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`

```
C c_func.c  ASM asm_utils.asm X
ex1 > ASM asm_utils.asm
1  [[bits 32]]
2  global function_from_asm
3  extern function_from_C
4  extern function_from_CPP
5
6  function_from_asm:
7      call function_from_C
8      call function_from_CPP
9      ret
```

最后在文件 `main.cpp` 中调用汇编函数 `function_from_asm`

```
C c_func.c  G+ main.cpp X
ex1 > G+ main.cpp > ...
1  #include <iostream>
2
3  extern "C" void function_from_asm();
4
5  int main() {
6      std::cout << "Call function from assembly." << std::endl;
7      function_from_asm();
8      std::cout << "Done." << std::endl;
9  }
```

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, 编译命令如下, 将它们写到 `Makefile` 中

```

ex1 > M Makefile
1  main.out: main.o c_func.o cpp_func.o asm_func.o
2      g++ -o main.out main.o c_func.o cpp_func.o asm_func.o -m32
3
4  c_func.o: c_func.c
5      gcc -o c_func.o -m32 -c c_func.c
6
7  cpp_func.o: cpp_func.cpp
8      g++ -o cpp_func.o -m32 -c cpp_func.cpp
9
10 main.o: main.cpp
11     g++ -o main.o -m32 -c main.cpp
12
13 asm_func.o: asm_func.asm
14     nasm -o asm_func.o -f elf32 asm_func.asm
15 clean:
16     rm *.o

```

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。

使用 `make main.out` 生成 `main.out` 文件并执行

```

wenny@owo:~/lab4/ex1$ make main.out
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
wenny@owo:~/lab4/ex1$ ./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done.
wenny@owo:~/lab4/ex1$

```

1.2 关键代码解释

```

1  #include <iostream>
2
3  extern "C" void function_from_CPP() {
4      std::cout << "This is a function from C++." << std::endl;
5  }

```

函数名前加上了 `extern "C"`, 加上 `extern "C"` 后, 会指示编译器这部分代码按 C 语言的进行编译, 而不是 C++ 的, 从而实现 C++ 代码调用其他 C 语言代码。

`extern "C"` 的主要作用就是为了能够正确实现 C++ 代码调用其他 C 语言代码。加上 `extern "C"` 后, 会指示编译器这部分代码按 C 语言 (而不是 C++) 的方式进行编译。由于 C++ 支持函数重载, 因此编译器编译函数

的过程中会将函数的参数类型也加到编译后的代码中，而不仅仅是函数名；而 C 语言并不支持函数重载，因此编译 C 语言代码的函数时不会带上函数的参数类型，一般只包括函数名。

这个功能十分有用处，因为在 C++ 出现以前，很多代码都是 C 语言写的，而且很底层的库也是 C 语言写的，为了更好的支持原来的 C 代码和已经写好的 C 语言库，需要在 C++ 中尽可能的支持 C，而 `extern "C"` 就是其中的一个策略。

```
1 [bits 32]
2 global function_from_asm
3 extern function_from_C
4 extern function_from_CPP
5
6 function_from_asm:
7     call function_from_C
8     call function_from_CPP
9     ret
```

`global` 关键字用来让一个函数（或变量）对链接器可见，可以供其他链接对象模块使用。`extern` 说明一个函数（或变量）为外部函数（或变量），调用的时候可以遍访所有文件找到该函数（或变量）并且使用它。

`global` 在汇编和 C 混合编程中，汇编程序中要使用 `.global` 伪操作声明汇编程序为全局的函数，意即可被外部函数调用，同时 C 程序中要使用 `extern` 声明要调用的汇编语言程序。

`extern` 我们知道，程序的编译单位是源程序文件，一个源文件可以包含一个或若干个函数。在函数内定义的变量是局部变量，而在函数之外定义的变量则称为外部变量，外部变量也就是我们所讲的全局变量。它的存储方式为静态存储，其生存周期为整个程序的生存周期。全局变量可以为本文件中的其他函数所共用，它的有效范围为从定义变量的位置开始到本源文件结束。然而，如果全局变量不在文件的开头定义，有效的作用范围将只限于其定义处到文件结束。如果在定义点之前的函数想引用该全局变量，则应该在引用之前用关键字 `extern` 对该变量作“外部变量声明”，表示该变量是一个已经定义的外部变量。有了此声明，就可以从“声明”处起，合法地使用该外部变量。

```
1 main.out: main.o c_func.o cpp_func.o asm_func.o
2     g++ -o main.out main.o c_func.o cpp_func.o asm_func.o -m32
3
4 c_func.o: c_func.c
5     gcc -o c_func.o -m32 -c c_func.c
```

```

6
7  cpp_func.o:  cpp_func.cpp
8      g++ -o cpp_func.o -m32 -c cpp_func.cpp
9
10 main.o:  main.cpp
11      g++ -o main.o -m32 -c main.cpp
12
13 asm_func.o:  asm_func.asm
14      nasm -o asm_func.o -f elf32 asm_func.asm
15 clean:
16      rm *.o

```

Makefile 规则主要是两个部分组成，分别是依赖的关系和执行的命令，其结构如下所示：

```

1  targets : prerequisites
2      command

```

或者是

```

1  targets : prerequisites; command
2      command

```

- targets: 规则的目标，可以是 Object File（一般称它为中间文件），也可以是可执行文件，还可以是一个标签；
- prerequisites: 是我们的依赖文件，要生成 targets 需要的文件或者是目标。可以是多个，也可以是没有；
- command: make 需要执行的命令（任意的 shell 命令）。可以有多条命令，每一条命令占一行。

上述代码实现的功能依次生成编译可重定向文件 main.o、cpp_func.o、asm_func.o，再将它们链接生成可执行文件 main.out。

```

1  main.out:  main.o c_func.o cpp_func.o asm_func.o
2      g++ -o main.out main.o c_func.o cpp_func.o asm_func.o -m32

```

其中 main.out 是的目标文件，也是我们的最终生成的可执行文件。依赖文件就是 main.o、cpp_func.o、asm_func.o 源文件，重建目标文件需要执行的操作是 g++ -o main.out main.o c_func.o cpp_func.o asm_func.o -m32。

```

1  c_func.o:  c_func.c

```

```
2 gcc -o c_func.o -m32 -c c_func.c
```

其中 `c_func.o` 是的目标文件，也是我们的最终生成的可执行文件。依赖文件就是 `c_func.c` 源文件，重建目标文件需要执行的操作是 `gcc -o c_func.o -m32 -c c_func.c`。

其他文件依次类推。

```
1 clean :  
2     rm *.o
```

`clean` 操作清除所有格式为 `.o` 的文件。

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。

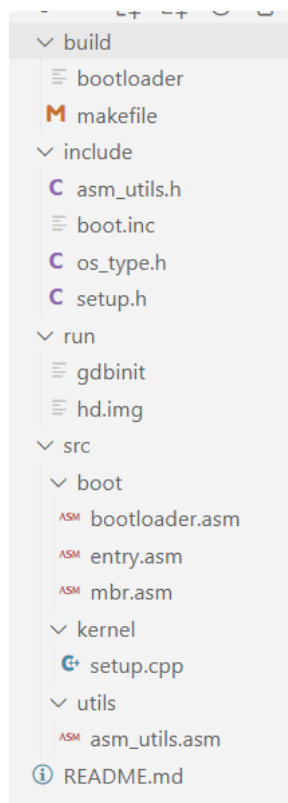
2.1 复现 Example 2：内核的加载

需要完成的任务如下：

在 **Example 2** 中，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 **“Hello World”**。

接上，在 lab4 文件夹中新建文件夹 ex2，用于存放 Example 2 的代码，作为 Example 2 的项目文件。

ex2 的文件目录如下：



Project 目录下包含 build、run、src、include 四个子文件夹，各个子文件夹的含义如下：

- **build** 存放 Makefile，make 之后生成的中间文件如.o，.bin 等会放置在这里，目的是防止这些文件混在代码文件中。
- **include** 存放.h 等函数定义和常量定义的头文件等。

- **run** 存放 gdb 配置文件，硬盘映像.img 文件等。
- **src** 存放.c, .cpp 等函数实现的文件。

2.1.1 build

build 文件夹下包含一个 Makefile, Makefile 的使用在前面的实验中已略有了解。文件内容如下：

```

1  ASM_COMPILER = nasm
2  C_COMPILER = gcc
3  CXX_COMPILER = g++
4  CXX_COMPILER_FLAGS = -g -Wall -march=i386 -m32 -nostdlib -fno-
    builtin -ffreestanding -fno-pic
5  LINKER = ld
6
7  SRCDIR = ../src
8  RUNDIR = ../run
9  BUILDDIR = build
10 INCLUDE_PATH = ../include
11
12 CXX_SOURCE += $(wildcard $(SRCDIR)/kernel/*.cpp)
13 CXX_OBJ += $(CXX_SOURCE:$(SRCDIR)/kernel/%.cpp=%o)
14
15 ASM_SOURCE += $(wildcard $(SRCDIR)/utils/*.asm)
16 ASM_OBJ += $(ASM_SOURCE:$(SRCDIR)/utils/%.asm=%o)
17
18 OBJ += $(CXX_OBJ)
19 OBJ += $(ASM_OBJ)
20
21 build : mbr.bin bootloader.bin kernel.bin kernel.o
22     dd if=mbr.bin of=$(RUNDIR)/hd.img bs=512 count=1 seek=0 conv
    =notrunc
23     dd if=bootloader.bin of=$(RUNDIR)/hd.img bs=512 count=5 seek
    =1 conv=notrunc
24     dd if=kernel.bin of=$(RUNDIR)/hd.img bs=512 count=145 seek=6
    conv=notrunc
25 # nasm的include path有一个尾随/
26
27 mbr.bin : $(SRCDIR)/boot/mbr.asm
28     $(ASM_COMPILER) -o mbr.bin -f bin -I$(INCLUDE_PATH)/ $(
    SRCDIR)/boot/mbr.asm
29
30 bootloader.bin : $(SRCDIR)/boot/bootloader.asm
31     $(ASM_COMPILER) -o bootloader.bin -f bin -I$(INCLUDE_PATH)/

```

```

32          $(SRCDIR)/boot/bootloader.asm
33 entry.obj : $(SRCDIR)/boot/entry.asm
34          $(ASM_COMPILER) -o entry.obj -f elf32 $(SRCDIR)/boot/entry.
asm
35
36 kernel.bin : entry.obj $(OBJ)
37          $(LINKER) -o kernel.bin -melf_i386 -N entry.obj $(OBJ) -e
enter_kernel -Ttext 0x00020000 --oformat binary
38
39 kernel.o : entry.obj $(OBJ)
40          $(LINKER) -o kernel.o -melf_i386 -N entry.obj $(OBJ) -e
enter_kernel -Ttext 0x00020000
41
42 $(CXX_OBJ):
43          $(CXX_COMPILER) $(CXX_COMPILER_FLAGS) -I$(INCLUDE_PATH) -c $(
CXX_SOURCE)
44
45 asm_utils.o : $(SRCDIR)/utils/asm_utils.asm
46          $(ASM_COMPILER) -o asm_utils.o -f elf32 $(SRCDIR)/utils/
asm_utils.asm
47 clean:
48          rm -f *.o* *.bin
49
50 run:
51          qemu-system-i386 -hda $(RUNDIR)/hd.img -serial null -
parallel stdio -no-reboot
52
53 debug:
54          qemu-system-i386 -S -s -parallel stdio -hda $(RUNDIR)/hd.img
-serial null&
55          @sleep 1
56          gnome-terminal -e "gdb -q -tui -x $(RUNDIR)/gdbinit"

```

解释:

定义汇编编译器为 ASM_COMPILER, C 语言编译器为 C_COMPILER, C++ 编译器为 CXX_COMPILER, C++ 的编译参数 CXX_COMPILER_FLAGS, 链接器 LD。

各个子目录名, SRCDIR、RUNDIR、BUILDDIR、INCLUDE_PATH。

指定汇编和 C++ 源文件和 obj 文件的目录 ASM_SOURCE、CXX_SOURCE、ASM_OBJ、CXX_OBJ。

build 步骤依次生成 mbr.bin、bootloader.bin、kernel.bin、kernel.o，再将生成的文件加载到硬盘 hd.img 中。后面的代码包含了生成 mbr.bin、bootloader.bin、kernel.bin、kernel.o 的指令。

clean 步骤清除所有.o 格式和.bin 格式的文件。

run 步骤将 run 目录下的硬盘文件 hd.img 作为起始扇区启动 qemu-system-i386。

debug 步骤在启动 qemu 时加入调试参数，在新的终端启动 gdb，并以 gdbinit 的参数初始化 gdb。

2.1.2 include

boot.inc 包含了一些常量的定义：

```
1 ; 常量定义区
2 ; _____Loader_____
3 ; 加载器扇区数
4 LOADER_SECTOR_COUNT equ 5
5 ; 加载器起始扇区
6 LOADER_START_SECTOR equ 1
7 ; 加载器被加载地址
8 LOADER_START_ADDRESS equ 0x7e00
9 ; _____GDT_____
10 ; GDT起始位置
11 GDT_START_ADDRESS equ 0x8800
12 ; _____Selector_____
13 ;平坦模式数据段选择子
14 DATA_SELECTOR equ 0x8
15 ;平坦模式栈段选择子
16 STACK_SELECTOR equ 0x10
17 ;平坦模式视频段选择子
18 VIDEO_SELECTOR equ 0x18
19 VIDEO_NUM equ 0x18
20 ;平坦模式代码段选择子
21 CODE_SELECTOR equ 0x20
22
23 ; _____kernel_____
24 KERNEL_START_SECTOR equ 6
25 KERNEL_SECTOR_COUNT equ 200
26 KERNEL_START_ADDRESS equ 0x20000
```

头文件 asm_utils.h 包含了所用到的汇编工具单，在此只有一个输出 Hello world 的函数 asm_hello_world:

```

1 #ifndef ASM_UTILS_H
2 #define ASM_UTILS_H
3
4 extern "C" void asm_hello_world();
5
6 #endif

```

在头文件 os_type.h 中进行 C++ 到汇编的类型定义：

```

1 #ifndef OS_TYPE_H
2 #define OS_TYPE_H
3
4 // 类型定义
5 typedef unsigned char byte;
6 typedef unsigned char uint8;
7
8 typedef unsigned short uint16;
9 typedef unsigned short word;
10
11 typedef unsigned int uint32;
12 typedef unsigned int uint;
13 typedef unsigned int dword;
14
15 #endif

```

在 setup.h 中定义内核进入点：

```

1 #ifndef ENTRY_H
2 #define ENTRY_H
3
4 extern "C" void setup_kernel();
5
6 #endif

```

2.1.3 run

包含 gdbinit 文件用于初始化 gdb：

```

1 target remote:1234
2 file ../build/kernel.o
3 set disassembly-flavor intel
4 set architecture i386

```

gdb 初始化操作包括设置反汇编语言风格为 Intel，设置工程为 i386 以利于对齐 32 位代码。

2.1.4 src

src 目录还包含了 3 个子目录：

- **boot** 用于存放引导扇区代码 mbr.asm、加载内核部分代码 bootloader.asm、从入口跳转到内核的接口的实现 entry.asm
- **kernel** 内核本体
- **asm_utils** asm_utils.asm 包含了各个汇编函数的实现，由于一些语句只在汇编中有而 C++ 中没有，因此需要在 C++ 源文件中调用这些函数再用汇编实现

mbr.asm、bootloader.asm 在之前的实验中已有记录，不同的部分是内核入口函数（位于 entry.asm）：

```
1 global enter_kernel
2 extern setup_kernel
3 enter_kernel:
4     jmp setup_kernel
```

至此项目建立完成，下面开始运行。

执行指令 make：

```
wenny@owo: ~/lab4/ex2/build
wenny@owo:~/lab4/ex2/build$ make
nasm -o mbr.bin -f bin -I../include/ ../src/boot/mbr.asm
nasm -o bootloader.bin -f bin -I../include/ ../src/boot/bootloader.asm
nasm -o entry.obj -f elf32 ../src/boot/entry.asm
g++ -g -Wall -march=i386 -m32 -nostdlib -fno-builtin -ffreestanding -fno-pic -I.
../include -c ../src/kernel/setup.cpp
nasm -o asm_utils.o -f elf32 ../src/utils/asm_utils.asm
ld -o kernel.bin -melf_i386 -N entry.obj setup.o asm_utils.o -e enter_kernel -Tt
ext 0x00020000 --oformat binary
ld -o kernel.o -melf_i386 -N entry.obj setup.o asm_utils.o -e enter_kernel -Ttex
t 0x00020000
dd if=mbr.bin of=../run/hd.img bs=512 count=1 seek=0 conv=notrunc
记录了1+0 的读入
记录了1+0 的写出
512字节已复制, 0.00483888 s, 106 kB/s
dd if=bootloader.bin of=../run/hd.img bs=512 count=5 seek=1 conv=notrunc
记录了0+1 的读入
记录了0+1 的写出
281字节已复制, 0.000426082 s, 659 kB/s
dd if=kernel.bin of=../run/hd.img bs=512 count=145 seek=6 conv=notrunc
记录了145+0 的读入
记录了145+0 的写出
74240字节 (74 kB, 72 KiB) 已复制, 0.00232881 s, 31.9 MB/s
wenny@owo:~/lab4/ex2/build$
```

执行 make run:

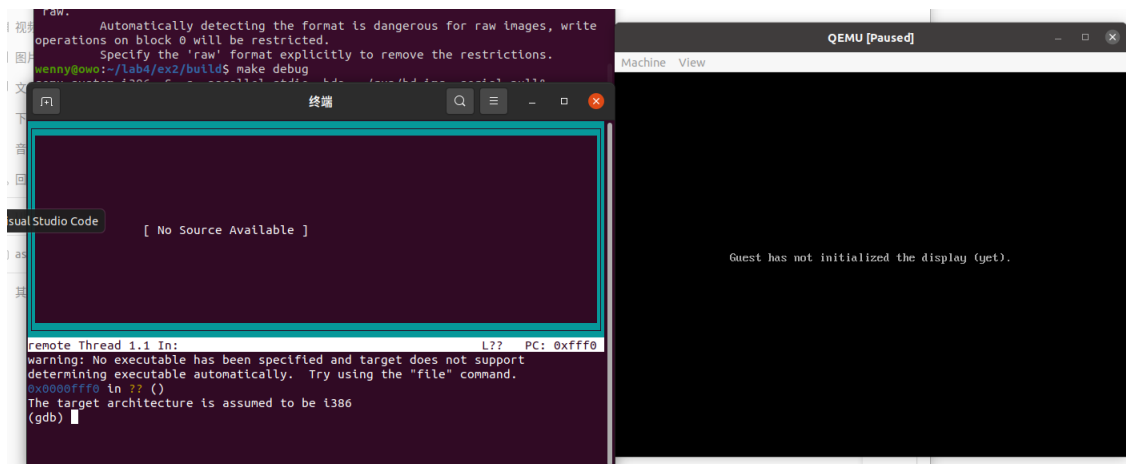
```
wenny@owo:~/lab4/ex2/build$ make run
qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot
QEMU
Machine View
Hello Worldrsion 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...
```

屏幕上输出了 Hello world。

执行 make debug:



成功连接上 gdb 并完成初始化操作。

复现 Example 2 成功。

2.2 输出学号

Hello world 的输出函数为 `asm_hello_world`, 若想输出学号只需修改 `asm_hello_world` 的输出参数即可。

修改的代码如下：

```

1  asm_hello_world:
2      push  eax
3      xor  eax,  eax
4
5      mov  ah,  0x03 ;青色
6      mov  al,  '1'
7      mov  [gs:2 * 0],  ax
8
9      mov  al,  '9'
10     mov  [gs:2 * 1],  ax
11
12     mov  al,  '3'
13     mov  [gs:2 * 2],  ax
14
15     mov  al,  '3'
16     mov  [gs:2 * 3],  ax
17
18     mov  al,  '5'
19     mov  [gs:2 * 4],  ax
20
21     mov  al,  '0'
22     mov  [gs:2 * 5],  ax

```



```

23
24     mov al, '7'
25     mov [gs:2 * 6], ax
26
27     mov al, '4'
28     mov [gs:2 * 7], ax
29
30     pop eax
31     ret

```

执行 make && make run, 输出如下:

```

wenny@owo:~/lab4/ex2/build$ make run
qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot

```

QEMU

Machine View

19335074(version 1.13.0-lubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...

0.04.2.0 LTS amd64

成功输出学号。

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。

3.1 复现 Example 3: 初始化 IDT

初始化 IDT 的 256 个中断, 这 256 个中断的中断处理程序均是向栈中压入 0xdead-beef 后做死循环, 并在内核中尝试调用中断处理程序。

在 lab4 文件夹中新建文件夹 ex3, 用于存放 Example 3 的代码, 作为 Example 4 的项目文件。

该中断处理程序如下, 它将会输出 “Unhandled interrupt happened, halt...”:

```
1 ASM_UNHANDLED_INTERRUPT_INFO db 'Unhandled interrupt happened,
    halt...'
2                                     db 0
3
4 ; void asm_unhandled_interrupt()
5 asm_unhandled_interrupt:
6     cli
7     mov esi, ASM_UNHANDLED_INTERRUPT_INFO
8     xor ebx, ebx
9     mov ah, 0x03
10 .output_information:
11     cmp byte[esi], 0
12     je .end
13     mov al, byte[esi]
14     mov word[gs:bx], ax
15     inc esi
16     add ebx, 2
17     jmp .output_information
18 .end:
19     jmp $
```

实验步骤如下:

- 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。

为了便于中断管理,构建中断处理器类 InterruptManager,定义在头文件 include/interrupt.h 中,如下所示:

```
1  #ifndef INTERRUPT_H
2  #define INTERRUPT_H
3
4  #include "os_type.h"
5
6  class InterruptManager
7  {
8  private:
9      // IDT起始地址
10     uint32 *IDT;
11
12 public:
13     InterruptManager();
14     // 初始化
15     void initialize();
16     // 设置中断描述符
17     // index 第index个描述符, index=0, 1, ..., 255
18     // address 中断处理程序的起始地址
19     // DPL 中断描述符的特权级
20     void setInterruptDescriptor(uint32 index, uint32 address, byte
        DPL);
21 };
22
23 #endif
```

在函数 InterruptManager::initialize 中初始化 IDT,这也是最关键的步骤:先设置 IDTR,然后再初始化 256 个中断描述符。

```
1  void InterruptManager::initialize()
2  {
3      // 初始化IDT
4      IDT = (uint32 *)IDT_START_ADDRESS;
5      asm_lidt(IDT_START_ADDRESS, 256 * 8 - 1);
6
7      for (uint i = 0; i < 256; ++i)
8      {
9          setInterruptDescriptor(i, (uint32)
              asm_interrupt_empty_handler, 0);
10     }
11 }
```

通过 `IDT = (uint32 *)IDT_START_ADDRESS;` 给 IDT 赋值, 此时 IDT (变量名) 存的值即中断向量表的起始地址, 下面将依次说明调用到的函数, 先是汇编函数 `asm_lidt`, 再是 C 函数 `InterruptManager::setInterruptDescriptor`, 先来说明函数 `asm_lidt`。

将要在 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 `src/utils/asm_utils.asm` 中, 如下所示:

```
1 ; void asm_lidt(uint32 start, uint16 limit)
2 asm_lidt:
3     push ebp
4     mov ebp, esp
5     push eax
6
7     mov eax, [ebp + 4 * 3]
8     mov [ASM_IDTR], ax
9     mov eax, [ebp + 4 * 2]
10    mov [ASM_IDTR + 2], eax
11    lidt [ASM_IDTR]
12
13    pop eax
14    pop ebp
15    ret
16
17 ASM_IDTR dw 0
18          dd 0
```

该函数依次将 16 位的 IDT 表界限、32 位的基地址存储到 `ASM_IDTR` 中, 再通过指令 `lidt` 将其装载到 IDTR 中。

现在说明描述符表设置函数 `InterruptManager::setInterruptDescriptor`。

描述符的设置函数如下, 它的实现放置在 `interrupt.cpp` 中:

```
1 // 设置中断描述符
2 // index    第index个描述符, index=0, 1, ..., 255
3 // address  中断处理程序的起始地址
4 // DPL      中断描述符的特权级
5 void InterruptManager::setInterruptDescriptor(uint32 index, uint32
        address, byte DPL)
6 {
7     IDT[index * 2] = (CODE_SELECTOR << 16) | (address & 0xffff);
8     IDT[index * 2 + 1] = (address & 0xffff0000) | (0x1 << 15) | (DPL
```

```

9     << 13) | (0xe << 8);
    }

```

- IDT 是中断向量表的指针，因此可将 IDT 看作一个数组，类型为 uint32
- 每个描述符占用两个 unit32, 因此要赋值的数组元素为 IDT[index * 2] 和 IDT[index * 2 + 1]
- 第二步设置给定特权值。

在类声明头文件 include/os_modules.h 中声明中断管理器类，以便于其他 CPP 文件使用：

```

1  #ifndef OS_MODULES_H
2  #define OS_MODULES_H
3
4  #include "interrupt.h"
5
6  extern InterruptManager interruptManager;
7
8  #endif

```

在常量定义头文件 include/os_constant.h 中添加 IDT 的起始地址和代码段选择子：

```

1  #ifndef OS_CONSTANT_H
2  #define OS_CONSTANT_H
3
4  #define IDT_START_ADDRESS 0x8880
5  #define CODE_SELECTOR 0x20
6
7  #endif

```

最后在内核启动函数 setup.cpp 中实例化中断管理器，初始化 256 个中断，尝试触发之：

```

1  // 中断管理器
2  InterruptManager interruptManager;
3
4  extern "C" void setup_kernel()
5  {
6      // 中断处理部件
7      interruptManager.initialize();
8
9      // 尝试触发除 0 错误

```

```

10     int a = 1 / 0;
11
12     // 死循环
13     asm_halt();
14 }

```

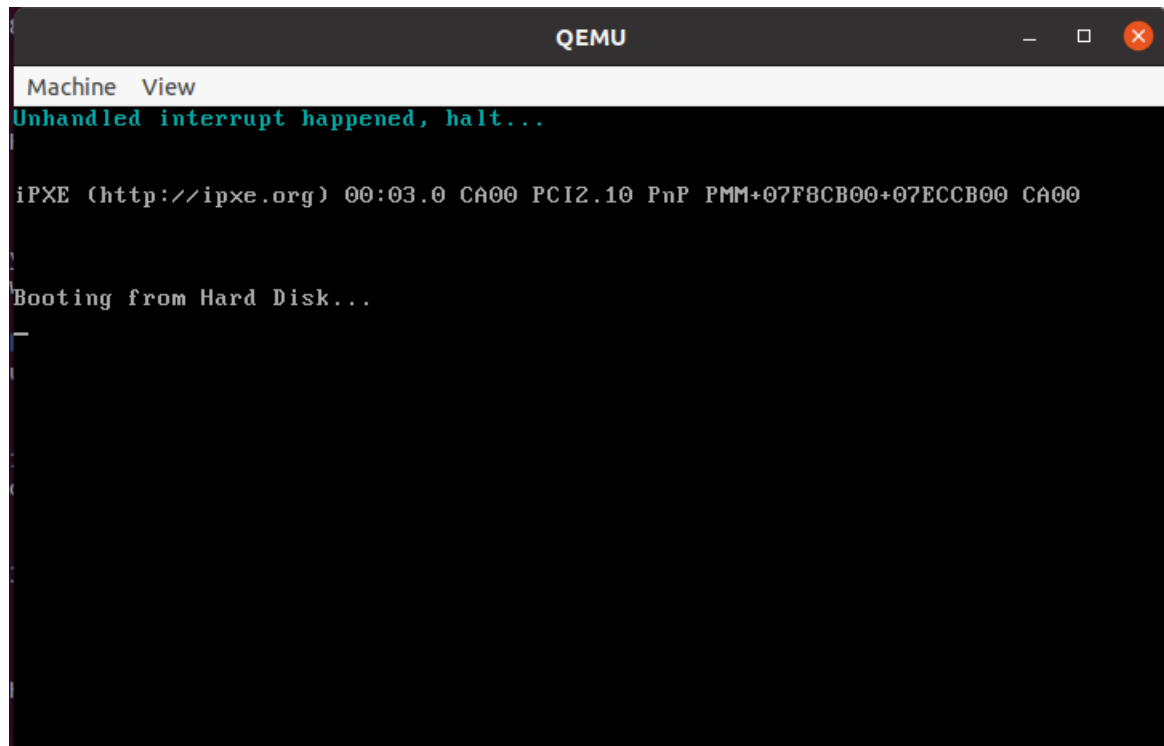
执行指令 `make && make run`:

```

wenny@owo:~/lab4/ex3/build$ make && make run
dd if=mbr.bin of=../run/hd.img bs=512 count=1 seek=0 conv=notrunc
记录了1+0 的读入
记录了1+0 的写出
512字节已复制, 0.00161773 s, 316 kB/s
dd if=bootloader.bin of=../run/hd.img bs=512 count=5 seek=1 conv=notrunc
记录了0+1 的读入
记录了0+1 的写出
281字节已复制, 0.000436614 s, 644 kB/s
dd if=kernel.bin of=../run/hd.img bs=512 count=145 seek=6 conv=notrunc
记录了145+0 的读入
记录了145+0 的写出
74240字节 (74 kB, 72 KiB) 已复制, 0.0126952 s, 5.8 MB/s
qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '../run/hd.img' and probing guessed
raw.
       Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
       Specify the 'raw' format explicitly to remove the restrictions.

```

结果如下:



```

Machine View
Unhandled interrupt happened, halt...

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...

```

符合预期效果。

在触发中断前调用 `asm_halt`，以利于后续在此处设断点：`setup.cpp`

```
1 #include "asm_utils.h"
2 #include "interrupt.h"
3
4 // 中断管理器
5 InterruptManager interruptManager;
6
7 extern "C" void setup_kernel()
8 {
9     // 中断处理部件
10    interruptManager.initialize();
11    // 死循环
12    asm_halt();
13    // 尝试触发除0错误
14    int a = 1 / 0;
15 }
```

执行 `make && make debug`，在 `asm_halt` 处设置断点，这时中断管理器刚好完成初始化工作，此时我们查看 IDT 的值：

```
0x8880: 0x00028e000020018c    0x00028e000020018c
0x8890: 0x00028e000020018c    0x00028e000020018c
0x88a0: 0x00028e000020018c    0x00028e000020018c
0x88b0: 0x00028e000020018c    0x00028e000020018c
0x88c0: 0x00028e000020018c    0x00028e000020018c
0x88d0: 0x00028e000020018c    0x00028e000020018c
0x88e0: 0x00028e000020018c    0x00028e000020018c
0x88f0: 0x00028e000020018c    0x00028e000020018c
0x8900: 0x00028e000020018c    0x00028e000020018c
0x8910: 0x00028e000020018c    0x00028e000020018c
0x8920: 0x00028e000020018c    0x00028e000020018c
0x8930: 0x00028e000020018c    0x00028e000020018c
0x8940: 0x00028e000020018c    0x00028e000020018c
0x8950: 0x00028e000020018c    0x00028e000020018c
0x8960: 0x00028e000020018c    0x00028e000020018c
0x8970: 0x00028e000020018c    0x00028e000020018c
0x8980: 0x00028e000020018c    0x00028e000020018c
0x8990: 0x00028e000020018c    0x00028e000020018c
0x89a0: 0x00028e000020018c    0x00028e000020018c
0x89b0: 0x00028e000020018c    0x00028e000020018c
0x89c0: 0x00028e000020018c    0x00028e000020018c
0x89d0: 0x00028e000020018c    0x00028e000020018c
0x89e0: 0x00028e000020018c    0x00028e000020018c
0x89f0: 0x00028e000020018c    0x00028e000020018c
0x8a00: 0x00028e000020018c    0x00028e000020018c
--Type <RET> for more, q to quit, c to continue without paging--
```

说明已经成功放入默认的中断描述符，可以验证上面的输出结果符合预期。

3.2 修改中断处理函数

修改中断处理函数为字符弹射程序，只需要在 tag asm_unhandled_interrupt 处进行修改，贴入代码过于冗长，故不在实验报告中展示，代码见附件。

实验结果如下：



输出结果符合预期。

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。

4.1 复现 Example 4: 8259A 编程

在本实验中，将对 8259A 芯片进行编程，添加处理实时钟中断的函数。

在 lab4 文件夹中新建文件夹 ex4，用于存放 Example 4 的代码，作为 Example 4 的项目文件。

首先需要在中断管理器中加入处理时钟中断的成员变量和函数：

```
1  class InterruptManager
2  {
3  private:
4      uint32 *IDT;           // IDT起始地址
5
6      uint32 IRQ0_8259A_MASTER; // 主片中断起始向量号
7      uint32 IRQ0_8259A_SLAVE;  // 从片中断起始向量号
8
9  public:
10     InterruptManager();
11     void initialize();
12     // 设置中断描述符
13     // index 第index个描述符, index=0, 1, ..., 255
14     // address 中断处理程序的起始地址
15     // DPL 中断描述符的特权级
16     void setInterruptDescriptor(uint32 index, uint32 address, byte
17         DPL);
18
19     // 开启时钟中断
20     void enableTimeInterrupt();
21     // 禁止时钟中断
22     void disableTimeInterrupt();
23     // 设置时钟中断处理函数
24     void setTimeInterrupt(void *handler);
25 private:
26     // 初始化8259A芯片
```

```

27     void initialize8259A();
28 };

```

在实现过程中涉及 in、out 指令的调用，而 C 语言中没有 in、out 指令，故需要用汇编函数进行实现封装，从而在 C 程序中调用：

```

1  ; void asm_in_port(uint16 port, uint8 *value)
2  asm_in_port:
3      push ebp
4      mov ebp, esp
5
6      push edx
7      push eax
8      push ebx
9
10     xor eax, eax
11     mov edx, [ebp + 4 * 2] ; port
12     mov ebx, [ebp + 4 * 3] ; *value
13
14     in al, dx
15     mov [ebx], al
16
17     pop ebx
18     pop eax
19     pop edx
20     pop ebp
21     ret

```

```

1  ; void asm_out_port(uint16 port, uint8 value)
2  asm_out_port:
3      push ebp
4      mov ebp, esp
5
6      push edx
7      push eax
8
9      mov edx, [ebp + 4 * 2] ; port
10     mov eax, [ebp + 4 * 3] ; value
11     out dx, al
12
13     pop eax
14     pop edx
15     pop ebp
16     ret

```

在使用 8259A 芯片之前，首先要对其初始化，初始化的代码放置在成员函数 initialize8259A 中，初始化 8259A 芯片的过程是通过设置一系列的 ICW 字来完成的，尚未建立处理 8259A 中断的任何函数，因此在初始化的最后需要屏蔽主片和从片的所有中断：

```
1 void InterruptManager::initialize8259A ()
2 {
3     // ICW 1
4     asm_out_port(0x20, 0x11);
5     asm_out_port(0xa0, 0x11);
6     // ICW 2
7     IRQ0_8259A_MASTER = 0x20;
8     IRQ0_8259A_SLAVE = 0x28;
9     asm_out_port(0x21, IRQ0_8259A_MASTER);
10    asm_out_port(0xa1, IRQ0_8259A_SLAVE);
11    // ICW 3
12    asm_out_port(0x21, 4);
13    asm_out_port(0xa1, 2);
14    // ICW 4
15    asm_out_port(0x21, 1);
16    asm_out_port(0xa1, 1);
17
18    // OCW 1 屏蔽主片所有中断，但主片的 IRQ2 需要开启
19    asm_out_port(0x21, 0xfb);
20    // OCW 1 屏蔽从片所有中断
21    asm_out_port(0xa1, 0xff);
22 }
```

接下来定义中断处理函数 c_time_interrupt_handler。由于需要显示中断发生的次数，因此在 src/kernel/interrupt.cpp 中定义一个全局变量来充当计数变量，如下所示。

```
1 int times = 0;
```

中断处理函数 c_time_interrupt_handler 如下：

```
1 // 中断处理函数
2 extern "C" void c_time_interrupt_handler ()
3 {
4     // 清空屏幕
5     for (int i = 0; i < 80; ++i)
6     {
7         stdio.print(0, i, ' ', 0x07);
8     }
9 }
```

```

10 // 输出中断发生的次数
11 ++times;
12 char str[] = "interrupt happend: ";
13 char number[10];
14 int temp = times;
15
16 // 将数字转换为字符串表示
17 for(int i = 0; i < 10; ++i ) {
18     if(temp) {
19         number[i] = temp % 10 + '0';
20     } else {
21         number[i] = '0';
22     }
23     temp /= 10;
24 }
25
26 // 移动光标到(0,0)输出字符
27 stdio.moveCursor(0);
28 for(int i = 0; str[i]; ++i ) {
29     stdio.print(str[i]);
30 }
31
32 // 输出中断发生的次数
33 for( int i = 9; i > 0; --i ) {
34     stdio.print(number[i]);
35 }
36 }

```

调用了 `stdio::print()` 函数，这是一个封装起来的控制显示的类，它的声明位于 `include/stdio.h` 中，如下所示：

```

1 #ifndef STDIO_H
2 #define STDIO_H
3
4 #include "os_type.h"
5
6 class STDIO
7 {
8 private:
9     uint8 *screen;
10
11 public:
12     STDIO();
13     // 初始化函数

```

```

14     void initialize();
15     // 打印字符c, 颜色color到位置(x,y)
16     void print(uint x, uint y, uint8 c, uint8 color);
17     // 打印字符c, 颜色color到光标位置
18     void print(uint8 c, uint8 color);
19     // 打印字符c, 颜色默认到光标位置
20     void print(uint8 c);
21     // 移动光标到一维位置
22     void moveCursor(uint position);
23     // 移动光标到二维位置
24     void moveCursor(uint x, uint y);
25     // 获取光标位置
26     uint getCursor();
27
28 public:
29     // 滚屏
30     void rollUp();
31 };
32
33 #endif

```

它的实现没有太复杂的结构上问题，故不在此一一分析。

中断的返回需要使用 `iret` 指令，而 C 语言的任何函数编译出来的返回语句都是 `ret`。因此，我们只能在汇编代码中完成保护现场、恢复现场和中断返回。

中断处理函数的实现思路如下：

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。C 语言编写的函数返回后，指令的执行流程会返回到 `call` 指令的下一条汇编代码。此时，我们使用汇编代码恢复保存的寄存器的内容，最后使用 `iret` 返回。

即保护现场-> 中断处理-> 恢复现场。

完整的中断处理程序如下：

```

1 asm__time__interrupt__handler:
2     pushad
3
4     nop ; 否则断点打不上去
5     ; 发送EOI消息，否则下一次中断不发生
6     mov al, 0x20

```

```

7      out 0x20, al
8      out 0xa0, al
9
10     call c_time_interrupt_handler
11
12     popad
13     iret

```

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：

```

1 void InterruptManager::setTimeInterrupt(void *handler)
2 {
3     setInterruptDescriptor(IRQ0_8259A_MASTER, (uint32)handler, 0);
4 }

```

封装开启和关闭时钟中断的函数。关于 8259A 上的中断开启情况，可以通过读取 OCW1 来得知；如果要修改 8259A 上的中断开启情况，需要先读取再写入对应的 OCW1，如下所示：

```

1 void InterruptManager::enableTimeInterrupt()
2 {
3     uint8 value;
4     // 读入主片OCW
5     asm_in_port(0x21, &value);
6     // 开启主片时钟中断，置0开启
7     value = value & 0xfe;
8     asm_out_port(0x21, value);
9 }
10
11 void InterruptManager::disableTimeInterrupt()
12 {
13     uint8 value;
14     asm_in_port(0x21, &value);
15     // 关闭时钟中断，置1关闭
16     value = value | 0x01;
17     asm_out_port(0x21, value);
18 }

```

最后，我们在 `setup_kernel` 中定义 `STDIO` 的实例 `stdio`，初始化内核的组件，然后开启时钟中断和开中断：

```

1 #include "asm_utils.h"
2 #include "interrupt.h"
3 #include "stdio.h"

```

```

4
5 // 屏幕IO处理器
6 STDIO stdio;
7 // 中断管理器
8 InterruptManager interruptManager;
9
10 extern "C" void setup_kernel()
11 {
12     // 中断处理部件
13     interruptManager.initialize();
14     // 屏幕IO处理部件
15     stdio.initialize();
16     interruptManager.enableTimeInterrupt();
17     interruptManager.setTimeInterrupt((void *)
18         asm_time_interrupt_handler);
19     asm_enable_interrupt();
20     asm_halt();
21 }

```

在类声明头文件 include/os_modules.h 中声明 STDIO 类，以便于其他 CPP 文件使用

```

1 #ifndef OS_MODULES_H
2 #define OS_MODULES_H
3
4 #include "interrupt.h"
5
6 extern InterruptManager interruptManager;
7 extern STDIO stdio;
8
9 #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：

```

1 ; void asm_enable_interrupt()
2 asm_enable_interrupt:
3     sti
4     ret

```

执行 make && make run:

```
wenny@owo:~/lab4/ex4/build$ make
g++ -g -Wall -march=i386 -m32 -nostdlib -fno-builtin -ffreestanding -fno-pic -I.
./include -c ../src/kernel/setup.cpp ../src/kernel/interrupt.cpp ../src/kernel/s
tdio.cpp
nasm -o asm_utils.o -f elf32 ../src/utils/asm_utils.asm
ld -o kernel.bin -melf_i386 -N entry.obj setup.o interrupt.o stdio.o asm_utils.o
-e enter_kernel -Ttext 0x00020000 --oformat binary
ld -o kernel.o -melf_i386 -N entry.obj setup.o interrupt.o stdio.o asm_utils.o -
e enter_kernel -Ttext 0x00020000
dd if=mbr.bin of=../run/hd.img bs=512 count=1 seek=0 conv=notrunc
记录了1+0 的读入
记录了1+0 的写出
512字节已复制, 0.000597173 s, 857 kB/s
dd if=bootloader.bin of=../run/hd.img bs=512 count=5 seek=1 conv=notrunc
记录了0+1 的读入
记录了0+1 的写出
281字节已复制, 0.000395124 s, 711 kB/s
dd if=kernel.bin of=../run/hd.img bs=512 count=145 seek=6 conv=notrunc
记录了145+0 的读入
记录了145+0 的写出
74240字节 (74 kB, 72 KiB) 已复制, 0.0024561 s, 30.2 MB/s
wenny@owo:~/lab4/ex4/build$ make run
qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '../run/hd.img' and probing guessed
raw.
    Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
    Specify the 'raw' format explicitly to remove the restrictions.
```

显示如下:



```
QEMU
Machine View
interrupt happend: 000000004

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...
```

可以看到, 第一行显示了目前中断发生的次数, 符合预期效果。

4.2 时钟中断处理程序：跑马灯显示学号

该部分主要是对时钟中断处理函数进行修改，由于 8259A 芯片产生的时钟中断信号频率较高，若持续输出肉眼不好观察，故考虑对齐进行分频，引入中间变量 freq：

```
1 int freq = 0;
```

每次触发时钟中断 freq 累计 1，times 的值为 freq/7，最后再以 times 的改变频率来实现跑马灯。

定义学号常量：

```
1 char stu_id[] = "19335074";
```

完整的中断处理函数如下：

```
1 // 中断处理函数
2 extern "C" void c_time_interrupt_handler()
3
4 {
5
6     // 清空屏幕
7     for (int i = 0; i < 80; ++i)
8     {
9         for(int j = 0; j < 25; ++j)
10            stdio.print(j, i, ' ', 0x07);
11     }
12
13     // 分频
14     ++freq;
15     times = freq / 7;
16
17     // 输出中断发生的次数
18     char str[] = "interrupt happend: ";
19     char number[10];
20     int temp = times;
21     // 将数字转换为字符串表示
22     for(int i = 0; i < 10; ++i ) {
23         if(temp) {
24             number[i] = temp % 10 + '0';
25         } else {
26             number[i] = '0';
27         }
28         temp /= 10;
```

```

29     }
30
31     // 移动光标到 (0,0) 输出字符
32     stdio.setCursor(0);
33     for(int i = 0; str[i]; ++i) {
34         stdio.print(str[i]);
35     }
36
37     // 输出中断发生的次数
38     for( int i = 9; i > 0; --i ) {
39         stdio.print(number[i]);
40     }
41
42     // 跑马灯输出学号
43     if(times < 8)
44         stdio.print(1, times, stu_id[times], 0x05);
45 }

```

执行指令 `make && make run`, 输出结果如下:

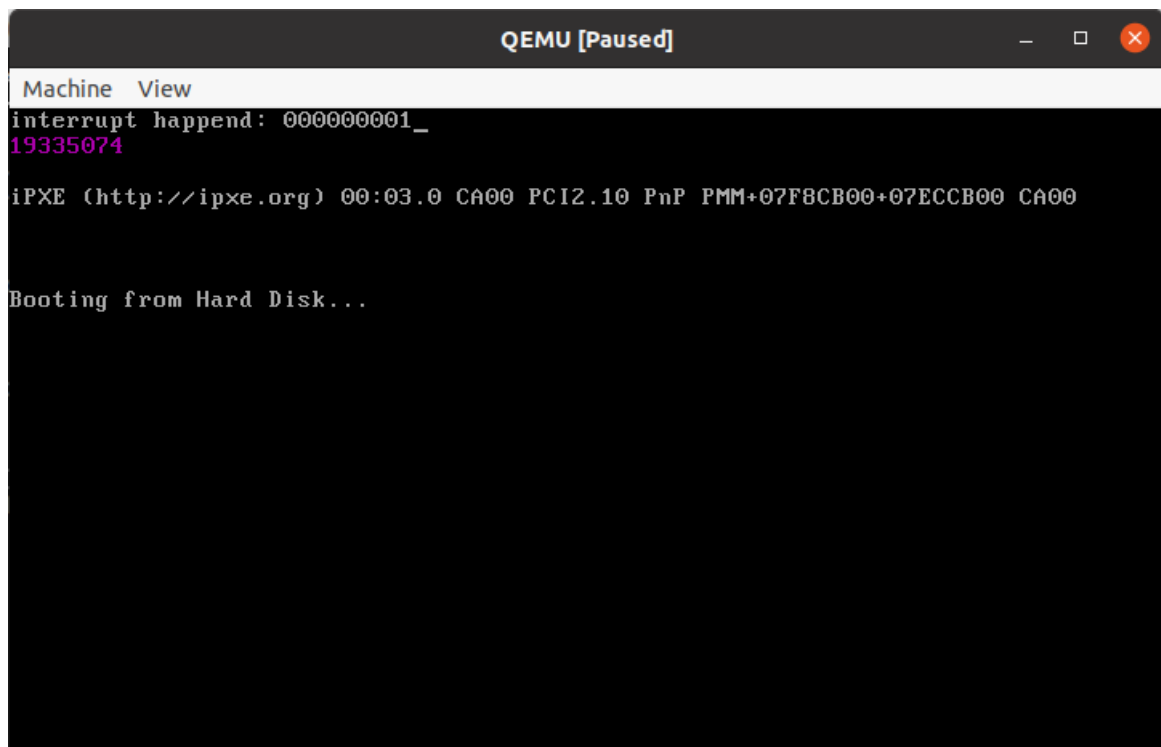
```

Machine View
interrupt happend: 000000000
1933507
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...

```

ual Studio Code



可以看到，程序以人眼可以适应的频率跑马灯输出学号，碍于实验报告只能展示静态的图片，无法展示动画效果，若要查看显示动画效果建议运行以上代码。

实验总结

在本次实验中，我了解了从代码到可执行文件的过程，包括预处理、编译、汇编、链接 4 个阶段，g 在 C/C++ 和汇编混和编程中，gcc 和 g++ 作为编译器将 C 和 CPP 文件预处理、编译、汇编成可重定位文件，nasm 将汇编代码处理成可重定向文件，最后通过链接器 LD 链接起来成为可执行文件。

此外，我还对 Makefile 的使用有了进一步的了解，比如变量的定义、不同目录下文件如何相互链接调用等。

在阅读和编写汇编代码和 C/C++ 代码，对一些关键字（global、extern 等）的含义和作用有了初步的认识，也学会了汇编函数和 C/C++ 函数相互调用的方法。

在汇编函数和 C/C++ 函数相互调用时值得注意的是参数的传递，对于 C/C++ 来说，函数的调用时的输入参数和返回值是有明确指向的，而在汇编函数中，参数传递是通过将参数入栈，在汇编程序中根据栈指针的值再去读取来获得，其中还包含了一些固定的调用规则。

在学习中断时了解了中断程序的调用过程，以及中断程序的运行过程，主要是保护现场-> 运行程序-> 恢复现场。在这个过程对很多细碎的东西都有了进一步的认识。在复现 Example 3 的过程中对中断处理有了更进一步的了解，我们实验的主要工作是定义中断管理器类（有且仅有一个实例），使用了 C++ 封装特性，便于不同功能的类的管理。使用中断管理器来初始化中断向量表，添加中断描述符，这让我们得以干净简洁的添加和管理中断。

在我看来时钟中断实验是对中断处理的一个延伸，其中我了解了 8259A 这一可编程中断控制器，它包括主片和从片，使用它的过程中更多是一些硬性的规则和参数的传递问题。在这个实验中我们封装了 STDIO 类，用于控制管理基本输入输出。最终我们实现了接收并处理 8259A 产生的时钟中断，处理过程中利用 STDIO 类实例 stdio 为我们输出所需信息。

前面是我在本次实验的收获的一些知识点的总结，此外还有很多没有提到，这次实验使我收益良多，这次实验涉及很多方面（特别是一些固定规则），因此我还需要在以后的实验继续吸收。

对我而言，此次实验最难的部分是实验报告的编写，实验指导讲的很详尽，提供的代码又是完整的，需要动手操作的部分很少，因此做完实验时不知道怎么编写实验报告，最后更多的展示的是一点自己的理解。

总而言之，此次实验使我受益匪浅。