



中山大學  
SUN YAT-SEN UNIVERSITY

# 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: 第三章 中断

专业名称: 信息与计算科学

学生姓名: 郑鸿鑫

学生学号: 22336313

实验地点: 实验中心 D503

实验时间: 2024/4/8

## Section 1 实验概述

在本章中，我们首先介绍一份 C 代码是如何通过预编译、编译、汇编和链接生成最终的可执行文件。接着，为了更加有条理地管理我们操作系统的代码，我们提出了一种 C/C++ 项目管理方案。在做了上面的准备工作后，我们开始介绍 C 和汇编混合编程方法，即如何在 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 编写的函数。介绍完混合编程后，我们来到了本章的主体内容——中断。我们介绍了保护模式下的中断处理机制和可编程中断部件 8259A 芯片。最后，我们通过编写实时钟中断处理函数来将本章的所有内容串联起来。

## Section 2 预备知识与实验环境

- 预备知识：x86 汇编语言程序设计、IA-32 处理器体系结构，LBA 方式读写硬盘和 CHS 方式读写硬盘的相关知识。
- 实验环境：

- 虚拟机版本/处理器型号：

11th Gen Intel® Core™ i5-11320H @ 3.20GHz × 2

- 代码编辑环境：VS Code

- 代码编译工具：g++

■ 重要三方库信息：Linux 内核版本号：linux-5.10.210

Ubuntu 版本号：Ubuntu 18.04.6LTS，Busybox 版本号：

Busybox\_1\_33\_0

## Section 3 实验任务

- 实验任务 1：学习混合编程的基本思路
- 实验任务 2：用 C/C++ 编写内核
- 实验任务 3：中断的处理
- 实验任务 4：时钟中断的处理

## Section 4 实验步骤与实验结果

### ----- 实验任务 1 -----

- 任务要求：学习混合编程的基本思路
- 思路分析：

根据指导书中“C/C++和汇编混合编程”及“一个混合编程的例子”这两节的知识，按要求完成实验

- 实验步骤：

实验要求 1：

复现混合编程例子，修改为“Done by 22336313 ZHX”

1. 我们首先在文件 `c_func.c` 中定义 C 函数 `function_from_C`。
2. 然后在文件 `cpp_func.cpp` 中定义 C++ 函数 `function_from_CPP`。
3. 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`。
4. 然后在文件 `main.cpp` 中调用汇编函数 `function_from_asm`。
5. 最后在终端配合 `makefile` 文件，用 `make` 和 `./main.out` 运行程序。

#### 实验要求 2:

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"`？

答:

1. `global` 关键字用于使汇编函数在链接时可见，这样 C 代码中的 `extern` 声明才能找到并调用该函数。
2. 在汇编代码中调用 C 函数，需要使用 `extern` 关键字来声明 C 函数。

例如在文件 `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`，这两处地方采取的关键字有所不同，定义汇编函数 `function_from_asm` 时是使用 `global`，而调用两个在其他文件中已经定义好的函数

`function_from_C` 和 `function_from_CPP` 则用 `extern`。

3. 在 C++ 函数前需要加上 `extern "C"` 的原因是: C++ 默认使用名称修饰 (Name Mangling) 来支持重载等特性。这意味着 C++ 编译器会改变函数名，以包含类型信息和作用域等。这导致 C 代码无法直接链接 C++ 编译器生成的函数。为了解决这个问题，C++ 引入了 `extern "C"` 关键字，它可以告诉编译器使用 C 的链接方式，即不进行名称修饰。这样，C 代码就可以链接 C++ 编译器生成的函数了。

- 实验结果展示:

通过 `./main.out` 命令得到运行结果:

(其中注意需要修改 `makefile` 文件中部分代码，详见 Section6 中详细说明)

```
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

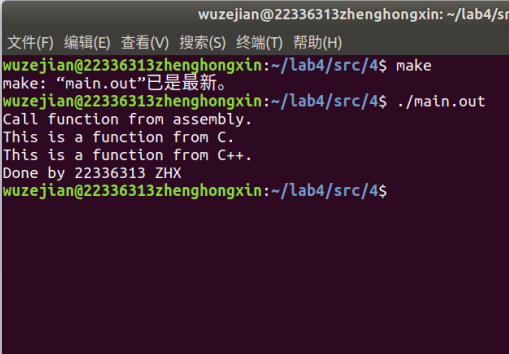
c_func.o: c_func.c
gcc -o c_func.o -m32 -c c_func.c

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_utils.asm
nasm -o asm_func.o -f elf32 asm_utils.asm

clean:
rm *.o
```



## ----- 实验任务 2 -----

- 任务要求：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，将输出 `Hello World` 改为输出学号 + 姓名首字母。
- 思路分析：按照指导书的步骤进行操作，并按要求在对应位置修改代码。
- 实验步骤：
  1. 在 `build` 文件夹下，使用 `make` 命令进行编译，`make run` 命令进行运行以复现“内核的加载”部分。
  2. 将 `asm_utils.asm` 文件修改代码如下所示：

```
[bits 32]
global asm_hello_world
asm_hello_world:
    push eax
    xor eax, eax
    mov ah, 0x03 ;青色
    mov al, '2'
    mov [gs:2 * 0], ax
    mov al, '2'
    mov [gs:2 * 1], ax
    mov al, '1'
    mov [gs:2 * 2], ax
```

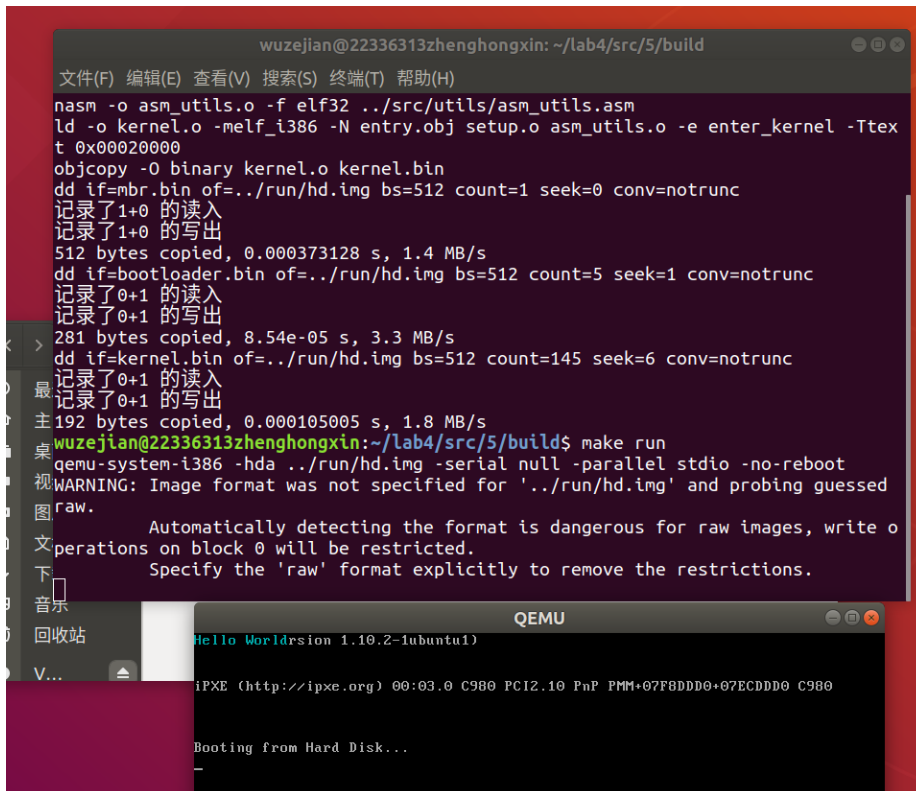
```
mov al, '3'
mov [gs:2 * 3], ax
mov al, '3'
mov [gs:2 * 4], ax
mov al, '6'
mov [gs:2 * 5], ax
mov al, '3'
mov [gs:2 * 6], ax
mov al, '1'
mov [gs:2 * 7], ax
mov al, '3'
mov [gs:2 * 8], ax
mov al, 'Z'
mov [gs:2 * 9], ax
mov al, 'H'
mov [gs:2 * 10], ax
mov al, 'X'
mov [gs:2 * 11], ax
pop eax
ret
```

3.在 build 文件夹中还存在上一次编译生成的文件，所以需要先执行命令 `make clean` 以清除。

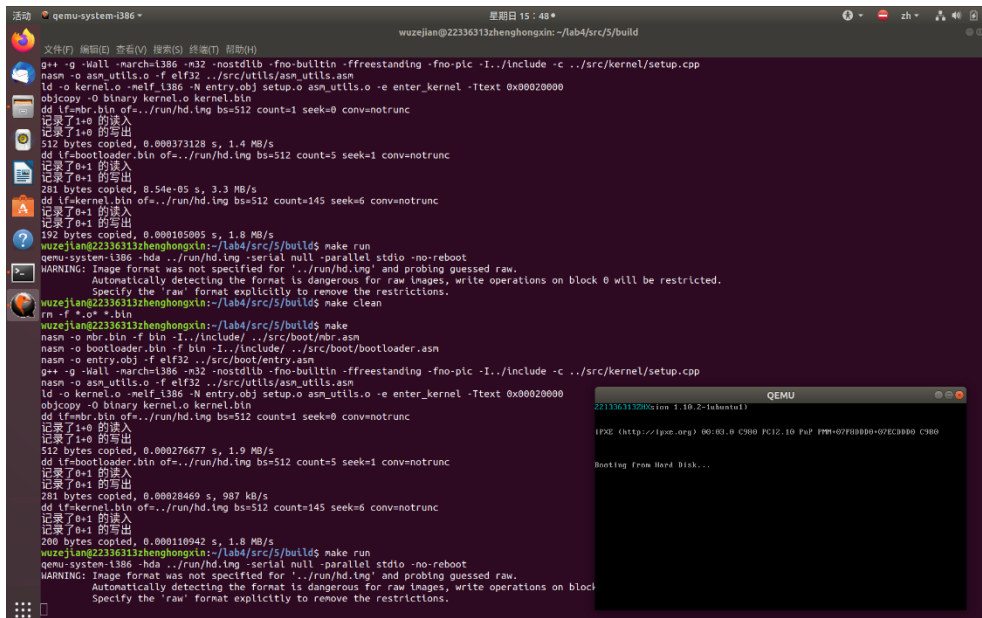
4. 再次执行 `make` 和 `make run` 即可得到结果

- 实验结果展示:

复现“内核的加载”结果如下:



修改代码后再次运行：



## 实验任务 3



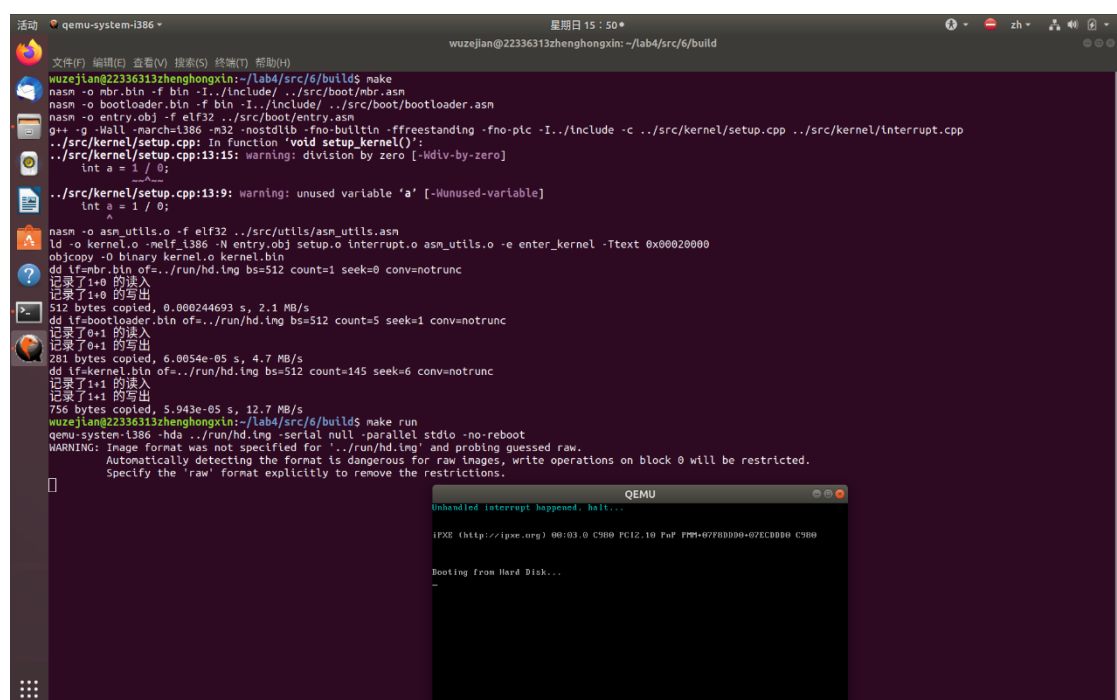
- 任务要求：中断的处理：复现网址中“初始化 IDT”部分，更改默认的中断处理函数为自己编写的函数，然后触发之。调用处理函数时输出包含个人学号或姓名信息。
- 思路分析：阅读指导书学习中断的处理，并在对应文件修改代码。
- 实验步骤：
  1. 在 build 文件夹下，使用 make 命令进行编译，make run 命令进行运行以复现“初始化 IDT”部分。
  2. 修改代码如下所示：

```
asm_unhandled_interrupt:
    cli
    mov esi, ASM_UNHANDLED_INTERRUPT_INFO
    xor ebx, ebx
    mov ah, 0x03
.output_information:
    mov ah, 0x03 ;青色
    mov al, '2'
    mov [gs:2 * 0], ax
    mov al, '2'
    mov [gs:2 * 1], ax
    mov al, '3'
    mov [gs:2 * 2], ax
    mov al, '3'
    mov [gs:2 * 3], ax
    mov al, '6'
    mov [gs:2 * 4], ax
    mov al, '3'
    mov [gs:2 * 5], ax
    mov al, '1'
    mov [gs:2 * 6], ax
    mov al, '3'
    mov [gs:2 * 7], ax
.end:
    jmp $
```

3. 注意在 build 文件夹下使用 `make clean` 清除前面产生的文件，防止后续产生干扰，然后再通过 `make` 和 `make run` 指令编译运行得到结果。

- 实验结果展示：通过执行前述代码，可得下图结果。

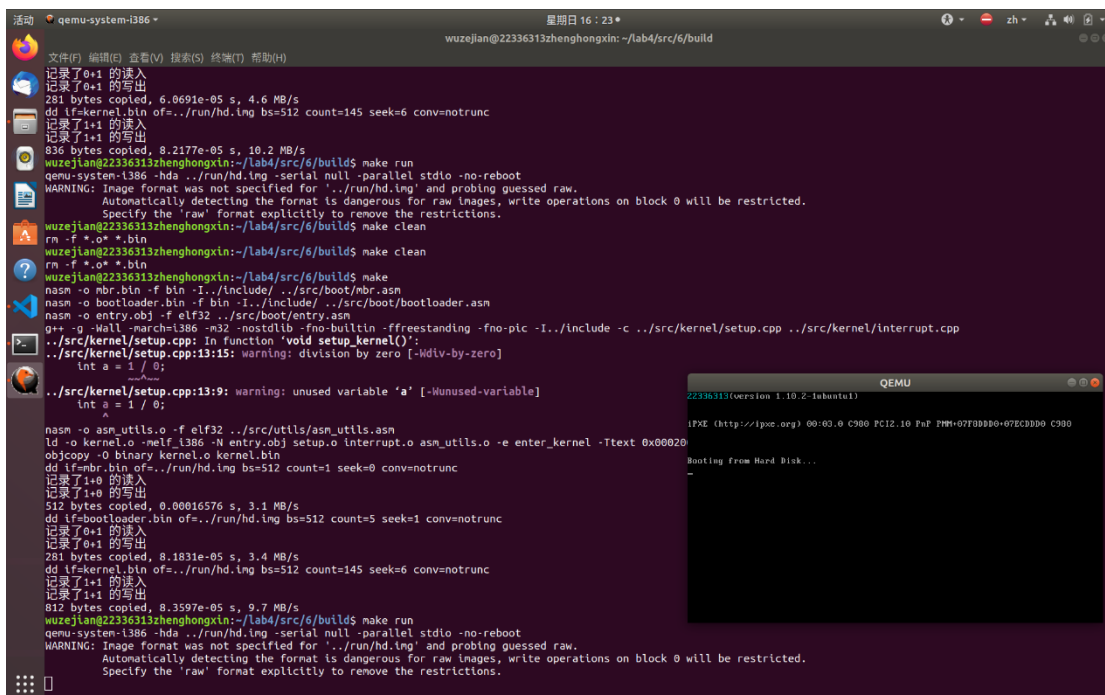
复现“初始化 IDT”结果如下：



```
wuzejian@22336313zhenghongxin:~/lab4/src/6/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 ../src/kernel/interrupt.cpp
../src/kernel/setup.cpp: in function 'void setup_kernel()':
../src/kernel/setup.cpp:13:15: warning: division by zero [-Wdiv-by-zero]
    int a = 1 / 0;
                  ^
../src/kernel/setup.cpp:13:9: warning: unused variable 'a' [-Wunused-variable]
    int a = 1 / 0;
        ^
nasm -o asm_utils.o -f elf32 ../src/utils/asm_utils.asm
ld -o kernel.o -melf_i386 -N entry.obj setup.o interrupt.o asm_utils.o -e enter_kernel -Ttext 0x00020000
objcopy -O binary kernel.o kernel.bin
dd if=mbr.bin of=../run/hd.img bs=512 count=1 seek=0 conv=notrunc
记录1+0 的写入
512 bytes copied, 0.000244693 s, 2.1 MB/s
dd if=bootloader.bin of=../run/hd.img bs=512 count=5 seek=1 conv=notrunc
记录0+1 的写入
记录0+1 的写入
281 bytes copied, 6.0054e-05 s, 4.7 MB/s
dd if=kernel.bin of=../run/hd.img bs=512 count=145 seek=6 conv=notrunc
记录1+1 的写入
记录1+1 的写入
756 bytes copied, 5.943e-05 s, 12.7 MB/s
wuzejian@22336313zhenghongxin:~/lab4/src/6/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 operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

QEMU
Unhandled interrupt happened, halt...
iPXE (http://ipxe.org) 00:03:0 C800 FC12:10 PaP FPM+07F0DD00+07ECDD00 C800
Booting from Hard Disk...
```

修改代码后输出自己的学号：



```
活动 qemu-system-i386 星期日 16:23 * wuzejian@22336313zhenghongxin: ~/lab4/src/6/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
记录了0+1 的读入
记录了0+1 的写出
231 bytes copied, 6.0691e-05 s, 4.6 MB/s
dd if=kernel.bin of=../run/hd.img bs=512 count=145 seek=6 conv=notrunc
记录了1+1 的读入
记录了1+1 的写出
836 bytes copied, 8.2177e-05 s, 10.2 MB/s
wuzejian@22336313zhenghongxin:~/lab4/src/6/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 operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
wuzejian@22336313zhenghongxin:~/lab4/src/6/build$ make clean
rm -f *.o* *.bin
wuzejian@22336313zhenghongxin:~/lab4/src/6/build$ make clean
rm -f *.o* *.bin
wuzejian@22336313zhenghongxin:~/lab4/src/6/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 ../src/kernel/interrupt.cpp
../src/kernel/setup.cpp: In function 'void setup_kernel()':
../src/kernel/setup.cpp:13:15: warning: division by zero [-Wdiv-by-zero]
    int a = 1 / 0;
                ^
../src/kernel/setup.cpp:13:9: warning: unused variable 'a' [-Wunused-variable]
    int a = 1 / 0;
        ^
nasm -o asm_utils.o -f elf32 ../src/utils/asm_utils.asm
ld -o kernel.o -melf_i386 -N entry.obj setup.o interrupt.o asm_utils.o -e enter_kernel -Ttext 0x00020
objcopy -O binary kernel.o kernel.bin
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 bytes copied, 0.00016576 s, 3.1 MB/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 bytes copied, 8.1831e-05 s, 3.4 MB/s
dd if=kernel.bin of=../run/hd.img bs=512 count=145 seek=6 conv=notrunc
记录了1+1 的读入
记录了1+1 的写出
812 bytes copied, 8.3597e-05 s, 9.7 MB/s
wuzejian@22336313zhenghongxin:~/lab4/src/6/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 operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

## 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，要求：仿照该章节中使用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(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)，保存结果截图并说说你的思路 and 做法。
- 思路分析：学习指导书中时钟中断的处理，复现结果及按要求修改代码后展示
- 实验步骤：

1. 在 build 文件夹下输入 make 进行编译，再输入 make run 运行即可复现“8259A 编程——实时钟中断的处理”。
2. 利用自己的类实现跑马灯，按以下步骤修改代码：
  - a. 在 stdio.h 下声明两个新的函数：

```
//new function
void clearLine(uint line);
void displayScrollingText(uint line, const char* text, int
pos);
```

其中 clearLine()用于清除某一行的输出，displayscrollingtext()则是打印跑马灯的关键函数。它们的具体实现会在 stdio.cpp 中补充给出。

- b. 在 stdio.cpp 中实现这两个函数的具体代码。

STDIO::clearLine 函数负责清除指定行的内容。它通过计算行的起始位置在显存中的地址，然后逐渐增加遍历整行，将每个字符位置设置为一个空格，并将字符属性设置为灰色背景。这样做可以确保之前在该行全部显示为空格，从而达到清除行内容的效果。

STDIO::displayScrollingText 函数用于在指定行显示滚动文本。它首先计算输入文本的长度，然后调用 clearLine 函数清空当前行。计算文本在屏幕上的开始位置后，它通过一个循环将文本字符一个接一个地写入显存，如果到达文本末尾则循环回到文本开始。文本字符使用青色字体显示。

```

void STDIO::clearLine(uint line) {
    if (line >= 25) return;
    uint position = line * 80 * 2;
    for (int i = 0; i < 80; ++i) {
        screen[position++] = ' '; // ASCII 空格
        screen[position++] = 0x07; // 属性字节: 灰色背景
    }
}

void STDIO::displayScrollingText(uint line, const char* text, int pos) {
    if (line >= 25 || !text) return;
    int len = 0;
    while (text[len] != '\0') ++len; // 计算文本长度
    clearLine(line); // 清除该行
    uint position = line * 80 * 2;

    for (int i = 0; i < 80; ++i) {
        screen[position++] = text[(pos + i) % len]; // 循环显示文本
        screen[position++] = 0x03; // 青色字体
    }
}

```

c. 实现两个辅助函数之后，继续实现中断处理函数以实现跑马灯：

```

size_t my_strlen(const char* str) {
    size_t len = 0;
    while (str[len]) len++;
    return len;
}

extern "C" void c_time_interrupt_handler()
{
    // 清空屏幕
    for (int i = 0; i < 80; ++i)
    {
        stdio.print(0, i, ' ', 0x07);
    }

    static int scrollPosition = 0;
    const char* scrollingText = " 22336313 ZHX ";
    // 设置跑马灯的行为第10行，边框在第9行和第11行
    stdio.clearLine(9); // 清除上边框
    stdio.clearLine(10); // 清除跑马灯行
    stdio.clearLine(11); // 清除下边框
    stdio.displayScrollingText(10, scrollingText, scrollPosition);
    scrollPosition = (scrollPosition + 1) %
my_strlen(scrollingText); // 更新位置
}

```

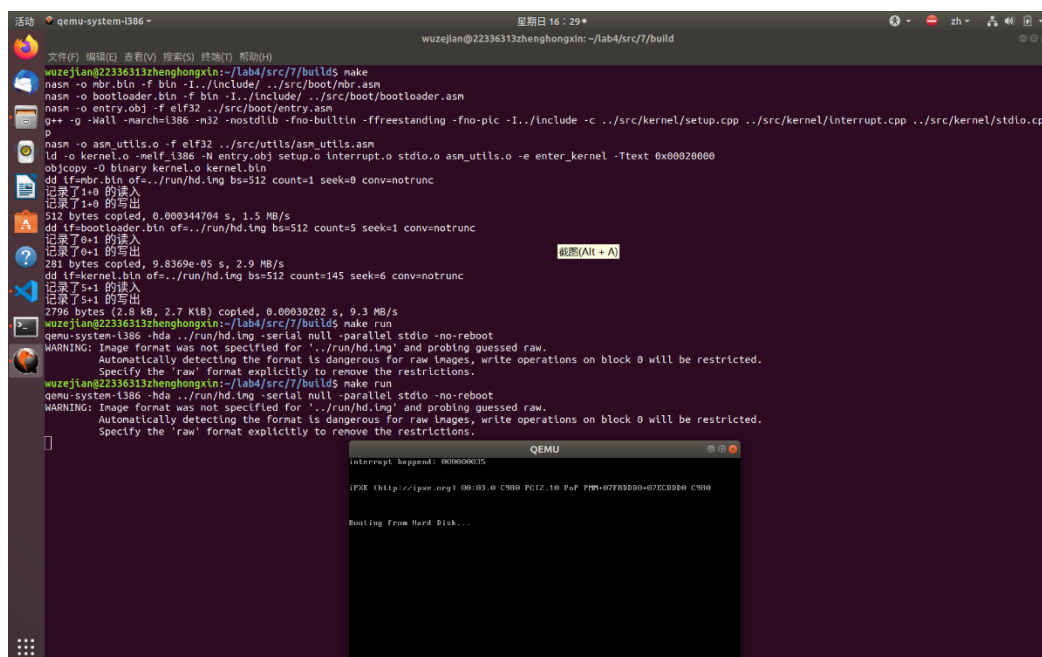
```
}
```

my\_strlen()函数用于求字符串的长度（此处是因为无法直接使用 C 语言库中的 strlen()函数，故自己实现一个简单的求字符串长度的函数）。c\_time\_interrupt\_handler()函数用于实现跑马灯，调用前述两个辅助函数以实现

3. 在 build 文件夹下进行编译运行，注意先使用 make clean 以删除前导的文件，以免产生干扰。

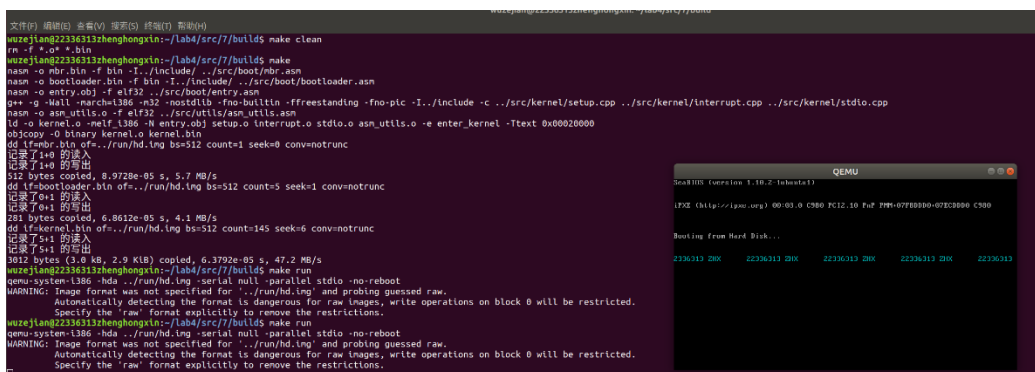
● 实验结果展示：通过执行前述代码，可得下图结果

复现“8259A 编程——实时钟中断的处理”结果如下：



```
qemu-system-i386 -wuzejian@22336313zhenghongxin:~/lab4/src/7/build
wuzejian@22336313zhenghongxin:~/lab4/src/7/build$ make
nasm -o mbr.bin -f bin -I. -I../include/ ../src/boot/mbr.asm
nasm -o bootloder.bin -f bin -I../include/ ../src/boot/bootloder.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 ../src/kernel/interrupt.cpp ../src/kernel/stdio.cpp
nasm -o asm_utils.o -f elf32 ../src/utlis/asm_utils.asm
ld -o kernel.o -melf_i386 -N entry.obj setup.o interrupt.o stdio.o asm_utils.o -e enter_kernel -Ttext 0x0020000
objcopy -O binary kernel.o kernel.bin
dd if=mbr.bin of=../run/hd.tng bs=512 count=1 seek=0 conv=notrunc
记录了1+0 的读入
记录了1+0 的写入
512 bytes copied, 0.000344704 s, 1.5 MB/s
dd if=bootloder.bin of=../run/hd.tng bs=512 count=5 seek=1 conv=notrunc
记录了0+1 的读入
记录了0+1 的写入
281 bytes copied, 9.8369e-05 s, 2.9 MB/s
dd if=kernel.bin of=../run/hd.tng bs=512 count=145 seek=6 conv=notrunc
记录了5+1 的读入
记录了5+1 的写入
2796 bytes (2.8 kB, 2.7 KiB) copied, 0.00030202 s, 9.3 MB/s
wuzejian@22336313zhenghongxin:~/lab4/src/7/build$ make run
qemu-system-i386 -hda ../run/hd.tng -serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '../run/hd.tng' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
wuzejian@22336313zhenghongxin:~/lab4/src/7/build$ make run
qemu-system-i386 -hda ../run/hd.tng -serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '../run/hd.tng' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
QEMU
interrupt happen: 000000035
IPXE (http://ipxe.org): 00:03:0 C800 FC12:10 PuP 7MM-677B0000-07EC0000 C800
Booting from Hard Disk...
```

混合编程编写跑马灯结果如下：



## Section 5 实验总结与心得体会

### 1. 编译过程：

- 了解了 C 代码从源文件到可执行文件的完整编译过程，包括预编译、编译、汇编和链接四个阶段。

- 预编译主要处理宏定义、头文件的包含等；编译将 C 代码转换成汇编代码；汇编将汇编代码转换成机器码；链接则将多个目标文件和库链接成最终的可执行文件。

### 3. C 与汇编混合编程：

- 掌握了在 C 代码中嵌入汇编代码的方法，如内联汇编和 extern "C" 关键字的使用。
- 了解了如何在汇编代码中调用 C 函数，以及如何处理 C 和汇编之间的接口，包括函数调用约定和数据类型转换。

### 4. 中断处理：

- o 深入理解了中断的概念，包括中断向量、中断服务例程（ISR）和中断优先级。

- o 学习了保护模式下的中断处理机制，特别是在 x86 架构下，如何设置中断描述符表（IDT）和处理中断。

#### 5. 8259A 可编程中断控制器：

- o 了解了 8259A 芯片的工作原理和编程方法，包括初始化、中断屏蔽和优先级设置。

- o 掌握了如何使用 8259A 来管理多个硬件中断源，并实现中断的嵌套和级联。

#### 6. 实时钟中断处理：

- o 通过编写实时钟中断处理函数，实践了中断编程的技巧，并将前面学习的知识点综合运用。

- o 学习了如何使用中断来实现操作系统的定时任务，如更新系统时间、调度等。

### **Section 6 对实验的改进建议和意见**



```
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_utils.asm
nasm -o asm_func.o -f elf32 asm_utils.asm

clean:
rm *.o
```



```
wuzejian@22336313zhenghongxin: ~/lab4/src
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
wuzejian@22336313zhenghongxin:~/lab4/src/4$ make
make: "main.out"已是最新。
wuzejian@22336313zhenghongxin:~/lab4/src/4$ ./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done by 22336313 ZHX
wuzejian@22336313zhenghongxin:~/lab4/src/4$
```

在实验任务 1 中提到：makefile 中部分代码有误，主要体现在于：在进行编译时将 asm\_utils.asm 错误的写为 asm\_func.asm，但文件夹中并未存在该名称的文件，故修改。最后成功复现“一个混合编程的例子”。

## Section 7 附录：参考资料清单

指导书网站：[SYSU-2023-Spring-Operating-System: 中山大学 2023 学年春季操作系统课程 - Gitee.com](#)

混合编程参考：[C 语言&汇编混合编程 - 知乎 \(zhihu.com\)](#)