



本科生实验报告

实验课程：____操作系统原理实验____

实验名称：____中断____

专业名称：____计算机科学与技术____

学生姓名：____熊彦钧____

学生学号：____23336266____

实验地点：____实验楼 B203____

实验成绩：____

报告时间：____2025 年 4 月 8 日____

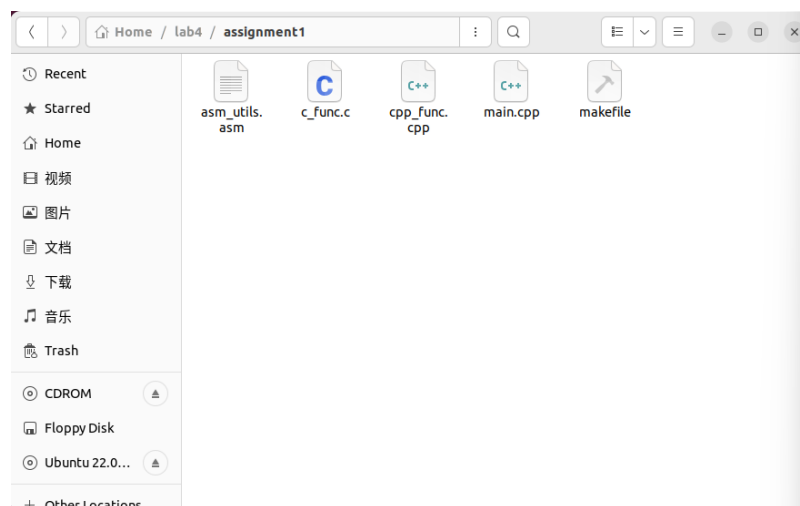
Section 1 实验概述

- 实验任务 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。
- 实验任务 2: 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。
- 实验任务 3: 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即可。
 - 实验任务 3 思考题: 使用尽可能多的方法触发段错误, 并在你的实验报告里总结一下, 引发段错误都有哪几种方式。
- 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, 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 （对应 assignment1） -----

- 任务要求: 复现 example1: 一个 c 语言和汇编语言混合编程的例子。
- 实验步骤: ①将代码复制到本地, `make` 之前文件夹的文件如下:



②运行 `makefile`, 在终端输入 `make` 进行编译。第一次编译时发现编译失败, 出现如下报错:

```
jhinx@jhinx-virtual-machine: ~/lab4/assignment1
jhinx@jhinx-virtual-machine:~/lab4/assignment1$ make
g++ -o main.o -m32 -c main.cpp
gcc -o c_func.o -m32 -c c_func.c
g++ -o cpp_func.o -m32 -c cpp_func.cpp
make: *** No rule to make target 'asm_func.asm', needed by 'asm_func.o'. Stop.
jhinx@jhinx-virtual-machine:~/lab4/assignment1$
```

经过检查，发现是 makefile 文件的内容有误，经过修改后，成功编译。

```
jhinx@jhinx-virtual-machine: ~/lab4/assignment1
jhinx@jhinx-virtual-machine:~/lab4/assignment1$ make clean
rm *.o
jhinx@jhinx-virtual-machine:~/lab4/assignment1$ make
g++ -o main.o -m32 -c main.cpp
gcc -o c_func.o -m32 -c c_func.c
g++ -o cpp_func.o -m32 -c cpp_func.cpp
nasm -o asm_utils.o -f elf32 asm_utils.asm
g++ -o main.out main.o c_func.o cpp_func.o asm_utils.o -m32
jhinx@jhinx-virtual-machine:~/lab4/assignment1$
```

③运行程序。输入指令 `./main.out` 即可运行打包后的程序，运行结果如下：

```
jhinx@jhinx-virtual-machine: ~/lab4/assignment1
jhinx@jhinx-virtual-machine:~/lab4/assignment1$ ./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done.
jhinx@jhinx-virtual-machine:~/lab4/assignment1$
```

经过对照，程序输出正确，实验复现任务完成。

④分析代码细节：这一实验任务中有以下几点需要注意：

- (1) **global** 关键字的作用：**global** 关键字的作用是将 `function_from_asm` 标记为一个全局符号，允许其他源文件（如 `main.cpp`）链接并调用这个汇编函数，告诉编译器和链接器这个符号需要对外部模块可见。如果没有 `global` 声明，其他文件就无法看到并调用这个汇编函数；
- (2) **extern** 关键字的作用：告诉编译器跟在 `extern` 后面的函数在其他文件中定义，在链接时需要解析这些符号。


```
asm_utils.h
~/lab4/assignment2/include

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

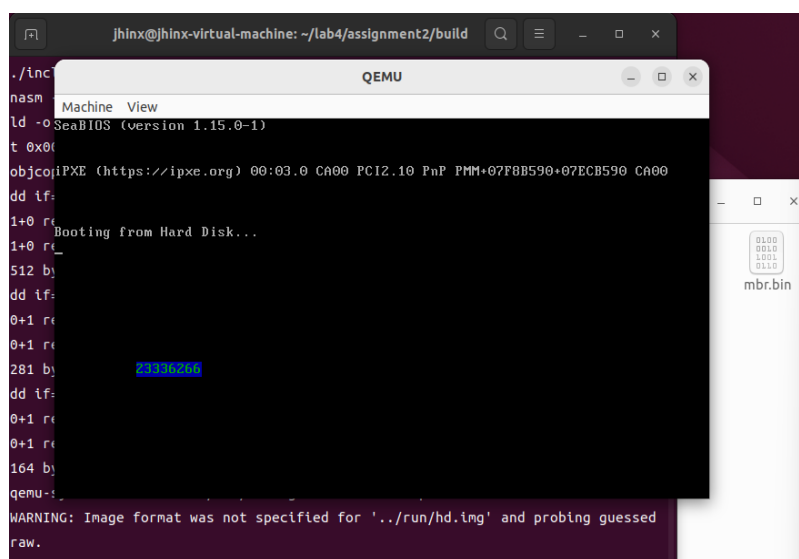
```
setup.cpp
~/lab4/assignment2/src/kernel

1 #include "asm_utils.h"
2
3 extern "C" void setup_kernel()
4 {
5     asm_show_number();
6     while(1) {
7     }
8 }
9 }
```

```
asm_utils.asm
~/lab4/assignment2/src/Utils

1 [bits 32]
2
3 global asm_show_number
4
5 asm_show_number:
6     push eax
7     xor eax, eax
8
9     mov ah, 0x12
10    mov al, '2'
11    mov [gs:2 * 1290], ax
12
13    mov al, '3'
14    mov [gs:2 * 1291], ax
15
16    mov al, '3'
17    mov [gs:2 * 1292], ax
18
19    mov al, '3'
20    mov [gs:2 * 1293], ax
21
22    mov al, '6'
23    mov [gs:2 * 1294], ax
24
25    mov al, '2'
26    mov [gs:2 * 1295], ax
27
28    mov al, '6'
29    mov [gs:2 * 1296], ax
30
31    mov al, '6'
32    mov [gs:2 * 1297], ax
33
34    pop eax
35    ret
```

在 `asm_utils.asm` 中，我们除了修改显示的字符，还修改了 `ah` 的值为 `0x12`，即蓝色前景色和蓝色背景色，以及修改了字符显示的位置，最终输出结果如下：



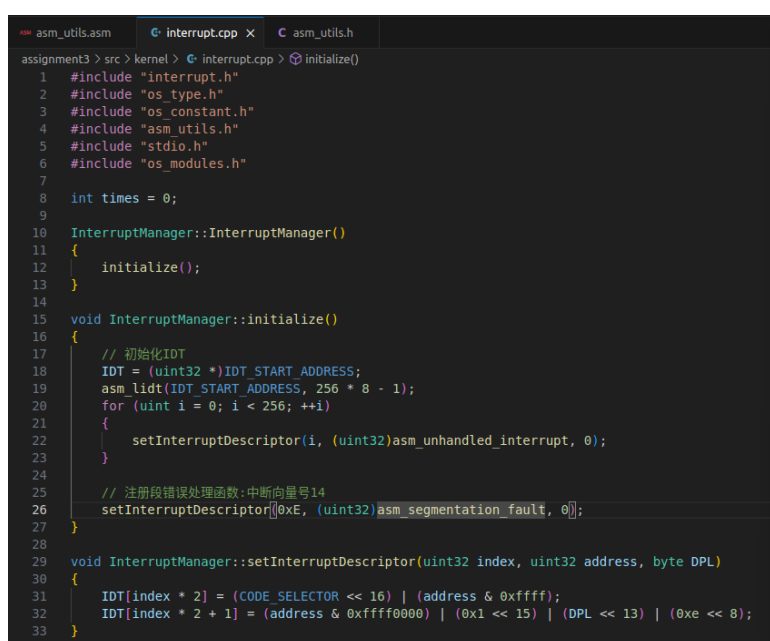
The screenshot shows a QEMU virtual machine window. The terminal displays assembly code being executed, including instructions like `ld -0 SeaBIOS (version 1.15.0-1)`, `objcopy`, and `dd if=`. A warning message at the bottom states: "WARNING: Image format was not specified for '../run/hd.img' and probing guessed raw." On the right side, a file explorer window shows a file named `mbr.bin`.

经过检验，我们可以看到程序成功输出了本人的学号，实验任务完成。

----- 实验任务 3（对应 assignment3） -----

- 任务要求：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。
- 实验步骤：由于实验指导已有源代码的详细解释和输出示例，这里我们仅展示需要修改的地方。

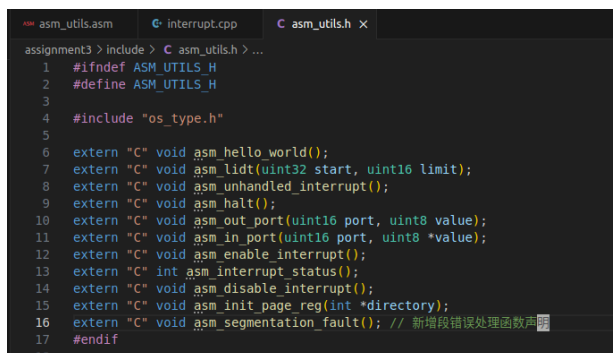
①注册段错误处理函数：首先，我们需要在中断描述符中注册段错误的中断处理函数。经过查询互联网，我们了解到段错误的中断向量号是 14。因此，我们需要在 `interrupt.cpp` 文件中作出如下修改：



The screenshot shows a code editor with the `interrupt.cpp` file open. The code includes headers like `interrupt.h`, `os_type.h`, `os_constant.h`, `asm_utils.h`, `stdio.h`, and `os_modules.h`. It defines a `times` variable and an `InterruptManager` class. The `initialize` method of `InterruptManager` is shown, which initializes the IDT and registers the segment fault interrupt handler. The handler is registered using `setInterruptDescriptor` with the index 14 and the address `asm_segmentation_fault`. The `setInterruptDescriptor` method is also shown, which sets the IDT entries for the given index and address.

这一操作其实就相当于函数重载，原本中断向量为 14 的地方采用的是默认的中断处理函数，这里我们用自定义的段错误处理函数覆盖了默认函数，从而实现段错误中断处理的注册。

② 声明段错误处理函数，这一部分没有什么好说明的，只需要在 `asm_utils.h` 中添加一行声明即可，修改如下：



```
asm_utils.asm | interrupt.cpp | C asm_utils.h x
assignment3 > include > C asm_utils.h > ...
1  #ifndef ASM_UTILS_H
2  #define ASM_UTILS_H
3
4  #include "os_type.h"
5
6  extern "C" void asm_hello_world();
7  extern "C" void asm_ldt(uint32 start, uint16 limit);
8  extern "C" void asm_unhandled_interrupt();
9  extern "C" void asm_halt();
10 extern "C" void asm_out_port(uint16 port, uint8 value);
11 extern "C" void asm_in_port(uint16 port, uint8 *value);
12 extern "C" void asm_enable_interrupt();
13 extern "C" int asm_interrupt_status();
14 extern "C" void asm_disable_interrupt();
15 extern "C" void asm_init_page_reg(int *directory);
16 extern "C" void asm_segmentation_fault(); // 新增段错误处理函数声明
17 #endif
18
```

③ 段错误处理函数的实现：这一部分是本任务的主要内容，段错误处理函数需要在出现段错误的时候中断函数，并且输出段错误相应的信息来提醒用户。这些内容可以仿照 `asm_utils.asm` 中的 `asm_unhandled_interrupt` 函数，只需修改一些具体内容即可。段错误处理函数实现代码如下：

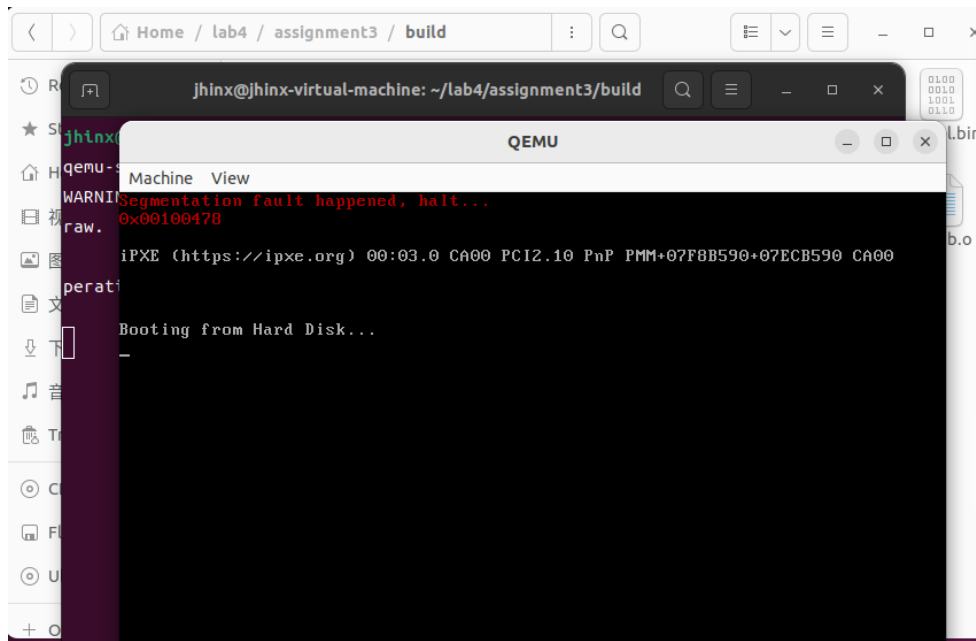
```

asm asm_utils.asm x 设置 interrupt.cpp C asm_utils.h
assignment3 > src > utils > asm_utils.asm
14 ASM_UNHANDLED_INTERRUPT_INFO db 'Unhandled interrupt happened, halt...'
16
17 ASM_PAGE_FAULT_INFO db 'Segmentation fault happened, halt...'
18 | | | | db 0
19
20 ASM_IDTR dw 0
21 | | dd 0
22
23 ; void asm_segmentation_fault()
24 asm_segmentation_fault:
25 cli
26 mov esi, ASM_PAGE_FAULT_INFO
27 xor ebx, ebx
28 mov ah, 0x04
29 .output_information:
30 cmp byte[esi], 0
31 je .end
32 mov al, byte[esi]
33 mov word[gs:ebx], ax
34 inc esi
35 add ebx, 2
36 jmp .output_information
37 .end:
38 ; 获取导致错误的地址 (存储在CR2寄存器中)
39 mov eax, cr2
40 ; 输出错误地址 (这里简化处理, 实际上可以格式化输出地址)
41 add ebx, 88 ; 下一行显示
42 mov ah, 0x04
43 mov al, '0'
44 mov word[gs:ebx], ax
45 add ebx, 2
46 mov al, 'x'
47 mov word[gs:ebx], ax
48 add ebx, 2
49
50 mov ecx, 8 ; 8个十六进制字符
51 mov edx, eax ; 保存原始地址
52
53 .hex_loop:
54 rol edx, 4 ; 循环左移4位
55 mov al, dl
56 and al, 0x0F ; 取低4位
57
58 ; 转换为ASCII字符
59 cmp al, 9
60 jbe .digit
61 add al, 7 ; A-F转换
62 .digit:
63 add al, '0' ; 0-9转换
64 mov [gs:ebx], ax
65 add ebx, 2
66 loop .hex_loop
67
68 jmp $
69

```

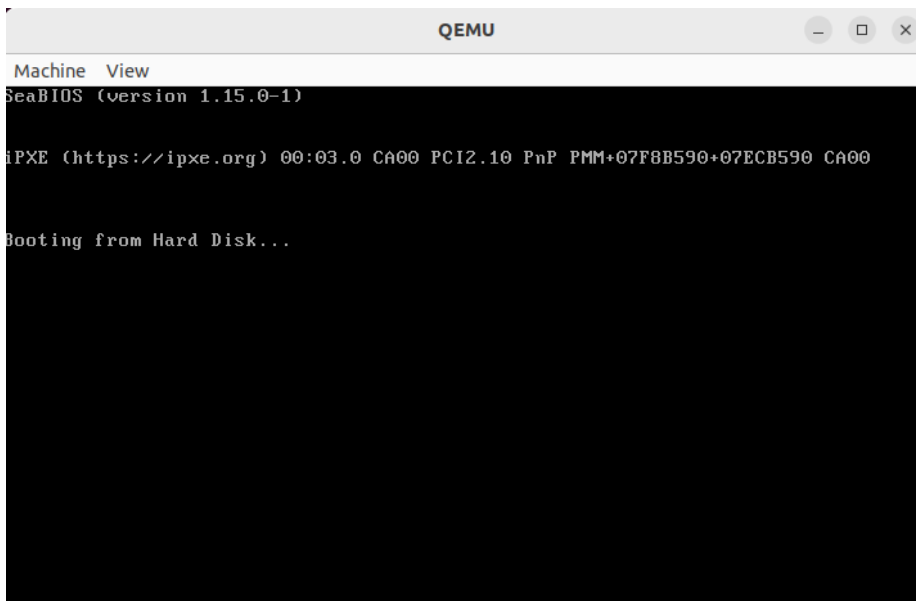
我们可以发现，除了输出段错误提示之外，我的函数还实现了输出段错误发生地址这一功能，这一部分的代码借助了 AI 的帮助，实现的原理是：Linux 内核有专门的寄存器 cr2 来存储发生错误的地址，我们将地址信息逐一转换为 ASCII 字符，并且输出在 qemu 显示屏的第二行中。这里的 add ebx, 88 是经过实践，第一行输出完段错误信息后，光标移动 44 格后刚好到第二行。

在不修改 setup.cpp 的情况下，即我们使用指令 `*(int*)0x100000 = 1;` 来触发段错误，程序运行结果如下：

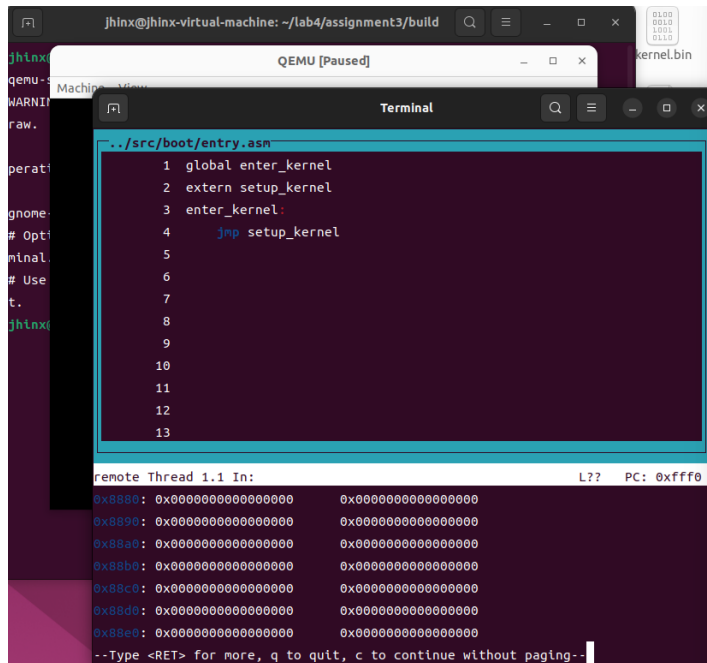


我们可以看到成功触发了段错误处理函数，输出了信息”segmentation fault happened, halt...”，并且在下一行输出了发生段错误的地址。程序设计成功。

作为对照，我们还截图了没有输出段错误的情况，如下：



④debug: 根据实验指导，我们需要查看我们是否已经放入中断描述符，我们在输入 `make debug` 进入 `gdb` 后，如果直接输入 `x/256gx 0x8880`，会发现相应的地址内容全都为 0，如下图所示：



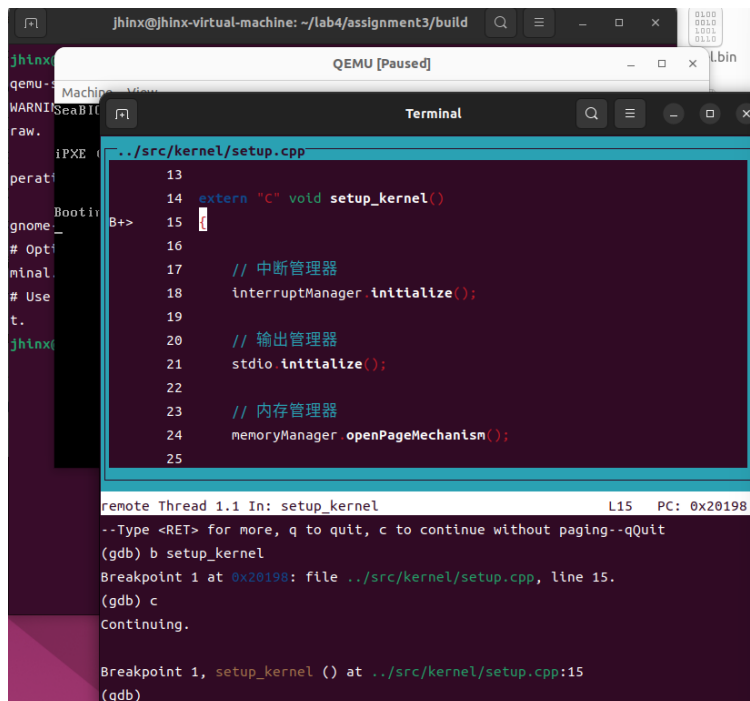
```
./src/boot/entry.asm
1 global enter_kernel
2 extern setup_kernel
3 enter_kernel:
4     jmp setup_kernel
5
6
7
8
9
10
11
12
13

remote Thread 1.1 In: L?? PC: 0xffff0
0x8880: 0x0000000000000000 0x0000000000000000
0x8890: 0x0000000000000000 0x0000000000000000
0x88a0: 0x0000000000000000 0x0000000000000000
0x88b0: 0x0000000000000000 0x0000000000000000
0x88c0: 0x0000000000000000 0x0000000000000000
0x88d0: 0x0000000000000000 0x0000000000000000
0x88e0: 0x0000000000000000 0x0000000000000000
--Type <RET> for more, q to quit, c to continue without paging--
```

经过询问同学，了解到是因为如果直接就查看地址，此时还没有运行 `setup_kernel` 函数，相当于实际上会注册中断描述符的函数还没有运行，因此我们需要输入如下指令

```
b setup_kernel
b 24
c
```

输入这些内容后，再次输入 `x/256gx 0x8880`，我们就能看到如下内容：



```
./src/kernel/setup.cpp
13
14 extern "C" void setup_kernel()
15
16
17 // 中断管理器
18 interruptManager.initialize();
19
20 // 输出管理器
21 stdio.initialize();
22
23 // 内存管理器
24 memoryManager.openPageMechanism();
25

remote Thread 1.1 In: setup_kernel L15 PC: 0x20198
--Type <RET> for more, q to quit, c to continue without paging--qQuit
(gdb) b setup_kernel
Breakpoint 1 at 0x20198: file ../src/kernel/setup.cpp, line 15.
(gdb) c
Continuing.

Breakpoint 1, setup_kernel () at ../src/kernel/setup.cpp:15
(gdb)
```

```
Terminal
../src/kernel/setup.cpp
16
17 // 中断管理器
18 interruptManager.initialize();
19
20 // 输出管理器
21 stdio.initialize();
22
23 // 内存管理器
24 memoryManager.openPageMechanism();
25
26 // 除零错误
27 // int t = 1 / 0;
28

remote Thread 1.1 In: setup_kernel L24 PC: 0x201be
0x8880: 0x00028e0000200b20 0x00028e0000200b20
0x8890: 0x00028e0000200b20 0x00028e0000200b20
0x88a0: 0x00028e0000200b20 0x00028e0000200b20
0x88b0: 0x00028e0000200b20 0x00028e0000200b20
0x88c0: 0x00028e0000200b20 0x00028e0000200b20
0x88d0: 0x00028e0000200b20 0x00028e0000200b20
0x88e0: 0x00028e0000200b20 0x00028e0000200b20
--Type <RET> for more, q to quit, c to continue without paging--
```

我们可以看到中断描述符信息成功录入到相应地址，实验任务完成。

⑤加分项，总结会触发段错误的方法：我们一共在 setup.cpp 中尝试了以下这 14 种可能的方法：

```
// 尝试方法 1: 访问 1MB 以上内存
//*(int*)0x100000 = 1;

// 尝试方法 2: 使用汇编直接触发页错误
// 通过直接调用会强制引发缺页异常的汇编指令
//asm(
//    //"movl $0x500000, %eax\n" // 5MB, 可能未映射
//    //"movl $1, (%eax)\n"      // 写入操作以触发错误
//);

// 尝试方法 3: 设置一个无效的指针并解引用
//int* badPtr = (int*)0xA0000000; // 远离已映射区域的地址
//*badPtr = 123;

// 尝试方法 4: 强制清零 CR3 寄存器（这将导致无效的页表基址）
//asm(
//    //"movl $0, %eax\n"
//    //"movl %eax, %cr3\n" // 设置无效的页目录表基址
//);

// 尝试方法 5: 使用函数指针调用无效地址的函数
//typedef void (*FuncPtr)();
//FuncPtr badFunction = (FuncPtr)0x12345678;
//badFunction();
```

```

// 尝试方法 6: 访问内核代码区域并尝试修改（只读区域）
//uint8 *kernelCode = (uint8 *)0x20000; // 内核代码的起始地址
//*kernelCode = 0x90; // 尝试修改内核代码（NOP 指令）

// 尝试方法 7: 尝试访问 4GB 虚拟地址空间边界
*(int*)(0xFFFFF000) = 1;

// 尝试方法 8: 尝试从用户态切换到特权指令（在用户态执行特权指令）
//asm(
    // "pushf\n"          // 保存标志寄存器
    // "cli\n"            // 清除中断标志（特权指令）
    // "hlt\n"            // 停机（特权指令）
//);

// 尝试方法 9: 非对齐访问（某些架构会因此触发异常）
//int *unaligned = (int*)0x10001;
//*unaligned = 0x12345678;

//尝试方法 10A: 数组越界访问
//int arr[5];
//arr[10] = 42; // 越界访问

//尝试方法 10B: 数组越界访问
//int arr[5];
//arr[100000000] = 42; // 越界访问

//尝试方法 11: 修改只读内存
//char *str = "hello"; // 字符串常量存储在只读段
//str[0] = 'H';         // 尝试修改只读内存

//尝试方法 12: 访问非法地址
//int *ptr = (int *)0x12345678;
//*ptr = 42;

//尝试方法 13: 执行非代码段
//void (*func)() = (void (*)())0xdeadbeef;
//func(); // 跳转到非法地址执行

// 尝试方法 14: 尝试访问其他进程的内存（通常会被操作系统阻止）
//int *ptr = (int *)0x7ff000000; // 假设是其他进程的地址
//*ptr = 42;

```

注：上面的注释和触发方式均为本人询问大模型后生成。

首先有一个不理解的地方，就是尝试方法 4 会导致 qemu 启动后立即闪退，可能是因为 cr3 寄存器受到内核的保护，清零 cr3 这一操作是系统不允许的？

此外，还有一些操作比如访问空指针（`int *ptr=NULL;`）由于过不了编译，就没有尝试是否触发段错误……

通过上面的尝试，我们总结出了一些规律：

在本次实验中，由于我们在 `memory.cpp` 中作出了以下定义：我们实际上对 0-1MB 的内存映射到页表的前 256 页，而其他地址还未进行注册，也即虚拟内存地址大于等于 0x100000 的，都没有登记。因此起初助教给出的触发段错误的方式就是访问了第一个没有被登记的虚拟内存地址。

然后，通过观察上面 14 种方法的代码我们不难发现，其实无论 AI 对这个尝试方法的描述是怎麼样的，也无论这个尝试方法的代码如何，只有涉及到访问的内存大于等于 0x100000，就会触发段错误，比如 `movl $0x500000, (int*)0xA0000000`、`(FuncPtr)0x12345678`、`(void (*)())0xdeadbeef` 等，其实都是同样的道理。

然后对于数组越界操作，即尝试方法 10A 和 10B，经过检验我们发现 10A 不会触发段错误，10B 会触发段错误。思考后我认为这一现象的原理如下：我们都知道数组采用的是偏移寻址，首地址会设置在我们注册过的 1kb 中的某一个地方，当我们只越界了“一点点”的时候比如 10A 中的 `arr[10]`，偏移寻址求得的地址依然在注册了的区域内，因此不会发生段错误。

如果这样说的话，由于我们只注册了 4kb 的内存，而一个 `int` 类型变量大小为 4b，按道理来说，`arr[1025]` 就必然会引发段错误。但是实际上我们经过检验发现 `arr[1025]` 仍然不会引发段错误，经过询问助教，可能是因为整个内核的其他程序已经申请了这个注册的页面后面的地址，因此偏移到这些地址的时候依然不违规。

经过二分查找，我尝试出了在我的电脑中，`arr[254215]` 不会触发段错误，而 `arr[254216]` 会触发段错误，我们可以断定，这个偏移地址就是整个注册页面的边界。虽然找到了这个边界好像没什么用，而且我也因此留下了疑问：每个同学用自己的虚拟机做这个实验，得到的边界是一样的吗？是否在后面实验进行过程中，会因为后面实验的代码而使得这个边界产生了改变？这两个问题还有

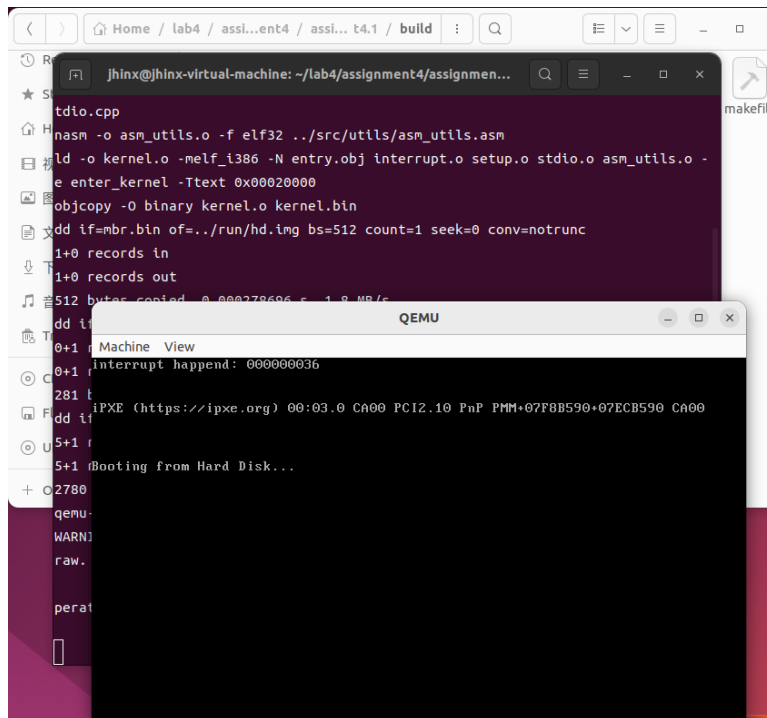
待考证。

最后还有一个疑点，就是对于尝试方法 7，AI 一开始给出的代码是 `*(int*)(0xFFFFFFFFFC) = 1`;很明显这个地址超出了我们注册的 4kb 地址，但是却没有触发段错误。后来经过询问助教，得知 Linux 内核好像是有一个机制，会把最后一个页面指向回最开始的页面，而 `0xFFFFFFFFxxx`（这里的 xxx 可以是 0-F 中的任何一个数）对应的就是最后一个页面。经过实践发现，当我们改成 `0xFFFFFEFFF`，也就是上一个页面的末尾时，确实会触发中断，而改成 `0xFFFFF000`，也就是最后一个页面的最前面，的确不会触发中断。至于这个机制是否真实存在，或者说是不是的确因为这个原因产生的现象，还有待考证。

至此，我对思考题的全部理解就到这里了，assignment3 的结果展示结束。

----- 实验任务 4 （对应 assignment4） -----

- 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。将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。
- 实验步骤：①复现 example4，由于这一部分和上面的实验区别不大，因此我们直接展示输出结果后的截图：



②编写程序：首先我们需要声明一个新的类来实现字符回旋程序，这个类的代码在新建的文件`/include/Character_Rotation.h`中。我们运用c++关于类的知识，并仿照中断处理类的声明，写出了如下代码：

```

// 移动方向枚举
enum Direction{
    RIGHT=0,
    DOWN=1,
    LEFT=2,
    UP=3
};

class CharacterRotation{
private:
    //光标当前位置
    uint position;
    //光标当前前进方向
    Direction direction;
    //字符前景色
    uint8 front_color;
    //字符背景色
    uint8 back_color;
    //计数器:记录当前背景色已经输出了多少个字符
    uint counter;
    //计数器:记录当前前景色已经输出了多少个字符
    uint8 color_counter;
    //当前字符索引:记录现在是“数组”中的第几个数
    uint8 current_char;
    //“数组”:用于存储伪随机数
    char chars[10];

public:
    CharacterRotation();
    void initialize();
    //更新位置
    void update_position();
    //显示当前字符
    void show_current_char();
    //在时钟中断中调用的主要方法
    void rotate();
};

```

其中 private 声明的变量其实都是 lab2 中完全一样的变量，只是变量类型我们替换成了 os_types.h 中 typedef 后的名字，而 public 中声明的函数都是 lab2 中完全一样的函数，不同的是：我们现在不需要延时函数了，我们现在只需要通过已有的时钟中断来输出字符。

其次，为了采用时钟中断来运行字符回旋程序，我们需要修改时钟中断处理函数的实现，让他在触发时钟中断时运行我们刚刚定义的类中的实现函数，修改 interrupt.cpp 的内容如下：


```

#include "interrupt.h"
#include "os_type.h"
#include "os_constant.h"
#include "asm_utils.h"
#include "stdio.h"
#include "Character_Rotation.h"

extern STDIO stdio;
extern CharacterRotation my_program;

```

我们要在这一文件中 `extern` 一个我们自己的类，以为了下面代码中调用类中的函数，即把函数的实现改成调用这个类的 `rotate()` 函数。

```

89 // 中断处理函数
90 extern "C" void c_time_interrupt_handler()
91 {
92
93     my_program.rotate();
94 }

```

当然，想要运行我们自己的类的函数，我们还需要在内核启动的时候对类进行初始化，因此，模仿中断类的初始化，我们在 `setup.cpp` 文件中做了如下修改，主要修改的地方是第 20 行：

```

character_rotation.cpp
setup.cpp

1 #include "asm_utils.h"
2 #include "interrupt.h"
3 #include "stdio.h"
4 #include "Character_Rotation.h"
5
6 // 屏幕Io处理器
7 STDIO stdio;
8 // 中断管理器
9 InterruptManager interruptManager;
10 // 字符旋转器
11 CharacterRotation my_program;
12
13 extern "C" void setup_kernel()
14
15 // 中断处理部件
16 interruptManager.initialize();
17 // 屏幕Io处理部件
18 stdio.initialize();
19 // 字符回旋程序初始化
20 my_program.initialize();
21
22 interruptManager.enableTimeInterrupt();
23 interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);
24 asm_enable_interrupt();
25 asm_halt();
26

```

最后，我们再花比较大的篇幅来讲解我们这个字符回旋类的实现：我们在

src/kernel 中新建了一个文件，叫 character_rotation.cpp，它的内容如下：

```
#include "Character_Rotation.h"
#include "stdio.h"
#include "os_modules.h"

extern STDIO stdio;

CharacterRotation::CharacterRotation(){
    initialize();
}

void CharacterRotation::initialize(){
    position=-2;
    direction=RIGHT;
    front_color=1;
    back_color=0x20;
    counter=0;
    color_counter=0;
    current_char=0;

    //初始化“数组”
    chars[0]='1';
    chars[1]='3';
    chars[2]='5';
    chars[3]='7';
    chars[4]='9';
    chars[5]='0';
    chars[6]='2';
    chars[7]='4';
    chars[8]='6';
    chars[9]='8';

    //显示学号
    char message[]="jhinx 23336266";
    for(int i=0;message[i]!='\0';i++){
        stdio.print(12,32+i,message[i],0xEC);
    }
}

void CharacterRotation::update_position(){
    uint newPos=position;

    switch(direction){
        case RIGHT:
```

```

        newPos+=2;
        if(newPos==160){
            //到达右边界
            newPos-=2;
            newPos+=160;//向下移动一行
            direction=DOWN;
        }
        break;

    case DOWN:
        newPos+=160;
        if(newPos>=4000){
            //到达下边界
            newPos-=160;
            newPos-=2;//向左移动一列
            direction=LEFT;
        }
        break;

    case LEFT:
        newPos-=2;
        if(newPos==3838){
            //到达左边界
            newPos+=2;
            newPos-=160;//向上移动一行
            direction=UP;
        }
        break;

    case UP:
        newPos-=160;
        if(newPos==-160){
            //到达上边界
            newPos+=160;
            newPos+=2;//向右移动一列
            direction=RIGHT;
        }
        break;
    }

    position=newPos;
}

void CharacterRotation::show_current_char(){

```

```

//获取当前要显示的字符
char ch=chars[current_char];

//设置颜色属性
uint8 color=back_color+front_color;

//计算屏幕位置的行列
uint row=position/160;
uint col=(position%160)/2;

//使用 stdio 显示字符
stdio.print(row,col,ch,color);

//更新颜色计数器
color_counter++;
if (color_counter>=3){
    color_counter=0;
    front_color++;
    if(front_color>=16){
        front_color=1;
    }
}

//更新字符索引
current_char++;
if(current_char>=10){
    current_char=0;
}

//更新计数器和背景色
counter++;
if(counter%14==0){
    back_color+=0x10;
    if(back_color==0){
        back_color=0x20;
    }
}
}

void CharacterRotation::rotate(){
    update_position();
    show_current_char();
}

```

首先，我们实现了字符回旋类的初始化（即 `initialize()`），内容包括给

各个变量赋初值，以及在屏幕中间显示学号，在这里显示学号，我们直接使用了 `stdio` 封装的函数。

随后，我们实现了更新光标的函数，其逻辑是通过我们前面实验学到的 `qemu` 光标坐标计算公式，以及对于光标移动方向的判断，实现了光标的回旋，在书写这一程序的时候，对于边界条件的判断起初不太顺利，最后是通过不断尝试并且修改，得到了最终正确的边界条件判断。

接着，我们实现了输出相应前景色和背景色字符的函数。在这里我们把前面用公式转换成的光标位置重新拆分成了行和列，并且借助了 `stdio` 封装的函数进行输出，而更新颜色和字符的部分比较简单，这里就不再阐述了。

最后，我们实现了最终整个类的执行函数：`rotate()`，它的内容是先调用更新坐标函数，再调用显示字符函数。

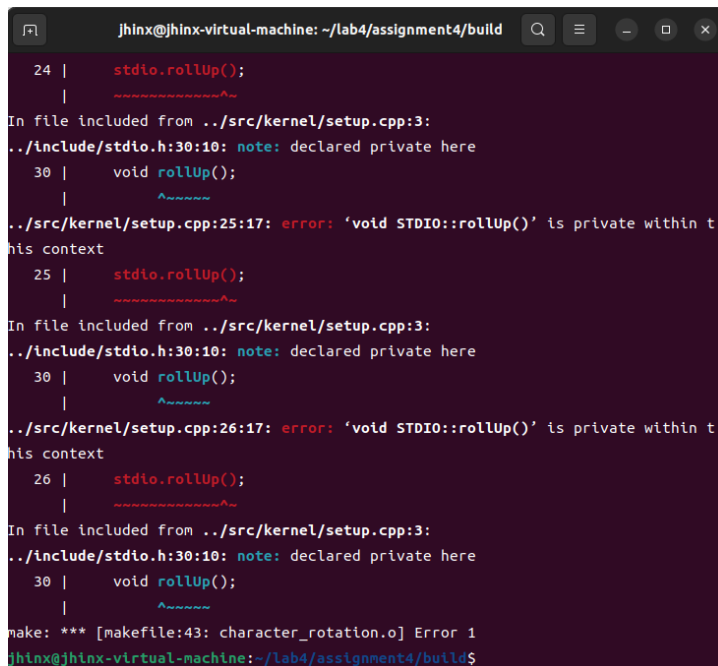
需要注意的是，在 `lab2` 的时候，我们有一个延迟函数来减慢字符输出频率，我们可以通过改变延迟函数中的循环次数来改变输出的快慢。但是在这一实验中，由于我们通过时钟中断来输出字符，每有一次时钟中断就会输出一个字符，而本人尚未发现如何控制时钟中断发生的频率。因此，即使引入延时函数，是不是通过时钟中断来实现字符回旋这一方法，只能减慢，不能加快？这一问题有待商榷。

最后，我们附上代码执行的结果：



此外，我们在 `lab2` 中通过 `int 10h` 中断实现了 `qemu` 显示屏的清屏，而在这一实验中，起初我想借用 `stdio` 的 `rollUp()` 函数来实现清屏，但是编译的时

候显示这一函数是 `stdio` 的 `private` 函数，如下图所示：



```
jhinx@jhinx-virtual-machine: ~/lab4/assignment4/build
24 |     stdio.rollUp();
    |
In file included from ../src/kernel/setup.cpp:3:
../include/stdio.h:30:10: note: declared private here
30 |     void rollUp();
    |
../src/kernel/setup.cpp:25:17: error: 'void STDIO::rollUp()' is private within t
his context
25 |     stdio.rollUp();
    |
In file included from ../src/kernel/setup.cpp:3:
../include/stdio.h:30:10: note: declared private here
30 |     void rollUp();
    |
../src/kernel/setup.cpp:26:17: error: 'void STDIO::rollUp()' is private within t
his context
26 |     stdio.rollUp();
    |
In file included from ../src/kernel/setup.cpp:3:
../include/stdio.h:30:10: note: declared private here
30 |     void rollUp();
    |
make: *** [makefile:43: character_rotation.o] Error 1
jhinx@jhinx-virtual-machine:~/lab4/assignment4/build$
```

随后，本人采用了一些“旁门左道”的办法，即把源代码中的 `private` 给注释掉，如下图所示：

```
//private:
// 滚屏
void rollUp();
};
```

并且在 `setup.cpp` 中加入了以下代码：

```
extern "C" void setup_kernel()
{
    // 中断处理部件
    interruptManager.initialize();
    // 屏幕io处理部件
    stdio.initialize();
    stdio.rollUp();
    stdio.rollUp();
    stdio.rollUp();
    stdio.rollUp();
    stdio.rollUp();
    stdio.rollUp();
    stdio.rollUp();
    stdio.rollUp();
    // 字符回旋程序初始化
    my_program.initialize();

    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);
    asm_enable_interrupt();
    asm_halt();
}
```

因为每次 `rollUp()` 会使 `qemu` 屏幕往上移动一行，因此经过实际测试，我们运行 8 次 `rollUp()` 后，刚好可以把 `qemu` 显示屏附带的信息给清除掉。当然，我们也可以通过修改 `rollUp()` 函数的实现，改为一次性往上移动八行来

达成同样的目的，不过这个区别无关紧要，虽然现在这样代码丑陋了一点，但是我们总归是达成了目的。

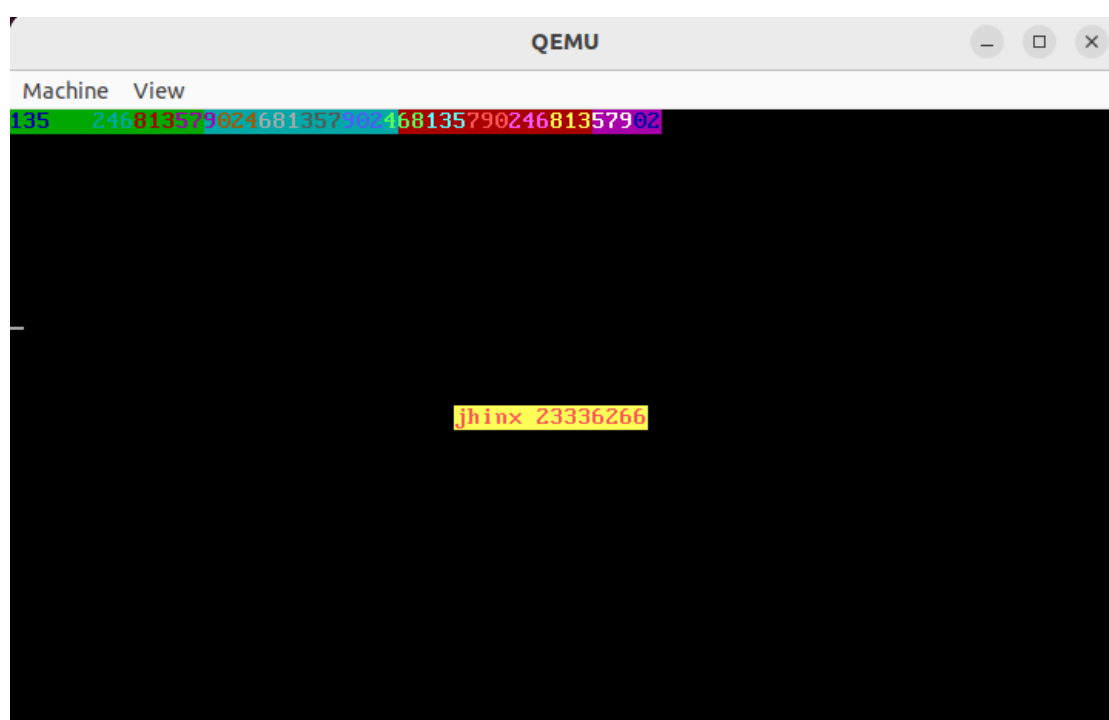
（本人“于心不忍”，在后来又补充了刚刚说的实现方式，修改 `stdio.cpp` 如下图：

```
void STDIO::rollUp()
{
    uint length;
    length = 25 * 80;
    for (uint i = 720; i < length; ++i)
    {
        screen[2 * (i - 720)] = screen[2 * i];
        screen[2 * (i - 720) + 1] = screen[2 * i + 1];
    }

    for (uint i = 24 * 80; i < length; ++i)
    {
        screen[2 * i] = ' ';
        screen[2 * i + 1] = 0x07;
    }
}
```

通过修改 `screen` 数组中的数值，我们就能将更多行往上移动，从而实现只用一个 `stdio.rollUp()` 就能清除掉所有的自带文字这一效果。

最后运行结果如下：



经过检验，运行情况符合要求，实验任务完成。

Section 5 实验总结与心得体会

本次实验我们了解了中断是如何介入到系统中并且发挥作用的，以及了解到

了中断他不仅仅可以中断程序，还可以通过自定义中断后的行为，通过中断实现自己想要达成的目的。与此同时，通过 assignment3 中触发段错误的尝试，我们了解到其实我们自己实现的系统（无论是前面 4 个实验还是将来会做的实验）其实是很稚嫩的，他没有更多的符合常理的约束条件来保护系统免受非法操作的破坏。不过，正是这一“稚嫩”，可以为我们将来的学习注入更多的动力，不断精益求精。但是话又说回来，毕竟时间有限，对于中断的研究就到这里吧（^_^）。

Section 6 附录：代码清单

代码请见附件。