



中山大學  
SUN YAT-SEN UNIVERSITY

# LAB4 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: 中断

专业名称: 计算机科学与技术

学生姓名: 马福泉

学生学号: 23336179

实验地点: 实验中心 B202

实验时间: 2025.4.9

## 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。
- 通过本章的学习，同学们将掌握使用 C 语言来编写内核的方法，理解保护模式的中断处理机制和处理时钟中断，为后面的二级分页机制和多线程/进程打下基础。

## Section 2 预备知识与实验环境

- 预备知识：汇编语言基础，计算机体系结构，操作系统基础，调试工具的使用（如 gdb）等。
- 实验环境：
  - 虚拟机版本/处理器型号：VMware-Ubuntu22.04.5/i386（32 位）
  - 代码编辑环境： 编辑器：VSCode  
插件：C/C++ 插件，汇编插件
  - 代码编译工具： 编译器：gcc  
工具链：binutils、make、qemu、nasm 等  
调试工具：gdb
  - 重要三方库信息：GNU binutils：工具集  
gdb：调试工具  
QEMU、Bochs：模拟器用于测试 等

## Section 3 实验任务

**实验任务 1:** 混合编程的基本思路。复现网址中“一个混合编程的例子”部分。要求:

1. 将原例子中最后一行的输出"Done"改为"Done by 学号 姓名首字母"
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"` 等, 保存结果截图并说说你是怎么做的;
3. 学习 `make` 的使用, 并用 `make` 来构建项目, 保存结果截图并说说你是怎么做。

**实验任务 2:** 使用 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”, 保存结果截图并说说你是怎么做的


**实验任务 3:** 中断的处理。复现网址中“初始化 IDT”部分, 你可以更改默认的中断处理函数为你编写的函数, 然后触发之, 对结果进行截图并说说你是怎么做的。要求: 调用处理函数时输出个人学号或姓名信息。

**实验任务 4:** 时钟中断的处理。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。注意: 不要使用纯汇编的方式来实现。

## Section 4 实验步骤与实验结果

### ----- 实验任务 1 -----

- 任务要求: 复现网址中“一个混合编程的例子”部分。
- 思路分析: 修改 `main.cpp`, 编写 `makefile`, 编译运行
- 实验步骤:
  - ① 修改 `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 by 23336179 MFQ" << std::endl;
9  }

```

## ② 编写 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

```

## ③ 编译整个项目

```
make
```

这会依次执行：

将 .c/.cpp 编译为 .o 文件

将 .asm 汇编为 .o 文件

链接所有.o 生成 main.out

## ④ 运行程序

```
./main.out
```

- ⑤ 清理所有中间文件（但保留 main.out 可执行文件）

```
make clean
```

- 实验结果展示：

```
mafq5@mafq5-virtual-machine:~/lab4$ 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_func.o -f elf32 asm_func.asm
g++ -o main.out main.o c_func.o cpp_func.o asm_func.o -m32
mafq5@mafq5-virtual-machine:~/lab4$ ./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done by 23336179 MFQ
```

## ----- 实验任务 2 -----

- 任务要求：使用 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”，保存结果截图并说说你是怎么做的
- 思路分析：修改代码，可以修改一下接口函数声明，重新运行即可。
- 实验步骤：

### 1.修改 kernel\_setup 为 asm\_show\_string



The screenshot shows two code editors. The top editor, titled 'asm\_utils.h' with path '~/lab4/task2/include', contains the following code:

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

The bottom editor, titled 'setup.cpp' with path '~/lab4/task2/src/kernel', contains the following code:

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

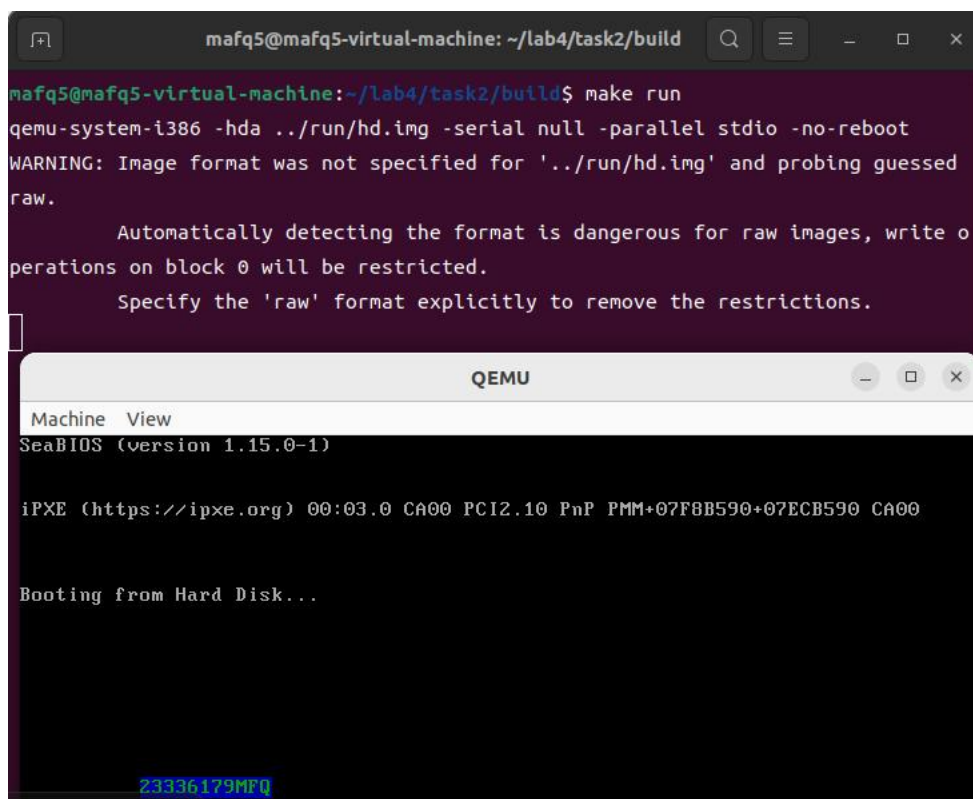
### 2 修改 asm\_utils.asm

仿照源代码，逐个字符写入到指定的内存地址中（通过 gs 段寄存器访问）



```
1  [bits 32]
2
3  global asm_show_string
4
5  asm_show_string:
6      push eax
7      xor eax, eax
8      mov ah, 0x12 ; 设置高字节为0x12（根据需要调
9      整）
10     mov al, '2'
11     mov [gs:2 * 1290], ax
12     mov al, '3'
13     mov [gs:2 * 1291], ax
14     mov al, '3'
15     mov [gs:2 * 1292], ax
16     mov al, '3'
17     mov [gs:2 * 1293], ax
18     mov al, '6'
19     mov [gs:2 * 1294], ax
20     mov al, '1'
21     mov [gs:2 * 1295], ax
22     mov al, '7'
23     mov [gs:2 * 1296], ax
24     mov al, '9'
25     mov [gs:2 * 1297], ax
26     mov al, 'M'
27     mov [gs:2 * 1298], ax
28     mov al, 'F'
29     mov [gs:2 * 1299], ax
30     mov al, 'Q'
31     mov [gs:2 * 1300], ax
32     pop eax
33     ret
34
```

- 实验结果展示



```
mafq5@mafq5-virtual-machine: ~/lab4/task2/build
mafq5@mafq5-virtual-machine:~/lab4/task2/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.

QEMU

Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00

Booting from Hard Disk...

23336179MFQ
```

### ----- 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，你可以更改默认的中断处理函数为你编写的函数，然后触发之，对结果进行截图并说说你是怎么做的。要求：调用处理函数时输出个人学号或姓名信息。

- 思路分析：

- 1.初始化中断描述符表（IDT），设置中断向量的处理函数。
2. 替换默认的中断处理函数，编写一个自定义的中断处理函数，使其在触发中断时输出个人学号信息。
3. 触发一个已知的中断（如除零异常），验证自定义中断处理函数是否正确执行

- 实验步骤：

1. 注册段错误处理函数：在 interrupt.cpp 文件中，将中断向量号为 14 的默认中断处理函数替换为自定义的段错误处理函数。

```
interrupt.cpp
~/lab4/task3/src/kernel

1 #include "interrupt.h"
2 #include "os_type.h"
3 #include "os_constant.h"
4 #include "asm_utils.h"
5 #include "stdio.h"
6 #include "os_modules.h"
7
8 int times = 0;
9
10 InterruptManager::InterruptManager()
11 {
12     initialize();
13 }
14
15 void InterruptManager::initialize()
16 {
17     // 初始化IDT
18     IDT = (uint32 *)IDT_START_ADDRESS;
19     asm_lidt(IDT_START_ADDRESS, 256 * 8 - 1);
20     for (uint i = 0; i < 256; ++i)
21     {
22         setInterruptDescriptor(i, (uint32)asm_unhandled_interrupt, 0);
23     }
24
25     // 注册段错误处理函数:中断向量号14
26     setInterruptDescriptor(0xE, (uint32)asm_segmentation_fault, 0);
27 }
28
29 void InterruptManager::setInterruptDescriptor(uint32 index, uint32 address, byte DPL)
30 {
31     IDT[index * 2] = (CODE_SELECTOR << 16) | (address & 0xffff);
32     IDT[index * 2 + 1] = (address & 0xffff0000) | (0x1 << 15) | (DPL << 13) | (0xe << 8);
33 }
```

2. 声明段错误处理函数: 在 `asm_utils.h` 文件中添加段错误处理函数的声明。

```
asm_utils.h
~/lab4/task3/include

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_lidt(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
```

3. 实现段错误处理函数: 在 `asm_utils.asm` 文件中, 实现段错误处理函数, 输出段错误提示信息和发生段错误的地址。寄存器 `cr2` 存储发生错误的地址, 我们借助 `ai` 可以简单编写程序输出错误地址。

(代码比较长, 字体较小, 可以打开附录看)

```

1  [bits 32]
2
3  global asm_hello_world
4  global asm_lidt
5  global asm_unhandled_interrupt
6  global asm_halt
7  global asm_out_port
8  global asm_in_port
9  global asm_enable_interrupt
10 global asm_disable_interrupt
11 global asm_interrupt_status
12 global asm_init_page_reg
13 global asm_segmentation_fault
14 ASM_UNHANDLED_INTERRUPT_INFO db 'Unhandled interrupt happened, halt...'
15                                db 0
16 ASM_PAGE_FAULT_INFO db '23336179_MFQ_interrupt'
17                                db 0
18 ASM_IDTR dw 0
19            dd 0
20 ;;;;
21 asm_segmentation_fault:
22     cli
23     mov esi, ASM_PAGE_FAULT_INFO
24     xor ebx, ebx
25     mov ah, 0x04
26 .output_information:
27     cmp byte[esi], 0
28     je .end
29     mov al, byte[esi]
30     mov word[gs:ebx], ax
31     inc esi
32     add ebx, 2
33     jmp .output_information
34 .end:
35     ; 获取导致段错误的地址（存储在CR2寄存器中）
36     mov eax, cr2
37     ; 输出错误地址（这里简化处理，实际上可以格式化输出地址）
38     add ebx, 116; 下一行显示
39     mov ah, 0x04
40     mov al, '0'
41     mov word[gs:ebx], ax
42     add ebx, 2
43     mov al, 'x'
44     mov word[gs:ebx], ax
45     add ebx, 2
46
47     mov ecx, 8            ; 8个十六进制字符
48     mov edx, eax          ; 保存原始地址
49
50 .hex_loop:
51     rol edx, 4            ; 循环左移4位
52     mov al, dl
53     and al, 0x0F          ; 取低4位
54
55     ; 转换为ASCII字符
56     cmp al, 9
57     jbe .digit
58     add al, 7             ; A-F转换
59 .digit:
60     add al, '0'           ; 0-9转换
61     mov [gs:ebx], ax
62     add ebx, 2
63     loop .hex_loop
64
65     jmp $

```

4. 调试验证：使用 GDB 调试工具，通过设置断点和查看中断描述符表的内容，验证段错误处理函数是否已经成功注册。

```
list
b setup_kernel
c
b 26
x/256gx 0x8880
```

### 注意指令顺序

**b setup\_kernel**：设置断点在 `setup_kernel` 函数的入口处。这样可以确保在进入 `setup_kernel` 函数之前暂停程序执行。

**c**：继续运行程序，直到触发上述设置的断点。

**b 24**：在 `setup_kernel` 函数的第 24 行设置断点。

`x/256gx 0x8880` 是 GDB 的内存查看命令，用于查看从地址 `0x8880` 开始的 256 个 8 字节（64 位）的内容。这个命令用于查看中断描述符表的内容。**x**：表示查看内存内容。**/256g**：表示查看 256 个 8 字节（64 位）的内容。**x 0x8880**：表示从地址 `0x8880` 开始查看。



```
终端
../src/kernel/setup.cpp
14  extern "C" void setup_kernel()
B+> 15  {
16      // 中断管理器
17      interruptManager.initialize();
18
19      // 输出管理器
20      stdio.initialize();
21
22      // 内存管理器
23      memoryManager.openPageMechanism();
24
25      *(int*)0x100000 = 1;
b+ 26
27
28      asm_halt(); // 这行代码不会执行，因为之前的代码会触发段错误
29  }
```

```
remote Thread 1.1 In: setup_kernel L15 PC: 0x20198
(gdb) c
Continuing.

Breakpoint 1, setup_kernel () at ../src/kernel/setup.cpp:15
warning: Source file is more recent than executable.
(gdb) b 24
Breakpoint 2 at 0x201ce: file ../src/kernel/setup.cpp, line 26.
```

### ● 实验结果展示：

```
终端
../src/kernel/setup.cpp
10 // 内存管理器
11 MemoryManager memoryManager;
12
13
14 extern "C" void setup_kernel()
B+> 15 {
16     // 中断管理器
17     interruptManager.initialize();
18
19     // 输出管理器
20     stdio.initialize();
21
22     // 内存管理器

remote Thread 1.1 In: setup_kernel L15 PC: 0x20198
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--
```

```
终端
../src/kernel/setup.cpp
21
22 // 内存管理器
23 memoryManager.openPageMechanism();
24
25 *(int*)0x100000 = 1;
B+> 26
27
28 asm_halt(); // 这行代码不会执行，因为之前的代码会触发段错误
29 }
30
31
32
33

remote Thread 1.1 In: setup_kernel L26 PC: 0x201ce
0x8880: 0x00028e0000200b16 0x00028e0000200b16
0x8890: 0x00028e0000200b16 0x00028e0000200b16
0x88a0: 0x00028e0000200b16 0x00028e0000200b16
0x88b0: 0x00028e0000200b16 0x00028e0000200b16
0x88c0: 0x00028e0000200b16 0x00028e0000200b16
0x88d0: 0x00028e0000200b16 0x00028e0000200b16
0x88e0: 0x00028e0000200b16 0x00028e0000200b16
--Type <RET> for more, q to quit, c to continue without paging--
```

## ----- 实验任务 4 -----

- 任务要求：时钟中断的处理。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。注意：不要使用纯汇编的方式来实现。
- 思路分析：

① 首先我们需要声明一个新的类来实现走马灯程序，这个类的代码在新建的文件/include/Marquee.h

```
1  #ifndef MARQUEE_H
2  #define MARQUEE_H
3
4  #include "os_type.h"
5
6  class Marquee {
7  private:
8      // 要显示的文本
9      char text[20];
10     // 文本长度
11     uint8 text_length;
12     // 当前位置
13     int position;
14     // 屏幕宽度
15     int screen_width;
16     // 前景色
17     uint8 front_color;
18     // 背景色
19     uint8 back_color;
20     // 颜色变化计数器
21     uint8 color_counter;
22 public:
23     Marquee();
24     void initialize();
25     // 更新并显示跑马灯
26     void update();
27 };
28 #endif
```

② 为了采用时钟中断来运行走马灯程序，我们需要修改 `interrupt.cpp` 的内容，修改时钟中断处理 `extern "C" void c_time_interrupt_handler()` 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实现跑马灯的函数，并且我们要在这一文件中 `extern` 一个我们自己的类的对象，以为了下面代码中调用类中的函数。

```
1 // 中断处理函数
2 extern "C" void c_time_interrupt_handler()
3 {
4     // 增加时钟中断计数
5     times++;
6
7     // 调用跑马灯更新函数
8     marquee.update();
9
10    // 向主片发送EOI信号
11    asm_out_port(0x20, 0x20);
12 }
```

```
1 #include "interrupt.h"
2 #include "os_type.h"
3 #include "os_constant.h"
4 #include "asm_utils.h"
5 #include "stdio.h"
6 #include "Marquee.h"
7
8 extern STDIO stdio;
9 extern Marquee marquee; // 声明外部跑马灯对象
10
11 int times = 0;
12
13 InterruptManager::InterruptManager()
14 {
15     initialize();
16 }
17
```

③ 重点是我们在 `src/kernel` 中新建了一个文件，叫 `maquee.cpp`，

#### 初始化：

设置文本内容、长度、初始位置、屏幕宽度、前景色和背景色。

清除屏幕第一行，用背景色填充。

直接使用了 `stdio` 封装的函数

#### 更新显示：

每次更新时，先清除屏幕第一行。

计算文本字符在屏幕上的位置，实现循环滚动。

使用动态计算的颜色显示文本。


更新文本位置和背景色，实现滚动和颜色变化效果。

#### 核心逻辑：

通过 `position` 和取模操作实现循环滚动。

通过颜色计数器和取模操作实现颜色变化。

循环调用 `update()` 函数即可实现跑马灯效果，在 `extern "C" void c_time_interrupt_handler()`中调用。



```
1  #include "Marquee.h"
2  #include "stdio.h"
3  #include "os_modules.h"
4
5  extern STDIO stdio;
6
7  Marquee::Marquee() {
8      initialize();
9  }
```

```
1 void Marquee::initialize() {
2     text[0] = '2';
3     text[1] = '3';
4     text[2] = '3';
5     text[3] = '3';
6     text[4] = '6';
7     text[5] = '1';
8     text[6] = '7';
9     text[7] = '9';
10    text[8] = 'M';
11    text[9] = 'F';
12    text[10] = 'Q';
13    text[11] = '\0';
14
15    text_length = 11; // 文本长度更新为11
16    position = 0;    // 初始位置（屏幕最左侧）
17    screen_width = 80; // 屏幕宽度（列数）
18    front_color = 0x0F; // 白色前景
19    back_color = 0x10; // 蓝色背景
20    color_counter = 0;
21
22    // 清除第一行
23    for (int i = 0; i < screen_width; i++) {
24        stdio.print(0, i, ' ', back_color);
25    }
26 }
27
28 void Marquee::update() {
29     // 清除第一行
30     for (int i = 0; i < screen_width; i++) {
31         stdio.print(0, i, ' ', back_color);
32     }
33
34     // 显示跑马灯文本
35     for (int i = 0; i < text_length; i++) {
36         int pos = (position + i) % screen_width;
37         if (pos >= 0 && pos < screen_width) {
38             // 使用彩色前景
39             uint8 color = back_color + ((front_color + i) % 15 + 1);
40             stdio.print(0, pos, text[i], color);
41         }
42     }
43
44     // 更新位置 - 实现从左到右的跑马灯效果
45     position = (position + 1) % screen_width;
46
47     // 定期更改背景色
48     color_counter++;
49     if (color_counter >= 10) {
50         color_counter = 0;
51         back_color += 0x10;
52         if (back_color >= 0x70) {
53             back_color = 0x10;
54         }
55     }
56 }
```

#### 4 我们还需要修改 setup.cpp,添加 marquee 类的初始化

```
1  #include "asm_utils.h"
2  #include "interrupt.h"
3  #include "stdio.h"
4  #include "Marquee.h"
5
6  // 屏幕IO处理器
7  STDIO stdio;
8  // 中断管理器
9  InterruptManager int_manager;
10 // 跑马灯对象
11 Marquee marquee;
12
13 extern "C" void setup_kernel()
14 {
15     // 中断处理部件
16     int_manager.initialize();
17     // 屏幕IO处理部件
18     stdio.initialize();
19     // 初始化跑马灯
20     marquee.initialize();
21     // 设置时钟中断处理函数
22     int_manager.setTimeInterrupt((void *)asm_time_interrupt_handler);
23     // 开启时钟中断
24     int_manager.enableTimeInterrupt();
25     // 开启中断
26     asm_enable_interrupt();
27     asm_halt();
28 }
```

- 实验步骤：按上述编写代码后，进入 build 目录，输入以下指令。

```
make
make run
```

- 实验结果展示：23336179MFQ 跑马灯



## Section 5 实验总结与 debug 心得体会

### (1) 使用 vscode SSH 远程连接出现乱码问题

```
mafq5@mafq5-virtual-machine:~/lab4$ nasm -f elf32 -g asm_func.asm -o asm_func.o
asm_func.asm:1: error: parser: instruction expected
mafq5@mafq5-virtual-machine:~/lab4$ cat -v asm_func.asm
M-oM-;M-?[bits 32]
global function_from_asm
extern function_from_C
extern function_from_CPP

function_from_asm:
    call function_from_C
    call function_from_CPP
    ret
```

解决方法:

#### ① 删除特殊字符

首先, 删除这些特殊字符。可以使用 `sed` 或 `tr` 等工具删除这些字符。以下是一个简单的命令, 删除文件中的非 ASCII 字符:

```
sed -i 's/[^[:print:]]//g' asm_func.asm
```

#### ② 检查文件内容

重新检查文件内容, 确保没有其他特殊字符:

```
cat -v asm_func.asm
```

#### ③ 重新保存文件

如果文件内容仍然有问题, 可以手动重新保存文件。使用命令行编辑器 (如 `nano` 或 `vim`) 重新编辑文件:

```
nano asm_func.asm
```

然后粘贴以下内容:

```
[bits 32]
global function_from_asm
extern function_from_C
extern function_from_CPP

function_from_asm:
    call function_from_C
    call function_from_CPP
    ret
```

保存并退出。

#### ④ 转换文件格式

确保文件格式正确。可以使用 `dos2unix` 转换文件格式:

```
sudo apt install dos2unix
dos2unix asm_func.asm
```

#### ⑤ 确认文件编码

确保文件的编码是 UTF-8 无 BOM。(可在 vsc 设置) 可以使用 `file` 命令检查

文件类型:

```
file asm_func.asm
```

输出应该类似于:

```
asm_func.asm: assembler source, ASCII text
```

(2) 在 Ubuntu 系统中使用 QEMU 虚拟机时, 可能会遇到启动过程中鼠标光标闪烁的问题。这不仅影响用户体验, 还可能让人误以为系统出现故障。本文将深入分析这一问题, 并提供详细的解决方案。

问题现象: 在 Ubuntu 下启动 QEMU 虚拟机时, 鼠标光标在屏幕上会不断闪烁, 无法正常使用。

问题原因: 图形驱动问题: QEMU 虚拟机启动时, 可能会遇到图形驱动不兼容或配置错误的问题, 导致鼠标光标闪烁。

QEMU 配置问题: 虚拟机的配置文件中可能存在导致启动闪烁的错误设置。  
硬件加速问题: 启用硬件加速后, 有时会出现兼容性问题, 导致鼠标光标闪烁。

解决方案

#### ① 1 修改 QEMU 配置文件

首先, 我们需要找到 QEMU 的配置文件。通常情况下, 配置文件位于虚拟机目录下的 `.qemu` 文件夹中。

```
cd /path/to/vm/directory
```

```
nano .qemu/qemu:///system.conf
```

在配置文件中, 找到以下行:

```
vga = std
```

将其修改为:

```
vga = none
```

保存并关闭文件。重新启动虚拟机, 看是否解决问题。

#### ② 更新图形驱动

如果修改配置文件后问题依然存在, 可以尝试更新图形驱动。

```
sudo apt-get update
```

```
sudo apt-get install firmware-linux firmware-linux-free firmware-linux-nonfree
```

重启虚拟机, 查看问题是否解决。

#### ③ 关闭硬件加速

如果启用硬件加速导致闪烁, 可以尝试关闭硬件加速。

在 QEMU 配置文件中, 找到以下行:

```
accel = kvm
```

将其修改为:

```
accel = none
```

保存并关闭文件。重新启动虚拟机, 查看问题是否解决。

#### ④ 修改内核参数

如果以上方法都无法解决问题, 可以尝试修改内核参数。

```
echo "vga=none" | sudo tee /etc/initramfs-tools/initramfs.conf.d/qemu.conf
```

```
sudo update-initramfs -u
```

重启虚拟机, 查看问题是否解决。

(3) 在 task3 中如果先 `make run` 再 `make debug` 就会进入死循环, 无法进行设置断点, 查看 IDT.如图:

```
Machine View
23336179_MFQ_interrupt0-1)
0x00100478

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00

Booting from Hard Disk...

终端

../../src/utls/asm_utils.asm
63     loop .hex_loop
64
> 65     jmp $
66
67     ; void asm_init_page_reg(int *directory);
68 asm_init_page_reg:
69     push ebp
70     mov ebp, esp
71
72     push eax
73
74     mov eax, [ebp + 4 * 2]
75     mov cr3, eax ; 放入页目录表地址

remote Thread 1.1 In: asm_segmentation_fault[digit] L65 PC: 0x20ac8
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x00020ac8 in ?? ()
(gdb) b setup_kernel
Breakpoint 1 at 0x2019e: file ../../src/kernel/setup.cpp, line 17.
(gdb) c
Continuing.
```

会出现无响应的情况（应该是进入了死循环）  
应该直接 make debug.

## Section 6 附录：参考资料清单

1. 部分代码 debug 与指令除了参考实验指导书，还参考了大语言模型
2. 参考文章，博客部分如下：

[中断方式实现跑马灯-阿里云开发者社区](#)

[中断函数实现八个跑马灯 - CSDN 文库](#)

[CC2530———中断方式实现跑马灯-CSDN 博客](#)