



中山大學  
SUN YAT-SEN UNIVERSITY

## 操作系统实验 4

### 混合编程与中断

实验课程: 操作系统原理实验

实验名称: 混合编程与中断

专业名称: 计算机科学与技术

学生姓名: 钟旺烜

学生学号: 23336342

实验地点: 实验楼 B203

实验成绩:

报告时间: 2025 年 4 月 18 日

## Section 1 实验概述

### ●实验任务 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。

### ●实验任务 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。

### ●实验任务 3：中断的处理

请在 `src/8` 的基础上, 仿照 Example 3 编写段错误的中断处理函数, 助教已经 `src/8/src/kernel/setup.cpp` 中实现了对应的段错误触发语句, 只要你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。

额外思考题: 请你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：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, 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。

## Section 2 实验步骤与实验结果

### ----- 实验任务 1 -----

#### ●任务要求:

#### 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。

#### ●思路分析:

实验 1 要求我们复现 Example 1, Example 1 的相关代码已经在实验材料里给出, 将代码复制到虚拟机中进行相关操作即可实现复现。

有关 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, 在实验材料中也有介绍, 将关键

内容整理归纳即可得出。

有关 Makefile 的使用，根据实验材料给出的参考资料地址 <https://c.biancheng.net/makefile/> 中进行学习，掌握使用 Makefile 编译 C/C++ 项目的基本方法和操作。

#### ●实验步骤：

首先我们来复现 Example1。Example1 的代码已经在实验材料里给出，并且提供了 Makefile，但是提供的 Makefile 代码有些许错误的地方，其中进行操作的文件名和实际编写的代码文件名有差错，故需要进行修改，修改后的 Makefile 代码如下所示

```
main.out: main.o c_func.o cpp_func.o asm_utils.o
    g++ -o main.out main.o c_func.o cpp_func.o asm_utils.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_utils.o: asm_utils.asm
    nasm -o asm_utils.o -f elf32 asm_utils.asm

clean:
    rm *.o
```

修改完 Makefile 代码后，我们直接于 terminal 中在 Makefile 对应的文件目录下输入 make 指令：

```
make
```

即可编译生成 main.out 可执行文件。再在 terminal 中输入以下指令

```
./main.out
```

即可运行编译生成好的程序。

以上我们利用实验资料提供的代码并且修改 Makefile 文件快速复现了 Example1。接下来我们结合 Example1 中具体的代码来说明 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。

以下为 Example1 中各个文件中的代码：

#### **c\_func.c:**

```
#include <stdio.h>

void function_from_C() {
    printf("This is a function from C.\n");
}
```

### cpp\_func.cpp:

```
#include <iostream>

extern "C" void function_from_CPP() {
    std::cout << "This is a function from C++." << std::endl;
}
```

### asm\_utils.asm

```
[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
```

### main.cpp

```
#include <iostream>

extern "C" void function_from_asm();
int main() {
    std::cout << "Call function from assembly." << std::endl;
    function_from_asm();
    std::cout << "Done." << std::endl;
}
```

可以看到，在文件 `c_func.c` 中定义 C 函数 `function_from_C`；然后在文件 `cpp_func.cpp` 中定义 C++ 函数 `function_from_CPP`；接着在文件 `asm_utils.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`；最后在文件 `main.cpp` 中调用汇编函数 `function_from_asm`。

我们首先来看如何在汇编代码中调用 C/C++ 的函数。

当我们需要在汇编代码中使用 C 的函数时，我们需要在汇编代码中声明这个函数来自于外部。在函数定义前加上 **extern** 来完成声明，如 **asm\_utils.asm** 在调用 `function_from_C` 和 `function_from_CPP` 时便提前进行了以下声明：

```
extern function_from_C
extern function_from_CPP
声明后便可直接使用：
call function_from_C
call function_from_CPP
```

但是，如果我们需要在汇编代码中使用来自 C++ 的函数 `function_from_CPP` 时，我们需要现在 C++ 代码的函数定义前加上 **extern "C"**。因为 C++ 支持函数重载，为了区别同名的重载函数，C++ 在编译时会进行名字修饰。也就是说，`function_from_CPP` 编译后的标号不再是 `function_from_CPP`，而是要

带上额外的信息。而 C 代码编译后的标号还是原来的函数名。因此，`extern "C"` 目的是告诉编译器按 C 代码的规则编译，不进行名字修饰。

所以我们可以看到 `cpp_func.cpp` 文件中 `function_from_CPP()` 前缀加上了 `extern "C"`：

```
extern "C" void function_from_CPP()
```

如果不加 `extern "C"`，那么我们在汇编代码中声明的标号就不是 `function_from_CPP`，而是 `function_from_CPP` 经过名字修饰后的标号，这将会变得非常麻烦。

接下来看如何在 C/C++ 代码中调用汇编代码中的函数。

在 C/C++ 调用汇编函数之前，我们先需要在汇编代码中将函数声明为 `global`。

在 `asm_utils.asm` 我们看到其中进行了以下声明：

```
global function_from_asm
```

我们在 C/C++ 中用前缀 `extern` 将汇编函数声明来自外部即可，如 `main.cpp` 中所示：

```
extern "C" void function_from_asm();
```

注意在 C++ 中需要声明为 `extern "C"`。

以上便是 C 代码调用汇编函数和汇编代码调用 C 函数的基本语法，但我们知道在很多情况下，我们调用的函数是带有返回值和参数的，下面我们来解释这一部分的语法。

因为 Example1 中有关函数的调用并没有涉及到返回值和参数，故下面的解释会用全新的代码实例来解释：

C/C++ 函数带有参数和返回值的调用规则如下所示：

- 如果函数有参数，那么参数从右向左依次入栈
- 如果函数有返回值，返回值放在 `eax` 中。
- 放置于栈的参数一般使用 `ebp` 来获取

我们以一个例子来说明。我们有两个函数，一个是汇编函数 `function_from_asm`，一个是 C 函数 `function_from_C`，并且两个函数在 C 代码中声明如下：

```
extern int function_from_asm(int arg1, int arg2);
```

```
int function_from_C(int arg1, int arg2);
```

当我们需要在汇编代码中调用函数 `function_from_C`，调用的形式是 `function_from_C(1,2)`，此时的汇编代码如下。

```
push 2      ; arg2
```

```
push 1      ; arg1 (从右到左依次入栈)
```

```
call function_from_C
```

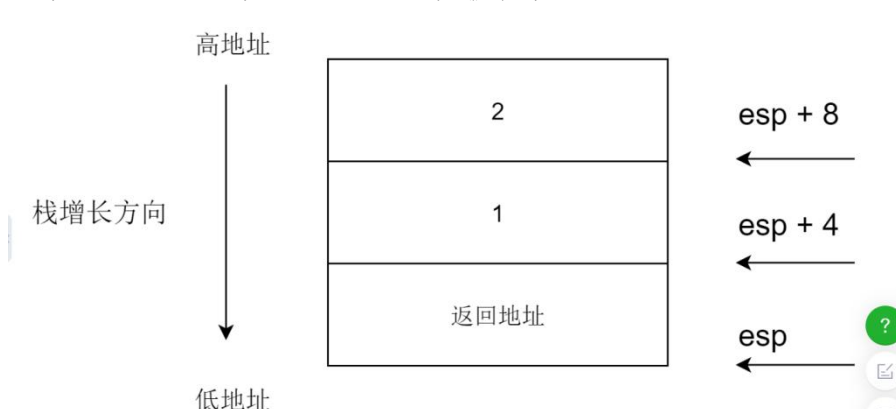
```
add esp, 8   ; 清除栈上的参数
```

`call` 指令返回后，函数的返回值被放在了 `eax` 中。

当我们需要在 C 代码中调用函数 `function_from_asm` 时，使用如下语句即可

```
int ret = function_from_asm(1, 2);
```

函数调用后，参数 2，1，返回值被依次放在栈上，如下所示



此时，我们实现的汇编函数 `function_from_asm` 必须要遵循 C/C++ 的函数调用规则才可以被正常调用。一个遵循了 C/C++ 的函数调用规则的汇编函数如下所示。

```

function_from_asm:
    push ebp
    mov ebp, esp
    ; 下面通过 ebp 引用函数参数
    ; [ebp + 4 * 0]是之前压入的 ebp 值
    ; [ebp + 4 * 1]是返回地址
    ; [ebp + 4 * 2]是 arg1
    ; [ebp + 4 * 3]是 arg2
    ; 返回值需要放在 eax 中
    ...
    pop ebp
    ret

```

特别注意，汇编函数并没有函数参数和返回值的概念，因此汇编函数也被称为过程，不过是一段指令序列而已。函数参数和返回值的概念是我们在 C/C++ 函数调用规则中人为给出规定的，汇编函数并不知情。

上面我们已经解决了有关函数调用的基本问题，下面我们来看如何编写 Makefile:

Makefile 文件包含了一系列的“规则”，每个规则的基本结构如下：

目标(target)...: 依赖(prerequisites)...

<tab>命令(command)

- **目标(target):** 通常是要生成的文件的名称，也可以是执行的动作名称，如“clean”。
- **依赖(prerequisites):** 生成目标所需要的文件或中间过程生成的目标。
- **命令(command):** 通过执行命令对依赖操作生成目标。命令前必须是一个 Tab 字符，不能是空格。

我们根据此规则来看前面我们编写的 Makefile 代码,并作出进一步解释（在 Makefile 中，#代表着注释）：

```

main.out: main.o c_func.o cpp_func.o asm_utils.o
#生成 main.out 文件，依赖于 main.o c_func.o cpp_func.o asm_utils.o 文件
    g++ -o main.out main.o c_func.o cpp_func.o asm_utils.o -m32 #生成文件命令

c_func.o: c_func.c #生成 c_func.o 文件，依赖于 c_func.c 文件
    gcc -o c_func.o -m32 -c c_func.c
    #g++是一个编译 c++语法的编译器，gcc 是一个编译 c 语音语法的编译器

cpp_func.o: cpp_func.cpp #生成 cpp_func.o 文件，依赖于 cpp_func.cpp 文件
    g++ -o cpp_func.o -m32 -c cpp_func.cpp

main.o: main.cpp #生成 main.o 文件，依赖于 main.cpp 文件
    g++ -o main.o -m32 -c main.cpp

asm_utils.o: asm_utils.asm #生成 asm_utils.o 文件，依赖于 asm_utils.asm 文件
    nasm -o asm_utils.o -f elf32 asm_utils.asm

clean:
    rm *.o

```

## Makefile 的工作原理

**1、检查依赖：**在生成目标之前，`make` 会检查规则中的依赖是否存在。如果不存在，则寻找是否有规则用来生成该依赖文件。

**2、检查更新：**如果依赖存在，`make` 会检查依赖的时间戳是否比目标的时间戳新。如果依赖的时间戳更新，则执行命令更新目标。

**3、执行命令：**如果目标需要更新，则按照规则中的命令执行操作。

## Makefile 的常用命令和选项

### **make 命令：**

**功能：**默认执行 Makefile 文件中的第一个目标

例子：我们在复现 Example1 中的实验中，直接输入了 `make` 指令，故默认执行 Makefile 文件中的第一个目标：

```
main.out: main.o c_func.o cpp_func.o asm_utils.o
```

```
g++ -o main.out main.o c_func.o cpp_func.o asm_utils.o -m32 #生成文件命令
```

而根据 Makefile 的工作原理，在生成目标之前，`make` 会检查规则中的依赖是否存在。如果不存在，则寻找是否有规则用来生成该依赖文件（**检查依赖**）。故 Makefile 会找到并依次执行生成 `main.o`；`c_func.o`；`cpp_func.o`；`asm_utils.o` 的命令，进而最后生成 `main.out`。因此，在编写 Makefile 时，最好将要生成的最终目标文件作为 Makefile 文件中的第一个目标。

### **make clean**

**功能：**通常用于清除编译过程中产生的中间文件和最终生成的目标文件。

其他 Makefile 的常用命令和选项在 Example1 中没有使用，故不再进一步赘述。

下面来解释一下 Makefile 中伪目标的设置和使用：

在 Makefile 中，`.PHONY` 是一个特殊的指令，用于声明一些“伪目标”（**phony targets**）。伪目标并不是实际要生成的文件名，而是用来执行一些命令或作为依赖关系的标签。通过使用 `.PHONY`，你可以告诉 `make` 这些目标不应该被当作文件名来处理，即使它们与当前目录中的文件名相同。

### **执行伪目标：**

在命令行中，通过 `make` 命令后跟伪目标名称来执行它。如果 `make` 不带任何参数运行，它通常会尝试构建 Makefile 中的第一个目标（但这取决于 `make` 的实现和 Makefile 的结构，因此使用 `.PHONY` 和明确的默认目标如 `all` 是更好的做法）。

### **tips:**

`all` 是 Makefile 中一个常见的默认目标名称。它通常被用作一个集合目标，依赖于其他需要构建或执行的目标。当你在 Makefile 中定义了 `all` 目标，并为其指定了依赖项时，运行 `make`（不带任何参数）通常会尝试构建 `all` 目标及其所有依赖项。

如在 Example1 中的 Makefile 中，以下代码：

```
clean:
```

```
rm *.o
```

`clean` 便是伪目标：其中的操作为

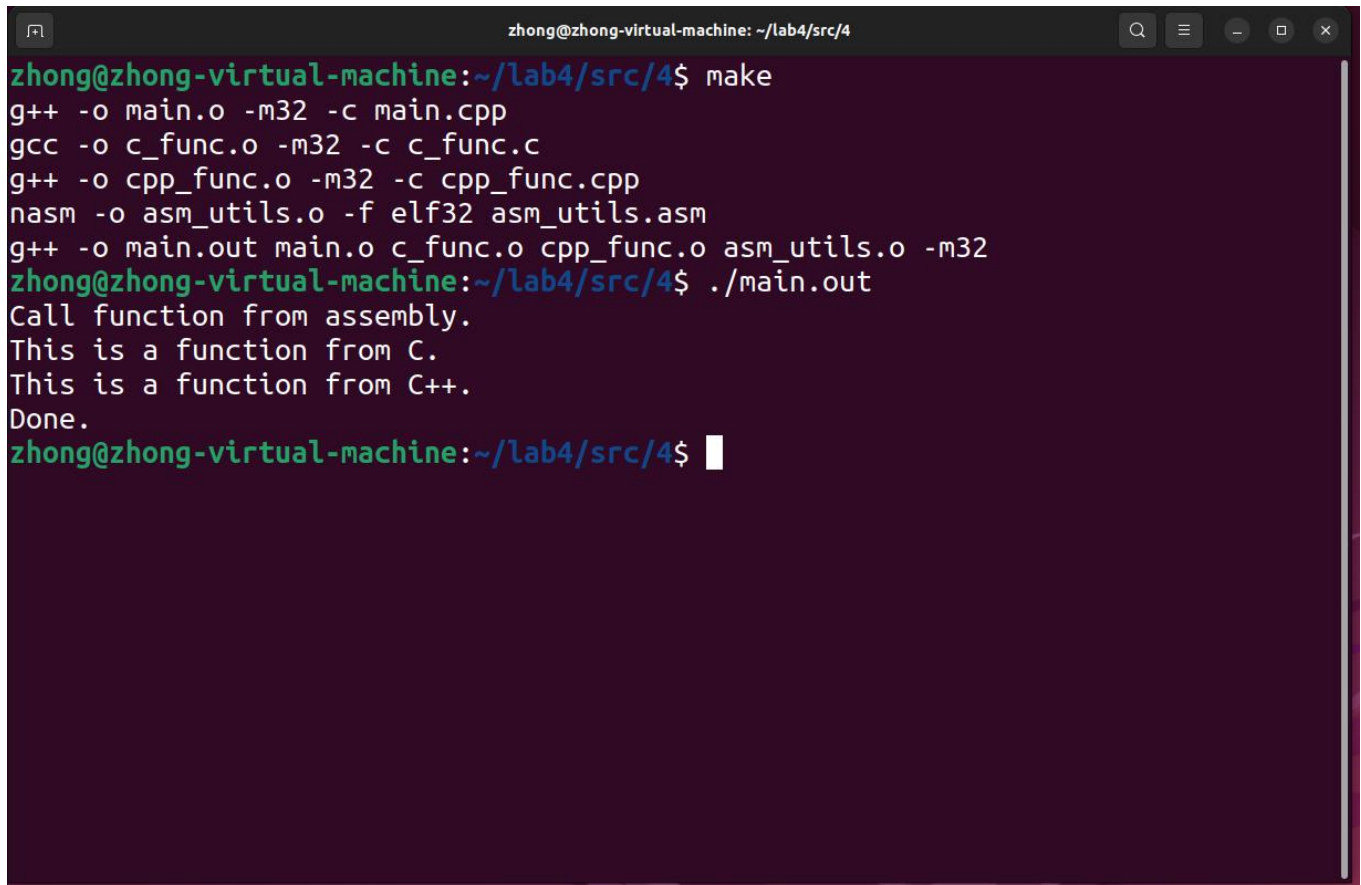
**rm:**这个命令用于删除文件.

**\*.o:**删除当前目录及其子目录下所有扩展名为.o 的文件

其他的 Makefile 规则在 Example1 中并未使用到，到后续实验中若有使用再进行解释。



●实验结果展示：通过执行前述代码，可得下图结果。

A terminal window titled 'zhong@zhong-virtual-machine: ~/lab4/src/4' with standard window controls. The terminal shows the following commands and output:

```
zhong@zhong-virtual-machine:~/lab4/src/4$ 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
zhong@zhong-virtual-machine:~/lab4/src/4$ ./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done.
zhong@zhong-virtual-machine:~/lab4/src/4$
```

按照实验步骤中的内容进行操作后，可以看到 main.out 程序正常编译并运行，打印字符结果如上图所示



## 实验任务 2

### ●任务要求:

### 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。

### ●思路分析:

实验 2 要求复现 Example2, Example2 的相关代码已经在实验资料中给出, 有关内核的加载的相关操作和代码已经在实验材料里详细给出, 在实验报告中不再做重复展示。这里着重解释如何在进入 `setup_kernel` 函数后输出自己的学号。

我们可以在 Example2 提供的代码中观察到在进入 `setup_kernel` 函数后, 调用了汇编函数 `asm_hello_world()`; 进而实现了输出 Hello World。因此, 我们只需要编写一个 `asm_id()` 函数来实现输出学号, 然后在到 `setup_kernel` 函数中调用即可。

### ●实验步骤:

我们首先编写 `asm_id()` 函数, 放置在 `src/utils/asm_utils.h` 下, 编写的代码如下:

```
asm_id:
push eax
xor eax, eax

mov ah, 0x74; 白底红字
mov al, '2'
mov [gs:2 * 0], ax

mov al, '3'
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, '4'
mov [gs:2 * 6], ax

mov al, '2'
```

```
mov [gs:2 * 7], ax
```

```
pop eax
```

```
ret
```

然后我们在在文件 `include/asm_utils.h` 中加入 `asm_id()` 声明的汇编函数，

```
#ifndef ASM_UTILS_H
#define ASM_UTILS_H

extern "C" void asm_hello_world();
extern "C" void asm_id();

#endif
```

随后我们要修改 `setup_kernel` 函数中的内容来调用 `asm_id()` 函数，修改后的代码如下：

```
#include "asm_utils.h"

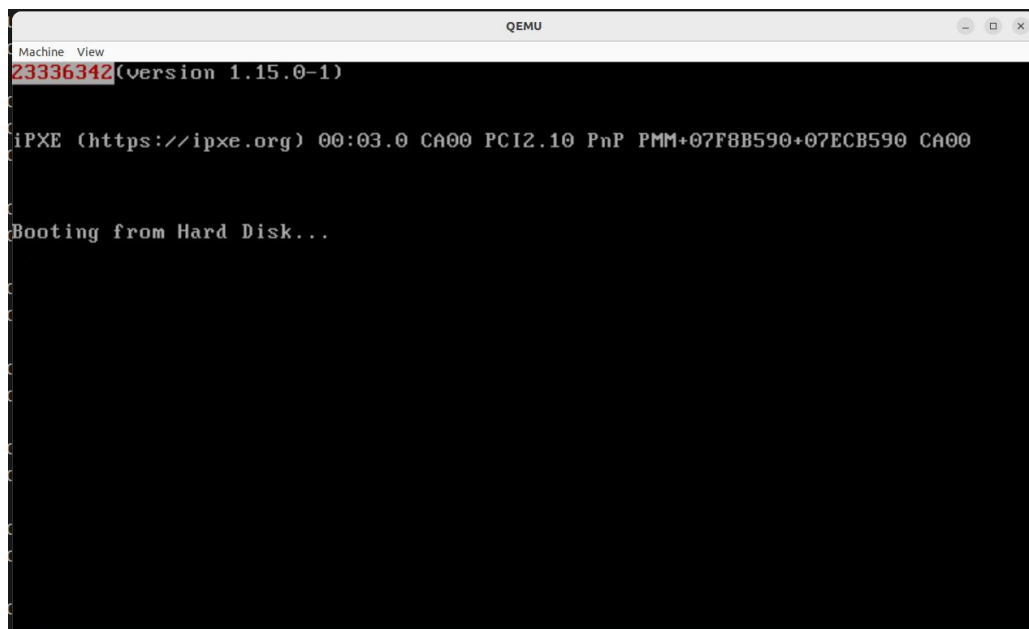
extern "C" void setup_kernel()
{
    asm_id();
    while(1) {

    }
}
```

至此我们成功在原代码文件中添加了函数并调用了他，接下来只要利用 `Makefile` 文件中的命令在 terminal 中于 `../src/build` 文件目录下输入以下指令即可：

```
make build && make run
```

●实验结果展示：通过执行前述代码，可得下图结果。



## 实验任务 3

### ●任务要求:

## Assignment 3 中断的处理

在过去的一个多月里，经常有同学来问助教为什么自己的程序会出现段错误（Segmentation Fault），助教觉得这是一个值得深刻思考的问题，决定继续用段错误折磨大家让大家深入学习段错误的成因，同时让大家稍微提前地学习到内存管理的部分相关知识。

请在 `src/8` 的基础上，仿照Example 3编写段错误的中断处理函数，助教已经 `src/8/src/kernel/setup.cpp` 中实现了对应的段错误触发语句，只要你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。

因为必须引入内存管理之后，才会有段错误（其实是**页面错误**），所以助教引入了一些lab7中的页面管理代码（`memory.cpp`等），同学们可以忽略这部分的代码，暂时不要求掌握。

**额外思考题：**请你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。

### ●思路分析:

实验 3 要求我们仿照 Example3 编写段错误的中断处理函数，根据助教实现的段错误触发语句，我们可以知道其触发的段错误会给出 14 向量号于操作系统，因此我们编写一个显示“Segment fault happened,halt” 的函数，并将其放置到 14 向量号对应的地址即可

### ●实验步骤:

首先我们用汇编语言编写一个 `asm_segment_interrupt` 函数，来处理可能触发的段错误，函数编写在 `asm_utils.asm` 中。代码如下：

```
global asm_segment_interrupt ;使 C 代码可以调用此汇编函数
ASM_SEGMENT_INTERRUPT_INFO db 'Segment interrupt happened, halt...'
                             db 0

; void asm_segment_interrupt()
asm_segment_interrupt:
    cli
    mov esi, ASM_SEGMENT_INTERRUPT_INFO
    xor ebx, ebx
    mov ah, 0x03
.output_information: ; 打印字符串
    cmp byte[esi], 0
    je .end
    mov al, byte[esi]
    mov word[gs:bx], ax
    inc esi
    add ebx, 2
    jmp .output_information
.end:
    jmp $ ; 进入死循环
```

我们编写的 `asm_segment_interrupt` 函数在被调用后, 会在 qemu 屏幕上输出字符串“Segment interrupt happened, halt...” 并随后跳入死循环

添加完函数后, 我们要把新创建的函数定义加入到 `asm_utils.h` 中, 修改后的代码如下:

```
#ifndef ASM_UTILS_H
#define ASM_UTILS_H

#include "os_type.h"

extern "C" void asm_hello_world();
extern "C" void asm_lidt(uint32 start, uint16 limit);
extern "C" void asm_unhandled_interrupt();
extern "C" void asm_segment_interrupt();//新创建的函数的定义
extern "C" void asm_halt();
extern "C" void asm_out_port(uint16 port, uint8 value);
extern "C" void asm_in_port(uint16 port, uint8 *value);
extern "C" void asm_enable_interrupt();
extern "C" int asm_interrupt_status();
extern "C" void asm_disable_interrupt();
extern "C" void asm_init_page_reg(int *directory);

#endif
```

紧接着我们在 `setup.cpp` 文件下的 `setup_kernel` 函数中添加向中断描述符中进行对应注册的代码段。我们可以直接使用 `interruptManager` 中已经实现的 `setInterruptDescriptor` 函数, 来注册对应的中断描述符。与段错误相关的中断向量为 11-14 号, 因此我们向中断描述符的 11-14 中断向量号处都注册我们编写好的处理段错误的中断函数。代码段如下:

```
extern "C" void setup_kernel()
{
    // 中断管理器
    interruptManager.initialize();
    // 注册对应的中断描述符
    for (uint i = 11; i < 15; ++i)
    {
        interruptManager.setInterruptDescriptor(i, (uint32)asm_segment_interrupt, 0);
    }
    // 输出管理器
    stdio.initialize();
    // 内存管理器
    memoryManager.openPageMechanism();
    // 除零错误
    // int t = 1 / 0;
    // 段错误触发
    *(int*)0x100000 = 1;
    asm_halt();
}
```

紧接着在我们加载并运行系统后便能看到系统触发了中断并调用了段错误中断处理函数。

### 额外思考题：

接下来我们来探索触发段错误的方法

#### 1、加载无效的段选择子

尝试将一个无效的段选择子加载到段寄存器（如 DS），触发一般保护故障（GPF）。

步骤如下：

a、选择无效的段选择子：例如，使用未在 GDT 中定义的索引或权限不符的选择子。

b、加载到段寄存器：在汇编中执行 `mov ds, ax`（假设 ax 包含无效选择子）。

#### 2、访问越界内存（段界限限制）

配置段描述符的界限，使访问超出该界限时触发异常。

操作如下：

a、加载段选择子后访问超过其段界限的越界地址

如 `mov eax,[ds:0x100000]`

#### 3、从非执行段执行代码

尝试跳转到标记为不可执行的段。

操作如下：

`mov ax, 0x10` ;数据段选择子（不可执行）

`jmp far eax` ; 跳转到数据段执行代码，触发故障

上面的三种触发段错误的操作使用汇编语言更好实现，因此我们在 `asm_utils.asm` 中新添加一个可以被 C++ 程序调用的 `asm_cause_segment_fault` 函数，在此函数中我们来实现上述三个触发段错误的操作，新添加的函数代码如下所示：

```
global asm_cause_segment_fault

asm_cause_segment_fault:
    ;method 1
    mov ax, 0x1234
    mov ds, ax

    ;method 2
    mov eax,[ds:0x100000]

    ;method 3
    mov ax, 0x10      ; 数据段选择子（假设不可执行）
    jmp far eax       ; 跳转到数据段执行代码，触发故障

    ret
```

同样我们要在 `asm_utils.h` 中添加此函数的定义，以便 C++ 程序调用，添加的定义代码如下：

```
extern "C" void asm_cause_segment_fault();
```

紧接着，我们在 `setup_kernel()` 函数里调用我们编写好的 `asm_cause_segment_fault()` 函数，依次使用 3 个触发段错误的函数，发现均成功触发了段错误并调用了处理段错误的中断处理函数。

结合上面分析内容，总结以下触发段错误的几种方式：

1、访问无效段选择子：加载段寄存器（如 DS、ES）时，选择子指向的段描述符不存在（Present=0）。

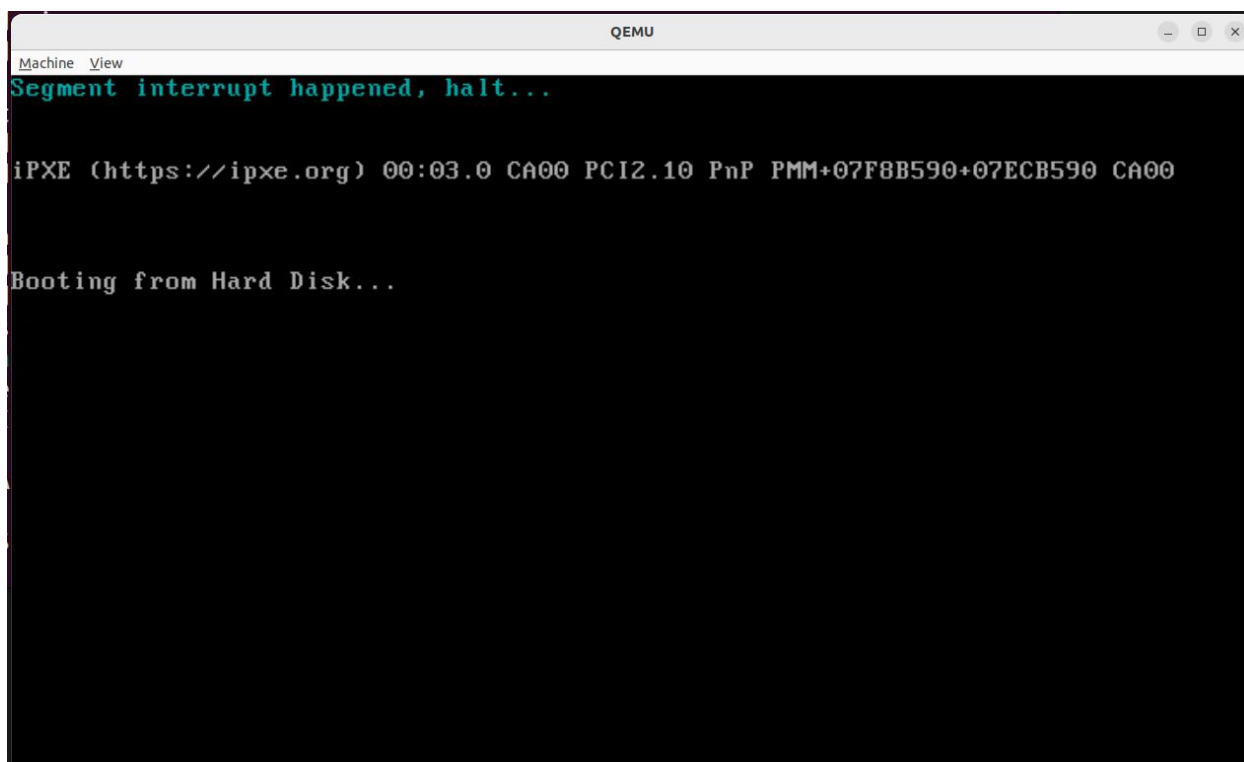
2、越界访问：访问的地址超出段界限（Limit）。

3、从非执行段执行代码，跳转到的数据段代码不可执行。

除了上面 3 种方法外，还有违反段权限（尝试向只读段写入数据）操作可以触发段错误。

- 实验结果展示：通过执行前述代码，可得下图结果。

正确的触发了段错误中断后，调用编写好的段错误中断处理函数后显示的结果如下所示：



```
Machine View
Segment interrupt happened, halt...

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

Booting from Hard Disk...
```

## ----- 实验任务 4 -----

- 任务要求：

### 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，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。

- 思路分析：

我们先来看复现 Example4：因为 Example4 的代码已经在实验资料中给出，所以只要复制到虚拟

机中再利用 Makefile 编译并启动系统即可。

接下来我们利用这个时钟中断的性质，使用 C++ 语言来复刻 lab2 的 assignment4 的字符回旋程序。

我们注意到，处理 8253 的芯片以一定的频率来产生时钟中断的中断函数会以一定的间隔来不停的调用，我们可以把字符回旋程序的进行程序放入这个中断函数中，每次调用都打印一个字符，随着其不断的调用，字符回旋程序便可以不断打印字符而持续运行。因此，我们可以定义一个 Program 类，来封装我们的字符回旋程序并构造相依的函数，首先在 program.h 中对此类进行定义，再到 program.cpp 中对定义的函数进行相应的实现。随后我们便可以再 setup.cpp 中创建一个全局 Program 类实例，并在 setup\_kernel 函数中对其进行初始化，接着我们在处理时间中断的函数中便可调用我们创建好的程序运行函数，使字符回旋程序正常运行。

### ●实验步骤：

首先，我们来复习 Example4,我们从实验资料上复制下代码后，在 terminal 中于 src/8/build 下输入以下指令：

```
make && make run
```

即可在 qemu 屏幕上观察到记录时间中断函数调用次数的字符串。

接下来，我们来实现利用这个时间中断函数来复现 lab2 的 assignment4 的字符回旋程序。

首先我们来编写 program.h，定义我们要使用的 Program 类。代码如下：

```
#ifndef PROGRAM_H
#define PROGRAM_H

#include "os_type.h"

class Program
{
private:
    uint direction;
    uint8 color;
    uint charIndex;
    uint row;
    uint col;
    uint left;
    uint right;
    uint top;
    uint bottom; //定义程序进行需要使用的的基本参数

    void draw(); //实现在 qemu 屏幕上显示相应字符的函数

public:
    Program();
```



```
void initialize(); //初始化函数

void pro_start(); //程序进行函数
};
#endif
```

接着我们在 `program.cpp` 中来实现我们定义的函数。代码如下：

```
#include "interrupt.h"
#include "os_type.h"
#include "os_constant.h"
#include "asm_utils.h"
#include "stdio.h"
#include "program.h"

extern STDIO stdio;

Program::Program() //构造函数模块
{
    initialize();
}

void Program::initialize() //初始化函数
{
    direction = 0;
    color = 0x1f;
    charIndex = 0;
    row = 0;
    col = 0;
    left = 0;
    right = 79;
    top = 0;
    bottom = 24;

    for (int i = 0; i < 25; ++i)
    {
        for(int j = 0; j < 80; ++j)
            stdio.print(i, j, ' ', 0x07);
    }

    char str[] = "23336342-ZWX";

    stdio.moveCursor(995);
    for(int i = 0; str[i]; ++i ) {
        stdio.print(str[i], 0x74);
    }
}

void Program::pro_start() //程序进行程序
{
    if(direction == 0)
```

```

        col++;
        if(col >= right)
        {
            direction = 1;
        }
        draw();
    }
    else if(direction == 1)
    {
        row++;
        if(row >= bottom)
        {
            direction = 2;
        }
        draw();
    }
    else if(direction == 2)
    {
        col--;
        if(col <= left)
        {
            direction = 3;
        }
        draw();
    }
    else
    {
        row--;
        if(row <= top)
        {
            direction = 0;
        }
        draw();
    }
}

```

```

void Program::draw() //打印字符函数
{
    uint8 chars[] = "23336342";
    stdio.print(row,col,chars[charIndex],color);
    color++;
    charIndex++;
    if(charIndex >= 8)
    {
        charIndex = 0;
    }
}

```

下图接着我们修改处理时间中断的函数，使每一次调用函数便进行一次字符回旋程序：

修改后的代码如下：

```
extern Program program;

extern "C" void c_time_interrupt_handler()
{
    program.pro_start();
}
```

可以看到，每次进入此时间中断函数，我们都进行了一次字符的打印，即使字符回旋程序向前进行了一部分。

接下来，我们修改以下 `setup.cpp` 下的代码，将我们创建的 `Program` 类加载入内核中以便运行：

修改后的代码如下：

```
#include "asm_utils.h"
#include "interrupt.h"
#include "stdio.h"
#include "program.h"//

// 屏幕 IO 处理器
STDIO stdio;
// 中断管理器
InterruptManager interruptManager;
// 字符回旋程序
Program program;

extern "C" void setup_kernel()
{
    // 中断处理部件
    interruptManager.initialize();
    // 屏幕 IO 处理部件
    stdio.initialize();
    // 字符回旋程序
    program.initialize();
    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);
    asm_enable_interrupt();
    asm_halt();
}
```

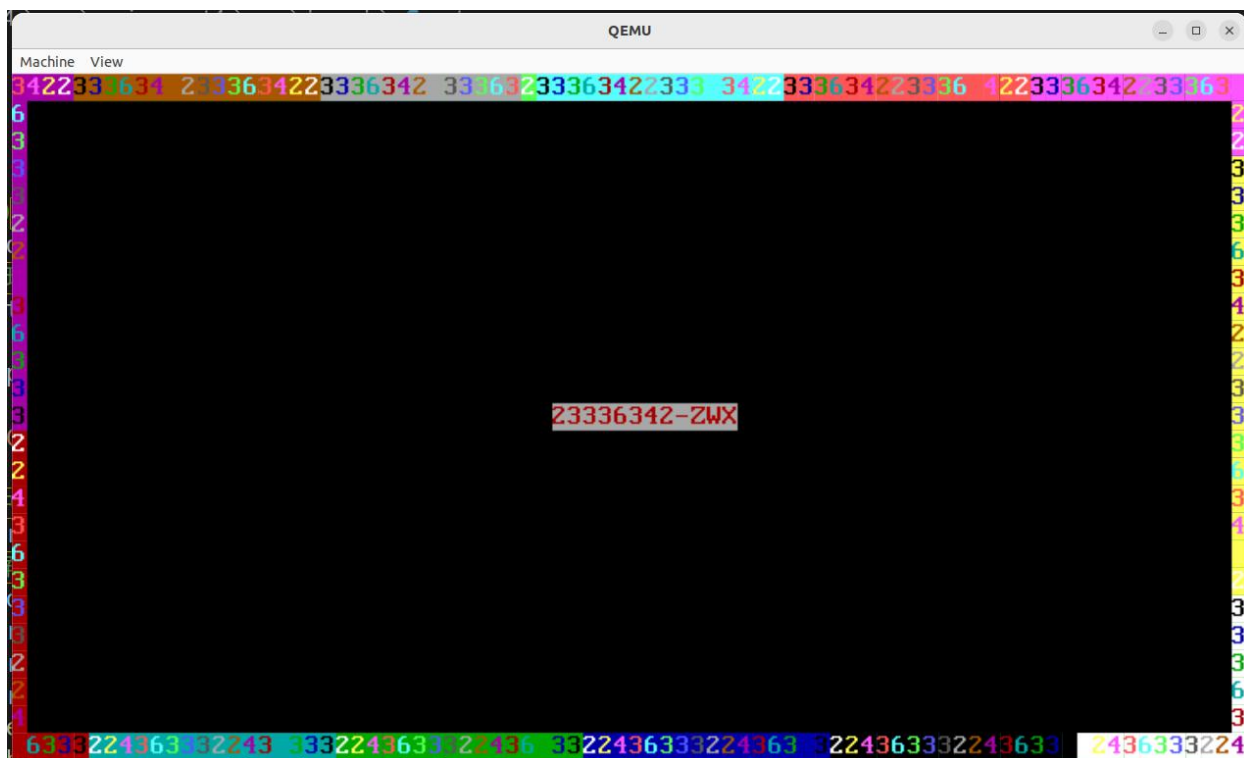
至此我们已经完成了所有的代码修改，使用已经创建的 `Makefile` 文件在 `terminal` 上于对应的文件目录 `lab4/src/assignment4/build` 下输入以下指令：

```
make && make run
```

便可使字符回旋程序正常运行。

●实验结果展示：通过执行前述代码，可得下图结果。

字符回旋程序的运行结果如下所示：



## Section 3 实验总结与心得体会

1、本次实验我深入了解了 C/C++和汇编混合编程的语法，能够正确的编写 C/C++和汇编函数并在不同的文件中进行调用。对汇编函数如何处理 C/C++函数传递的参数有了更加深刻的认识并进行了相关的应用。

2、本次实验使用了 Makefile 编译 C/C++项目，通过阅读大量网上有关 Makefile 编写的相关博客，我了解了 Makefile 编写的基本规则，对其基本运行指令有了更深的理解，特别是对伪目标的构造和使用有了进一步的认识。在本次实验里，对构造的 Makefile 文件我能正确的理解其操作内容并仿造编写对应的 Makefile 文件来简化编译操作。

3、对于系统内核的加载，根据实验材料给出的示例，我对如何编写初步的系统内核有了清晰的认识，对如何系统的管理加载内核的相关代码文件以及如何对其进行分类，进行抽象模块化处理有了清晰的认识并会进行具体的操作与运用。

4、对于保护模式下的中断机制有了进一步的了解，学会了如何通过设置 IDTR 中的内容来设置 IDT 的数量并正确设置 IDT(中断描述符)并编写对应的中断处理函数来对中断描述符对应的中断进行相应的定义和处理。对保护模式下系统对发生的中断的处理过程有了深刻的认识。

5、对什么是段错误以及如何使用不同的方式触发段错误进行了深入学习并有了更清晰的认识

6、学习了如何使用操作系统对硬中断进行相应的处理，根据实验内容，知晓并实现了对 8259A 芯片发出的中断的处理过程。学习了中断程序的编写思路：保护现场、中断处理、恢复现场。对如何处理硬中断有了深刻的认识与理解。

7、学习了如何利用中断的特性来实现各种程序，以更加高效简单的来实现我们想要实现的功能。

## Section 4 对实验的改进建议和意见

1、在对于 assignment3 中断的处理任务中希望给出例子来说明什么是对段错误中断的正常处理函数

2、对于如何在用户模式下触发段错误能给出系统的总结与归纳。

## Section 5 附录：参考资料清单

<https://gitee.com/apshuang/sysu-2025-spring-operating-system/tree/master/lab4>

[https://blog.csdn.net/Nire\\_Yeyu/article/details/106373974](https://blog.csdn.net/Nire_Yeyu/article/details/106373974)

[https://blog.csdn.net/2202\\_75305885/article/details/140375672](https://blog.csdn.net/2202_75305885/article/details/140375672)

[https://blog.csdn.net/qq\\_28453735/article/details/52439137](https://blog.csdn.net/qq_28453735/article/details/52439137)