1 实验概述

- 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。
- 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
- 仿照 Example 3 编写段错误的中断处理函数, 实现段错误的中断处理并正确地在中断描述符中注册。
- 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。

2 实验过程

2.1 实验任务一

2.1.1 混合编程

首先在 c 语言中实现此函数

```
void function_from_C() {
    printf("C function.\n");
}
```

接着在 c++ 中实现此函数, 在这里使用 **extern "C"**, 是因为要告诉编译器按 C 代码的规则编译，不进行名字修饰。因为 c++ 编译时会进行名字修饰，编译后的标号会带上额外的信息。

```
extern "C" void function_from_CPP() {
    std::cout << "C++ function" << std::endl;
}
```

在 asm 的汇编语言文件中调用上述两个函数，并组织成函数供 main.cpp 调用。
global 声明函数为全局，这样它便可以被其它 c/c++ 文件调用。
extern 是声明外部函数，这样便可以在这个 asm 文件中调用外部函数。

```
[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
```

最后在 main.cpp 中调用 asm 中的函数

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

最后是 makefile 文件的编写, 基于 example1 的 makefile 文件，我额外实现了 run 功能，这样 make run 便可直接运行代码。
其中 makefile 文件的格式为:
第一行：目标文件: [列出依赖文件]
第二行：需要执行的命令

```
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

run: main.out
	./main.out
```

因此在 makefile 文件所在目录下执行 make && make run，便可以快速编译执行文件

2.1.2 实验结果展示

```
david@David:~/i386/Lab4/assignment1$ make && make run
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
/usr/bin/ld: warning: asm_utils.o: missing .note.GNU-stack section implies executable stack
/usr/bin/ld: NOTE: This behaviour is deprecated and will be removed in a future version of the linker
./main.out
Call function from assembly.
C function.
C++ function
Done.
```

2.2 实验任务二

2.2.1 进入内核

首先仿照 Lab3 中，在 mbr 中读取 bootloader。接着在 bootloader 中读取内核，其中内核是从第六个扇区开始的 200 个扇区

```
load_kernel:
push eax
push ebx
call asm_read_hard_disk  ; 读取硬盘
```

```
add esp, 8
inc eax
add ebx, 512
loop load_kernel

jmp dword CODE_SELECTOR:KERNEL_START_ADDRESS  ;最后远跳转进入内核
```

在内核开始的代码中，远跳转到我们 c/c++ 实现的函数

```
global enter_kernel
extern setup_kernel
enter_kernel:
    jmp setup_kernel
```

在 c/c++ 中，调用 asm 中实现的在 qemu 显示屏输出学号的函数

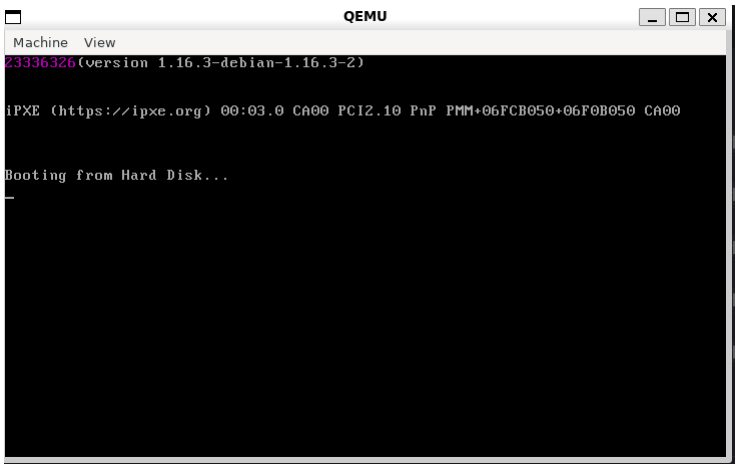
```
extern "C" void setup_kernel()
{
    asm_hello_world();
    while(1) {

    }
}
```

最后在 asm 中实现输出学号的操作

```
[bits 32]

global asm_hello_world

asm_hello_world:
    push eax
    xor eax, eax

    mov ah, 0x05 ;青色
    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, '2'
    mov [gs:2 * 6], ax
    mov al, '6'
    mov [gs:2 * 7], ax
    pop eax
```

```
27      ret
```

2.2.2 实验结果展示



2.3 实验任务三

2.3.1 实现页面错误的中断处理

先在 interrupt.cpp 编写中断处理函数, 清屏 + 在 qemu 输出特定语句

```
extern "C" void c_page_interrupt_handler()
{
    for (int i=0;i<25;i++)
        for (int j=0;j<80;j++)
        stdio.print(i,j,' ',0x07);
    stdio.moveCursor(0);

    char str[]="#PF Page fault";
    for (char*c=str;*c;c++)
    {
        stdio.print(0,c-str,*c,0x03);
    }
}
```

接着在 asm 中编写中断处理函数，调用上述 cpp 中的函数

```
asm_page_fault_handler:
    call c_page_interrupt_handler
jmp $
```

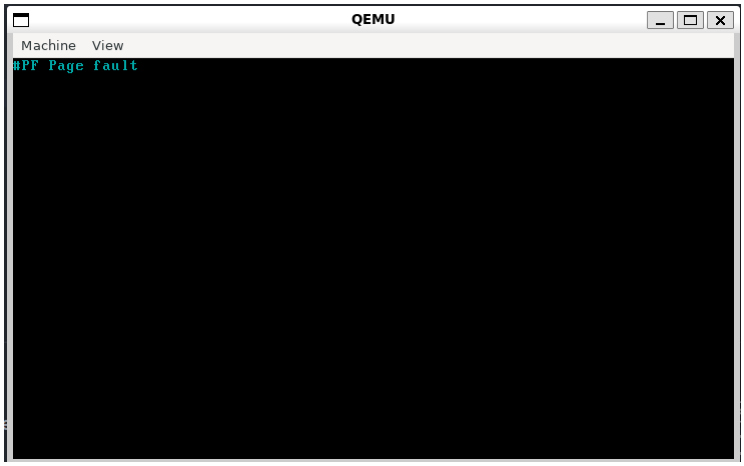
最后在 InterruptManager 类中的初始化函数中注册中断描述符

```
void InterruptManager::initialize()
{
// 初始化IDT
IDT = (uint32 *)IDT_START_ADDRESS;
asm_lidt(IDT_START_ADDRESS, 256 * 8 - 1);
for (uint i = 0; i < 256; ++i)
{
    setInterruptDescriptor(i, (uint32)asm_unhandled_interrupt, 0);
}
setInterruptDescriptor(14,(uint32)asm_page_fault_handler,0);//页面错误的中断向量号14
}
```

最后在 setup_kernel 函数中通过访问未定义的页面触发页面错误

```
*(int*)0x100000 = 1;
```

2.3.2 实验结果



2.3.3 思考题

常见引起段错误的方式:

- 访问未映射的内存地址，如空指针解引用，或者访问 1MB 以上的内存地址 (本实验中)
- 写入只读区域，如试图修改字符串字面量
- 动态内存释放后访问
- 数组访问越界且访问了未映射区域

为什么数组越界未引起段错误 (页面错误)?
对此进行测试，我编写了以下代码

```
int a[10];
a[100]=1;
```

运行后，发现没触发段错误，因此使用 gdb 进行 debug，分别查看了 a 和 a[100] 的地址:

```
      32    //*(int*)0x100000 = 1;
      33    int a[10];
 >    34    a[100]=1;
      35
      36    asm_halt();
      37 }
emote Thread 1.1 (src) In: setup_kernel
   at ../src/kernel/memory.cpp:49
gdb) s
etup_kernel () at ../src/kernel/setup.cpp:34
gdb) x/d &a[100]
x7d5c: 0
gdb) x/d &a
x7bcc: 0
gdb)
```

如图，a 的地址是 0x7bcc,a[100] 的地址是 0x7d5c, 而在 gdb 中看到的这些地址都是虚拟地址，且在 1MB 的虚拟地址以内。因此虽然此数组越界访问，但访问的虚拟地址仍在我们注册的虚拟地址范围之内，能够被成功映射至物理地址，因此不会发生段错误 (页面错误)。

2.4 实验任务四

2.4.1 时钟中断

在加载内核和初始化中断向量符后，在中断管理类中初始化 8259A 芯片。
首先向特定端口发送四个初始化命令字，其中 ICW2 指定了主片和从片的中断向量号。接着发送特定的中断屏蔽信号，但要注意主片的 IRQ2 引脚连接了从片，要保持一直开启状态。

```
void InterruptManager::initialize8259A()
{
    // ICW 1
    asm_out_port(0x20, 0x11);
    asm_out_port(0xa0, 0x11);
    // ICW 2
    IRQ0_8259A_MASTER = 0x20;
    IRQ0_8259A_SLAVE = 0x28;
    asm_out_port(0x21, IRQ0_8259A_MASTER);
    asm_out_port(0xa1, IRQ0_8259A_SLAVE);
    // ICW 3
    asm_out_port(0x21, 4);
    asm_out_port(0xa1, 2);
    // ICW 4
    asm_out_port(0x21, 1);
    asm_out_port(0xa1, 1);

    // OCW 1 屏蔽主片所有中断，但主片的IRQ2需要开启
    asm_out_port(0x21, 0xfb);
    // OCW 1 屏蔽从片所有中断
    asm_out_port(0xa1, 0xff);
}
```

接着是开中断和关中断的逻辑，时钟中断对于 IRQ0 引脚

```
void InterruptManager::enableTimeInterrupt()
{
    uint8 value;
    // 读入主片OCW
    asm_in_port(0x21, &value);
    // 开启主片时钟中断，置0开启
    value = value & 0xfe;
    asm_out_port(0x21, value);
}

void InterruptManager::disableTimeInterrupt()
{
    uint8 value;
    asm_in_port(0x21, &value);
    // 关闭时钟中断，置1关闭
    value = value | 0x01;
    asm_out_port(0x21, value);
}
```

注册中断描述符

```
void InterruptManager::setTimeInterrupt(void *handler)
{
    setInterruptDescriptor(IRQ0_8259A_MASTER, (uint32)handler, 0);
}
```

编写 asm 的中断处理函数，中断处理时要发送 EOI 消息，否则下一次中断不会发生

```
asm_time_interrupt_handler:
pushad

; 发送EOI消息，否则下一次中断不发生
mov al, 0x20
out 0x20, al
out 0xa0, al

call c_time_interrupt_handler

```

```
popad
iret
```

并在 c++ 中实现对应的字符旋转逻辑，这里使用了 c++ 编程来实现

```
extern "C" void c_time_interrupt_handler()
{
    // 清空屏幕
    for (int i = 0; i < 25; ++i)
    {
        for (int j=0;j<80;j++)
            stdio.print(i, j, ' ', 0x07);
    }
    stdio.moveCursor(0);
    ++times;
    char num='0';
    int color=0;
    int row=0,col=0;
    int direct=1;
    while(1) {
        stdio.print(row,col,num,color);
        if (direct==1) {
            col++;
            if (col==80) {
                col=79;
                row++;
                direct=2;
            }
        } else
        if (direct==2) {
            row++;
            if (row==25) {
                row=24;
                col--;
                direct=3;
            }
        } else
        if (direct==3) {
            col--;
            if (col==-1) {
                col=0;
                row--;
                direct=4;
            }
        } else
        if (direct==4) {
            row--;
            if (row==-1) {
                row=0;
                col++;
                direct=1;
            }
        }
        num++;
        if (num>'9') num='0';
        color++;
        if (color==256) {
            color=0;
        }
        for (int i=0;i<5e4;i++);//延时
    }
}
```

2.4.2 实验结果展示

3 实验总结以及心得体会

- C/C++ 与汇编的混合项目编程的文件组织方式:
  - build 目录: 存放 makefile 文件，使用 make 进行运行
  - include 目录: 存放头文件
  - run 目录：存放调试文件和磁盘
  - src 目录：
    - boot 目录: 存放 mbr, 加载磁盘，以及进入内核的程序
    - kernel 目录: 存放内核的建立，中断的实现等程序
    - utils 目录: 存放汇编文件
- makefile 的运用:
  编写 makefile 时，先编写 make 相应指令的实现，接着需要涉及目标文件，对应依赖，以及进行的指令，这样可以方便快捷的进行多文件编程。
- 擅用 gdb 进行调试: 在遇到段错误触发等问题时，要擅于利用调试查看对应地址，从而快速定位问题。