



# 本科生实验报告

实验课程:\_\_\_\_\_操作系统原理实验\_\_\_\_\_

实验名称:\_\_\_\_\_实验 4 中断\_\_\_\_\_

专业名称:\_\_\_\_\_计算机科学与技术(超级计算方向) \_\_\_\_\_

学生姓名:\_\_\_\_\_林天皓\_\_\_\_\_

学生学号:\_\_\_\_\_18324034\_\_\_\_\_

实验地点:\_\_\_\_\_

实验成绩:\_\_\_\_\_

报告时间:\_\_\_\_\_2021. 4. 14\_\_\_\_\_

# 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。

## 1. 汇编代码中 `global` 和 `extern` 的作用

`global` 和 `extern` 在 `nasm` 汇编语言中都是 `Assembler Directives`。使用 `extern` 关键字可以声明一个未在当前模块中的符号定义，但是假定在其他模块中定义。当一个模块使用 `extern` 引用别的模块时，被引用模块的该符号必须被声明为 `global` 来防止链接错误。

我们可以通过 `objdump` 查看 `global` 关键字对符号表的改变，首先测试未添加 `global` 生成的 `obj` 文件

```
lthos lab4/assignment1 =Σ((( つ^w^)) objdump asm_func.o -x

asm_func.o:      file format elf32-i386
asm_func.o
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
  0 .text          0000000b  00000000  00000000  00000130  2**4
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 asm_func.asm
00000000 l    d  .text  00000000 .text
00000000 l    .text  00000000 function_from_asm
00000000 *UND*  00000000 function_from_C
00000000 *UND*  00000000 function_from_CPP

RELOCATION RECORDS FOR [.text]:
OFFSET  TYPE           VALUE
00000001 R_386_PC32      function_from_C
00000006 R_386_PC32      function_from_CPP
```

图 1-未添加 `global` 关键字的符号表

然后测试添加 `global` 关键字的 `obj` 文件

```
lthos lab4/assignment1 =Σ((( つ^w^ )  objdump asm_func.o -x

asm_func.o:      file format elf32-i386
asm_func.o
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
 0 .text          0000000b  00000000  00000000  00000130  2**4
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE

SYMBOL TABLE:
00000000 l    df *ABS*  00000000  asm_func.asm
00000000 l    d  .text  00000000  .text
00000000      *UND*  00000000  function_from_C
00000000      *UND*  00000000  function_from_CPP
00000000 g      .text  00000000  function_from_asm
```

图 2-添加 global 关键字的符号表

添加 global 符号后，符号表中有 g 描述，即为 global 符号，在可以在其他文件中被链接。

## 2.c++中的 extern “C”

extern “C”使得 cpp 函数能够被 c 语言链接，首先去除 cpp\_func 中的 extern “C”，执行 make 会编译成功但是不能成功链接。

```
lthos lab4/assignment1 =Σ((( つ^w^ )  make run
g++ -o cpp_func.o -m32 -c cpp_func.cpp
g++ -o main.out main.o c_func.o cpp_func.o asm_func.o -m32
asm_func.o: In function `function_from_asm':
asm_func.asm:(.text+0x6): undefined reference to `function_from_CPP'
collect2: error: ld returned 1 exit status
Makefile:2: recipe for target 'main.out' failed
make: *** [main.out] Error 1
```

图 3-未添加 extern “C”关键字链接出现 undefined reference

下面对比添加 extern “C”关键字前后生成 obj 的变化

添加过 extern “C”关键字生成的 obj 文件的符号表中可见具有 function\_from\_CPP 的 global 符号，按照前面的学习，这是可以运行的。下面去除 extern “C”后再测试

```

00000000 l d .note.GNU-stack 00000000 .note.GNU-stack
00000000 l d .eh_frame 00000000 .eh_frame
00000000 l d .comment 00000000 .comment
00000000 l d .group 00000000 .group
00000000 l d .group 00000000 .group
00000000 g F .text 00000046 function_from_CPP
00000000 g F .text. __x86.get_pc_thunk.bx 00000000 .hidden __x86.get_pc_thunk.bx
00000000 *UND* 00000000 _GLOBAL_OFFSET_TABLE_
00000000 *UND* 00000000 _ZSt4cout
00000000 *UND* 00000000 _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_P
00000000 *UND* 00000000 _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0
00000000 *UND* 00000000 _ZNSoIsEPFRSoS_E
00000000 *UND* 00000000 _ZNSt8ios_base4InitC1Ev

```

图 4-添加 extern “C”后符号表中发现该全局符号

可见这时候的符号表只有 `_Z17function_from_CPPv`，这就是出现 undefined reference to 'function\_from\_CPP' 的原因，如果将汇编语言中的 `function_from_CPP` 替换为 `_Z17function_from_CPP` 也能编译执行成功。

```

00000000 l d .eh_frame 00000000 .eh_frame
00000000 l d .comment 00000000 .comment
00000000 l d .group 00000000 .group
00000000 l d .group 00000000 .group
00000000 g F .text 00000046 _Z17function_from_CPPv
00000000 g F .text. __x86.get_pc_thunk.bx 00000000 .hidden __x86.g
00000000 *UND* 00000000 _GLOBAL_OFFSET_TABLE_
00000000 *UND* 00000000 _ZSt4cout
00000000 *UND* 00000000 _ZStlsISt11char_traitsIcEERSt13basic_os
00000000 *UND* 00000000 _ZSt4endlIcSt11char_traitsIcEERSt13basi
00000000 *UND* 00000000 _ZNSoIsEPFRSoS_E

```

图 5-去除 extern “C”后符号表中发现全局符号被改名

这就涉及了 c++ 语言编译过程中的函数名重整技术

### 3.c++的函数名重整

为什么我们定义的名字出现了变化，这是由于 c++ 语言的函数名重整特性，为了实现 c++ 中重载与命名空间等特性，c++ 在编译时候会将函数名改名，因此添加 extern “C” 可以避免编译器对该函数的 Name mangling，使得 c++ 语言的代码可以和 c 语言，fortran 或者 asm 语言进行链接。

#### 4.将c++语言中的函数添加参数，并在汇编中调用

```
#include <iostream>
extern "C" void function_from_CPP(int a,char b[]) {
    std::cout << "This is a function from C++." << std::endl;
    std::cout << "id: "<< a <<" name: "<< b << std::endl;
}
```

将 function\_from\_CPP 添加一个 int 参数和字符串参数。

根据 x86 函数调用规则，需要将参数按照从低到高放入栈空间。

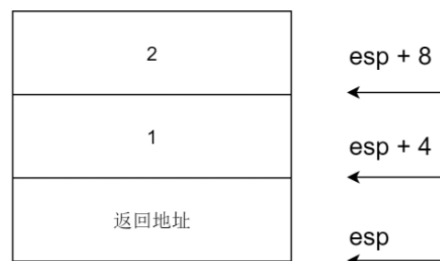


图 6- x86 函数调用规则

对应的，我们修改汇编，在汇编中传递参数，先 sub esp,8 申请两个字节栈空间，然后按照参数方式填入学号和名称。调用 c++语言的函数。调用结束后再 add esp,8 释放栈空间。

```
[bits 32]
global function_from_asm
extern function_from_C
extern function_from_CPP
extern function_from_Rust
function_from_asm:
    call function_from_C
    sub esp,8
    mov eax,18324034
    mov [esp + 4 * 0],eax
    mov eax,name
    mov [esp + 4 * 1],eax
    call function_from_CPP
    add esp,8
    call function_from_Rust
    ret
name db 'Avarpow'
db 0
```

运行结果如下，可以输出学号和名称

```
lthos lab4/assignment1 =Σ((( つ^ω^)) make run
g++ -o cpp_func.o -m32 -c cpp_func.cpp
g++ -o main.out main.o c_func.o cpp_func.o asm_func.o rs_func.a -m32 -lpthread -ldl
./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
id: 18324034 name: Avarpow
This is a function from Rust.
Done.
```

图 7- 输出学号与名称

## 5.汇编与 rust 语言混合编程

创建 rust 文件 rs\_func.rs，内容如下。其中使用#[no\_mangle]作用类似 c++中的 extern “C”，防止编译器的函数名重整。

```
#[allow(non_snake_case)]
#[no_mangle]
fn function_from_Rust(){
    println!("This is a function from Rust.");
}
```

编译为 rust 静态链接库

```
rustc rs_func.rs --target=i686-unknown-linux-gnu --crate-
type=staticlib -o rs_func.a
```

在 gcc 生成 out 文件时候需加上-lpthread 和-ldl 参数链接 pthread 和 libdl.so。

```
lthos lab4/assignment1 =Σ((( つ^ω^)) 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_func.o -f elf32 asm_func.asm
rustc rs_func.rs --target=i686-unknown-linux-gnu --crate-type=staticlib -o rs_func.a
g++ -o main.out main.o c_func.o cpp_func.o asm_func.o rs_func.a -m32 -lpthread -ldl
./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
id: 18324034 name: Avarpow
This is a function from Rust.
Done.
```

图 8- 输出 rust 中的 println 语句

如图成功输出 This is a function from Rust.

综上，该实验中完成了使用 C 语言，C++语言，Rust 语言和汇编语言的混合编程，以及在汇编中传递参数给 c++语言的函数。

## 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。

Example2 中代码的执行包含以下几个部分。1.在 mbr 中循环读取 5 个扇区，进入 bootloader。2.在 bootloader 中开启保护模式，并读取 200 个扇区，跳转进入 entry。3.在 entry 中执行汇编函数输出 helloworld。相关知识均已在 lab3 中展示过，此处不再赘述。

直接运行 `make build && make run` 复现 example2



```
lthos assignment2/build =I((( ㄿ^w^ㄿ 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
operations on block 0 will be restricted.
      Specify the 'raw' format explicitly to remove the restrictions.
[
QEMU
Hello Worldrsion rel-1.13.0-0-gf21b5a4aeb02-prebuilt.qemu.org)
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F909D0+07EF09D0 CA00
Booting from Hard Disk...
```

图 9- 复现 example2

输出为 Hello world。

下面我们将输出内容输出为学号。由于在 assignment2 中我们学习了函数调用的做法，我们这里使用在 C++ 中定义函数，汇编中实现的方法实现输出学号。

先修改 `seup.cpp` 文件为输出字符串函数，参数为字符串首地址与输出颜色。

```
#include "asm_utils.h"
char s[]="18324034 Avarpow";
char color=0x4e;
extern "C" void setup_kernel()
{
    asm_print_string(s,color);
    while(1) {}
}
```

```
}
```

然后修改 `ams_utils.asm` 文件，添加 `asm_print_string` 函数。首先将字符串的首地址写入 `ecx` 寄存器，然后逐个字节读取 `ecx` 寄存器所表示内存中的值，直到为 0 跳出循环。

```
global asm_print_string

asm_print_string:
    ;mov ah, 0x4e ;
    mov ecx,[esp + 4 * 1]
    mov ah,[esp + 4 * 2]
    mov ebx,0
.put:
    mov al,byte[ecx]
    cmp al,0
    je .end
    mov [gs:2 * ebx], ax
    add ecx,1
    inc ebx
    jmp .put
.end:
    ret
```

执行 `make build && make run`

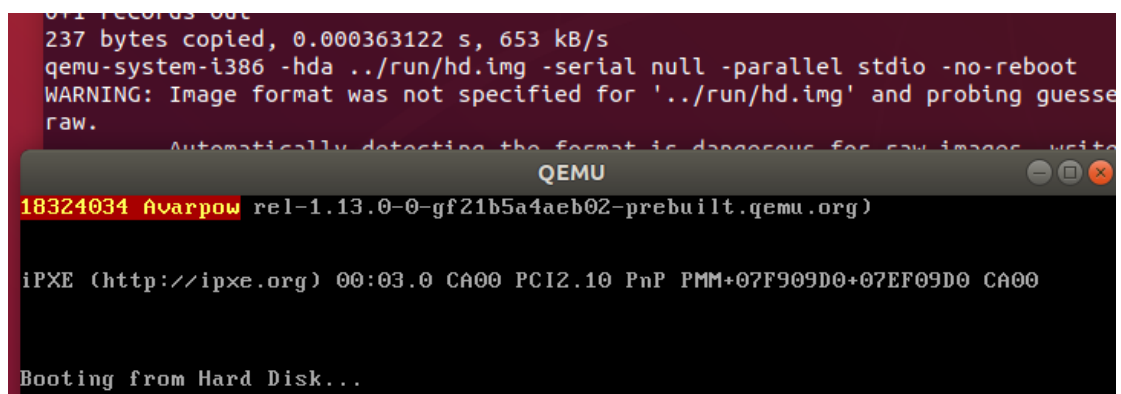


图 10- 保护模式中调用 `c++` 函数输出学号

综上，该实验完成了复现 `example2`，并通过在 `c++` 中调用汇编实现的带参函数完成学号的显示。



## Assignment 3 中断的处理

复现 **Example 3**，你可以更改 **Example** 中默认的中断处理函数为你编写的函数，然后触发之，结果截图并说说你是怎么做的。

Example3 中的程序除了 assignment2 中初始化系统部分之外，还包含以下几个步骤，

1.首先通过 `lidt` 指令初始化 IDTR，这部分被封装在 `asm_lidt` 函数中，通过指定中断描述表基地址和表界限，初始化表空间。

C++调用：

```
asm_lidt(IDT_START_ADDRESS, 256 * 8 - 1);
```

汇编定义：

```
; void asm_lidt(uint32 start, uint16 limit)
asm_lidt:
    push ebp
    mov ebp, esp
    push eax
    mov eax, [ebp + 4 * 3]
    mov [ASM_IDTR], ax
    mov eax, [ebp + 4 * 2]
    mov [ASM_IDTR + 2], eax
    lidt [ASM_IDTR]
    pop eax
    pop ebp
    ret
```

2.然后使用循环，将默认中断处理函数 `setInterruptDescriptor` 写入所有的中断描述表中。

```
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);
    }
}
```

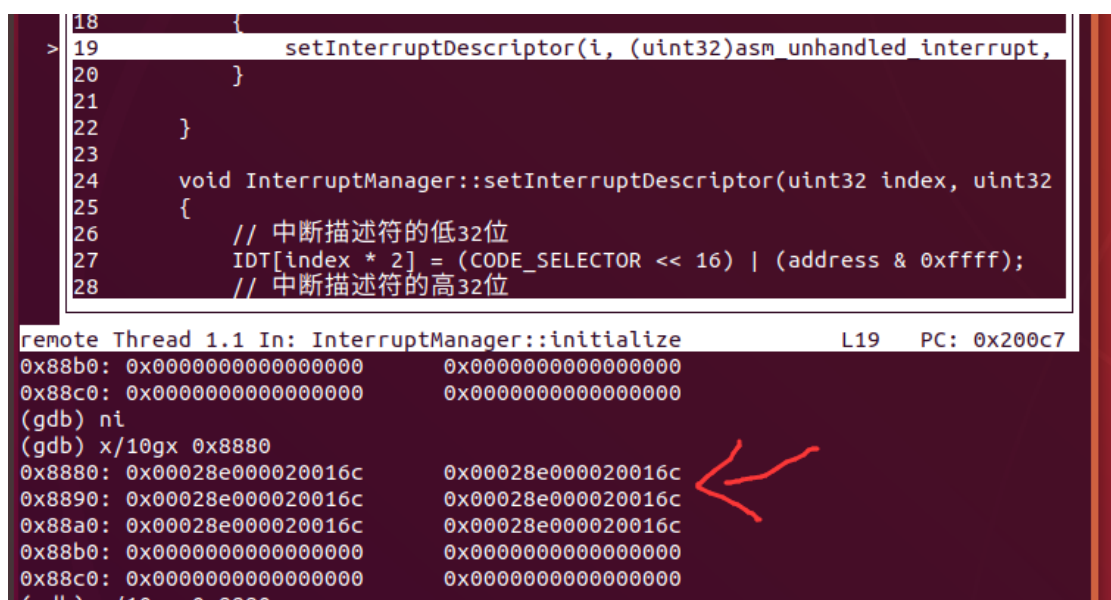
```

    }
    setInterruptDescriptor(3, (uint32)asm_breakpoint_interrupt, 0);
    setInterruptDescriptor(0x32, (uint32)asm_int_32_interrupt_warpper,
0);
}
void InterruptManager::setInterruptDescriptor(uint32 index, uint32 address, byte DPL)
{
    // 中断描述符的低 32 位
    IDT[index * 2] = (CODE_SELECTOR << 16) | (address & 0xffff);
    // 中断描述符的高 32 位
    IDT[index * 2 + 1] = (address & 0xffff0000) | (0x1 << 15) | (DPL <<
13) | (0xe << 8);
}

```

通过 make build && make debug 开始 debug 模式运行

在 InterruptManager::initialize 设置断点，并通过 ni 执行接下来几条指令，然后通过 x/10gx 0x8880 查看 IDT 表中的值



```

18      {
19          setInterruptDescriptor(i, (uint32)asm_unhandled_interrupt,
20      )
21      }
22  }
23
24  void InterruptManager::setInterruptDescriptor(uint32 index, uint32
25  {
26      // 中断描述符的低32位
27      IDT[index * 2] = (CODE_SELECTOR << 16) | (address & 0xffff);
28      // 中断描述符的高32位

```

remote Thread 1.1 In: InterruptManager::initialize L19 PC: 0x200c7

0x88b0:	0x0000000000000000	0x0000000000000000
0x88c0:	0x0000000000000000	0x0000000000000000
(gdb) ni		
(gdb) x/10gx 0x8880		
0x8880:	0x00028e000020016c	0x00028e000020016c
0x8890:	0x00028e000020016c	0x00028e000020016c
0x88a0:	0x00028e000020016c	0x00028e000020016c
0x88b0:	0x0000000000000000	0x0000000000000000
0x88c0:	0x0000000000000000	0x0000000000000000
(gdb) x/10gx 0x8880		

图 11- 中断向量被写入中断向量描述符表

可见已经有一些中断描述符被写入内存地址 0x8880

接下来通过编写 C++代码尝试引起中断，在 setup.cpp 中有一条除以零的语句，会触发中断处理向量号 0 的处理函数；。

```
// 尝试触发除 0 错误
int a = 1 / 0;
```

debug 运行

```
(gdb) layout next
(gdb) break setup_kernel
Breakpoint 1 at 0x20005: file ../src/kernel/setup.cpp, line 8.
(gdb) c
Continuing.

Breakpoint 1, setup_kernel () at ../src/kernel/setup.cpp:8
(gdb) ni
(gdb) si
(gdb) display $pc
1: $pc = (void (*)(void)) 0x20020 <setup_kernel()+27>
(gdb) si
1: $pc = (void (*)(void)) 0x20025 <setup_kernel()+32>
(gdb) so
source command requires file name of file to source.
(gdb) si
1: $pc = (void (*)(void)) 0x20026 <setup_kernel()+33>
(gdb) si
0x0002016d in asm_unhandled_interrupt ()
1: $pc = (void (*)(void)) 0x2016d <asm_unhandled_interrupt+1>
(gdb) si
0x00020172 in asm_unhandled_interrupt ()
1: $pc = (void (*)(void)) 0x20172 <asm_unhandled_interrupt+6>
(gdb) □
```

图 12- debug 执行过程，跳转进去中断处理函数

```
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restriction
□
QEMU
Unhandled interrupt happened, halt...4aeb02-prebuilt.qemu.org)
```

图 13- 在中断处理函数中输出字符串

在执行到除以零的指令之后，进入了中断处理函数 `asm_unhandled_interrupt` 函数。屏幕输出了 `unhandled interrupt happened, halt.` 完成了对 `example3` 的复现。

## 2. 添加自己的中断处理函数，并触发中断

下面我们写一个自己的中断处理函数，例如断点中断（`int 3`），再设置一个用户自定义中断 `0x32`。

首先修改 `setup.cpp`，添加一个 `int 3` 指令触断点中断

```
extern "C" void setup_kernel()
{
    // 中断处理部件
```

```

    interruptManager.initialize();
    // 尝试触发 breakpoint 错误
    __asm__("int $0x03");
    __asm__("int $0x32");
    // 尝试触发除 0 错误
    int a = 1 / 0;
    asm_halt();
}

```

然后改动默认中断向量初始化函数，添加两行将 `asm_breakpoint_interrupt` 和 `asm_int_32_interrupt_warpper` 写入 IDT 的对应位置。

```

extern "C" void asm_breakpoint_interrupt();
extern "C" void asm_int_32_interrupt_warpper();
extern "C" void asm_print_string(char *,char);
InterruptManager::InterruptManager()
{
    initialize();
}
extern "C" void int_32_handle(char color){
    asm_print_string("int32 interrupt handle",0x4e);
}

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(3, (uint32)asm_breakpoint_interrupt, 0);
    setInterruptDescriptor(0x32, (uint32)asm_int_32_interrupt_warpper,
0);
}

```

在 `asm_utils.asm` 中创建处理函数，最后使用 `iret` 返回

```

global asm_hello_world
global asm_lidt
global asm_unhandled_interrupt
global asm_breakpoint_interrupt

```

```

global asm_halt
global asm_print_string
global asm_int_32_interrupt_warpper
extern int_32_handle

ASM_UNHANDLED_INTERRUPT_INFO db 'Unhandled interrupt happened, halt...'
                                db 0

ASM_IDTR dw 0
          dd 0

asm_print_string:
    ;mov ah, 0x4e ;
    mov ecx,[esp + 4 * 1]
    mov ah,[esp + 4 * 2]
    mov ebx,160
.put:
    mov al,byte[ecx]
    cmp al,0
    je .end
    mov [gs:2 * ebx], ax
    add ecx,1
    inc ebx
    jmp .put
.end:
    ret

asm_int_32_interrupt_warpper:
    call int_32_handle
    iret

; void asm_unhandled_interrupt()
asm_unhandled_interrupt:
    cli
    mov esi, ASM_UNHANDLED_INTERRUPT_INFO
    mov ebx,160
    mov ah, 0x03
.output_information:
    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 $

ASM_BREAKPOINT_INFO db 'Breakpoint interrupt happened, halt...'
                    db 0
; void asm_breakpoint_interrupt()
asm_breakpoint_interrupt:
    cli
    mov esi, ASM_BREAKPOINT_INFO
    xor ebx, ebx
    mov ah, 0x03
.output_information:
    cmp byte[esi], 0
    je .end
    mov al, byte[esi]
    mov word[gs:bx], ax
    inc esi
    add ebx, 2
    jmp .output_information
.end:
    iret

```

为什么这里要通过一个 `asm_int_32_interrupt_warpper`，而不能直接使用 `c++` 中的函数？中断函数要求使用 `iret` 指令返回，而 `c++` 中的普通函数只会使用 `ret` 返回，因此需要一个 `warpper` 套一层完成这个转换。

`make build && make run` 运行

The screenshot shows a terminal window with the following content:

```

1+1 records in
1+1 records out
1020 bytes (1.0 kB) copied, 0.000150676 s, 6.8 MB/s
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
Breakpoint interrupt happened, halt...aeb02-prebuilt.qemu.org)
Unhandled interrupt happened, halt...
int32 interrupt handle
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F909D0+07EF09D0 CA00

Booting from Hard Disk...

```

图 14- 分别输出汇编定义中断，`c++` 定义中断，自定义中断

按照预期触发了三种中断，分别是中断 0，中断 3，中断 32，可见都根据我们实现的中断处理函数执行正确。

综上，该实验完成了对 example3 的复现并实现了自定义的中断处理函数。

## 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，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。（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。）

在复现 example4 的过程中，我们通过 8259A 芯片的通信来进一步完成对外设（此处为 8253 可编程时钟芯片）的中断。设置 8259A 的流程包括以下步骤。

1.通过向主片和从片端口按顺序发送 4 次初始化命令字 ICW，并通过设置命令字设置优先级与屏蔽。相应代码如下。

```
// 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);
```

2.将中断向量函数写入 IDT

```
setInterruptDescriptor(IRQ0_8259A_MASTER, (uint32)handler, 0);
```

3.通过 sti 命令开始中断。

```
asm_enable_interrupt:  
    sti  
    ret
```

通过 make build && make run 运行测试

```
raw.  
    Automatically detecting the format is dangerous for raw im  
perations on block 0 will be restricted.  
    Specify the 'raw' format explicitly to remove the restrict  
[  
QEMU  
interrupt happend: 000000017_  
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F909D0+07EF09D
```

图 15- 时钟中断，显示中断发生次数

输入随着时间增长，完成对 example4 的复现。

## 2.实现自己输出学号名称跑马灯。

将 interrupt.cpp 中的时钟中断处理函数替换为输出学号与名称的函数如下。

```
extern "C" void c_time_interrupt_handler()  
{  
    ++times;  
    times=times%80;  
    char str[20] = "18324034 Avarpow";  
    char buffer[80]={0};  
    uint8 color=0x4e;  
    int temp = times;  
    for(int i = 0; i < 16; ++i ) {  
        buffer[(times+i)%80]=str[i];  
    }  
    stdio.moveCursor(0);  
    for(int i = 0; i<80 ; ++i ) {  
        stdio.print(buffer[i],color);  
    }  
}
```

运行效果如下



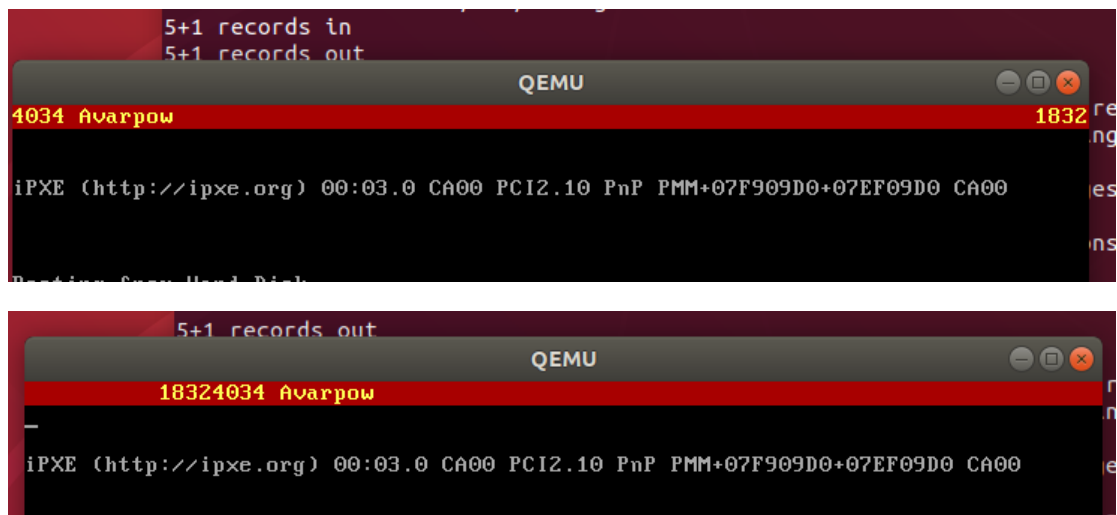


图 15- 跑马灯显示学号与名称

实现了首行中显示学号跑马灯效果。但是速度移动速度稍快，下面我们通过设置 8253 芯片调整时钟中断速度。

### 3.调整 8253 芯片中断频率。

通过向 0x40 端口写入值可以调整 8253 芯片的重载值，进而控制时钟中断的频率。在 initialize8259A 中添加如下代码，其中 count 越小则时钟频率越低，运行后跑马灯的速度降低到了一秒钟移动两格。

```
int count=0x0008;
asm_out_port(0x40,count&0xFF);    // Low byte
asm_out_port(0x40,(count&0xFF00)>>8);  // High byte
```

运行结果：

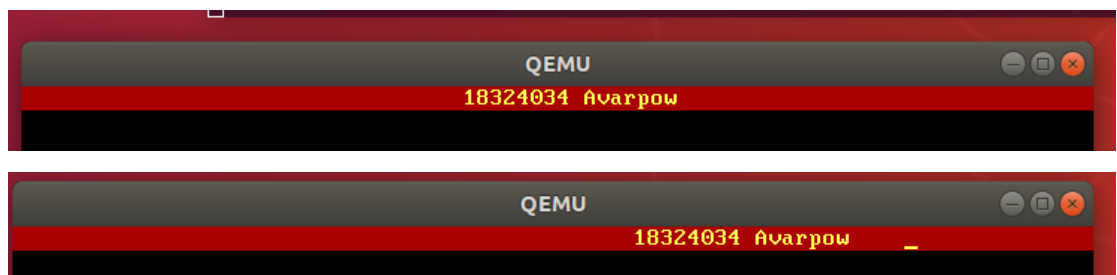


图 15- 降低速度的跑马灯显示学号与名称

综上，该实验完成了设置 8259A 与设置 8253 芯片的频率，并通过时钟中断函数完成对学号名称的跑马灯显示。

## 实验感想

感谢助教非常详细的实验指导，本次实验除了 rust 部分之外没有遇到什么问题地顺利完成了。在本次实验中第一次使用 c/c++ 语言，rust 与汇编语言进行混合编程，完成了汇编语言与 c/c++ 语言函数的互相调用，通过这种方式，可以用我们所熟悉的 C/C++ 语言编写操作系统。接下来通过了解 x86 函数调用约定，在汇编语言中填写参数执行 C++ 语言所定义的函数。

在对 c++ 关键字 `extern "C"` 的理解中，学习了 C++ 其中函数名重整的技术，通过这种技术可以实现命名空间，函数重载等特性，但是也失去了与直接与汇编混合编程的特点。

然后学习了 x86 中中断的处理，通过向中断描述符表中写入中断函数的入口地址，我们可以触发自己定义的中断。同时 x86 中的 8259A 可编程中断处理芯片和 8253 可编程时钟芯片可以使得我们定时的触发时钟中断。

除此之外，在对其他架构的了解过程中，发现有一些操作系统没有 `in`, `out` 对外设端口的访问指令。而是通过内存映射方式 (Memory-mapped)，例如与 RISC 指令系统的 CPU（如 ARM、PowerPC 等）通常只实现一个物理地址空间，外设 I/O 端口成为内存的一部分。此时，CPU 可以像访问一个内存单元那样访问外设 I/O 端口，而不需要设立专门的外设 I/O 指令。

在阅读一些 riscv 系统代码中，没有像 x86 中读取光标位置，输出字符等端口，而是通过 SBI 这样一个硬件抽象层 (HAL) 完成对屏幕的写入，键盘输出处理这些基本外设操作。时钟中断不像 x86 中需要通过 8253 芯片发送中断，而是 cpu 中已经包含了时钟中断相关寄存器。不同架构上的基本功能大都差不多，只是各种实现方式不同。对于操作系统有了更全面的认识。

## 参考资料

<https://www.nasm.us/xdoc/2.15.05/html/nasmdoc7.html#section-7.7>

[http://en.wikipedia.org/wiki/Name\\_mangling](http://en.wikipedia.org/wiki/Name_mangling)

<https://stackoverflow.com/questions/20369672/undefined-reference-to-dlsym>

<https://stackoverflow.com/questions/39688526/error-when-linking-c-rust-hello-world-undefined-reference-to-stdiostdi>

[https://wiki.osdev.org/Interrupt\\_Descriptor\\_Table](https://wiki.osdev.org/Interrupt_Descriptor_Table)

<https://pdos.csail.mit.edu/6.828/2018/readings/hardware/8259A.pdf>

<http://www.cpcwiki.eu/imgs/e/e3/8253.pdf>

[https://wiki.osdev.org/Programmable\\_Interval\\_Timer](https://wiki.osdev.org/Programmable_Interval_Timer)

<https://bochs.sourceforge.io/techspec/PORTS.LST>

[https://rcore-os.github.io/rCore\\_tutorial\\_doc/chapter3/part5.html](https://rcore-os.github.io/rCore_tutorial_doc/chapter3/part5.html)