

中山大學

SUN YAT-SEN UNIVERSITY

实验课程: 操作系统

实验名称: 中断

专业名称: 计算机科学与技术

学生姓名: 吴臻

学生学号: 21307371

实验地点: 实验中心大楼D栋501

实验成绩:

报告时间: 2023/4/7

- 1. 实验要求
- 2. 预备知识与实验环境
 - 代码编译的四个阶段
 - 中断处理机制
 - 8259A芯片
 - 介绍
 - 8259A的初始化
 - 8259A的工作流程
 - 优先级、中断屏蔽字和EOI消息的动态改变
 - 中断程序的编写思路

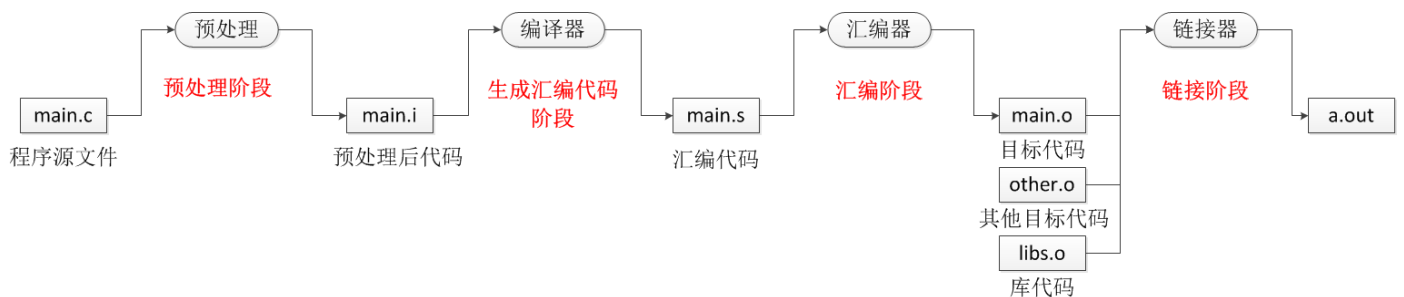
- 3. 实验任务
- 4. 实验步骤/关键代码/实验结果
 - Assignment 1 混合编程的基本思路
 - Assignment 2 使用C/C++编写内核
 - Assignment 3 中断的处理
 - Assignment 4 时钟中断的处理
- 5. 总结(对实验过程中遇到的问题进行总结，可以提出对实验设置的改进意见)
- 6. 参考资料清单

1. 实验要求

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。

2. 预备知识与实验环境

代码编译的四个阶段



- 预处理。处理宏定义，如 `#include`, `#define`, `#ifndef` 等，生成预处理文件，一般以 `.i` 为后缀。 `gcc -o main.i -E main.c`

- 编译。将预处理文件转换成汇编代码文件，一般以 `.s` 为后缀。

```
gcc -o hello.s -S hello.c -masm=intel
```

- 汇编。将汇编代码文件转换成可重定位文件，一般以 `.o` 为后缀。

```
gcc -o main.o -c main.c
```

- 链接。将多个可重定位文件链接生成可执行文件，一般以 `.o` 为后缀。

```
gcc -o main.out main.c print.c
```

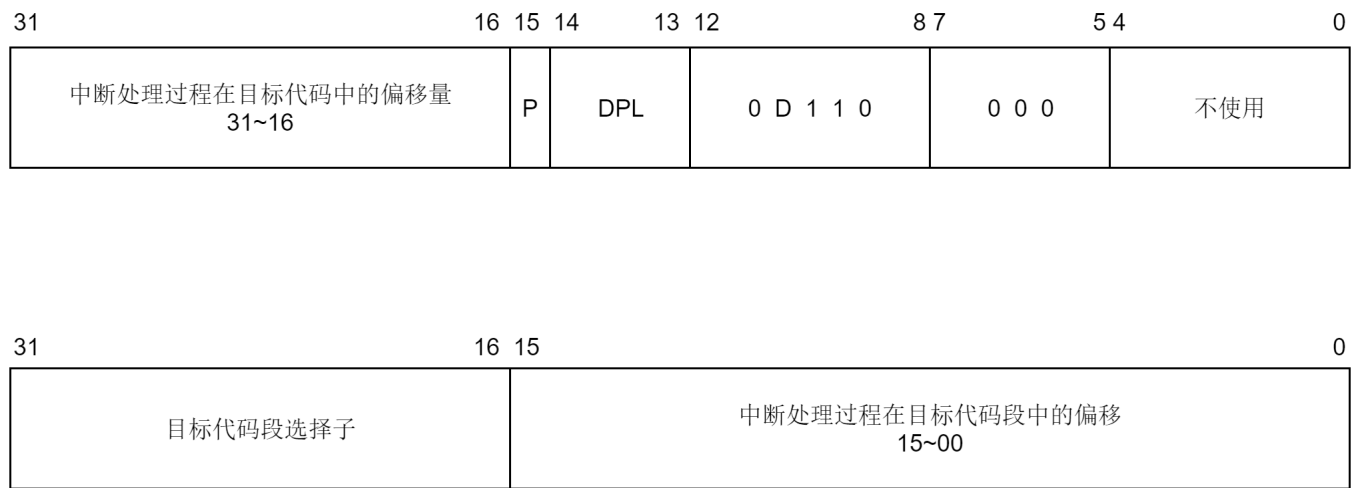
中断处理机制

为了实现在保护模式下的中断程序，我们首先需要了解在保护模式下计算机是如何处理中断程序的。下面是保护模式下中断程序处理过程。

- 中断前的准备。
- CPU 检查是否有中断信号。
- CPU根据中断向量号到IDT中取得处理中断向量号对应的中断描述符。
- CPU根据中断描述符中的段选择子到 GDT 中找到相应的段描述符。
- CPU 根据特权级设定即将运行程序的栈地址。
- CPU保护现场。
- CPU跳转到中断服务程序的第一条指令开始处执行。
- 中断服务程序运行。
- 中断服务程序处理完成，使用iret返回。

我们下面分别来看上面的各个步骤。

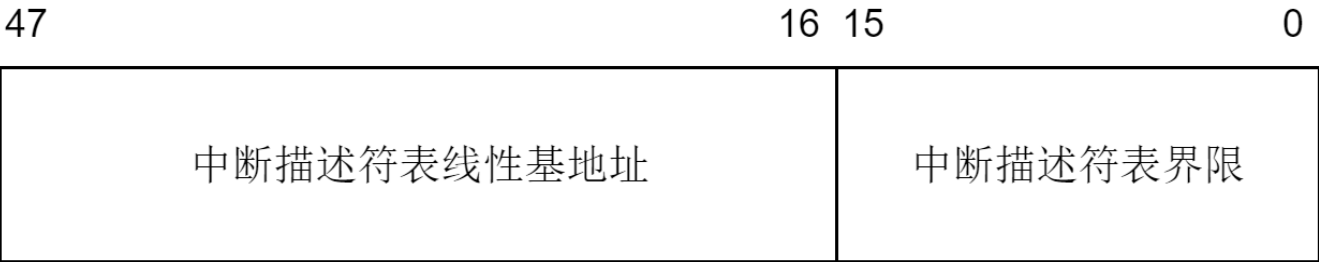
1. 中断前的准备 。为了标识中断处理程序的位置，保护模式使用了中断描述符。一个中断描述符由 64 位构成，其详细结构如下所示。



- 段选择子：中断程序所在段的选择子。
- 偏移量：中断程序的代码在中断程序所在段的偏移位置。
- P位：段存在位。 0表示不存在， 1表示存在。
- DPL： 特权级描述。 0-3 共4级特权， 特权级从0到3依次降低。
- D位： D=1表示32位代码， D=0表示16位代码。
- 保留位：保留不使用。

和段描述符一样，这些中断描述符也需要一个地方集中放置，这些中断描述符的集合被称为中断描述符表 IDT(InterruptDescriptorTable)。和GDT一样，IDT的位置可以任意放

置。但是，为了让CPU能够找到IDT的位置，我们需要将IDT的位置信息等放在一个特殊的寄存器内，这个寄存器是IDTR。CPU则通过IDTR的内容找到中断描述符表的位置，IDTR的结构如下所示。



注意，CPU只能处理前256个中断。因此，我们只会在IDT中放入256个中断描述符。当我们确定了IDT的位置后就使用 `lidt` 指令对IDTR赋值。通过上述步骤，我们便完成了中断的事先准备。

2. CPU检查是否有中断信号
- 除了我们主动调用中断或硬件产生中断外，CPU每执行完一条指令后，CPU还会检查中断控制器，如8259A芯片，看看是否有中断请求。若有中断请求，在相应的时钟脉冲到来时，CPU就会从总线上读取中断控制器提供的中断向量号。
3. CPU根据中断向量号到IDT中取得处理这个向量的中断描述符
- 注意，中断描述符没有选择子的说法。也就是说，中断向量号直接就是中断描述符在IDT的序号。
4. CPU根据中断描述符中的段选择子到GDT中找到相应的段描述符
- CPU会解析在上一步取得的中断描述符，找到其中的目标代码段选择子，然后根据选择子到GDT（段描述符表）中找到相应的段描述符。
5. CPU根据特权级的判断设定即将运行程序的栈地址
- 由于我们后面会实现用户进程，用户进程运行在用户态下，而每一个用户进程都会有自己的栈。因此当中断发生，我们从用户态陷入内核态后，CPU会自动将栈从用户栈切换到内核栈。但在这里，我们只会在内核态编写我们的代码，因此我们的栈地址不会被切换，此步可以暂时忽略。
6. CPU保护现场
- CPU依次将EFLAGS、CS、EIP中的内容压栈。其实，这是在特权级不变时候的情况，如果特权级发生变换，如从用户态切换到内核态后，CPU会依次将SS，ESP，EFLAGS、CS、EIP压栈。不过这里我们暂时只运行在特权级0下。读者在学习操作系统时还会提到CPU会压入错误码，但只有部份中断才会压入错误码，详情见前面的保护模式中断的表格。
7. CPU跳转到中断服务程序的第一条指令开始处执行
- 保护完现场后，CPU会将中断描述符中的段选择子送入CS段寄存器，然后将中断描述符中的目标代码段偏移送入EIP。同时，CPU更新EFLAGS寄存器，若涉及特权级的变换，SS和ESP还会发生变化。当CS和EIP更新后，CPU实际上就跳转到中断服务程序的第一条指令处。

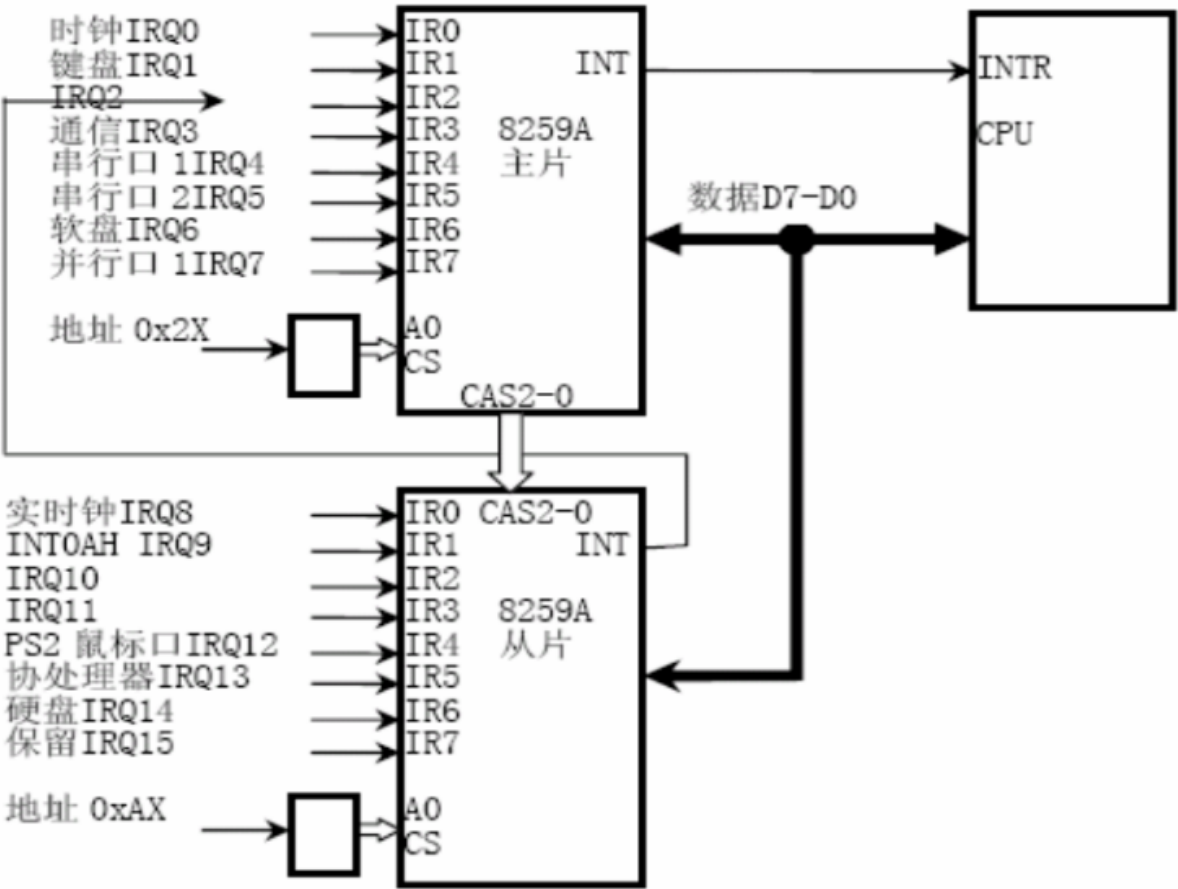
- 中断服务程序运行。此后，中断服务程序开始执行。
- 中断服务程序处理完成，使用`iret`返回。在特权级不发生变化的情况下，`iret`会将之前压入栈的`EFLAGS`，`CS`，`EIP`的值送入对应的寄存器，然后便实现了中断返回。若特权级发生变化，`CPU`还会更新`SS`和`ESP`。

8259A芯片

介绍

8259A芯片又被称为可编程中断控制器（PIC: Programmable Interrupt Controller）。可编程的意思是说，我们可以通过代码来修改8259A的处理优先级、屏蔽某个中断等。在PC中，8259A芯片有两片，分别被称为主片和从片，其结构如下。

8259A



每个8259A都有8根中断请求信号线，IRQ0-IRQ7，默认优先级从高到低。这些信号线与外设相连，外设通过IRQ向8259A芯片发送中断请求。由于历史原因，从片

默认是连接到主片的IRQ2的位置。为了使8259A芯片可以正常工作，我们必须先要对8259A芯片初始化。注意到，主片的IRQ1的位置是键盘中断的位置。此时，熟悉实模式的读者可能会产生疑问：在实模式下即使不进行8259A芯片的相关操作，也可以使用int指令来调用中断。实际上，实模式也是需要操作8259A芯片的，只不过BIOS帮我们做了这个工作。同学们需要特别注意，在保护模式下原先的实模式中断不可以被使用。此时，我们不得不去编程8259A芯片来实现保护模式下的中断程序。

8259A的初始化

在使用8259A芯片之前我们需要对8259A的两块芯片进行初始化。初始化过程是依次通过向8259A的特定端口发送4个ICW，ICW1~ICW4（初始化命令字，Initialization Command Words）来完成的。四个ICW必须严格按照顺序依次发送。

下面是四个ICW的结构。

- ICW1。发送到0x20端口（主片）和0xA0端口（从片端口）。

7	6	5	4	3	2	1	0
0	0	0	1	M	0	C	I

- I位：若置1，表示ICW4会被发送。置0表示ICW4不会被发送。我们会发送ICW4，所以I位置1。
- C位：若置0，表示8259A工作在级联环境下。8259A的主片和从片我们都会使用到，所以C位置0。
- M位：指出中断请求的电平触发模式，在PC机中，M位应当被置0，表示采用“边沿触发模式”。

- ICW2。发送到0x21（主片）和0xA1（从片）端口。

7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	0	0	0

对于主片和从片，ICW2都是用来表示当IRQ0的中断发生时，8259A会向CPU提供的中断向量号。此后，IRQ0，IRQ1，...，IRQ7的中断号为ICW2，ICW2+1，ICW2+2，...，ICW+7。其中，ICW2的低3位必须是0。这里，我们置主片的ICW2为0x20，从片的ICW2为0x28。

- **ICW3**。发送到0x21（主片）和0xA1（从片）端口。ICW3只有在级联工作时才会被发送，它主要用来建立两处PIC之间的连接，对于主片和从片，其结构是不一样的，主片的结构如下所示。

7	6	5	4	3	2	1	0
IRQ7	IRQ6	IRQ5	IRQ4	IRQ3	IRQ2	IRQ1	IRQ0

上面的相应位被置1，则相应的IRQ线就被用作于与从片相连，若置0则表示被连接到外围设备。前面已经提到，由于历史原因，从片被连接到主片的IRQ2位，所以，主片的ICW3=0x04，即只有第2位被置1。

从片的结构如下。

7	6	5	4	3	2	1	0
0	0	0	0	0	IRQ		

IRQ指出是主片的哪一个IRQ连接到了从片，这里，从片的ICW3=0x02，即IRQ=0x02，其他位置均为0。

- **ICW4**。发送到0x21（主片）和0xA1（从片）端口。

7	6	5	4	3	2	1	0
0	0	0	0	0	0	EOI	80x86

- **EOI位**：若置1表示自动结束，在PC位上这位需要被清零，详细原因在后面再提到。
- **80x86位**：置1表示PC工作在80x86架构下，因此我们置1。

到这里，读者已经发现，其实ICW1，ICW3，ICW4的值已经固定，可变的只有ICW2。

8259A的工作流程

对于8259A芯片产生的中断，我们需要手动在中断返回前向8259A发送EOI消息。如果没有发送EOI消息，那么此后的中断便不会被响应。

一个发送EOI消息的示例代码如下，OCW2在后面会介绍。

```
;发送OCW2字
mov al, 0x20
```



```
out 0x20, al
out 0xa0, al
```

8259A的中断处理函数末尾必须加上面这段代码，否则中断不会被响应。

优先级、中断屏蔽字和EOI消息的动态改变

初始化8259A后，我们便可以在任何时候8259A发送OCW字(Operation Command Words)来实现优先级、中断屏蔽字和EOI消息的动态改变。

OCW有3个，分别是OCW1，OCW2，OCW3，其详细结构如下。

- OCW1。中断屏蔽，发送到0x21（主片）或0xA1（从片）端口。

7	6	5	4	3	2	1	0
IRQ7	IRQ6	IRQ5	IRQ4	IRQ3	IRQ2	IRQ1	IRQ0

相应位置1表示屏蔽相应的IRQ请求。同学们很快可以看到，在初始化8259A的代码末尾，我们将0xFF发送到0x21和0xA1端口。这是因为我们还没建立起处理8259A芯片的中断处理函数，所以暂时屏蔽主片和从片的所有中断。

- OCW2。一般用于发送EOI消息，发送到0x20（主片）或0xA0（从片）端口。

7	6	5	4	3	2	1	0
R	SL	EOI	0	0	L2	L1	L0

EOI消息是发送 0x20，即只有EOI位是1，其他位置为0。

- OCW3。用于设置下一个读端口动作将要读取的IRR或ISR，我们不需要使用。

中断程序的编写思路

- 保护现场。现场指的是寄存器的内容，因此在处理中断之前，我们需要手动将寄存器的内容放置到栈上面。待中断返回前，我们会将这部分保护在栈中的寄存器内容放回到相应的寄存器中。
- 中断处理。执行中断处理程序。
- 恢复现场。中断处理完毕后恢复之前放在栈中的寄存器内容，然后执行 `iret` 返回。在执行 `iret` 前，如果有错误码，则需要将错误码弹出栈；如果是8259A芯片

产生的中断，则需要在中断返回前发送EOI消息。注意，8259A芯片产生的中断不会错误码。事实上，只有中断向量号1-19的部分中断才会产生错误码。

其代码描述如下。

```
interrupt_handler_example:
    pushad
    ... ; 中断处理程序
    popad

    ; 非必须

    ; 1 弹出错误码，没有则不可以加入
    add esp, 4

    ; 2 对于8259A芯片产生的中断，最后需要发送EOI消息，若不是则不可以加入
    mov al, 0x20
    out 0x20, al
    out 0xa0, al

    iret
```

注意，中断返回使用的是 `iret` 指令。

3. 实验任务

1. 混合编程的基本思路
2. 使用C/C++编写内核
3. 中断的处理
4. 时钟中断的处理

4. 实验步骤/关键代码/实验结果

Assignment 1 混合编程的基本思路

复现指导书中“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"（参考下图）改为"Done by 你的学号 你的姓名首字母"；

```
WZ@WZ-VirtualBox:~/lab4/src/4$ ./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done by 21307371 WZ
```

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；

C代码调用汇编函数的语法:

- 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`。例如我们需要调用汇编函数`function_from_asm`，那么我们首先需要在汇编代码中将其声明为`global`，否则在链接阶段会找不到函数的实现。

```
global function_from_asm
```

- 然后我们在C/C++中将其声明来自外部即可，如下所示。

```
extern void function_from_asm();
```

在C++中需要声明为`extern "C"`，如下所示。

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

汇编代码调用C函数的语法:

- 当我们需要在汇编代码中使用C的函数`function_from_C`时，我们需要在汇编代码中声明这个函数来自于外部。`extern function_from_C`声明后便可直接使用，例如

```
call function_from_C
```
- 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"`

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

原因：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。

同样地，在汇编代码中声明这个函数即可。

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的

- Makefile基本格式如下：

```
target ... : prerequisites ...  
    command  
    ...  
    ...
```

其中，

- target - 目标文件，可以是 Object File，也可以是可执行文件
- prerequisites - 生成 target 所需要的文件或者目标
- command - make需要执行的命令 (任意的shell命令)，Makefile中的命令必须以 [tab] 开头

- g++/gcc表示编译器

-o指定了生成的可执行文件的名称

```
main.out: main.o c_func.o cpp_func.o asm_func.o  
    g++ -o main.out main.o c_func.o cpp_func.o asm_func.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_func.o: asm_func.asm  
    nasm -o asm_func.o -f elf32 asm_func.asm  
  
clean:  
    rm *.o
```

Assignment 2 使用C/C++编写内核

要求:复现指导书中“内核的加载”部分，在进入setup_kernel函数后，将输出 Hello World改为输出你的学号+姓名首字母，保存结果截图并说说你是怎么做的。

步骤:

1. 在 `mbr` 将 `bootloader` 加载到内存中，并跳转到 `bootloader`
2. 在 `bootloader` 中只负责完成进入保护模式的内容，然后加载操作系统内核到内存中，最后跳转到操作系统内核的起始地址
3. 首先，我们在 `src/boot/entry.asm` 下定义内核进入点。

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

我们会在链接阶段巧妙地将 `entry.asm` 的代码放在内核代码的最开始部份，使得 `bootloader` 在执行跳转到 `0x20000` 后，即内核代码的起始指令，执行的第一条指令是 `jmp setup_kernel`。

4. `setup_kernel` 的定义在文件 `src/kernel/setup.cpp` 中，内容如下。

```
#include "asm_utils.h"

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

    }
}
```

5. `asm_hello_world` 具体实现如下：

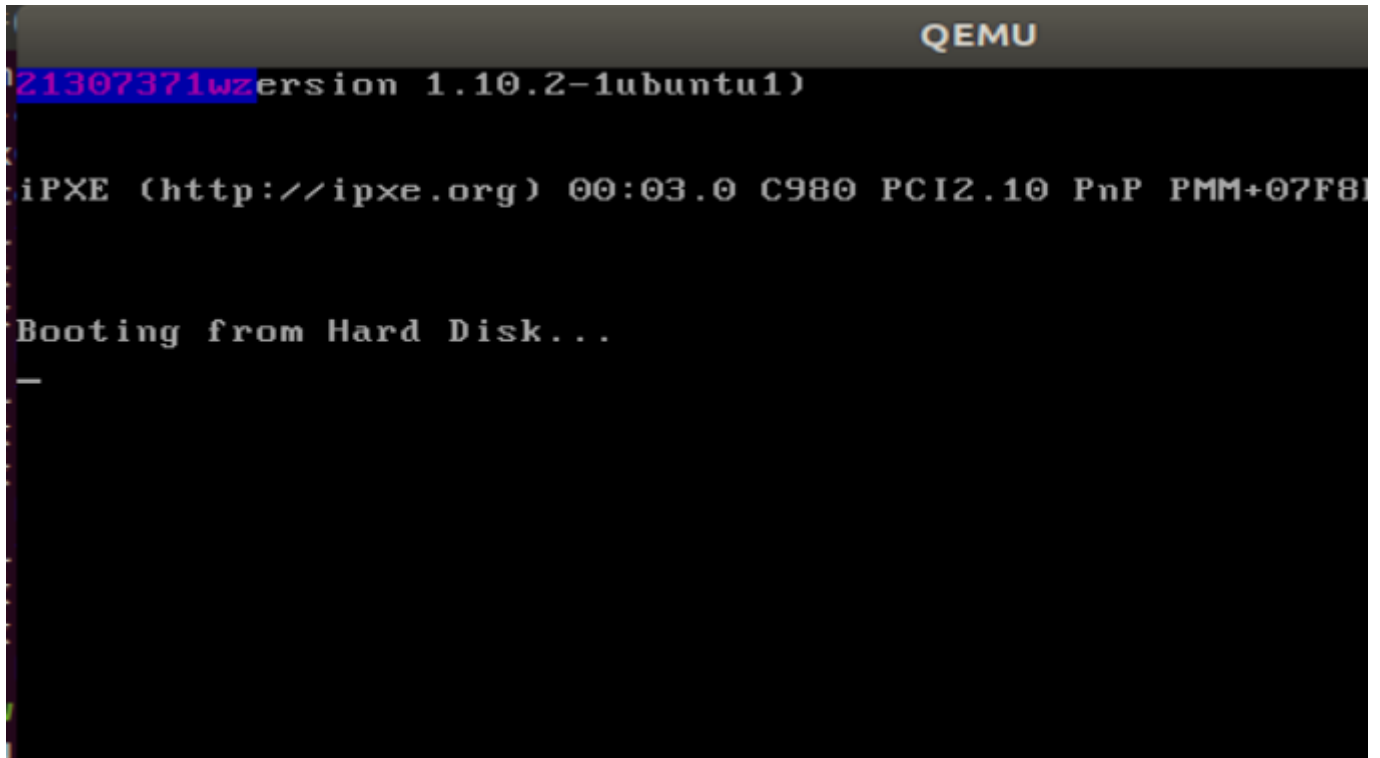
```
asm_hello_world:
    push eax
    xor eax, eax

    mov ecx, my_string_end - my_string
    xor ebx, ebx
    mov esi, my_string
output_my_string:
    mov ah, 0x15
    mov al, [esi]
    mov word[gs:bx], ax
    inc esi
    add ebx, 2
    loop output_my_string
```

```
        pop eax
        ret

my_string db '21307371wz'
my_string_end:
```

结果截图：



Assignment 3 中断的处理

要求:复现指导书中“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，结果截图并说说你是怎么做的。要求：调用处理函数时输出个人特征信息

步骤：

1. 确定IDT的地址。
2. 定义中断默认处理函数。
3. 初始化256个中断描述符。

```
#define IDT_START_ADDRESS 0x8880
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);
}

```

asm_lidt函数实现

```

; 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

ASM_IDTR dw 0
         dd 0

```

默认中断处理函数asm_unhandled_interrupt

```

ASM_UNHANDLED_INTERRUPT_INFO db 'Unhandled interrupt happened, halt by 21307371
WZ'
                                db 0
; void asm_unhandled_interrupt()
asm_unhandled_interrupt:
    cli
    mov esi, ASM_UNHANDLED_INTERRUPT_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

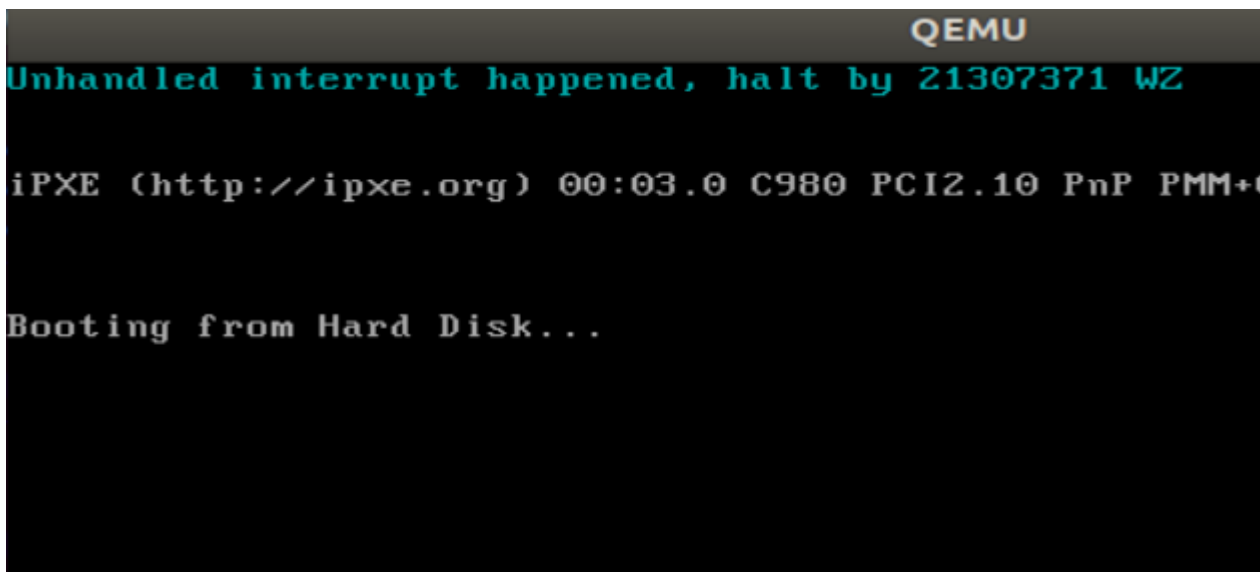
```

```
.end:
    jmp $
```

函数InterruptManager::setInterruptDescriptor的实现

```
// 设置中断描述符
// index    第index个描述符, index=0, 1, ..., 255
// address  中断处理程序的起始地址
// DPL      中断描述符的特权级
void InterruptManager::setInterruptDescriptor(uint32 index, uint32 address, byte
DPL)
{
    IDT[index * 2] = (CODE_SELECTOR << 16) | (address & 0xffff);
    IDT[index * 2 + 1] = (address & 0xffff0000) | (0x1 << 15) | (DPL << 13) |
(0xe << 8);
}
```

结果截图：



Assignment 4 时钟中断的处理

要求：复现指导书中“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，保存结果截图并说说你的思路 and 做法。(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。) 注意：不可以使用纯汇编的方式来实现。

步骤：

1. 首先要对8259A芯片初始化

```
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);
}
```

2. 编写中断处理函数。

```
// 中断处理函数
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);
        }
    }c

    // 记录中断发生的次数
    ++times;
    char number[] = "21307371WZ";

    //定义slow_time来降低跑马灯的频率
    if(times % 8 == 0){
        slow_time++;
    }
    //对slow_time取余防止数组越界
    slow_time %= 10;
    stdio.moveCursor(slow_time);
    stdio.print(number[slow_time], 0x15);
}
```

3. 设置主片IRQ0中断对应的中断描述符。

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

4. 开启时钟中断。

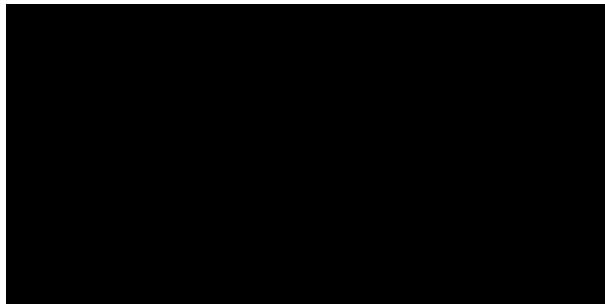
```
void InterruptManager::enableTimeInterrupt()
{
    uint8 value;
    // 读入主片OCW
    asm_in_port(0x21, &value);
    // 开启主片时钟中断, 置0开启
    value = value & 0xfe;
    asm_out_port(0x21, value);
}

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);
}
```

5. 开中断。

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

结果:



5. 总结(对实验过程中遇到的问题进行总结，可以提出对实验设置的改进意见)

此次实验内容比较多，新知识比较多，阅读了几次课件才勉强了解了此次实验的具体步骤，在阅读汇编代码时还是比较吃力，不过好在此次实验任务基本是使用c代码就能解决，还有一个比较复杂的问题就是项目编写时文件的存放位置，此次实验需要使用的文件很多，在一步步操作时也渐渐懂得了相应文件应该存在哪里，最后就是makefile的使用，虽然已经能简单使用了，但是很多语法还是不懂

6. 参考资料清单

c语言: <https://www.cnblogs.com/linzworld/p/13690620.html>

makefile: https://www.cnblogs.com/wang_yb/p/3990952.html