Day 6

Phase 1

SubPhase 1

分割源文件。这里分割源文件的主要方法和我们之前学习c和c++的不太一样：我们暂时还没有使用include语句，我们只是把他们拆分开来，然后编译的时候各自生成目标文件，最后再链接到一起去。

SubPhase 2

Makefile支持使用%作为通配符，因此我们使用%可以将重复相近的生成规则缩成一条来写，以免形成冗长的难以阅读和调试的Makefile。

> https://seisman.github.io/how-to-write-makefile 这是一篇很好的教授如何写makefile的教程

SubPhase 3

整理头文件，我们将定义放在`bootpack.h`头文件中。

其实Phase 1就是简单的拆分一下文件，学习过c语言多文件组织并不是什么难事，这里也没有什么值得多说的地方。

不过我想补充一下，我鼓捣出了一个makefile的修改方法，可以替代之前出错的copy指令

```
haribote.sys : asmhead.bin bootpack.hrb Makefile
    copy /B asmhead.bin+bootpack.hrb haribote.sys
# 替换成
haribote.sys : asmhead.bin bootpack.hrb Makefile
    cmd /c"copy /B asmhead.bin+bootpack.hrb haribote.sys"
```

```
run :
    $(MAKE) img
    $(COPY) haribote.img ..\z_tools\qemu\fdimage0.bin
    $(MAKE) -C ../z_tools/qemu
# 替换成
run :
    $(MAKE) img
    cmd /c"cp haribote.img ..\z_tools\qemu\fdimage0.bin"
    $(MAKE) -C ../z_tools/qemu
```

但实际上为何之前的会出错，这样不会出错，原因不得而知。我觉得应该是make的版本原因。

Phase 2

SubPhase 1

之前我们并不理解`load_gdtr`，今天我们根据书上的解释来理解一波。

```
_load_gdtr:     ; void load_gdtr(int limit, int addr);
        MOV     AX,[ESP+4]      ; limit
        MOV     [ESP+6],AX
        LGDT    [ESP+6]
        RET
```

`lgdt`指令接受一个内存地址，从那里读取6个字节，然后赋值给gdtr寄存器。这6个字节中低16位将作为段上限，高32位将作为gdt的开始地址。

我们要组合出来这6个字节：先将limit存入16位寄存器AX（高位将被丢掉），然后将AX的内容放到ESP+6的位置，这16位的内容和参数addr所处的内存位置正好是我们要的6个字节。于是直接以ESP+6的位置作为lgdt的参数咯。

SubPhase 2

段设置

段上限中一个比较有趣的设置位是Gbit。由于设计上的一些限制，以字节为单位最多只能指定1MB的段大小，设置Gbit后，limit将被解释为页。由于一页是4KB，所以段最多可以指定到4GB了。

段访问权属性

CPU到底是处于系统模式还是应用模式，取决于执行中的应用程序是位于访问权为0x9a的 段，还是位于访问权为0xfa的段。

Phase 3

PIC是programmable interrupt controller，意思是可编程中断控制器。CPU只能单独处理一个中断，这无法满足众多的低速外设的中断需求。所以使用PIC将众多中断信号合成一个中断信号。一个PIC可以监听8个信号，使用两个PIC，其中一个直接与CPU相连，另一个与第一个PIC相连。与CPU直接相连的PIC叫做主PIC，另一个PIC叫做从PIC，这样总共能处理15个中断信号了。

很多设定是要查看PIC文档才能知道如何设置的，我们就先暂时囫囵吞枣一下吧。

从PIC接在主PIC的IRQ2口上。

SubPhase 1

```
void init_pic(void)
/* PICの初期化 */
{
    io_out8(PIC0_IMR, 0xff ); /* 禁止所有中断 */
    io_out8(PIC1_IMR, 0xff ); /* 禁止所有中断 */
    io_out8(PIC0_ICW1, 0x11 ); /* 边沿触发模式 (edge trigger mode)  */
    io_out8(PIC0_ICW2, 0x20 ); /* IRQ0-7由INT20-27接收 */
    io_out8(PIC0_ICW3, 1 << 2); /* PIC1由IRQ2连接 */
    io_out8(PIC0_ICW4, 0x01 ); /* 无缓冲区模式 */
    io_out8(PIC1_ICW1, 0x11 ); /* 边沿触发模式 (edge trigger mode)  */
    io_out8(PIC1_ICW2, 0x28 ); /* IRQ8-15由INT28-2f接收 */
    io_out8(PIC1_ICW3, 2 ); /* PIC1由IRQ2连接 */
    io_out8(PIC1_ICW4, 0x01 ); /* 无缓冲区模式 */
    io_out8(PIC0_IMR, 0xfb ); /* 11111011 PIC1以外全部禁止 */
    io_out8(PIC1_IMR, 0xff ); /* 11111111 禁止所有中断 */
```

```
    return;
}
```

我们一会将要编写0x2c和0x21的中断handler，来实现对鼠标和键盘的响应。

SubPhase 2

中断的处理不能只靠C语言进行，因为要进行中断返回（iretd）。这个指令是C语言无法直接调用的，所以我们还是得用nask来写，但我们还是想用c语言怎么办呢？我们可以先写c语言程序，然后在nask程序中调用它。这样我们就可以兼得nask和c语言的好处了！

```
void inthandler21(int *esp)
/* 来自PS/2键盘的中断 */
{
    struct BOOTINFO *binfo = (struct BOOTINFO *) ADR_BOOTINFO;
    boxfill8(binfo->vram, binfo->scrnx, COL8_000000, 0, 0, 32 * 8 - 1, 15);
    putfonts8_asc(binfo->vram, binfo->scrnx, 0, 0, COL8_FFFFFF, "INT 21 (IRQ-1) : PS/2
keyboard");
    for (;;) {
        io_hlt();
    }
}
```

```
    EXTERN _inthandler21, _inthandler2c
_asm_inthandler21:
    PUSH ES
    PUSH DS
    PUSHAD
    MOV EAX,ESP
    PUSH EAX
    MOV AX,SS
    MOV DS,AX
    MOV ES,AX
    CALL _inthandler21
    POP EAX
    POPAD
    POP DS
    POP ES
    IRETD
```

其中`pushad`相当于

```
PUSH EAX
PUSH ECX
PUSH EDX
PUSH EBX
PUSH ESP
PUSH EBP
PUSH ESI
PUSH EDI
```

而`popad`相当于

```
POP EDI
POP ESI
POP EBP
POP ESP
POP EBX
POP EDX
POP ECX
POP EAX
```

这两个指令相当于保存和回复CPU现场。

对了，PUSH和POP是栈的概念，这个我们早就在数据结构等相关课程上学习过了，书上的看一下也就明白了

```
MOV AX,SS
MOV DS,AX
MOV ES,AX
```

这三个命令是因为C语言自以为是地认为“DS也好，ES也好，SS也好，它们都是指同一个段”，所以如果不按照它的想法设定的话，函数inthandler21就不能顺利执行。

既然改变了DS和ES的值，那么我们在改变之前也应该保存现场和恢复现场。

因此我们有

```
push es
push ds
; ---------
pop ds
pop es
```

然后我们再按照相同方法搞一下0x2c中断服务程序（鼠标）就可以了。

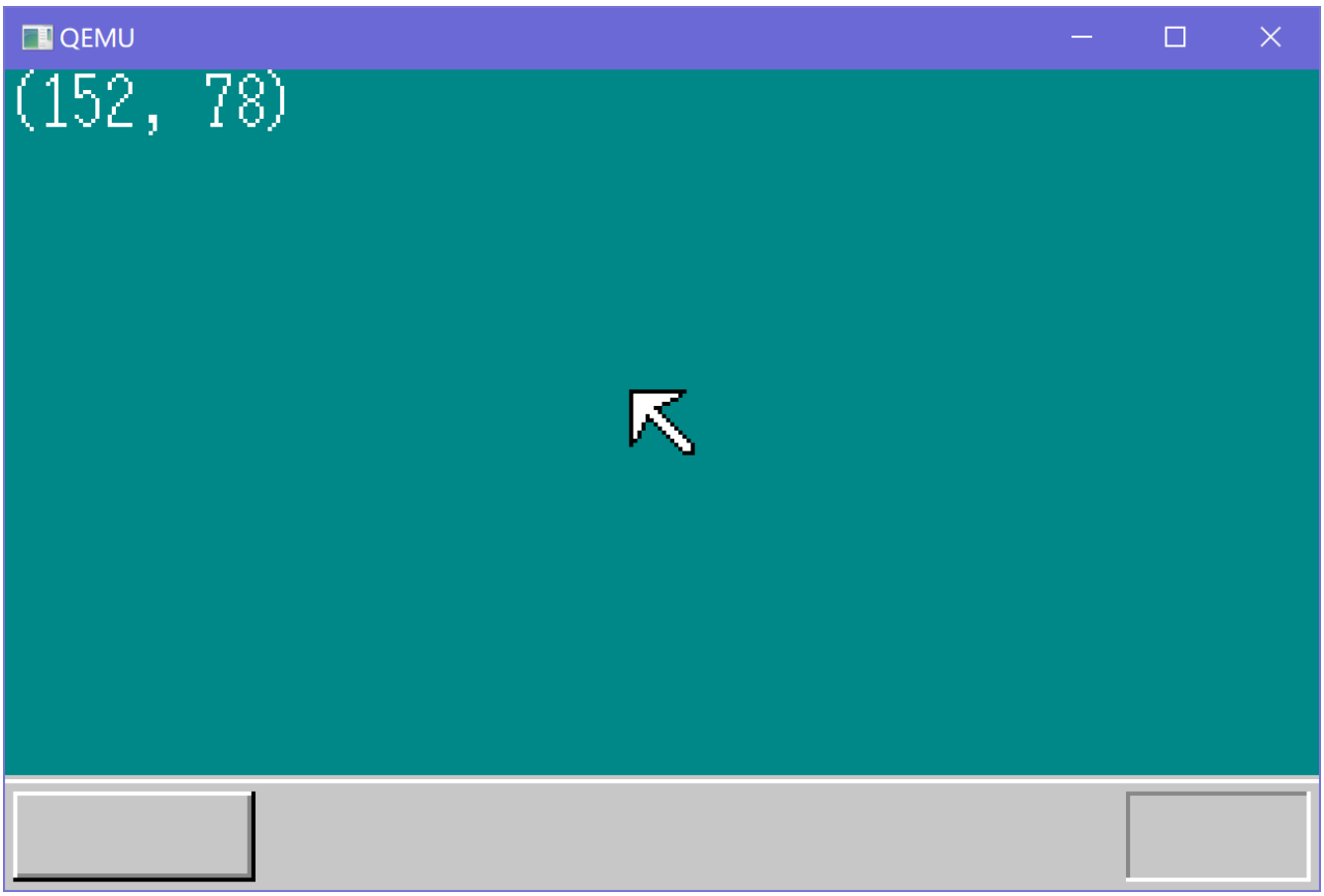
SubPhase 3

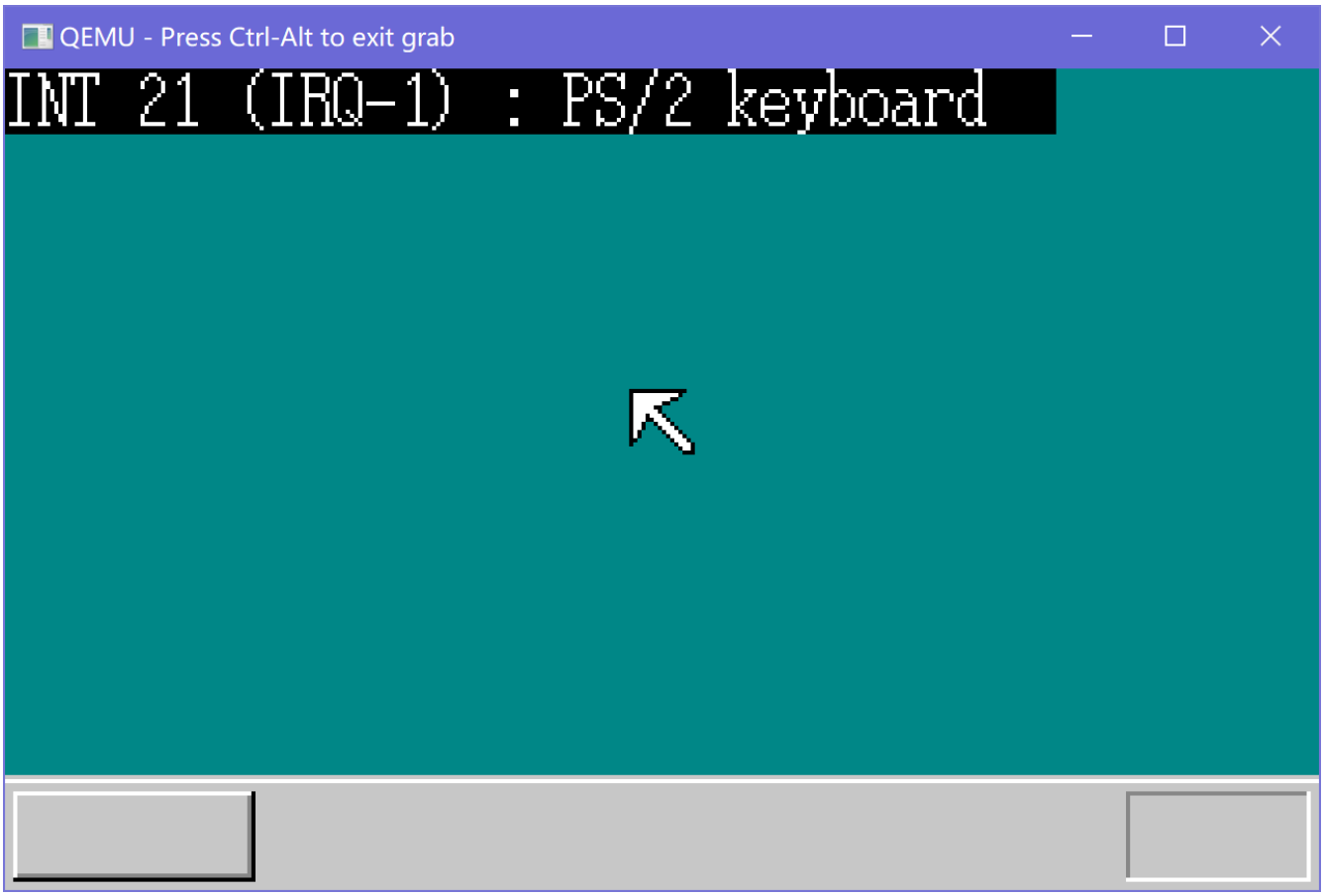
然后我们就可以注册中断处理程序啦

```
set_gatedesc(idt + 0x21, (int) asm_inthandler21, 2 * 8, AR_INTGATE32);
set_gatedesc(idt + 0x2c, (int) asm_inthandler2c, 2 * 8, AR_INTGATE32);
```

2的意思是段号是2，乘8的原因是低3位有别的意思。

然后我们就来试一试吧！





鼠标的中断似乎还不能正常处理。今天就先到这里了。