第6天　分割编译与中断处理

- 分割源文件（harib03a）
- 整理Makefile（harib03b）
- 整理头文件（harib03c）
- 意犹未尽
- 初始化PIC（harib03d）
- 中断处理程序的制作（harib03e）

1　分割源文件（**harib03a**）

本来想接着详细讲解一下昨天剩下的程序，但一上来就说这些，有点乏味，所以还是先做点准备活动吧。不经意地看一下bootpack.c，发现它竟然已长达近300行，是太长了点。所以我们决定把它分割为几部分。

将源文件分割为几部分的利弊，大致如下。

优点

1. 按照处理内容进行分类，如果分得好的话，将来进行修改时，容易找到地方。

2. 如果Makefile写得好，只需要编译修改过的文件，就可以提高make的速度。

3. 单个源文件都不长。多个小文件比一个大文件好处理。

4. 看起来很酷（笑）。

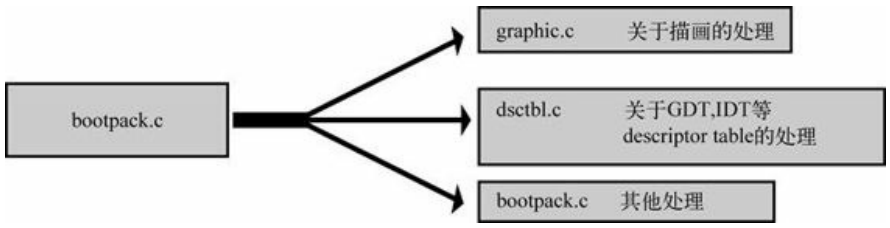
缺点

5. 源文件数量增加。

6. 分类分得不好的话，修改时不容易找到地方。

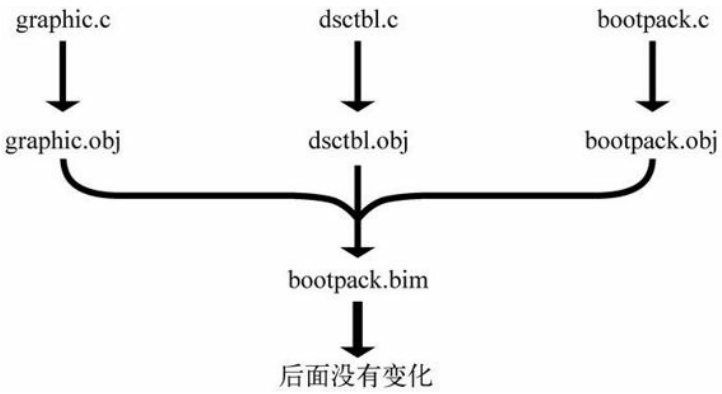
■■■■■

我们先将源文件按下图分割一下看看。



分割并不是很难，但有一点很关键。比如如果graphic.c也想使用naskfunc.nas的函数，就必须要写上“void io_out8（int port,int data）;”这种函数声明。虽然这都已经写在bootpack.c里了，但编译器在编译graphic.c时，根本不知道有bootpack.c存在。

这样整理一下看起来就清爽多了。对应源文件的分割，我们还要修改Makefile，流程如下：



理解了这个流程，Makefile也就很容易看懂了。

现在再来“make run”。运行起来一点问题也没有，分割成功了。

2　整理Makefile（harib03b）

分割虽然成功了，但现在Makefile又有点长了，足足有113行。虽说出现这种情况是情有可原，但是，像这样：

```
bootpack.gas : bootpack.c Makefile
    $(CC1) -o bootpack.gas bootpack.c

graphic.gas : graphic.c Makefile
    $(CC1) -o graphic.gas graphic.c

dsctbl.gas : dsctbl.c Makefile
    $(CC1) -o dsctbl.gas dsctbl.c
```

或者像这样：

```
bootpack.nas : bootpack.gas Makefile
    $(GAS2NASK) bootpack.gas bootpack.nas

graphic.nas : graphic.gas Makefile
    $(GAS2NASK) graphic.gas graphic.nas

dsctbl.nas : dsctbl.gas Makefile
    $(GAS2NASK) dsctbl.gas dsctbl.nas
```

它们做的都是同样的事。为什么要写这么多同样的东西呢？每次增加新的源文件，都要像这样增加这么多雷同的编译规则，看着都烦。

■■■■■

其实有一个技巧可以将它们归纳起来，这就是利用一般规则。我们可以把上面6个独立的文件生成规则，归纳成以下两个一般规则。

```
%.gas : %.c Makefile
    $(CC1) -o $*.gas $*.c

%.nas : %.gas Makefile
    $(GAS2NASK) $*.gas $*.nas
```

哦，这玩意儿好！真方便。

make.exe会首先寻找普通的生成规则，如果没找到，就尝试用一般规则。所以，即使一般规则和普通生成规则有冲突，也不会有问题。这时候，普通生成规则的优先级更高。比如虽然某个文件的扩展名也是.c，但是想用单独的规则来编译它，这也没问题。真聪明呀。

所以，Makefile中可以用一般规则的地方我们都换成了一般规则。这样程序就精简成了92行。减了21行呢，感觉太棒了。

我们来确认一下，运行“make run”。很好，完全能正常运行。

3　整理头文件（**harib03c**）

Makefile变短了，真让人高兴。我们继续把源文件也整理一下。现在的文件大小如下。

graphic.c 187行

dsctbl.c 67行

bootpack.c 81行

合计 **..... 335**行

这比分割前的280行多了不少。主要原因在于各个源文件都要重复声明“vold io_out8（int port, int data）;”等，虽然说这也是迫不得已，但还是不甘心。所以，我们在这儿再下点工夫。

■■■■■

首先将重复部分全部去掉，把他们归纳起来，放到名为bootpack.h的文件里。虽然扩展名变了，但它也是C语言的文件。已经有一个文件名叫bootpack.c了，我们根据一般的做法，将文件命名为bootpack.h。因为是第一次接触到.h文件，所以我们截取bootpack.h内容靠前的一段放在下面。

bootpack.h的内容

```
/* asmhead.nas */
struct BOOTINFO { /* 0x0ff0-0x0fff */
    char cyls; /* 启动区读硬盘读到何处为止 */
    char leds; /* 启动时键盘LED的状态 */
    char vmode; /* 显卡模式为多少位彩色 */
    char reserve;
    short scrnx, scrny; /* 画面分辨率 */
    char *vram;
};
#define ADR_BOOTINFO    0x00000ff0

/* naskfunc.nas */
void io_hlt(void);
void io_cli(void);
void io_out8(int port, int data);
int io_load_eflags(void);
void io_store_eflags(int eflags);
void load_gdtr(int limit, int addr);
void load_idtr(int limit, int addr);

/* graphic.c */
void init_palette(void);
void set_palette(int start, int end, unsigned char *rgb);
void boxfill8(unsigned char *vram, int xsize, unsigned char c, int x0, int y0, int x1, int y1
void init_screen8(char *vram, int x, int y);
    （以下略）
```

这个文件里不仅仅罗列出了函数的定义，还在注释中写明了函数的定义在哪一个源文件里。想要看一看或者修改函数定义时，只要看一下文件bootpack.h就能知道该

函数定义本身在哪个源文件里。这就像目录一样，很方便。

在编译graphic.c的时候，我们要让编译器去读这个头文件，做法是在graphic.c的前面加上如下一行：

```
#include "bootpack.h"
```

编译器见到了这一行，就将该行替换成所指定文件的内容，然后进行编译。所以，写在“bootpack.h”里的所有内容，也都间接地写到了“graphic.c”中。同样道理，在“dsctbl.c”和“bootpack.c”的前面也都加上一行“#include "bootpack.h"”。

■■■■■

像这样，仅由函数声明和#define等组成的文件，我们称之为头文件。头文件英文为header，顾名思义，是指放在程序头部的文件。为什么要放在头部呢？因为像“void io_out8（int port,int data）;”这种声明必须在一开始就让编译器知道。

前面曾经提到，要使用spintf函数，必须在程序的前面写上#include 语句。这正是因为stdio.h中含有对sprintf函数的声明。虽然括住文件名的记号有引号和尖括号的区别，但那也只是文件所处位置的不同而已。双引号（"）表示该头文件与源文件位于同一个文件夹里，而尖括号（<　>）则表示该头文件位于编译器所提供的文件夹里。

这次用了很多#define语句，把用到的地址都只写在了bootpack.h文件里。之所以这么做是因为，如果以后想要变更地址的话，只修改bootpack.h一个文件就行了。

好了，我们运行一下每次必做的“make　run”确认一下。挺好挺好，运行结果没有问题。现在再来确认一下源文件的长度。

bootpack.h 69行

graphic.c 156行

dsctbl.c 51行

bootpack.c 25行

合计 **..... 301**行

整体共缩短了34行[1]，真是太好了。

[1] 分割前是280行，这样算来结果还增加了21行，不过因为我们进行了分割，所以无法避免这种情况。而我们分割的目的也不是为了缩短源文件，所以总的来说还是比较满意的。（可在6.1节确认分割的目的）

4　意犹未尽

好了，现在来详细讲一下昨天遗留下来的问题。首先来说明一下naskfunc.nas的_load_gdtr。

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

这个函数用来将指定的段上限（limit）和地址值赋值给名为GDTR的48位寄存器。这是一个很特别的48位寄存器，并不能用我们常用的MOV指令来赋值。给它赋值的时候，唯一的方法就是指定一个内存地址，从指定的地址读取6个字节（也就是48位），然后赋值给GDTR寄存器。完成这一任务的指令，就是LGDT。

该寄存器的低16位[1]（即内存的最初2个字节）是段上限，它等于“GDT的有效字节数 - 1”。今后我们还会偶尔用到上限这个词，意思都是表示量的大小，一般为“字节数 - 1”。剩下的高32位（即剩余的4个字节），代表GDT的开始地址。

[1] 对于一个多位数字组成的数，靠近右边的位称为低位。反之，靠近左边的位称为高位。

在最初执行这个函数的时候，DWORD[ESP + 4]里存放的是段上限，DWORD[ESP+8]里存放的是地址。具体到实际的数值，就是0x0000ffff和0x00270000。把它们按字节写出来的话，就成了[FF FF 00 00 00 27 00]（要注意低位放在内存地址小的字节里[2]）。为了执行LGDT，笔者希望把它们排列成[FF FF 00 00 00 27 00]的样子，所以就先用“MOV AX,[ESP + 4]”读取最初的0xffff，然后再写到[ESP + 6]里。这样，结果就成了[FF FF FF FF 00 27 00 00]，如果从[ESP + 6]开始读6字节的话，正好是我们想要的结果。

[2] 请大家回想一下2.2节。

■■■■■

naskfunc.nas的_load_idtr设置IDTR的值，因为IDTR与GDTR结构体基本上是一样的，程序也非常相似。

最后再补充说明一下dsctbl.c里的set_segmdesc函数。这个有些难度，我们仅介绍一些与本书相关的内容。

本次的**dsctbl.c**节选

```
struct SEGMENT_DESCRIPTOR {
    short limit_low, base_low;
    char base_mid, access_right;
    char limit_high, base_high;
};

void set_segmdesc(struct SEGMENT_DESCRIPTOR *sd, unsigned int limit, int base, int ar)
```

```
{
    if (limit > 0xfffff) {
        ar |= 0x8000; /* G_bit = 1 */
        limit /= 0x1000;
    }
    sd->limit_low    = limit & 0xffff;
    sd->base_low     = base & 0xffff;
    sd->base_mid     = (base >> 16) & 0xff;
    sd->access_right = ar & 0xff;
    sd->limit_high   = ((limit >> 16) & 0x0f) | ((ar >> 8) & 0xf0);
    sd->base_high    = (base >> 24) & 0xff;
    return;
}
```

说到底，这个函数是按照CPU的规格要求，将段的信息归结成8个字节写入内存的。这8个字节里到底填入了什么内容呢？昨天已经讲到，有以下3点：

- 段的大小
- 段的起始地址
- 段的管理属性（禁止写入，禁止执行，系统专用等）

为了写入这些信息，我们准备了struct SEGMENT_DESCRIPTOR这样一个结构体。下面我们就来说明这个结构体。

■■■■■

首先看一下段的地址。地址当然是用32位来表示。这个地址在CPU世界的语言里，被称为段的基址。所以这里使用了base这样一个变量名。在这个结构体里base又分为low（2字节），mid（1字节），high（1字节）3段，合起来刚好是32位。所以，这里只要按顺序分别填入相应的数值就行了。虽然有点难懂，但原理很简单。程序中使用了移位运算符和AND运算符往各个字节里填入相应的数值。

为什么要分为3段呢？主要是为了与80286时代的程序兼容。有了这样的规格，80286用的操作系统，也可以不用修改就在386以后的CPU上运行了。

■■■■■

下面再说一下段上限。它表示一个段有多少个字节。可是这里有一个问题，段上限最大是4GB，也就是一个32位的数值，如果直接放进去，这个数值本身就要占用4个字节，再加上基址（base），一共就要8个字节，这就把整个结构体占满了。这样一来，就没有地方保存段的管理属性信息了，这可不行。

因此段上限只能使用20位。这样一来，段上限最大也只能指定到1MB为止。明明有4GB，却只能用其中的1MB，有种又回到了16位时代的错觉，太可悲了。在这里英特尔的叔叔们又想了一个办法，他们在段的属性里设了一个标志位，叫做Gbit。这个标志位是1的时候，limit的单位不解释成字节（byte），而解释成页（page）。页是什么呢？在电脑的CPU里，1页是指4KB。

这样一来，4KB × 1M = 4GB，所以可以指定4GB的段。总算能放心了。顺便说一

句，G bit的“G”，是“granularity”的缩写，是指单位的大小。

这20位的段上限分别写到limit_low和limit_high里。看起来它们好像是总共有3字节，即24位，但实际上我们接着要把段属性写入limit_high的高4位里，所以最后段上限还是只有20，好复杂呀。

■■■■■

最后再来讲一下12位的段属性。段属性又称为“段的访问权属性”，在程序中用变量名access_right或ar来表示。因为12位段属性中的高4位放在limit_high的高4位里，所以程序里有意把ar当作如下的16位构成来处理：

xxxx0000xxxxxxxx(其中**x**是**0**或**1)**

ar的高4位被称为“扩展访问权”。为什么这么说呢？因为这高4位的访问属性在80286的时代还不存在，到386以后才可以使用。这4位是由“GD00”构成的，其中G是指刚才所说的G bit，D是指段的模式，1是指32位模式，0是指16位模式。这里出现的16位模式主要只用于运行80286的程序，不能用于调用BIOS。所以，除了运行80286程序以外，通常都使用D=1的模式。

■■■■■

ar的低8位从80286时代就已经有了，如果要详细说明的话，够我们说一天的了，所以这里只是简单地介绍一下。

00000000（**0x00**）：未使用的记录表（**descriptor table**）。

10010010（**0x92**）：系统专用，可读写的段。不可执行。

10011010（**0x9a**）：系统专用，可执行的段。可读不可写。

11110010（**0xf2**）：应用程序用，可读写的段。不可执行。

11111010（**0xfa**）：应用程序用，可执行的段。可读不可写。

“系统专用”，“应用程序用”什么的，听着让人摸不着头脑。都是些什么东西呀？在32位模式下，CPU有系统模式（也称为“ring0”[3]）和应用模式（也称为“ring3”）之分。操作系统等“管理用”的程序，和应用程序等“被管理”的程序，运行时的模式是不同的。

[3] 除此之外，还有ring1和ring2，这些中间阶段，由device driver（设备驱动器）等使用。ring原意是轮子或环，有时用它来表示阶段，故得此名。

比如，如果在应用模式下试图执行LGDT等指令的话，CPU则对该指令不予执行，并马上告诉操作系统说 “那个应用程序居然想要执行LGDT，有问题！”。另外，当应用程序想要使用系统专用的段时，CPU也会中断执行，并马上向操作系统报告“那个应用程序想要盗取系统信息。也有可能不仅要盗取信息，还要写点东西来破坏系统呢。”

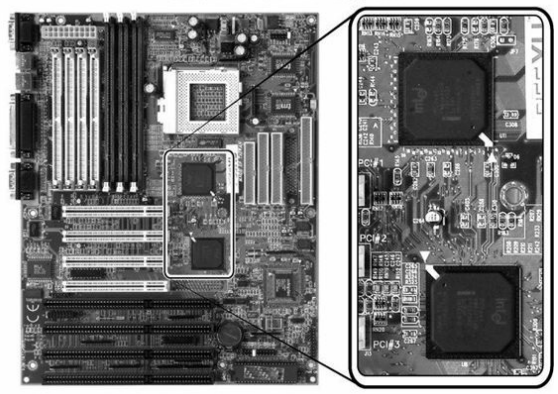
“想要盗取系统信息这一点我明白，但要阻止LGDT的执行这一点，我还是不懂。”可能有人会有这种疑问。当然要阻止啦。因为如果允许应用程序执行LGDT，那应用程序就会根据自己的需要，偷偷准备GDT，然后重新设定LGDT来让它执行自己准备的GDT。这可就麻烦了。有了这个漏洞，操作系统再怎么防守还是会防不胜防。

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。

5　初始化PIC（harib03d）

那好，现在欠债（指昨天没讲完的部分）也还清了，就继续往后讲吧。我们接着昨天继续做鼠标指针的移动。为达到这个目的必须使用中断，而要使用中断，则必须将GDT和IDT正确无误地初始化。

主板上配置的芯片组



那就赶紧使用中断吧……但是，还有一件该做的事没做——还没有初始化PIC。那么我们现在就来做。

所谓PIC是“programmable interrupt controller”的缩写，意思是“可编程中断控制器”。PIC与中断的关系可是很密切的哟。它到底是什么呢？在设计上，CPU单独只能处理一个中断，这不够用，所以IBM的大叔们在设计电脑时，就在主板上增设了几个辅助芯片。现如今它们已经被集成在一个芯片组里了。

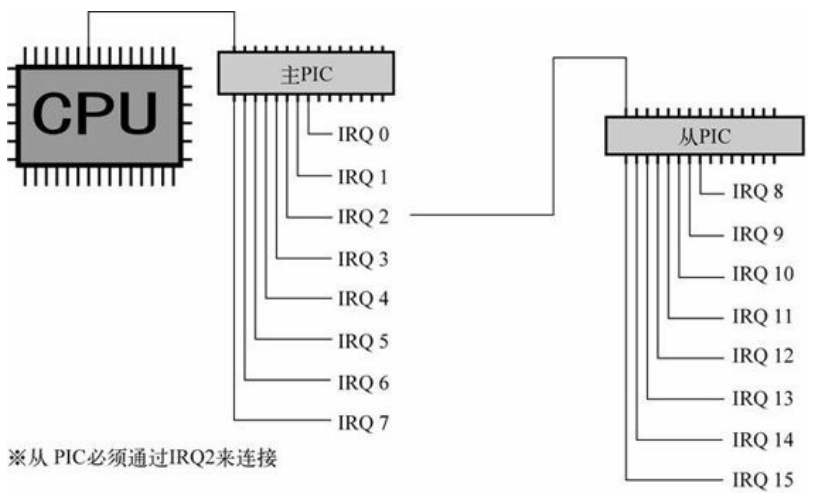
PIC是将8个中断信号[1]集合成一个中断信号的装置。PIC监视着输入管脚的8个中断信号，只要有一个中断信号进来，就将唯一的输出管脚信号变成ON，并通知给CPU。IBM的大叔们想要通过增加PIC来处理更多的中断信号，他们认为电脑会有8个以上的外部设备，所以就把中断信号设计成了15个，并为此增设了2个PIC。

[1] 英文是interrupt request，缩写为IRQ。

那它们的线路是如何连接的呢？如下页图所示。

与CPU直接相连的PIC称为主PIC（master PIC），与主PIC相连的PIC称为从PIC（slave PIC）。主PIC负责处理第0到第7号中断信号，从PIC负责处理第8到第15号中断信号。master意为主人，slave意为奴隶，笔者搞不清楚这两个词的由来，但现在结果是不论从PIC如何地拼命努力，如果主PIC不通知给CPU，从PIC的意思也就不能传达给CPU。或许是从这种关系上考虑，而把它们一个称为主人，一个称为奴隶。

另外，从PIC通过第2号IRQ与主PIC相连。主板上的配线就是这样，无法用软件来改变。



为什么是第2号IRQ呢？事实上笔者也搞不清楚。是不是因为第0号和第1号已经被占用了，而第2号现在还空着，所以就用它了呢。嗯……如果有人想进一步了解这个问题，请一定打电话问问IBM的大叔们。

■■■■■

有人可能会纳闷儿，怎么突然讲起硬件来了？这是因为，如果不懂得这部分的硬件结构，就无法顺利设定PIC。

int.c的主要组成部分

```
void init_pic(void)
/* PIC的初始化 */
{
    io_out8(PIC0_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;
}
```

以上是PIC的初始化程序。从CPU的角度来看，PIC是外部设备，CPU使用OUT指令进行操作。程序中的PIC0和PIC1，分别指主PIC和从PIC。PIC内部有很多寄存器，用端口号码对彼此进行区别，以决定是写入哪一个寄存器。

具体的端口号码写在bootpack.h里，请参考这个程序。但是，端口号相同的东西有很多，可能会让人觉得混乱。不过笔者并没有搞错，写的是正确的。因为PIC有些很细微的规则，比如写入ICW1之后，紧跟着一定要写入ICW2等，所以即使端口号相同，也能够很好地区别开来。

■■■■■

现在简单介绍一下PIC的寄存器。首先，它们都是8位寄存器。IMR是“interrupt mask register”的缩写，意思是“中断屏蔽寄存器”。8位分别对应8路IRQ信号。如果某一位的值是1，则该位所对应的IRQ信号被屏蔽，PIC就忽视该路信号。这主要是因为，正在对中断设定进行更改时，如果再接受别的中断会引起混乱，为了防止这种情况的发生，就必须屏蔽中断。还有，如果某个IRQ没有连接任何设备的话，静电干扰等也可能会引起反应，导致操作系统混乱，所以也要屏蔽掉这类干扰。

ICW是“initial control word”的缩写，意为“初始化控制数据”。因为这里写着word，所以我们会想，“是不是16位”？不过，只有在电脑的CPU里，word这个词才是16位的意思，在别的设备上，有时指8位，有时也会指32位。PIC不是仅为电脑的CPU而设计的控制芯片，其他种类的CPU也能使用，所以这里word的意思也并不是我们觉得理所当然的16位。

ICW有4个，分别编号为1~4，共有4个字节的数据。ICW1和ICW4与PIC主板配线方式、中断信号的电气特性等有关，所以就不详细说明了。电脑上设定的是上述程序所示的固定值，不会设定其他的值。如果故意改成别的什么值的话，早期的电脑说不定会烧断保险丝，或者器件冒　烟[2]；最近的电脑，对这种设定起反应的电路本身被省略了，所以不会有任何反应。

[2]　电路上，+5V与GND（地）短路时，就会发生保险丝熔断、器件冒烟的现象。这可不是吓唬你，而是真的会发生。

ICW3是有关主—从连接的设定，对主PIC而言，第几号IRQ与从PIC相连，是用8位来设定的。如果把这些位全部设为1，那么主PIC就能驱动8个从PIC（那样的话，最大就可能有64个IRQ），但我们所用的电脑并不是这样的，所以就设定成00000100。另外，对从PIC来说，该从PIC与主PIC的第几号相连，用3位来设定。因为硬件上已经不可能更改了，如果软件上设定不一致的话，只会发生错误，所以只能维持现有设定不变。

■■■■■

因此不同的操作系统可以进行独特设定的就只有ICW2了。这个ICW2，决定了IRQ以哪一号中断通知CPU。“哎？怎么有这种事？刚才不是说中断信号的管脚只有1根吗？”嗯，话是那么说，但PIC还有个挺有意思的小窍门，利用它就可以由PIC来设定中断号了。

大家可能会对此有兴趣，所以再详细介绍一下。中断发生以后，如果CPU可以受理这个中断，CPU就会命令PIC发送2个字节的数据。这2个字节是怎么传送的呢？CPU与PIC用IN或OUT进行数据传送时，有数据信号线连在一起。PIC就是利用这个信号线发送这2个字节数据的。送过来的数据是“0xcd　0x??”这两个字节。由于电路设计的原因，这两个字节的数据在CPU看来，与从内存读进来的程序是完全一样的，所以CPU就把送过来的“0xcd 0x??”作为机器语言执行。这恰恰就是把数据当作程序来执行的情况。这里的0xcd就是调用BIOS时使用的那个INT指令。我们在程序里写的“INT　　0x10”，最后就被编译成了“0xcd 0x10”。所以，CPU上了PIC的当，按照PIC所希望的中断号执行了INT指令。

这次是以INT 0x20~0x2f接收中断信号IRQ0~15而设定的。这里大家可能又会有疑问

了。“直接用INT　0x00~0x0f就不行吗？这样与IRQ的号码不就一样了吗？为什么非要加上0x20？”不要着急，先等笔者说完再问嘛。是这样的，INT　0x00~0x1f不能用于IRQ，仅此而已。

之所以不能用，是因为应用程序想要对操作系统干坏事的时候，CPU内部会自动产生INT 0x00~0x1f，如果IRQ与这些号码重复了， CPU就分不清它到底是IRQ，还是CPU的系统保护通知。

这样，我们就理解了这个程序，把它保存为int.c。今后要进行中断处理的还有很多，所以我们就给它另起了一个名字。从bootpack.c的HariMain调用init_pic。

我们来运行一下“make　run”。因为这只是内部设定，所以画面上没有什么变化，虽然觉得不过瘾没有特别大的成就感，但看起来可以正常运行。

6　中断处理程序的制作（**harib03e**）

[1]　重印时的补充说明：本文中只讲到了IRQ1和IRQ12的中断处理程序。事实上附属光盘中还有IRQ7的中断处理程序。要它干什么呢？因为对于一部分机种而言，随着PIC的初始化，会产生一次IRQ7中断，如果不对该中断处理程序执行STI（设置中断标志位，见第4章），操作系统的启动会失败。关于inthandler27的处理内容，大家读一读７.1节会更容易理解。

今天的内容所剩不多了，大家再加一把劲。鼠标是IRQ12，键盘是IRQ1，所以我们编写了用于INT 0x2c和INT 0x21的中断处理程序（handler），即中断发生时所要调用的程序。

int.c的节选

```
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();
    }
}
```

正如大家所见，这个函数只是显示一条信息，然后保持在待机状态。鼠标的程序也几乎完全一样，只是显示的信息不同而已。“只写鼠标程序不就行了吗，怎么键盘也写了呢？”，因为键盘与鼠标的处理方法很相像，所以顺便写了一下。inthandler21接收了esp指针的值，但函数中并没有用。在这里暂时不用esp，不必在意。

■■■■■

如果这样就能运行，那就太好了，可惜还不行。中断处理完成之后，不能执行“return;”（=RET指令），　而是必须执行IRETD指令，真不好办。而且，这个指令还不能用C语言写[2]。所以，还得借助汇编语言的力量修改naskfunc.nas。

[2]　对于我们今天这个程序来说，在中断处理程序中无限循环，IRETD指令得不到执行，所以怎么都行。之所以说“不能用C语言来写”，是为了今后。

本次的**naskfunc.nas**节选

```
        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
```

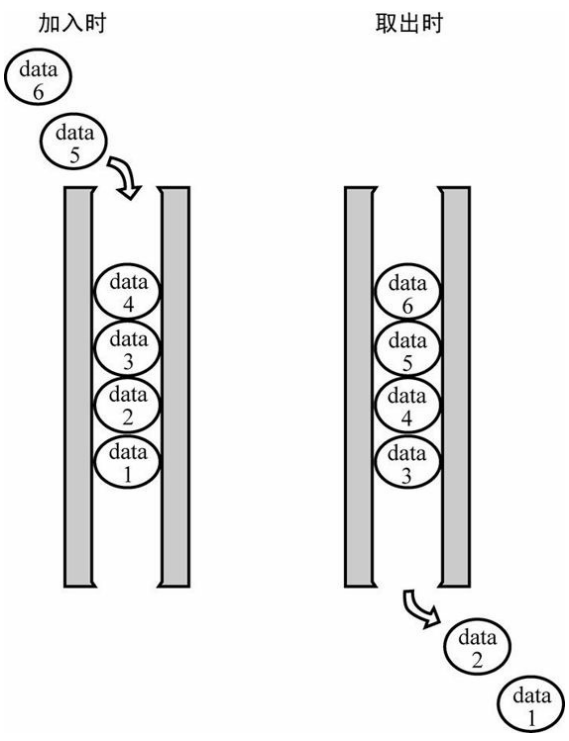
我们只解释键盘程序，因为鼠标程序和它是一样的。最后的IRETD刚才已经讲过了。最开头的EXTERN指令，在调用（CALL）的地方再进行说明。这样一来，问题就只剩下PUSH和POP了。

■■■■■

继续往下说明之前，我们要先好好解释一下栈（stack）的概念。

写程序的时候，经常会有这种需求——虽然不用永久记忆，但需要暂时记住某些东西以备后用。这种目的的记忆被称为缓冲区（buffer）。突然一下子接收到大量信息时，先把它们都保存在缓冲区里，然后再慢慢处理，缓冲区一词正是来源于这层意思。根据整理记忆内容的方式，缓冲区分为很多种类。

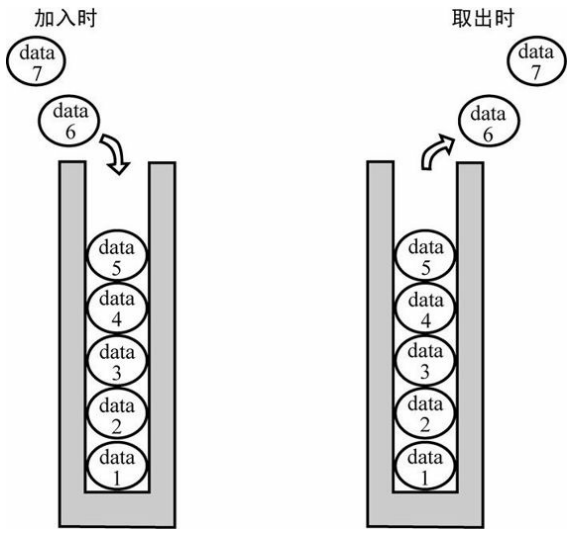
最简单明了的方式，就是将信息从上面逐渐加入进来，需要时再从下面一个个取出。



缓冲的种类（**1**）

最先加入的信息也最先取出，所以这种缓冲区是“先进先出”（first in, first out），简称FIFO。这应该是最普通的方式了。有的书中也会称之为“后进后出”（last in, last out），即LILO。叫法虽然不同，但实质上是同样的东西。

下面要介绍的一种方式，有点类似于往桌上放书，也就是信息逐渐从上面加入进来，而取出时也从最上面开始。



缓冲的种类（**2**）

最先加入的信息最后取出，所以这种缓冲区是“先进后出”（first in, last out），简称FILO。有的书上也称之为“后进先出”（last in, first out），即LIFO。

■■■■■

这里要说明的栈，正是FILO型的缓冲区。PUSH将数据压入栈顶，POP将数据从栈顶取出。PUSH EAX这个指令，相当于：

```
ADD ESP,-4
MOV [SS:ESP],EAX
```

也就是说，ESP的值减去4，以所得结果作为地址值，将寄存器中的值保存到该地址所对应内存里。反过来，POP EAX指令相当于：

```
MOV EAX,[SS:ESP]
ADD ESP,4
```

CPU并不懂栈的机制，它只是执行了实现栈功能的指令而已。所以，即使是PUSH太多，或者POP太多这种没有意义的操作，基本上CPU也都会遵照执行。

所以，如果写了以下程序，

```
PUSH EAX
PUSH ECX
PUSH EDX
各种处理
POP  EDX
POP  ECX
POP  EAX
```

在“各种处理”那里，即使把EAX，ECX，EDX改了，最后也还会恢复回原来的值……其实ES、DS这些寄存器，也就是靠PUSH和POP等操作而变回原来的值的。

■■■■■

还有一个不怎么常见的指令PUSHAD，它相当于：

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

反过来，POPAD指令相当于按以上相反的顺序，把它们全都POP出来。

■■■■■

结果，这个函数只是将寄存器的值保存到栈里，然后将DS和ES调整到与SS相等，再调用_inthandler21，返回以后，将所有寄存器的值再返回到原来的值，然后执行IRETD。内容就这些。如此小心翼翼地保存寄存器的值，其原因在于，中断处理发生在函数处理的途中，通过IREDT从中断处理返回以后，如果寄存器的值乱了，函数就无法正常处理下去了，所以一定要想尽办法让寄存器的值返回到中断处理前的状态。

关于在DS和ES中放入SS值的部分，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。所以，虽然麻烦了一点，但还是要这样做。

这么说来，CALL也是一个新出现的指令，它是调用函数的指令。这次要调用一个没有定义在naskfunc.nas中的函数，所以我们最初用一个EXTERN指令来通知nask：“马上要使用这个名字的标号了，它在别的源文件里，可不要搞错了”。

■■■■■

好了，这样_asm_inthandler21的讲解就没有问题了吧。下面要说明的，就是要将这个函数注册到IDT中去这一点。我们在dsctbl.c的init_gdtidt里加入以下语句。

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

asm_inthandler21注册在idt的第0x21号。这样，如果发生中断了，CPU就会自动调用asm_inthandler21。这里的2 * 8表示的是asm_inthandler21属于哪一个段，即段号是2，乘以8是因为低3位有着别的意思，这里低3位必须是0。

所以，“2 * 8” 也可以写成 “2<<3”， 当然，写成16也可以。

不过，号码为2的段，究竟是什么样的段呢？

```
set_segmdesc(gdt + 2, LIMIT_BOTPAK, ADR_BOTPAK, AR_CODE32_ER);
```

程序中有以上语句，说明这个段正好涵盖了整个bootpack.hrb。

最后的AR_INTGATE32将IDT的属性，设定为0x008e。它表示这是用于中断处理的

有效设定。

■■■■■

还有就是对bootpack.c的HariMain的补充。“io_sti();”仅仅是执行STI指令，它是CLI的逆指令。就是说，执行STI指令后，IF（interrupt flag，中断许可标志位）变为1，CPU接受来自外部设备的中断（参考4.6节）。CPU的中断信号只有一根，所以IF也只有一个，不像PIC那样有8位。

在HariMain的最后，修改了PIC的IMR，以便接受来自键盘和鼠标的中断。这样程序就完成了。只要按下键盘上某个键，或动一动鼠标，中断信号就会传到CPU，然后CPU执行中断处理程序，输出信息。

■■■■■

那好，我们运行一下试试看。“make　run”……然后按下键盘上的“A”……哦！显示了一行信息。

按下字母**A**之后

让我们先退出程序，再运行一次“make　run”吧。这次我们随便转转鼠标。但怎么让鼠标转起来呢？首先我们在QEMU画面的某个地方单击一下，这样就把鼠标与QEMU绑定在一起了，鼠标事件都会由QEMU接受并处理。然后我们上下左右移动鼠标，就会产生中断。哎？怎么没反 应呢？

哎？明明动了鼠标嘛？！

在这个状态下，我们不能对Windows进行操作，所以只好按下Ctr键再按Alt键，先把鼠标从QEMU中解放出来。然后点击“×”，关闭QEMU窗口。

虽然今天的结果还不能让人满意，但天色已经很晚了，就先到此为止吧。原因嘛，让我们来思考一夜。但不论怎么说，键盘的中断设定已经成功了，至于鼠标的问题，肯定也能很快找到原因的。我们明天再继续吧。