



第 4 天

C语言与画面显示的练习

- 用C语言实现内存写入 (harib01a)
- 条纹图案 (harib01b)
- 挑战指针 (harib01c)
- 指针应用 (1) (harib01d)
- 指针应用 (2) (harib01e)
- 色号设定 (harib01f)
- 绘制矩形 (harib01g)
- 今天的成果 (harib01h)

1 用 C 语言实现内存写入 (harib01a)

昨天我们成功地让画面显示黑屏了,但只做到这一步没什么意思,还是往画面上画点儿什么东西比较有趣。想要画东西的话,只要往VRAM里写点什么就可以了。但是在C语言中又没有直接写入指定内存地址的语句^①。嗯,真是不方便。所以,我们干脆就创建一个有这种功能的函数。下面就来修改一下naskfunc.nas。

naskfunc.nas里添加的部分

```
_write_mem8:      ; void write_mem8(int addr, int data);  
    MOV     ECX, [ESP+4]      ; [ESP + 4]中存放的是地址, 将其读入ECX  
    MOV     AL, [ESP+8]      ; [ESP + 8]中存放的是数据, 将其读入AL  
    MOV     [ECX], AL  
    RET
```

这个函数类似于C语言中的“write_mem8 (0x1234,0x56);”语句,动作上相当于“MOV BYTE[0x1234],0x56”。顺便说一下,addr是address的缩写,在这里用它来表示地址。

※※※※※

在C语言中如果用到了write_mem8函数,就会跳转到_write_mem8。此时参数指定的数字就

① “怎么会? 分明有啊!” 如果你有这样的疑问,那么作为本书的读者,你已经知道得相当多了。

存放在内存里，分别是：

第一个数字的存放地址：[ESP + 4]
 第二个数字的存放地址：[ESP + 8]
 第三个数字的存放地址：[ESP + 12]
 第四个数字的存放地址：[ESP + 16]
 (以下略)

我们想取得用参数指定的数字0x1234或0x56的内容，就用MOV指令读入寄存器。因为CPU已经是32位模式，所以我们积极使用32位寄存器。16位寄存器也不是不能用，但如果用了的话，不只机器语言的字节数会增加，而且执行速度也会变慢，没什么好处。

在指定内存地址的地方，如果使用16位寄存器指定[CX]或[SI]之类的就会出错，但使用32位寄存器，连[ECX]、[ESP]等都OK，基本上没有不能使用的寄存器。真方便。另外，在指定地址时，不光可以指定寄存器，还可以使用往寄存器加一个常数，或者减一个常数的方式。另外说一下，在16位模式下，也能使用这种方式指定，但那时候没有什么地方用得上，所以没有使用。

如果与C语言联合使用的话，有的寄存器能自由使用，有的寄存器不能自由使用，能自由使用的只有EAX、ECX、EDX这3个。至于其他寄存器，只能使用其值，而不能改变其值。因为这些寄存器在C语言编译后生成的机器语言中，用于记忆非常重要的值。因此这次我们只用EAX和ECX。

■■■■■■■■

这次还给naskfunc.nas增加了一行，那就是INSTRSET指令。它是用来告诉nask“这个程序是给486用的哦”，nask见了这一行之后就知道“哦，那见了EAX这个词，就解释成寄存器名”。如果什么都不指定，它就会认为那是为8086这种非常古老的、而且只有16位寄存器的CPU而写的程序，见了EAX这个词，会误解成标签(Label)，或是常数。8086那时候写的程序中，曾偶尔使用EAX来做标签，当时也没想到这个单词后来会成为寄存器名而不能再随便使用。

上面虽然写着486用，但并不是说会出现仅能在486中执行的机器语言，这只是单纯的词语解释的问题。所以486用的模式下，如果只使用16位寄存器，也能成为在8086中亦可执行的机器语言。“纸娃娃操作系统”也支持386，所以虽然这里指定的是486，但并不是386中就不能用。可能会有人问，这里的386，486都是什么意思啊？我们来简单介绍一下电脑的CPU（英特尔系列）家谱。

8086→80186→286→386→486→Pentium→PentiumPro→PentiumII→PentiumIII→Pentium4→...

从上面的家谱来看，386已经是非常古老的CPU了。到286为止CPU是16位，而386以后CPU是32位。

■■■■■■■■

现在，汇编这部分已经准备好了，下面来修改C语言吧。这次我们导入了变量。

本次的bootpack.c内容

```

void io_hlt(void);
void write_mem8(int addr, int data);

void HariMain(void)
{
    int i; /*变量声明: i是一个32位整数*/

    for (i = 0xa0000; i <= 0xfffff; i++) {
        write_mem8(i, 15); /* MOV BYTE [i],15 */
    }

    for (;;) {
        io_hlt();
    }
}

```

for语句是初次登场它是循环语句，会循环执行花括号（{}）括起来的部分。圆括号（()）中写的是循环执行的条件。共有3个条件，各个条件之间以分号（;）隔开，最初一个条件是初始值。所以上文第一个for语句中，把0xa0000赋值给i。任何for语句的初始值设定语句总是要执行，这是C语言的规定。

下一个部分“i <= 0xfffff”是循环条件。for语句会判断是否满足这个条件，如果不满足，就跳出“{}”括起来的循环体部分。另外，这个部分在第一次执行时就要判断，所以，有时候循环体部分有可能一次都得不到执行。不过这次的for语句中，最初的i值是0xa0000，满足条件，所以循环体部分能够被执行。

最后一个部分是“i++”，这是“i = i+1;”的省略形式，也就是i的值增加1。这个语句在循环体执行完以后肯定要执行一次，然后判断循环条件。

只看文字说明不易于理解，我们写成代码形式来辅助说明。

for (A; B; C){D; } 与以下程序等价

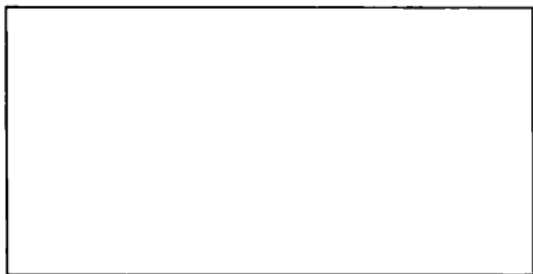
```

A;
label:
    if (B) {
        D;
        C;
        goto label;
    }

```

for语句的3个条件，全都可以省略。这种情况下，不做任何初值设定，循环条件永远成立，“{}”内的循环体部分执行完以后，不做任何处理。也就是单纯的无限循环。我们在“io_hlt();”处使用了这种循环。

下一步是运行“make run”还是“make install”呢？两个都可以，但不管执行哪个，画面都不是黑屏，而是白屏。哦？这是怎么回事呢？因为VRAM全部都写入了15，意思是全部像素的颜色都是第15种颜色，而第15种颜色碰巧是纯白，所以画面就成了白色。还是画面上有点什么变化才好。



太好了，成功了！但看不出来……

最初做成的时候，还挺高兴的，但在写这本书的时候，才发觉这是一次失败。纯白的截图放到书里还是一片白，什么都看不出来。

2 条纹图案 (harib01b)

所以，为了在印成书后能看出效果，我们就显示成有条纹的图案吧。修改也很简单，只要稍微改动一下bootpack.c就可以了。

```
for (i = 0xa0000; i <= 0xfffff; i++) {  
    write_mem8(i, i & 0x0f);  
}
```

哪儿变了呢？是write_mem8那里。地址部分虽然和之前一样，但写入的值由15变成了 i & 0x0f。

在这里&是“与”运算，是数学中没有的一种运算。很久以前，CPU就不仅能处理数值数据，还能处理图形数据。在处理图形数据的时候，加减乘除这种数学上的计算功能几乎没什么用。因为所处理的数据虽然是二进制数，但它们并不是作为数字来使用的，重点是0和1的排列方式，对于图形来说，这种排列方式本身更重要。

那么对于图形数据应该进行什么样的运算呢？可以将某些特定的位变为1，某些特定的位变为0，或者是反转^①特定的位等，做这样的运算。

※ 图形数据

① 反转指让0变为1、1变为0的操作，形象地说就好比照片的底片一样。

先来看看让特定位变成1的功能。这可以通过“或”(OR)运算来实现。

```
0100 OR 0010 → 0110
1010 OR 0010 → 1010
```

计算“AOR B”的时候,每一位分别计算,对于某一位,A和B的该位只要有一个是1,“或”运算的结果,该位就是1。否则(A和B的该位都是0)结果就是0。也就是说,如果某个图像数据放在变量i里,让i与0010进行或运算,1所在的那一位(从右往左第2位)就一定会变为1。对于其他的位则没有任何影响。如果i的该位(从右往左第2位)原本就是1,则i不变。

下面说说让特定位变成0的功能。这可以通过“与”(AND)运算来实现。

```
0100 AND 1101 → 0100
1010 AND 1101 → 1000
```

计算“AAND B”的时候,也是每一位分别计算,对于某一位,A和B的该位都是1的时候,“与”运算的结果,该位才是1,否则结果就是0。也就是说,如果某个图像数据放在变量i里,让i与1101进行“与”运算,则0所在的那一位(从右往左第2位)就一定会变为0。如果i的该位(从右往左第2位)原本就是0,则i不变。跟“或”运算不同,“与”运算中不想改变的部分要设为1,想改为0的部分要设为0(也就是说,一个是i与0010进行“或”运算,一个是i与1101进行“与”运算)。这一点需要注意。

最后我们来看让特定位反转的功能。这可以通过“异或”(XOR)运算来实现。

```
0100 XOR 0010 → 0110
1010 XOR 0010 → 1000
```

计算“A XOR B”的时候,同样也是每一位分别计算,对于某一位,A和B该位的值如果不相同,“异或”运算的结果,该位是1,否则就是0。也就是说,如果某个图像数据放在变量i里,让i与0010进行“异或”运算,就可以对该位进行反转,而别的位不受影响。如果i与所有位都是1(即0xffffffff)的数进行“异或”,则全部位都反转。

原图

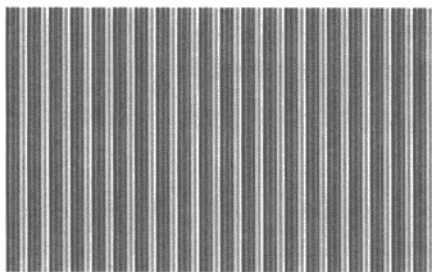
这次我们用的是“与”(AND)运算。将地址值与0x0f进行“与”运算会怎么样呢?低4位原封保留,而高4位全部都变成0。所以,写入的值是:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 00 01 02 03 04 05 06 ...
```

就像这样,每隔16个像素,色号就反复一次。会出现什么效果呢?运行一下“make run”就知道了。

出现了下面这种条纹图案。

印成书之后也能看得很清楚,成功啦!



执行后的结果

3 挑战指针 (harib01c)

前面说过“C语言中没有直接写入指定内存地址的语句”，实际上这不是C语言的缺陷，因为有替代这种命令的语句。一般大多数程序员主要使用那种替代语句，像这次这样，做一个函数 `write_mem8` 的，也就只有笔者了。如果有替代方案的话，大家肯定想用一下，笔者也想试试看。

```
write_mem3(i, i & 0x0f);
```

替代以上语句的是：

```
*i = i & 0x0f;
```

两个语句有点像，但又不尽相同。不管那么多了，先换成后面一种写法看看吧。好了，改完了，用“make run”命令运行一下。唉？奇怪，怎么会出错呢？

```
invalid type argument of `unary *'
```

类型错误？

调试器截图

没错，就是类型错误。这种写法，从本质上讲没问题，但这样就是无法顺利运行。我们从编译器的角度稍微想想就能明白为什么会出错了。回想一下，如果写以下汇编语句，会发生什么情况呢？

```
MOV [ 0x1234], 0x56
```

是的，会出错。这是因为指定内存时，不知道到底是BYTE，还是WORD，还是DWORD。只有在另一方也是寄存器的时候才能省略，其他情况都不能省略。

其实C编译器也面临着同样的问题。这次，我们费劲写了一条C语句，它的编译结果相当于下面的汇编语句所生成的机器语言，

```
MOV [i], (i & 0x0f)
```

但却不知道[i]到底是BYTE，还是WORD，还是DWORD。刚才就是出现了这种错误。

那怎么才能告诉计算机这是BYTE呢?

```
char *p;    /*, 变量p是用于内存地址的专用变量*/
```

声明一个上面这样变量p, p里放入与i相同的值, 然后执行以下语句。

```
*p = i & 0x0f;
```

这样, C编译器就会认为“p 是地址专用变量, 而且是用于存放字符(char)的, 所以就是BYTE。”顺便解释一下类似语句:

```
char *p;    /*用于BYTE类地址*/
```

```
short *p;   /*用于WORD类地址*/
```

```
int *p;     /*用于DWORD类地址*/
```

这次我们是一个字节一个字节地写入, 所以使用了char。

既然说到这里, 那我们再介绍点相关知识, “char i;”是类似AL的1字节变量, “short i;”是类似AX的2字节变量, “int i;”是类似EAX的4字节变量。

而不管是“char *p”, 还是“short *p”, 还是“int *p”, 变量p都是4字节。这是因为p是用于记录地址的变量。在汇编语言中, 地址也像ECX一样, 用4字节的寄存器来指定, 所以也是4字节。

编译并运行

这样准备工作就OK了。再用“make run”运行一遍以下内容。

```
void HariMain(void)
{
    int i; /*变量声明。变量i是32位整数*/
    char *p; /*变量p, 用于BYTE型地址*/

    for (i = 0xa0000; i <= 0xfffff; i++) {
        p = i; /*代入地址*/
        *p = i & 0x0f;

        /*这可以替代write_mem8(i, i & 0x0f);*/
    }

    for (;;) {
        io_hlt();
    }
}
```

哇, 居然不使用write_mem8就能显示出条纹图案, 真是太好了。

嗯? 且慢! 仔细看看画面, 发现有一行警告。

warning: assignment makes pointer from integer without a cast

这个警告的意思是说,“赋值语句没有经过类型转换,由整数生成了指针”。其中有两个单词的意思不太明白。类型转换是什么?指针又是什么?

类型转换是改变数值类型的命令。一般不必每次都注意类型转换,但像这次的语句中,如果不明确进行类型转换,C编译器就会每次都发出警告:“喂,是不是写错了?”顺便说一下,cast在英文中的原意是压入模具,让材料成为某种特定的形状。

指针是表示内存地址的数值。C语言中不用“内存地址”这个词,而是用“指针”。在C语言中,普通数值和表示内存地址的数值被认为是两种不同的东西,虽然笔者也觉得它们没什么不同,但也只能接受这种设计思想了。基于这种设计思想,如果将普通整数赋值给内存地址变量,就会有警告。为了避免这种情况的发生,可以这样写:

```
p = (char *) i;
```

这就对i进行了类型转换,使之成为表示内存地址的整数。(其实这样转换以后,数值一点都没变,但对于C编译器来说,类型的不同有着很大的差别。)以后再进行这样的赋值时,就不会出现这种讨厌的警告了。于是我们这样修改一下。

再运行一次“make run”吧。好了,不再出现那种烦人的警告了。write_mem8已经没用了,所以可以将它从naskfunc.nas中删除。

这样的写法虽然有点绕圈子了,但我们实现了只用C语言写入内存的功能。

COLUMN-2

只要使用类型转换,就可以不用指针之类的方法吗?

好不容易介绍完了类型转换,我们来看一个应用实例吧。如果定义:

```
p = (char *) i;
```

那么将上式代入下面语句中。

```
*p = i & 0x0f;
```

这样就能得到下式:

```
*((char *) i) = i & 0x0f;
```

这个语句执行起来毫无问题。虽然读起来不是很容易理解,但这样可以不特意声明p变量,所以笔者偶尔还是会使用的。

有没有觉得这种写法与“BYTE[i] = i & 0x0f;”有些相像吗?在特别喜欢汇编语言的笔者看来,会有这种感觉呢。(笑)

COLUMN-3

还是不能理解指针

能有这种想法,说明你很诚实。那好,我们再尽量详细地讲解一下。

如果你曾经使用过C语言,并且听说过“指针”这个词,那么刚才的说明肯定让你觉得混乱,摸不着头脑。倒是那些从未接触过C语言的人更能理解一些。

这里,特别重要的一点是,必须想点办法让C语言完成以下功能:

```
MOV BYTE [i], (i & 0x0f)
```

也就是,向内存的第*i*号地址写入*i* & 0x0f的计算结果。而程序只是偶然地写成了:

```
int i;
char *p;

p = (char *) i;
*p = i & 0x0f;
```

必须要先理解以上程序。这可能与你所知道的指针的使用方法完全不同,不过暂时先不要想这个。总之上面4行,是MOV语句的替代物,这一点是最重要的。

从没听说过C语言指针的人,仅仅会想“哦,原来C语言中是这么写的,没那么复杂么。”的确如此,没什么不懂的嘛。

图形界面

下面再稍微深入说明一下。我们常见的两个语句是:

```
p = (char *) i;
*p = i & 0x0f;
```

这两个语句有什么区别呢?这是不懂汇编的人常有的疑问。将以上语句按汇编的习惯写一下吧。假设*p*相当于ECX,那么写出来就是:

```
MOV ECX, i
MOV BYTE [ECX], (i & 0x0f)
```

它们的区别很清楚,即一个是给ECX寄存器赋值,一个给ECX号内存地址赋值。这完全是两回事。存储它们的半导体也不一样,一个在CPU里,一个在内存芯片里。在C语言中,虽然*p*与**p*只有一字之差,但意思上的差别却如此之大。

如果执行顺序调过来会怎么样呢?也就是像这样:

```
*p = i & 0x0f;
p = (char *) i;
```

不是很熟悉指针的人可能认为这样也行。但是,这相当于:

```
MOV BYTE [ECX], (i & 0x0f)
MOV ECX, i
```

如果这么做,第一个MOV的时候,ECX的值不确定,是个随机数,这会导致*i* & 0x0f的

结果写入内存的某个不可知的地址中。这样的后果很严重。

—————

另一个比较常见的疑问，是关于声明的。在 C 语言中，如果不声明变量就不能使用。所谓声明，就是类似“int i;”这种语句。有了这句话，变量 i 就可以使用了（与此不同的是汇编语言中，EAX, DL 等，不声明也可以自由使用）。在 C 语言中，声明了 10 个变量，就可以用 10 个变量，这是理所当然的事。

既然如此，那为什么只声明了“char *p;”却不仅能使用 p，还可以使用 *p 呢？这让人搞不懂……确实，这个程序中，给 p 和 *p 赋值了。看上去，能够使用的变量数比实际声明的变量数要多。

遇到这种情况时，我们先回到汇编语言中看看。

```
MOV ECX, i
MOV BYTE [ECX], (i & 0x0f)
```

看着这个程序，就不会再有人认为其中有 2 个变量了。其中只有一个 ECX。而且，同样是“MOV AL, [ECX]”，ECX 是 123 的时候，和 ECX 是 124 的时候，放入 AL 的值也是不同的（只要这两处地址存放的不是同样的值）。这是因为地址不同，代表的内存区域不同。就好比不同的住址，住的人也不一样。

所以，同样是 *p，因为 p 值的不同，记录的值也不同。

```
*p = 3;
p = p + 3;
i = *p;
```

也就是说如果执行以上片段，i 不一定是 3，因为地址已经变了。

费了半天劲，其实笔者想说的就是，*p 并不是什么变量。确实，我们可以给 *p 赋值，也可以引用 *p 的值，这看起来就像变量一样。但即便如此，*p 也不是一个变量，变量只有 p。所谓 *p，就相当于汇编中 BYTE [p] 这种语句的代替。

如果你还执拗地说 *p 是一个变量，那照这种逻辑，变量可远不止 2 个，还有很多很多。因为只要给 p 赋上不同的值，*p 就代表完全不同区域的内存内容。

—————

下一个问题也是关于声明的：“char *p;”声明的是 *p，还是 p 呢？

这也是一个常见的问题。先给出结论吧，声明的是 p。“既然如此，那为什么不写成 char *p; 呢？”有这种想法，说明你这方面的直觉很好。笔者也认为这样写对于初学者来说更简单易懂。事实上，在 C 语言中写成“char *p;”也可以，既不出错，也不出警告，运行也没问题。

但这种写法有点小问题。如果写成“char *p,q;”，我们看上去会觉得 p 和 q 都表示地址

的变量,但C编译器却不那样认为,q会被看作是一般的1字节的变量。也就是被解释成“char *p,q”。为了避免这样的误解,一般的程序员不写成“char* p;”,所以笔者也按照这个习惯编写程序。另外,如果想要声明两个地址变量,就写成“char *q,*p;”。

继续看代码

今天的专栏写得好长呀,我们来整理总结一下吧。首先,本书中出现的“char *p;”不必看作指针,这是最重要的诀窍。p不是指针,而是地址变量。不要使用“p是指针”这种模棱两可的说法,“p是地址变量”这种说法比较好。

将地址值赋给地址变量是理所当然的。并且,既然地址代表的是内存的地址,可以让该地址存放自己想放的任何值。虽然也可以将地址变量说成是指针,但笔者听到指针这个说法也很茫然,所以除了跟别人讨论时以外,笔者也不说指针什么的。

C语言中地址变量的声明,以及给内存地址赋值的,写法不是很习惯,但终究这只是写法的不同,思考问题的方法与汇编语言差不多。在C语言开发人员看来,“C语言的*p比汇编语言 BYTE[p],更短小精悍”,确实,简洁是一个长处,但就是因为简洁,才让初学者不好理解。

C语言的很多初学者都在学习指针时受挫,以至于会想“如果没有指针就好了”。而事实上,没有指针的语言也确实存在的。但这种语言很不好用,因为没有指针就无法往指定内存的地址存入数据,那怎么往VRAM上绘制图像呢?这种语言只能让写操作系统变得更困难。

笔者也认为,C语言指针的语法很难理解,所以希望能改善。但它像汇编语言一样,能直接访问地址,这一点非常好。所以希望大家能这样想:“不是要废除指针,而是把指针改善得更直观易懂。”

4 指针的应用(1)(harib01d)

绘制条纹图案的部分,也可以写成以下这样:

```
p = (char *) 0xa0000; /*给地址变量赋值*/

for (i = 0; i <= 0xffff; i++) {
    *(p + i) = i & 0x0f;
}
```

本质上讲,所做的事跟之前一样。这里只是想说明,C语言还能用这种方法书写。

5 指针的应用(2)(harib01e)

C语言中,*(p+i)还可以改写成p[i]这种形式,所以上片段也可以写成这样:

```
p = (char *) 0xa0000; /*将地址赋值进去*/  
  
for (i = 0; i <= 0xffff; i++) {  
    p[i] = i & 0x0f;  
}
```

其实要做的事还是没有什么变化, 这里想要告诉大家各种写法, 今后可以根据自己的喜好区别使用。

COLUMN-4 p[i]是数组吗?

写得不好的 C 语言教科书里, 往往会说 $p[i]$ 是数组 p 的第 i 个元素。这虽然也不算错, 但终究有些敷衍。如果读者不懂汇编语言, 这种敷衍的说法是最省事的。

$p[i]$ 与 $*(p+i)$ 意思完全相同。要是嫌后者太长太麻烦, 或者是为了看起来好看就会使用这种写法。在这个例子里, $*(p+i)$ 是 6 个字符, 而 $p[i]$ 只有 4 个字符。区别只有这一点, 所以大家可以根据喜好使用。 $p[i]$ 不过是一个看起来像数列的使用了地址变量的省略写法而已。

反过来说, 也可以将 $p[0]$ 写成 $*p$, 写成指针的形式反倒是节省了 2 个字符。总之, 根据情况, 按自己喜欢的方式写就行了。

不是说改变一下写法, 地址变量就变成数组了。大家不要被那些劣质的教科书骗了。编译器生成的机器语言也完全一样。这比什么都更能证明, 意思没有变化, 只是写法不同。

说个题外话, 加法运算可以交换顺序, 所以将 $*(p+i)$ 写成 $*(i+p)$ 也是可以的。同理, 将 $p[i]$ 写成 $i[p]$ 也是可以的 (可能你会不相信, 但这样写既不会出错, 也能正常运行)。 $a[2]$ 也可以写成 $2[a]$ (这当然是真的)。难道还能说这是名为 2 的数组的第 a 个元素吗? 当然不能。所以, $p[i]$ 也好, $i[p]$ 也好, 仅仅是一种省略写法, 本质上讲, 与数组没有关系。

4

6 色号设定 (harib01f)

好了, 到现在为止我们的话题都是以 C 语言为中心的, 但我们的目的不是为了掌握 C 语言, 而是为了制作操作系统, 操作系统中是不需要条纹图案之类的。我们继续来做操作系统吧。

可能大家马上就想描绘一个操作系统模样的画面, 但在此之前要先做一件事, 那就是处理颜色问题。这次使用的是 320×200 的 8 位颜色模式, 色号使用 8 位 (二进制) 数, 也就是只能使用 0 ~ 255 的数。我想熟悉电脑颜色的人都会知道, 这是非常少的。一般说起指定颜色, 都是用 #ffffff 一类的数。这就是 RGB (红绿蓝) 方式, 用 6 位十六进制数, 也就是 24 位 (二进制) 来指定颜色。8 位数完全不够。那么, 该怎么指定 #ffffff 方式的颜色呢?

这个 8 位彩色模式, 是由程序员随意指定 0 ~ 255 的数字所对应的颜色的。比如说 25 号颜色对应 #ffffff, 26 号颜色对应 #123456 等。这种方式就叫做调色板 (palette)。

如果像现在这样, 程序员不做任何设定, 0 号颜色就是 #000000, 15 号颜色就是 #ffffff。其他

号码的颜色, 笔者也不是很清楚, 所以可以按照自己的喜好来设定并使用。

笔者通过制作OSAKA知道: 要想描绘一个操作系统模样的画面, 只要有以下这16种颜色就足够了。所以这次我们也使用这16种颜色, 并给它们编上号码0-15。

#000000: 黑	#00ffff: 浅亮蓝	#000084: 暗蓝
#ff0000: 亮红	#ffffff: 白	#840084: 暗紫
#00ff00: 亮绿	#c6c6c6: 亮灰	#008484: 浅暗蓝
#ffff00: 亮黄	#840000: 暗红	#848484: 暗灰
#0000ff: 亮蓝	#008400: 暗绿	
#ff00ff: 亮紫	#848400: 暗黄	

所以我们要给bootpack.c添加很多代码。

源程序清单

本次的bootpack.c

```
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);

/*就算写在同一个源文件里, 如果想在定义前使用, 还是必须事先声明一下。*/

void init_palette(void);
void set_palette(int start, int end, unsigned char *rgb);

void HariMain(void)
{
    int i; /* 声明变量。变量i是32位整数型 */
    char *p; /* 变量p是BYTE [...]用的地址 */

    init_palette(); /* 设定调色板 */

    p = (char *) 0xa0000; /* 指定地址 */

    for (i = 0; i <= 0xffff; i++) {
        p[i] = i & 0x0f;
    }

    for (;;) {
        io_hlt();
    }
}

void init_palette(void)
{
    static unsigned char table_rgb[16 * 3] = {
        0x00, 0x00, 0x00, /* 0:黑 */
```

```

    0xff, 0x00, 0x00, /* 1:亮红 */
    0x00, 0xff, 0x00, /* 2:亮绿 */
    0xff, 0xff, 0x00, /* 3:亮黄 */
    0x00, 0x00, 0xff, /* 4:亮蓝 */
    0xff, 0x00, 0xff, /* 5:亮紫 */
    0x00, 0xff, 0xff, /* 6:浅亮蓝 */
    0xff, 0xff, 0xff, /* 7:白 */
    0xc6, 0xc6, 0xc6, /* 8:亮灰 */
    0x84, 0x00, 0x00, /* 9:暗红 */
    0x00, 0x84, 0x00, /* 10:暗绿 */
    0x84, 0x84, 0x00, /* 11:暗黄 */
    0x00, 0x00, 0x84, /* 12:暗青 */
    0x84, 0x00, 0x84, /* 13:暗紫 */
    0x00, 0x84, 0x84, /* 14:浅暗蓝 */
    0x84, 0x84, 0x84 /* 15:暗灰 */
};
set_palette(0, 15, table_rgb);
return;

/* C语言中的static char语句只能用于数据, 相当于汇编中的DB指令 */
}

void set_palette(int start, int end, unsigned char *rgb)
{
    int i, eflags;
    eflags = io_load_eflags(); /* 记录中断许可标志的值 */
    io_cli(); /* 将中断许可标志置为0, 禁止中断 */
    io_out8(0x03c8, start);
    for (i = start; i <= end; i++) {
        io_out8(0x03c9, rgb[0] / 4);
        io_out8(0x03c9, rgb[1] / 4);
        io_out8(0x03c9, rgb[2] / 4);
        rgb += 3;
    }
    io_store_eflags(eflags); /* 复原中断许可标志 */
    return;
}

```

程序的头部罗列了很多的外部函数名, 这些函数必须在naskfunc.nas中写。这有点麻烦, 但也无法。先跳过这一部分, 我们来看看主函数HariMain。函数里只是增加了一行调用调色板设置的函数, 变更并不是太大。我们接着往下看。

图例说明

函数init_palette开头一段以static开始的语句, 虽然很长, 但结果无非就是声明了一个常数table_rgb。它太长了, 有些晦涩难懂, 所以我们来简化一下。

```

void init_palette(void)
{
    table_rgb的声明;
    set_palette(0, 15, table_rgb);
}

```

```
return;
}
```

简而言之, 就是这些内容。除了声明之外没什么难点, 所以我们仅仅解说声明部分。

```
char a[3];
```

C语言中, 如果这样写, 那么a就成为了常数, 以汇编的语言来讲就是标志符。标志符的值当然就意味着地址。并且还准备了“RESB 3”。总结一下, 上面的叙述就相当于汇编里的这个语句:

```
a:
    RESB 3
```

nasm中RESB的内容能够保证是0, 但C语言中不能保证所以里面说不定含有某种垃圾数据。

记录符号表

另外, 在这个声明的后面加上“={ ... }”, 还可以写上数据的初始值。比如:

```
char a[3]= { 1,2,3 };
```

这与下面的内容基本等价。

```
char a[3];
a[0] = 1;
a[1] = 2;
a[2] = 3;
```

这里, a是表示最初地址的数字, 也就是说它被认为是指针。

那么这次, 应该代入的值共有 $16 \times 3 = 48$ 个。笔者不希望大家做如此多的赋值语句。每次赋值都至少要消耗3个字节, 这样算下来光这些赋值语句就要花费将近150字节, 这太不值了。

其实写成下面这样一般的DB形式, 不就挺好吗。

```
table_rgb:
    DB 0x00, 0x00, 0x00, 0xff, 0x00, 0x00, 0x00, 0xff, 0x00, ...
```

只要48字节就够了。所以说, 就像在汇编语言中用DB指令代替RESB指令那样, 在C语言中也有类似的指示方法, 那就是在声明时加上static。这次我们也加上它。

下面来看unsigned。它的意思是: 这里所处理的数据是BYTE(char)型, 但它是没有符号(sign)的数(0或者正整数)。

char型的变量有3种模式, 分别是signed型、unsigned型和未指定型。signed型用于处理-128~127的整数。它虽然也能处理负数, 扩大了处理范围, 很方便, 但能够处理的最大值却减小了一半。unsigned型能够处理0~255的整数。未指定型是指没有特别指定时, 可由编译器决定是unsigned还是signed。



在这个程序里，多次出现了0xff这个数值，也就是255，我们想用它来表示最大亮度，如果它被误解成负数（0xff会被误解成-1）就麻烦了。虽然我们不清楚亮度比0还弱会是什么概念，但无论如何不能产生这种误解。所以我们决定将这个数设定为unsigned。顺便提一句，int和short也分signed和unsigned。……好了，关于init_palette的说明就到此为止。

详细源代码

下面要讲的是C语言说明部分最后的函数set_palette。这个函数虽然很短，干的事儿可不少。首先让我们仔细看看以下精简之后的记述吧。

```
void set_palette(int start, int end, unsigned char *rgb)
{
    int i;
    io_out8(0x03c8, start);
    for (i = start; i <= end; i++) {
        io_out8(0x03c9, rgb[0] / 4);
        io_out8(0x03c9, rgb[1] / 4);
        io_out8(0x03c9, rgb[2] / 4);
        rgb += 3;
    }
    return;
}
```

4

程序被如此精简后还可以正确运行。其实可以在一开始就介绍这个程序，但由于想给大家介绍精简之前的正确方法，所以才写了那么长。这个先放一边，我们来说说精简的程序吧。

这个程序所做的事情，仅仅是多次调用io_out8。函数io_out8是干什么的呢？以后在naskfunc.nas中还要详细说明，现在大家只要知道它是往指定装置里传送数据的函数就行了。

详细源代码

我们前面已经说过，CPU的管脚与内存相连。如果仅仅是与内存相连，CPU就只能完成计算和存储的功能。但实际上，CPU还要对键盘的输入有响应，要通过网卡从网络取得信息，通过声卡发送音乐数据，向软盘写入信息等。这些都是设备（device），它们当然也都要连接到CPU上。

既然CPU与设备相连，那么就有向这些设备发送电信号，或者从这些设备取得信息的指令。向设备发送电信号的是OUT指令；从设备取得电气信号的是IN指令。正如为了区别不同的内存要使用内存地址一样，在OUT指令和IN指令中，为了区别不同的设备，也要使用设备号码。设备号码在英文中称为port（端口）。port原意为“港口”，这里形象地将CPU与各个设备交换电信号的行为比作了船舶的出港和进港。

所以，我们执行OUT指令时，出港信号就要挥泪告别CPU了。这就好像它在说：“妈妈，我要走了。我在显卡中，会很好的，不用担心。”我想不用说大家也会感觉得到，在C语言中，没有与IN或OUT指令相当的语句，所以我们只好拿汇编语言来做了。唉，汇编真是关键时刻显身手

的语言呀。

画面编辑器

如果我们读一读程序的话，就会发现突然蹦出了0x03c8、0x03c9之类的设备号码，这些设备号码到底是如何获得的呢？随意写几个数字行不行呢？这些号码当然不是能随便乱写的。否则，别的什么设备胡乱动作一下，会带来很严重的问题。所以事先必须仔细调查。笔者的参考网页如下：

<http://community.osdev.info/?VGA>

网页的叙述太长了，不好意思（注：这一页也是笔者写的）。网页正中间那里，有一个项目，叫做“video DA converter”，其中有以下记述。

- 调色板的访问步骤。
- 首先在一连串的访问中屏蔽中断（比如CLI）。
- 将想要设定的调色板号码写入0x03c8，紧接着，按R，G，B的顺序写入0x03c9。如果还想继续设定下一个调色板，则省略调色板号码，再按照RGB的顺序写入0x03c9就行了。
- 如果想要读出当前调色板的状态，首先要将调色板的号码写入0x03c7，再从0x03c9读取3次。读出的顺序就是R，G，B。如果要继续读出下一个调色板，同样也是省略调色板号码的设定，按RGB的顺序读出。
- 如果最初执行了CLI，那么最后要执行STI。

我们的程序在很大程度上参考了以上内容。

画面编辑器

到这里，该说明的部分都说明得差不多了。总结一下就是：

```
void set_palette(int start, int end, unsigned char *rgb)
{
    int i, eflags;
    eflags = io_load_eflags(); /* 记录中断许可标志的值 */
    io_cli();                 /* 将许可标志置为0，禁止中断 */
    已经说明的部分
    io_store_eflags(eflags); /* 恢复许可标志的值 */
    return;
}
```

在“调色板的访问步骤”的记述中，还写着CLI、STI什么的。下面来看看它们可以做什么。

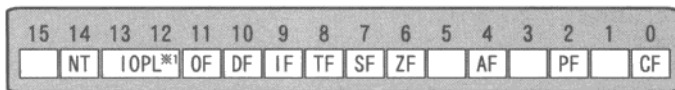
首先是CLI和STI。所谓CLI，是将中断标志（interrupt flag）置为0的指令（clear interrupt flag）。STI是要将这个中断标志置为1的指令（set interrupt flag）。而标志，是指像以前曾出现过的进位标志一样的各种标志，也就是说在CPU中有多种多样的标志。更改中断标志有什么好处呢？正如其名所示，它与CPU的中断处理有关系。当CPU遇到中断请求时，是立即处理中断请求（中断标志

为1), 还是忽略中断请求 (中断标志为0), 就由这个中断标志位来设定。

那到底什么是中断呢? 大家可能会有这种疑问, 可如果现在来讲这个问题的话, 就与我们“描绘一个操作系统模样的画面”这个主题渐行渐远了, 所以等以后有机会再讲吧。

插图 4-1

下面再来介绍一下EFLAGS这一特别的寄存器。这是由名为FLAGS的16位寄存器扩展而来的32位寄存器。FLAGS是存储进位标志和中断标志等标志的寄存器。进位标志可以通过JC或JNC等跳转指令来简单地判断到底是0还是1。但对于中断标志, 没有类似的JI或JNI命令, 所以只能读入EFLAGS, 再检查第9位是0还是1。顺便说一下, 进位标志是EFLAGS的第0位。



※1 IOPL将第13, 第12位这两位放在一起处理

空白位没有特殊意义 (或许留给将来的CPU用?)

set_palette中想要做的事情是在设定调色板之前首先执行CLI, 但处理结束以后一定要恢复中断标志, 因此需要记住最开始的中断标志是什么。所以我们制作了一个函数io_load_eflags, 读取最初的eflags值。处理结束以后, 可以先看看eflags的内容, 再决定是否执行STI, 但仔细想一想, 也没必要搞得那么复杂, 干脆将eflags的值代入EFLAGS, 中断标志位就恢复为原来的值了。函数o_store_eflags就是完成这个处理的。

估计不说大家也知道了, CLI也好, STI也好, EFLAGS的读取也好, EFLAGS的写入也好, 都不能用C语言来完成。所以我们就努力一下, 用汇编语言来写吧。

插图 4-2

我们已经解释了bootpack.c程序, 那么现在来说说naskfunc.nas。

```
; naskfunc
; TAB=4

[FORMAT "wcoff"]           ; 制作目标文件的模式
[INSTRSET "i486p"]         ; 使用到486为止的指令
[BITS 32]                  ; 制作32位模式用的机器语言
[FILE "naskfunc.nas"]      ; 源程序文件名

GLOBAL _io_hlt, _io_cli, _io_sti, _io_stihlt
GLOBAL _io_in8, _io_in16, _io_in32
GLOBAL _io_out8, _io_out16, _io_out32
GLOBAL _io_load_eflags, _io_store_eflags

[SECTION .text]

_io_hlt:    ; void io_hlt(void);
```

82 第4天: C语言与画面显示的练习

```
    HLT
    RET
_io_cli:    ; void io_cli(void);
    CLI
    RET

_io_sti:    ; void io_sti(void);
    STI
    RET

_io_stihlt: ; void io_stihlt(void);
    STI
    HLT
    RET

_io_in8:    ; int io_in8(int port);
    MOV     EDX,[ESP+4]    ; port
    MOV     EAX,0
    IN      AL,DX
    RET

_io_in16:   ; int io_in16(int port);
    MOV     EDX,[ESP+4]    ; port
    MOV     EAX,0
    IN      AX,DX
    RET

_io_in32:   ; int io_in32(int port);
    MOV     EDX,[ESP+4]    ; port
    IN      EAX,DX
    RET

_io_out8:   ; void io_out8(int port, int data);
    MOV     EDX,[ESP+4]    ; port
    MOV     AL,[ESP+8]     ; data
    OUT     DX,AL
    RET

_io_out16:  ; void io_out16(int port, int data);
    MOV     EDX,[ESP+4]    ; port
    MOV     EAX,[ESP+8]    ; data
    OUT     DX,AX
    RET

_io_out32:  ; void io_out32(int port, int data);
    MOV     EDX,[ESP+4]    ; port
    MOV     EAX,[ESP+8]    ; data
    OUT     DX,EAX
    RET

_io_load_eflags: ; int io_load_eflags(void);
    PUSHFD    ; 指 PUSH EFLAGS
    POP      EAX
    RET

_io_store_eflags: ; void io_store_eflags(int eflags);
    MOV      EAX,[ESP+4]
```

```

PUSH    EAX
POPFD   ; 指 POP EFLAGS
RET

```

到现在为止的说明，想必大家都已经懂了，尚且需要说明的只有与EFLAGS相关的部分了。如果有“MOV EAX,EFLAGS”之类的指令就简单了，但CPU没有这种指令。能够用来读写EFLAGS的，只有PUSHFD和POPFD指令。

※※※※※

PUSHFD是“push flags double-word”的缩写，意思是将标志位的值按双字长压入栈。其实它所做的，无非就是“PUSH EFLAGS”。POPFD是“pop flags double-word”的缩写，意思是按双字长将标志位从栈弹出。它所做的，就是“POP EFLAGS”。

栈是数据结构的一种，大家暂时只要理解到这个程度就够了。往栈登录数据的动作称为push（推），请想象一下往烤箱里放面包的情景。从栈里取出数据的动作称为pop（弹出）。

也就是说，“PUSHFD POP EAX”，是指首先将EFLAGS压入栈，再将弹出的值代入EAX。所以说它代替了“MOV EAX,EFLAGS”。另一方面，PUSH EAX POPFD正与此相反，它相当于“MOV EFLAGS,EAX”。

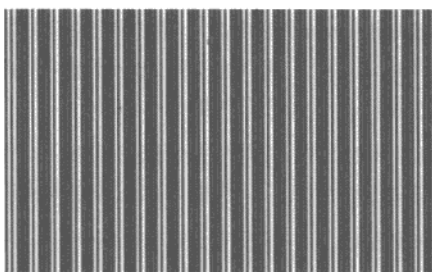


※※※※※

最后要讲的是`io_load_eflags`。它对我们而言，是第一个有返回值的函数的例子，但根据C语言的规约，执行`RET`语句时，`EAX`中的值就被看作是函数的返回值，所以这样就可以。

另外，虽然还有几个函数是不必要的，但因为将来会用到，所以这里就顺便做了。虽然不知道什么时候用，用于什么目的，但通过到目前为止的讲解也能明白其中的意义。

好了，讲解完了以后执行一下吧。运行“`make run`”。条纹的图案没有变化，但颜色变了！成功了！



仔细看看，颜色可不一样哟！

7 绘制矩形（harib01g）

颜色备齐了，下面我们来画“画”吧。首先从VRAM与画面上的“点”的关系开始说起。在当前画面模式中，画面上有 $320 \times 200 (=64\,000)$ 个像素。假设左上点的坐标是 $(0,0)$ ，右下点的坐标是 $(319,199)$ ，那么像素坐标 (x,y) 对应的VRAM地址应按下式计算。

$$0xa0000 + x + y * 320$$

其他画面模式也基本相同，只是`0xa0000`这个起始地址和`y`的系数`320`有些不同。

根据上式计算像素的地址，往该地址的内存里存放某种颜色的号码，那么画面上该像素的位置就出现相应的颜色。这样就画出了一个点。继续增加`x`的值，循环以上操作，就能画一条长长的水平直线。再向下循环这条直线，就能够画很多的直线，组成一个有填充色的长方形。

根据这种思路，我们制作了函数`boxfill8`。源程序就是`bootpack.c`。并且在程序`HariMain`中，我们不再画条纹图案，而是使用这个函数3次，画3个矩形。也不知能不能正常运行，我们来“`make run`”看看。哦，好像成功了。

本次的bootpack.c节选

```
#define COL8_000000    0
#define COL8_FF0000    1
#define COL8_00FF00    2
#define COL8_FFFF00    3
```

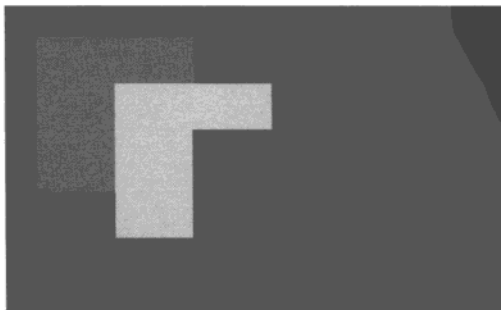
```

#define COL8_0000FF      4
#define COL8_FF00FF      5
#define COL8_00FFFF      6
#define COL8_FFFFFFFF    7
#define COL8_C6C6C6      8
#define COL8_840000      9
#define COL8_008400     10
#define COL8_848400     11
#define COL8_000084     12
#define COL8_840084     13
#define COL8_008484     14
#define COL8_848484     15

void HariMain(void)
{
    char *p; /* p变量的地址 */
    init_palette(); /* 设置调色板 */
    p = (char *) 0xa0000; /* 将地址赋值进去 */
    boxfill18(p, 320, COL8_FF0000, 20, 20, 120, 120);
    boxfill18(p, 320, COL8_00FF00, 70, 50, 170, 150);
    boxfill18(p, 320, COL8_0000FF, 120, 80, 220, 180);
    for (;;) {
        io_hlt();
    }
}

void boxfill18(unsigned char *vram, int xsize, unsigned char c, int x0, int y0, int x1, int y1)
{
    int x, y;
    for (y = y0; y <= y1; y++) {
        for (x = x0; x <= x1; x++)
            vram[y * xsize + x] = c;
    }
    return;
}

```



画了3个矩形哦

这次新出现了#define声明方式，它用来表示常数声明。要记住哪种色号对应哪种颜色实在太

麻烦了，所以为了便于理解，做了以上声明。

8 今天的成果（harib01h）

我们已经努力到现在了，再加最后一把劲儿。这次我们只修改HariMain程序。让我们看看执行结果会是什么样呢？

本次的HariMain

```
void HariMain(void)
{
    char *vram;
    int xsize, ysize;

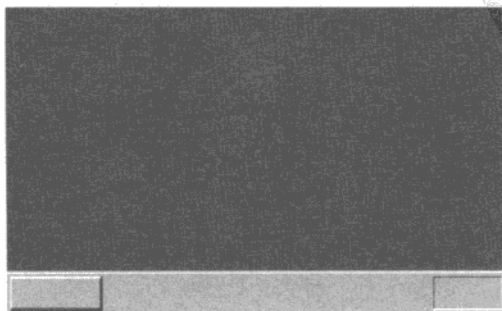
    init_palette();
    vram = (char *) 0xa0000;
    xsize = 320;
    ysize = 200;

    boxfill8(vram, xsize, COL8_008484, 0, 0, xsize - 1, ysize - 29);
    boxfill8(vram, xsize, COL8_C6C6C6, 0, ysize - 28, xsize - 1, ysize - 28);
    boxfill8(vram, xsize, COL8_FFFFFFFF, 0, ysize - 27, xsize - 1, ysize - 27);
    boxfill8(vram, xsize, COL8_C6C6C6, 0, ysize - 26, xsize - 1, ysize - 1);

    boxfill8(vram, xsize, COL8_FFFFFFFF, 3, ysize - 24, 59, ysize - 24);
    boxfill8(vram, xsize, COL8_FFFFFFFF, 2, ysize - 24, 2, ysize - 4);
    boxfill8(vram, xsize, COL8_848484, 3, ysize - 4, 59, ysize - 4);
    boxfill8(vram, xsize, COL8_848484, 59, ysize - 23, 59, ysize - 5);
    boxfill8(vram, xsize, COL8_000000, 2, ysize - 3, 59, ysize - 3);
    boxfill8(vram, xsize, COL8_000000, 60, ysize - 24, 60, ysize - 3);

    boxfill8(vram, xsize, COL8_848484, xsize - 47, ysize - 24, xsize - 4, ysize - 24);
    boxfill8(vram, xsize, COL8_848484, xsize - 47, ysize - 23, xsize - 47, ysize - 4);
    boxfill8(vram, xsize, COL8_FFFFFFFF, xsize - 47, ysize - 3, xsize - 4, ysize - 3);
    boxfill8(vram, xsize, COL8_FFFFFFFF, xsize - 3, ysize - 24, xsize - 3, ysize - 3);

    for (;;) {
        io_hlt();
    }
}
```



怎么样？（笑）

任务条 (task bar) 有点大了, 这是因为像素数太少的缘故吧。但很有进步, 已经有点操作系统的样子了。总算到了这一步。从什么都不会开始, 到现在只用了四天。嗯, 干得不错嘛。现在的haribote.sys是1216字节, 大概是1.2KB吧。虽然这个操作系统很小, 但已经有这么多功能了。好, 今天先到此为止, 明天再见啦。