第 10 天

叠加处理

- 内存管理（续）（harib07a）
- 叠加处理（harib07b）
- 提高叠加处理速度（1）（harib07c）
- 提高叠加处理速度（2）（harib07d）

1 内存管理（续）（harib07a）

得益于昨天的努力，我们终于可以进行内存管理了。不过仔细一看会注意到，bootpack.c都已经有254行了。笔者感觉这段程序太长了，决定整理一下，分出一部分到memory.c中去。（整理中）……好了，整理完了。现在bootpack.c变成95行了。

为了以后使用起来更加方便，我们还是把这些内存管理函数再整理一下。memman_alloc和memman_free能够以1字节为单位进行内存管理，这种方式虽然不错，但是有一点不足——在反复进行内存分配和内存释放之后，内存中就会出现很多不连续的小段未使用空间，这样就会把man->frees消耗殆尽。

因此，我们要编写一些总是以0x1000字节为单位进行内存分配和释放的函数，它们会把指定的内存大小按0x1000字节为单位向上舍入（roundup），而之所以要以0x1000字节为单位，是因为笔者觉得这个数比较规整。另外，0x1000字节的大小正好是4KB。

本次的*memory.c节选

```
unsigned int memman_alloc_4k(struct MEMMAN *man, unsigned int size)
{
    unsigned int a;
    size = (size + 0xfff) & 0xfffff000;
    a = memman_alloc(man, size);
    return a;
```

```
}

int memman_free_4k(struct MEMMAN *man, unsigned int addr, unsigned int size)
{
    int i;
    size = (size + 0xfff) & 0xfffff000;
    i = memman_free(man, addr, size);
    return i;
}
```

下面我们来看看这次增加的部分，这里的关键是向上舍入，可是如果上来就讲向上舍入的话可能不太好懂，所以我们还是先从向下舍入（round down）讲起吧。

讲数字问题时，以钱为例大家可能更容易理解，所以我们就用钱来举例说明。比如，把123元以10元为单位进行向下舍入，就是120元；把456元以100元为单位进行向下舍入，就是400元。通过这些例子，你会发现“所谓向下舍入，就是把最后几位数字强制变为0”。

所以如果将0x12345678以0x1000为单位进行向下舍入，得到的就应该是0x12345000吧？没错，这就是正确答案。这样一来，用纸笔就可以进行向下舍入运算了。不过，如果我们要写个程序让电脑来做同样的事，那该怎么办才好呢？

在二进制下，如果我们想把某位变为0，只要进行“与运算”就可以了，这在4.2节已经介绍过了。而十六进制其实就是把二进制数4位4位地排在一起，所以要想把十六进制的某一位设置为0，同样只进行“与运算”就可以。

```
0x12345678 & 0xfffff000 = 0x12345000
```

因此把变量i中的数字以0x1000为单位进行向下舍入的式子如下：

```
i  = i & 0xfffff000;
```

顺便告诉大家一下，以0x10为单位向下舍入的式子是“i =i & 0xfffffff0;”。这样我们就掌握了向下舍入的方法。

下面我们来看看向上舍入。如果把123元以10元为单位进行向上舍入，就是130元；把456元以100元为单位进行向上舍入，就是500元。嗯嗯，原来如此，看来先向下舍入，再在它的结果上做个加法运算就可以了。

以0x1000为单位对0x12345678进行向上舍入的结果为0x12346000。所以有人可能会问：“要是用程序来表达这个过程的话就应该写成这样吧？”

```
i = (i & 0xfffff000) + 0x1000;
```

看起来貌似确实不错，但其实这并不是正确答案。因为如果“i＝0x12345000;”时执行上述命令，结果就变成了“i=0x12346000;”。这当然不对啦。这就相当于，以10元为单位对120元进行向上舍入，结果为130元，但实际上120 元向上舍入后应该还是120元。

所以我们要对程序进行改进。具体做法是：先判断最后几位，如果本来就是零则什么也不做，如果不是零就进行下面的运算。

```
if ((i & 0xfff) ! = 0)  { i = (i  & 0xfffff000) + 0x1000;}
```

这样问题就解决了。大功告成。

现在我们可以灵活自由地进行向下舍入和向上舍入了，而实际上向上舍入还有改进的“窍门”，那就是：

```
i  = (i + 0xfff) & 0xfffff000;
```

这是怎么回事呢？实际上这是“加上0xfff后进行向下舍入”的运算。不论最后几位是什么，都可以用这个公式进行向上舍入运算。真的吗？

由于十六进制不易理解，所以我们还是以钱的十进制运算为例来说明吧。如使用这个方法以100元为单位对456元进行向上舍入，就相当于先加上99元再进行向下舍入。456元加上99元是555元，向下舍入后就是500元了。嗯，这方法做出来的答案没错，456元向上舍入的结果确实就是500元。

那么如果对400元进行向上舍入呢？先加上99元，得到499元，再进行向下舍入，结果是400元。看，400元向上舍入的结果还是400元。

这种方法多方便呀，可比if语句什么的好用多了。不过其中的原理是什么呢？其实加上99元就是判断进位，如果最后两位不是00，就要向前进一位，只有当最后两位是00时，才不需要进位。接下来再向下舍入，这样就正好把因为加法运算而改变的后两位设置成00了。看，向上舍入就成功了。

这个技巧并不是笔者想出来的，忘了是从哪本书上看到的。能想到这么做的人真是相当聪明呢。既然有了这么方便的技巧，我们没道理不用，在此笔者大力推荐给大家。而在memman_alloc_4k和memman_free_4k中也大量使用了该技巧。

那么试着“make run”一下吧。可是没有任何反应呀！那当然了，这次做的新函数，还没有被调用呢。

COLUMN-6 十进制数的向下舍入

上面介绍了“与运算”可以应用于二进制数和十六进制数的向下舍入，那么对于我们所熟悉的十进制数的向下舍入，也能使用“与运算”吗？

以0x10为单位对变量i进行向下舍入时，实际上是进行了“i=i&0xfffffff0;”处理，可以将它看成“i=i&(0x100000000-0x10);”。同样，以0x100为单位进行向下舍入时，进行的是“i=i&0xffffff00;”处理，所以这也可以看成是“i=i&(0x100000000-0x100);”。也就是说，我们好像可以归纳出“i=i&(0x100000000-向下舍入单位);”。

按照以上思路，如果以100为单位对变量i进行向下舍入，就可以按照“i=i&(0x100000000-100);”，即“i=i&0xffffff9c;”来处理。但这样做并不成功。假设“i=123;”，结果是123&0xffffff9c=24，没有得到我们的预期答案100。

不愿轻易放弃的人可以尝试更多的计算式，不过没有一个能成功，因为不能用“与运算”来进行十进制数的向下舍入处理，“与运算”只能用于二进制数的向下舍入处理。而十六进制数因为是4位4位排在一起的二进制数，所以凑巧成功了。

这倒不是说无法对二进制和十六进制以外的数进行向下舍入处理，只不过是不能使用“与运算”而已。如果允许使用其他方法，一样可以轻松地进行计算。例如“i=(i/100)*100;”，只需要先除以100再乘以100就可以了。我们来假设“i=123”，123除以100，结果是1（当整数除以整数时，答案还是整数，所余的数值叫做“余数”），再用1乘以100就得到了我们的预期结果100。再假设“i=456;”，那么先除以100得到4，再扩大100倍结果就是400，这个答案也是正确的。

我们还可以把计算方法进一步改进一下，写成“i=i-(i%100);”，意思是用i减去i除以100所得的余数。这种方法只用了除法和减法计算，比既用除法又用乘法要快。

不管采用以上哪种方法，在以2^n（n>0）以外的数为单位进行向下舍入和向上舍入处理时，都必须要使用除法命令，而它恰恰是CPU最不好处理的命令之一，所以计算过程要花费较长的时间（当然，在我们看来是一瞬间就结束了）。而“与”命令是所有CPU命令中速度最快的命令之一，和除法命令相比其执行速度要快10倍到100倍。

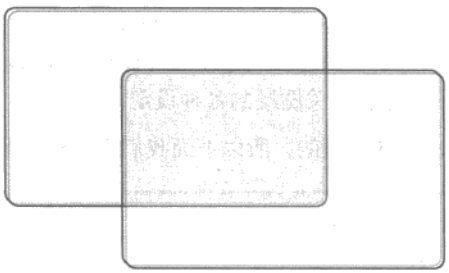
由此可见，如果以1000字节或4000字节单位进行内存管理的话，每次分配内存时，都不得不进行繁琐的除法计算。但如果以1024字节或4096字节为单位进行内存管理的话（两者都是在二进制下易于取整的数字。附带说明:0x1000=4096），在向上舍入的计算中就可以使用“与运算”，这样也能够提高操作系统的运行速度，因此笔者认为这个设计很高明。

2 叠加处理（harib07b）

上一节我们为了转换心情，做了内存管理的探讨，现在还是回过头来，继续解决鼠标的问题吧。从各方面深入思考鼠标的叠加处理确实很有意思，不过考虑到今后我们还面临着窗口的叠加

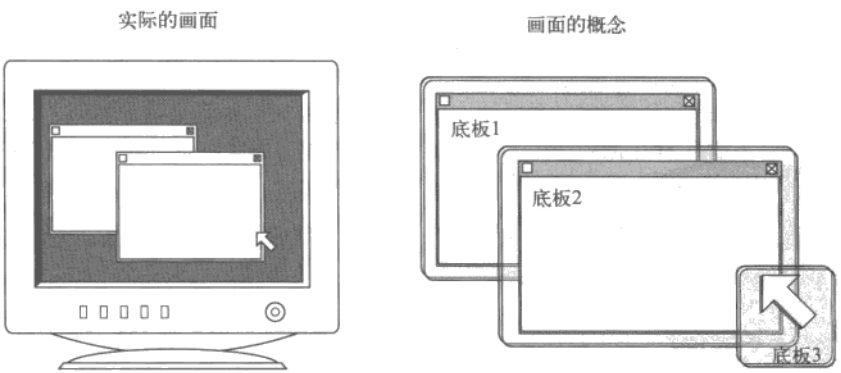
处理问题，所以笔者想做这么一段程序，让它不仅适用于鼠标的叠加处理，也能直接适用于窗口的叠加处理。

其实在画面上进行叠加显示，类似于将绘制了图案的透明图层[①]叠加在一起。



实际上，我们并不是像上面那样仅仅把两张大小相同的图层重叠在一起，而是要从大到小准备很多张图层。

最上面的小图层用来描绘鼠标指针，它下面的几张图层是用来存放窗口的，而最下面的一张图层用来存放桌面壁纸。同时，我们还要通过移动图层的方法实现鼠标指针的移动以及窗口的移动。



我们想法已经有了，下面就把它们变成程序吧。首先来考虑如何将一个图层的信息编成程序。

① 读者朋友如果对图像处理软件中的“层”有所了解，也许脑海中会立刻浮现出这个概念。

```
struct SHEET {
    unsigned char *buf;
    int bxsize, bysize, vx0, vy0, col_inv, height, flags;
};
```

暂时先写成这样就可以了。程序里的sheet这个词，表示“透明图层”的意思。笔者觉得英文里没有和“透明图层”接近的词，就凭感觉选了它。buf是用来记录图层上所描画内容的地址（buffer的略语）。图层的整体大小，用bxsize*bysize表示。vx0和vy0是表示图层在画面上位置的坐标，v是VRAM的略语。col_inv表示透明色色号，它是color（颜色）和invisible（透明）的组合略语。height表示图层高度。Flags用于存放有关图层的各种设定信息。

只有一个图层是不能实现叠加处理的，所以下面我们来创建一个管理多重图层信息的结构。

```
#define MAX_SHEETS      256

struct SHTCTL {
    unsigned char *vram;
    int xsize, ysize, top;
    struct SHEET *sheets[MAX_SHEETS];
    struct SHEET sheets0[MAX_SHEETS];
};
```

我们创建了SHTCTL结构体，其名称来源于sheet control的略语，意思是“图层管理”。MAX_SHEETS是能够管理的最大图层数，这个值设为256应该够用了。

变量vram、xsize、ysize代表VRAM的地址和画面的大小，但如果每次都从BOOTINFO查询的话就太麻烦了，所以在这里预先对它们进行赋值操作。top代表最上面图层的高度。sheets0这个结构体用于存放我们准备的256个图层的信息。而sheets是记忆地址变量的领域，所以相应地也要先准备256份。这是干什么用呢？由于sheets0中的图层顺序混乱，所以我们把它们按照高度进行升序排列，然后将其地址写入sheets中，这样就方便多了。

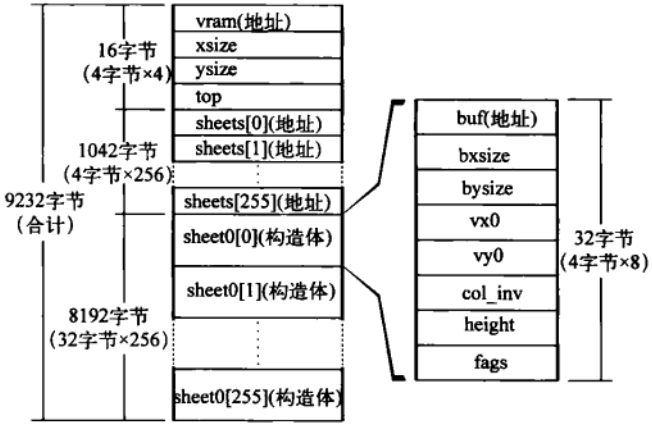
不知不觉我们已经写了很多了，不过也许个别地方大家还不太明白，与其在这纸上谈兵，不如直接看程序更易于理解。所以前面的说明部分，大家即使不懂也别太在意，先往下看吧。

在这里我们稍微说一下结构体吧。内容不难，只是确认大家是不是真正理解了这个概念。struct SHTCTL结构体的内部既有子结构体，又有结构体的指针数组，稍稍有些复杂，不过却是一个不错的例子。

我们的这个例子并不是用文字来解说，而是通过图例展示给大家。请大家看看下面这幅图，确认一下是否理解了结构体。

我们提到的图层控制变量中，仅仅sheets0的部分大小就有32× 256=8 192，即8KB，如果再加上sheets的话，就超过了9KB。对于空间需要如此大的变量，我们想赶紧使用memman_alloc_4k

来分配内存空间，所以就编写了对内存进行分配和初始化的函数。



本次的*sheet.c节选

```
struct SHTCTL *shtctl_init(struct MEMMAN *memman, unsigned char *vram, int xsize, int ysize)
{
    struct SHTCTL *ctl;
    int i;
    ctl = (struct SHTCTL *) memman_alloc_4k(memman, sizeof (struct SHTCTL));
    if (ctl == 0) {
        goto err;
    }
    ctl->vram = vram;
    ctl->xsize = xsize;
    ctl->ysize = ysize;
    ctl->top = -1; /*一个SHEET没都有 */
    for (i = 0; i < MAX_SHEETS; i++) {
         ctl->sheets0[i].flags = 0; /* 标记为未使用 */
     }
err:
    return ctl;
}
```

这段程序是什么的呢？首先使用memman_alloc_4k来分配用于记忆图层控制变量的内存空间，这时必须指定该变量所占空间的大小，不过我们可以使用sizeof（struct SHTCTL）这种写法，让C编译器自动计算。只要写sizeof（变量型），C编译器就会计算出该变量型所需的字节数。

接着，我们给控制变量赋值，给其下的所有图层变量都加上“未使用”标签。做完这一步，这个函数就完成了。

下面我们再做一个函数，用于取得新生成的未使用图层。

本次的*sheet.c节选

```
#define SHEET_USE       1

struct SHEET *sheet_alloc(struct SHTCTL *ctl)
{
    struct SHEET *sht;
    int i;
    for (i = 0; i < MAX_SHEETS; i++) {
        if (ctl->sheets0[i].flags == 0) {
            sht = &ctl->sheets0[i];
            sht->flags = SHEET_USE; /* 标记为正在使用*/
            sht->height = -1; /* 隐藏 */
            return sht;
        }
    }
    return 0;   /* 所有的SHEET都处于正在使用状态*/
}
```

在sheets0[ ]中寻找未使用的图层，如果找到了，就将其标记为“正在使用”，并返回其地址就可以了，这里没有什么难点。高度设为-1，表示图层的高度还没有设置，因而不是显示对象。

程序中出现的&ctl->sheets0[i]是“ctl->sheets0[i]的地址”的意思。也就是说，指的是&(ctl->sheets0[i])，而不是（&ctl）-> sheets0[i]。

本次的*sheet.c节选

```
void sheet_setbuf(struct SHEET *sht, unsigned char *buf, int xsize, int ysize, int col_inv)
{
    sht->buf = buf;
    sht->bxsize = xsize;
    sht->bysize = ysize;
    sht->col_inv = col_inv;
    return;
}
```

这是设定图层的缓冲区大小和透明色的函数，这也没什么难点。

接下来我们写设定底板高度的函数。这稍微有些复杂，所以我们在程序中加入了不少注释。这里的updown就是“上下”的意思。

本次的*sheet.c节选

```
void sheet_updown(struct SHTCTL *ctl, struct SHEET *sht, int height)
{
    int h, old = sht->height; /* 存储设置前的高度信息 */

    /*  如果指定的高度过高或过低，则进行修正 */
    if (height > ctl->top + 1) {
        height = ctl->top + 1;
    }
    if (height < -1) {
        height = -1;
    }
    sht->height = height; /* 设定高度 */

    /* 下面主要是进行sheets[ ]的重新排列 */
    if (old > height) { /* 比以前低 */
        if (height >= 0) {
            /* 把中间的往上提 */
            for (h = old; h > height; h--) {
                ctl->sheets[h] = ctl->sheets[h - 1];
                ctl->sheets[h]->height = h;
            }
            ctl->sheets[height] = sht;
        } else {    /* 隐藏 */
            if (ctl->top > old) {
                /* 把上面的降下来 */
                for (h = old; h < ctl->top; h++) {

                    ctl->sheets[h] = ctl->sheets[h + 1];
                    ctl->sheets[h]->height = h;
                }
            }
            ctl->top--; /* 由于显示中的图层减少了一个，所以最上面的图层高度下降 */
        }
        sheet_refresh(ctl); /* 按新图层的信息重新绘制画面 */
    } else if (old < height) {  /* 比以前高 */
        if (old >= 0) {
            /* 把中间的拉下去 */
            for (h = old; h < height; h++) {
                ctl->sheets[h] = ctl->sheets[h + 1];
                ctl->sheets[h]->height = h;
            }
            ctl->sheets[height] = sht;
        } else {    /* 由隐藏状态转为显示状态 */
            /* 将已在上面的提上来 */
            for (h = ctl->top; h >= height; h--) {
                ctl->sheets[h + 1] = ctl->sheets[h];
                ctl->sheets[h + 1]->height = h + 1;
            }
            ctl->sheets[height] = sht;
            ctl->top++; /* 由于已显示的图层增加了1个，所以最上面的图层高度增加 */
        }
        sheet_refresh(ctl); /* 按新图层信息重新绘制画面 */
```

```
    }
    return;
}
```

程序稍稍有些长，不过既然大家能看懂前面的程序，那么这个程序应该也是可以看明白的。每一条语句并不比之前的语句难，只是整个程序变长了而已。最初可能很难看进去，但是如果一直坚持读下去的话，阅读程序的能力就会越来越强。

程序中间有“ctl->sheets[h]->height = h;”这样一句话。两个[->]一起出现估计还是第一次，不过大家应该懂吧。这当然是“(*(*ctl).sheets[h]).height = h;”的意思了。

要是改写为下面这样，就好理解了。

```
struct SHEET *sht2;
sht2 = ctl->sheets[h];
sht2 -> height = h;
```

下面来说说在sheet_updown中使用的sheet_refresh函数。这个函数会从下到上描绘所有的图层。refresh是“刷新”的意思。电视屏幕就是在1秒内完成多帧的描绘才做出动画效果的，这个动作就被称为刷新。而这种对图层的刷新动作，与电视屏幕的动作有些相似，所以我们也给它起名字叫做刷新。

本次的*sheet.c节选

```
void sheet_refresh(struct SHTCTL *ctl)
{
    int h, bx, by, vx, vy;
    unsigned char *buf, c, *vram = ctl->vram;
    struct SHEET *sht;
    for (h = 0; h <= ctl->top; h++) {
        sht = ctl->sheets[h];
        buf = sht->buf;
        for (by = 0; by < sht->bysize; by++) {
            vy = sht->vy0 + by;
            for (bx = 0; bx < sht->bxsize; bx++) {
                vx = sht->vx0 + bx;
                c = buf[by * sht->bxsize + bx];
                if (c != sht->col_inv) {
                    vram[vy * ctl->xsize + vx] = c;
                }
            }
        }
    }
    return;
}
```

对于已设定了高度的所有图层而言，要从下往上，将透明以外的所有像素都复制到VRAM中。由于是从下开始复制，所以最后最上面的内容就留在了画面上。

现在我们来看一下不改变图层高度而只上下左右移动图层的函数——sheet_slide。slide原意是“滑动”，这里指上下左右移动图层。

本次的*sheet.c节选

```
void sheet_slide(struct SHTCTL *ctl, struct SHEET *sht, int vx0, int vy0)
{
    sht->vx0 = vx0;
    sht->vy0 = vy0;
    if (sht->height >= 0) { /* 如果正在显示*/
        sheet_refresh(ctl); /* 按新图层的信息刷新画面 */
    }
    return;
}
```

最后是释放已使用图层的内存的函数sheet_free。这个简单。

本次的*sheet.c节选

```
void sheet_free(struct SHTCTL *ctl, struct SHEET *sht)
{
    if (sht->height >= 0) {
        sheet_updown(ctl, sht, -1); /* 如果处于显示状态，则先设定为隐藏 */
    }
    sht->flags = 0; /* "未使用"标志  */
    return;
}
```

下面我们将以上与图层相关的程序汇总到sheet.c中，所以就要改造HariMain函数了。

本次的*bootpack.c节选

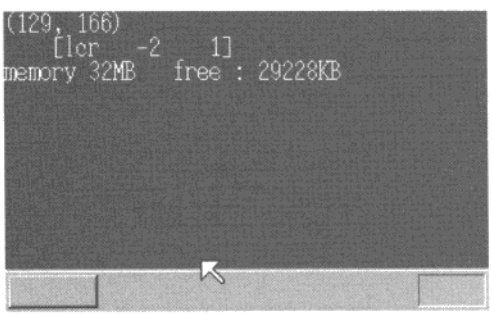
```
void HariMain(void)
{
    （中略）
    struct SHTCTL *shtctl;
    struct SHEET *sht_back, *sht_mouse;
    unsigned char *buf_back, buf_mouse[256];
```

```
(中略)

init_palette();
shtctl = shtctl_init(memman, binfo->vram, binfo->scrnx, binfo->scrny);
sht_back  = sheet_alloc(shtctl);
sht_mouse = sheet_alloc(shtctl);
buf_back  = (unsigned char *) memman_alloc_4k(memman, binfo->scrnx * binfo->scrny);
sheet_setbuf(sht_back, buf_back, binfo->scrnx, binfo->scrny, -1); /* 没有透明色 */
sheet_setbuf(sht_mouse, buf_mouse, 16, 16, 99); /* 透明色号99 */
init_screen8(buf_back, binfo->scrnx, binfo->scrny);
init_mouse_cursor8(buf_mouse, 99); /* 背景色号99 */
sheet_slide(shtctl, sht_back, 0, 0);
mx = (binfo->scrnx - 16) / 2; /* 按显示在画面中央来计算坐标 */
my = (binfo->scrny - 28 - 16) / 2;
sheet_slide(shtctl, sht_mouse, mx, my);
sheet_updown(shtctl, sht_back,  0);
sheet_updown(shtctl, sht_mouse, 1);
sprintf(s, "(%3d, %3d)", mx, my);
putfonts8_asc(buf_back, binfo->scrnx, 0, 0, COL8_FFFFFF, s);
sprintf(s, "memory %dMB   free : %dKB",
        memtotal / (1024 * 1024), memman_total(memman) / 1024);
putfonts8_asc(buf_back, binfo->scrnx, 0, 32, COL8_FFFFFF, s);
sheet_refresh(shtctl);

for (;;) {
    io_cli();
    if (fifo8_status(&keyfifo) + fifo8_status(&mousefifo) == 0) {
        io_stihlt();
    } else {
        if (fifo8_status(&keyfifo) != 0) {
            i = fifo8_get(&keyfifo);
            io_sti();
            sprintf(s, "%02X", i);
            boxfill8(buf_back, binfo->scrnx, COL8_008484,  0, 16, 15, 31);
            putfonts8_asc(buf_back, binfo->scrnx, 0, 16, COL8_FFFFFF, s);
            sheet_refresh(shtctl);
        } else if (fifo8_status(&mousefifo) != 0) {
            i = fifo8_get(&mousefifo);
            io_sti();
            if (mouse_decode(&mdec, i) != 0) {
                /* 因为已得到3字节的数据所以显示 */
                sprintf(s, "[lcr %4d %4d]", mdec.x, mdec.y);
                if ((mdec.btn & 0x01) != 0) {
                    s[1] = 'L';
                }
                if ((mdec.btn & 0x02) != 0) {
                    s[3] = 'R';
                }
                if ((mdec.btn & 0x04) != 0) {
                    s[2] = 'C';

                }
                boxfill8(buf_back, binfo->scrnx, COL8_008484, 32, 16, 32 + 15 * 8 - 1, 31);
                putfonts8_asc(buf_back, binfo->scrnx, 32, 16, COL8_FFFFFF, s);
                /* 移动光标 */
                mx += mdec.x;
```

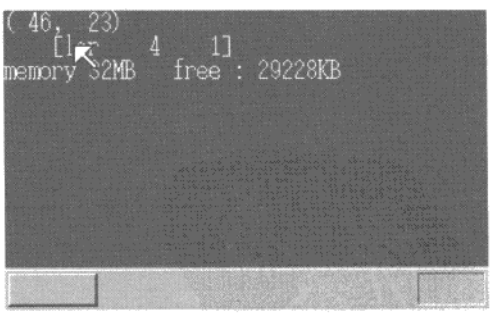
```
                my += mdec.y;
                if (mx < 0) {
                    mx = 0;
                }
                if (my < 0) {
                    my = 0;
                }
                if (mx > binfo->scrnx - 16) {
                    mx = binfo->scrnx - 16;
                }
                if (my > binfo->scrny - 16) {
                    my = binfo->scrny - 16;
                }
                sprintf(s, "(%3d, %3d)", mx, my);
                boxfill8(buf_back, binfo->scrnx, COL8_008484, 0, 0, 79, 15); /* 消坐标 */
                putfonts8_asc(buf_back, binfo->scrnx, 0, 0, COL8_FFFFFF, s); /* 写坐标 */
                sheet_slide(shtctl, sht_mouse, mx, my); /* 包含sheet_refresh含sheet_refresh */
            }
        }
    }
}
```

我们准备了2个图层，分别是sht_back和sht_mouse，还准备了2个缓冲区buf_back和buf_mouse，用于在其中描绘图形。以前我们指定为binfo -> vram的部分，现在有很多都改成了buf_back。而且每次修改缓冲区之后都要刷新。这段代码不是很难，只要大家认认真真地读，肯定能理解。

好了，终于可以“make run”了，真是激动人心的一刻！



成功啦！



很好很好！

成功地运行啦！真开心！由于使用的内存增加，从而导致剩余内存相对减少，但这也是不可避免的，现在这样就可以了。

不过其实这里面还是有问题。从图片来看确实很完美，可实际操作一下，你恐怕就要喊“吐血啦！”。没错，它太慢了，而且画面还一闪一闪的。动一下鼠标就要郁闷一次，哪个用户想用这样的操作系统呢？所以下面我们就来解决这个问题吧。

3 提高叠加处理速度（1）（harib07c）

那么怎样才能提高速度呢？既然其他操作系统都能处理得那么快，就肯定有好的方法。首先，我们从鼠标指针的移动，也就是图层的移动来思考一下。

鼠标指针虽然最多只有16×16=256个像素，可根据harib07b的原理，只要它稍一移动，程序就会对整个画面进行刷新，也就是重新描绘320×200=64 000个像素。而实际上，只重新描绘移动相关的部分，也就是移动前后的部分就可以了，即256×2=512个像素。这只是64 000像素的0.8%而已，所以有望提速很多。现在我们根据这个思路写一下程序。

本次的*sheet.c节选

```
void sheet_refreshsub(struct SHTCTL *ctl, int vx0, int vy0, int vx1, int vy1)
{
    int h, bx, by, vx, vy;
    unsigned char *buf, c, *vram = ctl->vram;
    struct SHEET *sht;
    for (h = 0; h <= ctl->top; h++) {
        sht = ctl->sheets[h];
        buf = sht->buf;
        for (by = 0; by < sht->bysize; by++) {
            vy = sht->vy0 + by;
            for (bx = 0; bx < sht->bxsize; bx++) {
                vx = sht->vx0 + bx;
                if (vx0 <= vx && vx < vx1 && vy0 <= vy && vy < vy1) {
                    c = buf[by * sht->bxsize + bx];
                    if (c != sht->col_inv) {
                        vram[vy * ctl->xsize + vx] = c;
                    }
                }
            }
        }
    }
    return;
}
```

这个函数几乎和sheet_refresh一样，唯一的不同点在于它能使用vx0 ~ vy1指定刷新的范围，而我们只追加了一个if语句就实现了这个新功能。另外，程序中的&&运算符是我们之前没有见过的，所以在这里详细解释一下。

&&运算符是把多个条件关系式连接起来的运算符。当用它连接的所有条件都满足时，就执行{}中的程序；只要有一个条件不满足，就不执行（如果有else，就执行else后的语句）。另外，还有一个跟它很像的运算符“||”。“||”也是把多个条件关系式连接起来的运算符，不过由它连接的各个条件，只要其中一个满足了，就执行{}中的程序。简而言之，&&就是“而且”，而||是“或者”。

条件“vx大于等于vx0且小于vx1”可以用数学式vx0 <= vx < vx1来表达，但在C语言中不能这样写，我们只能写成 vx0 <= vx && vx < vx1。

现在我们使用这个refreshsub函数来提高sheet_slide的运行速度。

本次的*sheet.c节选

```
void sheet_slide(struct SHTCTL *ctl, struct SHEET *sht, int vx0, int vy0)
{
    int old_vx0 = sht->vx0, old_vy0 = sht->vy0;
    sht->vx0 = vx0;
    sht->vy0 = vy0;
    if (sht->height >= 0) { /* 如果正在显示，则按新图层的信息刷新画面 */
        sheet_refreshsub(ctl, old_vx0, old_vy0, old_vx0 + sht->bxsize, old_vy0 + sht->bysize);
        sheet_refreshsub(ctl, vx0, vy0, vx0 + sht->bxsize, vy0 + sht->bysize);
    }
    return;
}
```

这段程序所做的是：首先记住移动前的显示位置，再设定新的显示位置，最后只要重新描绘移动前和移动后的地方就可以了。

估计大家会认为“这次鼠标的移动就快了吧”，但移动鼠标时，由于要在画面上显示坐标等信息，结果又执行了sheet_refresh程序，所以还是很慢。为了不浪费我们付出的各种努力，下面我们就来解决一下图层内文字显示的问题。

我们所说的在图层上显示文字，实际上并不是改写图层的全部内容。假设我们已经写了20个字，那么8×16×20=2560，也就是仅仅重写2560个像素的内容就应该足够了。但现在每次却要重写64 000个像素的内容，所以速度才那么慢。

这么说来，这里好像也可以使用refreshsub，那么我们就来重新编写函数sheet_refresh吧。

本次的*sheet.c节选

```
void sheet_refresh(struct SHTCTL *ctl, struct SHEET *sht, int bx0, int by0, int bx1, int by1)
{
    if (sht->height >= 0) { /* 如果正在显示，则按新图层的信息刷新画面*/
        sheet_refreshsub(ctl, sht->vx0 + bx0, sht->vy0 + by0, sht->vx0 + bx1, sht->vy0 + by1);
    }
    return;
}
```

所谓指定范围，并不是直接指定画面内的坐标，而是以缓冲区内的坐标来表示。这样一来，HariMain就可以不考虑图层在画面中的位置了。

我们改动了refresh，所以也要相应改造updown。做了改动的只有sheet_refresh（ctl）这部分（有两处），修改后的程序如下：

```
        sheet_refreshsub(ctl, sht->vx0, sht->vy0, sht->vx0 + sht->bxsize, sht->vy0 + sht->bysize);
```

最后还要改写HariMain。

本次的*bootpack.c节选

```
void HariMain(void)
{
    (中略)
    sprintf(s, "(%3d, %3d)", mx, my);
    putfonts8_asc(buf_back, binfo->scrnx, 0, 0, COL8_FFFFFF, s);
    sprintf(s, "memory %dMB   free : %dKB",
            memtotal / (1024 * 1024), memman_total(memman) / 1024);
    putfonts8_asc(buf_back, binfo->scrnx, 0, 32, COL8_FFFFFF, s);
    sheet_refresh(shtctl, sht_back, 0, 0, binfo->scrnx, 48); /* 这里！ */

    for (;;) {
        io_cli();
        if (fifo8_status(&keyfifo) + fifo8_status(&mousefifo) == 0) {
            io_stihlt();
        } else {
            if (fifo8_status(&keyfifo) != 0) {
                (中略)
                sheet_refresh(shtctl, sht_back, 0, 16, 16, 32); /* 这里！ */
            } else if (fifo8_status(&mousefifo) != 0) {
                i = fifo8_get(&mousefifo);
                io_sti();
                if (mouse_decode(&mdec, i) != 0) {
                    (中略)
                    boxfill8(buf_back, binfo->scrnx, COL8_008484, 32, 16, 32 + 15 * 8 - 1, 31);
                    putfonts8_asc(buf_back, binfo->scrnx, 32, 16, COL8_FFFFFF, s);
                    sheet_refresh(shtctl, sht_back, 32, 16, 32 + 15 * 8, 32); /* 这里！ */
                    (中略)
                    sprintf(s, "(%3d, %3d)", mx, my);
                    boxfill8(buf_back, binfo->scrnx, COL8_008484, 0, 0, 79, 15); /* 消去坐标 */
                    putfonts8_asc(buf_back, binfo->scrnx, 0, 0, COL8_FFFFFF, s); /* 写出坐标 */
                    sheet_refresh(shtctl, sht_back, 0, 0, 80, 16); /* 这里！ */
                    sheet_slide(shtctl, sht_mouse, mx, my);
                }
            }
        }
    }
}
```

这里我们仅仅改写了sheet_refresh，变更点共有4个。只有每次要往buf_back中写入信息时，才进行sheet_refresh。

这样应该可以顺利运行了。我们赶紧试一试。“make run”。哦，确实比以前快多了。太好了，撒花！不过还是欠缺一些东西……

4 提高叠加处理速度（2）（harib07d）

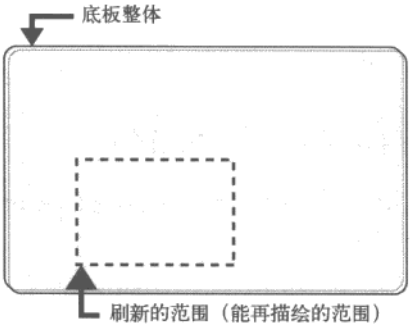
虽然我们想了如此多的办法，但结果还是没有达到我们的期望，真让人郁闷。到底是怎么回事呢？原来还是refreshsub有些问题。

不是太快的refreshsub

```
void sheet_refreshsub(struct SHTCTL *ctl, int vx0, int vy0, int vx1, int vy1)
{
    int h, bx, by, vx, vy;
    unsigned char *buf, c, *vram = ctl->vram;
    struct SHEET *sht;
    for (h = 0; h <= ctl->top; h++) {
        sht = ctl->sheets[h];
        buf = sht->buf;
        for (by = 0; by < sht->bysize; by++) {
            vy = sht->vy0 + by;
            for (bx = 0; bx < sht->bxsize; bx++) {
                vx = sht->vx0 + bx;
                if (vx0 <= vx && vx < vx1 && vy0 <= vy && vy < vy1) {
                    c = buf[by * sht->bxsize + bx];
                    if (c != sht->col_inv) {
                        vram[vy * ctl->xsize + vx] = c;
                    }
                }
            }
        }
    }
    return;
}
```

依照这个程序，即使不写入像素内容，也要多次执行if语句，这一点不太好，如果能改善一下，速度应该会提高不少。

按照上面这种写法，即便只刷新图层的一部分，也要对所有图层的全部像素执行if语句，判断“是写入呢，还是不写呢”。而对于刷新范围以外的部分，就算执行if判断语句，最后也不会进行刷新，所以这纯粹就是一种浪费。既然如此，我们最初就应该把for语句的范围限定在刷新范围之内。



基于以上思路，我们做好了改良版本。

本次的*sheet.c节选

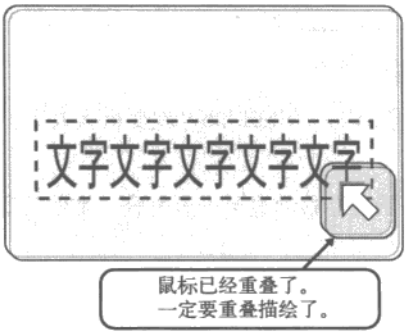
```
void sheet_refreshsub(struct SHTCTL *ctl, int vx0, int vy0, int vx1, int vy1)
{
    int h, bx, by, vx, vy, bx0, by0, bx1, by1;
    unsigned char *buf, c, *vram = ctl->vram;
    struct SHEET *sht;
    for (h = 0; h <= ctl->top; h++) {
        sht = ctl->sheets[h];
        buf = sht->buf;
        /* 使用vx0～vy1，对bx0～by1进行倒推/
        bx0 = vx0 - sht->vx0;
        by0 = vy0 - sht->vy0;
        bx1 = vx1 - sht->vx0;
        by1 = vy1 - sht->vy0;
        if (bx0 < 0) { bx0 = 0; } /* 说明(1) */
        if (by0 < 0) { by0 = 0; }
        if (bx1 > sht->bxsize) { bx1 = sht->bxsize; } /* 说明(2) */
        if (by1 > sht->bysize) { by1 = sht->bysize; }
        for (by = by0; by < by1; by++) {
            vy = sht->vy0 + by;
            for (bx = bx0; bx < bx1; bx++) {
                vx = sht->vx0 + bx;
                c = buf[by * sht->bxsize + bx];
                if (c != sht->col_inv) {
                    vram[vy * ctl->xsize + vx] = c;
                }
            }
        }
    }
    return;
}
```

改良的关键在于，bx在for语句中并不是在0到bxsize之间循环，而是在bx0到bx1之间循环（对于by也一样）。而bx0和bx1都是从刷新范围“倒推”求得的。倒推其实就是把公式变形转换了一下，具体如下：

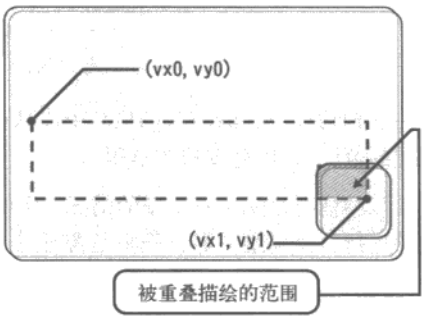
```
vx = sht->vx0 + bx;    →  bx = vx - sht->vx0;
```

计算vx0的坐标相当于bx中的哪个位置，然后把它作为bx0。其他的坐标处理方法也一样。

这样算完以后，就该执行以上程序中说明(1)的地方了。这行代码用于处理刷新范围在图层外侧的情况。什么时候会出现这种情况呢？比如在sht_back中写入字符并进行刷新，而且刷新范围的一部分被鼠标覆盖的情况。

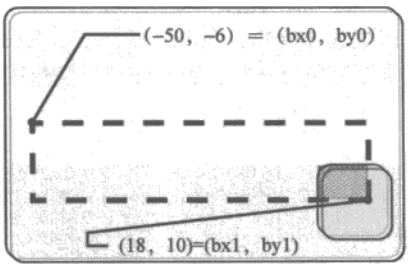


刷新范围与鼠标的图层像这样重叠在一起



这时候的vx和vy

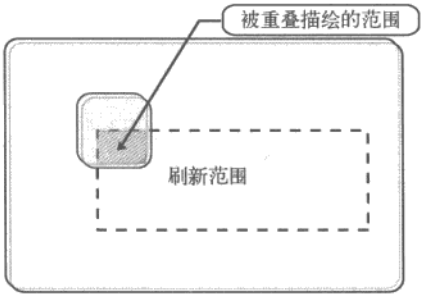
假设在这种情况下h=1，且想要重复刷新鼠标的图层，那么就变成了下面这样。



对sheets[1]进行bx0～bx1计算的时候

在这里必须要进行重复描绘的只有与鼠标图层重叠的那一小块范围，而其他部分并没有被要求刷新，所以不能刷新。这样的话，可以把bx0和by0置0。

程序中“说明(2)”部分所做的，是为了应对不同的重叠方式。



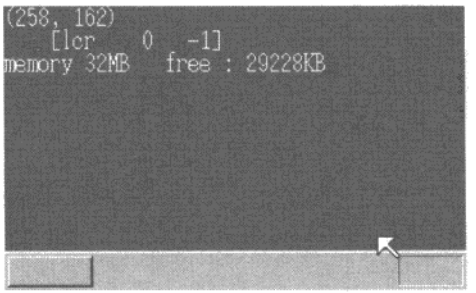
需要执行“说明(2)”部分的情形

在这种情况下，bx0和by0虽然可以从vx0和vy0顺利求取， 但bx1和by1就变得太大了（超出了图层的范围），因此要修改这里。

第三种情况是完全不重叠的情况。例如，鼠标的图层往左移动直至不再重叠。此时当然完全不需要进行重复描绘，那么程序是否可以正常运行呢？

利用倒推计算得出的bx0和bx1都是负值，在说明(1)中，仅仅bx0被修正为0，而在说明(2) 中bx1没有被修正，还是负的。这样的话，for（bx = bx0;bx < bx1;bx++）这个语句里的循环条件bx < bx1从最开就不成立，所以for语句中的命令得不到循环，这样就完全不会进行重复描绘了，很好。

仅仅改了这些地方，就可以提高速度吗？我们来试一下。“make run”（要等待一会儿）吗？哦，这次感觉很好，操作系统正在迅速地运行，太开心了！虽然从表面上看不出有什么不同，不过这次我们要附上照片，展示一番。太棒了！



太好了！真开心。

纪念照片也拍完了（笑），在这里我们看一下haribote.sys的大小吧。哦，是11 104字节。除以1024的话，大约是10.8，也就是10.8KB。……我们的系统正在茁壮成长！ 到这里可以暂时告一段落了，那好，我们今天就到此结束吧。明天见！