Day 9

Phase 1

整理源文件，将键盘鼠标的处理函数和中断处理函数分别移动到新建的`keyboard.c`和`mouse.c`中

不要忘记对`Makefile`进行修改

Phase 2

内存容量检查

SubPhase 1

检查CPU是否是486以上的型号

> 感觉这个好老啊，虚拟机应该都是模拟486以上的CPU吧

检测原理是设置eflags的AC位，由于386没有AC位，所以设置AC位之后再次读取eflags，AC位还会是0

```
#define EFLAGS_AC_BIT 0x00040000
char is486 = 0;
unsigned int eflg;
eflg = io_load_eflags();
eflg |= EFLAGS_AC_BIT;
io_store_eflags(eflg);
eflg = io_load_eflags();
if (eflg & EFLAGS_AC_BIT) is486 = 1;
```

SubPhase 2

关闭缓存

```
if (is486) {
    cr0 = load_cr0();
    cr0 |= CR0_CACHE_DISABLE; /* 禁止缓存 */
    store_cr0(cr0);
}
```

允许缓存

```
if (is486) {
    cr0 = load_cr0();
    cr0 &= ~CR0_CACHE_DISABLE; /* 允许缓存 */
    store_cr0(cr0);
}
```

注意cr0寄存器不是c语言能够直接读写的，所以我们需要增加一个naskfunc，由于没啥特别的地方，这里就不体现代码了

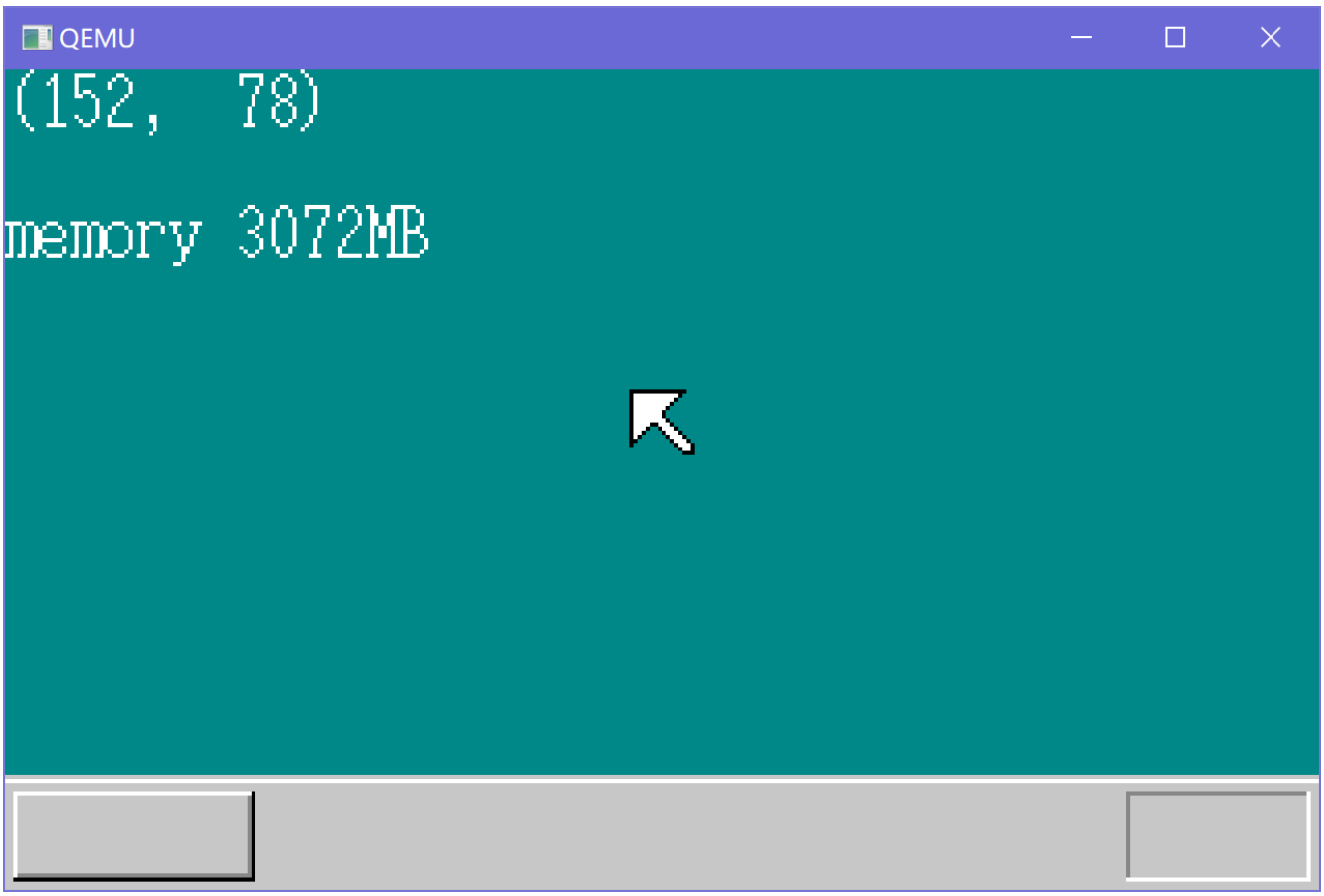
SubPhase 3

在关闭缓存后，我们可以开始测试内存的大小了。具体原理是先读一个内存位值，然后经过多次修改存取，如果有一次结果不符合预期，则说明这个内存位置不可用，否则说明这个内存位置可用，则这个位置之前的内存也可用。

```
unsigned int memtest_sub(unsigned int start, unsigned int end)
{
    unsigned int i, *p, old, pat0 = 0xaa55aa55, pat1 = 0x55aa55aa;
    for (i = start; i <= end; i += 0x1000) {
        p = (unsigned int *) (i + 0xffc);
        old = *p;
        *p = pat0;
        *p ^= 0xffffffff;
        if (*p != pat1) {
not_memory:
            *p = old;
            break;
        }
        *p ^= 0xffffffff;
        if (*p != pat0) {
            goto not_memory;
        }
        *p = old;
    }
    return i;
}
```

为了避免挨个测试所带来的效率上的降低，我们决定每次跳过4KB，并查验这4个KB块中最后的4个字节。

先测试一下



如书上接下来提到的那样，没有出现期望的正常结果。这是因为编译器优化使我们的这个trick失效了。

作者的解决方案使啥？手撸汇编。太不优雅了！

只要我们想办法关闭编译器对这段代码的优化就好了

通过上网搜寻代码，我得到了几个解决方案

第一种是修改Makefile，关闭所有的编译器优化。这可太蠢了，为了一点点优雅的写法而放弃性能，这得不偿失。

第二种是为函数声明及添加`__attribute__((optimize("-O0")))`关键字，关闭这个函数的优化。但编译器提示如下

```
bootpack.c:7: warning: `optimize' attribute directive ignored
bootpack.c:131: warning: `optimize' attribute directive ignored
```

编译器似乎忽略了我们的指令

第三种是利用宏命令暂时关闭优化

在函数之前添加

```
#pragma GCC push_options
#pragma GCC optimize ("O0")
```

函数之后添加

```
#pragma GCC pop_options
```

这样应该会临时为这一小段代码关闭优化。

但遗憾的是，编译器再一次忽略了我们的“建议”

```
bootpack.c:129: warning: ignoring #pragma GCC push_options
bootpack.c:130: warning: ignoring #pragma GCC optimize
bootpack.c:153: warning: ignoring #pragma GCC pop_options
```

没关系，我们还有**第四种**方法，这就是使用`volatile`关键字，我们在可能被编译器进行优化的变量的声明前面加上`volatile`关键字，这样编译器就知道这个变量的值可能被外部改变了。

```
volatile unsigned int i, *p, old, pat0 = 0xaa55aa55, pat1 = 0x55aa55aa;
```

如上代码。

`make run`一发！结果正常！

修改`qemu-win.bat`提升模拟器内存到64MB，再试一下

成功!

Phase 3

内存管理初探

我们要弄一个简单的表来维护可用内存区段，作者使用数组实现了这样的一个简单的版本，但实际上，使用链表会降低内存无法归并情况下插入数据时的移位开销。如果使用平衡树的话，则可以把整体的复杂度降低到$O(logN)$级别。由于代码耦合度比较高，修改不易，我们这里只进行部分修改。

```
int memman_free(struct MEMMAN *man, unsigned int addr, unsigned int size) {
    int i, j, p, k;
    for (p = man->head; p >= 0; i = p, p = man->free[p].next) {
        if (man->free[p].addr > addr) {
            break;
        }
    }
    if (p != man->head) {
        if (man->free[i].addr + man->free[i].size == addr) {
            man->free[i].size += size;
            if (man->free[p].next >= 0) {
                k = man->free[p].next;
                if (addr + size == man->free[k].addr) {
                    man->free[k].size += man->free[p].size;
                    man->free[k].addr = addr;
                    recycle(man, p);
                }
```

```
            }
        }
    }
    if (man->free[p].next >= 0) {
        k = man->free[p].next;
        if (addr + size == man->free[k].addr) {
            man->free[k].addr = addr;
            man->free[k].size += size;
            return 0;
        }
    }
    if (man->top > 0) {
        i = manalloc(man);
        man->free[i].next = man->free[j].next;
        man->free[j].next = i;
        man->free[i].addr = addr;
        man->free[i].size = size;
        return 0;
    }
    man->losts++;
    man->lostsize += size;
    return -1;
}
```

Day 10

Phase 1

内存分配按4KB为单位进行向上舍入，这样能够一定程度上减少内存碎片的产生，提高内存管理的效率。

```
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;
}
```

Phase 2

SubPhase 1

引入图层概念。

为了正确处理堆叠，我们需要引入图层的概念。图层的概念我们在日常使用计算机的过程中经常会接触到。我们可以为每个图层单独保存他们的画面，然后根据需要对整个屏幕进行绘制。

一个图层都需要什么样的信息呢?

以下信息是必须的

变量名	含义
buf	存储画面具体内容的指针
bxsize	画面横向宽度
bysize	画面纵向宽度
vx0	图层左上角位于屏幕上位置的横坐标
vy0	图层左上角位于屏幕上位置的纵坐标
col_inv	透明色号
height	图层纵向顺序编号
flags	其他设定信息

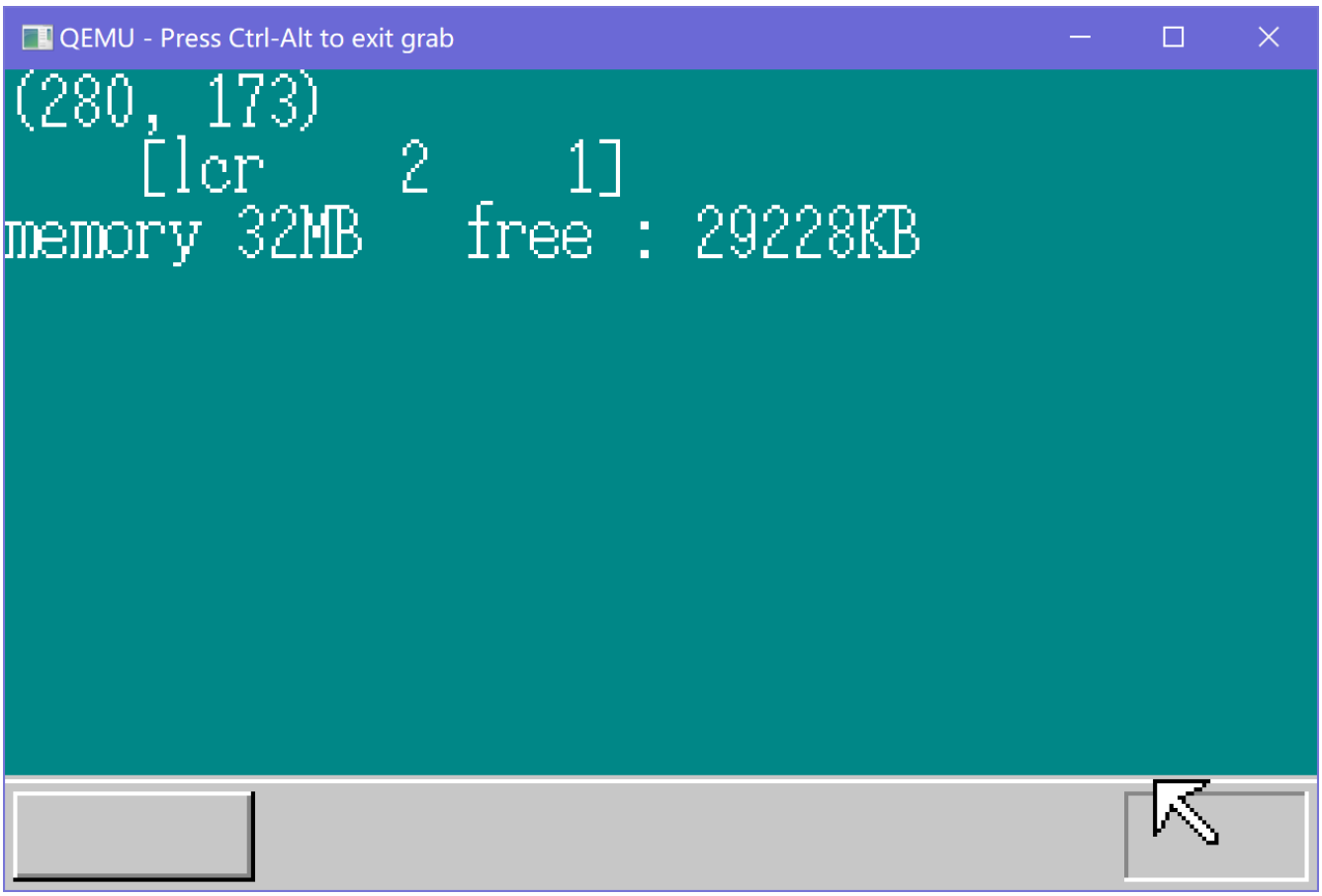
一个这样的sheet结构体共8个四字节变量（32位机指针是4字节）。

我们只准备256个这样的sheet，因为这个看起来够用了，毕竟是实验性的系统，哪怕是使用win10，也很少有人会同时开256个窗口。

我们需要像管理内存一样管理这256个sheets，所以也需要编制相应的分配及回收程序。

我们还需要编制相应的修改height的代码，以及绘制整个屏幕的函数。

完成相应的编制之后，我们并不需要修改putfont putblock等函数，因为作者之前留下vram参数大概是早有预谋，是为了避免添加叠加处理功能的时候修改更多的代码吧。



看起来一切工作正常

SubPhase 2

其实书上提到的屏幕闪烁问题，我并没有遇到。想必大家也没有遇到，估计是因为作者在写这本书的时候计算机机能并不好，而现在的机能相比以前提升很大，屏幕不闪烁也是意料之中了。

不过屏幕刷新还是有优化的地方的，我们每次鼠标移动，都会重绘整个屏幕，这是十分效率低下的。实际上，我们只需要重绘鼠标那16x16个像素就好了。

同样的，文字打印也会重绘整个屏幕，这里也需要优化的。所以我们重新编制一下绘图函数，令他只重绘屏幕的一个矩形部分，而不是整个屏幕范围。

另外作者终于打算在这里处理打印部分超出现有图层/屏幕外部的情况了（笑），我们早在之前刚开始做字符串打印的时候就考虑到了这种情况。

不过他还是暂时还不打算让鼠标的右下部份超出屏幕。

```
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;
        bx0 = vx0 - sht->vx0;
        by0 = vy0 - sht->vy0;
        bx1 = vx1 - sht->vx0;
        by1 = vy1 - sht->vy0;
        if (bx0 < 0)
            bx0 = 0;
        if (by0 < 0)
            by0 = 0;
        if (bx1 > sht->bxsize)
            bx1 = sht->bxsize;
        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;
}
```

Day 11

Phase 1

SubPhase 1

激动！作者终于要处理鼠标不能移到屏幕外面的这个feature了

现阶段的代码如果硬改的话，是会产生以下我们不希望看到的效果的。

我们来修改以下图层绘制函数吧

```
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;
    if (vx0 < 0) { vx0 = 0; }
    if (vy0 < 0) { vy0 = 0; }
    if (vx1 > ctl->xsize) { vx1 = ctl->xsize; }
    if (vy1 > ctl->ysize) { vy1 = ctl->ysize; }
    for (h = 0; h <= ctl->top; h++) {
        sht = ctl->sheets[h];
        buf = sht->buf;
```

```
        bx0 = vx0 - sht->vx0;
        by0 = vy0 - sht->vy0;
        bx1 = vx1 - sht->vx0;
        by1 = vy1 - sht->vy0;
        if (bx0 < 0) { bx0 = 0; }
        if (by0 < 0) { by0 = 0; }
        if (bx1 > sht->bxsize) { bx1 = sht->bxsize; }
        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;
}
```

SubPhase 2

优化代码。

之前*SHTCTL都是作为参数传进去的，优势后我们明明有了SHEET却还要准备*SHTCTL，很不方便，我们来优化一下代码结构，将*SHTCTL纳入到SHEET当中，这样就可以少写一些代码了。

Phase 2

SubPhase 1

窗口

做法是新建一个sheet，然后在上面打印一个看起来像窗口的图像就可以了。

SubPhase 2

高速计数器

如书上所言，窗体更新的部分底色不停的变，在闪烁。这是因为许多时间被浪费在了绘制一定会被覆盖的像素上。我们可以对sheet_refresh稍加修改，只让他绘制更新的图层和更新的图层以上的图层。

但我们发现，仅仅使用这种解决方案是不够的。当鼠标移到上面的时候还是会闪动。这是因为我们多次直接在vram上修改，我们的修改并不一定是我们最后想要显示的结果，所以如果碰见了显示器的某次刷新，那么显示设备也会忠实的把这些中间过程显示出来。

我们可以稍微绕一下，先用一个临时的空间进行绘制，然后再把绘制的结果送到真正的显存当中去，这样就可以避免绘制过程的中间值对显示效果所造成的影响了。

以上是我的解决方法。

作者的解决方法是，对于要重绘的区域，新建一个map数组，记录每一个像素最终对应的是哪一个图层，最后再一起更新，到各个图层中的对应位置找这个像素到底应该是什么颜色。

作者的方法比我的更高效，因为如果各个图层间的覆盖关系没有改变的话，重绘甚至不需要便利整个图层，只需要找对应图层的对应像素点就可以了。

```
void sheet_refreshsub(struct SHTCTL *ctl, int vx0, int vy0, int vx1, int vy1, int h0, int
h1) {
    int h, bx, by, vx, vy, bx0, by0, bx1, by1;
    unsigned char *buf, *vram = ctl->vram, *map = ctl->map, sid;
    struct SHEET *sht;
    if (vx0 < 0)
        vx0 = 0;
    if (vy0 < 0)
        vy0 = 0;
    if (vx1 > ctl->xsize)
        vx1 = ctl->xsize;
    if (vy1 > ctl->ysize)
        vy1 = ctl->ysize;
    for (h = h0; h <= h1; h++) {
        sht = ctl->sheets[h];
        buf = sht->buf;
        sid = sht - ctl->sheets0;
        bx0 = vx0 - sht->vx0;
        by0 = vy0 - sht->vy0;
        bx1 = vx1 - sht->vx0;
        by1 = vy1 - sht->vy0;
        if (bx0 < 0)
            bx0 = 0;
        if (by0 < 0)
            by0 = 0;
        if (bx1 > sht->bxsize)
            bx1 = sht->bxsize;
        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;
                if (map[vy * ctl->xsize + vx] == sid) {
                    vram[vy * ctl->xsize + vx] = buf[by * sht->bxsize + bx];
                }
            }
        }
    }
    return;
}
```