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

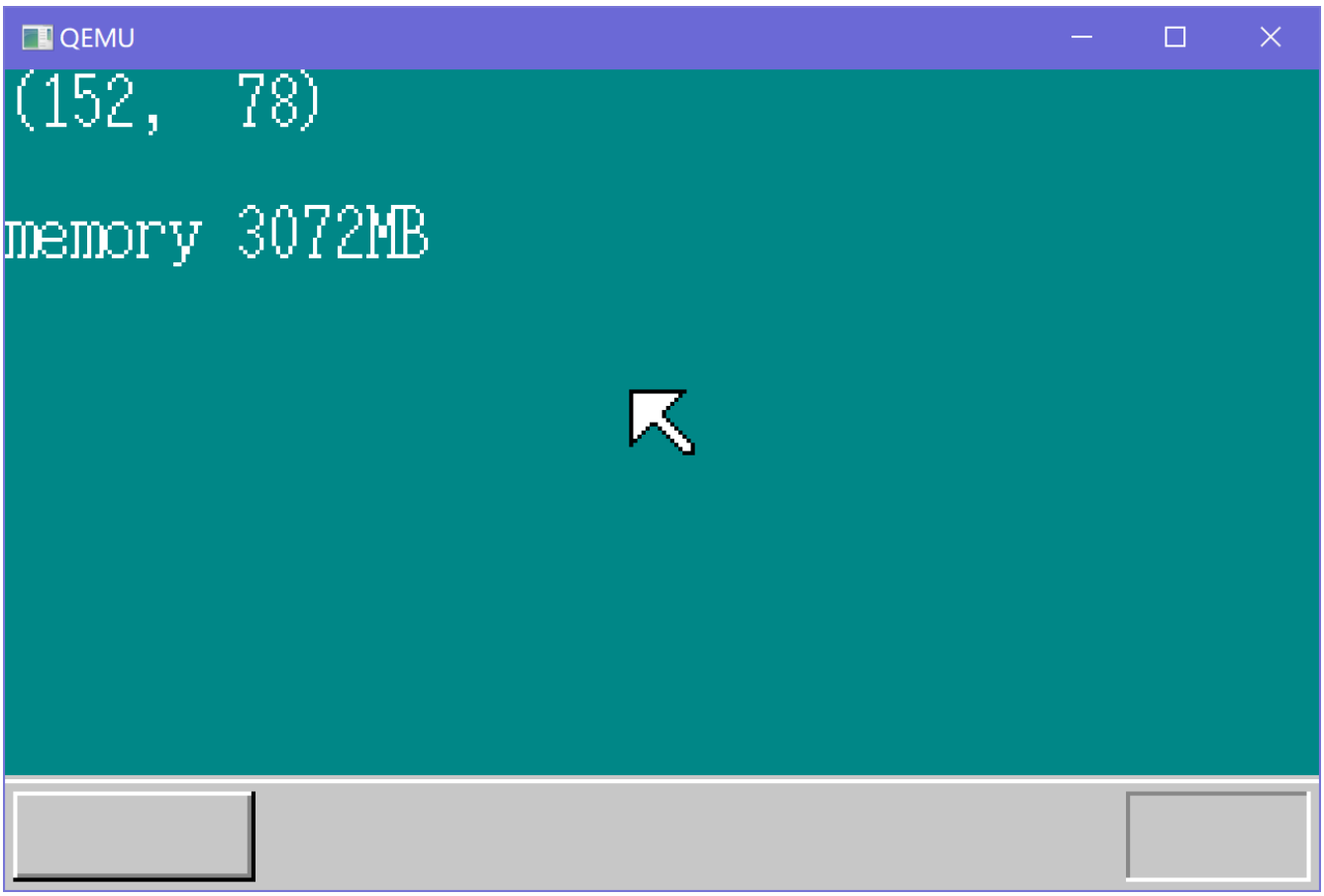
SubPhase 3

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;
}
```