第9天

内存管理

- 整理源文件（harib06a）
- 内存容量检查（1）（harib06b）
- 内存容量检查（2）（harib06c）
- 挑战内存管理（harib06d）

1 整理源文件（harib06a）

现在我们还残留一个问题，就是鼠标指针的叠加处理不太顺利。不过如果一味进行鼠标处理的话，大家可能很容易腻烦，所以我们今天干点儿别的。鼠标指针的叠加处理问题迟早会解决的，大家不用担心，暂时先忘掉这个事情吧。

那么，今天做什么呢？我们今天就做内存管理吧。好不容易变成了32位模式，终于可以使用电脑的全部内存了，大家肯定也想用一用试试吧。

刚想改造bootpack.c，却发现为了解决鼠标处理问题而大加修改程序导致程序变大了很多，足足有182行。嗯，程序太长了，怎么看都不舒服，所以笔者决定将程序整理一下。

本次的程序整理表

函数名	移动前	移动后
wait_KBC_sendready	bootpack.c	keyboard.c
init_keyboard	bootpack.c	keyboard.c
enable_mouse	bootpack.c	mouse.c
mouse_decode	bootpack.c	mouse.c
inthandler21	init.c	keyboard.c
inthandler2c	init.c	mouse.c

要做的事情很简单，仅仅是把函数写到不同的地方而已。此时，如果不知道哪个函数写在什么地方，可就麻烦了，所以在bootpack.h里还要加上函数声明，在Makefile的“OBJS_BOOTPACK=”那里，要将keyboard.obj和mouse.obj也补进去。

我们顺便确认一下运行情况。“make run”，不错不错，还能像以前那样运行。这样bootpack.c就减到了86行。真清爽！

2 内存容量检查（1）（harib06b）

现在我们要进行内存管理了。首先必须要做的事情，是搞清楚内存究竟有多大，范围是到哪里。如果连这一点都搞不清楚的话，内存管理就无从谈起。

在最初启动时，BIOS肯定要检查内存容量，所以只要我们问一问BIOS，就能知道内存容量有多大。但问题是，如果那样做的话，一方面asmhead.nas会变长，另一方面，BIOS版本不同，BIOS函数的调用方法也不相同，麻烦事太多了。所以，笔者想与其如此，不如自己去检查内存。

下面介绍一下做法。

首先，暂时让486以后的CPU的高速缓存（cache）功能无效。回忆一下最初讲的CPU与内存的关系吧。我们说过，内存与CPU的距离地与CPU内部元件要远得多，因此在寄存器内部MOV，要比从寄存器MOV到内存快得多。但另一方面，有一个问题，CPU的记忆力太差了，即使知道内存的速度不行，还不得不频繁使用内存。

考虑到这个问题，英特尔的大叔们在CPU里也加进了一点存储器，它被称为高速缓冲存储器（cache memory）。cache这个词原是指储存粮食弹药等物资的仓库。但是能够跟得上CPU速度的高速存储器价格特别高，一个芯片就有一个CPU那么贵。如果128MB内存全部都用这种高价存储器，预算上肯定受不了。高速缓存，容量只有这个数值的千分之一，也就是128KB左右。高级CPU，也许能有1MB高速缓存，但即便这样，也不过就是128MB的百分之一。

为了有效使用如此稀有的高速缓存，英特尔的大叔们决定，每次访问内存，都要将所访问的地址和内容存入到高速缓存里。也就是存放成这样：18号地址的值是54。如果下次再要用18号地址的内容，CPU就不再读内存了，而是使用高速缓存的信息，马上就能回答出18号地址的内容是54。

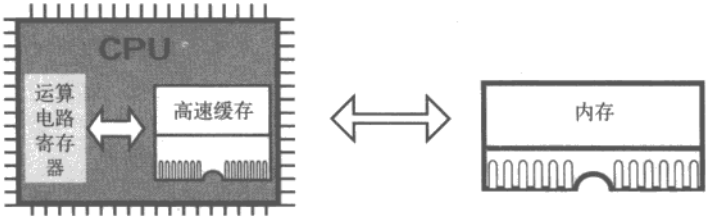
往内存里写入数据时也一样，首先更新高速缓存的信息，然后再写入内存。如果先写入内存的话，在等待写入完成的期间，CPU处于空闲状态，这样就会影响速度。所以，先更新缓存，缓存控制电路配合内存的速度，然后再慢慢发送内存写入命令。

观察机器语言的流程会发现，9成以上的时间耗费在循环上。所谓循环，是指程序在同一个地方来回打转。所以，那个地方的内存要一遍又一遍读进来。从第2圈循环开始，那个地方的内

存信息已经保存到缓存里了，就不需要执行费时的读取内存操作了，机器语言的执行速度因而得以大幅提高。

另外，就算是变量，也会有像“for(i = 0; i < 100; i++){}”这样，i频繁地被引用，被赋值的情况，最初是 0，紧接着是 1，下一个就是 2。也就是说，要往内存的同一个地址，一次又一次写入不同的值。缓存控制电路观察会这一特性，在写入值不断变化的时候，试图不写入缓慢的内存，而是尽量在缓存内处理。循环处理完成，最终i的值变成100以后，才发送内存写入命令。这样，就省略了99次内存写入命令，CPU几乎不用等就能连续执行机器语言。

386的CPU没有缓存，486的缓存只有8-16KB，但两者的性能就差了6倍以上[①]。286进化到386时，性能可没提高这么多。386进化到486时，除了缓存之外还有别的改善，不能光靠缓存来解释这么大的性能差异，但这个性能差异，居然比16位改良到32位所带来的性能差异还要大，笔者认为这主要应该归功于缓存。



内存检查时，要往内存里随便写入一个值，然后马上读取，来检查读取的值与写入的值是否相等。如果内存连接正常，则写入的值能够记在内存里。如果没连接上，则读出的值肯定是乱七八糟的。方法很简单。但是，如果CPU里加上了缓存会怎么样呢？写入和读出的不是内存，而是缓存。结果，所有的内存都“正常”，检查处理不能完成。

所以，只有在内存检查时才将缓存设为OFF。具体来说，就是先查查CPU是不是在486以上，如果是，就将缓存设为OFF。按照这一思路，我们创建了以下函数memtest。

本次的bootpack.c节选

```
#define EFLAGS_AC_BIT       0x00040000
#define CR0_CACHE_DISABLE   0x60000000

unsigned int memtest(unsigned int start, unsigned int end)
{
    char flg486 = 0;
    unsigned int eflg, cr0, i;
```

① 这里用来比较的是386DX-33MHz与486DX4-100MHz（据ICOMP1.0）。

```
    /* 确认CPU是386还是486以上的 */
    eflg = io_load_eflags();
    eflg |= EFLAGS_AC_BIT; /* AC-bit = 1 */
    io_store_eflags(eflg);
    eflg = io_load_eflags();
    if ((eflg & EFLAGS_AC_BIT) != 0) { /* 如果是386，即使设定AC=1，AC的值还会自动回到0 */
        flg486 = 1;
    }
    eflg &= ~EFLAGS_AC_BIT; /* AC-bit = 0 */
    io_store_eflags(eflg);

    if (flg486 != 0) {
        cr0 = load_cr0();
        cr0 |= CR0_CACHE_DISABLE; /* 禁止缓存 */
        store_cr0(cr0);
    }

    i = memtest_sub(start, end);

    if (flg486 != 0) {
        cr0 = load_cr0();
        cr0 &= ~CR0_CACHE_DISABLE; /* 允许缓存 */
        store_cr0(cr0);
    }

    return i;
}
```

最初对EFLAGS进行的处理，是检查CPU是486以上还是386。如果是486以上，EFLAGS寄存器的第18位应该是所谓的AC标志位；如果CPU是386，那么就没有这个标志位，第18位一直是0。这里，我们有意识地把1写入到这一位，然后再读出EFLAGS的值，继而检查AC标志位是否仍为1。最后，将AC标志位重置为0。

将AC标志位重置为0时，用到了AND运算，那里出现了一个运算符“~”，它是取反运算符，就是将所有的位都反转的意思。所以，~EFLAGS_AC_BIT与0xfffbffff一样。

为了禁止缓存，需要对CR0寄存器的某一标志位进行操作。对哪里操作，怎么操作，大家一看程序就能明白。这时，需要用到函数load_cr0和store_cr0，与之前的情况一样，这两个函数不能用C语言写，只能用汇编语言来写，存在naskfunc.nas里。

本次的naskfunc.nas节选

```
_load_cr0:      ; int load_cr0(void);
        MOV     EAX,CR0
        RET

_store_cr0:     ; void store_cr0(int cr0);
        MOV     EAX,[ESP+4]
        MOV     CR0,EAX
        RET
```

另外，memtest_sub函数，是内存检查处理的实现部分。

最开始的memtest_sub

```
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 += 4) {
        p = (unsigned int *) i;
        old = *p;           /* 先记住修改前的值 */
        *p = pat0;          /* 试写 */
        *p ^= 0xffffffff;   /* 反转 */
        if (*p != pat1) {   /* 检查反转结果 */
not_memory:
            *p = old;
            break;
        }
        *p ^= 0xffffffff;   /* 再次反转 */
        if (*p != pat0) {   /* 检查值是否恢复 */
            goto not_memory;
        }
        *p = old;           /* 恢复为修改前的值 */
    }
    return i;
}
```

这个程序所做的是：调查从start地址到end地址的范围内，能够使用的内存的末尾地址。要做的事情很简单。首先如果p不是指针，就不能指定地址去读取内存，所以先执行“p=i;”。紧接着使用这个p，将原值保存下来（变量old）。接着试写0xaa55aa55，在内存里反转该值，检查结果是否正确[①]。如果正确，就再次反转它，检查一下是否能回复到初始值。最后，使用old变量，将内存的值恢复回去。……如果在某个环节没能恢复成预想的值，那么就在那个环节终止调查，并报告终止时的地址。

关于反转，我们用XOR运算来实现，其运算符是“^”。“*p^=0xffffffff;”是“*p=*p^0xffffffff;”的省略形式。

i的值每次增加4是因为每次要检查4个字节。之所以把变量命名为pat0、pat1是因为这些变量表示测试时所用的几种形式。

笔者试着执行了一下这个程序，发现运行速度特别慢，于是就对memtest_sub做了些改良，不过只修改了最初的部分。

① 有些机型即便不进行这种检查也不会有问题。但有些机型因为芯片组和主板电路等原因，如果不做这种检查就会直接读出写入的数据，所以要反转一下。

本次的bootpack.c节选

```
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;           /* 先记住修改前的值 */
```

改变的内容只是for语句中i的增值部分以及p的赋值部分。每次只增加4，就要检查全部内存，速度太慢了，所以改成了每次增加0x1000，相当于4KB，这样一来速度就提高了1000倍。p的赋值计算式也变了，这是因为，如果不进行任何改变仍写作“p=i;”的话，程序就会只检查4KB最开头的4个字节。所以要改为“p=i + 0xffc;”，让它只检查末尾的4个字节。

毕竟在系统启动时内存已经被仔细检查过了，所以像这次这样，目的只是确认容量的话，做到如此程度就足够了。甚至可以说每次检查1MB都没什么问题。

那好，下面我们来改造HariMain。添加的程序如下：

本次的bootpack.c节选

```
    i = memtest(0x00400000, 0xbfffffff) / (1024 * 1024);
    sprintf(s, "memory %dMB", i);
    putfonts8_asc(binfo->vram, binfo->scrnx, 0, 32, COL8_FFFFFF, s);
```

暂时先使用以上程序对0x00400000～0xbfffffff范围的内存进行检查。这个程序最大可以识别3GB范围的内存。0x00400000号以前的内存已经被使用了（参考8.5节的内存分布图），没有内存，程序根本运行不到这里，所以我们没做内存检查。如果以byte或KB为单位来显示结果不容易看明白，所以我们以MB为单位。

也不知道能不能正常运行。如果在QEMU上运行，根据模拟器的设定，内存应该为32MB。运行“make run”。

内存容量怎么不对呀？

哎？怎么回事？内存容量怎么不是32MB，而是3072MB？这不就是3GB吗？为什么会失败

呢？明明已经将缓冲OFF掉了。

3 内存容量检查（2）（harib06c）

这种做法本身没有问题，笔者在OSASK上确认过，所以看到上述结果很纳闷。这种内存检查方法在很多机型上都能运行，所以笔者非常自信地向大家推荐了它。虽然笔者坚信程序没有问题，可运行结果……

经过多方调查，终于搞清楚了原因。如果我们不用“make run”，而是用“make -r bootpack.nas”来运行的话，就可以确认bootpack.c被编译成了什么样的机器语言。用文本编辑器看一看生成的bootpack.nas会发现，最下边有memtest_sub的编译结果。我们将编译结果列在下面。（为了读起来方便，笔者还添加了注释。）

harib06b中，memtest_sub的编译结果

```
_memtest_sub:
    PUSH    EBP                 ; C编译器的固定语句
    MOV EBP,ESP
    MOV EDX,DWORD [12+EBP]      ; EDX = end;
    MOV EAX,DWORD [8+EBP]       ; EAX = start; /* EAX是i */
    CMP EAX,EDX                 ; if (EAX > EDX) goto L30;
    JA  L30
L36:
L34:
    ADD EAX,4096                ; EAX += 0x1000;
    CMP EAX,EDX                 ; if (EAX <= EDX) goto L36;
    JBE L36
L30:
    POP EBP                     ; 接收前文中PUSH的EBP
    RET                         ; return;
```

有些细节大家可能不太明白，但是可以跟memtest_sub比较一下。可以发现，以上的编译结果有点不正常。

harib06b中，memtest_sub的内容

```
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;           /* 先记住修改前的值 */
        *p = pat0;          /* 试写 */
        *p ^= 0xffffffff;   /* 反转 */
        if (*p != pat1) {   /* 检查反转结果 */
not_memory:
            *p = old;
            break;
```

```
        }
        *p ^= 0xffffffff;    /* 再次反转 */
        if (*p != pat0) {    /* 检查值是否恢复 */
            goto not_memory;
        }
        *p = old;            /* 恢复为修改前的值 */
    }
    return i;
}
```

大家会发现，编译后没有XOR等指令，而且，好像编译后只剩下了for语句。怪不得显示结果是3GB呢。但是，为什么会这样呢？

笔者开始以为这是C编译器的bug，但仔细查一查，发现并非如此。反倒是编译器太过优秀了。

编译器在编译时，应该是按下面思路考虑问题的。

首先将内存的内容保存到old里，然后写入pat0的值，再反转，最后跟pat1进行比较。这不是肯定相等的吗？if语句不成立，得不到执行，所以把它删掉。怎么？下面还要反转吗？这家伙好像就喜欢反转。这次是不是要比较*p和pat0呀？这不是也肯定相等吗？这些处理不是多余么？为了提高速度，将这部分也删掉吧。这样一来，程序就变成了：

编译器脑中所想的（1）

```
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;            /* 先记住修改前的值*/
        *p = pat0;           /* 试写 */
        *p ^= 0xffffffff;    /* 反转 */
        *p ^= 0xffffffff;    /* 再次反转 */
        *p = old;            /* 恢复为修改前的值 */
    }
    return i;
}
```

反转了两次会变回之前的状态，所以这些处理也可以不要嘛。因此程序就变成了这样：

编译器脑中所想的（2）

```
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;           /* 先记住修改前的值 */
        *p = pat0;          /* 试写 */
        *p = old;           /* 恢复为修改前的值 */
    }
    return i;
}
```

还有，“*p＝pat0;”本来就没有意义嘛。反正要将old的值赋给*p。因此程序就变成了：

编译器脑中所想的（3）

```
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;           /* 先记住修改前的值 */
        *p = old;           /* 恢复为修改前的值 */
    }
    return i;
}
```

这程序是什么嘛？结果，*p里面不是没写进任何内容吗？这有什么意义？

编译器脑中所想的（4）

```
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);
    }
    return i;
}
```

这里的地址变量p，虽然计算了地址，却一次也没有用到。这么说来，old、pat0、pat1也都是用不到的变量。全部都舍弃掉吧。

编译器脑中所想的（5）

```
unsigned int memtest_sub(unsigned int start, unsigned int end)
{
    unsigned int i;
    for (i = start; i <= end; i += 0x1000) { }
    return i;
}
```

好了，这样修改后，速度能提高许多。用户肯定会说：“这编译器真好，速度特别快！”

根据以上编译器的思路，我们可以看出，它进行了最优化处理。但其实这个工作本来是不需要的。用于应用程序的C编译器，根本想不到会对没有内存的地方进行读写。

如果更改编译选项，是可以停止最优化处理的。可是在其他地方，我们还是需要如此考虑周密的最优化处理的，所以不想更改编译选项。那怎样来解决这个问题呢？想来想去，还是觉得很麻烦，于是决定memtest_sub也用汇编来写算了。

这次C编译器只是好心干了坏事，但意外的是，它居然会考虑得如此周到、缜密来进行最优化处理……这个编译器真是聪明啊！顺便说一句，这种事偶尔还会有的，所以能够看见中途结果很有用。而且懂汇编语言很重要。

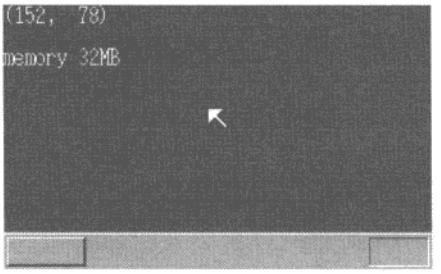
笔者用汇编语言写的程序列举如下。

本次的naskfunc.nas节选

```
_memtest_sub:   ; unsigned int memtest_sub(unsigned int start, unsigned int end)
        PUSH    EDI                     ;   (由于还要使用EBX, ESI, EDI)
        PUSH    ESI
        PUSH    EBX
        MOV     ESI,0xaa55aa55          ; pat0 = 0xaa55aa55;
        MOV     EDI,0x55aa55aa          ; pat1 = 0x55aa55aa;
        MOV     EAX,[ESP+12+4]          ; i = start;
mts_loop:
        MOV     EBX,EAX
        ADD     EBX,0xffc               ; p = i + 0xffc;
        MOV     EDX,[EBX]               ; old = *p;
        MOV     [EBX],ESI               ; *p = pat0;
        XOR     DWORD [EBX],0xffffffff  ; *p ^= 0xffffffff;
        CMP     EDI,[EBX]               ; if (*p != pat1) goto fin;
        JNE     mts_fin
        XOR     DWORD [EBX],0xffffffff  ; *p ^= 0xffffffff;
        CMP     ESI,[EBX]               ; if (*p != pat0) goto fin;
        JNE     mts_fin
        MOV     [EBX],EDX               ; *p = old;
        ADD     EAX,0x1000              ; i += 0x1000;
        CMP     EAX,[ESP+12+8]          ; if (i <= end) goto mts_loop;
       JBE     mts_loop
        POP     EBX
        POP     ESI
        POP     EDI
        RET
mts_fin:
        MOV     [EBX],EDX               ; *p = old;
        POP     EBX
        POP     ESI
        POP     EDI
        RET
```

笔者好久没写过这么长的汇编程序了。程序里加上了足够的注释，应该很好懂。虽然XOR指令（异或）是第一次出现，不过不用特别解释大家也应该能明白。

那好，我们删除bootpack.c中的memtest_sub函数，运行一下看看。“make run”。结果怎么样呢？



内存容量显示正常了

太好了！现在可以回到内存管理这个正题上来了。

4 挑战内存管理（harib06d）

刚才笔者一个劲儿地说内存管理长，内存管理短的，那到底什么是内存管理呢？为什么要进行内存管理呢？

比如说，假设内存大小是128MB，应用程序A暂时需要100KB，画面控制需要1.2MB……，像这样，操作系统在工作中，有时需要分配一定大小的内存，用完以后又不再需要，这种事会频繁发生。为了应付这些需求，必须恰当管理好哪些内存可以使用（哪些内存空闲），哪些内存不可以使用（正在使用），这就是内存管理。如果不进行管理，系统会变得一塌糊涂，要么不知道哪里可用，要么多个应用程序使用同一地址的内存。

内存管理的基础，一是内存分配，一是内存释放。“现在要启动应用程序B了，需要84KB内存，哪儿空着呢？”如果问内存管理程序这么一个问题，内存管理程序就会给出一个能够自由使用的84KB的内存地址，这就是内存分配。另一方面，“内存使用完了，现在把内存归还给内存管理程序”，这一过程就是内存的释放过程。

先假设有128MB的内存吧。也就是说，有0x08000000个字节。另外我们假设以0x1000个字节（4KB）为单位进行管理。大家会如何管理呢？答案有很多，我们从简单的方法开始介绍。

0x08000000/0x1000＝0x08000＝32768，所以首先我们来创建32768字节的区域，可以往其中写入0或者1来标记哪里是空着的，哪里是正在使用的。

```
    char a[32768];
    for (i = 0; i < 1024; i++) {
        a[i] = 1; /* 一直到4MB为止，标记为正在使用 */
    }
    for (i = 1024; i < 32768; i++) {
        a[i] = 0; /* 剩下的全部标记为空 */
    }
```

比如需要100KB的空间，那么只要从a中找出连续25个标记为0的地方就可以了。

```
    j = 0;
再来一次:
    for (i = 0; i < 25; i++) {
        if (a[j + i] != 0) {
            j++;
            if (j < 32768 - 25) goto 再来一次;
            "没有可用内存了";
        }
    }
    "从a[j]到a[j + 24]为止，标记连续为0";
```

如果找到了标记连续为0的地方，暂时将这些地方标记为“正在使用”，然后从j的值计算出对应的地址。这次是以0x1000字节为管理单位的，所以将j放大0x1000倍就行了。

```
    for (i = 0; i < 25; i++) {
        a[j + i] = 1;
    }
    "从 j * 0x1000 开始的100KB空间得到分配";
```

如果要释放这部分内存空间，可以像下面这样做。比如，如果遇到这种情况：“刚才取得的从0x00123000开始的100KB，已经不用了，现在归还。谢谢你呀。”那该怎么办呢？用地址值除以0x1000，计算出j就可以了。

```
    j = 0x00123000 / 0x1000;
    for (i = 0; i < 25; i++) {
        a[j + i] = 0;
    }
```

很简单吧。以后再有需要内存的时候，这个地方又可以再次被使用了。

上面这个方法虽然很好懂，但是有一点问题。如果内存是128MB，管理表只需要32768字节（32KB）；如果内存最大是3GB，管理表是多大呢？0xc0000000 / 0x1000 = 0xc0000 = 786432，也

就是说光管理表就需要768KB。当然，虽说768KB不小，但从3GB看来，只不过是0.02%。

事实上，0.02%的比例是与容量没有关系的。用32KB管理128MB时，比例也是0.02%。如果容量是个问题，这个管理表可以不用char来构成，而是使用位（bit）来构成。归根到底，储存的只有0和1，用不了一个字节，一位就够了。这样做，程序会变得复杂些，但是管理表的大小可缩减到原来的1/8。如果是3GB内存，只需要96KB就可以管理整个内存了。这个比例只有0.003%。

我们后面还会讲到，这虽然不是最好的方法，但Windows的软盘管理方法，与这个方法很接近（1.44MB的容量，以512字节为单位分块管理）。

除了这个管理方法之外，还有一种列表管理的方法，是把类似于“从xxx号地址开始的yyy字节的空间是空着的”这种信息都列在表里。

```
struct FREEINFO {   /* 可用状况 */
    unsigned int addr, size;
};

struct MEMMAN {     /* 内存管理 */
    int frees;
    struct FREEINFO free[1000];
};

    struct MEMMAN memman;
    memman.frees = 1; /* 可用状况list中只有1件 */
    memman.free[0].addr = 0x00400000;   /* 从0x00400000号地址开始，有124MB可用 */
    memman.free[0].size = 0x07c00000;
```

大体就是这个样子。之所以有1000个free，是考虑到即使可用内存部分不连续，我们也能写入到这1000个free里。memman是笔者起的名字，代表memory manager。

比如，如果需要100KB的空间，只要查看memman中free的状况，从中找到100MB以上的可用空间就行了。

```
    for (i = 0; i < memman.frees; i++) {
        if (memman.free[i].size >= 100 * 1024) {
            "找到可用空间！";
            "从地址memman.free[i].addr开始的100KB空间，可以使用哦！";
        }
    }
    "没有可用空间";
```

如果找到了可用内存空间，就将这一段信息从“可用内存空间管理表”中删除。这相当于给这一段内存贴上了“正在使用”的标签。

```
memman.free[i].addr += 100 * 1024; /* 可用地址向后推进了100KB */
memman.free[i].size -= 100 * 1024; /* 减去100KB */
```

如果size变成了0，那么这一段可用信息就不再需要了，将这条信息删除，frees减去1就可以了。

释放内存时，增加一条可用信息，frees加1。而且，还要调查一下这段新释放出来的内存，与相邻的可用空间能不能连到一起。如果能连到一起，就把它们归纳为一条。

能够归纳为一条的例子：

```
free[0]:地址0x00400000号开始，0x00019000字节可用
free[1]:地址0x00419000号开始，0x07be7000字节可用
```

可以归纳为

```
free[0]:地址0x00400000号开始，0x07c00000字节可用
```

如果不将它们归纳为一条，以后系统要求“请给我提供0x07bf0000字节的内存”时，本来有这么多的可用空间，但以先前的查找程序却会找不到。

上述新方法的优点，首先就是占用内存少。memman是8×1000 + 4 = 8004，还不到8KB。与上一种方法的32KB相比，差得可不少。而且，这里的1000是个很充裕的数字。可用空间不可能如此零碎分散（当然，这与内存的使用方法有关）。所以，这个数字或许能降到100。这样的话，只要804字节就能管理128MB的内存了。

如果用这种新方法，就算是管理3GB的内存，也只需要8KB左右就够了。当然，可用内存可能更零碎些，为了安全起见，也可以设定10000条可用区域管理信息。即使这样也只有80KB。

这样新方法，还有其他优点，那就是大块内存的分配和释放都非常迅速。比如我们考虑分配10MB内存的情形。如果按前一种方法，就要写入2560个“内存正在使用”的标记“1”，而释放内存时，要写入2560个“0”。这些都需要花费很长的时间。

另一方面，这种新方法在分配内存时，只要加法运算和减法运算各执行一次就结束了。不管是10MB也好，100MB也好，还是40KB，任何情况都一样。释放内存的时候虽然没那么快，但是与写入2560个“0”相比，速度快得可以用“一瞬间”来形容。

事情总是有两面性的，占用内存少，分配和释放内存速度快，现在看起来全是优点；但是实际上也有缺点，首先是管理程序变复杂了。特别是将可用信息归纳到一起的处理，变得相当复杂。

还有一个缺点是，当可用空间被搞得零零散散，怎么都归纳不到一块儿时，会将1000条可用

空间管理信息全部用完。虽然可以认为这几乎不会发生，但也不能保证绝对不能发生。这种情况下，要么做一个更大的MEMMAN，要么就只能割舍掉小块内存。被割舍掉的这部分内存，虽然实际上空着，但是却被误认为正在使用，而再也不能使用。

为了解决这一问题，实际上操作系统想尽了各种办法。有一种办法是，暂时先割舍掉，当memman有空余时，再对使用中的内存进行检查，将割舍掉的那部分内容再捡回来。还有一种方法是，如果可用内存太零碎了，就自动切换到之前那种管理方法。

那么，我们的“纸娃娃系统”（haribote OS）会采用什么办法呢？笔者经过斟酌，采用了这样一种做法，即“割舍掉的东西，只要以后还能找回来，就暂时不去管它。”。如果我们陷在这个问题上不能自拔，花上好几天时间，大家就会厌烦的。笔者还是希望大家能开开心心地开发“纸娃娃系统”。而且万一出了问题，到时候我们再回过头来重新修正内存管理程序也可以。

根据这种思路，笔者首先创建了以下程序。

本次的bootpack.c节选

```
#define MEMMAN_FREES        4090    /* 大约是32KB*/

struct FREEINFO {   /* 可用信息 */
    unsigned int addr, size;
};

struct MEMMAN {     /* 内存管理 */
    int frees, maxfrees, lostsize, losts;
    struct FREEINFO free[MEMMAN_FREES];
};

void memman_init(struct MEMMAN *man)
{
    man->frees = 0;         /* 可用信息数目 */
    man->maxfrees = 0;      /* 用于观察可用状况：frees的最大值 */
    man->lostsize = 0;      /* 释放失败的内存的大小总和 */
    man->losts = 0;         /* 释放失败次数 */
    return;
}

unsigned int memman_total(struct MEMMAN *man)
/* 报告空余内存大小的合计 */
{
    unsigned int i, t = 0;
    for (i = 0; i < man->frees; i++) {
        t += man->free[i].size;
    }
    return t;
}
```

```
unsigned int memman_alloc(struct MEMMAN *man, unsigned int size)
/* 分配 */
{
    unsigned int i, a;
    for (i = 0; i < man->frees; i++) {
        if (man->free[i].size >= size) {
            /* 找到了足够大的内存 */
            a = man->free[i].addr;
            man->free[i].addr += size;
            man->free[i].size -= size;
            if (man->free[i].size == 0) {
                /* 如果free[i]变成了0，就减掉一条可用信息 */
                man->frees--;
                for (; i < man->frees; i++) {
                    man->free[i] = man->free[i + 1]; /* 代入结构体 */
                }
            }
            return a;
        }
    }
    return 0; /* 没有可用空间 */
}
```

一开始的struct MEMMAN，只有1000组的话，可能不够。所以，我们创建了4000组，留出不少余量。这样一来，管理空间大约是32KB。其中还有变量maxfrees、lostsize、losts等，这些变量与管理本身没有关系，不用在意它们。如果特别想了解的话，可以看看函数memman_init的注释，里面有介绍。

函数memman_init对memman进行了初始化，设定为空。主要工作，是将frees设为0，而其他的都是附属性设定。这里的init，是initialize（初始化）的缩写。

函数memman_total用来计算可用内存的合计大小并返回。笔者觉得有这个功能应该很方便，所以就创建了这么一个函数。原理很简单，不用解释大家也会明白。total这个英文单词，是“合计”的意思。

最后的memman_alloc函数，功能是分配指定大小的内存。除了free[i].size变为0时的处理以外的部分，在前面已经说过了。alloc是英文allocate（分配）的缩写。在编程中，需要分配内存空间时，常常会使用allocate这个词。

memman_alloc函数中free[i].size等于0的处理，与FIFO缓冲区的处理方法很相似，要进行移位处理。希望大家注意以下写法：

```
man->free[i].addr = man->free[i+1].addr;
man->free[i].size = man->free[i+1].size;
```

我们在这里将其归纳为了：

```
man->free[i] = man->free[i+1];
```

这种方法被称为结构体赋值，其使用方法如上所示，可以写成简单的形式。

释放内存函数，也就是往memman里追加可用内存信息的函数，稍微有点复杂。

本次的bootpack.c节选

```
int memman_free(struct MEMMAN *man, unsigned int addr, unsigned int size)
/* 释放 */
{
    int i, j;
    /* 为便于归纳内存，将free[]按照addr的顺序排列 */
    /* 所以，先决定应该放在哪里 */
    for (i = 0; i < man->frees; i++) {
        if (man->free[i].addr > addr) {
            break;
        }
    }
    /* free[i - 1].addr < addr < free[i].addr */
    if (i > 0) {
        /* 前面有可用内存 */
        if (man->free[i - 1].addr + man->free[i - 1].size == addr) {

            /* 可以与前面的可用内存归纳到一起 */
            man->free[i - 1].size += size;
            if (i < man->frees) {
                /* 后面也有 */
                if (addr + size == man->free[i].addr) {
                    /* 也可以与后面的可用内存归纳到一起 */
                    man->free[i - 1].size += man->free[i].size;
                    /* man->free[i]删除  */
                    /* free[i]变成0后归纳到前面去 */
                    man->frees--;
                    for (; i < man->frees; i++) {
                        man->free[i] = man->free[i + 1]; /* 结构体赋值 */
                    }
                }
            }
            return 0; /* 成功完成 */
        }
    }
    /* 不能与前面的可用空间归纳到一起 */
    if (i < man->frees) {
        /* 后面还有 */
        if (addr + size == man->free[i].addr) {
            /* 可以与后面的内容归纳到一起 */
            man->free[i].addr = addr;
            man->free[i].size += size;
            return 0; /* 成功完成 */
        }
    }
```

```
    /* 既不能与前面归纳到一起，也不能与后面归纳到一起 */
    if (man->frees < MEMMAN_FREES) {
        /* free[i]之后的，向后移动，腾出一点可用空间 */
        for (j = man->frees; j > i; j--) {
            man->free[j] = man->free[j - 1];
        }
        man->frees++;
        if (man->maxfrees < man->frees) {
            man->maxfrees = man->frees; /* 更新最大值 */
        }
        man->free[i].addr = addr;
        man->free[i].size = size;
        return 0; /* 成功完成 */
    }
    /* 不能往后移动 */
    man->losts++;
    man->lostsize += size;
    return -1; /* 失败 */
}
```

程序太长了，用文字来描述不易于理解，所以笔者在程序里加了注释。如果理解了以前讲解的原理，现在只要细细读一读程序，大家肯定能看懂。

另外，我们前面已经说过，如果可用信息表满了，就按照舍去之后带来损失最小的原则进行割舍。但是在这个程序里，我们并没有对损失程度进行比较，而是舍去了刚刚进来的可用信息，这只是为了图个方便。

最后，将这个程序应用于HariMain，结果就变成了下面这样。写着“（中略）”的部分，笔者没做修改。

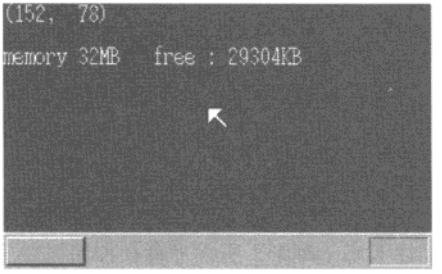
本次的bootpack.c节选

```
#define MEMMAN_ADDR         0x003c0000

void HariMain(void)
{
    （中略）
    unsigned int memtotal;
    struct MEMMAN *memman = (struct MEMMAN *) MEMMAN_ADDR;
    （中略）
    memtotal = memtest(0x00400000, 0xbfffffff);
    memman_init(memman);
    memman_free(memman, 0x00001000, 0x0009e000); /* 0x00001000 - 0x0009efff */
    memman_free(memman, 0x00400000, memtotal - 0x00400000);
    （中略）
    sprintf(s, "memory %dMB   free : %dKB",
            memtotal / (1024 * 1024), memman_total(memman) / 1024);
    putfonts8_asc(binfo->vram, binfo->scrnx, 0, 32, COL8_FFFFFF, s);
```

memman需要32KB，我们暂时决定使用自0x003c0000开始的32KB（0x00300000号地址以后，今后的程序即使有所增加，预计也不会到达0x003c0000，所以我们使用这一数值），然后计算内存总量memtotal，将现在不用的内存以0x1000个字节为单位注册到memman里。最后，显示出合计可用内存容量。在QEMU上执行时，有时会注册成632KB和28MB。632+28672=29304，所以屏幕上会显示出29304KB。

那好，运行一下“make run”看看。哦，运行正常。今天已经很晚了，我们明天继续吧。



能正常显示出29304KB