# 操作系统实验日志

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学号 | 201808010718 | 姓名 | 肖鹏 | 专业年级班级 | 智能1802 |
| 实验日期 | 2020.11.18 | 实验项目 | 第9天：内存管理 | | |

## 一、实验主要内容

1、内存容量检查

**内容：**

**a）**预处理——检测CPU型号：由于检查内存的方式是——向一个内存单元写入一个 值，然后马上取出这个内存里的值，来检查写入和读出的值是否一致。如果使用缓存 的话，那么“写入”和“读出”都将是对缓存操作的，这样会导致一个假象——“所 有的值都是符合的，即所有的内存单元都是正常可用的”，那么将失去内存检查的预期 结果；所以我们在进行内存检查之前需要判断CPU型号（386下没有cache，486有）， 然后把有cache机制的CPU，做相应的禁用cache处理；

**b）**内存检查部分——：通过对[start , end]范围内的内存进行检测，原理是：①取出地 址内的原始值（以便后续对数据恢复）②向这个地址写入一个特殊的值0xaa55aa55 ③ 将内存中的这个值反转，检查是否能够得到0x55aa55aa ④如果上一步正确，再次反 转，检测是否能够得到初始值0xaa55aa55 ⑤如果上述任何一部出现错误，终止检测并 且返回出错时的地址指针；否则说明这一区域内的内存使用正常

最初的代码运行的比较慢，这是因为每次只对4-bytes的内存进行了检测；为了加快速 度，我们在这一步不需要细致到每一个地址都检查，所以增大每次检测的步长——每 次检查0x1000-bytes，也就是4KB的大小，如果这4KB的最后4-bytes通过我们上述的 检测，就说明这4KB内存是可以的；

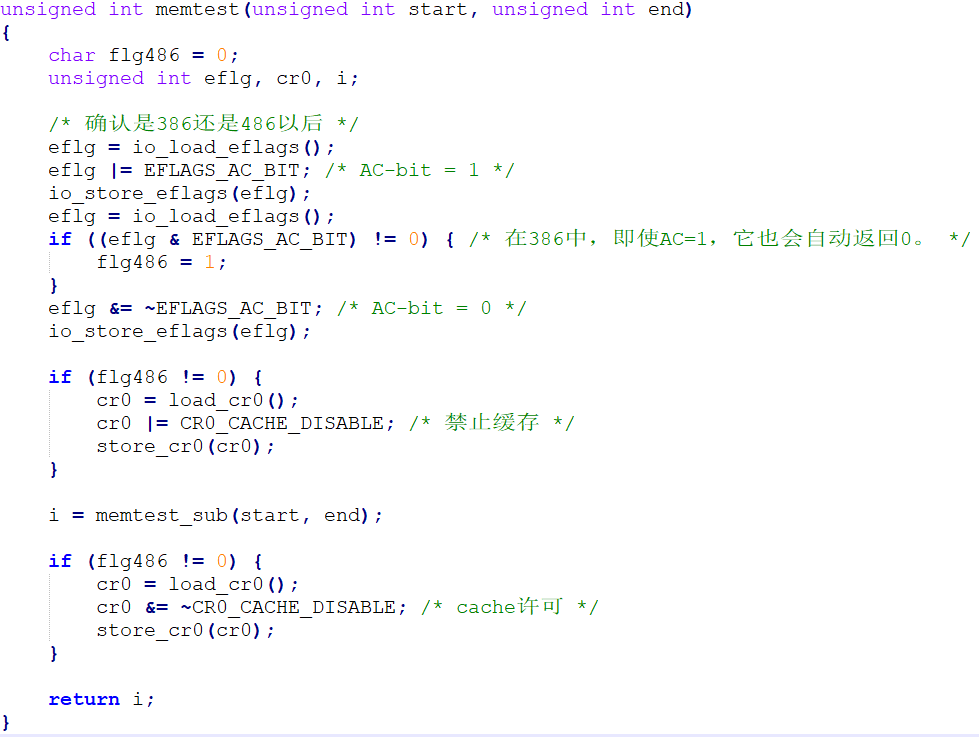
**c）**C语言失败——转汇编：由于上述方法写出来的C代码不能跑出正确的结果，对其 编译后生成的代码检查后发现，我们检测的过程被编译器当作了完全无效的代码块， 所以编译器在优化编译的时候，直接我们检测内存的部分删去了，只留下了一个for循 环的空壳，所以自然不能得到正常结果；

**重点总结：**

**a）**EFLAGS寄存器中第18位为AC标志位：如果CPU是386，则没有该位，第18位一 直为0；判断CPU型号的原理是：将AC标志位置位，如果是386的CPU，那么就算被 置位，AC的值还会自动变成0；否则就是486；之后对于486的CPU，通过对CR0寄 存器的值修改，实现禁止缓存；

**b）**在系统启动的时候，内存已经被仔细检查过了，所以我们在刚刚检测的那一段对步 长的增加是合理的，毕竟我们的目的只是检测容量，甚至可以每次检查1MB都没什么 问题；

**关键代码及注释：**



*（bootpack.c中，关闭cache）*



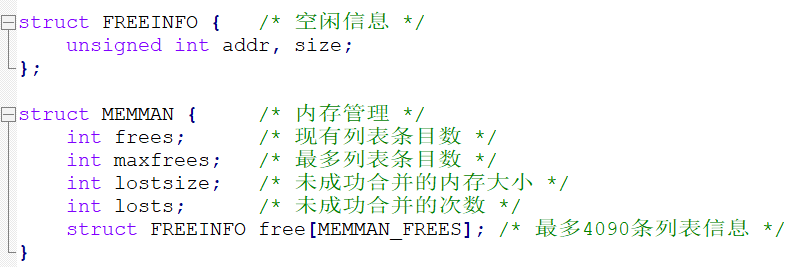
*（naskfunc.asm中，汇编代码写的检测内存）*

2、内存管理

**内容：**

**a）**内存管理使用的方法：笔者比较了位图法和列表法；位图法就是把内存按4KB为 单位，每个4KB都只用1-bit的二进制数表示使用情况；而列表法中保存的是每一个 空闲块的起始地址和大小；由于位图法在大内存分配和释放的时候，需要修改的位数 很多，而不像列表只需要修改一次起始地址和大小就能够完成一大块内存的分配或释 放；所以选择的是后者，但列表中也存在一些棘手的问题——当表满了之后，释放出 来的内存怎么记录？还有，如何及时地合并被释放的内存？

**b）**列表数据结构：



**c）**内存分配：在使用列表法管理内存的模式内，堆内存分配按以下流程：①从头开 始，搜索空闲条目，查看每个空闲条目记录的空闲空间大小是否大于所需分配的大 小；②如果不满足，则继续遍历空闲条目，直到找到一个足够大的区域；③如果找到 了，则分配，并且将该条目的size和起始地址addr都做相应修改；特殊的，当某个 空闲条目所记录的空闲块被完全分配完，则直接删除这一个条目，frees数减一，然 后把所有在后面的free条目前移；

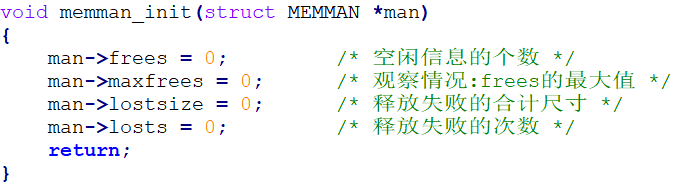
**d）**内存释放：使用完的内存，我们需要将其整理，重新登记到列表中；对于一般的 情况下，对于所释放的内存，我们优先采用将其与现有空闲条目合并；也就是， 如 果这一块释放内存前面，有被记录在列表中的空闲块，①而且这个空闲块刚好与释放 内存连在一起——空闲内存终止地址=释放内存起始地址，那么我们就把这两块内存 合并，然后更新空闲条目；②如果释放内存后面有紧挨着的空闲内存，那么可以再次 进行合并（这里可能是（释放）+（后面的空闲）、也可能是（前面的空闲）+（释 放）+（后面的空闲））；③如果前后都没有紧挨着的，那就只能委屈记录空闲地址的 起始地址大于释放内存的起始地址的那些空闲条目，只能让它们往后移一个空间，以 便新创一个空闲条目给这一块孤独的释放内存。④当然，有的时候并不是后面的空闲 条目不想“让位”，而是实在让不了了，后面已经顶着限制的最大条目数了，那没有 办法，我们只能舍弃这一块内存，但是还是有点可惜，怎么办呢，那就把这次丢的内 存大小给先记录一下，然后再把一共这样舍弃了多少次内存也记录一下，以便后面有 能力了，再把这些内存捡起来！

重点总结：

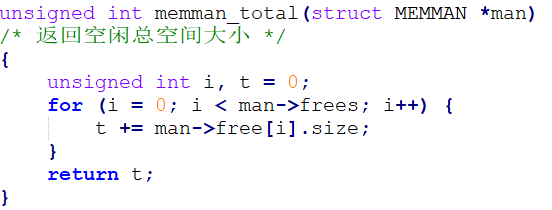
**a）**使用位图法，虽然更简洁，每个4KB只需要一个位来表示，用一个bit来存储区 域是否被使用，这样空间开销是容量的0.003%。但是这样也会造成后续使用的一些 麻烦，比如大容量的分配、释放时，需要写很多位，比列表法要慢许多；

**b）**列表法的优点： 1、占用内存小，假如分1000个条目，那么内存管理仅 需要 1000\*8+4=8004个字节。2、处理更快，占用内存时，只需要加法（原地址加上使用 的空间）和减法运算（size减去使用的空间）各执行一次，释放空间也比逐一将0写 入内存快得多。 缺点： 会造成外部碎片，也就是使用这种方法时间变长，内存就会被分的零零散 散。

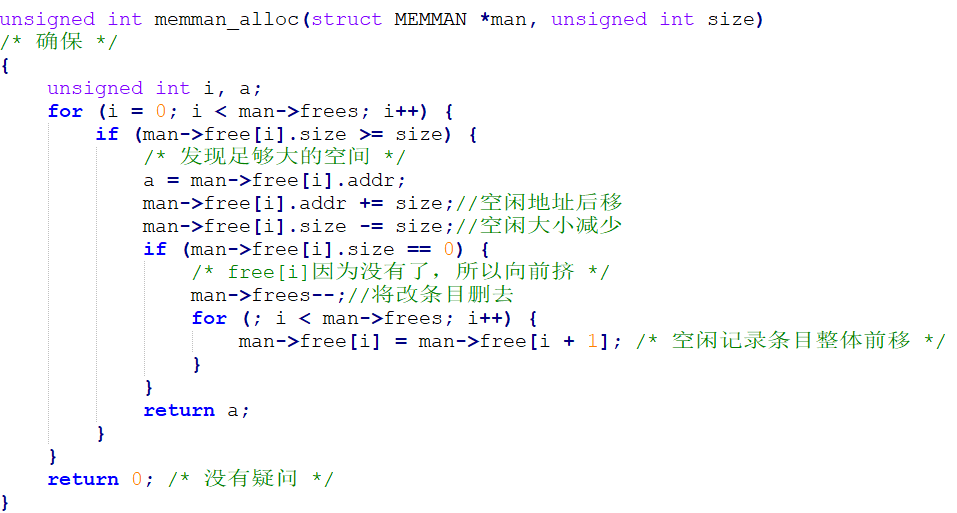
**关键代码及注释：**



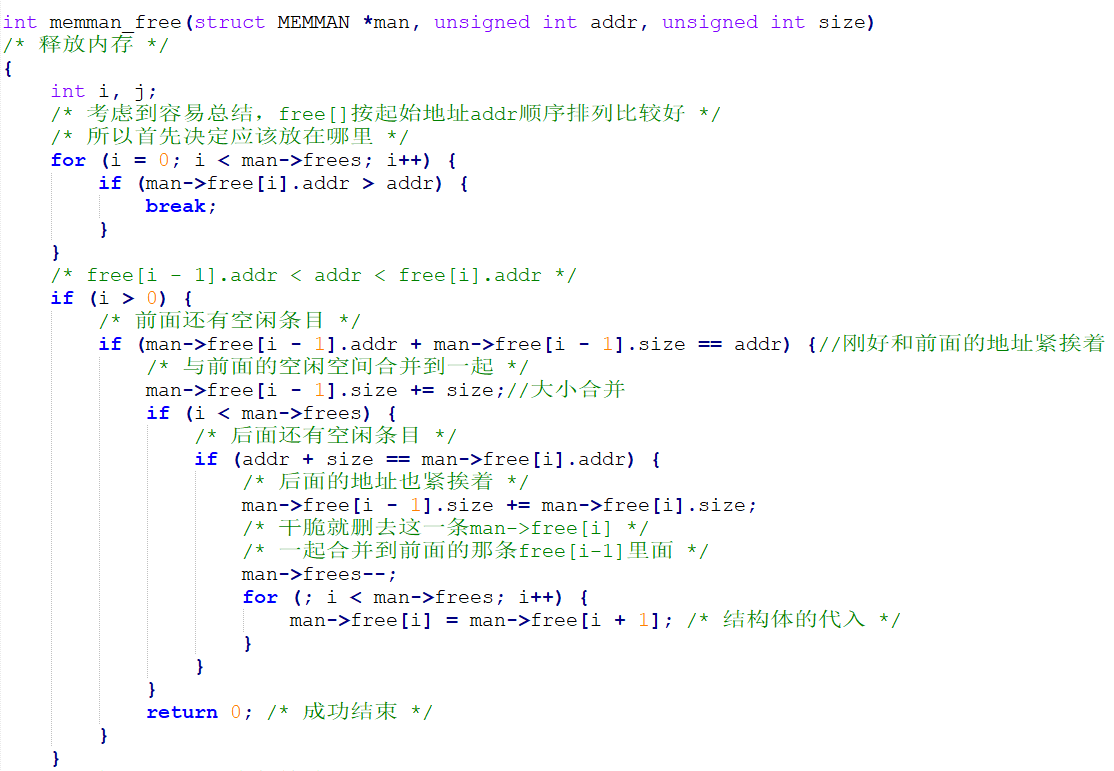
*（初始化函数）*



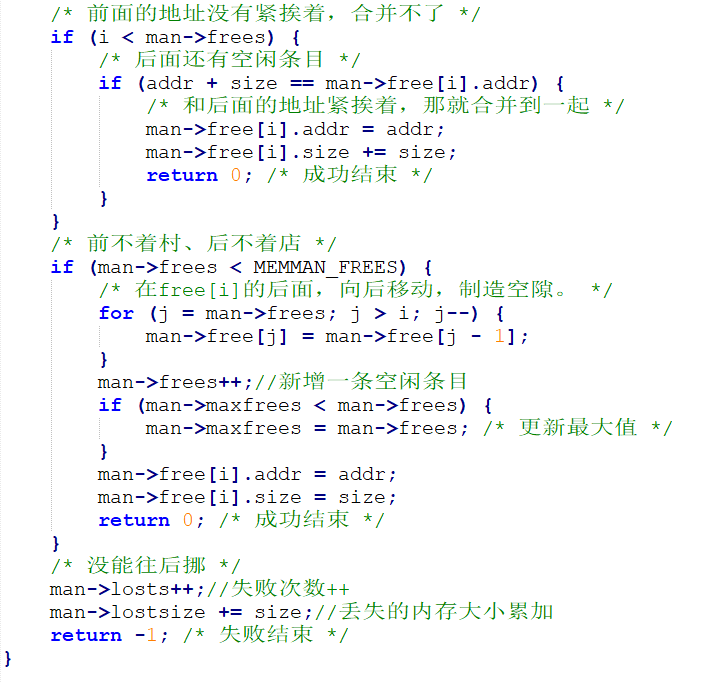
*（返回总空闲大小）*



*（内存分配）*



*（释放内存（1））*



*（释放内存（2））*

## 二、遇到的问题及解决方法

1、

问题描述：在memtest\_sub函数中，检查是否写入到内存时，为什么要先将写入的数据反 转在检查比较？

解决方法：需要写入数据再反转是因为有些机型由于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。

2、

问题描述：一次只检查4kb内存的最后4个地址，不免会有疑问，如果不可使用的内存正 好处于这4kb之间怎么办？

解决方法：其实无需担心这个问题，因为在系统启动的时候内存已经被检查过了，像存储 了bios这样的我们不能使用的内存都是连续的而且占用比较大，所以我们能使用的部分也 是一整块，我们检测的其实就是它的上界和下界。

## 三、程序设计创新点

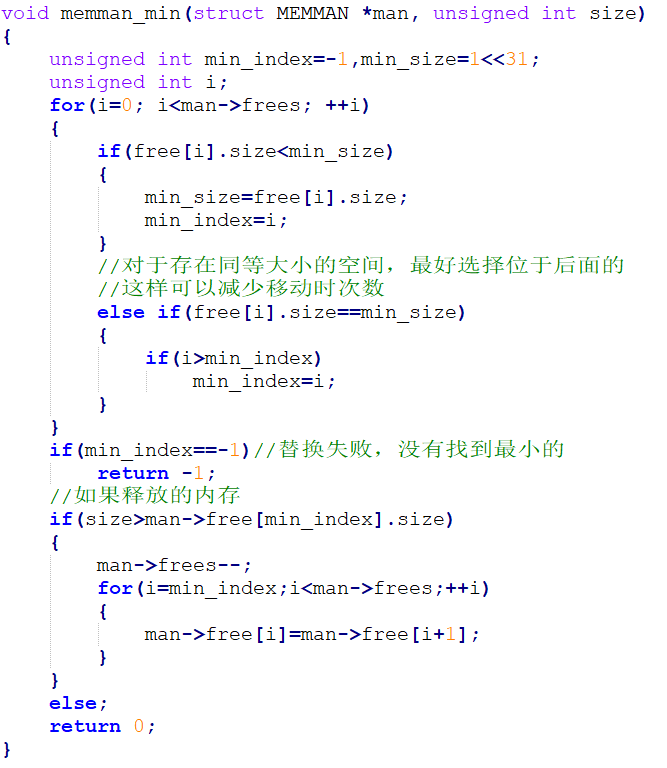
1、

**创新点：**在释放空间的函数中，使用替换策略，替换方法是当列表已满时，调用 memman\_min函数寻找当前列表中空间大小最小的一个条目，然后比较与我们将要释放的 内存的大小关系；如果我们将要释放的内存更大，则换出这个已有的条目，腾出空间为我 们将要释放的空间；反之不处理；

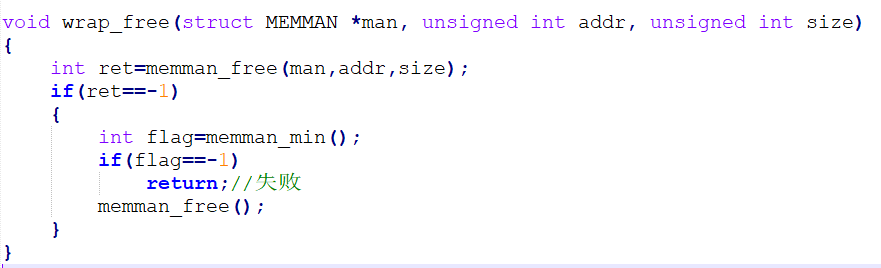
在memman\_min函数实现中，稍微有一点优化的地方，就是当存在相同大小的最小内存 时，我们偏向选择位置更后的那一个，因为这样能使我们在做列表前移的时候次数减少；

最后我们实现一个包装函数wrap\_free，用它来代替memman\_free，这样当出现可以替换 的情况时，它将自动为我们处理；

**实现代码：**



*（memman\_min函数）*



*（wrap\_free函数）*

**效果截图：**



*（正常运行）*

## 四、实验心得体会

这次试验主要还是以概念层面上的内容为主，通过对内存检测、内存管理的分配、释放，加深了我对内存的理解；当然这次试验采用的内存管理方式还是比较简单的，没有在操作系统课上学的那么复杂，所以总体来讲还是能够理解的。