练习

对实验报告的要求：

- 基于markdown格式来完成，以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 完成实验后，请分析ucore_lab中提供的参考答案，并请在实验报告中说明你的实现与参考答案的区别
- 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）
- 列出你认为OS原理中很重要，但在实验中没有对应上的知识点

练习0：填写已有实验

本实验依赖实验1。请把你做的实验1的代码填入本实验中代码中有“LAB1”的注释相应部分并按照实验手册进行进一步的修改。具体来说，就是跟着实验手册的教程一步步做，然后完成教程后继续完成完成exercise部分的剩余练习。

练习1：理解first-fit 连续物理内存分配算法（思考题）

first-fit 连续物理内存分配算法作为物理内存分配一个很基础的方法，需要同学们理解它的实现过程。请大家仔细阅读实验手册的教程并结合`kern/mm/default_pmm.c`中的相关代码，认真分析default_init，default_init_memmap，default_alloc_pages， default_free_pages等相关函数，并描述程序在进行物理内存分配的过程以及各个函数的作用。
请在实验报告中简要说明你的设计实现过程。请回答如下问题：

- 你的first fit算法是否有进一步的改进空间?

first-fit 连续物理内存分配算法是一个基础的内存分配方法，它尝试在可用内存块中找到第一个足够大的块以满足分配请求。下面是关于 `kern/mm/default_pmm.c` 中相关函数的分析以及程序的内存分配过程：

1. `default_init` 函数：这个函数初始化了物理内存管理器。它通过调用 `list_init` 来初始化空闲内存块的链表 `free_list`，并将 `nr_free` 设置为 0。 `free_list` 是用来记录未使用的内存页的链表， `nr_free` 记录了当前的未使用内存页数目。
2. `default_init_memmap` 函数：这个函数用于初始化一个物理内存块，它接受一个指向物理内存块的指针 `base` 和块的大小 `n` 作为参数。在初始化过程中，它首先确保 `n` 大于 0。然后，它遍历该块内的每个页，并执行以下操作：
    - 检查每个页是否已被保留（reserved）。
    - 将每个页的标志位 `flags` 和属性 `property` 设置为 0。
    - 将每个页的引用计数 `ref` 设置为 0。
    - 使用 `list_add` 将每个页链接到 `free_list` 链表中。
    - 更新 `nr_free`，增加空闲页数目。
    - 如果 `free_list` 为空，直接将当前页添加到链表头部；否则，遍历 `free_list`，根据地址的大小将当前页插入合适的位置。

3. `default_alloc_pages` 函数：这个函数用于分配连续的内存页。它接受一个参数 `n`，表示要分配的页数。首先，它确保 `n` 大于 0，并检查是否有足够的空闲页数。然后，它遍历 `free_list` 链表，查找第一个拥有足够空闲页数的内存块。如果找到了符合条件的块，它会执行以下操作：
    - 从链表中移除这个块。
    - 如果这个块的空闲页数大于 `n`，则将剩余的页数更新到新块，并重新插入到链表中。
    - 更新 `nr_free`，减少空闲页数目。
    - 将被分配的页的标志位 `flags` 设置为 PG_reserved，表示这些页已被分配。
    - 返回分配的页的指针。
4. `default_free_pages` 函数：这个函数用于释放连续的内存页。它接受一个指向物理内存块的指针 `base` 和块的大小 `n` 作为参数。在释放过程中，它首先确保 `n` 大于 0 并检查要释放的页是否都是可释放的。然后，它执行以下操作：
    - 将每个页的标志位 `flags` 设置为 0，表示这些页未被分配。
    - 将每个页的引用计数 `ref` 设置为 0。
    - 使用 `list_add` 将每个页链接到 `free_list` 链表中。
    - 更新 `nr_free`，增加空闲页数目。
    - 尝试合并相邻的空闲块，以减少内存碎片。
5. `default_nr_free_pages` 函数：这个函数返回当前可用的空闲页数。

进一步的改进空间：

- 算法优化：算法使用的是first fit策略，即找到第一个满足要求的空闲页面进行分配。然而，由于物理页面的分布不一定均匀，可能存在很多小的空闲页面，导致后续大的内存请求无法满足。因此，可以考虑引入更高效的内存分配策略，比如best fit或者next fit，以提高内存利用率。
- 内存回收：可以实现内存回收机制，及时释放不再使用的内存块，以提高内存利用率。内存合并算法可以进一步完善，目前的合并算法只考虑了当前释放的页面与其相邻的前后页面是否可合并，但没有考虑更远距离的页面合并情况。可以进行更加全面的内存合并分析，将更多的相邻页面进行合并，以减少碎片。
- 大页支持：支持分配和释放大页，以减少内存管理开销。
- 性能优化：针对特定硬件架构和应用场景进行性能优化，以提高内存分配的效率。
- 可以在系统初始化时预留一部分空闲页面，以备未来的分配需要，或者在内存不足时可以回收一部分不常用的页面作为空闲页面供其他进程使用，以提高整体系统的内存利用率。

练习2：实现 Best-Fit 连续物理内存分配算法（需要编程）

在完成练习一后，参考kern/mm/default_pmm.c对First Fit算法的实现，编程实现Best Fit页面分配算法，算法的时空复杂度不做要求，能通过测试即可。
请在实验报告中简要说明你的设计实现过程，阐述代码是如何对物理内存进行分配和释放，并回答如下问题：

- 你的 Best-Fit 算法是否有进一步的改进空间?

Best-Fit 算法是一种内存分配算法，它尝试在空闲内存块中找到大小最接近需求的块来分配内存。下面是关于 Best-Fit 算法的代码以及它对物理内存的分配和释放的过程：

1. `best_fit_init` 函数：这个函数初始化了 Best-Fit 内存管理器。它通过调用 `list_init` 来初始化空闲内存块的链表 `free_list`，并将 `nr_free` 设置为 0。`free_list` 用于记录未使用的内存页的链表，`nr_free` 记录了当前的未使用内存页数目。
2. `best_fit_init_memmap` 函数：这个函数用于初始化一个物理内存块，它接受一个指向物理内存块的指针 `base` 和块的大小 `n` 作为参数。在初始化过程中，它首先确保 `n` 大于 0。然后，它遍历该块内的每个页，并执行以下操作：
    - 检查每个页是否已被保留（reserved）。
    - 将每个页的标志位 `flags` 和属性 `property` 设置为 0。
    - 将每个页的引用计数 `ref` 设置为 0。
    - 使用 `list_add` 将每个页链接到 `free_list` 链表中。
    - 更新 `nr_free`，增加空闲页数目。
    - 如果 `free_list` 为空，直接将当前页添加到链表头部；否则，遍历 `free_list`，根据地址的大小将当前页插入合适的位置。
3. `best_fit_alloc_pages` 函数：这个函数用于分配连续的内存页。它接受一个参数 `n`，表示要分配的页数。首先，它确保 `n` 大于 0 并检查是否有足够的空闲页数。然后，它遍历 `free_list` 链表，查找大小最接近 `n` 的内存块。如果找到了符合条件的块，它会执行以下操作：
    - 从链表中移除这个块。
    - 如果这个块的空闲页数大于 `n`，则将剩余的页数更新到新块，并重新插入到链表中。
    - 更新 `nr_free`，减少空闲页数目。
    - 将被分配的页的标志位 `flags` 设置为 PG_reserved，表示这些页已被分配。
    - 返回分配的页的指针。
4. `best_fit_free_pages` 函数：这个函数用于释放连续的内存页。它接受一个指向物理内存块的指针 `base` 和块的大小 `n` 作为参数。在释放过程中，它首先确保 `n` 大于 0 并检查要释放的页是否都是可释放的。然后，它执行以下操作：
    - 清除每个页的标志位 `flags`，表示这些页未被分配。
    - 将每个页的引用计数 `ref` 设置为 0。
    - 使用 `list_add` 将每个页链接到 `free_list` 链表中。
    - 更新 `nr_free`，增加空闲页数目。
    - 尝试合并相邻的空闲块，以减少内存碎片。
5. `best_fit_nr_free_pages` 函数：这个函数返回当前可用的空闲页数。

与first-fit区别最大的地方就是best_fit_alloc_pages，需要寻找到空间相差最小的那个块。

```
  // 遍历空闲链表，查找满足需求的最小空闲页框

    list_entry_t *best_fit = NULL; // 记录最佳匹配
    size_t smallest_gap = (size_t) -1; // 记录最小的超出的空闲空间，初始化为最大的size_t值

    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        // 检查是否满足需求并且空闲空间小于之前找到的最小空闲空间
        if (p->property >= n && (p->property - n) < smallest_gap) {
            best_fit = le;
            smallest_gap = p->property - n;
```

```
        }
    }

    // 如果找到了合适的页框
    if (best_fit != NULL) {
        struct Page *page = le2page(best_fit, page_link);
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add(prev, &(p->page_link));
        }
        nr_free -= n;
        ClearPageProperty(page);
        return page;
    }

    return NULL;


}
```

Best-Fit 算法的改进空间：

- 性能优化：可以针对特定硬件架构和应用场景进行性能优化，以提高内存分配的效率。可以考虑使用数据结构来加速分配和释放操作。
- 内存回收：实现内存回收机制，及时释放不再使用的内存块，以提高内存利用率。
- 分区分配：实现不同大小的内存块分配策略，以更好地满足不同大小的内存需求。

扩展练习Challenge：buddy system（伙伴系统）分配算法（需要编程）

Buddy System算法把系统中的可用存储空间划分为存储块(Block)来进行管理, 每个存储块的大小必须是2的n次幂(Pow(2, n)), 即1, 2, 4, 8, 16, 32, 64, 128...

- 参考伙伴分配器的一个极简实现， 在ucore中实现buddy system分配算法，要求有比较充分的测试用例说明实现的正确性，需要有设计文档。

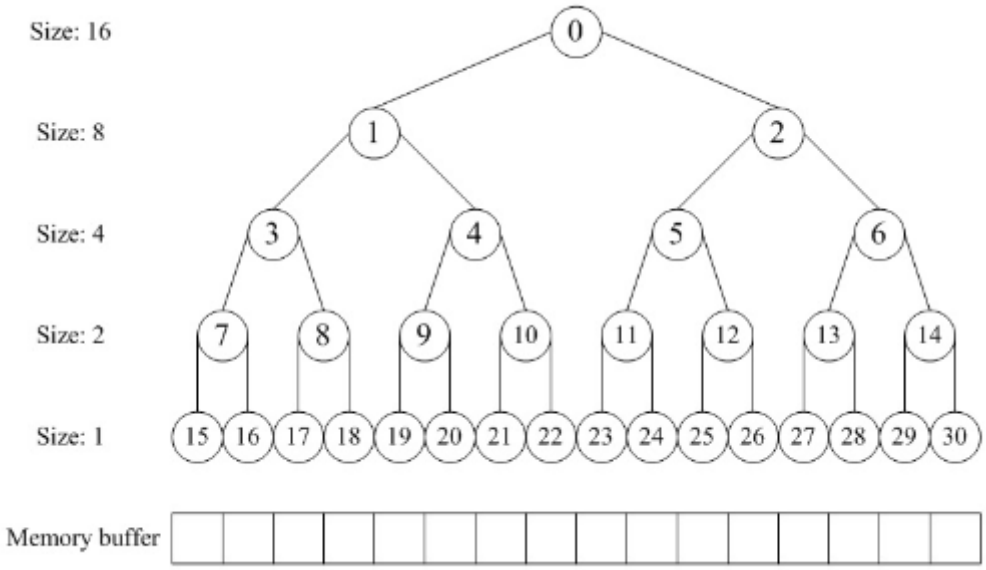
在first-fit和best-fit算法中，因为每次分配空间的时候，都有可能会产生很多的小碎片空间，而这些切割下来的碎片空间，大多数是不够其他程序使用的，所有为了避免产生大量碎片空间，我们尝试将每次切割下来的空间尽可能的大，即切割下来的空间至少和被使用的空间一样大，这样可以极大程度上，减少被切割下来的空间不能使用的可能性。

为了实现buddy system分配算法，我们使用了一种类似于二叉树的结构，如图。对于以2的幂单位大小的内存空间，我们使用一个数组形式的完全二叉树来管理内存。比如图中是16个单位的内存空间，我们声明一个大小为31的数组来保存这颗二叉树，其中根节点管理16个单位的内存，其子节点分别管理一半，也就是8个单位的内存，以此类推。当我们需要3个单位的空间时，首先要找到最小的2的幂，即4，所以我们要分配出4个单位的内存空间。我们从根节点开始遍历，以左子树优先的原则，找到第一个管理4个单位空间的结点，也就是3号结点，再通过结点序号与内

存偏移之间的换算（offset = node_size * (index + 1) - total_size），我们可以得到分配内存的起始地址也就是0，再结合3号结点所管理的大小——4个单位，得到我们分配的空间起始地址与大小。

之后要做的，就是做上标记，告诉系统，这一部分已经被分配。我们要将3号结点标记为已经被分配，即这一部分的可使用内存变为0，向上回溯，1号结点实际管理的可分配内存变为4，0号结点可以分配的内存变为8。这是是取左右子树的最大值，而不是之和，是因为之和的含义是一整块，而这里不一定是一整块，不一定真的有之和那么多的内存，而取最大值，也能方便下一次分配时寻找合适的空间。

分配之和就是释放内存。释放内存我们通过内存偏移，计算得到管理该内存的结点叶子结点（index = offset + self->size - 1），通过叶子结点一步一步向上回溯找到被设置为0的结点，将其设置回实际管理的内存。从根节点向下检测，如果左右子树的大小之和符合父节点实际管理的内存，那么就进行合并，父节点的空闲内存设置为左右子树内存之和，否则设置为最大的内存。



扩展练习Challenge：任意大小的内存单元slub分配算法（需要编程）

slub算法，实现两层架构的高效内存单元分配，第一层是基于页大小的内存分配，第二层是在第一层基础上实现基于任意大小的内存分配。可简化实现，能够体现其主体思想即可。

- 参考linux的slub分配算法/，在ucore中实现slub分配算法。要求有比较充分的测试用例说明实现的正确性，需要有设计文档。

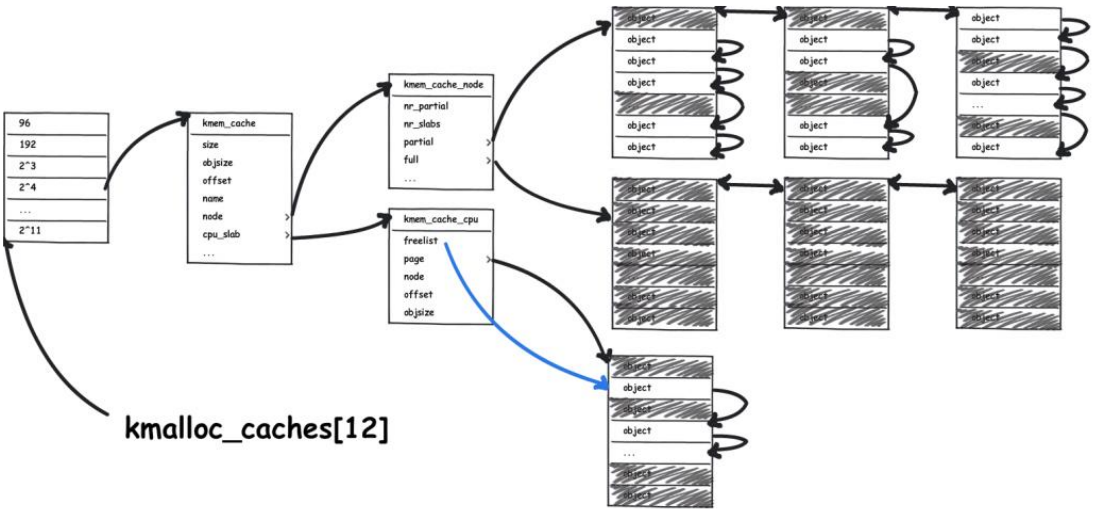
上述代码实现了slub分配算法，使用了两层架构：第一层是基于页大小的内存分配，第二层是基于任意大小的内存分配。在第二层中，我们使用了slab和cache的概念来管理不同大小的内存块，以提高内存分配的效率。

为了实现任意的内存分配，我们设计了一种数据结构，将内存分为12组，每个组分别包含 8、64、512、...2048个字节，每一组包含一个链表，一个结点管理对应的一小片内存，同时每个结点有一个指向下一个结点的指针，所以其中每个结点实际可分配内存要减去四，因为有四个字节是指向下一个结点的指针，我们将这样的一个结点成为object。为了优化，我们设计了一个缓存结构，这个缓存结构指向三个链表，分别是未分配内存链表，可分配链表，已完全分配链表，我们将可分配链表称为slub。

首先是设计内存分配，当我们要分配一个4字节大小的内存时，我们通过数组找到4字节大小内存对应的缓存，通过这个内存缓存来经行下一步的内存分配。首先遍历可分配链表，查看是否还有未分配的object，如果有则按顺序分配，如果没有那么就从未分配链表中取，将可分配链表设置为已完全分配链表，将未分配链表设置可分配链表。

接下来是内存释放，考虑三种情况：第一，object在释放之前slub是full状态的时候（slub中的object都是被占用的），释放该object后，这是该slub就是半满（partail）的状态了，这时需要把该slab添加到kmem_cache_node中的partial链表中。第二，slab是partial状态时（slab中既有object被占用，又有空闲的），直接把该object加入到该slab的空闲队列中即可。第三，该object在释放后，slab中的object全部是空闲的，还需要把该slab释放掉。

> 在分配缓存块的时候，要分两种路径，fast path和slow path，也就是快速通道和普通通道。其中 kmem_cache_cpu 就是快速通道，kmem_cache_node 是普通通道。每次分配的时候，要先从 kmem_cache_cpu 进行分配。如果 kmem_cache_cpu 里面没有空闲的块，那就到 kmem_cache_node 中进行分配；如果还是没有空闲的块，才去伙伴系统分配新的页。



扩展练习Challenge：硬件的可用物理内存范围的获取方法（思考题）

- 如果 OS 无法提前知道当前硬件的可用物理内存范围，请问你有何办法让 OS 获取可用物理内存范围？

1. **BIOS/UEFI 接口**：操作系统可以通过 BIOS/UEFI 提供的接口来获取硬件信息，包括可用物理内存范围。在启动过程中，BIOS/UEFI 初始化硬件并记录可用内存范围，操作系统可以通过这些接口查询。

   在Linux中有很多方法获取内存容量，如果一种方法失败，就会调用其他方法，但是这些方法的共性是调用BIOS中断的0x15实现的，分别是 0x15 的三个子功能，子功能号要放在寄存器EAX或者AX中。

   BIOS中断可以返回已安装的硬件信息，由于BIOS及其中断也只是一组软件，它要访问硬件也要依靠硬件提供的接口，所以，获取内存信息，其内部是通过连续调用硬件的应用程序接口（Application Program Interface，API）来获取内存信息的。另外，由于每次调用BIOS中断都是有一定的代价的（比如至少要将程序的上下文保护起来以便从中断返回时可以回到原点继续向下执行），所以尽量在一次中断中返回足量的信息，由用户程序自己挑出重点内容。

   中断 0x15 子功能 0xe820

这个子功能能够获取系统的内存布局，由于系统内存各部分的类型属性不同，BIOS就按照类型属性来划分这片系统内存，所以这种查询呈迭代式，每次BIOS只返回一种类型的内存信息，直到将所有内存类型返回完毕。子功能0xE820的强大之处是返回的内存信息较丰富，包括多个属性字段，所以需要一种格式结构来组织这些数据。

内存信息的内容是用地址范围描述符来描述的，用于存储这种描述符的结构称之为地址范围描述符(ARDS)

地址范围描述符结构如下表

字节偏移量	属性名称	描述
0	BaseAddrLow	基地址的低32位
4	BaseAddrHigh	基地址的高32位
8	LengthLow	内存长度的低32位，单位字节
12	LengthHigh	内存长度的高32位，单位字节
16	Type	本段内存的类型

此结构中的字段大小都是4字节，共5个字段，所以一共是20字节，每次执行这个中断，BIOS就返回这样一个数据结构，上面是4个字段很好理解，Type字段为1表示这段内存可以被操作系统使用，为2则表示内存使用中或者被系统保留，操作系统不可以用此内存，Type字段一共就两个定义。

为什么BIOS会按照类型来返回内存信息呢，原因是这段内存可能是系统的ROM，或者ROM用到了这部分内存，设备内存映射到了这部分内存，由于某种原因，这段内存不适合标准设备使用。

由于我们在32位环境下工作，所以在ARDS结构属性中，我们只用到了低32位属性，BaseAddrLow+LengthLow是一片内存区域上限，单位是字节。正常情况下，不会出现较大的内存区域不可用的情况，除非安装的物理内存极其小。

BIOS中断只是一段函数例程，调用它就要为其提供参数，现在介绍下BIOS中断0x15的0xe820子功能需要哪些参数

1.EAX: 子功能号，这里输入为 0xE820

2.EBX: ARDS后续值，内存信息需要多次返回，EBX告诉中断下一次调用时返回哪个ARDS，第一次置0，后续不需要关注

3.ES:DI : ARDS缓冲区：BIOS将获取到的内存信息写入此寄存器指向的内存，每次都以ARDS格式返回

4.ECX: ARDS结构的字节大小：用来指示BIOS写入的字节数，调用者和BIOS都同时支持的大小是20字节

5.EDX: 固定为签名标记0x534d4150，此十六进制数字是字符串SMAP的ASCII码：BIOS将调用者正在请求的内存信息写入ES：DI寄存器所指向的ARDS缓冲区后，再用此签名校验其中的信息

返回后

1.CF: 若CF位为0表示调用未出错，CF为1，表示调用出错

2.EAX: 字符串SMAP的ASCII码0x534d4150

3.ES:DI : ARDS缓冲区地址，同输入值是一样的，返回时此结构中已经被BIOS填充了内存信息

4.ECX：BIOS写入到ES:DI所指向的ARDS结构中的字节数，BIOS最小写入20字节

5.EBX :后续值：下一个ARDS的位置。每次中断返回后，BIOS会更新此值，不需要操作

所以该中断的调用方式为

1、填写好“调用前输入”中列出的寄存器。

2、执行中断调用int 0x15。

3、在CF位为0的情况下，“返回后输出”中对应的寄存器便会有对应的结果。

中断 0x15 子功能 0xe801

这个方法虽然获取内存的方式较为简单，但是功能也不强大，只能识别最大4GB内存，不过这对于32位的地址线也够用了。

此方法检测到的内存是分别存储到两个寄存器中的，低于15MB的内存以1KB为单位来记录，记录结果保存在AX和CX中，其中AX和CX中的数据是一样的，所以在15MB空间以下的实际内存容量为 A X ∗ 1024 AX * 1024 AX∗1024 。AX的最大值为 0x3c00，所以知道为什么最大只能15MB了吧，16MB~4GB的内存时以64KB为单位来记录的，单位数量在寄存器BX和DX中记录

所以这个中断子功能需要的参数

1.AX ： 子功能号 0xE801

返回后

1.CF：CF为0表示调用未出错，CF为1，表示调用出错

2.AX：1KB的容量

3.BX：64KB的容量

4.CX：1KB的容量

5.DX：64KB的容量

功能我们介绍完了其实怎么用就很好说了，但是有没有一些疑问，比如说，为什么小于15MB的内存和大于16MB的内存要分开记录，中间的1MB去哪儿了，比如为什么AX和CX一样，BX和DX一样?

我们先解决第一个问题，这是一个著名的历史遗留问题，80286拥有24位地址线，其寻址空间是16MB。当时有一些ISA设备要用到地址15MB以上的内存作为缓冲区，也就是此缓冲区为1MB大小，所以硬件系统就把这部分内存保留下来，操作系统不可以用此段内存空间。保留的这部分内存区域就像不可以访问的黑洞，这就成了内存空洞memory hole。现在虽然很少很少能碰到这些老ISA设备了，但为了兼容，这部分空间还是保留下来，只不过是通过BIOS选项的方式由用户自己选择是否开启。BIOS厂商不同，一般的菜单选项名称也不相同，不过大概意思都差不多。

起初这个 0x801的中断子功能也是为了支持ISA服务的，如果内存的容量大于16MB，那么 A X ∗ 1024 AX*1024 AX∗1024*必然是小于等于15MB，而 B X ∗ 64 ∗ 1024 BX*64*1024 BX∗64∗1024肯定大于0，所以很容易就检测出内存空洞。当然如果物理内存在16MB以下，此方法就不灵了，但检测到的内存依然会小于实际内存1MB。

至于第二个问题，我们可以理解为，BIOS就是这么设计的

总结，此中断的调用步骤为：

1、将AX寄存器写入0xE801。

2、执行中断调用int 0x15。

3、在CF位为0的情况下，“返回后输出”中对应的寄存器便会有对应的结果。

中断 0x15 子功能 0x88

这个方法最简单，得到的结果也最简单，简单到只能识别最大64MB的内存。即使内存容量大于64MB，也只会显示63MB，大家可以自己在bochs中试验下。为什么只显示到63MB呢？因为此中断只会显示1MB之上的内存，不包括这1MB，咱们在使用的时候记得加上1MB。

所以这个中断子功能需要的参数

1.AX： 子功能号 0x88

返回后

1.CF：CF为0表示调用未出错，CF为1，表示调用出错

2.AX：1KB的容量，内存空间1MB之上的连续单位数量，不包括低端1MB内存。故内存大小为AX*1024字节+1MB

此中断的调用步骤为：

1、将AX寄存器写入0x88。

2、执行中断调用int 0x15。

3、在CF位为0的情况下，“返回后输出”中对应的寄存器便会有对应的结果。

2. **物理内存映射表**：一些体系结构或硬件平台提供了物理内存映射表，其中包含了可用内存块的信息。操作系统可以解析这些表以获取可用物理内存范围。
3. **系统调用**：操作系统可以定义一些特定的系统调用，供用户空间程序查询可用物理内存范围。用户空间程序可以通过这些系统调用与操作系统通信以获取物理内存信息。
4. **Bootloader 传递参数**：一些引导加载程序（Bootloader）可以检测硬件配置并传递相关参数给操作系统。操作系统可以从引导加载程序接收这些参数，包括可用内存范围。
5. **内存探测算法**：操作系统可以实现一些内存探测算法，通过尝试访问不同的物理地址来检测内存的可用性。这种方法需要小心处理，以避免访问无效地址导致系统崩溃。