LAB2实验说明：

这篇报告将指导手册上的知识平摊到了每一个实验的具体流程中，可能会便于大家定位知识的用途到底在哪里。

实验一、default_pmm.c

在这个实验中主要是要完成一个first-fit算法的实现，这个算法的大体含义就是用一个双向链表维护很多空闲块，并且返回第一个找到的可以满足要求的空闲块，所谓的满足要求就是这个空闲块的物理内存的大小比需要的内存大小要大就可以。

需要修改的文件：mm/default_pmm.c

（1）文件首先定义了一个free_area_t，它维护了一个双向链表

```
/* free_area_t - maintains a doubly linked list to record free (unused) pages */
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // # of free pages in this free list
} free_area_t;

//上面的list_entry_t其实是一个list_entry
typedef struct list_entry list_entry_t;//定义在list.h 相当于给list_entry起了个新名字

//别看这里好像说明都没有，只有一个前后指针，实际上这个list_entry是一个Page的元素，如果我们拿到了一个list_entry对象，那么就可以通过特定的函数找到其嵌入的page对象
struct list_entry {
    struct list_entry *prev, *next; //list_entry的定义，包括一个前面的指针一个后面的指针
};
```

（2）default_init函数

```
static void
default_init(void) {
    list_init(&free_list);  //初始化链表，让他的前后指针都指向自己
    nr_free = 0;            //还没有页数
}

//在list.h中定义
/* *
 * list_init - initialize a new entry
 * @elm:        new entry to be initialized
 * */
static inline void
list_init(list_entry_t *elm) {
    elm->prev = elm->next = elm;
}
```

（3）default_init_memap函数：这个函数的作用就是初始化一段内存空间并把它加入到双向链表中（其实这么说可能不太准确，因为双向链表的类实际上实在page中）初始化一个块，这个块由很多页组成，注意这里的初始化和后面的alloc不是一个概念，这个初始化是将一些页变成进程可以使用的页

```
static void
default_init_memmap(struct Page *base, size_t n) {//插入一个块，这个块可能由很多页组成
    assert(n > 0); //插入的页数必须大于零
    struct Page *p = base;      //base就是这些块开始的那个页
    for (; p != base + n; p ++) {
        assert(PageReserved(p));  //必须保证这些页都是被系统保留的页，下面会有专门的专题来说为什么这里必须是被系统保留的页
        p->flags = p->property = 0; //把property都置0，这里的property代表着这个块有多少个页（只对块的首页生效）
        set_page_ref(p, 0);     //还没有其他的页表引用这块物理内存
    }
    base->property = n;       //更改初始页的property
    SetPageProperty(base);      //更改初始页的属性值
    nr_free += n;               //根据页数进行累加，这是代表着那个全局的空页数有多少，要和前面那个base->property区分一下
    list_add(&free_list, &(base->page_link));    //把新的块加入到已有的双向链表中
}

//首先来看Page的定义 定义在memlayout.h中
/* *
 * struct Page - Page descriptor structures. Each Page describes one
 * physical page. In kern/mm/pmm.h, you can find lots of useful functions
 * that convert Page to other data types, such as phyical address.
 * */
//一个page就描述了一个物理内存的页

//ref表示了有多少虚拟页引用了这个物理内存页

//flags记录这个物理页的状态，查看flags的定义：
#define PG_reserved                 0       // if this bit=1: the Page is reserved for kernel, cannot be used in alloc/free_pages; otherwise, this bit=0
//如果reserved是1就说明这个也是被内核所占有的，其他人不能分配
#define PG_property                 1       // if this bit=1: the Page is the head page of a free memory block(contains some continuous_addrress pages), and can be used in alloc_pages; if this bit=0: if the Page is the the head page of a free memory block, then this Page and the memory block is alloced. Or this Page isn't the head page.
//如果这一位是1，说明这个页是某一个连续块的块首，并且它还没被分配，可以用于分配。如果他是零就有两种可能：第一，他是一个块首页但是它已经被分配了或者它根本就不是一个块首页。

//然后就是一些改变或者获得状态位的函数：
//set就是置位 clear就是清楚 test就是检验，具体的函数实现在atomic.h可以查看
#define SetPageReserved(page)       set_bit(PG_reserved, &((page)->flags))
#define ClearPageReserved(page)     clear_bit(PG_reserved, &((page)->flags))
#define PageReserved(page)          test_bit(PG_reserved, &((page)->flags))
#define SetPageProperty(page)       set_bit(PG_property, &((page)->flags))
#define ClearPageProperty(page)     clear_bit(PG_property, &((page)->flags))
#define PageProperty(page)          test_bit(PG_property, &((page)->flags))

//property 这里我觉得注释给的不太明确，这个property就代表着以该页为页首的块有几页，如果不是块首页根本就直接是0
```

```
//最后这个list_entry_t就是代表着那个维护空闲块的双向链表
struct Page {
    int ref;                        // page frame's reference counter
    uint32_t flags;                 // array of flags that describe the status
of the page frame
    unsigned int property;          // the num of free block, used in first fit
pm manager
    list_entry_t page_link;         // free list link
};
```

(4) defualt_alloc_pages函数：分配一个大小为n的页

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);//我要确保我分配的页数必须是个正数
    if (n > nr_free) {//如果我这个双向链表中所有的页数加起来都不如n大的话那肯定分配失败，返
回一个NULL就好了
        return NULL;
    }
    struct Page *page = NULL;//用来保存我最后返回的那个内存块的第一页
    //开始遍历空闲链表
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {//只要最后没有遍历回来就一直遍历
        //找到当前链表指向的页表，如果这个内存页数大于我们需要的页数，则直接从这个内存块取n页
        struct Page *p = le2page(le, page_link);//这句话的意思就是，我遍历只能拿到一个
list_entry这个变量的东西，但是我想要的是一个Page，所以我需要调用这个函数帮我实现转换
        if (p->property >= n) {
            page = p;
            //SetPageReserved(page);
            break;
        }
    }
    //如果找到了，才做下面的操作
    if (page != NULL) {
        //有一种可能就是会剩下几个页，那么就需要重新组织剩余的空页
        if (page->property > n) {
            //因为我们取了n页，内存块可能还有部分内存页，需要当前内存块头偏移n个`Page`位置就
是
            //内存块剩下的页组成新的内存块结构，新的页头描述这个小内存块
            //p指向了那块小内存
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);//记得做这步，把property设为1，表示我是这个块的首页并且可
以被分配
            //往空闲链表里加入这个新的小内存
            list_add(&free_list, &(p->page_link));
    }
        list_del(&(page->page_link));//删除掉原来的块
        nr_free -= n;//整个链表保存的页也减少了
        ClearPageProperty(page);//原来的首页不能再被分配了
    }
    return page;
```

```
}

//这个函数有什么用呢，就是通过一个类的一个元素来定位这个类在哪，他的实现方式是我知道这个元素相对于这个类的偏移，然后我顺着去减掉这个偏移就行了
//le2page其实就是把list_entry变成page的函数，它也是调用了to_struct函数来完成的
// convert list entry to page
#define le2page(le, member)                 \
    to_struct((le), struct Page, member)

/* 在defs.h中可以找到
 * to_struct - get the struct from a ptr
 * @ptr:    a struct pointer of member 指向的就是嵌入其他对象的那个对象，也就是member
 * @type:   the type of the struct this is embedded in 被嵌入的那个对象的类是啥
 * @member: the name of the member within the struct，就是他在被嵌入的对象里的名字叫啥
 * */
#define to_struct(ptr, type, member)                               \
    ((type *)((char *)(ptr) - offsetof(type, member)))

/* Return the offset of 'member' relative to the beginning of a struct type */
#define offsetof(type, member)                                     \
    ((size_t)(&((type *)0)->member))
```

（5）default_free_pages函数：注意这个free的含义，这个free的意思是，我把这块物理内存free掉，那么这代表着什么呢，代表着没有人再用这块物理内存了，这个物理内存的信息已经没用了，下一次再访问同样的空间我就不想再知道这块物理内存以前存了什么了，而是想存入新的东西，所以说，已经没有人再去ref它了，记得把他的ref清零。

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);//首先要保证我们要释放的页数必须大于0
    struct Page *p = base;
    //首先遍历页表，把flags全部置0，并将ref清0，说明此时没有逻辑地址引用这块内存
    for (; p != base + n; p ++) {
        //如果pagereserved是1说明是给内核预留的页，我们不能free掉，同理没有被分配的页或者不是块首的页也不能被free
        assert(!PageReserved(p) && !PageProperty(p));
        //先把所有的标志位都置零
        p->flags = 0;
        //把物理内存释放了，所以没有虚拟地址会再指向它
        set_page_ref(p, 0);
    }
    //同样的道理，我释放了n页，那么个n页形成新的一个大一点的内存块，我们需要设置这个内存块的第一个
    //设置它后面跟了n个页并且目前可以被分配
    base->property = n;
    //设置块首页flag为可分配的物理内存
    SetPageProperty(base);
    //遍历空闲链表，目的找到有没有地址空间是连在一起的内存块，把他们合并
    list_entry_t *le = list_next(&free_list);
    while (le != &free_list) {
        //同理把le转换成page
        p = le2page(le, page_link);
        le = list_next(le);
        //意思就是如果我这个base块的结尾正好和块p的开头连在一起了，那就说明可以合并
```

```
        //这个就是基于base向后合并
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        //这个就是基于base向前合并
        else if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            //注意这里也要删掉p因为后面我们会对base完成插入，如果不删除相当于插了两次
            list_del(&(p->page_link));
        }
    }
    nr_free += n;
    //遍历空闲链表，因为空闲链表是from low to high（见实验指导书153页）
    //只需要遍历找打第一个地址比他高的，把释放的内存插入到他前面就行
    le = list_next(&free_list);
    while (le != &free_list) {
        p = le2page(le, page_link);
        if (base + base->property <= p) {//需要注意到的是page是一个指针，所以可以以其存在的虚拟地址表征其代表的物理地址
            //必须保证不能合并的
            assert(base + base->property != p);
            break;
        }
        le = list_next(le);
    }
    //把第二个参数插入到第一个参数的前面
    list_add_before(le, &(base->page_link));
}

/* *
 * list_add_before - add a new entry
 * @listelm:    list head to add before
 * @elm:        new entry to be added
 *
 * Insert the new element @elm *before* the element @listelm which
 * is already in the list.
 * */
static inline void
list_add_before(list_entry_t *listelm, list_entry_t *elm) {
    __list_add(elm, listelm->prev, listelm);
}
```

实验二、pmm.c getpte函数

这个实验要干的一件事就是我给你一个虚拟地址，你把这个虚拟地址对应的二级页表项拿出来，如果这个虚拟地址还不存在一个二级页表那就分配一个。

```
pde_t *pdep = &pgdir[PDX(la)];//首先我拿到一个页目录项，这个页目录项的来源就是我用虚拟地址做索引在页目录表中找到的一个地址
    if (!(*pdep & PTE_P)) {//如果说这个地址的内容和PTE_P相与不为1，那么我就要为其分配一个页
```

```
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL) {//为page分配空间，这个page是一个新的二级页表
            return NULL;
        }
        set_page_ref(page, 1);//这个二级页表目前被页目录表（一级页表）索引
        uintptr_t pa = page2pa(page);//拿到代表这个页所管理的物理地址
        memset(KADDR(pa), 0, PGSIZE);//把对应物理地址内容都置零
        *pdep = pa | PTE_U | PTE_W | PTE_P;//变为存在 用户可用 可写，注意看家人们，对应的页目录项存储的是二级页表的物理地址！！！！
    }
    return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];//返回对应的页目录项


//标志位定义
/* page table/directory entry flags */
#define PTE_P           0x001                   // Present
#define PTE_W           0x002                   // Writeable
#define PTE_U           0x004                   // User
#define PTE_PWT         0x008                   // Write-Through
#define PTE_PCD         0x010                   // Cache-Disable
#define PTE_A           0x020                   // Accessed
#define PTE_D           0x040                   // Dirty
#define PTE_PS          0x080                   // Page Size
#define PTE_MBZ         0x180                   // Bits must be zero
#define PTE_AVAIL       0xE00                   // Available for software use
                                                // The PTE_AVAIL bits aren't used
by the kernel or interpreted by the
                                                // hardware, so user processes
are allowed to set them arbitrarily（任意）.
```

(1) pde_t*是什么

```
//按照命名规则的理解，pde_t*是一个二级页表的指针实际上我们通过逐层推进找到它的定义：
typedef uintptr_t pde_t;
typedef uint32_t uintptr_t;
typedef unsigned int uint32_t;
//果不其虽然，这个pde_t实际上就代表这一个32bits的地址
```

（2）pgdir是什么

```
//the kernel virtual base address of PDT
//上面是代码注释中对参数的解释
//在实验指导书164页对这个pgdir的解释如下：
//注意：pgdir实际不是表项，而是一级页表本身。实际上应该新定义一个类型pgd_t来表示一级页表本身
//如果还不理解没关系，先看下一个东西
//如果你已经理解了PDX，那么你就对pgdir[PDX(la)]这样的写法不陌生了
//也就是说，pgdir就是一级页表，我用一级页表的索引去里面找二级页表。
```

（3）PDX是什么

```
// A linear address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
```

```
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |     Index      |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
//  \---------- PPN(la) -----------/
//
// The PDX, PTX, PGOFF, and PPN macros decompose linear addresses as shown.
// To construct a linear address la from PDX(la), PTX(la), and PGOFF(la),
// use PGADDR(PDX(la), PTX(la), PGOFF(la)).
//这些都是mmu.h中的注释，什么意思呢，就是ucore使用的二级页表，通过一个虚拟地址的前十位来索引一级页表，用中间十位索引二级页表，最后的十二位作为偏移寻找一个页里面的具体字节。
// page directory index
//那么PDX是用来干什么的呢，就是用来提取出一级页表索引的
#define PDX(la) ((((uintptr_t)(la)) >> PDXSHIFT) & 0x3FF)
#define PDXSHIFT        22                      // offset of PDX in a linear address
//看明白了吗，也就是说把一个虚拟地址右移22位是不是就只剩高10位了？再把它与001111111111进行与，那么就剩什么了？其实就剩原来的高10位本身了。
```

(4) PTE_P是什么

```
//到此为止，我们已经拿到了二级页表，但我们需要的是二级页表项，所以我们还需要做一次映射，但在做映射之前需要检查这个页表是否存在，怎么检查呢？就是用PTE_P
/* page table/directory entry flags */
#define PTE_P           0x001                   // Present
//上面的注释可以在mmu.h中找到，所以一个简单的与操作就可以判定该页表是否合法
```

(5) page2pa是什么

```
//先说结论，这个函数的功能是获得Page管理的物理页的物理地址
//page2pa的具体函数在pmm.h中实现，可以看到，该函数将一个地址左移了12位
//page就是我们刚刚传入的新被分配的页的地址
static inline ppn_t
page2ppn(struct Page *page) {
    return page - pages;
}

static inline uintptr_t
page2pa(struct Page *page) {
    return page2ppn(page) << PGSHIFT;
}
#define PGSHIFT         12                      // log2(PGSIZE)

//于是，还是要补充那个alloc_page是什么，毕竟新获得的这个Page就是alloc出来的。不难看出，默认在这里只分配一页
#define alloc_page() alloc_pages(1)
//下面是alloc_pages的具体实现，还是通过调用pmm_manager来实现的内存分配
struct Page *
alloc_pages(size_t n) {
    struct Page *page=NULL;
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        page = pmm_manager->alloc_pages(n);
```

```
    }
    local_intr_restore(intr_flag);
    return page;
}

//那么，减数pages是什么呢？
// virtual address of physicall page array
struct Page *pages;

//减数pages是一个虚拟地址，我们查看他的赋值过程
pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);
//参照实验指导书第150页，和这里一模一样，由于bootloader加载ucore的结束地址（用全局指针变量end记录）以上的空间没有被使用，所以我们可以把end按页大小为边界去整后，作为管理页级物理内存空间所需的Page结构的内存空间
//所以说你看懂它是干什么的了吗
//pages指向了内存中第一个Page的虚拟地址，然后我又拿到了一个Page，我用page-pages就得到了page在pages中的偏移，这个偏移就是物理页的页号，然后我把它左移十二位就能得到物理地址
```

(6) KADDR是什么?

```
//kaddr其实就是把物理空间转换为内核虚拟空间的函数，问了助教，内核虚拟空间的意思就是，你已经知道了一个物理地址，但是你在写程序的时候没办法直接操作物理地址，必须要把他转为虚拟地址，这个函数就是起到的这个作用。，但是在内核通过物理地址转虚拟地址和在外部通过虚拟地址转物理地址的方式是不一样的，在内核中可以通过简单的算术运算来获得物理地址，但是在进程中需要顺着table一路找过来。
/* *
 * KADDR - takes a physical address and returns the corresponding kernel virtual
 * address. It panics if you pass an invalid physical address.
 * */
#define KADDR(pa) ({                                                    \
            uintptr_t __m_pa = (pa);                                    \
            size_t __m_ppn = PPN(__m_pa);                               \
            if (__m_ppn >= npage) {                                     \
                panic("KADDR called with invalid pa %08lx", __m_pa);    \
            }                                                           \
            (void *) (__m_pa + KERNBASE);                               \
        })
#define KERNBASE            0xC0000000

//如果看过实验指导书到这里应该就不陌生了，这是说明东西呢？就是ucore中虚拟地址和物理地址的映射关系，具体可以查看实验指导书第162页
//看一下这玩意具体都是怎么实现的
// page number field of address
//PPN函数：把一个虚拟地址右移12位
#define PPN(la) (((uintptr_t)(la)) >> PTXSHIFT)
#define PTXSHIFT        12                      // offset of PTX in a linear address
//npage的定义：代表了总共会有多少个页
npage = maxpa / PGSIZE;
//那么很明显了，如果你右移之后的这个数比最多的页数还要多，那么只有一种可能就是你给的地址就不对
```

(6) PTE_U PTE_W PTE_P是什么

```
*pdep = pa | PTE_U | PTE_W | PTE_P;
//在mmu.h中他们的作用分别是：存在 用户可写 用户可读
#define PTE_P           0x001                   // Present
#define PTE_W           0x002                   // Writeable
#define PTE_U           0x004                   // User
//只有当一级二级页表的项都设置了用户写权限后，用户才能对对应的物理地址进行读写。 所以我们可以在一级页表先给用户写权限，再在二级页表上面根据需要限制用户的权限，对物理页进行保护
```

(7)简而言之：最后返回了个啥：

```
//从里向外看，第一步先获取了页表或者的入口地址：把最后面的12位全部置零
// address in page table or page directory entry
#define PTE_ADDR(pte)   ((uintptr_t)(pte) & ~0xFFF)
#define PDE_ADDR(pde)   PTE_ADDR(pde)
//然后获得这个东西的物理地址
//最后再使用一个指针指向这个地址，所以说最后拿到的是虚拟地址。
//利用这个页表的虚拟地址，就可以使用传入的虚拟地址的中间10位进行索引并取出对应的页表项了
// page table index
#define PTX(la) ((((uintptr_t)(la)) >> PTXSHIFT) & 0x3FF)
#define PTXSHIFT        12                      // offset of PTX in a linear address

//上面的这玩意看看就行了，是抄的网上答案，来说一下我自己的理解把

return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];
//最后返回的东西是上面这一坨，这是什么呢
//pdep是一个页表项的指针，而对他取内容就相当于取它所指向的位置的内容，这个内容就是二级页表，也就是一个页表的基地址
//然后PDE_ADDR把他的后12位清零，为什么要清零，一个很简单的道理就是：页目录项和页表项存的地址都是末尾12位为0的地址：
//也表项就不用多说了，因为物理地址就需要以4k对齐，4k就是4096字节，这是什么概念呢，一个页表中有1024个页表项，每一个页表项保存了一个32位的地址，是4字节，所以一个页表也需要4k，整好是用一页来存储
//而我们知道也目录项和页表项中存的都是物理地址，要想用它做指针必须拿出虚拟地址来，于是获得了页表的基地址的虚拟地址，并把他强制类型转换为pte_t*也就是页表项，其实就是一个32位数，没什么特别的
//然后通过索引，右移12位再与03ff来做索引所以需要的页表项，然后你拿到的是一个地址，这个地址是物理页，也就是页目录项的内容，所以需要再取地址，获得一个pte_t的指针
```

实验三：释放某虚地址所在的页并取消对应二级页表项的映射

```
if (*ptep & PTE_P) {
        struct Page *page = pte2page(*ptep);//调用了使用pt找到page结构
        if (page_ref_dec(page) == 0) {
            free_page(page);
        }
        *ptep = 0;
        tlb_invalidate(pgdir, la);
    }
```

（1）pte2page是什么

```
//定义在pmm.h中，最后返回了pte对应的物理地址的Page结构
static inline struct Page *
```

```
pte2page(pte_t pte) {//首先输入进来的是一个二级页表项本身，也就是一个物理地址，或者理解成一个指针，指向了一个物理地址
    if (!(pte & PTE_P)) {
        panic("pte2page called with invalid pte");
    }
    return pa2page(PTE_ADDR(pte));//PTE_ADDR的作用是把后12位清零，这样就找到了一个完整的页框的开头位置，是一个物理地址
}

static inline struct Page *
pa2page(uintptr_t pa) {
    if (PPN(pa) >= npage) {
        panic("pa2page called with invalid pa");
    }
    return &pages[PPN(pa)];//PPN的作用是右移12位，用它作为page指针的索引即可
}
```

(2) page_ref_dec是什么?

```
//这个函数不难理解，就是把引用次数减一。
static inline int
page_ref_dec(struct Page *page) {
    page->ref -= 1;
    return page->ref;
}
```

(3) 如何理解下面这段代码:

```
if (page_ref_dec(page) == 0) {
            free_page(page);
   }
*ptep = 0;
//如果这个页表已经没有人引用了，就把他释放掉
//如果还有人在引用，那没关系，至少我需要清理二级页表，因为我已经绝对不再引用它了，怎么清除呢？就是把该页表的地址置0，二级页表不能再去寻找它了，但它并不影响其他页目录表对这个页表的引用。
```

(4) tlb_invalidate是什么

```
tlb_invalidate(pde_t *pgdir, uintptr_t la) {
    if (rcr3() == PADDR(pgdir)) {
        //还记得CR3是说明吗？可以查看实验指导书第158页，他就指向了页目录表的基地址，而pgdir是什么呢？可以看一下实验二中的说明，他其实也就是页目录表，所以这一步就是为了判断被修改的页目录表究竟是不是现在的进程在被使用的那个页目录表
        invlpg((void *)la);
    }
}

//由于我们释放了一些进程正在使用的页表，所以说我们需要刷新tlb，保证tlb不会残留被释放的地址
static inline uintptr_t
rcr3(void) {
    uintptr_t cr3;
    asm volatile ("mov %%cr3, %0" : "=r" (cr3) :: "memory");
    return cr3;
}
```

```
static inline void
invlpg(void *addr) {
    asm volatile ("invlpg (%0)" :: "r" (addr) : "memory");
}
//这里函数的目的就是取消va对应物理页之间的关联，相当于刷新TLB，每次我们调整虚拟页和物理页之间的映射关系的时候，我们都要刷新TLB，调用这个函数或invlpg汇编指令 .因为tlb直接保存了虚拟页和物理页之间的关系，所以当我们释放虚拟页的时候必须调整tlb
```

在上面的过程中我们有一个问题没有解决，就是关于init_memmap中的问题，为什么那个页要是保留的，这个问题需要从内存探测讲起。

一个基本的概念就是一开始操作系统是不知道计算机的物理内存是怎么分布的，于是需要BIOS中断来完成这个工作（在实模式下完成）于是BIOS通过系统内存映射地址描述符格式来表示系统物理内存布局

```
Offset    Size     Description
00h       8字节     base address                #系统内存块基地址
08h       8字节     length in bytes             #系统内存大小
10h       4字节     type of address range       #内存类型
```

用这样的格式来描述可用的计算机内存，其中type的取值是如下之一，表示不同内存块的不同性质

```
Values for System Memory Map address type:
#内存，是可以留给操作系统的
01h  memory, available to OS
#保留的，不能留给操作系统
02h  reserved, not available (e.g. system ROM, memory-mapped device)
# ACPI表示高级配置和电源管理接口
03h  ACPI Reclaim Memory (usable by OS after reading ACPI tables)
04h  ACPI NVS Memory (OS is required to save this memory between NVS sessions)
othernot defined yet -- treat as Reserved
```

BIOS会将得到的信息写入内存es:di，保存地址范围描述符结构的缓冲区就是e820map，他的定义如下：

```
struct e820map {
    int nr_map;
        struct {
            long long addr;
            long long size;
            long type;
        } map[E820MAX];
};
//常数值是已经定义好的20，是其中的实体个数
#define E820MAX             20      // number of entries in E820MAP
```

探测的代码也在下面贴出来了，

```
probe_memory:
//对0x8000处的32位单元清零,即给位于0x8000处的
//struct e820map的成员变量nr_map清零
movl $0, 0x8000
xorl %ebx, %ebx
```

```
//表示设置调用INT 15h BIOS中断后，BIOS返回的映射地址描述符的起始地址
movw $0x8004, %di
start_probe:
movl $0xE820, %eax // INT 15的中断调用参数
//设置地址范围描述符的大小为20字节，其大小等于struct e820map的成员变量map的大小
movl $20, %ecx
//设置edx为534D4150h (即4个ASCII字符“SMAP”)，这是一个约定
movl $SMAP, %edx
//调用int 0x15中断，要求BIOS返回一个用地址范围描述符表示的内存段信息
int $0x15
//如果eflags的CF位为0，则表示还有内存段需要探测
jnc cont
//探测有问题，结束探测
movw $12345, 0x8000
jmp finish_probe
cont:
//设置下一个BIOS返回的映射地址描述符的起始地址
addw $20, %di
//递增struct e820map的成员变量nr_map
incl 0x8000
//如果INT0x15返回的ebx为零，表示探测结束，否则继续探测
cmpl $0, %ebx
jnz start_probe
finish_probe:
```

上面的代码我也没看太懂，但有一件事很重要就是BIOS探测出来的结果存在了0x8000,之后page_init函数会来这里找e820map来完成对机器的物理内存管理。

管不了那么多了，直接来看pmm.c中page_init的代码：

```
//这玩意定义的就是上面我们说的type of address range
#define E820_ARM            1       // address range memory
#define E820_ARR            2       // address range reserved
/* All physical memory mapped at this address */
#define KERNBASE            0xC0000000
#define KMEMSIZE            0x38000000                  // the maximum amount of
physical memory
#define KERNTOP             (KERNBASE + KMEMSIZE)
//首先，这是第一段代码
//这一行的意思就是去找BIOS写好的那个结构，当然毫无疑问的，0x8000当然要从kernelBase开始算起，
当然，这个东西是虚拟地址
struct e820map *memmap = (struct e820map *)(0x8000 + KERNBASE);
    uint64_t maxpa = 0;

    cprintf("e820map:\n");
    int i;
//遍历每一个探测到的计算机内存块
    for (i = 0; i < memmap->nr_map; i ++) {
        //找到起始地址和终止地址
        uint64_t begin = memmap->map[i].addr, end = begin + memmap->map[i].size;
        cprintf("  memory: %08llx, [%08llx, %08llx], type = %d.\n",
                memmap->map[i].size, begin, end - 1, memmap->map[i].type);
        //如果这个内存块的类型是操作系统可用的，就查一下，我最多能使用的地址空间变了没有
        if (memmap->map[i].type == E820_ARM) {
```

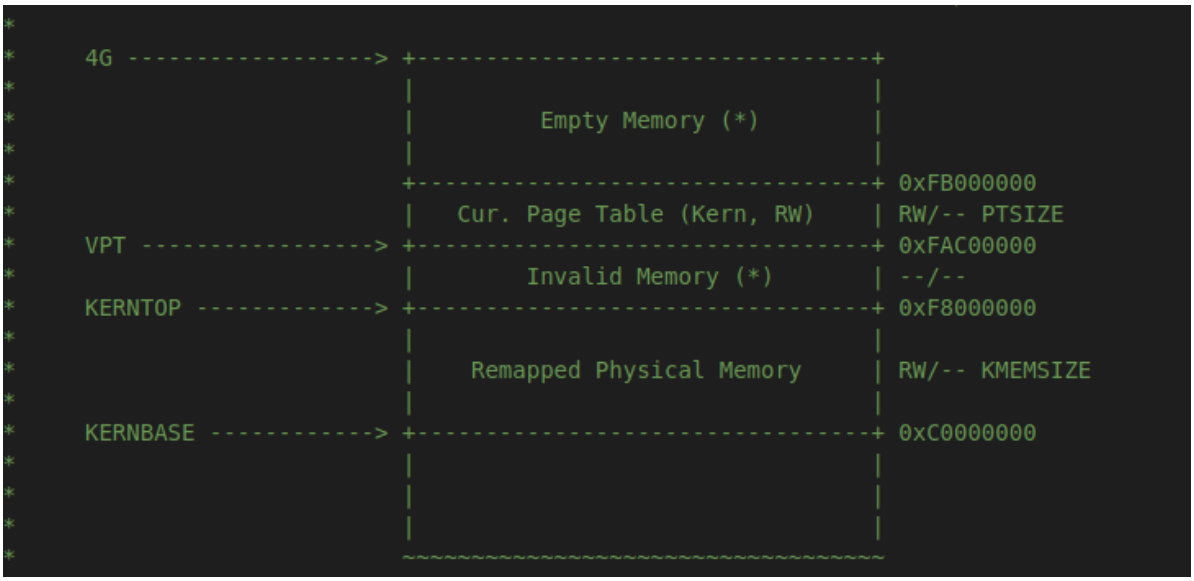
```
            //这两个是啥意思我解释下，就是如果有更大的end说明有更大的maxpa，因为探测出的都是物理地址，没有做虚拟映射，所以end和maxpa实际上是一致的。而第二个判断根据助教的意思就是如果begin的大小超过了KMEMSIZE是不会有映射的，因为更高的地址是不可用的。
            if (maxpa < end && begin < KMEMSIZE) {
                maxpa = end;
            }
        }
    }
//设定了一个上界，不能超过KMEMSIZE。因为maxpa的概念是我最多能用的物理地址是多少所以他的上线就设在KMEMSIZE
//注意理解kmemsize的含义，他的含义是我最多能使用的kernel的大小是多少，而不是最后的地址是多少
    if (maxpa > KMEMSIZE) {
        maxpa = KMEMSIZE;
    }
// “end”表示BSS段的结束地址,如果你忘记了BSS段的位置的话可以去看一看实验指导手册第152页的图片
    extern char end[];
//算一下要管理这么大的空闲需要多少页
    npage = maxpa / PGSIZE;
    pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);//这里pages是虚拟地址
//然后把所有操作系统能管的内存全部置成reserved
//到这里似乎能够回答那个问题了，看吧，所有操作系统能用的页全都被置成reserved了，所以你在分配它的时候才要把他解开。
    for (i = 0; i < npage; i ++) {
        SetPageReserved(pages + i);
    }
```

```
//第二阶段
//先计算一个从pages开始能留给其他进程分配的内存空间，这个空间并不是从end开始就可以，而是还要跑去存储管理物理页的Page所占的空间，剩下的才能留给其他进程使用
uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) * npage);

    for (i = 0; i < memmap->nr_map; i ++) {
        uint64_t begin = memmap->map[i].addr, end = begin + memmap->map[i].size;
        if (memmap->map[i].type == E820_ARM) {
            //这个就很好理解了，要分配的物理页不能低于freePages
            if (begin < freemem) {
                begin = freemem;
            }
            //同理最大也不能超过KMEMSIZE
            if (end > KMEMSIZE) {
                end = KMEMSIZE;
            }
            //在这中间的块都是可以分配的，使用init_memmap进行分配，在这个时候这些页都还是reserved的状态，所以你知道为什么这个函数里面要有保证所有的页都是reserved的断言了吧，这样这些页就可以被当成分配给进程的物理页了
            if (begin < end) {
                begin = ROUNDUP(begin, PGSIZE);
                end = ROUNDDOWN(end, PGSIZE);
                if (begin < end) {
                    init_memmap(pa2page(begin), (end - begin) / PGSIZE);
                }
            }
        }
    }
```

然后再说一下这两张图的关系：这张图片表示的一个机器的内存空间的物理地址！！！什么ucore里的0xc0000000在这里都不好用，因为这个就是实打实的物理地址

图3

这张图片的含义就是虚拟地址了，我们可以简单的理解为下面这张图里的0XC0000000就代表着上面图里的0x00000000，下图中的内核空间就对应着上面图中的从0x00000000到实际物理内存空间结束地址



challenge1：buddy-system

参考网址：通过ucore学习OS（X）：实现buddy system（伙伴系统） - 知乎 (zhihu.com)

```
//相当于总共有2^14页可以分
//为什么要+1不清楚？
#define MAX_BUDDY_ORDER 14
/* buddy system 的结构体 */
typedef struct {
    unsigned int max_order;                                 // 伙伴二叉树的层数
    list_entry_t free_array[MAX_BUDDY_ORDER + 1];       // 链表数组(现在默认有14层，即2^14 = 16384个可分配物理页)，每个数组元素都一个free_list头
    unsigned int nr_free;                                   // 伙伴系统中剩余的空闲页总数
} free_buddy_t;
```

```
#include <pmm.h>
#include <list.h>
#include <string.h>
#include <buddy_pmm.h>

free_buddy_t buddy_s;
#define buddy_array (buddy_s.free_array) //链表数组
#define max_order (buddy_s.max_order) //最大的层数
#define nr_free (buddy_s.nr_free) //剩余的空闲块

// 第一个能被分配的物理页页号（用于buddy system中寻找伙伴块）
ppn_t first_ppn = 0;

// 判断是否是二的整数倍
static int IS_POWER_OF_2(size_t n) {
    //https://blog.csdn.net/navyifanr/article/details/19496459
    if (n & (n - 1)) { //巧妙的利用位运算判断是否是二的整数倍
        return 0;
    }
    else {
        return 1;
    }
}
//计算是二的几次方--是二的n次方时使用
static unsigned int getOrderOf2(size_t n) {
    unsigned int order = 0;
    while (n >> 1) {
        n >>= 1;
        order ++;
    }
    return order;
}
//向下取整
static size_t ROUNDDOWN2(size_t n) {
    size_t res = 1;
    if (!IS_POWER_OF_2(n)) {
        while (n) {
            n = n >> 1;
            res = res << 1;
        }
```

```
        return res>>1;
    }
    else {
        return n;
    }
}
//向上取整
static size_t ROUNDUP2(size_t n) {
    size_t res = 1;
    if (!IS_POWER_OF_2(n)) {
        while (n) {
            n = n >> 1;
            res = res << 1;
        }
        return res;
    }
    else {//bu
        return n;//
    }
}

//在测试的时候使用，打印buddy array
static void
show_buddy_array(void) {
    cprintf("[!]BS: Printing buddy array:\n");
    for (int i = 0;i < max_order + 1;i ++) {
        cprintf("%d layer: ", i);
        list_entry_t *le = &(buddy_array[i]);
        while ((le = list_next(le)) != &(buddy_array[i])) {
            struct Page *p = le2page(le, page_link);
            cprintf("%d ", 1 << (p->property));
        }
        cprintf("\n");
    }
    cprintf("---------------------------\n");
    return;
}

/*
 *  初始化buddy结构体
 */
//这个数组像一个管理堆结构的数组，就像软件安全那个考试题一样，根据数组的索引来判断每一个数组项后面跟随的块的大小是多少，数组里面也存了一个指向链表首的指针，数组里只维护还没被分配的块
static void
buddy_init(void) {
    // 初始化链表数组中的每个free_list头
    for (int i = 0;i < MAX_BUDDY_ORDER;i ++){
        //buddy_array就是那个真正的数组，但是需要+1？和上面代码块的问题一样，为什么要+1？
        list_init(buddy_array + i);
    }
    max_order = 0;
    nr_free = 0;
    return;
}
```

```
/*
 *  获取以page页为头页的块的伙伴块
 *  从函数名上看猜测是返回一个块的buddy块
 *  first_ppn表示第一个可分配物理内存页在pages数组的下标.用代码中的异或计算便可得到伙伴块的头页在pages数组中的下标
 */
static struct Page*
buddy_get_buddy(struct Page *page) {
    //有多少个页
    unsigned int order = page->property;//拿到阶数
    //这里注意，firstppn也是要从pages开始的，因为ppn其实也是可以给进程能分配的第一个块的地址
    unsigned int buddy_ppn = first_ppn + ((1 << order) ^ (page2ppn(page) - first_ppn));//我觉得可以姑且理解为拿到他的buddy_ppn相对于pages的偏移是多少
    cprintf("[!]BS: Page NO.%d 's buddy page on order %d is: %d\n", page2ppn(page), order, buddy_ppn);
    if (buddy_ppn > page2ppn(page)) {//buddy的首page在free的首page之后
        return page + (buddy_ppn - page2ppn(page));
    }
    else {
        return page - (page2ppn(page) - buddy_ppn);
    }

}

//在这里因为作者说qemu只会模拟初始化一块内存，所以init_memmap也只会被调用一次，所以可以直接指定块的大小。ucore中实际可以给我们分配的大约不到30000页，但是向下取整直接少了14000页，有点离谱，但反正重点也不在这里，无所谓
static void
buddy_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    size_t pnum;
    unsigned int order;
    pnum = ROUNDDOWN2(n);        // 将页数向下取整为2的幂
    pnum = 8; // 为了test而设置的，真正用的时候把这一行注释掉就行，pnum由上一行来决定。
    order = getOrderOf2(pnum);   // 求出页数对应的2的幂
    cprintf("[!]BS: AVA Page num after rounding down to powers of 2: %d = 2^%d\n", pnum, order);
    struct Page *p = base;
    // 初始化pages数组中范围内的每个Page，和default中的保持基本一致
    for (; p != base + pnum; p ++) {
        assert(PageReserved(p));
        p->flags = 0;
        p->property = -1;   // 全部初始化为非头页
        set_page_ref(p, 0);
    }
    //这些条件的初始化都是建立在一次初始化整个块的基础上的
    max_order = order;
    nr_free = pnum;
    //添加到最大的数组的一个元素中
    list_add(&(buddy_array[max_order]), &(base->page_link)); // 将第一页base插入数组的最后一个链表，作为初始化的最大块--16384,的头页
    base->property = max_order;                          // 将第一页base的property设为最大块的2幂
    return;
}
```

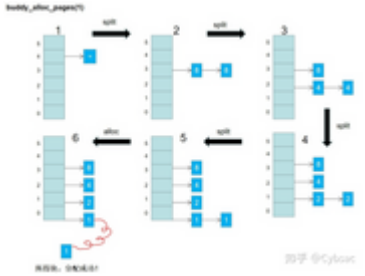
```
// 默认分裂数组中第n条链表的第一块
/*
buddy_split函数专门用在buddy_alloc_pages函数中，删除给定阶的对应链表中第一个空闲块，并将分裂出来的两个空闲块插入比原阶小1的对应链表中。
*/
static void buddy_split(size_t n) {
    assert(n > 0 && n <= max_order);
    assert(!list_empty(&(buddy_array[n])));//要拆分的肯定不为空
    cprintf("[!]BS: SPLITTING!\n");
    struct Page *page_a;
    struct Page *page_b;

    page_a = le2page(list_next(&(buddy_array[n])), page_link);//我觉得这个是指向了要被分的那个块的首页
    page_b = page_a + (1 << (n - 1));//page_b也平分原来的页，向后挪动一半相当于
    //各分一半
    page_a->property = n - 1;
    page_b->property = n - 1;

    //删掉原来的，然后插入分裂之后的
    list_del(list_next(&(buddy_array[n])));
    list_add(&(buddy_array[n-1]), &(page_a->page_link));
    list_add(&(page_a->page_link), &(page_b->page_link));

    return;
}

/*
step 1：把请求的物理页数向上取整至2的幂，并求出此时的阶order

step 2：查找伙伴数组对应位置，即buddy_array[order]处的链表是否为空。若不为空则直接分配，跳到step 4。若为空则继续

step 3：从当前阶order开始，向上寻找buddy_array的首个非空链表。找到的第一个非空链表下标为i，则说明当前空闲的最小块大小为2^i，我们执行buddy_split对其进行分裂。跳到step 2

step 4：返回所分配物理块的头页

*/
static struct Page *
buddy_alloc_pages(size_t n) {
    // require n > 0, or panic
    assert(n > 0);

    // if the number of required pages beyond what we have currently, return NULL
    //要的比有的还多直接返回
    if (n > nr_free) {
        return NULL;
    }

    struct Page *page = NULL;
    size_t pnum = ROUNDUP2(n);  // 处理所要分配的页数，向上取整至2的幂
```

```
    size_t order = 0;

    //order就是要分配的页数大小
    order = getOrderOf2(pnum);  // 求出所需页数对应的幂pow
    cprintf("[!]BS: Allocating %d-->%d = 2^%d pages ...\n", n, pnum, order);
    cprintf("[!]BS: Buddy array before ALLOC:\n");
    show_buddy_array();
find:
    // 若pow对应的链表中含有空闲块，则直接分配
    if (!list_empty(&(buddy_array[order]))) {//如果不空调用下面的代码
        page = le2page(list_next(&(buddy_array[order])), page_link);//别忘了这个函数是干吗的，链表元素是嵌入在类对象中的，这个函数就是通过链表对象把类对象拿出来，要下一个是因为数组里面也是存了指针。仿写default_pmm的内容。
        list_del(list_next(&(buddy_array[order])));//直接把这一页删掉，因为大小正好，所以不需要进行额外操作
        SetPageProperty(page); // 将分配块的头页设置为已被占用
        cprintf("[!]BS: Buddy array after ALLOC NO.%d page:\n", page2ppn(page));
        show_buddy_array();
        goto done;
    }
    else {
        for (int i = order;i < max_order + 1;i ++) {//一直往后找，直到找到一个数组项满足要求为止（够大、有空位置）
            // 找到pow后第一个非空链表，分裂空闲块
            if (!list_empty(&(buddy_array[i]))) {//不为空
                buddy_split(i);//需要分裂一个空闲块，分裂完可能就有了，所以返回重新找，如果没找到，继续向上，这个时候你能找到的块肯定就比上一次找的小了一级
                cprintf("[!]BS: Buddy array after SPLITT:\n");
                show_buddy_array();
                goto find;      // 重新检查现在是否可以分配
            }
        }
    }

done:
    nr_free -= pnum;//总共还有多少块要剪掉
    cprintf("[!]BS: nr_free: %d\n", nr_free);
    cprintf("----------------------------\n");
    return page;//返回分配的块
}

/*
释放比较复杂。

简单来说，我们在对一个物理块进行释放后，需要检查其伙伴块是否空闲。若空闲则将这两块合并成更大的块，继续检查更大块的伙伴块是否空闲，一直重复此过程，直到无法再合并。

buddy_get_buddy函数的作用是，返回给定块的伙伴块。

这里引用了一个全局变量first_ppn，first_ppn是新声明的全局变量，表示第一个可分配物理内存页在pages数组的下标.用代码中的异或计算便可得到伙伴块的头页在pages数组中的下标.

buddy_free_pages的思路如下:

step 1: 将阶为i，大小为2^i的当前块插入伙伴数组的第i条链表。获取当前块的伙伴块
```

```
step 2:检查当前阶为i, 大小为2^i块的伙伴块是否空闲.若不空闲则条至step 6

step 3: 从第i条链表中将当前块与伙伴块删除

step 4: 判断当前块的伙伴块是否为左块(伙伴块头页地址小于当前块), 若为左块则将当前块的指针指向伙伴块

step 5:将当前块的property加一, i = i +1, 跳至step 1

step 6: 释放完成返回。
*/

static void
buddy_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    unsigned int pnum = 1 << (base->property);//回来的页的大小，因为是二的整数倍所以直接左移
    assert(ROUNDUP2(n) == pnum);//判断必须能被二整除，因为分配的时候都是按2的整数倍整除的，所以回来也理所应当回来2的整数倍的页
    cprintf("[!]BS: Freeing NO.%d page leading %d pages block: \n", page2ppn(base), pnum);
    //base就是要合并的
    struct Page* left_block = base;
    //这个姑且理解为base的伙伴块吧
    struct Page *buddy = NULL;
    //如果左右块反了交换时用到的中间块
    struct Page* tmp = NULL;

    buddy = buddy_get_buddy(left_block);
    list_add(&(buddy_array[left_block->property]), &(left_block->page_link));
    cprintf("[!]BS: add to list\n");
    show_buddy_array();
    // 当伙伴块空闲，且当前块不为最大块时
    while (!PageProperty(buddy) && left_block->property < max_order) {
        cprintf("[!]BS: Buddy free, MERGING!\n");
        if (left_block > buddy) { // 若当前左块为更大块的右块
            left_block->property = -1; //这个指令没看懂，为什么要减掉然后后面再加回来？？？？
            ClearPageProperty(left_block);//清除掉这个标识位
            tmp = left_block;
            left_block = buddy;
            buddy = tmp;
        }
        //删掉原有的
        list_del(&(left_block->page_link));
        list_del(&(buddy->page_link));
        left_block->property += 1; //这个我觉得也没必要吧，直接set就好了
        list_add(&(buddy_array[left_block->property]), &(left_block->page_link)); // 头插入相应链表
        show_buddy_array();
        buddy = buddy_get_buddy(left_block);
    }
    cprintf("[!]BS: Buddy array after FREE:\n");
    ClearPageProperty(left_block); // 将回收块的头页设置为空闲
```

```
    nr_free += pnum;//剩余的多了就加上
    show_buddy_array();

    cprintf("[!]BS: nr_free: %d\n", nr_free);
    cprintf("--------------------------\n");
    return;
}

static size_t
buddy_nr_free_pages(void) {
    return nr_free;
}//返回还剩多少页可以分


static void
basic_check(void) {
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;
    assert((p0 = alloc_page()) != NULL);
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);
    free_page(p0);
    free_page(p1);
    free_page(p2);
    show_buddy_array();

    assert((p0 = alloc_pages(4)) != NULL);
    assert((p1 = alloc_pages(2)) != NULL);
    assert((p2 = alloc_pages(1)) != NULL);
    free_pages(p0, 4);
    free_pages(p1, 2);
    free_pages(p2, 1);
    show_buddy_array();

    assert((p0 = alloc_pages(3)) != NULL);
    assert((p1 = alloc_pages(3)) != NULL);
    free_pages(p0, 3);
    free_pages(p1, 3);

    show_buddy_array();
}
static void
buddy_check(void) {
    basic_check();
}
const struct pmm_manager buddy_pmm_manager = {
    .name = "buddy_pmm_manager",
    .init = buddy_init,
    .init_memmap = buddy_init_memmap,
    .alloc_pages = buddy_alloc_pages,
    .free_pages = buddy_free_pages,
    .nr_free_pages = buddy_nr_free_pages,
    .check = buddy_check,
};
```

alloc分配所示图像：



验证的效果图：