练习一：实现first-fit 连续物理内存分配算法

根据实验指导书中的实验执行流程概述，先了解分析ucore如何对物理内存进行管理，再完成实验练习。

在对物理内存进行管理之前，需要先进行物理内存布局的探测，探测得到的内存映射存放在e820map中。物理内存以页的形式进行管理，页的信息保存在Page结构中，而Page以链式结构保存在链表free_area_t中。对于物理内存的管理，ucore是通过定义一个pmm_manager实现的，其中包含了初始化需要管理的物理内存空间，初始化链表，内存空间分配释放等功能的函数。在内核启动后，会初始化pmm_manager，调用page_init，使用init_memmap将e820map中的物理内存纳入物理内存页管理（将空闲页初始化并加入链表），接下来就可以使用pmm_manager中的空间分配和释放等函数进行内存管理了。

1.探测系统物理内存布局

对物理内存空间进行管理，首先要对物理内存布局进行探测。本实验中是在bootloader进入保护模式之前通过BIOS中断获取。基于INT 15h，通过e820h中断探测物理内存信息。并将探测到的物理内存对应的映射存放在物理地址0x8000处，定义e820map结构保存映射。在bootasm.S中有内存探测部分的代码，e820map的结构定义则在memlayout.h中。

```
//memlayout.h
struct e820map {
    int nr_map;
    struct {
        uint64_t addr;
        uint64_t size;
        uint32_t type;
    } __attribute__((packed)) map[E820MAX];
};
//bootasm.S
probe_memory:
    movl $0, 0x8000             #存放内存映射的位置
    xorl %ebx, %ebx
    movw $0x8004, %di           #0x8004开始存放map
start_probe:
    movl $0xE820, %eax          #设置int 15h的中断参数
    movl $20, %ecx              #内存映射地址描述符的大小
    movl $SMAP, %edx
    int $0x15
    jnc cont                    #CF为0探测成功
    movw $12345, 0x8000
    jmp finish_probe
cont:
    addw $20, %di               #下一个内存映射地址描述符的位置
    incl 0x8000                 #nr_map+1
    cmpl $0, %ebx               #ebx存放上次中断调用的计数值，判断是否继续进行探测
    jnz start_probe
```

2.以页为单位管理物理内存

物理内存是以页为单位进行管理的。探测可用物理内存空间的情况后，就可以建立相应的数据结构来管理物理内存页了。每个内存页使用一个Page结构来表示，Page的定义在memlayout.h中，如下：

```
//memlayout.h中的Page定义
struct Page {
    int ref;                        // 引用计数
    uint32_t flags;                 // 状态标记
    unsigned int property;          // 在first-fit中表示地址连续的空闲页的个数，空闲块头部才有该属性
    list_entry_t page_link;         // 双向链表
};
//状态
#define PG_reserved                 0       //表示是否被保留
#define PG_property                 1       //表示是否是空闲块第一页
```

初始情况下，空闲物理页可能都是连续的，随着物理页的分配与释放，大的连续内存空闲块会被分割为地址不连续的多个小连续内存空闲块。为了管理这些小连续内存空闲块，使用一个双向链表进行管理，定义free_area_t数据结构，包含一个list_entry结构的双向链表指针，记录当前空闲页个数的无符号整型：

```
typedef struct {
    list_entry_t free_list;         // 链表
    unsigned int nr_free;           // 空闲页个数
} free_area_t;
```

对以页为单位进行内存管理，还有两个问题需要解决。一个是找到管理页级物理内存空间所需的Page结构的内存空间的位置，另一个是要找到空闲空间开始的位置。

可以根据内存布局信息找到最大物理内存地址maxpa计算出页的个数，并计算出管理页的Page结构需要的空间。ucore的结束地址（即.bss段的结束地址，用全局变量end表示）以上的空间空闲，从这个位置开始存放Pages结构，而存放Pages结构的结束的位置以上就是空闲物理内存空间。将空闲的物理内存空间使用init_memmap()函数纳入物理内存管理器，部分代码如下：

```
//pmm.c中的page_init的部分代码
    npage = maxpa / PGSIZE;                                    //页数
    pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);       //Page结构的位置
    for (i = 0; i < npage; i ++) {
        SetPageReserved(pages + i);                            //每一个物理页默认标记为保留
    }
    //空闲空间起始
    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) {
            if (begin < freemem) {
                begin = freemem;                               //限制空闲地址最小值
            }
            if (end > KMEMSIZE) {
                end = KMEMSIZE;                                //限制空闲地址最大值
            }
            if (begin < end) {
                begin = ROUNDUP(begin, PGSIZE);                //对齐地址
                end = ROUNDDOWN(end, PGSIZE);
                if (begin < end) {
                    //将空闲内存块映射纳入内存管理
                    init_memmap(pa2page(begin), (end - begin) / PGSIZE);
                }
            }
        }
    }
```

3.物理内存空间管理的初始化

ucore中建立了一个物理内存页管理框架对内存进行管理，定义如下：

```
//pmm.h中的pmm_manager定义
struct pmm_manager {
    const char *name;                                 // 管理器名称
    void (*init)(void);                               // 初始化管理器
    void (*init_memmap)(struct Page *base, size_t n); // 设置并初始化可管理的内存空间
    struct Page *(*alloc_pages)(size_t n);            // 分配n个连续物理页，返回首地址
    void (*free_pages)(struct Page *base, size_t n);  // 释放自Base起的连续n个物理页
    size_t (*nr_free_pages)(void);                    // 返回剩余空闲页数
    void (*check)(void);                              // 用于检测分配释放是否正确
};
```

其中，init_memmap用于对页内存管理的Page的初始化，在上面提到的page_init负责确定探查到的物理内存块与对应的struct Page之间的映射关系，Page的初始化工作由内存管理器的init_memmap()来完成。而init则用于初始化已定义好的free_area_t结构的free_area。

在内核初始化的kern_init中会调用pmm_init()，pmm_init()中会调用init_pmm_manager()进行初始化，使用默认的物理内存页管理函数，即使用default_pmm.c中定义的default_init等函数进行内存

管理，本实验中需要实现的分配算法可以直接通过修改这些函数进行实现。

```
//pmm.c中的init_pmm_manager()定义，在pmm_init()中调用，在kernel_init()中初始化
static void
init_pmm_manager(void) {
    pmm_manager = &default_pmm_manager;              //pmm_manager指向default_pmm_manager
    cprintf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}
//default_pmm.c中的default_init()
static void
default_init(void) {
    list_init(&free_list);                           //初始化链表
    nr_free = 0;
}
//init_memmap
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));                     //检查是否为保留页
        //property设置为0，只有空闲块的第一页使用该变量,标志位清0
        p->flags = p->property = 0;
        set_page_ref(p, 0);                          //引用计数为0
    }
    base->property = n;                              //设置第一页的property
    SetPageProperty(base);
    nr_free += n;                                    //空闲页+n
    list_add(&free_list, &(base->page_link));        //将空闲块加入列表
}
```

4.first-fit算法的实现

在pmm_manager完成链表的初始化，page_init完成页的初始化后，就可以使用pmm_manager的函数进行内存空间管理了。该练习中使用默认的pmm_manager进行管理，其中的default_init和default_init_memmap可以直接使用，只需要修改default_alloc_pages，default_free_pages，实现first-fit算法以及内存空间释放。

使用的函数及宏定义

使用的链表相关操作和宏定义如下。ucore对空闲内存空间管理使用的链表与常见的链表不同，链表只包含了前后节点的信息，而链表包含在Page结构中，这样做是为了实现链表通用性（c语言中并不支持泛型功能，因此采用这种方式）。而通过链表节点获取节点数据是通过宏定义le2page实现的。链表定义和部分相关函数如下：

```
//双向链表的定义
struct list_entry {
    struct list_entry *prev, *next;
};
typedef struct list_entry list_entry_t;
 //返回下一个节点
static inline list_entry_t *
list_next(list_entry_t *listelm) {
    return listelm->next;
}
//删除当前节点
static inline void
list_del(list_entry_t *listelm) {
    __list_del(listelm->prev, listelm->next);
}
//前插节点，还有类似的list_add_after
static inline void
list_add_before(list_entry_t *listelm, list_entry_t *elm) {
    __list_add(elm, listelm->prev, listelm);
}
```

le2page通过page_link地址减去其相对于Page结构的偏移，实现从链表节点找到对应的数据。

```
#define le2page(le, member)
    to_struct((le), struct Page, member
#define to_struct(ptr, type, member)
    ((type *)((char *)(ptr) - offsetof(type, member))))
```

还有一些其他函数需要使用，以设置Page的信息。

```
//pmm.h中定义的设置引用计数的函数
static inline void
set_page_ref(struct Page *page, int val) {
    page->ref = val;
}
//memlayou.h中的关于页标志位设置的宏定义
#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))
```

使用first-fit实现default_alloc_pages

使用first-fit实现空闲空间分配，只需要遍历空闲链表，找到合适的块大小，重新设置标志位并从链表中删除该页，如果找到的块大于需要的大小，则需要进行分割，最后更新空闲页数，返回分配好的页块的地址。

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    //空闲页不够，直接返回
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    //寻找合适的空闲块
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        ClearPageProperty(page);                //page已被分配
        //如果空闲块过大则进行分割
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);                 //空闲块
            list_add_after(&(page->page_link), &(p->page_link));
        }
        list_del(&(page->page_link));
        nr_free -= n;
    }
    return page;
}
```

default_free_pages

释放空间，首先确定需要释放的n都页是未被保留且非空闲的页，然后将这些页的标志位和引用计数清零，并将释放空间的第一页的property设置为n，即空闲块共有n页。接下来完成空闲块的合并，对于相邻的空间，将高地址的块合并到低地址的空闲块中，删除被合并的块，并重新设置空闲块的第一页的property，最后由于空闲链表按地址空间由低到高排列空闲块，还需要找到插入的位置。

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    //将需要释放的页设为空闲状态
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);                //引用计数清0
    }
    SetPageProperty(base);
    base->property = n;                    //空闲块第一页，property=n
    //空闲块的合并
    list_entry_t *le = list_next(&free_list);
    while (le != &free_list) {
        p = le2page(le, page_link);
        le = list_next(le);
        if (base + base->property == p) {
            base->property += p->property;
            p->property=0;                 //不再是空闲块第一页，property清0
            SetPageProperty(base);         //设置为空闲块第一页
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            p->property += base->property;
            base->property=0;
            SetPageProperty(p);             //设置为空闲块第一页
            list_del(&(base->page_link));
            base=p;                         //更新空闲块第一页
        }
    }
    le = list_next(&free_list);
    //找到插入位置
     while (le != &free_list)
    {
        p = le2page(le, page_link);
        if (base + base->property <= p)
        {
            break;
        }
        le = list_next(le);
    }
    //将base插入到正确位置
    list_add_before(le, &(base->page_link));
    nr_free += n;
}
```

改进空间

每次查找链表都需要进行遍历，时间复杂度较高，且在空闲链表开头会产生许多小的空闲块，仍然有优化空间。

练习2：实现寻找虚拟地址对应的页表项

1.段页式管理

在保护模式中，内存地址分成三种：逻辑地址、线性地址和物理地址。逻辑地址即是程序指令中使用的地址，物理地址是实际访问内存的地址。段式管理的映射将逻辑地址转换为线性地址，页式管理的映射将线性地址转换为物理地址。

ucore中段式管理仅为一个过渡，逻辑地址与线性地址相同。而页式管理是通过二级页表实现的，地址的高10位为页目录索引，中间10位为页表索引，低12位为偏移（页对齐，低12位为0）。一级页表的起始物理地址存放在 boot_cr3中。

2.页目录项和页表项的组成

问题一：请描述页目录项（Pag Director Entry）和页表（Page Table Entry）中每个组成部分的含义和以及对ucore而言的潜在用处

页目录项的组成

- 前20位表示该PDE对应的页表起始位置
- 第9-11位保留给OS使用
- 第8位可忽略
- 第7位用于设置Page大小，0表示4KB
- 第6位为0
- 第5位表示该页是否被写过
- 第4位表示是否需要进行缓存
- 第3位表示CPU是否可直接写回内存
- 第2位表示该页是否可被任何特权级访问
- 第1位表示是否允许读写
- 第0位为该PDE的存在位

页表项的组成

- 前20位表示该PTE指向的物理页的物理地址
- 第9-11位保留给OS使用
- 第8位表示在 CR3 寄存器更新时无需刷新 TLB 中关于该页的地址
- 第7位恒为0
- 第6位表示该页是否被写过
- 第5位表示是否可被访问
- 第4位表示是否需要进行缓存

- 第0-3位与页目录项的0-3位相同

3.页访问异常的处理

问题二：如果ucore执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情?

当发生页访问异常时，硬件需要将发生错误的地址存放在cr2寄存器中，向栈中压入EFLAGS，CS，EIP等，如果异常发生在用户态，还需要进行特权级切换，最后根据中断描述符找到中断服务例程，接下来由中断服务例程处理该异常。

4.实现get_pte寻找页表项

练习二要求实现get_pte函数，使该函数能够找到传入的线性地址对应的页表项，返回页表项的地址。为了完成该函数，需要了解ucore中页式管理的相关函数及定义。

```
//PDX(la): la为线性地址，该函数取出线性地址中的页目录项索引
#define PDX(la) ((((uintptr_t)(la)) >> PDXSHIFT) & 0x3FF)
//取出线性地址中的页表项索引
#define PTX(la) ((((uintptr_t)(la)) >> PTXSHIFT) & 0x3FF)
//KADDR(pa): 返回物理地址pa对应的虚拟地址
#define KADDR(pa) {...}
//set_page_ref(page,1): 设置该页引用次数为VAL
static inline void
set_page_ref(struct Page *page, int val) {
    page->ref = val;
}
//page2pa: 找到page结构对应的页的物理地址
static inline uintptr_t
page2pa(struct Page *page) {
    return page2ppn(page) << PGSHIFT;
}
//alloc_page(): 分配一页
#define alloc_page() alloc_pages(1)
//页目录项、页表项的标志位
#define PTE_P           0x001                   // 存在位
#define PTE_W           0x002                   // 是否可写
#define PTE_U           0x004                   // 用户是否可访问
//页目录项和页表项类型
typedef uintptr_t pte_t;
typedef uintptr_t pde_t;
//以下两个函数用于取出页目录项中的页表地址，取出页表项中的页地址
#define PDE_ADDR(pde)   PTE_ADDR(pde)
#define PTE_ADDR(pte)   ((uintptr_t)(pte) & ~0xFFF)
```

需要完成的get_pte函数原型如下，其中pgdir是一级页目录的起始地址，la为线性地址，creat表示是否可以为页表分配新的页。

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create)
```

取出线性地址对应的页表项，首先在页目录中找到对应的页目录项，并判断是否有效（二级页目录是否存在）。如果不存在则根据create判断是否需要创建新的一页存放页表，如果需要则调用alloc_page分配新的一页，将这一页的地址结合标志位设置为页目录项。最后返回页表项的线性地址，使用PDE_ADDR取出页目录项中保存的页表地址，再加上使用PTX从线性地址取出的页表索引，就找到了页表项的位置，使用KADDR转换为线性地址返回（先将页表地址转换为线性地址再加索引同样可行），注意类型转换。

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    pde_t *pdep = &pgdir[PDX(la)];              //找到页目录项
    if (!*pdep&PTE_P) {                         //判断二级页表是否存在
        if(!create) return NULL;                //create=0则直接返回
        else{
            struct Page* page=alloc_page();     //分配一页用于存放二级页表
            if(page==NULL) return NULL;
            set_page_ref(page,1);               //引用计数设为1
            uintptr_t pte_pa=page2pa(page);     //分配的页物理地址
            memset(KADDR(pte_pa),0,PGSIZE);     //清除页面内容
            *pdep=pte_pa | PTE_P | PTE_W | PTE_U;
        }
    }
    return ((pte_t*)KADDR(PDE_ADDR(*pdep)))+PTX(la);     // 返回页表项的地址
}
```

练习3：释放某虚地址所在的页并取消对应二级页表项的映射

1.Page与页目录项和页表项的关系

问题一：数据结构Page的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系？如果有，其对应关系是啥？

当页目录项与页表项均有效时，有对应关系。每个页目录项记录一个页表的位置，每个页表项则记录一个物理页的位置，而Page变量保存的就是物理页的信息，因此每个有效的页目录项和页表项，都对应了一个page结构，即一个物理页的信息。

2.实现虚拟地址与物理地址相等

问题二：如果希望虚拟地址与物理地址相等，则需要如何修改lab2，完成此事？

由附录可知，在lab1中，虚拟地址=线性地址=物理地址，ucore的起始虚拟地址（也即物理地址）从0x100000开始。而在lab2中建立了从虚拟地址到物理地址的映射，ucore的物理地址仍为

0x100000，但虚拟地址变为了0xC0100000，即最终建立的映射为：virt addr = linear addr = phy addr + 0xC0000000。

```
//memlayout.h给出了直观的映射关系
*     4G ------------------> +---------------------------------+
*                            |                                 |
*                            |         Empty Memory (*)        |
*                            |                                 |
*                            +---------------------------------+ 0xFB000000
*                            |   Cur. Page Table (Kern, RW)    | RW/-- PTSIZE
*     VPT -----------------> +---------------------------------+ 0xFAC00000
*                            |        Invalid Memory (*)       | --/--
*     KERNTOP -------------> +---------------------------------+ 0xF8000000
*                            |                                 |
*                            |    Remapped Physical Memory     | RW/-- KMEMSIZE
*                            |                                 |
*     KERNBASE ------------> +---------------------------------+ 0xC0000000
*                            |                                 |
*                            |                                 |
*                            |                                 |
*                            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

只要取消这个映射，就可以实现虚拟地址和物理地址相等，将ld工具形成的ucore的虚拟地址修改为0x100000，就可以取消映射。

```
ENTRY(kern_entry)
SECTIONS {
            /* Load the kernel at this address: "." means the current address */
            . = 0xC0100000;         //修改为0x100000就可以实现虚拟地址=物理地址
            .text : {
                        *(.text .stub .text.* .gnu.linkonce.t.*)
            }
```

还需要在memlayout.h中将KERNBASE即虚拟地址基址设置为0，并关闭entry.S中对页表机制的开启。

```
//memlayout.h中定义的KERNBASE
#define KERNBASE 0x0
//页表机制开启
    # enable paging
    movl %cr0, %eax
    orl $(CR0_PE | CR0_PG | CR0_AM | CR0_WP | CR0_NE | CR0_TS | CR0_EM | CR0_MP), %eax
    andl $~(CR0_TS | CR0_EM), %eax
    movl %eax, %cr0         //将这句注释掉
```

pmm_init中的check_pgdir和check_boot_pgdir都假设kernbase不为0，对线性地址为0的地址进行页表查询等，因此会产生各种错误，可以将这两个函数注释掉。

3.实现page_remove_pte释放虚拟页并取消二级映射

本练习中需要完成的是page_remove_pte函数，该函数将传入的虚拟页释放，取消二级页表的映射，并且需清除TLB中对应的项。原型如下：

```
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep)
```

使用的相关函数定义如下：

```
//pte2page：找到页表项对应的页
struct Page *page pte2page(*ptep)
//free_page：释放页
free_page(page)
//page_ref_dec：引用计数-1
page_ref_dec(page)
//tlb_invalidate：清除TLB
tlb_invalidate(pde_t *pgdir, uintptr_t la)
```

对于传入的页表项，首先要判断其是否有效，如果有效，将引用计数-1，当引用计数为0时释放该页，再清0二级页表映射，使用tlb_invalidate清除对应的TLB项。最终实现如下：

```
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    //判断页表项是否有效
    if (*ptep & PTE_P) {
        struct Page *page = pte2page(*ptep);          //找到对应的页
        page_ref_dec(page);                           //引用计数-1
        if(page->ref==0) free_page(page);             //引用计数为0释放该页
        *ptep=0;                                      //清除二级页表映射
        tlb_invalidate(pgdir,la);                     //修改TLB
        return;
    }
    else return;
}
```

Challenge：buddy system（伙伴系统）分配算法

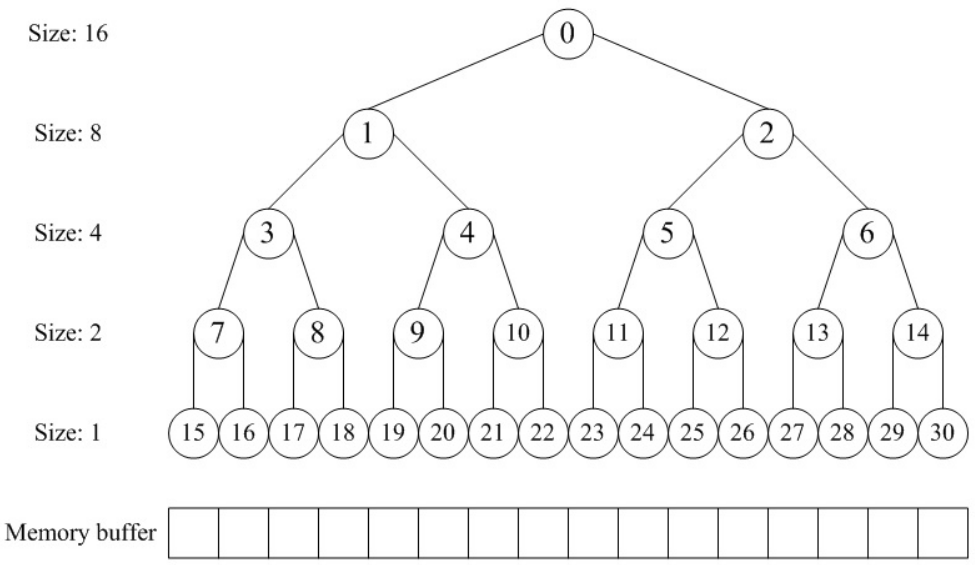
challenge没有自己完成ucore中的实现，主要是读懂和理解参考资料中的伙伴系统。

1.伙伴系统分配算法

伙伴系统是一种采用二分的方式分割和合并空闲空间的内存分配算法。空闲空间被视为2^n的空间，当有内存空间分配请求时，将空闲空间一分为二，直到刚好可以满足请求大小。在空间释放时，

分配程序会检查该块同大小的伙伴块是否空闲，如果是则可以进行合并，并继续上溯，直到完成全部可能的合并。

分配器使用伙伴系统，是通过数组形式的二叉树实现的。二叉树的节点标记相应的内存块是否使用，通过这些标记进行块的分离与合并。二叉树的情况如下图（总大小为16），其中节点数字为在数组中的索引：



2.伙伴系统的实现

下面是对实验指导书中给出的伙伴系统的分析。

首先是数据结构和一些宏定义，主要是伙伴系统定义，计算节点等：

```
struct buddy2 {
  unsigned size;//表明物理内存的总单元数
  unsigned longest[1]; //二叉树的节点标记，表明对应内存块的空闲单位
};
#define LEFT_LEAF(index) ((index) * 2 + 1)//左子树节点的值
#define RIGHT_LEAF(index) ((index) * 2 + 2)//右子树节点的值
#define PARENT(index) ( ((index) + 1) / 2 - 1)//父节点的值
#define IS_POWER_OF_2(x) (!((x)&((x)-1)))//x是不是2的幂
#define MAX(a, b) ((a) > (b) ? (a) : (b))//判断a，b大小
#define ALLOC malloc//申请内存
#define FREE free//释放内存
```

在分配时，分配的空间大小必须满足需要的大小，且为2的幂次方，以下函数用于找到合适的大小：

```
static unsigned fixsize(unsigned size) {//找到大于等于所需内存的2的倍数
  size |= size >> 1;
  size |= size >> 2;
  size |= size >> 4;
  size |= size >> 8;
  size |= size >> 16;
  return size+1;
}
```

接下来是分配器的初始化及销毁。初始化传入的参数是需要管理的内存空间大小，且这个大小应该是2的幂次方。在函数中node_size用于计算节点的大小，每次除2，初始化每一个节点。

```
struct buddy2* buddy2_new( int size ) {//初始化分配器
  struct buddy2* self;
  unsigned node_size;//节点所拥有的内存大小
  int i;

  if (size < 1 || !IS_POWER_OF_2(size))
    return NULL;

  self = (struct buddy2*)ALLOC( 2 * size * sizeof(unsigned));
  self->size = size;
  node_size = size * 2;

  for (i = 0; i < 2 * size - 1; ++i) {
    if (IS_POWER_OF_2(i+1))
      node_size /= 2;
    self->longest[i] = node_size;
  }
  return self;
}

void buddy2_destroy( struct buddy2* self) {
  FREE(self);
}
```

内存分配的实现如下，传入分配器，需要分配的空间大小，首先判断是否可以进行分配，并将空间大小调整为2的幂次方，然后进行分配。分配的过程为遍历寻找合适大小的节点，将找到的节点大小清0表示以被占用，并且需要更新父节点的值，最后返回的值为所分配空间相对于起始位置的偏移。

```
int buddy2_alloc(struct buddy2* self, int size) {
  unsigned index = 0;         //节点在数组的索引
  unsigned node_size;
  unsigned offset = 0;

  if (self==NULL)             //无法分配
    return -1;
  if (size <= 0)              //分配不合
    size = 1;
  else if (!IS_POWER_OF_2(size))//调整大小为2的幂次方
    size = fixsize(size);
  if (self->longest[index] < size)//可分配内存不足
    return -1;
  //从根节点开始向下寻找合适的节点
  for(node_size = self->size; node_size != size; node_size /= 2 ) {
    if (self->longest[LEFT_LEAF(index)] >= size)
      index = LEFT_LEAF(index);
    else
      index = RIGHT_LEAF(index);//左子树不满足时选择右子树
  }
  self->longest[index] = 0;      //将节点标记为已使用
  offset = (index + 1) * node_size - self->size;//计算偏移量
    //更新父节点
  while (index) {
    index = PARENT(index);
    self->longest[index] =
      MAX(self->longest[LEFT_LEAF(index)], self->longest[RIGHT_LEAF(index)]);
  }
  return offset;
}
```

内存释放时，先自底向上寻找已被分配的空间，将这块空间的大小恢复，接下来就可以匹配其大小相同的空闲块，如果块都为空闲则进行合并。

```
void buddy2_free(struct buddy2* self, int offset) {
  unsigned node_size, index = 0;
  unsigned left_longest, right_longest;
  //判断请求是否出错
  assert(self && offset >= 0 && offset < self->size);
  node_size = 1;
  index = offset + self->size - 1;
  //寻找分配过的节点
  for (; self->longest[index] ; index = PARENT(index)) {
    node_size *= 2;
    if (index == 0)                //如果节点不存在
      return;
  }
  self->longest[index] = node_size;//释放空间
  //合并
  while (index) {
    index = PARENT(index);
    node_size *= 2;

    left_longest = self->longest[LEFT_LEAF(index)];
    right_longest = self->longest[RIGHT_LEAF(index)];

    if (left_longest + right_longest == node_size) //如果可以则合并
      self->longest[index] = node_size;
    else
      self->longest[index] = MAX(left_longest, right_longest);
  }
}
```

以上就是参考资料中给出的伙伴系统的实现（另外还有两个函数分别用于返回当前节点大小和打印内存状态），在ucore中实现伙伴系统的原理相同，但需要对具体的页进行处理分配以及释放，完成对应的buddy.h头文件和buddy.c文件后，修改pmm.c中的init_pmm_manager，将默认使用的分配器修改为伙伴系统分配器就可以在ucore中实现伙伴系统了。

实验总结

与参考答案的对比

练习一

first-fit物理内存分配算法与回收内存空间算法参考答案都给出了基本完整的实现。主要是理解ucore的页式内存管理机制，了解ucore所实现的物理内存探测及内存空间管理初始化，物理内存管理框架等内容，熟悉相关数据结构和函数。回收内存空间的算法补充了空闲块对插入位置的查找，相较于答案缺少了对空闲链表的空闲块检验。

练习二

与参考答案基本一致，只有细节上的差别，先通过creat参数判断是否需要分配新的页存放二级页表，再判断是否分配成功，答案将这两步放在了一起。

练习三

与参考答案基本一致。代码中给出了详尽的步骤提示，因此实现与答案基本相同。关于问题二，理论上只要取消虚拟地址相对于物理地址的偏移，但仅仅取消偏移会产生其他问题。由于对于ucore建立映射的三个阶段（不同版本的指导书中既有说明为四个阶段，也有说明为三个阶段，根据目前的ucore代码，应该为三个阶段）还不够理解，这个问题其实没有完全的解决。

challenge 1

主要是理解和分析指导书给出的参考资料中的代码，学习伙伴系统的具体实现，没有自己实现ucore中的伙伴系统。

重要知识点

- 分页机制
- 空闲链表
- 内存分配算法（first-fit）
- 多级页表
- 虚拟地址的转换
- 页表项与页目录项的组成

lab2主要是关于物理内存管理机制的实现，是OS原理中内存空间管理的具体实现，细节上更加复杂，且ucore通过创建一个物理内存管理框架进行内存管理，在OS原理知识中未涉及到。通过lab2还学习了物理内存空间探测，段页机制的配合等知识，在物理内存空间管理方面比OS原理的知识更加完整深入。关于内存空间分配算法，lab2中只实现了首次匹配算法，没有涉及下次匹配，最优匹配等其他分配策略。

未涉及的内容

- 页交换策略
- 进程调度及切换等