

LAB2：物理内存管理

▪ PB18051183 白欣雨

LAB2：物理内存管理

实验目的

实验内容

实验过程

练习0:填写已有实验

练习1：实现 first-fit 连续物理内存分配算法

实验准备

函数实现

练习2：实现寻虚拟地址对应查找到第一个的页表项

实验准备

函数实现

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

Challenge1：Buddy System实现

感想与总结

参考资料

相关知识点

实验目的

- 理解基于段页式内存地址的转换机制
- 理解页表的建立和使用方法收到的
- 理解物理内存的管理方法

实验内容

- 了解如何发现系统中的物理内存。
- 了解如何建立对物理内存的初步管理，即了解连续物理内存管理。
- 了解页表相关的操作，即如何建立页表来实现虚拟内存到物理内存之间的映射，对段页式内存管理机制有一个比较全面的了解。

实验过程

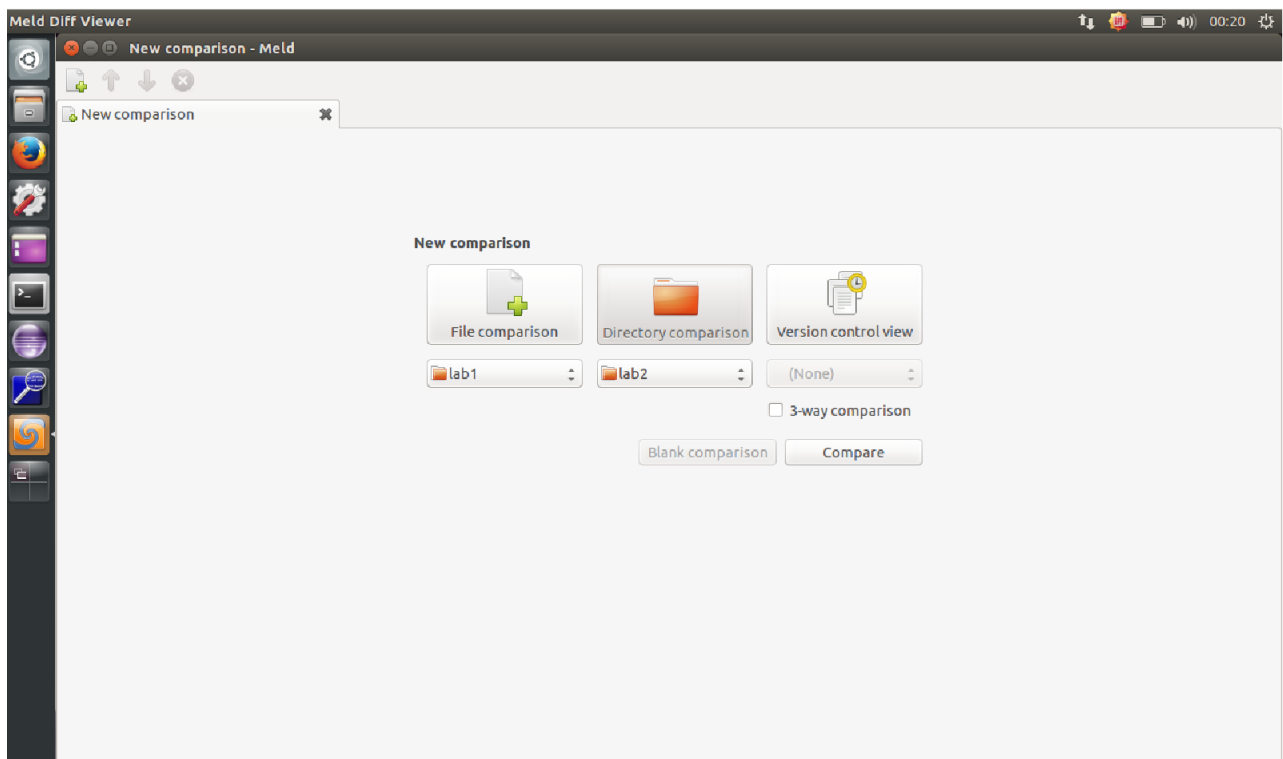
练习0: 填写已有实验

本实验依赖于实验1，可以直接将实验1的代码复制过来。根据实验指导书提供的方法，有diff和patch工具。本次实验采用meld，一种图形化的方式进行手动合并，具体过程为：

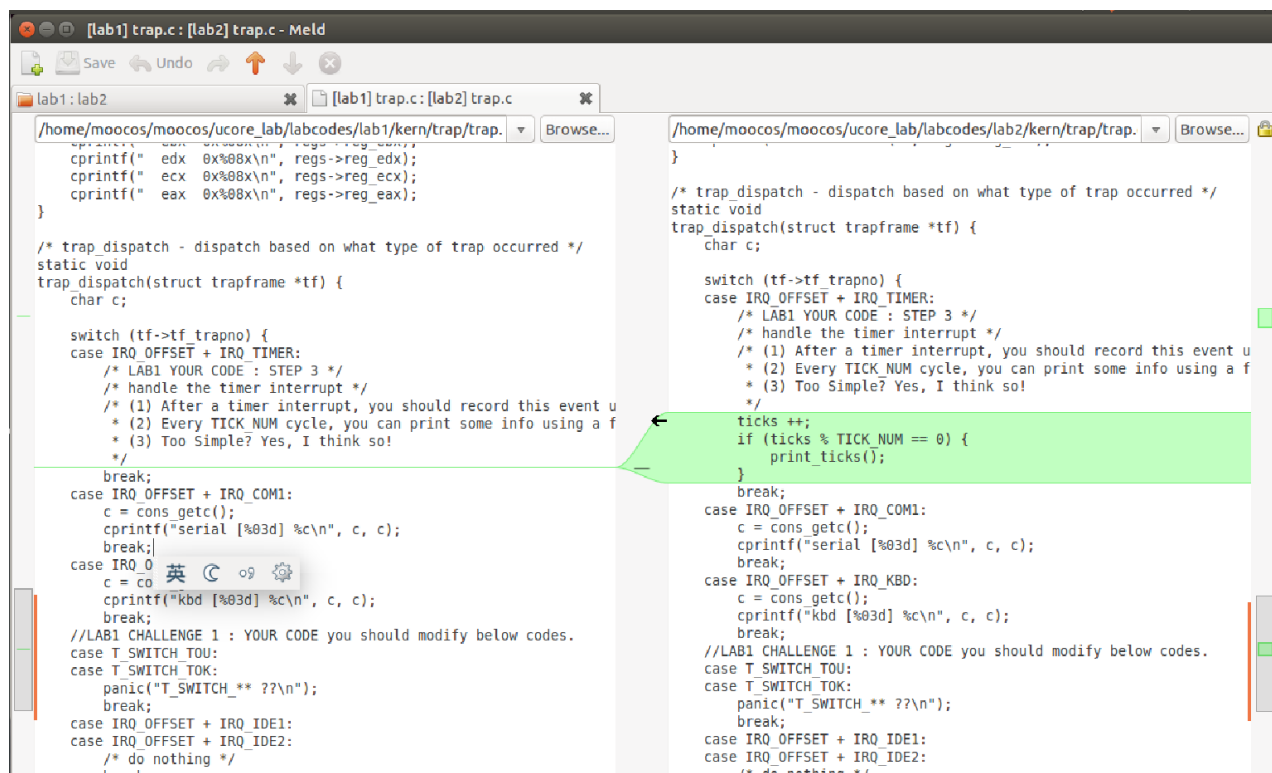
1. 在Terminal中调用meld：



2. 选择要比较的两个目录



3. 点击“compare”后，即可进行内部文件的对比，在本次实验中，需要填补代码的文件为：init.c、trap.c 和 kdebug.c，点击箭头即可进行添加或者对不同的地方进行替换。



```
/* trap_dispatch - dispatch based on what type of trap occurred */
static void
trap_dispatch(struct trapframe *tf) {
    char c;

    switch (tf->tf_trapno) {
    case IRQ_OFFSET + IRQ_TIMER:
        /* LAB1 YOUR CODE : STEP 3 */
        /* handle the timer interrupt */
        /* (1) After a timer interrupt, you should record this event u
        * (2) Every TICK_NUM cycle, you can print some info using a f
        * (3) Too Simple? Yes, I think so!
        */
        break;
    case IRQ_OFFSET + IRQ_COM1:
        c = cons_getc();
        cprintf("serial [%03d] %c\n", c, c);
        break;
    case IRQ_0:
        c = co
        cprintf("kbd [%03d] %c\n", c, c);
        break;
    //LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
    case T_SWITCH_TOK:
        panic("T_SWITCH_** ??\n");
        break;
    case IRQ_OFFSET + IRQ_IDE1:
    case IRQ_OFFSET + IRQ_IDE2:
        /* do nothing */
        break;
    }
}

/* trap_dispatch - dispatch based on what type of trap occurred */
static void
trap_dispatch(struct trapframe *tf) {
    char c;

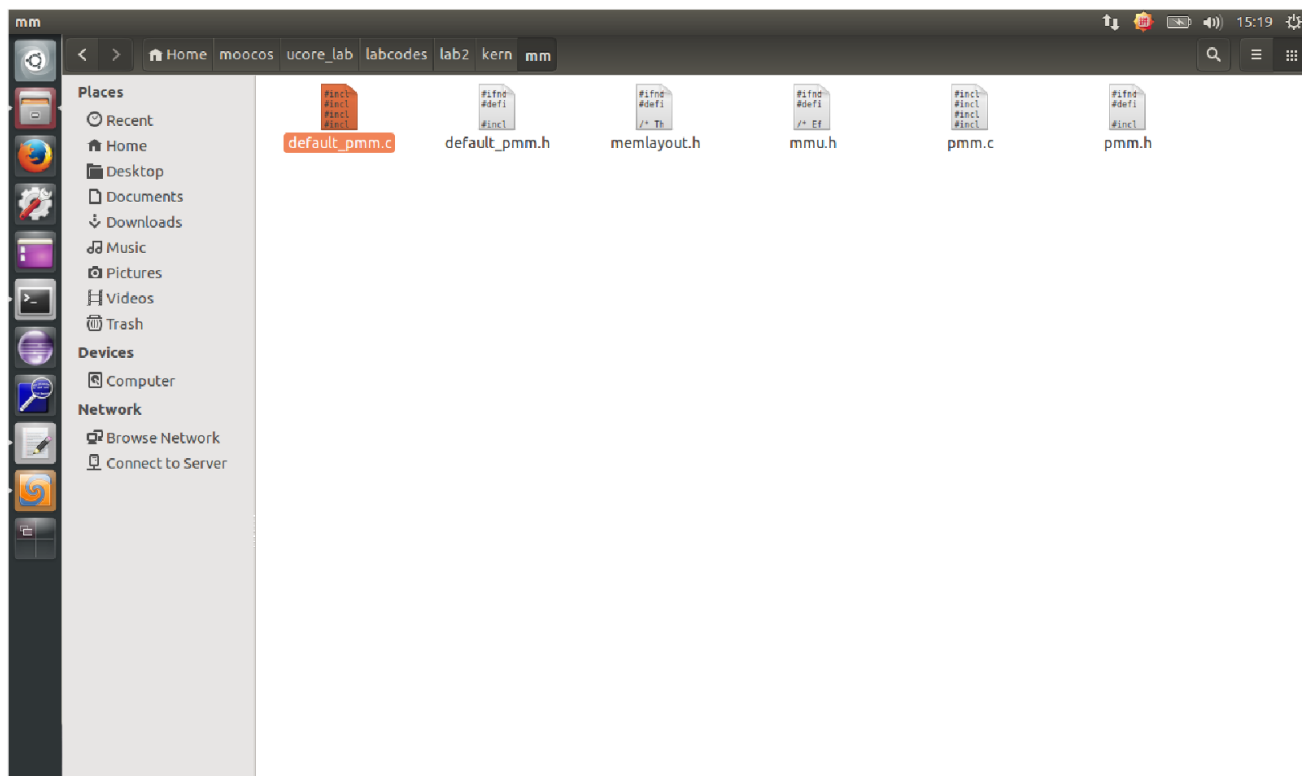
    switch (tf->tf_trapno) {
    case IRQ_OFFSET + IRQ_TIMER:
        /* LAB1 YOUR CODE : STEP 3 */
        /* handle the timer interrupt */
        /* (1) After a timer interrupt, you should record this event u
        * (2) Every TICK_NUM cycle, you can print some info using a f
        * (3) Too Simple? Yes, I think so!
        */
        ticks++;
        if (ticks % TICK_NUM == 0) {
            print_ticks();
        }
        break;
    case IRQ_OFFSET + IRQ_COM1:
        c = cons_getc();
        cprintf("serial [%03d] %c\n", c, c);
        break;
    case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        cprintf("kbd [%03d] %c\n", c, c);
        break;
    //LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
    case T_SWITCH_TOK:
        panic("T_SWITCH_** ??\n");
        break;
    case IRQ_OFFSET + IRQ_IDE1:
    case IRQ_OFFSET + IRQ_IDE2:
        /* do nothing */
        break;
    }
}
```

4. 填补结果可以直接在后续运行中验证。

练习1：实现first-fit 连续物理内存分配算法

在完成练习1之前，我们首先要了解如何进行分配内存空间的操作以及相关算法，通常情况下，分配给一个进程的地址空间必须是连续的。为了提高利用率，在分配物理内存时有着不同的算法，比如根据进程需要的内存空间，选择分配给与其大小最相近的空间（best fit），当然还有其他算法，比如最差匹配法（worst fit）、最先匹配法等（first fit）等。本次实验将实现first fit算法，并对其进行可能的相关的改进。

接下来，我们按照实验指导书的提示，找到default_pmm.c文件，仔细阅读其注释，进行练习1。



实验准备

在实验开始之前，我们先了解一下文件中的相关定义与数据结构。首先是Page，在文件中它指的是每一个页的属性。其位于kern/mm/memlayout.h中。

```
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
};
```

- ref表示这样页被页表的引用记数，这里是指映射此物理页的虚拟页个数。如果某页表中有一个页表项设置了虚拟页到这个Page管理的物理页的映射关系，就会把Page的ref加一。反之，若页表项取消，即映射关系解除，会把ref减一。
- flags表示此物理页的状态标记，有两个标志位，前一个（bit0）表示是否被保留，如果被保留了则设为1（比如内核代码占用的空间），此时该页不能放到空闲页链表中，不能动态分配与释放。后一个（bit1）表示此页是否是free的。如果设置为1，表示这页是free的，可以被分配；如果设置为0，表示这页已经被分配出去了，不能被再次分配。
- property用来记录某连续内存空闲块的大小（即地址连续的的空闲页个数），需要注意的是用到此成员变量的这个Page一定是连续内存块的开始地址（即头一页）。；连续内存空闲块利用这个页的成员变量property来记录此块内的空闲页个数。
- page_link是便于把多个连续内存空闲块链接在一起的双向链表指针，连续内存空闲块利用这个页的成员变量page_link来链接比它地址小和大的其他连续内存空闲块。

接下来定义一个free_area_t数据结构，这个数据结构可以管理随物理页分配与释放而产生的多个地址不连续的小连续内存空闲块。

```
typedef struct {
    list_entry_t free_list;           // the list header
    unsigned int nr_free;             // # of free pages in this free list
} free_area_t;
```

- free_list是一个list_entry结构的双向链表指针。
- nr_free记录当前空闲页的个数。

函数实现

根据实验指导书的介绍，本实验主要修改kern/mm/default_pmm.c文件，根据注释我们知道，我们需要修改default_init、default_init_memmap、default_alloc_pages、default_free_pages。我们依次进行查看与修改：

1. default_init

首先是初始化free_list并将nr_free设置为0的default_init：

```
static void
default_init(void) {
    list_init(&free_list);
    nr_free = 0;
}
```

没什么可以修改的内容，其作用就是初始化，可以参照其注释：

```
/*default_init: you can reuse the demo default_init fun to init the free_list and set
nr_free to 0.
*          free_list is used to record the free mem blocks. nr_free is the total number
for free mem blocks.
```

free_list用于记录空闲mem块，nr_free是空闲内存块的总数。

2. default_init_memmap

接下来是default_init_memmap，首先是未修改前的代码：

```
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));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    list_add(&free_list, &(base->page_link));
}
```

根据注释：

```

/*default_init_memmap: CALL GRAPH: kern_init --> pmm_init-->page_init-->init_memmap-->
pmm_manager->init_memmap
*           This fun is used to init a free block (with parameter: addr_base,
page_number).
*           First you should init each page (in memlayout.h) in this free block, include:
*           p->flags should be set bit PG_property (means this page is valid. In
pmm_init fun (in pmm.c),
*           the bit PG_reserved is setted in p->flags)
*           if this page is free and is not the first page of free block, p-
>property should be set to 0.
*           if this page is free and is the first page of free block, p->property
should be set to total num of block.
*           p->ref should be 0, because now p is free and no reference.
*           We can use p->page_link to link this page to free_list, (such as:
list_add_before(&free_list, &(p->page_link)); )

```

我们知道此函数用于初始化一个空闲块（带参数：addr_base、page_number），将块的每一个页都初始化，包括p->flags等，其中要注意的是：空闲页的第一页p->property应设置为总块数，其余页p->property设置为0。

因此根据注释，我们很容易得到修改后的代码：

```

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)); //判断本页是否为保留页
        //设置标志位
        p->flags = 0;
        p->property = 0;
        SetPageProperty(p);
        set_page_ref(p, 0); //清空引用
        list_add_before(&free_list, &(p->page_link)); //插入空闲页链表
    }
    nr_free = nr_free + n; //计算连续内存空闲块大小
    //第一个块的property特殊处理
    base->property = n; //连续内存空闲块的大小为n
}

```

此函数错误其实比较明显，只要理解了函数的作用和各个参数作用即可修改。我们在循环体部分进行了修改，之前代码的问题在于插入空闲页链表项的次数不对，应该放在循环内。与答案对比只是一些顺序不同。

3. default_alloc_pages

此函数源代码如下：

```

static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
}

```

```

}
if (page != NULL) {
    list_del(&(page->page_link));
    if (page->property > n) {
        struct Page *p = page + n;
        p->property = page->property - n;
        list_add(&free_list, &(p->page_link));
    }
    nr_free -= n;
    ClearPageProperty(page);
}
return page;
}

```

此函数的主要作用就是从空闲页块的链表中去遍历找到第一块大于n的块，把它从空闲页链表中一出移除，分配出去，然后如果有多余的，把分完剩下的部分再次加入会空闲页链表中即可。

由于first-fit算法维护的是一个有序空闲块，要求空闲表中的空闲空间按照地址从小到大排序。这样，我们在查询空闲空间时就可以从链表头部开始，找到第一个符合要求的空间即可，将这个空间从空闲表中删除。如果空闲空间在分配完要求数量的物理页之后有剩余，那么需要将剩余的部分作为新的空闲空间插入到原空间位置，以保证空闲表中空闲空间地址顺序。

此函数的设计思路以及修改思路就按照上述进行，得到修改后的代码：

```

static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);    //若全部空闲页空间加起来都小于需求，则直接返回NULL
    if (n > nr_free) {
        return NULL;
    }
    list_entry_t *le, *len;
    le = &free_list;

    //从空闲块链表的头指针开始，遍历空闲页链表，寻找合适的页块
    while((le=list_next(le)) != &free_list) {
        struct Page *p;
        p = le2page(le, page_link);    //将地址转换成页的结构
        if(p->property >= n){           //找到第一个大于等于n的空闲块
            int i = 0;                  //递归初始化每一个选中的空闲块链表中的页结构
            for(;i<n;i++){
                len = list_next(le);
                //让pp指向分配的那一页
                //le2page将le的地址根据链表元素找到对应的Page指针p
                struct Page *pp;
                pp = le2page(le, page_link);
                //设置每一页的标志位
                SetPageReserved(pp);
                ClearPageProperty(pp);
                list_del(le);
                le = len;
            }
            if(p->property>n){
                (le2page(le,page_link))->property = p->property - n;//如果选中的块空间大于n，只取其中的大小为n的块，剩余块保留。
            }
            ClearPageProperty(p);
            SetPageReserved(p);
            nr_free = nr_free - n;//当前空闲页的数目减n
            return p;
        }
    }
}

```

```
return NULL;    //没有大于等于n的连续空闲页块，返回NULL
}
```

第一次完成函数时报错，查看错误才知道这里for循环里不能直接使用int i = 0，要在外面声明。

4. default_free_pages

此函数的主要功能就是将需要释放的空间标记为空之后，需要找到空闲表中合适的位置将其加到free_list中。

由于空闲表中的记录都是按照物理页地址排序的，所以如果插入位置的前驱或者后继刚好和释放后的空间邻接，那么需要将新的空间与前后邻接的空间合并形成更大的空间。

函数源代码：

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    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;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
    nr_free += n;
    list_add(&free_list, &(base->page_link));
}
```

根据注释的讲解与提示，函数运行具体步骤如下：

1. 在空闲页链表中查找合适的位置，将被释放页块插入。
2. 修改被释放页块的标志位与头部计数器。
3. 在空闲页链表中地址合并。

容易得到修改后的代码：

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    assert(PageReserved(base));
    list_entry_t *le = &free_list;
    struct Page *p;
    while((le=list_next(le)) != &free_list) {    //查找待插入的位置le
```



```

    p = le2page(le, page_link);
    if(p>base){
        break;
    }
}
for(p=base;p<base+n;p++){ //向前插入n个页（空闲），并设置标志位
    list_add_before(le, &(p->page_link));
}
base->flags = 0;
set_page_ref(base, 0);
ClearPageProperty(base);
SetPageProperty(base);
base->property = n; //将页块信息记录在头部
//向高地址合并
p = le2page(le,page_link) ;
if( base+n == p ){
    base->property = base->property + p->property;
    p->property = 0;
}
//向低地址合并
le = list_prev(&(base->page_link));
p = le2page(le, page_link);
//若低地址已分配则不需要合并
if(le!=&free_list && p==base-1){
    while(le!=&free_list){
        if(p->property){
            p->property += base->property;
            base->property = 0;
            break;
        }
        le = list_prev(le);
        p = le2page(le,page_link);
    }
}

nr_free += n;
return ;
}

```

至此，所有需要修改的函数都已经修改完毕，总体上感觉有点复杂，首先要读懂函数的作用，又要理解每个参数的意义，对一些宏定义可能要去头文件中查找（使用grep -r命令）。参考了很多资料，并和答案对比完成了最终的函数定义。

算法改进

根据前述实验的分析，我们发现用此方法实现的first-fit算法的复杂度主要来自于查找合适的内存块。本实验完成的是用一个线性表，用顺序查找的方式来查找合适的块，这种方式最坏的情况要遍历整个表，时间复杂度为 $O(n)$ 。因此我们可以从查找效率上入手，可以考虑采用二叉查找的方式：用一个平衡二叉树来代替原来的链表，这样可以保证在 $O(\log n)$ 复杂度下查找到第一块合适的物理内存。虽然使运算更加复杂，但是较好的改进了时间复杂度。

练习2：实现寻虚拟地址对应查找到第一个的页表项

实验准备

根据实验指导书的要求，本练习主要是补全kern/mm/pmm.c中的get_pte函数，以实现其功能。该函数的功能是根据一个虚地址找到其所对应的二级页表项的内核虚地址，若该二级页表项不存在，则分配一个包含该项的二级页表。

我们先看看注释的提示：

```
/* LAB2 EXERCISE 2: YOUR CODE
 *
 * If you need to visit a physical address, please use KADDR()
 * please read pmm.h for useful macros
 *
 * Maybe you want help comment, BELOW comments can help you finish the code
 *
 * Some Useful MACROs and DEFINES, you can use them in below implementation.
 * MACROs or Functions:
 *   PDX(la) = the index of page directory entry of VIRTUAL ADDRESS la.
 *   KADDR(pa) : takes a physical address and returns the corresponding kernel virtual
address.
 *   set_page_ref(page,1) : means the page be referenced by one time
 *   page2pa(page): get the physical address of memory which this (struct Page *) page
manages
 *   struct Page * alloc_page() : allocation a page
 *   memset(void *s, char c, size_t n) : sets the first n bytes of the memory area pointed
by s
 *
 *                                     to the specified value c.
 * DEFINES:
 *   PTE_P           0x001           // page table/directory entry flags bit :
Present
 *   PTE_W           0x002           // page table/directory entry flags bit :
Writeable
 *   PTE_U           0x004           // page table/directory entry flags bit : User
can access
 */
```

函数实现

注释提示了我们可能用到的函数与定义以及参数，结合实验指导书，完成后的源代码，代码的具体设计思路见注释：

```
pte_t *
get_pte(pte_t *pgdir, uintptr_t la, bool create) {
//尝试获取页表
    pte_t *pdep = &pgdir[PDX(la)]; //找到 PDE这里的 pgdir 可以看做是页目录表的基址
    if (!(*pdep & PTE_P)) { //若获取不成功则申请一页
        struct Page *page = alloc_page();
        if(!creat || (page = all_page())==NULL){
            return NULL;
        }
        set_page_ref(page, 1); //引用次数需要加1
        uintptr_t pa = page2pa(page); //获取页的线性地址
        memset(KADDR(pa), 0, PGSIZE); //这一页清空，将线性地址转换为内核虚拟地址
        *pdep = pa | PTE_U | PTE_W | PTE_P; //设置PDE权限
    }
//返回页表地址
```

```
return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(1a)];
}
```

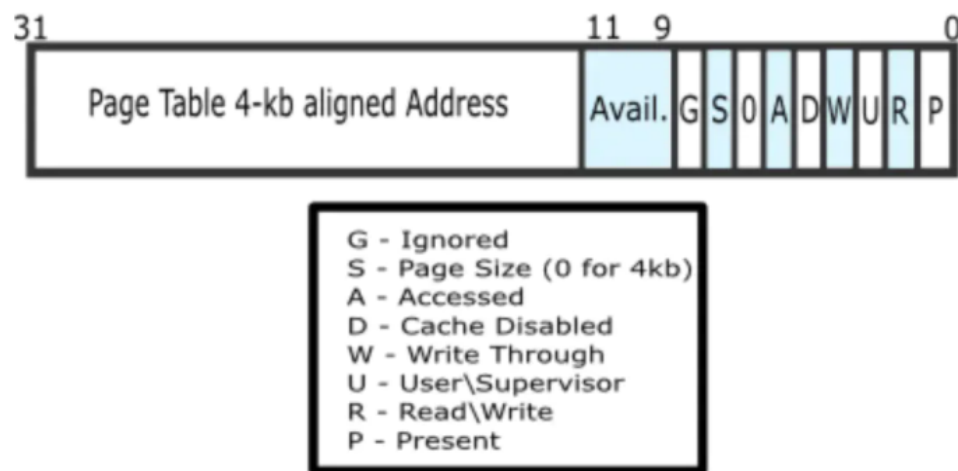
完成函数后，与答案对比发现自己缺少若分配失败怎么返回，代码的鲁棒性没有考虑，其余功能都顺利完成（主要是通过练习1了解了不少），还有一些宏定义没有搞清楚，在看了注释以及答案的方法最后将bug消除了。

- 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？

会进行换页操作。首先CPU将产生页访问异常的线性地址放到cr2寄存器中；然后在中断栈中依次压入EFLAGS, CS, EIP, 以及页访问异常码error code, 如果page fault是发生在用户态，则还需要先压入ss和esp, 并且切换到内核栈；设置错误代码error_code, 触发Page Fault异常，然后压入error_code 中断服务例程后，将外存的数据换到内存中来，最后退出中断，回到进入中断前的状态。

- 请描述目录项和页表项中每个组成部分的含义和以及对ucore而言的潜在用处

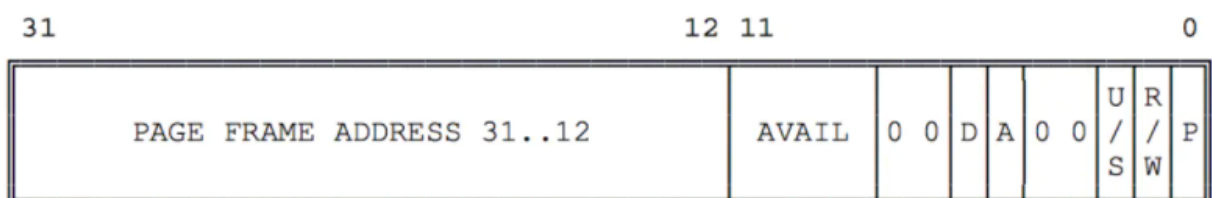
接下来描述页目录项的每个组成部分，PDE（页目录项）的具体组成如下图所示；描述每一个组成部分的含义如下：



每一个组成部分的含义如下：

- P(Present)位：表示改页保存在物理内存中；
- R(Read/Write)位：表示该页是否可以被读写；
- U(User)位：表示该页可以被何等级的权限用户访问；
- W(Write Through)位：表示CPU可以直接写回内存；
- D(Cache Disable)位：表示不需要被CPU缓存；
- A(Access)位：表示该页是否被写过；
- S(Size)位：表示一个页4KB；
- 接下来的第8位可忽略；
- 第9-11位未被CPU使用，可保留给OS使用；
- 前20位表示4K对齐的该PDE对应的页表起始位置(物理地址，该物理地址的高20位即PDE中的高20位，低12位为0)。

接下来描述PTE（页表项）中的每个组成部分的含义，具体组成如下图所示：



P - PRESENT
 R/W - READ/WRITE
 U/S - USER/SUPERVISOR
 D - DIRTY
 AVAIL - AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.

每个部分具体含义如下：

- 0-3位同PDE。
- C(Cache Disable)位：同PDED位。
- A(Access)位:同PDE。
- D(Dirty)位:表示该页是否被写过。
- G (Global)位：表示在CR3寄存器更新时无需刷新TLB中关于该页的地址。
- 9-11位保留给OS使用。
- 12-31位指明物理页基址。

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

具体实验之前，我们先看一下代码中给出的注释：

```

/* LAB2 EXERCISE 3: YOUR CODE
 *
 * Please check if ptep is valid, and tlb must be manually updated if mapping is updated
 *
 * Maybe you want help comment, BELOW comments can help you finish the code
 *
 * Some Useful MACROs and DEFINES, you can use them in below implementation.
 * MACROs or Functions:
 *   struct Page *page pte2page(*ptep): get the according page from the value of a ptep
 *   free_page : free a page
 *   page_ref_dec(page) : decrease page->ref. NOTICE: ff page->ref == 0 , then this page
should be free.
 *   tlb_invalidate(pde_t *pgdir, uintptr_t la) : Invalidate a TLB entry, but only if the
page tables being
 *
 *                                     edited are the ones currently in use by the processor.
 * DEFINES:
 *   PTE_P          0x001                // page table/directory entry flags bit :
Present
 */

```

因此在实现过程中要首先判断页被引用次数，如果只引用了一次，则直接释放掉这一页，否则删掉该页的入口。大致过程为：将物理页的引用数目减一，如果变为零，那么释放页面；然后将页目录项清零；刷新TLB。

因此，可以得到补全后的代码。具体实现过程注释：

```

static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    if (*ptep & PTE_P) {          //判断页表中该表项是否存在
        struct Page *page;        //声明一个页
        page = pte2page(*ptep);    //将页表项转换为页数据结构
        if (page_ref_dec(page) == 0) {
            free_page(page);        //若只被引用了一次，直接释放掉此页
        }
        else *ptep = 0;            //若被多次引用，释放二级页表的表项
        tlb_invalidate(pgdir, la);
    }
}

```

- 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?

有对应关系：

PDX: 页目录表索引

PTX:页表索引

PPN:线性地址的高20位，即PDX +PTxPA:物理地址 KA - KERNBASE

KA:内核虚地址PA +KERNBASE

page - pages = PPN

PPN<<12 = PA

&pages [PPN(pa)]= page

- 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？鼓励通过编程来具体完成这个问题。

bootloader把ucore放在了起始物理地址为0x100000的物理内存空间。要使得虚拟地址与物理地址相等:首先更改KERNBASE，从0xC000000 到0x00000000，在没有开启页式地址转换前，内核逻辑地址经过一次段地址转换即直接得到物理地址:PA（物理地址）= LA（线性地址）=VA（逻辑地址）- KERNBASE;其次，虚拟地址由0xC0100000改成0x00100000。

至此Lab2的主要内容均已经完成，下面是程序运行结果：

```

memory management: default_pmm_manager
e820map:
memory: 0009fc00, [00000000, 0009fbff], type = 1.
memory: 00000400, [0009fc00, 0009ffff], type = 2.
memory: 00010000, [000f0000, 000fffff], type = 2.
memory: 07efe000, [00100000, 07ffdfff], type = 1.
memory: 00002000, [07ffe000, 07ffffff], type = 2.
memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
----- BEGIN -----
PDE(0e0) c0000000-f8000000 38000000 urw
|-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
|-- PTE(000e0) faf00000-fafe0000 000e0000 urw
|-- PTE(00001) fafeb000-fafec000 00001000 -rw
----- END -----
++ setup timer interrupts
100 ticks

```

Challenge1: Buddy System实现

此部分为拓展实验部分，主要实现伙伴系统（Buddy System）分配算法。

通过查阅相关资料得知，与first-fit不同的是Buddy System使用一棵完全二叉树来储存或者标记一个内存块，对于一颗有着 n 片叶子的完全二叉树来说，所有节点的总数为 $2n-1 \approx 2n$ 。如果Buddy System的可分配空间为 n 页的话，那么就需要额外保存 $2n-1$ 个节点信息。

具体分配过程，查阅相关资料：

Buddy System 要求分配空间为2的幂，所以首先将请求的页数向上对齐到2的幂。接下来从二叉树的根节点(1号节点)开始查找满足要求的节点。对于每次检查的节点：

- 如果子树的最大可用空间小于请求空间，那么分配失败;
- 如果子树的最大可用空间大于等于请求空间，并且总空间大小等于请求空间，说明这个节点对应的空间没有被分割和分配，并且满足请求空间大小，那么分配这个空间;
- 如果子树的最大可用空间大于等于请求空间，并且总空间大小大于请求空间，那么在这个节点的子树中查找:
 - 如果这个节点对应的空间没有被分割过（最大可用空间等于总空间大小），那么分割空间，在左子树（左半部分）继续查找;
 - 如果左子树包含大小等于请求空间的可用空间，那么在左子树中继续查找;
 - 如果右子树包含大小等于请求空间的可用空间，那么在右子树中继续查找;
- 如果左子树的最大可用空间大于等于请求空间，那么在左子树中继续查找;
- 如果右子树的最大可用空间大于等于请求空间，那么在右子树中继续查找。

实现此部分，需要和first-fit一样，建立一个头文件，一个包含主程序的C代码文件，然后查找pmm.c文件，将引用改为自己写的buddy system：

```
pmm_manager = &buddy_pmm_manager;
```

对于头文件，我直接仿照了default_pmm.h，模仿实现了buddy_pmm.h：

```
#ifndef __KERN_MM_BUDDY_PMM_H__
#define __KERN_MM_BUDDY_PMM_H__
#include <pmm.h>

extern const struct pmm_manager buddy_pmm_manager;

#endif /* ! __KERN_MM_BUDDY_PMM_H__ */
```

然后是主程序buddy_pmm.c:

```
#include <pmm.h>
#include <list.h>
#include <string.h>
#include <buddy_pmm.h>
#define BUDDY_MAX_DEPTH 30
static unsigned int* buddy_page;
static unsigned int buddy_page_num; // store buddy system
static unsigned int max_pages; // maintained by buddy
static struct Page* buddy_allocatable_base;
#define max(a, b) ((a) > (b) ? (a) : (b))
static void buddy_init(void) {}
static void buddy_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    // calc buddy alloc page number
```

```

max_pages = 1;
for (int i = 1; i < BUDDY_MAX_DEPTH; ++i, max_pages <= 1)
if (max_pages + (max_pages >> 9) >= n)
break;
max_pages >>= 1;
buddy_page_num = (max_pages >> 9) + 1;
cprintf("buddy init: total %d, use %d, free %d\n", n, buddy_page_num, max_pages);
// set these pages to reserved
for (int i = 0; i < buddy_page_num; ++i)
SetPageReserved(base + i);
// set non-buddy page to be allocatable
buddy_allocatable_base = base + buddy_page_num;
for (struct Page *p = buddy_allocatable_base; p != base + n; ++p) {
ClearPageReserved(p);
SetPageProperty(p);
set_page_ref(p, 0);
}
// init buddy page
buddy_page = (unsigned int*)KADDR(page2pa(base));
for (int i = max_pages; i < max_pages << 1; ++i)
buddy_page[i] = 1;
for (int i = max_pages - 1; i > 0; --i)
buddy_page[i] = buddy_page[i << 1] << 1;
}

static struct Page* buddy_alloc_pages(size_t n) {
assert(n > 0);
if (n > buddy_page[1]) return NULL;
unsigned int index = 1, size = max_pages;
for (; size >= n; size >>= 1) {
if (buddy_page[index << 1] >= n) index <= 1;
else if (buddy_page[index << 1 | 1] >= n) index = index << 1 | 1;
else break;
}
buddy_page[index] = 0;
// allocate all pages under node[index]
struct Page* new_page = buddy_allocatable_base + index * size - max_pages;
for (struct Page* p = new_page; p != new_page + size; ++p)
set_page_ref(p, 0), ClearPageProperty(p);
for (; (index >>= 1) > 0; ) // since destory continuous, use MAX instead of SUM
buddy_page[index] = max(buddy_page[index << 1], buddy_page[index << 1 | 1]);
return new_page;
}

static void buddy_free_pages(struct Page *base, size_t n) {
assert(n > 0);
unsigned int index = (unsigned int)(base - buddy_allocatable_base) + max_pages, size = 1;
// find first buddy node which has buddy_page[index] == 0
for (; buddy_page[index] > 0; index >>= 1, size <= 1);
// free all pages
for (struct Page *p = base; p != base + n; ++p) {
assert(!PageReserved(p) && !PageProperty(p));
SetPageProperty(p), set_page_ref(p, 0);
}
// modify buddy_page
for (buddy_page[index] = size; size <= 1, (index >>= 1) > 0;)
buddy_page[index] = (buddy_page[index << 1] + buddy_page[index << 1 | 1] == size) ? size :
max(buddy_page[index << 1], buddy_page[index << 1 | 1]);
}

static size_t buddy_nr_free_pages(void) { return buddy_page[1]; }
static void buddy_check(void) {
int all_pages = nr_free_pages();
struct Page* p0, *p1, *p2, *p3;

```

```

assert(alloc_pages(all_pages + 1) == NULL);
p0 = alloc_pages(1);
assert(p0 != NULL);
p1 = alloc_pages(2);
assert(p1 == p0 + 2);
assert(!PageReserved(p0) && !PageProperty(p0));
assert(!PageReserved(p1) && !PageProperty(p1));
p2 = alloc_pages(1);
assert(p2 == p0 + 1);
p3 = alloc_pages(2);
assert(p3 == p0 + 4);
assert(!PageProperty(p3) && !PageProperty(p3 + 1) && PageProperty(p3 + 2));
free_pages(p1, 2);
assert(PageProperty(p1) && PageProperty(p1 + 1));
assert(p1->ref == 0);
free_pages(p0, 1);
free_pages(p2, 1);
p2 = alloc_pages(2);
assert(p2 == p0);
free_pages(p2, 2);
assert((*p2 + 1).ref == 0);
assert(nr_free_pages() == all_pages >> 1);
free_pages(p3, 2);
p1 = alloc_pages(129);
free_pages(p1, 256);
}

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,
};

```

实验结果:

通过make qemu命令, 我们查看一下运行结果:

```

(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc0106978 (phys)
  edata  0xc0119a36 (phys)
  end    0xc011a988 (phys)
Kernel executable memory footprint: 107KB
ebp:0xc0118f38 eip:0xc01009d0 args:0x00010094 0x00000000 0xc0118f68 0xc01000bc
  kern/debug/kdebug.c:308: print_stackframe+21
ebp:0xc0118f48 eip:0xc0100cbf args:0x00000000 0x00000000 0x00000000 0xc0118fb8
  kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc0118f68 eip:0xc01000bc args:0x00000000 0xc0118f90 0xffff0000 0xc0118f94
  kern/init/init.c:47: grade_backtrace2+33
ebp:0xc0118f88 eip:0xc01000e5 args:0x00000000 0xffff0000 0xc0118fb4 0x0000002a
  kern/init/init.c:52: grade_backtrace1+38
ebp:0xc0118fa8 eip:0xc0100103 args:0x00000000 0xc010002a 0xffff0000 0x0000001d
  kern/init/init.c:57: grade_backtrace0+23
ebp:0xc0118fc8 eip:0xc0100128 args:0xc010699c 0xc0106980 0x00000f52 0x00000000
  kern/init/init.c:62: grade_backtrace+34
ebp:0xc0118ff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00

```



```
kern/init/init.c:27: kern_init+84
memory management: buddy_pmm_manager
e820map:
memory: 0009fc00, [00000000, 0009fbff], type = 1.
memory: 00000400, [0009fc00, 0009ffff], type = 2.
memory: 00010000, [000f0000, 000fffff], type = 2.
memory: 07efe000, [00100000, 07ffdfdf], type = 1.
memory: 00002000, [07ffe000, 07ffffff], type = 2.
memory: 00040000, [fffc0000, ffffffff], type = 2.
buddy init: total 32323, use 33, free 16384
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
----- BEGIN -----
PDE(0e0) c0000000-f8000000 38000000 urw
|-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
|-- PTE(000e0) faf00000-fafe0000 000e0000 urw
|-- PTE(00001) fafeb000-fafec000 00001000 -rw
----- END -----
++ setup timer interrupts
100 ticks
End of Test.
kernel panic at kern/trap/trap.c:18:
EOT: kernel seems ok.
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
```

buddy init 显示了内存分配情况，可分配的物理页面数一共有 32291 个，由于 buddy system 所能利用的内存大小必须为2的整数次幂，因此实际能分配的页面数为 16384 个，为了管理分配的页面还需额外的 33 个页面分配给 buddy system 供其使用。

感想与总结

第一次做操作系统的实验，也第一次接触这么大的程序项目，前前后后查看了很多资料，使用grep命令查找各种宏定义，了解物理内存分配的细节，感觉做起来还是有困难，自己写的代码也出现了bug，和答案对比发现有很大的提升空间。当然通过这第一次实验，我熟悉了Linux的操作环境，能熟练掌握一些操作命令；除此以外，我对frist-fit、页表等知识点也有了更深层次的理解。我还尝试通过GitHub等渠道，查阅资料，实现了challenge1，总之收获颇丰。

参考资料

- 1.<https://coolshell.cn/articles/10427.html>
- 2.<https://github.com/Trinkle23897/os2019/tree/master>
- 3.<https://blog.csdn.net/rikeyone/article/details/85054231>

相关知识点

本实验中相关的知识点

- 首次匹配
- 分页
- 二级页表

本实验未涉及

- 碎片整理