

# OS\_Lab2\_Experimental report

---

湖南大学信息科学与工程学院

计科 210X 甘晴void (学号 202108010XXX)

## 前言

实验一过后大家做出来了一个可以启动的系统，实验二主要涉及操作系统的物理内存管理。操作系统为了使用内存，还需高效地管理内存资源。在实验二中大家会了解并且自己动手完成一个简单的物理内存管理系统。

## 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
- 理解物理内存的管理方法

## 实验内容

本次实验包含三个部分。

- 首先了解如何发现系统中的物理内存；
- 然后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；
- 最后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。

本实验里面实现的内存管理还是非常基本的，并没有涉及到对实际机器的优化，比如针对cache的优化等。如果大家有余力，尝试完成扩展练习。

## 练习要求

为了实现lab2的目标，lab2提供了3个基本练习和2个扩展练习，要求完成实验报告。

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

## 实验执行流程概述

本次实验主要完成ucore内核对物理内存的管理工作。参考ucore总控函数kern\_init的代码，可以清楚地看到在调用完成物理内存初始化的pmm\_init函数之前和之后，是已有lab1实验的工作。

lab2相对于lab1有两个方面的扩展。

- 首先，bootloader的工作有增加，在bootloader中，完成了对物理内存资源的探测工作（可进一步参阅附录A和附录B），让ucore kernel在后续执行中能够基于bootloader探测出的物理内存情况进行物理内存管理初始化工作。
- 其次，bootloader不像lab1那样，直接调用kern\_init函数，而是先调用位于lab2/kern/init/entry.S中的kern\_entry函数。kern\_entry函数的主要任务是为执行kern\_init建立一个良好的C语言运行环境（设置堆栈），而且临时建立了一个段映射关系，为之后建立分页机制的过程做一个准备。完成这些工作后，才调用kern\_init函数。

kern\_init函数在完成一些输出并对lab1实验结果的检查后，将进入物理内存管理初始化的工作，即调用pmm\_init函数完成物理内存的管理，这也是我们lab2的内容。接着是执行中断和异常相关的初始化工作，即调用pic\_init函数和idt\_init函数等，这些工作与lab1的中断异常初始化工作的内容是相同的。

为了完成物理内存管理，这里首先需要探测可用的物理内存资源：了解到物理内存位于什么地方，有多大之后，就以固定页面大小来划分整个物理内存空间，并准备以此为最小内存分配单位来管理整个物理内存，管理在内核运行过程中每页内存，设定其可用状态（free的，used的，还是reserved的），这其实就对应了我们在课本上讲到的连续内存分配概念和原理的具体实现；接着ucore kernel就要建立页表，启动分页机制，让CPU的MMU把预先建立好的页表中的页表项读入到TLB中，根据页表项描述的虚拟页（Page）与物理页帧（Page Frame）的对应关系完成CPU对内存的读、写和执行操作。这一部分其实就对应了我们在课本上讲到内存映射、页表、多级页表等概念和原理的具体实现。

内存管理相关的总体控制函数是pmm\_init函数，它完成的主要工作包括：

1. 初始化物理内存页管理器框架pmm\_manager;
2. 建立空闲的page链表，这样就可以分配以页（4KB）为单位的空闲内存了；
3. 检查物理内存页分配算法；
4. 为确保切换到分页机制后，代码能够正常执行，先建立一个临时二级页表；
5. 建立一一映射关系的二级页表；
6. 使能分页机制；
7. 从新设置全局段描述符表；
8. 取消临时二级页表；
9. 检查页表建立是否正确；
10. 通过自映射机制完成页表的打印输出（这部分是扩展知识）

另外，主要注意的相关代码内容包括：

- boot/bootasm.S中探测内存部分（从probe\_memory到finish\_probe的代码）；
- 管理每个物理页的Page数据结构（在mm/memlayout.h中），这个数据结构也是实现连续物理内存分配算法的关键数据结构，可通过此数据结构来完成空闲块的链接和信息存储，而基于这个数据结构的管理物理页数组起始地址就是全局变量pages，具体初始化此数组的函数位于page\_init函数中；
- 用于实现连续物理内存分配算法的物理内存页管理器框架pmm\_manager，这个数据结构定义了实现内存分配算法的关键函数指针，而我们需要完成这些函数的具体实现；
- 设定二级页表和建立页表项以完成虚实地址映射关系，这与硬件相关，且用到不少内联函数，源代码相对难懂一些。具体完成页表和页表项建立的重要函数是boot\_map\_segment函数，而get\_pte函数是完成虚实映射关键的关键。

## 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。

提示：可采用diff和patch工具进行半自动的合并（merge），也可用一些图形化的比较/merge工具来手动合并，比如meld，eclipse中的diff/merge工具，understand中的diff/merge工具等。

实际上，Lab1中改动的代码部分有限，因此手动对以下三个文件进行复制：

- kern/debug/kdebug.c
- kern/init/init.c
- kern/trap/trap.c

## 练习1：实现 **first-fit** 连续物理内存分配算法（需要编程）

在实现first fit 内存分配算法的回收函数时，要考虑地址连续的空闲块之间的合并操作。

提示:在建立空闲页块链表时，需要按照空闲页块起始地址来排序，形成一个有序的链表。可能会修改default\_pmm.c中的default\_init, default\_init\_memmap, default\_alloc\_pages, default\_free\_pages等相关函数。请仔细查看和理解default\_pmm.c中的注释。

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

阅读default\_pmm.c的注释部分

您应该重写函数： *default\_init, default\_init\_memmap, default\_alloc\_pages, default\_free\_pages*

所以我们需要重点关注这几个函数。但在这之前，需要先了解分配内存之前都需要干些什么（怎么初始化）

### 1.探测系统物理内存布局

当 ucore 被启动之后，最重要的事情就是知道还有多少内存可用，一般来说，获取内存大小的方法有以下两种

- BIOS 中断调用（只在实模式下完成）
- 直接探测（只在保护模式下完成）

通过 BIOS 中断获取内存布局有三种方式，都是基于INT 15h中断，分别为88h、e801h、e820h。但是并非在所有情况下这三种方式都能工作。在 Linux kernel 里，采用的方法是依次尝试这三种方法。而在本实验中，我们通过e820h中断获取内存信息。

因为e820h中断必须在实模式下使用，所以我们在 bootloader 进入保护模式之前调用这个 BIOS 中断，并且把 e820 映射结构保存在物理地址0x8000处。具体实现详见 boot/bootasm.S，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

```

总结来说，在实模式下，我们就通过e820H来探测了物理内存有多少可用。

## 2.以页为单位管理物理内存

在获得可用物理内存范围后，系统需要建立相应的数据结构来管理以物理页（按4KB对齐，且大小为4KB的物理内存单元）为最小单位的整个物理内存，以配合后续涉及的分页管理机制。每个物理页可以用一个 **Page** 数据结构来表示。由于一个物理页需要占用一个 **Page** 结构的 **空间**，**Page** 结构在设计时须尽可能小，以减少对内存的占用。**Page** 的定义在 `kern/mm/memlayout.h` 中。以页为单位的物理内存分配管理的实现在 `kern/default_pmm`

如下为物理页的 **Page** 结构（重要）

```

//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
};
//状态标记（在kern/mm/memlayout.h中）
#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

```

解读上述成员变量：

- **ref**: 如果这个页被页表引用了，即在某页表中有一个页表项设置了一个虚拟页到这个Page管理的物理页的映射关系，就会把Page的ref加1；反之，若页表项取消，即映射关系解除，就会把Page的ref减1。
- **flags**: 此物理页的状态标记（若为0则表示被保留，不能被分配与释放；若为1，表示free，可以被分配）
- **property**: 记录某连续内存空闲块的大小（即地址连续的空闲页的个数）
- **page\_link**: 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

在初始情况下，这个物理内存的空闲物理页都是连续的，这样就形成了一个大的连续内存空闲块。但随着物理页的分配与释放，这个大的连续内存空闲块会分裂为一系列地址不连续的多个小连续内存空闲块，且每个连续内存空闲块内部的物理页是连续的。那么为了有效地管理这些小连续内存空闲块。所有的连续内存空闲块可用一个双向链表管理起来，便于分配和释放，为此定义了一个**free\_area\_t**数据结构，具体实现如下。

```

typedef struct {
    list_entry_t free_list; // 链表:连接空闲物理页
    unsigned int nr_free;   // 空闲页个数
} free_area_t;

```

有了这两个数据结构，ucore就可以管理起来整个以页为单位的物理内存空间。接下来需要解决两个问题：

- 管理页级物理内存空间所需的Page结构的内存空间从哪里开始，占多大空间？
- 空闲内存空间的起始地址在哪里？

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 $npage = maxpa / PGSIZE$ ,

并计算出管理页的Page结构需要的空间 $sizeof(struct Page) * npage$ 。

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

其实实验二在内存分配和释放方面最主要的作用是建立了一个物理内存页管理器框架，这实际上是一个函数指针列表，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（初始化page）
    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
};

```

init\_memmap用于对页内存管理的Page的初始化，init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: 将page进行打包, 放入free_list内
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。本练习中使用默认的pmm\_manager进行管理, 其中的default\_init可以直接使用, 只需要修改default\_init\_memmap, default\_alloc\_pages, default\_free\_pages这三个就可以实现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。

★宏定义le2page可以实现通过链表节点获取节点数据。(这个在后面会反复用到)

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域。

```
#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。如下。

```

//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;
}
//memlayout.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))

```

## default\_init函数实现

按照注释解释，该部分代码无需修改，可直接使用

```

//default_init:
//初始化空闲链表
static void
default_init(void) {
    list_init(&free_list);
    nr_free = 0;
}

```

`free_list`是全局变量`free_area`的`list_entry_t`型成员变量。使用`list_init`函数后，它的前驱和后继都将指向自己，形成闭环；同时，`nr_free`是全局变量`free_area`的整型成员变量，表示空闲列表里所有空闲块可用的页数目，初始化为0。于是，该函数构建了一个只有头节点、没有空闲块的初始空闲列表。

## default\_init\_memmap函数实现

根据注释要求，此函数与原来相比只需要修改最后一行，使用list\_add\_before即可。

```
After that, We can use p->page_link to link this page into free_list.(e.g.:  
list_add_before(&free_list, &(p->page_link));)
```

实现如下：

```
//default_init_memmap:  
//初始化内存中的各个Page变量（这些变量连续存放在内存中，起始地址是参数base，总数是n）  
static void  
default_init_memmap(struct Page *base, size_t n) {  
    //n为页数，base为起始地址  
    assert(n > 0); //断言：必须满足n>0，否则警告  
    struct Page *p = base;  
    for (; p != base + n; p++) {  
        //base+n表示第n+1个页的指针（指针加法），这里使用循环遍历所有page  
        assert(PageReserved(p)); //断言：必须满足若页面被设置为可被分配，否则警告  
        p->flags = p->property = 0; //设置为被保留且不可分配（因为已经归base统一管理）  
        set_page_ref(p, 0); //设置为未被引用  
    }  
    //开始构建空闲列表  
    base->property = n; //初始所有物理页都空闲，拥有n的巨大空闲块  
    SetPageProperty(base); //将property位设置为1，表示这个空闲块可被分配  
    nr_free += n; //初始空闲块有完整的n个页大小  
    list_add_before(&free_list, &(base->page_link)); //将base指向的Page的list_entry_t成员page_link加入双向链表，从而将这个Page作为一个结点加入空闲列表  
}  
//完成此函数后，空闲列表里有两个结点，一个是头节点，另一个是base，它是连续n个Page里的第一个Page，表示初始空闲块有n个页大，而且是可分配的。
```

## default\_alloc\_pages函数实现

```
//函数作用：负责分配n个页，返回值是一个Page类型的指针，该指针描述了那个被真实分配的一个空闲区域的首个物理页。  
static struct Page *  
default_alloc_pages(size_t n) {  
    assert(n > 0);
```

```

if (n > nr_free) {
return NULL;
} //要分配的页数比总空闲页还多，直接返回空，分配失败（供不应求）
struct Page *page = NULL;
list_entry_t *le = &free_list;
while ((le = list_next(le)) != &free_list) {
    //不停用le指向后继结点的list_entry_t型成员，可以达到遍历的目的，但要注意范围，避免死循环
    struct Page *p = le2page(le, page_link);
    //使用le2page通过链表节点获取节点数据
    if (p->property >= n) {
        //property值说明了此空闲块里含有的页数；循环不断进行，直到发现一个Page的
        //property≥n，说明这个Page象征的空闲块足够大，可供分配
        page = p;
        break;
    }
}
if (page != NULL) {
nr_free -= n; //找到了合适的空闲块，成功分配
ClearPageProperty(page); //可分配，所以可用的空闲页必须减少，同时清除page的有效位
//处理该块中剩余的page
if (page->property > n) { //当前空闲块被分割n个页后还有剩余的可作为新空闲块
    struct Page *newnode=NULL; //需要新的空闲块
    newnode=page + n; //新的空闲块的指针指向page后的第n个页
    newnode->property = page->property - n; //设置新空闲块的空闲页数目
    SetPageProperty(newnode); //同时设置它是可分配的
    list_add_after(&(page->page_link), &(newnode->page_link));
    //接在旧的节点后面，然后删除旧的节点的时候，新的自然就取代了它
    list_del(&(page->page_link));
}
else
{ //否则，直接删除就行了
list_del(&(page->page_link));
}
}
return page;
}

```

//注意：若干Page变量本身是顺序存放的，对应着有序的若干物理页；只有空闲块中的第一个物理页对应的Page会被放入空闲列表并用指针将其le2page成员和空闲列表其他结点的le2page成员连接起来。虽然说“放入了空闲列表”，但各个Page的具体位置并不变化，只是其le2page成员的指针指向发生了变化而已。

## default\_free\_pages函数实现

```
//函数作用：释放n个页
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0); //断言
    struct Page *p = base;
    for (; p != base + n; p++) {
        assert(!PageReserved(p) && !PageProperty(p));
        //Property只有某个连续空闲块的顶部的页才有，相应的flags里页才会有这个位存在。如果PageReserved成立，说明是保留页，不能释放；如果是PageProperty成立，说明是某个空闲块的第一个页，不能释放。
        //总的来说，如果这个页是已经保留的，或者是有property的，就终止。
        p->flags = 0;
        set_page_ref(p, 0);
    } //为了释放，p不断向下移动，把沿途的页的flags全部变为0，设置其ref引用也为0
    //总之，for循环把需要释放的p个页的flags和ref全部都设置为0
    base->property = n;
    SetPageProperty(base);
    //把n个页的属性重新设置之后，它们变成一个还没有加入空闲表的空闲块，且第一个页的地址就是参数base。它的property在分配前后都是大于等于n的，现在必须设为n，表示它是一个n个页的空闲块。同时，它的property位被再次设为有效。
    list_entry_t *le = list_next(&free_list);

    //接下来遍历已有空闲列表，试图发现被释放的head有没有可能与已有的空闲块相连
    while (le != &free_list) { //while循环会开始遍历当前的双向链表。这个链表挤满了可用空闲块的首页（property不为0）
        p = le2page(le, page_link); //用le2page取出双向链表中list_entry_t类所在的Page结构指针
        le = list_next(le);
        //此时，p所指向的Page是当前双向链表中的某个空闲块的第一个页。base指向要释放的空闲块的第一个页，base->property是要释放的页的个数
        if (base + base->property == p) { //表示相连且base在p前面
            base->property += p->property; //合并延长
            ClearPageProperty(p); //合并后，p不是单独的且不是第一个，被取消掌管property的权力
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) { //表示相连且p在base前面
            p->property += base->property; //合并延长
            ClearPageProperty(base); //合并后，base不是单独的且不是第一个，被取消掌管property的权力
        }
    }
}
```

```

base = p;
list_del(&(p->page_link));
}
}
nr_free += n; //可用页增加
//将base指向的页的list_entry_t类放入双向链表。
//为。用while找出第一个地址比base自己大的页p; p作为空闲块的第一个页，地址大于
base，那么就意味着base对应的页应该放到它的前面。这样一来，双向链表中各个空闲块的
排序就是按照地址从小到大的，符合first_fit算法的要求。
le = &free_list;
while ((le=list_next(le)) != &free_list)
{
p = le2page(le, page_link);
if(base<p)
{ break; }
}
list_add_before(le, &(base->page_link));
}

```

## 改进空间

- 时间复杂度改进：用线段树可以实现 **alloc** 和 **free** 达到  $O(\log n)$  级别的时间复杂度，不过空间（复杂度虽然还是  $O(n)$  不变）要增加一倍。
- 空间改进：在空闲链表开头会产生许多小的空闲块，需要处理。
- 遍历顺序：可以通过适当提前较大的空闲块达到更好的成功命中效果。

## 5.其他实现算法概览

### 1、First Fit(最先匹配算法)

该算法从空闲分区链首开始查找，直至找到一个能满足其大小要求的空闲分区为止。然后再按照作业的大小，从该分区中划出一块内存分配给请求者，余下的空闲分区仍留在空闲分区链中。

- 优点：该算法倾向于使用内存中低地址部分的空闲区，在高地址部分的空闲区很少被利用，从而保留了高地址部分的大空闲区。显然为以后到达的大作业分配大的内存空间创造了条件。
- 缺点：低地址部分不断被划分，留下许多难以利用、很小的空闲区，而每次查找又都从低地址部分开始，会增加查找的开销。



## 2、Next Fit(循环首次匹配算法)

该算法是由首次适应算法演变而成的。在为进程分配内存空间时，不再每次从链首开始查找，直至找到一个能满足要求的空闲分区，并从中划出一块来分给作业。

- 优点：使内存中的空闲分区分布的更为均匀，减少了查找时的系统开销。
- 缺点：缺乏大的空闲分区，从而导致不能装入大型作业。

## 3、Best Fit(最佳匹配算法)

该算法总是把既能满足要求，又是最小的空闲分区分配给作业。为了加速查找，该算法要求将所有的空闲区按其大小排序后，以递增顺序形成一个空白链。这样每次找到的第一个满足要求的空闲区，必然是最优的。孤立地看，该算法似乎是最优的，但事实上并不一定。因为每次分配后剩余的空间一定是最小的，在存储器中将留下许多难以利用的小空闲区。同时每次分配后必须重新排序，这也带来了一定的开销。

- 优点：每次分配给文件的都是最合适该文件大小的分区。
- 缺点：内存中留下许多难以利用的小的空闲区。

## 4、Worst Fit(最差匹配算法)

该算法按大小递减的顺序形成空闲区链，分配时直接从空闲区链的第一个空闲区中分配（不能满足需要则不分配）。很显然，如果第一个空闲分区不能满足，那么再没有空闲分区能满足需要。这种分配方法初看起来不太合理，但它也有很强的直观吸引力：在大空闲区中放入程序后，剩下的空闲区常常也很大，于是还能装下一个较大的新程序。

最坏适应算法与最佳适应算法的排序正好相反，它的队列指针总是指向最大的空闲区，在进行分配时，总是从最大的空闲区开始查寻。该算法克服了最佳适应算法留下的许多小的碎片的不足，但保留大的空闲区的可能性减小了，而且空闲区回收也和最佳适应算法一样复杂。

- 优点：给文件分配分区后剩下的空闲区不至于太小，产生碎片的几率最小，对中小型文件分配分区操作有利。
- 缺点：使存储器中缺乏大的空闲区，对大型文件的分区分配不利。

## 练习2：实现寻找虚拟地址对应的页表项（需要编程）

通过设置页表和对应的页表项，可建立虚拟内存地址和物理内存地址的对应关系。其中的 `get_pte` 函数是设置页表项环节中的一个重要步骤。此函数找到一个虚地址对应的二级页表项的内核虚地址，如果此二级页表项不存在，则分配一个包含此项的二级页表。本练习需要补全 `get_pte` 函数 in `kern/mm/pmm.c`，实现其功能。请仔细查看和理解 `get_pte` 函数中的注

释。get\_pte函数的调用关系图如下所示：

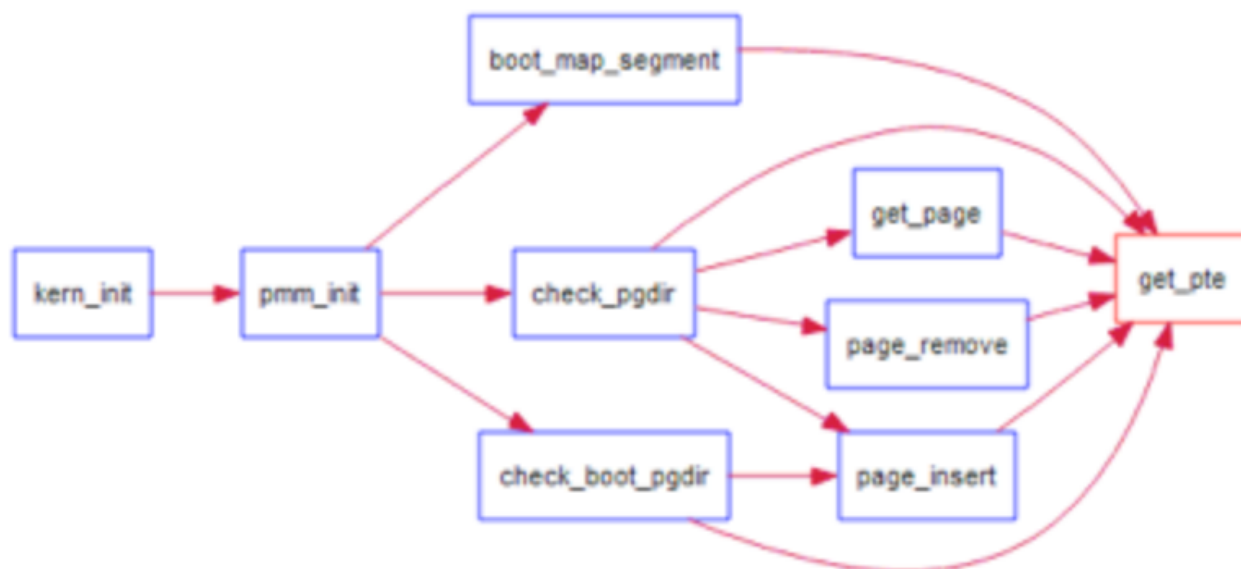


图1 get\_pte函数的调用关系图

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：

- 请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中每个组成部分的含义以及对ucore而言的潜在用处。
- 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？

## 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

# x86 MMU – 页机制概述

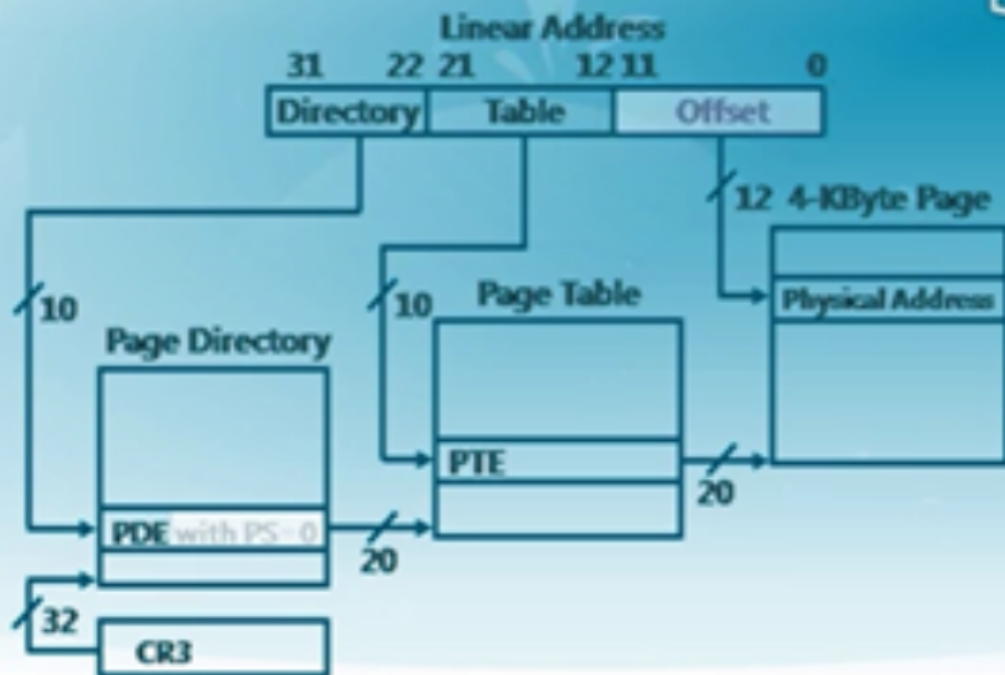


图 基于页机制的地址转换

## (1) 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

```
/* page table/directory entry flags */  
//在mmu.h文件中，对页目录项各个位的构成有说明如下：  
#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.
```

## (2) 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

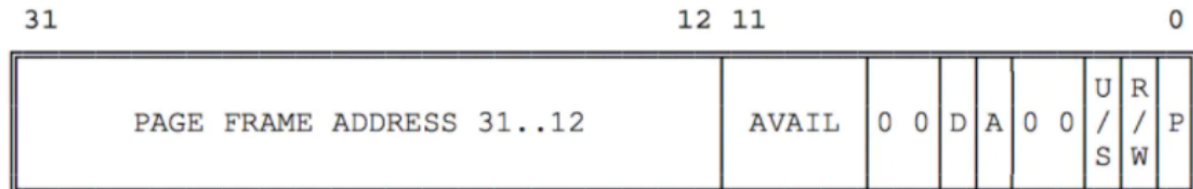
第7位恒为0

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



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.

## 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，转交给软件处理。

## 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1
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)
//补充:
*memset(void*s, char c, size_t n): 将s指向的内存区域的前n个字节设置为指定的值c。
```

需要完成的get\_pte函数原型如下

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

这个函数的作用为：获得二级页表中的页表项pte，并返回此pte的内核虚拟地址以供la使用。如果该pte不存在，根据create参数决定是否为该pte分配一页。

参数含义：

- pgdir: 一级页目录的起始地址，
- la: 需要映射的线性地址
- create: 表示是否可以为页表分配新的页

注释对该函数的步骤说明：

1. 查找页目录表项目（搜寻二级页表在一级页表里保存的地址）
2. 查看这样的目录项是否存在
3. 根据create变量查看有没有必要在不存在的时候创建表，如果需要的话，为二级页表申请分配一个页
4. 设置页的引用（表示这个物理页被一个进程使用，分配出去了。如果在目录项里查不到对应的二级页表，且需要分配一个页存放二级页表，那么这个被分配的页就需要给当前进程使用，所以要设置它被引用一次。）
5. 获得页的线性地址
6. 用memset清除页的内容
7. 设置页目录表项的权限
8. 返回页表项

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。

函数实现：

```
//函数功能：返回已知线性地址所对应页表项的指针
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    uintptr_t index=PDX(la); //获取目录表的索引
    pde_t *pde_ad = &pgdir[index]; //根据索引，获得目录表的表项的地址，这样方便
    后续修改这个表项的内容

    //若不存在，则需要创建或直接退出
    if(!(*pde_ad & PTE_P)) //先看bit 1位是否为1，即物理页是否存在。如果不存在，
    根据情况决定是否分配
```



```

{
if(create)
{
    struct Page *newpage=alloc_page();//分配一个页
    set_page_ref(newpage, 1);//设置这个页被引用，因为它是一个存放二级页表的页
    uintptr_t pa=page2pa(newpage); //获得这个页的物理地址，用于填写到pde目录项
    之1中
    *pde_ad =(pa & ~0x0FFF) | PTE_U | PTE_W | PTE_P;//重置页目录项
    memset(KADDR(pa), 0, PGSIZE);//设置这个页为全0，初始化清空
}
else
{
    return NULL; //如果不要求新建页，也返回空
}
}

//正式开始工作，返回其页表项
uintptr_t pt_pa=PDE_ADDR(*pde_ad);//从页目录项里，获得二级页表的物理地址
pte_t *mypte=KADDR(pt_pa);//将这个二级页表的物理地址转化为虚拟地址，然后赋值
给指针
index=PTX(1a);//获得在二级页表里的索引
mypte+=index;//增加以指向二级页表里相应的表项
return mypte;//返回这个指向表项的指针
}

```

### 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（需要编程）

当释放一个包含某虚地址的物理内存页时，需要让对应此物理内存页的管理数据结构Page做相关的清除处理，使得此物理内存页成为空闲；另外还需把表示虚地址与物理地址对应关系的二级页表项清除。请仔细查看和理解page\_remove\_pte函数中的注释。为此，需要补全在 kern/mm/pmm.c中的page\_remove\_pte函数。page\_remove\_pte函数的调用关系图如下所示：





图2 page\_remove\_pte函数的调用关系图

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：

- 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？
- 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

## 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。

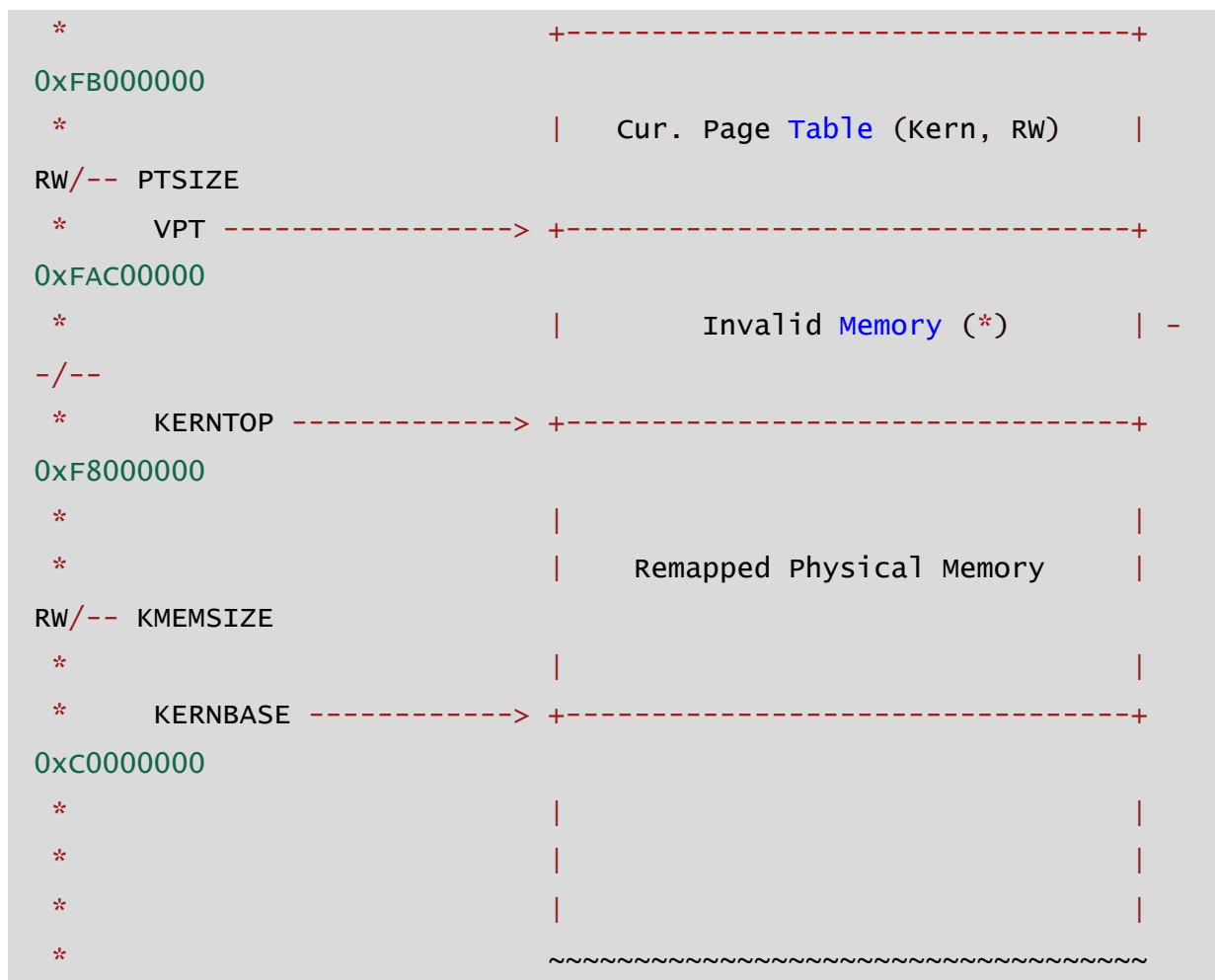
而在lab2中建立了从虚拟地址到物理地址的映射，ucore的物理地址仍为0x100000，但虚拟地址变为了0xC0100000，即最终建立的映射为：virt addr = linear addr = phy addr + 0xC0000000。

//memlayout.h给出了直观的映射关系

```

*      4G -----> +-----+
*                                     |
*                                     |      Empty Memory (*)      |
*                                     |
*                                     |

```



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)

```

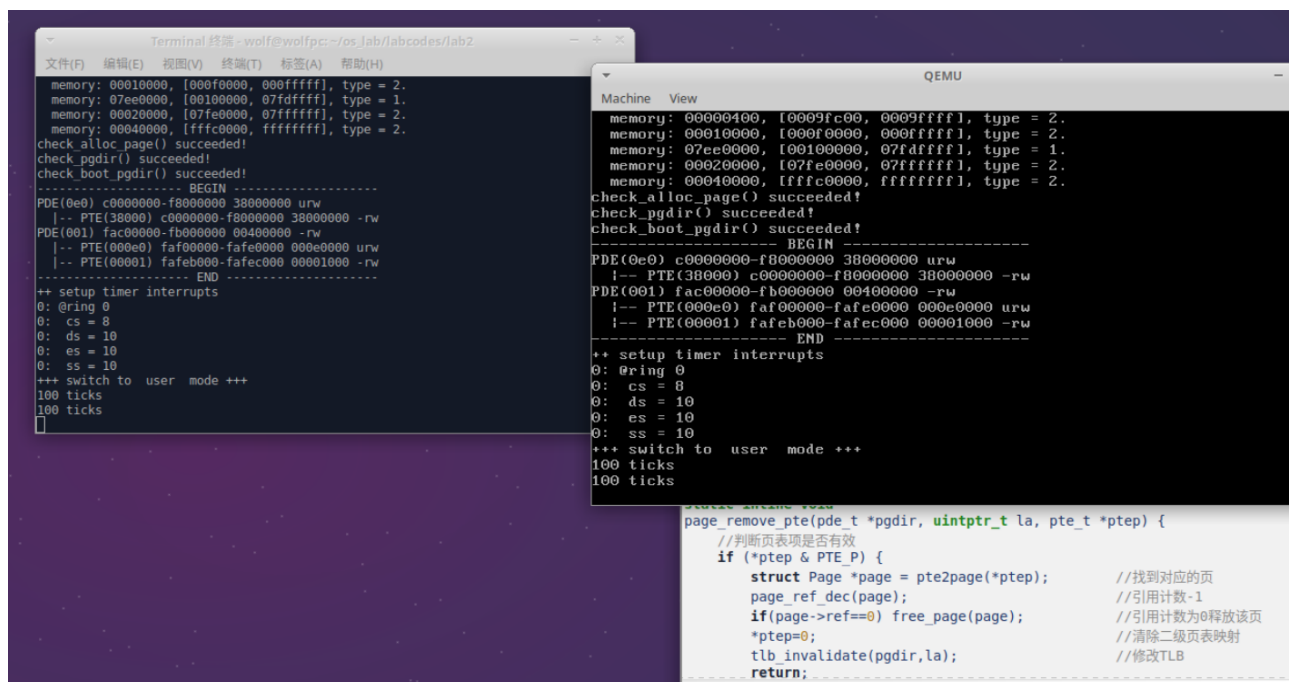
对于传入的页表项，首先要判断其是否有效，如果有效，将引用计数-1，当引用计数为0时释放该页，再清0二级页表映射，使用tlb\_invalidate清除对应的TLB项。最终实现如下：

```
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;
    }
    return;
}
```

## 结果测试

首先输入make clean,make qemu运行构建操作系统:



```
Terminal 终端 - wolf@wolfpc:~/os_lab/labcodes/lab2
文件(F) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)

memory: 00010000, [000f0000, 000fffff], type = 2.
memory: 07ee0000, [00100000, 07fdffff], type = 1.
memory: 00020000, [07fe0000, 07fffff], type = 2.
memory: 00040000, [ffffc000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
----- BEGIN -----
PDE(0e0) c0000000-f0000000 38000000 urw
|-- PTE(38000) c0000000-f0000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
|-- PTE(000e0) faf00000-fafe0000 000e0000 urw
|-- PTE(00001) fafeb000-fafec000 00001000 -rw
----- END -----
++ setup timer interrupts
0: @ring 0
0: cs = 8
0: ds = 10
0: es = 10
0: ss = 10
+++ switch to user mode +++
100 ticks
100 ticks
[ ]

QEMU
Machine View
memory: 00000400, [0009fc00, 0009ffff], type = 2.
memory: 00010000, [000f0000, 000fffff], type = 2.
memory: 07ee0000, [00100000, 07fdffff], type = 1.
memory: 00020000, [07fe0000, 07fffff], type = 2.
memory: 00040000, [ffffc000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
----- BEGIN -----
PDE(0e0) c0000000-f0000000 38000000 urw
|-- PTE(38000) c0000000-f0000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
|-- PTE(000e0) faf00000-fafe0000 000e0000 urw
|-- PTE(00001) fafeb000-fafec000 00001000 -rw
----- END -----
++ setup timer interrupts
0: @ring 0
0: cs = 8
0: ds = 10
0: es = 10
0: ss = 10
+++ switch to user mode +++
100 ticks
100 ticks
----- BEGIN -----
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;
    }
    return;
}
```

操作系统正常运行,显示检查成功(三个succeeded)

再次输入make clean,并用make grade检查测试结果:

```
Terminal 终端 - wolf@wolfpc: ~/os_lab/labcodes/lab2  
文件(F) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
100 ticks  
wolf@wolfpc:~/os_lab/labcodes/lab2$ make clean  
wolf@wolfpc:~/os_lab/labcodes/lab2$ make grade  
Check PMM: (2.1s)  
-check pmm: OK  
-check page table: OK  
-check ticks: OK  
Total Score: 50/50  
wolf@wolfpc:~/os_lab/labcodes/lab2$
```

通过测试。

### 扩展练习**Challenge1: 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...

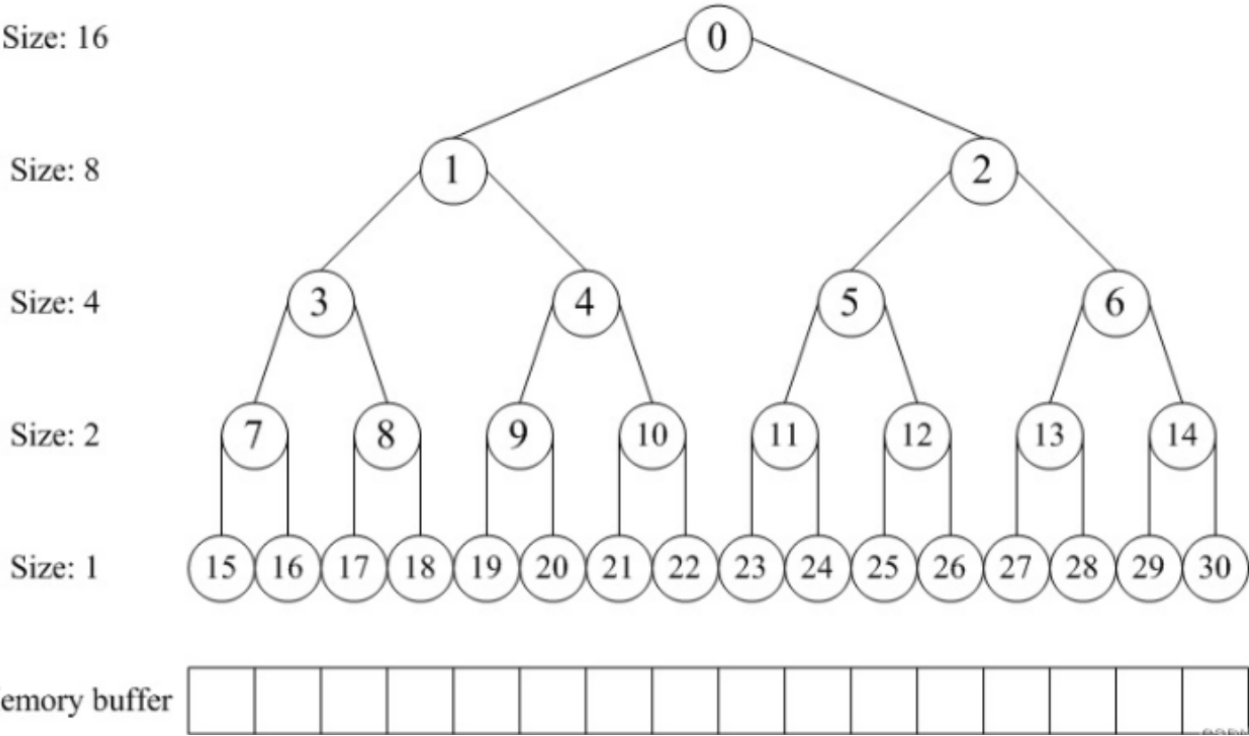
- 参考[伙伴分配器的一个极简实现](#)，在ucore中实现buddy system分配算法，要求有比较充分的测试用例说明实现的正确性，需要有设计文档。

## 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。

高层节点对应大的块，低层节点对应小的块，在分配和释放中我们就通过这些节点的标记属性来进行块的分离合并。

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：



	0	128k	256k	512k	1024k
start	1024k				
A=70K	A	128	256	512	
B=35K	A	B 64	256	512	
C=80K	A	B 64	C 128	512	
A ends	128	B 64	C 128	512	
D=60K	128	B D	C 128	512	
B ends	128	64 D	C 128	512	
D ends	256		C 128	512	
C ends	512			512	
end	1024k				

2.伙伴系统的实现

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)
        size = 1;
    else if (!IS_POWER_OF_2(size)) //调整为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。

## 扩展练习**Challenge2**: 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。

**SLAB** 分配器为每种使用的内核对象建立单独的缓冲区。每种缓冲区由多个 **slab** 组成，每个 **slab** 就是一组连续的物理内存页框，被划分成了固定数目的对象。根据对象大小的不同，节省情况下一个 **slab** 最多可以由 1024 个物理内存页框构成。

内核使用 **kmem\_cache** 数据结构管理缓冲区。由于 **kmem\_cache** 自身也是一种内核对象，所以需要有一个专门的缓冲区。所有缓冲区的 **kmem\_cache** 控制结构被组织成以 **cache\_chain** 为队列头的一个双向循环队列，同时 **cache\_chain** 全局变量指向 **kmem\_cache** 对象缓冲区的 **kmem\_cache** 对象。每个 **slab** 都需要一个类型为 **struct slab** 的描述符数据结构管理其状态，同时还需要一个 **kmem\_bufctl\_t**（被定义为无符号整数）的结构数组来管理空闲对象。如果对象不超过 1/8 个物理内存页框的大小，那么这些 **slab** 管理结构直接存放在 **slab** 的内部，位于分配给 **slab** 的第一个物理内存页框的起始位置；否则的话，存放在 **slab** 外部，位于由 **kmalloc** 分配的通用对象缓冲区中。

**slab** 中的对象有 2 种状态：已分配或空闲。

为了有效地管理 **slab**，根据已分配对象的数目，**slab** 可以有 3 种状态，动态地处于缓冲区相应的队列中：

1. **Full** 队列，此时该 **slab** 中没有空闲对象。
2. **Partial** 队列，此时该 **slab** 中既有已分配的对象，也有空闲对象。
3. **Empty** 队列，此时该 **slab** 中全是空闲对象。

在 **SLUB** 分配器中，一个 **slab** 就是一组连续的物理内存页框，被划分成了固定数目的对象。**slab** 没有额外的空闲对象队列（这与 **SLAB** 不同），而是重用了空闲对象自身的空间。**slab** 也没有额外的描述结构，因为 **SLUB** 分配器在代表物理页框的 **page** 结构中加入 **freelist**，**inuse** 和 **slab** 的 **union** 字段，分别代表第一个空闲对象的指针，已分配对象的数

目和缓冲区 `kmem_cache` 结构的指针，所以 `slab` 的第一个物理页框的 `page` 结构就可以描述自己。

每个处理器都有一个本地的活动 `slab`，由 `kmem_cache_cpu` 结构描述。

以下是 `slab_alloc` 的实现

```
static __always_inline void *slab_alloc(struct kmem_cache *s, gfp_t
gfpflags, int node, void *addr)
{
    void **object;
    struct kmem_cache_cpu *c;
    unsigned long flags;
    local_irq_save(flags);
    c = get_cpu_slab(s, smp_processor_id()); //获取本处理器的
kmem_cache_cpu 数据结构
    if (unlikely(!c->freelist || !node_match(c, node)))
        object = __slab_alloc(s, gfpflags, node, addr, c); // 假如当
前活动 slab 没有空闲对象，或本处理器所在节点与指定节点不一致，则调用
__slab_alloc 函数
    else {
        object = c->freelist; //获得第一个空闲对象的指针，然后更新指针使其指
向下一个空闲对象
        c->freelist = object[c->offset];
        stat(c, ALLOC_FASTPATH);
    }
    local_irq_restore(flags);
    if (unlikely((gfpflags & __GFP_ZERO) && object))memset(object,
0, c->objsize);
    return object; //返回对象地址
}
```

`static void *__slab_alloc(struct kmem_cache *s, gfp_t gfpflags, int node, void *addr, struct kmem_cache_cpu *c)`函数实现

```
static void *__slab_alloc(struct kmem_cache *s, gfp_t gfpflags, int
node, void *addr, struct kmem_cache_cpu *c)
{
    void **object;
    struct page *new;
    gfpflags &= ~__GFP_ZERO;
```

```

    if (!c->page) // (a): 如果没有本地活动 slab, 转到 (f) 步骤获取 slab
        goto new_slab;
    slab_lock(c->page);
    if (unlikely(!node_match(c, node))) // (b): 检查处理器活动 slab 没
        有空闲对象, 转到 (e) 步骤
        goto another_slab;
    stat(c, ALLOC_REFILL);
    load_freelist:object = c->page->freelist;
    if (unlikely(!object)) // (c): 此时活动 slab 尚有空闲对象, 将 slab
        的空闲对象队列指针复制到 kmem_cache_cpu 结构的 freelist 字段, 把 slab 的空闲
        对象队列指针设置为空, 从此以后只从 kmem_cache_cpu 结构的 freelist 字段获得空
        闲对象队列信息
        goto another_slab;
    if (unlikely(SlabDebug(c->page)))goto debug;
    c->freelist = object[c->offset]; // (d): 此时活动 slab 尚有空闲对
        象, 将 slab 的空闲对象队列指针复制到 kmem_cache_cpu 结构的 freelist 字段, 把
        slab 的空闲对象队列指针设置为空, 从此以后只从 kmem_cache_cpu 结构的 freelist
        字段获得空闲对象队列信息

    c->page->inuse = s->objects;
    c->page->freelist = NULL;
    c->node = page_to_nid(c->page);
    unlock_out:slab_unlock(c->page);
    stat(c, ALLOC_SLOWPATH);
    return object;
    another_slab:deactivate_slab(s, c); // (e): 取消当前活动 slab, 将其
        加入到所在 NUMA 节点的 Partial 队列中
    new_slab:new = get_partial(s, gfpflags, node); // (f): 优先从指定
        NUMA 节点上获得一个 Partial slab
    if (new) {
        c->page = new;
        stat(c, ALLOC_FROM_PARTIAL);
        goto load_freelist;
    }
    if (gfpflags & __GFP_WAIT) // (g): 加入 gfpflags 标志置有
        __GFP_WAIT, 开启中断, 故后续创建 slab 操作可以睡眠
        local_irq_enable();
    new = new_slab(s, gfpflags, node); // (h): 创建一个 slab, 并初始化
        所有对象
    if (gfpflags & __GFP_WAIT)local_irq_disable();
    if (new) {
        c = get_cpu_slab(s, smp_processor_id());
        stat(c, ALLOC_SLAB);

```

```

        if (c->page) flush_slab(s, c);
        slab_lock(new);
        setSlabFrozen(new);
        c->page = new;
        goto load_freelist;
    }
    if (!(gfpflags & __GFP_NORETRY) && (s->flags &
__PAGE_ALLOC_FALLBACK))
    {
        if (gfpflags & __GFP_WAIT) local_irq_enable();
        object = kmalloc_large(s->objsize, gfpflags); // (i): 如果内存不
        足, 无法创建 slab, 调用 kmalloc_large (实际调用物理页框分配器) 分配对象
        if (gfpflags & __GFP_WAIT)
            local_irq_disable();
        return object;
    }
    return NULL;

debug:
    if (!alloc_debug_processing(s, c->page, object, addr))
        goto another_slab;
    c->page->inuse++;
    c->page->freelist = object[c->offset];
    c->node = -1;
    goto unlock_out;
}

```

以下是slab\_free的实现

```

static __always_inline void slab_free(struct kmem_cache *s, struct
page *page, void *x, void *addr)
{
    void **object = (void *)x;
    struct kmem_cache_cpu *c;
    unsigned long flags;
    local_irq_save(flags);
    c = get_cpu_slab(s, smp_processor_id());
    debug_check_no_locks_freed(object, c->objsize);
    if (likely(page == c->page && c->node >= 0))
    { // (a)如果对象属于处理器当前活动的 slab, 或处理器所在 NUMA 节点号不为
    -1 (调试使用的值), 将对象放回空闲对象队列

```

```

    object[c->offset] = c->freelist;
    c->freelist = object;
    stat(c, FREE_FASTPATH);
}
else __slab_free (s, page, x, addr, c->offset); // (b) 否则调用
__slab_free 函数
    local_irq_restore(flags);
}

```

以下是static void \_\_slab\_free(struct kmem\_cache \*s, struct page \*page, void \*x, void \*addr, unsigned int offset)函数的实现

```

static void __slab_free(struct kmem_cache *s, struct page *page,
void *x, void *addr, unsigned int offset)
{
    void *prior; void **object = (void *)x;
    struct kmem_cache_cpu *c;
    c = get_cpu_slab(s, raw_smp_processor_id());
    stat(c, FREE_SLOWPATH);
    slab_lock(page);
    if (unlikely(SlabDebug(page))) goto debug;
    checks_ok: prior = object = page->freelist; // (a) 执行本函数表明对
象所属 slab 并不是某个活动 slab。保存空闲对象队列的指针，将对象放回此队列，最后把
已分配对象数目减一。
    page->freelist = object; page->inuse--;
    if (unlikely(SlabFrozen(page)))
    {
        stat(c, FREE_FROZEN);
        goto out_unlock;
    }
    if (unlikely(!page->inuse)) // (b) 如果已分配对象数为 0，说明 slab 处
于 Empty 状态，转到 (d) 步骤。
        goto slab_empty;
    if (unlikely(!prior))
    { // (c) 1. 如果原空闲对象队列的指针为空，说明 slab 原来的状态为 Full，那
么现在的状态应该是 Partial，将该 slab 加到所在节点的 Partial 队列中。
        add_partial(get_node(s, page_to_nid(page)), page, 1);
        stat(c, FREE_ADD_PARTIAL);
    }
    out_unlock: slab_unlock(page);
    return;
}

```



```

slab_empty:
if (prior) { // (d)如果 slab 状态转为 Empty, 且先前位于节点的 Partial 队
列中, 则将其剔除并释放所占内存空间。
    remove_partial(s, page);
    stat(c, FREE_REMOVE_PARTIAL);
}
slab_unlock(page);
stat(c, FREE_SLAB);
discard_slab(s, page);
return;

debug:if (!free_debug_processing(s, page, x, addr))goto
out_unlock;
goto checks_ok;
}

```

## 参考答案对比

### 练习1

1. default\_init函数实现未在原基础上做修改;
2. default\_init\_memmap函数在原基础上做的修改只是按照注释要求使用了 List\_add\_before而不是list\_add, 但由于初始构建空闲列表时只有两个结点、顺序无关, 所以实质上效果一样。和参考答案一致。
3. default\_alloc\_pages函数在原基础上, 在找到第一个合适的空闲块并拆分新空闲块的实现上, 虽然基本原理与参考一致, 但执行思路不同; 此外, 似乎不如参考答案的实现更加简洁。
4. default\_free\_pages函数在原基础上只修改了将空闲块添加到空闲列表时的方式, 保证了空闲块按照地址从小到大分步在空闲列表中。采用的循环检查方式和比较方式均不同于答案, 而答案中对空闲块的大小进行比较时明显更加严谨一些, 而且能够排除更多的潜在错误情况。

### 练习2

- 与参考答案相比, 使用的临时变量较多, 不是十分简洁; 判断是否分配页的时候, 不如参考答案精简, 对页目录项的赋值处理和返回指针的方式较为繁琐。

## 练习3

- 释放部分原本没有考虑到对引用的处理和辨析，所以最后借鉴了参考报告的思路（详见后续参考文献），基本和参考答案一致。

## 重要知识点和对应原理

### 实验中的重要知识点

- 连续内存管理机制
- 物理内存分配算法具体实现
- 实现双向链表的数据结构
- 利用函数指针和结构体近似面向对象功能
- 段机制与页机制相关数据结构和操作方法
- 虚拟地址到物理地址的映射和转换

### 对应的OS原理知识点

- 连续空间分配算法
- 分段机制、分页机制和多级页表
- 虚拟地址空间到物理地址空间的映射关系

## 二者关系

- 本实验设计的知识是对OS原理的具体实现，在细节上非常复杂。

### 未对应的知识点

- 页交换和页分配机制
- TLB快速缓存实现方法
- 页中断处理的详细软件机制
- 虚存地址空间实现方法
- 操作系统代码的映射关系和内核栈的具体实现

## 参考文献

对于slub分配算法参考了<https://www.bbsmax.com/A/gAJG6BegzZ/>

对于练习二与练习三的理解参考了<https://blog.csdn.net/Aaron503/article/details/130189764>

对于基础原理的理解部分参考了老师提供的2019级曹书与同学的实验报告