实验2报告

计科1904 曹书与 201908010416

预备知识

X86在操作系统初始化的过程中，已经启动了一个最基本的分段机制。程序中操作的地址被称为逻辑地址，逻辑地址经过分段机制的映射之后，将会得到一个线性地址。

在没有开启分页机制之前，线性地址本身可以作为物理地址使用；在开启分页机制之后，线性地址需要进一步通过分页机制的映射，才能得到物理地址。

所以，ucore中的分页机制是在分段机制的基础之上增加的。

ucore中的分页机制是通过二级页表实现的，其中一级页表称为页目录表，二级页表称为页表。

在32位的线性地址之中，以31-22位作为页目录索引，可以从页目录表中找到一个目录项，从中得到二级页表的基址，从而定位二级页表；以21-12位作为二级页表的索引，可以从二级页表中找到一个页表项，从中得到相应物理页的基址；将物理页基址与线性地址的11-0位偏移量结合起来，可以得到真正的物理地址。

学堂在线教学示意图如下：

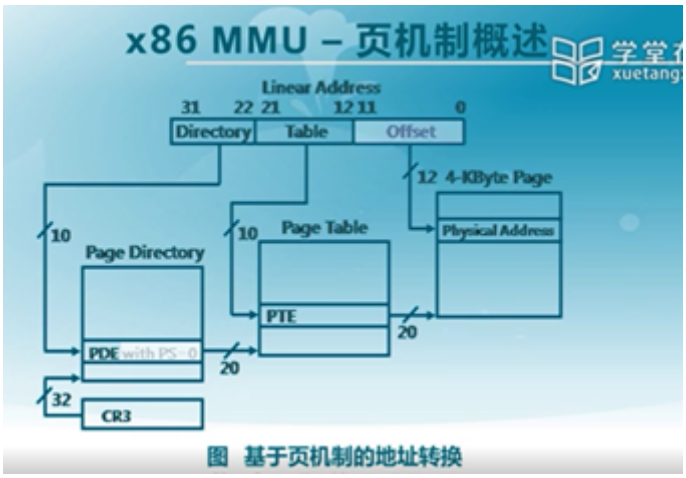


图 基于页机制的地址转换

可以看出，目录页表的基址保存在CR3寄存器之中；11-0位作为物理页的偏移量有12位，从而一个物理页含有2^12个字节即4K字节。

物理内存管理框架

lab1对页机制的考察较少，主要在于空闲空间的管理问题。在kern/init目录下的kern_init函数之中，通过调用pmm_init函数完成对物理内存的管理。

```
int
kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);
    cons_init();                // init the console
    const char *message = "(THU.CST) os is loading ...";
    cprintf("%s\n\n", message);
    print_kerninfo();
```

```
    grade_backtrace();
    pmm_init();                 // init physical memory management

    pic_init();                 // init interrupt controller
    idt_init();                 // init interrupt descriptor table
    clock_init();               // init clock interrupt
    intr_enable();              // enable irq interrupt
    /* do nothing */
    while (1);
}
```

kern/mm目录下的pmm_init函数内容如下：

```
//pmm_init - setup a pmm to manage physical memory, build PDT&PT to setup paging
mechanism
//         - check the correctness of pmm & paging mechanism, print PDT&PT
void
pmm_init(void) {
    // We've already enabled paging
    boot_cr3 = PADDR(boot_pgdir);
    //We need to alloc/free the physical memory (granularity is 4KB or other
size).
    //So a framework of physical memory manager (struct pmm_manager)is defined in
pmm.h
    //First we should init a physical memory manager(pmm) based on the
framework.
    //Then pmm can alloc/free the physical memory.
    //Now the first_fit/best_fit/worst_fit/buddy_system pmm are available.
    init_pmm_manager();
    // detect physical memory space, reserve already used memory,
    // then use pmm->init_memmap to create free page list
    page_init();

    //use pmm->check to verify the correctness of the alloc/free function in a
pmm
    check_alloc_page();
    check_pgdir();
    static_assert(KERNBASE % PTSIZE == 0 && KERNTOP % PTSIZE == 0);

    // recursively insert boot_pgdir in itself
    // to form a virtual page table at virtual address VPT
    boot_pgdir[PDX(VPT)] = PADDR(boot_pgdir) | PTE_P | PTE_W;

    // map all physical memory to linear memory with base linear addr KERNBASE
    // linear_addr KERNBASE ~ KERNBASE + KMEMSIZE = phy_addr 0 ~ KMEMSIZE
    boot_map_segment(boot_pgdir, KERNBASE, KMEMSIZE, 0, PTE_W);

    // Since we are using bootloader's GDT,
    // we should reload gdt (second time, the last time) to get user segments and
the TSS
    // map virtual_addr 0 ~ 4G = linear_addr 0 ~ 4G
    // then set kernel stack (ss:esp) in TSS, setup TSS in gdt, load TSS
    gdt_init();

    //now the basic virtual memory map(see memalyout.h) is established.
    //check the correctness of the basic virtual memory map.
    check_boot_pgdir();
```

```
    print_pgdir();

}
```

上述代码中的**init_pmm_manager()**函数用于初始化一个可以对物理内存进行管理的数据结构，内容如下：

```
//init_pmm_manager - initialize a pmm_manager instance
static void
init_pmm_manager(void) {
    pmm_manager = &default_pmm_manager;
    cprintf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}
```

其中pmm_manager是一个全局变量，它是一个pmm_manager结构体类型的数据；它内部包含多个成员变量，其中大部分成员变量都是函数指针。在c语言中，函数指针可以用于指向某个函数，从而调用函数指针时就可以调用函数；一个有许多函数指针作为成员变量的结构体，使用了面向对象设计的思想，类似于c++中的类。可以通过这个pmm_manager类型的指针（名也为pmm_manager）来获得内部成员函数指针，从而调用一系列与物理内存操作有关的函数。**它为操作系统其他的部分提供了对物理内存进行管理的接口。**

```
// physical memory management
const struct pmm_manager *pmm_manager;
```

对于操作系统而言，获得可用的物理内存范围之后，需要建立相应的数据结构对页进行管理。每一页的大小为4K，将页作为管理内存的最小单位，则需要能够对物理内存空间进行空间的分配和释放操作，这是分页机制的基础。**Ucore中使用physical memory manager也就是物理内存管理器来进行空闲空间的分配、释放以及空闲列表的相关操作，这个框架在一个名为pmm_manager的结构体中实现；它位于kern/pmm.h中，内容如下：**

```
// pmm_manager is a physical memory management class. A special pmm manager -
XXX_pmm_manager
// only needs to implement the methods in pmm_manager class, then XXX_pmm_manager
can be used
// by ucore to manage the total physical memory space.
struct pmm_manager {
    const char *name;                                 // XXX_pmm_manager's name
    void (*init)(void);                               // initialize internal
description&management data structure
                                                      // (free block list, number
of free block) of XXX_pmm_manager
    void (*init_memmap)(struct Page *base, size_t n); // setup
description&management data structcure according to
                                                      // the initial free
physical memory space
    struct Page *(*alloc_pages)(size_t n);            // allocate >=n pages,
depend on the allocation algorithm
    void (*free_pages)(struct Page *base, size_t n);  // free >=n pages with
"base" addr of Page descriptor structures(memlayout.h)
    size_t (*nr_free_pages)(void);                    // return the number of
free pages
    void (*check)(void);                              // check the correctness
of XXX_pmm_manager
```

```
};
```

利用这个数据结构的成员变量可以对物理内存进行相关操作。各成员变量含义如下：

```
const char *name;   //该物理内存管理器的名字，可用字符串赋值
void (*init)(void); //一个返回值为void的函数的指针，指针名称为init，用于初始化。
void (*init_memmap)(struct Page *base, size_t n);//就是一个返回值为void的函数的指针，指针名称为init_memmap，函数参数有两个,用于初始化空闲列表
struct Page *(*alloc_pages)(size_t n);//函数指针的名字是alloc_pages，参数只有一个，返回值类型居然也是一个指针，而且指向一个struct Page类型。用于对页进行分配。
void (*free_pages)(struct Page *base, size_t n);  //返回值为void的函数的指针，指针名称为free_pages，函数参数有两个。用于释放页。
size_t (*nr_free_pages)(void);   //一个返回值为size_t的函数的指针，指针名称为nr_free_pages，不需要参数。用于返回当前列表
void (*check)(void);  //一个函数指针，指向的函数的返回值为void，负责检查其他函数的实现效果。
```

在介绍Page类型之前，先要了解对pmm_manager的初始化是如何进行的。

```
//init_pmm_manager - initialize a pmm_manager instance
static void
init_pmm_manager(void) {
    pmm_manager = &default_pmm_manager;
    cprintf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}
```

重新查看此函数，发现它将全局变量pmm_manager(一个指向pmm_manager类型结构体的指针)以default_pmm_manager的地址赋值，然后在第三行里，调用了其成员的初始函数。可见，default_pmm_manager是一个已经为成员函数指针赋值完毕的pmm_manager类变量。其内容如下（kern/mm目录下，default_pmm.c文件里）：

```
const struct pmm_manager default_pmm_manager = {
    .name = "default_pmm_manager",
    .init = default_init,
    .init_memmap = default_init_memmap,
    .alloc_pages = default_alloc_pages,
    .free_pages = default_free_pages,
    .nr_free_pages = default_nr_free_pages,
    .check = default_check,
};
```

可见，它将作为整个操作系统用来管理物理内存的接口；操作系统可以通过这个接口，调用上述已经将地址赋予函数指针的函数，如default_init，default_init_memmap等函数（**它们都是函数名，函数名作为函数的起始地址被赋值给结构体的函数指针成员变量**）。顾名思义，它们是所谓的默认函数，即不进行任何操作时，率先使用这些函数来管理物理内存。

由此可知，练习1的要求就是对这些默认的物理内存管理函数进行修改，让它们满足first_fit内存分配算法。即，对空闲列表进行操作，在分配和释放空间时，遵循first_fit算法的实现逻辑。

Page——描述页的数据结构

实验要求以4K大小的页为基本单位管理物理内存。在Ucore中，视物理内存为一个由连续的页组成的大型数组；其中，某些页被分配了，某些没有。那些没有被分配的连续的页可以组成一个个较大的空闲块，所谓**空闲列表就是由描述这些空闲块位置和大小的结点所组成的链表**，通过对空闲列表进行查询，可以得知物理内存中当前有哪些空闲块（也就是连续的、可分配的物理页）。在分配页和释放页的过程中，需要对空闲列表进行相应的处理维护。

构建空闲列表之前，操作系统需要对每一个页的属性进行描述。每一个物理页都有一个名为Page的结构体变量加以描述，它的实现如下：

```
/* *
 * 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.
 * */
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
};
```

Page结构体描述了物理内存中一个页的状况，每一个物理页都有一个Page类型结构体对应描述。

成员变量ref为整型，它表示当前物理页被引用了多少次。在分页机制中，进程虚拟地址空间的每一个虚拟页都可以通过页机制被映射为一个真实存在的物理页；由于良好的共享机制，多个进程各自拥有的某一个虚拟页可以映射为同一个物理页，物理页被这些进程同时引用，ref就说明了物理页被多少个进程使用者。当ref为0时，意味着没有进程在使用这个物理页，该物理页是空闲可分配的。

成员变量flags为32位无符号整型变量，它作为一组标志位描述了页的状态。其配合如下宏使用：

```
/* Flags describing the status of a page frame */
#define PG_reserved                 0       // the page descriptor is reserved
for kernel or unusable
#define PG_property                 1       // the member 'property' is valid
```

flags的PG_reserved位也就是第0位为1的时候，说明这个Page描述的物理页是保留页；保留页不能被放入空闲页链表中，不能动态分配与释放（比如，操作系统内核占用的物理页就是保留页，无论如何都不能对内核所占据的空间进行分配释放操作）

如果flags的PG_reserved位也就是第0位为0，说明这个页不是保留页，可以参与到分配和释放中。此时，如果flags的PG_property位也就是第1位为1，说明这个物理页是空闲的，可以用于分配；反之，这个物理页已经被分配，直到被**所有引用它的进程释放**才能作为空闲页。

成员变量property为无符号整型，表示地址连续的空闲页的数目，涉及到空闲列表的实现机制。在ucore之中，物理内存中可能有若干空闲块，每个空闲块都是由一组连续的空闲物理页构成的；每一个空闲的物理页自然都有相应的Page进行描述，那么这些连续空闲页中的第一个空闲页的Page里，需要为其成员变量property赋值，表示连续空闲页的数目。因此，对于连续空闲页构成的空闲块，可以通过查找其第一个物理页的Page描述的property成员，来了解这个空闲块到底有多大，由多少个页组成。

list_entry_t类型的成员变量page_link是实现空闲列表的核心，它负责连接空闲列表的各个结点，其作用和含义将在接下来对双向链表的介绍中阐明。

实现双向链表的数据结构

Ucore中将用一个首尾闭合的双向链表作为空闲列表，存放各个空闲块的描述信息。list_entry是实现这一机制的核心。相关定义位于list.h文件中。

```
struct list_entry {
    struct list_entry *prev, *next;
};

typedef struct list_entry list_entry_t;
```

list_entry结构只有两个成员变量，分别是前驱和后继指针，可以指向其他list_entry类型的变量，构成一个双向链表。这个结构体拥有的类名是list_entry_t。

对list_entry_t类型构成的链表可以进行初始化、添加与删除操作，具体如下：

```
//初始化函数将一个list_entry_t类型的链表结点的前驱和后继都指向自己，形成闭环。
static inline void
list_init(list_entry_t *elm) {
    elm->prev = elm->next = elm;
}

//此函数获得指向后继结点的指针。
static inline list_entry_t *
list_next(list_entry_t *listelm) {
    return listelm->next;
}

//此函数获得指向前驱结点的指针。
static inline list_entry_t *
list_prev(list_entry_t *listelm) {
    return listelm->prev;
}

//此函数将elm结点插入到prev参数结点和next结点之间，将二者作为前驱和后继
static inline void
__list_add(list_entry_t *elm, list_entry_t *prev, list_entry_t *next) {
    prev->next = next->prev = elm;
    elm->next = next;
    elm->prev = prev;
}

//此函数将elm结点放在listelm和listelm原本的前驱之间，从而插入到listelm前。
list_add_before(list_entry_t *listelm, list_entry_t *elm) {
    __list_add(elm, listelm->prev, listelm);
}

//此函数将elm结点放在listelm和listelm原本的后继之间，从而插入到listelm后。
static inline void
list_add_after(list_entry_t *listelm, list_entry_t *elm) {
    __list_add(elm, listelm, listelm->next);
}

//此函数与list_add_after等价。
static inline void
list_add(list_entry_t *listelm, list_entry_t *elm) {
    list_add_after(listelm, elm);
```

```
}

//此函数将prev和next结点连在一起。
static inline void
__list_del(list_entry_t *prev, list_entry_t *next) {
    prev->next = next;
    next->prev = prev;
}

//通过链接一个结点的前驱和后继结点，结点自身从链表中被删除。
static inline void
list_del(list_entry_t *listelm) {
    __list_del(listelm->prev, listelm->next);
}
```

ucore使用两种数据结构构成空闲列表，一个是前文的Page类型，一个是下方的free_area_t。

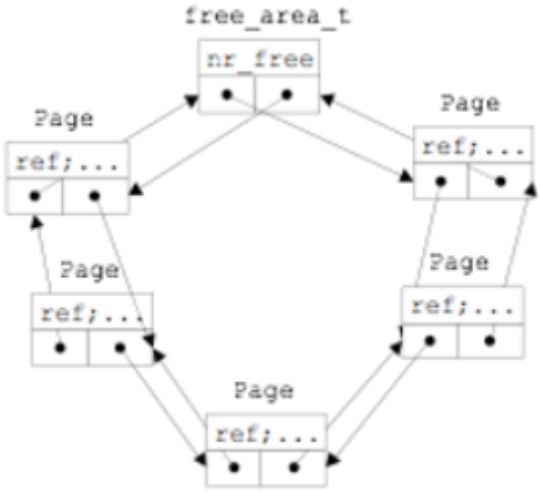
```
/* free_area_t - maintains a doubly linked list to record free (unused) pages */
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // # of free pages in this free list
} free_area_t;
```

free_area_t和Page类型都含有一个list_entry_t类型的成员变量。定义一个free_area_t类型的全局变量free_area，它将作为ucore中空闲列表的头节点——它自身不是Page类型变量，但是它的free_list成员变量可以通过指针指向其他Page类型变量的free_list成员。如下，free_area的两个成员均获得宏定义，从而 free_list表示指向双向链表下一结点的list_entry_t变量，nr_free表示整个空闲列表里所有的空闲页的数目（每一个空闲列表里的空闲块里全部空闲页的数目之和）。

```
free_area_t free_area;

#define free_list (free_area.free_list)
#define nr_free (free_area.nr_free)
```

最后，ucore里维护的空闲列表的结构如下：



将这个空闲列表视为双向链表，头节点为free_area_t类型的全局变量free_area，剩余结点均为Page类型。它们内部均含有list_entry_t类型的成员变量，这些成员变量通过指针互相联系起来。可以从free_area的成员free_list开始，根据指针依次寻找空闲列表中的各个Page结构的list_entry_t类型，然后用下文的le2page宏，根据从指针获得的list_entry_t而得到其所在的Page变量。

```
// convert list entry to page
#define le2page(le, member)                 \
    to_struct((le), struct Page, member)
```

简而言之，虽然不能通过指针直接指向各个Page结构，但是可以通过用指针指向双向列表某节点的list_entry_t类型，用le2page来获得它所在的Page。

空闲列表的构建

如上文所示，空闲列表由很多Page类型构成，每一个Page代表了一个由若干连续页构成的空闲块。由于每一个空闲块都是很多连续页构成的，**连续页里的第一个页对应的Page结构，将用来代表整个空闲块，被加入到空闲列表之中。**在空闲列表里的每一个Page类型，其成员变量property是以它自己开头的空闲块拥有的物理页的数目。

在ucore的实现中，首先需要初始化这些描述各个物理页的Page变量，然后在已经初始化完毕的各个Page变量的基础上，利用全局变量free_area来构建空闲列表。

描述物理页的Page的初始化：

内存中需要专门开辟一段位置用于存放描述了所有物理页的Page类型，而且应当连续存放。

根据实验指导书说明抄录如下：

首先根据bootloader给出的内存布局信息找出最大的物理内存地址maxpa（定义在page_init函数中的局部变量），由于x86的起始物理内存地址为0，所以可以得知需要管理的物理页个数为

```
npage = maxpa / PGSIZE
```

这样，我们就可以预估出管理页级物理内存空间所需的Page结构的内存空间所需的内存大小为：

```
sizeof(struct Page) * npage)
```

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**，记为：

```
pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);
```

为了简化起见，**从地址0到地址pages+ sizeof(struct Page) * npage)结束的物理内存空间**设定为已占用物理内存空间（起始0~640KB的空间是空闲的），**地址pages+ sizeof(struct Page) * npage)以上的空间为空闲物理内存空间**，这时的空闲空间起始地址为

```
uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) * npage);
```

(进行上述操作后，内存中从pages指向的地址，到pages+ sizeof(struct Page) * npage))指向地址，这一部分内存专门存放连续的Page类型变量；这一步空间一共有npage个Page结构变量，每一个都描述了某一个物理页。

为此我们需要把这两部分空间给标识出来。首先，对于所有物理空间，通过如下语句即可实现占用标记：

```
for (i = 0; i < npage; i ++) {
SetPageReserved(pages + i);
}
`
```

（由于pages是Page类型的指针，遍历的本质是遍历了连续存放的一个个Page变量；这些变量每一个都描述了一个物理页，使用SetPageReserved函数可以将这些Page的flags标志位设置为已保留状态。这一操作在后续的初始化中会被更新——因为，不可能让每一个物理页都是已保留的，不然还怎么进行分配和释放呢？上面这一步操作的目的只是为了标识出已经找到的物理页而已。）

然后，根据探测到的空闲物理空间，通过如下语句即可实现空闲标记：

```
//获得空闲空间的起始地址begin和结束地址end
……
init_memmap(pa2page(begin), (end - begin) / PGSIZE);
```

init_memmap函数则是把空闲物理页对应的Page结构中的flags和引用计数ref清零，并加到free_area.free_list指向的双向列表中，为将来的空闲页管理做好初始化准备工作。

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。

已完成，补充了实验1中练习5和6要求填写的代码。

练习1：实现 first-fit 连续物理内存分配算法

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

上述函数实现位于default_pmm.c文件里，该文件提供了大量注释。我会把注释的内容穿插在对函数实现的解释中。

概述：

在First Fit算法中，分配器保留一个空闲块列表（称为空闲列表）。一旦收到内存分配请求，**它将沿着列表扫描第一个足够大以满足请求的块。如果选择的块明显大于请求的块，则通常将其拆分，剩余的块将作为另一个空闲块添加到列表中。**

为了实现First Fit Memory Allocation（FFMA），我们应该使用列表管理空闲内存块。结构“free_area_t”用于管理可用内存块。（free_area就是free_area_t类型的头节点）

可以重用default_init函数来初始化free_list并将nr_free设置为0。

default_init函数实现：

```
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：

```
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_before(&free_list, &(base->page_link));
}
```

根据注释要去，此函数与原来相比只需要修改最后一行，使用list_add_before。

该函数的作用是初始化内存中的各个Page变量（这些变量连续存放在内存中，起始地址正是参数base，总数是n）。

函数首先用assert，在n>0不满足的时候警告。参数n是页的数目。然后用Page类型的指针p指向base。

base本身是指针，所以base+n实际上是第n+1个页的指针（时刻记住指针加法的特殊算法）。

接着，可用这种方式遍历内存里保存的每一个Page。由于之前已经SetPageReserved函数把这些Page的flags标志位设置为已保留状态，所以此处用assert进行确认；确认之后，需要把flags设置为0的，表示它们既没有保留也可以分配（两个标志位都被设置为0，说明这些物理页不是保留页，而且不可以分配）。property初始化为0，且**用set_page_ref将每一个页的引用位设置为0，表示没有页被引用过。**

遍历完之后，准备构建空闲列表。初始的连续n个物理页将被视为一个整体的空闲块，所以base指向的空闲块的第一个Page将代表这初始的巨大空闲块；它的property被设置为n，表示初始空闲块有完整的n个页大小；用 SetPageProperty将这个Page的property标志位设置为1，说明这个空闲块可以被分配；nr_free加上n，表示整个空闲列表一共n个空闲页；最后，用list_add_before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，一个是一个Page，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：

```
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) {
        nr_free -= n;
        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
        if (page->property > n) {
            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;
}
```

该函数负责分配n个页。参数n是要分配出来的页数，返回值是一个Page类型的指针。既然有多个Page变量在内存里连续存放，那么返回的Page指针所指向的Page，也就描述了那个被真实分配的一个空闲区域的首个物理页。

该函数基于原本的基础上进行了细化修改。基本思路分步介绍如下：

```
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;
        }
    }
```

以上部分不必修改。首先检查n。如果要分配的页数比总空闲页还多，直接返回空，分配失败。接下来设置一个Page型指针NULL， list_entry_t型指针le。free_list是空闲列表头节点free_area的成员变量，通过它构建的双向链表可以用上述方式进行遍历：不停用le指向后继结点的list_entry_t型成员，然后用le2page宏得到指向这个Page结点的指针。

前文说过，空闲列表由一个头节点和若干Page结点构成；每一个Page是一个空闲块里的第一个页。虽然Page在内存里连续排放，但是空闲列表里的各个Page只是被指针连起来了而已。此处用le2page在遍历空闲列表的时候指向列表里的Page，其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大于等于n，说明这个Page象征的空闲块足够大。根据First_fit算法，将把这个空闲块的前n个页分配出去，剩下的部分构成新的空闲块。返回值自然就是page，它是被分配出去的连续页里的第一个页对应的Page的指针。

```
if (page != NULL) {
        nr_free -= n;
        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
        if (page->property > n) {
            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(&(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));
        }

    }
```

接下来，上述代码在原本基础上进行了修改。

当page不为空，说明找到了合适的空闲块；于是，可以把将空闲列表头节点的nr_free减去n，表示可用空闲页数少了n；用 ClearPageProperty清除Page变量的property标志位，意味着它所代表的空闲块不再可分配。

接着，当前空闲块被分出n个页之后，剩下的部分需要作为新的空闲块；也就是说，原本空闲块的第一个Page作为空闲块的代表被放入了空闲列表，现在由于前n个页被分配出去，需要把第n+1个页作为新空闲块的代表放入空闲列表。

当page->property > n成立，说明当前空闲块被分割n个页后还有剩余的可作为新空闲块；于是，newnode指针将指向第n+1个原本空闲块的页，方法是令newnode=page + n，newnode指向的Page将代表新的空闲块。

于是，newnode指向的Page变量的property被设置为原本空闲块总页数减去n，即剩余页数；它的property标志位被设置为1，表示可分配；然后先将它加入到空闲列表里原本空闲块结点的后面，再删除原本的空闲块结点，就相当于这个新空闲块结点代替了原本的空闲块结点，称为空闲列表的一部分。

当page->property > n不成立，不需要设置新的空闲块，删掉原本的空闲块结点也就是将被分配的page即可。

一定要记住，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：

```
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;

    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));
}
```

函数负责释放n个页。分步介绍实现：

```
 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);
    }
```

释放函数的参数是指向Page的指针，还有一个参数是指定的要释放的页的数目。

用p指向参数。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。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。**如果这个页是已经保留的，或者是有property的，就终止。**

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了。

之前分配了一个n个页的空闲块，那么空闲块的第一个页的property的值必然是大于等于n的，但是其分配出去之后，flag的property标准位必然是0表示已经被分配出去了。释放的n个页的property位必须都是0且reserved位也都是0，即这些页不可分配、不是保留页。

```
base->property = n;
   SetPageProperty(base);
   list_entry_t *le = list_next(&free_list);
```

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。

然后，用le指向空闲列表头节点往后的第一个list_entry_t类。这也是空闲链表中的第一个Page结点拥有的list_entry_t类。如果内存没有被分配完毕，那么它就是一个空闲块的第一个页Page的list_entry_t类。

```
 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;
```

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。）

用le2page取出双向链表中list_entry_t类所在的Page结构指针，然后让le指向下一个。

现在，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，如果二者相加等于p，意味着要释放的空闲块是和双向链表中的这个空闲块是连在一起的。于是它们可以合并，base->property增加。同时，p不再是一个单独的空闲块，它的property位会被取消，然后从双向链表里删除。

假如p所代表的空闲块位于要释放的块的前面且相接，那么p将会合并在前，base则指向新空闲块的头部，将p的list_entry_t类从双向链表里删除。最后，base指向新的大空闲块的第一个页（它的list_entry_t类依然不在链表里），而被它合并的块的首页的list_entry_t类都被删除。

最后，nr_free增加n，表示可用的页多了。

```
    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));
```

现在要把base指向的页的list_entry_t类放入双向链表。最粗暴的方法，就是直接把新的list_entry_t类放在头的后面。如果每次都这样搞，那么各次放入的新的list_entry_t类对应的页地址就是杂乱无章的。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。

进一步的改进空间：

1、我的算法在查找第一个可用空闲块时采用的是用while进行遍历的方式，理论上明显有更快的方法进行查找，比如给空闲列表设置哈希值。

2、我并没有考虑到足够的异常情况，所有的算法都基于可以正确分配和释放内存的假设，没有考虑可能出现的非法操作或其他异常情况。举例来说，如果空闲列表因某种原因损坏或失常，我无法检测出这个问题。

3、我采用的first_fit的空闲列表是按照地址由小到大排列各个空闲块的。但是为了能早些找到合适的空闲块，也许可以尝试在释放的时候把较大的空闲块放在前面。这样做并不会影响释放空闲块时的合并操作，所以可能加快找到合适空闲块的速度。

练习2：实现寻找虚拟地址对应的页表项

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。**
- **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?**

为了补全此函数，首先前往对应文件查看注释。

```
//get_pte - get pte and return the kernel virtual address of this pte for la
//        - if the PT contians this pte didn't exist, alloc a page for PT
// parameter:
//  pgdir:  the kernel virtual base address of PDT
//  la:     the linear address need to map
//  create: a logical value to decide if alloc a page for PT
// return vaule: the kernel virtual address of this pte
```

get_pte函数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，请为该pte分配一页

参数：pgdir:PDT的内核虚拟基址；la：需要映射的线性地址；create：一个逻辑值，用于决定是否为PT分配页面；返回值：此pte的内核虚拟地址。

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

解析：多级页表分页机制下，需要一级页表（又称页目录表）和二级页表（简单称为页表）共同进行寻址。

get_pte函数的第一个参数是一个pde_t类型的指针也就是pgdir。其中，pde是页目录项的缩写，即页目录表中的每一个条目都是pde_t类型的，从而pgdir这个指向pde_t类型变量的指针，可以作为整个页目录表的起始地址。

函数的第二个参数是无符号整型la，它是一个线性地址。在32位的线性地址之中，以31-22位作为页目录索引，可以从页目录表（起始地址是pgdir）中找到一个目录项（目录项类型为pde_t），从中得到二级页表的基址，从而定位二级页表（被称为PT，即Page Table缩写）；以21-12位作为二级页表的索引，可以从二级页表中找到一个页表项（页表项类型为pte_t），从中得到相应物理页的基址；将物理页基址与线性地址的11-0位偏移量结合起来，可以得到真正的物理地址。

函数的第三个参数是布尔变量create。在一级页表中按照某个索引进行查询的时候，某个pde页目录项理论上对应一个二级页表的基址；但是，也有可能并不存在这样的一个二级页表，即未分配。此时，根据create的值决定是否专门分配一个页的空间用来创建一个二级页表，然后将其基址构成页目录表对应索引处的新pde。

由实验指导书说明：

设一个32bit线性地址la有一个对应的32bit物理地址pa，**如果在以la的高10位为索引值的页目录项中的存在位（PTE_P）为0**，表示缺少对应的页表空间，则可通过alloc_page获得一个空闲物理页给页表，页表起始物理地址是按4096字节对齐的，这样填写页目录项的内容为

```
页目录项内容 = (页表起始物理地址 &~0x0FFF) | PTE_U | PTE_W | PTE_P
```

页目录项的本质是一个32位的整数。由于一个二级页表占据一个页且页是4k对齐的，所以二级页表的起始物理地址的低12位为0。因此，将其和~0x0FFF相与以得到高20位、将低12位置零，然后依次与PTE_U 、 PTE_W 、 PTE_P相或（它们是常量，可以给32位数据的位3、位2、位1置1）。

一个页目录项中，前20位作为基址，取出后左移12位就能得到二级页表的起始地址；后12位是标志位，用于描述和二级页表有关的信息，比如PTE_U对应的bit3表示用户态的软件可以读取对应地址的物理内存页内容 、 PTE_W对应的bit2表示物理内存页内容可写 、 PTE_P对应的bit1表示物理内存页存在。

之所以这些置位的常量以PTE开头，是因为pde目录项和pte页表项的结构是类似的，都是前20位为基址，后12位为标志位，使用的时候取二级页表的页表项pte的前20位并左移12位，就能得到最终物理页的起始地址。同样的，根据物理页基址构建页表项的方法如下：

进一步对于页表中以线性地址la的中10位为索引值对应页表项的内容为

```
页表项内容 = (pa & ~0x0FFF) | PTE_P | PTE_W
```

其中：

PTE_U：0x004，位2，表示用户态的软件可以读取对应地址的物理内存页内容

PTE_W：0x002，位1，表示物理内存页内容可写

PTE_P：0x001，位0，表示物理内存页存在

注释提示使用的宏或函数：

*PDX（la）=虚拟地址la的页目录项索引。
*KADDR（pa）：获取物理地址并返回相应的内核虚拟地址。（pa是物理地址，physical address）
*set_page_ref（page,1）：表示该页面被一次引用
*page2pa（page）：page的类型是Page*指针，根据练习1可知，它管理一个物理页；可以用它返回这个物理页的物理地址。
*struct Page *alloc_page（）：分配一个页
*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。

```
static inline void
set_page_ref(struct Page *page, int val) {
    page->ref = val;
}
```

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memeset清除页的内容
7、 设置页目录表项的权限
8、 返回页表项

函数实现：

```
pte_t *
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
    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
        }

    }
```

```
    //返回一个地址
    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(la);//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
}
```

逐步分析：

```
 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
```

参数la作为uintptr_t整型变量，其前10位可以作为一级页表页目录表的索引，这段内容可以用PDX宏取出。

```
// page directory index
#define PDX(la) ((((uintptr_t)(la)) >> PDXSHIFT) & 0x3FF)
```

取出索引后用Index保存，将pgdir理解为由目录项构成的数组，可以取出索引对应的目录项；由于目录项是一个pde_t类型的元素且有可能需要修改，用pde_t类型的指针pde_ad指向相应目录项的地址。pde_ad指向的pde_t数据就是目录项，它的前20位可以作为二级页表的基址，后12位含有一些标志位。

```
 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
        }

    }
```

*pde_ad可以取出指针指向的pde目录项的值然后和PTE_P相与，得到bit 1位的值，这个位可以用于判断页目录表的这一条目是否对应着一个二级页表；在if判断中，如果为0，说明当前不存在对应的二级页表，需要根据参数create的取值判断是否有必要新建一个二级页表。

如果create为真，则需要新建一个二级页表。用alloc_page()分配一个页，返回一个描述物理页的Page类型指针newpage表示新分配的物理页，调用set_page_ref将这个物理页设置为已经引用了一次；

接着，用page2pa得到这个分配的新物理页的物理地址。取物理地址的高20位并在低12位按照上文描述的方法给三个标志位赋值，得到一个32位的整型数字可以作为页目录表的新表项，并通过pde_ad进行指针赋值修改页目录表。

最后，用memeset将已分配的物理页清零。KADDR(pa)获得了这个物理页的虚拟地址，用于作为函数参数；PGSIZE是一个页的大小，0做为参数把整个页清零。由于这个页分配出来是用于存放二级页表的，所以需要把这个页先清空

如果create为假，则不需要新建一个二级页表，直接返回空指针就行了。

在完成对页目录项的更新以后，需要继续根据线性地址的中间10位作为索引，从而去二级页表里获得页表项（各个页表项类型是pte_t），并返回页表项的虚拟地址。方法如下：

```
    //返回一个地址
    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(la);//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
```

*pde_ad是根据线性地址前10位为索引、在页目录表里获得的页目录项，它的前20位通过PDE_ADDR宏取出并左移12位得到了二级页表的物理地址pt_pa；接着用KADDR将这个物理地址转化为虚拟地址，赋值给pte_t类型指针。

```
// address in page table or page directory entry
#define PTE_ADDR(pte)   ((uintptr_t)(pte) & ~0xFFF)
#define PDE_ADDR(pde)   PTE_ADDR(pde)
```

由于二级页表可以视为pte_t类型元素构成的数组，因此用mypte指针作为二级页表开始地址可以快速查找其中的元素。用PTX宏取出线性地址中间10位作为新索引index，并移动mypte指针，就可以指向二级页表里相应项的位置（虚拟地址），最后返回即可。

- **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。**

在mmu.h文件中，对页目录项各个位的构成有说明如下：

```
/* 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.
```

目录项项的高20位是对应二级页表基址的高20位。

bit 0位为存在位，用于说明对应的二级页表是否存在。

bit 1位为可写位，用于说明对应的二级页表是否可写。

bit 2位为特权位，用于表示用户态的软件可以读取对应地址的物理内存页内容。

bit 3设置是否使用Write-Through缓存写。

bit 4位为1时表示不对其进行缓存。

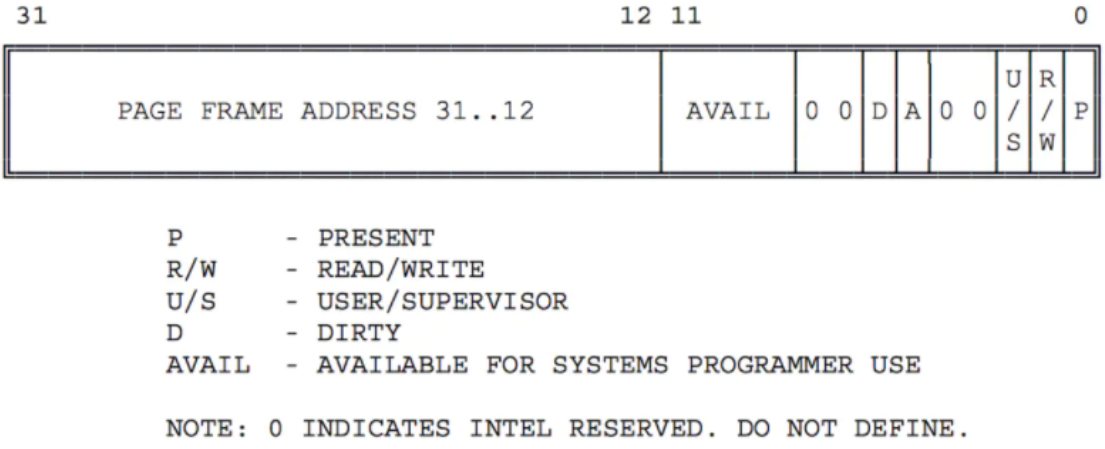
bit 5位表示页是否已经使用过。

bit 7设置页的大小。

bit 8必须为0。

bit 9-11给软件使用。

页表项构成如下：



基本和页目录项类似。

高20位是指向的物理页的基址；

bit 9-11位留给软件使用；

bit 7-8位为0；

bit 6位表示该页是否为脏页；

bit 5位表示是否被操作过；

bit 3-4位为0；

bit 0-2位左移作用和目录项类似。

- **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?**

页访问异常会引发中断，从而硬件将在内核栈中保存CS，EIP，EFLAGS等寄存器内容以及错误码，接着根据错误发生时的特权级决定是否需要发生特权级变化；接着在内核态下根据中断号找到中断描述符，跳转到终端服务例程处理页错误。剩下的部分是软件行为。

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

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。

- 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？

函数相关注释和参数、返回值描述如下：

```
//page_remove_pte - free an Page sturct which is related linear address la
//                - and clean(invalidate) pte which is related linear address la
//note: PT is changed, so the TLB need to be invalidate
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep)
```

释放一个包含某虚拟地址的物理页。使用的参数包含页目录表的起始地址pgdir，需要释放的页所拥有的线性地址la，以及指向二级页表里一个项的指针ptep。

page_remove_pte函数：**释放一个与线性地址la相关的Page结构，同时清理（使无效化）一个与线性地址la相关的pte(页表项)。由于一个pte（页表项）是管理一个物理页的，既然物理页自己都要释放了，这个条目自然页不需要存在了，应该无效化。**

注：PT（页表，实际上就是二级页表）已更改，因此TLB需要失效。

需要涉及的函数及相关说明：

```
* pte2page（*ptep）：从ptep的值获取相应页面。Ptep是一个指向二级页表某条目的指针，可以用这个函数获取二级页表相应条目所对应的物理页Page。
*free_page：释放页面。释放页面之后还需要无效化pte页表项。
 *page_ref_dec(page)：减少page->ref，即修改page的成员变量，减少引用次数。注意：当且仅当page->ref==0，此页面才是自由页面。Page->ref的值实际上说明了这个物理页被多少个进程所引用着，释放一个页以后，自然要让其值减去1；如果减去1后这个值为0了，说明不再有任何一个进程引用这个物理页，此时才可以释放它。
*tlb_invalidate（pde_t*pgdir，uintpttr_t la）：使tlb条目无效，但仅当正在编辑的页表是处理器当前正在使用的页表时。
```

执行步骤：

```
1、  使用PTE_P检查页目录是否有对应的物理页
2、  找到pte所对应的页面
3、  减少页面的ref值，意味着使用这个页面的进程少了一个
4、  如果ref变成0了，直接释放这个页面
5、  清除二级页表的相关条目
6、  更新tlb。
```

函数实现：

```
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep){
    if(*ptep & PTE_P)
    {   struct Page *mypage = pte2page(*ptep);
        int reference=page_ref_dec(mypage);
        if (reference==0)
        { free_page(mypage); }
        *ptep = 0;
        tlb_invalidate(pgdir, la);
    }
}
```

要删除ptep指针所指向的这个页表项。首先，将该页表项的具体值和PTE_P相与以得到存在位，即是否存在对应的物理页。

如果这个存在位确实是1，说明物理页存在，则需要指向操作；先使用pte2page取出物理页的Page类型指针。由于当前进程放弃了这个物理页，所以使用page_ref_dec让Page类型的ref成员（引用此页的进程数）减去1；如果发现减去1之后引用数变成0，说明不再有进程使用此物理页，直接调用free_page释放即可。

ptep页表项不再需要，直接全部设置为0；接着调用 tlb_invalidate使tlb条目无效。

```
static inline int
page_ref_dec(struct Page *page) {
    page->ref -= 1;
    return page->ref;
}
```

- **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。页目录项中的高20位左移12位可以得到32位的二级页表基址；在相应二级页表里根据线性地址的索引查到页表项，页表项的前20位左移12位可以得到32位的物理页地址，这个物理页的状态和属性利用Page结构描述。

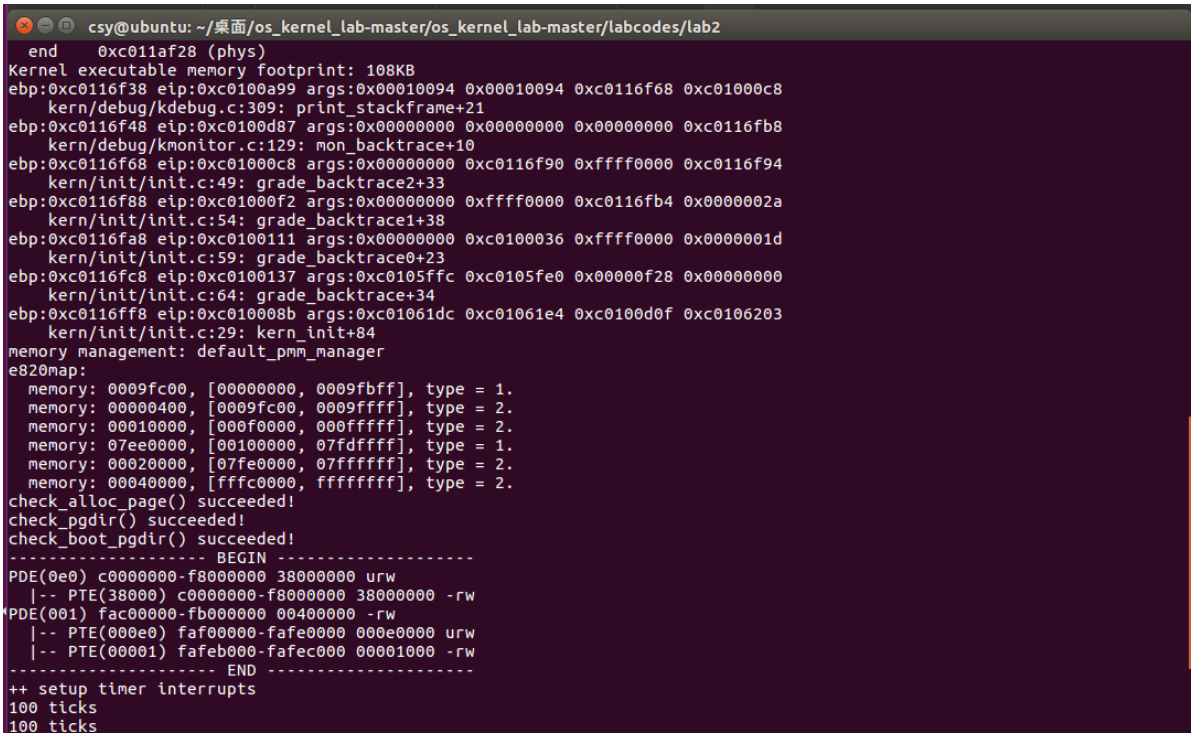
- **如果希望虚拟地址与物理地址相等，则需要如何修改lab2，完成此事？**

在lab2最后建立的映射关系中，逻辑地址（虚拟地址）=线性地址=物理地址+0xC000 0000。

想要让虚拟地址和物理地址相等，需要修改段映射关系，使得：逻辑地址（虚拟地址）=线性地址-0xC000 0000=物理地址。

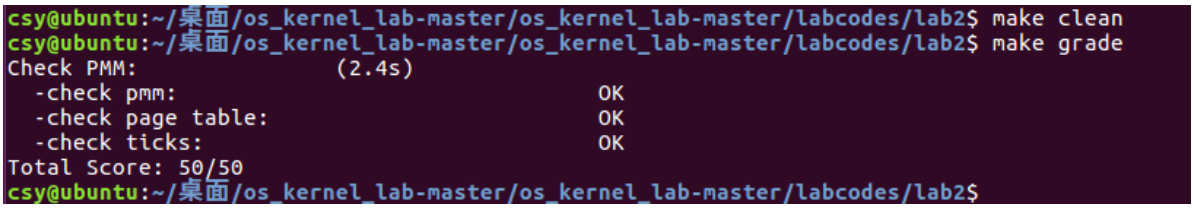
结果测试

首先输入make clean,make qemu运行构建操作系统：



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：



通过测试。

参考答案对比

练习1：

default_init函数实现未在原基础上做修改；

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。和参考答案自然也一致。

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shixian 更加简洁。

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达分步在空闲列表中。采用的循环检查方式和比较方式均不同于答案，而答案中在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，本次实现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似乎页显得啰嗦了一些。唯一可取之处在于步骤虽然繁琐但是清楚，但参考答案似乎是更加高级的编写方式。

练习3：

释放部分原本没有考虑到对引用的处理和辨析，所以最后借鉴了他人思路，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