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

list_entry 结构体：

Page 结构体：

free_area_t 结构体：

pmm_manager 结构体：

default_init 函数：

default_init_memmap 函数：

Answer：

default_free_pages 函数：

Answer：

default_alloc_pages 函数：

Answer：

练习2：实现寻找虚拟地址对应的页表项

Answer：

参考链接：

练习0：填写已有实验

没啥用，复制粘贴就行

练习1：实现 first-fit 连续物理内存分配算法

因为完全不会，所以去 csdn 找了几篇文章先看了看

list_entry 结构体：

写于：**libs/list.h**

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

typedef struct list_entry list_entry_t;
```

这里定义了一个双向链表

Page 结构体：

写于：**kern/mm/memlayout.h**

```
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
};
```

1. ref 表示的是：这个页被页表的引用记数，也就是映射此物理页的虚拟页个数

   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 加一

   反之，若页表项取消，即映射关系解除，就会把 Page 的 ref 减一

2. flags 表示此物理页的状态标记，有两个标志位状态

   为 1 的时候，代表这一页是 free 状态，可以被分配，但不能对它进行释放

   如果为 0，那么说明这个页已经分配了，不能被分配，但是可以被释放掉

3. property 用来记录某连续空闲页的数量

   这里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（第一页的地址）**

4. 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

   释放的时候只要将这个空间通过指针放回到双向链表中

free_area_t 结构体：

写于：**kern/mm/mmlayout.h**

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

在初始情况下，也许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

包含了一个 list_entry 结构的双向链表指针和记录当前空闲页的个数的无符号整型变量 nr_free

其中的链表指针指向了空闲的物理页

该数据结构中含有两个成员：

free_list：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

其中，在 **kern/mm/pmm.c** 定义了一个该数据结构的实例，用于管理实际分配当中的空闲空间

ucore 利用一个物理内存管理类 pmm_manager 确定使用需要的分配算法：**kern/mm/pmm.h**

pmm_manager 结构体：

```
struct pmm_manager {
    const char *name;  // 物理内存页管理器的名字
    void (*init)(void);  // 初始化内存管理器
    void (*init_memmap)(struct Page *base, size_t n);  // 初始化管理空闲内存页的数据结构
    struct Page *(*alloc_pages)(size_t n);  // 分配 n 个物理内存页
    void (*free_pages)(struct Page *base, size_t n);  // 释放 n 个物理内存页
    size_t (*nr_free_pages)(void);  // 返回当前剩余的空闲页数
    void (*check)(void);  // 用于检测分配/释放实现是否正确的辅助函数
};
```

总的来说，一个物理内存管理类包含这个类的名字和一些函数，类似：初始化、分配页、释放页等操作

可以看做类似于 lab6 中的调度的五元组的绑定方法

同样，我们所实现的函数，也是通过名字绑定的，具体的绑定过程在 **kern/mm/pmm.c**：

```
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,
};
```

接下来，我们来实现 First fit 算法的相关函数：

default_init，default_init_memmap，default_alloc_pages， default_free_pages

---

default_init 函数：

```
free_area_t free_area;

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

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

调用库函数 list_init 初始化掉 free_area_t

即管理所有连续的空闲内存空间块的数据结构 free_area_t 的双向链表和空闲块数

1. **list_init 函数：**

   写于：**libs/list.h**

   ```
   static inline void
   list_init(list_entry_t *elm) {
       elm->prev = elm->next = elm;
   }
   ```

   大致作用就是初始化双向循环链表

这里有两个宏定义，在 **kern/mm/pmm**：

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

从这里看出来，初始化的对象是 free_area_t

这是与具体物理内存分配算法无关的，因此直接使用默认的函数实现即可

default_init_memmap 函数：

init_memmap 函数主要实现的是一个根据现有的内存情况构建空闲块列表的初始状态的功能

何时应该执行这个函数呢，因此我们需要研究一下这个函数是如何被调用的：

调用过程是：kern_init → pmm_init → page_init → init_memmap

1. **kern_init 函数：**

   写于 **kern/init/init.c**

   这个函数是进入 ucore 操作系统之后第一个执行的函数，对内核进行初始化

   其中调用了初始化物理内存的函数 pmm_init

2. **pmm_init 函数：**

   写于：**kern/mm/pmm.c**

这个函数主要是完成对于整个物理内存的初始化，页初始化只是其中的一部分

调用位置偏前，函数之后的部分可以不管，直接进入 page_init 函数

3. **page_init 函数：**

   写于：**kern/mm/pmm.c**

```
static void
page_init(void) {
struct e820map *memmap = (struct e820map *)(0x8000 + KERNBASE);
    // 首先声明一个e820map类的对象memmap，与物理内存相关
    // 在本实验中，我们获取内存信息的方式是通过e820中断（一种特殊的内核中断模式）
    uint64_t maxpa = 0;

    cprintf("e820map:\n");
    int i;
    for (i = 0; i < memmap->nr_map; i ++) {
        // 这里可以看做一个遍历，第一轮遍历是遍历物理地址空间
        // 获取物理地址的最大值maxpa（探测物理内存布局）
        uint64_t begin = memmap->map[i].addr, end = begin + memmap->map[i].size;
        cprintf("  memory: %08llx, [%08llx, %08llx], type = %d.\n",
                memmap->map[i].size, begin, end - 1, memmap->map[i].type);
        if (memmap->map[i].type == E820_ARM) {
            if (maxpa < end && begin < KMEMSIZE) {
                maxpa = end;
            }
        }
    }
    // maxpa不能大于所允许的最大值，这个最大值宏定义在memlayout的第57行
    // 物理地址所允许的最大值为0x38000000(KMEMSIZE)
    if (maxpa > KMEMSIZE) {
        maxpa = KMEMSIZE;
    }

    extern char end[];

    // 创建的页数量等于物理地址最大值除以页大小，其中页大小为4096字节，即4KB
    // 该定义在mmu.h的第226行
    npage = maxpa / PGSIZE;
    pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);

    // 将所有的页设置为保留页，在实际初始化页面init_memmap的时候，又会更改回非保留
    // 推测是在初始化过程中这样处理，是为了防止页面被分配，结构被破坏。
    for (i = 0; i < npage; i ++) {
        SetPageReserved(pages + i);
    }

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

    // 第二次遍历物理内存，这一次遍历主要是调用init_memmap初始化各个页表
    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;
            }
            if (end > KMEMSIZE) {
                end = KMEMSIZE;
            }
            if (begin < end) {
                begin = ROUNDUP(begin, PGSIZE);
                end = ROUNDDOWN(end, PGSIZE);
                if (begin < end) {
                    init_memmap(pa2page(begin), (end - begin) /
PGSIZE);
                }
            }
        }
    }
}
```

page_init 函数主要是完成了一个**整体物理地址的初始化过程**，包括设置标记位，探测物理内存布局等操作

上面函数的注释中，标出了几个关键位置代码

但是，其中最关键的部分，也是和实验相关的页初始化，交给了 init_memmap 函数处理

这里我们主要研究一下上方 init_memmap 传入的两个参数：

1. **pa2page(begin)**

   写于：**kern/mm/pmm.h**

```
static inline struct Page *
pa2page(uintptr_t pa) {
    if (PPN(pa) >= npage) {
        panic("pa2page called with invalid pa");
    }
    return &pages[PPN(pa)];
}
```

   其中 **PPN 是物理地址页号**

   该函数的作用是：返回传入参数 pa 开始的第一个物理页

   其基地址是 base（default_init_memmap 函数里有写）

2. **(end - begin) / PGSIZE**

   由于 end 和 begin 都是循环中记录位置的标记，PGSIZE 为 4 KB 的页大小，这里就代表物理页的个数

接下来看 default_init_memmap 函数：

```
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));
}
```

这里先看几个不理解的函数：

1. **set_page_ref 函数：**

   写于：**kern/mm/pmm.h**

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

   这个函数就是设置记录映射此物理页的虚拟页的变量的数值

2. **test_bit 函数：**

   写于：**libs/atomic.h**

   ```
   static inline bool
   test_bit(int nr, volatile void *addr) {
       int oldbit;
       asm volatile ("btl %2, %1; sbbl %0,%0" : "=r" (oldbit) : "m" (*(volatile long *)addr), "Ir" (nr));
       return oldbit != 0;
   }
   ```

   这里 bt 的作用是把 *addr 的第 nr 位复制到 cf

   btl 里的 l 是用于操作 32 位的长字值所使用的后缀

   sbb 与 sub 的区别：
   sub ax, bx 的结果是 ax - bx
   sbb ax, bx 的结果是 ax - bx - cf（进/借位标志）

   这里 oldbit 初始化的时候不用管值是多少，最后经过 sbbl 汇编，只会有两个值

   一个是 0，一个是 -1

   那么最后检查 oldbit 的值是否为 0，若 oldbit 的值为 0，则 *addr 第 nr 位的值为 0，否则为 1

   也就是说，这个函数返回 true 的话，该测试位上的值就为 0，反之则为 1

3. **PageReserved 函数：**

   写于：**kern/mm/memlayout.h**

   ```
   #define PageReserved(page)          test_bit(PG_reserved, &((page)->flags))
   ```

   接下来看 **PG_reserved** 的宏定义：

   ```
   #define PG_reserved                 0       // if this bit=1: the Page is reserved for kernel, cannot be used in alloc/free_pages; otherwise, this bit=0
   ```

这个函数的意思就是检查 `(page)->flags` 这个值的第 0 位是否为 0

如果为 0，就可以顺利通过 `assert(PageReserved(p));` 语句

如果为 1，则报告错误并终止程序

Reserved 表示该页被操作系统保留，不能进行分配或者释放

那么该函数的意思就是如果该页不是被操作系统保留的，那么就不报错，顺利通过

4. **set_bit 函数：**

   写于：**libs/atomic.h**

```
/* *
 * set_bit - Atomically set a bit in memory
 * @nr:     the bit to set
 * @addr:   the address to start counting from
 *
 * Note that @nr may be almost arbitrarily large; this function is not
 * restricted to acting on a single-word quantity.
 * */
static inline void
set_bit(int nr, volatile void *addr) {
    asm volatile ("btsl %1, %0" :"=m" (*(volatile long *)addr) : "Ir"
(nr));
}
```

   bts 在执行 bt 命令的同时（把 *addr 的第 nr 位复制到 cf），把操作数的指定位置设置为 1

   这个函数的功能就是将 *addr 的第 nr 位设为 1

   第一个占位符 %0 与 C 语言变量 addr 对应，第二个占位符 %1 与 C 语言变量 nr 对应

   因此上面的汇编语句代码与此伪代码等价：`btsl nr, addr`

   该指令的两个操作数不能全是内存变量，因此将 nr 的限定字符串指定为 Ir

   将 nr 与立即数或者寄存器相关联，这样两个操作数中只有 addr 为内存变量

5. **SetPageProperty 函数：**

   写于：**kern/mm/memlayout.h**

```
#define SetPageProperty(page)       set_bit(PG_property, &((page)-
>flags))
```

   接下来看 **PG_property** 的宏定义：

```
#define PG_property                 1       // if this bit=1: the Page
is the head page of a free memory block(contains some
continuous_addrress pages), and can be used in alloc_pages; if this
bit=0: if the Page is the the head page of a free memory block, then
this Page and the memory block is alloced. Or this Page isn't the head
page.
```

   这个函数的意思就是修改 `(page)->flags` 这个值的第 1 位为 1

6. **list_add 函数等：**

   写于：**libs/atomic.h**

```
static inline void
```

```
list_add(list_entry_t *listelm, list_entry_t *elm) {
    list_add_after(listelm, elm);
}

static inline void
list_add_before(list_entry_t *listelm, list_entry_t *elm) {
    __list_add(elm, listelm->prev, listelm);
}

static inline void
list_add_after(list_entry_t *listelm, list_entry_t *elm) {
    __list_add(elm, listelm, listelm->next);
}

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;
}
```

所以能看出来 list_add 实质上就是 list_add_after，它和 list_add_before 都调用了 __list_add

Answer：

```
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));
}
```

这个函数的主要目的就是初始化所有空闲物理页的 Page 结构，具体初始化方式为：

`assert(n > 0);` 确保不会有分配 0 页的情况发生

`struct Page *p = base;` 设置指针变量 p，为了在之后遍历 n 页

`for (; p != base + n; p ++)` 循环内容：

`assert(PageReserved(p));` 确保该页没有被操作系统保留（检测 **p->flags** 的第 0 位）

`p->flags = p->property = 0;` 将 Page 结构中的两个成员变量的值置零，可以当作手动初始化

将遍历到的物理页的状态标记设置为 0，说明该页已经被分配了

`set_page_ref(p, 0);` 清空这些物理页的引用计数

`base->property = n;` 设置页头中记录连续空闲页数量的值为 n

`SetPageProperty(base);` 修改 `(base)->flags` 值的第 1 位为 1

`nr_free += n;` 更新总空闲块数目（该变量一开始在 **default_init 函数**中初始化）

`list_add_before(&free_list, &(base->page_link));`

将 **&(base->page_link)** 插入到 **(&free_list)->prev** 和 **&free_list** 的中间

这里因为 **&free_list** 是双向链表，所以插入是没有问题的，用 **list_add** 也没问题

default_free_pages 函数:

先看初始函数:

```
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));
}
```

继续先看不认识的函数:

1. **PageProperty 函数:**

   写于：**kern/mm/memlayout.h**

   ```
   #define PageProperty(page)          test_bit(PG_property, &((page)->flags))
   ```

   这个函数跟 **PageReserved 函数**是一个意思

   不过这次是检查 `(page)->flags` 这个值的第 1 位是否为 0

   如果为 0，就可以顺利通过 `assert(PageProperty(p));` 语句

   如果为 1，则报告错误并终止程序

说白点就是如果该页不是连续空闲空间的第一页，那么就不报错，顺利通过

2. **list_next 函数：**

   写于：**libs/list.h**

```
/* *
 * list_next - get the next entry
 * @listelm:    the list head
 **/
static inline list_entry_t *
list_next(list_entry_t *listelm) {
    return listelm->next;
}
```

   字面意思，获得该列表的下一个元素

3. **offsetof 函数：**

   写于：**libs/defs.h**

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

   又学到了，先说 **&((type *)0)->member** 是啥：

   ANSI C 标准允许值为 0 的常量被强制转换成任何一种类型的指针，并且转换的结果是 NULL

   因此 **((type *)0)** 的结果就是一个类型为 **type *** 的 NULL 指针

   如果利用这个 NULL 指针来访问 type 的成员当然是非法的

   但 **&((type *)0)->member** 意图是想取 member 的地址

   所以编译器不会生成访问 type 成员的代码，而会优化为直接取地址

   该函数的作用是获取结构体中 member 成员相对于该结构体首元素地址的偏移量

4. **to_struct 函数：**

   写于：**libs/defs.h**

```
/* *
 * to_struct - get the struct from a ptr
 * @ptr:    a struct pointer of member
 * @type:   the type of the struct this is embedded in
 * @member: the name of the member within the struct
 * */
#define to_struct(ptr, type, member)                               \
    ((type *)((char *)(ptr) - offsetof(type, member)))
```

   ptr 为一个真实地址，同时也是 **type *** 结构体中 member 成员变量的地址

   减去自身在 **type *** 中的偏移就可以得到这个结构体的起始地址

   那么这个函数就是用来获得这个成员变量所在结构体的起始地址

5. **le2page 函数：**

   写于：**kern/mm/memlayout.h**

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

经过 grep 命令，可以发现这个函数里的 le 变量基本都来自这里：**list_entry_t *le = list_next(&free_list);**

这个函数的作用就是依靠作为 Page 结构体中 member 成员变量的 le 变量，得到 le 成员变量所对应的结构体头变量

6. **clear_bit 函数：**

   写于：**libs/atomic.h**

```
/* *
 * clear_bit - Atomically clears a bit in memory
 * @nr:     the bit to clear
 * @addr:   the address to start counting from
 * */
static inline void
clear_bit(int nr, volatile void *addr) {
    asm volatile ("btrl %1, %0" :"=m" (*(volatile long *)addr) : "Ir" (nr));
}
```

   btr 在执行 bt 命令的同时，把操作数的指定位置为 0

   所以这个函数是用来将 *addr 的第 nr 位的值设置为 0 的

7. **ClearPageProperty 函数：**

   写于：**kern/mm/memlayout.h**

```
#define ClearPageProperty(page)     clear_bit(PG_property, &((page)->flags))
```

   这个函数就是把 **(page)->flags** 的第 1 位设置为 0

   意思就是将该页标记为并非是连续空闲空间的第一页

8. **__list_del 函数：**

   写于：**libs/list.h**

```
/* *
 * Delete a list entry by making the prev/next entries point to each other.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 * */
static inline void
__list_del(list_entry_t *prev, list_entry_t *next) {
    prev->next = next;
    next->prev = prev;
}
```

   双向链表删除元素的日常操作，就是把中间的元素断开，将上下两个元素相连

   但是这里没有清除掉中间的元素，算是一个心里的疑惑

9. **list_del 函数：**

写于：**libs/list.h**

```
/* *
 * list_del - deletes entry from list
 * @listelm:    the element to delete from the list
 *
 * Note: list_empty() on @listelm does not return true after this, the
entry is
 * in an undefined state.
 * */
static inline void
list_del(list_entry_t *listelm) {
    __list_del(listelm->prev, listelm->next);
}
```

将 **listelm->prev** 与 **listelm->next** 相连，使 **listelm** 节点断连

Answer：

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);  // 防止出现 n == 0 的情况发生
    struct Page *p = base;  // 设置 p 变量，一会要遍历，先从头开始
    /* 开始遍历所有的物理空闲页，从 base 开始，经过 n 页后结束 */
    for (; p != base + n; p ++) {
        /* 如果该页既是连续空闲空间的第一页，又被系统保留了，就不能通过 */
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;  // 将记录标志位的变量置零
        set_page_ref(p, 0);  // 将记录映射此物理页的虚拟页的变量的数值清零
    }
    base->property = n;  // 将第一页结构体中记录连续空闲页数量的变量更新为 n
    SetPageProperty(base);  // 将第一页的 flags 变量中标记上该页是第一页
    list_entry_t *le = list_next(&free_list);  // 获取 &free_list 变量的下一个
列表内容
    /* *
     * 循环遍历双向链表，初始地址为 &free_list 的下一个地址
     * 直到遍历到 &free_list 地址，其实就是遍历整个链表
     * 但是不进入链表头 &free_list 进行操作
     * */
    while (le != &free_list) {
        p = le2page(le, page_link);  // 获取当前链表块所对应的 Page 结构的结构体
头地址
        /* *
           下文没有用到 le 的，那么这里的意思就是更新 le 为下一个地址
           为的是在进入下一个 while 语句后，寻找其所对应的 Page 结构体头
         * */
        le = list_next(le);
        /* 被释放的空闲块后面紧接着空闲块 p */
        if (base + base->property == p) {
            /* *
             * 更新 base 页表头的 property 变量
             * 将接下来要被释放的页表头的 property 变量加到 base 页表头的 property
变量上
             * 差不多就是合并后面的空闲块 p 的意思
             * */
```

```
            base->property += p->property;
            /* *
             * 清除 p 结构体 flags 变量上的 Property 标志位
             * 意思就是让 p 不再成为某一页的页表头
             * */
            ClearPageProperty(p);
            list_del(&(p->page_link));  // 在链表中删除掉 p->page_link 的地址，即释放掉 p
        }
        /* 被释放的空闲块前面紧接着空闲块 p */
        else if (p + p->property == base) {
            p->property += base->property;  // 合并前面的空闲块 p
            ClearPageProperty(base);  // 让 base 不再成为某一页的页表头
            base = p;  // 更新 base 的地址为 p 的地址
            list_del(&(p->page_link));  // 释放掉自己
        }
    }
    nr_free += n;  // 更新总空闲块数目
    le = list_next(&free_list);  // 获取 &free_list 的下一个地址
    /* 与上面的 while 循环一样的遍历机制，不过这里新加了 break 的条件 */
    while (le != &free_list) {
        p = le2page(le, page_link);  // 获取当前链表块所对应的 Page 结构的结构体头地址
        /* *
         * 合并完成之后，由于需要维持链表有序，需要找到这个新的空闲块在链表中的插入位置
         * 这里采用了插入排序的思想，遍历链表，找到第一个地址比需要插入的空闲块结尾地址大的空闲块，插入到它的前面即可
         * 由于前面的过程保证了不会有空闲块越界和粘连的情况发生，所以判断条件改为 base <= p 也是可以的
         * */
        if (base + base->property <= p) {
            assert(base + base->property != p);
            break;
        }
        le = list_next(le);
    }
    list_add_before(le, &(base->page_link));
}
```

default_alloc_pages 函数:

先看初始函数:

```
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;
}
```

Answer:

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);  // 防止出现 n == 0 的情况发生
    /* 若是申请的空闲块数目大于原有的空闲块数目，就返回 NULL */
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;  // 声明一个 Page 结构体变量
    list_entry_t *le = &free_list;  // 声明一个 list_entry_t 结构体变量
    // TODO: optimize (next-fit)
    /* *
       根据 First-Fit 算法，从地址小到大遍历每一个空闲块，一旦找到了足够大的空闲块就马上分配
       也就是分配第一个足够大的空闲块给程序使用。找不到，说明没有这么大的连续空闲块，则返回空指针
     * */
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    /* *
       如果找到了这样的空闲块，还要检查空闲块的大小是否比需要分配的大小更大
       如果是，则需要进行分裂。分裂的操作便是将空闲块的后面的空闲块重新形成空闲块
       这个空闲块的第一个页面是 page + n，大小是 page->property - n
       设置好 property 和 flags 中的 property 位后，在链表中加入到当前空闲块的后面（因为地址更大）
     * */
    if (page != NULL) {
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add_after(&(page->page_link), &(p->page_link));
        }
        list_del(&(page->page_link));  // 删除 page 节点
        nr_free -= n;  // 更新物理空闲块数目
        ClearPageProperty(page);  // page 页不再是第一页
    }
```

```
    return page;
}
```

至此就完成了 First-Fit 算法

练习2：实现寻找虚拟地址对应的页表项

先看初始函数：

```
//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
pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    /* 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
     */
#if 0
    pde_t *pdep = NULL;   // (1) find page directory entry
    if (0) {              // (2) check if entry is not present
                          // (3) check if creating is needed, then alloc page for page table
                          // CAUTION: this page is used for page table, not for common data page
                          // (4) set page reference
```

```
        uintptr_t pa = 0; // (5) get linear address of page
                          // (6) clear page content using memset
                          // (7) set page directory entry's permission
    }
    return NULL;          // (8) return page table entry
#endif
}
```

先看不了解的定义和函数：

PDX 定义：

写于：**kern/mm/mmu.h**

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

这里 **PDXSHIFT** 的定义也在该文件中

```
#define PDXSHIFT        22                  // offset of PDX in a linear
address
```

page2ppn 函数：

写于：**kern/mm/pmm.h**

```
static inline ppn_t
page2ppn(struct Page *page) {
    return page - pages;
}
```

pages 的定义也在该文件里，如下：

```
extern struct Page *pages;
```

所以 pages 的值应该是依靠外部代码改变的，具体的内容暂且还不知道

这个函数的意思是：**获取物理页对应的物理页号**

page2pa 函数：

写于：**kern/mm/pmm.h**

```
static inline uintptr_t
page2pa(struct Page *page) {
    return page2ppn(page) << PGSHIFT;
}
```

PGSHIFT 的定义在 **kern/mm/mmu.h** 中，如下：

```
#define PGSHIFT         12                  // log2(PGSIZE)
```

该函数的意思是：**获取物理页对应的物理地址**

__panic 函数：

写于：**kern/debug/panic.c**

```
static bool is_panic = 0;

/* *
 * __panic - __panic is called on unresolvable fatal errors. it prints
 * "panic: 'message'", and then enters the kernel monitor.
 * */
void
__panic(const char *file, int line, const char *fmt, ...) {
    if (is_panic) {
        goto panic_dead;
    }
    is_panic = 1;

    // print the 'message'
    va_list ap;
    va_start(ap, fmt);
    cprintf("kernel panic at %s:%d:\n    ", file, line);
    vcprintf(fmt, ap);
    cprintf("\n");

    cprintf("stack trackback:\n");
    print_stackframe();

    va_end(ap);

panic_dead:
    intr_disable();
    while (1) {
        kmonitor(NULL);
    }
}
```

该函数的作用在英文的已经有了标注：在不可解决的致命错误上调用。它输出 `panic: 'message'`，然后进入内核监视器。

panic 宏定义：

写于：**kern/debug/assert.h**

```
#define panic(...)                                      \
    __panic(__FILE__, __LINE__, __VA_ARGS__)
```

读代码就知道这个宏定义是对 **__panic 函数**的封装

KADDR 宏定义：

写于：**kern/mm/pmm.h**

```
/* *
 * KADDR - takes a physical address and returns the corresponding kernel virtual
 * address. It panics if you pass an invalid physical address.
 * */
#define KADDR(pa) ({                                                    \
            uintptr_t __m_pa = (pa);                                    \
            size_t __m_ppn = PPN(__m_pa);                               \
            if (__m_ppn >= npage) {                                     \
                panic("KADDR called with invalid pa %08lx", __m_pa);    \
            }                                                           \
            (void *) (__m_pa + KERNBASE);                               \
        })
```

看英文的意思，该宏定义的作用就是通过物理地址找到对应的逻辑(虚拟)地址

PDE_ADDR 宏定义：

写于：**kern/mm/mmu.h**

```
#define PTE_ADDR(pte)   ((uintptr_t)(pte) & ~0xFFF)
#define PDE_ADDR(pde)   PTE_ADDR(pde)
```

Answer：

```
pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    /* *
     * pdep -- Page Directory Entry Pointer
     * 这里的 pgdir 可以看做是页目录表的基址
     * 获取到页目录表中给定线性地址对应到的页目录项
     * */
    pde_t *pdep = &pgdir[PDX(la)];
    /* *
     * 检查查找到的页目录项是否存在，如果存在直接放回找到的页表项即可
     * 即检查是否设置了 Present 位，也就是 PDE_P 位
     * 不过实际上并没有 PDE_P 这个宏，使用注释里告诉我们用等价的 PTE_P 来检查
     * 注释里告诉我们这个位是 PDE 和 PTE 通用的
     * */
    if (!(*pdep & PTE_P)) {
        struct Page *page;
        /* *
         * 如果该页目录项是不存在的，那么就不创建新的页表 (!create)
         * 或者物理空间不足 (page)，直接返回 NULL
         * alloc_pages 函数就是之前练习一里写的 default_alloc_pages 函数
         * 通过 alloc_pages() 分配的页地址 并不是真正的页分配的地址
         * 实际上只是 Page 这个结构体所在的地址而已
         * 故而需要通过使用 page2pa() 将 Page 这个结构体的地址转换成真正物理页地址的线性地址
         * 然后需要注意的是无论是 * 或是 memset 都是对虚拟地址进行操作的
         * 所以需要将 真正的物理页地址再转换成内核逻辑(虚拟)地址
         * */
        if (!create || (page = alloc_page()) == NULL) {
            return NULL;
```

```
29          }
30          set_page_ref(page, 1);  // 更新该物理页的引用计数
31          uintptr_t pa = page2pa(page);  // 获取物理页对应的物理地址
32          /* *
33           * 利用 KADDR 宏定义获取该物理页对应的逻辑(虚拟)地址
34           * 此时已经启动了page机制，内核地址空间，这是因为CPU执行的指令中使用的已经是虚拟地址了
35           * 新创建的页表进行初始化
36           * */
37          memset(KADDR(pa), 0, PGSIZE);
38          *pdep = pa | PTE_U | PTE_W | PTE_P;  // 设置 PDE 权限
39      }
40      /* *
41       * */
42      return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];
43  }
```

参考链接：

https://blog.csdn.net/weixin_34327223/article/details/85450897

https://blog.csdn.net/abraham_li/article/details/49474721

https://blog.csdn.net/rosetta/article/details/90746936

http://ilinuxkernel.com/?p=1062