

# 操作系统原理实验

## 实验三 - 物理内存管理

数据科学与计算机学院 软件工程专业 教务2班 17343131 许滨楠

### 实验准备

主机环境: macOS X 10.14.4 Mojave

虚拟机环境: Linux Ubuntu 14.04 LTS


虚拟机搭载软件: Parallels Desktop.app

命令行终端: Linux 下 Terminal

终端shell: zsh

上一次的实验二中, 实验用虚拟机硬盘文件.vdi搭建的虚拟机环境存在兼容性不好, 卡顿严重, 分辨率也不高等问题。故从这次实验开始, 在Github上将原实验项目clone下来, 自己按照指导书第一章讲的环境配置, 安装所需支持, 配置实用工具进行实验。

为满足实验要求, 将命令行用户名字段指定为姓名。(需要在zsh主题文件中修改显示。因为中文姓名在命令行终端会影响显示效果, 故用我的中大NetID作为署名标记: 许滨楠 - xubn)



```
xubn@ubuntu ~  
xubn@ubuntu ~  
xubn@ubuntu ~  
xubn@ubuntu ~  
xubn@ubuntu ~
```

### 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

- 了解如何发现系统中的物理内存；
- 了解如何建立对物理内存的初步管理，了解连续物理内存管理；
- 了解页表的相关操作，包括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。

## 练习

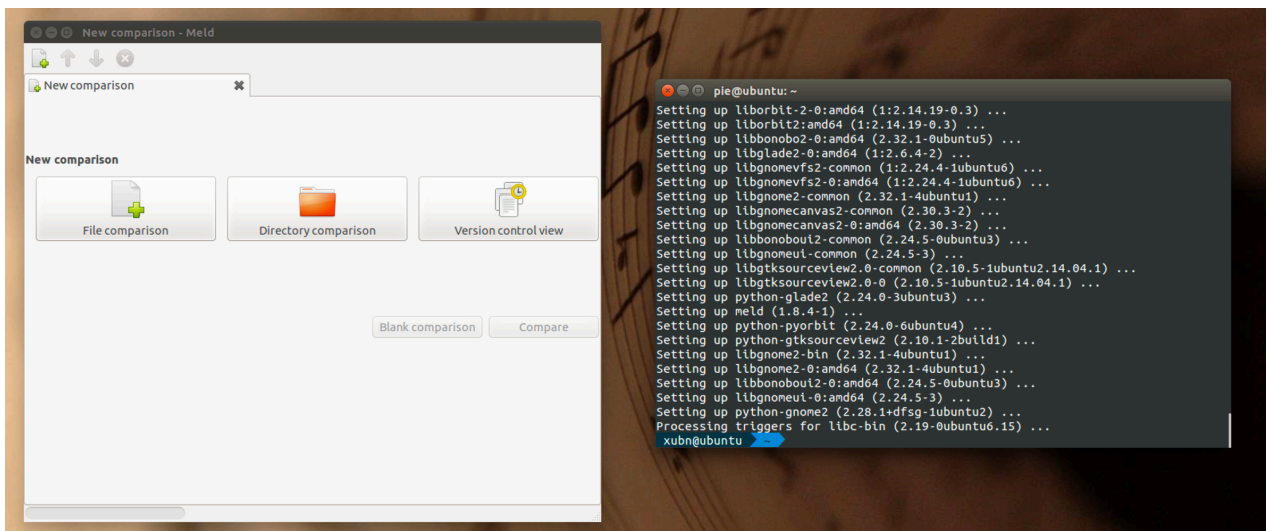
### 练习0

填写已有实验。

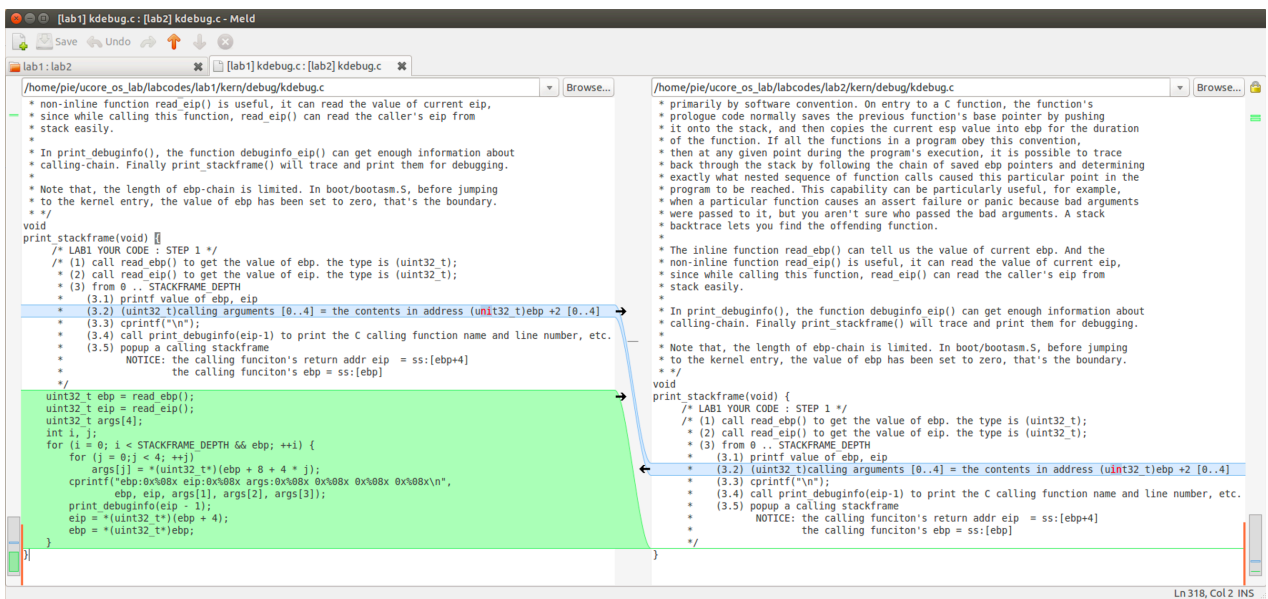
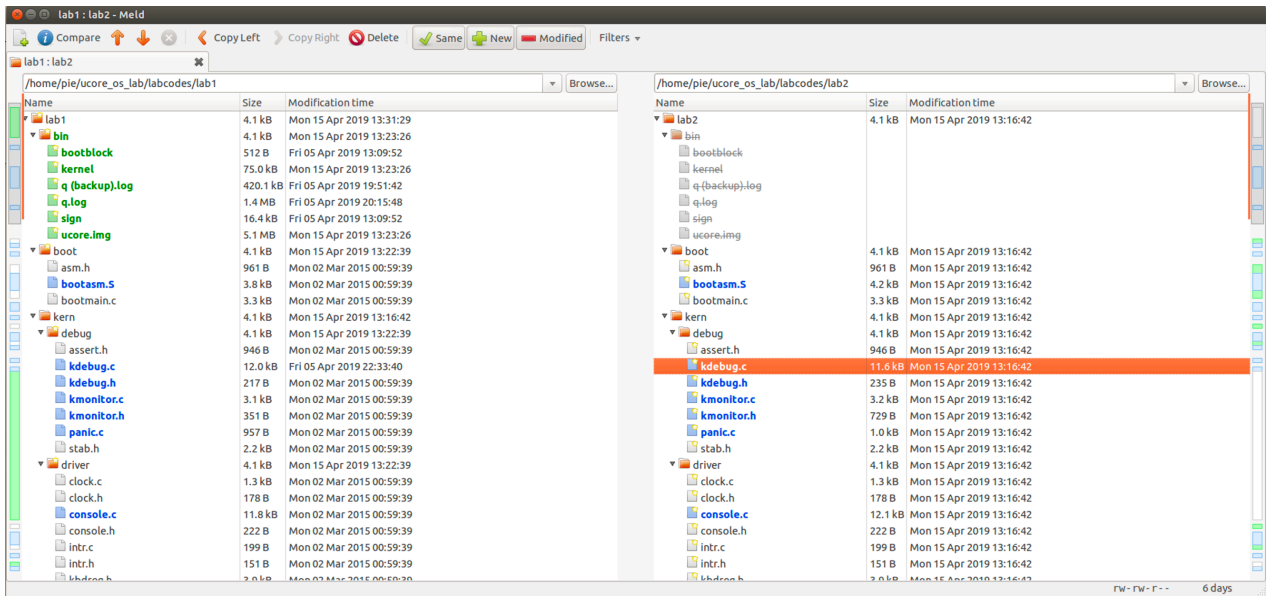
图形化应用meld是比较受欢迎的merge工具，它可以通过图形化的操作，对比两个文件之间的差别。在进行下面的实验前，需要把前面第一次UCore实验的代码合并更新到lab2代码中。这里我利用了meld来完成这项工作。

命令行安装meld：

```
1 apt-get install meld
```



对比上一次实验中完成的lab1代码目录和本次实验的源代码目录，可以看到除了一些生成文件不同，以及本次实验中新增的内容之外，在实验一中被修改过的源代码不同也被标注出来。在实验一已经完成，可以基本正常运行，符合要求的基础上，直接将这些增加的代码内容合并到lab2中。



meld工具的合并操作非常简便，它显示文件差异的形式非常直观，方便检查。同时其显示界面也是编辑界面，可以直接在上面编辑保存文件，还可以通过差异条之间的箭头来一键直接合并差异。在meld工具的帮助下，源代码改动部分的搬运内容顺利完成。

## 练习1

实现first-fit连续物理内存分配算法。

## 基本认识

first-fit的内存分配算法在查找空闲内存块的时候不需要过于复杂的查找，只要找到空间大小能满足需要的即可。但这种算法的回收函数稍复杂些，实现中需要考虑地址连续的空间块之间的合并。

根据指导课件提示：在建立空闲页块链表时，需要按照空闲页块的起始地址排序，形成一个有序的链表。主要涉及的源文件是 **default\_pmm.c**。内存管理相关文件目录位于 **/kern/mm**，进入该目录，查看几个源文件和头文件，确定基本结构和代码框架。在 **"memlayout.h"** 中，定义了基本的内存管理结构，其中比较关键的，需要进行管理的是struct Page定义的内存物理页结构：

```

1  /* *
2   * struct Page - Page descriptor structures. Each Page describes one
3   * physical page. In kern/mm/pmm.h, you can find lots of useful functions
4   * that convert Page to other data types, such as physical address.
5   * */
6  struct Page {
7      int ref;                // page frame's reference counter
8      uint32_t flags;        // array of flags that describe the
// status of the page frame
9      unsigned int property;  // the num of free block, used in
// first fit pm manager
10     list_entry_t page_link;  // free list link
11 };

```

其中包含映射此物理页的虚拟页的个数ref，描述物理页属性的类型为uint32\_t（通过typedef的int数组）的flags（其中包括是否被保留，是否空闲等信息），描述所属连续内存块大小的property，双向链接前后Page的page\_link链表。

至于需要重点关注和实现的空闲页管理，该头文件中相关的结构定义如下：

```

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

```

该结构用双向链表的方式管理空闲的页块，free\_list存放空闲块的头节点，nr\_free存放连续空闲页的数量。

在 **default\_pmm.c** 中的开篇大段注释中也有非常详细的提示和指导，不浪费篇幅贴上了，有了这些结构上的基础认识，可以在 **default\_pmm.c** 进行基本的修改了。

## 实现思路

对每个由连续的空闲页组成的内存块，用其地址最小的页管理。分配时遍历Page，查询其property，当其大小满足分配要求时即使用；对于原内存块中剩下空闲页的分割，将原来的头页从链表删除，加入分割后的新的头页。对于释放内存后的更新，释放后的合并一样通过修改链表完成。

## 初始化

文件注释中关于init函数的注释如下：

```

1  /* CALL GRAPH: `kern_init` --> `pmm_init` --> `page_init` -->
// `init_memmap` -->
2   * `pmm_manager` --> `init_memmap`.
3   * This function is used to initialize a free block (with parameter
// `addr_base`,
4   * `page_number`). In order to initialize a free block, firstly, you
// should
5   * initialize each page (defined in memlayout.h) in this free block. This

```

```

6  * procedure includes:
7  * - Setting the bit `PG_property` of `p->flags`, which means this page
   is
8  * valid. P.S. In function `pmm_init` (in pmm.c), the bit `PG_reserved`
   of
9  * `p->flags` is already set.
10 * - If this page is free and is not the first page of a free block,
11 * `p->property` should be set to 0.
12 * - If this page is free and is the first page of a free block, `p-
   >property`
13 * should be set to be the total number of pages in the block.
14 * - `p->ref` should be 0, because now `p` is free and has no reference.
15 * After that, We can use `p->page_link` to link this page into
   `free_list`.
16 * (e.g.: `list_add_before(&free_list, &(p->page_link));` )
17 * Finally, we should update the sum of the free memory blocks: `nr_free
   += n`.
18 */

```

初始状态中将每一页的ref, flags, property置为零，然后设置地址最小的那一页的相关属性，如ref标示虚拟页的数量，flags中描述帧属性，property中统计以其为头页的连续空闲内存块的大小，以此来表示每一个空闲块。修改 **default\_init\_memmap(struct Page \* base, size\_t n)** 函数，注释中的大部分操作已经完成，只需要根据上面注释第15-16行，将添加头页到空闲块链表中的操作进行修改完善即可。

```

1  static void
2  default_init_memmap(struct Page *base, size_t n) {
3      assert(n > 0);
4      struct Page *p = base;
5      for (; p != base + n; p++) {
6          assert(PageReserved(p));
7          p->flags = p->property = 0;
8          set_page_ref(p, 0);
9      }
10     base->property = n;
11     SetPageProperty(base);
12     nr_free += n;
13     list_add_before(&free_list, &(base->page_link));
14 }

```

## 内存分配

文件中关于alloc函数的注释如下：

```

1  /* (4) `default_alloc_pages`:
2   * Search for the first free block (block size >= n) in the free list
   and reszie
3   * the block found, returning the address of this block as the address
   required by
4   * `malloc`.
5   * (4.1)
6   * So you should search the free list like this:

```

```

7      *      list_entry_t le = &free_list;
8      *      while((le=list_next(le)) != &free_list) {
9      *          ...
10     *      (4.1.1)
11     *          In the while loop, get the struct `page` and check if `p-
>property`
12     *      (recording the num of free pages in this block) >= n.
13     *          struct Page *p = le2page(le, page_link);
14     *          if(p->property >= n){ ...
15     *      (4.1.2)
16     *          If we find this `p`, it means we've found a free block with
its size
17     *      >= n, whose first `n` pages can be malloced. Some flag bits of
this page
18     *      should be set as the following: `PG_reserved = 1`, `PG_property =
0`.
19     *      Then, unlink the pages from `free_list`.
20     *      (4.1.2.1)
21     *          If `p->property > n`, we should re-calculate number of
the rest
22     *      pages of this free block. (e.g.: `le2page(le,page_link))-
>property
23     *      = p->property - n;`)
24     *      (4.1.3)
25     *          Re-calucate `nr_free` (number of the the rest of all
free block).
26     *      (4.1.4)
27     *          return `p`.
28     *      (4.2)
29     *          If we can not find a free block with its size >=n, then
return NULL.
30     */

```

函数的架构非常清晰：首先搜索存放空闲块头页的链表，找到每个空闲块的头页以查询每个块的属性，当块的大小（property）满足要求（>=所需页数n）时，即为first-fit，此时设定相关的标志位信息，然后将其从空闲块头页的链表中移除，如果需要进行分割（空闲块原先页数>n），则进行相关计算，更新统计值，最后将通过first-fit方式取到的内存块的地址返回，如果没有找到符合要求的空闲块，返回的值应该为NULL。

原函数已经基本完善，根据注释，我们只需要修改找到空闲块后的数据更新即可，需要在找到空闲块判断页数符合要求时，先重新计算剩余空闲页的个数，更新空闲块头页的链表，若需要则插入分割后的余下空闲块头页，再行删除原节点：

```

1  static struct Page *
2  default_alloc_pages(size_t n) {
3      assert(n > 0);
4      // judge for enough free pages
5      if (n > nr_free) {
6          return NULL;
7      }
8      // search free page block

```

```

9      struct Page *page = NULL;
10     list_entry_t *le = &free_list;
11     while ((le = list_next(le)) != &free_list) {
12         struct Page *p = le2page(le, page_link);
13         if (p->property >= n) {
14             page = p;
15             break;
16         }
17     }
18     // judge and split
19     if (page != NULL) {
20         if (page->property > n) {
21             struct Page *p = page + n;
22             p->property = page->property - n;
23             SetPageProperty(p);
24             list_add_after(&(page->page_link), &(p->page_link));
25         }
26         list_del(&(page->page_link));
27         nr_free -= n;
28         ClearPageProperty(page);
29     }
30     return page;
31 }

```

## 内存释放

文件中关于free函数的注释如下：

```

1  /* (5) `default_free_pages`:
2   * re-link the pages into the free list, and may merge small free blocks
   * into
3   * the big ones.
4   * (5.1)
5   *     According to the base address of the withdrew blocks, search
   * the free
6   * list for its correct position (with address from low to high), and
   * insert
7   * the pages. (May use `list_next`, `le2page`, `list_add_before`)
8   * (5.2)
9   *     Reset the fields of the pages, such as `p->ref` and `p->flags`
   * (PageProperty)
10  * (5.3)
11  *     Try to merge blocks at lower or higher addresses. Notice: This
   * should
12  * change some pages' `p->property` correctly.
13  */

```

释放内存块时，需要重新将所释放的块的头页加入空闲块链表，并进行必要的合并。合并过程中，只需要检查前后有无空闲页并将自身空闲块与检测到的前后空闲块连接起来，更新连续页数量等信息。原源文件中，已经通过查询前后空闲块头页，如果发现空闲块则合并：将地址较低的空闲块头页的property更新为两者之和，清除地址较高的空闲块头页（因为此时它已经变成空闲块内页）来完成。



```

1  static void
2  default_free_pages(struct Page *base, size_t n) {
3      assert(n > 0);
4      struct Page *p = base;
5      // free the block and set relative flags
6      for (; p != base + n; p++) {
7          assert(!PageReserved(p) && !PageProperty(p));
8          p->flags = 0;
9          set_page_ref(p, 0);
10     }
11     base->property = n;
12     SetPageProperty(base);
13     // search for neiboring block and merge
14     list_entry_t *le = list_next(&free_list);
15     while (le != &free_list) {
16         p = le2page(le, page_link);
17         le = list_next(le);
18         if (base + base->property == p) {
19             base->property += p->property;
20             ClearPageProperty(p);
21             list_del(&(p->page_link));
22         }
23         else if (p + p->property == base) {
24             p->property += base->property;
25             ClearPageProperty(base);
26             base = p;
27             list_del(&(p->page_link));
28         }
29     }
30     nr_free += n;
31     // search for proper position to insert the head
32     le = list_next(&free_list);
33     while (le != &free_list) {
34         p = le2page(le, page_link);
35         if (base + base->property <= p) {
36             assert(base + base->property != p);
37             break;
38         }
39         le = list_next(le);
40     }
41     list_add_before(le, &(base->page_link));
42 }

```

### 思考改进

- 在合并空闲页的过程中，对整个列表的遍历查找来找到相邻空闲页会带来很多无效查找浪费资源，可能可以改为从所释放的空闲页开始向上向下查找，若查找过程中遇到非空闲页则停止，（向上）遇到空闲页则一直找到其所属空闲块的头页，（向下）遇到空闲页应为地址较高的相邻内存块头页，然后将新的头页和释放的头页合并，以实现连续空闲空间的合并。
- 在分配空间块后对剩余空闲空间进行分割时，容易因为这样的分割操作产生较多的外碎片，若其大小较大，可能还有机会被所需页数较少的进程利用起来，但是若其只有少数的若干页，则很难利用，产



生外碎片。与其做这样浪费时间去分割又不能加以利用，释放后还要再花时间将其合并的操作，不如索性将剩下的少数几页留作内碎片，节省分割和合并时间。所以可能可以设置一个值，当分配空间时剩下的空闲页少于某个值的时候不进行分割，当剩下的空闲页数量达到一定程度，比较有可能被利用到的时候再进行分割。

## 练习2

实现寻找虚拟地址对应的页表项。

### 基本认识

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

主要函数的前导注释如上。练习2主要针对的是页表这一重要的页式内存管理的概念。通过设置页表以及其中的对应页表项，建立虚拟内存地址和物理内存地址的对应关系。其中的关键函数是 **kern/mm/pmm.c** 中的 **get\_pte**，其负责找到一个虚拟地址对应的二级页表项的内核虚拟地址，若不存在此二级页表项，则进行分配。

根据注释指引，阅读 **kern/mm/pmm.h**，查看相关宏定义，可以了解到两个重要宏定义对地址的解析和访问原理：

```
1 /* *
2  * PADDR - takes a kernel virtual address (an address that points above
  KERNBASE),
3  * where the machine's maximum 256MB of physical memory is mapped and
  returns the
4  * corresponding physical address. It panics if you pass it a non-kernel
  virtual address.
5  * */
6 #define PADDR(kva) ({
7     uintptr_t __m_kva = (uintptr_t)(kva);
8     if (__m_kva < KERNBASE) {
9         panic("PADDR called with invalid kva %08lx", __m_kva);
10    }
11    __m_kva - KERNBASE;
12 })
13
14 /* *
15  * KADDR - takes a physical address and returns the corresponding kernel
  virtual
16  * address. It panics if you pass an invalid physical address.
17  * */
```

```

18 #define KADDR(pa) ({
19     uintptr_t __m_pa = (pa);
20     size_t __m_ppn = PPN(__m_pa);
21     if (__m_ppn >= npage) {
22         panic("KADDR called with invalid pa %08lx", __m_pa);
23     }
24     (void *) (__m_pa + KERNBASE);
25 })

```

函数中的提示性注释也能给予许多有效的信息：

```

1  /* Some Useful MACROs and DEFINES, you can use them in below
   implementation.
2  * MACROs or Functions:
3  *   PDX(la) = the index of page directory entry of VIRTUAL ADDRESS la.
4  *           访问虚拟地址页目录项的对应索引
5  *   KADDR(pa) : takes a physical address and returns the corresponding
   kernel virtual address.
6  *           访问传入的物理地址对应的内核虚拟地址
7  *   set_page_ref(page,1) : means the page be referenced by one time
8  *           标志页面被引用访问的次数
9  *   page2pa(page): get the physical address of memory which this (struct
   Page *) page manages
10 *           访问由某页进行管理的内存物理地址
11 *   struct Page * alloc_page() : allocation a page
12 *           用于当所查询的二级页表项不存在时分配页面
13 *   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.
14 * DEFINES:
15 *   PTE_P           0x001    // page table/directory entry flags bit :
   Present
16 *   物理内存页存在标志
17 *   PTE_W           0x002    // page table/directory entry flags bit :
   Writeable
18 *   物理内存页可写标志
19 *   PTE_U           0x004    // page table/directory entry flags bit :
   User can access
20 *   标志用户态下可以访问的物理内存页内容
21 */

```

## 实现思路

根据原代码中的注释内容，可以归纳出大致的操作步骤：

利用PDX(la)找到la对应的pde的虚拟地址

若PTE\_P位为0

若函数参数 bool create 为false，表示若不存在不创建二级页表，return null

否则创建二级页表

若创建失败，返回NULL

若创建成功

将page\_ref设置为1

转换该页物理地址为虚拟地址

清空页面

填表 - pde，包括所得页物理地址及其标记

找到la对应的pte的虚拟地址返回

完善实现后代码如下：

```
1  pte_t *
2  get_pte(pde_t *pgdir, uintptr_t la, bool create) {
3      pde_t *pdep = &pgdir[PDX(la)]; // find page directory entry
4      if (!(*pdep & PTE_P)) { //check if entry is not present
5          struct Page *page;
6          // check if creating is needed, then alloc page for page table
7          if (!create || !(page = alloc_page())) return NULL;
8          set_page_ref(page, 1); // set page reference
9          uintptr_t pa = page2pa(page); // get linear address of page
10         memset(KADDR(pa), 0, PGSIZE); // clear page content
11         // set page directory entry's permission
12         *pdep = pa | PTE_U | PTE_W | PTE_P;
13     }
14     // return page table entry
15     return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];
16 }
```

## 思考题

描述 Page Directory Entry 和 Page Table Entry 中每个组成部分的含义以及对UCore而言的潜在用处。

查看 `/kern/mmu.h` 可以看到相关定义：

```
1  /* page table/directory entry flags */
2  #define PTE_P      0x001 // Present
3  #define PTE_W      0x002 // Writeable
4  #define PTE_U      0x004 // User
5  #define PTE_PWT    0x008 // Write-Through
6  #define PTE_PCD    0x010 // Cache-Disable
7  #define PTE_A      0x020 // Accessed
8  #define PTE_D      0x040 // Dirty
9  #define PTE_PS     0x080 // Page Size
10 #define PTE_MBZ     0x180 // Bits must be zero
11 #define PTE_AVAIL   0xE00 // Available for software use
12 // The PTE_AVAIL bits aren't used by the kernel or interpreted by
   the
```

```

13      // hardware, so user processes are allowed to set them
    arbitrarily.
14
15 #define PTE_USER      (PTE_U | PTE_W | PTE_P)

```

各部分含义分别为：

- PTE\_P 前述已经提及，标志页表地址指向的页是否在内存中，1表示存在，0表示不存在
- PTE\_W 权限标志位，标志页目录项/页表项是否可写，1表示可写
- PTE\_U 权限标志位，标志页目录项/页表项是否可在用户态被访问，1表示可访问
- PTE\_PWT 标志是否采用既写RAM内存又写cache高速缓存的write-through写透方式，1表示采用
- PTE\_PCD 标志是否启用高速缓存，1表示启用
- PTE\_A 标志该页目录项/页表项是否正在被访问，1表示正在被访问
- PTE\_D 标志该页目录项/页表项是否被修改过，1表示修改过，所以剥夺时需要写回磁盘进行同步更新
- PTE\_PS Page Size标志，1表示该页目录项指向的是4MB的页面（仅用于页目录项）
- PTE\_MBZ 必须为0的位置
- PTE\_AVAIL 保留位，操作系统内核并不适用，用户进程可以自由设定

通过设置及读取这些标志位的内容，UCore系统内核可以直接地得知相关页块的信息，完成对应的操作和管理。

如果UCore执行过程中访问内存，出现了页访问异常，硬件要做哪些事情？

- 保存因为页错误而需要中止暂停的进程CPU现场、内存管理情况等相关信息，即保存上下文；
- 跳转到中断服务程序，执行中断处理；
- 将引起页访问异常的地址装载到寄存器CR2，触发缺页错误（14号中断）

## 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。

当一个包含某虚地址的物理内存页被释放时，其对应的管理数据结构Page需要做相应的清除处理，使得相关的资源正式被回收，物理内存页重新变成空闲，可以再次分配，相关二级页表项也应该清除。

主要的相关函数为 `kern/mm/pmm.c` 中的 `page_remove_pte` 函数：

```

1 //page_remove_pte - free an Page sturct which is related linear address
  la
2 //                  - and clean(invalidate) pte which is related linear
  address la
3 //note: PT is changed, so the TLB need to be invalidate
4 static inline void
5 page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep);
6 /* LAB2 EXERCISE 3: YOUR CODE
7  *
8  * Please check if ptep is valid, and tlb must be manually updated if
  mapping is updated

```

```

9  *
10 * Maybe you want help comment, BELOW comments can help you finish the
    code
11 *
12 * Some Useful MACROs and DEFINEs, you can use them in below
    implementation.
13 * MACROs or Functions:
14 *   struct Page *page pte2page(*ptep): get the according page from the
    value of a ptep
15 *   free_page : free a page
16 *   page_ref_dec(page) : decrease page->ref. NOTICE: ff page->ref == 0 ,
    then this page should be free.
17 *   tlb_invalidate(pde_t *pgdir, uintptr_t la) : Invalidate a TLB entry,
    but only if the page tables being
18 *                                           edited are the ones currently in use by the
    processor.
19 * DEFINEs:
20 *   PTE_P           0x001                // page table/directory
    entry flags bit : Present
21 */

```

根据其中详尽的注释，函数的具体操作流程为：

检查PTEP - PTE对应Page是否存在

找到相应的物理页面

将物理页对应的Page引用数（在实验二中的操作环节中已设置）自减

    若此时引用数已经达到0，说明该页不再被需要，释放

清空二级页表项PTE

更新TLB快表

实现代码如下：

```

1  static inline void
2  page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
3      if(*ptep & PTE_P) { // if it present
4          struct Page *page = pte2page(*ptep); // get page
5          page_ref_dec(page); // decrease page_ref
6          if(page->ref==0) free_page(page); // free the page when its
    ref==0
7          *ptep = 0; // invalidate pte
8          tlb_invalidate(pgdir, la); //flush tlb
9      }
10 }

```

## 实验结果

利用Makefile和Tool中定义的Shell脚本，进行测试：

```
xubn@ubuntu ~ j lab2
/home/pie/Desktop/Link to ucore os lab/labcodes/lab2
xubn@ubuntu labcodes/lab2 master make grade
Check PMM: (2.3s)
- check pmm: OK
- check page table: OK
- check ticks: OK
Total Score: 50/50
xubn@ubuntu labcodes/lab2 master make clean
xubn@ubuntu labcodes/lab2 master
```

## 实验总结

### 分析与区别

#### 练习1

根据注释提示修改了插入列表函数的使用、搜索位置插入释放后的空闲页表，均与参考答案（参考答案于近期从Github上重新clone下来，可能有部分更新，先前参考答案应该是将所有空闲页插入而非只插入空闲空间块的头页）一致。（原先在分配内存分割剩余内存的时候添加了一点判断，若剩余的连续空闲页，则保留为内碎片，减少因分割和后续合并带来的时间浪费。但因为这样的改动将改变功能，最后测试无法通过，所以放弃更改。）

#### 练习2

根据注释流程实现，基本一致，在看过参考答案后对原先不规范的，不合理的语法进行了调整。

#### 练习3

根据注释，与答案一致。

### 重要知识点

- 内存空间块管理中标志位的定义和使用：理论知识的学习中只学习到其存在和含义，不知其在实际的实现中其实是非常重要的页面信息来源和存储单元。是对某页进行可用性、合法性判断的重要依据。
- First-Fit的内存分配算法：从理论上很直观的找到即分配到具体的实验，具体的回收、合并的管理方式，更加细节，具体了。

### 补充知识点

- 段式内存空间管理，在本次以页式管理为主题的实验中就没有体现了。因为其不定长的特点，管理起

来应该会更加复杂；

- 更好的分配算法优化，练习1中实现了first-fit的分配算法，但我们所知该算法其实存在不少缺点和局限性，原理中还有许多更优秀的算法，当然也更加复杂了；
- 虚拟地址向实际物理地址的转换关系。

## 体会与反思

第二次的UCore实验，对比第一次更加具体细节了。从第一次的Makefile阅读分析，启动过程了解以及运行的尝试，到现在开始针对内存管理中的一些小细节，如分页管理，内存分配等问题进行研究理解，学习的过程是Top-Down，从抽象到具象，从整体到局部细节的。所以项目看起来也不会那么庞大费解了。加上老师的实验指导已经帮我们定位好相应需要修改的源文件、函数，指导书中则有更加详细的指引，且源文件中的注释都足够详尽，理解和上手起来都比较顺利，可能也因为有了上一次的实验经验，这一次走的弯路少了很多。花费的时间也更加有效了。

这一次的实验中，我们主要对物理内存管理的一些基本操作进行了实践，从段页式内存地址转换，从物理内存分配算法的初始化、分配、释放开始，到页表的管理，到物理内存的管理，结合理论课上的理论知识来理解，还是比较直观易懂的。但因为理论学习中的抽象性和概括性，真的实践起来发现好多概念已经模糊，需要重新温习，更深入地理解。一方面说明理论学习上其实有"不求甚解"的地方，另一方面也感受到确实还是要实验和理论相结合的。

因为有足够的指导、指引注释和参考资料，才完成了实验，其实自身的知识还有比如虚拟地址物理地址的映射转换等许多缺口和漏洞要补；还有许多比如要是不看指引自己很难准确定位到参考的、相关的内容和声明之类的学习能力上的不足之处。继续学习吧。