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。

实验目的如下

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

在开始实验前，别忘了将lab1中已经完成的代码填入本实验中代码有LAB1的注释的相应部分。可以采用diff和patch工具进行半自动的合并(merge)，也可以用一些图形化的比较/merge工具来合并，比如meld，eclipse中的diff/merge工具，understand中的diff/merge工具

练习0：合并Lab0和Lab1代码

使用meld工具打开文件进行比较，因为lab1只修改了trap.c，init.c和kdegub.c这三个文件，所以一一比较合并即可。

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

ucore中采用面向对象编程的思想，将物理内存管理的内容抽象成若干个特定的函数，并以结构体pmm\_manager作为物理内存管理器封装各个内存管理函数的指针，这样在管理物理内存时只需调用结构体内封装的函数，从而可将内存管理功能的具体实现与系统中其他部分隔离开。pmm\_manager中保存的函数及其功能如下所述：

|  |  |
| --- | --- |
|  | COPY  struct pmm\_manager {  const char \*name; /\*某种物理内存管理器的名称（可根据算法等具体实现的不同自定义新的内存管理器，这样也更加符合面向对象的思想）\*/  void (\*init)(void); /\*物理内存管理器初始化，包括生成内部描述和数据结构（空闲块链表和空闲页总数）\*/   void (\*init\_memmap)(struct Page \*base, size\_t n); /\*初始化空闲页，根据初始时的空闲物理内存区域将页映射到物理内存上\*/  struct Page \*(\*alloc\_pages)(size\_t n); //申请分配指定数量的物理页  void (\*free\_pages)(struct Page \*base, size\_t n); //申请释放若干指定物理页  size\_t (\*nr\_free\_pages)(void); //查询当前空闲页总数  void (\*check)(void); //检查物理内存管理器的正确性 }; |

上图为pmm.h当中定义的结构体pmm\_manager

涉及的结构体和宏定义：

memlayout.h中：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | COPY  struct Page {  int ref; // page frame's reference counter  uint32\_t flags; //描述物理页帧状态的标志位  unsigned int property; //只在空闲块内第一页中用于记录该块中页数，其他页都是0  list\_entry\_t page\_link; //空闲物理内存块双向链表 }; |

|  |  |
| --- | --- |
|  | COPY  /\* Flags describing the status of a page frame \*/ #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  #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.  #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)) |

|  |  |
| --- | --- |
|  | COPY  /\* free\_area\_t - maintains a doubly linked list to record free (unused) pages \*/ typedef struct {  list\_entry\_t free\_list; // the list header(@ with a forward ptr & a backward ptr)  unsigned int nr\_free; // # of free pages in this free list } free\_area\_t; |

list.h：

|  |  |
| --- | --- |
|  | COPY  struct list\_entry {  struct list\_entry \*prev, \*next; }; /\*list\_entry\_t是双链表结点的两个指针构成的集合，这个空闲块链表实际上是将各个块首页的指针集合（由prev和next构成）的指针（或者说指针集合所在地址）相连\*/ typedef struct list\_entry list\_entry\_t; |

练习一共需实现四个函数：

default\_init：初始化物理内存管理器；

default\_init\_memmap：初始化空闲页；

default\_alloc\_pages：申请分配指定数量的物理页；

default\_free\_pages: 申请释放若干指定物理页；

直接修改default\_pmm.c中的内存管理函数来实现。

default\_init：

|  |  |
| --- | --- |
|  | COPY  free\_area\_t free\_area; /\*allocate blank memory for the doublely linked list\*/  #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; } |

初始化双向链表，将空闲页总数nr\_free初始化为0。

default\_init\_memmap：

初始化一整个空闲物理内存块，将块内每一页对应的Page结构初始化，参数为基址和页数（因为相邻编号的页对应的Page结构在内存上是相邻的，所以可将第一个空闲物理页对应的Page结构地址作为基址，以基址+偏移量的方式访问所有空闲物理页的Page结构，根据指导书，这个空闲块链表正是将各个块首页的指针集合（由prev和next构成）的指针（或者说指针集合所在地址）相连，并以基址区分不同的连续内存物理块）。

根据注释，具体流程为：遍历块内所有空闲物理页的Page结构，将各个flags置为0以标记物理页帧有效，将property成员置零，使用 SetPageProperty宏置PG\_Property标志位来标记各个页有效（具体而言，如果一页的该位为1，则对应页应是一个空闲块的块首页；若为0，则对应页要么是一个已分配块的块首页，要么不是块中首页；另一个标志位PG\_Reserved在pmm\_init函数里已被置位，这里用于确认对应页不是被OS内核占用的保留页，因而可用于用户程序的分配和回收），清空各物理页的引用计数ref；最后再将首页Page结构的property置为块内总页数，将全局总页数nr\_free加上块内总页数，并用page\_link这个双链表结点指针集合将块首页连接到空闲块链表里。

写出代码如下：

|  |  |
| --- | --- |
|  | COPY  static void  default\_init\_memmap(struct Page \*base, size\_t n) {   assert(n > 0);  struct Page \*p = base;//块基址  for (; p != base + n; p ++)  {  assert(PageReserved(p));//确认本页不是给OS的保留页  p->flags = p->property = 0;  SetPageProperty(p);//设置标志位  set\_page\_ref(p, 0);//清空引用  }  nr\_free += n;//增加全局总页数  base->property=n;//本块首页的property设为n  list\_add\_before(&free\_list, &(base->page\_link));//首页的指针集合插入空闲页链表 } |

default\_alloc\_pages(size\_t n)：

该函数分配指定页数的连续空闲物理内存空间，返回分配的空间中第一页的Page结构的指针。

流程：从起始位置开始顺序搜索空闲块链表，找到第一个页数不小于所申请页数n的块（只需检查每个Page的property成员，在其值>=n的第一个页停下），如果这个块的页数正好等于申请的页数，则可直接分配；如果块页数比申请的页数多，要将块分成两半，将起始地址较低的一半分配出去，将起始地址较高的一半作为链表内新的块，分配完成后重新计算块内空闲页数和全局空闲页数；若遍历整个空闲链表仍找不到足够大的块，则返回NULL表示分配失败。

|  |  |
| --- | --- |
|  | COPY  static struct Page \* default\_alloc\_pages(size\_t n) {  assert(n>0);  if(n>nr\_free) return NULL;//如果所有空闲页的总数都不够，直接返回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);//链表内地址转换为Page结构指针  if (p->property >= n) {//遇到第一个页数不小于n的块  page = p;//可得块首页  break;  }  }  if (page != NULL) { //如果找到了满足条件的空闲内存块  for (struct Page \*p = page; p != (page + n); ++p)  ClearPageProperty(p); //将分配出去的内存页标记为非空闲  if (page->property > n) { /\*如果原先找到的空闲块大小大于需要的分配内存大小，进行分裂\*/ struct Page \*p = page + n; //分裂出来的新的小空闲块首页  p->property = page->property - n; //更新新的空闲块的大小信息  list\_add(&(page->page\_link), &(p->page\_link)); /\*将新空闲块插入空闲块列表中\*/ }  list\_del(&(page->page\_link)); //从链表里删除分配出去的空闲块  nr\_free -= n; //更新全局空闲页数  }  return page; } |

default\_free\_pages(struct Page \*base, size\_t n)：

释放从指定的某一物理页开始的若干个被占用的连续物理页，将这些页连回空闲块链表，重置其中的标志信息，最后进行一些碎片整理性质的块合并操作。

首先根据参数提供的块基址，遍历链表找到待插入位置，插入这些页。然后将引用计数ref、flags标志位置位，最后调用merge\_blocks函数迭代地进行块合并，以获取尽可能大的连续内存块。规则是从新插入的块开始，首先正序遍历链表，不断将链表内基址与新插入块物理地址较大一端相邻的空闲块合并到新插入块里（也是对应着分配内存块时将物理基址较大的块留在链表里）；然后反序遍历链表，不断将链表内的基址与新插入块物理地址较小一端相邻的空闲块合并到新插入块里。

代码实现如下：

|  |  |
| --- | --- |
|  | COPY  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));/\*确认各个页状态是被OS占用的或是已分配的，如果释放了空闲的内存则产生异常\*/  p->flags = 0;  SetPageProperty(p);//PG\_Property重置为空闲状态  set\_page\_ref(p, 0);  }//先对块内各页进行标志信息重置  //块首页的成员property重新赋值为n  base->property = n;   list\_entry\_t \*le = list\_next(&free\_list);//以链表头部的下一个为起始结点，开始顺序搜索  while (le != &free\_list && le < &base->page\_link) {  le = list\_next(le);  }  list\_add\_before(&free\_list, &(base->page\_link));  nr\_free += n;  while (merge\_block(base)); /\*块合并\*/  for (list\_entry\_t \*i = list\_prev(&(base->page\_link)); i!= &free\_list; i = list\_prev(i)) { //再将新插入的空闲块和其物理地址小的一段的所有相邻的物理空闲块进行合并  if (!merge\_block(le2page(i, page\_link))) break; } }  static unsigned int merge\_block(struct Page \*base){  list\_entry\_t \*le = list\_next(&(base->page\_link)); //获取链表中下一空闲块的地址  if (le == &free\_list) return 0; /\*如果当前空闲块是物理地址最大的空闲块，则无法进行向后合并，返回合并失败\*/  struct Page \*p = le2page(le, page\_link);  if (PageProperty(p) == 0) return 0;//异常情况  if (base + base->property != p) return 0; //如果两空闲块不相邻，合并终止    //合并操作  base->property += p->property;  p->property = 0;  //将合并前的物理地址较大的空闲块从链表中删去  list\_del(le);   return 1; //返回合并成功 } |

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（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？

首先需要知道PDT(页目录表),PDE(页目录项),PTT(页表),PTE(页表项)之间的关系:页表保存页表项，页表项被映射到物理内存地址；页目录表保存页目录项，页目录项映射到页表。

PDE和PTE的各部分含义及用途

PDE和PTE都是4B大小的一个元素，其高20bit被用于保存索引，低12bit用于保存属性，但是由于用处不同，内部具有细小差异，如图所示：

| bit | PDE | PTE |
| --- | --- | --- |
| 0 | Present位，0不存在，1存在下级页表 | 同 |
| 1 | Read/Write位，0只读，1可写 | 同 |
| 2 | User/Supervisor位，0则其下页表/物理页用户无法访问，1可以访问 | 同 |
| 3 | Page level Write Through，1则开启页层次的写回机制，0不开启 | 同 |
| 4 | Page level Cache Disable， 1则禁止页层次缓存，0不禁止 | 同 |
| 5 | Accessed位，1代表在地址翻译过程中曾被访问，0没有 | 同 |
| 6 | 忽略 | 脏位，判断是否有写入 |
| 7 | PS，当且仅当PS=1且CR4.PSE=1，页大小为4M，否则为4K | 如果支持 PAT 分页，间接决定这项访问的页的内存类型，否则为0 |
| 8 | 忽略 | Global 位。当 CR4.PGE 位为 1 时,该位为1则全局翻译 |
| 9 | 忽略 | 忽略 |
| 10 | 忽略 | 忽略 |
| 11 | 忽略 | 忽略 |

出现页访问异常时，硬件执行的工作

首先需要将发生错误的线性地址la保存在CR2寄存器中

这里说一下控制寄存器CR0-4的作用

CR0的0位是PE位，如果为1则启动保护模式，其余位也有自己的作用

CR1是未定义控制寄存器，留着以后用

CR2是页故障线性地址寄存器，保存最后一次出现页故障的全32位线性地址

CR3是页目录基址寄存器，保存PDT的物理地址

CR4在Pentium系列处理器中才实现，它处理的事务包括诸如何时启用虚拟8086模式等

之后需要往中断时的栈中压入EFLAGS,CS,EIP,ERROR CODE，如果这页访问异常很不巧发生在用户态，还需要先压入SS,ESP并切换到内核态

最后根据IDT表查询到对应的也访问异常的ISR，跳转过去并将剩下的部分交给软件处理。

代码的实现

为了获取PTE，很自然需要先获取PDE，通过PDX(la)可以获得对应的PDE的索引，之后在pkdir中根据索引找到对应的PDE指针，写成代码即为

|  |  |
| --- | --- |
| 1 | COPY  pde\_t \*pdep = &pgdir[PDX(la)]; |

之后判断其Present位是否为1(若不为1则需要为其创建新的PTT)，在创建时如果失败(创建标志为0或者分配物理内存页失败)需要立即返回，对应代码为

|  |  |
| --- | --- |
|  | COPY  if(!(\*pdep & PTE\_P)) {  struct Page \*temPage;   if(!create || (temPage = alloc\_page()) == NULL )  {  return NULL;  }   } |

分配物理内存页之后，把这个物理页的引用的计数通过set\_page\_ref更新。

获取这个temPage的物理地址，然后把这个物理页全部初始化为0(需要用KADDR转换为内核虚拟地址)。到此完成了pte的设置，还需要更新pde中的几个标识位。对应代码如下

|  |  |
| --- | --- |
|  | COPY  set\_page\_ref(temPage,1); uintptr\_t pa = page2pa(temPage); memset(KADDR(pa), 0, PGSIZE); // pa的地址，用户级，可写，存在 \*pdep = pa | PTE\_U | PTE\_W | PTE\_P; |

最后需要返回这个对应的PTE，而目前我们有的是一个pdep，存放着PTT的物理地址，根据PDE的结构，可以得出这样一个流程

抹去低12位，只保留对应的PTT的起始基地址

用PTX(la)获得PTT对应的PTE的索引

用数组和对应的索引得到PTE并返回

|  |  |
| --- | --- |
|  | COPY  return &((pte\_t \*)KADDR(PDE\_ADDR(\*pdep)))[PTX(la)]; |

完整代码如下：

|  |
| --- |
| COPY  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    pde\_t \*pdep = &pgdir[PDX(la)];    if(!(\*pdep & PTE\_P))  {  struct Page \*temPage;  if(!create || (temPage = alloc\_page()) == NULL)  {  return NULL;  }  set\_page\_ref(temPage,1);  uintptr\_t pa = page2pa(temPage);  memset(KADDR(pa),0,PGSIZE);  \*pdep = pa | PTE\_U | PTE\_W | PTE\_P;   }   return &((pte\_t \*)KADDR(PDE\_ADDR(\*pdep)))[PTX(la)]; } |

la,PDX,PTX,PTE\_ADDR,PDE\_ADDR的定义可以在mmu.h下看到，具体内容如下

|  |
| --- |
| COPY  // A linear address 'la' has a three-part structure as follows: // // +--------10------+-------10-------+---------12----------+ // | Page Directory | Page Table | Offset within Page | // | Index | Index | | // +----------------+----------------+---------------------+ // \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/ // \----------- PPN(la) -----------/ // // The PDX, PTX, PGOFF, and PPN macros decompose linear addresses as shown. // To construct a linear address la from PDX(la), PTX(la), and PGOFF(la), // use PGADDR(PDX(la), PTX(la), PGOFF(la)).  // page directory index #define PDX(la) ((((uintptr\_t)(la)) >> PDXSHIFT) & 0x3FF)  // page table index #define PTX(la) ((((uintptr\_t)(la)) >> PTXSHIFT) & 0x3FF)  // offset in page #define PGOFF(la) (((uintptr\_t)(la)) & 0xFFF)  // address in page table or page directory entry #define PTE\_ADDR(pte) ((uintptr\_t)(pte) & ~0xFFF) #define PDE\_ADDR(pde) PTE\_ADDR(pde) |

其高10位用作PDT的索引，中10位用作PTT的索引，末12位用作物理页偏移量

KADDR，page2pa可以在pmm.h中看到

|  |
| --- |
| COPY  /\* \*  \* 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); \  })  static inline ppn\_t page2ppn(struct Page \*page) {  return page - pages; } // PGSHIFT = log2(PGSIZE) = 12 static inline uintptr\_t page2pa(struct Page \*page) {  return page2ppn(page) << PGSHIFT; } |

遇到的坑

PDE,PDT,PTT,PTE真的很容易让人弄混，而且他们的发音真的很像，一不留神就弄混了。搞清楚关系和结构看一下注释就能很快地搞定。

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。

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

关于代码：

练习3可以看成是练习2的简单的逆过程，在理解了练习2的基础上，释放虚地址所在的页和取消对应二级页表项的映射可以大概分为如下步骤：

通过PTE\_P判断该ptep是否存在；

判断Page的ref的值是否为0，若为0，则说明此时没有任何逻辑地址被映射到此物理地址，换句话说当前物理页已没人使用，因此调用free\_page函数回收此物理页，使得该物理页空闲；若不为0，则说明此时仍有至少一个逻辑地址被映射到此物理地址，因此不需回收此物理页；

把表示虚地址与物理地址对应关系的二级页表项清除；

更新TLB；

根据Lab2中的代码提示，我们将MACROs 和 DEFINES拿出进行理解：

|  |
| --- |
| COPY  static inline struct Page \* pte2page(pte\_t pte) {//从ptep值中获取相应的页面  if (!(pte & PTE\_P)) {  panic("pte2page called with invalid pte");  }  return pa2page(PTE\_ADDR(pte)); }  //减少该页的引用次数，返回剩下的引用次数。 static inline int  page\_ref\_dec(struct Page \*page) {  page->ref -= 1;  return page->ref; }  //当修改的页表目前正在被进程使用时，使之无效。 void  tlb\_invalidate(pde\_t \*pgdir, uintptr\_t la) {  if (rcr3() == PADDR(pgdir)) {  invlpg((void \*)la);  } }    #define PTE\_P 0x001 // Present，即最低位为1； |

在此基础之上，我们给出完整的代码：

|  |
| --- |
| COPY  static inline void page\_remove\_pte(pde\_t \*pgdir, uintptr\_t la, pte\_t \*ptep) {  if (\*ptep & PTE\_P) {  // 如果对应的二级页表项存在，如果\*ptep存在，则其与PTE\_P相与，得到的是  struct Page \*page = pte2page(\*ptep); // 获得\*ptep对应的Page结构  // 关联的page引用数自减1，page\_ref\_dec的返回值是现在的page->ref  if(page\_ref\_dec(page) == 0) {  // 如果自减1后，引用数为0，需要free释放掉该物理页  free\_page(page);  }  // 把表示虚地址与物理地址对应关系的二级页表项清除(通过把整体设置为0)  \*ptep = 0;  // 由于页表项发生了改变，需要使TLB快表无效  tlb\_invalidate(pgdir, la);  }   } |

关于问题：

Q1：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？

A：所有的物理页都有一个描述它的Page结构，所有的页表都是通过alloc\_page()分配的，每个页表项都存放在一个Page结构描述的物理页中；如果 PTE 指向某物理页，同时也有一个Page结构描述这个物理页。所以有两种对应关系：

(1)可以通过 PTE 的地址计算其所在的页表的Page结构：

将虚拟地址向下对齐到页大小，换算成物理地址(减 KERNBASE), 再将其右移 PGSHIFT(12)位获得在pages数组中的索引PPN，&pages[PPN]就是所求的Page结构地址。

(2)可以通过 PTE 指向的物理地址计算出该物理页对应的Page结构：

PTE 按位与 0xFFF获得其指向页的物理地址，再右移 PGSHIFT(12)位获得在pages数组中的索引PPN，&pages[PPN]就 PTE 指向的地址对应的Page结构。

Q2：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。

A：此时虚拟地址和物理地址的映射关系是：phy addr + KERNBASE = virtual addr

所以如果想让虚拟地址=物理地址，则只要让KERNBASE = 0，因此去修改memlayout.h的宏定义

|  |  |
| --- | --- |
|  | COPY  #define KERNBASE 0xC0000000 |