

lab2 .....	2
物理内存管理 .....	3
Exercise 1 .....	4
实现 boot_alloc 函数 .....	5
丰富 mem_init 函数 .....	6
实现 page_init 函数 .....	6
实现 page_alloc 函数 .....	9
实现 page_free 函数 .....	9
虚拟内存管理/页表映射管理 .....	11
Exercise 2 .....	12
Exercise 3 : .....	13
Question : .....	14
Exercise 4: .....	14
完成 pgdir_walk 函数 .....	15
完成 boot_map_region 函数 .....	16
实现 page_lookup 函数 .....	16
实现 page_remove 函数 .....	17
实现 page_insert 函数 .....	17
内核的地址空间 .....	18
Exercise 5 .....	19
丰富 mem_int 函数，初始化虚拟内存 .....	19
Question .....	20
总结 : .....	21

## lab2

第一部分讲的是物理内存管理，要进行内存管理首先需要知道哪些物理内存是空闲的，哪些是被使用的。还需要实现一些函数对这些物理内存进行管理。

第二部分讲的是虚拟内存。一个虚拟地址如何被映射到物理地址，将实现一些函数来操作页目录和页表从而达到映射的目的。

第三部分讲的是内核的地址空间。将结合第一部分和第二部分的成果，来对内核地址空间进行映射。

lab2 包括了新的源文件：

- `inc/memlayout.h`
- `kern/pmap.c`
- `kern/pmap.h`
- `kern/kclock.h`
- `kern/kclock.c`

`memlayout.h` 描述了必须通过修改 `pmap.c` 来实现虚拟地址空间的布局。

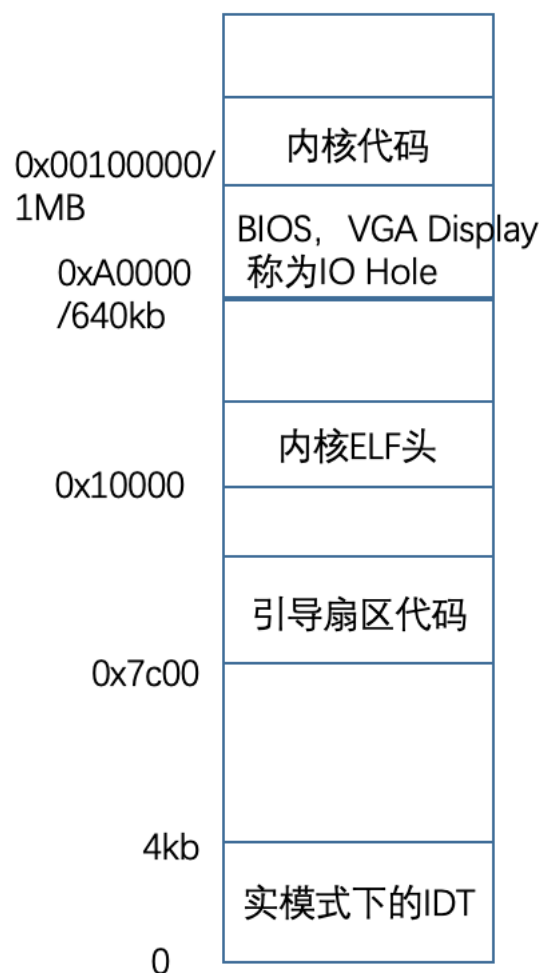
`memlayout.h` 和 `pmap.h` 定义了 `PageInfo` 结构，您将使用它来跟踪哪些物理内存页是空闲的。

`kclock.c` 和 `kclock.h` 操作 PC 机的电池支持的时钟和 CMOS RAM 硬件，其中 BIOS 记录 PC 机包含的物理内存，以及其他内容。

注意：

要特别注意 memlayout.h 和 pmap.h，因为这个 lab 要求你使用和理解它们包含的许多定义。可能也要看 inc/mmu.h，因为它也包含了一些对这个 lab 有用的定义

## 物理内存管理



前一个 lab 中，我们了解到内存的分布大概如下：

- 1, 0x00000 - 0xA0000: 这部分叫做 basemem，是可用的。
- 2, 0xA0000 - 0x00100000 : 这部分是 IO Hole，是不可用的。

3, 0x00100000 - 往上 : 这边是 extmem, 可用。

在 pmap.c 文件中, 定义了全局变量 npages 和 npages\_basemem。

```
// These variables are set by i386_detect_memory()
size_t npages;           // Amount of physical memory (in pages)
static size_t npages_basemem; // Amount of base memory (in pages)
```

npages: 表示所有物理内存的页数。

npages\_basemem: 所有 basemem 部分可用的内存的页数。

## Exercise 1

**Exercise 1.** In the file kern/pmap.c, you must implement code for the following functions (probably in the order given).

```
boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()
```

check\_page\_free\_list() and check\_page\_alloc() test your physical page allocator. You should boot JOS and see whether check\_page\_alloc() reports success. Fix your code so that it passes. You may find it helpful to add your own assert()s to verify that your assumptions are correct.

处理 pmap.c 文件中的一些子函数。

通过观察 pmap.c 文件, 这是内存管理的文件, 主要管理物理内存和虚拟内存。

主函数为 mem\_init() 函数。

在 lab1 中, 进入内核后, 调用的第一个函数是 i386\_init() 函数, 该函数会调用 mem\_init() 函数。下面分析该函数的调用子函数初始化过程。

- 1) 首先, 调用 i386\_detect\_memory() 函数, 这个函数检测现在系统中还有多少可用的内存空间, 并初始化 npages 和 npages\_basemem。
- 2) 接着调用 boot\_alloc() 函数, 这边这个函数需要添加代码:

## 实现 boot\_alloc 函数

```
static void *
boot_alloc(uint32_t n)
{
    static char *nextfree; // virtual address of next byte of free memory
    char *result;

    // Initialize nextfree if this is the first time.
    // 'end' is a magic symbol automatically generated by the linker,
    // which points to the end of the kernel's bss segment:
    // the first virtual address that the linker did *not* assign
    // to any kernel code or global variables.
    if (!nextfree) {
        extern char end[];
        nextfree = ROUNDUP((char *) end, PGSIZE);
    }

    // Allocate a chunk large enough to hold 'n' bytes, then update
    // nextfree. Make sure nextfree is kept aligned
    // to a multiple of PGSIZE.
    //
    // LAB 2: Your code here.
    return NULL;
}
```

申请 n 个字节的内存空间。

添加代码如下：

```
// Allocate a chunk large enough to hold 'n' bytes, then update
// nextfree. Make sure nextfree is kept aligned
// to a multiple of PGSIZE.
//
// LAB 2: Your code here.
result=nextfree;
nextfree=ROUNDUP((char *)result+n,PGSIZE); //change nextfree to next free place

//ensure have free place to allocate
if((uint32_t)nextfree-KERNBASE > (npages*PGSIZE)){
    panic("out of memory!\n");
}
return result;
```

- 3) 依靠 boot\_alloc ( ) 函数是用来建立初始化的页目录：

```
////////////////////////////////////
// create initial page directory.
kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
memset(kern_pgdir, 0, PGSIZE);
```

kern\_pgdir 指针指向分配的一页的内存，该内存存储页目录表，分配大小为 PGSIZE。并且初始化为 0。**这个页紧跟在内核之后。**

- 4) 下一个命令：

```
// Permissions: kernel R, user R
kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
```

该语句的作用是，为页目录表中添加第一个页目录表项，

```

*          | Cur. Page Table (User R-) | R-/R- PTSIZE
*   UVPT   -----> +-----+ 0xef400000
*          |          RO  PAGES       | R-/R- PTSIZE
*   UPAGES  -----> +-----+ 0xef000000
*          |          RO  ENVS        | R-/R- PTSIZE
*   UTOP,UENVS -----> +-----+ 0xeec00000
*   UXSTACKTOP -/      | User Exception Stack | RW/RW PGSIZE
*                      +-----+ 0xeebfff00

```

可以看到 UVPT 定义的一段虚拟地址，首位置是 0xef400000，这个虚拟地址就是存放的就是 kern\_pgdir，将其与 kern\_pgdir 的物理地址映射起来，使用 PADDR() 函数获取到 kern\_pgdir 的真实物理地址。

- 5) 下一个任务是需要分配一块内存，内存中存放 struct PageInfo 的数组，每个 PageInfo 代表是内存中的一个页。根据这个数组来追踪内存的使用情况的。

## 丰富 mem\_init 函数

添加如下：

```

////////////////////////////////////
// Allocate an array of npages 'struct PageInfo's and store it in 'pages'.
// The kernel uses this array to keep track of physical pages: for
// each physical page, there is a corresponding struct PageInfo in this
// array. 'npages' is the number of physical pages in memory. Use memset
// to initialize all fields of each struct PageInfo to 0.
// Your code goes here:
pages=(struct PageInfo*) boot_alloc(npages * sizeof(struct PageInfo));
memset(pages,0,sizeof(struct PageInfo) * npages);

```

- 6) 接着调用 page\_init() 函数，这个函数的目的是初始化 pages 数组，初始化 pages\_free\_list 链表，用来保存空闲页的信息。

## 实现 page\_init 函数

需要去实现这个函数：

```

void
page_init(void)
{
    // The example code here marks all physical pages as free.
    // However this is not truly the case. What memory is free?
    // 1) Mark physical page 0 as in use.
    // This way we preserve the real-mode IDT and BIOS structures
    // in case we ever need them. (Currently we don't, but...)
    // 2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
    // is free.
    // 3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
    // never be allocated.
    // 4) Then extended memory [EXTPHYSMEM, ...).
    // Some of it is in use, some is free. Where is the kernel
    // in physical memory? Which pages are already in use for
    // page tables and other data structures?
    //
    // Change the code to reflect this.
    // NB: DO NOT actually touch the physical memory corresponding to
    // free pages!
    size_t i;
    for (i = 0; i < npages; i++) {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}

```

从注释可以知道哪些是已分配使用的，哪些是未使用的。该函数通过 for 循环，遍历所有 PageInfo 结构体，去修改结构的状态，如果 pp\_ref 的值为 1，则表示该页已分配使用；否则，则是未分配。

- A) page 0 是已经分配使用的。
- B) 其他的 basemem，也就是 [PGSIZE, npages\_basemem \* PGSIZE] 区域是空闲的。
- C) IO hole 是分配出去的，可以发现这部分页的大小为  $(1024-640)/4$ ，一共 96 个页。
- D) 扩展的内存，一些是使用的，一些是未使用的。

添加代码如下：

```

void
page_init(void)
{
    // The example code here marks all physical pages as free.
    // However this is not truly the case. What memory is free?
    // 1) Mark physical page 0 as in use.
    //    This way we preserve the real-mode IDT and BIOS structures
    //    in case we ever need them. (Currently we don't, but...)
    // 2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
    //    is free.
    // 3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
    //    never be allocated.
    // 4) Then extended memory [EXTPHYSMEM, ...).
    //    Some of it is in use, some is free. Where is the kernel
    //    in physical memory? Which pages are already in use for
    //    page tables and other data structures?
    //
    // Change the code to reflect this.
    // NB: DO NOT actually touch the physical memory corresponding to
    // free pages!
    size_t i;
    page_free_list=NULL;
    //extmem counts of in use ; boot_alloc(0) means the first unused place address
    int numAlloc=((uint32_t) boot_alloc(0) - KERNBASE) / PGSIZE;
    //IO hole counts
    int numIo=96; // 1024-640 / 4 ;
    for (i = 0; i < npages; i++) {
        if(i==0){
            pages[i].pp_ref=1;
        }
        else if(i>=npages_basemem && i<npages_basemem+numIo+numAlloc){
            pages[i].pp_ref=1;
        }else{
            pages[i].pp_ref = 0;
            pages[i].pp_link = page_free_list;
            page_free_list = &pages[i];
        }
    }
}

```

7) 初始化完一系列操作后，接着就是检查操作。

```

check_page_free_list(1);
check_page_alloc();
check_page();

```

首先是 `check_page_free_list(1)` 这个函数的功能就是检查 `page_free_list` 链表的所谓空闲页，是否真的都是合法的，空闲的。当输入参数为 1 时，这个函数要在检查前先进行一步额外的操作，对空闲页链表 `free_page_list` 进行修改，经过 `page_init`，`free_page_list` 中已经存放了所有的空闲页表，但是他们的顺序是按照页表的编号从大到小排列的。当前操作系统所采用的页目录表 `entry_pgdir`（不是 `kern_pgdir`）中，并没有对大编号的页表进行映射，所以这部分页表我们还不能操作。但是小编号的页表，即从 0 号页表开始到 1023 号页表，已经映射过了，所以可以对这部分页表进行操作。那么 `check_page_free_list(1)` 要完成的就是把这部分页表对应的 `PageInfo` 结构体移动到 `free_page_list` 的前端，供操作系统现在使用。剩下的就是对 `page_free_list` 进行检查。



接着执行的操作是 `check_page__alloc()` 函数，这个函数去检查 `page_alloc()` 函数和 `page_free()` 函数是否合法。

## 实现 `page_alloc` 函数

这个 `page_alloc()` 函数需要添加修改：

```
struct PageInfo *
page_alloc(int alloc_flags)
{
    // Fill this function in
    return 0;
}
```

首先取出一个空闲页，再调整空闲页链表的表头，最后初始化拿出来的空闲页。

```
struct PageInfo *
page_alloc(int alloc_flags)
{
    // Fill this function in
    struct PageInfo* result;
    if(page_free_list==NULL){
        return NULL;
    }
    result=page_free_list;
    page_free_list=result->pp_link;
    result->pp_link=NULL;

    if(alloc_flag & ALOC_ZERO){
        memset(page2kva(result),0,PGSIZE);
    }
    return result;
}
```

## 实现 `page_free` 函数

`page_free()` 函数也需要添加：

```
void
page_free(struct PageInfo *pp)
{
    // Fill this function in
    // Hint: You may want to panic if pp->pp_ref is nonzero or
    // pp->pp_link is not NULL.
}
```

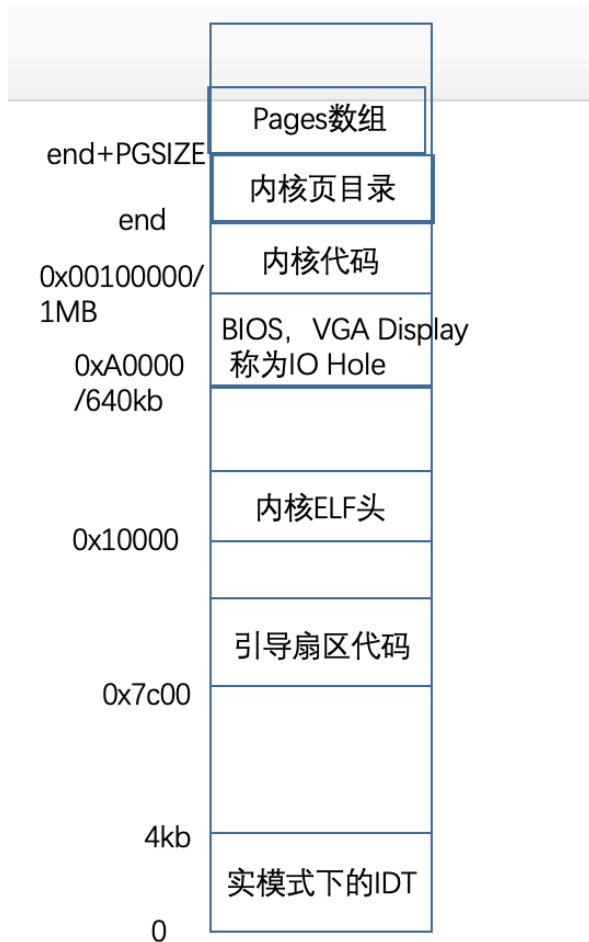
首先修改 `pp` 的相关信息，接着接在空闲链表头部。

```
void
page_free(struct PageInfo *pp)
{
    // Fill this function in
    // Hint: You may want to panic if pp->pp_ref is nonzero or
    // pp->pp_link is not NULL.
    assert(pp->pp_ref==0);
    assert(pp->pp_link==NULL);

    pp->pp_link=page_free_list;
    page_free_list=pp;
}
```

至此：mem\_init（）函数中的物理内存管理方面的内容，暂时结束。

经过上述内存分配后，物理内存的布局如下：



```
// Initialize nextfree if this is the first time.
// 'end' is a magic symbol automatically generated by the linker,
// which points to the end of the kernel's bss segment:
// the first virtual address that the linker did *not* assign
// to any kernel code or global variables.
if (!nextfree) {
    extern char end[];
    nextfree = ROUNDUP((char *) end, PGSIZE);
}
```

end 的定义看注释！

## 虚拟内存管理/页表映射管理

## Exercise 2

**Exercise 2.** Look at chapters 5 and 6 of the [Intel 80386 Reference Manual](#), if you haven't done so already. Read the sections about page translation and page-based protection closely (5.2 and 6.4). We recommend that you also skim the sections about segmentation; while JOS uses the paging hardware for virtual memory and protection, segment translation and segment-based protection cannot be disabled on the x86, so you will need a basic understanding of it.

这个练习是需要了解 X86 保护模式下的，分段和分页机制。  
查看 intel 80386 用户手册。

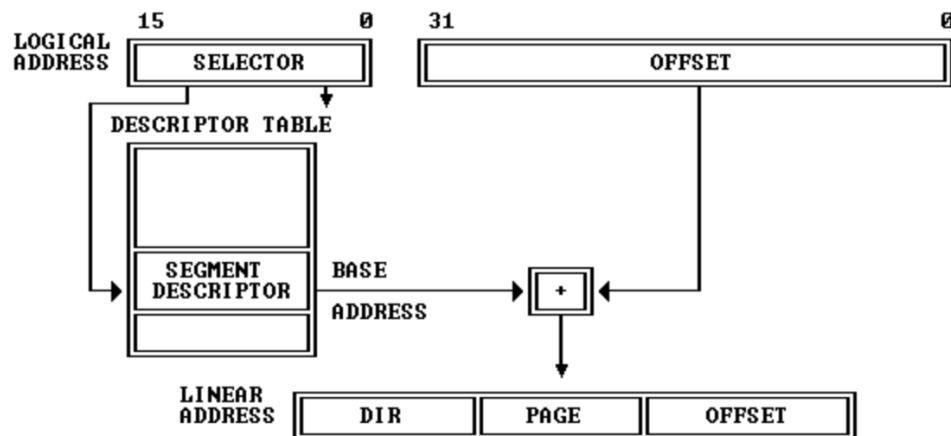
知识点：

分段机制：

段的转换主要包括下面 4 种数据结构：

- Descriptors
- Descriptor tables
- Selectors
- Segment Registers

Figure 5-2. Segment Translation

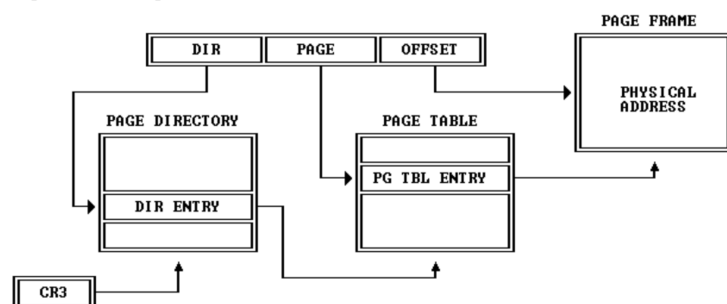


具体细节看手册！

分页机制：

CR3 控制寄存器中存放的是页目录的物理地址。

Figure 5-9. Page Translation



整个机制是两级页表机制，页目录可以包含 1k 个页表；每个页表可以包含 1k 个页。因此可以覆盖  $1k \times 1k = 1M$  个页。 $2^{20}$  次方，又每个每个页占 4k，则整个就占  $2^{32}$ 。

### Exercise 3 :

**Exercise 3.** While GDB can only access QEMU's memory by virtual address, it's often useful to be able to inspect physical memory while setting up virtual memory. Review the QEMU [monitor commands](#) from the lab tools guide, especially the `xp` command, which lets you inspect physical memory. To access the QEMU monitor, press `Ctrl-a c` in the terminal (the same binding returns to the serial console).

Use the `xp` command in the QEMU monitor and the `x` command in GDB to inspect memory at corresponding physical and virtual addresses and make sure you see the same data.

Our patched version of QEMU provides an `info pg` command that may also prove useful: it shows a compact but detailed representation of the current page tables, including all mapped memory ranges, permissions, and flags. Stock QEMU also provides an `info mem` command that shows an overview of which ranges of virtual addresses are mapped and with what permissions.

该练习主要了解一下 qemu 的指令，观察物理地址中存放的数据。

通过 GDB，我们只能通过虚拟地址来查看内存所存放的内容，但是如果我们能够访问物理内存的话，肯定会更有帮助的。我们可以看一下 QEMU 中的一些常用指令，特别是 `xp` 指令，可以允许我们去访问物理内存地址。

打开 monitor 后，我们可以输入如下比较常见的指令：

`xp/Nx paddr` -- 查看 paddr 物理地址处开始的，N 个字的 16 进制的表示结果。

`info registers` -- 展示所有内部寄存器的状态。

`info mem` -- 展示所有已经被页表映射的虚拟地址空间，以及它们的访问优先级。

info pg -- 展示当前页表的结构。

Question :

To summarize:

C type	Address type
T*	Virtual
uintptr_t	Virtual
physaddr_t	Physical

**Question**

1. Assuming that the following JOS kernel code is correct, what type should variable x have, uintptr\_t Or physaddr\_t?

```
mystery_t x;  
char* value = return_a_pointer();  
*value = 10;  
x = (mystery_t) value;
```

value 是一个虚拟地址，所以这边应该是 uintptr\_t 类型！

Exercise 4:

**Exercise 4.** In the file kern/pmap.c, you must implement code for the following functions.

```
pgdir_walk()  
boot_map_region()  
page_lookup()  
page_remove()  
page_insert()
```

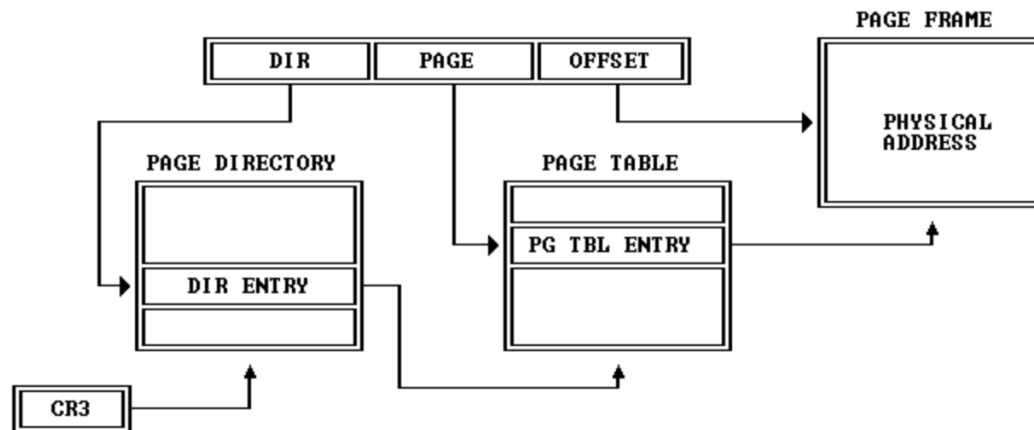
check\_page(), called from mem\_init(), tests your page table management routines. You should make sure it reports success before proceeding.

**要注意理解 mmu.h 文件中的信息！**

练习 4 是关于页表的管理，包括插入和删除 线性地址和物理地址映射关系，创建页表等操作。

页转换机制：

Figure 5-9. Page Translation



完成 pgdir\_walk 函数

该函数的作用：给定指向页目录的指针 pgdir，该函数返回指向线性地址 “ va” 的页表 entry（PTE）的指针。

这需要遍历两级页面表结构。

**Tips:** 如果出现多个虚拟地址映射到同样的一块物理地址，则物理页的 pp\_ref 的引用计数加 1.

```

pte_t *
pgdir_walk(pte_t *pgdir, const void *va, int create)
{
    // Fill this function in
    //获取到线性地址的 页目录index 和 页表index
    //使用连个指针指向页目录和页表
    pde_t* pde=NULL;
    pte_t* pte=NULL;

    struct PageInfo* pp;

    //获取到va地址所对应的页表,使用PDX(va)获取到页目录index 和PTX(va)获取到页表index
    pde=&pgdir[PDX(va)];
    if(*pde & PTE_P) //判断页目录是否有效
    {
        pte=(KADDR(PTE_ADDR(*pde))); //KADDR将物理地址转变为虚拟地址
    }else{
        if(!create){
            return NULL;
        }
        if(!(pp=page_alloc(ALLOC_ZERO))){
            return NULL;
        }
        pte=(pte_t *)page2kva(pp); //设置指向页表的指针
        pp->pp_ref++; //分配页的引用计数加一
        *pde=PADDR(pte) | PTE_P | PTE_W | PTE_U; //设置页目录项, pte是虚拟地址, 则将其装变为物理地址。
    }
    return &pte[PTX(va)]; //最终返回线性地址所对应的页表项指针
}

```

## 完成 boot\_map\_region 函数

该函数将[va,va+size]对应的虚拟地址空间映射到物理地址空间[pa,pa+size]。  
va 和 pa 是页对齐的。

Perm 相当于是权限位！

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
    // Fill this function in
    uint32_t pgNum=size/PGSIZE;
    if((size % PGSIZE)!=0){
        pgNum++;
    }
    //计算size所占的页数
    for(uint32_t i=0;i<pgNum;++i){
        pte_t *pte=pgdir_walk(pgdir,(void*) va,1);
        if(pte==NULL){
            panic("out of memory!");
        }
        *pte= pa | PTE_P | perm; //修改va对应的PTE的值
        va+=PGSIZE;
        pa+=PGSIZE;
    }
}
```

## 实现 page\_lookup 函数

该函数的作用是返回虚拟地址 va 映射的物理地址所在页面信息。

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // Fill this function in
    struct PageInfo* pp=NULL;

    pte_t* pte=pgdir_walk(pgdir,va,0);
    if(pte==NULL){
        return NULL;
    }
    if(!(*pte & PTE_P)){
        return NULL;
    }
    pp=pa2page(PTE_ADDR(*pte)); //将xpte转变为物理地址，通过pa2page转变为该地址下的page指针

    if(pte_store!=NULL){
        *pte_store=pte;
    }

    return pp;
}
```



## 实现 page\_remove 函数

该函数的作用是将虚拟地址 va 和物理页的映射关系删除。

- 1) pp\_ref 值要减一。
- 2) 如果 pp\_ref 减为 0，要把这个页回收。
- 3) 这个页对应的页表项应该置为 0

```
void
page_remove(pde_t *pgdir, void *va)
{
    // Fill this function in
    pte_t* pte;
    struct PageInfo* pp=page_lookup(pgdir,va,&pte);
    if(pp==NULL){
        return;
    }
    page_decref(pp);    //减少引用计数
    tlb_invalidate(pgdir,va);    //失效 TLB缓存
    *pte=0;            //将pte清空。
}
```

## 实现 page\_insert 函数

该函数实现了将物理内存页 pp 和 va 进行一个映射关系，

- 1) 通过 pgdir\_walk 函数获得 va 对应页表项指针
- 2) 将 pp 页的引用计数加 1
- 3) 如果该 va 已经映射过，则需要重置它。
- 4) 将 pp 的物理地址拿到
- 5) 修改 pte 指针内容，将 va 与物理页的映射关系加入到页表项中。

```

int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    // Fill this function in
    pte_t *pte=pgdir_walk(pgdir,va,1);
    if(pte==NULL){
        return -E_NO_MEM;
    }
    pp->pp_ref++;

    if((*pte) & PTE_P){ //如果虚拟地址va已经被映射过了，需要对其先释放
        page_remove(pgdir,va);
    }
    physaddr_t pa=page2pa(pp); //获得pp的物理地址。
    *pte= pa | perm | PTE_P; //修改pte

    pgdir[PDX(va)] |=perm;

    return 0;
}

```

## 内核的地址空间

JOS 将虚拟地址划分为两个部分，其中用户环境占据低地址部分，叫做用户地址空间；操作系统内核总是占据高地址部分，叫做内核地址空间。这两部分的分界线定义在 memlayout.h 文件中的一个宏 ULIM 处，JOS 为内核空间保留了接近 256MB 的虚拟地址空间。

由于内核和用户进程只能访问各自的地址空间，所以我們必須在 x86 页表中使用访问权限位(Permission Bits)来使用户进程的代码只能访问用户地址空间，而不是内核地址空间。否则用户代码中的一些错误可能会覆写内核中的数据，最终导致内核的崩溃。

处在用户地址空间中的代码不能访问高于 ULIM 的地址空间，但是内核可以读写这部分空间。而内核和用户对于地址范围[UTOP, ULIM]有着相同的访问权限，那就是可以读取但是不可以写入。这一个部分的地址空间通常被用于把一些只读的内核数据结构暴露给用户地址空间的代码。在 UTOP 之下的地址范围是给用户进程使用的，用户进程可以访问，修改这部分地址空间的内容。

## Exercise 5

**Exercise 5.** Fill in the missing code in `mem_init()` after the call to `check_page()`.

Your code should now pass the `check_kern_pgdir()` and `check_page_installed_pgdir()` checks.

### 丰富 `mem_init` 函数，初始化虚拟内存

将 `pages` 对应的数组映射到 `UPAGES`，大小为 `PTSIZE`；`PADDR` 函数将虚拟地址转换为物理地址，指针都是虚拟地址。因为内核空间 and 用户空间都可以访问，所以这边权限位为 `PTE_U`。

```
////////////////////////////////////  
// Now we set up virtual memory  
  
////////////////////////////////////  
// Map 'pages' read-only by the user at linear address UPAGES  
// Permissions:  
//   - the new image at UPAGES -- kernel R, user R  
//   (ie. perm = PTE_U | PTE_P)  
//   - pages itself -- kernel RW, user NONE  
// Your code goes here:  
boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);
```

这边是映射内核的堆栈区域，把从 `bootstack` 变量所标记的物理地址范围映射到堆栈中。内核的堆栈区域包括 `[KSTACKTOP-PTSIZE, KSTACKTOP)`，不过要把这个范围划分成两部分：

- \* `[KSTACKTOP-KSTKSIZE, KSTACKTOP)` 这部分映射关系加入的页表中。

- \* `[KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE]` 这部分不进行映射。

因为这边只能内核访问，用户空间不可以访问，所以设置为 `PTE_W`。

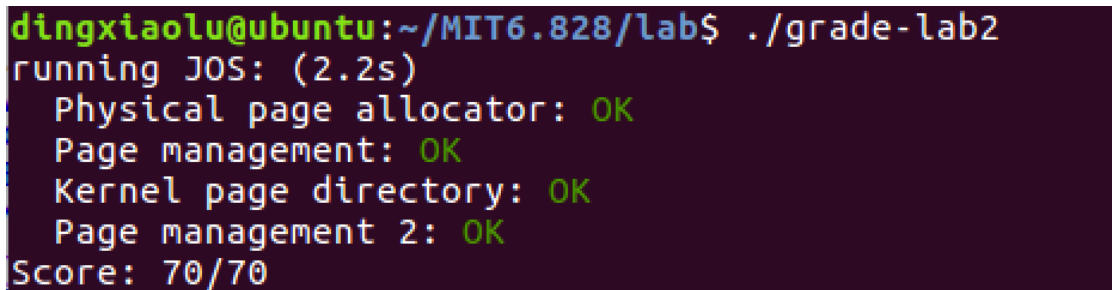
```
////////////////////////////////////  
// Use the physical memory that 'bootstack' refers to as the kernel  
// stack. The kernel stack grows down from virtual address KSTACKTOP.  
// We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)  
// to be the kernel stack, but break this into two pieces:  
//   * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory  
//   * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if  
//     the kernel overflows its stack, it will fault rather than  
//     overwrite memory. Known as a "guard page".  
// Permissions: kernel RW, user NONE  
// Your code goes here:  
boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE_W);  
////////////////////////////////////
```

这边映射的是整个内核操作系统，将虚拟地址范围是[KERNBASE,  $2^{32}$ ]，物理地址范围是[0,  $2^{32}$  - KERNBASE]。

访问权限是，kernel space 可以读写，user space 无权访问。

```
// Map all of physical memory at KERNBASE.
// Ie. the VA range [KERNBASE, 2^32) should map to
// the PA range [0, 2^32 - KERNBASE)
// We might not have 2^32 - KERNBASE bytes of physical memory, but
// we just set up the mapping anyway.
// Permissions: kernel RW, user NONE
// Your code goes here:
boot_map_region(kern_pgdir, KERNBASE, 0xffffffff - KERNBASE, 0, PTE_W);
```

至此 lab2 结束！



```
dingxiaolu@ubuntu:~/MIT6.828/lab$ ./grade-lab2
running JOS: (2.2s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
```

## Question

3. We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?
4. What is the maximum amount of physical memory that this operating system can support? Why?
5. How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?
6. Revisit the page table setup in kern/entry.s and kern/entrypgdir.c. Immediately after we turn on paging, EIP is still a low number (a little over 1MB). At what point do we transition to running at an EIP above KERNBASE? What makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE? Why is this transition necessary?

问题 3) 为什么用户代码不会去读取或写入内核内存？什么机制在保护内核内存？

因为如果用户代码读取或写入内核内存的话，导致内核数据破坏，造成整个系统程序的崩溃。

分段和分页机制在保护内核内存。通过阅读 intel 80386 的用户手册有段机制的保护和分页机制的保护：

All five aspects of protection apply to segment translation:

1. Type checking
2. Limit checking
3. Restriction of addressable domain
4. Restriction of procedure entry points
5. Restriction of instruction set

Two kinds of protection are related to pages:

1. Restriction of addressable domain.
2. Type checking.

问题 4) 这个操作系统可以支持的最大的物理内存占多大？为什么？

因为这个操作系统利用一个大小为 4MB 的空间 UPAGES 来存放所有的页的 PageInfo 结构体信息，每个结构体的大小为 8B，所以一共可以存放 512K 个 PageInfo 结构体，所以一共可以出现 512K 个物理页，每个物理页大小为 4KB，自然总的物理内存占 2GB。

问题 5) 如果现在的物理内存页达到最大个数，那么管理这些内存所需要的额外空间开销有多少？

此时存放所有的 PageInfo，需要 4MB；存放页目录表 kern\_pgdir，4KB；存放当前的页表，大小为 2MB。则总大小为 6MB+4KB。

## 总结：

该实验总共做了 3 件事：

- 1) 提供物理内存管理的数据结构和函数

注释:

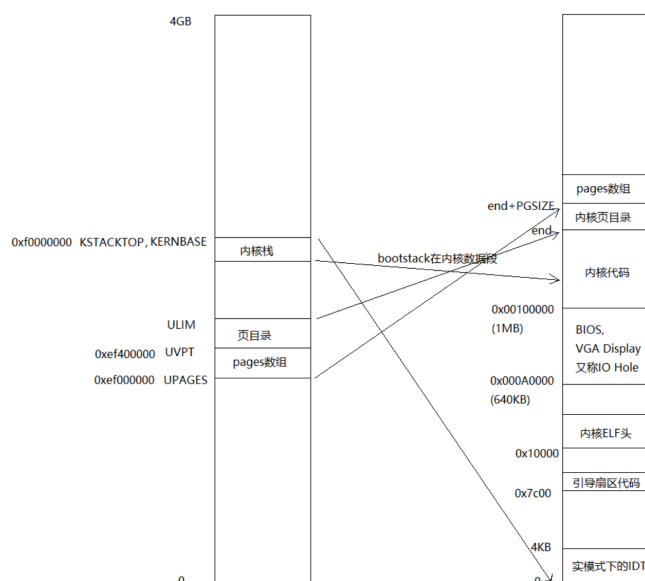
- 1.每个物理页对应pages数组中的一个PageInfo结构, 可以使用page2pa()和pa2page()转换PageInfo结构地址和物理页首地址。
- 2.page\_free\_list维护一个PageInfo结构链表, 该链表中的PageInfo代表的物理页是没有被使用的。
- 3.page\_alloc()从page\_free\_list中取一个PageInfo结构返回, 表示该PageInfo对应的物理页可以被调用者使用。
- 4.page\_free(struct PageInfo \*pp): 将pp指向的PageInfo重新放入page\_free\_list链表中, 以便后续分配。

- 2) 提供修改页目录和页表关系的函数, 实现虚拟地址到物理地址的映射函数。

页表管理函数:

- 1.page\_insert(pde\_t \*pgdir, struct PageInfo \*pp, void \*va, int perm): 建立映射线性地址到物理地址的映射关系。pgdir为页目录的地址, va为虚拟地址, pp代表物理页, perm代表权限。通过pgdir和va找到页表的一项PTE, 将物理页首地址和perm结合生成PTE项, 放入该PTE中。本质是操作pgdir指向的树结构。
- 2.page\_remove(pde\_t \*pgdir, void \*va): 解除线性地址va的映射关系。和page\_insert()类似。

- 3) 将内核空间相关线性区域映射到物理内存区域。



现在我们可以直接使用 **UPAGES** 这个虚拟地址直接访问 **pages** 数组，使用 **UVPT** 访问内核页目录表，使用 **KSTACKTOP** 访问内核栈。