Lab1_1

Question1:
1):
问：At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
答：At the point “7c2d: ea 32 7c 08 00 66 b8 ljmp $0xb866,$0x87c32”,the processor starts executing 32-bit code.
0x7c2d 的前几句的指令如下图。图中指令将程序从 16 转到 32 位，即实模式到保护模式的转换。
The code in boot.S ;
lgdt gdtdesc;movl %cr0,%eax;orl $CR)_PE_ON,%eax;movl %eax,%cr0.
截图：

```
lgdt    gdtdesc
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0
```

指令地址截图：

```
movl    %eax, %cr0
  7c2a:       0f 22 c0                mov    %eax,%cr0
```

解析：These codes change PE flag in CR0 register from 0 to 1,0 stand for 16-bit,1 stand for32-bit。
gdt:global descripton table.可以在 boot.S 的最底部看到 gdt 的定义。其中第一段为空段（规定），第二句是 mmu.h 中的宏定义，第一个参数是 type,X 为可执行，R 为可读，W 为可写。第二个和第三个参数分别是起始地址和段的大小。由下图可见，该 gdt 定义了 4G 大小的段。但是后续实验中我们知道，我们并没有使用段来管理页面，而是使用了页表管理。但是在开启页表之前，我们仍然直接访问物理内存。其实在程序进入 entry.S 中后，便开启了 cr3 的页表，但是使用的只是提前映射好的虚拟地址表。在 lab2 中才正式使用页表管理。

```
gdt:
  SEG_NULL                                  # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff)         # code seg
  SEG(STA_W, 0x0, 0xffffffff)               # data seg
```

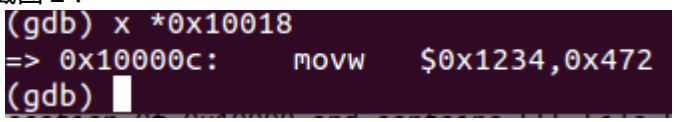
2):
问：What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?
答：a. The last instruction of the boot loader executed is"((void (*)(void)) (ELFHDR->e_entry))();"
Assambel（汇编代码）:call *0x10018
b. The first instruction of the kernel when it just loaded is "movw $0x1234 0x472"
解析：a. boot/boot.S 的功能是将实模式转换为保护模式，之后通过“call bootmain”进入 boot/main.c 文件中，boot/main.c 的主要功能时加载内核文件。在 boot/mian.c 中可以看到在加载完内核文件之后，程序会通过 ELF 文件（该数组结构可以在 inc/elf.h 中找到）中的 entry 变量进入内核文件。

```
// call the entry point from the ELF header
// note: does not return!
((void (*)(void)) (ELFHDR->e_entry))();
```

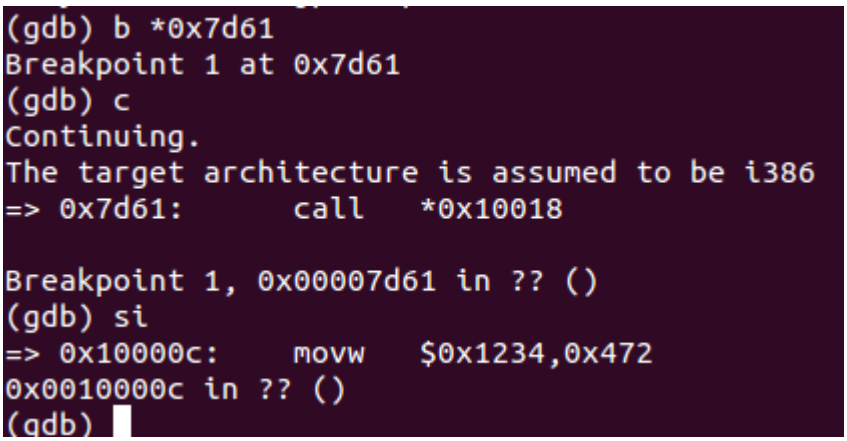
b.通过 obj/boot/boot.asm 中的汇编源码，找到如下源码：

```
    // call the entry point from the ELF header
    // note: does not return!
    ((void (*)(void)) (ELFHDR->e_entry))();
7d61:       ff 15 18 00 01 00       call   *0x10018
```

知道 entry 的入口地址是 0x10018 的内容。可以通过 gdb 的命令 x *0x10018 查看指令。
也可以将 gdb 的断点设为 0x7d61,则程序执行到 boot/mian.c 的最后一条指令。si 单步执行即进入内核文件第一条指令。
截图 1：

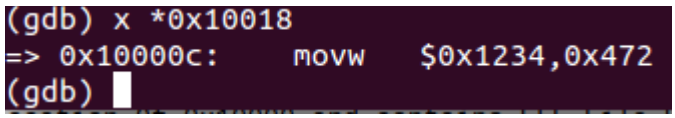


截图 2：



3):
问：Where is the first instruction of the kernel?

答：At address 0x0010000c.位于/kerne/entry.S 文件中
解析：由 2）中的答案可知，第一条指令的地址在 0x10018 中。可以通过 gdb 命令：x *0x10018 来查看 0x10018 的内容，即内核第一条指令地址



4):
问：How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

答：Boot loader decides how many sectors to read using information stored in ELF header.（elf 文件结构可以在 inc/elf.h 中查看）
在 boot/main.c 中
First:#define ELFHDR ((struct Elf *)0x10000) The starting memory addr to load kermel file。

Second:function readseg((uint32_t)ELFHDR,SEGTSIZE*8,0) load 8 sectors from disk,which starting on location 0f 0x10000 and contains ELF File Header.
ELF File Header:It comes at the very beginning of the executable, and can be read directly from the first e_ehsize (default: 512) bytes of the file into this structure.

Third:"if (ELFHDR->e_magic != ELF_MAGIC) goto bad;"checked is whether this is a correct ELF file by checking if magic number(first four bytes) of ELF file are correct.

Fourth:According to information in ELF File Header,finding the program

header,then loading each program segment with p_pa,p_memsz and p_offset arguments
加载 elf 数据段的截图：

```
// load each program segment (ignores ph flags)
ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
eph = ph + ELFHDR->e_phnum;
for (; ph < eph; ph++)
        // p_pa is the load address of this segment (as well
        // as the physical address)
        readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
```

```
struct Elf {
        uint32_t e_magic;       // must equal ELF_MAGIC
        uint8_t e_elf[12];
        uint16_t e_type;
        uint16_t e_machine;
        uint32_t e_version;
        uint32_t e_entry;
        uint32_t e_phoff;
        uint32_t e_shoff;
        uint32_t e_flags;
        uint16_t e_ehsize;
        uint16_t e_phentsize;
        uint16_t e_phnum;
        uint16_t e_shentsize;
        uint16_t e_shnum;
        uint16_t e_shstrndx;
};
```

总结：首先 mian.c 把 ELF 头文件（位于 disk 的第一页）读入内存，内核文件是分为若干段的，段的数目由 ELFHDR 头文件中的 e_phnum 变量决定，e_phoff 变量是段目录结构距 ELFHDR 的偏移。ph 为指向首个段目录数据结构的指针，eph 指向最后一个段目录。每个段目录结构包含该内核段的具体信息。
然后使用 readseg 来读取每一个段的具体内容。

question2:
1)：
答：console.c exports "void cputchar(int c)"
princtf.c is based on the kernel console's cputchar().When printf.c calls vprintfmt in function vcprintf,function putch as a parameter,while function calls function cputchar().
解析：

```
// `High'-level console I/O.  Used by readline and cprintf.

void
cputchar(int c)
{
        cons_putc(c);
}
```

由上图可知在 kern/console.c 中，注
释表明 cputchar 是高层级的控制端 I/O，被 cprintf 调用。其调用的 cons_putc 函数向端口输出一个字符。

```
int
cprintf(const char *fmt, ...)
{
        va_list ap;
        int cnt;

        va_start(ap, fmt);
        cnt = vcprintf(fmt, ap);
        va_end(ap);

        return cnt;
}

 int
 vcprintf(const char *fmt, va_list ap)
 {
         int cnt = 0;

         vprintfmt((void*)putch, &cnt, fmt, ap);
         return cnt;
 }

static void
putch(int ch, int *cnt)
{
        cputchar(ch);
        *cnt++;
}
```

由上面三幅图可知，在 kern/printf.c 中，函数 printf 首先调用函数 vcprintf，vcprintf 再调用函数 putch，putch 再调用 kern/console.c 中的函数 cputchar。

2)：

When the screen is full,scroll down on row to show newer information.

解析:In the console.h,"#define CRT_SIZE (CRT_ROWS*CRT_COLS)",so the CRT_SIZE is the biggest num the screen can print.

When crt_pos(注：cursorposition)>=CRT_SIZE,call function memmove( ) in lib/string.c which can move char from crt_buffer+CRT_COLS to crt_buffer,so that the screen display a blank row.

注：memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));crt_buf 指向当前缓冲区屏幕第一行字符，crt_buf+CRT_COLS，即 crt_buf 加上屏幕的列数，便是第二行首地址，(CRT_SIZE - CRT_COLS) * sizeof(uint16_t)为除去第一行后的字符数，所以该函数是将屏幕从第二行开始上移进而将第一行覆盖掉。然后 for 循环将最后一行置为空格。然后“crt_pos -= CRT_COLS;”使得 crt_pos 指向最后一行行首。

Work1:

In the printfmt.c,replace the original code on row 208-row 212 with "nun=getuint(&ap,lflag);base=8;goto number".

解析：在 lib/printfmt.c 中，可以知道省略的是输出八进制的格式。参照十进制格式便可得到答案。

```
// unsigned decimal
case 'u':
        num = getuint(&ap, lflag);
        base = 10;
        goto number;

// (unsigned) octal
case 'o':
        // Replace this with your code.
        num = getuint(&ap, lflag);
        base = 8;
        goto number;
```

Practice3:

We can know 0x20 the ebp incresed each time it gets called.So every time it push 8 32-bit words.According to section 2.3,在函数调用前压进栈的数据为：参数，eip(next is，以便返回原函数)，entering function body,push ebp,mov %esp,%ebp,push ebp.So 8 32-bit words are:return addr,old ebp,new ebp,parater.

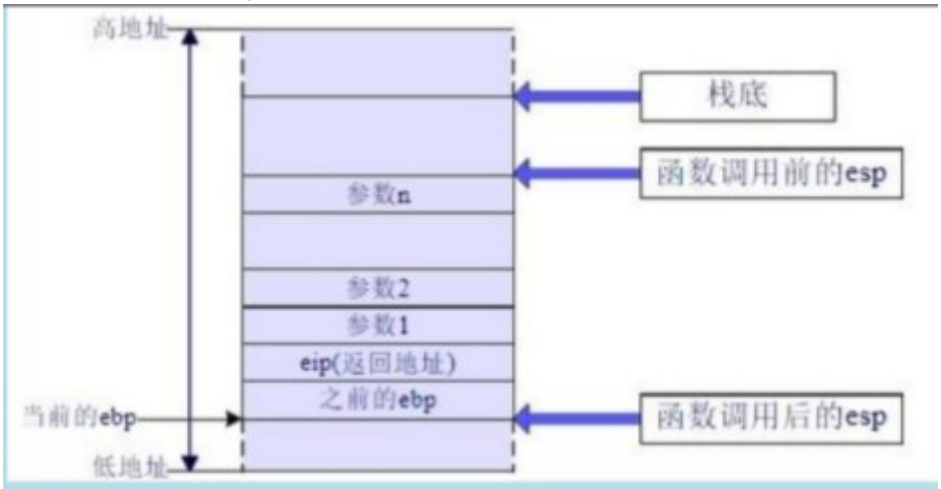
```
Stack backtrace:
ebp f010ff18 eip f0100087args 0 0 0 0 f0100965
ebp f010ff38 eip f0100069args 0 1 f010ff78 0 f0100965
ebp f010ff58 eip f0100069args 1 2 f010ff98 0 f0100965
ebp f010ff78 eip f0100069args 2 3 f010ffb8 0 f0100965
ebp f010ff98 eip f0100069args 3 4 0 0 0
ebp f010ffb8 eip f0100069args 4 5 0 10094 10094
ebp f010ffd8 eip f01000eaargs 5 1aac 660 0 0
ebp f010fff8 eip f010003eargs 111021 0 0 0 0
```

Work2:

函数代码

According to practice 3,we push info into stack when we call function all the time.

（层次图见练习 3 的图）.



解释一下 mon_backtrace()函数，该函数的原型在 kern/monitor.c 中

在调用函数时依次压入栈的参数为：五个参数，下一条指令地址（即返回地址 eip），参数指针 ebp。

我们使用 inc/x86.h 中的函数 read_ebp() and read_eip()来读取 ebp 和 eip 的内容。

不过在每次调用完函数后，就将 esp 赋值给 ebp。所以在 mon_backtrace（）中 read_ebp()是 esp 的值，即当前栈顶的值。

至于命令行添加，可以参见 monitor.c 中 help 和 kerninfo 的设置。

Lab1_2

该部分的代码修改主要在 kern/pmap.c 中
pmap.c 主体在函数 mem_init()中，该函数通过调用其他函数来完成对内存的初始化和管理。

其实在 lab1_1 中我们便接触了虚拟内存，在进入 entry.S 内核文件后（entry.S 主要功能是开启页式管理），便紧接着开启了虚拟内存的管理，这是内核文件被链接器链接到了虚拟的高地址空间，所以需要进行地址的映射。但是当时我们还没有进行虚拟内存管理的各种函数，所以在 entry.S 中的第 56 行可以看到，我们加载了一个地址映射表(entry_pgdir,该文件在 kern/entrypgdir.c 中)到页表寄存器 cr3 中。然后才在 entry.S 的 79 行 call i386_init（该函数在 lab2/kern/init.c 中）进入内核文件的 c 代码中，之后从 init.c 调用"mem_init();"。mem_init()在 kern/pmap.c 中，自此正式开始页面的管理。

Work3:
在进行物理页面分配中，根据注释，我们知道一共涉及 4 个函数的补充。
依次解释各个函数
1）：

```
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.
        result=nextfree;
        nextfree+=ROUNDUP(n,PGSIZE);
        //if(nextfree>(char*)(KERNBASE+npages*PGSIZE))
                //panic("Run out of memeory,nextfree=%x\r\n",nextfree);
        return result;
}
```

boot_alloc(n)：该函数根据参数 n 分配能够容纳 n 个字节的若干页。该函数主要在 mem_int()中用来申请初始页目录 kern_pgdir 和页表数组 pages。
思路：首先初始化 nextfree,如果其是 NULL，则使 nextfree 指向对齐后的 end(连接器自动生成的，即下一个空页的首地址)。然后根据 n 的大小，分配空间若干页，然后返回刚才所分页的首页地址，其次更新 nextfree 使其指向下一个空页。
详解 pmap.c 中的 ROUNDUP((char*)end,PGSIZE).
ROUNDUP(a,n)其实是在 inc/types.h 中的宏定义。其具体原型如下：

```
#define ROUNDDOWN(a, n)                                    \
({                                                  \
      uint32_t __a = (uint32_t) (a);                        \
            (typeof(a)) (__a - __a % (n));                        \
            })
#define ROUNDUP(a,n)
```

```
({uint32_t __n=(uint32_t)n;
(typeof(a))(ROUNDDOWN(uint32_t)(a)+__n-1,__n});
```

分析后可知其具体功能是 nextfree 和 end 始终和 4K 空间对齐，简单来说就是 nextfree 和 end 始终指向每个 4K 空间的首位，即对 PGSIZE 取余得 0。
其中 typeof(a)意思是获得 a 的数据类型，然后将结果进行强制类型转换为 a 的类型。

2）：

```
// Allocate an array of npages 'struct Page'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 Page in this
// array.  'npages' is the number of physical pages in memory.
// Your code goes here:
pages=(struct Page *)boot_alloc(npages*sizeof(struct Page));
```

根据 mem_int()注释提示，首先注释掉 panic("mem_init.....")语句。
然后根据提示在指定位置使用 boot_alloc()为 pages 数组分配足够空间。该数组的每一项对应于一个具体的物理页。数组的下标最大为 npages-1（npages 由函数 i386_detect_memory()得到）.pages 数组控制了所有的物理页。

3）：

```
        size_t i;
        //extern char end[];
        page_free_list=NULL;
        int lower_pgnum=PGNUM(IOPHYSMEM);//函数PGNUM()在inc/mmu.h中
        uint32_t upper_pgnum=PADDR(boot_alloc(0))/PGSIZE;
        int extphy=PGNUM(EXTPHYSMEM);
        cprintf("lower_pgnum:%d\r\nupper_pgnum:%d\r\nextphy:%d\r\nnpages:%d\r\nbasenum:%d\r
\n",lower_pgnum,upper_pgnum,extphy,npages,npages_basemem);
        page_free_list=NULL;
        for (i = 0; i < npages; i++) {
                if(i==0){
                pages[i].pp_ref = 1;
                pages[i].pp_link=NULL;
                continue;
        }else if(i < npages_basemem)
        {
            pages[i].pp_ref = 0;
            pages[i].pp_link = page_free_list;
            page_free_list = &pages[i];
        } else if(i>=lower_pgnum&&i<upper_pgnum){
                pages[i].pp_ref=1;
                pages[i].pp_link=NULL;
                continue;
        }else{
                pages[i].pp_ref=0;
                pages[i].pp_link = page_free_list;
                page_free_list = &pages[i];}
        }
```

page_init()：初始化 pages 数组.即找出哪些页表已经被使用了。
思路：首先需要搞明白在调用 page_init()时，哪些 page 已经被之前程序和数据占用了。根据提示和对之前代码的分析可以得到如下结论：
a.内存的第一个页被 IDT(中断描述标占用)
b.[PGSIZE,npages_basemem*PGSIZE)为空闲的
c.[IOPHYSMEM,EXTPHYMEM)被占用
d.[KERNBASE,nextfree)的空间被内核文件和 pages 数组占用
注：除了上述的空间其他都是空闲的。而且 c,d 两段地址是连续的[IOPHYSMEM,nextfree)

4）：

page_alloc()：当

```
struct Page *
page_alloc(int alloc_flags)
{
        // Fill this function in
        if(!page_free_list)
          return NULL;
        struct Page *ret=page_free_list;
        page_free_list=page_free_list->pp_link;
        if(alloc_flags&ALLOC_ZERO)
            memset(page2kva(ret),0,PGSIZE);
        return ret;
}
```

page_free_list 不是空时，为调用者分配一个空闲页。返回使 page_free_list 当前指向的 page，同时更新 page_free_list 使其指向 page_free_list->pp_link。

5）：

```
void
page_free(struct Page *pp)
{
        // Fill this function in
    pp->pp_link=page_free_list;
    page_free_list=pp;
}
```

page_free(pp)：当 pp->pp_ref==0 时，调用此函数。
逻辑为将页 pp 加入 page_free_list。使将要释放的页的 pp_link 指向 page_free_list，然后 page_free_list 指向将释放的页。

question3:
x is uintptr_t.
答：Because the kernel operates data uses the virtual addr.所以只能对虚拟地址进行解引用。所以是虚拟地址。

Work4:
代码在/kern/pmap.c 中。此处解析各个函数的思路和用处
1）：

```
//pte_t是在inc/memlayout.h中定义的uint32_32类型
//PTE_ADDR()将地址的后十二位变为0，后十二位为权限标志为
//PTX()：地址右移12,然后在取结果的后十位

//pgdir()：查找va对应的页表项
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
        // Fill this function in
        pde_t *va_pgdir=kern_pgdir+PDX(va);
        pte_t *va_pgtable=NULL;//va_pgtable为页表位置
        struct Page *pg=NULL;
//在inc/mmu.h中PTE_ADDR(pte)：((physaddr_t) (pte) & ~0xFFF)
//标志位PTE_P表示当前页面是否存在
        if(*va_pgdir&PTE_P){
                va_pgtable=KADDR(PTE_ADDR(*va_pgdir));
                return va_pgtable+PTX(va);
        }
        else if(create){
                pg=page_alloc(ALLOC_ZERO);//分配空闲页面并且清空它;
                if(pg==NULL) return NULL;
                pg->pp_ref++;
                va_pgtable=(pte_t*)page2kva(pg);
                *va_pgdir=PADDR(va_pgtable)|PTE_P;
                return va_pgtable+PTX(va);
        }
        else return NULL;
}
```

pgdir_walk(pgdir,va,create):该函数接收一个虚拟地址 va，将 va 对应的二级页表中的页表项地址返回。
思路：首先通过 va_pgdir=kern_pgdir+PDX(va)得到页目录项，查看*va_pgdir 中的 PTE_P 标志位首否为 1,为 1 则存在对应页表，此时通过 PTE_ADDR(*va_pgdir)得到页表的入口地址（即页表首地址），加上 PTX(va)即得页表项的地址。否则不存在，此时根据 create 的值决定是否创建一个页表。使用 page_alloc()函数分配一个页作为页表 va_pgtable，将此地址赋值给*va_pagdir。最后 va_pgtable+PTX(va)得到页表项地址。

2）：

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
        // Fill this function in
        pte_t* pte=NULL;//定义页的入口变量
        assert(size%PGSIZE==0);//判断size是否真的是PGSIZE的倍数
        int i;
        for(i=0;i<size/PGSIZE;i++){
        pte=pgdir_walk(pgdir,(void*)va,1);
        if(!pte)return;
        *pte=(PTE_ADDR(pa))|perm|PTE_P;//perm:
        //修改目录项的权限
        pgdir[PDX((void*)va)]=PADDR(pte)|perm|PTE_P;
        va+=PGSIZE;
        pa+=PGSIZE;
        }
}
```

page_map_region():该函数将虚拟地址[va,va+size)映射到物理地址[pa,pa+size)上，其中 size 时 PGSIZE 的整数倍。
思路：使用 page_walk()函数得到 va 的页表项地址 pte,将 pte 的内容*pte 修改为 PTE_ADDR(pa)|perm|PTE_P，这就完成了一个页的映射。然后循环止至 va 增加到 va+size。

3）：

```
//page_lookup()查找va对应的具体物理页
struct Page *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
        // Fill this function in
        pte_t*pte=pgdir_walk(pgdir,va,0);
        if(pte_store) *pte_store=pte;
        if(!pte)return NULL;
        return pa2page(PTE_ADDR(*pte));
}
```

page_lookup():该函数查找 va 对应的具体物理页地址。
思路：首先通过 page_walk(, ,create=0)得到 va 对应的页表项地址，如果没有对应的页表则直接返回 NULL，否则得到页表项地址 pte。返回 pte 中的地址内容 PTE_ADDR(*pte).
根据注释提示，当 pte_store 不为空时，将页表项地址赋值给 pte_store,这个语句将会在 page_remove 中用到。

4）：

```
//page_decref(page):将当前页的引用减一，如果为0则调用page_free();
//
void
page_remove(pde_t *pgdir, void *va)
{
        // Fill this function in
        pte_t* pte;
        struct Page* phypage=page_lookup(pgdir,va,&pte);
        if(phypage!=NULL){
        page_decref(phypage);
        *pte=0;
        tlb_invalidate(pgdir,va);
        }

}
```

pgae_remove()：该函数删除虚拟地址 va 对应的具体物理页。
思路：通过 page_lookup()查找 va 的具体物理页，如果不存在（即返回 NULL）则无需处理。如果存在，则调用 page_decref()来删除 va 对物理页 phypg 的引用(即 phypg->pp_ref--,如果为 0,则调用 page_free()来释放 phypg)。然后把*pte 的值改为 0,从函数 page_lookup 可知当前 pte 存储着 va 对应页表项的地址，所以将*pte 置 0,即标志删除物理页映射。最后调用 tlb_incalidate()使 va 无效。

5）：

```
//TLB(Translation Lookaside Buffer)传输后备缓冲器是一个内存管理单元,用于改进虚拟地址到物理地址转换速度的缓存。
int
page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{
        // Fill this function in
        pte_t*pte=pgdir_walk(pgdir,va,1);
        if(pte==NULL)
           return -E_NO_MEM;
                //如果已经有物理页映射，则删除
                if(*pte&PTE_P){
                        if(PTE_ADDR(*pte)==page2pa(pp)){
                        *pte=page2pa(pp)|perm|PTE_P;
                        pgdir[PDX(va)]=PADDR(pte)|perm|PTE_P;
                        }
                        else{
                        page_remove(pgdir,va);
                        *pte=page2pa(pp)|perm|PTE_P;
                        pgdir[PDX(va)]=PADDR(pte)|perm|PTE_P;
                        pp->pp_ref++;
                        }
                }
                else{
                *pte=page2pa(pp)|perm|PTE_P;
                pgdir[PDX(va)]=PADDR(pte)|perm|PTE_P;
                pp->pp_ref++;
                }
    return 0;
}
```

page_insert()：该函数将一个指定物理页 pp 映射成一个指定的虚拟地址 va
思路：调用 page_walk(，，1)（create 为 1,表示如果 va 没有页表则分配）得到 va 的页表项地址 pte;检查 PTE_ADDR(*pte)是否已经指向其他物理页了，如果是，则先调用 page_remove()删除 va 对原物理页的引用，再将*pte 指向 pp;否则还有两种情况，其一：*pte 已经指向了 pp，此时只用按照参数 perm 来修改权限即可，其二：*pte 指向空，此时修改*pte 的值和权限，并且修改目录项(pgdir(PDX(va)))的权限。

Work5:

使用 boot_map_region()函数根据注释提示映射 inc/memlayout.h 中的各个虚拟地址到物理页。

1）：

```
        //将UPAGES及其以上的空间映射到pages
        boot_map_region(kern_pgdir,UPAGES,ROUNDUP(npages*sizeof(struct Page),PGSIZE),PADDR
((uintptr_t*)pages),PTE_U);
```

首先将 UPAGES 映射到 pages 空间，此后便能在 UPAGES 虚拟空间中访问 pages。

结合 mem_init()中

2）：

```
//KSTKSIZE为8个PGSIZE大小
boot_map_region(kern_pgdir,KSTACKTOP-KSTKSIZE,KSTKSIZE,PADDR((uintptr_t*)bootstack),PTE_W);
```

然后将内核栈 KSTACKTOP 映射到 bootstack。其中 bootstack 为 entry.S 中定义的地址空间，大小为 KSTKSIZE。

3）：

```
// Your code goes here:
boot_map_region(kern_pgdir,KERNBASE,~KERNBASE+1,(physaddr_t)0,PTE_W);
```

最后将内核空间映射到物理内存的 0 ~ 256M 上。

question4：

1）：在 pmap.c 的 mem_int()函数中，我们可以看到进行了了四次页目录的映射操作。

a.首先是将 UVPT 映射到 kern_pgdir,根据 UVPT:0xef400000,PDX(UVPT)=957,所以页目录的第 957 个入口被占用。该语句主要把 UVPT 映射到 kern_pgdir.

b.其次是将虚拟地址 UPAGES 映射到 pages 数组，根据 qemu 返回的 pages 数组大小约为 33*4k;由于 UPAGES-UVPT 的大小仅为 4M,所以 pages 最多为 4M，UPAGES:0xef000000.因此只占用 kern_pgdir 的一个目录 kern_pgdir[956].该语句把虚拟地址 UPAGES--UPAGES+pagesSize 映射的 pages 的物理地址(根据 qemu 打印信息可知该地址为 0x119000)上.

c.再者就是将虚拟地址 KSTACKTOP-KSTKSIZE(大小为 4M)映射盛放内核文件的栈 bootstack(根据 qemu 的打印信息，bootstack:10d00)上.KSTACKTOP-KSTKSIZE:0xefb00000.所有 kern_pgdir[958]被占用。

d.最后把虚拟地址 KERNBASE-0xffffffff 映射到物理地址的 0-256M。这个需要占用 64 个页目录 kern_pgdir[960-1023].

Entry	Base Virtual Addr	Point to
956	0xef000000	指向 pages 数组
957	0xef400000	指向页目录 kern_pgdir
958	0xefb00000	指向 bootstack
960-1023	0xf0000000	指向物理地址 0-256M

其实根据实验说明的提示，使用 Ctrl+a c 的调用 info pg 页可以得到答案。一下是截图。其中方括号中的地址都是被占用的。

```
(qemu) info pg
VPN range     Entry         Flags        Physical page
[ef000-ef3ff]  PDE[3bc]     --S----U-P
  [ef000-ef020]  PTE[000-020] -------U-P 00119-00139
[ef400-ef7ff]  PDE[3bd]     -------U-P
  [ef7bc-ef7bc]  PTE[3bc]     --S----U-P 003fd
  [ef7bd-ef7bd]  PTE[3bd]     -------U-P 00118
  [ef7be-ef7be]  PTE[3be]     -GSDACTUWP 003fe
  [ef7c0-ef7ff]  PTE[3c0-3ff] -GSDACTUWP 003ff 003fc 003fb 003fa 003f9 003f8 ..
[ef800-efbff]  PDE[3be]     -GSDACTUWP
  [efbf8-efbff]  PTE[3f8-3ff] --------WP 0010d-00114
[f0000-f03ff]  PDE[3c0]     -GSDACTUWP
  [f0000-f0000]  PTE[000]     --------WP 00000
  [f0001-f009f]  PTE[001-09f] ---DA---WP 00001-0009f
  [f00a0-f00b7]  PTE[0a0-0b7] --------WP 000a0-000b7
  [f00b8-f00b8]  PTE[0b8]     ---DA---WP 000b8
  [f00b9-f00ff]  PTE[0b9-0ff] --------WP 000b9-000ff
  [f0100-f0105]  PTE[100-105] ----A---WP 00100-00105
  [f0106-f0113]  PTE[106-113] --------WP 00106-00113
  [f0114-f0114]  PTE[114]     ---DA---WP 00114
  [f0115-f0116]  PTE[115-116] --------WP 00115-00116
  [f0117-f0118]  PTE[117-118] ---DA---WP 00117-00118
  [f0119-f0119]  PTE[119]     ----A---WP 00119
  [f011a-f011a]  PTE[11a]     ---DA---WP 0011a
  [f011b-f0139]  PTE[11b-139] ----A---WP 0011b-00139
  [f013a-f03bd]  PTE[13a-3bd] ---DA---WP 0013a-003bd
  [f03be-f03ff]  PTE[3be-3ff] --------WP 003be-003ff
[f0400-f3fff]  PDE[3c1-3cf] -GSDACTUWP
  [f0400-f3fff]  PTE[000-3ff] ---DA---WP 00400-03fff
[f4000-f43ff]  PDE[3d0]     -GSDACTUWP
  [f4000-f40fe]  PTE[000-0fe] ---DA---WP 04000-040fe
  [f40ff-f43ff]  PTE[0ff-3ff] --------WP 040ff-043ff
[f4400-fffff]  PDE[3d1-3ff] -GSDACTUWP
  [f4400-fffff]  PTE[000-3ff] --------WP 04400-0ffff
(qemu)
```

2）：使用权限标志位 PTE_U 来限制用户程序的访问权限。在页目录和页表中后 12 位是各种形式的标志位。可以在 inc/mmu.h 中找到各种标志位。

3）：2G。因为 pages 数组映射到 UPAGES 虚拟内存中，而 UPAGES 为 4M，而 sizeof(struct Page)=8B,所以最多可以容纳 4M/8B=512k 个数组单元，即最大支持 512k 个物理页。512k×4k=2G。

4）：**第一部分**是页目录的空间开销：在 kern/pmap.c 的 mem_init()中，由 kern_pgdir=(pde_t*)boot_alloc(PGSIZE)可知 kern_pgdir 占用了 4K 空间

第二部分是 pages 数组占用的空间：这个跟实际的物理内存有关。假设我们管理的物理内存为 256M，首先通过 sizeof(struct Page)得到一个 pages 单元占 8 字节。所以此时 pages 数组大小为 2G/4K=8×64×1024;则所占空间为 8×64×1024×8B=4M。在本实验中，通过打印 npages 得到 16639,所以 pages 大约占用 129.486k。

第三部分是页表所占用的空间：在 page_walk()中我们知道我们需要给每个目录项分配一个大小为 PGSIZE 的页作为页表空间，因为本 os 种 kern_pgdir 有 1024 个目录项，一个页目录项对应 4M，所以最多有 1024 个目录项，每个目录项对应一个 4K 页表，所以页表占用 1024×4K=4M。

综上三部分，在 2G 物理内存下，内存管理消耗 8M+4K 得空间。

减小管理内存使用空间的大小或许可以增大每个页的大小的实现。比如将每个页扩大至 8K。

分析：假设我们把页面扩大到 8K。此时虚拟地址格式如下，前 9 位为页目录项入口地址，中间 10 位为页表项入口地址，最后 13 位为页内地址。

此时，我们只需要使用 256K 个 pages 数组项来管理 2G 的内存。pages 数组消耗的空间减小一半至 2M。

页目录还是之前的 4K 空间没有变化。

pages 数组减小一般，那么页表数就相应的减少一半至 512，但每页大小变成了 8K，所以页表占用的空间也没有变。

所以一共就减小了 2M-4K