作业 3:
check_page_free_list() and check_page_alloc() test your physical page allocator.
操作系统必需跟踪哪些物理 RAM 是空闲的，哪些正在使用。这个 exercise 主要编写物理页面分配器。它利用一个 PageInfo 结构体组成的链表记录哪些页面空闲，每个结构体对应一个物理页。因为页表的实现需要分配物理内存来存储页表，在虚拟内存的实现之前，我们需要先编写物理页面分配器。
boot_alloc 函数实现如下（仅展示添加代码）：

```
        // LAB 2: Your code here.
        if(n==0)
                return nextfree;
        result = nextfree;
        nextfree += n;
        nextfree = ROUNDUP( (char*)nextfree, PGSIZE);
        return result;
```

实现功能：所以在系统第一次调用boot_alloc()这个函数的时候，首先nextfree会被指向第一个空闲页的首地址。接下来，根据输入的n，来分配地址。如果n=0，则返回nextfree，否则分配n字节的地址，返回分配地址的首地址。
设计方法：
end符号向上均为未使用空间，只要返回这些空间就行。
首先将 end 符号向上和 4K 字节对齐（JOS 已经帮我们完成），然后将 nextfree 加上要分配的空间并依然 4K 字节对齐，接着返回原先的 nextfree 即可。

mem_init 函数：
主要是要为 struct PageInfo 的结构体的指针 pages 申请一定的地址空间。首先来看在 inc/memlayout.h 中 struct PageInfo 的定义：

这个结构体，主要是用来保存内存中的所有物理页面的信息的。每一个 PageInfo 对应一个物理页面。

```
struct PageInfo {
        // Next page on the free list.
        struct PageInfo *pp_link;

        // pp_ref is the count of pointers (usually in page table
entries)
        // to this page, for pages allocated using page_alloc.
        // Pages allocated at boot time using pmap.c's
        // boot_alloc do not have valid reference count fields.

        uint16_t pp_ref;
};
```

pageInfo 主要有两个变量:
pp_link 表示下一个空闲页，如果 pp_link=0,则表示这个页面被分配了，否则，这个页面未被分配，是空闲页面。
pp_ref 表示页面被引用数，如果为 0，表示是空闲页。(这个变量类似于智能指针中指针的引用计数)。

我们要做的就是补充能够代码为 pages 申请足够的空间(npages 的页面)，来存放这些结构体，并且用 memset 来初始化，代码如下：

```
        // 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 = boot_alloc(npages * sizeof (struct PageInfo));
        memset(pages, 0, npages*sizeof(struct PageInfo));

        ////////////////////////////////////////////////////////////////////
        // Now that we've allocated the initial kernel data
```

所做的就是为 pages 分配内存

关于 pages 的一些理解如下：

pages 是用来管理物理内存页的一个结构体。首先，pages 是整个系统物理内存的第 0 页，pages 的值是一个虚拟地址。由于结构体是连续的，所以通过 pages[i]-pages 可以得到页的编号 i，在通过 i<<12 就可以得到 pages[i]所对应的页的物理内存，由于实现系统的物理内存和虚拟内存的转换比较简单，虚拟内存=物理内存+ 0xF0000000.所以通过 pages 这个结构体，在知道具体的物理页时，就可以很容易得到物理页对应的物理地址和虚拟地址

page_init()函数的实现：

所做工作：pages数组以及page_free_list的初始化：

在page_init()里系统首先初始化了pages数组以及page_free_list，可以看到这个page_free_list指向了所有的Page结构，换句话说此时认为所有的页面都是空闲可分配的，我们需要做的是要从中把一些我们已经用的内存页面从中剔除出去，这包括0地址向上的第一个页面（包括IDT等），IO hole（0xA0000--0x100000，包括vga display ,bios等），kernel地址之上的部分（kernel本身+kern_pgdir+pages）。巧合的是，IO hole，和kernel之上部分是连续的地址，因为kernel就加载在0x100000处，所以其实只需要剔除两块地址，第一块是0地址开始的第一个页面，第二块就是io hole开始的向上的一组连续的页面。

page_free_list 链表是从 pages 数组的末尾开始从高地址指向低地址，所以先计算出要剔除的 Page 的地址，然后用过指针剔除即可。

在函数里的代码实现如下：

```
        // free pages!
        size_t i;
        for (i = 0; i < npages; i++) {
                if(i == 0)
                        {       pages[i].pp_ref = 1;
                                pages[i].pp_link = NULL;
                        }
                else if(i>=1 && i<npages_basemem)
                {
                        pages[i].pp_ref = 0;
                        pages[i].pp_link = page_free_list;
                        page_free_list = &pages[i];
                }
                else if(i>=IOPHYSMEM/PGSIZE && i< EXTPHYSMEM/PGSIZE )
                {
                        pages[i].pp_ref = 1;
                        pages[i].pp_link = NULL;
                }

                else if( i >= EXTPHYSMEM / PGSIZE &&
                                i < ( (int)(boot_alloc(0)) - KERNBASE)/PGSIZE)
                {
                        pages[i].pp_ref = 1;
                        pages[i].pp_link =NULL;
                }
                else
                {
                        pages[i].pp_ref = 0;
                        pages[i].pp_link = page_free_list;
                        page_free_list = &pages[i];
                }
        }
}
```

部分代码分析：

其中：

```
pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
```

是把页面设为空闲，并插入链表头。

```
size_t first_free_address = PADDR(boot_alloc(0));
```

实际需要利用 boot_alloc 函数来找到第一个能分配的页面。相同的思想在已经写好的check_free_page_list函数中也可以找到。

page_alloc 函数

page_alloc()是页面申请函数，就是通过读取和更新 page_free_list 来申请页面。注意空闲页面被申请完的情况就可以了。从 page_free_list 头剔除一个 Page，然后改变 page_free_list 使其为 pp_link。代码如下：

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

page_free 就是释放页面，需要注意 pp_ref 和 pp_link 是否为 0。做法为将释放

页面的 pp_link 指向 page_free_list 头，然后改变 page_free_list 使其为 pp（要释放页面）。代码效果如下：

```
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.
        if(pp->pp_link != 0  || pp->pp_ref != 0)
                panic("page_free is not right");
        pp->pp_link = page_free_list;
        page_free_list = pp;
        return;
}
```

运行之后，可以看到结果如下图，

```
user@user-VirtualBox:~/lab1/src$ cd lab1_2
user@user-VirtualBox:~/lab1/src/lab1_2$ make qemu
+ as kern/entry.S
+ cc kern/entrypgdir.c
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/pmap.c
+ cc kern/kdebug.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
qemu -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log
VNC server running on `127.0.0.1:5900'
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
kernel panic at kern/pmap.c:128: mem_init: This function is not finished

Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

问题 3 假设以下内核代码是正确的，那么变量 x 将会是什么类型，uintptr_t 或者 physaddr_t? mystery_t x; char* value = return_a_pointer(); *value = 10; x = (mystery_t) value;
解答： JOS 系统里面，有物理地址和虚拟地址。由于系统会经常要进行地址的相关运算，所以经常要进行强制转化，把一个 unsigned int 型变量转化为一个地址，或者相反，所以在程序里面，需要对两者进行区分。
上面题目的问题，value是可以直接操作虚拟内存的char * ，x是从value强制转化而来，没有经过虚拟内存转物理内存的步骤，所以mystery_t应该也是虚拟内存的一种，故x是uintptr_t类型。(在程序里面，任何指针都是虚拟地址(段偏移)。)

作业4：

作业 4 在文件 kern/pmap.c 文件中，你必须实现以下函数的代码。

pgdir_walk()

boot_map_region()

page_lookup()

page_remove()

page_insert()

mem_init()调用的 check_page()，用于测试你的页表管理方法。

解答：

原理：首先从硬件机制说起，当cpu拿到一个地址并根据这个地址访问主存时，在x86体系架构下要经过至少两级的地址变换，第一级成为段式地址变换而第二级成为页式地址变换。

最原始的地址叫做虚拟地址，根据规定，将前16位作为段选择子，后32位作为偏移。根据段选择子查找gdt/ldt，查到的内容替加上偏移，此时的地址就变成了线性地址。

线性地址前10位被称作页目录入口（page directory entry也就是pde），其含义为该地址在页目录中的索引，中间10位为页表入口（page table entry，也就是pte），代表在页表中的索引，最后12位是偏移。

当一个线性地址进入页式地址变换机制时，首先cpu从cr3寄存器里得到页目录（page directory）在主存中的地址，然后根据这个地址加上pde得到该地址在页目录中对应的项。无论是页目录的项还是页表的项均是32位，前20位为地址，后12位为标志位。当获取了相应的页目录项之后，根据前20位地址得到页表所在地址，加上偏移pte得到页表项，取出前20位加上线性地址本身的后12位组成物理地址，整个变换过程结束。

这个过程完全是由硬件实现，在这个部分的实验中要做的是初始化并维护页目录与页表，当页目录与页表维护好了，然后使cr3装载新的页目录，一切就交由硬件去处理地址变换了。

关于虚拟、线性和物理地址：

虚拟地址

最原始的地址，也是 C/C++ 指针使用的地址。由前 16bit 段 (segment) 选择器和后 32bit 段内的偏移 (offset) 组成，显然一个段大小为 4GB。通过虚拟地址可以获得线性地址。

线性地址

前 10bit 为页目录项(page directory entry, PDE)，即该地址在页目录中的索引。中间 10bit 为页表项(page table entry, PTE)，代表在页表中的索引，最后 12bit 为偏移，也就是每页 4kB。通过线性地址可以获得物理地址。

物理地址

经过段转换以及页面转换，最终在 RAM 的硬件总线上的地址。

1、 pgdir_walk 函数

函数需要做的是返回 va 对应的二级页表的地址(PTE)，给定一个虚拟地址va和pgdir(page director table 的首地址)，返回va所对应的pte(page table entry)。当 va 对应的二级页表存在时，只需要直接按照页面翻译的过

程给出 PTE 的地址就可以了。但是，当 va 对应的二级页表还没有被创建的时候，就需要手动的申请页面，并且创建页面了。过程比较简单，但是在最后返回 PTE 的地址的时候，需要返回 PTE 地址对应的虚拟地址，而不能直接把 pte 的物理地址给出。因为程序里面只能执行虚拟地址，给出的物理地址也会被当成是虚拟地址，一般会引发段错误。

函数实现方法：

查找页目录表，根据宏PDX取得页目录项（相关宏定义在mmu.h中），如果不为空，取出该项内容的前20位（PTE_ADDR宏），这是物理地址，通过此物理地址查找对应的Page结构（pa2page宏），然后获得此Page的虚拟地址（page2kva宏）。

此时的地址为页表的虚拟地址，根据偏移得到页目录项，在返回此页目录项地址。

如果前20位不为空，检查create，如果为0，返回null。否则新分配一个Page作为页表，然后自增Page 的引用，让该页目录项的前20位为页表物理地址（page2pa得到物理地址），并设置一些权限符号（不加通不过最后的检测函数），在通过此页表的虚拟地址得到相应页表项的虚拟地址并返回。

```
//
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
        // Fill this function in
        int pdeIndex = (unsigned int)va >>22;
        if(pgdir[pdeIndex] == 0 && create == 0)
                return NULL;
        if(pgdir[pdeIndex] == 0){
                struct PageInfo* page = page_alloc(1);
                if(page == NULL)
                        return NULL;
                page->pp_ref++;
                pte_t pgAddress = page2pa(page);
                pgAddress |= PTE_U;
                pgAddress |= PTE_P;
                pgAddress |= PTE_W;
                pgdir[pdeIndex] = pgAddress;
        }
        pte_t pgAdd = pgdir[pdeIndex];
        pgAdd = pgAdd>>12<<12;
        int pteIndex =(pte_t)va >>12 & 0x3ff;
        pte_t * pte =(pte_t*) pgAdd + pteIndex;
        return KADDR( (pte_t) pte );//返回虚拟地址，物理地址会引发错误。
}
```

2. boot_map_region: [va, va+size)映射到[pa, pa+size)，也就是映射一片指定虚拟页到指定物理页，实现方法是反复利用pgdir_walk。此时的 va 类型是 uintptr_t，调用 pgdir_walk 时需要转换为 void *

代码如下：

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int
perm)
{
        // Fill this function in
        while(size)
                pgdir_walk(pgdir, (void* )va, 1);
                if(pte == NULL)
                        return;
                *pte= pa |perm|PTE_P;

                size -= PGSIZE;
                pa  += PGSIZE;
                va  += PGSIZE;
        }
}
```

3. page_lookup:
返回虚拟地址va对应的物理地址页面page，对应于物理地址pa的内存
代码如下：

```
//
struct PageInfo *
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 == NULL)
                return NULL;
        pte_t pa =  *pte>>12<<12;
        if(pte_store != 0)
                *pte_store = pte ;
        return pa2page(pa);
}

//
```

4. page_remove：对 va 和其对应的页面取消映射，对 va 对应的页面进行释放，换句话说，就是移除 va 虚拟地址与物理地址的映射。在这里进行页面释放之后，需要把 va 对应的 PTE 里面存储的值进行清零操作，否则查询 va 对应的 PTE 时，会发生错误，系统会误以为 va 和 pa 还是存在对应关系。
代码如下：

```
..
void
page_remove(pde_t *pgdir, void *va)
{
        // Fill this function in
        pte_t* pte;
        struct PageInfo* page = page_lookup(pgdir, va, &pte);
        if(page == 0)
                return;
        *pte = 0;
        page->pp_ref--;
        if(page->pp_ref ==0)
                page_free(page);
        tlb_invalidate(pgdir, va);
}

..
```

5. page_insert():把va映射到指定的物理页表page
这个函数要考虑三种情况：

1. va没有对应的映射page
2：va有对应的映射page，但是不是指定的page
3.：va有对应的映射page，并且和指定的page相同。
对于情况1，最简单，直接把va和page映射就可以了，具体方法就是把va对应的pte求出，然后修改pte里面的值，使其为对应的page的物理地址。
对于情况2，先要把va对应的page释放(remove,ref-1)，然后和情况1一样的处理方法。
对于情况3，当两个page相同的时候，不能直接返回。因为page_insert函数不仅要进行虚拟地址和页面的映射，它还要对页面的特权(PTE_U,PTE_P,PTE_W)进行设置，如果原来的page的特权和现在的特权不一样，那么直接return，就会存在问题。

代码如下：

终端

```
(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;
        if( (pte[0] &  ~0xfff) == page2pa(pp))
                pp->pp_ref--;
        else if(*pte != 0)
                page_remove(pgdir, va);

        *pte = (page2pa(pp) & ~0xfff) | perm | PTE_P;
        pp->pp_ref++;
        return 0;
}

//
```

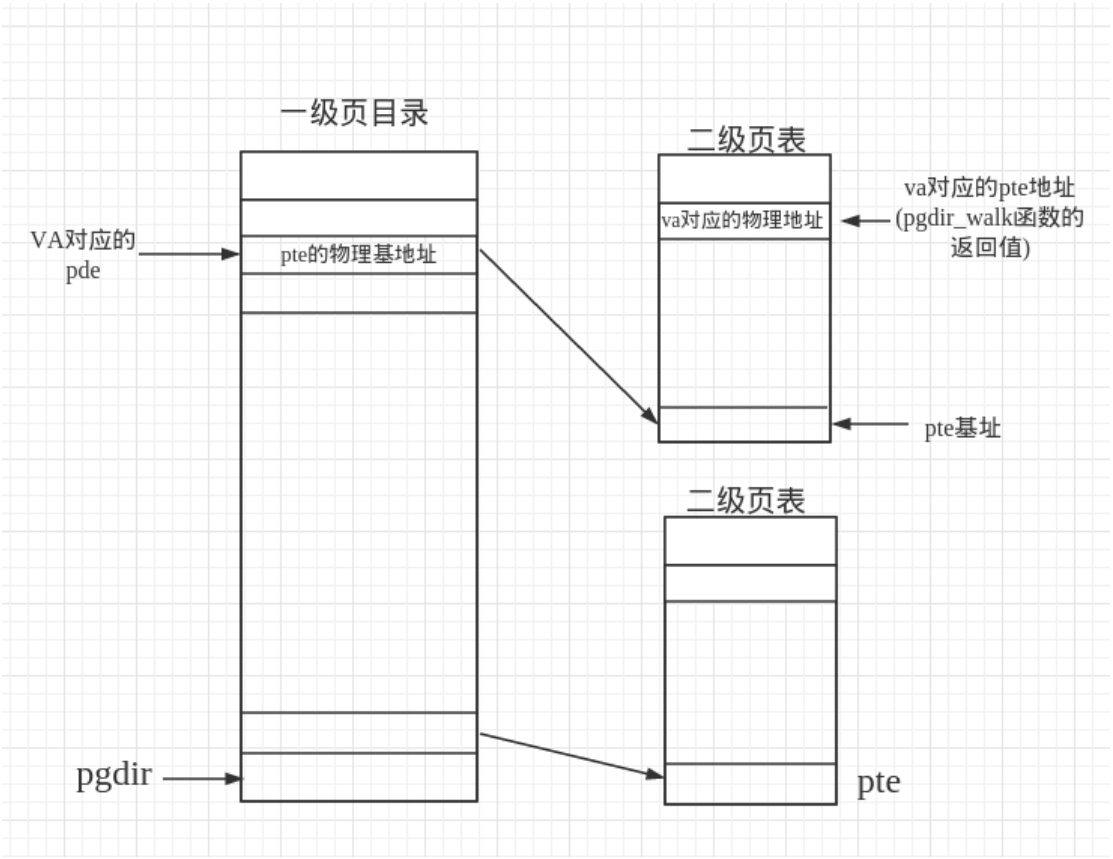
重新运行 QEMU 测试结果如下：

```
user@user-VirtualBox:~/lab1/src/lab1_2$ make qemu
+ cc kern/pmap.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
qemu -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log
VNC server running on `127.0.0.1:5900'
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
kernel panic at kern/pmap.c:128: mem_init: This function is not finished

Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

因为此时还没有完成 mem_init()函数的全部，没删除 panic("mem_init: Thisfunction is not finished\n");这句话，所以看到上边提示。

现在对 part2 部分的页面管理做一个总结：



part2 里面的各种函数其实就是对上面的二级页目录做一个管理，主要就是 va 和 pa 的映射。

pgdir_walk()函数就是返回 va 所对应的二级页表的 pte 的地址，在查询一级页目录时，如果 va 对应的 pde 里面没有存储对应的 pte 的物理基址，表明 va 还没有进行地址映射。此时，根据 creat，来决定是否需要进行二级页面的创建。若需要，则申请一个空闲的页面，作为 va 的二级页表，然后返回对应的二级页表的地址(pte 的地址)。

boot_map_region()函数就是向 va 对应的 pte 中填入对应的物理地址，以完成虚拟地址到物理地址的映射。

page_lookup()函数则是返回 va 对应的物理地址所在的页面 page。

page_remove()函数则是取消 va 和对应的 page 的映射，即是把 va 对应的 pte 中的内容清零，然后由于对应的 page 和 va 失去了映射，所以需要把 pp_ref-1.

page_insert()函数则是把 va 和要求的 page 建立映射关系，就是向 pte 里面填入 page 对应的物理地址。

综上，总的来说，part2 就是写各种使用建立二级页面的函数。

作　　业　　五

作业 5 在调用 check_page()之后，填写 mem_init()丢失的代码。

你的代码需要通过 check_kern_pgdir()和 check_installed_pgdir 的检验。

由文件我们知道剩下 3 个函数：

第一个：要求把pages结构体所在的页面和虚拟地址UPAGES相互映射。这里只要计算出pages结构体的大小，就可以通过page_insert()进行映射了。JOS 将处

理器的 32 位线性地址分为用户环境（低位地址）以及内核环境（高位地址）。该练习中主要映射了三段虚拟地址到物理页上。

UPAGES（0xef000000 ~ 0xef400000）最多 4MB

剩下的工作就是要完善 mem_init()函数，现在要完善的功能就是把关于操作系统的一些重要的地址范围映射到现在的新页目录项上 kern_pgdir 上。这里我们可以利用前面定义过的 boot_map_region 函数。

首先我们要映射的范围是把 pages 数组映射到线性地址 UPAGES，大小为一个 PTSIZE。UPAGES（0xef000000 ~ 0xef400000）最多 4MB

这是 JOS 记录物理页面使用情况的数据结构，即 exercise 1 中完成的东西，只有 kernel 能够访问。由于用户空间同样需要访问这个数据结构，我们将用户空间的一块内存映射到存储该数据结构的物理内存上。很自然联想到了 boot_map_region 这个函数。

所以我们添加的代码是：

```
        boot_map_region(kern_pgdir, (uintptr_t) UPAGES, npages*sizeof(struct PageInfo),
PADDR(pages), PTE_U | PTE_P);
```

需要注意的是目前只建立了一个页目录，即 kern_pgdir，所以第一个参数显然为 kern_pgdir。第二个参数是虚拟地址，UPAGES 本来就是以虚拟地址形式给出的。第三个参数是映射的内存块大小。第四个参数是映射到的物理地址，直接取 pages 的物理地址即可。权限 PTE_U 表示用户有权限读取。

2、内核栈（ 0xefff8000 ~ 0xf0000000）32kB

bootstack 表示的是栈地最低地址，由于栈向低地址生长，实际是栈顶。常数 KSTACKTOP = 0xf0000000，KSTKSIZE = 32kB。在此之下是一块未映射到物理内存的地址，所以如果栈溢出时，只会报错而不会覆盖数据。因此我们只用映射 [KSTACKTOP-KSTKSIZE, KSTACKTOP) 区间内的虚拟地址即可。代码如下：

```
        //     * [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".
Firefox 网络浏览器
        // Your code goes here:
oot_map_region(kern_pgdir, (uintptr_t) (KSTACKTOP-KSTKSIZE), KSTKSIZE,
PADDR(bootstack), PTE_W | PTE_P);PET_P);
        //////////////////////////////////////////////////////////////////////
        // 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
```

3、内核（ 0xf0000000 ~ 0xffffffff ）256MB

之前在 lab1 中，通过 kernel/entrypgdir.c 映射了 4MB 的内存地址，这里需要映射全部 0xf0000000 至 0xffffffff 共 256MB 的内存地址。代码如下：

```
        //      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, (uintptr_t) KERNBASE, ROUNDUP(0xffffffff - KERNBASE, PGSIZE),
0, PTE_W | PTE_P);
        // Check that the initial page directory has been set up correctly.
        check_kern_pgdir();

        // Switch from the minimal entry page directory to the full kern_pgdir
```

这里的 size 参数做了 roundup，也就是说从 0x0fffffff 变为了 0x10000000。在 boot_map_region 中，再利用 va + size，显然会溢出得 0。即 boot_map_region 中的 for 循环一开始就判断 va > end_addr。这是显然的，因为 `end_addr = 0xf0000000 + 0x1000000 = 0x00000000`。
因此，实际上 boot_map_region 的更佳实现是直接用页数，避免溢出。

最终运行检验如下：

```
user@user-VirtualBox:~/lab1/src/lab1_2$ make qemu
+ cc kern/pmap.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
qemu -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log
VNC server running on `127.0.0.1:5900'
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
EAX=f08d6ef8 EBX=ef7be000 ECX=00000000 EDX=000008d6
ESI=ff8627f8 EDI=008d7005 EBP=f0114f78 ESP=f0114f40
EIP=f010105a EFL=00000006 [-----P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
```

调试成功。

问题 4：

问题 4

1)在这一点上页目录中的哪些行已经被填写了？他们映射了什么地址，指向了哪？换言之，尽量填写以下表：

Entry	Base Virtual Address	Points to (logically):
1023	?	Page table for top 4MB of phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	[see next question]

2)我们已经将内核和用户环境放在了相同的地址空间。为什么用户程序不能读或者写内核内存？什么样的具体机制保护内核内存？

3)这个操作系统最大能支持多大的物理内存？为什么？

4)管理内存有多大的空间开销，如果我们拥有最大的物理内存？这个空间开销如何减小？

解答：

Entry	Base virtual address	Points to(logically)
1023	0xffc0000	Page table for top 4MB of phys memory
1022	0xff80000	Page table for 248MB—(252MB-1)phys mem
…	…	…
960	0xf000000	Page table for kernel code & static data 0—(4MB-1)phys mem
959	0xefc0000	Page directory self
958	0xef80000	ULIM
957	0xef40000	State register
956	0xef00000	UPAGES, page table for struct Pages[]
…	…	…
2	0x0080000	NULL
1	0x0040000	NULL
0	0x0000000	Same as 960

2) 页表和页目录有PTE_U位，可以来控制用户是否可以访问某页。

3) pages 这个数组只能占用最多 4MB 的空间，而每个 PageInfo 占用 8Byte，也就是说最多只能有512k页，每页容量4kB，总共最多 2GB

4）在lab1_2/inc/mmu.h中可以看到如下内容：

```
// construct linear address from indexes and offset
#define PGADDR(d, t, o) ((void*) ((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))

// Page directory and page table constants.
#define NPDENTRIES      1024            // page directory entries per page
directory
#define NPTENTRIES      1024            // page table entries per page table

#define PGSIZE          4096            // bytes mapped by a page
#define PGSHIFT         12              // log2(PGSIZE)

#define PTSIZE          (PGSIZE*NPTENTRIES) // bytes mapped by a page directory
entry
#define PTSHIFT         22              // log2(PTSIZE)

#define PTXSHIFT        12              // offset of PTX in a linear address
#define PDXSHIFT        22              // offset of PDX in a linear address
```

由此我们知道：

PageInfo开销：4MB（一个PageInfo Object代表一个物理页）

页表开销：2MB ，一个页表有1K个页表项,每个页表项是4Bytes，即每个页表是4KB（一页的大小）,共有页表项512K项（对应512K物理页），需要512*K*1*K*/=512个页表，故空间开销是512×4KB=2MB；

页目录开销：4KB ，一个页目录项对应一个页表，512个页表只需一个页目录表即可，一个页目录有1K个页目录项,每个页目录项是4Bytes ，即每个页目录是4KB（一页的大小），故空间开销是4KB；

空间总开销:6MB+4KB

减小开销的方法：使用多级页表