实验报告

Lab3

姓名：朱瑞媛

学号：14307130373

注意事项：

1.请在每个 exercise 之后简要叙述实验原理，详细描述实验过程。
2.请将你认为的关键步骤附上必要的截图。
3.有需要写代码的实验，必须配有代码、注释以及对代码功能的说明。
4.你还可以列举包括但不局限于以下方面：实验过程中碰到的问题、你是如何解决的、实验之后你还留有哪些疑问和感想。

5.请在截止日期前将代码和报告上传到 ftp 的指定目录下，文件名为 os_lab3_学号.zip，该压缩文件中应包含实验报告和代码，其中实验报告格式为 pdf，置于压缩文件的根目录。
6.如果实验附有 question，请在对应 exercise 后作答，这是实验报告评分的重要部分。
7.Challenge 为加分选作题。每个 lab 可能有多个 challenge，我们会根据完成情况以及难度适当加分，这部分的实验过程描述应该比 exercise 更加详细。

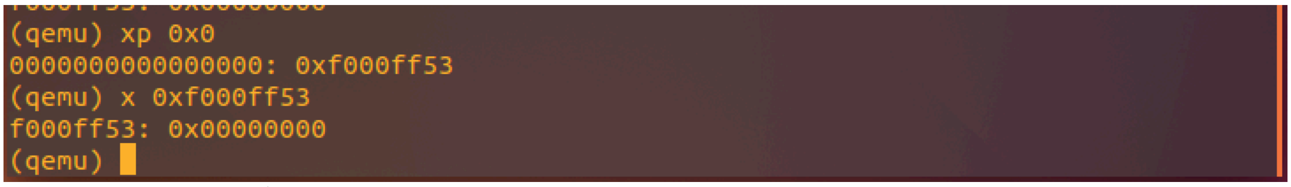
Exercise 1. Look at chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. Read the sections about page translation and page-based protection closely (5.2 and 6.4). We recommend that you also skim the sections about segmentation; while JOS uses paging for virtual memory and protection, segment translation and segment-based protection cannot be disabled on the x86, so you will need a basic understanding of it.

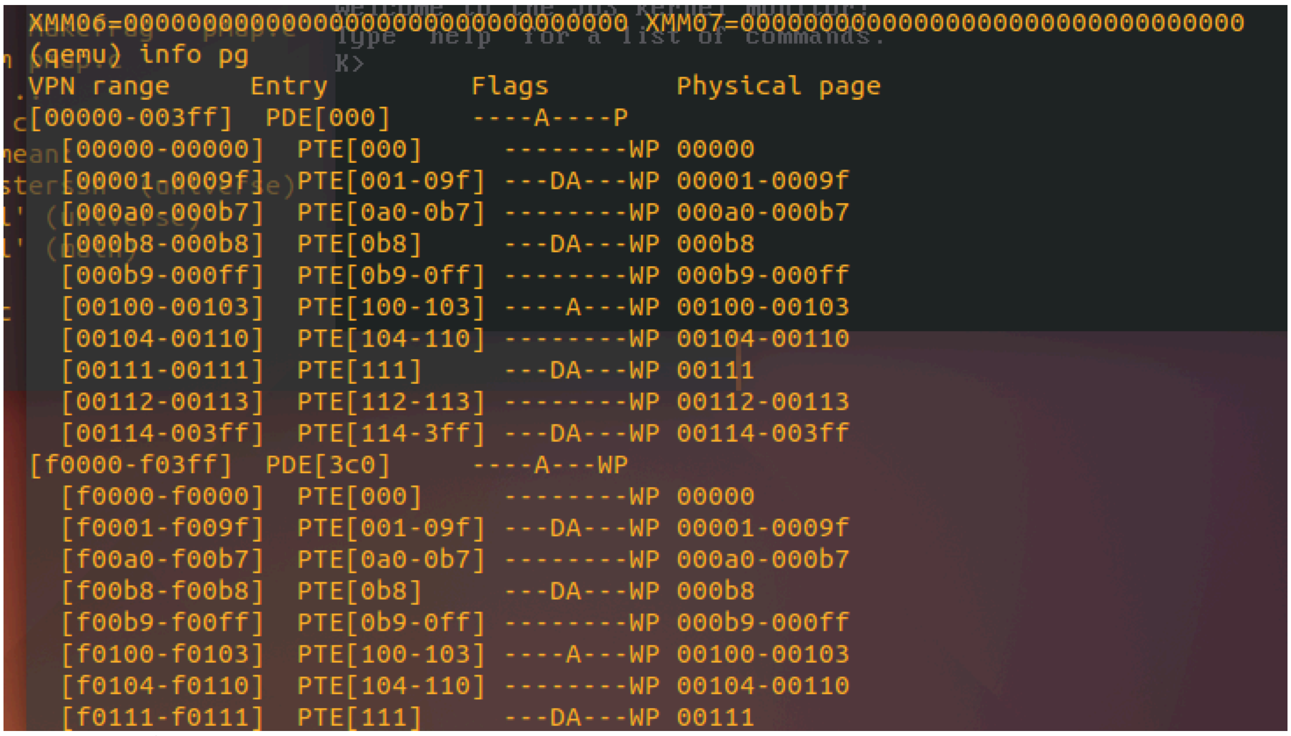
已认真阅读手册关于分段及分页的章节。

Exercise 2. 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.

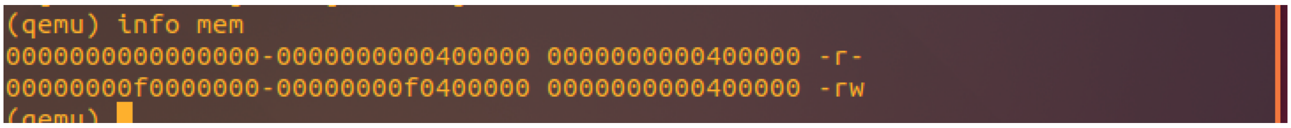
Our patched version of QEMU provides an `info pg` command that may also prove useful: it shows a compact but detailed representation of the current page tables, including all mapped memory ranges, permissions, and flags. Stock QEMU also provides an `info mem` command that shows an overview of which ranges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


```
(qemu) xp 0x0
0000000000000000: 0xf000ff53
(qemu) x 0xf000ff53
f000ff53: 0x00000000
(qemu) 
```
可以看出物理地址和虚拟地址的对应。



```
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
(qemu) info pg
VPN range     Entry         Flags        Physical page
[00000-003ff]  PDE[000]     ----A----P
  [00000-00000]  PTE[000]     --------WP 00000
  [00001-0009f]  PTE[001-09f] ---DA---WP 00001-0009f
  [000a0-000b7]  PTE[0a0-0b7] --------WP 000a0-000b7
  [000b8-000b8]  PTE[0b8]     ---DA---WP 000b8
  [000b9-000ff]  PTE[0b9-0ff] --------WP 000b9-000ff
  [00100-00103]  PTE[100-103] ----A---WP 00100-00103
  [00104-00110]  PTE[104-110] --------WP 00104-00110
  [00111-00111]  PTE[111]     ---DA---WP 00111
  [00112-00113]  PTE[112-113] --------WP 00112-00113
  [00114-003ff]  PTE[114-3ff] ---DA---WP 00114-003ff
[f0000-f03ff]  PDE[3c0]     ----A---WP
  [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-f0103]  PTE[100-103] ----A---WP 00100-00103
  [f0104-f0110]  PTE[104-110] --------WP 00104-00110
  [f0111-f0111]  PTE[111]     ---DA---WP 00111
```

查看页表信息。



```
(qemu) info mem
0000000000000000-0000000000400000 0000000000400000 -r-
00000000f0000000-00000000f0400000 0000000000400000 -rw
(qemu) 
```

查看被页表映射的虚拟地址。

Question

1. Assuming that the following JOS kernel code is correct, what type should variable x have, uintptr_t or physaddr_t?

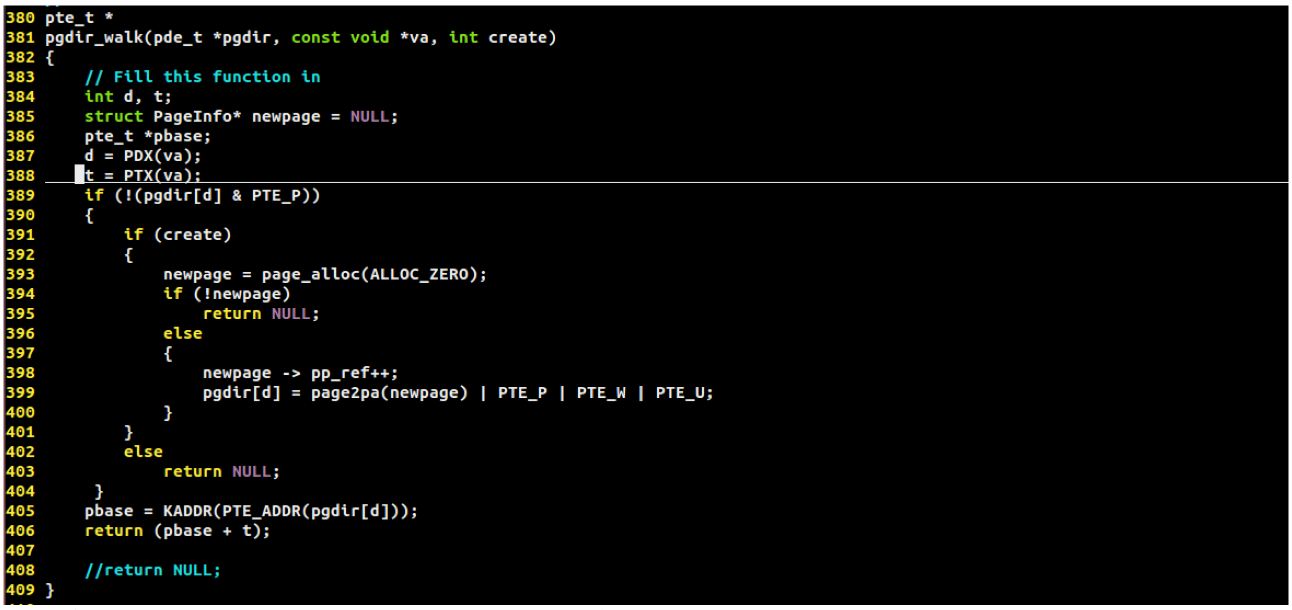
```
2.   mystery_t x;
3.   char* value = return_a_pointer();
4.   *value = 10;
     x = (mystery_t) value;
```

C 语言中所有使用了 * 操作符解析地址的变量 x 都应该是虚拟地址， 所以 x 是 uintptr_t 类型。

Exercise 3. In the file kern/pmap.c, you must implement code for the following functions.

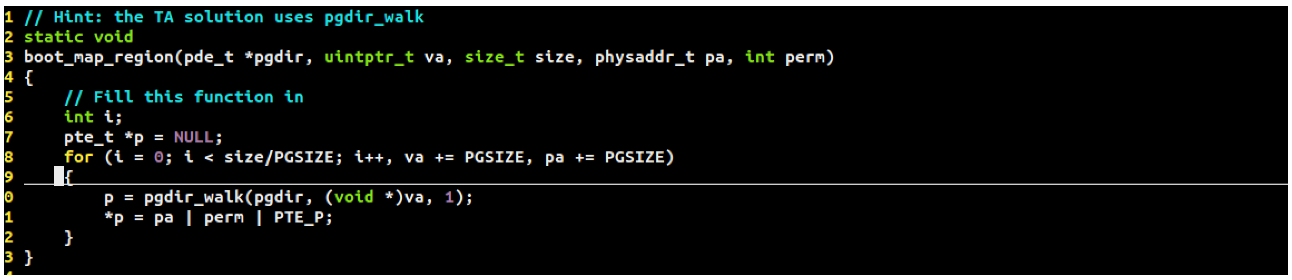
```
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

check_page(), called from mem_init(), tests your page table management routines. You should make sure it reports success before proceeding.



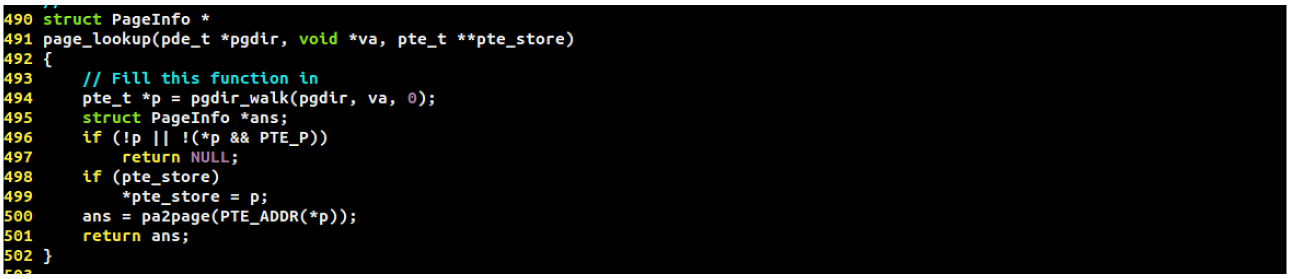
```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    // Fill this function in
    int d, t;
    struct PageInfo* newpage = NULL;
    pte_t *pbase;
    d = PDX(va);
    t = PTX(va);
    if (!(pgdir[d] & PTE_P))
    {
        if (create)
        {
            newpage = page_alloc(ALLOC_ZERO);
            if (!newpage)
                return NULL;
            else
            {
                newpage -> pp_ref++;
                pgdir[d] = page2pa(newpage) | PTE_P | PTE_W | PTE_U;
            }
        }
        else
            return NULL;
     }
    pbase = KADDR(PTE_ADDR(pgdir[d]));
    return (pbase + t);

    //return NULL;
}
```

对于给定的页目录指针 pgdir，pgdir_walk()函数返回线性地址 va 的页表项指针。
在页目录表中求得这个虚拟地址所在的页表页对于与页目录中的页目录项地址是否已经在内存中。
如果不在内存中，
如果 create == 0，返回 NULL。
其他情况下，分配新页并将其添加至页目录项，并返回对应页表项地址。
如果在内存中，计算页表页基址，返回基址加上偏移量。



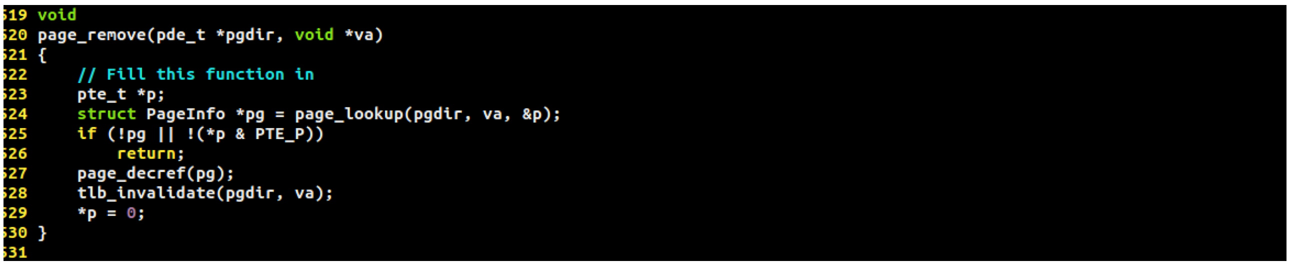
```
// Hint: the TA solution uses pgdir_walk
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
    // Fill this function in
    int i;
    pte_t *p = NULL;
    for (i = 0; i < size/PGSIZE; i++, va += PGSIZE, pa += PGSIZE)
    {
        p = pgdir_walk(pgdir, (void *)va, 1);
        *p = pa | perm | PTE_P;
    }
}
```

boot_map_region()函数把虚拟地址空间范围[va, va+size]映射到物理空间[pa, pa+size]的映射关系加入到页表 pgdir 中。
通过 for 循环遍历所有 size/PGSIZE 页，将每个虚拟页和物理页的对应关系加入相应的页表项。



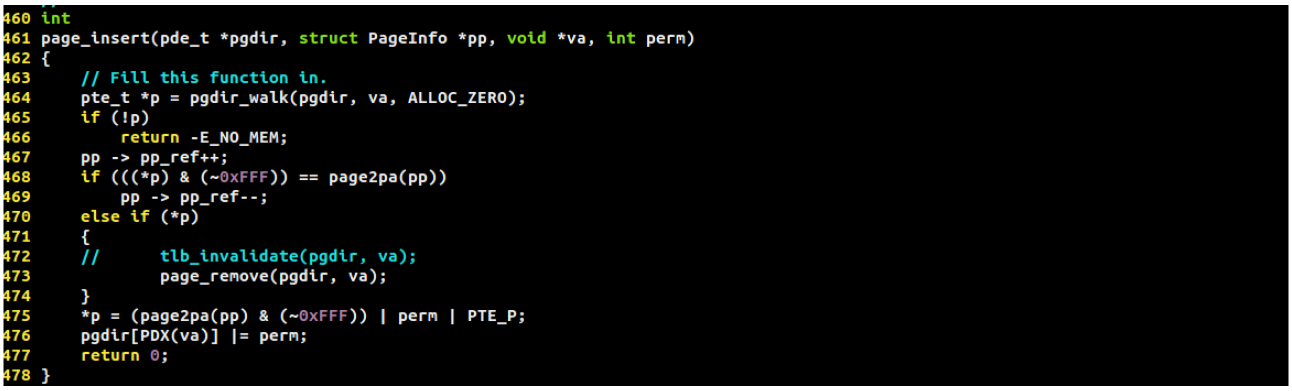
```c
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // Fill this function in
    pte_t *p = pgdir_walk(pgdir, va, 0);
    struct PageInfo *ans;
    if (!p || !(*p && PTE_P))
        return NULL;
    if (pte_store)
        *pte_store = p;
    ans = pa2page(PTE_ADDR(*p));
    return ans;
}
```

返回虚拟地址 va 所映射的物理页的 PageInfo 结构体的指针，如果 pte_store 参数不为 0，则把这个物理页的页表项地址存放在 pte_store 中。
调用 pgdir_walk 函数获取这个 va 对应的页表项，判断这个页是否已经在内存中，如果在则返回这个页的 PageInfo 结构体指针。并且把这个页表项的内容存放到 pte_store 中。



```c
void
page_remove(pde_t *pgdir, void *va)
{
    // Fill this function in
    pte_t *p;
    struct PageInfo *pg = page_lookup(pgdir, va, &p);
    if (!pg || !(*p & PTE_P))
        return;
    page_decref(pg);
    tlb_invalidate(pgdir, va);
    *p = 0;
}
```

page_remove()函数功能是把虚拟地址 va 和物理页的映射关系删除。
调用 page_decref()函数，使得每次删除后 pp_ref 减 1，且当减为 0 时 free 此页。
将此页页表项置 0.



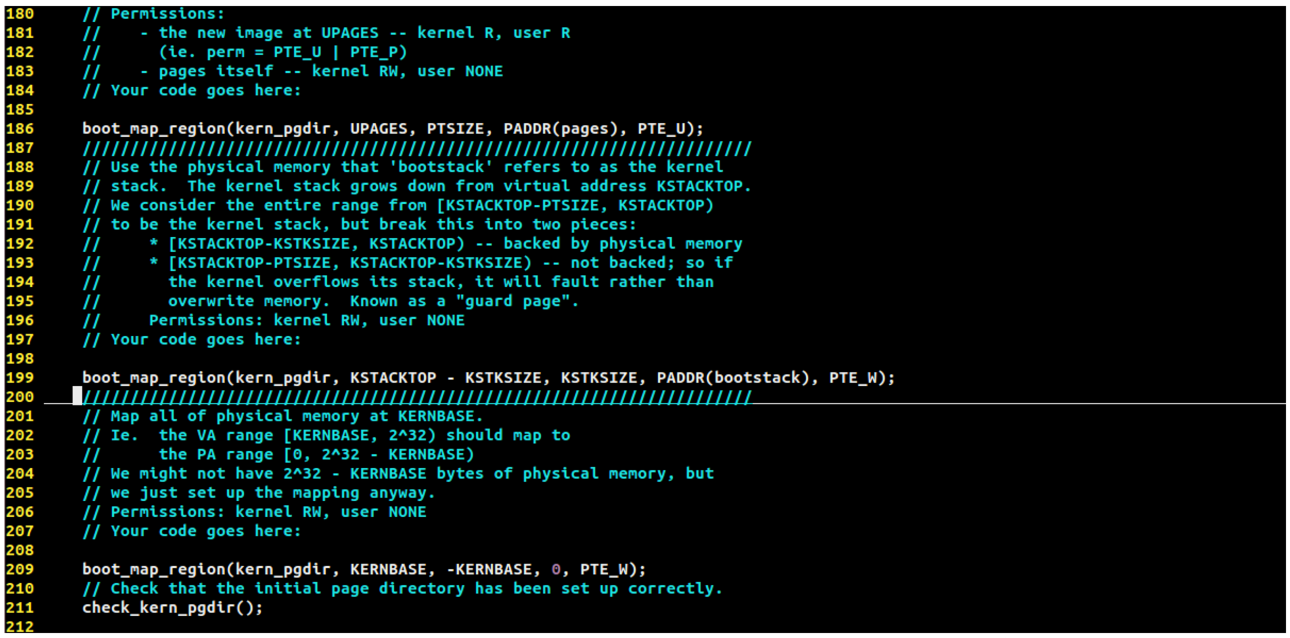
```c
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    // Fill this function in.
    pte_t *p = pgdir_walk(pgdir, va, ALLOC_ZERO);
    if (!p)
        return -E_NO_MEM;
    pp -> pp_ref++;
    if (((*p) & (~0xFFF)) == page2pa(pp))
        pp -> pp_ref--;
    else if (*p)
    {
    //      tlb_invalidate(pgdir, va);
            page_remove(pgdir, va);
    }
    *p = (page2pa(pp) & (~0xFFF)) | perm | PTE_P;
    pgdir[PDX(va)] |= perm;
    return 0;
}
```

page_insert()函数的功能是把一个物理内存中页 pp 与虚拟地址 va 建立映射关系。
调用 pgdir_walk()函数取出 va 对应的物理页的页表项地址。如果没有对应物理页则为其分配。
如果无法分配则返回-E_NO_MEM.
将对应的 pp_ref 加 1.
如果 va 对应的物理页是指定的物理页，则将前面加 1 再减去。
如果 va 对应的物理页是其他的物理页，将这个对应关系取消。
然后将其对应到指定物理页上。

运行后，通过 check_page()函数测验。

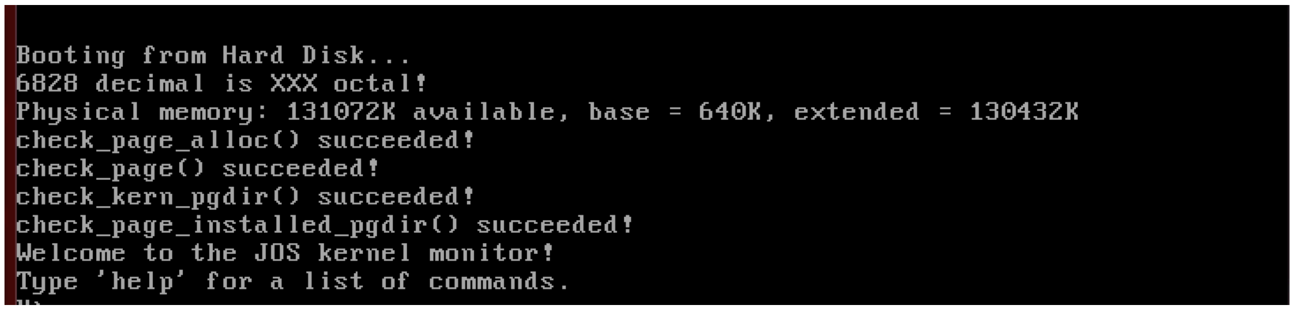
Exercise 4. Fill in the missing code in mem_init() after the call to check_page().

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



```
    // 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);
    //////////////////////////////////////////////////////////////////////
    // 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);
    //////////////////////////////////////////////////////////////////////
    // 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, -KERNBASE, 0, PTE_W);
    // Check that the initial page directory has been set up correctly.
    check_kern_pgdir();
```

mem_init()函数要完善的功能就是利用前面定义过的 boot_map_region 函数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。
第一处要映射的范围是把 pages 数组映射到线性地址 UPAGES，大小为一个 PTSIZE。
这部分空间是 kernel space 和 user space 中的代码都能访问的，所以要设置 PTE_U。
第二处是映射内核的堆栈区域，把由 bootstack 变量所标记的物理地址范围映射给内核的堆栈。[KSTACKTOP-KSTKSIZE, KSTACKTOP] 这部分映射关系加入的页表中。这部分地址的访问权限是，kernel space 可以读写，user space 无权访问，设置为 PTE_W。
第三处映射整个操作系统内核，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，设置为 PTE_W。



```
Booting from Hard Disk...
6828 decimal is XXX octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

运行后，通过 check_kern_pgdir(),check_page_installed_pgdir()函数测验。

Question

2. What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

Entry	Base Virtual Address	Points to (logically):
1023	0x02800000	Page table for top 4MB of phys memory
1022	0x02400000	0x02000000
.	0x02000000	0x01600000

.	0x01600000	0x01200000
.	0x01200000	0x00800000
2	0x00800000	0x00400000
1	0x00400000	0x00000000
0	0x00000000	[see next 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?

2. 3BD 号页目录项，指向的是 kern_pgdir。
    3BC 号页目录项，指向的是 pages 数组。
    3BF 号页目录项，指向的是 bootstack。
    3C0～3FF 号页目录项，指向的是 kernel。
3. 用户程序不能去随意修改内核中的代码，数据，否则可能会破坏内核，造成程序崩溃。
   目前通过分页机制来实现，通过把页表项中的 Supervisor/User 位置 0，那么用户态的代码就不能访问内存中的这个页。
4. 由于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。
5.page 结构体大小为 UPAGES 大小 4MB。页目录表为 4KB。当前页表存储需要 2MB。
6. 在 entry.S 文件中有一个指令 jmp *%eax，这个指令要完成跳转，就会重新设置 EIP 的值，把它设置为寄存器 eax 中的值，而这个值是大于 KERNBASE 的，所以就完成了 EIP 从小的值到大于 KERNBASE 的值的转换。在 entry_pgdir 这个页表中，也把虚拟地址空间[0, 4MB)映射到物理地址空间[0, 4MB)上，所以当访问位于[0, 4MB]之间的虚拟地址时，可以把它们转换为物理地址。

实验感想：由于本次实验难度有提升，加上最近正值期中，所以 challenge 部分并没有完成，希望助教体谅=w=

Challenge! We consumed many physical pages to hold the page tables for the KERNBASE mapping. Do a more space-efficient job using the PTE_PS ("Page Size") bit in the page directory entries. This bit was *not* supported in the original 80386, but is supported on more recent x86 processors. You will therefore have to refer to Volume 3 of the current Intel manuals. Make sure you design the kernel to use this optimization only on processors that support it!

Challenge! Extend the JOS kernel monitor with commands to:

- Display in a useful and easy-to-read format all of the physical page mappings (or lack thereof) that apply to a particular range of virtual/linear addresses in the currently active address space. For example, you might enter '`showmappings 0x3000 0x5000`' to display the physical page mappings and corresponding permission bits that apply to the pages at virtual addresses 0x3000, 0x4000, and 0x5000.
- Explicitly set, clear, or change the permissions of any mapping in the current address space.
- Dump the contents of a range of memory given either a virtual or physical address range. Be sure the dump code behaves correctly when the range extends across page boundaries!
- Do anything else that you think might be useful later for debugging the kernel. (There's a good chance it will be!)

Challenge! Write up an outline of how a kernel could be designed to allow user environments unrestricted use of the full 4GB virtual and linear address space. Hint: the technique is sometimes known as "*follow the bouncing kernel*." In your design, be sure to address exactly what has to happen when the processor transitions between kernel and user modes, and how the kernel would accomplish such transitions. Also describe how the kernel would access physical memory and I/O devices in this scheme, and how the kernel would access a user environment's virtual address space during system calls and the like. Finally, think about and describe the advantages and disadvantages of such a scheme in terms of flexibility, performance, kernel complexity, and other factors you can think of.

Challenge! Since our JOS kernel's memory management system only allocates and frees memory on page granularity, we do not have anything comparable to a general-purpose `malloc/free` facility that we can use within the kernel. This could be a problem if we want to support certain types of I/O devices that require *physically contiguous* buffers larger than 4KB in size, or if we want user-level environments, and not just the kernel, to be able to allocate and map 4MB *superpages* for maximum processor efficiency. (See the earlier challenge problem about PTE_PS.)

Generalize the kernel's memory allocation system to support pages of a variety of power-of-two allocation unit sizes from 4KB up to some reasonable maximum of your choice. Be sure you have some way to divide larger allocation units into smaller ones on demand, and to coalesce multiple small allocation units back into larger units when possible. Think about the issues that might arise in such a system.