JOS 2016 Lab 2实习报告

陈一茹 1400012976

October 17, 2016

Contents

1 Introduction

这个lab主要是对内存的管理。

第一部分是内核的物理内存分配器。 第二个是虚拟内存的管理。

2 Getting started

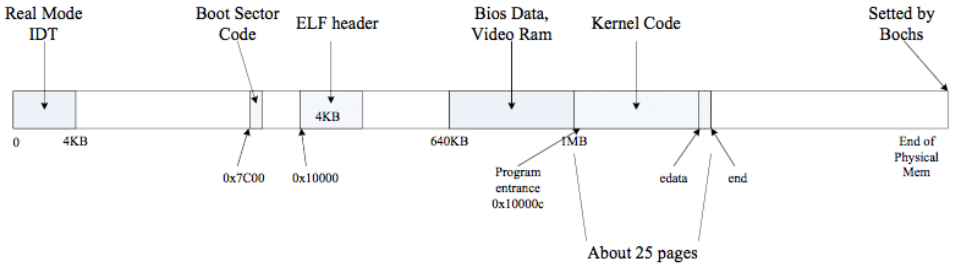
这部分我就是照着说明一步步做的。有一点要注意的是因为我没有athena的权限，所以我就没有办法的merge，这样会导致之后没有办法"make"。这时就需要手动修改一些文件，我就是修改了"conf/lab.mk"和"GNUmakefile"之后，将原来关于lab1的内容变成关于lab2的， 这样就没有问题了。

这部分的指导还告诉我，添加了几个文件：

```
1 inc/memlayout.h
2 kern/pmap.c
3 kern/pmap.h
4 kern/kclock.h
5 kern/kclock.c
```

inc/memlayout.h这个是虚拟内存的分布图。
kern/pmap.c:这个是我在lab2中主要关注和实现的文件。
这里还是回顾一下lab1吧，从开机到现在做了：

- BIOS首先执行，进行有关初始化。
- BIOS将512byte的Boot Loader加载进入内存0x7c00，然后用jmp跳转到0x7c00的位置，将控制权交给Boot Loader。
- Boot Loader首先执行的是boot.S，然后是boot/main.c中的bootmain()，继续进行有关初始化设置。
- Boot Loader中的代码开始执行后，会从磁盘上紧接着自己的第 2 个扇区开始，一直读 8 个扇 区的内容(一共是 8×512=4KB，ELF 头的大小)到 0x10000(64KB)的地方，然后，通过 对 ELF 头的解析，得到 kernel 模块编译出来后所占的大小，并将 kernel 读到物理内存 0x100000(1MB)开始的地方
- 设置好GDT之后，JOS内核首先执行的是entry.S，然后调用kern/init.c中的i386 init()，在这个函数中，i386 init()调用cons init()初始化控制台，还对显示器，键盘等硬件进行初始化。JOS内核目前的主要流程都在这个函数中。
- 然后调用kern/pmap.c中的mem init()进行存储管理的有关设置，进行了内存初始化——这个函数是lab2的主要实现对象。
- 最后调用monitor()启动命令行，等待用户从终端输入的命令。



这是i386 init()之前的内存分布。大概能够说明这些流程对内存的造成的影响。

3 Part 1: Physical Page Management

这部分关注的是物理页面的管理，暂时和页面转换机制没有什么关系。

JOS在管理内存的时候，以页为最小单位进行管理。首先我要做的就是以页来划分，然后建立一些数据结构对其管理。

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.

这个部分只有一个Exercise，描述非常的简练，看来要自己探索了。

3.1 boot_alloc()

In the file kern/pmap.c,

```
1 // This simple physical memory allocator is used only while JOS
      is setting
2 // up its virtual memory system.  page_alloc() is the real
     allocator.
3 //
4 // If n>0, allocates enough pages of contiguous physical memory
      to hold 'n'
```

```
// bytes.  Doesn't initialize the memory.  Returns a kernel virtual address.
//
// If n==0, returns the address of the next free page without allocating
// anything.
//
// If we're out of memory, boot_alloc should panic.
// This function may ONLY be used during initialization,
// before the page_free_list list has been set up.
static void *
boot_alloc(uint32_t n)
```

先看boot_alloc的函数。

boot_alloc()是用于JOS建立虚拟内存管理的时候分配内存的， 当JOS虚拟内存系统完善的时候，就不用这个函数了。而page_alloc（）才是真正的内存分配器。

boot_alloc()的两个用法： :在n为0时返回 下一个可用的空闲页面地址，当n大于0时返回足够容纳n个字节的连续页面— —返回的值是一个核的虚拟地址。

注意， 两者地址都是虚拟地址! 由lab 1，我已经知道了链接地址（虚拟地址）和物理地址是不同的。

PGSIZE是定义在inc/mmu.h中的，为一个物理页的大小4KB = 4096B。PTSIZE，为一个页表对应实际物理内存的 大小，即1024 * 4KB = 4MB

从代码的注释中，我可以知道，可用内存指针nextfree是从end, 也就是 紧接着内核之后开始的。 这个可用内存的计算就在boot_alloc()函数的上方，是由 i386_detect_memory(void) 计算得到的，计算得到值放在全局变量npages中。这就规定了整个可用的物理内存不能够超过这个值。而nextfree由于是C语言的指 针，所以其中的值是一个虚拟地址;而npages* PGSIZE表示的可用内存上限，是一个物理地址。所以，在对二者进行比较时需要进行一次转化，这里我做的是使用PADDR宏将nextfree转换到对应的物理地址。PADDR宏定义 在kern/pmap.h中。同样的，还存在将物理地址转化为虚拟地址的KADDR宏。

注释里面还提到一个panic() 函数，经查找， 这是用来出错时显示调试信息的，这样，代码就不难写了：

```
// 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.
if (n == 0) return (void*)nextfree;
n = ROUNDUP(n, PGSIZE);
if (PADDR(nextfree + n) > npages * PGSIZE)
{
panic("kern/pmap.c: boot_alloc()");
return NULL;
}
```

```
13   result = nextfree;
14   nextfree += n;
15   return result;
```

3.2 mem_init(void)

接下来是mem_init(void)函数，这个函数是用来初始化内核部分的地址空间，

这个函数用到一个pages数组，

struct PageInfo *pages; // Physical page state array

由此可知，这个数组类型是PageInfo，用上述已经实现的boot_alloc来实现这个函数，大小是 npages * sizeof(struct PageInfo)。至此，函数已经可以填写了：

```
1 //
      ////////////////////////////////////////////////////////////////////
2   // Allocate an array of npages 'struct PageInfo's and store
        it in 'pages'.
3   // The kernel uses this array to keep track of physical pages
        : for
4   // each physical page, there is a corresponding struct
        PageInfo in this
5   // array.  'npages' is the number of physical pages in memory
        .  Use memset
6   // to initialize all fields of each struct PageInfo to 0.
7   // Your code goes here:
8
9   pages = (struct PageInfo*)boot_alloc(npages * sizeof(struct
        PageInfo));
10  memset(pages, 0, npages * sizeof(struct PageInfo)));
```

为了增进对PageInfo 的了解，我又找了找PageInfo的位置，在inc/memorylayout.h中：

```
1 /*
2  * Page descriptor structures, mapped at UPAGES.
3  * Read/write to the kernel, read-only to user programs.
4  *
5  * Each struct PageInfo stores metadata for one physical page.
6  * Is it NOT the physical page itself, but there is a one-to-
       one
7  * correspondence between physical pages and struct PageInfo's.
8  * You can map a struct PageInfo * to the corresponding
       physical address
9  * with page2pa() in kern/pmap.h.
10 */
11 struct PageInfo {
12   // Next page on the free list.
13   struct PageInfo *pp_link;
14
```

```
15   // pp_ref is the count of pointers (usually in page table
         entries)
16   // to this page, for pages allocated using page_alloc.
17   // Pages allocated at boot time using pmap.c's
18   // boot_alloc do not have valid reference count fields.
19
20   uint16_t pp_ref;
21 };
```

每个PageInfo结构存储一个物理页面的信息， PageInfo 有两个变量，一个是*pp_link，显然是用来形成链表的，然后pp_ref是用来计算引用的次数，注释中还说了，这个引用次数只能计算用page_alloc分配的页，不能计算boot_alloc分配的。

注： page2pa这个函数可以计算她的地址，我发现PageInfo中并没有一个域用来存放它所对应的页面的地址，这是因为在JOS中，一个PageInfo对象所对应的页面可以通过它在pages数组中的下标来获知，所以，pages[0]对应的页面自然就是第1个页面，地址就 是0*PGSIZE;pages[1]对应的页面就是第2个页面，地址是1*PGSIZE，以此类 推，可以直接计算得到，所以页没有必要存在一个域来存放这个地址。这里的page2pa()宏就是完成这一工作的。

这是我发现，有些宏还是很好用的，所以我再回去看一下mmu.h的宏，以方便我之后写代码。事实证明之后还真是用到了。

```
1  // A linear address 'la' has a three-part structure as follows:
2  //
3  // +--------10------+-------10-------+---------12----------+
4  // | Page Directory |   Page Table    | Offset within Page  |
5  // |      Index     |      Index      |                     |
6  // +----------------+-----------------+---------------------+
7  //  \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
8  //  \---------- PGNUM(la) ----------/
9  //
10 // The PDX, PTX, PGOFF, and PGNUM macros decompose linear
       addresses as shown.
11 // To construct a linear address la from PDX(la), PTX(la), and
       PGOFF(la),
12 // use PGADDR(PDX(la), PTX(la), PGOFF(la)).
13
14 // page number field of address
15 #define PGNUM(la) (((uintptr_t) (la)) >> PTXSHIFT)
16
17 // page directory index
18 #define PDX(la)   ((((uintptr_t) (la)) >> PDXSHIFT) & 0x3FF)
19
20 // page table index
21 #define PTX(la)   ((((uintptr_t) (la)) >> PTXSHIFT) & 0x3FF)
22
23 // offset in page
24 #define PGOFF(la) (((uintptr_t) (la)) & 0xFFF)
25
26 // construct linear address from indexes and offset
```

```
27 #define PGADDR(d, t, o) ((void*) ((d) << PDXSHIFT | (t) <<
       PTXSHIFT | (o)))
```

这部分直接看注释就好了，就不加赘述了。

3.3 page_init()

接下来我实现page_init()函数，

注释里面摘要一点重点： page_init()之后，不要再用boot_alloc()函数，因为不再分配内核的空间了，替代的，我要使用的是page的allocator来分配和释放物理内存通过 page_free_list。

另外并不是所有内存都是自由的，以下的4部分内存是不可以分配的：

- 第0页 ： 这部分要用来保存实模式的IDT， 和BIOS的结构
- IO hole [IOPHYSMEM, EXTPHYSMEM)， 为了向前兼容所需要保存的 IO hole。
- 存放JOS内核的部分内容:从0x00100000开始。还有存页表和其它数据结构的空间。
- 其他已经被分配的内存。

在inc/memorylayout.h中可以 查到EXTPHYSMEM的值就是0x00100000，所以第1段和第2段是前后相邻的; 由于截至到目前为止分配内存都是通过boot alloc()进行的，而boot alloc()是 从紧接着JOS内核之后的部分开始分配内存的，所以第2段和第3段也相邻。 而综上所述，这不可分配的3部分在内存中是相邻的，可以当作一段来处 理，从IOPHYSMEM开始，直到被boot alloc()分配出去的最后一字节内存， 即boot alloc()中的nextfree的前一字节结束。这些弄清楚代码就很好写了:

```
1 //
2 // Initialize page structure and memory free list.
3 // After this is done, NEVER use boot_alloc again.  ONLY use
      the page
4 // allocator functions below to allocate and deallocate
      physical
5 // memory via the page_free_list.
6 //
7 void
8 page_init(void)
9 {
10   // The example code here marks all physical pages as free.
11   // However this is not truly the case.  What memory is free?
12   //  1) Mark physical page 0 as in use.
13   //     This way we preserve the real-mode IDT and BIOS
       structures
14   //     in case we ever need them.  (Currently we don't, but
       ...)
```

```
15   //  2) The rest of base memory, [PGSIZE, npages_basemem *
         PGSIZE)
16   //      is free.
17   //  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which
         must
18   //      never be allocated.
19   //  4) Then extended memory [EXTPHYSMEM, ...).
20   //      Some of it is in use, some is free. Where is the
         kernel
21   //      in physical memory?  Which pages are already in use
         for
22   //      page tables and other data structures?
23   //
24   // Change the code to reflect this.
25   // NB: DO NOT actually touch the physical memory
         corresponding to
26   // free pages!
27   size_t i;
28       assert(page_free_list == 0);
29       unsigned used_top = PADDR(boot_alloc(0));
30       for (i = 0; i < npages; i++) {
31     if (i == 0 || (page2pa(&pages[i]) >= IOPHYSMEM && page2pa(&
           pages[i]) < used_top))
32       continue;
33   pages[i].pp_ref = 0;
34   pages[i].pp_link = page_free_list;
35   page_free_list = &pages[i];
36     }
37 }
```

这样这个函数就算完成了。

3.4 page_alloc()

这里有提示要用到 page2kva()，它和page2pa()很相似，是将一个PageInfo转换为 对应的页面的虚拟地址。

```
1 //
2 // Allocates a physical page.  If (alloc_flags & ALLOC_ZERO),
      fills the entire
3 // returned physical page with '\0' bytes.  Does NOT increment
      the reference
4 // count of the page - the caller must do these if necessary (
      either explicitly
5 // or via page_insert).
6 //
7 // Be sure to set the pp_link field of the allocated page to
      NULL so
8 // page_free can check for double-free bugs.
9 //
10 // Returns NULL if out of free memory.
11 //
12 // Hint: use page2kva and memset\_
```

```
13 struct PageInfo *
14 page\_alloc(int alloc\_flags)
15 {
16   // Fill this function in
17   if (alloc_flags & ALLOC_ZERO)
18     memset(page2kva(ret), 0, PGSIZE);
19   ret->pp_link = NULL;
20   return ret;
21 }
```

不难实现。

3.5 page_free()

```
1 // Return a page to the free list.
2 // (This function should only be called when pp->pp_ref reaches
      0.)
3 //
4 void
5 page_free(struct PageInfo *pp)
6 {
7   // Fill this function in
8   // Hint: You may want to panic if pp->pp_ref is nonzero or
9   // pp->pp_link is not NULL.
10  if(pp->pp_ref == 0){
11        pp->pp_link = page_free_list;
12        page_free_list = pp;
13  }
14  else
15  {
16    panic("pp->pp_ref is not zero. Wrong call of the page_free
          !!!");
17  }
18 }
```

4 Part 2: Virtual Memory

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 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.

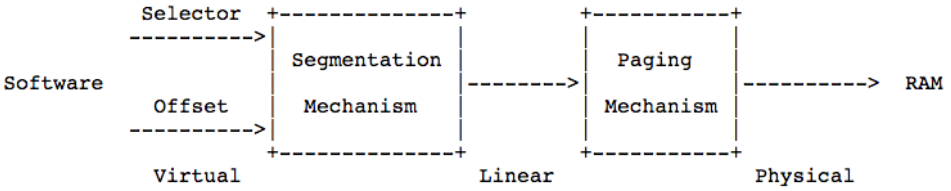
这里让我先看看Intel手册，可能是Intel手册的版本不一样，这里给出的

章节号是不对的。看的主要内容是虚拟内存和保护相关知识，内容比较多，这里就不详细展开了，之后有涉及的再说。

4.1 Virtual, Linear, and Physical Addresses

这里有三个概念：虚拟地址，线性地址，物理地址。

指导里面有张图，非常不错：



在程序运行中使用的都是虚拟地址，虚拟地址给出的形式是“段选择子+ 偏移”。段选择子放置在CS寄存器中，而我在C语言程序中所使用的地址(比如 一个指针的值)就是这个段内偏移。

线性地址（Linear Address）是逻辑地址到物理地址变换之间的中间层。在分段部件中逻辑地址是段中的偏移地址，然后加上基地址就是线性地址，就是相当于虚拟地址进行了一次运算得到的结果。在JOS中，代码段和数据段都只有一个，基址也都是0，所以虚拟地址和线性地 址总是一样的，段式地址转换并不明显。

物理地址，如果采用分页机制的话， 还需要从线性地址进行页式地址转换，就可以得到物理地址，也就是信息在内存中的真实地址了。

回想上一个lab， 我建立了一个页表，所以我能够map 4 MB的内容，使得链接地址在0xf0100000，而物理地址在0x00100000。现在我要map整个256MB的内存。

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 memory are mapped and with what permissions.

我之前用GDB都是只能看到虚拟地址，现在我需要用qemu来观察物理地址。本来以为我的电脑有问题。不支持ctrl a c 这种指令，后来查了官方文档，发现只有在qemu-nox的时候才能使用，这样就没有问题了。

这个Exercise 是让我用这个两种方法观察同一个数，从而增进对虚拟地址和物理地址的理解。

我的观察流程是这样的，（还是记录一下，毕竟qemu 的观察是第一次使用呢）

这是用qemu查看的物理内存的值：

```
K> (qemu) info mem
0000000000000000-0000000000400000 0000000000400000 -r-
00000000f0000000-00000000f0400000 0000000000400000 -rw
(qemu) xp 0x00000321
0000000000000321: 0x53f000ff
```

这是gdb 查看的虚拟内存的值：

```
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b mem
memcmp       memfind      memlayout.h  memset
memcpy       mem_init     memmove
(gdb) b mem_init
Breakpoint 1 at 0xf0100f85: file kern/pmap.c, line 128.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf0100f85 <mem_init>:       push   %ebp

Breakpoint 1, mem_init () at kern/pmap.c:128
128     {
(gdb) x/x 0xf0000321
0xf0000321:     0x53f000ff
```

我们发现他们是一样的。

指导里面接着讲了，一旦我们进入了保护模式，（首先是boot/boot.S）,这就不能直接用一个线性或者物理地址。所有的地址指针全部都是虚拟地址，并且需要由MMU翻译，这意味着所有的C 指针都是虚拟地址。

这里 the type uintptr_t represents opaque virtual addresses, and physaddr_t represents physical addresses，但是这里都是32位的整数，并不能用来解指针。

Summary

C type	Address type
`T*`	Virtual
`uintptr_t`	Virtual
`physaddr_t`	Physical

另外就是casting 的注意点，我们可以cast uintptr_t到指针类型，并且正确的解指针，但是不能够把physaddr_t变成指针，因为硬件会把这个当成虚拟地址来看待。

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;
```

解答：

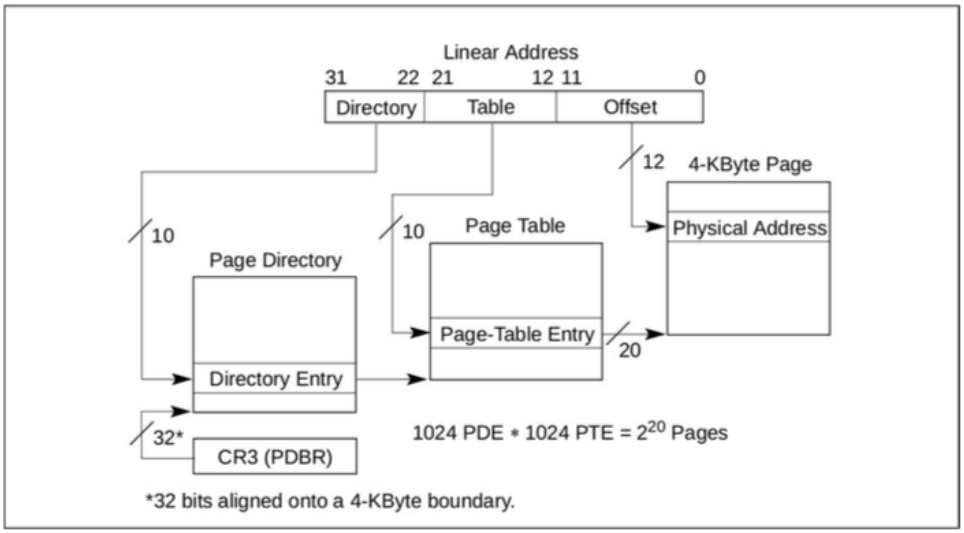
x的值是变量value的地址，之前已经说过C语言中的地址都是虚拟地址，所以这里x的类型当然应该是uintptr_t。

4.2 Reference counting

以后一个页面可能会同时 map 到许多不同的虚拟地址。一个页面如果没有被引用，则说明它变成了一个空闲页面，就可以回收了。之前我们也说过在PageInfo结构中有一个pp ref域来记录一个页面的被引用数。这个数还有其他用处，可以用来计数，记录我们应该保存到页目录的指针数。

4.3 Page Table Management

ICS的存储管理不太够用，补充一下在intel手册中看到的：



这是一个二级页式存储管理，其中第一级为页目录，第二级为页表，它们都存放在内存中。页目录只有一个，其中共有1024个页目录项，每个4字节，存放描述一个页表的信息。所以页目录大小为4KB，正好为一页大

小;页表 有1024个。每个页表中也有1024个页表项，每个也是4字节，存放描述一个页 的信息。所以，每个页表的大小也是4KB，为一页的大小。这样一来，一共可 以管理1024 1024 = 220个页，由于每个页大小为4KB，所以可以管理的内存 为220 4096B = 232B = 4GB。页目录项、页表项的4字节中的信息很丰富，包 括相关地址、访问权限等等，更多格式细节都可以在Intel手册的有关章节中查 到。 页表的基址存放在CR3寄存器中。对于一个32位线性地址，其高10位[31:22]代 表的是页目录项索引，用它可以找到页目录项，然后找到页表。线性地址的中 间10位[21:12]代表的是页表项索引，用它可以在页表中找到页表项，从而找到 页面。线性地址的最低12位[11:0]表示该地址在页内的偏移，通过它就可以在页 面中最终找到物理地址了。

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.

4.3.1 pgdir_walk()

这个函数会给一个页目录地址和线性地址，要求返回一个指向该页的页表项的指针。

```
1  // Given 'pgdir', a pointer to a page directory, pgdir_walk
       returns
2  // a pointer to the page table entry (PTE) for linear address '
       va'.
3  // This requires walking the two-level page table structure.
4  //
5  // The relevant page table page might not exist yet.
6  // If this is true, and create == false, then pgdir_walk
       returns NULL.
7  // Otherwise, pgdir_walk allocates a new page table page with
       page_alloc.
8  //    - If the allocation fails, pgdir_walk returns NULL.
9  //    - Otherwise, the new page's reference count is
       incremented,
10 //  the page is cleared,
11 //  and pgdir_walk returns a pointer into the new page table
       page.
12 //
```

```
13 // Hint 1: you can turn a PageInfo * into the physical address
       of the
14 // page it refers to with page2pa() from kern/pmap.h.
15 //
16 // Hint 2: the x86 MMU checks permission bits in both the page
       directory
17 // and the page table, so it's safe to leave permissions in the
        page
18 // directory more permissive than strictly necessary.
19 //
20 // Hint 3: look at inc/mmu.h for useful macros that mainipulate
        page
21 // table and page directory entries.
22 //
23 pte_t *
24 pgdir_walk(pde_t *pgdir, const void *va, int create)
25 {
26     // Fill this function in
27       if (!(pgdir[PDX(va)] & PTE_P)) {
28   if (!create) return NULL;
29   struct PageInfo *page = page_alloc(ALLOC_ZERO);
30   if (!page) return NULL;
31   page->pp_ref = 1;
32   pgdir[PDX(va)] = page2pa(page) | PTE_P | PTE_U | PTE_W;
33     }
34     return KADDR(PTE_ADDR(pgdir[PDX(va)])+ PTX(va) * sizeof(
          pte_t*)) ;
35 }
```

解释一下这里用到的宏和相关的变量：

- PDX()、PTX()、PGOFF()三个宏，可以直接从一个线性地址中取出页目 录项索引、页表项索引、页内偏移。
- PTE *，是页目录项、页表项中的权限位，有很多个，不一一列举了。
- page2pa(page) 用于将一个PageInfo转换成一个物理地址
- PTE_P — PTE_U — PTE_W权限，这个是按照注释要求，尽量给页目录更多的权限，因为最终的权限会经过两次检查。
- KADDR()是将一个物理地址返回成一个相应的虚拟地址。因为C语言中的指针都是虚拟地址。

这样代码就不难理解了。

4.3.2 boot_map_region()

这个函数是把[va, va+size) of virtual address space 映射到 physical [pa, pa+size)上。只用于初始化建立UTOP之上的映射，不改变映射域的pp_ref值。

按照他的提示用pgdir_walk()来写，然后把权限加上，就完成了呢。

```
1 //
2 // Map [va, va+size) of virtual address space to physical [pa,
      pa+size)
3 // in the page table rooted at pgdir.  Size is a multiple of
      PGSIZE.
4 // Use permission bits perm|PTE_P for the entries.
5 //
6 // This function is only intended to set up the ``static''
      mappings
7 // above UTOP. As such, it should *not* change the pp_ref field
       on the
8 // mapped pages.
9 //
10 // Hint: the TA solution uses pgdir_walk
11     static void
12 boot_map_region(pde_t *pgdir, uintptr_t va, size_t size,
      physaddr_t pa, int perm)
13 {
14     // Fill this function in
15     int i;
16     for (i = 0; i < size; i += PGSIZE) {
17   pte_t *pte = pgdir_walk(pgdir, (void*)(va + i), 1);
18   if (pte) *pte = (pa + i) | perm | PTE_P;
19     }
20 }
```

4.3.3 page_lookup()

用于查找虚拟地址va所在物理页的PageInfo，如果va没有映射的物 理地址则返回NULL。 另外，还有一个PTE_STORE需要被设置一下， 有了前面的基础，这里就很好写了。

```
1 //
2 // Return the page mapped at virtual address 'va'.
3 // If pte_store is not zero, then we store in it the address
4 // of the pte for this page.  This is used by page_remove and
5 // can be used to verify page permissions for syscall arguments
      ,
6 // but should not be used by most callers.
7 //
8 // Return NULL if there is no page mapped at va.
9 //
10 // Hint: the TA solution uses pgdir_walk and pa2page.
11 //
12 struct PageInfo *
13 page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
14 {
15   // Fill this function in
16       pte_t *pte = pgdir_walk(pgdir, va, 0);
17     if (pte_store != NULL) *pte_store = pte;
18       if (pte == NULL || !(*pte & PTE_P)) return NULL;
```

```
19        return pa2page(PTE_ADDR(*pte));
20
21 }
```

4.3.4 page_remove()

page_remove()解除虚拟地址va到现有页的对应。要求就是：

ref数要减1，如果到达零了，就要释放物理页。注释中提示我们要调用tlb_invalidate()清空TLB，并且利用page_decref()来进少当前页面的引用数(如果减少到0则会回收页面)。

实现也不难：

```
1 //
2 // Unmaps the physical page at virtual address 'va'.
3 // If there is no physical page at that address, silently does
      nothing.
4 //
5 // Details:
6 //   - The ref count on the physical page should decrement.
7 //   - The physical page should be freed if the refcount
      reaches 0.
8 //   - The pg table entry corresponding to 'va' should be set
      to 0.
9 //     (if such a PTE exists)
10 //   - The TLB must be invalidated if you remove an entry from
11 //     the page table.
12 //
13 // Hint: The TA solution is implemented using page_lookup,
14 //  tlb_invalidate, and page_decref.
15 //
16     void
17 page_remove(pde_t *pgdir, void *va)
18 {
19     // Fill this function in
20     struct PageInfo *page = page_lookup(pgdir, va, 0);
21     pte_t *pte = pgdir_walk(pgdir, va, 0);
22     if (page != NULL) page_decref(page);
23     if (pte != NULL) {
24   *pte = 0;
25   tlb_invalidate(pgdir, va);
26     }
27 }
```

4.3.5 page_insert()

page_insert() 用于建立虚拟地址所在页与物理页的映射，注意如果之前虚拟地址 映射到了其他页，要先解除映射。这里还要在判断一下是否pp已经和va对应了，这里要特殊处理一下。

```
1 //
2 // Map the physical page 'pp' at virtual address 'va'.
3 // The permissions (the low 12 bits) of the page table entry
4 // should be set to 'perm|PTE_P'.
5 //
6 // Requirements
7 //   - If there is already a page mapped at 'va', it should be
      page_remove()d.
8 //   - If necessary, on demand, a page table should be
      allocated and inserted
9 //     into 'pgdir'.
10 //   - pp->pp_ref should be incremented if the insertion
      succeeds.
11 //   - The TLB must be invalidated if a page was formerly
      present at 'va'.
12 //
13 // Corner-case hint: Make sure to consider what happens when
      the same
14 // pp is re-inserted at the same virtual address in the same
      pgdir.
15 // However, try not to distinguish this case in your code, as
      this
16 // frequently leads to subtle bugs; there's an elegant way to
      handle
17 // everything in one code path.
18 //
19 // RETURNS:
20 //   0 on success
21 //   -E_NO_MEM, if page table couldn't be allocated
22 //
23 // Hint: The TA solution is implemented using pgdir_walk,
      page_remove,
24 // and page2pa.
25 //
26 int
27 page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int
      perm)
28 {
29   // Fill this function in
30   pte_t *pte = pgdir_walk(pgdir, va, 1);
31       if (pte == NULL) return -E_NO_MEM;
32       if (*pte & PTE_P) {
33     if (PTE_ADDR(*pte) == page2pa(pp)) {
34         pp->pp_ref--;
35         tlb_invalidate(pgdir, va);
36     }
37     else {
38         page_remove(pgdir, va);
39     }
40       }
41       *pte = page2pa(pp) | perm | PTE_P;
42       pp->pp_ref++;
43       return 0;
```

```
44 }
```

5 Part 3: Kernel Address Space

JOS把处理器的32位线性地址分成了两个部分，一个是较低部分，一个是较高的地址部分。其中，用户态只能使用和控制较低部分的地址，较高部分的地址交给核控制。这个分界线就是inc/memlayout.h里面的ULIM。

5.1 Permissions and Fault Isolation

用户环境没有对ULIM以上的内存的权限，但是kernel可以在这部分读和写。

[UTOP,ULIM)这部分kernel和user有相同的权限，只读不写。这段是来暴露某些核的只读数据结构给用户。

UTOP之下是给用户使用的，用户可以给内存分配权限。终于解释了我一直困惑到现在的问题，感动，这个指导。

5.2 Initializing the Kernel Address Space

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.

用必要回顾一下boot_map_region()的功能。

```
1 // Map [va, va+size) of virtual address space to physical [pa,
      pa+size)
2 // in the page table rooted at pgdir.
3 static void
4 boot_map_region(pde_t *pgdir, uintptr_t va, size_t size,
      physaddr_t pa, int perm)
```

所以第一个函数

```
1 ////////////////////////////////////////////////////////////
2   // Map 'pages' read-only by the user at linear address UPAGES
3   // Permissions:
4   //     - the new image at UPAGES -- kernel R, user R
5   //       (ie. perm = PTE_U | PTE_P)
6   //     - pages itself -- kernel RW, user NONE
7   // Your code goes here:
8 boot_map_region(kern_pgdir, UPAGES, ROUNDUP(sizeof(struct
      PageInfo)* npages, PGSIZE), PADDR(pages), PTE_U);
```

第2段代码

将内核栈的虚拟地址对应到bootstack的物理地址处。

```
1 //////////////////////////////////////////////////////
2   // Use the physical memory that 'bootstack' refers to as the
        kernel
3   // stack.  The kernel stack grows down from virtual address
        KSTACKTOP.
4   // We consider the entire range from [KSTACKTOP-PTSIZE,
        KSTACKTOP)
5   // to be the kernel stack, but break this into two pieces:
6   //     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by
        physical memory
7   //     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed
        ; so if
8   //       the kernel overflows its stack, it will fault rather
         than
9   //       overwrite memory.  Known as a "guard page".
10  //     Permissions: kernel RW, user NONE
11  // Your code goes here:
12   boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, ROUNDUP(
        KSTKSIZE, PGSIZE), PADDR(bootstack), PTE_W);
```

和上面要实现的内容非常相似，直接实现了就好了。

第3段代码

将0开始物理地址全部对应到KERNBASE开始的虚拟地址处。

```
1 //////////////////////////////////////////////////////
2   // Map all of physical memory at KERNBASE.
3   // Ie.  the VA range [KERNBASE, 2^32) should map to
4   //      the PA range [0, 2^32 - KERNBASE)
5   // We might not have 2^32 - KERNBASE bytes of physical memory
        , but
6   // we just set up the mapping anyway.
7   // Permissions: kernel RW, user NONE
8   // Your code goes here:
9       boot_map_region(kern_pgdir, (uintptr_t)KERNBASE , -
            KERNBASE, (physaddr_t)0, PTE_W);
```

代码部分就算完成了。还有几个问题回答一下：

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	?	Page table for top 4MB of phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
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?

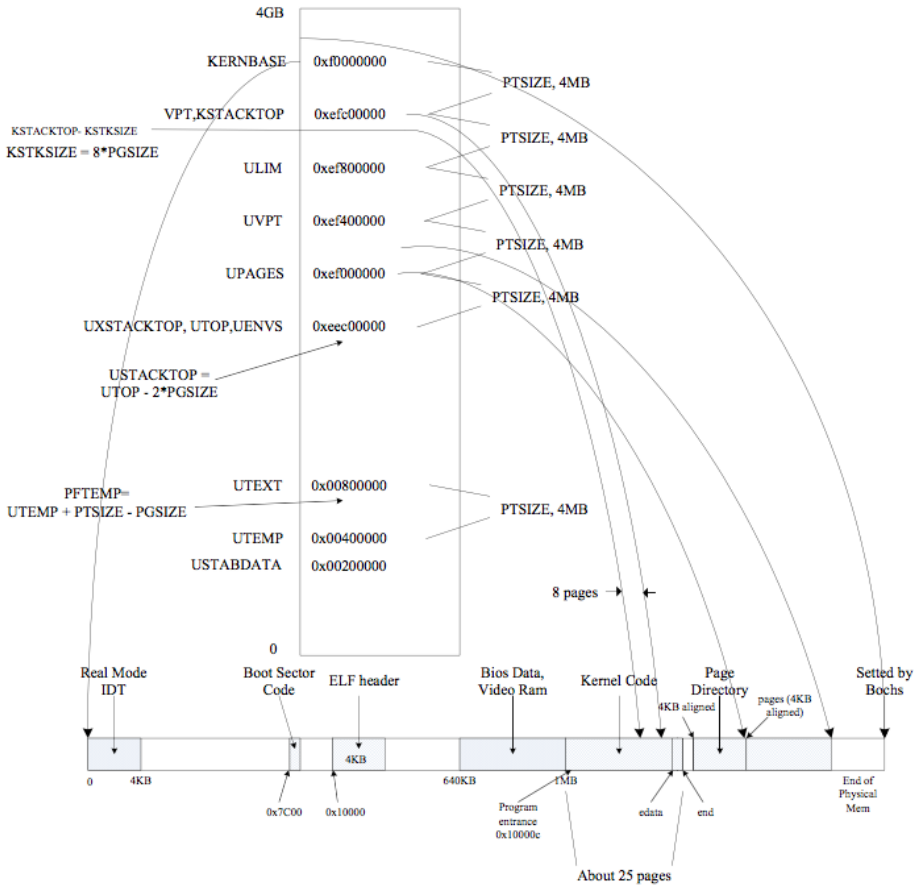
- 这道题的答案就是Exercise 5的练习。再看一下exercise 5的工作。

  [UPAGES, sizeof(PAGES)] $->$ [pages, sizeof(PAGES)] 这里 PAGES 代表页面管理结构所占用的空间;

  [KSTACKTOP – KSTKSIZE, 8] $->$ [bootstack, 8] 其中 bootstack 为内核编译时预先留下的 8 个页面(用做内核堆栈);

  [KERNBASE, pages in the memory] $->$ [0, pages in the memory] 这个地址映射范围比较广，含盖了所有物理内存。

  看一下这个图非常好的说明这个问题：



所以这个表就能够很容易去填了。以上我们做了三个映射，这三个映射的虚拟地址，我们是可以查找出来的：

UPAGES：0xef000000 KSTACKTOP： 0xf0000000 KERNBASE ：0xf0000000

KSTACKTOP – KSTKSIZE - KSTACKTOP = [0xf0000000 - 8page-size, 0xf0000000] KSTKSIZE = 8*PGSIZE

#define PGSIZE 4096 bytes mapped by a page

所以：

Base Virtual Address − > Points to (logically)

0xf0000000 − > 物理地址 0

0xefff8000 − > 内核栈

0xef000000 − > 页表信息pages

- 页表项中的权限位实现的控制

- JOS将所有物理内存(从0开始)对应到了虚拟地址KERNBASE以上的部分，而KERNBASE以上到4GB的封顶只有共0x10000000字节，256MB空间，所以JOS最多只能管理256MB空间。

- 256MB共256*1024KB/4KB=65536个页，需要64个二级页表，加上一个一级页表，共260KB。再加上pages数组里面pageinfo结构体的开销，65536*8B=512KB，共772KB

```
1 # Now paging is enabled, but we're still running at a low
    EIP
2   # (why is this okay?).  Jump up above KERNBASE before
      entering
3   # C code.
4   mov $relocated, %eax
5   jmp *%eax
```

就在kern/entry.S 这个位置，我们跳转到了高地址。

```
1 __attribute__((__aligned__(PGSIZE)))
2 pde_t entry_pgdir[NPDENTRIES] = {
3   // Map VA's [0, 4MB) to PA's [0, 4MB)
4   [0]
5     = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P,
6   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
7   [KERNBASE>>PDXSHIFT]
8     = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P +
        PTE_W
9 };
```

就在kern/entrypgdir.c这里，将虚拟地址[0, 4M)也映射到了物理地址[0, 4M)，这使得JOS在低地址也可以继续运行。

6 Challenge!

先做一个想做的challenge，其他的看时间吧，这个challenge能帮我调试，而且对前面一个question解答也能起到帮助。

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 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.

这个challeng 有三个任务：

1．实现了 pgmap 指令，用于查看虚拟地址到物理地址的映射

2．实现了 pgperm 指令，用于修改页表的权限，可添加、删除、修改

3. 实现了 memdump 指令，用于观察内存中的内容，可接受虚拟地址或物理地址

不管其他，先给monitor.c 加个头文件 #include ¡kern/pmap.h¿

```
1   int
2 mon_pgmap(int argc, char **argv, struct Trapframe *tf)
3 {
4     uintptr_t va1, va2, va;
5     struct PageInfo *pg;
6   pte_t *pte;
7     if (argc != 3) {
8   cprintf("Usage: pgmap va1 va2\n Display physical memory
        mapping from virtual memory va1 to va2\nva1 and va2 are
        hex\n");
9   return 0;
10    }
11    else {
12  for (va1 = strtol(argv[1], 0, 16), va2 = strtol(argv[2], 0,
        16); va1 < va2; va1 += PGSIZE) {
13      va = va1 & ~0xfff;
14      pg = page_lookup(kern_pgdir, (void*)va, 0);
15    pte = pgdir_walk(kern_pgdir, (void* )va,0);
16      if (pg){
17    cprintf("[%x, %x) ---> [%x, %x)    ", va, va + PGSIZE,
          page2pa(pg), page2pa(pg) + PGSIZE);
18       if(*pte & PTE_U)
19      cprintf("user: ");
20       else
21      cprintf("kernel: ");
22
23       if(*pte &PTE_W)
24      cprintf("read/write ");
25       else
26      cprintf("read only ");
27    }else
28    cprintf("[%x, %x) ---> NULL    ", va, va + PGSIZE);
29
30    cprintf("\n");
31  }
32    }
33    return 0;
34 }
```

```
K> pgmap 0xf0000000 0xf0004000
[f0000000, f0001000) ---> [0, 1000)    kernel: read/write
[f0001000, f0002000) ---> [1000, 2000)    kernel: read/write
[f0002000, f0003000) ---> [2000, 3000)    kernel: read/write
[f0003000, f0004000) ---> [3000, 4000)    kernel: read/write
K> pgmap 0xefff8000 0xf0000000
[efff8000, efff9000) ---> [10e000, 10f000)    kernel: read/write
[efff9000, efffa000) ---> [10f000, 110000)    kernel: read/write
[efffa000, efffb000) ---> [110000, 111000)    kernel: read/write
[efffb000, efffc000) ---> [111000, 112000)    kernel: read/write
[efffc000, efffd000) ---> [112000, 113000)    kernel: read/write
[efffd000, efffe000) ---> [113000, 114000)    kernel: read/write
[efffe000, effff000) ---> [114000, 115000)    kernel: read/write
[effff000, f0000000) ---> [115000, 116000)    kernel: read/write
K> pgmap 0xef000000 0xef008000
[ef000000, ef001000) ---> [11a000, 11b000)    user: read only
[ef001000, ef002000) ---> [11b000, 11c000)    user: read only
[ef002000, ef003000) ---> [11c000, 11d000)    user: read only
[ef003000, ef004000) ---> [11d000, 11e000)    user: read only
[ef004000, ef005000) ---> [11e000, 11f000)    user: read only
[ef005000, ef006000) ---> [11f000, 120000)    user: read only
[ef006000, ef007000) ---> [120000, 121000)    user: read only
[ef007000, ef008000) ---> [121000, 122000)    user: read only
```

实现出来的效果是这样的，这几个地址是前面问题中的，先打印出来，验证一下前面的答案。

```
1    int
2 mon_pgperm(int argc, char **argv, struct Trapframe *tf)
3 {
4     uintptr_t va, perm;
5     if (argc != 4) {
6     cprintf("Usage: pgperm +/-/= perm va\nset perm of page
          which contains va, va is hex\n");
7     return 0;
8     }
9     else {
10    va = strtol(argv[3], 0, 16);
11    perm = strtol(argv[2], 0, 16);
12    pte_t *pte = pgdir_walk(kern_pgdir, (void*)va, 0);
13    if (!pte) {
14      cprintf("0x%x is not mapped\n", va);
15    }
16    else {
17      if (argv[1][0] == '+') *pte |= perm;
18      if (argv[1][0] == '0') *pte &= ~perm;
19      if (argv[1][0] == '=') *pte = PTE_ADDR(*pte) | perm;
20    }
21    }
22    return 0;
23 }
```

实现的过程很简单，下面展示一下实现过后的效果图。

```
K> pgmap 0xf0000000 0xf0004000
[f0000000, f0001000) ---> [0, 1000)    kernel: read/write
[f0001000, f0002000) ---> [1000, 2000)    kernel: read/write
[f0002000, f0003000) ---> [2000, 3000)    kernel: read/write
[f0003000, f0004000) ---> [3000, 4000)    kernel: read/write
K> pgperm + 0x004 0xf0002222
K> pgmap 0xf0000000 0xf0004000
[f0000000, f0001000) ---> [0, 1000)    kernel: read/write
[f0001000, f0002000) ---> [1000, 2000)    kernel: read/write
[f0002000, f0003000) ---> [2000, 3000)    user: read/write
[f0003000, f0004000) ---> [3000, 4000)    kernel: read/write
```

用于观察内存中的内容，可接受虚拟地址区间或物理地址区间

```
1
2     int
3 mon_memdump(int argc, char **argv, struct Trapframe *tf)
4 {
5     uintptr_t a1, a2, a;
6     struct PageInfo *pg;
7     if (argc != 4) {
8   cprintf("Usage: memdump p/v a1 a2\n Dump memory content via
        virtual or physical address\na1 and a2 are hex\n");
9   return 0;
10    }
11    else {
12  a1 = strtol(argv[2], 0, 16), a2 = strtol(argv[3], 0, 16);
13  if (argv[1][0] == 'p') a1 = (int)KADDR(a1), a2 = (int)KADDR(
        a2);
14  for (a = a1; a < a2 && a >= KERNBASE; a += 4) {
15      if (!((a - a1) & 0xf)) cprintf("\n%x:\t", a);
16      cprintf(" %x", *(int*)(a));
17  }
18  cprintf("\n");
19    }
20    return 0;
21 }
```

```
memdump - Dump the contents of a range of memory
K> memdump v 0xf0003000 0xf0003070

f0003000:             97979797 97979797 97979797 97979797
f0003010:             97979797 97979797 97979797 97979797
f0003020:             97979797 97979797 97979797 97979797
f0003030:             97979797 97979797 97979797 97979797
f0003040:             97979797 97979797 97979797 97979797
f0003050:             97979797 97979797 97979797 97979797
f0003060:             97979797 97979797 97979797 97979797
K> memdump p 0x00003000 0x00003070

f0003000:             97979797 97979797 97979797 97979797
f0003010:             97979797 97979797 97979797 97979797
f0003020:             97979797 97979797 97979797 97979797
f0003030:             97979797 97979797 97979797 97979797
f0003040:             97979797 97979797 97979797 97979797
f0003050:             97979797 97979797 97979797 97979797
f0003060:             97979797 97979797 97979797 97979797
```

这是效果图，非常的完美啊。

This completes the lab.

```
make[1]: Leaving directory `/home/parallels/6.
running JOS: (1.4s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
```