操作系统JOS实习第二次报告

张弛 00848231,
zhangchitc@gmail.com

March 26, 2011

Contents

1 Introduction

我在实验中主要参考了华中科技大学邵志远老师写的JOS实习指导，在邵老师的主页上http://grid.hust.edu.cn/zyshao/OSEngineering.htm 可以找到。但是这次实验的指导远远不如lab1的指导详尽，所以我这里需要补充的内容会很多。

2 Physical Page Management

> **Exercise 1.** In the file kern/pmap.c, you must implement code for the following functions.
>
> boot_alloc()
> page_init()
> page_alloc()
> page_free()
>
> You also need to add some code to i386_vm_init() in pmap.c, as indicated by comments there. For now, just add the code needed leading up to the call to check_page_alloc().
>
> You probably want to work on boot_alloc(), then i386_vm_init(), then page_init(), page_alloc(), and page_free().
>
> check_page_alloc() tests 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.

这部分实验的内容暂时和页面转换机制没有关系，我们需要重点关注的是物理页面的规划以及管理。主要需要关注JOS内核代码中的inc/queue.h文件以及kern/pmap.c文件。

2.1 Physical page and its data structure

请仔细阅读邵老师的讲义中4.3章第一节页面管理。其中重点需要掌握以下内容：

1. 物理页和Page数据结构的对应关系
2. 对Page*页面链表的宏操作，在inc/queue.h中

里面我唯一碰到的问题就是没看懂为什么JOS给出的链表模板要写成下面这样的形式：

inc/queue.h

```
165 /*
166  * Reset the list named "head" to the empty list.
```

```
167  */
168 #define LIST_INIT(head) do {                                    \
169         LIST_FIRST((head)) = NULL;                              \
170 } while (0)
```

这个宏的的目的是将链表初始化为空。但我奇怪的是为什么要写成一个只执行一次的while循环的形式，而不是直接大括号包住这一段语句就可以了？实验室的白光东师兄给出了一个详尽满意的答案。他给了我下面这样的程序：

test.c

```
1  #include <stdio.h>
2
3  #define MACRO() do { \
4    printf ("hello\n"); \
5  } while (0)
6
7  int main () {
8    int x;
9    scanf ("%d\n", &x);
10
11   if (x)
12     MACRO ();
13   else
14     printf ("this is else\n");
15   return 0;
16 }
```

然后使用gcc -E test.c编译这段程序，参数E的目的是为了让编译器仅仅进行预编译以后即停下来，并输出预编译后的程序结果，那么我们得到这样的输出

```
int main () {
  int x;
  scanf ("%d\n", &x);

  if (x)
    do { printf ("hello\n"); } while (0);
  else
    printf ("this is else\n");
  return 0;
}
zhangchi@zhangchi-desktop:/tmp/test$
```

很明显我们看到相关的`MACRO ();`调用变成了其相应的宏展开，请特别注意，调用的时候我们是以单个语句的形式调用`MACRO ()`的，那么展开以后的形式还是满足了单个语句(一个while循环)，并且，在其后面加上了分号。如果我们把MACRO改成：

test.c

```
1  #define MACRO() { \
2    printf ("hello\n"); \
3  }
```

那么再次展开的结果会变成：

```
int main () {
  int x;
```

```
  scanf ("%d\n", &x);

  if (x)
    { printf ("hello\n"); };
  else
    printf ("this is else\n");
  return 0;
}
zhangchi@zhangchi-desktop:/tmp/test$
```

明显可以看出这样的转换造成语法错误，用大括号包裹的代码块后不需要分号。这里出错的原因就在于我们调用MACRO ()之前是当成一个单个语句，调用展开后变成了一个代码块。那么相应的语法结构就出现了变化导致出错。

真是考虑得非常细致！感谢师兄！

2.2 Physical memory layout

请仔细阅读邵老师的讲义中的4.3章第一节页面管理中“页面管理链表在内存中的存储和放置”小节。重点理解

1. pages数组和Page*链表的对应关系
2. pages所在的空间是怎样分配的
3. 整个物理内存的布局

这里要说的是，在做完lab1以后，我们知道了在实模式下物理页面前640KB的一些分配情况，如BIOS的载入地址、boot loader载入地址、操作系统内核ELF文件头的临时存放空间等等，具体的布局应该如下图所示：

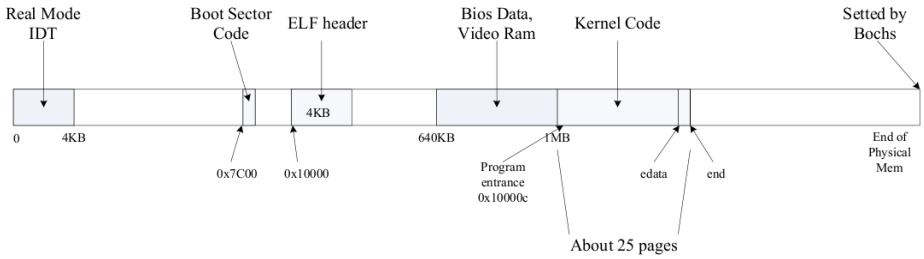

图 4-1. 调用 i386_init()函数以前内存的 layout

在lab1完成后，我们从0x000100000这个位置放入内核，直到end结束。end是链接器作链接时得到的内核结束地址。

那么在建立物理页面对应的Page* 链表时，我们需要为这个链表分配实际的物理内存空间，在二级页地址映射机制中，系统还需要一个页目录存下所有二级页表的地址，这个也是需要操作系统预先分配空间的，所以结合上图，我们第一步完成之后物理内存布局应该如下图所示：

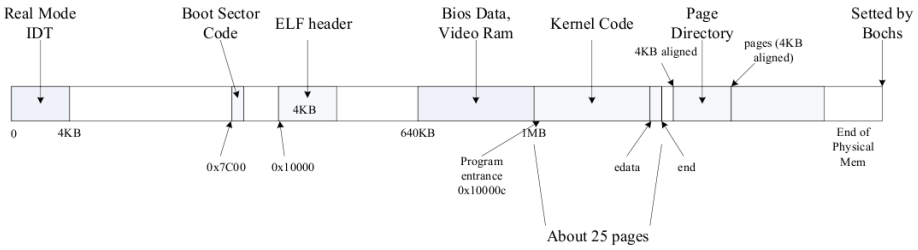

图 4-4. 页面管理空间的放置

我们第一步的目的就是为页目录和pages分配好空间，并建立起空闲页表page_free_list。

开始写代码的时候，首先需要弄清楚kern/pmap.c中的几个基本的变量：

kern/pmap.c

```
12 // These variables are set by i386_detect_memory()
13 static physaddr_t maxpa;        // Maximum physical address
14 size_t npage;                   // Amount of physical memory (in pages)
15 static size_t basemem;          // Amount of base memory (in bytes)
16 static size_t extmem;           // Amount of extended memory (in bytes)
17
18 // These variables are set in i386_vm_init()
19 pde_t* boot_pgdir;              // Virtual address of boot time page directory
20 physaddr_t boot_cr3;            // Physical address of boot time page directory
21 static char* boot_freemem;      // Pointer to next byte of free mem
22
23 struct Page* pages;             // Virtual address of physical page array
24 static struct Page_list page_free_list; // Free list of physical pages
```

这里只需要知道两个变量，boot_freemem和boot_pgdir，前者是当前可用内存的开始地址（也就是说，内核载入以后，系统管理所需要的内容就从end以后开始分配，即从一开始boot_freemem是等于end，这个在接下来的代码里就能看到）；boot_pgdir则是系统页目录所在空间的开始地址。**注意，两者地址都是虚拟地址！在这次lab中一定要搞清楚的一个细节就是虚拟地址，线性地址和物理地址的区别，以及我们使用的地址变量哪些是虚拟地址，哪些是物理地址。**

接下来我们可以看到`i386_vm_init ()`，从这里开始我们这次的lab。一开始就看到关于页目录的初始化代码：

kern/pmap.c: i386_vm_init ()

```
1         ////////////////////////////////////////////////////////////////////
2         // create initial page directory.
3         pgdir = boot_alloc(PGSIZE, PGSIZE);
4
5         memset(pgdir, 0, PGSIZE);
6         boot_pgdir = pgdir;
7         boot_cr3 = PADDR(pgdir);
```

其中boot_alloc ()为其分配内存空间地址，然后将分配的地址段清空。然后将其物理地址（PADDR）放入boot_cr3准备启动x86的页面地址转换机制。这里要注意几点：

- PGSIZE为一个物理页的大小4KB = 4096B，定义在inc/mmu.h中，其中还有我们后面要用的重要常量PTSIZE，为一个页表对应实际物理内存的大小，即1024 * 4KB = 4MB
- 从boot_alloc ()得到的页面是不会作相应的初始化工作的，所以如果对分配到的空间有要求清空，必须自己亲自动手
- memset接受的清空地址是pgdir即一个虚拟地址，这个在我们后面的工作中对实际分配到的**物理页面**进行初始化时提醒，清空时使用memset也一定要使用实际物理页面对应的**内核虚拟地址**
- boot_cr3得到的是一个**物理地址**，这个和我们前面强调的分清每个地址变量到底是虚拟地址还是物理地址有密切联系

接下来我们来看一下第一个需要实现的函数boot_alloc ()

kern/pmap.c: boot_alloc ()

```
1  static void*
2  boot_alloc(uint32_t n, uint32_t align)
3  {
4          extern char end[];
5          void *v;
6
7          // Initialize boot_freemem if this is the first time.
8          // 'end' is a magic symbol automatically generated by the linker,
9          // which points to the end of the kernel's bss segment -
10         // i.e., the first virtual address that the linker
11         // did _not_ assign to any kernel code or global variables.
12         if (boot_freemem == 0)
13                 boot_freemem = end;
14
15         // LAB 2: Your code here:
16         //      Step 1: round boot_freemem up to be aligned properly
17         //              (hint: look in types.h for some handy macros)
18         //      Step 2: save current value of boot_freemem as allocated chunk
19         //      Step 3: increase boot_freemem to record allocation
20         //      Step 4: return allocated chunk
21
22         v = ROUNDUP (boot_freemem, align);
23         boot_freemem = (char*) v + n;
24
25         return v;
26 }
```

这个函数的问题不大。

看接下来的代码：

kern/pmap.c: i386_vm_init ()

```
1          //////////////////////////////////////////////////////////////////////
2          // Recursively insert PD in itself as a page table, to form
3          // a virtual page table at virtual address VPT.
```

```
4         // (For now, you don't have understand the greater purpose of the
5         // following two lines.)
6
7         // Permissions: kernel RW, user NONE
8         pgdir[PDX(VPT)] = PADDR(pgdir)|PTE_W|PTE_P;
9
10        // same for UVPT
11        // Permissions: kernel R, user R
12        pgdir[PDX(UVPT)] = PADDR(pgdir)|PTE_U|PTE_P;
13
14        //////////////////////////////////////////////////////////////////////
15        // Allocate an array of npage 'struct Page's and store it in 'pages'.
16        // The kernel uses this array to keep track of physical pages: for
17        // each physical page, there is a corresponding struct Page in this
18        // array.  'npage' is the number of physical pages in memory.
19        // User-level programs will get read-only access to the array as well.
20        // Your code goes here:
21
22
23        pages = boot_alloc (npage * sizeof (struct Page), PGSIZE);
24
25        //////////////////////////////////////////////////////////////////////
26        // Now that we've allocated the initial kernel data structures, we set
27        // up the list of free physical pages. Once we've done so, all further
28        // memory management will go through the page_* functions. In
29        // particular, we can now map memory using boot_map_segment or page_insert
30        page_init();
31
32        check_page_alloc();
33
34        page_check();
```

前两句对pgdir的操作我们可以先不用管他，在后来设置页表的时候我们会回过头来看这两句话的含义。在23行里为pages分配空间以后，就进入倒`page_init ()`对链表进行初始化了。

在进行接下来的编码之前，我们先需要了解JOS对于地址编码的一些规定，在inc/mmu.h中，我们可以找到一组详尽的宏：

inc/mmu.h

```
16  // A linear address 'la' has a three-part structure as follows:
17  //
18  // +--------10------+-------10-------+----------12----------+
19  // | Page Directory |   Page Table    | Offset within Page  |
20  // |      Index     |      Index      |                     |
21  // +----------------+-----------------+---------------------+
22  //  \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
23  //  \---------- PPN(la) -----------/
24  //
25  // The PDX, PTX, PGOFF, and PPN macros decompose linear addresses as shown.
26  // To construct a linear address la from PDX(la), PTX(la), and PGOFF(la),
27  // use PGADDR(PDX(la), PTX(la), PGOFF(la)).
28
29  // page number field of address
30  #define PPN(la)         (((uintptr_t) (la)) >> PTXSHIFT)
31  #define VPN(la)         PPN(la)         // used to index into vpt[]
32
33  // page directory index
34  #define PDX(la)         ((((uintptr_t) (la)) >> PDXSHIFT) & 0x3FF)
35  #define VPD(la)         PDX(la)         // used to index into vpd[]
36
37  // page table index
38  #define PTX(la)         ((((uintptr_t) (la)) >> PTXSHIFT) & 0x3FF)
39
```

```
40 // offset in page
41 #define PGOFF(la)       (((uintptr_t) (la)) & 0xFFF)
42
43 // construct linear address from indexes and offset
44 #define PGADDR(d, t, o) ((void*) ((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
```

其中PDX，PTX和PGOFF都很好理解，需要注意的是PPN，一个**线性地址**的PPN其实没有什么意义，但是如果我们对一个**物理地址**取PPN的话，就可以利用这个PPN直接访问**这个物理地址在pages数组中的对应页**！这个宏所以非常的好用。

我们来看到前面提到对物理页面链表进行初始化的`page_init ()`过程：

kern/pmap.c: boot_init ()

```
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, basemem) is free.
16     //  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM).
17     //     Mark it as in use so that it can never be allocated.
18     //  4) Then extended memory [EXTPHYSMEM, ...).
19     //     Some of it is in use, some is free. Where is the kernel
20     //     in physical memory?  Which pages are already in use for
21     //     page tables and other data structures?
22     //
23     // Change the code to reflect this.
24     //
25
26     int i;
27     int lower_ppn = PPN (IOPHYSMEM);
28     int upper_ppn = PPN (ROUNDUP (boot_freemem, PGSIZE));
29
30     LIST_INIT(&page_free_list);
31     for (i = 0; i < npage; i++) {
32         pages[i].pp_ref = 0;
33
34         if (i == 0) continue;
35         if (lower_ppn <= i && i < upper_ppn) continue;
36
37         LIST_INSERT_HEAD(&page_free_list, &pages[i], pp_link);
38     }
39 }
```

这个函数的具体工作就是建立其每个物理页面对应的实际链表节点，然后把那些被操作系统占用或是系统预留空间从链表里去除掉。通过对照2.2中提到的物理内存的使用布局，可以总结出以下几个使用的物理地址区域：

$[0, \mathrm{PGSIZE})$ ：
存放中断向量表IDT以及BIOS的相关载入程序

[IOPHYSMEM, EXTPHYSMEM) ：
存放输入输出所需要的空间，比如VGA的一部分显存直接映射这个地址

[EXTPHYSMEM, end) ：
存放操作系统内核kernel

[PADDR(boot_pgdir), PADDR(boot_pgdir) + PGSIZE) ：
存放页目录

[PADDR(pages), boot_freemem) ：
存放pages数组

但是除了第一项之外，后面的4段区域实际上是一段连续内存[IOPHYSMEM, boot_freemem)，所以上面的代码在实现时，把这段区域对应的物理页下标算出来，那么如果是第一个物理页或者是上面区间内的物理页，就不加入空闲页链表里。

接下来我们还要完成两个的对物理页链表的操作：申请和释放，先来看申请的操作`page_alloc ()`

kern/pmap.c: page_alloc ()

```
int
page_alloc(struct Page **pp_store)
{
    // Fill this function in

    if (!LIST_EMPTY (&page_free_list)) {
        *pp_store = LIST_FIRST (&page_free_list);
        LIST_REMOVE (LIST_FIRST (&page_free_list), pp_link);
        return 0;
    }

    return -E_NO_MEM;
}
```

这个很简单，直接按照注释来做即可。再看释放页面的`page_free ()`

kern/pmap.c: page_free ()

```
void
page_free(struct Page *pp)
{
    // Fill this function in

    LIST_INSERT_HEAD (&page_free_list, pp, pp_link);
}
```

好像更简单了...好吧亚

这个时候我们重新编译内核后启动JOS，应该可以通过`check_page_alloc()`的所有测试了。

3 Virtual Memory

Exercise 2. Read chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. You can skip 6.3. Although JOS relies most heavily on page translation, you will also need a basic understanding of how segmentation works in protected mode to understand what's going on in JOS.

貌似我没怎么看...，这个部分的lab请仔细阅读绍老师课件里4.3中第二小节"页表管理"。

3.1 Virtual, Linear, and Physical Addresses

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, 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.

QEMU's info mem command may also prove useful in the lab. We've also added an info pg command to our patched version of QEMU that prints out the current page table.

这里提到的调试命令貌似都没用到过。不过这一章里提到用两种数据类型来区分虚拟和物理地址。这样在我们编写程序时通过函数内部的声明能够很清晰的看出对应操作的地址是哪一类，以便于我们理解代码。

这里提到的有关地址类型的问题

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;
```

因为在内核中操作数据都是以内核虚拟地址进行的，所以x的类型应该是uintptr_t

3.2 Reference counting

从这里就开始提到虚拟内存空间了。里面出现了UTOP这样的地址，那么我们来看看详细定义在inc/memlayout.h中的虚拟内存布局：

inc/memlayout.h

```
/*
 * Virtual memory map:                                Permissions
 *                                                    kernel/user
 *
 *    4 Gig --------->  +------------------------------+
 *                      |                              | RW/--
 *                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                      :              .               :
 *                      :              .               :
 *                      :              .               :
 *                      |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
 *                      |                              | RW/--
 *                      |   Remapped Physical Memory   | RW/--
 *                      |                              | RW/--
 *    KERNBASE ----->   +------------------------------+ 0xf0000000
 *                      |  Cur. Page Table (Kern. RW)  | RW/--  PTSIZE
 *    VPT,KSTACKTOP--> +------------------------------+ 0xefc00000      --+
 *                      |         Kernel Stack         | RW/--  KSTKSIZE   |
 *                      | - - - - - - - - - - - - - - -|                 PTSIZE
 *                      |      Invalid Memory (*)      | --/--             |
 *    ULIM     ------> +------------------------------+ 0xef800000      --+
 *                      |  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
 *                      +------------------------------+ 0xeebff000
 *                      |       Empty Memory (*)       | --/--  PGSIZE
 *    USTACKTOP  --->   +------------------------------+ 0xeebfe000
 *                      |      Normal User Stack       | RW/RW  PGSIZE
 *                      +------------------------------+ 0xeebfd000
 *                      |                              |
 *                      |                              |
 *                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                      .                              .
 *                      .                              .
 *                      .                              .
 *                      |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
 *                      |     Program Data & Heap      |
 *    UTEXT -------->   +------------------------------+ 0x00800000
 *    PFTEMP ------->   |       Empty Memory (*)       |        PTSIZE
 *                      |                              |
 *    UTEMP -s------->   +------------------------------+ 0x00400000      --+
 *                      |       Empty Memory (*)       |                   |
 *                      | - - - - - - - - - - - - - - -|                   |
 *                      |  User STAB Data (optional)   |                 PTSIZE
 *    USTABDATA ---->   +------------------------------+ 0x00200000        |
 *                      |       Empty Memory (*)       |                   |
 *    0 ------------>   +------------------------------+                 --+
 *
 * (*) Note: The kernel ensures that "Invalid Memory" (ULIM) is *never*
 *     mapped.  "Empty Memory" is normally unmapped, but user programs may
 *     map pages there if desired.  JOS user programs map pages temporarily
 *     at UTEMP.
 */
```

这个页面布局代表的是启用地址转换以后，无论是操作系统还是用户程序，看到的虚拟内存布局（这也就是说，**操作系统和用户程序使用的是同一套页目录和页表**，这个在绍老师的讲义里有提到），那么这个虚拟地址和我们在前面2.2中看到的实际物理页面布局之间有什么联系呢？我们先来看看这个页表里有哪些组成部分：

[USTABDATA, UTEXT) ：
这个部分暂时在这个lab里还没有涉及过，我们先可以不用管它有什么具体用途

[UTEXT, USTACKTOP) ：
用户程序的.text段以及用户堆栈共用区域，两个区域从两头向中间生长。

[USTACKTOP, UPAGES) ：
这个也用不到，不用关注，后面的lab会用到。

[UPAGES, UVPT) ：
在虚拟地址中这段内存就是对应的在**实际物理地址里pages数组**对应储存位置。（在后面的代码部分我们可以看到相应的操作语句）。可以看到这段地址在ULIM之下，也就是说操作系统开放pages数组便于让用户程序可以访问。为什么呢？比如说假如有的程序想要知道一个物理页面被引用了多少次，那么根据相应的pages[i].pp_ref就可以知道了。我们看到这个区域在布局中分配了PTSIZE的大小，那么这个大小够么？因为我们知道在物理页面中pages所占用的大小为npage $\times$ sizeof (struct Page) = 12B $\times$ npage。我们看看PTSIZE的空间可以装struct Page的个数为1024 $\times$ 4KB / 12B = 4MB / 12B = $\frac{1}{3}$M。一个Page结构对应一个实际大小为4KB的物理页面，所以这个PTSIZE大小的虚拟地址空间能够管理$\frac{1}{3}$M $\times$ 4KB = 1.3GB的物理内存，这个对于我们的QEMU模拟器来讲应该还是足够了。

[UVPT, ULIM)**和**[VPT, KERNBASE) ：
这两个地址映射的是同一个**系统页目录**，即原来在程序中看到的变量pgdir，开放给用户只读，可以使用户得到当前内存中**某个虚拟地址**对应的**物理页面地址**是多少。这段空间在虚拟地址上分配了PTSIZE，但是实际上只使用了物理页面上的PGSIZE个空间（可以回头去看看pgdir的空间分配参数是多少），所以这里很容易被误导和搞迷糊，请注意。还记得2.2 中提到的那两句没有解释的语句么，我们现在回过头去看看:

kern/pmap.c: i386_vm_init ()

```
1  //////////////////////////////////////////////////////////////////////
2  // Recursively insert PD in itself as a page table, to form
3  // a virtual page table at virtual address VPT.
4  // (For now, you don't have understand the greater purpose of the
5  // following two lines.)
6
7  // Permissions: kernel RW, user NONE
8  pgdir[PDX(VPT)] = PADDR(pgdir)|PTE_W|PTE_P;
9
10 // same for UVPT
11 // Permissions: kernel R, user R
12 pgdir[PDX(UVPT)] = PADDR(pgdir)|PTE_U|PTE_P;
```

我们只看UVPT好了。12行的语句字面来看是讲是将UVPT对应到的虚拟地址对应在系统页目录中的表项设置成它自己的物理地址。也就是说，如果一个用户程序想访问页目录的话，只要把对应的虚拟地址设置成UVPT即可。得到页目录有什么用？如果我们想查找一个**任意虚拟地址**所在的页面的**物理地址**和其对应的**二级页表物理地址**的话，只要有页目录地址就可以做到！具体来讲，如果要查询的虚拟地址是addr

= PDX|PTX|OFFSET的话，显然只有PDX|PTX决定了addr物理页面的地址。查找方法如下：

查找addr对应的物理页面地址 ：
构造虚拟地址vaddr = UVPT[31:22]|PDX|PTX|00在虚拟内存空间里查询。根据两级页表翻译机制：

1. 系统首先取出vaddr的前10位，即UVPT[31:22]，去页目录里查询，根据

```
pgdir[PDX(UVPT)] = PADDR(pgdir)|PTE_U|PTE_P;
```

注意UVPT[31:22]等价于PDX(UVPT)，所以得到的二级页表地址A0，还是页目录pgdir本身。

2. 再取出vaddr的中间10位，即PDX，去二级页表中A0（即页目录），查找到的是addr所在**二级页表的物理地址A1**，注意！！不是addr的物理页面地址，是二级页表的地址！！
3. 最后取出vaddr的最后12位，即PTX|00，去A1物理页中（即addr所在二级页表）查找，得到的就是addr最后所在页面的物理地址

查找addr对应的二级页表物理地址 ：
构造虚拟地址vaddr = UVPT[31:22]|UVPT[31:22]|PDX|00在虚拟内存空间里查询。注意这个地址等价于PDX(UVPT)|PDX(UVPT)|PDX|00。

1. 根据上面的分析，我们可以知道页式转换的前两步，地址转换系统都会跳回到页目录本身
2. 最后一步取出vaddr的最后12位，即PDX|00，去pgdir的物理页查询，查到的就是addr对应的二级页表物理地址

我自我感觉应该是说清楚了的...\(￣3￣)/

相应的[VPT, KERNBASE)就一样了，只不过它和[UVPT, ULIM)权限不同，前者开放给操作系统，可读可写，后者只允许用户读。

[KERNBASE, 4GB) ：
这个部分映射实际物理内存中从0开始的中断向量表IDT、BIOS程序、IO端口以及操作系统内核等。即内核使用的虚拟地址KERNBASE + x进入这段地址查找到实际上的物理地址就是它自身。所以这段空间是为操作系统准备的。这个在后面解释这段具体的代码和4.2中关于Exercise的提问时还会提到。

从这个布局结构我们就可以看出来，实际上虚拟内存中分配给用户的只有[0, ULIM)大概3.7GB而非4GB的空间（ULIM = 0xef800000），其余的空间需要分配给操作系统。

3.3 Page Table Management

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

        pgdir_walk()
        boot_map_segment()
        page_lookup()
        page_remove()
        page_insert()

page_check(), called from i386_vm_init(), tests your page table management
 routines. You should make sure it reports success before proceeding.
```

现在开始我们就要开始真正的地址转换部分的编程了。在这之前，我们先看看定义在kern/pmap.h中一组好用的宏：

kern/pmap.h

```
63 static inline ppn_t
64 page2ppn(struct Page *pp)
65 {
66         return pp - pages;
67 }
68
69 static inline physaddr_t
70 page2pa(struct Page *pp)
71 {
72         return page2ppn(pp) << PGSHIFT;
73 }
74
75 static inline struct Page*
76 pa2page(physaddr_t pa)
77 {
78         if (PPN(pa) >= npage)
79                 panic("pa2page called with invalid pa");
80         return &pages[PPN(pa)];
81 }
82
83 static inline void*
84 page2kva(struct Page *pp)
85 {
86         return KADDR(page2pa(pp));
87 }
```

我们知道每个物理页面对应一个Page的struct和一个物理页号PPN和唯一的物理首地址，这组宏就是作这几个量之间的对应关系的，其中：

- 一个Page对应的PPN就是page2ppn (struct Page)，而一个PPN对应的struct Page则是pages[PPN]
- 一个Page对应的物理地址是page2pa (struct Page)，而一个物理地址对应的struct Page则是pa2page (pa)

最后还提供了一个页面到内核虚拟地址的转换宏page2kva，这个也很好用。根据Exercise的提示，我们先来看看要填写的第一个函数`pgdir_walk()`

kern/pmap.c: pgdir_walk ()

```
1  pte_t *
2  pgdir_walk(pde_t *pgdir, const void *va, int create)
3  {
4      pde_t *pt = pgdir + PDX(va);
5      void *pt_kva;
6
7      if (*pt & PTE_P) {
8          pt_kva = (void*) KADDR (PTE_ADDR (*pt));
9          return (pte_t*) pt_kva + PTX (va);
10     }
11
12     struct Page *newpt;
13
14     if (create == 1 && page_alloc (&newpt) == 0) {
15
16         memset (page2kva (newpt), 0, PGSIZE);
17         newpt -> pp_ref = 1;
18
19         *pt = PADDR (page2kva (newpt))|PTE_U|PTE_W|PTE_P;
20         pt_kva = (void*) KADDR (PTE_ADDR (*pt));
21         return (pte_t*) pt_kva + PTX (va);
22     }
23
24     return NULL;
25 }
```

简单说一下这个函数的作用，其实这个函数就是实现虚拟地址到实际物理地址的翻译过程。根据给出的虚拟地址，返回其二级页表中对应的页表项。根据参数create的值，如果等于1，则如果虚拟地址还没有对应的物理页面，则分配一个给它。

代码中有几个地方需要注意：

1. 第7行代码：判断页目录中va对应二级页表是否存在应该看页目录的对应PDX(va)项最后一位状态位present是否为0，我一开始写成了if (*pt != 0)了，这样显然不对
2. 第16行代码：一定记得memset参数使用的是内核虚拟地址
3. 第19行代码：这里我们为新建的物理页设置页目录表项时，页目录后12位的权限位为什么是PTE_U|PTE_W|PTE_P呢?

第三条提出的问题在注释中其实已经提示了，但是我还是查询了Intel x86的手册：在Reference里的**IA-32 Intel Architecture Software Developer's Manuals**，其中的**Volume 3A: System Programming Guide, Part 1**。在里面的**COMBINING PAGE AND SEGMENT PROTECTION**一节里，提到了分段分页保护机制的实现。当一个程序试图访问一个虚拟地址的数据时，x86系统的保护机制运行为：

- 先检查段权限位DPL，这个是所访问数据段或者Call gate的权限级别，和当前权限级别CPL进行比较，如果不够则产生权限异常（具体机制请参考手册，在这个问题上我们不用管它），否则进入下一步
- 再检查页目录相应表项的访问权限，如果不够也产生异常

- 最后检查二级页表相应表项的访问权限，不够就产生异常

可以看到，在任意一次检查上违例了，那么访问失效。实际上我们知道页目录中一个表项代表的就是内存中的1024个物理页，这些页中很可能对访问的控制各不相同，比如有的可以给用户写权限，有的只能读，有的连读都不行。那么在无法提前知道这些需求的话，最明智的办法就是不在页目录这一环节限制太多，让最终的访问控制在二级页表这一环节上再去具体设置。

实际上Intel手册上给出了一个页目录加页表的访问控制的组合控制效果：

Table 4-3. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

NOTE:

* If CR0.WP = 1, access type is determined by the R/W flags of the page-directory and page-table entries. IF CR0.WP = 0, supervisor privilege permits read-write access.

这个图应该是很容易看明白的，只有一个不太清楚的地方，就是为什么当最终组合效果的权限要求是内核态时，读写权限全部是可读也可写的？

助教告诉我，因为在内核态下，内核想要写入任意地址都是可行的。即便内核在写入一个表项时发现该页表项权限只能允许它读的话，通过修改CR0.WP寄存器，内核也能最终写入这个地址，所以访问类型就显得没有意义了，感谢张顺廷师兄的解释！！

接下来看看要填的第二个函数`boot_map_segment()`:

kern/pmap.c: boot_map_segment ()

```
1 static void
2 boot_map_segment(pde_t *pgdir, uintptr_t la, size_t size, physaddr_t pa, int perm)
```

```
{
    int offset;
    pte_t *pte;

    for (offset = 0; offset < size; offset += PGSIZE) {
        pte = pgdir_walk (pgdir, (void*)la, 1);

        *pte = pa|perm|PTE_P;

        pa += PGSIZE;
        la += PGSIZE;
    }
}
```

这个函数的作用是将线性地址la开始的size大小的区域映射到物理页面pa开始的同样大小区域。代码很直接，没什么好说的。实现完这个函数后，我们可以回到`i386_vm_init ()`从上次停下的地方继续往下看：

kern/pmap.c: i386_vm_init ()

```
        //////////////////////////////////////////////////////////////////////
        // 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_segment (
            pgdir,
            UPAGES,
            ROUNDUP (npage * sizeof (struct Page), PGSIZE),
            PADDR ((uintptr_t) pages), // PADDR returns a (void*)
            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_segment (
            pgdir,
            KSTACKTOP - KSTKSIZE,
            KSTKSIZE, // 8 * PGSIZE
            PADDR ((uintptr_t) 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_segment (
```

```
46              pgdir,
47              KERNBASE,
48              ~KERNBASE + 1,
49              // 2^32 - KERNBASE = (2 ^ 32 - 1 - KERNBASE) + 1 = ~KERNBASE + 1
50              (physaddr_t) 0,
51              PTE_W);
```

其中最后一段就是映射的KERNBASE以上的虚拟空间到物理地址操作系统本身。这个是为了将来打开页式转换以后操作系统也能使用统一的虚拟地址来访问其自身的数据。

按照注释的指示设置好映射关系即可，唯一需要留意的地方就是传参数的时候记住将内核虚拟地址的变量都用uintptr_t转换一下类型。

接下来的几个函数都不是很难：`page_lookup ()` 查找一个虚拟地址对应的具体Page：

kern/pmap.c: page_lookup ()

```
1  struct Page *
2  page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
3  {
4      pte_t *pte = pgdir_walk (pgdir, va, 0);
5
6      if (pte_store != 0) {
7          *pte_store = pte;
8      }
9
10     if (pte != NULL && (*pte & PTE_P)) {
11         return pa2page (PTE_ADDR (*pte));
12     }
13
14         return NULL;
15 }
```

`page_lookup ()` 在页目录中删除一个虚拟地址对应的二级页表表项。

kern/pmap.c: page_remove ()

```
1  void
2  page_remove(pde_t *pgdir, void *va)
3  {
4      pte_t *pte;
5      struct Page *physpage = page_lookup (pgdir, va, &pte);
6
7      if (physpage != NULL) {
8          page_decref (physpage);
9          *pte = 0;
10         tlb_invalidate (pgdir, va);
11     }
12 }
```

还有`page_insert ()`：

kern/pmap.c: boot_alloc ()

```
1  int
2  page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
3  {
4      pte_t *pte = pgdir_walk (pgdir, va, 1);
```

```
5
6       if (pte == NULL) {
7           return -E_NO_MEM;
8       }
9
10      if (*pte & PTE_P) {
11          if (PTE_ADDR(*pte) == page2pa (pp)) {
12              tlb_invalidate (pgdir, va);
13              pp -> pp_ref --;
14          } else {
15              page_remove (pgdir, va);
16          }
17      }
18
19      *pte = page2pa (pp)|perm|PTE_P;
20      pp -> pp_ref ++;
21
22      return 0;
23  }
```

这个函数负责将一个虚拟地址映射到它实际对应的物理页面上去，这里有几种调用的情况：

1. 这个虚拟地址所在地址在二级页表上没有挂载页面，那么这时直接修改相应的二级页表表项即可
2. 如果已经挂载了页面，且页面和当前分配的物理页面不一样，那么就卸下原来的页面，再挂载新的页面。
3. 如果已经挂载了页面，而且已挂载页面和当前分配的物理页面是同样的，这是什么情况呢？这种情况非常普遍，就是当内核要修改一个页面的访问权限时，它会将同一个页面重新插入一次，传入不同的perm参数，即完成了权限修改。

所以注意第13行代码，修改权限并没有增加一个物理页面的引用数目，所以这里先减了1，方便后面加回来。第12行代码重置了TLB但第15行没有，因为`page_remove ()`函数中已经将移除的物理页自动重置TLB了，这里就没有必要重复清空了。

至此，我们已经把所有要求的函数填完了，重新编译运行JOS，连同page_check()和check_boot_pgdir() 应该都可以通过了。即我们提前把Part 3的内容也做完了。如命令行打印：

```
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
page_check() succeeded!
check_boot_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

4 Kernel Address Space

4.1 Permissions and Fault Isolation

关于[UTOP, ULIM)的空间我们早已讨论过。

4.2 Initializing the Kernel Address Space

Exercise 5. Fill in the missing code in i386_vm_init() after the call to page_check().

Your code should now pass the check_boot_pgdir() check.

已完成

Question

1. 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]

2. After check_boot_pgdir(), i386_vm_init() maps the first four MB of virtual address space to the first four MB of physical memory, then deletes this mapping at the end of the function. Why is this mapping necessary? What would happen if it were omitted? Does this actually limit our kernel to be 4MB? What must be true if our kernel were larger than 4MB?
3. (From Lecture 4) 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?

---

1. 具体的虚拟地址到物理地址的映射关系在3.3中已经结合JOS代码详细解释过了

2. 我们来看看那段代码具体是如何作的：

kern/pmap.c: i386_vm_init ()

```
1   //////////////////////////////////////////////////////////////////////
2   // On x86, segmentation maps a VA to a LA (linear addr) and
3   // paging maps the LA to a PA.  I.e. VA => LA => PA.  If paging is
4   // turned off the LA is used as the PA.  Note: there is no way to
5   // turn off segmentation.  The closest thing is to set the base
6   // address to 0, so the VA => LA mapping is the identity.
7
8   // Current mapping: VA KERNBASE+x => PA x.
9   //     (segmentation base=-KERNBASE and paging is off)
10
11  // From here on down we must maintain this VA KERNBASE + x => PA x
12  // mapping, even though we are turning on paging and reconfiguring
13  // segmentation.
14
15  // Map VA 0:4MB same as VA KERNBASE, i.e. to PA 0:4MB.
16  // (Limits our kernel to <4MB)
17  pgdir[0] = pgdir[PDX(KERNBASE)];
18
19  // Install page table.
20  lcr3(boot_cr3);
21
22  // Turn on paging.
23  cr0 = rcr0();
24  cr0 |= CR0_PE|CR0_PG|CR0_AM|CR0_WP|CR0_NE|CR0_TS|CR0_EM|CR0_MP;
25  cr0 &= ~(CR0_TS|CR0_EM);
26  lcr0(cr0);
27
28  // Current mapping: KERNBASE+x => x => x.
29  // (x < 4MB so uses paging pgdir[0])
30
31  // Reload all segment registers.
32  asm volatile("lgdt␣gdt_pd");
33  asm volatile("movw␣%%ax,%%gs" :: "a" (GD_UD|3));
34  asm volatile("movw␣%%ax,%%fs" :: "a" (GD_UD|3));
35  asm volatile("movw␣%%ax,%%es" :: "a" (GD_KD));
36  asm volatile("movw␣%%ax,%%ds" :: "a" (GD_KD));
37  asm volatile("movw␣%%ax,%%ss" :: "a" (GD_KD));
38  asm volatile("ljmp␣%0,$1f\n␣1:\n" :: "i" (GD_KT));  // reload cs
39  asm volatile("lldt␣%%ax" :: "a" (0));
40
41  // Final mapping: KERNBASE+x => KERNBASE+x => x.
42
43  // This mapping was only used after paging was turned on but
44  // before the segment registers were reloaded.
45  pgdir[0] = 0;
46
47  // Flush the TLB for good measure, to kill the pgdir[0] mapping.
48  lcr3(boot_cr3);
49  }
```

这段代码中顺序有这么几个主要的操作：

- 第17行到第26行：重置页目录的第一项，并打开页表转换机制，**在这个之前，没有页式地址转换**，所以所有的虚拟地址经过一次段地址转换就直接得到物理地址了，即paddr = laddr = vaddr − KERNBASE，laddr 到paddr 是等价的。在这个之后，线性地址laddr到物理地址paddr的页式转换打开。注意，这个时候，**会操作内存的程序只有内核**，而内核访问内存的地址都是内核虚拟地址（≥KERNBASE），且这个时候页表的第一项pgdir[0]被设置成

了pgdir[PDX[KERNBASE]]，所以任意一个内核访问vaddr=KERNBASE + x都先经过一次段式转换得到laddr = vaddr − KERNBASE = x，然后laddr经过页式转换实际上等价于访问原来虚拟地址KERNBASE上面的同样地址，所以还是访问到内核自己的东西。把pgdir[0]重设的目的就是让内核地址在打开页式转换以后仍然能找到其自己所在的位置，因为后面如第33行内联汇编代码需要使用到内核中定义的常量GD_UD等，内核在引用它们时都是使用的虚拟地址。

- 第32行到39行：关闭段式地址转换，即把原来是-KERNBASE的段基址赋值为零即可。这个时候一个内核虚拟地址KERNBASE+x转化为的线性地址laddr就是KERNBASE + x本身，那么这个时候进入页式转换，查找到的就是内核其自身的物理地址，这个前面已经解释过了。
- 第45到第48行：内核已经能通过关闭段式转换只依靠页式转换实现正常访问其自身了，那么这时虚拟地址的低4MB的设置已经没用，需要还原清空，然后重置TLB

3. 用户当然不能写内核地址的数据...，这个问题不予回答
4. 这个问题我们在前面已经分析过了，其中UPAGES存放的是系统pages的空间，系统分配的虚拟空间是PTSIZE，对应能管理的实际物理内存大概是1.3G左右
5. 管理物理内存实际的内存开销为1个页目录+1024个页表，即1025×4KB大概为4MB的样子。

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!)

4.3 Address Space Layout Alternatives

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.

Challenge! Extend the JOS kernel monitor with commands to allocate and free pages explicitly, and display whether or not any given page of physical memory is currently allocated. For example:

```
        K> alloc_page
                0x13000
        K> page_status 0x13000
                allocated
        K> free_page 0x13000
        K> page_status 0x13000
                free
```

Think of other commands or extensions to these commands that may be useful for debugging, and add them.