操作系统JOS实习第二次报告

张弛 00848231,
zhangchitc@gmail.com

March 24, 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.

这次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：

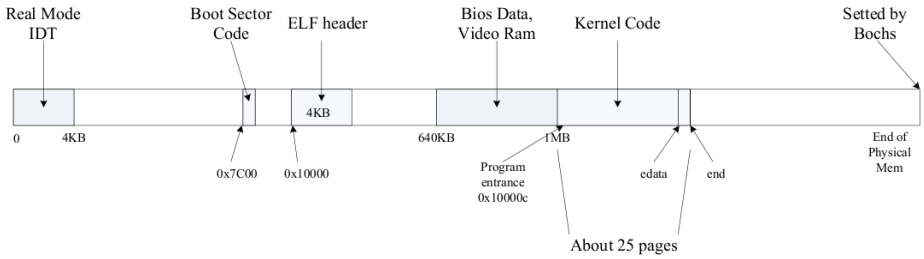


图 4-1. 调用 i386_init()函数以前内存的 layout

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：

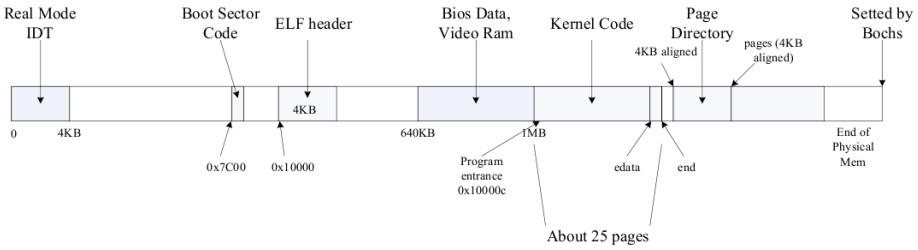


图 4-4. 页面管理空间的放置

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：

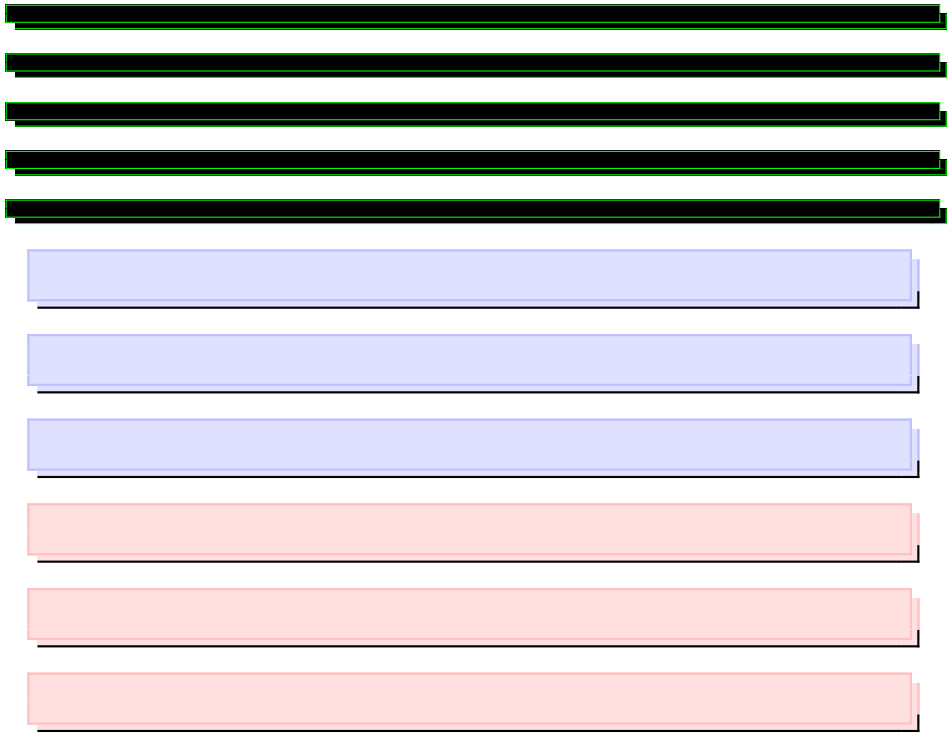
3.3 Page Table Management

kern/pmap.c: boot_alloc ()

kern/pmap.c: boot_alloc ()

kern/pmap.c: boot_alloc ()

kern/pmap.c: boot_alloc ()



boot/boot.S

boot/boot.S

boot/boot.S

boot/boot.S

boot/boot.S

boot/boot.S