JOS 实验二实验记录

作者：卓达城
指导老师：邵志远
单位：华中科技大学集群网络与服务计算实验室

第一步：理解下面这个函数

stab_binsearch(stabs,& region_left,& region_right, type, addr)函数

This function is a bi*ch ! ! !

背景：

JOS 的作者通过 kerne.ld 把调试信息和内核一起加载到内存中去。并提供了四个相应的宏以记录相关信息。

下图是 kernel.ld 中的代码片段：

```
  /* Include debugging information in kernel memory */
  .stab : {
          PROVIDE(__STAB_BEGIN__ = .);
          *(.stab);
          PROVIDE(__STAB_END__ = .);
          BYTE(0)          /* Force the linker to allocate space
                              for this section */
  }

  .stabstr : {
          PROVIDE(__STABSTR_BEGIN__ = .);
          *(.stabstr);
          PROVIDE(__STABSTR_END__ = .);
          BYTE(0)          /* Force the linker to allocate space
                              for this section */
  }
```

当我们运行完虚拟机之后，调试信息和内核一起加载到 0xf0000000（virtual address）中

从以下代码片段中，我们可以找到，调试信息是如何被赋值到 stab 的：

```
  if (addr >= ULIM) {
          stabs = __STAB_BEGIN__;
          stab_end = __STAB_END__;
          stabstr = __STABSTR_BEGIN__;
          stabstr_end = __STABSTR_END__;
```

调试信息的结构是怎么样的呢？

我们可以轻松地在代码中发现下面的结构：

```
struct Stab {
        uint32_t n_strx;        // index into string table of name
        uint8_t n_type;         // type of symbol
        uint8_t n_other;        // misc info (usually empty)
        uint16_t n_desc;        // description field
        uintptr_t n_value;      // value of symbol
};
```

但是它到底放的是什么，这令人满头雾水。

我们可以从 init.s 中找到我们想要的答案。init.s 通过命令：

```
gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c
```

生成。

```
.stabs  "test_backtrace:F(0,18)",36,0,0,test_backtrace
```

观察以上代码片段：

我们把它放入 stab 结构中，我们可以知道：

n_strx 是函数名，n_type 是 36 ，n_other 是 0，n_desc 也是 0，n_value 是函数的段偏移（eip）

```
.stabn  68,0,14,.LM1-.LFBB1
```

观察以上代码片段：

我们可以猜测：

n_str 为空， n_type 为 68， n_other 为 0, n_desc 为行号（对一下源代码就知道了），n_value 为函数内偏移量。（I just want to say f*ck！ Waste my time for 3 hours！！！）

```
#define N_FUN           0x24
```

这就是 type 的类型，0x24 是 36，跟上面的代码片段对应。

有了以上背景知识，我们开始研究 stab_binsearch 函数

先说说这个函数的返回值和参数：

返回值：

函数成功找到代码区域就返回 0 否则返回 -1

region_left 和 region_right:根据 type 返回代码区域的起始地址。如果是函数，返回函数的起始地址和终点地址，如果是代码，返回代码的起始地址的终点地址（翻译成汇编之后）。如果是文件或者其它类型，同上。

参数：

stabs 调试信息的起始地址

region_left：要搜索的区域的起始地址

region_right： 要搜索的区域的终点

type：要搜索的代码类型

addr ： eip

这个函数用了二分搜索法进行搜索，具体实现请看代码，比较难讲清楚，我看也看了很久才明白其中奥妙。但是如果了解了上面的背景再看，就会事半功倍。这里的所有的 stab 已经跟我们的代码运行顺序是一致的。

看完代码之后我们就开始实现我们的 Exercise 1，完成 debuginfo_eip 函数

首先：

在 debuginfo_eip 函数中，我看到这句代码

```
if (stabs[lfun].n_strx < stabstr_end - stabstr)
        info->eip_fn_name = stabstr + stabs[lfun].n_strx;
```

这句代码是判断函数名字有没有超界的。如果没有，就赋值。

```
info->eip_fn_addr = stabs[lfun].n_value;
addr -= info->eip_fn_addr;
```

这两句代码求出我们所要找的代码在具体函数中的偏移量。记着，我们前面说过，当 type 是代码的时候，n_value 表示的是偏移量。第一行代码中的 n_value 表示 addr 所在的函数的代码区域的首地址，addr - info->eip_fn_addr 就可以求出偏移量，下面就可以用 stab_binsearch 函数找到具体的行号。

以下是我们要添加的代码：

```
    // Hint:
    //      There's a particular stabs type used for line numbers.
    //      Look at the STABS documentation and <inc/stab.h> to find
    //      which one.
    // Your code here.

    stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
    if(lline <= rline){
            info->eip_line = stabs[lline].n_desc;
    } else {
            info->eip_line = 0;
            return -1;
    }
    // Search backwards from the line number for the relevant filenam
```

按照要求，我们还要找出函数的参数个数：

所再添加如下代码，我们已经知道了 stab 的结构方式（stab 的顺序和我们的代码顺序一样），所以以下代码及可以找出参数的个数。

```
    // Set eip_fn_narg to the number of arguments taken by the function,
    // or 0 if there was no containing function.
    // Your code here.

    for(;lfun<=rfun;lfun++){
            if(stabs[lfun].n_type == N_PSYM){
                    info->eip_fn_narg++;
            }
    }
```

然后，在 monitor.c 中加入如下代码就完成了第一个 Exercise

```
    char name[100];
    if( debuginfo_eip(read_eip(), &debug_info) >= 0)
    {
            cprintf("%s:",debug_info.eip_file);
            cprintf("%d:",debug_info.eip_line);
    //      memcpy(name,debug_info.eip_fn_name,debug_info.eip_fn_namelen);
    //      name[debug_info.eip_fn_namelen] = '\0';
    //      debug_info.eip_fn_name[debug_info.eip_fn_namelen] = '\0';
            for(i=0;i<debug_info.eip_fn_namelen ; i++){
                    name[i] =  debug_info.eip_fn_name[i];
            }
            name[i] = '\0';
            cprintf("%s:",name);
            cprintf("args_num: %d\n",debug_info.eip_fn_narg);
     }
     else
     {
            cprintf("debuginfo_eip fail\n");
     }
```

^^^^^^^^^^^^^^^^^^^^^^^^zhuodc@qq.com^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

写到这里，我确实不想再写了，但是作为一个软件工程的学生，我深深体会到文档的重要性，所以再写。

在开始做下面的实验之前，我们想要了解一下 JOS 的内存结构。
先从最基本的 boot.S 开始
在 boot.S 中，gdt 如下

NULL	0x00
0x00000000	0x08
0x00000000	0x10

然后系统跳转到 main.c,由于段选择符为 0x08，所以基地址为 0x0000000
JOS 把第一个扇区加载到 0x10000 处(elf),然后根据 elf 的参数把内核部分加载到 0x1000000 处,现在观察 kernel.ld，里面定义了开始地址为 0xF01000000，所以 entry.S 里面的代码的 eip 都是 0xF0100000++,然后 bootmain 的最后一句跳转到 0x10000c 处,开始执行 entry.S 的代码.
entry.S 中
一开始就 lgdt,把 gdt 变了,在 lgdt 的时候有一个宏 RELOC,这个宏的作用是把虚拟地址转换成实地址

  0xF0100000
- 0xF0000000(KERNBASE)
  0x00100000

所以,程序能够正确的加载到 mygdtdesc
加载后 gdt 如下:

NULL	0x00
-KERNBASE	0x08
-KERNBASE	0x10

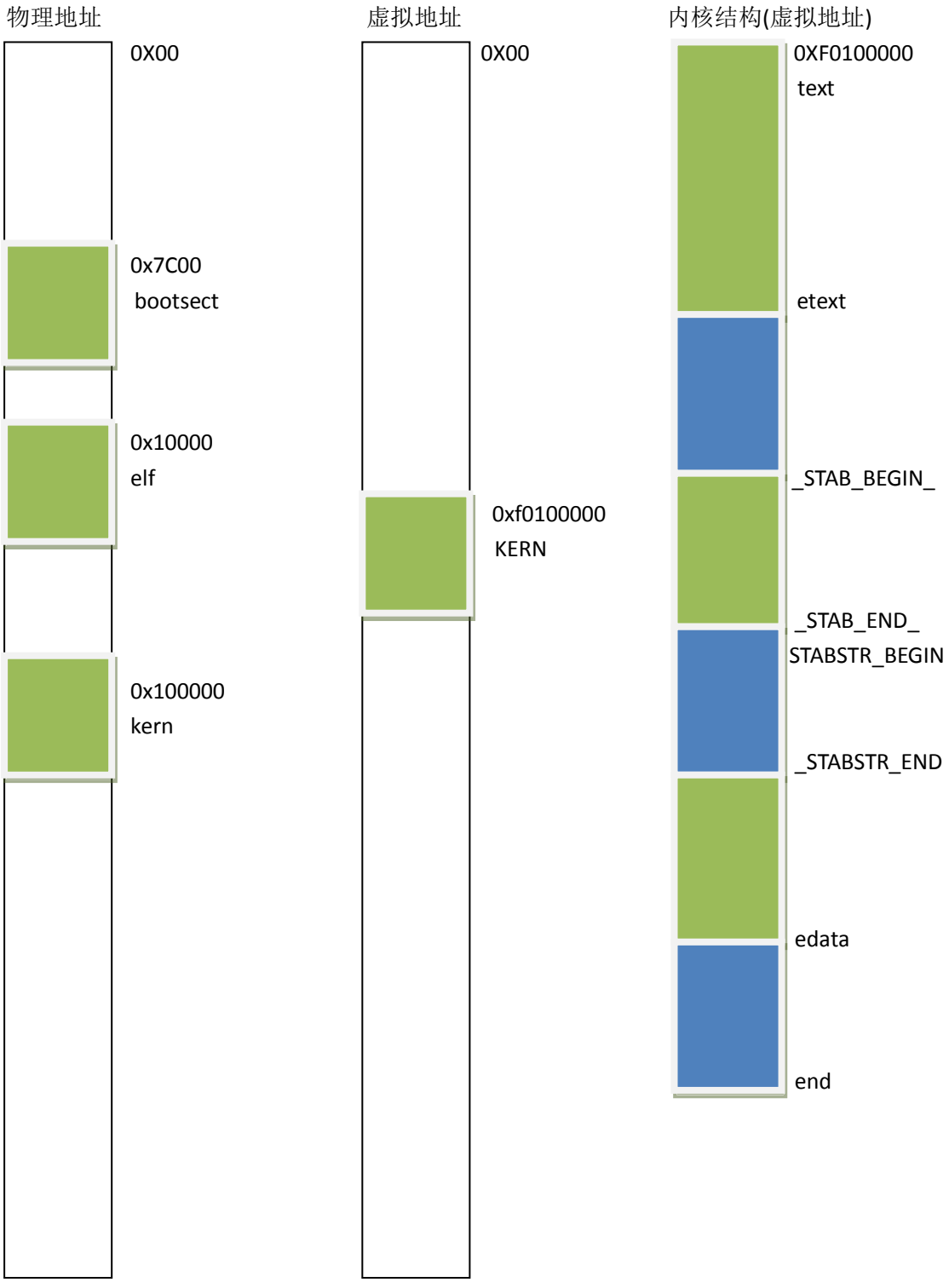
然后跳转到 CODE_SEL,从现在开始,段基地址变成 0X10000000,所以所有虚拟地址

(0xF0100000++)的地址,经过段转换之后都会变成 0x00100000++

```
 0x10000000
+0xF0100000 ++(EIP)
 0x00100000++
```

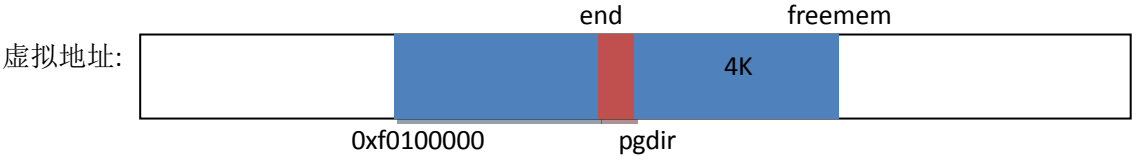
现在 EIP 是 0xf0100000 基地址是 0x10000000，程序可以正确运行.

分析到这里,我们可以画出比较完整的内存分布图:



以上数据可以从 kernel.ld 中得到。
下面进入 pmap.c
I386_detct_memory 中，可以得到一些全局变量
base = 640K
extmem = 31744K
maxpa = EXTPHYSMEM + extmem = 32M
npage = maxpa/1024

现在进入 vm_init
先去 boot_alloc 函数，这个函数的作用是在内核地址后分配一块 nbyte 的内存空间，返回内存空间的首地址，空间的首地址必须是 4k 的倍数。free_mem 指向内存空间末尾。
pages = boot_alloc(),先内存空间如下图所示:



boot_alloc 中应该添加的代码如下:

```
        boot_freemem = ROUNDUP(boot_freemem, align);
//      cprintf("0->\n");
        v = boot_freemem;
//      boot_freemem += n;
        boot_freemem += ROUNDUP(n,align);
//      cprintf("ff-->>\n");
```

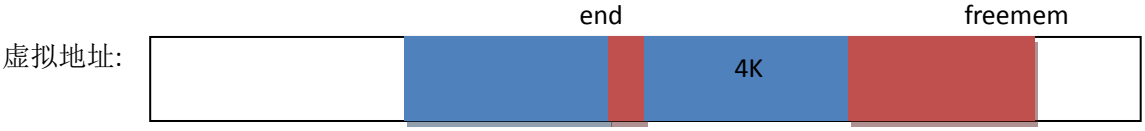
然后回到 vminit 中:

```
        // Permissions: kernel RW, user NONE
        pgdir[PDX(VPT)] = PADDR(pgdir)|PTE_W|PTE_P;

        // same for UVPT
        // Permissions: kernel R, user R
        pgdir[PDX(UVPT)] = PADDR(pgdir)|PTE_U|PTE_P;
```

这两句代码暂时不知道是做什么用的.
然后是

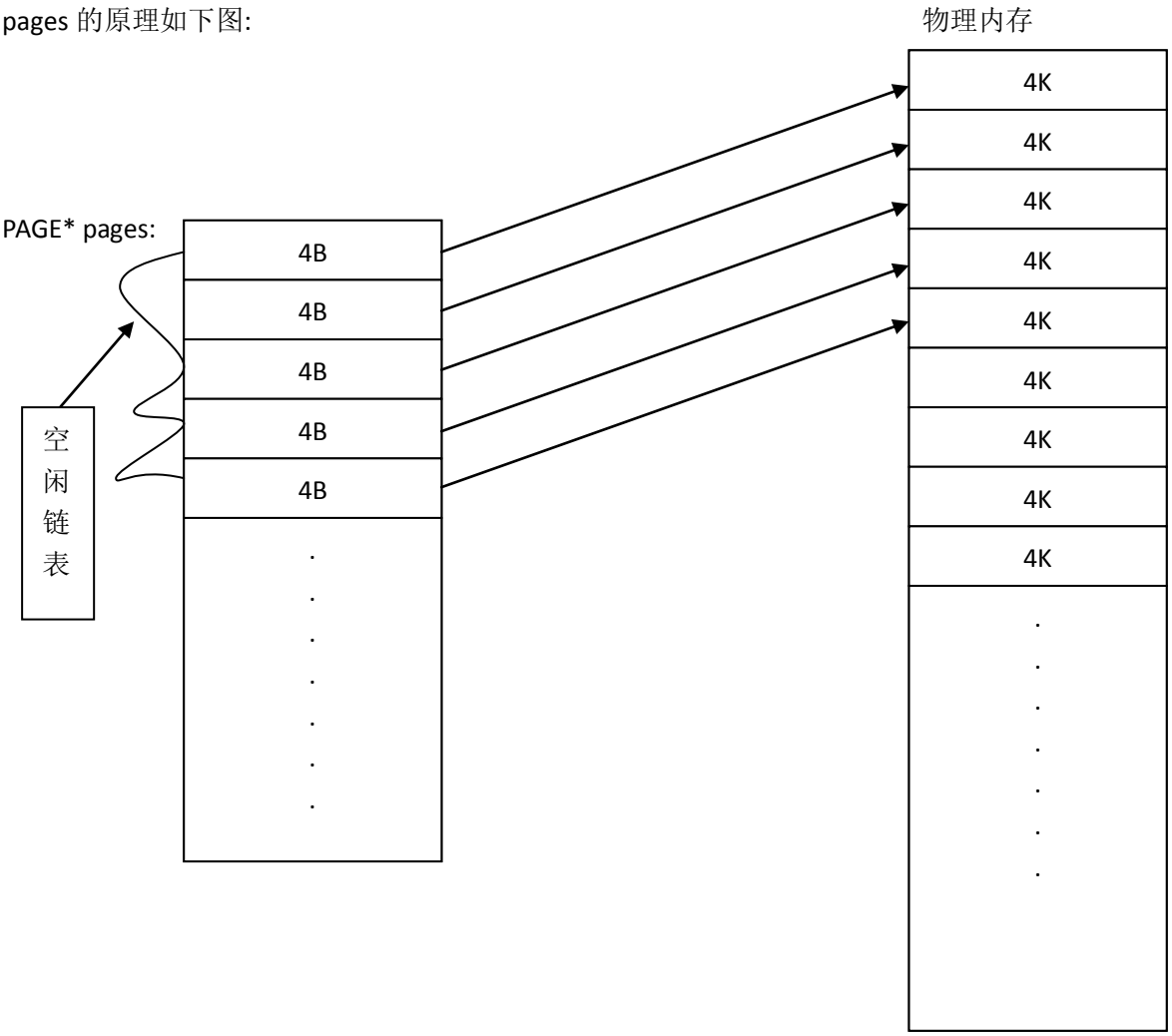
```
  pages = boot_alloc(npage * sizeof( struct Page),PGSIZE);
```

分配 pages
分配完以后,内存布局如下:

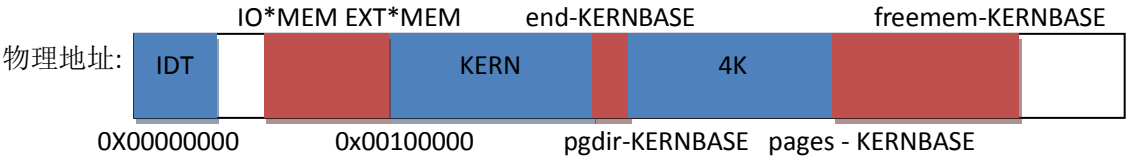


0xf0100000 pgdir pages

接着是 page_init:
要明白这个函数到底是什么回事,必须要明白 pages 数组,到底是怎么回事.
pages 的原理如下图:



这里,我们只要知道 pages[x]的地址,就可以计算出对应的物理内存页的物理地址（很容易算的）.然后 pages[x]里面有存储着链表需要的指针,这个链表是记录着那些也是空闲的.pages[x]里面还保存了一个 pp_ref,用来记录页面的引用次数.
然后,我们看看物理内存里面的数据分布是怎样,如下:



到这里，page_init 的作用就是把以上有颜色的部分标记为已经使用。然后再空闲链表中把以上页删除。
代码如下：

```
        pages[0].pp_ref = 1;
        LIST_REMOVE( &pages[0], pp_link);
        for( i = IOPHYSMEM; i < EXTPHYSMEM; i += PGSIZE){
                pages[i / PGSIZE].pp_ref = 1;
                LIST_REMOVE( &pages[i / PGSIZE], pp_link);
        }
        for(i = EXTPHYSMEM; i < PADDR((unsigned int) boot_freemem); i += PGSIZE){
                pages[i / PGSIZE].pp_ref = 1;
                LIST_REMOVE( &pages[i / PGSIZE], pp_link);
        }
```

现在进入 page_alloc 函数：
这个函数是用来分配一个页。
具体是从空闲链表中删除一个页，然后返回这个页。代码如下：

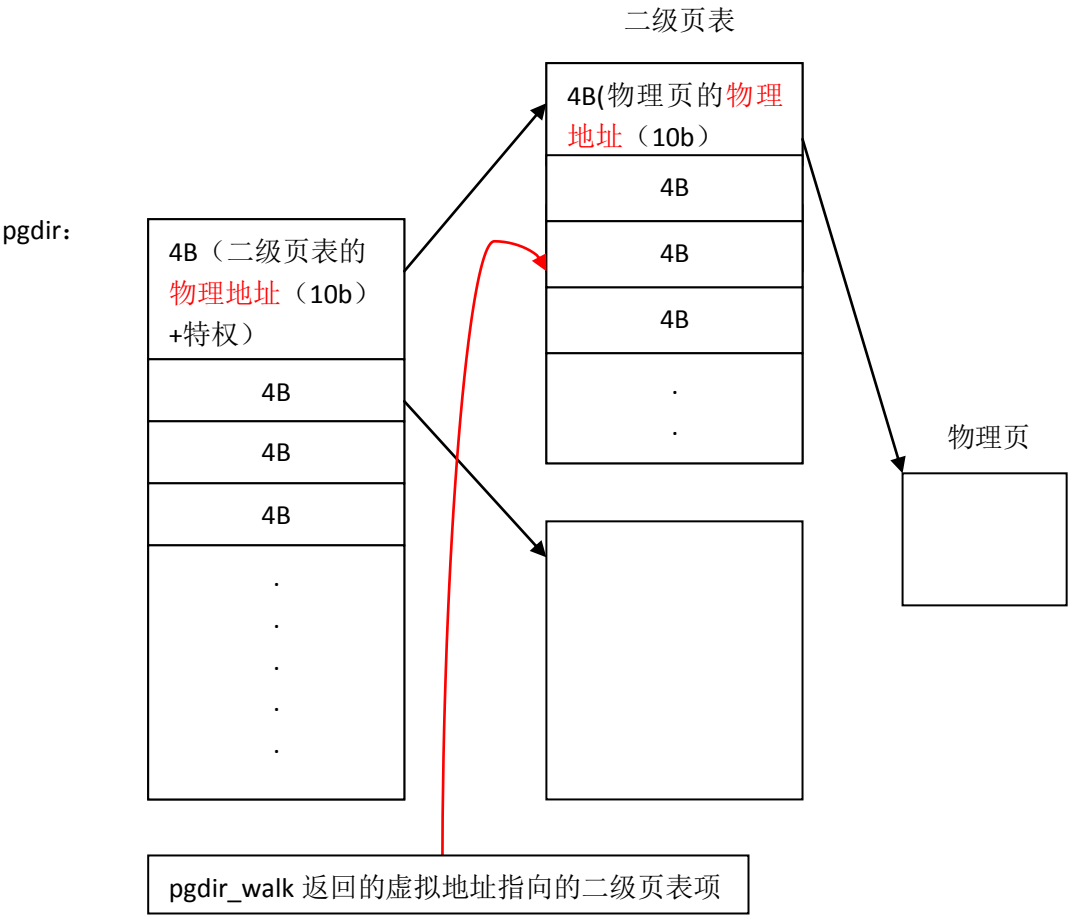
```
int
page_alloc(struct Page **pp_store)
{
        // Fill this function in
        if( LIST_FIRST( &page_free_list) == NULL){
        //      cprintf("function: page_alloc fail!!\n");
                return -E_NO_MEM;
        }else {
                *pp_store = LIST_FIRST( &page_free_list);
                LIST_REMOVE( *pp_store,pp_link);
        //      page_initpp(*pp_store);
                return 0;
        }
}
```

page_free 函数：
就是从空闲链表中插入一个空闲页，代码如下：

```
void
page_free(struct Page *pp)
{
        // Fill this function in
        if(pp -> pp_ref != 0){
                return;
        } else {
                page_initpp(pp);
                LIST_INSERT_HEAD( &page_free_list, pp, pp_link);
                return;
        }
}
```

pgdir_walk 函数：
这个函数是一个重要的函数，基本上如果明白了这个函数之后，所有的函数都理解，也就是说，理解了这个函数之后，整个虚拟内存管理系统都会了然于胸。
具体功能是：给出线性地址，然后返回线性地址对应的二级页表的页表项的虚拟地址（虚拟地址，因为在代码中用到的地址必须是虚拟地址，但是放在页目录和二级页表中的地址必须是物理地址，因为是 cpu 直接操作这些数据，不是我们的 JOS 操作）

现在先看看页目录的结构：



有了以上分析，实现这个函数就不难了，但是一定要搞清楚什么时候用虚拟地址，什么时候用物理地址，总的来说就是 cpu 自动进行的操作用物理地址，但是我们 JOS 进行的操作（也就是我们写的代码）就用虚拟地址。代码如下：

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
        pte_t *pt_addr_v;
        struct Page *pg;
        if(( pgdir[PDX(va)] & PTE_P) !=0){
                pt_addr_v =(pte_t *)KADDR(PTE_ADDR(pgdir[PDX(va)]));
                return &pt_addr_v[PTX(va)];
        /*
                cprintf("--->%x\n",(unsigned int)(&pt_addr_v[PTX(va)]));
                pt_addr_v = (pte_t*)(PTE_ADDR(pgdir[PDX(va)]));
                cprintf("--->%x\n",(unsigned int)pt_addr_v);
                cprintf("--->%x\n",pt_addr_v + PTX(va));
                return (pt_addr_v + PTX(va));
        */
        } else {
                if( create == 0){
                        return  NULL;
                } else {
                        if( page_alloc(&pg) != 0){
                                return  NULL;
                        }else {
                                pg->pp_ref = 1;
                                //this line below is unnecessary!But ......
                                memset(KADDR(page2pa(pg)), 0, PGSIZE);
                                pgdir[PDX(va)] = page2pa(pg);
                                pgdir[PDX(va)] = pgdir[PDX(va)] | PTE_U | PTE_W | PTE_P;
                                pt_addr_v = (pte_t *)KADDR(PTE_ADDR(pgdir[PDX(va)]));
                                return &pt_addr_v[PTX(va)];
                        /*      pt_addr_v =(pte_t *)PTE_ADDR(pgdir[PDX(va)]);
                                return (pt_addr_v + PTX(va));
                        */
                        }
                }
        }
        return NULL;
}
```

把上述道理弄明白，基本上已经达到我们实验的目的了。下面的函数实现不是事问题：

boot_map_segment 函数：

把指定的线性地址和物理地址绑定。这很简单，只要在线性地址对应的二级页表中写入物理地址对应的物理页的地址就可以了。代码如下：

```
static void
boot_map_segment(pde_t *pgdir, uintptr_t la, size_t size, physaddr_t pa, int perm)
{
        // Fill this function in
        unsigned int i;
        pte_t * pg;
        size = ROUNDUP(size, PGSIZE);
//      la = ROUNDDOWN(la, PGSIZE);
//      pa = ROUNDDOWN(pa, PGSIZE);
/*      for(i = la; i < la + size ;i += PGSIZE){
                pg = pgdir_walk(pgdir,(void *)i,1);
                if(pg == NULL){
                        cprintf("error in boot_map_segment\n");
                        assert(pg != NULL);
                }
        //      *pg = (pa+i ) | perm | PTE_P;
                *pg = (pa + i - la) | perm | PTE_P;
        }
*/
        for( i = 0;i < size; i +=PGSIZE){
                pg = pgdir_walk(pgdir,(void *)(la + i),1);
                if(pg == NULL){
                        assert(pg != NULL);
                }
                *pg = (pa + i) | perm |PTE_P;
        }
        return;
}
```

page_lookup 函数：
给出线性地址 va，我只能说代码里面把它写成虚拟地址害人不浅，虽然我不能说它错！
返回 va 对应的页的虚拟地址，就是 pages 里面的一项的地址。代码如下：

```
struct Page *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
        // Fill this function in
        pte_t *pg;
//      va =(void *)PADDR((unsigned int)va);
        pg = pgdir_walk(pgdir,va,0);
        if(pg == NULL){
                return 0;
        }
        if( pte_store != NULL){
                *pte_store = pg;
        }
        return pa2page(*pg);
}
```

page_remove 函数：
删除线性地址 va 对应的一个页。
这里把二级页表中的表项置 0，当然还要涉及到空闲链表，但是这些事情 page_decref 已经帮我们做了。代码如下：

```
void
page_remove(pde_t *pgdir, void *va)
{
        // Fill this function in
        struct Page *pg;
        pte_t *p_pte;
        pg = page_lookup(pgdir,va,&p_pte);
        if(pg == NULL){
                return;
        }else {
                page_decref(pg);
        }
        if(p_pte !=NULL){
                *p_pte = 0;
        }
        tlb_invalidate(pgdir,va);
}
```

page_insert 函数：
这个函数就是在线性地址 va 对应的二级页表项插入指定页。

具体代码如下：

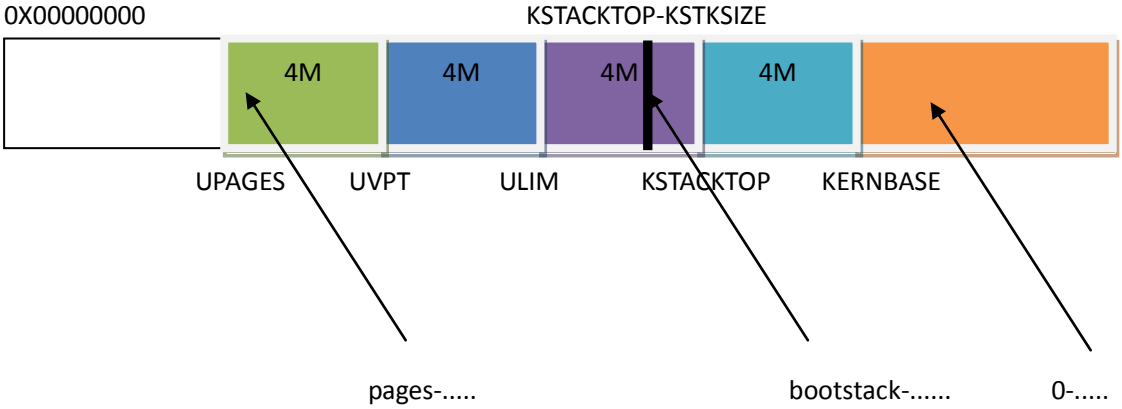
```
int
page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{
        // Fill this function in
        pte_t *pte;
        pte = pgdir_walk(pgdir,va,1);
        if( pte == NULL){
                return -E_NO_MEM;
        } else {
                pp -> pp_ref ++;
                if((*pte & PTE_P) !=0){
                        page_remove(pgdir,va);
                }
                *pte = page2pa(pp) |PTE_P |perm;
                return 0;
        }

        return 0;
}
```

现在是在 i386_vm_init 里面添加代码：

```
        // Make 'pages' point to an array of size 'npage' of 'struct Page'.
        // The kernel uses this structure to keep track of physical pages;
        // 'npage' equals the number of physical pages in memory.  User-level
        // programs will get read-only access to the array as well.
        // You must allocate the array yourself.
        // Your code goes here:

//      cprintf("---->pgdir \n");
//      int n;
//       n = ROUNDUP(npage*sizeof(struct Page), PGSIZE);
        pages = boot_alloc(npage * sizeof( struct Page),PGSIZE);

//      pages = boot_alloc(sizeof(struct Page) * npage,PGSIZE);
//      page_init();
//      cprintf("----->boot_alloc\n");
        //////////////////////////////////////////////////////////////////////
        // Now that we've allocated the initial kernel data structures, we set
        // up the list of free physical pages. Once we've done so, all further
        // memory management will go through the page_* functions. In
        // particular, we can now map memory using boot_map_segment or page_insert
        page_init();

//      cprintf("page_init\n");

        check_page_alloc();

        page_check();

        cprintf("---------->page_check\n");

        //////////////////////////////////////////////////////////////////////
        // Now we set up virtual memory
```

现在分页管理系统的基本函数已经完成，接下来就是要按照 JOS 作者的设计意图把线性地址映射到对应的物理地址.
具体如下图：

线性地址：



上面所写的 pages-.....的意思是 pages 的物理地址到后面的一段空间，这里是 pages-pages+npage*sizeof（Page），bootstack 同上，0 代表物理地址 0。

代码如下：

```
//////////////////////////////////////////////////////////////////////
// Map 'pages' read-only by the user at linear address UPAGES
// (ie. perm = PTE_U | PTE_P)
// Permissions:
//    - pages -- kernel RW, user NONE
//    - the read-only version mapped at UPAGES -- kernel R, user R
// Your code goes here:
n = ROUNDUP(npage * sizeof( struct Page),PGSIZE);
boot_map_segment(pgdir,UPAGES,n,PADDR(pages),PTE_P |PTE_U);


//////////////////////////////////////////////////////////////////////
// Map the kernel stack (symbol name "bootstack").  The complete VA
// range of the stack, [KSTACKTOP-PTSIZE, KSTACKTOP), breaks into two
// pieces:
//     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
//     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed => faults
//     Permissions: kernel RW, user NONE
// Your code goes here:

boot_map_segment(pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE,PADDR(bootstack), PTE_W |PTE_P);
//boot_map_segment(pgdir,KSTACKTOP - PTSIZE,PTSIZE - KSTKSIZE,0,0);
//////////////////////////////////////////////////////////////////////
// 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 amapping anyway.
// Permissions: kernel RW, user NONE
// Your code goes here:
boot_map_segment(pgdir,KERNBASE,0xFFFFFFFF - KERNBASE + 1,0,PTE_W|PTE_P);
// Check that the initial page directory has been set up correctly.
check_boot_pgdir();
```

关于一些特权级的说明如下：

```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PWT         0x008   // Write-Through
#define PTE_PCD         0x010   // Cache-Disable
#define PTE_A           0x020   // Accessed
#define PTE_D           0x040   // Dirty
#define PTE_PS          0x080   // Page Size
#define PTE_MBZ         0x180   // Bits must be zero
```

EXERCISE 6

问题一：上面的图已经回答一切。

问题二：这个问题比较有趣

我们之前通过

```
// Your code goes here:
boot_map_segment(pgdir, KERNBASE, 0xFFFFFFFF - KERNBASE + 1, 0, PTE_W|PTE_P);
// Check that the initial page directory has been set up correctly.
```

把线性地址[KERNBASE,0XFFFFFFFF)映射到物理地址的[0,0XFFFFFFFF - KERNBASE +1) 处

也就是说，如果分页机制开了，我们给出线性地址 0xf0100000 ++ ，那么就会映射到物理地址 0x00100000++上。

由于当前虚拟地址的值是 0XF0100000 ++ ，经过段转换之后变成 0x00100000，如果没有开启分页机制，那么刚好的内核所在的物理地址。

但是如果开启了分页机制之后，0x00100000 二进制是 0000000000 0100000000 0000000000000，对应的是页目录的第一项，但是页目录的第一项我们是没有设置映射的，所以虚拟地址对应的物理地址是？？？？我们不知道，但是如果我们把 pgdir[0] = pgdir[pdx(kernbase)]之后，pgdir[pdx(0x00100000)]和 pgdir[pdx(0xf0100000)]的值是一样的，也就是说线性地址 0x00100000++也是映射到了物理地址 0x00100000++后面。

问题三：

因为有保护位的原因 PTE_W　PTE_U 这些宏决定了内存空间的可访问性。

问题四：

4G，因为是 32 位的地址线。

问题五：

开销是：npage * sizeof(Pages) + pgdir[4k] + 1024 * 4k(如果全部二级页表都用上的话)，如果是 4G 的内存，那么 npage 的值就会使 4G/4096

如果完全明白上面的东西 challenge 应该问题不大。