JOS2实验报告

王正1110379017

2012 年 7 月 12 日

1 准备工作

首先需要提交lab1的源代码。

```
wang@ThinkPad:~/lab$ git commit -am 'my␣solution␣to␣
   lab1'
[lab1 934bc25] my solution to lab1
 4 files changed, 82 insertions(+), 7 deletions(-)
```

然后建立新的分支。

```
wang@ThinkPad:~/lab$ git pull
Already up-to-date.
wang@ThinkPad:~/lab$ git checkout -b lab2 origin/lab2
Branch lab2 set up to track remote branch lab2 from
   origin.
Switched to a new branch 'lab2'
```

合并lab1的代码。

```
wang@ThinkPad:~/lab$ git merge lab1
Merge made by recursive.
 kern/console.c |    7 ++++++-
 kern/kdebug.c  |    7 ++++++--
 kern/monitor.c |   39
    ++++++++++++++++++++++++++++++++++++++++
 lib/printfmt.c |   36
    ++++++++++++++++++++++++++++++++----
 4 files changed, 82 insertions(+), 7 deletions(-)
```

在合并的过程中发现，因为自己曾经修改过kern/init.c 增加了彩色打印的功能，

```
cprintf("color test: \033[0;32;40m hello \033[0;36;41mworld\033[0;37;40m\n");
```

但是发现无法合并，所以就恢复了init.c到初始状态下。

2 物理页面管理

lab2主要是实现内存的分页管理和虚存的实现。通常管理内存的方式有两种，分段和分页，这里jos实现的是分页管理。

> 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.

代码集中在kern/pmap.c中。主要修改boot_alloc()，mem_init()，page_init()，page_alloc()，page_free()这几个函数。可以通过check_page_free_list() 和check_page_alloc()检测分配是否正确。正确之后会有check_page_alloc() succeeded!的提示。boot_alloc()初始化页目录和页表的空间。在pmap.c中mem_init()调用

```
// create initial page directory.
kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
memset(kern_pgdir, 0, PGSIZE);
```

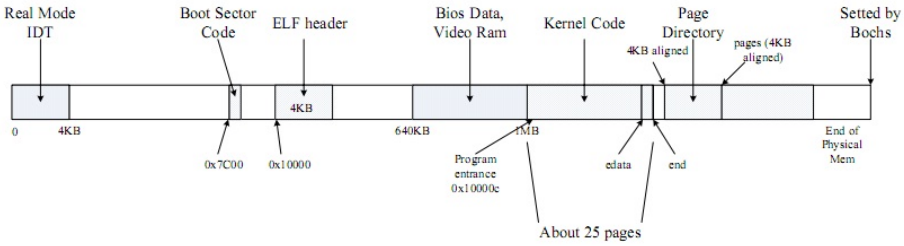
和

```
// array.  'npages' is the number of physical pages in memory.
// Your code goes here:

pages =(struct Page *) boot_alloc(npages* sizeof (struct Page));
memset(pages, 0, npages* sizeof (struct Page));
```

实现，这里页表是模仿页目录的写法而写成的。下面可以看一下页表的数据结构

```
struct Page {
struct Page *pp_link;
uint16_t pp_ref;
};
```



这个就是个简单的链表的节点，pp_link指向下个节点，pp_ref表示引用次数。空闲的节点引用次数为0. 页目录和页面链表紧接着内核空间后分配。 我们可以从这张图上看到实际上的内核层次。Page Directory 在内核代码end之后按页对齐的地方开始，在一页之后是页面的链表。所以boot_alloc()主要功能就是给这两项分配内存。

```
// 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.
result = ROUNDUP(nextfree, PGSIZE);
nextfree = result + n;
//cprintf("\nnextfree:0x%08x",nextfree);
return result;
```

所做的添加如上，这里result是分配的内存起始位置，nextfree是下一块空闲区域开始的地方。这里的地址都不是是物理地址，而是虚拟地址，返回的是指向该虚拟地址的指针。

page_init()是对页面链表进行初始化。初始化的方法是，设定page_free_list指向空闲链表的表头。这里需要处理下不可使用的内存空间，令他们的pp_ref=1。不可使用的内存空间共有两种情况。1是页表0，2是从IOPHYSMEM到上面所说的页面链表分配位置的结束处。具体的实现如下：

```
size_t i;
//size_t a=0;
//size_t b=0;
//size_t c=0;
page_free_list = NULL;
physaddr_t pgnum_IOPHYSMEM = PGNUM (IOPHYSMEM);
physaddr_t pgnum_EXTPHYSMEM =PGNUM ( PADDR
```

```
(ROUNDUP(pages+npages* sizeof (struct Page),PGSIZE)));
//PGNUM (ROUNDUP(pages+npages* sizeof
//(struct Page),PGSIZE))-PGNUM(kern_pgdir)+PGNUM(EXTPHYSMEM);
for (i = 1; i < npages; i++)
{


if(i<pgnum_IOPHYSMEM)
{
pages[i].pp_ref = 0;
pages[i].pp_link = page_free_list;
page_free_list = &pages[i];
//a++;
}
else if( i>pgnum_EXTPHYSMEM)
{
pages[i].pp_ref = 0;
pages[i].pp_link = page_free_list;
page_free_list = &pages[i];
//b++;
}
else
{
pages[i].pp_ref = 1;
//c++;
}
}
```

这里实现之前犯了一个错误，就是直接将ROUNDUP(pages+npages* sizeof (struct Page),PGSIZE))的逻辑地址取页表号的操作，这样的话页表号就会很大，导致断言assert(nfree_extmem > 0);报错，也就是在大于已分配空间没有了空闲地址。所以实际上要取物理页号的操作。也就是将该地址减去KERNBASE 0xF0000000。减去之后可以使得断言通过。之后特别的用physaddr_t来强调一下这里指的是实际地址页号。page_init()基本完成。

page_alloc()是分配页表的过程，简而言之就是从page_free_list取出一个节点，将其对应的页表分配出去，并且空闲列表中删除出去。具体实现很简单：

```
struct Page *
page_alloc(int alloc_flags)
{
```

```
// Fill this function in
if ((alloc_flags==0 ||alloc_flags==ALLOC_ZERO)
&& page_free_list!=NULL)
{
struct Page * temp_alloc_page = page_free_list;
if(page_free_list->pp_link!=NULL)
page_free_list=page_free_list->pp_link;
else
page_free_list=NULL;
if(alloc_flags==ALLOC_ZERO)
memset(page2kva(temp_alloc_page), 0, PGSIZE);
return temp_alloc_page;
}
else
return NULL;
}
```

这里也遇到了一个问题是对alloc_flags的定义不明确，查看断言之后估计ALLOC_ZERO是清空内存为0的操作，所以就memset 分配的页为0.而且注意的一点是这里memset是对temp_alloc_page代表的那一页清空。所以需要找出temp_alloc_page代表的那页的虚拟地址。用page2kva 即可实现。

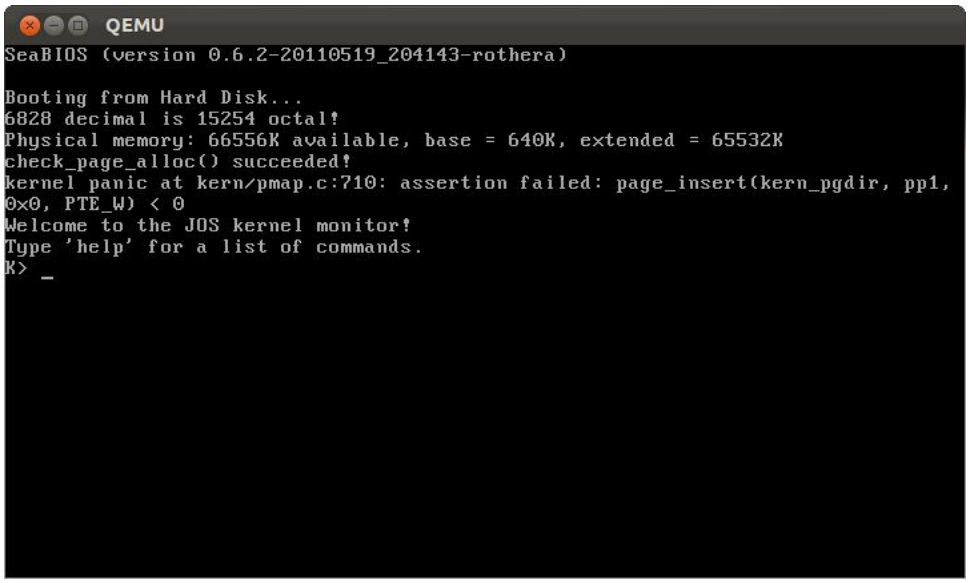
page_free()就是将一个页表节点再放回空闲列表中。再将该节点的pp_ref设置为0即可。

```
void
page_free(struct Page *pp)
{
// Fill this function in
pp->pp_ref = 0;
pp->pp_link = page_free_list;
page_free_list = pp;
}
```

经过如上的修改之后，运行结果如下：

3 虚拟存储

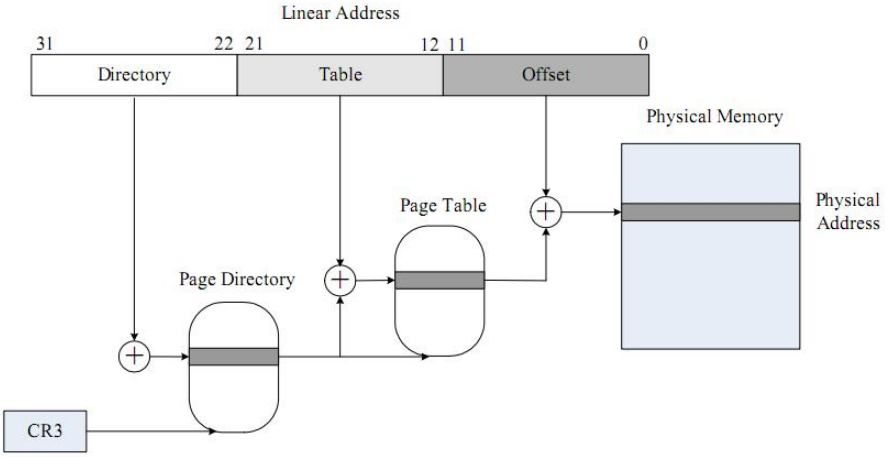
> Exercise 2. Look at chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. Read the sections about page translation



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.

由于之前看过深入理解Linux内核的部分内容，所以对内存管理这块还是有些熟悉的。

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.

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.

页面转换的示意图如图所示： 这里pgdir_walk()完成的过程是针对输入的线性地址va,a返回一个指针指向页表项。这就需要走完二级页表。也就是上图中的Table中的表项。这里开始时产生了误解，以为是如果没有的话就产生一个新的物理页，这里发现如果不存在的话是新产生一个新的二级Table。该函数的具体实现如下：

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
// Fill this function in
```

```
pde_t *pde;//page directory entry,
pte_t *pte;//page table entry
pde=(pde_t *)pgdir+PDX(va);//get the entry of pde

if (*pde & PTE_P)//the address exists
{
pte=(pte_t *)KADDR(PTE_ADDR(*pde))+PTX(va);
//point to the virtual address of PTE
return pte;
}
//the page does not exist
if (create )//create a new page table
{
struct Page *pp;
pp=page_alloc(ALLOC_ZERO);
if (pp!=NULL)
{
pp->pp_ref=1;
*pde = page2pa(pp)|PTE_U|PTE_W|PTE_P ;
pte=(pte_t *)KADDR(PTE_ADDR(*pde))+PTX(va);
return pte;
}
}
return NULL;
}
```

需要注意的是，这里页表中的项指向的都是虚拟地址。但是页表项中储存的内容都是实际的页的编号加上一些控制位。控制位的选择是在pde表项中给予最多的权限，也就是Hint 2中的内容。

- P – Present，该位用来判断对应物理页面是否存在，如果存在，该位置1，否则为0；
- R/W – Read/Write，该位用来判断对所指向的物理页面的访问权限，如果为1，表示页面可写，否则，是只读页面；
- U/S – User/Supervisor，该位用来定义页面的访问者应该具备的权限。如果为1，表示该页面是User权限的，大家都可以访问，如果为0，表示只能是Ring0 中的程序能访问；

boot_map_region() 作用是将虚拟地址[va, va+size)映射到物理地址[pa, pa+size)。这里的size要求是页表大小的整数倍，否则就向上取整。该函数基本

也就是一段物理地址空间初始化映射到page table中的过程。

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va,
size_t size, physaddr_t pa, int perm)
{
// Fill this function in
if(size%PGSIZE!=0)
size=ROUNDUP(size,PGSIZE);//panic(" Size must be a multiple of PGSIZE.");
pte_t *pte ;
size_t i=0;
while(i<size)
{
pte=pgdir_walk(pgdir, (void *)va, 1);
*pte= pa|perm;
pa+=PGSIZE;
va+=PGSIZE;
i+=PGSIZE;
}
}
```

page_lookup()是返回一个虚拟地址va所对应的页，也就是这个页对应在链表中的地址。并且将这个页储存在store中。

```
struct Page *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
pte_t *pte = pgdir_walk(pgdir,(void *)va, 0);
if (pte==NULL)
{
return NULL;
}
if (pte_store != 0)
{
*pte_store = pte;
}
if (*pte & PTE_P)
{
return pa2page (PTE_ADDR (*pte));
}
```

```
return NULL;
}
```

page_remove()也就是取消va对某个物理地址的映射。

```
void
page_remove(pde_t *pgdir, void *va)
{
pte_t *pte;
struct Page *pp;
    pp=page_lookup (pgdir, va, &pte);
if (pp != NULL)
{
page_decref (pp);
//- The ref count on the physical page should decrement.
//- The physical page should be freed if the refcount reaches 0.
if(pte!=NULL)
*pte = 0;
// The pg table entry corresponding to 'va' should be set to 0. (if such a PTE exists)
tlb_invalidate (pgdir, va);
//The TLB must be invalidated if you remove an entry from  the page table.
}
}
```

page_insert()将虚拟地址va映射到物理地址上。

```
int
page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{

pte_t * pte = pgdir_walk(pgdir, (void *)va, 1) ;
//   - If necessary, on demand, a page table should be allocated and inserted
//     into 'pgdir'.
if (pte==NULL)
return -E_NO_MEM;
//-E_NO_MEM, if page table couldn't be allocated
if (*pte & PTE_P) {
if (PTE_ADDR(*pte) == page2pa(pp))
{
tlb_invalidate(pgdir, va);
//The TLB must be invalidated if a page was formerly present at 'va'.
```

```
}
else
{
page_remove (pgdir, va);
//If there is already a page mapped at 'va', it should be page_remove()d.
}
}

*pte = page2pa(pp)|perm|PTE_P;
if (!(*pte&PTE_P&PTE_ADDR(*pte) == page2pa(pp)))
pp->pp_ref++;
//pp->pp_ref should be incremented if the insertion succeeds.
return 0;
//0 on success
}
```

这里的映射一共有三种情况。

- 原先的pte有效，就是该页，删除tlb的值，并且修改该pte的值为此页即可；
- 原先的pte有效，不是该页，删除映射，并且修改该pte的值为此页即可；
- 原先的pte表项无效，这样直接修改该pte的值为此页即可；

4 内核地址空间

> 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.

该练习也就是将内核的一部分空间初始化。设置UTOP以上部分的地址空间。这里的三个设置都有很明显的注释。按照注释的要求调用函数即可。

```
//////////////////////////////////////////////////////////////////////
// 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_region(kern_pgdir,UPAGES,npages * sizeof (struct Page),
PADDR (pages), PTE_U|PTE_P);
//////////////////////////////////////////////////////////////////////
// Use the physical memory that 'bootstack' refers to as the kernel
// stack.  The kernel stack grows down from virtual address KSTACKTOP.
// We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
// to be the kernel stack, but break this into two pieces:
//     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
//     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
//       the kernel overflows its stack, it will fault rather than
//       overwrite memory.  Known as a "guard page".
//     Permissions: kernel RW, user NONE
// Your code goes here:
boot_map_region (kern_pgdir,KSTACKTOP - KSTKSIZE,KSTKSIZE,
PADDR(bootstack),PTE_W|PTE_P);
//////////////////////////////////////////////////////////////////////
// Map all of physical memory at KERNBASE.
// Ie.  the VA range [KERNBASE, 2^32) should map to
//      the PA range [0, 2^32 - KERNBASE)
// We might not have 2^32 - KERNBASE bytes of physical memory, but
// we just set up the mapping anyway.
// Permissions: kernel RW, user NONE
// Your code goes here:
boot_map_region (kern_pgdir,KERNBASE,0xffffffff-KERNBASE+1,0,PTE_W|PTE_P);
```

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;
   ```

   内核中的操作都在虚地址下完成，所以该操作也是uintptr_t。

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]

一共用boot_map_region()初始化了三块区域，虚拟地址分别为UPAGES，KSTACKTOP - KSTKSIZE和KERNBASE。根据虚拟地址的前十位计算表项中的位置。其中KSTACKTOP - KSTKSIZE只是ULIM到KSTACKTOP中的一段区域，但是要求ULIM到KSTACKTOP - KSTKSIZE是无效的，相当于这一段也被映射了。填出来的表如下：

Entry	Base Virtual Address	Points to (logically):
1023	?	Page table for top 4MB of phys memory
.	?	?
960	0xf0000000	KERNBASE
958	0xef800000	ULIM
956	0xef000000	UPAGES
.	?	
0	0x00000000	[see next question]

3. (From Lecture 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?

   由于在boot_map_region()使用了控制符，比如说PTE_W，设定了权限，所以只有内核有权限进行访问。

4. What is the maximum amount of physical memory that this operating system can support? Why?

   最多支持4G的物理内存，32位系统最多支持4G的物理内存，超过4G的内存无法寻址。

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?

一共有一个页目录，npages* sizeof (struct Page)/PGSIZE个页表。而npages是实际系统的内存中的页数。如果是最大物理内存，4G，则有1025*4k的空间用来管理内存。降低额外空间的方法是使用扩展分页技术，也就是采用4M的页大小。

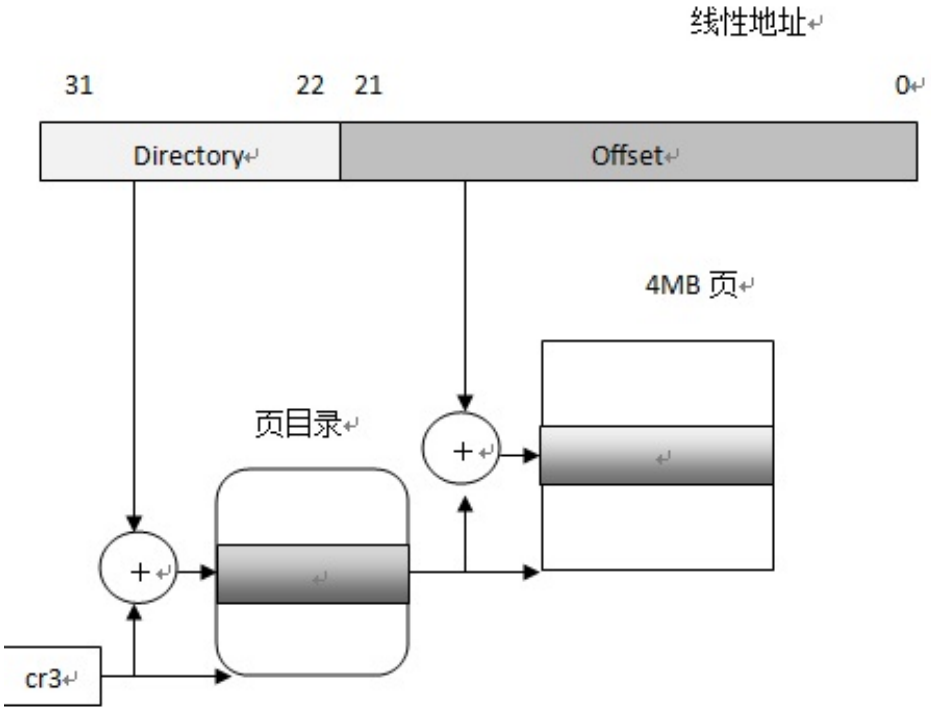
6. Revisit the page table setup in kern/entry.S and kern/entrypgdir.c. Immediately after we turn on paging, EIP is still a low number (a little over 1MB). At what point do we transition to running at an EIP above KERNBASE? What makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE? Why is this transition necessary? 在下面这段代码下使得EIP在KERNBASE之上。

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

由于实际的物理地址代表的虚拟地址和高地址的虚拟地址指向的是同一块物理地址，这就使得在开始分页和提升EIP之间能够运行。

> 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!

根据深入理解Linux内核中第二章的解释，Intel在80x86处理器引入了扩展分页技术，允许的页框大小是4MB而不是4KB。通过设置页目录项的Page Size标志启用扩展分页功能。通过设置cr4处理器的PSE标志位可以使得扩展分页和常规分页并存。而且20位物理地址只有最高10位是有意义的。


线性地址
31
22
21
0
Directory
Offset
4MB 页
页目录
cr3