

OSlab4

18300240005 罗铭源

make grade 结果

```
K> rivership@ubuntu:~/share/share/OS/lab4$ make grade
make clean
make[1]: Entering directory '/home/rivership/share/share/OS/lab4'
rm -rf obj .gdbinit jos.in qemu.log
make[1]: Leaving directory '/home/rivership/share/share/OS/lab4'
./grade-lab2
make[1]: Entering directory '/home/rivership/share/share/OS/lab4'
+ as kern/entry.S
+ cc kern/entrypgdir.c
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/pmap.c
+ cc kern/kclock.c
+ cc kern/printf.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 390 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/rivership/share/share/OS/lab4'
running JOS: (1.7s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
```

exercise1

`boot_alloc()` `mem_init()` `page_init()` `page_alloc()` `page_free()`

boot_alloc

先求出分配的边界(对齐 `PGSIZE`), 检查该边界是否超出物理内存的范围, 若不出, 就将 `nextfree` 向后移动到边界处, 并返回原来的 `nextfree` 值

```
char *tmp = ROUNDUP(nextfree+n, PGSIZE);
if (PGNUM(PADDR(tmp)) >= npages)
    panic("boot_alloc: out of memory\n");
result = nextfree;
nextfree = tmp;
return result;
```

page_init

page_init 的任务是初始化 pages，设置部分页面的分配关系，以及将 free 的页面串在链表 page_free_list 上

根据注释，pages[0] 预先保留，设置为已分配状态

```
pages[0].pp_ref=1;
pages[0].pp_link=NULL;
```

[PGSIZE, npages_basemem*PGSIZE) 部分的物理内存为 free，串到 page_free_list 上

```
for (i = 1; i < npages_basemem; i++) {
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}
```

[IOPHTSMEM, EXTPHYSMEM) 留给IO设备，设置为已分配

```
for(i=PGNUM(IOPHYSMEM); i<PGNUM(EXTPHYSMEM); i++){
    pages[i].pp_ref = 1;
    pages[i].pp_link = NULL;
}
```

接下来是 EXTPHYSMEM 往后的内存。在调用 boot_alloc 前，[EXTPHYSMEM, end) 都是已被分配的 (end 指向 kernel bss 结尾)。调用 boot_alloc，会后推 nextfree 来分配page (原本指向 end)，所以 [EXTPHYSMEM, nextfree) 内的物理页都是已分配的，可以通过 boot_alloc(0) 来获得 nextfree

```
char* NextFreePage = boot_alloc(0);
size_t npages_next_free = PGNUM(PADDR(NextFreePage));
for(i = PGNUM(EXTPHYSMEM); i<npages_next_free; i++){
    pages[i].pp_ref = 1;
    pages[i].pp_link = NULL;
}
```

page_alloc

首先检查是否还有空余page

```
if(!page_free_list)
    return NULL;
```

若有空闲page，将其从 page_free_list 中取出，并对 page_free_list 状态进行更新

```
struct PageInfo * victim_page;
victim_page = page_free_list;
page_free_list = victim_page->pp_link;
victim_page->pp_link = NULL;
```

若调用函数要求 `ALLOC_ZERO`，就将该内存页使用 `memset` 设置成 0，由于 `memset` 接受虚拟地址作为参数，且我们处于内核态，所以需要 `page2kva`

```
if(alloc_flags & ALLOC_ZERO)
    memset(page2kva(victim_page), 0, PGSIZE);
return victim_page;
```

page_free

检查 `pp_ref` 和 `pp_link` 是否为 0。若均非 0，将 `pp` 插入 `page_free_list`

```
void
page_free(struct PageInfo *pp){
    if(pp->pp_ref || pp->pp_link){
        panic("page_free: pp_ref or pp_link not zero");
    }
    pp->pp_link = page_free_list;
    page_free_list = pp;
}
```

mem_init

exercise1 中的 `mem_init` 要求写到 `check_page_free_list(1)` 之前，只需要补充对 `pages` 的空间分配（当然还需要写好 `page_init`）

为 `pages` 分配空间只需要使用我们写的 `boot_alloc` 即可

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

exercise1 结果

可以看到 `check_page_free_list` 以及 `check_page_alloc` 已经通过

```
K> rivership@ubuntu:~/share/share/OS/lab4$ make qemu
+ cc kern/pmap.c
+ ld obj/kern/kernel
ld: warning: section '.bss' type changed to PROGBITS
+ mk obj/kern/kernel.img
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
kernel panic at kern/pmap.c:717: assertion failed: page_insert(kern_pgdir, pp1,
0x0, PTE_W) < 0
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```

exercise4

`pgdir_walk` `boot_map_region` `page_lookup` `page_remove` `page_insert`

pgdir_walk

该函数返回在页目录 `pgdir` 中，虚拟地址 `va` 对应的页表项 (PTE)，参数 `create` 可以指定当 `va` 对应的页表还未申请时，是否申请一个page来存放页表。

首先找到 `va` 对应的页目录项，即指向页表的指针。宏 `PTE_ADDR` 将 `PDE` 的后12bit全部置0，即去掉 `permission bit`

```
pte_t *pgtable = (pte_t *)PTE_ADDR((physaddr_t)pgdir[PDX(va)]);
```

随后，判断是否有 `pgtable == NULL`，若有，则需要根据 `create` 决定是否要申请页表

```
if(!pgtable){
    // if caller don't want to create a page table
    if(!create){
        return NULL;
    }
    else{
        // check if there is one free page to alloc
        if(! (pp = page_alloc(ALLOC_ZERO)))
            return NULL;
        pp->pp_ref ++;
        pgtable = (pte_t *)page2pa(pp);
        pgdir[PDX(va)] = (physaddr_t)pgtable | PTE_P | PTE_U | PTE_W;
    }
}
```

最后需要返回页表中 `va` 对应的 `pte` 的地址（注意页表项中存的是物理地址，需要转成kernel的虚拟地址，实际上就是加上一个 `kernbase`）。

```
return KADDR( (uint32_t)pgtable + PTX(va)*sizeof(pte_t) );
```

boot_map_region

参数要求对齐 `PGSIZE`，进行检查。

(`assert` 是一个可以根据给定布尔条件是否满足来调用 `panic` 的宏，错误时它还能输出自身所判断的条件，对于定位错误很有帮助)

```
assert( va%PGSIZE==0 && size%PGSIZE==0 && pa%PGSIZE==0 );
```

从 `va` 开始，逐个将页面映射到物理地址

```

for(i=0;i<num;i++){
    pte = pgdir_walk(pgdir,(void*)(va+i*PGSIZE),1);
    // check if pte is NULL (when there is no free pages, it is NULL)
    assert(pte);
    *pte = (pa+i*PGSIZE) | perm | PTE_P;
}

```

page_lookup

该函数返回虚拟地址 `va` 在页目录 `pgdir` 中对应的物理页，用 `PageInfo` 结构表示，同时还返回对应的 `pte` 的指针

其中，对应的 `pte` 的指针只要用 `pgdir_walk` 获得，同时要检查 `pgdir_walk` 是否返回了 `NULL`，即是否有 `va` 对应的 `page table` 还未分配的情况（还要检查函数的参数 `pte_store` 是否为 `NULL`，否则会有引用空指针）

```

assert(pte_store != NULL);
*pte_store = pgdir_walk(pgdir,va,0);
if(!(*pte_store))
    return NULL;

```

随后还要判断页表项的内容是否为 `NULL`，即是否有 `**pte_store == NULL`，这里若不检查其是否为 `NULL`，然后直接返回 `pa2page` 的结果的话，就会实际上返回了 `pages` 而不是 `NULL`

```

return PTE_ADDR(**pte_store) ? pa2page(PTE_ADDR(**pte_store)) : NULL ;

```

page_remove

`page_remove` 在页目录 `pgdir` 中去除一个对 `va` 的映射

首先通过 `page_lookup` 查看对 `va` 的映射，获取 `va` 对应的物理页面

```

pte_t *pte;
struct PageInfo* pp = page_lookup(pgdir,va,&pte);
// if va is not mapped, do nothing
if(!pp){
    return;
}

```

随后将物理页面的引用数减一，并且将 `tlb` 中关于 `va` 的 `pte`（若存在）无效化

最后将 `pte` 置0

```

page_decref(pp);
tlb_invalidate(pgdir,va);
*pte = 0;

```

page_insert

`page_insert` 需要建立一个 `va ---> pp` 的映射

首先获得`va`对应的页表项地址

```
pte_t *pte = pgdir_walk(pgdir, va, 1);
```

如果 `pte==NULL` , 由于在 `pgdir_walk` 中设置了 `create=1` , 所以只可能是已经没有可用的物理页了, 于是 `return -E_NO_MEM`

当 `pte != NULL` 时, 检查页表项的内容 `*pte` 是否为0;

若 `*pte != 0` , 说明已经存在一个`va`的映射, 注意**此时需要检查该映射是否与所要求建立的映射是同一个映射。**

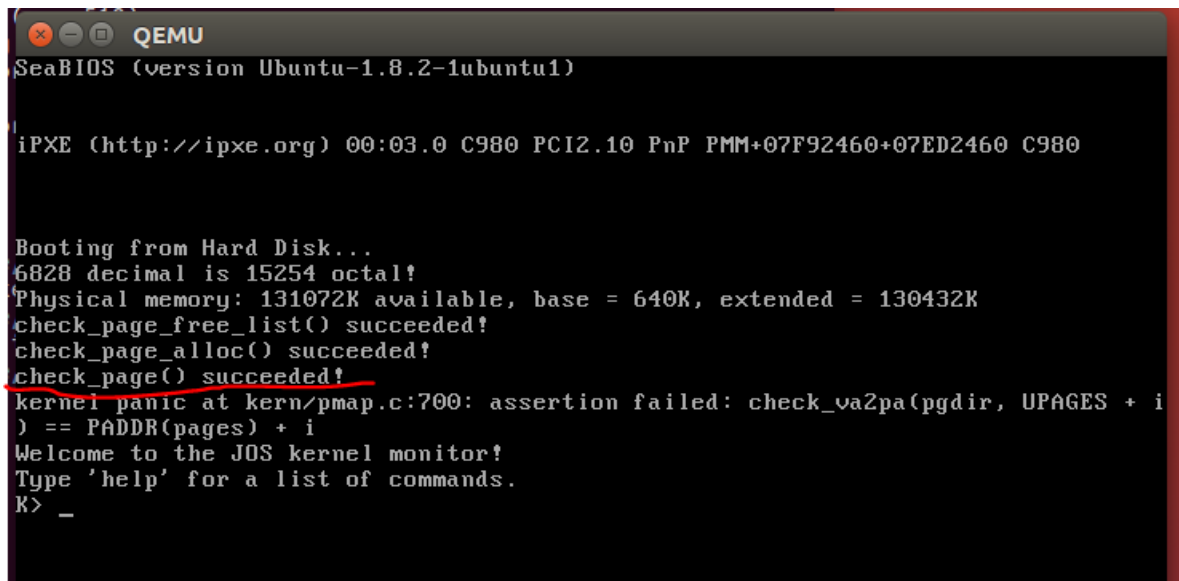
```
if(*pte){
    //va-->pp doesn't exist
    if(PTE_ADDR(*pte) != page2pa(pp)){
        pp->pp_ref++;
        page_remove(pgdir, va);
    }
    //va-->pp already exist
    *pte = page2pa(pp) | perm | PTE_P;
    tlb_invalidate(pgdir, va);
}
```

完整代码如下

```
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    // Fill this function in
    pte_t *pte = pgdir_walk(pgdir, va, 1);
    //check if there are no free pages for a new pagetable
    if(pte){
        //here, *pte should not equal NULL
        //when *pte==NULL and pp is pages[0]
        // will lead to pp->ref++, which is wrong action
        if(*pte){
            //va-->pp doesn't exist
            if(PTE_ADDR(*pte) != page2pa(pp)){
                pp->pp_ref++;
                page_remove(pgdir, va);
            }
            //va-->pp already exist, do nothing but change the content
            *pte = page2pa(pp) | perm | PTE_P;
            tlb_invalidate(pgdir, va);
        }
        else{
            pp->pp_ref++;
            *pte = page2pa(pp) | perm | PTE_P;
        }
        return 0;
    }
    else
        return -E_NO_MEM;
}
```

exercice4 结果

make qemu, 已经通过 check_page 检查



```
QEMU
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92460+07ED2460 C980

Booting from Hard Disk...
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
kernel panic at kern/pmap.c:700: assertion failed: check_va2pa(pgdir, UPAGES + i) == PADDR(pages) + i
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> _
```

exercise5

该 exercise 要求在 mem_init 给虚拟地址空间搭建一些区域的映射, 如 kernel stack, user pages 等等

首先, 将 pages 数组的物理区域 map 给虚拟地址 UPAGES 开始的虚拟地址区域, size = sizeof(struct PageInfo)*npages

```
size_t SIZE = ROUNDUP(sizeof(struct PageInfo)*npages, PGSIZE);
boot_map_region(kern_pgdir, UPAGES, SIZE, PADDR(pages), PTE_U | PTE_P);
```

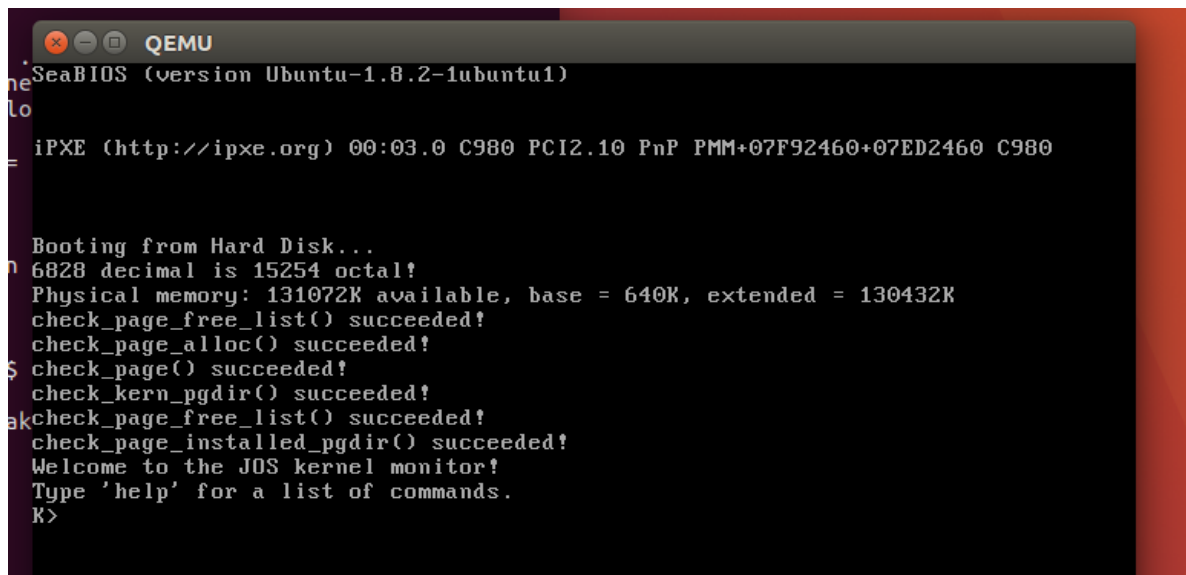
将 virtual address: [KSTACKTOP-KSTKSIZE, KSTACKTOP) map 到 bootstack 的物理地址

```
boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE_W);
```

将 [KERNBASE, 2³²) ---> [0, 2³² - KERNBASE), 这里映射中的物理地址有超出物理内存的大小, 不过仍然设置映射

```
boot_map_region(kern_pgdir, KERNBASE, 0x100000000-KERNBASE, 0, PTE_W);
```

make qemu, 发现能够正常进入 monitor



showva2pa

在 `commands` 中添加新指令 `showva2pa`，其中 `mon_showva2pa` 是 `monitor` 解析完指令后调用的函数

```
static struct Command commands[] = {
    { "help", "Display this list of commands", mon_help },
    { "kerninfo", "Display information about the kernel", mon_kerninfo },
    { "backtrace", "Display information about the kernel", mon_backtrace },
    { "showva2pa", "Display information about mapping virtual address to
    physical address", mon_showva2pa }
};
```

```
int mon_showva2pa(int argc, char **argv, struct Trapframe *tf);    // in line:
80 (monitor.c)
```

这里要求设置两种模式，一种是单地址查询，此时 `argc == 2`。发现 `inc/string.h` 有 `strtol` 函数可以用来将字符串转为数字

```
va=strtol(argv[1],&end,16);
pp = page_lookup(kern_pgdir,(void*)va,&pte);
if(!pp) {
    cprintf("VA: 0x%8x doesn't have a pa mapped\n");
}
else
    cprintf("VA: 0x%8x, PA: 0x%6x, pp_ref: %d, PTE_W: %d, PTE_U: %d\n",
        va, page2pa(pp), pp->pp_ref, ((*pte)&PTE_W)>>1, ((*pte)&PTE_U)>>2);
```

一种是范围查询，这里我要求范围参数的起点和重点必须对齐 `PGSIZE`

```
intptr_t va_pg_begin,va_pg_end;
//caculate the begin and end
va_pg_begin = strtol(argv[1],&end,16);
va_pg_end = strtol(argv[2],&end,16);
if( va_pg_begin % PGSIZE==0 && va_pg_end % PGSIZE==0 ){
    //output
    for(va=va_pg_begin; va<=va_pg_end; va+=PGSIZE){
```



```

    pp = page_lookup(kern_pgdir, (void *)va, &pte);
    if(!pp) {
        cprintf("VA: 0x%8x doesn't have a pa mapped\n");
        continue;
    }
    cprintf("VA: 0x%8x, PA: 0x%6x, pp_ref: %d, PTE_W: %d, PTE_U: %d\n",
           va, page2pa(pp), pp->pp_ref, ((*pte)&PTE_W)>>1,
           ((*pte)&PTE_U)>>2);
    }
}
else
    cprintf("arguments not aligned!\n");

```

运行结果:

蓝色部分为单地址查询

红色部分为范围地址查询

```

rivership@ubuntu: ~/share/share/OS/lab4
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> showva2pa 0xef7f0000
VA: 0xef7f0000, PA: 0x3cd000, pp_ref: 1, PTE_W: 1, PTE_U: 1
K> showva2pa 0xef800000
VA: 0xef800000 doesn't have a pa mapped
K> showva2pa 0xef7f0000 0xef800000
VA: 0xef7f0000, PA: 0x3cd000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f1000, PA: 0x3cc000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f2000, PA: 0x3cb000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f3000, PA: 0x3ca000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f4000, PA: 0x3c9000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f5000, PA: 0x3c8000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f6000, PA: 0x3c7000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f7000, PA: 0x3c6000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f8000, PA: 0x3c5000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7f9000, PA: 0x3c4000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7fa000, PA: 0x3c3000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7fb000, PA: 0x3c2000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7fc000, PA: 0x3c1000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7fd000, PA: 0x3c0000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7fe000, PA: 0x3bf000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef7ff000, PA: 0x3be000, pp_ref: 1, PTE_W: 1, PTE_U: 1
VA: 0xef800000 doesn't have a pa mapped
K>

```

问题回答

(1)

从 entry.s 中为 cr3 装载了页目录 entry_pgdir 时开始使用虚拟地址。

```

# Load the physical address of entry_pgdir into cr3. entry_pgdir
# is defined in entrypgdir.c.
movl    $(RELOC(entry_pgdir)), %eax
movl    %eax, %cr3
# Turn on paging.
movl    %cr0, %eax
orl    $(CR0_PE|CR0_PG|CR0_WP), %eax
movl    %eax, %cr0

```

(2)

```
kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
```

kern_pgdir 由 boot_alloc 申请来，且是第一次调用 boot_alloc，所以其位于 kernel bss 段之后的第一个 page 起始处，其虚拟地址为 0xf0118000，物理地址为 0xf0118000-KERNBASE == 0x00118000

```

137      memset(kern_pgdir, 0, PGSIZE);
gdb-peda$ p kern_pgdir
$2 = (pde_t *) 0xf0118000
gdb-peda$

```

之所以能够使用虚拟地址，是因为此前在 entry.S 中，预先为 cr3 装载了一个简单的页目录 entry_pgdir，定义在 kern/entrypgdir.c 中，这个页目录只有项 entry_pgdir[0] entry_pgdir[KERNBASE>>PDXSHIFT]，均指向同一个页表 entry_pgtable，这个页目录指定了如下的映射关系：

```

VA's [0, 4MB) to PA's [0, 4MB)
VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)

```

在 mem_init 装载 kern_pgdir 之前，我们都能访问位于上述映射区间中的虚拟地址。即我们可以访问的虚拟地址范围为 [0, 0x1000000) [0xF0000000, 0xF0100000)

(3)

位于 mem_init 中，1cr3(PADDR(kern_pgdir));

这句话将内核页表 kern_pgdir 装载到 cr3 寄存器中，之后可以通过该寄存器访问页表，即可以获得虚拟地址的映射关系

(4)

最大能有 2GB 物理内存

[UPAGES, UVPT) 之间的 4MB 空间映射到 pages 数组，而一个 struct PageInfo 是 8bytes，因此 pages 最多可以有 $2^{22-3}=2^{19}$ 项，共能表示 $2^{19} * PGSIZE = 2GB$ 的物理内存大小

(5)

JOS 采用二级页表，虚拟地址的 32bit 为：

```
|----10bit pgdir----|----10bit pgnum----|-----12bit offset-----|
```

(6)

- (7)**

Virtual memory map:		Permissions kernel/user	
4 Gig ----->	+-----+ RW/-- ~~~~~ : : : ~~~~~ RW/-- RW/-- at physical address 0 RW/-- RW/--		
KERNBASE, -----> KSTACKTOP	+-----+ Kernel Stack RW/ - - - - - kernel stack gap - - - - - +-----+ RW/-- KSTKSIZE	0xf0000000	
	 - - - - - : : : +-----+		
ULIM ----->	+-----+ 这一段的map被kern_pgdir指定 R-/R- PTSIZE 此时kern_pgdir不是作为页目录 而是作为一个页表	0xef800000	
UVPT ----->	+-----+ pages R-/R- PTSIZE	0xef400000	
UPAGES ----->	+-----+ 	0xef000000	

+-----+ 0x00000000