Lab2作业报告

515030910489 丁贞锴

**1 概述**

这个 lab 主要讲述的是 kernel virtual address 部分页表的建立，目前用户 virtual

address 没有建立。那么如何建立？这个 lab 的想法比较简单，之前的 lab1 中我们建立

了一个简单的页表，非常简单，就是将 virtual address-kernbase=physical address。这

是一个线性的映射，为何我们要这样做？由于我们现在已经处于 protected mode，我

们肯定是要用到页表的，所以在建立真正的页表之前，我们使用这个页表。有了这个页

表，那么我们就可以建立自己的页表了，建立好了之后，将 cr3 设置成新的页表的

physical address 就完成了真正的页表。

页表的结构？我们先 alloc 一个 pde，这个一级页表，实际上，我们的一级页表只

需要这一个 page 的 pde 就完全能够覆盖所有的物理空间。（virtual address 的分布，

// +--------10------+-------10-------+---------12----------+

// | Page Directory | Page Table | Offset within Page |

// | Index | Index | |

// +----------------+----------------+---------------------+

一个 page 为 4K，pde 就有 4K/4=1K 个 pte 的 entry，正好就是前 10 位），所以我们的

memory 的页表是一个一级页表，1024 个二级页表就可以了。由于我们是 page 对其的，

所以我们的页表地址低 12 位应该为 0，这就来了一个技巧，我们就使用低 12 位来设置

我们权限 bit ( good idea！！)。

页表的建立过程？在一级页表中（kern\_pgdir）根据 VA 前 10bit 找到二级页表的

entry，那么就存在以下情况？1）页表 entry 是 0，那么我们分配一个物理 page 来作为

二级页表，将他的地址给 entry 并设置权限；2）页表 entry 不为 0，但是 PET\_P 为 0，

代表不再内存中，操作如 1）；3）页表 entry 存在，但权限不够（由于我们是 kernel page，

所以不存在这个情况），返回 NULL；4）页表存在，而且权限可以，那么我们直接找到二

级页表就可以了。找到二级页表，通过 PTX（VA）+\*PDE 找到二级页表中的 entry，重

复上面的情况就可以了。

基本想法有了，下面介绍这个 lab 的详细情况。

Lab2相比lab1多了五个文件，memlayout.h pmap.c pmap.h kclock.h kclock.c

我们先来看看这五个文件讲了什么。

Memlayout.h: 这个头文件里面主要描述了虚拟内存的框架，定义了一些常量和page的数据结构。

pmap.c/pmap.h： 这是内存管理的主要文件，关于虚拟地址转换为物理地址，页表的建立，页的分配等相关函数都在这里面。

kclock.c/kclock.h：这个是更底层的实现，不过在本次lab里面，可以不考虑这两个文件。

**2 Part 1: 物理页管理**

这部分主要就是做一些初始化的工作和物理页的分配。

Exercise1要求我们实现五个函数boot\_alloc() mem\_init() page\_init() page\_alloc() page\_free()。

我们先看一下boot\_alloc()函数讲了什么。

static void \*

boot\_alloc(uint32\_t n)

{

if(n<0){

return NULL;

}

static char \*nextfree; // virtual address of next byte of free memory

// static variable will not be deleted when the fuction is gone, it will

// stay as time goes

char \*result;

// Initialize nextfree if this is the first time.

// 'end' is a magic symbol automatically generated by the linker,

// which points to the end of the kernel's bss segment:

// the first virtual address that the linker did \*not\* assign

// to any kernel code or global variables.

// end points the end of kernel in memory

if (!nextfree) {

extern char end[];

nextfree = ROUNDUP((char \*) end, PGSIZE);

}

// 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.

if(n==0) return nextfree;

result = nextfree; // start address

nextfree += n;

nextfree = KADDR(PADDR(nextfree));

// means noting ,just nextfree = nextfree -KNERBASE + KNERBASE

// just to check whether or not ,nextfree out of memory , just check if panic

nextfree = ROUNDUP(nextfree, PGSIZE);

nextfree = KADDR(PADDR(nextfree));

return result;

}

boot\_alloc( n ) 就是分配足够容纳n个byte的页，返回空闲页的起始地址。

注意boot\_alloc()函数里面有一个静态变量nextfree，这个是个全局变量。

nextfree指的是下一个没有被分配的页，注意这里面有一个ROUNDUP（nextfree, PGSIZE）函数，这个函数的作用是把nextfree变成PGSIZE的倍数。

我们再看下一个函数mem\_init()。

void

mem\_init(void)

{

uint32\_t cr0;

size\_t n;

// Find out how much memory the machine has (npages & npages\_basemem).

i386\_detect\_memory();

// Remove this line when you're ready to test this function.

//panic("mem\_init: This function is not finished\n");

//////////////////////////////////////////////////////////////////////

// create initial page directory.

kern\_pgdir = (pde\_t \*) boot\_alloc(PGSIZE);

memset(kern\_pgdir, 0, PGSIZE);

//////////////////////////////////////////////////////////////////////

// Recursively insert PD in itself as a page table, to form

// a virtual page table at virtual address UVPT.

// (For now, you don't have understand the greater purpose of the

// following two lines.)

// Permissions: kernel R, user R

kern\_pgdir[PDX(UVPT)] = PADDR(kern\_pgdir) | PTE\_U | PTE\_P;

//////////////////////////////////////////////////////////////////////

// Allocate an array of npages 'struct Page's and store it in 'pages'.

// The kernel uses this array to keep track of physical pages: for

// each physical page, there is a corresponding struct Page in this

// array. 'npages' is the number of physical pages in memory.

// Your code goes here:

pages = boot\_alloc(npages \* sizeof(struct Page));

memset(pages, 0, PGSIZE);

//////////////////////////////////////////////////////////////////////

// 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\_region

// or page\_insert

page\_init();

check\_page\_free\_list(1);

check\_page\_alloc();

check\_page();

}

这里就是初始化了页表，给页表分配了内存，也给pages（里面存的是物理页的信息pp\_link，pp\_ref）分配了内存。

我们再看page\_init()。

void

page\_init(void)

{

// The example code here marks all physical pages as free.

// However this is not truly the case. What memory is free?

// 1) Mark physical page 0 as in use.

// This way we preserve the real-mode IDT and BIOS structures

// in case we ever need them. (Currently we don't, but...)

// 2) The rest of base memory, [PGSIZE, npages\_basemem \* PGSIZE)

// is free.

// 3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must

// never be allocated.

// 4) Then extended memory [EXTPHYSMEM, ...).

// Some of it is in use, some is free. Where is the kernel

// in physical memory? Which pages are already in use for

// page tables and other data structures?

//

// Change the code to reflect this.

// NB: DO NOT actually touch the physical memory corresponding to

// free pages!

pages[0].pp\_ref = 1;

pages[0].pp\_link = NULL;

size\_t i;

for (i = 1; i < npages\_basemem; i++) {

pages[i].pp\_ref = 0;

pages[i].pp\_link = page\_free\_list;

page\_free\_list = &pages[i];

}

for (i = PGNUM(PADDR(boot\_alloc(0))); i < npages; i++) {

pages[i].pp\_ref = 0;

pages[i].pp\_link = page\_free\_list;

page\_free\_list = &pages[i];

}

chunk\_list = NULL;

}

page\_init()函数就是把刚刚分配的页初始化一下，一共npages个页，但是要注意这中间有一部分页是IO的，不能碰，不过我们可以用boot\_alloc(0)来获取nextfree。

我们再看page\_alloc()。

struct Page \*

page\_alloc(int alloc\_flags)

{

// Fill this function in

if(!page\_free\_list) return NULL;

struct Page \* returnpage = page\_free\_list;

page\_free\_list = page\_free\_list->pp\_link;

if(alloc\_flags & ALLOC\_ZERO){

memset(page2kva(returnpage), 0, PGSIZE);

}

return returnpage;

}

page\_alloc()函数就是取当前的nextfree并且返回，并且更新nextfree。

我们再看page\_free()。

void

page\_free(struct Page \*pp)

{

// Fill this function in

if(pp->pp\_ref==0){

pp->pp\_ref = 0;

pp->pp\_link = page\_free\_list;

page\_free\_list = pp;

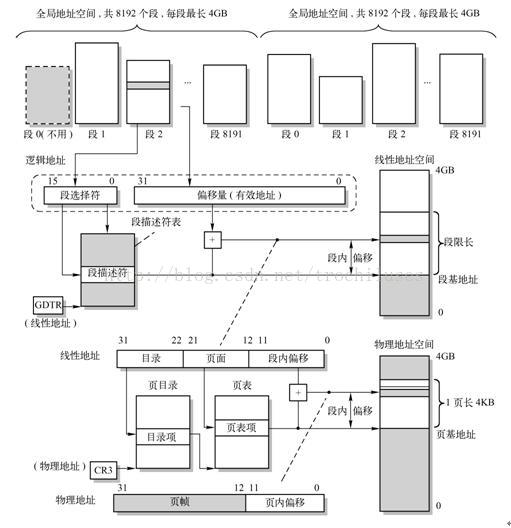
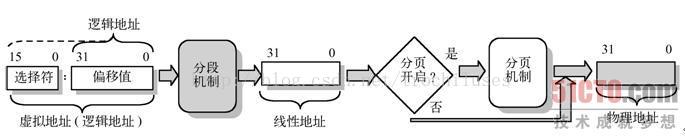
}

}

page\_free()就是把当前页的pp\_ref置为0，然后把当前页加到nextfree里面。Nextfree其实是个链表结构。

**3 Part 2 虚拟内存**

这里首先讲述了虚拟地址，线性地址跟物理地址的关系，我找了下面两张图来更具像地解释这三者之间的关系。



在这部分，我们主要是实现虚拟地址跟物理地址之间的映射。我们一共要实现五个函数

pgdir\_walk() boot\_map\_region() page\_lookup() page\_remove() page\_insert().

我们先看pgdir\_walk()。

pte\_t \*

pgdir\_walk(pde\_t \*pgdir, const void \*va, int create)

{

// Fill this function in

/\*char \*p1 = va>>22;

char \*PDE = pgdir + p1;

char \*p2 = (va<<10)>>22;

char \*PTE = (\*PDE) + p2;\*/

pde\_t \*PDE = PDX(va) + pgdir;

pte\_t \*PTE;

if( (\*PDE) & PTE\_P){

pte\_t \*PPN = (pte\_t \*)KADDR(PTE\_ADDR(\*PDE));

PTE = PPN + PTX(va);

}

else if(create == 0) return NULL;

else{

struct Page \*new\_page = page\_alloc(ALLOC\_ZERO);

if( new\_page == NULL){

return NULL;

}

new\_page->pp\_ref++;

\*PDE = page2pa(new\_page) | PTE\_P | PTE\_W | PTE\_U;

pte\_t \*PPN = (pte\_t \*)KADDR(PTE\_ADDR(\*PDE));

PTE = PPN + PTX(va);

}

return PTE;

}

page\_walk()函数的作用就是把虚拟地址转换为物理地址。具体就是先根据虚拟地址的高10位在pgdir里面找到page\_table,然后再根据中间的10位在page\_table里面找到对应的PTE。

我们再看看boot\_map\_region()。

static void

boot\_map\_region(pde\_t \*pgdir, uintptr\_t va, size\_t size, physaddr\_t pa, int perm)

{

int i=0;

pte\_t \*PTE;

for(;i<size ;i+=PGSIZE){

PTE = pgdir\_walk(pgdir, (void \*)va, 1);

\*PTE = pa | perm | PTE\_P;

va += PGSIZE;

pa += PGSIZE;

}

}

这个函数就是把pa映射到va的位置，一共映射size个byte。

我们再看看page\_lookup()。

struct Page \*

page\_lookup(pde\_t \*pgdir, void \*va, pte\_t \*\*pte\_store)

{

// Fill this function in

pte\_t \*PTE = pgdir\_walk(pgdir, va, 0);

if((PTE==NULL) || ((\*PTE)&PTE\_P) ==0) return NULL;

if(pte\_store!=0){

\*pte\_store = PTE;

}

return pa2page(PTE\_ADDR(\*PTE));

}

这个函数就是根据va找到对应的物理页。

我们再看看page\_remove()

void

page\_remove(pde\_t \*pgdir, void \*va)

{

pte\_t \*\*PTE = &pgdir ;

struct Page \*lookpage = page\_lookup(pgdir, va, PTE);

if(lookpage){

page\_decref(lookpage);

\*\*PTE = 0;

tlb\_invalidate(pgdir,va);

}

}

这个函数就是把va对应的物理页remove，其实就是pp\_ref减一,并且当pp\_ref小于1的时候，释放这个页。

我们再看看page\_insert()。

int

page\_insert(pde\_t \*pgdir, struct Page \*pp, void \*va, int perm)

{

pte\_t \*PTE = pgdir\_walk(pgdir, va, 1);

if(PTE==NULL) return -E\_NO\_MEM;

/\*if((\*PTE)&PTE\_P){

if(PTE\_ADDR(\*PTE) == page2pa(pp)){

tlb\_invalidate(pgdir,va);

pp->pp\_ref--;

}

else{

page\_remove(pgdir, va);

}

//page\_remove(pgdir, va);

}

\*PTE = page2pa(pp) | perm | PTE\_P;

pp->pp\_ref++;

return 0;\*/

if(((\*PTE)&PTE\_P) == 0){

;

}

else if(PTE\_ADDR(\*PTE) == page2pa(pp)){

tlb\_invalidate(pgdir,va);

pp->pp\_ref--;

}

else{

page\_remove(pgdir, va);

}

\*PTE = page2pa(pp) | perm | PTE\_P;

pp->pp\_ref++;

return 0;

}

这个函数就是把物理页pp置为va对应的物理页。我一开始想的是，删掉之前va对应的物理页，然后把pp插进去。后来我看注释，要考虑之前va对应的物理页其实就是pp的情况，在这种情况下，要删除之前TLB里面的条目。

**4 Part 3 内核地址空间**

这部分我们要开始做内核地址空间的映射，把相应的物理页映射到对应的虚拟地址空间。

我们看最开始的函数mem\_init()

void

mem\_init(void)

{

//////////////////////////////////////////////////////////////////////

// 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\_region(kern\_pgdir, UPAGES, PTSIZE, PADDR(pages), 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\_region(kern\_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE, PADDR(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:

lcr4(rcr4() | CR4\_PSE);

boot\_map\_region\_large(kern\_pgdir, KERNBASE, 0x10000000 , 0, PTE\_W);

//boot\_map\_region(kern\_pgdir, KERNBASE, 0x10000000 , 0, PTE\_W);

// Check that the initial page directory has been set up correctly.

check\_kern\_pgdir();

// Switch from the minimal entry page directory to the full kern\_pgdir

// page table we just created. Our instruction pointer should be

// somewhere between KERNBASE and KERNBASE+4MB right now, which is

// mapped the same way by both page tables.

//

// If the machine reboots at this point, you've probably set up your

// kern\_pgdir wrong.

lcr3(PADDR(kern\_pgdir));

check\_page\_free\_list(0);

// entry.S set the really important flags in cr0 (including enabling

// paging). Here we configure the rest of the flags that we care about.

cr0 = rcr0();

cr0 |= CR0\_PE|CR0\_PG|CR0\_AM|CR0\_WP|CR0\_NE|CR0\_MP;

cr0 &= ~(CR0\_TS|CR0\_EM);

lcr0(cr0);

// Some more checks, only possible after kern\_pgdir is installed.

check\_page\_installed\_pgdir();

}

我们要加的其实就是三个boot\_map\_region。boot\_map\_region(kern\_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE\_U);映射页表。boot\_map\_region(kern\_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE\_W);映射内核栈。boot\_map\_region\_large(kern\_pgdir, KERNBASE, 0x10000000 , 0, PTE\_W);映射256MB的物理内存。

**5 总结**

这个lab主要实现内存管理相关的工作，具体就是实现虚拟内存的layout，物理内存的layout，已经虚拟内存与物理内存之间的映射。我在这里面也踩了很多坑，比如对物理内存里面的各个部分里面存了什么理解有误，尤其是pages那一块，后来反复看代码才知道了。地址间的映射要注意地址有没有越界，在页表那里，我对于也表里面存的是什么产生了误解。我一开始以为二级页表里面存的PTE就直接是加了offset的物理地址，其实没有，只是PPN。