# Lab2

## Part 1：Physical Page Management

Exercise 1. 在文件 kern/pmap.c 中，你必须要完成以下几个子函数的代码

boot\_alloc();

mem\_init();

page\_init();

page\_alloc();

page\_free();

check\_page\_free\_list()和check\_page\_alloc()两个函数将会检测你写的页分配器代码是否正确。

答：

jos把整个物理内存空间划分成三个部分：

一个是从0x00000~0xA0000，这部分也叫basemem，是可用的。

紧接着是0xA0000~0x100000，这部分叫做IO hole，是不可用的，主要被用来分配给外部设备了。

再紧接着就是0x100000~0x，这部分叫做extmem，是可用的，这是最重要的内存区域。

这个子函数中包括三个变量，其中npages记录整个内存的页数，npages\_basemem记录basemem的页数，npages\_extmem记录extmem的页数。

执行完这个函数，下一条指令为：

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

memset(kern\_pgdir, 0, PGSIZE);

其中kern\_pgdir是一个指针，pde\_t \*kern\_pgdir，它是指向操作系统的页目录表的指针，操作系统之后工作在虚拟内存模式下时，就需要这个页目录表进行地址转换。我们为这个页目录表分配的内存大小空间为PGSIZE，即一个页的大小。并且首先把这部分内存清0。

boot\_alloc() 只能被kernel调用，分配内存

它只是被用来暂时当做页分配器，之后我们使用的真实页分配器是page\_alloc()函数。而这个函数的核心思想就是维护一个静态变量nextfree，里面存放着下一个可以使用的空闲内存空间的虚拟地址，所以每次当我们想要分配n个字节的内存时，我们都需要修改这个变量的值。

再看下一条命令：

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

这一条指令就是再为页目录表添加第一个页目录表项。通过查看memlayout.h文件，我们可以看到，UVPT的定义是一段虚拟地址的起始地址，0xef400000，从这个虚拟地址开始，存放的就是这个操作系统的页表kern\_pgdir，所以我们必须把它和页表kern\_pgdir的物理地址映射起来，PADDR(kern\_pgdir)就是在计算kern\_pgdir所对应的真实物理地址。

下一条命令需要我们去补充，这条命令要完成的功能是分配一块内存，用来存放一个struct PageInfo的数组，数组中的每一个PageInfo代表内存当中的一页。操作系统内核就是通过这个数组来追踪所有内存页的使用情况的。我写的代码如下：

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

memset(pages, 0, npages \* sizeof(struct PageInfo));

page\_init()

1. 初始化pages数组

2.初始化pages\_free\_list链表，这个链表中存放着所有空闲页的信息

check\_page\_free\_list(1)执行完成，我们将进入下一个检查函数check\_page\_alloc()，这个函数的功能是检查page\_alloc()，page\_free()两个子函数是否能够正确运行。所以我们首先要实现这两个子函数。

page\_alloc()

分配一个物理页。而函数的返回值就是这个物理页所对应的PageInfo结构体。

1. 从free\_page\_list中取出一个空闲页的PageInfo结构体

2. 修改free\_page\_list相关信息，比如修改链表表头

3. 修改取出的空闲页的PageInfo结构体信息，初始化该页的内存

page\_free()

这个方法的功能就是把一个页的PageInfo结构体再返回给page\_free\_list空闲页链表，代表回收了这个页。

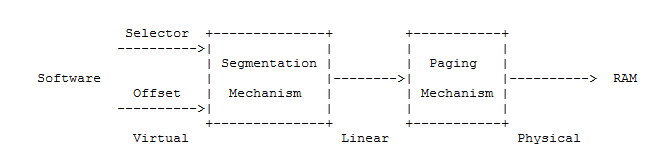
1. 修改被回收的页的PageInfo结构体的相应信息。

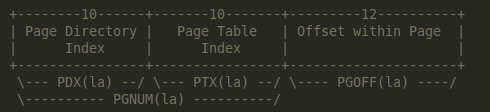
2. 把该结构体插入回page\_free\_list空闲页链表。

## Part 2: Virtual Memory

git log 历史提交

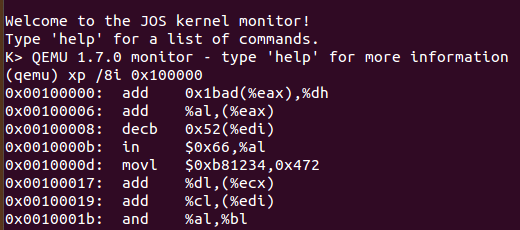
git reset --hard HEAD^ 返回至最新提交版本





注意本实验用的是二级页表结构，先查目录表page directory，再查页表page table

Exercise 3:



一旦进入保护模式，我们就不能直接使用线性地址或者物理地址了。所有代码中的地址引用都是虚拟地址的形式，然后被MMU系统所转换，所以C语言中的指针其实都是虚拟地址。

JOS内核通常需要把地址按照以一种模糊的值或者整数值的形式来操纵，而不是直接解析引用，比如物理内存分配器。有时使用虚拟地址，有时使用物理地址。为了能够帮助我们记录代码，JOS源文件中的地址被区分为两种情况：

uintptr\_t -- 表示虚拟地址

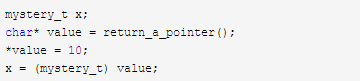
physaddr\_t -- 表示物理地址

这两种类型其实都是32位的整型数(uint32\_t)，所以如果你把一个类型的变量的值赋给另一个类型变量，编译器不会报错。但是由于他们都是整型数，所以如果你打算解引用(deference)他们，编译器会报错。

JOS内核可以先对uintptr\_t类型的值进行强制类型转换，然后再解析引用。但是对于physaddr\_t的值，我们不能这么做，因为内核是需要MMU（内存管理单元）来首先对你输入的地址进行转化的，如果你对physaddr\_t进行强制类型装换再解引用，最终你得到的你要访问的地址，可能不是你要找的真实物理地址。

问题：

假设下述JOS内核代码是正确的，那么变量x应该是uintptr\_t类型呢，还是physaddr\_t呢？



答：

由于这里使用了 \* 操作符解析地址，所以变量x应该是uintptr\_t类型。

### Permission and Fault Isolation

由于内核和用户进程只能访问各自的地址空间，所以我们必须在x86页表中使用访问权限位(Permission Bits)来使用户进程的代码只能访问用户地址空间，而不是内核地址空间。否则用户代码中的一些错误可能会覆写内核中的数据，最终导致内核的崩溃。

处在用户地址空间中的代码不能访问高于ULIM的地址空间，但是内核可以读写这部分空间。而内核和用户对于地址范围[UTOP, ULIM]有着相同的访问权限，那就是可以读取但是不可以写入。这一个部分的地址空间通常被用于把一些只读的内核数据结构（kernel data structures）暴露给用户地址空间的代码。在UTOP之下的地址范围是给用户进程使用的，用户进程可以访问，修改这部分地址空间的内容。

Exercise 4. 完成kern/pmap.c中的下面几个子函数的编码

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

check\_page()子函数将会被用来检查你所编写的这些程序是否正确。

答：

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

**给定一个页目录表指针 pgdir ，该函数应该返回线性地址va所对应的页表项指针。**

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

把虚拟地址空间范围[va, va+size)映射到物理空间[pa, pa+size)的映射关系加入到页表pgdir中。这个函数主要的目的是为了设置虚拟地址UTOP之上的地址范围，这一部分的地址映射是静态的，在操作系统的运行过程中不会改变，所以这个页的PageInfo结构体中的pp\_ref域的值不会发生改变。

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

把一个物理内存中页pp与虚拟地址va建立映射关系

这里要注意，pp->pp\_ref++这条语句，一定要放在page\_remove之前，这是为了处理一种特殊情况：pp已经映射到va上了。如过位置颠倒且pp已经映射到va，则在pp->pp\_ref -1之后就为0，直接free该page。

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

返回虚拟地址va所映射的物理页的PageInfo结构体的指针，如果pte\_store参数不为0，则把这个物理页的页表项地址存放在pte\_store中。

void page\_remove(pde\_t \*pgdir, void \*va)，功能就是把虚拟地址va和物理页的映射关系删除。注释里面还提示了要注意的几个细节：

1. pp\_ref值要减一

2. 如果pp\_ref减为0，要把这个页回收

3. 这个页对应的页表项应该被置0

## Part 3 Kernel Address Space

Exercise 5

继续完善mem\_init()函数，你的程序现在必须能够通过check\_kern\_pgdir()和check\_page\_installed\_pgdir()函数的检测。

答：

首先我们要映射的范围是把pages数组映射到线性地址UPAGES，大小为一个PTSIZE。

所以我们添加的代码是：

boot\_map\_region(kern\_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE\_U);

其中perm变量之所以设置为PTE\_U，是因为这部分空间是kernel space和user space中的代码都能访问的，所以要设置PTE\_U。

然后映射内核的堆栈区域，把由bootstack变量所标记的物理地址范围映射给内核的堆栈。内核堆栈的虚拟地址范围是[KSTACKTOP-PTSIZE, KSTACKTOP)，不过要把这个范围划分成两部分：

\* [KSTACKTOP-KSTKSIZE, KSTACKTOP) 这部分映射关系加入的页表中。

\* [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) 这部分不进行映射。

对这部分地址的访问权限是，kernel space 可以读写，user space 无权访问，所以代码如下：

boot\_map\_region(kern\_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE\_W)

最后映射整个操作系统内核，虚拟地址范围是[KERNBASE, 2^32]，物理地址范围是[0，2^32 - KERNBASE]。

访问权限是，kernel space 可以读写，user space 无权访问，所以代码如下：

boot\_map\_region(kern\_pgdir, KERNBASE, 0xffffffff - KERNBASE, 0, PTE\_W);

Question:

1. 到目前为止页目录表中已经包含多少有效页目录项？他们都映射到哪里？

UVPT 0xef400000 PDX(VPT) = 3BD 指向的是kern\_pgdir

UPAGES 0xef000000 PDX(UPAGES) = 3BC 指向的是pages数组

KSTACKTOP – KSTKSIZE = 0xefc00000 PDX = 3BF 指向的是bootstack

KERNBASE =>PDX: 3C0 到 0xffffffff – KERNBASE => 3FF 指向的是kernel

1. 如果我们把kernel和user environment放在一个相同的地址空间中。为什么用户程序不能读取，写入内核的内存空间？用什么机制保护内核的地址范围。

用户程序不能去随意修改内核中的代码，数据，否则可能会破坏内核，造成程序崩溃。

正常的操作系统通常采用两个部件来完成对内核地址的保护，一个是通过段机制来实现的，但是JOS中的分段功能并没有实现。二就是通过分页机制来实现，通过把页表项中的 Supervisor/User位置0，那么用户态的代码就不能访问内存中的这个页。这个操作系统的可以支持的最大数量的物理内存是多大？

1. 这个操作系统的可以支持的最大数量的物理内存是多大？

由于这个操作系统利用一个大小为4MB的空间UPAGES来存放所有的页的PageInfo结构体信息，每个结构体的大小为8B，所以一共可以存放512K个PageInfo结构体，所以一共可以出现512K个物理页，每个物理页大小为4KB，自然总的物理内存占2GB。

1. 如果现在的物理内存页达到最大个数，那么管理这些内存所需要的额外空间开销有多少？

首先需要存放所有的PageInfo，需要4MB，

页表项总数为512K，故页表项累计大小为2MB

需要2MB/4K = 512个目录，故2KB

所以总的开销就是6MB + 4KB。

1. 回顾entry.S文件中，当分页机制开启时，寄存器EIP的值仍旧是一个小的值。在哪个位置代码才开始运行在高于KERNBASE的虚拟地址空间中的？当程序位于开启分页之后到运行在KERNBASE之上这之间的时候，EIP的值是小的值，怎么保证可以把这个值转换为真实物理地址的？

在entry.S文件中有一个指令 jmp \*%eax，这个指令要完成跳转，就会重新设置EIP的值，把它设置为寄存器eax中的值，而这个值是大于KERNBASE的，所以就完成了EIP从小的值到大于KERNBASE的值的转换。

在entry\_pgdir这个页表中，也把虚拟地址空间[0, 4MB)映射到物理地址空间[0, 4MB)上，所以当访问位于[0, 4MB)之间的虚拟地址时，可以把它们转换为物理地址。