实验Lab2——内存管理

在本次实验中，先在本地新建lab2分支，再将在lab1分支中所做的更改合并到lab2分支中。观察到实验2包含以下新的源文件：

inc/memlayout.h

kern/pmap.c

kern/pmap.h

kern/kclock.c

kern/kclock.h

memlayout.h描述了虚拟地址空间的结构，我们需要通过修改pmap.c文件来实现这个结构。memlayout.h和pmap.h文件定义了一个PageInfo结构，利用这个结构可以记录有哪些物理页是空闲的。kclock.c和kclock.h文件中操作的是用电池充电的时钟，以及CMOS RAM设备。在这个设备中记录着PC机拥有的物理内存的数量。

1.物理页面管理

内存管理部分有两个组件。第一个组件是内核的物理内存分配器，因此内核可

以在需要时分配内存并用完后释放它。自己编写的分配器将以4096个字节（4k）为单位进行操作，这个单位被称为一个页(page)。需要维护一个数据结构，用于记录哪些物理页面是空闲的，哪些物理页面已经被分配，以及每个分配的页面正在被几个进程所共享。第二个组件是虚拟内存，将在下一部分进行深入分析。

**练习1.** 在文件kern/pmap.c中，必须实现以下函数的代码（可能按照给定的顺

序）。

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。

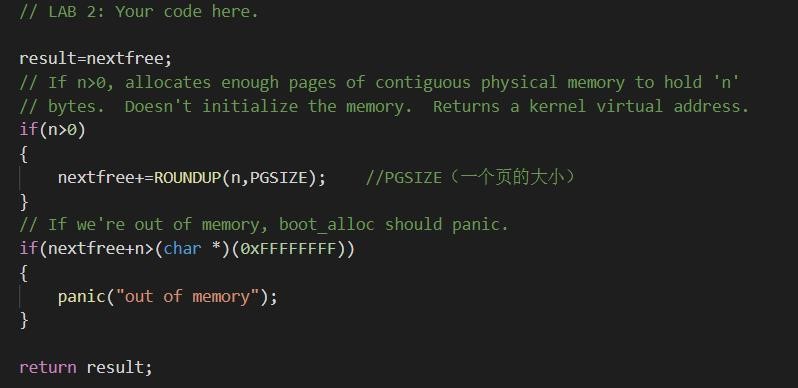
答：该实验利用一个 PageInfo 结构体组成的链表记录哪些页面空闲，每个结构体对应一个物理页。因为页表的实现需要分配物理内存来存储页表，在虚拟内存的实现之前，我们需要先编写物理页面分配器。

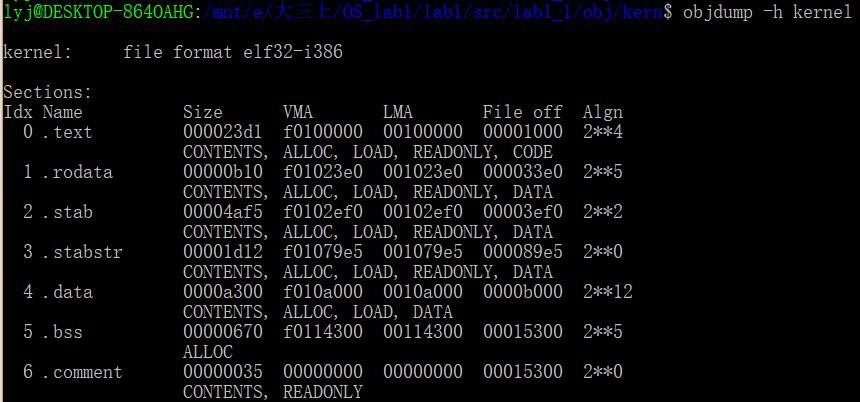
（1）打开pmap.c文件，找到第一个需要补全的函数 boot\_alloc()。根据注释中

的说明，理解到该函数负责开辟出n个Byte的空闲空间，并返回这段空闲空间的首地址。boot\_alloc()函数只是被用来暂时当做页分配器，之后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，nextfree指向下一块可用空间的起始地址，我们只需将nextfree

初始化为end的4K字节对齐，然后加上我们要分配的空间大小，并返回初始时的nextfree即可。

根据注释内容补全代码；  nextfree存储着下一块可用空闲内存的虚拟地址，被初始化赋值为当前end， end指向内核bss段的末尾。bss 段已经是内核的最后一段。end以上均为可用地址，因此，end 指向的是第一个未使用的虚拟内存地址。使用ROUNDUP宏定义是因为内存区块是对齐的，所以每块都是固定的大小。



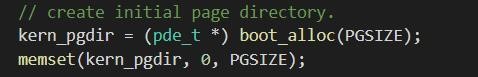
boot\_alloc()函数传入参数n,也就是待分配的内存大小。若n> 0，分配足够的连续物理内存页来保存n字节。返回更新之后的netfree虚拟地址。如果我们内存不足以分配n字节大小的空间，boot\_alloc就会输出错误提示。

（2）继续阅读代码，找到mem\_init()函数,在内核刚开始运行时就会调用这个子函数，对整个操作系统的内存管理系统进行一些初始化的设置，比如设定页表等等操作。

首先这个函数调用 i386\_detect\_memory 子函数，这个子函数的功能就是检测现在系统中有多少可用的内存空间。



执行完这个函数，下一条指令为：



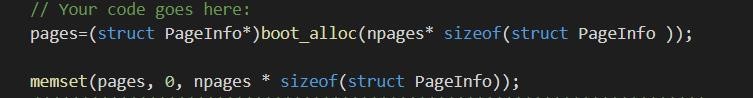
我们为这个页目录表分配的内存大小空间为PGSIZE，即一个页的大小，并且这个页就是紧跟着操作系统内核之后。把这部分内存清0。

下一条指令是递归地将PD本身作为页表插入，以在虚拟地址UVPT上形成虚拟页表。



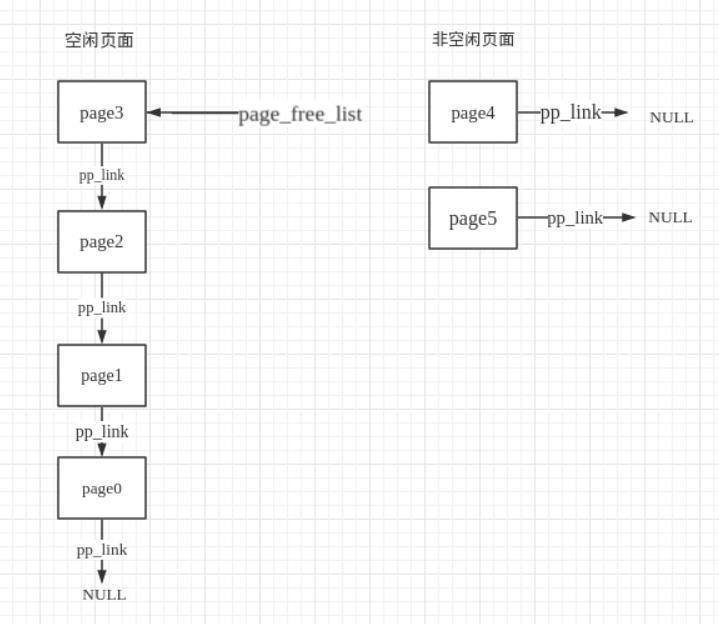
下面就是我们需要补全的代码了，根据注释，这部分代码分配一块内存，用来存放一个struct PageInfo的数组，数组中的每一个PageInfo代表内存当中的一页。初始化所有的struct PageInfo 为 0。首先确定PageInfo的大小，然后用boot\_alloc分配内存，接着用memset初始化。操作系统内核就是通过这个数组来追踪所有内存页的使用情况的。

代码如下：

 使用总物理内存/4k得到所需要的Page数量，赋值给npages，换句话说 npages代表所需Page结构体的数量。

需要注意的是分配内存用的是 boot\_alloc。这是一个仅用于 JOS 设置自身虚拟内存系统时使用的物理内存分配器，仅用mem\_init 函数。当初始化页面以及空闲内存列表后，不再使boot\_alloc，而使用 page\_alloc。

（3）继续阅读mem\_init（）函数，我们观察在补全的代码后，函数调用了page\_init()函数，该函数初始化pages数组 以及pages\_free\_list链表（管理空闲内存的链表）。



整个函数是由一个for循环构成，它会遍历所有内存页所对应npages

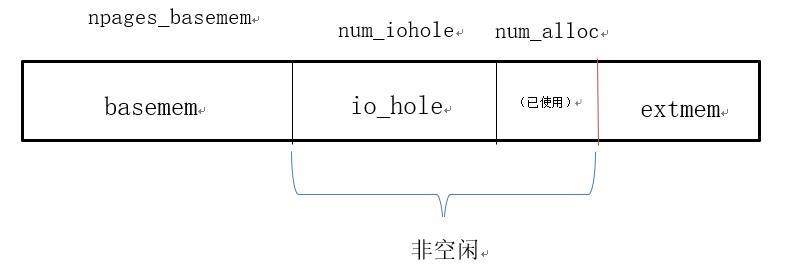
数组中的PageInfo结构体，并且根据这个页当前的状态来修改这个结构体的状态。原始代码将pages中的物理页都标志成free，并且把这些页加入到了pages\_free\_list链表中。但是阅读注释信息我们知道有三部分内存已经被占用，需要将它们的pp\_ref属性设置为1，分别是第0页，io\_hole部分,还有在 extmem区域还有一部分已经被占用。

这里我们回忆一下JOS对整个物理内存空间划分

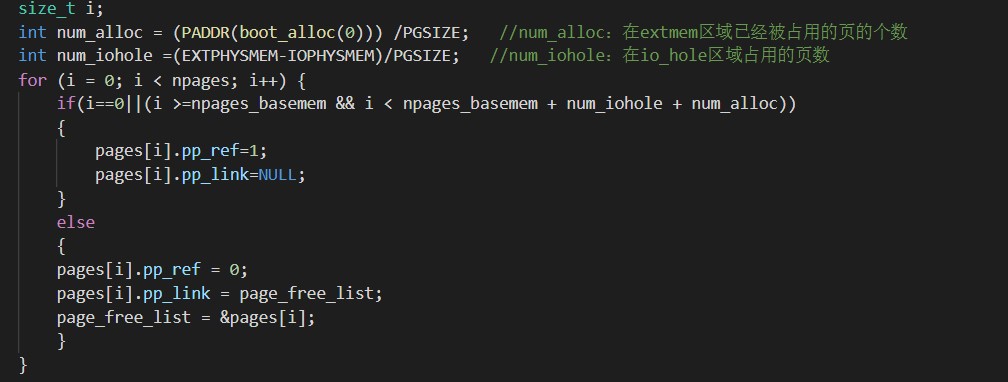
一个是从0x00000~0xA0000，这部分也叫basemem，是可用的。

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，部分可用的，这是最重要的内存区域。



添加代码如下：



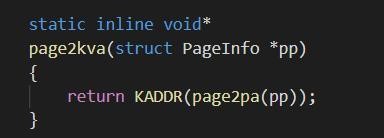
1. 补全page\_init()初始化函数之后，继续阅读mem\_init()函数

 首先找到page\_alloc()函数进行补全，通过注释我们可以知道这个函数的

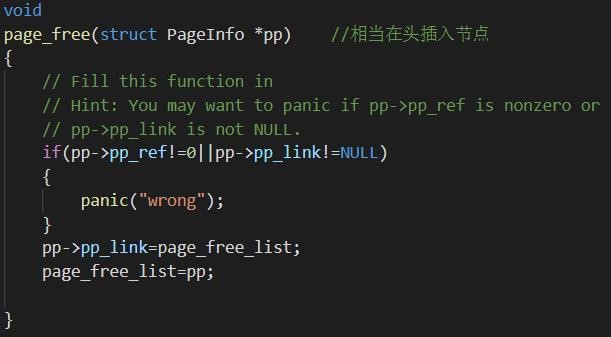
功能就是分配一个物理页。返回值就是这个物理页所对应的PageInfo结构体。分配一个物理页也就是从空闲链表中删除节点，利用数据结构所学知识可以很快补全该部分代码。

 注释中提示If(alloc\_flags & ALLOC\_ZERO)，直接分配一个以0赋值的物理页，并且提示page2kva和memset函数。

在pmap.h中找到page2kva（）的定义，page2kva 函数的作用就是通过物理页获取其内核虚拟地址：

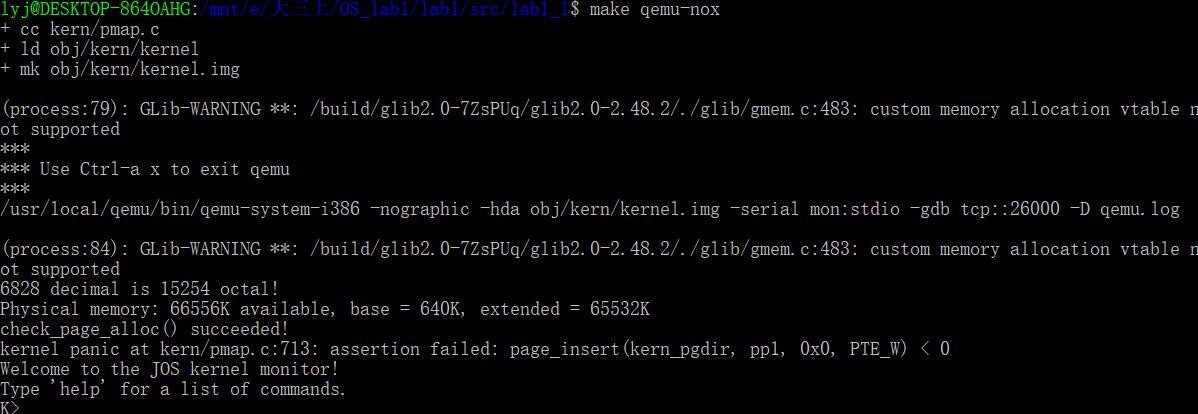


1. 下面补全 page\_free()函数，这个函数是回收页，只有当 pp\_ref 为 0 时，才回收。需要把这个页的PageInfo结构体再返回给page\_free\_list空闲页链表，想到与插入节点。代码如下：



完成五个函数的补全之后，运行qemu测试，check\_page\_alloc() succeeded!

说明补全正确，结果如下图：



虚拟内存

内存管理的第二个组件是虚拟内存。首先介绍逻辑地址，线性地址，物理地址的概念与三者之间的关系。

虚拟地址，线性地址和物理地址

逻辑地址（Logical Address） 是指由程序产生的与段相关的偏移地址部分。例如，你在进行C语言指针编程中，可以读取指针变量本身值(&操作)，实际上这个值就是逻辑地址，它是相对于你当前进程数据段的地址，不和绝对物理地址相干。只有在Intel实模式下，逻辑地址才和物理地址相等（因为实模式没有分段或分页机制,cpu不进行自动地址转换）；逻辑地址是在Intel 保护模式下程序执行代码段限长内的偏移地址（假定代码段、数据段如果完全一样）。应用程序员仅需与逻辑地址打交道，应用程序员虽然自己可以直接操作内存，那也只能在操作系统给你分配的内存段操作。

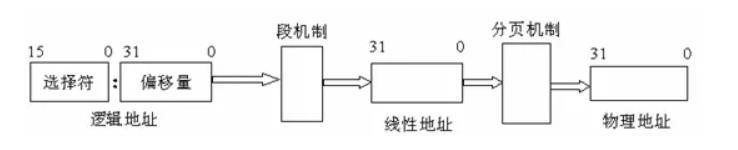
线性地址（Linear Address） 是逻辑地址到物理地址变换之间的中间层。程序代码会产生逻辑地址，或者说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址可以再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。Intel

80386的线性地址空间容量为4G（2的32次方即32根地址总线寻址）。

物理地址（Physical Address） 是指出现在CPU外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。如果启用了分页机制，那么线性地址会使用页目录和页表中的项变换成物理地址。如果没有启用分页机制，那么线性地址就直接成为物理地址了。

虚拟地址到物理地址的转化方法一般来说有分段、分页两种。MMU负责从虚拟地址到物理地址的转化。逻辑地址是段标识+段内偏移量的形式，MMU通过查询段表，可以把逻辑地址转化为线性地址。如果cpu没有开启分页功能，那么线性地址就是物理地址；如果cpu开启了分页功能，MMU还需要查询页表来将线性地址转化为物理地址：

逻辑地址 ----（段表）---> 线性地址 — （页表）—> 物理地址



逻辑地址与物理地址的“差距”是0xf0000000，是由于虚拟地址->线性地址->物理地址映射正好差这个值。要将该区域中的虚拟地址转换为物理地址，内核部分可以简单地减去0xf0000000。可以用PADDR(va)来完成这个减法。

不同的逻辑地址可以映射到同一个线性地址上；不同的线性地址也可以映

射到同一个物理地址上；所以是多对一的关系。另外，同一个线性地址，在发生换页以后，也可能被重新装载到另外一个物理地址上。所以这种多对一的映射关系也会随时间发生变化。

虚拟内存（Virtual Memory） 是指计算机呈现出要比实际拥有的内存大得多的内存量。因此它允许程序员编制并运行比实际系统拥有的内存大得多的程序。这使得许多大型项目也能够在具有有限内存资源的系统上实现。内存分配其实就是虚拟内存地址映射物理内存的过程，内存回收则是解除映射关系的过程。

虚拟内存中，0-3G是用户控制，3-4G是内核空间。用户层不能直接访问内核层，可以通过Unix/Linux的系统函数访问内核层。

分页机制与分段机制

练习2.请参阅“ Intel 80386参考手册”的第5章和第6章 。仔细阅读有关页转换和基于页的保护部分（5.2和6.4）。我们建议你同时浏览有关分段的部分; 虽然JOS使用的分页机制实现的虚拟内存和保护，但是段转换和段保护功能无法在x86上禁用，因此你需要对其有基本的了解。

答：

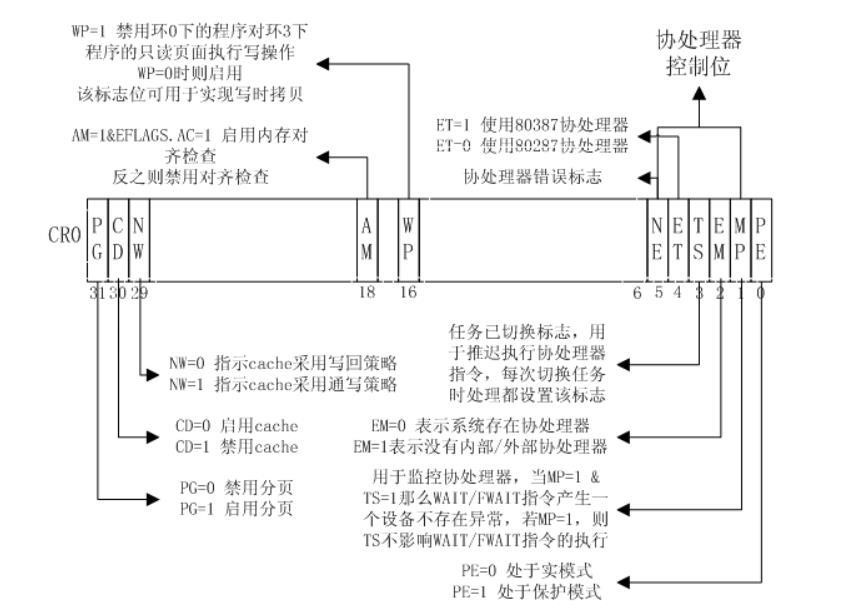
分页内存管理方案

分页的最大作用就在于：使得进程的物理地址空间可以是非连续的。物理

内存被划分为一小块一小块，每块被称为帧(Frame)。分配内存时，帧是分配时

的最小单位，最少也要给一帧。在逻辑内存中，与帧对应的概念就是页(Page)。

处理器中与分页单元有关的寄存器为CR0-CR3控制寄存器，其中CR1被处理器保留，CR2寄存器则用于存放页故障线性地址，当根据某个线性地址所寻址的页不在内存中时将触发一个缺页异常，此时处理器负责将该线性地址加载至CR2寄存器从而把适当的页重新加载到内存中。与分页单元联系最紧密的当属CR0和CR3控制寄存器，其中CR0寄存器中各个位的组织结构如下所示：

 重点关注这两个标志位：

* PE位用于实模式与保护模式之间的转换。
* PG=0时禁用分页机制，PG=1时则启用分页机制。若禁用分页那么线性地址

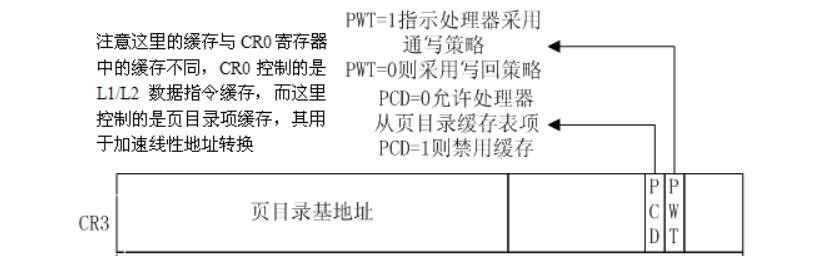
与物理地址一一对应，但通常将该位置1以启用分页机制，此时线性地址需要通过分页单元的转换才能形成物理地址。

* CD位用于启用或禁用高速缓存，注意这里的高速缓存所对应的是L1，L2数据或指令高速缓存而并非是用于加速线性地址转换的TLB。

下面是CR3寄存器的组织结构图：

CR3是页目录基址寄存器，保存页目录表的物理地址。页目录表总是放在

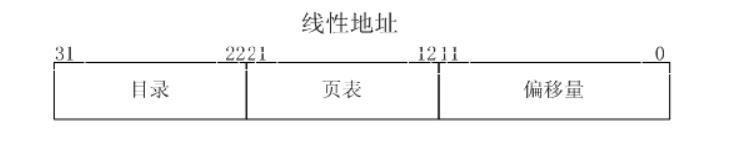
以4K字节为单位的存储器边界上。因此，它的地址的低12位总为0，不起作用，即使写上内容，也不会被理会。

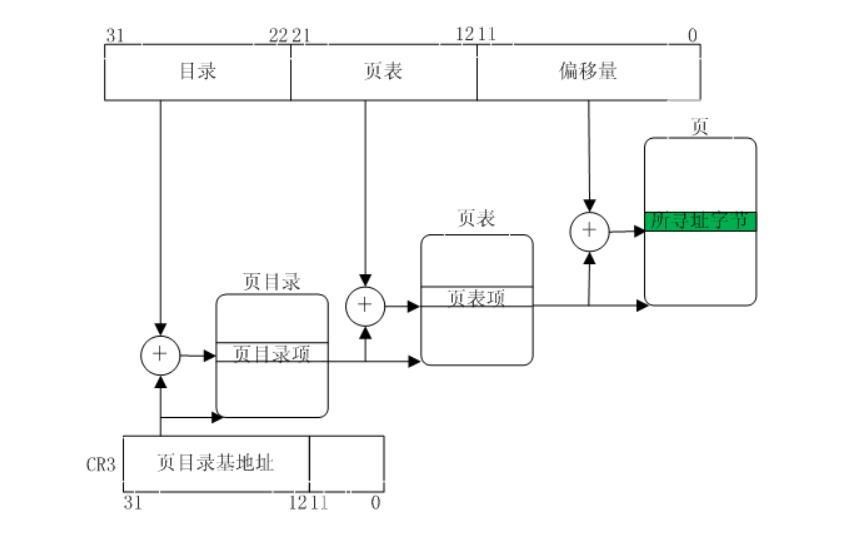


分页机制的实现原理：

在分页模式下，32位的线性地址被划分成3个域：

* Directory(目录)，在线性地址中为最高 10 位，用于访问页目录表中的项。
* Table(页表)，在线性地址中为中间10位，用于访问页表中的项。
* Offset(偏移量)，最低12位，用于访问页内的某个存储单元。



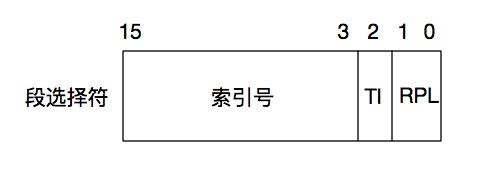
采用两级页表结构完成线性地址到物理地址的转化。首先由CR3寄存器中的页目录基地址域找到相应的页目录表，通过线性地址中的目录域找到对应的页目录项，随后根据页目录项中存放的页表的基地址找到相应的页表，再通过线性地址中的页表域找到相应的页表项，最后根据页表项中存放的页的基地址及线性地址中的偏移量便完成了线性地址到物理地址的转换。  由于每次存储器访问都要存取两级页表，这大大降低了访问速度。所以，为了提高速度，计算机设计使用一个最近存取页面的高速缓存硬件机制（TLB）。当线性地址被第一次使用时，通过页目录/页表计算得出相应的物理地址，这个地址在使用后将被缓存在TLB中，以备将来对同一线性地址引用时直接从TLB中得到其对应的物理地址。这里还要注意的是，当CR3控制寄存器被更新时，硬件将自动使TLB中的所有项均无效，因为CR3被更改后将存放新的页目录基地址，所以线性地址转换时不允许再引用TLB中的表项。

分段内存管理方案

分段机制使用在实模式下，可以将逻辑地址转化为线性地址。逻辑地址空间由一组段组成。每个段都有名字和长度。地址指定了段名称和段内偏移。因此用户通过两个量来指定地址：段名称和偏移。为了得到线性地址，需要从相应的段寄存器中取出16位的段标识符（段选择符），通过这个段标识符可以得到一个段基址。然后将得到的段基址与指令中的段内偏移相加，从而得到一个线性地址。

下面具体分析分段机制

段寄存器的唯一目的就是存放段选择符，段选择符是一个16位长的字段：



第01位表示的是请求者的特权，第2位是表指示器，用于指定所使用的描述符表。3-15位是一个索引号，定位到描述符表中的某一项段描述符。

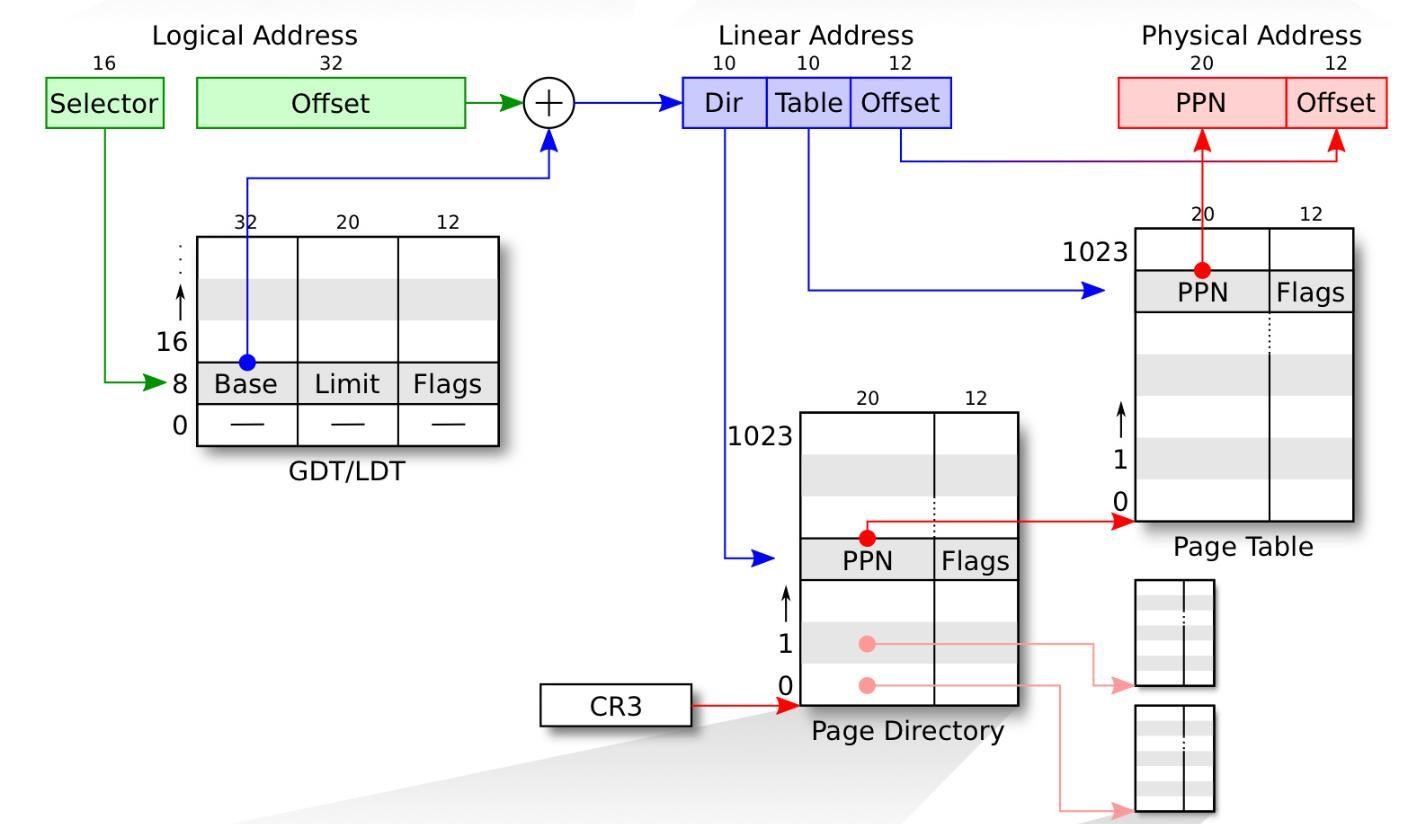
段描述符就是用来描述段的特征，段描述表放在全局描述符表（GDT）或者局部描述符表（LDT），每个段描述符为 8 个字节大小。GDT 的地址和大小存放在寄存器gdtr中，LDT的地址和大小存放在寄存器ldtr中。

不同类型的段对应不同的段描述符；代码段描述符，数据段描述符，任务描述段描述符，局部描述符表描述符等。

当指令给出一个地址后，硬件的处理顺序如下：

1. 根据指令类型选择好段选择符，检查段选择符的TI字段，决定使用GDT还是LDT，再从gftr寄存器或者ldtr寄存器中获得描述符表的地址。
2. 使用段选择符的索引号，定位到段描述符，描述符地址 = 索引号 \* 8 + gdtr(ldtr)中的地址。
3. 将段描述符中的Base值与给出的地址相加，从而得到线性地址。
4. 将线性地址通过分页单元换换成物理地址。

下面为分段单元与分页单元结合的完整的逻辑地址转化过程图：



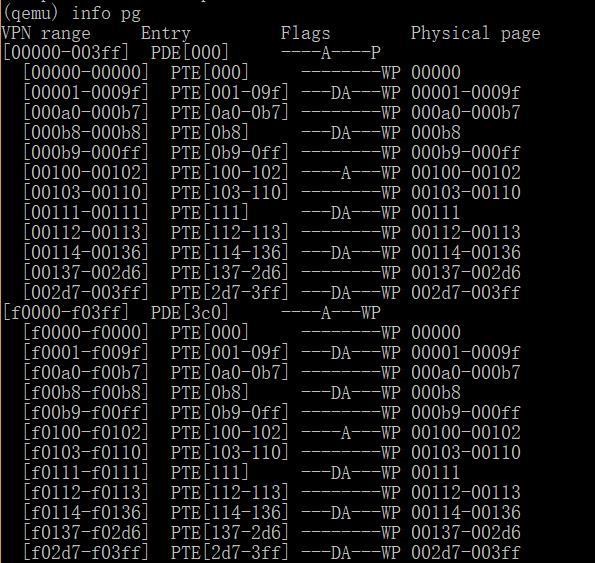
练习3. 虽然GDB只能通过虚拟地址访问QEMU的内存，但在设置虚拟内存的

同时，查看物理内存通常很有用。从实验室工具指南里查看QEMU [监视器命令，](http://oslab.mobisys.cc/pdos.csail.mit.edu/6.828/2014/labguide.html#qemu)特别注意 xp命令，它可以让你查看物理内存的内容。要调用QEMU监视器，

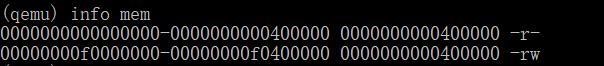
请在终端中下 Ctrl-a c（注：这个表示先同时按下ctrl和a，然后再按下c）使用***QEMU***监视器中***xp***命令和***GDB***中的的***x***命令查看相应物理地址和虚拟

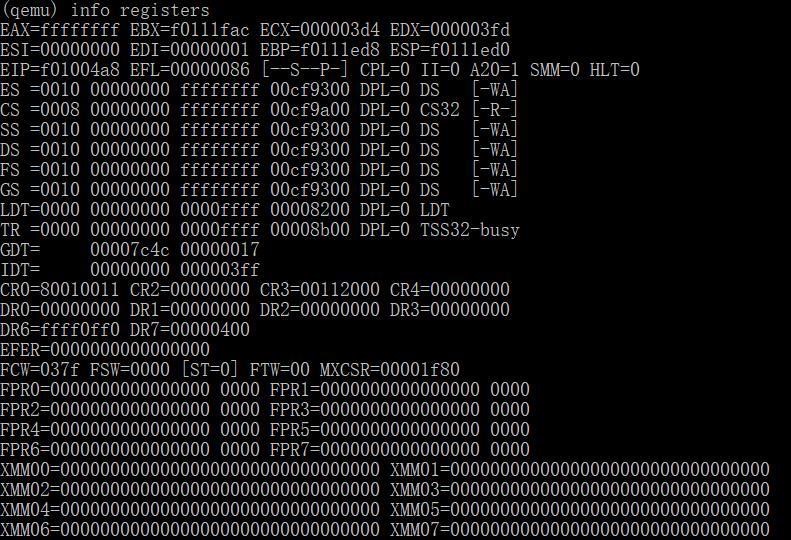
地址上内存的内容，并确保看到相同的数据。

显示当前页表详细信息，包括所有内存映射的范围，权限和标志：

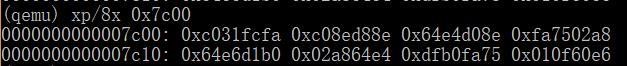


显示虚拟内存的映射范围和配置了哪些权限：

 显示所有内部寄存器的状态：



查看相应物理地址上的内容：



问题1:假设以下*JOS*内核代码是正确的，变量*x*应该是什么类型，uintptr\_t 还是physaddr\_t？（类型 uintptr\_t 代表不可见的虚拟地址，physaddr\_t 代表物理地址，这种表示方式只是为了程序员编写代码时更加清晰，实际上两个类型都是相同的32位整数）

mystery\_t x;

char \* value = return\_a\_pointer（）;

\* value = 10;

x =（mystery\_t）value;

答：x 是 uintptr\_t 类型。在内核中对数据操作都是以内核虚拟地址形式进行

的， 则在第三句“\*value = 10;”使用 uintprt\_t 类型值对\*value 进行赋值，

所以 value 肯定是一个虚拟地址，虽然在“x=(mystery\_t)value;”一句中看似对 x 进行了强制类型转换，但给 x一个物理地址是没有意义的，因此 x 仍为 uintptr\_t 类型。

参考计数

在实验中，我们经常将同一物理页面同时映射到多个虚拟地址（或多个进程的地址空间中）。在与物理页面对应的struct PageInfo的pp\_ref字段中保留了对每个物理页面的引用数量的计数。当物理页面的此计数为零时，该页面可以被释放，因为它不再被使用。一般来说，这个计数应该等于物理页面在所有页表中UTOP之下的部分出现的次数。

页面管理

完成页表管理：插入删除线性到物理地址的映射和创建页表。

作业 4 在文件 kern/pmap.c 文件中，你必须实现以下函数的代码： pgdir\_walk() boot\_map\_region() page\_lookup() page\_remove()

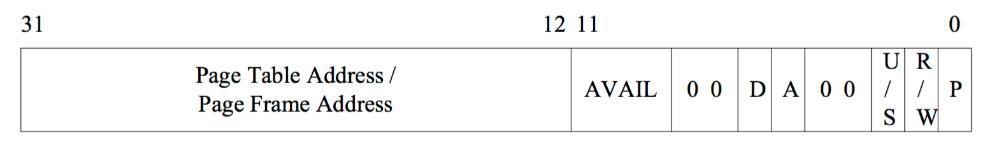
page\_insert() mem\_init()调用的 check\_page()，用于测试你的页表管理方法。

答：

需要用到的宏定义：取得页目录项（PDX）取得页表项（PTX）

取出该项内容的前20位（PTE\_ADDR）（页目录项的前20位是物理地址）通过物理地址查找对应的Page结构（pa2page）获得此Page的虚拟地址（page2kva）

根据Page结构得到其物理地址（page2pa）物理地址转化为虚拟地址（KADDR）虚拟地址转化为物理地址（PADDR）



1. pgdir\_walk()

pgdir\_walk函数根据参数虚拟地址va，查找其对应在二级页表中的page

table entry（PTE），并返回指向此PTE的指针，如果暂时没有此PTE，则根据create参数的指示选择是否创建这一PTE。查找的方式就是从一级页表中获取va对应的第PDX(va)项页目录，从而找到对应的二级页表，从二级页表中获取va对应的第PTX(va)项页目录，即找到了要返回的PTE。

参考注释中给出代码逻辑：

相关的页面表页面可能尚不存在

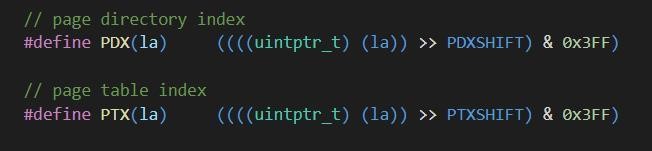
如果是真的，并且create == false，则pgdir\_walk返回NULL。

否则，pgdir\_walk将使用page\_alloc分配新的页表页面。

* + 如果分配失败，pgdir\_walk将返回NULL。
  + 否则，新页面的引用计数会递增，页面被清除，并且pgdir\_walk返回指

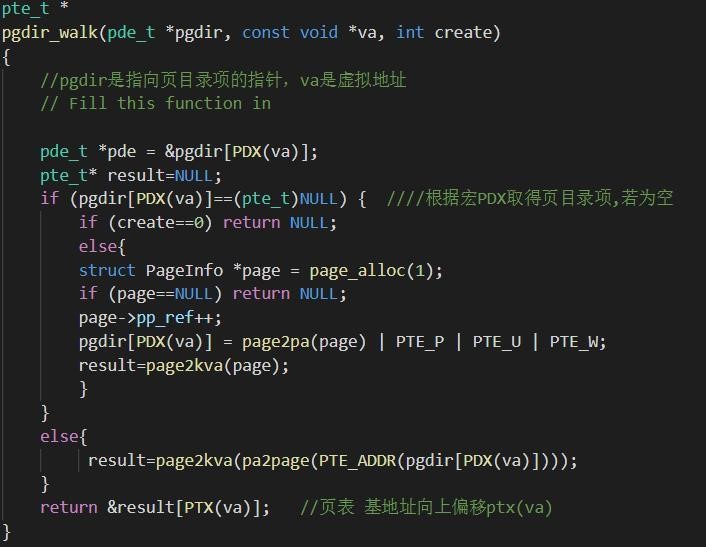
向新页表页面的指针。

使用到的inc/mmu.h中的宏定义：





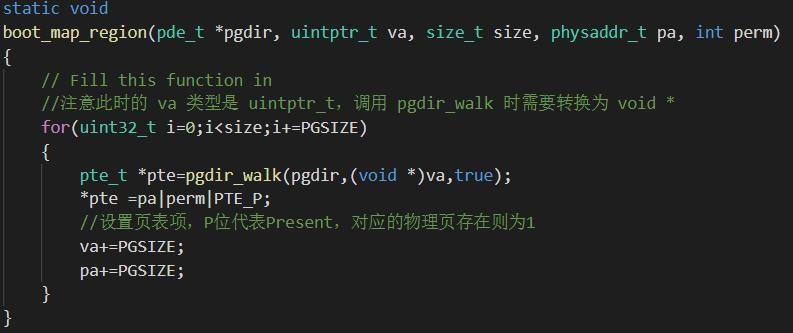
代码如下（代码解释见注释）：



1. boot\_map\_region()

注释说明这个函数将虚拟地址[va, va+size)映射到物理地址[pa, pa+size)。并将每一个对应的页表项的permission设置为perm|PTE\_P,可以借

助create参数为1的pgdir\_walk函数完成。代码如下：

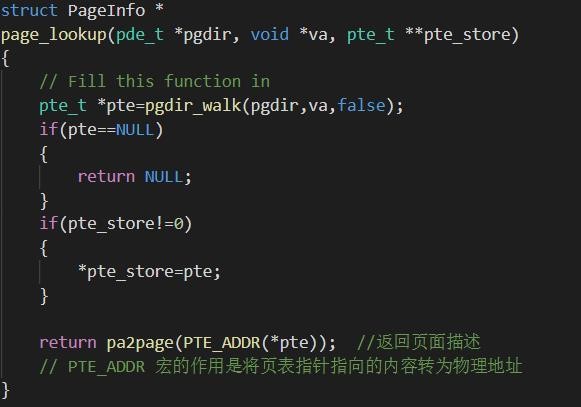


1. page\_lookup()

该函数查找并返回虚拟地址va映射的物理页的描述，如果返回为NULL，说

明没有找到。同时如果pte\_store参数不为空，则将找到的pte放到 pte\_store指向的地址中。查找的方式是使用create参数为false的 pgdir\_walk函数。

代码如下：

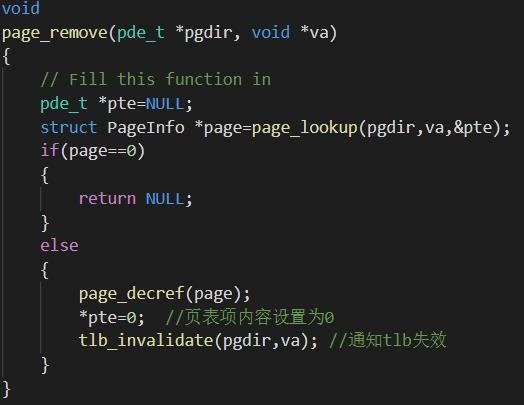


1. page\_remove()

该函数在虚拟地址'va'处取消映射物理页面。首先通过page\_lookup函数

查找va对应的Page，如果返回为NULL，那么不用进行处理，否则要对返回的 Page执行ref-1操作，如果减一后ref变为0，则要回收此Page（上述操作都

封装在page\_decref函数中），最后通知TLB高速缓存这一缓存失效（调用 tlb\_invalidate函数）。代码如下：

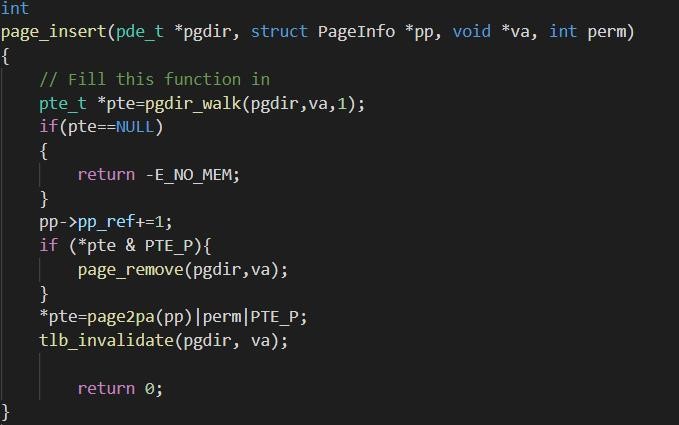


1. page\_insert()

page\_insert函数将参数虚拟地址va映射到参数struct Page\* pp所对应的物理页，页表项的权限（低12位）应该设置为'perm | PTE\_P'。

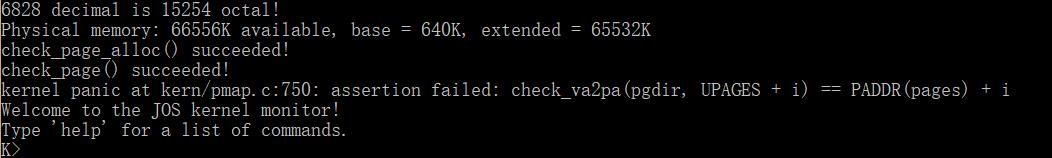
具体实现方式：首先使用 pgdir\_walk 查询该地址的 pte（若不存在则建立，

第三个参数传入1），如果pte为空，说明没有额外的空间分配页表，因此返回-E\_NO\_MEM。该操作是查找该va是否已经映射到了某个物理页面，如果映射到了，则解除映射。注释中指出对va本来映射的Page就是pp对应的Page的情况，可以暂时忽略。使用pgdir\_walk查询该地址的pte（若不存在则建立，第三个参数传入1），如果pte为空，说明没有额外的空间分配页表，因此返回-E\_NO\_MEM。否则将该pte的内容填充成相应物理页面的物理地址，增加这个物理页面的引用次数。



这里有一点需要注意：pp->pp\_ref+1的操作要在remove操作之前进行，因为若顺序颠倒则可能出现ref原本为1，先被remove（ref-1）ref=0，自动加入空闲链表中，在使其ref+1，就会造成空闲链表中出现ref不为0的错误页。

全部更改完毕运行qemu，结果如下，check\_page()succeeded!：



内核地址空间

上一部分主要是完成了一些分页相关机制的工作，但还没有真正的去使用这个分页系统。这一部分主要就是让我们使用part2中完成的映射机制来初始化内核的页目录和页表，并将此页目录加载到cr3里，让os真正去使用我们初

始化之后的页目录以取代kernpgdir.c里面简单的页目录。

权限和故障隔离

内核和用户内存都在各自的环境地址空间中，必须在x86页表中使用访问

权限位(Permissions bits)来使用户代码只访问用户的地址空间，否则用户的代码bug会覆盖内核数据，造成系统崩溃。其中可写权限位(PTE\_W)可以同时影响用户和内核代码。

高于ULIM的内存内核可以读写，而用户环境没有权限。内核和用户在地址 [UTOP,ULIM)有同样的权限：可读但不可写，这部分地址空间通常是一些特定的

内核数据，让用户环境可以读取。最后，地址UTOP之下的是用户环境。

初始化内核地址空间

练习***5.*** 补全在*mem\_init()*函数在调用*check\_page()*函数后的代码。

你的代码要通过*check\_kern\_pgdir()* 和 *check\_page\_installed\_pgdir()*两个函数的

检查。

答：首先查看inc/menlayout.h,下图描述了虚拟地址空间的结构





首先从高位地址说起，kernbase到最高为4g是一块remapped内存，这块很大的内存从低到高要映射整块物理内存。

其次kernbase往下PTSIZE是无用内存，再往下是KERNSTACKTOP也就是内核栈的栈顶，众所周知栈的生长顺序是从高地址向低地址生长，所以这个位置也就是栈开始生长的位置。从KERNSTACKTOP往下KSTKSIZE为内核栈区域，大概是8个页面，内核栈不能超过这个区域。从KERNSTACKTOP往下一个PTSIZE 这块区域除了内核栈之外还有一块空白区域，防止栈溢出的时候复写用户空间的数据。

接着是ULIM，代表用户空间的最高地址，换句话说此位置往下所有空间为

用户空间，用户空间有很多东西。UPAGES是JOS用户记录物理页面使用情况的数据结构，只有 kernel 能够访问。由于用户空间同样需要访问这个数据结构，

我们将用户空间的一块内存映射到存储该数据结构的物理内存上（PAGES映射到UPAGES位置）。

（1)pages映射到UPAGES-UVPT地址空间

映射的实质就是往页表和页目录里写入相应的值，使UPAGES这个符号经过

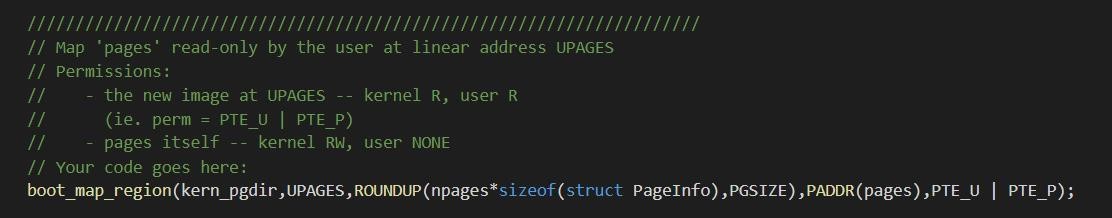
地址变换之后指向的地址恰好是pages的物理地址。我们在上一部分补全的

boot\_map\_region()函数就是将虚拟内存映射到物理内存的。主要关注该函数参数：

 目前只建立了一个页目录，即 kernel\_pgdir，所以第一个参数为

kernel\_pgdir。第二个参数是虚拟地址，UPAGES 本来就是以虚拟地址形式给出的。第三个参数是映射的内存块大小，获取pages数组的大小，并向上对齐到 PGSIZE得到要映射的页面。第四个参数是映射到的物理地址，直接取 pages

的物理地址即可。根据注释权限 perm = PTE\_U | PTE\_P。（用户空间内核只读） mem\_init()补全代码如下：

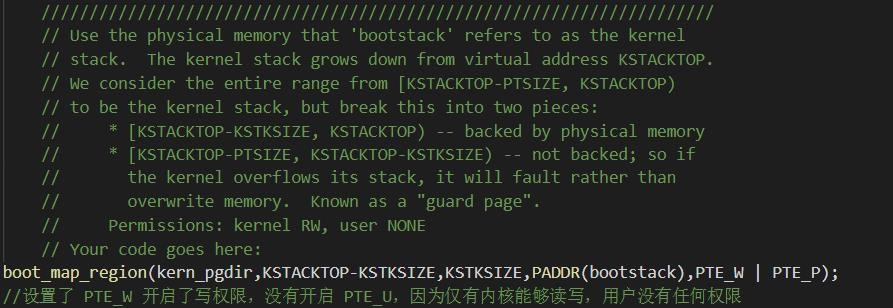


(2）内核栈的映射（bootstack到Kernel的stack）

内核栈的符号地址是bootstack表示栈顶，需要将KSTACKTOP-KSTKSIZE映

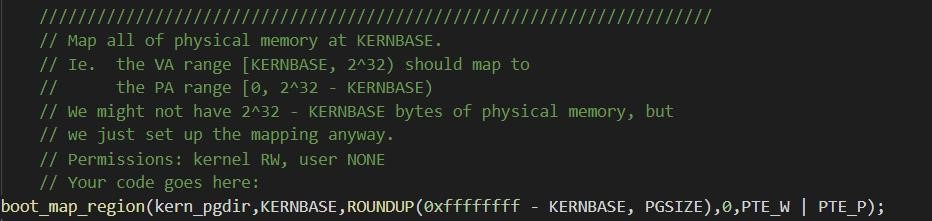
射到这个位置，并向上按顺序映射KSTKSIZE大小的内存区域。权限为内核读写。

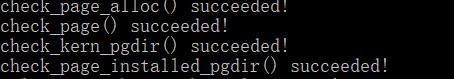
类似的使用boot\_map\_region()函数，代码如下：



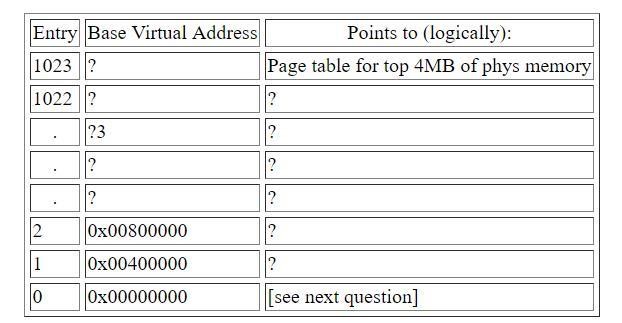
（3）映射整块内存区域

该部分需要将KERNBASE到0xffffffff总共256M内存的空间与从0开始的物理地址建立映射。权限为内核读写。

 至此三个映射全部建立，代码更改完成，运行qemu查看结果：



问题2：假设下图描述的是系统的页目录表，哪些条目（行）已经被填写了？他们是怎么样进行地址映射的？他们所指向的位置在哪里？请尽可能完善这张表的内容。

 答：根据memlayout.h的虚拟内存分布图进行填写：

|  |  |  |
| --- | --- | --- |
| Entry | Base Virtual Address | Points to (logically): |
| 1023 | 0xFFC00000 | Page table for top 4MB of phys memory |
| . | . | . |
| 960 | 0xF0000000 | Phys Memory |
| 959 | 0xEFC00000 | Kernel Stack |
| 958 | 0xEF800000 | Memory-mapped I/O |
| 957 | 0xEF400000 | Page Table |
| 956 | 0xEF000000 | Page Structures |
|  |  |  |
| 2 | 0x00800000 | Program Data &Heap |
| 1 | 0x00400000 | Empty Memory |
| 0 | 0x00000000 | Empty Memory |

问题3.我们已经将内核和用户环境放在同一地址空间内。为什么用户的程序不能读取内核的内存？有什么具体的机制保护内核空间吗？答：：将内核和用户环境放在了相同的地址空间，所以内核态和用户态的保护

机制无效。但是页表可以设置权限位。如果没有将 PTE\_U 置 1 则用户无权限读写内接内存。

问题4.JOS操作系统可以支持的最大物理内存是多少？为什么？

答：Page 结构有两个成员：pp\_link 和 pp\_ref，该结构体共占用 8B（sizeof 输出过）。 在 inc/memalyout.h 中显示出来的页面布局中 ,pages 数组是存放的 [UPAGES,UVPT)这段内存中，大小为 4MB。 我们可以计算得出我们可以存放的 Page 结构的个数为：4MB/8B=0.5M。 而一个 Page 对应的物理页面的实际大小为 4KB，所以这个操作系统最大能支持的物理内存大小为：

0.5M\*4KB=2GB。

这里UPAGES对应的就是pages这个链表。程序空间在利用虚拟地址访问

pages的时候。一旦大于4MB，比如越界到了UVPT这个空间。由于这部分虚拟地址是放到了kern\_pgdir里面。所以这个时候超出的部分就不能访问了。也就是物理空间上，pages占用多大空间都没有问题。但是虚拟地址空间在访问UPAGES

的时候就是不能访问全。因此，能支持的内存大小为 2GB。这是内存保护问题。

问题5.如果我们的硬件配置了可以支持的最大的物理内存，那么管理内存空间的开销是多少？如何减少这种开销?

答：如果我们有2GB的物理内存，我们需要4MB的PageInfos来管理内存加上

4MB的页目录表和页表，共8MB。

问题六：再次分析kern/entry.S和kern/entrypgdir.c 的页表设置的过程，在打开分页之后，EIP依然是一个数字（稍微超过1MB）。在什么时刻我们才开

始在KERNBASE上运行EIP。当我们启动分页，当我们开始在KERNBASE上运行 EIP之时，我们能否以低地址的EIP继续执行？这个过渡为什么是必须要的？

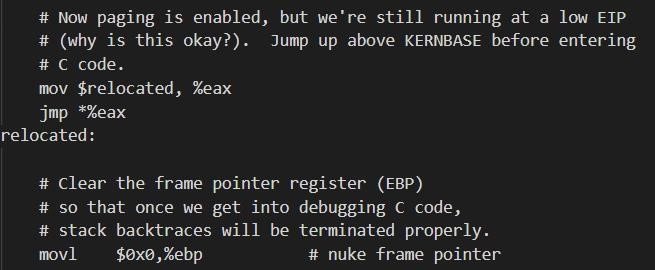
答：在jmp \*％eax完成之后。我们可以以低地址的EIP继续执行。

kern/entrypgdir.c 中将 0 - 4MB 和 KERNBASE - KERNBASE + 4 MB 的虚拟地

址都映射到了 0 - 4MB 的物理地址上，因此无论 EIP 在高位和低位都能执行。

这是过度是必要的，因为稍后将加载kern\_pgdir（页目录）并且将放弃va

[0,4M]。如果只映射高位地址，那么在开启分页机制的下一条语句就会 crash。



# Challenge

扩展*JOS*的内核监视器的命令：

1. 以有用且易于阅读的格式去展示特定地址范围的所有的物理页面映射（或缺页），仅展示当前活动地址空间中的内容，地址的范围可以是线性地址或虚拟地址。例如，输入‘*showmappings 0x3000 0x5000*’命令，可以显示物理页面映射以及映射在虚拟地址*0x3000 0x4000 0x5000* 位置的页所对应的权限位。
2. 可以手动设置，清除或者更改当前地址空间所映射的页的权限位。
3. 显示出指定虚拟地址或者物理范围内的内存的内容。确保当范围超过了页边界时，你的代码的功能依旧正常。

答：

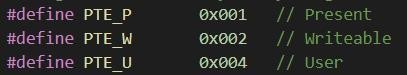
1. 首先需要了解strtol函数，函数原型为：

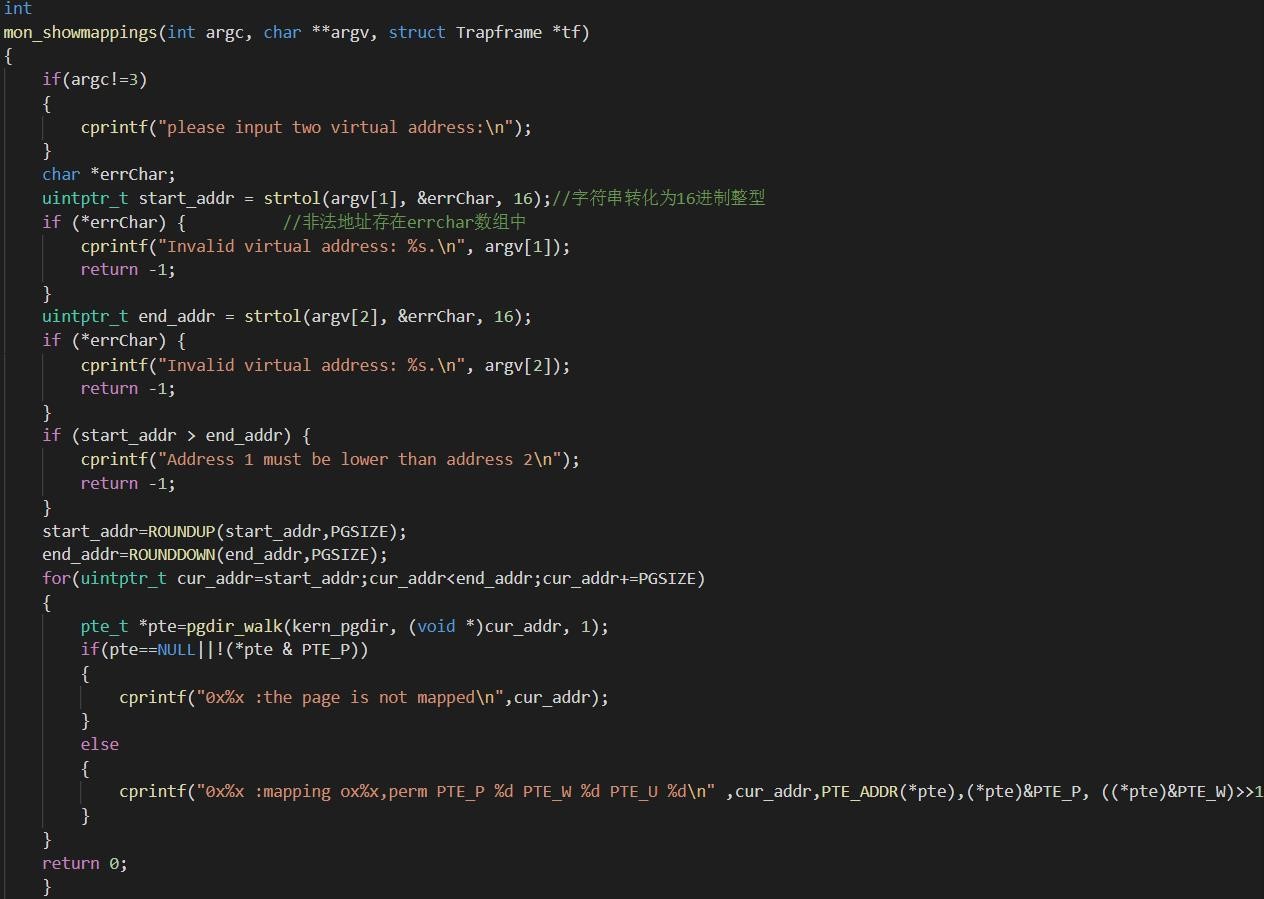
long int strtol(const char \*nptr,char \*\*endptr,int base);

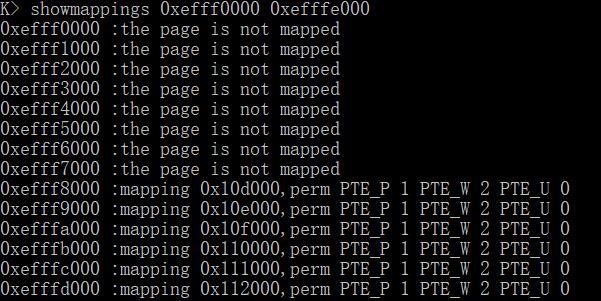
作用是将字符串转为整数，可以通过 base 指定进制，会将第一个非法字符的指针写入 endptr 中。在mon\_showmappings函数中，使用strtol函数将控制台输入的char型转化为16位整型地址。

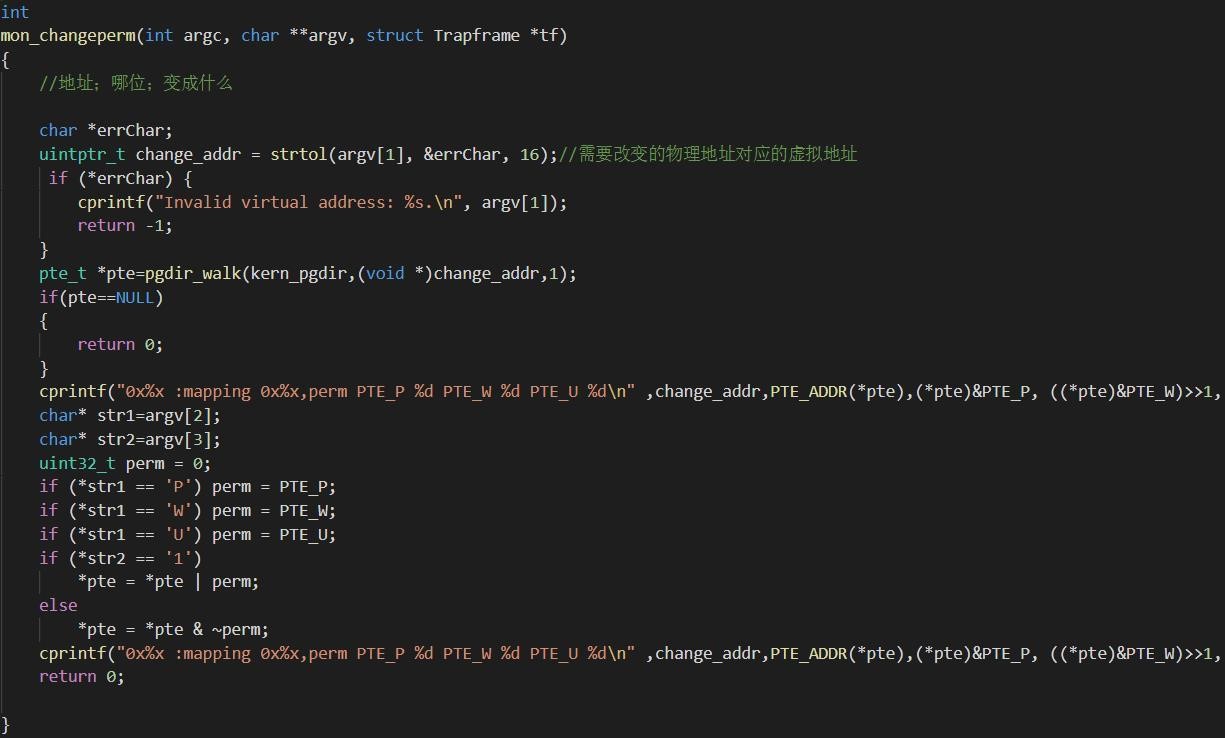
mon\_showmappings函数利用pgdir\_walk函数，通过一个循环，将从开始地址到结束地址范围内的虚拟地址对应的物理地址输出（利用PTE\_ADDR）以及

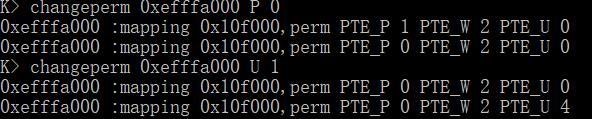
PTE页表项中的权限位情况（这里只输出P，W，U三位）



 运行结果如下：



1. 第二部分需要实现对权限位的自定义修改，输入格式为： changeperm va P/U/W(待操作的权限位) 0/1(置位还是清空)  输出结果如下图：

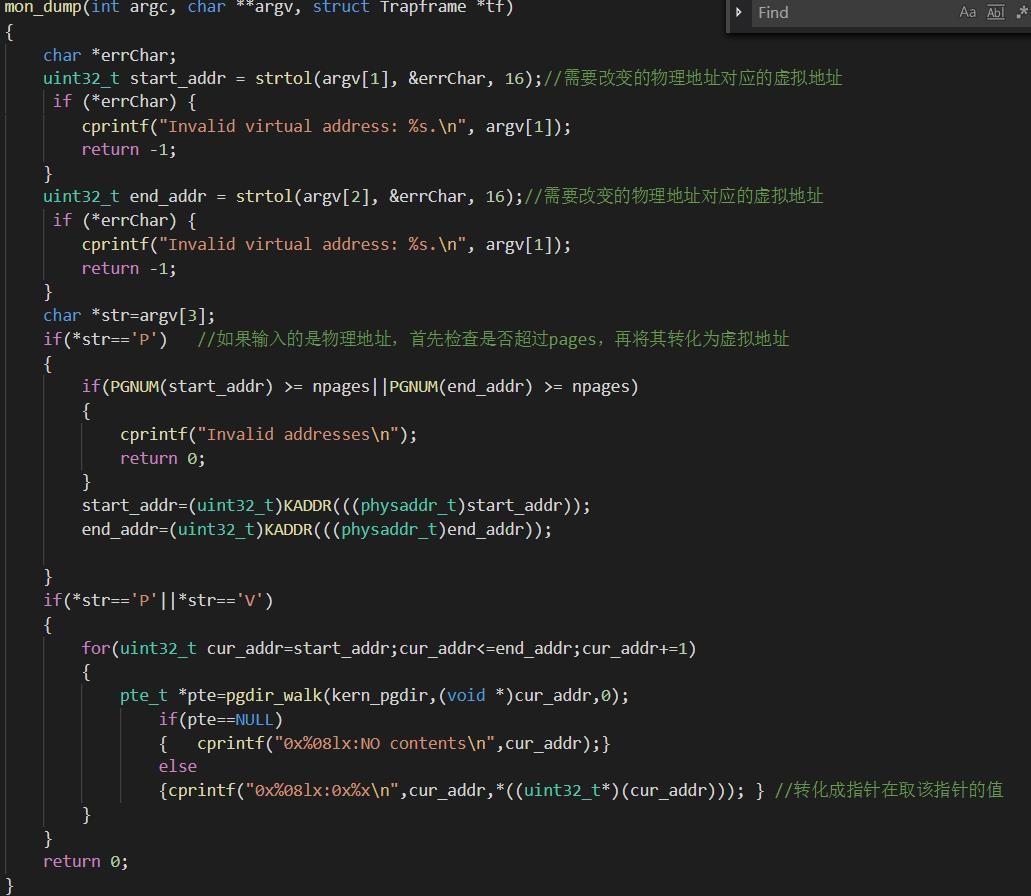


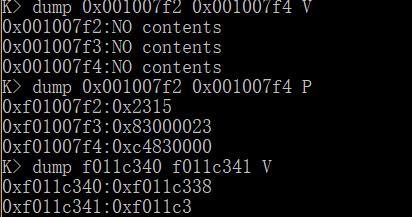
1. 为了分别显示出指定虚拟地址或者物理范围内的内存的内容，在输入时前两位分别是查询范围的开始地址和结束地址，第三位是判断为，当输入为 P 时，代表输入的地址是物理地址，V代表虚拟地址。在程序实现上，对于虚拟地址只需要将整型地址转化为指针，然后输出指针的内容即可。物理地址首先利用

PGNUM宏定义，检查其是否超过pages，再将其转化为虚拟地址。

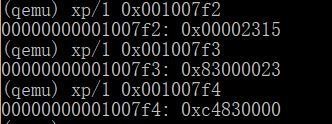
代码如下：

（%08lX表示用8位数字表达一个十六进制指针地址）

 运行结果如下：

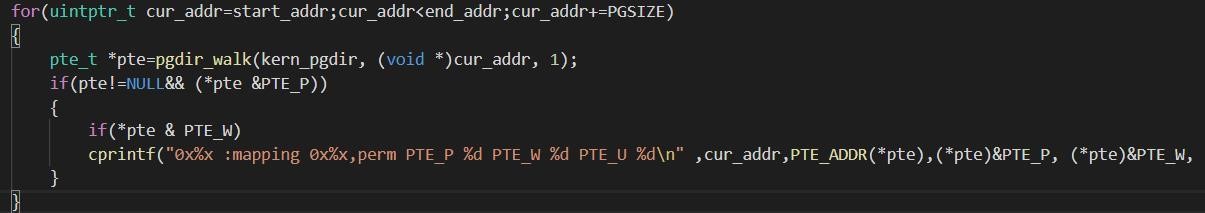


其中物理地址的值可以使用qemu下的xp命令查看，结果与dump命令下输出相同：

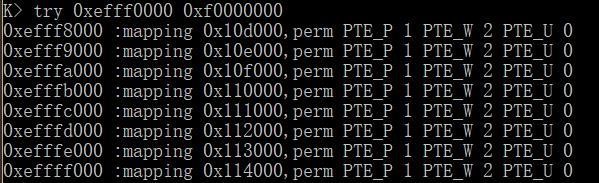


1. 自定义

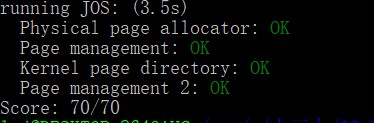
仅输出指定范围内PTE\_W为2的映射（内核可读写）仅需在showmappings加入一句判断即可：



输出结果为：



Make grade结果为：



至此，Lab2全部结束！