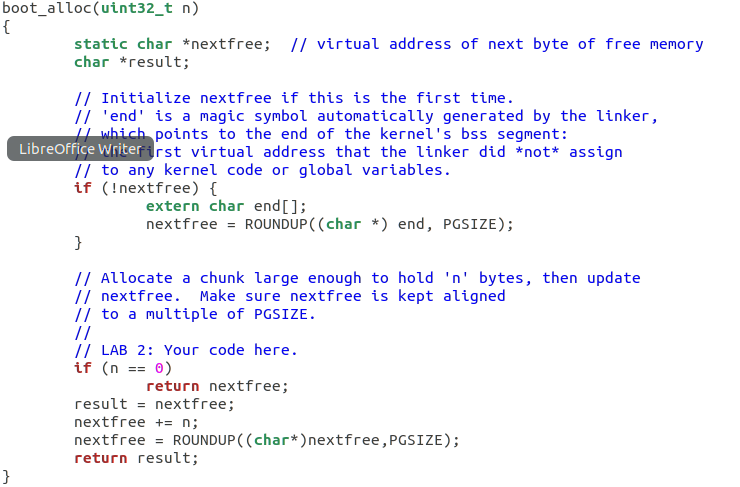
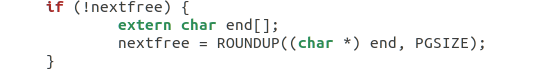
LAB2 内存管理

练习1：

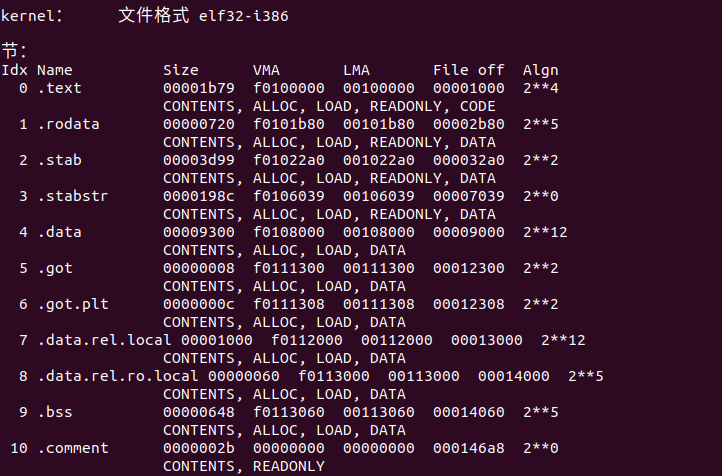
a) 

boot\_alloc(unit32\_t n)主要是申请n个字节的地址空间，返回申请空间的首地址。如果n是0，则返回nextfree(未分配空间的首地址)。其中，分配的地址是页对齐的，即4K对齐。

这个函数，巧妙应用了未初始化的全局变量和静态变量会被自动初始化为0这个特性。函数以首先定义静态局部变量nextfree，它指向空闲内存空间的首地址。由于未初始化，所以变量自动初始化为0，所以首次调用boot\_alloc()函数的时候，nextfree的值是0，会执行下面的语句：



end指向的是bss段的末尾，通过调用objdump -h kernel 可以看程序的各个段的信息。



从上面的图片可以看出，在整个程序里面，.bss段的位置是最下面的，注意.comment段是一些程序的注释信息，是不进内存的。所以end未初始化的全局变量，在.bss段中。所以end是指向的是整个内存的最后一个地址。

接下来的ROUNDUP就是一个宏定义，用来4K对齐的，最后会返回一个地址，指向end的下一个空闲页的首地址。

所以在系统第一次调用boot\_alloc()这个函数的时候，首先nextfree会被指向第一个空闲页的首地址。接下来，根据输入的n，来分配地址。如果n=0，则返回nextfree，否则分配n字节的地址，返回分配地址的首地址。注意，整个过程中，需要4K对齐。

b) 

这个函数只需要补充一部分就可以了。主要是要为struct PageInfo的结构体的指针pages申请一定的地址空间。

这个结构体，主要是用来保存内存中的所有物理页面的信息的。每一个PageInfo对应一个物理页面。

pageInfo主要有两个变量:

pp\_link表示下一个空闲页，如果pp\_link=0,则表示这个页面被分配了，否则，这个页面未被分配，是空闲页面。

pp\_ref表示页面被引用数，如果为0，表示是空闲页。(这个变量类似于智能指针中指针的引用计数)。

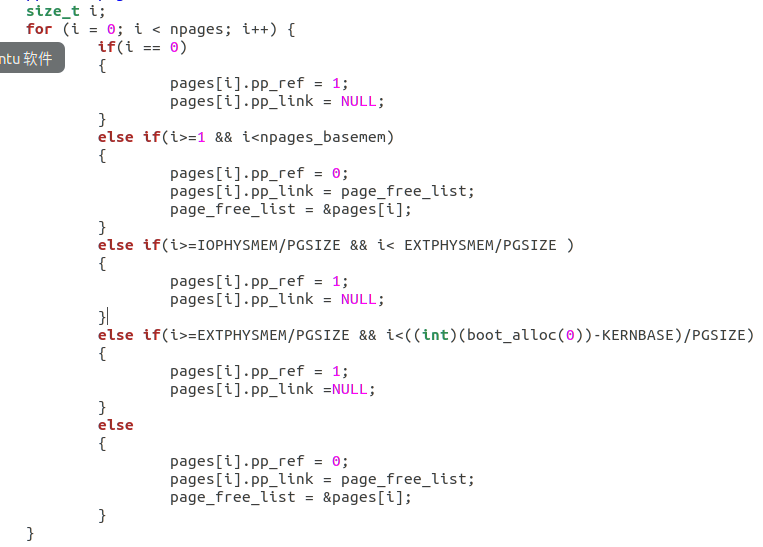
补充的代码比较简单，就是位pages申请足够的空间(npages的页面)，来存放这些结构体，并且用memset来初始化:



关于pages：

pages是用来管理物理内存页的一个结构体。首先，pages是整个系统物理内存的第0页，pages的值是一个虚拟地址。由于结构体是连续的，所以通过pages[i]-pages可以得到页的编号i，在通过i<<12就可以得到pages[i]所对应的页的物理内存，由于实现系统的物理内存和虚拟内存的转换比较简单，虚拟内存=物理内存+ 0xF0000000.所以通过pages这个结构体，在知道具体的物理页时，就可以很容易得到物理页对应的物理地址和虚拟地址

c) page\_init()函数主要就是结构体pages的初始化，就是在系统刚刚开始运行时，对物理内存分配进行初始化操作，按照上面的提示一步一步写就可以了，比较简单。

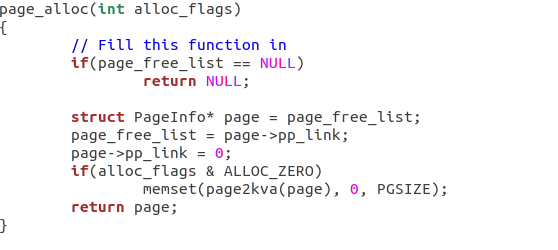


在这个初始化的函数里面，有一点需要注意。整个pages的结构通过pp\_link来区分页面是否是空闲页。对于非空闲页，只需要把pp\_ref置位，pp\_link置0就可以了，非空闲页不用通过什么东西来让他相互关联。但是对于空闲页，由于需要实时申请页面，在非空闲页释放页面之后，也需要把对应的页面加入到空闲页的结构中，所以空闲页的结构设计很重要。空闲页的结构设计要满足：申请页面时，能够较快的得到空闲页；释放页面时，能把页面较快的插入。

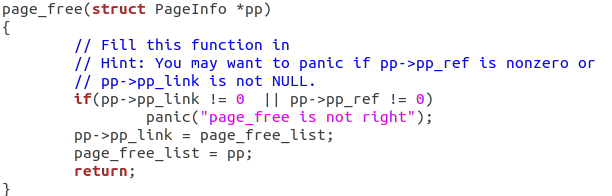
在这个实验系统中，采用了一个比较巧妙的结构，通过page\_free\_list这个变量来完成页面的申请和调用。

d) page\_alloc() 和 page\_free()

page\_alloc()是页面申请函数，整个函数思路比较简单，就是通过读取和更新page\_free\_list来申请页面。只要注意空闲页面被申请完的情况就可以了。



page\_free就是释放页面，也比较简单，只需要注意pp\_ref和pp\_link是否为0即可。



问题1：

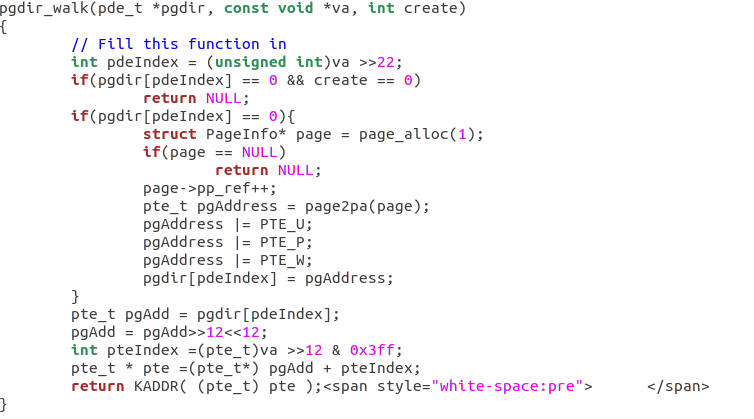
这道题木就是要分清楚在JOS系统里面，什么是物理地址，什么是虚拟地址。由于系统会经常要进行地址的相关运算，所以经常要进行强制转化，把一个unsigned int型变量转化为一个地址，或者相反，所以在程序里面，需要对两者进行区分。

上面题目的问题比较简单，x应该是uintptr\_t，在程序里面，任何指针都是虚拟地址(段偏移)。

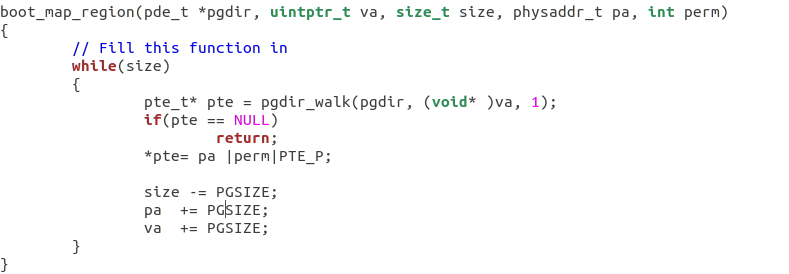
练习4：

1. pgdir\_walk()：返回va对应的二级页表的地址(PTE)

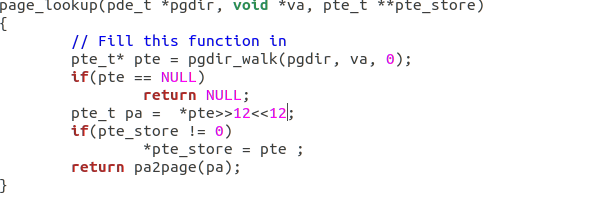
这个函数的主要作用是给定一个虚拟地址va和pgdir(page director table 的首地址), 返回va所对应的pte(page table entry)。当va对应的二级页表存在时，只需要直接按照页面翻译的过程给出PTE的地址就可以了。但是，当va对应的二级页表还没有被创建的时候，就需要手动的申请页面，并且创建页面了。过程比较简单，但是在最后返回PTE的地址的时候，需要返回PTE地址对应的虚拟地址，而不能直接把pte的物理地址给出。因为程序里面只能执行虚拟地址，给出的物理地址也会被当成是虚拟地址，一般会引发段错误。



1. boot\_map\_region： [va, va+size)映射到[pa, pa+size)

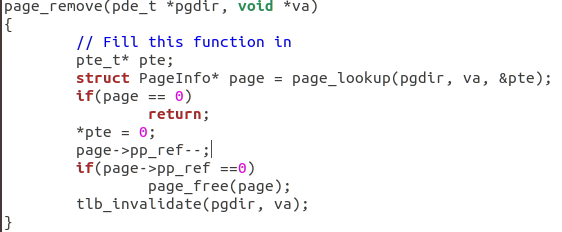


1. page\_lookup：返回虚拟地址va对应的物理地址的页面page



1. page\_remove：对va和其对应的页面取消映射

对va对应的页面进行释放，换句话说，就是取消va的映射。在这里注意，进行页面释放之后，需要把va对应的PTE里面存储的值进行清零操作，否则查询va对应的PTE时，会发生错误，系统会误以为va和pa还是存在对应关系。



1. page\_insert():把va映射到指定的物理页表page

这个函数要考虑三种情况：

1.va没有对应的映射page

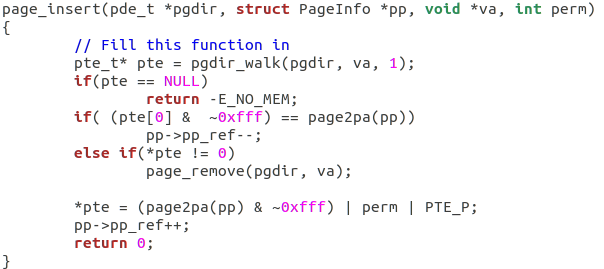
2：va有对应的映射page，但是不是指定的page

3.: va有对应的映射page，并且和指定的page相同。

对于情况1，最简单，直接把va和page映射就可以了，具体方法就是把va对应的pte求出，然后修改pte里面的值，使其为对应的page的物理地址。

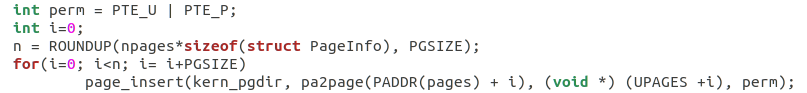
对于情况2，先要把va对应的page释放(remove,ref-1)，然后和情况1一样的处理方法。

对于情况3，要注意一点，当两个page相同的时候，不能直接返回。因为page\_insert函数不仅要进行虚拟地址和页面的映射，它还要对页面的特权(PTE\_U,PTE\_P,PTE\_W)进行设置，如果原来的page的特权和现在的特权不一样，那么直接return，就会存在问题。



练习5：

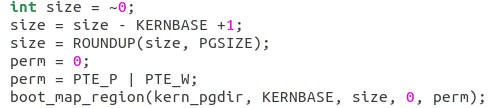
1. 第一个：要求把pages结构体所在的页面和虚拟地址UPAGES相互映射。这里只要计算出pages结构体的大小，就可以通过page\_insert()进行映射了。



1. 第二个：把虚拟地址[KSTACKTOP-KSTKSIZE, KSTACKTOP)映射到以bootstack为起点的物理地址(bootstack实际存储的是其虚拟地址，需要转换位物理地址)，这是地址之间的静态映射，所以用boot\_map\_region()就可以了



1. 第三个也是一个地址的静态映射，是把地址从[KERNBASE, 2^32)映射到[0, 2^32 - KERNBASE)。这里唯一的一点麻烦就是32位机子没有办法表示2^32，要通过一定的方法来得到size。



问题2：

| **Entry(PDENo.)** | **Base Virtual Address** | **Points to (logically):** |
| --- | --- | --- |
| 1023 | 0xfff00000 | Page table for top 4MB of phys memory |
| … | … | Page table for top 4MB of phys memory |
| 960 | 0xf0000000 | Page table for top 4MB of phys memory |
| 959 | 0xefff8000 | Kernel Stack & Invalid Memory |
| 958 | 0 | NULL |
| 957 | 0xef400000 | User read-only virtual page table |
| 956 | 0xef000000 | Read-only copies of the Page structures |
| 955 | NULL | NULL |
| … | NULL | NULL |
| 0 | 0x00000000 | 0 |

问题3：

通过权限控制，设置U/S = 0可阻止用户程序访问内核内存

问题4：

由于PAGES的大小为4MB而一个PageInfo的大小为8Byte，所以最多有524288个页面，即最大内存为524288 \* 4KB = 2GB

问题5：

无法拥有最大内存，管理内存的开销为1个一级页表与1024个二级页表，再加上4MB的PAGES，即 1025 \* 4KB + 4MB = 6100KB

问题6：

在entry.S中，指令

jmp \*%eax

执行之后eip位于KERNBASE之上，在设置页表后还有一小段指令是运行在低地址的，可以运行的原因是页表中同时也把虚拟地址[0, 4MB)映射到了物理地址[0, 4MB)，这是必须的，不然会找不到地址挑战3：用户态4G虚拟地址可用的设计方案

1、权限转化方式

只使用调用门来从用户态转向内核态。

2、内核映射方式

首先寻找未被使用的虚拟地址，如果找不到，则兑换出一块内存到硬盘，把兑换信息记录到内核中，把该块地址映射到内核，标志为kenel权限，并更改调用门中或者其他hardcoding的内核地址（为了这里修改方便，内核重要虚拟地址尽量不要写死，可以用宏的方式来实现）。跳转的新映射的内核地址，然后根据已保存信息将旧页恢复（标记为空闲或者把曾经兑换到硬盘的页读入进来）。

3、异常处理机制

用户程序访问到被内核占用的页面，会产生保护错误。在保护错误处理过程中，根据保护错误类型（这里是U/S权限出错）执行2操作。

4、优点：应用程序可以占用全部虚拟空间，提高了应用程序设计的灵活性，以及对需要超大内存空间程序的支持。

5、缺点：增加了内核的复杂度，降低了性能，尤其处理U/S权限保护错误的时候需要做很多操作。