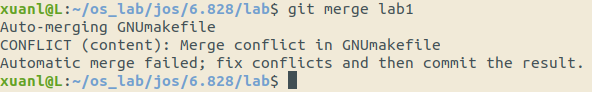
## LAB2

### Part1

git checkout -b lab2 origin/lab2

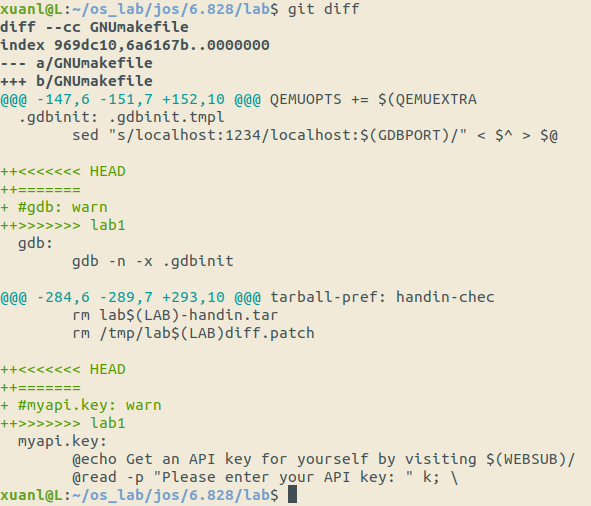
这个命令能够在本地创建lab2分支，并将远程的origin/lab2分支拷贝到本地。

注意记得git merge lab1



GNUmakefile的merge失败了。

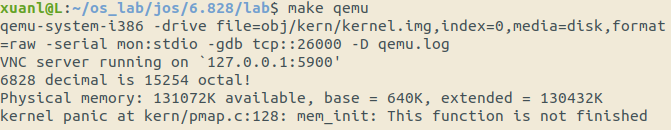
使用git diff 查看问题。



其实就是本来把那些warn，都注释掉了。现在同样都注释掉，然后commit一下。

然后就可以了。



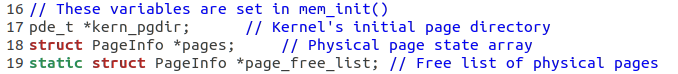


运行之后可以看到，显示mem\_init:This funciton is not finished。接下来就需要完成Exercise1.

#### Exercise 1

这个练习虽然网页上的介绍很少，但打开pmap.c发现有800多行，工作量还是不小的。

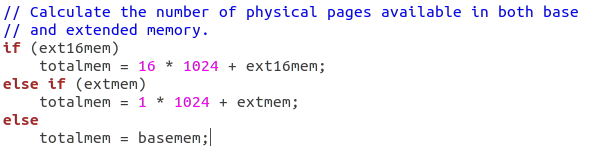
先把pmap.c过一遍吧。



这些变量需要在mem\_init()中初始化

i386\_detect\_memory（）函数初始化了几个变量，其中basemem表示基本内存（也就是主板上的内存），extended是扩展内存，也就是插入的内存条的内存大小，总内存是两者之和。

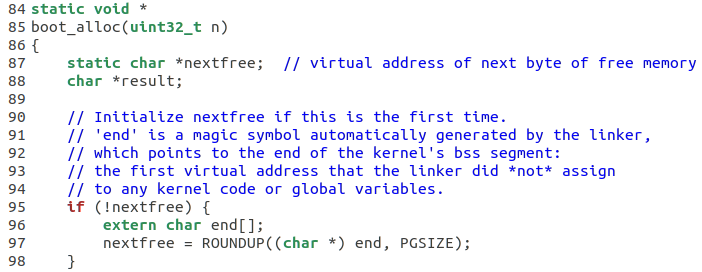
页数也分总页数和基本页数。



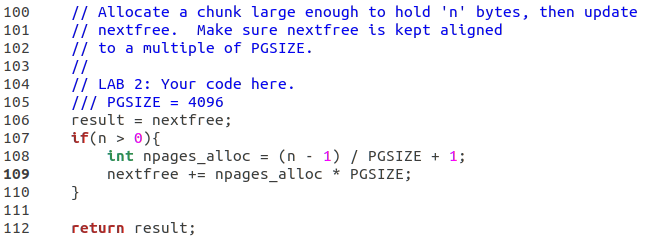
上面这三行不清楚是按照什么规范来的，这就得深入nvram\_read（）函数才能知道了。

下面遇到了需要完成的第一个函数boot\_alloc

这里面nextfree是静态变量，第一次进入这个函数时，其值为0，因此初始化为end。这里end变量就是在kernel.ld链接脚本中提供的变量，地址就是.bss段的最后。这里也可以知道，内核的堆区也是在内核的.bss上面的区域。当然，这和用户的堆区不在一起。



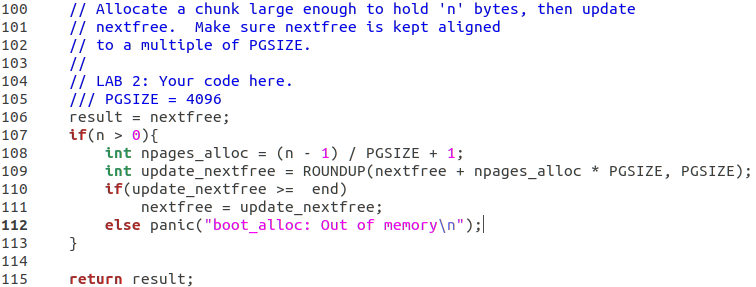
根据注释可知，这里需要考虑n是否等于0两种情况。



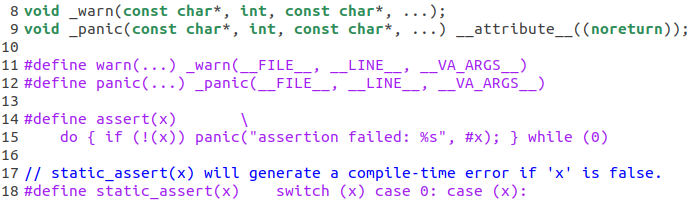
先就这样吧，暂时没考虑不够分配的问题，然后因为nextfree一开始就是按照PGSIZE对齐的，因此之后加上PGSIZE的倍数应该始终还是对齐的。

按照要求，如果不够分配需要panic。怎么判断不够分配了呢？我感觉堆区是向上生长的，因此最大就是0xffffffff，其实具体来说，就是肯定大于等于end变量。这样的话，可以比较两者的大小来判断能否分配。

修改后是这样的



这个地方的panic虽然在头文件里找到声明，但找不到实现。可能这里直接用了系统提供的库函数？



接下来要完成mem\_init函数的前半部分，后半部分建立虚拟内存之后再考虑。

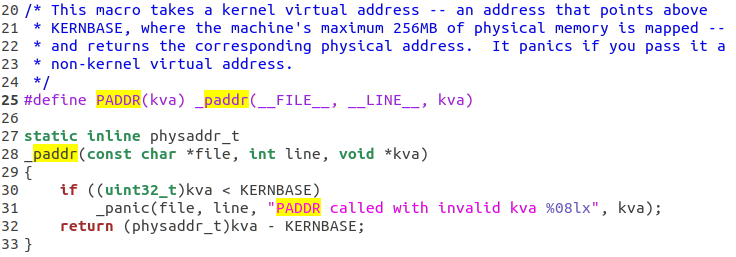
这里面经常会出现一些术语缩写，例如UTOP，这可以看memlayout.h里面的示意图。

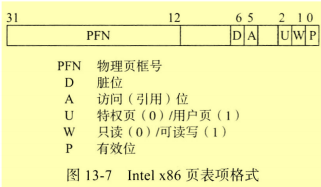
从UTOP到ULIM这部分，是用户可读但不可写的部分，再往上就是不可读不可写部分了。UTOP往下显示用户异常栈，然后是用户普通栈。

PTSIZE就是存放所有页表的空间的大小，在虚拟内存中很多的区域都是和这个大小保持一致。根据mmu.h可知，页表项数为1024，所以PTSIZE=1K×4KB，也就是4MB。



这句话，UVPT也就是存放所有页表的区域的起始虚拟地址，然后PDX宏取地址的高10位（即一级页表索引），后面的PADDR宏定义在pmap.h中（找了好久）

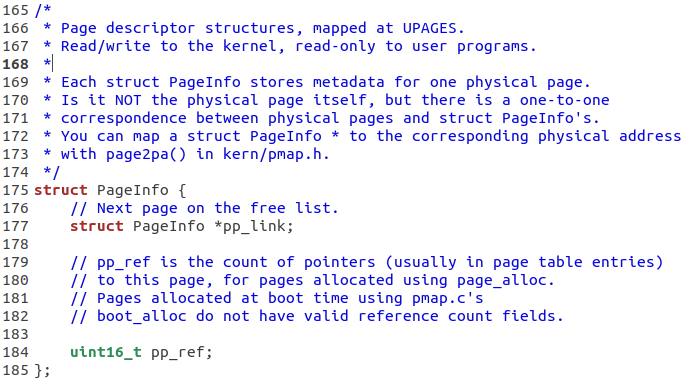




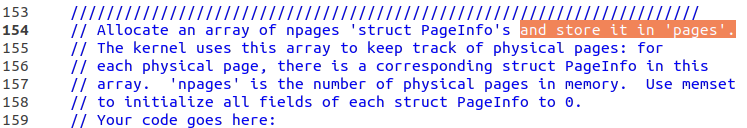
这里可以回顾p332（UNIX内核）的内容。

也就是说，页目录的UVPT这个位置的这一项的内容，就是这个页目录自身的物理地址。这个地方之后可能要再回来好好研究一下。

在memlayout.h底部的注释中有提到，页目录本身也是一个页表，所以其中有一个项是指向自身的。

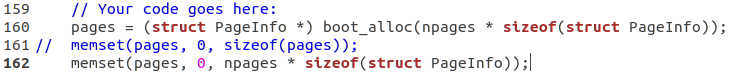


这是PageInfo数据结构



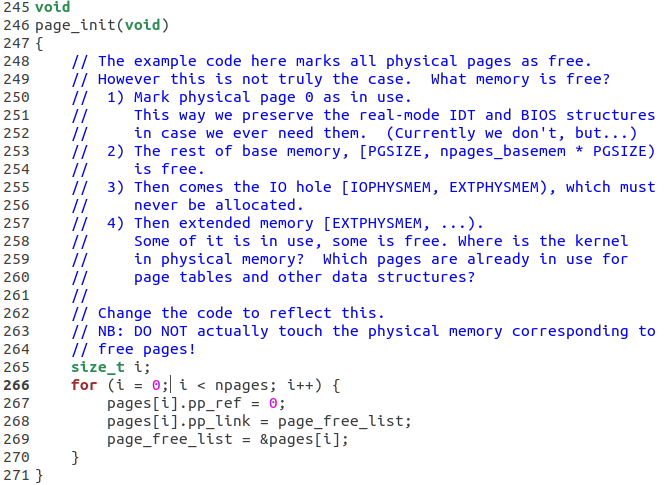
这里store in pages不太懂啥意思，好像是上面定义的那个全局变量？

PageInfo在memlayout.h中声明。



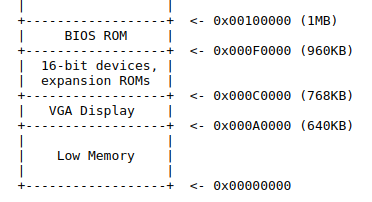
这里不知道用memset的时候应该直接用sizeof，还是得用实际大小去初始化。查阅sizeof发现，其实这是一个C关键字，而不是一个函数（从高亮颜色也可以看出来），这是在编译阶段就计算出大小的，所以这里应该不能用sizeof(pages)，因为这个是在运行阶段才能知道大小的。

事实上，如果这里用sizeof(pages)的话，结果会是4！因为pages是一个指针类型，所以占4个字节，这样结果就完全错误了！

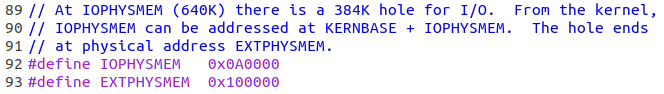


接下来要完成page\_init函数，这个感觉还是比较危险的，要考虑的细节不少。

首先page 0 应该标记为使用，那就不能出现在page\_free\_list中。



从640KB开始到1MB这部分都是被占用的。可以用memlayout.h中的宏



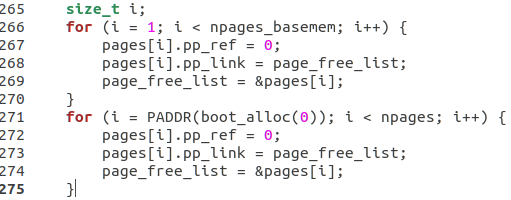
在实验1中知道，内核的物理地址是从0x00100000开始的，也就是0x00100000到0x0fffffff这部分都是内核占用的物理地址。

有些疑惑的是内核栈的位置，根据之前的汇编代码来看，此时内核栈还是在内核的.data段？而不是0xf0000000往下的那一个PTSIZE里面。

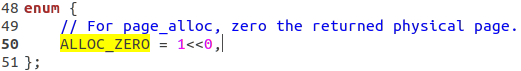
这样的话，内核全部先跳过，然后要考虑的就是页表了吧？但页表此时也在内核区，所以都跳过的话就没什么好考虑的了。

所以这里要考虑的细致一些，要考虑到内核的堆区（.bss结尾往上）的里面还有很多空间没有被分配（目前分配的只有页表和pages数组），所以没分配的那些也得标记为空闲。因为之后不会再用boot\_alloc分配了，所以那些如果标记为占用的话就再也没机会分配出来了。

而目前分配了多少内核堆区的资源呢？这个信息可以通过boot\_alloc(0)来获得！然后使用ADDR宏（pmap.h中定义）获得对应的物理地址。注意，这个宏只能作用于内核区的虚拟地址。



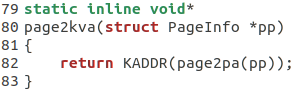
接下来要完成page\_alloc函数



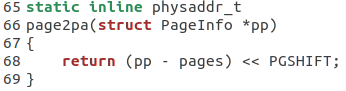
ALLOC\_ZERO不是宏，而是一个枚举类型。（意义何在？）

这里提示用page2kva，先去看看这是干啥的。

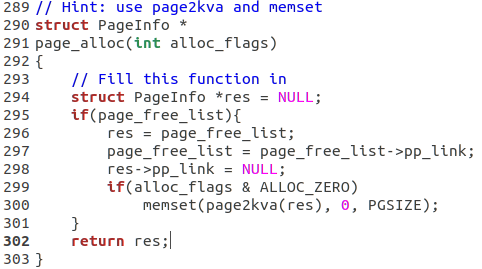
因为memset作用的对象是分配出来的内存空间，而不是page\_free\_list指向的空间！这一定要注意。



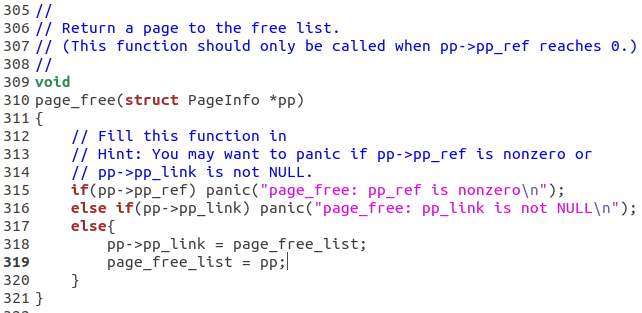
直接用地址相减得到偏移，就能够计算出页框编号，然后乘上页大小（左移12位）就能得到物理地址。最后物理地址转成内核的虚拟地址即可（这里依然假设分配出来的内存在内核区，所以不需要页表进行转换，直接加上KERNBASE即可。



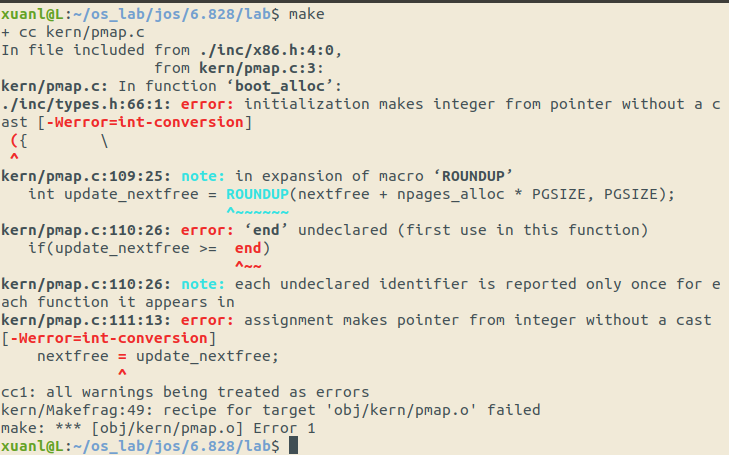
这样就能够完成page\_alloc函数了。



最后完成page\_free函数。

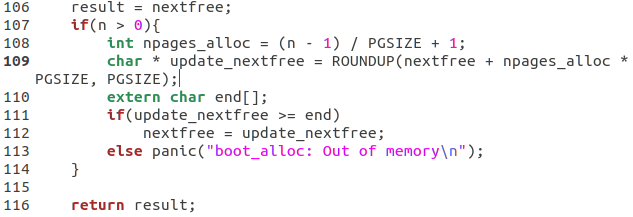


最后，把mem\_init函数中的panic注释掉，重新编译，就可以开始测试了。

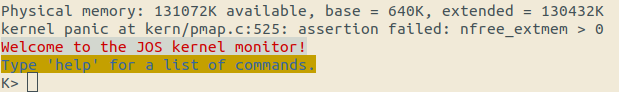


结果报错，是boot\_alloc里面，update\_nextfree的类型应该是char \*。

然后就是end没有定义。因为上文中的申明是在一个代码块中使用的，所以到下面之后end就失效了。要重新声明一下。



编译通过之后运行，发现出现了panic

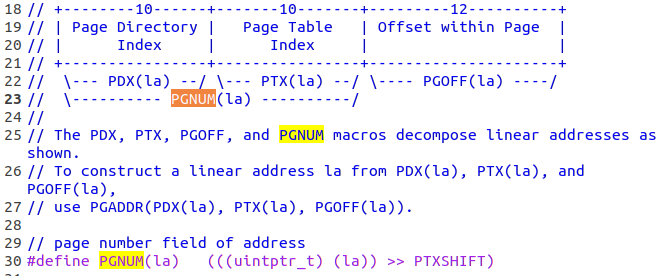


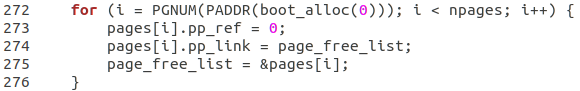
nfree\_extmem > 0，这个不知道啥意思。

总之是没通过测试。

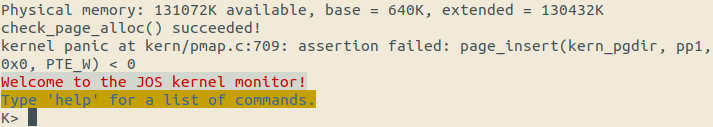
可以仔细看看check\_page\_free\_list来找出问题所在。

就是extended内存没有设置为free，原因是i初始化为PADDR(boot\_alloc(0))是错误的，i是索引而不是地址，应当改成i = PGNUM(PADDR(boot\_alloc(0)))。





修改完成后，check\_page\_alloc() succeeded!



### Part2

#### **Exercise 2**

这个练习就是阅读80386手册的5和6章，了解虚拟内存、分页、分段之类的具体实现。

手册第六章里提到，type checking位（R/W）只在U/S为1时才有效，因为内核态本来就可以访问所有的内存空间。

在手册6.3中有段保护相关的内容，这在lab3中可能会用到。但对于lab2的地址转换，不会用到段。

#### **Exercise 3**

这个练习也没有什么实质性的内容，主要是介绍QEMU的使用方法。

在模拟器的串行控制台输入终端状态下，按下ctrl+a, c 进入QEMU监视器终端，重复按一遍可回到串行控制台。

只有在QEMU才能查看物理地址的内容，而gdb只能查看虚拟地址的内容。

Xp命令查看物理地址内容

Info mem查看虚拟地址和物理地址转换

Info pg查看页表内容（一般用这个，比mem更详细）

#### **Question 1**

Assuming that the following JOS kernel code is correct, what type should variable x have, uintptr\_t or physaddr\_t?

mystery\_t x;

char\* value = return\_a\_pointer();

\*value = 10;

x = (mystery\_t) value;

答案应该是uintptr\_t吧？因为char \*是指针类型，而指针的值肯定是虚拟地址，所以x不可能是物理地址。

用KADDR（pa）来将物理地址转为虚拟地址

用PADDR（va）来将虚拟地址转为物理地址

pp\_ref是针对用户虚拟地址而言的（即UTOP以下的部分）

#### **Exercise 4**

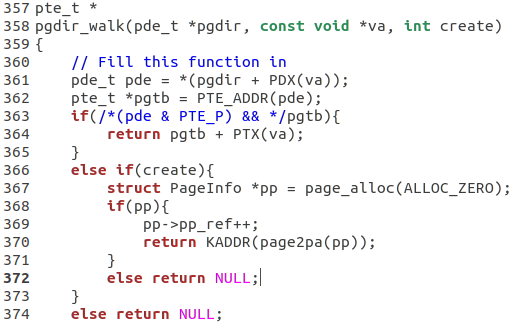
又要写代码了，还是原来的pmap.c源文件。

首先完成pgdir\_walk函数。

这个函数返回指向某个页表项的指针，作为对线性地址’va’的结果。

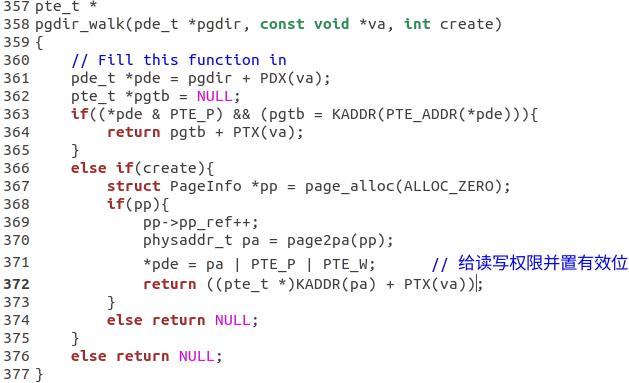
按照注释的意思，如果相应的页表页本身还不存在，似乎要新建一个空页，然后返回这个页的首地址。

这里到底怎么样才说明页表页本身不存在不太清楚，是PTE\_P位为0？还是说页框号为0？

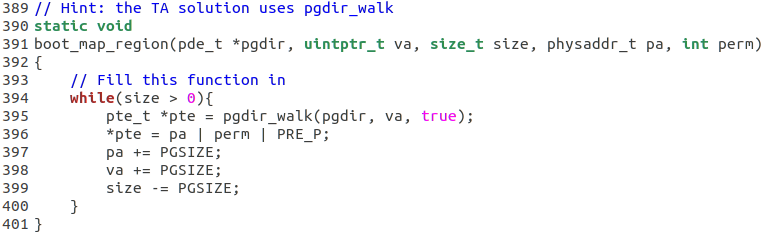


这里不太清楚要不要判断PTE\_P位，我怀疑不需要，要么先注释掉看看行不行。

看了boot\_map\_region函数的要求之后，感觉上面pgdir\_walk还是得修改一下pgdir的信息，如果需要新建一个页表的话，需要把页表的地址填入页目录中。按照x86手册上，页目录页表项里的页框表示的是物理地址，那这里还需要转换一下才能存。如果页目录里面还存虚拟地址，那就套娃了。



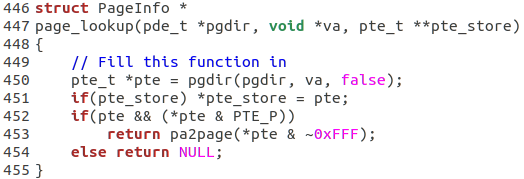
改写了一下，现在先判断PTE\_P位，然后再获取虚拟地址，避免panic。Create部分还将新建的页表页的物理地址存入页目录当中。最后返回的应该是新建的页表页中对应的页表项地址，否则这个接口根本没法用。注释中写的是and pgdir\_walk returns a pointer into the new page table page.注意这里用了“into”而不是”to“，微妙差别！



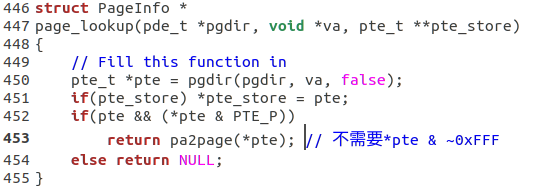
因为提示了用pgdir\_walk，所以这个的实现就相对容易一些。这里perm表示permission。

接下来按照lab2网页上的顺序，应当先完成page\_lookup和page\_remove两个函数，最后再实现page\_insert，因为那个里面要用到前两个函数。

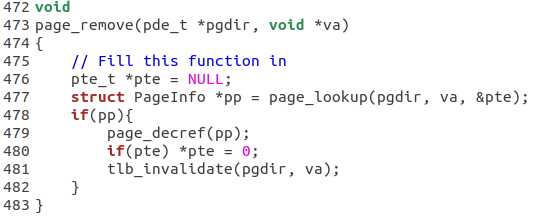
Page\_lookup



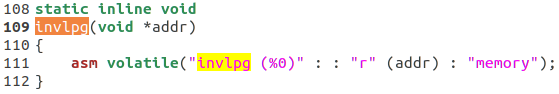
其实仔细看pa2page和PGNUM可以发现，上面不需要自己与上~0xfff。因此可以简化为下面的代码。



Page\_remove

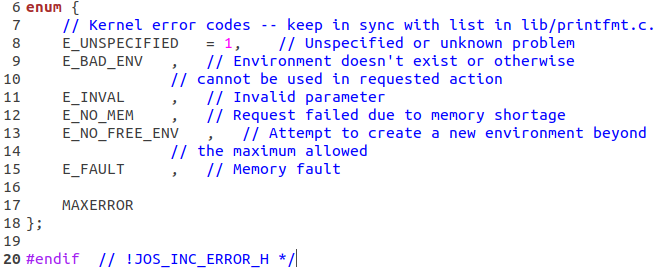


这里invlpg是tlb\_invalidate函数调用的子函数，其实就是一个X86汇编指令，能把与某个虚拟地址相关的在TLB快表中的页表项刷新（flush），估计就是把PTE\_P位置0或者全部置0.

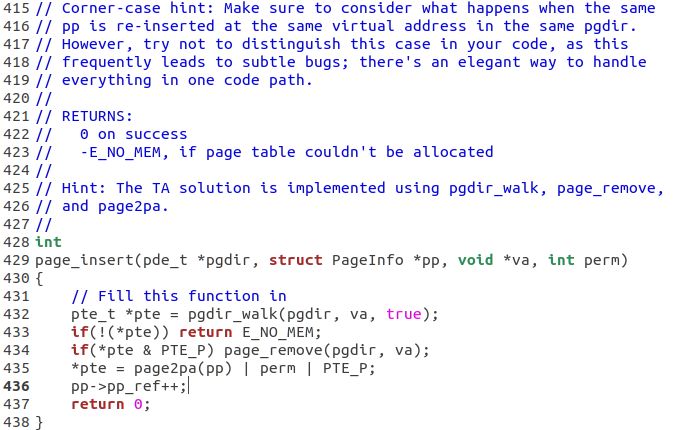


最后就是完成page\_insert函数了。

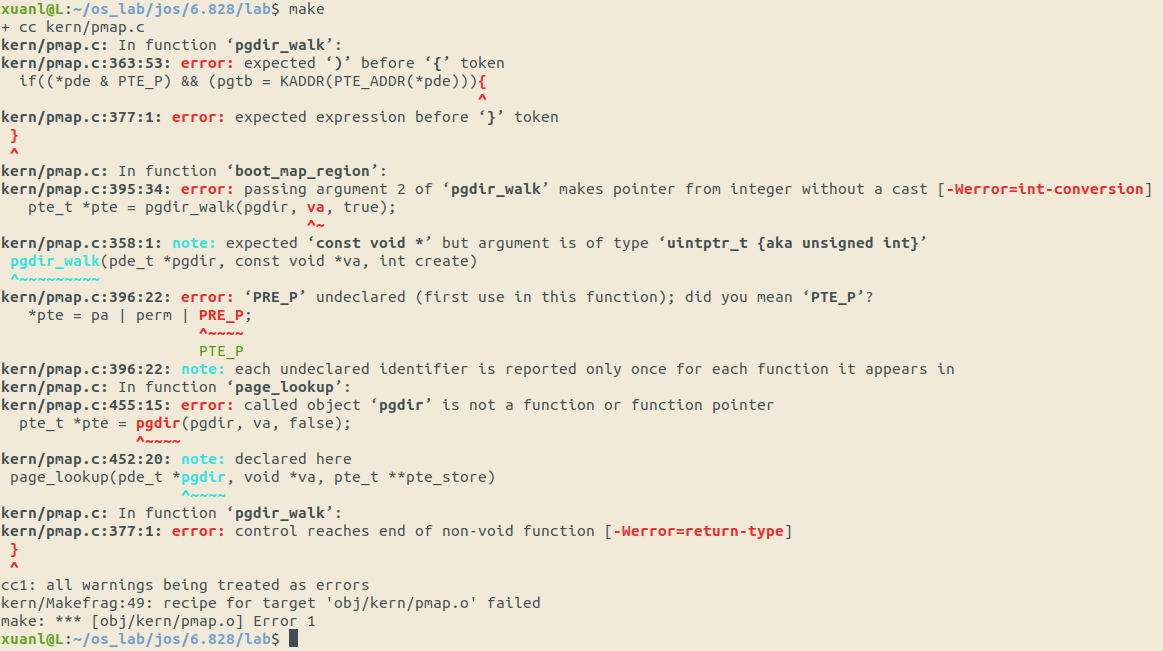
注意返回值中-E\_NO\_MEM是一个枚举类型，定义在error.h中，而且前面没有-（找了我好久）



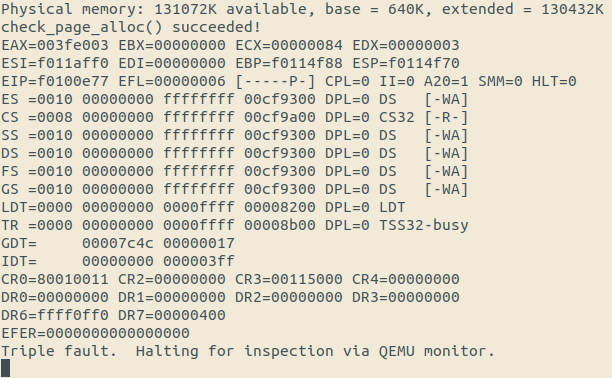
实现的比较简单，不知道行不行



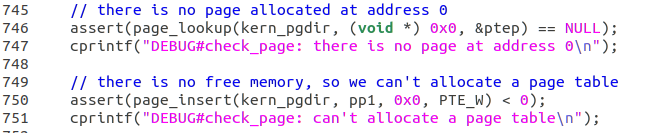
果然先是一堆报错



发现这个报错还挺人性化的，甚至会提醒PRE\_P应该是PTE\_P，这有点强啊。

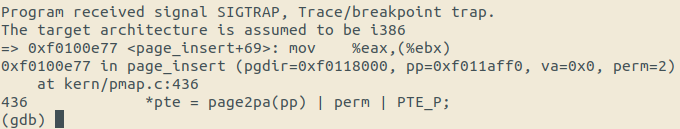


Triple fault！这下出大问题。



还是先用输出调试法找到出错位置，这里发现上一行能打印，下一行就不行了，所以问题出在page\_insert函数。

用GDB调试

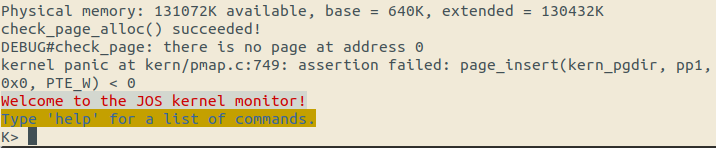


可以发现，本来这个时候调用page\_insert应该返回E\_NO\_MEM，也就是pgdir\_walk应该返回NULL.

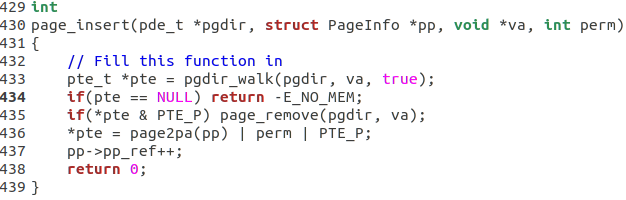


这里不应该判断\*pte，而应该是判断pte == NULL！

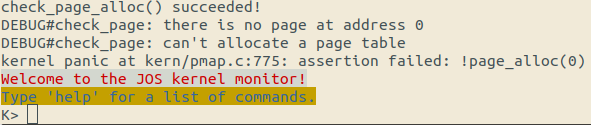
这下终于没有Triple fault了，但还是有问题



这里返回值应该是个负数，但是E\_NO\_MEM显然是个正数！当时前面有个-，其实是表示减号！结果不明所以地被忽略了。



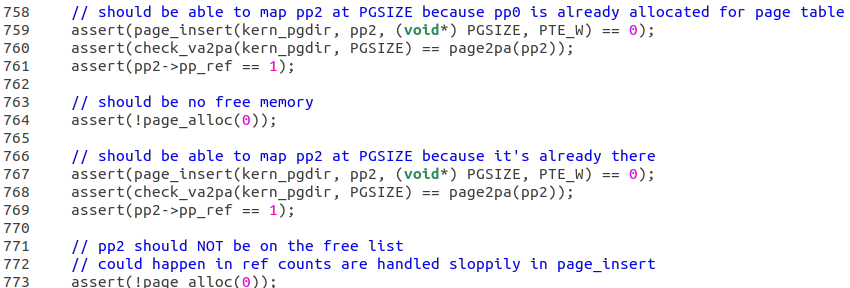
新问题



正如注释所说，对ref counts的处理比较马虎。



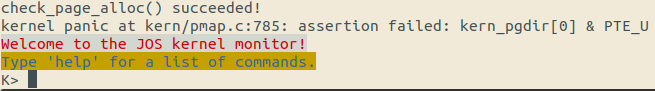
观察这里做的事情

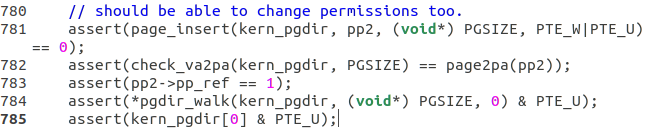


同一个物理块pp2被page\_insert了两次，因为第二次发现va已经被映射了，就会先page\_remove，然后pp2就进入了free\_list，因此之后pp\_ref明明>0，但是却在free\_list上。所以要额外判断一下pp是否在free\_list中？还是判断pp\_ref？

我认为，如果出现这样的情况，pp一定是刚刚进入free\_list，即在链表头，所以可以直接判断free\_list的头部是否为pp。

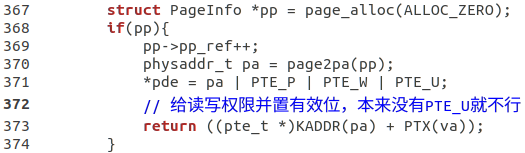
新问题





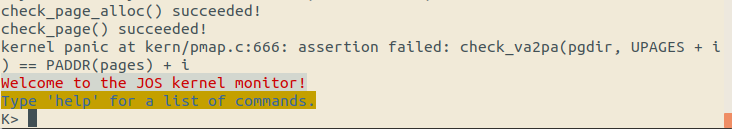
其实就是想要改变权限，但是只改变了二级页表项的权限，没有改变页表目录对应项的权限。

根据pgdir\_walk的注释，这里添加的权限还是多一些好了，因为目录项权限就不够的话，二级页表就不会看了。所以最后改权限可以都在二级页表上限制，而一级页表所有权限放开。



添加一个PTE\_U权限就行了

通过了check\_page测试！

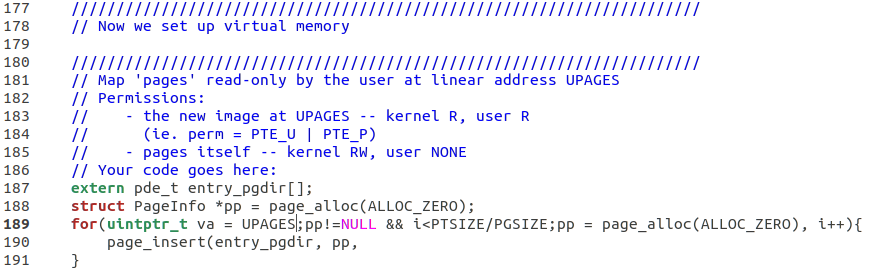


### Part3

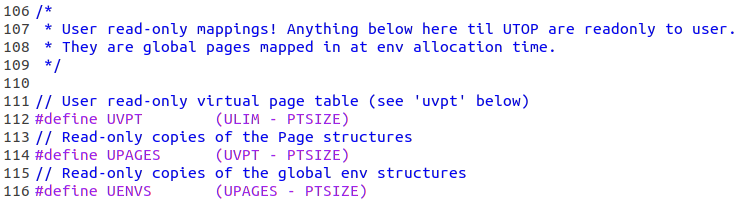
#### **Exercise 5**

这部分只需要完成mem\_init的剩余部分就可以了。

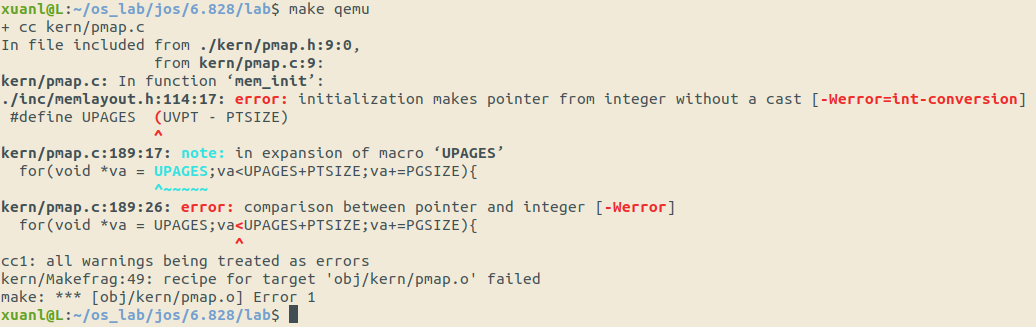
这里实在有些不明白，感觉照理来说UPAGES是在UTOP上的部分，按照注释应该用page\_map\_regioin进行区域映射。但这里实在不知道应该映射到哪一块物理内存区域。

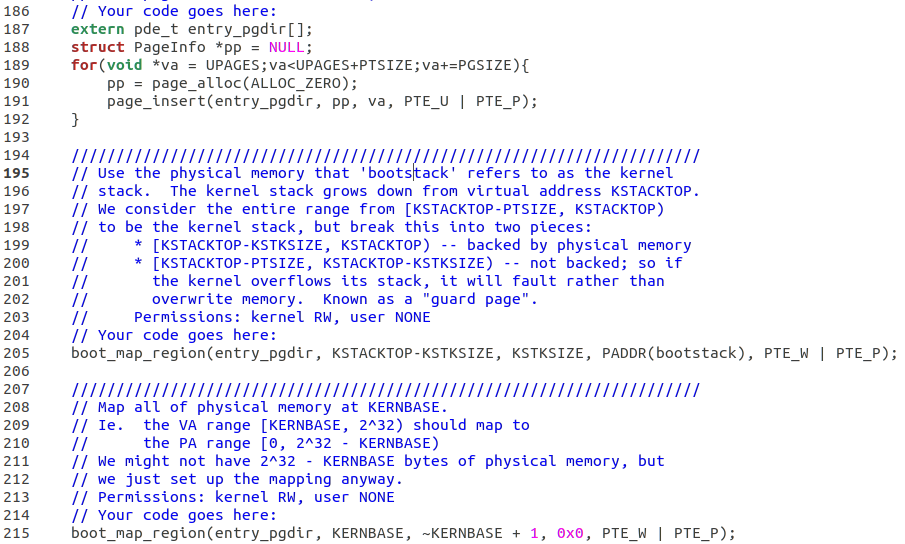


根据注释可以发现，UPAGES是page 数据结构的cpies，所以应该已经在物理内存中了？但还是不清楚是那一个部分。

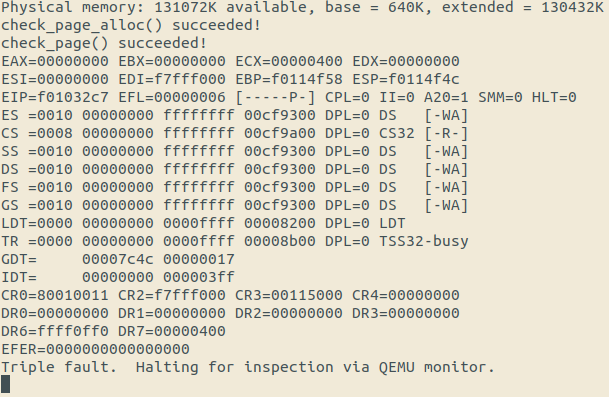


这一部分非常迷，先写一份代码测一下。



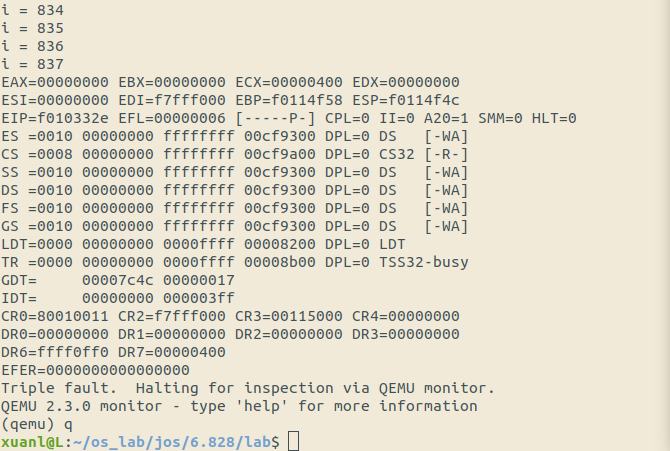


解决报错



Triple fault。

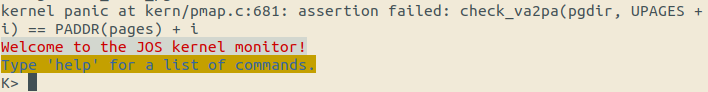
还是看一下测试程序，然后可能会明白具体要怎么做。



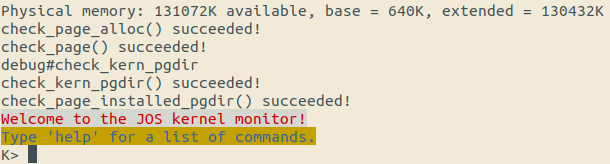
首先是应该用kern\_pgdir而不是用什么entry\_pgdir，那个是内核初始化的时候临时用的页表目录，现在要建立新的页表目录了。

其次检查循环次数发现，应该是次数不能太多，否则就会有问题。这里根本没法支持那么多。

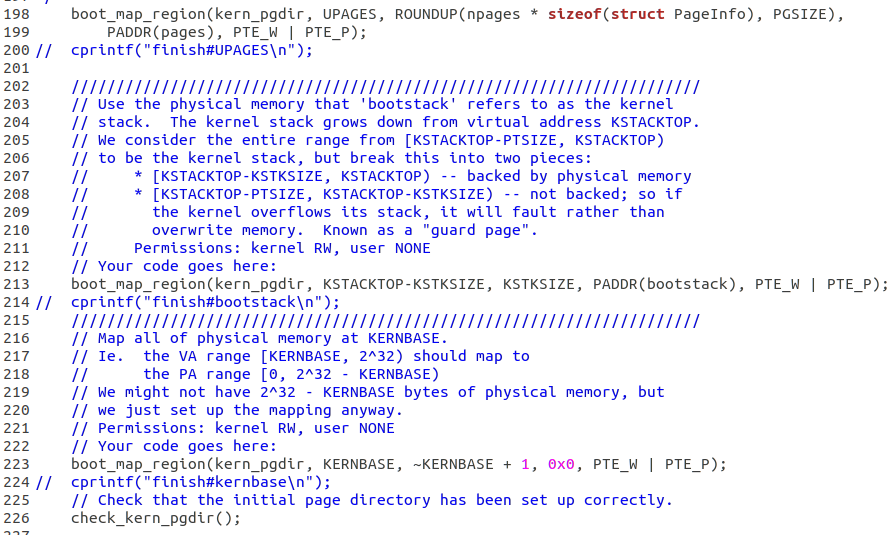
那只能先静态映射了。



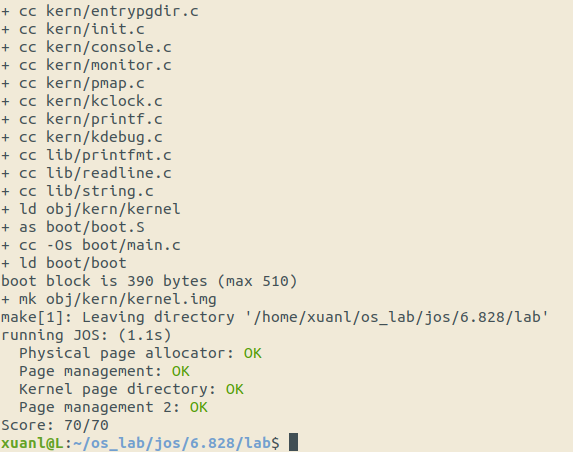
根据这个错误来看，应该映射到的就是pages数组那个位置。



改了之后就通过了。



其实就是简单的三个调用。。。



通过了所有测试，70/70。

#### **Question 2**

1. What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

|  |  |  |
| --- | --- | --- |
| Entry | Base Virtual Address | Points to (logically): |
| 1023 | ? | Page table for top 4MB of phys memory |
| 1022 | ? | ? |
| . | ? | ? |
| . | ? | ? |
| . | ? | ? |
| 2 | 0x00800000 | ? |
| 1 | 0x00400000 | ? |
| 0 | 0x00000000 | [see next question] |

首先这个问题，就是页表目录中哪些项能够确定。我感觉其实就是看mem\_init函数。



上面这行，是在所有初始化之前的，将UVPT这里的4MB虚拟空间映射到了kern\_pgdir所在的物理内存，且用户可读可写。UVPT的地址为0xef400000，所以其entry编号为0x3bd，十进制就是957。



接下来这里，将虚拟地址UPAGES映射到了pages数组的物理地址。注意到960是KERNBASE往上的第一块的编号，所以往下的内核栈就是959，内存I/O映射区就是958，当前页表是957，pages就是956，因此956项映射到pages数组所在的物理地址。



而KSTACKTOP这部分就应该是映射到bootstack对应的物理地址，即959项映射到PADDR(bootstack)。



最后就是960到1023这所有的内核部分的项，都映射到从地址0x0开始的256MB物理地址空间。

1. We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?

在223行的代码，将KERNBASE上面部分的虚拟地址映射到0x0开始的物理地址过程中，设置的权限位只有PTE\_W和PTE\_P，所以在对应的页表项中的权限位里PTE\_U这一项是0。因此，当用户态访问到内核区域的地址时，在进行转换的过程中会发现权限不够，因此无法对内核区的数据进行读写，从而保护了内核区的代码和数据。

1. What is the maximum amount of physical memory that this operating system can support? Why?

最大能支持4GB的物理内存，因为一级页表（页表目录）有1024项，每一项对应一个页表，每个页表有1024个页表项，每个页表项对应一页，一页有4KB内存，因此一共能映射到的物理内存最多有1K×1K×4KB=4GB

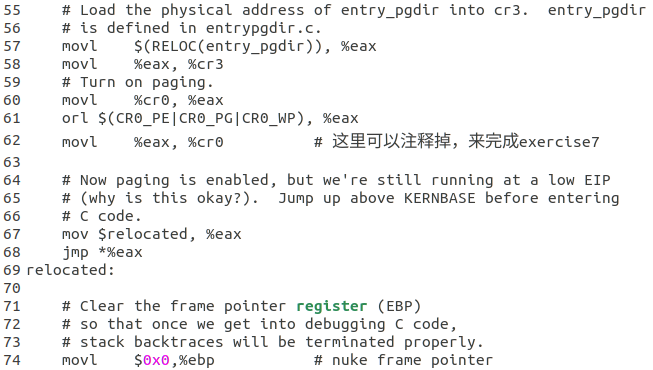
1. How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?

这里”overhead“表示”开销“的意思，所以这里是问内存管理所需的额外空间开销。一个页目录（即一级页表，这应该是x86手册对一级页表的术语）加上1024个页表，供1025\*4KB，即4100KB的空间开销来管理。当然，如果物理内存没有4GB，那这肯定可以减小一部分开销。此外要注意到，如果多个虚拟地址被映射到同一块物理块上，那么也可能减小这里的空间开销，因为可以让页目录的多个目录项指向同一个二级页表，从而减小二级页表的数量。如果有些虚拟地址空间没有进行映射，那相应的页表也可以省去，进一步节省空间。反之，如果使用一个很大的一级页表的话，就不得不把所有的表项都考虑到，必须占用最大可能的空间。

再进一步可以知道，三级页表、四级页表也可以进一步节省空间开销，但对系统的复杂性提升了很多。多级页表最大的优势就在于，大部分进程运行时不需要将整个虚拟内存空间全部映射，而只需要映射很小的一部分（代码段、数据段等），所以二级页表中的第二级就可以只开辟很小的一部分空间来管理内存，而一级页表则对于每个进程都需要开辟4MB的内存空间来进行管理，这样额外的空间开销就很大。对于二级页表，其最大可能的开销是4MB+4KB，还多了一点，但大部分时候都远小于4MB，就在于二级页表在不需要时可以不用开辟。

1. Revisit the page table setup in kern/entry.S and kern/entrypgdir.c. Immediately after we turn on paging, EIP is still a low number (a little over 1MB). At what point do we transition to running at an EIP above KERNBASE? What makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE? Why is this transition necessary?

这个问题其实在上一个实验里面就弄清楚了。



这里68行jmp \*%eax的时候，就跳转到高地址了。那之前为什么都是低地址呢？因为在kernel.ld中设置了代码执行从\_start开始，而entry.S中又设置了\_start = RELOC(entry)，所以\_start就是高地址entry对应的低地址。

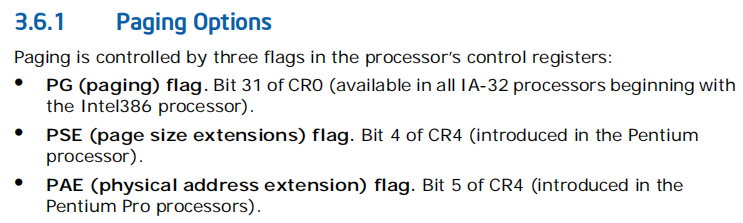
在开启页表之后，这个临时页表对两个区域都进行了映射，其一是虚拟地址KERNBASE向上4MB的部分，映射到物理地址0x0开始的4MB，另外还将虚拟地址0x0也映射到物理地址0x0开始的4MB，因此在建立页表之后，运行在低地址和高地址都不会出问题，因为它们映射之后的物理地址是相同的。

如果不进行这个简单的映射，那么后面内核代码中所有与地址有关的指令都无法正常执行。而在建立后面的二级页表之前还有很多工作要完成，所以必须先建立一个简单的映射。

#### **Challenge 1**

We consumed many physical pages to hold the page tables for the KERNBASE mapping. Do a more space-efficient job using the PTE\_PS ("Page Size") bit in the page directory entries. This bit was not supported in the original 80386, but is supported on more recent x86 processors. You will therefore have to refer to [Volume 3 of the current Intel manuals](https://pdos.csail.mit.edu/6.828/2016/readings/ia32/IA32-3A.pdf). Make sure you design the kernel to use this optimization only on processors that support it!

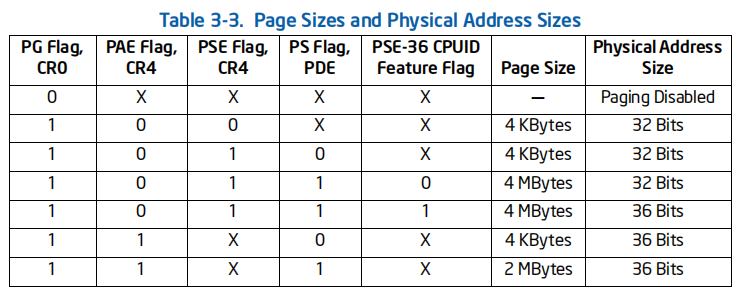
现在对于KERNBASE网上的256MB虚拟内存空间，使用了64个二级页表，即二级页表的空间开销为256KB。如果把页大小调整得更大，那么所需的页表就可以更少了。



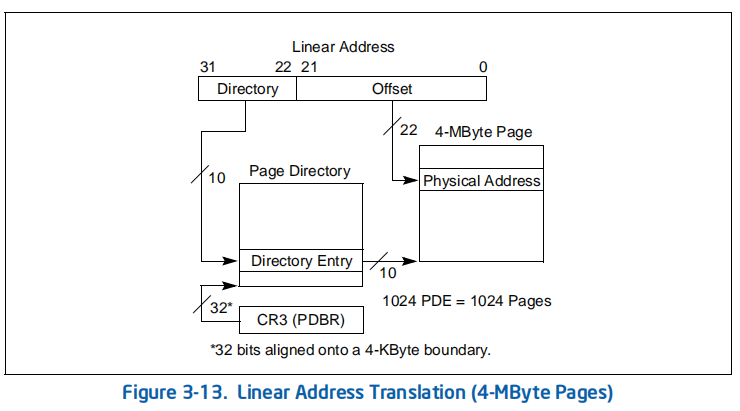
从奔腾处理器架构开始，就在寄存器cr4中引入了PSE标志位，用于修改页大小，可以修改2MB,4MB这两种。

感觉网页上应该是写错了，这里应该是PSE和PS这两位来控制页面的大小。

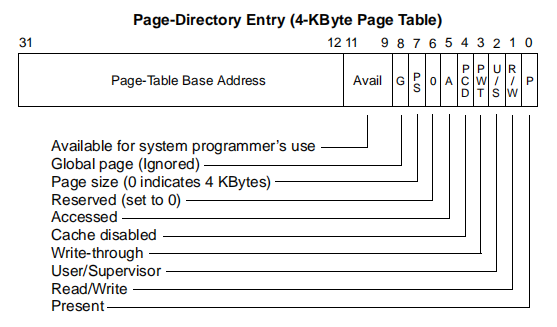
CR4中的PSE标志位置位之后，如果页目录项PDE的PS标志位也置位，则页大小为4MB，否则还是4KB，从而可以实现两者混合分页。

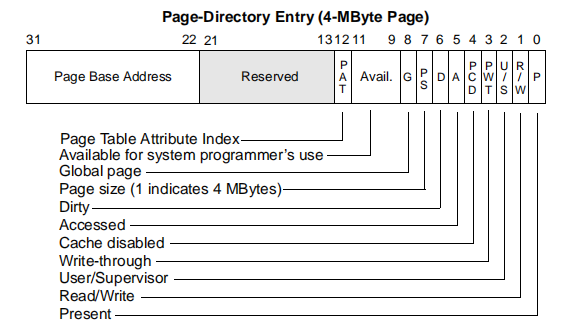


当使用4MB作为页大小时，相当于只用了一级页表，示意图如下

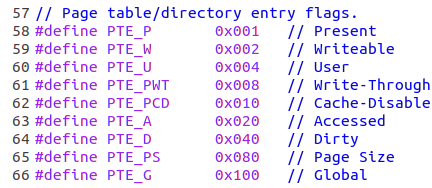


那大概明白了，就是在cr4里面把PSE置位，然后在kern\_pgdir里面最顶上的64项把PS置位即可。

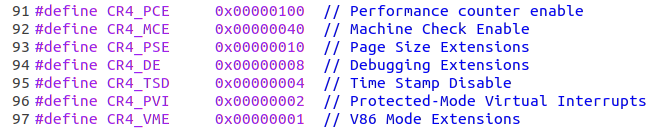




PS是页目录项的第8个bit位（7号位）。

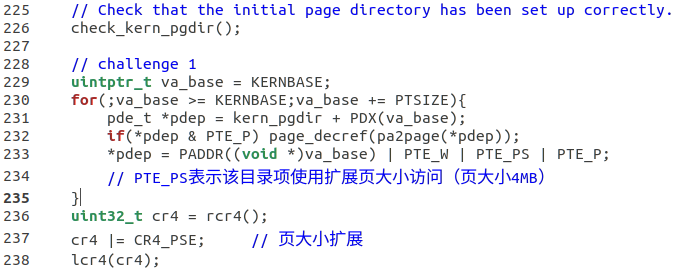


在头文件里面早就已经定义好了PTE\_PS宏

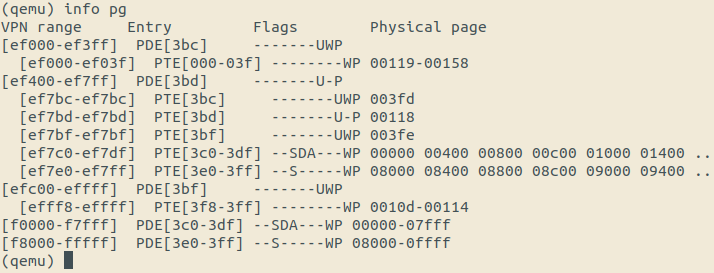


Cr4的相关的宏也都有。

直接改那部分的话，就通不过测试。因为check\_va2pa函数永远是按照二级页表来检查的。要么放在后面部分。



这里先把之前新建的多余页表释放掉（page\_decref），然后直接将物理地址填在页目录的页框内，并置PTE\_PS位。顺利运行，但要想办法确定现在确实和原来不同。



改了之后是上面这样。

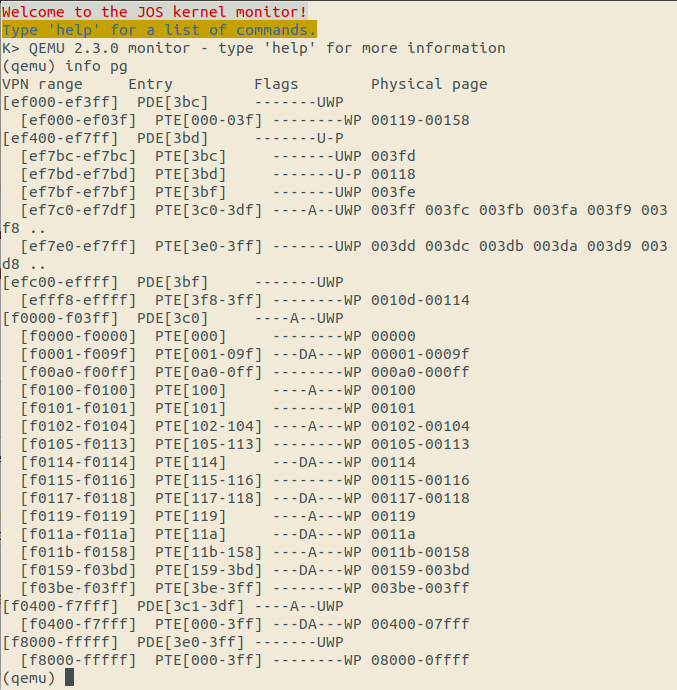
下面把新加的注释掉，再试一次。

可见原来不使用页扩展的话，占用的页数要多很多。因此，challenge 1顺利完成了。

这里的VPN应该是vitural page number的意思，所以要转成虚拟地址的话，后面要补上3位。例如，ef000-ef3ff，其实表示的是0xef000000-0xef3fffff，其中后面要补的是f而不是0！

这里中间的权限位的符号含义，可以对照上面x86手册的pde和pte的截图。上面的权限位里出现了符号’S‘，这就是表示PS位置1。D表示Dirty位，A表示Accessed位。

在上图可以看到，含有S符号的PDE是与物理地址范围直接对应。但也可以看到还有含S的PTE，这就不清楚是什么意思了。按照手册上，PTE的第8位表示PAT。在challenge2中知道，这些S其实就是PS位，因为这些PTE其实就是PDE！只不过这一块区域存放的就是页表，所以PDE中这一项指向的就是自身！经过这一层指针解引用之后，页目录本身就降级为了页表，但其内容都没有变。



总之，内核能正常运行，查看页表信息也能找到PDE与物理地址的对应关系，因此该challenge完成。

#### **Challenge 2**

Challenge! Extend the JOS kernel monitor with commands to:

Display in a useful and easy-to-read format all of the physical page mappings (or lack thereof) that apply to a particular range of virtual/linear addresses in the currently active address space. For example, you might enter 'showmappings 0x3000 0x5000' to display the physical page mappings and corresponding permission bits that apply to the pages at virtual addresses 0x3000, 0x4000, and 0x5000.

Explicitly set, clear, or change the permissions of any mapping in the current address space.

Dump the contents of a range of memory given either a virtual or physical address range. Be sure the dump code behaves correctly when the range extends across page boundaries!

Do anything else that you think might be useful later for debugging the kernel. (There's a good chance it will be!)

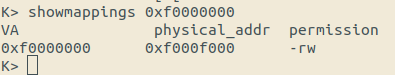
在控制台添加一些指令。

先添加一个showmappings吧。其他的可以之后有时间再加上。

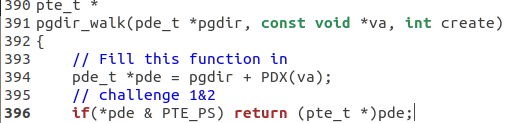
注意要在monitor.h中添加mon\_showmap函数声明，否则在定义之前在commands[]数组里面就会用到这个函数的地址，就会编译报错。

需要使用pmap.h头文件里声明的函数。

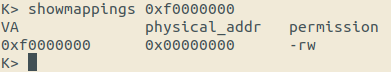
这里检查0xf0000000发现结果不对。其实就是pgdir\_walk里没有考虑页目录的SE位导致的。



加上额外的判断



这样showmappigns命令现在可以处理一个参数的情况了。

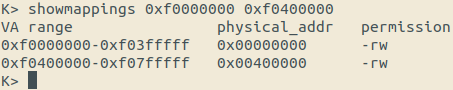


接下来考虑处理两个参数。

注意ROUNDUP和ROUNDDOWN是在type.h头文件里定义的宏。

这里碰到了不少问题，主要在于challenge 1中修改了分页的方式，使得需要额外判断当前虚拟地址对应的页表项是否有页大小扩展，即当前对应的目录项的页大小是多少。

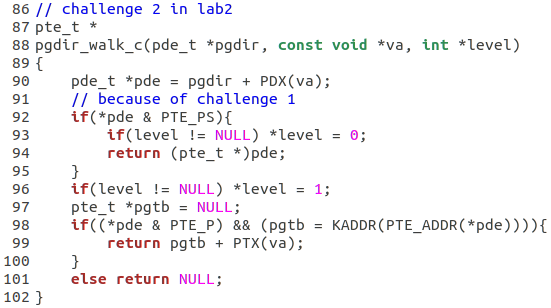
如果页大小为4MB，则按照4MB为单位显示，如下所示。



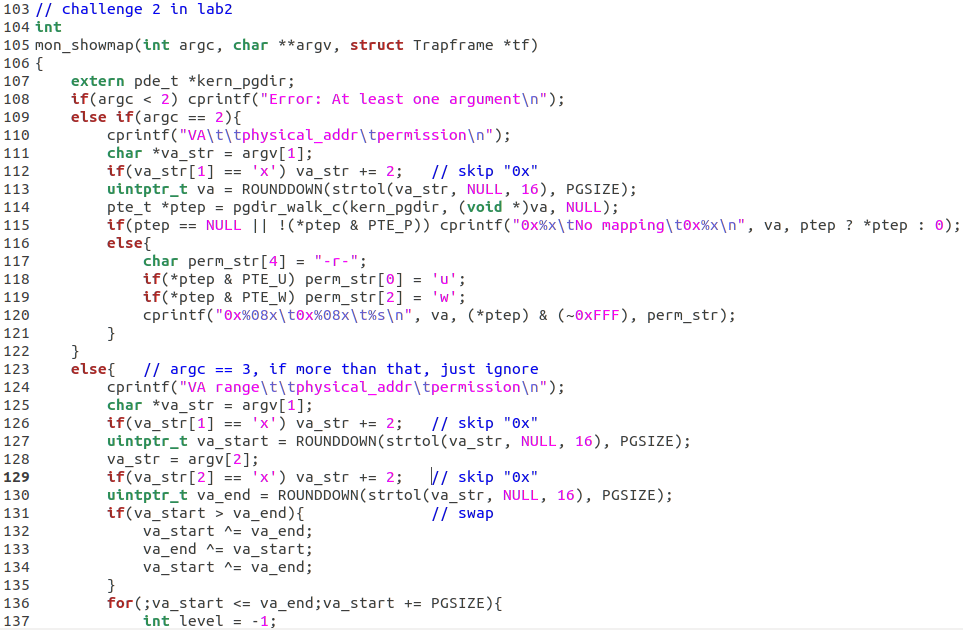
然后对0xef800000下面的部分测试的时候发现，在Page Table部分存放的都是页表的信息，其中甚至就有页目录自身！所以页目录自身也是一个页表，其中有一项就指向自身。这其实也就解释了为什么上面有些PTE也是S！

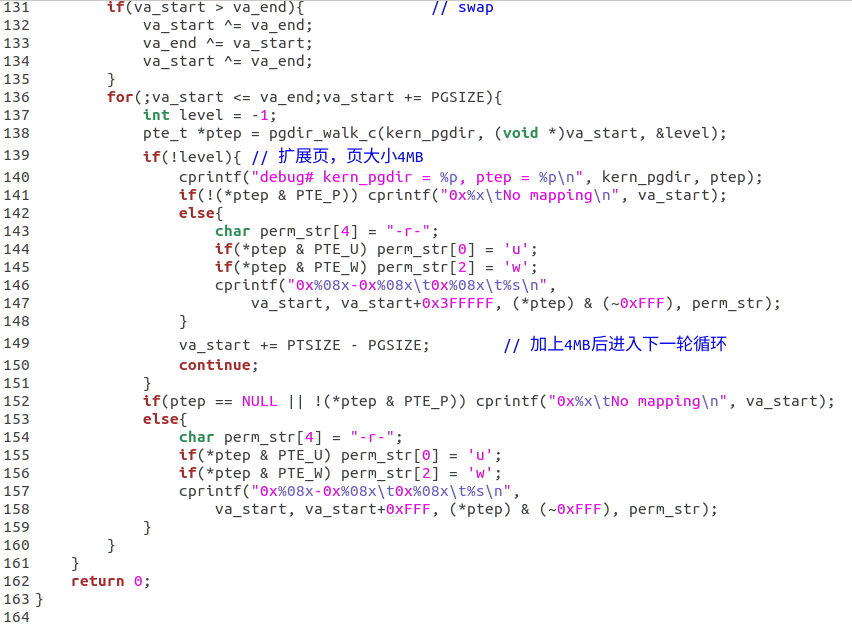


重新写了一个新的page\_walk\_c，使用一个额外的参数表示页表的级数level。



这样就可以了。





总之，先完成了这个challenge2的第一个命令，感觉还是挺艰难的啊。

#### **Challenge 3**

#### **Challenge 4**