Lab2

此次实验主要需要编写操作系统的内存管理部分。内存管理分为两部分：

1. 内核的物理内存分配器

使得内核可以分配、释放内存。该分配器以页为单位，jos中页一页是4kB。Lab2的任务是维护一个数据结构，该数据结构记录了物理内存的分配与释放，以及多少个进程正在共享各个已分配的页

1. 虚拟内存

将内核和用户程序使用的虚拟地址映射到物理内存的地址中。X86的内存管理单元会在指令拥戴内存时完成这个映射，查询一系列页表。

在lab2中，新加入了几个源文件：

memlayout.h描述了虚拟地址空间的布局，你需要通过修改pmap.c来实现它们。memlayout.h和pmap.h定义了一个PageInfo结构，用于描述物理内存页面是否可用。kclock.c和kclock.h用于管理PC的时钟（这个时钟是指用电池供电的系统常驻时钟）和CMOS RAM硬件（BIOS在其中记录PC包含的物理内存量，以及其他一些信息）。pmap.c中的代码需要读取设备硬件，以便了解设备上有多少物理内存

# Part 1 物理页面管理

操作系统必须知道物理RAM的哪些部分是空闲的，哪些部分正在被使用。JOS以页（page）为单位管理PC的物理内存，从而可以使用MMU映射和保护每一片分配的内存。

你现在将完成物理页分配器。它用于保存哪些页面是空闲的，在数据的组织上，由struct PageInfo对象构成的链表，每个PageInfo结构都对应一个物理页。在完成其余的虚拟内存实验之前，你需要编写物理页分配器，因为你的页表管理代码将需要分配存储页表的物理内存。

## **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()（仅完成调用check\_page\_free\_list(1) 之前的部分）

page\_init()

page\_alloc()

page\_free()

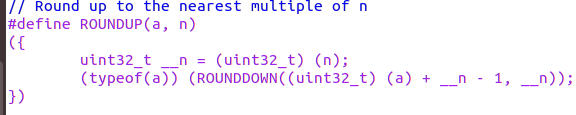
check\_page\_free\_list()和check\_page\_alloc()会测试你的物理页面分配器。你应该启动JOS并查看check\_page\_alloc() 是否报告成功。修改代码，并确定能够通过它的测试。你可以添加自己的assert()来验证你的假设是否正确。

### (1)boot\_alloc()

该函数维护一个static的指针nextfree，初始值是end，end是定义在/kern/kernel.ld中定义的符号，位于bss段的末尾，end 指向的是第一个未使用的虚拟地址

。也就是说从内核的末尾开始分配物理内存

inc/types.h



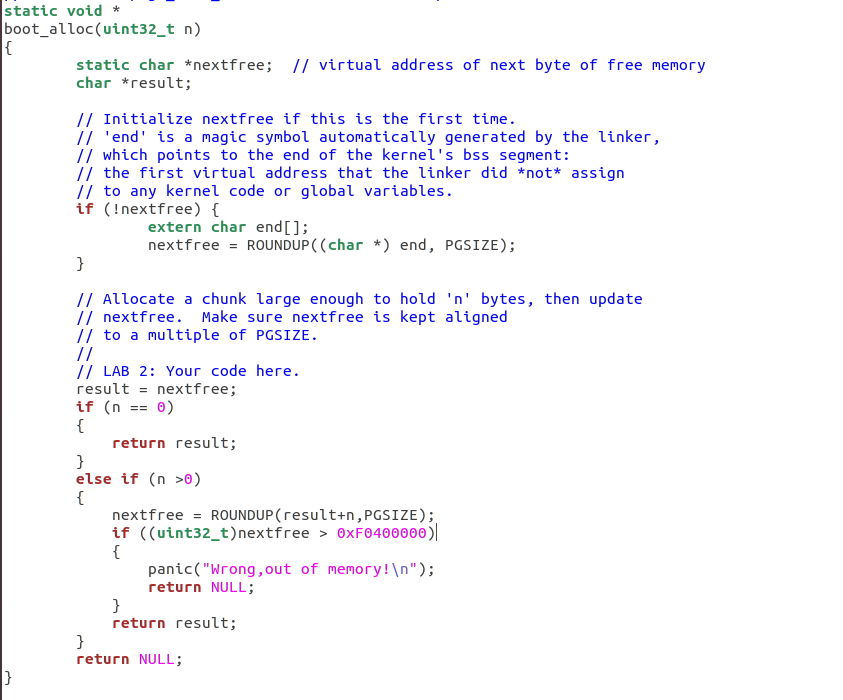
PGSIZE是４KB

inc/mmu.h



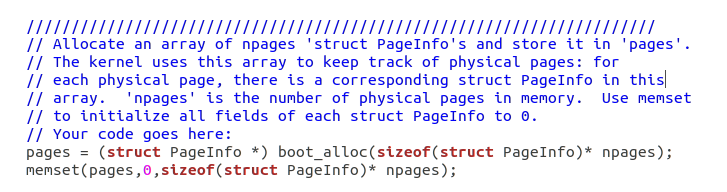
需要注意的是分配内存用的是 boot\_alloc。这是一个仅用于 JOS 设置自身虚拟内存系统时使用的物理内存分配器，仅用于 mem\_init 函数

每次调用都返回nextfree，然后根据参数n更新nextfree的值，使其指向下一个空闲地址处。



### (2)mem\_init()

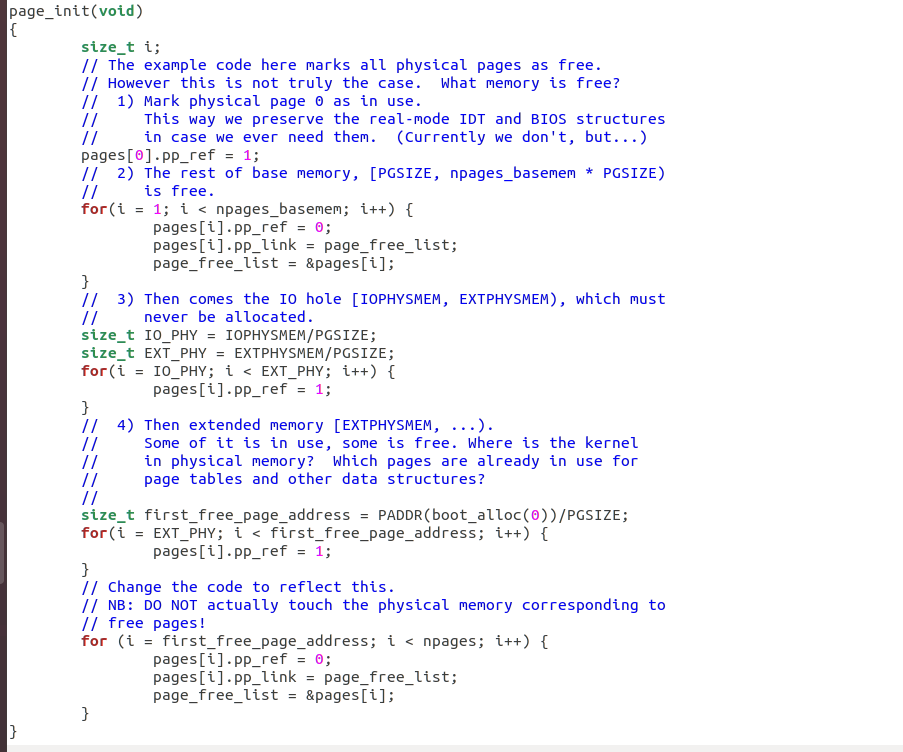
mem\_init()调用boot\_alloc()，将返回值赋给全局变量kern\_pgdir，kern\_pgdir保存的是内核页目录的物理地址



这段代码分配足够的的内存空间保存pages数组，pages数组的每一项是一个PageInfo结构，对应一个物理页的信息

### (3)page\_init()

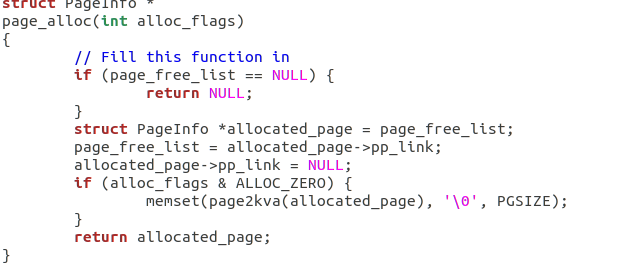
初始化页表



这个函数的主要作用是初始化之前分配的pages数组，并且构建一个PageInfo链表，保存空闲的物理页，表头是全局变量page\_free\_list

### (4)page\_alloc()

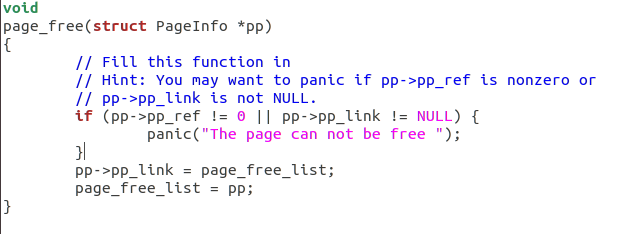
分配页面



该函数的作用是：从page\_free\_list指向的链表中取一个PageInfo结构返回，根据参数alloc\_flags决定要不要将这块内存初始化为0。需要注意的是，不需要增加PageInfo的pp\_ref字段。

### (5)page\_free

释放页面



该函数重新将参数pp对应的物理页设置为空闲状态。

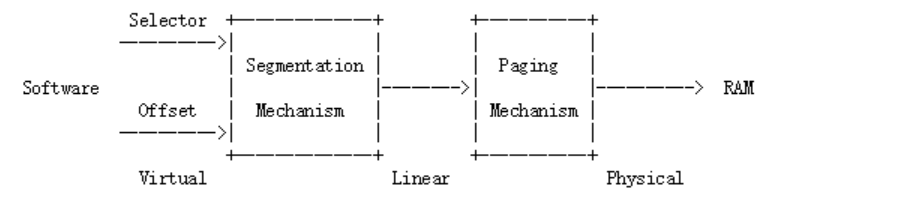
# Part2：虚拟内存

## 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

虚拟地址，线性地址和物理地址

在x86术语中，虚拟地址 由段选择器和段内的偏移组成。线性地址是在段转换之后，页转换之前的地址。物理地址是你在段和页转换之后最终获得的最终地址，也就是最终在硬件总线上最后出现在RAM中的物理地址。



从上图中可以看出来，我们编 程时使用的C语言指针用的是虚拟地址来描述的。在boot/boot.S中，我们加载了一个全局描述符表（GDT），通过将所有段基地址设置为0并将其有效地址上限设置为0xffffffff，这覆盖了整个32位的地址空间。因此，“段选择器”是不起作用的，线性地址总是等于虚拟地址。在实验3中，我们会更多地使用分段机制以设置权限级别，但是对于内存转换，我们可以在JOS实验中忽略分段机制，并且只关注页转换。

回想一下，在实验1的第3部分中，我们加载了一个简单的页表，使得内核可以在0xf0100000的链接地址运行，即使它实际上是加载在ROM BIOS的0x00100000上方的物理内存中。此页表仅映射了4MB内存。在本实验中，我们将要为整个JOS设置虚拟内存布局，映射前256MB物理内存到虚拟地址0xf0000000处，并映射虚拟内存的其他区域。

## 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，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）

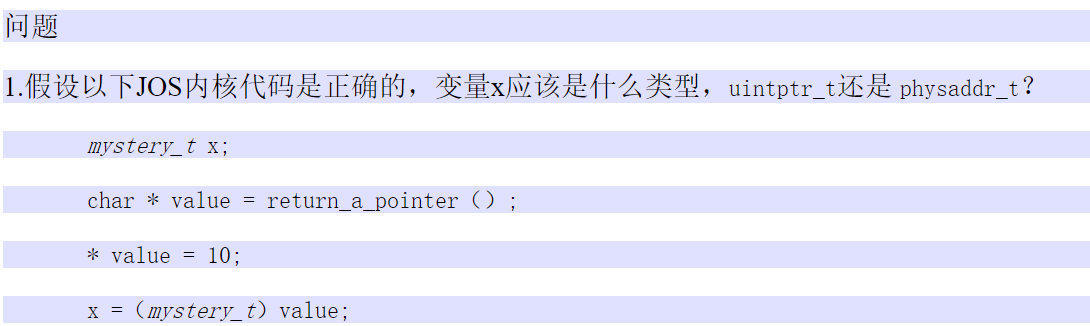
使用QEMU监视器中xp命令和GDB中的的x命令查看相应物理地址和虚拟地址上内存的内容，并确保看到相同的数据。

我们修补的QEMU版本提供了一个info pg 的命令：它显示了当前页表的详细的表示，包括所有映射的内存范围，权限和标志。QEMU中原来还提供了一个info mem命令，显示虚拟内存的映射范围以及配置了哪些权限。

从CPU上执行的代码，一旦我们处于保护模式（也就是我们在进入boot/boot.S中做的第一件事），就没有办法直接使用线性或物理地址。所有存储器引用被理解为虚拟地址并由MMU转换，这意味着C中的所有指针都是虚拟地址。

JOS 内核中操作的地址变量，经常是一些整数值或者常数，有的地址是不可见的虚拟地址，有的是物理地址，它们不来自某个变量的取地址操作，内核也不会按地址引用它们，例如在物理内存分配器中。为了帮助解释这部分的代码，JOS 源区别了两种情况：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（uint32\_t），因此编译器不会阻止你将一个类型赋值为另一个类型！由于他们是整形并非指针，如果你试图利用他们取值（\*运算符的功能），编译器将会报错。

JOS 内核可以通过将 uintptr\_t 类型转换赋值给一个指针类型，然后再对它进行取值操作。相较而言，内核不能简单的对一个物理地址进行操作， MMU 会对所有的内存操作进行地址转换。如果你一定要将一个 physaddr\_t 转换一个指针上并对它取值，你可能会可以加载和存储到一个另外的地址上（硬件会把它当作一个虚拟地址处理），但你将不会获得你想要的内存位置。



**参考计数**

在将来的实验中，您将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（UTOP上的映射大多由内核在启动时设置，不应该被释放，所以不需要为它们计算引用计数）。我们还将使用这个变量来保存指向页目录表所在页的指针的数量，以及页目录表对页表所在页的引用数量。

使用page\_alloc时要小心。 它返回的页面的引用计数为0，所以pp\_ref应该随着返回的页面完成某事（如将其插入到页面表中）而递增。 有时这是由其他功能（例如page\_insert）处理的，有时调用page\_alloc的函数必须直接执行

**页表管理**

## 练习4.

在文件kern / pmap.c中，必须实现以下函数的代码。

pgdir\_walk（）

boot\_map\_region（）

page\_lookup（）

page\_remove（）

page\_insert（）

check\_page()（由mem\_init()调用）可以用于测试你的页表管理程序。你应该确保测试报告成功之后再继续后面的实验。

### (1)pgdir\_walk()

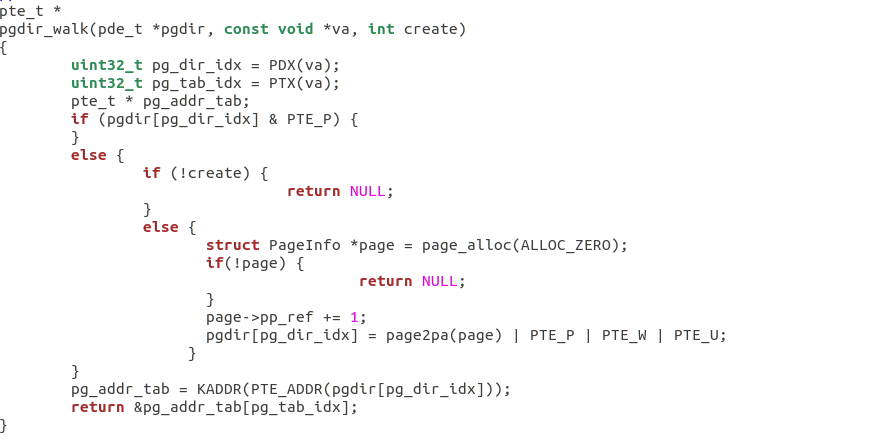
参数：

pgdir:页目录虚拟地址

va:虚拟地址

create:布尔值

返回值：页表条目的地址

作用：给定pgdir，指向一个页目录，该函数返回一个指针指向虚拟地址va对应的页表条目(PTE)。

### (2)boot\_map\_region()

参数：

pgdir:页目录指针

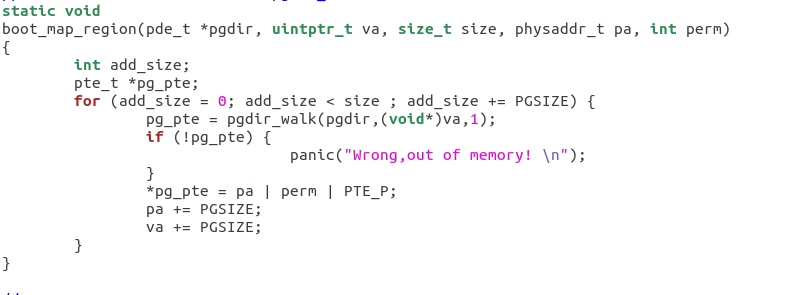
va:虚拟地址

size:大小

pa:物理地址

perm:权限

作用：通过修改pgdir指向的树，将[va, va+size)对应的虚拟地址空间映射到物理地址空间[pa, pa+size)。va和pa都是页对齐的。



### (3)page\_insert()

参数：

pgdir:页目录指针

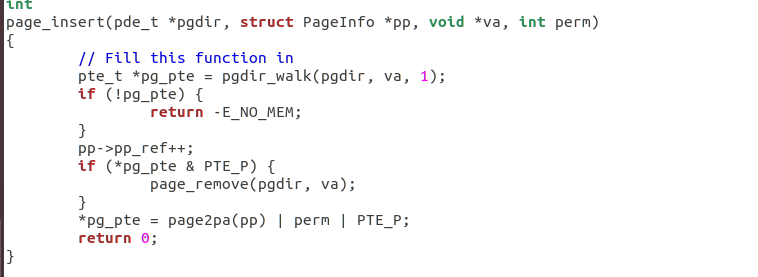
pp:PageInfo结构指针，代表一个物理页

va:线性地址

perm：权限

返回值：0代表成功，-E\_NO\_MEM代表物理空间不足。

作用：修改pgdir对应的树结构，使va映射到pp对应的物理页处。

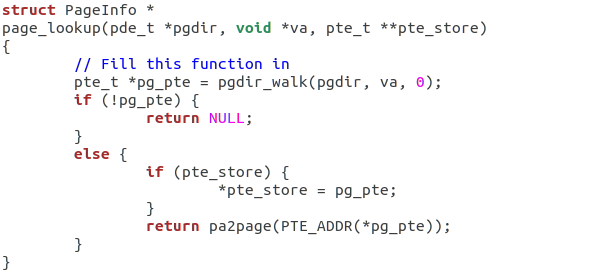


### (4)page\_lookup()

参数：

1. pgdir:页目录地址
2. va:虚拟地址
3. pte\_store:一个指针类型，指向pte\_t \*类型的变量

返回值：PageInfo\*  
作用：通过查找pgdir指向的树结构，返回va对应的PTE所指向的物理地址对应的PageInfo结构地址



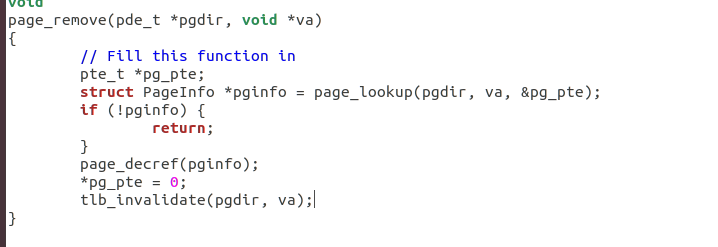
### (5)page\_remove()

参数：

pgdir:页目录地址

va:虚拟地址

作用：修改pgdir指向的树结构，解除va的映射关系。



# Part3:内核地址空间

JOS将32位的线性地址划分成两部分。下半部分(低地址段)供用户程序使用（我们将在实验3中开始设计加载和运行的用户环境（进程）的部分，用户环境是JOS中引入的概念，可以基本与课程中讲到的进程等同），而上半部分（高地址段）由内核始终保持对其完全控制。分隔线由inc/memlayout.h中的符号ULIM定义，为内核保留大约256MB的虚拟地址空间。这就解释了为什么我们需要在实验室1中给内核提供如此高的链接地址：否则在内核的虚拟地址空间中没有足够的地方用于映射同时运行的用户环境（进程）。

你会发现对于本次实验以及以后的实验中，你需要经常参考inc/memlayout.h的内存布局图。

**权限和故障隔离**

由于在每个用户环境的地址空间中，同时存在内核和用户内存，所以我们必须使用X86页表中的权限位，允许用户代码仅能进入到用户部分的地址空间内。否则在用户代码中的bug会覆盖掉内核的代码，这样会引起崩溃或者更为恶劣的故障；用户的代码也有可能盗取其它环境中的私有数据。

用户对任何超过ULIM以上的地址空间是没有任何权限的，而内核可以对这一部分空间进行读写操作。对于在[UTOP,ULIM]范围内的地址，用户和内核有这相同的权限：可以读但是不能对这一部分进行写操作。这一地址范围用于将内核中的一些数据结构以只读的方式对用户公开。最后，UTOP以下的地址空间是为用户部分使用的，设置这一部分空间的权限位以使用户环境可以自由读写它们。

**初始化内核地址空间**

现在，你要设置地址空间在UTOP以上的部分：内核部分的地址空间。inc/memlayout.h 文件中展示了我们使用的内存地址空间的布局，你将使用你写的函数去建立线性地址到物理地址映射，以建立这个布局。

## 练习5.

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

