

XV6 源码第三次阅读报告

虚存管理部分

XV6 源代码阅读——同步机制与虚存管理

一、阅读代码:

内存管理部分: kalloc.c vm.c 以及相关其他 件代码

**（一） vm.c**

**seginit( )：**

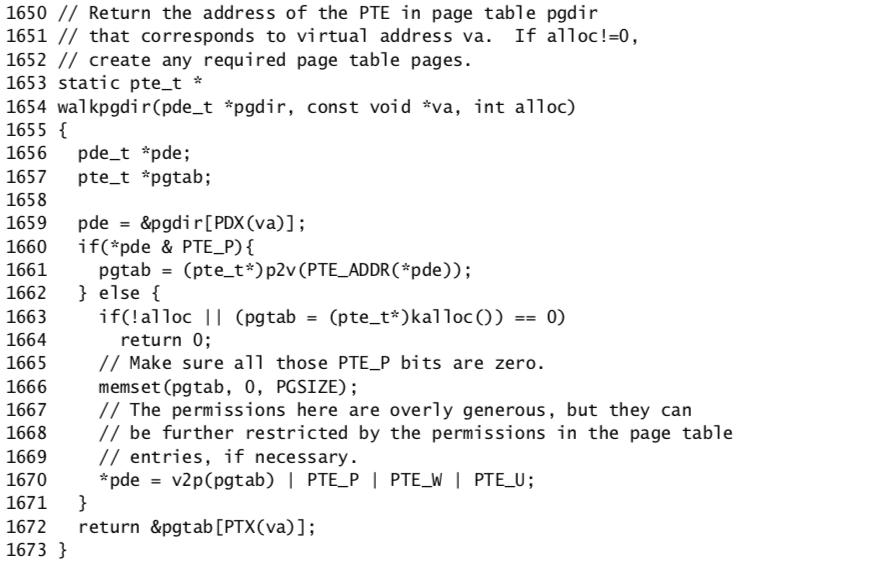
* main.c 中调用这函数重新设置全局描述符表，并补充未设置的内容，Task State Segment 会在第一个用户进程被创建时设置（在 switchuvm( )中）

**walkpgdir( )**

PTE\_P 表 PTE 是否陈列在页表中：如果不是，那么一个对该页的引用会引发错误（也就是：不允许被使用）

walkpgdir（1654）模仿 x86 的分页硬件为一个虚拟地址寻找 PTE 的过程（见图

* 2-1）。walkpgdir 通过虚拟地址的前 10 位来找到在页目录中的对应条目（1659），如果该条目不存在，说明要找的页表页尚未分配；如果 alloc 参数被设置了， walkpgdir 会分配页表页并将其物理地址放到页目录中。最后用虚拟地址的接下来 10 位来找到其在页表中的 PTE 地址（1672）。



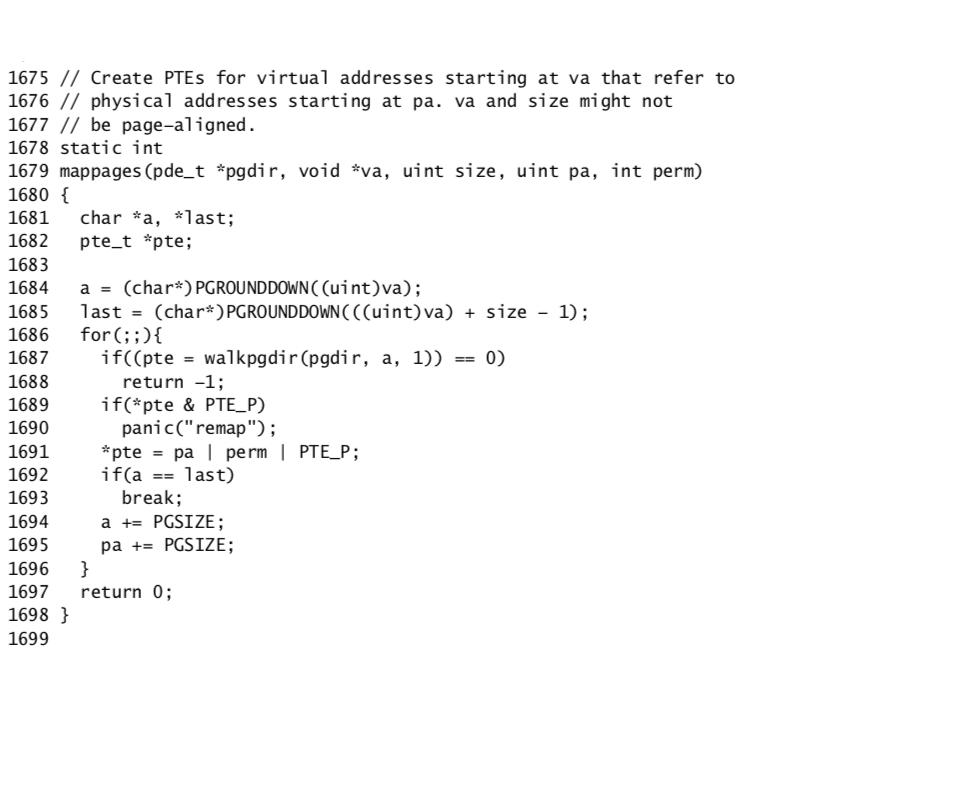
2

**mappages( ):**

mappages（1679）做的工作是在页表中建立一段虚拟内存到一段物理内存的映射。它是在页的级别，即一页一页地建立映射的。对于每一个待映射虚拟地址，

mappages 调用 walkpgdir 来找到该地址对应的 PTE 地址。然后初始化该 PTE 以保存对应物理页号、许可级别（PTE\_W 和/或 PTE\_U）以及 PTE\_P 位来标记该 PTE 是否是有效的（1691）。

一个 PTE 只能指向一个 4096 字节对齐的物理地址（即是 4096 的倍数），因此 PGROUNDOWNP 来保证物理地址的对齐。

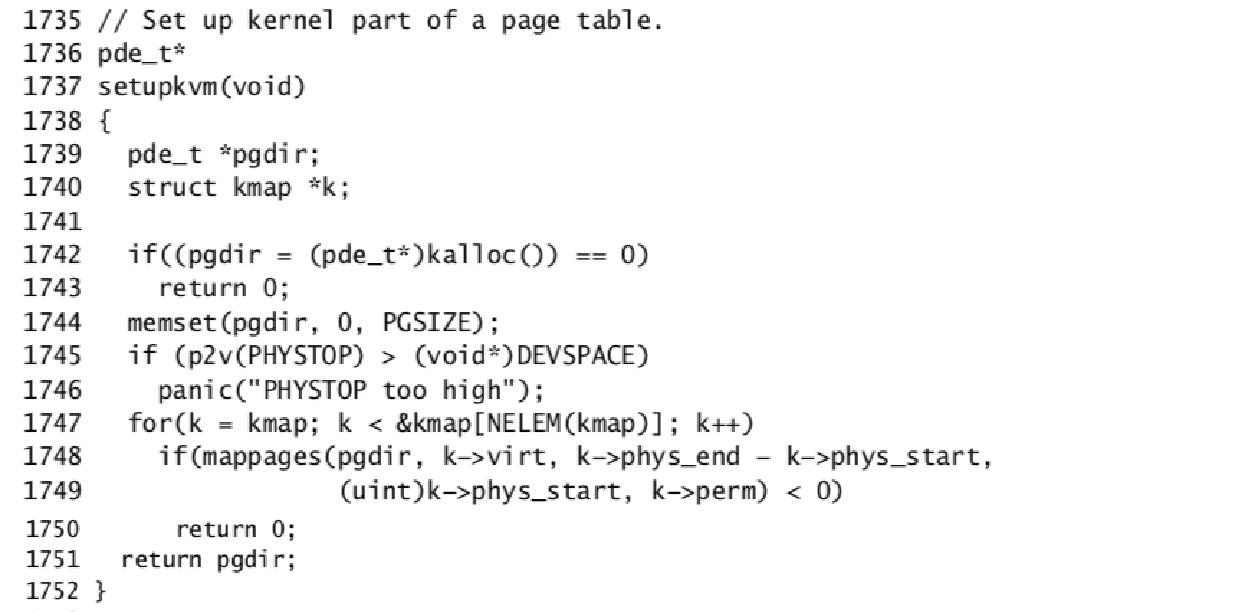
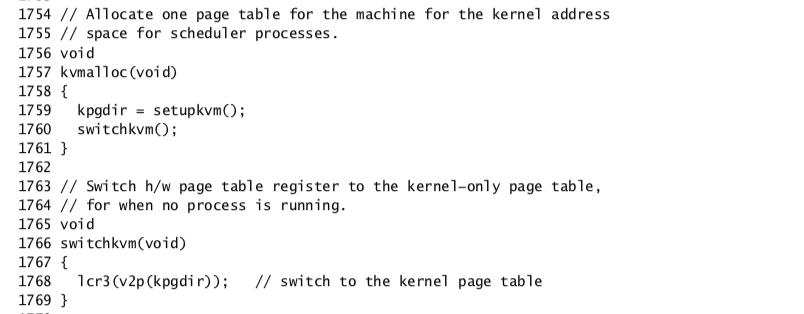


main 调 **kvmalloc**（1757），创建并切换到一个拥有内核运算所需

* KERNBASE 以上映射的页表。这里的大多数工作都是由 **setupkvm**（1737）完成

的。首先，它会分配一页内存来放置页目录，然后调用mappages 来建立内核需要的映射，这些映射可以在 kmap（1728）数组中找到。这里的映射包括内核的指令和数据， PHYSTOP 以下的物理内存，以及 I/O 设备所占的内存。setupkvm 不会建立任何用户内存的映射，这些映射稍后会建立。

3



**(二)** **kalloc.c:**

这一文件是 XV6 的物理内存分配器，分配器中的数据结构是一个由可分配物理内存页构成的*空闲链表*。这个空闲页的链表的元素是结构体 struct run（2764），分配器将每个空闲页的 run 结构体保存在该空闲页本 中，因为空闲页中没有其他数据。



分配器还用一个 spin lock（2764-2766）来保护空闲链表。链表和这个锁都封装在一个结构体中，进行XV6 的动态内存管理。

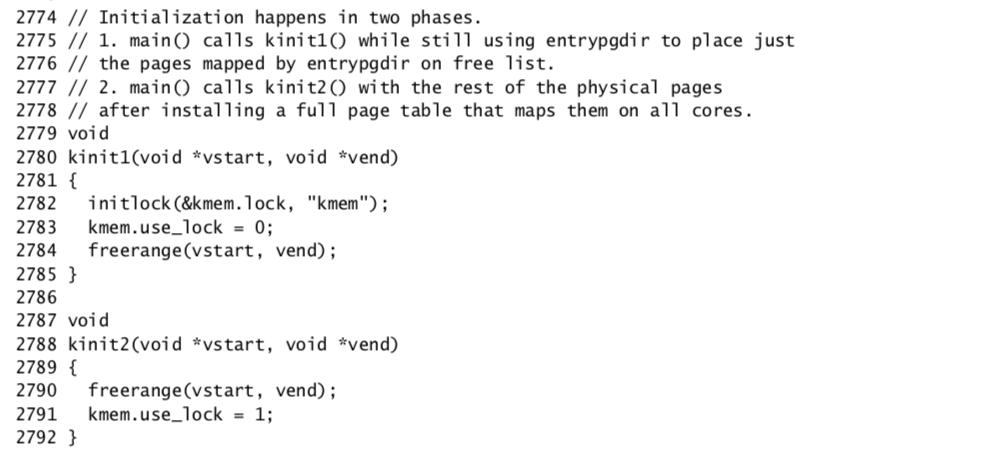


main 函数调用了 kinit1 和 kinit2 两个函数对分配器进行初始化（2780）。这样做是由于 main 中的一部分代码都不能使用锁以及 4MB 以上的内存。kinit1 在前 4MB

4

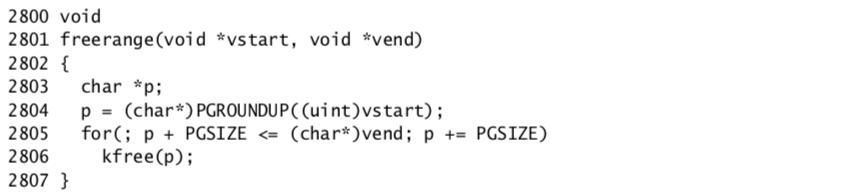
进行了不需要锁的内存分配。而 kinit2 允许了锁的使用，并使得更多的内存可用于分配。原本应该由 main 决定有多少物理内存可用于分配，但在 x86 上很难实现。所以它假设机器中有 240MB（PHYSTOP）物理内存，并将内核末尾和 PHYSTOP 之间的内存都作为 个初始的空闲内存池。

**kinit1** 和 **kinit2** 调用freerange将内存加入空闲链表中，freerange则是通过对每一页调用 kfree 实现该功能。一个 PTE 只能指向 个 4096 字节对齐的物理地址（即是4096 的倍数），因此 freerange用PGROUNDUP 来保证分配器只会释放对齐的物理地址。分配器原本一开始没有内存可用，正是对 kfree 的调用将可用内存交给了分配器来管理。



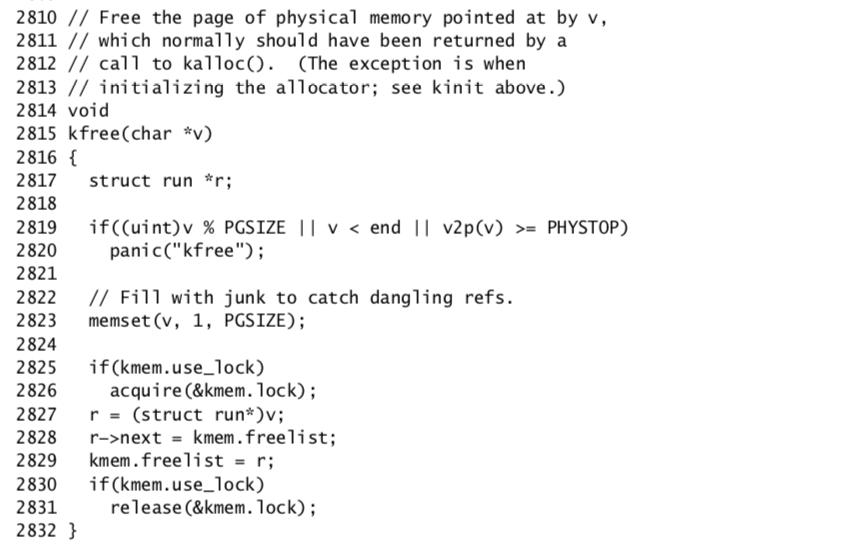
分配器用映射到高内存区域的虚拟地址找到对应的物理页，而非物理地址。所

* kinit 会使用p2v(PHYSTOP)来将 PHYSTOP（一个物理地址）翻译为虚拟地址。分配器有时将地址看作是整型，这是为了对其进行运算（譬如在 kinit 中遍历所有页）； 有时将地址看作读写内存用的指针（譬如操作每个页中的 run 结构体）；对地址的双重使用导致分配器代码中充满了类型转换。另外一个原因是，释放和分配内存隐性地改变了内存的类型。



5

函数 **kfree**（2815） 先将被释放内存的每一字节设为1。这使得访问已被释放内存的代码所读到的不是原有数据，而是垃圾数据；这样做能让这种错误的代码尽早崩溃。接下来 kfree 把 v 转换为 个指向结构体 struct run 的指针，在 r->next 中保存原有空闲链表的表头，然后将当前的空闲链表设置为 r。



**kalloc** 移除并返回空闲链表的表头。



二、问题讨论

请大家围绕如下一些问题阐述原理课的相关内容，以及 XV6 中是如何实现的。

1. XV6 初始化之后到执行 main.c 时，内存布局是怎样的(其中已有哪些内容)?

在调用main 函数之前，XV6 进行了如下操作：计算机启动之后，存放在 ROM 中 BIOS 从磁盘中的第一扇区读取引导程序，加载到内存地址 0x7c00，设置程序计数器跳转到地址，执行 bootloader。bootloader 从实模式切换到保护模式并，将 0x7c00 设置为临时栈，调用 bootmain 函数，在 0x10000 处加载并运行内核（操作系统的 elf

件），通过调用 entry 把控制权交给内核，设置 entrypgdir 并开始分页，在开启页表之后，便可以调用 main 函数。

在执行 main 函数前内存物理地址的内容：

6

|  |  |  |
| --- | --- | --- |
| 0x0000 – 0x7c00 |  | 引导程序的栈 |
| 0x7c00 – 0x7d00 |  | 引导程序的代码 |
| 0x10000 – 0x11000 |  | 内核 ELF 头 件 |
| 0xA0000 – 0x100000 |  | 设备区 |
| 0x100000 – 0x400000 |  | XV6 操作系统 |
| 其中设备区为硬件规定占用的区域。 | |  |

1. XV6 的动态内存管理是如何完成的? 有一个 kmem(链表)，用于管理可分配的物理内存页。(vend=0x00400000，也就是可分配的内存页最大为 4Mb)

在文件 kalloc.c 中定义了 XV6 的内存管理机制。动态内存管理就是把所有可用的空闲页串在一起，形成一个大链表 kmem。每个空闲页表结构 run 表示。每当有内存页被释放时，就通过 kfree( )（对每一页进行操作）把这页加入到空闲页链表中；当需要分配页表时，就使用kalloc( )从链表的头部取一个空闲页。这个内存分配器必须知道它要负责管理的内存范围，并在初始化时将整个物理地址空间都纳入其管理范围。

* kmem 结构体中还包括一个锁，一开始，锁是没有启动的，直到 main()函数调 了 kvinit2()之后锁才会被使用，因为从这里之后可能会有多个进程和多个处理器并发地访问这个数据结构。

1. XV6 的虚拟内存是如何初始化的?画出 XV6 的虚拟内存布局图，请说出每 部分对应的 内容是什么。 见 memlayout.h 和 vm.c 的 kmap 上的注释

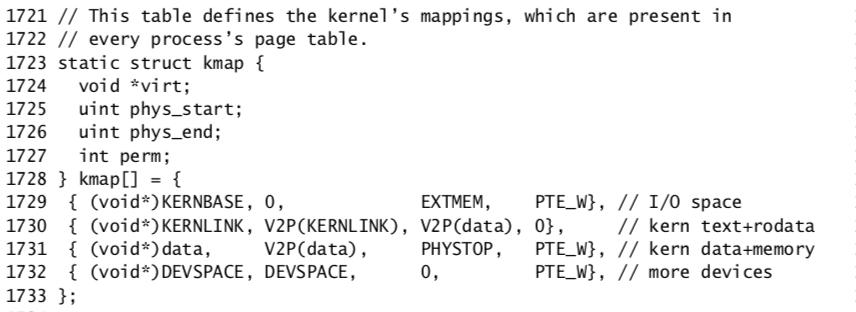
XV6 系统使用end 指针来标记 XV6 的 ELF 件所标记的结尾位置，这样， [PGROUNDUP(end), 0x400000]范围内的物理内存页是可以被用作内存页分配的。XV6调用kinit1(end, P2V(0x400000))来首先将这部分内存纳入虚拟内存页管理。虽然这部分在之前的页表中已经被映射为 4MB 大页，但是我们的目标是建立一个新的页表，这个页表使用的页大小为 4KB。由于这部分内存已经被分配为一个 4MB 内存 页，且硬件已经会自动执行虚拟内存地址翻译，故需要使用 P2V()函数将物理地址转换为虚拟地址。之后的代码里还会存在很多这样的虚拟地址到物理地址的转换。

XV6 的内存分配器必须知道它要负责管理的内存范围。由于此时虚拟内存已经开启，且页表表项只有两条，因此 XV6 必须利用已有的虚拟地址空间，在其中创建新的页表。这就是 main()函数中 kinit1()和 kvmalloc()所做的事情。

kinit1()函数会调用freerange()函数，按照前文叙述的方式，建立从PGROUNDUP(end)地址开始直到 0x400000 为 的全部内存页的链表。这样，我们得到了第一组可以使用的虚拟内存页，然后内核就可以运行 kvmalloc()使用这些内存页了。

7

kvmalloc()函数获得一个虚拟内存页并将其初始化一级页表。这个一级页表的内容在 vm.c 中的 kmap 处被定义，具体内容如下

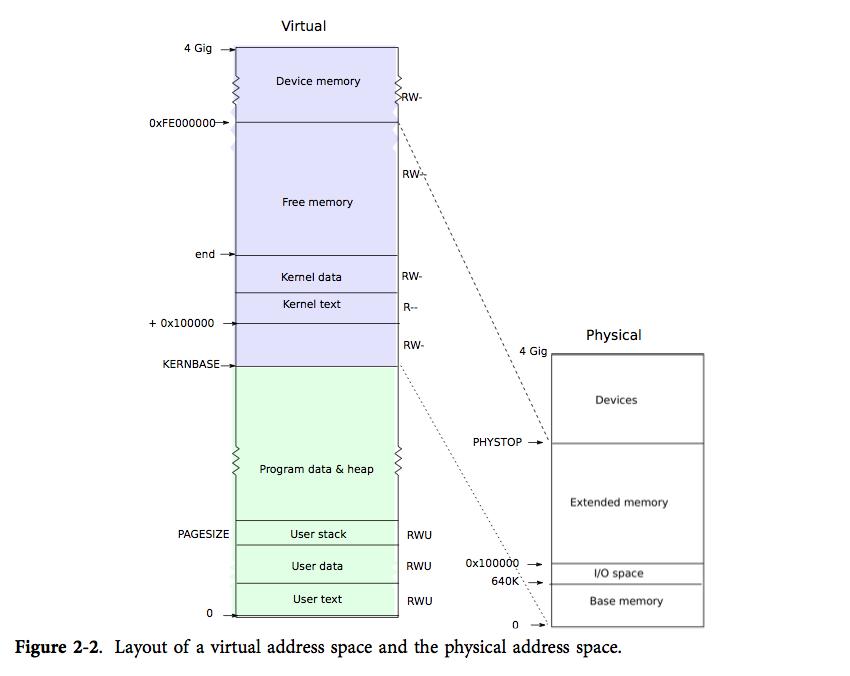


注意以上映射规则会被生成为 x86 所要求的对应一级页表和二级页表。需要的时候，kvmalloc()函数所调用的 walkpgdir()函数会申请新的内存页用作二级页表。

之后，main()函数会调用seginit()函数重新设置 GDT。新的 GDT 与之前的 GDT 的主要区别在于设置了用户数据段和用户代码段。虽然这些段依然是对 32 位偏移进 直接映射，但其执行权限与内核的段有所不同。GDT 中的 TSS 表项直到第一个用户进程创立时才会被设置，并且其内容会随着当前用户进程的切换而改变。

最后，main()函数会调用 kinit2()将[0x400000, 0xE00000]范围内的物理地址纳入 到内存页管理之中。至此，XV6 的内存页管理系统和内核页表已经全部建立完毕。需要注意的是，这个内核页表(kpgdir 变量)只会在调度器运行时被使用。对于每一个用户进程，都会拥有自己独自的完整页表，其中也包含了一份一模一样的内核页表。

XV6 的虚拟内存布局如图：

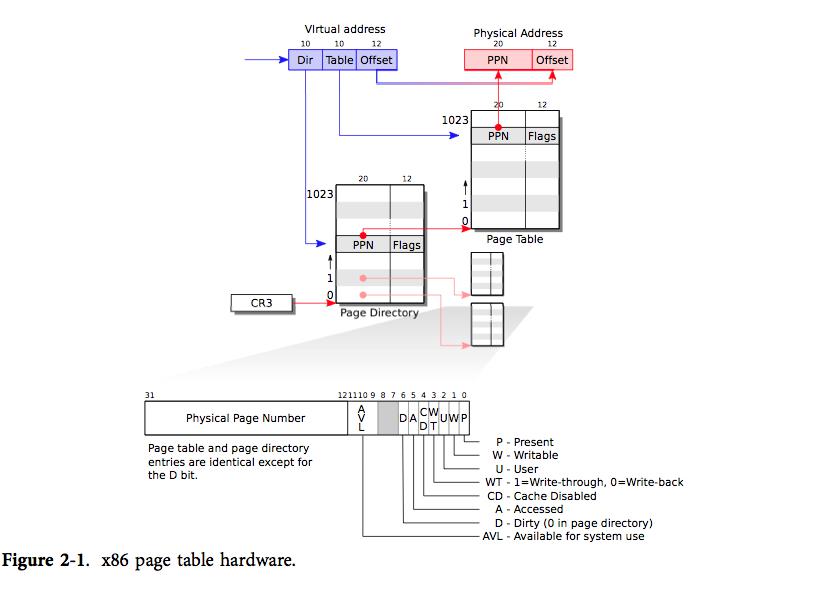


8

1. 关于 XV6 的内存页式管理。发 中断时， 哪个页表? 个内页是多 ?页 录有多少 项?页表有多少项?最 持多 的内存?画出从虚拟地址到物理地址的转换图。在 XV6 中，是如何将虚拟地址与物理地址映射的(调 了哪些函数实现了哪些功能)?

发生中断时，使用中断描述符表，将换入CPU 的进程的页表首地址存入 CR3 寄存器；XV6 一个内存页为 4K，页表项为 4B，故页表有 1024 项，页目录也有 1024 项；最大支持 4G 内存；

虚拟地址到物理地址的转化如下图：



* XV6 中，虚拟地址到物理地址的映射主要是通过 vm.c 文件中的 mappages 函数实现的，还涉及到了 walkpgdir 函数，具体函数的功能在上文源代码阅读中已提到，故不再赘述。

9