Lab 2: Memory Management

本次实验将会为OS编写内存管理代码。内存管理包括以下两个部分：

1. 内核物理内存分配器，内核借此来分配内存然后释放内存。分配器将以4096B为一个单位（页）进行操作。本次实验中需要维护一个记录哪一个物理页面为空闲或占有、页面由哪些进程共享的数据结构，还需编写分配和释放页面的规则。
2. 虚拟内存，即将内核和用户软件使用的虚拟地址映射到物理内存地址的部分。当指令占用内存、查询各类页表时x86硬件的内存管理单元MMU则会映像地址。本次实验中将会根据要求改进JOS以安装MMU页表。

第一部分 物理页面管理

OS必须保持跟踪物理RAM中哪些部分是空闲的、哪些部分是正在被占用的。JOS以页粒度管理PC的物理内存，因此可以使用MMU来映射、维护已分配内存的每一块。

首先编写物理页面分配器。它通过一个 `struct PackInfo` 对象的链表（与xv6不同，这些对象本身并没有嵌入在自由页面中）保持追踪页面的空闲状态，每一个对象对应一个物理页面。在完成虚拟内存的其他部分之前需要先完成物理页面分配器，因为页表管理的代码将会需要先分配存放页表的物理内存。

> **练习一：**
>
> 在文件 `kern/pmap.c` 中完成以下函数的实现：
>
> `boot_alloc()`
> `mem_init()` （完成至调用 `check_page_free_list(1)` 即可）
> `page_init()`
> `page_alloc()`
> `page_free()`
>
> 函数 `check_page_free_list()` 和 `check_page_alloc()` 可用于测试。应该启动JOS观察 `check_page_alloc()` 是否报告正确。可以添加一些 `assert()` 来验证假设是否正确。

第二部分 虚拟内存

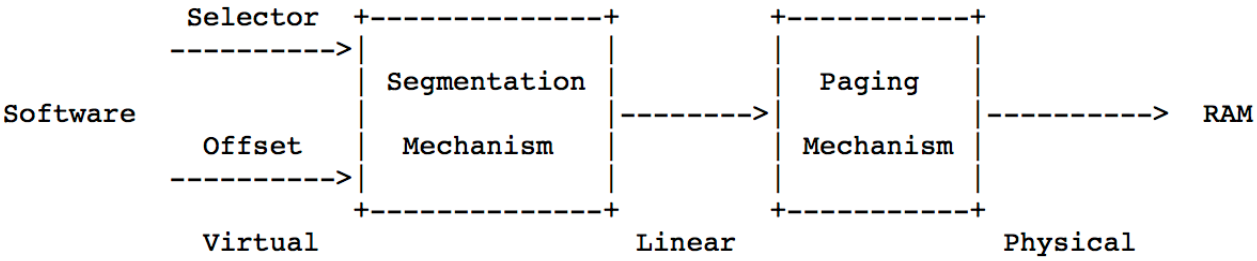
首先熟悉x86保护模式下的内存管理结构：分段和页面翻译。

练习二：

阅读Intel 80386 Reference Manual的第五、六章。阅读有关页面翻译和页面保护的部分（5.2和6.4）。大致浏览有关分段的部分，虽然JOS通过页实现虚拟内存和保护，但是段翻译和段保护在x86下也无法忽视。

1. 虚拟地址、线性地址和物理地址

在x86术语中，虚拟地址包含段描述符和段内偏移。线性地址是段翻译之后、页翻译之前的地址。物理地址则是在段页翻译后得到的最终出现在去往RAM的硬件总线上的地址。



一个C的指针即是虚拟地址中的偏移量。`boot/boot.s`中的全局描述符表GDT将所有分段的基地址置为0并限制其在0xffffffff内，从而有效地将段翻译省去。因此描述符就没有任何效果，线性地址和虚拟地址中的偏移量始终一致。在下一Lab中，我们将会与段进行进一步的交互，但在内存翻译中可以只关注页二暂时忽略分段。

回想Lab 1中的第三部分，我们安装了一个简单的页表从而内核可以于0x00100000处运行其链接地址，尽管实际上其加载到物理内存中的地址就在ROM BIOS的上方0x00100000处。这个页表仅仅映射了4MB的内存。在本次实验需要设置的虚拟内存布局中我么将会扩展页表至映射物理内存中的从0xf0000000开始的前256MB以及其他虚拟内存的区域。

练习三：

虽然GDB仅可以通过虚拟地址访问QEMU的内存，但是在设置虚拟内存的同时能够检查物理内存将是十分有利的。复习实验工具向导中QEMU的监视命令，尤其是`xp`命令，该命令可以检查物理内存。在终端中用`Control-a c`来访问QEMU监视器。

在QEMU监视器中使用`xp`命令、GDB中使用`x`命令来检查相对应的物理地址和虚拟地址处的内存，确保你看到的数据一致。

修补的QEMU版本提供了`info pg`命令，该命令详细显示了当前页表的内容，包括所有映射的内存范围，权限和标志。Stock QEMU也提供了`info mem`命令，该命令显示虚拟内存映射范围以及权限的概括。

由CPU执行的代码可知，一旦进入保护模式就再无法直接使用线性地址或者物理地址。所有的内存饮用都被

MMU翻译为虚拟地址，也就是说所有C中的指针都是虚拟地址。

JOS通常需要将地址操作为不透明值或整数值，而不将其解引用，例如在物理内存分配器中。这些地址有时候是虚拟地址，有时候是物理地址。为方便记录代码，JOS的源代码将两种情况做了区分：`uintptr_t`类型代表不透明的虚拟地址，`physaddr_t`类型则代表物理地址。这两种类型都是32位整数类型`uint32_t`的同义词，所以编译器允许两种类型之间的转换。又因为这两种类型都是整数类型而不是指针类型，所以编译器不允许解引用。

JOS内核可以先将`uintptr_t`转换成指针类型再对其解引用。但是内核不能解引用物理地址，因为所有的内存引用都由MMU翻译。如果将`physaddr_t`强制转换成指针类型然后对其解引用，也许可以访存得到的地址（硬件将其理解为虚拟地址），但是却可能得不到预期的内存位置。

总结起来就是：

C Type	Address
`T*`	Virtual
`uintptr_t`	Virtual
`physaddr_t`	Physical

JOS内核有时需要读取、修改只知道物理地址的内存。例如，向页表中添加映射时为了存放页目录会分配并初始化物理内存。但是内核和其他软件一样无法跳过虚拟内存翻译直接访存物理内存。JOS将从0开始的全部物理地址重新映射到虚拟内存0xf0000000处的一个原因就是为了帮助内核读写那些JOS已经知道是物理地址的内存。为了将物理地址转换成内核可以读写的虚拟地址，内核必须将0xf0000000添加到物理地址中以找到其在重新映射区域的虚拟地址，现可以通过`KADDR(pa)`命令来完成这一添加。

JOS内核有时候也需要找到内存中存放内核数据结构的虚拟地址对应的物理地址。内核全局变量和`boot_alloc()`分配的内存都位于从0xf0000000开始的内核加载的区域，这个区域同时也是映射所有物理内存的区域。因此内核只需简单地减去0xf0000000即可以将这个区域内的虚拟地址转换成物理地址，可以通过`PADDR(va)`命令来完成减法。

2. 引用计数

在后期的实验中可能经常会有同一物理页面同时映射到多个虚拟地址，或是不同环境下的地址空间。这时需要在`strcut PageInfo`中`pp_ref`字段中保留每一个物理页面的引用计数。当物理页面的引用计数变为0时，这个页面即可以被释放，因为不再被使用。通常来说，引用计数应当与所有页表中在`UTOP`下方出现的物理页面的数目，在`UTOP`上方出现的映射是内核在引导启动时设置好的，不会被释放，所以不需要引用这些页面的数目。引用计数还可能用于追踪保留到页面目录页的指针的数目，反过来说，页面目录引用页表页的数目。

使用`page_alloc`时要小心。其返回的页面的引用计数总是0，所以每当对返回的页面进行一定处理（例如将其插入页表），`pp_ref`都应该增加。有时候也有可能被其他函数如`page_insert`处理，也有可能是调用了`page_alloc`的函数直接执行。

3. 页表管理

现在应当实现一套管理页表的规则：插入、删除从线性地址到物理地址的映射，以及需要时创建页表页。

> **练习四：**
>
> 实现 `kern/pmap.c` 中的下列函数：
>
> `pgdir_walk()`
> `boot_map_region()`
> `page_lookup()`
> `page_remove()`
> `page_insert()`
>
> 由 `mem_init()` 调用的 `check_page()` 可以测试页表管理规则。

第三部分 内核地址空间

JOS将处理器的32位线性地址分成两个部分。我们将在下一实验中开始加载和运行的用户环境（进程）将会控制低部分的布局和内容，而高部分始终保持由内核控制。两个部分的分界线由 `inc/memlayout.h` 中的 `ULIM` 符号进行较为随意的定义，为内核保留大约256MB的虚拟地址空间。这解释了Lab 1中为什么要给内核设置这么高的链接地址：如果不这样的话，内核的虚拟地址空间将不会有足够的空间同时映射到下边的用户环境中。

参考 `inc/memlayout.h` 中的JOS内存布局图象对于今后的实验都将有所帮助。

1. 权限和故障隔离

内核和用户内存都位于每个环境的地址空间中，所以需要通过x86页表中的权限位来限制用户代码仅有权访问地址空间中的用户部分。否则用户代码中的错误将可能覆盖内核数据，导致崩溃甚至更严重的故障，此外也可以防止用户代码盗取其他环境的私密数据。需要注意的是，可编写的权限位 `PTE_W` 同时影响用户代码和内核代码。

用户环境不可能有对 `ULIM` 以上内存的访问权限，二内核可以读写这一部分内存。在[`UTOP` , `ULIM`)这一范围内，用户和内核具有相同的权限：仅可读而不可写。这一范围内的地址用于将特定的内核数据结构向用户环境公开。最后， `UTOP` 以下的地址空间供用户使用，用户环境将设置访问此内存环境的权限。

2. 初始化内核地址空间

首先设置的是高于 `UTOP` 的地址空间，即地址空间中的内核部分。 `inc/memlayout.h` 已经包含了将要用到的布局。使用之前写好的函数设置合适的从线性地址到物理地址的映射。

练习五：

补充 `mem_init()` 中调用 `check_page()` 后边遗失的代码。补充之后将会传递 `check_kern_pgdir()` 和 `check_page_installed_pgdir()`

3.地址空间布局替代方案

JOS所用的地址空间布局并不是唯一一种。一个OS可能将内核映射到低线性地址中，把线性地址的高部分保留给用户进程使用。但是x86并没有采取这种方法，因为x86下的一个向后兼容模式，虚拟8086模式，为了使用线性地址空间的底部而采取硬连线方式，如果将内核映射到那里，这个模式便不能使用。

将内核设计成不给自己保留任何线性或者虚拟地址的固定部分，而允许用户级进程毫无限制地使用全部4GB虚拟地址空间，同时还完全保护内核不被其他进程妨碍。这一方案虽然非常困难，但是仍然是可行的。