北京大学操作系统实习(实验班)报告

JOS-Lab 2:

Memory Management

黄睿哲 00948265
huangruizhe@pku.edu.cn

Feb 20, 2012

目录

内容一：总体概述

JOS-Lab2 实习的主要内容为：为系统添加内存管理代码，包括**物理页面管理**和**虚拟页式管理**。前者强调对机器拥有的物理内存进行管理，包括建立相应的数据结构、处理物理页分配和回收工作、管理空闲页链表等；后者是配合 x86 处理器的页式地址管理功能(MMU)，完成（虚拟）线性地址到物理地址的转换，包括建立页目录、页表等。

I: Physical Page Management

物理页面管理，就是将物理内存划分为大小相等的区域（物理页），并按页对物理内存进行分配和回收。

本次实习要求我们实现一个物理页面分配器，用链表记录空闲页面，其中链表中结点是 struct Page 结构体类型，每一个链表结点和物理页一一对应。页大小由 PGSIZE 指定，这里 PGSIZE 等于 4KB，因此物理页总数 npages 等于物理内存的总大小除以 4KB，也就是右移 12 位。在 Lab1 中，Kernel 已经被装载到物理内存 1MB 到 end 的空间中；在 Lab2 中，Kernel 首先接着 end 之后以 4KB 对齐的位置，为页目录 kern_pgdir 预留了空；然后在接下来的位置分配了 pages 数组，用来对应每一个物理页并记录物理页使用情况；之后就是用 page_free_list 链表将可用的空闲物理页链起来，这部分工作就基本完成了。

和这一部分内容相关的函数为文件 kern/pmap.c 中的 boot_alloc(), mem_init(), page_init(), page_alloc(), page_free()。

II: Virtual Memory

虚拟页式管理，主要是建立页目录和页表数据结构，并利用页目录和页表，将（虚拟）线性地址映射到物理地址。

首先，要区分下面几个概念：逻辑地址、线性地址、物理地址、内核地址。这在报告中将有详细叙述。由于 x86 的机制已经为我们完成了逻辑地址到线性地址也就是段式转换的过程，因此实习的任务主要是实现线性地址到物理地址的映射，也即填写页目录和页表项。此外，我们还需要了解包括：JOS 代码中提供的一些工具宏、页目录和页表的表项结构、地址转换过程等知识。

地址转换过程大致如下。给定一个 32 位线性地址，它可以被拆分为 10+10+12 位：高

10 位是该线性地址在页目录中对应的页目录项的下标，从而得到页目录项，页目录项也是一个 32 位无符号整数，其中高 20 位保存的是地址，低 12 位保存一些和地址相关的信息；中间 10 位是该线性地址在页表中对应页表项的下标，从而得到页表项，页表项和页目录项的结构相同；根据线性地址的高 20 位就可以从页表定位到真正存放数据的物理页，而低 12 位则是数据在物理页内的偏移。需要注意的是，写入页目录项和页表项的，应该是实际的物理地址。另外，我们注意到页目录和页表项中的物理地址只有 20 位，实际上这是真正物理地址 32 位中的高 20 位，由于对齐，低 12 位全为 0。

关于实现页目录和页表。页目录就是 kern_pgdir，而页表则应根据需要进行动态的申请。建立映射的过程，就是填写页目录和页表的过程，不会涉及目标地址的物理内存分配。给定一个 32 位线性地址，它的页目录项一定在 kern_pgdir 中存在，而根据页目录项的 P 位可以判断对应的页表是否存在，若不存在还需要为页表分配物理内存并进行一定的初始化，然后就可以填写相应的页表项了。

和这一部分内容相关的函数为文件 kern/pmap.c 中的 pagedir_walk(), boot_map_region(), page_lookup(), page_remove(), page_insert()。

III: Kernel Address Space

在 Lab1 中，我们其实已经开启了分页机制，但是这个分页机制只能将线性地址[KERNBASE, KERNBASE + 4MB)映射到物理地址的[0, 4MB)，这仅仅对于内核运行来说是足够的。在 Lab2 中，直到 Kernel 重置 CR3 之前（在 kern/pmap.c: mem_init()中），Kernel 进行地址转换仍使用 Lab1 的中简单页表。

因此，为保证正确运行，在 Kernel 切换到新的页表之前，我们必须在新页表中为 Kernel 建立好 Kernel 线性地址空间到物理内存的映射。具体来说，我们要将 pages 数组线性地址映射到物理地址 UPAGES 开始处，将 bootstack 映射到[KSTACKTOP-PTSIZE, KSTACKTOP)，将[KERNBASE, 2^32)映射到[0, 2^32 - KERNBASE)。这样一来，旧的页表就可以完全扔掉了。

Kernel 在 32 位线性地址空间中占据较高的部分，地址空间的较低部分留给用户使用。在 JOS 内存管理中只使用唯一的页目录，也就意味着 Kernel 和所有的用户程序，共用同一个 32 位的 4GB 地址空间。因此，还要注意通过页目录和页表表项中相应的信息位，来控制内核/用户，读/写权限。另外，在页目录或页表更新时，注意是否要使 TLB 无效。

在这一部分，我们还实现了 Challenge 的内容，包括开启 4M 页机制、输出控制台分页信息等。

内容二：任务完成情况

I: 任务完成列表

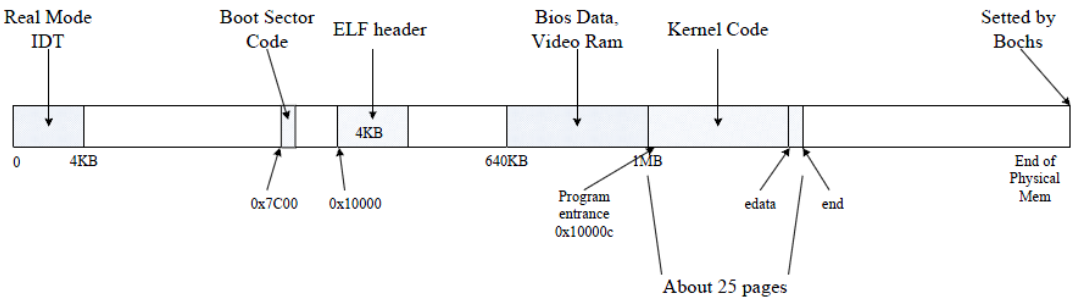
Exercise	1	2~4	5
第一周	Y	Y	
第二周			Y

Challenge	1	2	3	4
第三周	Y	Y		
第四周			?	Y

II: 具体 Exercise 完成情况

1. Physical Page Management

首先，我们应该了解到目前为止，物理内存的使用情况，如下图所示：



我们可以观察到，物理内存大致使用了四块：0~4KB 放置了和实模式 IDT 相关的一些信息，在此不必深究；0x7c00 开始保存了 boot loader 的可执行代码；0x10000（4 个 0）开始保存的是为了读入 Kernel 可执行代码的 ELF 文件头；从 640KB 开始到 end 指向的位置保存了 BIOS 等一些和 IO 有关的内容，以及从 0x100000（5 个 0，1MB）开始保存的 Kernel 可执行代码。end 是链接器作链接时得到的内核结束地址。edata 是什么，目前我还不太了解。

注意到，boot loader、Kernel 可执行代码的 ELF 文件头这两块内存的内容，我们今后不

再需要用到，因此是可以覆盖的。

目前为止，物理内存上还没有实现统一的管理，包括分配和回收空间等功能。因此我们在以下 exercise1 中将逐渐实现这些功能。

Exercise 1. In the file kern/pmap.c, you must implement code for the following functions (probably in the order given).

boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()

check_page_free_list() and check_page_alloc() test your physical page allocator. You should boot JOS and see whether check_page_alloc() reports success. Fix your code so that it passes. You may find it helpful to add your own assert()s to verify that your assumptions are correct.

1.1 boot_alloc()

boot_alloc()是一个临时的物理内存空间分配器。注意，它确实是临时的。为什么需要一个临时的函数呢？我们现在是想在物理内存上建立起统一的管理，简单地说，我们希望把物理内存像切蛋糕似的划分成一块一块，也就是物理页，然后根据需求按物理页进行内存非配。既然切成一块一块，我们就需要一些内存对这一块一块物理页的使用情况等信息进行记录，也就是一个数组。那么装下这个数组的空间从哪里来，由谁分配，我们的物理内存管理不是还没有建立起来么？没错，所以我们就只能通过一个临时的分配器 boot_alloc()——在尚未建立页式物理内存管理的条件下——对几乎裸露的物理内存进行分配，分配给即将建立起的、页式物理内存管理需要的数组空间。在此之后，页式物理内存管理已经建立起来，我们分配物理内存就要按页来分配了——程序员不管 OS 分给他哪一页，反正 OS 给他一页就好，但是 OS 要记录清楚每一页的使用情况——这时的真正的物理内存空间分配器是后面介绍的 page_alloc()函数。

boot_alloc()函数要实现的主要功能是：传入参数 n 指定分配的字节数，若物理内存空间足够分配，则返回分配的空间的起始内核地址。此外，boot_alloc ()还有一个小功能，就是将传入参数 n 设为 0，这时返回值就是当前可用的内核地址。注意，这里的内核地址是可以通过 lab1 简单页表一一映射为物理地址的，只要减去 0xf0000000 即可。

下面讨论 boot_alloc()函数实现细节。首先，我们注意到 nextfree 这个变量：

```
static char *nextfree;  // virtual address of next byte of free memory
```

小心 nextfree 是一个静态变量，也就是对 boot_alloc()函数的不同调用，访问的都是同一个变量！nextfree 指向当前物理内存下一个可用地址，并且要求这个地址是以 PGSIZE 对齐的。我们分配 n 个字节，指的就是 nextfree 起始的 n 个字节。

其次，我们注意到 end 这个外部变量：

```
extern char end[];
```

它是一个外部变量，根据注释，它的值是在 Kernel 链接完了的时候确定的，指向 Kernel 在虚拟地址空间之后的、可分配作其他用处的、第一个可用的虚拟地址。因为我们在 lab1 中曾经建立过一个简单的页表，所以现在我们所指的虚拟地址，都会通过这个简单页表一一映射成物理地址。我们输出看看这个 end 到底指向哪里：

```
end=f011996c
nextfree(roundup)=f011a000
```

这就说明了，Kernel 被装载到虚拟地址的 0xf0100000 至 0xf011996c 处，也即物理内存的 0x00100000 至 0x0011996c 处，大小为 103kB，大约 26 页。

最后，我们来看看“if we are out of memory”该怎么处理。注意到在 kern/pmap.c/i386_detect_memory()函数中，读取过物理内存大小，转化成物理页数保存在 npages 中，因此，npages*PGSIZE 就是物理内存的大小；而对于当前可用地址，我们要先将它转化成物理地址，也就是简单地减去 KERNBASE，然后加上待分配字节数 n，看看是否超过物理内存大小即可。

剩下的部分就很水了，我们只要记录起始地址，然后将 nextfree 往后移动 n 个字节即可，代码如下：

```
// Allocate a chunk large enough to hold 'n' bytes, then update
// nextfree.  Make sure nextfree is kept aligned
// to a multiple of PGSIZE.
//
// LAB 2: Your code here.

// @huangruizhe:2012-3-1
// For the case n>0 and n=0, we can use the following code uniformly without a branch
result = ROUNDUP(nextfree, PGSIZE);
// If we're out of memory, boot_alloc should panic.
// Allocate to physical address at: result-KERNBASE + n
// Total physical memory size:      npages * PGSIZE
// ("npages" is decided in i386_dectect_memory())
if((uint32_t)(result-KERNBASE + n) > (uint32_t)(npages * PGSIZE)){
    panic("boot_alloc(): out of physical memory!\n");
}
nextfree = result + n;  //note: nextfree is a *static* local variant
```

至此我们完成了第一个函数 boot_alloc()。不过，我们还可以再看看究竟哪里调用到了这个函数。共有两处：

1. mem_init()中，为 kern_pgdir 分配一页的空间，kern_pgdir 就是我们将要用到的页目录。

2. mem_init()中，为 pages 分配一页的空间，pages 就是我们记录物理页使用情况的数组。

这两处调用，都是在真正建立器物理内存管理前分配空间。我认为，其中 pages 的空间是必须调用 boot_alloc()的；而对于 kern_pgdir，它完全可以在建立起物理内存管理后，再分配空间，而现在在这里分配，可能是出于要保持特定内存布局的原因。

1.2 mem_init()

mem_init()这个函数，是 lab2 中用于控制流程的主函数，它在 kern/init.c/i386_init()中被调用。mem_init()的主要功能，是建立物理内存管理的数据结构，然后建立虚拟地址空间管理的数据结构，在这里是一个二级页表。

在 exercise1 中的 mem_init()，我们只需要走到调用 check_page_free_list(1)处即可（only up to the call to check_page_free_list(1)）。可惜悲剧的是，我当时没有理解清楚题目的这个意思，一直往下做，做啊做啊做啊，结果发现后面一点也看不懂，根本不知到在干什么。直到我实在不想写的时候，我才意识到题目中“up to”原来是这个意思，而不是“It’s up to you”的那个意思。囧。读材料，理解清楚真的很重要。

言归正传。在 exercise1 中的 mem_init()要求我们为 pages 数组分配空间。不管这么多了，他让你干什么你就干什么。前面我们一直叫嚣要“建立页式物理内存管理”，现在要开始了，第一步就是申请 pages 数组。

首先要了解 pages 数组和 struct Page 结构是什么。打个比方，如果班上有 50 个学生，要记录这 50 个学生的情况，我们就会用一张 50 行的表格，每一行和一个学生对应，哪个学生犯了错了就在他那一行画一个叉神马的；现在，我们的物理内存有 npages 页，要记录这些页的情况，我们就会用一个有 npages 个元素的数组 pages，每一个数组元素和物理页一一对应。页大小由 PGSIZE 指定，这里 PGSIZE 等于 4KB，因此物理页总数 npages 等于物理内存的总大小除以 4KB，也就是右移 12 位。

pages 数组的元素类型是 struct Page 结构类型，它的定义在 kern/memlayout.h 中如下：

```
struct Page {
    // Next page on the free list.
    struct Page *pp_link;

    // pp_ref is the count of pointers (usually in page table entries)
    // to this page, for pages allocated using page_alloc.
    // Pages allocated at boot time using pmap.c's
    // boot_alloc do not have valid reference count fields.

    uint16_t pp_ref;
};
```

这个结构体是用来描述一个物理页的使用情况的，显然它本身不是物理页，而是描述物理页的元数据，并和物理页一一对应。其中 pp_link 是 Page 在空闲页链表中下一个结点的指针，pp_ref 记录当前物理页被引用的次数。

我们说过，pages 数组和物理内存是一一对应的，JOS 为我们提供了一些非常有用的工具函数或工具宏，用来处理这个对应关系。具体来说，在 kern/pmap.h 中，page2pa()函数将一个 Page 对象的指针，转化成该 Page 对象对应的物理内存起始地址；pa2page()函数则将一个物理内存地址，转化成该地址所在物理页对应的 Page 对象的指针。这两个函数的实现是显而易见的，因为是一一对应的，而且对应的顺序也是保证的，只要知道 pages 数组的起始地址做一些偏移计算即可。

了解完上面的知识后，写代码的环节又容易了不少。给定页数 npages（这是在 kern/pmap.c/i386_detect_memory()函数中，根据物理内存大小和 PGSIZE 计算好的），调用 boot_alloc()分配 n=npages * sizeof(struct Page)字节空间，然后将这片空间初始化为全 0。代码如下：

```
////////////////////////////////////////////////////////////////////
// Allocate an array of npages 'struct Page's and store it in 'pages'.
// The kernel uses this array to keep track of physical pages: for
// each physical page, there is a corresponding struct Page in this
// array.  'npages' is the number of physical pages in memory.
// Your code goes here:

//@huangruizhe 2012-3-1
pages = (struct Page *)boot_alloc(npages * sizeof(struct Page));
memset(pages, 0, npages * sizeof(struct Page));
```

关于 mem_init()函数的整个流程，我们做完 lab2 后会回顾一遍，这样可以帮助我们对 JOS 内存管理有一个全局的清晰的认识。

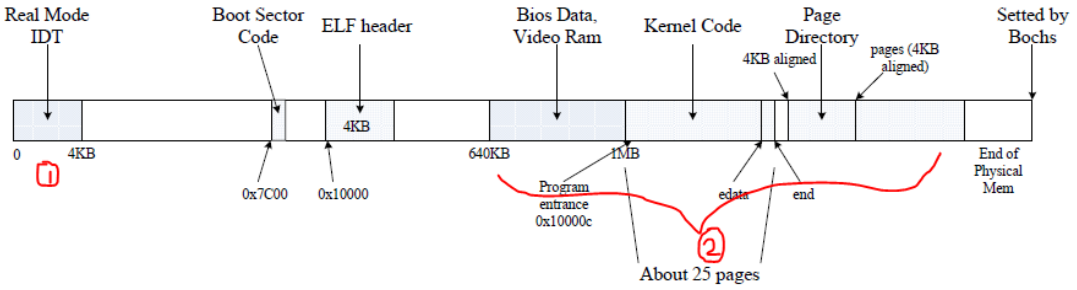
1.3 page_init()

建立页式物理内存管理的第二步，是初始化空闲页链表 page_free_list，记录当前物理内存中有那些页是可以分配出去的。这也是 page_init()函数的主要功能。

page_free_list 实际是一个 Page 结构的指针，指向空闲页链表的头部；链表的 next 域保存在每个 Page 结构的 pp_link 中。我们要理解好 page_free_list 链表和 pages 数组的

区别。我们的目标，就是遍历整个物理内存（的每一页），对于那些尚未被使用的、可以分配出去的物理页，就将其对应的 Page 结构链入 page_free_list 中；而那些被操作系统占用或系统预留空间，就不加入链表。

为此，我们需要了解已被占用的物理内存布局，如下图所示：



已被占用的物理内存有两块：

1. 0 至 4KB(第 0 页)；
2. 640KB 开始，至当前可用物理地址（利用 boot_alloc(0)获得，小心要通过 PADDR 转化成物理地址）；

这个内存布局图和前面的布局图的区别，仅仅在于在 kernel code 后，又增加了页目录 kern_pgdir 和 pages 数组的空间。另外，我们认为 boot loader、Kernel 可执行代码的 ELF 文件头这两块内存的内容是可以覆盖的。

因此，可以写出 page_init()的代码如下：

```
//
// Change the code to reflect this.
// NB: DO NOT actually touch the physical memory corresponding to
// free pages!

// @huangruizhe:2012-3-1
size_t i;
size_t i_IOPHYSMEM_begin = ROUNDDOWN(IOPHYSMEM, PGSIZE) / PGSIZE;
size_t i_EXTPHYSMEM_end = PADDR(boot_alloc(0)) / PGSIZE;
page_free_list = NULL;  // Be careful to initialize page_free_list!
for (i = 0; i < npages; i++) {
    pages[i].pp_ref = 0;
    if(i == 0){
        continue;
    }
    if(i_IOPHYSMEM_begin <= i && i < i_EXTPHYSMEM_end){
        continue;
    }
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}
```

1.4 page_alloc()

page_alloc()函数的功能，是基于 page_free_list 链表和 pages 数组分配物理页，也就是完全替代了 boot_alloc()函数的功能。

这个函数非常简单，在此不作赘述。代码如下：

```
struct Page *
page_alloc(int alloc_flags)
{
    // Fill this function in
    //huangruizhe 2012-3-1

    struct Page * page_return;
    if(page_free_list != NULL){
        page_return = page_free_list;
        page_free_list = (struct Page *)page_free_list->pp_link;
        if(alloc_flags & ALLOC_ZERO){
            memset(page2kva(page_return), '\0', PGSIZE);
        }
    }
    else{
        page_return = NULL;
    }
    return page_return;
}
```

1.5 page_free()

page_free()函数的功能是将物理页回收。具体来说，传入一个 Page 结构，我们将这个结构链回 page_free_list 链表，这样对应的物理页就被标记为空闲。代码如下：

```
void
page_free(struct Page *pp)
{
    // Fill this function in
    // @huangruizhe:2012-3-1
    pp->pp_link = page_free_list;
    page_free_list = pp;
    return;
}
```

2. Virtual Memory

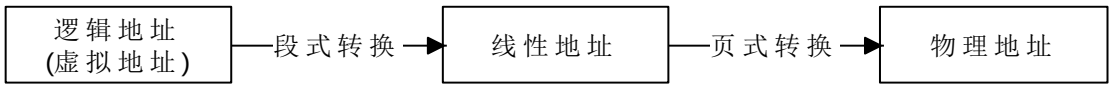
总的来说，虚拟内存管理，就是它给你一个虚拟地址，你就给它返回一个相应的物理地址。这就叫映射。

2.1 Page Translation and Page-based Protection for X86

Exercise 2. Look at chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. Read the sections about page translation and page-based protection closely (5.2 and 6.4). We recommend that you also skim the sections about segmentation; while JOS uses paging for virtual memory and protection, segment translation and segment-based protection cannot be disabled on the x86, so you will need a basic understanding of it.

这个任务是阅读 Intel 80386 Reference Manual 手册，了解 x86 体系结构虚拟地址转换、页面权限保护的有关内容。

总的来说，x86 体系结构虚拟地址转换过程包括两个步骤：段式转换，页式转换。如下图所示。其中段式转换是不可避免的，而页式转换是可选的。



在 x86 体系结构中，地址被分为三类：逻辑地址，线性地址和物理地址。逻辑地址是指程序在编译链接后，变量名字等的符号地址，也是我们在应用程序中访问的 C 指针的地址；线性地址是指逻辑地址经过 x86 保护模式的段式地址转换后的地址，该变换的过程是“段首地址+逻辑地址”；物理地址指的是内存存储单元的编址。在 JOS 中，这三类地址都是 32 位的。

对于所有对内存的访问，地址都被解释成虚拟地址，并由 MMU 执行过程。程序员的工作是设置和维护页目录和页表。

段式转换提供了将可执行代码、数据、堆栈模块分开的机制，和我们常说的 CS, DS, SS, ES 等以及段寄存器有关，这有利于多任务在同一个 CPU 上运行、切换和保护。具体来说，在 lab1 中，开启保护模式以后，段式转换将段寄存器中的值当作在 GDT 中对应描述符的索引，然后将逻辑地址用于和描述符中的该段的起始地址相加，得到线性地址。段式转换原理和过程如下图所示：

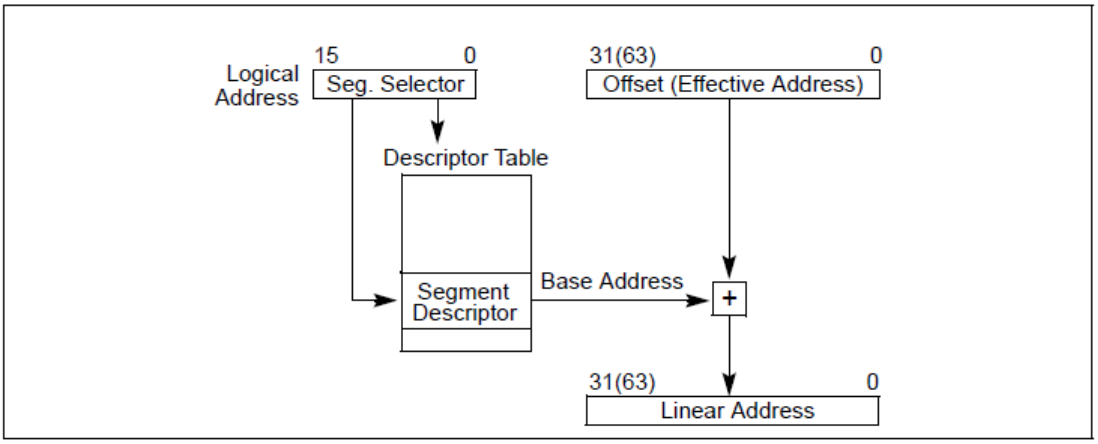


Figure 3-5. Logical Address to Linear Address Translation

值得注意的是，在 JOS 中，我们记得在 Lab1 的 boot/boot.S 的时候曾经加载了一个 GDT 将段基址设为 0，段界限设为 0xffffffff，这种方法有效地使段式转换“失效”，目的是为了简化。

页式转换提供了实现虚拟地址空间和按需分配内存的机制，根据需要，程序中仅部分内容被装载到物理内存中。页式转换是可选的，当 CR0 寄存器的 PG 位有效时页式转换开启。首先，物理内存被划分为 4KB 为单位的页框。其次，线性地址通过指定页表，页面在页表中的偏移，以及目标地址在页面中的偏移来访问物理地址。JOS 采用两级页表，页式转换的原理和过程如下图所示：

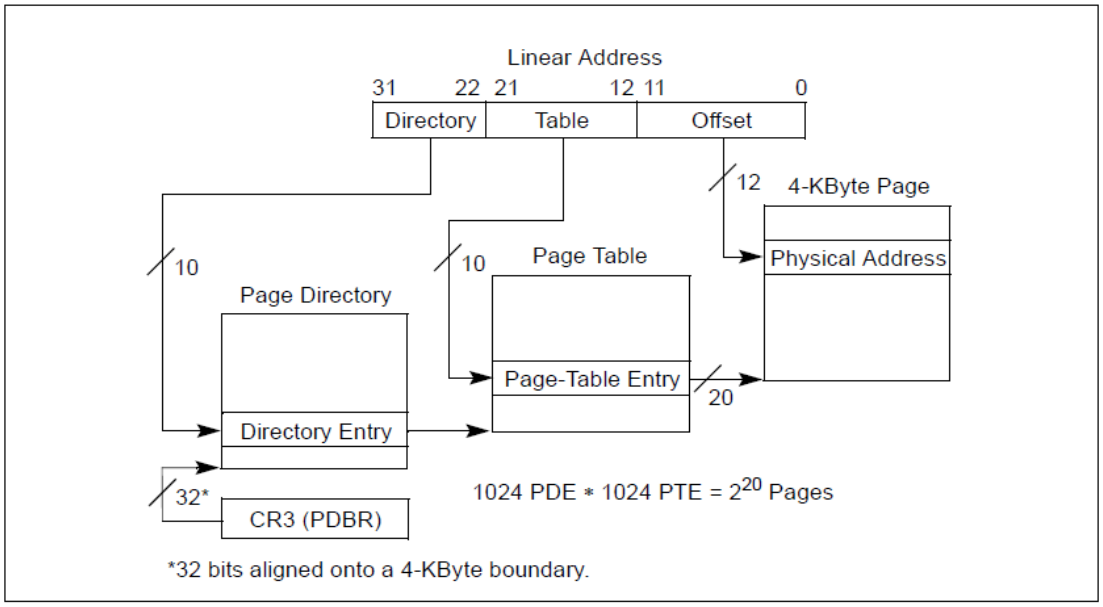
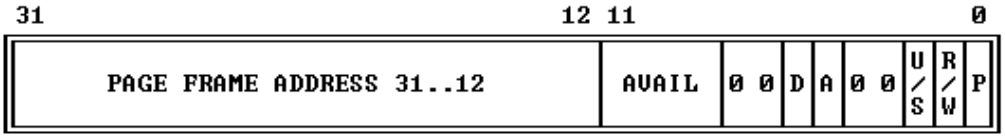


Figure 3-12. Linear Address Translation (4-KByte Pages)

Figure 5-10. Format of a Page Table Entry




```
P       - PRESENT
R/W     - READ/WRITE
U/S     - USER/SUPERVISOR
D       - DIRTY
AVAIL   - AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.
```

给定一个 32 位线性地址，它可以被拆分为 10+10+12 位：高 10 位是该线性地址在页目录 pgdir 中对应的页目录项 PDE 的下标，从而得到页目录项 PDE，页目录项 PDE 也是一个 32 位无符号整数，其中高 20 位保存的是地址，低 12 位保存一些和地址使用相关的信息；中间 10 位是该线性地址在页表中对应页表项 PTE 的下标，从而得到页表项 PTE，页表项 PTE 和页目录项 PDE 的结构相同；根据线性地址的高 20 位就可以从页表定位到真正存放数据的物理页，而低 12 位则是数据在物理页内的偏移。需要注意的是，写入页目录项 PDE 和页表项 PTE 的，应该是实际的物理地址。另外，我们注意到页目录和页表项中的物理地址只有 20 位，实际上这是真正物理地址 32 位中的高 20 位，由于对齐，低 12 位全为 0。

下面是关于页面保护的内容。页面保护有助于防止程序访问不该访问的内存地址，在即将访问一个地址前，硬件会帮助我们进行检查是否具有权限。这些权限包括 Supervisor/User，Read-only/Read-Write，在页目录和页表项的低位指定。两个权限位共有四种组合：管态只读，管态可写，目态只读，目态可写。另外，基于页的权限的设定，是由页目录和页表共同决定，Intel 手册穷举了所有的组合。由于 MMU 对页目录和页表的权限都会检查，因此在 JOS 中，可以令页目录的权限是最低权限，更高的权限可以由页表再来限制。

此外，x86 提供了 TLB 机制。为了提高地址转换的效率，系统将最常用到的页表数据保存在 TLB，只有当 TLB 不命中时才需要经过二级页表的转换。因此，操作系统在任何时候只要更改了页目录或页表，都必须刷新 TLB。Intel 手册提到了两种刷新 TLB 的方法，在 JOS 中，我们采用的是利用 MOV 指令重载 CR3 的方法。

2.2 Virtual, Linear, and Physical Addresses

逻辑地址（虚拟地址）、线性地址、物理地址，我们在 2.1 中已经进行了区分。

Exercise 3. While GDB can only access QEMU's memory by virtual address, it's often useful to be able to inspect physical memory while setting up virtual memory. Review the QEMU monitor commands from the lab tools guide, especially the xp command, which lets you inspect physical memory. To access the QEMU monitor, press Ctrl-a c in the terminal (the same binding returns to the serial console).

Use the xp command in the QEMU monitor and the x command in GDB to inspect memory at corresponding physical and virtual addresses and make sure you see the same data.

Our patched version of QEMU provides an info pg command that may also prove useful: it shows a compact but detailed representation of the current page tables, including all mapped memory ranges, permissions, and flags. Stock QEMU also provides an info mem command that shows an overview of which ranges of virtual memory are mapped and with what permissions.

这个任务是让我们使用和熟悉 QEMU monitor commands. 运行 QEMU，在控制台先按下 Ctrl+a，然后放开，迅速按下 c，就能够进入 QEMU 机器状态的监视器。目前觉得比较有用的命令和功能说明如下：

x/*N*x *vaddr*	显示虚拟内存地址 vaddr 开始 N 个字长的内容。
xp/*N*x *paddr*	显示物理内存地址 paddr 开始 N 个字长的内容。
info registers	显示所有机器内部寄存器状态，包括段选择子，GDT 等。
info mem	显示一段虚拟地址映射到的物理地址，及其权限。
info pg	显示当前的页目录、页表结构

其中在后面的 challenge 中，要求我们在 JOS 控制台实现类似 info pg，x/*N*x *vaddr*，xp/*N*x *paddr* 这样的功能。另外，在 QEMU monitor commands 中使用 help 指令还是很有帮助的。

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;
```

根据这段程序的意图，我们判断 x 应该是要保存 value 这个 C 指针的指针值，也就是指针所指向的虚拟地址，因此 x 的类型应该是 uintptr_t。

JOS 内核有时需要在只知道物理地址的情况下访问内存，例如，页目录和页表中指示的都是物理地址。然而，kernel 和其他程序一样，不可避免虚拟地址转换过程，因此不能直接访问物理地址。因此，JOS 将从 0 开始的物理地址，映射到从 0xf0000000 的虚拟地址，以便内核访问只知道物理地址的内存。在此基础之上，给定一个物理地址，kernel 需要将它加上 0xf0000000，然后再去访问。我们可以通过宏 KADDR(pa)来完成这个转换工作。

JOS 内核有时还需要在给定虚拟地址的情况下求出对应的物理地址。当然，这可以通过查询页表得到。但是，对于在 KERNBASE 以上的地址，即 kernel 中数据结构（如 kernel 中的全局变量，boot_alloc()函数的返回值等）的地址，我们可以有更加便捷的方法得到它们的物理地址。我们只要将这些虚拟地址减去 0xf0000000 即可！我们可以通过宏 PADDR(va)来完成这个转换工作。

2.3 Reference counting

这部分内容讨论了当同一个物理地址被虚拟地址映射多次时，要在相应 Page 结构的 pp_ref 记录物理页的引用次数；当 pp_ref 等于 0 时，物理页就可以被回收进空闲页链表中。

2.4 Page Table Management

Exercise 4. In the file kern/pmap.c, you must implement code for the following functions.

pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()

check_page(), called from mem_init(), tests your page table management routines. You should make sure it reports success before proceeding.

这个任务我们开始实现页表和页目录的功能。简单地说，给定任何线性地址，都能够找到一个对应的页目录项，然后就得到物理页起始地址，再加上偏移地址，就是我们要访问的物理地址。

注意!!! 写入页目录项和页表项的都应该是实际的物理地址!!! 也就是说如果要查页目录获得页表内核地址（也就是程序中能够使用的 C 指针），还要将物理地址转换成虚拟地址！

2.4.1 pgdir_walk()

pgdir_walk()这个函数的主要功能如下：

- 根据给定页目录指针 pgdir，返回线性地址 va 对应的页表项的指针。
- 如果相关页表还不存在，且 create==false，返回 NULL。
- 否则，通过 page_alloc 为页表分配一页物理内存，并相应初始化，清 0。返回 va 对应

的表项的指针。

既然说的是指针 C pointer，很显然，我们返回的应该是页表项虚拟地址。只要调用这个函数，调用者就能够根据返回结果修改页表项的内容，包括设定物理地址，设定 P 位，设定权限等。pgdir_walk()代码如下：

```
pte_t * pgdir_walk(pde_t *pgdir, const void *va, int create){
    pte_t *pgdir_entry = &pgdir[PDX(va)];
    // For 4MB paging
    if(*pgdir_entry & PTE_PS){
        return pgdir_entry;
    }
    // If the page table page is present in memory.
    if(*pgdir_entry & PTE_P){
        // Get the virtual starting address of target page table, using KADDR
        pte_t *pt_va = (pte_t *)KADDR(PTE_ADDR(*pgdir_entry));
        return pt_va + PTX(va);
    }
    // Else, the page table page does not exist, we may create it.
    else{
        if(create == 0){
            return NULL;
        }
        else{//allocate a new physical page
            struct Page * new_pgtbl_page = page_alloc(1);//parameter 1 is for clearing to all 0
            if(new_pgtbl_page == NULL){
                return NULL;
            }
            else{//allocate success
                new_pgtbl_page->pp_ref = 1;
                physaddr_t new_pgtbl_pa = page2pa(new_pgtbl_page);
                *pgdir_entry = new_pgtbl_pa | PTE_W | PTE_U | PTE_P;
                pte_t * pt_va = (pte_t *)KADDR(new_pgtbl_pa);
                return pt_va + PTX(va);
            }
        }
    }
    return NULL;
}
```

首先，根据 va 的高 10 位在 pgdir 中得到页目录项 pgdir_entry，这里，可以利用 PDX 从 va 中获得页目录项的下标。因为我们后面可能会修改到页目录项 pgdir_entry 本身的内容，因此我们最好操作页目录项的指针，而不是值。

其次，是判断是否是 4M 页，这个是 challenge 的内容，后面再介绍。

随后，我们通过 P 位判断获得的页目录项 pgdir_entry 指示的页表是否存在。如果存在，这个页表已经可以使用，因此我们可以通过 PTE_ADDR 从 pgdir_entry 中解析出页表的物理地址，通过 KADDR 转化成内核地址；然后我们再利用 PTX(va)得到 va 对应页表项在页表中的下标。最后，将页表起始内核地址加上下标就可以得到页目录项的地址，也就是页目录项的指针。

如果页表不存在，create==false，直接返回 NULL。否则 create!=false，我们就要为页表申请一页物理内存，这是通过 page_alloc 完成，返回的是一个 Page 的指针，我们将这个 Page 指针通过 page2pa 转化成物理页起始的物理地址，然后用这个物理地址设置页目录项 pgdir_entry，包括设置 P 位，S/U 位, R/W 位。然后我们就可以用和上面类似的方法返回页表项指针。

在书写这个函数的时候，头脑一定要保持清醒，各种地址、各种指、针各种值，还有几个工具宏，要了解每一步得到的结果是什么，而我们需要什么。就我的体会而言，只要明确两点就足够了：①页目录和页表中保存的是物理地址；②我们在内核代码中使用的指针是虚拟地址。

pgdir_walk()函数需要经历两级页表，估计这也是该函数得名的原因。这个函数在后面会经常用到。

2.4.2 boot_map_region()

boot_map_region()这个函数的主要功能如下：

• 根据指定页目录 pgdir，权限 perm，页大小 PGSIZE，将[va, va+size)的虚拟地址映射到物理地址的[pa, pa+size)。

由于这个函数只是当建立 UTOP 以上虚拟地址到物理内存映射的时候被调用，它并不是实际使用物理页，因此不改变 Page 结构的 pp_ref. 代码如下：

```
static void boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
    uintptr_t offset;
    pte_t *pte;
    for(offset = 0; offset < size; offset += PGSIZE){
        pte = pgdir_walk(pgdir, (void *)va, 1);
        *pte = pa | perm | PTE_P;
        pa += PGSIZE;
        va += PGSIZE;
    }
    return;
}
```

代码就是一个一重的循环，从 pa 开始遍历到 pa+size，每次 map 一个物理页；通过 pgdir_walk()获得 va 相应页表项，然后设置该页表项为 pa|perm|PTE_P 即可。这里需要小心的是，每次循环 pa，va 不要忘了各自加上 PGSIZE.

2.4.3 page_lookup()

page_lookup()这个函数的主要功能如下：

• 根据指定页目录 pgdir，虚拟地址 va，返回 va 对应物理页（如果有）的 Page 结构体指针；否则返回 NULL。

• 如果 pte_store 不为 NULL，将其置成 va 对应页表项的指针。

page_lookup()实现比较简单，代码如下：

```
struct Page * page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    pte_t *pte = pgdir_walk(pgdir, (void *)va, 0);
    if(pte_store != 0){
        *pte_store = pte;
    }
    if(pte == NULL || (*pte & PTE_P) == 0){
        return NULL;
    }
    return pa2page(PTE_ADDR(*pte));
}
```

2.4.4 page_remove()

page_remove()这个函数的主要功能如下：

• 根据指定页目录 pgdir，虚拟地址 va，将 va 到相应物理地址（如果有）的映射解除；否则直接返回。

• 物理页的引用次数要相应减 1，如果物理页引用次数减到 0，要将物理页放回空闲页链表中；同时记得修改页目录项。

这个函数也没有什么好说的，就根据提示使用了这几个函数 page_lookup, tlb_invalidate, page_decref。代码如下：

```
void page_remove(pde_t *pgdir, void *va)
{
    pte_t *pte;
    struct Page * pg = page_lookup(pgdir, va, &pte);
    if(pg != NULL){
        page_decref(pg);
        *pte = 0;
        tlb_invalidate(pgdir, va);
    }
    return;
}
```

2.4.5 page_insert()

page_insert()函数的主要功能如下：

• 指定页目录 pgdir，将虚拟地址 va 映射到 Page 结构指针 pp 对应的物理地址，同时设置权限位 perm。

• 如果 va 已经映射到某个物理地址，应先将该物理地址所在物理页 page_remove()掉。

• va 经过的页表如果不存在，就申请一页。

• 映射到的物理地址所在页的 Page 结构的 pp_ref 自增 1。

• 更新 TLB。

page_insert()和 boot_map_region()都是将虚拟地址映射到物理地址，不过功能上还有区别。page_insert()和 page_remove()的功能互为相反。代码如下：

```
int page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{
    pte_t *pte = pgdir_walk(pgdir, va, 1);
    if(pte == NULL){
        return -E_NO_MEM;
    }
    if(*pte & PTE_P){
        if(PTE_ADDR(*pte) == page2pa(pp)){
            pp->pp_ref --;
            tlb_invalidate(pgdir, va);
        }
        else{
            page_remove(pgdir, va);
            //page_remove中已更新tlb，因此不需要再次更新
        }
    }
    *pte = page2pa(pp) | perm | PTE_P;
    pp->pp_ref ++;
    return 0;
}
```

3. Kernel Address Space

JOS 将 32 位地址空间划分成两部分：User 部分和 kernel 部分，前者在低地址，后者在高地址。两者的分界线由 inc/memlayout.h 中的 ULIM 定义，在 ULIM 以上大致为 kernel 保留了 256MB 的虚拟地址空间。虚拟地址空间布局参见 inc/memlayout.h。

3.1 Permissions and Fault Isolation

这部分没什么好介绍的，就是说要防止用户程序访问超过 User 部分的地方，担心它一

不小心就修改了 Kernel 部分的数据，导致出错。那些用户不能使用的内存空间的地址，将不能通过页式转换将线性地址转换得到物理地址。

地址空间大致被分为三个部分：[0, UTOP)是用户可以读写的；[UTOP,ULIM)对用户和内核都是只读的，这部分地址的作用是向用户暴露一部分内核的数据内容；ULIM 以上的地址，用户没有任何权限。

3.2 Initializing the Kernel Address Space

Exercise 5. Fill in the missing code in mem_init() after the call to check_page().
Your code should now pass the check_kern_pgdir() and check_page_installed_pgdir() checks.

现在，我们要建立起 Kernel 部分的地址空间（UTOP 以上地址空间）到物理地址的映射。这个任务分为下面三个步骤。

3.2.1 Map 'pages' read-only to user

首先，将虚拟地址 UPAGES 映射到我们前面分配过的 pages 数组，其中对内核和用户的权限都是只读。注意 pages 要转化成物理地址。我想这样映射的目的是，可以让用户程序了解物理内存的使用情况，了解每个物理页面被引用的次数。我们看到这个区域在布局中分配了 PTSIZE 的大小，这个大小经过计算是足够映射 pages 数组的。代码如下：

```
boot_map_region(kern_pgdir,
                UPAGES,
                ROUNDUP(npages*sizeof(struct Page), PGSIZE),
                PADDR(pages),
                PTE_U | PTE_P);
```

3.2.2 Map kernel stack to 'bootstack'

其次，将虚拟地址[KSTACKTOP-KSTKSIZE, KSTACKTOP)映射到由 bootstack 指定的物理地址。小心，bootstack 实际上还是一个虚拟地址，我们可以将它输出看看，发现他是在 0xf0000000 以上的，因此实际的物理地址是 PADDR(bootstack)。

```
//////////////////////////////////////////////////////////////////////
// Use the physical memory that 'bootstack' refers to as the kernel
// stack.  The kernel stack grows down from virtual address KSTACKTOP.
// We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
// to be the kernel stack, but break this into two pieces:
//     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
//     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
//       the kernel overflows its stack, it will fault rather than
//       overwrite memory.  Known as a "guard page".
//     Permissions: kernel RW, user NONE
```

这部分的注释其实刚开始我看得不是很懂，于是我请教了林舒同学。**为什么 kernel stack 要分割成两个部分呢？** 原来，栈是从高地址向低地址向下生长的，[KSTACKTOP-KSTKSIZE, KSTACKTOP)是真正会使用到的堆栈空间；而[KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE)这部分，是为了防止栈溢出时，不小心修改到紧接着的地址空间中的数据，也就是说这是一个"guard page"，即缓冲区。这体现了 JOS 考虑问题的保守性和稳健性。所以[KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE)中的虚拟地址仅仅是保留的，不需要映射。代码如下：

```
boot_map_region(kern_pgdir,
                KSTACKTOP-KSTKSIZE,
                KSTKSIZE,
                PADDR(bootstack),
                PTE_W | PTE_P);
```

3.2.3 Map all of physical memory at KERNBASE

最后，我们将 KERNBASE 开始映射到整个物理地址空间。**这样做的目的，是为了稍后打开页式转换以后，Kernel 也能使用统一的虚拟地址来访问其自身数据**，即 Kernel 使用虚拟地址 KERNBASE+x 访问到的物理地址，正是物理地址为 x 处的内存。这部分映射的大小为 2^{32} – KERNBASE = 256 MB。

事实上，这部分映射的功能，和 lab1 简单页表的功能，完全一样！

代码如下：

```
boot_map_region(kern_pgdir,
                KERNBASE,
                1 << 28,
                0,
                PTE_W | PTE_P);
```

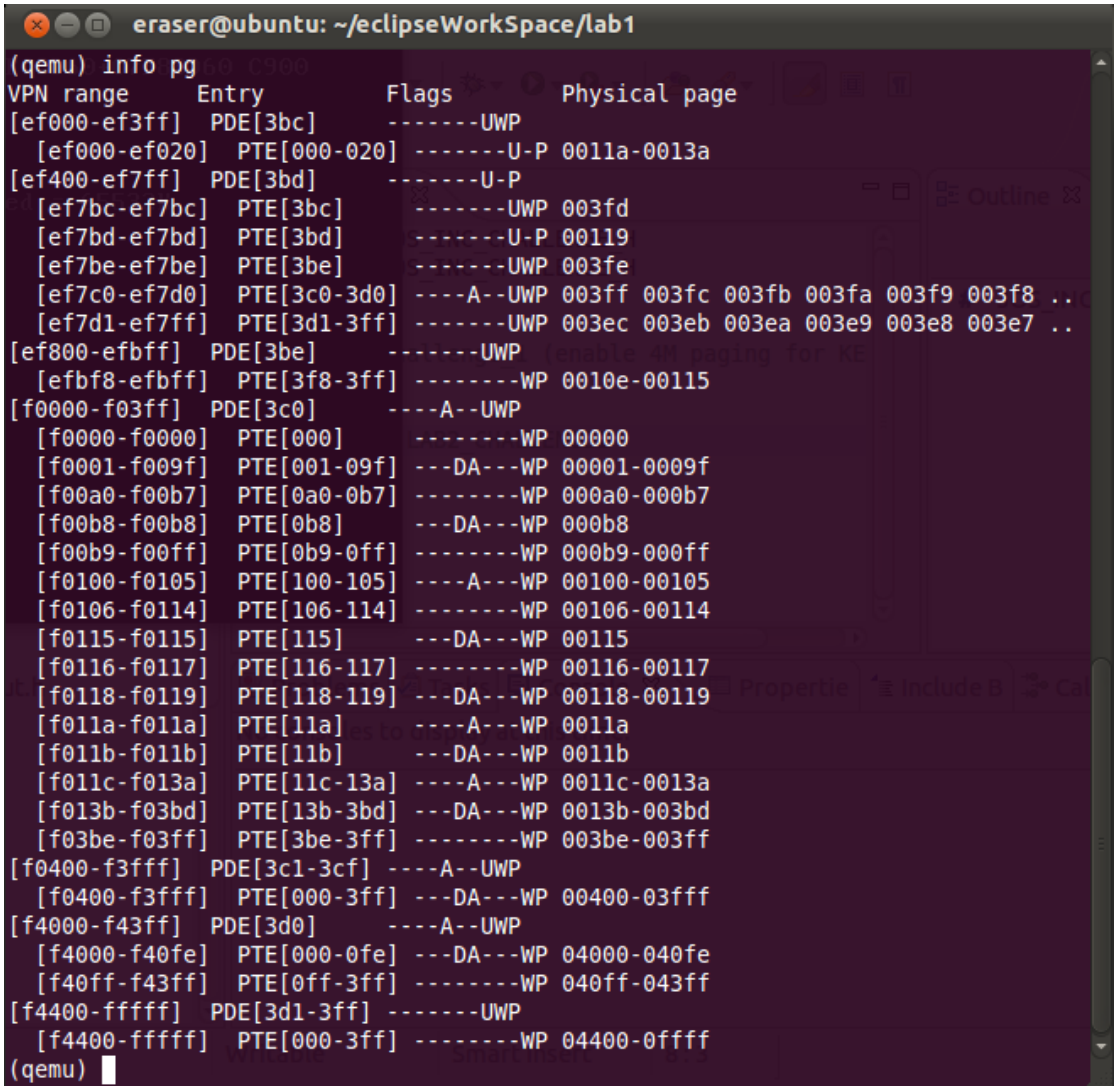
在完成上面的函数以后，Lab2 的 exercise 部分完成，能够通过所有的 check（说得轻松实际上还是调了不少时间，有时候是忘记转换成物理地址，有时候是不小心映射错了）。最后附上 make grade 的截图：

```
Physical page allocator: OK (1.5s)
Page management: OK (1.6s)
Kernel page directory: OK (1.6s)
Page management 2: OK (1.6s)
Score: 70/70
```

Questions:

2. 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:

这是让我们填写当前的页目录映射情况，我们使用 QEMU monitor commands 的 info pg 命令，就能够输出页目录和页表的当前映射情况，如下：



3. (From Lecture 3) 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?

要防止用户程序访问超过 User 部分（ULIM）的虚拟地址空间，以防它无意或有意地就修改了 Kernel 部分的数据，导致系统出错甚至崩溃。在 x86 中，在页目录项 PDE 和页表项 PTE 设置了相应的权限 U/S, R/W，使那些用户不能使用的内存空间的地址，将不能通过页式转换将线性地址转换得到物理地址或不能写入。

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

我本来觉得是 4GB 的……但是似乎 zhangchi 的报告说得有道理：由于在映射 pages 的时候，系统分配的空间是 PTSIZE（4M），那么可以计算 pages 数组中的元素个数为 1/3M（每个 Page 结构的大小是 12 字节），而一个 pages 数组中的元素对应 4KB 物理页，因此可以支持的总物理内存大小是 1/3M * 4KB = 1.33GB。这道题说明了，我们考虑问题要考虑瓶颈。

5. 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?

我们需要 1 张页目录，大小是 1 张物理页；1024 张页表，每张大小是 1 张物理页。但是实际上，我们注意到下面这行代码：

```
    //////////////////////////////////////////////////////////////////////
    // Recursively insert PD in itself as a page table, to form
    // a virtual page table at virtual address UVPT.
    // (For now, you don't have understand the greater purpose of the
    // following two lines.)

    // Permissions: kernel R, user R
    kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
```

这是一个自映射机制，也就是页目录将自己当成一张页表……所以总的物理内存开销是 1 + 1024 – 1 = 1024 张 = 4MB。

若我们希望减小管理内存带来的开销，那么我们就要减少页表张数，因此，我们可以开更大的页，如 4M 页。每开一个 4M 页，可以减少使用 1 张页表。

6. 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?

从标签 relocated 后，执行代码的 EIP 都是在 KERNBASE 之上，即需要通过页式转换来获得物理地址。

在设置 CR0 以后，我们 EIP 确实仍然在低地址运行，并执行了后面这两条指令：

```
    # Turn on paging.
    movl    %cr0, %eax
    orl $(CR0_PE|CR0_PG|CR0_WP), %eax
    movl    %eax, %cr0

    # Now paging is enabled, but we're still running at a low EIP
    # (why is this okay?).  Jump up above KERNBASE before entering
    # C code.
    mov $relocated, %eax
    jmp *%eax
relocated:
```

这两条指令为什么还能够正确执行呢？我们再看看 kern/entrypgdir.c，看到下面的注释，现在能看懂了！

```
// The entry.S page directory maps the first 4MB of physical memory
// starting at virtual address KERNBASE (that is, it maps virtual
// addresses [KERNBASE, KERNBASE+4MB) to physical addresses [0, 4MB)).
// We choose 4MB because that's how much we can map with one page
// table and it's enough to get us through early boot.  We also map
// virtual addresses [0, 4MB) to physical addresses [0, 4MB); this
// region is critical for a few instructions in entry.S and then we
// never use it again.
```

原来，lab1 的这个简单的页表，不仅仅将虚拟地址[KERNBASE, KERNBASE+4MB)映射到物理地址[0, 4MB)，它还将虚拟地址[0, 4MB)映射到物理地址[0, 4MB)，这非常重要！有了这个映射，跳转到高地址前的指令才能够正确执行，虽然才两条指令而已。

Challenge1 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. Make sure you design the kernel to use this optimization only on processors that support it!

这个 challenge 的意思是：我们在将 KERNBASE 开始的虚拟地址映射到整块物理内存的时候，由于物理内存较大，导致映射消耗了较多页表的空间开销。我们可以在相应页目录项 PDE 中打开 PTE_PS 位，从而为该映射开启 4M 页机制，不仅节省页表，还能够提高 TLB 命中率。在开启 4M 页前，我们还要注意检测处理器是否支持 4M 页。

Intel 手册中，对 4M 页映射机制的描述是一张图：

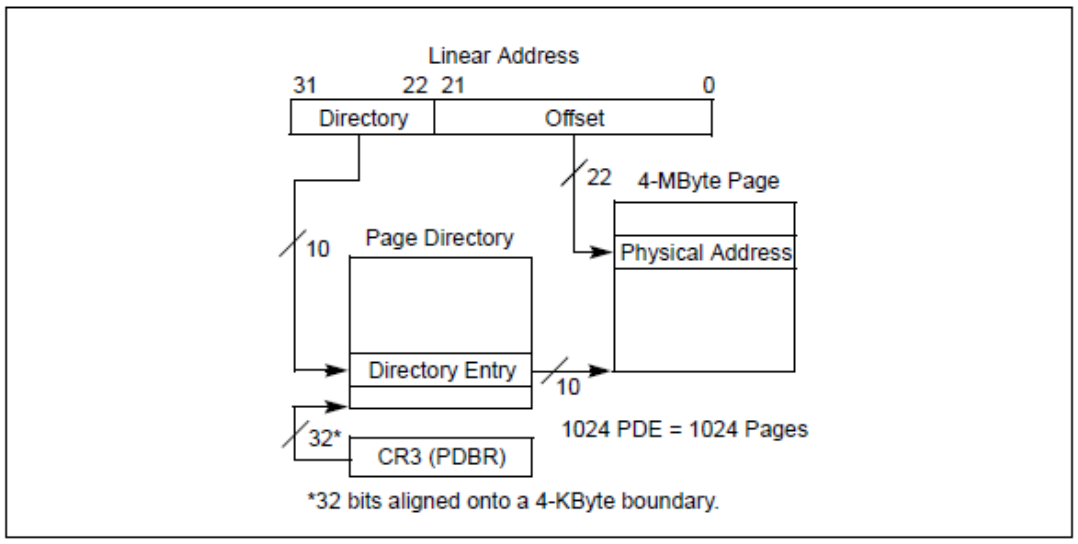


Figure 3-13. Linear Address Translation (4-MByte Pages)

从图上我们注意到，和 4K 页不一样，一个 32 位线性地址被划分为 10 + 22 位：高 10 位是对应页目录项 PDE 在页目录 pgdir 中的下标，从页目录项 PDE 中我们立即得到物理页的起始地址（实际指定了地址的高 10 位，第 22 位全 0）；低 22 位是数据在物理页内的偏移。

通过设置CR4中的PSE位，同时设置页目录项PDE的PS位，我们就可以使用该PDE指向一个4M的物理地址空间。此时，系统是4M页和4K页混合的，那些没有设置PS位的页目录项PDE使用方法和之前的4K页完全一样。

基于上面的知识，这个 challenge 就能实现出来，我主要修改了 boot_map_region(), mem_init(), pgdir_walk()这三个函数。首先，我们要认识到，我们将 KERNBASE 映射到整块物理内存，这片物理地址本身就已经是连续的，因此这和 page_alloc()这些函数无关，只有在映射的时候才需要处理。而实际上，这并不比 4K 页难处理，映射的基本原理完全是类似的，我们只需要设置页目录项 PDE。因此，在 boot_map_region()函数中的实现如下：

```c
static void boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm){
    #define PGSIZE_4M PGSIZE*1024
    uintptr_t offset;
    pte_t *pte;
    if(perm & PTE_PS){
        // 4MB paging
        for(offset = 0; offset < size; offset += PGSIZE_4M){
            pte = &pgdir[PDX(va)];
            *pte = pa | perm | PTE_P | PTE_PS;
            pa += PGSIZE_4M;
            va += PGSIZE_4M;
        }
    }
    else{
        // 4KB paging
        for(offset = 0; offset < size; offset += PGSIZE){
            pte = pgdir_walk(pgdir, (void *)va, 1);
            *pte = pa | perm | PTE_P;
            pa += PGSIZE;
            va += PGSIZE;
        }
    }
    return;
}
```

在 mem_init()中调用 boot_map_region()的代码如下：

```c
boot_map_region(kern_pgdir, KERNBASE, 1 << 28, 0, PTE_W | PTE_P | PTE_PS);
```

为了让 pgdir_walk()更具有鲁棒性，我加上了几行代码，处理 4M 页的情形：

```c
// for 4MB paging
if(*pgdir_entry & PTE_PS){
    return pgdir_entry;
}
```

这样，采用 4M 页将 KERNBASE 开始的虚拟地址映射到整块物理内存的功能就实现了。（其实还要另外修改 check_va2pa 的代码，否则 check 不通过，在此不赘述。）

实验结果如下：

```
(qemu) info pg
VPN range     Entry         Flags        Physical page
[ef000-ef3ff]  PDE[3bc]     -------UWP
  [ef000-ef020]  PTE[000-020] -------U-P 0011a-0013a
[ef400-ef7ff]  PDE[3bd]     -------U-P
  [ef7bc-ef7bc]  PTE[3bc]     -------UWP 003fd
  [ef7bd-ef7bd]  PTE[3bd]     -------U-P 00119
  [ef7be-ef7be]  PTE[3be]     -------UWP 003fe
  [ef7c0-ef7d0]  PTE[3c0-3d0] --SDA---WP 00000 00400 00800 00c00 01000 01400 ..
  [ef7d1-ef7ff]  PTE[3d1-3ff] --S-----WP 04400 04800 04c00 05000 05400 05800 ..
[ef800-efbff]  PDE[3be]     -------UWP
  [efbf8-efbff]  PTE[3f8-3ff] --------WP 0010e-00115
[f0000-f43ff]  PDE[3c0-3d0] --SDA---WP 00000-043ff
[f4400-fffff]  PDE[3d1-3ff] --S-----WP 04400-0ffff
(qemu)
```

我们要将这个截图和 Question2 中的截图对比一下，不难发现，结果变短了！没错，在上面这张截图中，我们注意到红色方框标出的部分，开启了 PS 位，因此它是一个 4M 页的映射，将[0xf0000000-0xffff000]映射到[0x00000000-0x0fffffff]（说明，截图中的地址全部都省略了最低位的 3 个 0，因为 16^3=4096 恰为一页）。

我们还发现，黄色框中出现了 PS 位开启现象，这是怎么回事？原来，这是页目录自映射造成的，页目录把它自己当成了一张页表，里面的内容当作页表项。

最后，关于检测硬件是否支持 4M 页，这部分据说很难，所以就不做了。呃。

Challenge2 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!)

完成上面的所有功能后，这个 challenge 算是很简单的，就是在控制台实现相应的功能，书写 mon_*()函数。由于比较简单，代码仅简述流程，不再给出。

第一个部分是让我们实现根据给定虚拟地址（区间）显示其映射到物理地址的情况，包括映射到哪里，以及映射权限。首先，得到虚拟地址 vafrom 和 vato，调用 strtol()函数将参数字符串转换成整数，并进行一定的安全性检查，如 vafrom 是否大于 vato 等。其次，通过一个循环调用 pgdir_walk 遍历 vafrom 到 vato 的区间即可。难点有两个。第一个是如果开启 4M 页，需要安排一些分支语句。第二个是 vafrom 和 vato 是否要对齐的问题，我没有要求用户自己对齐地址，而是将 vafrom ROUNDDOWN 到 PGSIZE 的倍数，vato 不作处理。事实上，就算 vato 不处理，vafrom 增长到 vato 上方第一个 PGSIZE 的倍数就会停止循环，包含了 vato.

实验结果截图如下：

```
K> showMappings 0xefbf8000 0xefbff000
0xefbf8000 - 0xefbf9000(4K)  ......  0x10f000  kern: read/write
0xefbf9000 - 0xefbfa000(4K)  ......  0x110000  kern: read/write
0xefbfa000 - 0xefbfb000(4K)  ......  0x111000  kern: read/write
0xefbfb000 - 0xefbfc000(4K)  ......  0x112000  kern: read/write
0xefbfc000 - 0xefbfd000(4K)  ......  0x113000  kern: read/write
0xefbfd000 - 0xefbfe000(4K)  ......  0x114000  kern: read/write
0xefbfe000 - 0xefbff000(4K)  ......  0x115000  kern: read/write
```

第二个部分是让我们实现能够在控制台设置 mapping 的权限。我不知道这个功能到底是拿来干嘛的，难道用户可以任意修改页目录页表么？反正这个功能也是用 pgdir_walk()函数就能实现。效果截图如下，我显示了修改前后的页表情况。

```
K> showMappings 0xefbf8000 0xefbff000
0xefbf8000 - 0xefbf9000(4K)  ......  0x10f000  kern: read/write
0xefbf9000 - 0xefbfa000(4K)  ......  0x110000  kern: read/write
0xefbfa000 - 0xefbfb000(4K)  ......  0x111000  kern: read/write
0xefbfb000 - 0xefbfc000(4K)  ......  0x112000  kern: read/write
0xefbfc000 - 0xefbfd000(4K)  ......  0x113000  kern: read/write
0xefbfd000 - 0xefbfe000(4K)  ......  0x114000  kern: read/write
0xefbfe000 - 0xefbff000(4K)  ......  0x115000  kern: read/write

K> setMappingPermission 0xefbfa000 00000100
0xefbfa000 - 0xf03fe000  ......  kern: read/write ---> user: read-only

K> showMappings 0xefbf8000 0xefbff000
0xefbf8000 - 0xefbf9000(4K)  ......  0x10f000  kern: read/write
0xefbf9000 - 0xefbfa000(4K)  ......  0x110000  kern: read/write
0xefbfa000 - 0xefbfb000(4K)  ......  0x111000  user: read-only
0xefbfb000 - 0xefbfc000(4K)  ......  0x112000  kern: read/write
0xefbfc000 - 0xefbfd000(4K)  ......  0x113000  kern: read/write
0xefbfd000 - 0xefbfe000(4K)  ......  0x114000  kern: read/write
0xefbfe000 - 0xefbff000(4K)  ......  0x115000  kern: read/write

K> _
```

第三个部分是给定物理地址或虚拟地址，让我们输出相应物理地址存储的内容。这就是 QEMU monitor commands 中的 x 和 xp 命令。给定虚拟地址的情况，很简单。直接将这个地址转化成一个 c 指针（类型）按照 16 进制整数输出那里的内容即可。若给定物理地址，这个就要转化一下。还记得我们前面将这个物理内存 map 到 KERNBASE 这个地方么，另外在 2.2 中我们也提到 JOS 有时需要根据物理地址获得虚拟地址。因此，我们只要把那个物理地址加一个 KERNBASE，就可以当做虚拟地址处理。

实验结果如下（我演示了给定物理地址 p 和虚拟地址 v 的情况，同时和 QEMU monitor commands 中的结果对比，发现二者是完全一样的）：

给定物理地址 dumpmem：

```
K> dumpmem p 0x00000010 20

0xf0000010: 0xf000ff53 0xf000ff53 0xf000ff53 0xf000ff53
0xf0000020: 0xf000fea5 0xf000e987 0xf000e97a 0xf000e97a
0xf0000030: 0xf000e97a 0xf000e97a 0xf000ef57 0xf000e97a
0xf0000040: 0xc00083f9 0xf000f84d 0xf000f841 0xf000e3fe
0xf0000050: 0xf000e739 0xf000f859 0xf000e82e 0xf000efd2

K> _
```

和 QEMU monitor commands 结果比较：

```
(qemu) xp/20x 0x00000010
0000000000000010: 0xf000ff53 0xf000ff53 0xf000ff53 0xf000ff53
0000000000000020: 0xf000fea5 0xf000e987 0xf000e97a 0xf000e97a
0000000000000030: 0xf000e97a 0xf000e97a 0xf000ef57 0xf000e97a
0000000000000040: 0xc00083f9 0xf000f84d 0xf000f841 0xf000e3fe
0000000000000050: 0xf000e739 0xf000f859 0xf000e82e 0xf000efd2
(qemu)
```

给定虚拟地址 dumpmem：

```
K> dumpmem v 0xf010000c 20

0xf010000c: 0x7205c766
0xf0100010: 0x34000004 0x7000b812 0x220f0011 0xc0200fd8
0xf0100020: 0x0100010d 0xc0220f80 0x10002fb8 0xbde0fff0
0xf0100030: 0x00000000 0x117000bc 0x0002e8f0 0xfeeb0000
0xf0100040: 0x83e58955 0x6cb818ec 0x2df01199 0xf0119300
0xf0100050: 0x08244489 0x042444c7 0x00000000

K> info pg
```

和 QEMU monitor commands 结果比较：

```
(qemu) x/20x 0xf010000c
f010000c: 0x7205c766 0x34000004 0x7000b812 0x220f0011
f010001c: 0xc0200fd8 0x0100010d 0xc0220f80 0x10002fb8
f010002c: 0xbde0fff0 0x00000000 0x117000bc 0x0002e8f0
f010003c: 0xfeeb0000 0x83e58955 0x6cb818ec 0x2df01199
f010004c: 0xf0119300 0x08244489 0x042444c7 0x00000000
(qemu)
```

3.3 Address Space Layout Alternatives

Challenge3 Write up an outline of how a kernel could be designed to allow user environments unrestricted use of the full 4GB virtual and linear address space. Hint: the technique is sometimes known as "follow the bouncing kernel." In your design, be sure to address exactly what has to

happen when the processor transitions between kernel and user modes, and how the kernel would accomplish such transitions. Also describe how the kernel would access physical memory and I/O devices in this scheme, and how the kernel would access a user environment's virtual address space during system calls and the like. Finally, think about and describe the advantages and disadvantages of such a scheme in terms of flexibility, performance, kernel complexity, and other factors you can think of.

这里提到了一种不同于 JOS 的安排用户和内核虚拟地址空间布局的机制，即用户可以完全使用 4GB 虚拟地址空间，不再被 ULIM 严格限制只能使用 4GB 的低地址部分；同时内核在虚拟地址空间中，也将不一定是连续的，所谓“地址跳跃的内核”；同时还要能够实现保护和隔离。

Google 了一下发现似乎没有什么可以参考的资料。对于这个 challenge 中提到的问题，我似乎并没有太多的概念，除了知道一个特权级别以外，并不了解 CPU 或 kernel 具体是怎么工作的。在 lab1 和 lab2 中也没有接触相关的内容。恳请老师助教课堂上解释一下？

我的直觉是，既然不连续就不不连续啊，反正只要在 map 内核地址空间的时候，能够正确地 map 过去不就可以了吗，权限仍然是由页表和页目录相应场位设定。至于带来的问题，显然 Kernel 部分的地址空间不连续了，那么我们就不能在某些场合用“加/减 KERNBASE”来进行虚拟和物理地址的转换，带来了 Kernel 的复杂性，也没有那么方便了。

Challenge4 Since our JOS kernel's memory management system only allocates and frees memory on page granularity, we do not have anything comparable to a general-purpose malloc/free facility that we can use within the kernel. This could be a problem if we want to support certain types of I/O devices that require physically contiguous buffers larger than 4KB in size, or if we want user-level environments, and not just the kernel, to be able to allocate and map 4MB superpages for maximum processor efficiency. (See the earlier challenge problem about PTE_PS.)

Generalize the kernel's memory allocation system to support pages of a variety of power-of-two allocation unit sizes from 4KB up to some reasonable maximum of your choice. Be sure you have some way to divide larger allocation units into smaller ones on demand, and to coalesce multiple small allocation units back into larger units when possible. Think about the issues that might arise in such a system.

这个 challenge 的意思是，希望让内核能够支持 4K 页到某一值之间的任意 2 的幂数值的大小的页面。注意，这里的页面指的是连续的物理页。

这个有点像伙伴系统。如果我们选择支持 4K~4M 页，那么我们就要能够分配 4K~4M 之间的任意 2 的幂数值的大小的连续物理页框：如果没有合适大小的页框，要对更大的页框进行剪裁；如果页框被归还回来，要主动将两个连续的页框块合并为一个较大的页框块。

在 JOS 中，空闲页框是使用 page_free_list 来管理的，它只能够申请大小为 4KB 的连续物理页框，如果超过 4KB，不能保证其地址是连续的。因此，实现上述功能要从空闲页框管理入手。我们可以把所有的空闲页框分为 11 组，用 11 个链表管理，每个链表链接的空闲页框大小为 4KB，8 KB，16 KB，32 KB，64 KB，128 KB，256 KB，512 KB，1MB，2MB，4MB。需要注意要让每个页框的起始物理地址为该页框大小的整数倍，这样我们根据起始地址就能

够知道这个页框多大。

假设现在我们要申请一个 128KB 的连续物理地址，我们首先从 128KB 页框链表查找空闲页框，如果有空闲的，那就返回之；如果没有空闲的，就从 256KB 页框链表中找一个空闲的，将它分成 2 个 128KB 的页框，其中一个分配出去，另一个移入 128KB 空闲页框链表。如果 256KB 页框链表没有空闲页框，那么就继续往上找，如果仍然没有就返回错误。

当回收页框的时候，我们只要检查页框地址及其前后地址的大小相同的页框（如果有）是否也是空闲的，如果空闲，还需要合并后放入高一级的页框链表。

至于虚拟地址的映射，是否要和物理页框大小匹配呢？我觉得这是不必要的。地址连续的物理页框，对虚拟地址映射的步骤可以是透明的；也就是说我们在映射的时候，根本不用管这个页框是多大，是不是连续，我们就全都按照 4K 页映射好了。就算希望在 PDE 或 PTE 中体现所映射连续物理页框的大小，我们可以利用那些保留位编码后作适当的标记信息，在此不作赘述。

暂且吹吹牛，这个 challenge 就“说”到这里。

内容三：遇到的困难以及解决方法

困难一：耐心看代码，关注宏展开

在这次实习中，用到了很多宏，这是逃不掉的。以前我看代码，遇到大写字母基本上随便看看知道一个大概就不管了。但是在这次实习中，宏的确帮了很大的忙，它们的作用实在是太大了，以致我不得不耐心地跳转到宏定义，去看看它是怎么实现或者有什么说明，以了解正确用法。有时候遇到嵌套的宏，在必要时更需要耐心去一层一层剥开它的内容。

困难二：英语太差了，看不懂题目

在 exercise1 中的 mem_init()，我们只需要走到调用 check_page_free_list(1)处即可（only up to the call to check_page_free_list(1)）。可惜悲剧的是，我当时没有理解清楚题目的这个意思，一直往下做，做啊做啊做啊，结果发现后面一点也看不懂，根本不知到在干什么。直到我实在不想写的时候，我才意识到题目中“up to”原来是“只用做到这里”这个意思，而不是“It's up to you”的那个意思。囧。读材料，读题目，理解清楚真的很重要。

困难三：在 check_page_free_list()出现了一个 panic

我在 exercise1 的 check_page_free_list()检查的时候遇到了第一个 panic：

```
kernel panic at kern/pmap.c:536: assertion failed: nfree_extmem > 0
```

这是我第一次遇到的、需要调试的比较大的错误（尽管后来习惯了）。错在哪里？根据错误信息，应该是 free_list 进行初始化时出错了，因为 nfree_extmem 即空闲列表中处在物理内存 extend memory 的页面数居然为 0。因此我打印出 kern_pgdir、pages 物理内存地址、申请物理内存字节数、每次 boot_alloc 分配后 nextfree 的地址、物理页面数等信息……观察发现，原来是 page_init 函数中对于已分配页面的上限计算错了！这个上限我们知道是刚刚分配完 pages 数组后的可用地址 nextfree 所在页面，这个页面的编号该怎么计算呢？因此，我们要知道 nextfree 的物理地址……再次仔细阅读邵老师的资料，发现有一句很重要的话“始终记住：kernel 得逻辑地址是从 KERNBASE=0xf0000000 开始的！”。恍然大悟，nextfree 是一个逻辑地址（此为线性地址），要转换成物理地址啊……

自从这次以后，调试就成了比较自然的事情，不会像 lab1 那么顺利了。另外，我发现打印一些数据到屏幕，观察它们是否符合预期，是一个非常好的调试方法。

困难四：还是没理解好材料……

pgdir_walk 中，有一句代码应该是：

```
*pgdir_entry = new_pgtbl_pa | PTE_W | PTE_U | PTE_P;
```

这里是页目录项，我当时没有理解好，没有加 PTE_W 导致 check 的时候又出错了。事实上，应该让页目录项 PDE 开最大权限，而让页表项 PTE 开合适的权限，即让页表项再来限制。

这个也让我调了半天，调试方法也是打印到屏幕的办法。

困难五：报告怎么写了那么久 T.T

我都写到快要崩溃了。不行，下次我要改变写报告的策略了……

内容四：收获和感想

主要的收获是，对“访问内存”的过程、虚拟地址映射物理地址、页目录页表等知识理解得更加深刻了。以前对于虚拟页式内存管理，还只是停留在背书考试阶段。这次自己实现了页目录页表，虽然不难，但是对整个地址转换、访问物理内存的过程都有了一个深刻的体会。

感觉 lab2 没有老师说的那么难，似乎在吓人。

要保持头脑清醒。各种概念区分清楚，对于多步骤的事情，要清楚自己现在处在那个步骤，手里拿着什么东西准备干什么事情，不要乱了套。例如，pgdir_walk().

代码调试能力的提高。主要是输出一些数据啦，看看是否符合预期神马的，关键是耐心。调试的时候如果看了 5 遍都觉得“咦，我觉得没问题啊”，那么最好现在去干另一件事情，明天换一个脑子再来调。要不然就和同学讨论啦。

最后，我要感谢刘驰同学，他耐心地指导了我很多不明白的地方。同时通过讨论，他纠正了我很多我本来理解错题目的地方，让我少走弯路。他写作业写得好快啊，一直是我追赶的目标和努力的方向。

然后，我要感谢林舒同学，他在 1 秒钟内告诉我 kernel stack 为什么分为两部分。

我还要感谢王仲禹同学，他给了我不少有用的经验。感谢刘智猷同学，我向他询问了关于如何检查硬件是否支持 PSE 的问题，他耐心地回答了我的问题，还告诉我应该去 Google 搜索什么关键词。然而由于需要阅读材料的行数和最终书写的代码行数的“比”太大了，导致我决定还是不要挑战这个极限了。

内容五：对课程的意见和建议

1、希望老师对我们提交的报告中的问题（是不是写得太冗长太罗嗦了，有哪些内容做错了，有没有更好的方法）有所具体反馈。我觉得这样我们获得的提升更大，同时能够增进语言表达能力。

2、出于好奇心，我很想知道其他同学，或者标准答案是怎么给出 challenge 的解决方案。例如，这次实习的 challenge3，里面很多概念我都不理解……（的确，也是我的错，我应该在那节课去问助教的……可惜……可惜当时我还没有开始写报告啊，我怎么知道我不知道那个问题……T.T）

内容六：参考文献

[1] MIT 6.828 课程资料，MIT，2011；
[2] 邵志远，JOS 实验讲义第四章，华中科技大学，2010；
[3] 王仲禹，系统的启动和初始化实习报告，北京大学，2011；
[4] 张驰，操作系统 JOS 第二次实习报告，北京大学，2011；
[5] 百度百科：伙伴系统。