[TOC]

练习1

i386_detect_memory

识别内存中有多少空间剩余以及base memory,extended memory

> 不太理解哪来的额外内存

所有的单位都是页数（一页4KB）

boot_alloc

> 暂时没弄懂end以及kernel bss段是什么意思

当做页分配器，维护静态变量nextfree，使它始终指向一个可用的虚拟内存地址，按照注释要求，每次申请变量空间时都需要修改nextfree的值 初始化时boot_alloc(0)返回的就是可用虚拟内存开始的地址

ROUNDUP

inc/types.h中定义 作用是取整，用来保持变量是4k的倍数（应该是向上取整）

mem_init

struct PageInfo

memlayout.h中定义

```
struct PageInfo {
        // Next page on the free list.
        struct PageInfo *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;
};
```

pageInfo主要有两个变量: pp_link表示下一个空闲页，如果pp_link=0,则表示这个页面被分配了，否则就表示未被分配，是空闲页。 pp_ref表示页面被引用的次数，如果为0，表示是空闲页。

第一次补充的代码

为pages申请地址空间，并初始化为0

memset

string.h中定义 `void * memset(void *dst, int c, size_t len);` 函数解释：将dst中当前位置后面的len个字节用c替换并返回 dst 。 作用是在一段内存块中填充某个给定的值，它是对较大的结构体或数组进行清零操作的一种最快方法。

page_init

初始化pages数组以及page_free_list，将已经被系统使用的剔除

> 挺坑的，一开始不知道boot_alloc中end那个还是虚拟地址，要减去KERENBASE转化为物理地址 利用 boot_alloc 函数来找到第一个能分配的页面

IOPHYSMEM && EXTPHYSMEM

memlayout.h中定义

```
// At IOPHYSMEM (640K) there is a 384K hole for I/O.  From the kernel,
// IOPHYSMEM can be addressed at KERNBASE + IOPHYSMEM.  The hole ends
// at physical address EXTPHYSMEM.
#define IOPHYSMEM       0x0A0000
#define EXTPHYSMEM      0x100000
```

KERNBASE

memlayout.h中定义

```
// All physical memory mapped at this address
#define KERNBASE        0xF0000000
```

page_alloc

申请一页的空间，需要对page_free_list进行更新，以及对页进行初始化

ALLOC_ZERO

pmap.h中定义

```
enum {
        // For page_alloc, zero the returned physical page.
        ALLOC_ZERO = 1<<0,
};
```

> 没太懂这个值的意义，以及把1左移0位的意义

page2kva

pmap.h中定义

```
static inline void*
page2kva(struct PageInfo *pp)
{
        return KADDR(page2pa(pp));
}

/* This macro takes a physical address and returns the corresponding kernel
 * virtual address.  It panics if you pass an invalid physical address. */
#define KADDR(pa) _kaddr(__FILE__, __LINE__, pa)
```

KADDR：将物理地址转换为虚拟地址 page2kva：将PageInfo结构转换为相应的虚拟地址

page_free

对申请的空间进行释放，同时对page_free_list和收回的page进行修改

*原本加了个**pp->ref**是否等于**0**的判断，发现结果与自己想的相反，后来查了才知道好像前面的代码在调用**page_free**的时候会提前把**pp_ref**和**pp_link**修改了，但没仔细去看过这部分代码*

check_page_alloc

检查page_alloc是否成功，成功则 `cprintf("check_page_alloc() succeeded!\n");`

assert

assert.h中定义

```
#define assert(x)               \
        do { if (!(x)) panic("assertion failed: %s", #x); } while (0)
```

> 不太懂panic的作用,貌似是个系统中断？感觉语法和printf差不多

```
void _panic(const char*, int, const char*, ...) __attribute__((noreturn));

#define panic(...) _panic(__FILE__, __LINE__, __VA_ARGS__)
```

练习2

Intel理解 对于Intel CPU 来说，分页标志位是CR0寄存器的第31位，为1 表示使用分页，为0 表示不使用分页。CPU 在执行代码时，自动检测CR0寄存器中的分页标志位是否被设定，若被设定就自动完成虚拟地址到物理地址的转换。

练习3

无法在运行着QEMU软件的terminal里面通过输入ctrl-a c，切换到监控器里，上网查询得知在lab目录下面输入：

```
qemu-system-i386 -hda obj/kern/kernel.img -monitor stdio -gdb tcp::26000 -D
qemu.log
```

可以在linux的terminal里进入到监控器。

练习4

padir_walk

主要是用过一个给定的虚拟地址va和pgdir(page director table的首地址), 返回va所对应的pte(page table entry)中的页表项。当va对应的页表存在时，只需要直接按照页面翻译的过程给出页表项的地址；当va对应的页表没有被创建的时候，就需要手动的申请并创建页面。 最后需要返回的是虚拟地址，而不能直接返回页表项的物理地址。因为程序里面只能执行虚拟地址，给出的物理地址也会被当成是虚拟地址，会引发错误。

pte_t,pde_t

memlayout.h中定义

```
typedef uint32_t pte_t;
typedef uint32_t pde_t;
```

mmu.h中有用的宏定义

PDX

取地址31-22bit，用作pd索引

```
// page directory index
#define PDX(la)         ((((uintptr_t) (la)) >> PDXSHIFT) & 0x3FF)
```

PTE_ADDR

将页目录项的后12位（flag 位）全部置 0 获得对应的页表项物理地址

```
// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((physaddr_t) (pte) & ~0xFFF)
```

PTX

取地址21到12bit，用作pt索引

```
// page table index
#define PTX(la)     ((((uintptr_t) (la)) >> PTXSHIFT) & 0x3FF)
```

PGOFF

取地址11-0bit，用作页间索引

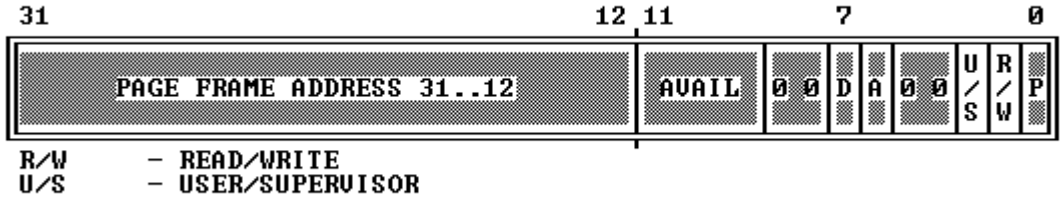
```
// offset in page
#define PGOFF(la)       (((uintptr_t) (la)) & 0xFFF)
```

页目录和页表权限位

```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present 存在
#define PTE_W           0x002   // Writeable 可写入
#define PTE_U           0x004   // User 用户可操作
```

> 关于权限位如果有不懂的地方看下图

Figure 6-10. Protection Fields of Page Table Entries



boot_map_region

> UTOP memlayout.h中定义 `UTOP,UENVS ------> +------------------------------+ 0xeec00000`

调用pgdir_walk，把虚拟地址和物理地址的映射放到页的首地址对应的页目录中,并设置权限等信息

page_lookup

根据虚拟地址，找到相应的page(pageInfo)的位置

pa2page

将虚拟地址转换为对应的PageInfo结构

page_remove

虚拟地址va和物理页的映射关系删除

tlb_invalidate

通知tlb失效。tlb是个高速缓存，用来缓存查找记录增加查找速度

```
// Invalidate a TLB entry, but only if the page tables being
// edited are the ones currently in use by the processor.
//
void
tlb_invalidate(pde_t *pgdir, void *va)
{
        // Flush the entry only if we're modifying the current address
space.
        // For now, there is only one address space, so always invalidate.
        invlpg(va);
}
```

具体解释在这个博客里有提到 https://blog.csdn.net/cinmyheart/article/details/39994769

page_decref

减少页的被引用次数，如果页的被引用次数为0就调用page_free将页面释放

```
// Decrement the reference count on a page,
// freeing it if there are no more refs.
//
void
page_decref(struct PageInfo* pp)
{
        if (--pp->pp_ref == 0)
                page_free(pp);
}
```

page_insert

把指定的物理地址和虚拟地址之间的联系放到页的首地址（如果虚拟地址已经有映射，删除之前的映射，添加新的）

练习5

mem_init

> 不太懂bootstack在哪被赋值，也就pamp.h中有提到 `extern char bootstacktop[], bootstack[];`

PTSIZE

mmu.h中定义 `#define PTSIZE (PGSIZE*NPTENTRIES) // bytes mapped by a page directory entry` PTSIZE 被定义为页目录项映射的bytes，一个页目录中有1024个页表项，每个页表项可映射一个物理页，每个物理页为4K，故PTSIZE为 4MB。

PADDR

将虚拟地址转化为对应的物理地址

```
/* This macro takes a kernel virtual address -- an address that points
above
 * KERNBASE, where the machine's maximum 256MB of physical memory is mapped
--
 * and returns the corresponding physical address.  It panics if you pass
it a
 * non-kernel virtual address.
 */
#define PADDR(kva) _paddr(__FILE__, __LINE__, kva)
```

问题

1

> 变量x应该是什么类型，uintptr_t还是 physaddr_t?

虚拟地址，因为使用了指针，指针指向的都是虚拟地址。

2

> 假设下图描述的是系统的页目录表，哪些条目（行）已经被填写了？他们是怎么样进行地址映射的？他们所指向的位置在哪里？请尽可能完善这张表的内容。

通过参考memlayout.h文件可以得知: [KERNBASE, 4G]此部分映射实际物理内存中从0开始的中断向量表IDT、BIOS程序、IO端口以及操作系统内核等，该部分虚拟地址 - KERNBASE 就是物理地址。 [UPAGES, UVPT]在虚拟地址中的这段内存就是对应的在实际物理地址里pages数组的存储位置。 [UVPT, ULIM]对应系统页目录kern_pgdir存储的物理地址。

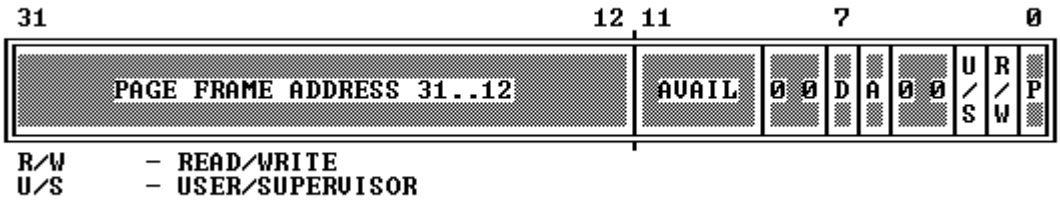
实际上虚拟内存中分配给用户的只有[0, ULIM)，其余的空间需要分配给操作系统。

3

> 为什么用户的程序不能读取内核的内存？有什么具体的机制保护内核空间吗？

只有分页机制，没有分段机制。通过对页表项的后12bit进行权限设置来限制对实际物理地址的操作。 把内核存储的地方权限设置为PTE_W | PTE_P，也就是把下图中寄存器的最后两位设置为1，把U/S位置为0

Figure 6-10. Protection Fields of Page Table Entries



参考

Intel i386手册

> The concept of privilege for pages is implemented by assigning each page to one of two levels:

```
Supervisor level (U/S=0) -- for the operating system and other systems
software and related data.
User level (U/S=1) -- for applications procedures and data.
```

4

> JOS操作系统可以支持的最大物理内存是多少？为什么?

通过前面练习1在mem_init中补全的代码可以知道，PageInfo结构存在pages数组中。练习5中给pages数组分配了PTSIZE，即4MB大小的空间。一个PageInfo结构有两个uint32_t类型的指针（pp_ref,pp_link）占8个字节，也就是最多存储512*1024个PageInfo类型的结构。一个PageInfo对应一个4K的物理页，因此JOS可以支持的最大物理内存是512*1024*4k=2GB。

5

> 如果我们的硬件配置了可以支持的最大的物理内存，那么管理内存空间的开销是多少？如何减少这种开销?

如果支持最大的物理内存，需要4MB空间存放所有的PageInfo，以及4KB的page_dir和1024个4KB的page_table，共需要8196KB。 可以通过扩大页的容量，使得相同的存储开销存储的页能够映射更多的物理地址空间来减少内存开销。

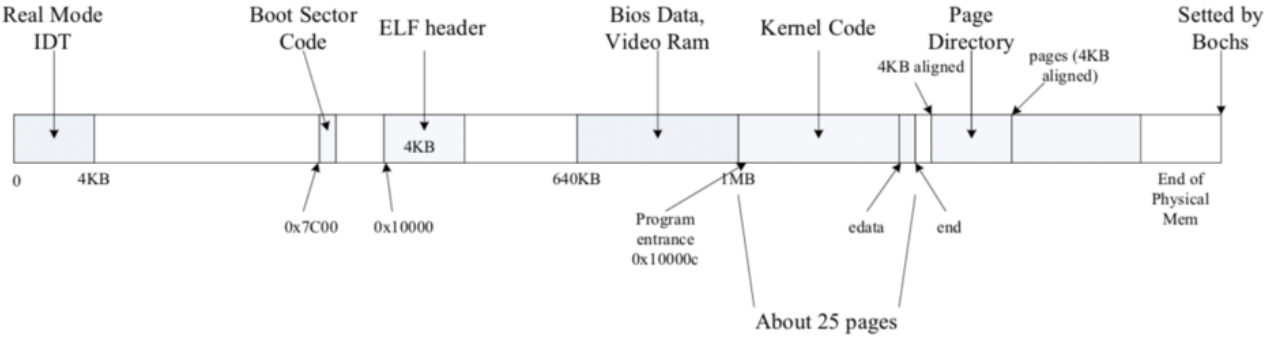
6

> 在什么时刻我们才开始在KERNBASE上运行EIP。当我们启动分页，当我们开始在KERNBASE上运行EIP之时，我们能否以低地址的EIP继续执行？这个过渡为什么是必须要的?

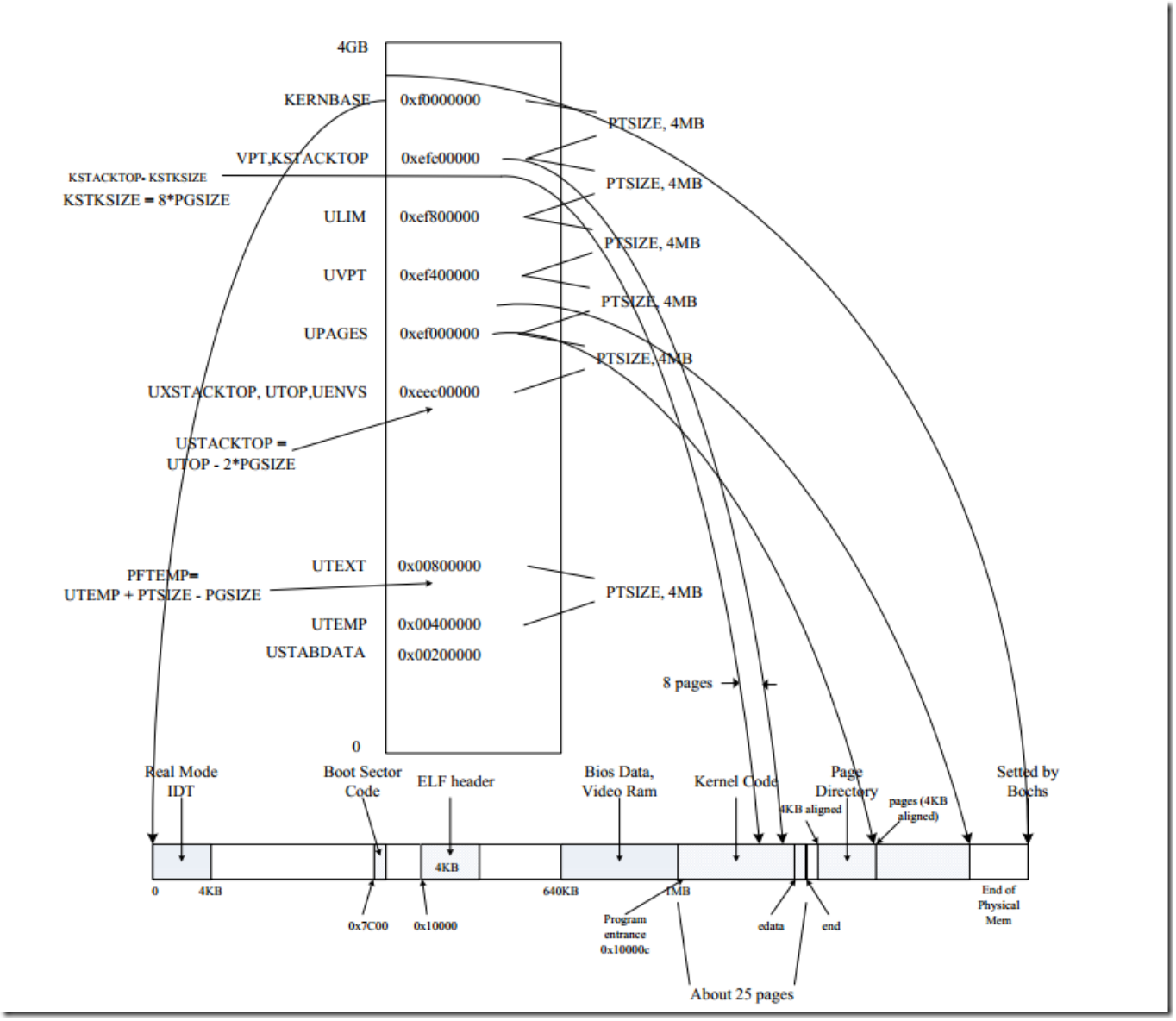
在entry.S文件中指令 jmp *%eax跳转完成后，就会重置EIP的值，置为寄存器eax中的值，而这个值是大于KERNBASE的，使EIP的值大于KERNBASE的值。由于在 kern/entrypgdir.c 中将 0-4MB 和 KERNBASE-KERNBASE + 4 MB 的虚拟地址都映射到了 0-4MB 的物理地址上，因此无论 EIP 在高位和低位都能执行。如果不支持低地址映射，那么就无法读取到0-4MB的内容了。

补充

内存布局



映射关系



> 后面的这些忘记摘自哪个博客了

JOS管理内存所用的数据结构

JOS使用Page数据结构来管理内存，一个Page代表一个PGSIZE（4K）大小的物理页面，Page数据结构的定义在memlayout.h中，共有两个域，第一个域是pp_link，指向下一个空闲Page结构的指针，第二个域是一个short整型，代表当前此物理页面的引用次数，若为0则是没有被引用也就是空闲页面。

其次使用free_page_list维护一个空闲物理内存的链表，free_page_list本身就是一个Page指针，然后通过Page结构体里面的pp_link域构成空闲链表。

再次一个Page代表4K，在pmap.c中的i386_memory_detect函数中检测内存后，使用总物理内存/4k得到所需要的Page数量，赋值给npages，换句话说npages代表所需Page结构体的数量。

最后所有Page在内存（物理内存）中的存放是连续的，存放于pages处，可以通过数组的形式访问各个Page，而pages紧接于end[]符号之上，end符号是编译器导出符号，其值约为kernel的bss段在内存（虚拟内存）中的地址+bss段的段长，对应物理和虚拟内存布局也就是在kernel向上的紧接着的高地址部分连续分布着pages数组。

JOS虚拟地址机制实现

最原始的地址叫做虚拟地址，根据规定，将前16位作为段选择子，后32位作为偏移。根据段选择子查找gdt/ldt，查到的内容替加上偏移，此时的地址就变成了线性地址。 线性地址前10位被称作页目录入口（page directory entry也就是pde），其含义为该地址在页目录中的索引，中间10位为页表入口（page table entry，也就是pte），代表在页表中的索引，最后12位是偏移。 当一个线性地址进入页式地址变换机制时，首先CPU从CR3寄存器里得到页目录（page directory）在主存中的地址，然后根据这个地址加上pde得到该地址在页目录中对应的项。无论是页目录的项还是页表的项均是32位，前20位为地址，后12位为标志位。当获取了相应的页目录项之后，根据前20位地址得到页表所在地址，加上偏移pte得到页表项，取出前20位加上线性地址本身的后12位组成物理地址，整个变换过程结束。 值得注意的是，这个过程完全是由硬件实现，在这个部分的实验中要做的是初始化并维护页目录与页表，当页目录与页表维护好了，然后使cr3装载新的页目录，一切就交由硬件去处理地址变换了。

JOS虚拟地址和线性地址一样

JOS的机制中，虽然使用段式地址变换，但和没使用完全一样，因为只定义了一个段，其长度为4G，换句话说，经过段式地址变换后的内容和之前完全一样，虚拟地址和线性地址完全一样。