OS_lab2

最近编辑过的 2011年3月21日

inc/mmu.h

1. 关于地址的概念：三个重要地址： 逻辑地址；线性地址；物理地址 。
逻辑地址为用户程序所使用的地址；
线性地址是操作系统根据x86段式地址转换将逻辑地址转换后的地址，具体来说：线性地址 = 逻辑地址 - KERNBASE
物理地址是操作系统地址转换系统将线性地址通过页表地址转换后得到的数据真实存储地址

2. mmu.h关于地址的一些组织方式说明

```
// A linear address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
//  \---------- PPN(la) ----------/
//
// The PDX, PTX, PGOFF, and PPN macros decompose linear addresses as shown.
// To construct a linear address la from PDX(la), PTX(la), and PGOFF(la),
// use PGADDR(PDX(la), PTX(la), PGOFF(la)).

// page number field of address
#define PPN(la)   (((uintptr t) (la)) >> PTXSHIFT)
#define VPN(la)   PPN(la)  // used to index into vpt[]

// page directory index
#define PDX(la)   ((((uintptr t) (la)) >> PDXSHIFT) & 0x3FF)
#define VPD(la)   PDX(la)  // used to index into vpd[]

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

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

// construct linear address from indexes and offset
#define PGADDR(d, t, o) ((void*) ((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
```

几个缩写规范：

la：线性地址Linear Address
PDX: 页目录索引Page Directory Index
PTX: 页表索引Page Table Index

PPN：页标号Page Number？ ,由PDX和PTX共同组成,
VPN：虚拟页标号？=PPN, 这两个东西有什么用？

VPD：虚拟页目录？

PGOFF：页内偏移Page Offset
PGADDR：页地址（线性地址）

3. 关于页目录和页表的几个重要常量：

```
// Page directory and page table constants.
#define NPDENTRIES  1024  // page directory entries per page directory
#define NPTENTRIES  1024  // page table entries per page table

#define PGSIZE   4096  // bytes mapped by a page
#define PGSHIFT   12  // log2(PGSIZE)

#define PTSIZE   (PGSIZE*NPTENTRIES) // bytes mapped by a page directory entry
#define PTSHIFT   22  // log2(PTSIZE)

#define PTXSHIFT 12  // offset of PTX in a linear address
#define PDXSHIFT 22  // offset of PDX in a linear address
```

几个缩写规范：

NPDENTRIES: 页目录项数Page Directory Entires
NPTENTRIES: 页表项数Page Table Entries

PGSIZE: 物理页面大小Page Size = 4096, 十六进制表示为0x00001000

PTSIZE: 页目录大小Page Table Size（或者是页表）， 十六进制表示为0x00400000

inc/memlayout.h

1. Global Descriptor Numbers是什么？GD_KT等等，用在哪里了？

2. 关于经过转换后的线性地址空间的内容：

注意！！！：

一般情况下，对于每个用户进程，都应该有一个自己的页目录，将其保存在用户上下文中。当用户进程被切换到运行状态时，将其用户进程页目录地址写入CR3，重新启动页管理

但是JOS只使用一个页目录！！就是说整个系统的线性地址空间只有4GB，并且所有代码（操作系统，用户程序）都使用这个页目录，这样的话就需要对线性地址空间进行详细的规划。

下面这个地址规划和lab1中那个地址规划的关系是：

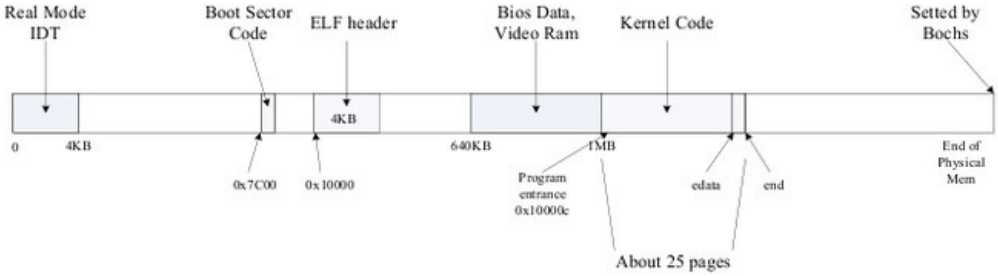

图 4-1. 调用 i386_init()函数以前内存的 layout

原来这个lab1的图是未启用页式管理之前的地址规划，所以只有段式转换：-KERNBASE， 启用页式转换以后，所有的内存访问（包括内核自己的），都要通过页式转换这个机制得到自己的使用地址，所以说，下面这个图是操作系统和用户程序在使用时线性地址所指向的区域，而上图是实际物理页面的布置。

下图的KERNBASE网上的区域对应的就是上图的0x00100000开始的区域，而且所有上面的已有区域都在下面的layout中有其对应（具体还没搞清楚哪个对哪个）

```
/*
 * Virtual memory map:                                Permissions
 *                                                    kernel/user
 *
 *    4 Gig --------->  +------------------------------+
 *                      |                              | RW/--
 *                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                      :              .               :
 *                      :              .               :
 *                      :              .               :
 *                      |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
 *                      |                              | RW/--
 *                      |   Remapped Physical Memory   | RW/--
 *                      |                              | RW/--
 *    KERNBASE ------>  +------------------------------+ 0xf0000000
 *                      |  Cur. Page Table (Kern. RW)  | RW/--  PTSIZE
 *    VPT,KSTACKTOP--> +------------------------------+ 0xefc00000      --+
 *                      |         Kernel Stack         | RW/--  KSTKSIZE   |
 *                      | - - - - - - - - - - - - - - -|                   PTSIZE
 *                      |      Invalid Memory (*)      | --/--             |
 *    ULIM     ------> +------------------------------+ 0xef800000      --+
 *                      |  Cur. Page Table (User R-)   | R-/R-  PTSIZE
 *    UVPT      ----->  +------------------------------+ 0xef400000
 *                      |          RO PAGES            | R-/R-  PTSIZE
 *    UPAGES    ----->  +------------------------------+ 0xef000000
 *                      |           RO ENVS            | R-/R-  PTSIZE
 * UTOP,UENVS ------>  +------------------------------+ 0xeec00000
 * UXSTACKTOP -/       |     User Exception Stack     | RW/RW  PGSIZE
 *                     +------------------------------+ 0xeebff000
 *                     |       Empty Memory (*)       | --/--  PGSIZE
 *    USTACKTOP  --->  +------------------------------+ 0xeebfe000
 *                     |      Normal User Stack       | RW/RW  PGSIZE
 *                     +------------------------------+ 0xeebfd000
 *                     |                              |
 *                     |                              |
 *                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                     .                              .
 *                     .                              .
 *                     .                              .
 *                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
 *                     |     Program Data & Heap      |
 *    UTEXT -------->  +------------------------------+ 0x00800000
 *    PFTEMP ------->  |       Empty Memory (*)       |        PTSIZE
 *                     |                              |
 *    UTEMP -------->  +------------------------------+ 0x00400000      --+
 *                     |       Empty Memory (*)       |                   |
 *                     | - - - - - - - - - - - - - - -|                   |
 *                     |  User STAB Data (optional)   |                 PTSIZE
 *    USTABDATA ---->  +------------------------------+ 0x00200000        |
 *                     |       Empty Memory (*)       |                   |
 *    0 ------------>  +------------------------------+                 --+
 *
 * (*) Note: The kernel ensures that "Invalid Memory" (ULIM) is *never*
 *     mapped.  "Empty Memory" is normally unmapped, but user programs may
 *     map pages there if desired.  JOS user programs map pages temporarily
 *     at UTEMP.
 */
```

从左边底下地址一直往上看：

第一部分

```
// Where user programs generally begin
#define UTEXT  (2*PTSIZE)
```

```
// Used for temporary page mappings.  Typed 'void*' for convenience
#define UTEMP   ((void*) PTSIZE)
// Used for temporary page mappings for the user page-fault handler
// (should not conflict with other temporary page mappings)
#define PFTEMP   (UTEMP + PTSIZE - PGSIZE)
// The location of the user-level STABS data structure
#define USTABDATA (PTSIZE / 2)
```

USTABDATA ：半个页目录大小，存放用户程序的STAB数据（调试信息？）
UTEMP ：一个页目录减去一个页（即1023个页），存放用户临时页映射？临时页映射是神码东西
PFTEMP ：一个页大小（4096），存放缺页中断处理的程序
UTEXT ：大小和用户栈共用一段内存，存放用户程序和数据

UTEMP江主席告诉我的解释是用户程序临时

第二部分

```
// Top of user-accessible VM
#define UTOP  UENVS
// Top of one-page user exception stack
#define UXSTACKTOP UTOP
// Next page left invalid to guard against exception stack overflow; then:
// Top of normal user stack
#define USTACKTOP (UTOP - 2*PGSIZE)
```

USTACKTOP ：大小和UTEXT共用（双向生长），存放用户栈，这个位置应该是栈底
UTOP ：用户有写权限的地址上限（从UTEXT开始的程序和数据区，然后是USTACKTOP的用户栈，直到UTOP，UTOP的位置是通过ULIM一直减去上面系统所需要占用的系统栈、用户页表等等得到的。最终用户能访问的地址空间大概是（0xEEC00000 - 0x00800000) = 0x0EE400000 = 3812MB = 3.8GB左右，不到4GB
UXSTACKTOP ：用户异常栈底User Exception Stack，这个栈的大小为一个物理页大小

第三部分

```
// Same as VPT but read-only for users
#define UVPT  (ULIM - PTSIZE)
// Read-only copies of the Page structures
#define UPAGES  (UVPT - PTSIZE)
// Read-only copies of the global env structures
#define UENVS  (UPAGES - PTSIZE)
```

UENVS ：只读环境结构，这个干吗的？在哪定义的
UPAGES ：只读页结构？干吗的？
UVPT ：只读页表？干吗？

VPT/UVPT
当前用户/内核进程的页表项映射的位置？
UVPT是为用户程序只读访问VPT而做的映射？

这个在后面就可以知道这两个东西指向的是同一片物理内存，只是程序在访问时需要的权限不同，UVPT规定了用户只能读不能写

第四部分

```
// Virtual page table.  Entry PDX[VPT] in the PD contains a pointer to
// the page directory itself, thereby turning the PD into a page table,
// which maps all the PTEs containing the page mappings for the entire
// virtual address space into that 4 Meg region starting at VPT.
#define VPT  (KERNBASE - PTSIZE)
#define KSTACKTOP VPT
#define KSTKSIZE (8*PGSIZE)     // size of a kernel stack
#define ULIM  (KSTACKTOP - PTSIZE)
```

VPT ：用户虚拟页表Virtual Page Table，大小为PTSIZE， 这个是内核使用的
KSTACKTOP ：内核栈
KSTKSIZE ：内核栈大小，8个页大小
ULIM ：用户态程序可以访问的地址上限,高于此地址的内存,用户不可读（UVPT到ULIM有用户页表，用户程序读页表干吗？）

内核栈是内核在运行时使用的栈？
UVPT倒ULIM的用户页表是为了满足有的底层用户程序为了读取页表给出的。（比如用户想知道自己给出的一个虚拟地址究竟对应到了哪个物理地址？在一些磁盘程序中）

kern/pmap.c

使用的变量

```
// These variables are set by i386_detect_memory()
static physaddr_t maxpa; // Maximum physical address
size_t npage;   // Amount of physical memory (in pages)
static size_t basemem;  // Amount of base memory (in bytes)
static size_t extmem;  // Amount of extended memory (in bytes)

// These variables are set in i386_vm_init()
```

```
pde_t* boot_pgdir;  // Virtual address of boot time page directory
physaddr_t boot_cr3;  // Physical address of boot time page directory
static char* boot_freemem; // Pointer to next byte of free mem

struct Page* pages;  // Virtual address of physical page array
static struct Page_list page_free_list; // Free list of physical pages
```

问题一：maxpa啥意思？

```
void
i386_detect_memory(void)
{
 // CMOS tells us how many kilobytes there are
 basemem = ROUNDDOWN(nvram_read(NVRAM_BASELO)*1024, PGSIZE);
 extmem = ROUNDDOWN(nvram_read(NVRAM_EXTLO)*1024, PGSIZE);

 // Calculate the maximum physical address based on whether
 // or not there is any extended memory.  See comment in <inc/mmu.h>.
 if (extmem)
  maxpa = EXTPHYSMEM + extmem;
 else
  maxpa = basemem;

 npage = maxpa / PGSIZE;

 cprintf("Physical memory: %dK available, ", (int)(maxpa/1024));
 cprintf("base = %dK, extended = %dK\n", (int)(basemem/1024), (int)(extmem/1024));
}
```

在上面这段代码中为什么会有基础内存和扩展内存的区别？

答：因为x86在实模式下和保护模式下的内存有差别，实模式下只有640KB的基本内存，保护模式下会有扩展内存，这个通过lab2初始启动看到的画面就可以知道
Physical memory: 66556K available, base = 640K, extended = 65532K，这个和我们lab1里作的时候那些地址的分配是一致的，比如实模式下0x7c00为boot loader加载地址，切换到保护模式以后就可以把内核载入到0x00100000的位置。。。

这里的basemen和extmem就是QEMU打印出的值，maxpa是物理地址中的最高地址，在后面用于计算总共的物理页面数npages

问题二：
boot_pgdir和boot_freemem干嘛的？

答：首先看`boot_freemem`的初始化过程：

```
static void*
boot_alloc(uint32_t n, uint32_t align)
{
 extern char end[];
 void *v;

 // Initialize boot_freemem if this is the first time.
 // 'end' is a magic symbol automatically generated by the linker,
 // which points to the end of the kernel's bss segment -
 // i.e., the first virtual address that the linker
 // did _not_ assign to any kernel code or global variables.
 if (boot_freemem == 0)
  boot_freemem = end;

 // LAB 2: Your code here:
 // Step 1: round boot_freemem up to be aligned properly
 //   (hint: look in types.h for some handy macros)
 // Step 2: save current value of boot_freemem as allocated chunk
 // Step 3: increase boot_freemem to record allocation
 // Step 4: return allocated chunk

 return NULL;
}
```

一开始的时候`boot_freemen = end`，这个是内核放进内存后的结束地址，也就是说，在内核之后的所有空间都算作空闲内存了，地址从这里开始分配。`boot_alloc`的作用，就是每次申请一块`n`字节的空间，那么就从`boot_freemen`开始的空间往后取`n`个字节分配出去，然后`boot_freemem`更新为这个尾地址。

然后再看`boot_pgdir`，这个是`boot_pgdir`指向的是 系统页目录 所在的首地址

```
 ////////////////////////////////////////////////////////////////////
 // create initial page directory.
 pgdir = boot_alloc(PGSIZE, PGSIZE);
 memset(pgdir, 0, PGSIZE);
 boot_pgdir = pgdir;
 boot_cr3 = PADDR(pgdir);
```

这段代码显示， 系统页目录存放在`kernel`后面的第一块`PGSIZE`大小的内存区域

问题三：`pages`干吗的？

这个就是lab2里除了boot_alloc之外第二个需要我们填写的代码，pages的信息请详细阅读huazhong的第四章第7页，这里提到pages可以用数组下标访问，和任意一个实际物理页面一一对应。这个在lab2里暂时还没有用到的地方，pages的意义就是为了开出npage个struct Page大小的空间，让空闲内存分配链表page_free_list有内存空间分配给它

法大赛
fdsa

关于宏，一共有**4**套宏

1. inc/mmu.h中关于线性地址取出其地址各部分的宏：

```
// A linear address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
//  \---------- PPN(la) -----------/
//
// The PDX, PTX, PGOFF, and PPN macros decompose linear addresses as shown.
// To construct a linear address la from PDX(la), PTX(la), and PGOFF(la),
// use PGADDR(PDX(la), PTX(la), PGOFF(la)).

// page number field of address
#define PPN(la)   (((uintptr_t) (la)) >> PTXSHIFT)
#define VPN(la)   PPN(la)   // used to index into vpt[]

// page directory index
#define PDX(la)   ((((uintptr_t) (la)) >> PDXSHIFT) & 0x3FF)
#define VPD(la)   PDX(la)   // used to index into vpd[]

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

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

// construct linear address from indexes and offset
#define PGADDR(d, t, o) ((void*) ((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
```

它的组织方式和我的思维很不一样，因此我把它们整理成：
对于线性地址，三部分：PDX(la), PTX(la)和PGOFF(la)， 前两个构成PPN(la)页号， 页号可以用来顺序索引**pages**
还有一个反操作重组的宏PGADDR(PDX(la), PTX(la), PGOFF(la)] = la

至于VPN和VPD，这个暂时可以先不用管，以后会遇到

2. inc/mmu.h中关于页表项的组成部分

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

这个宏是取出Page Table或者Page Directory的前20位的，因为只有前20位代表地址，而后12位则是各种flag，我们计算的时候用不到

3. kern/pmap.h中关于内核虚拟地址和物理地址的转换

```
#define PADDR(kva)
#define KADDR(pa)
```

4. kern/pmap.h中关于Page和物理地址的转换

```
static inline ppn_t
page2ppn(struct Page *pp)

static inline physaddr_t
page2pa(struct Page *pp)

static inline struct Page*
pa2page(physaddr_t pa)

static inline void*
page2kva(struct Page *pp)
```

1. 有了PPN，则通过pages[PPN]就可以找到对应Page*，如果有Page*，则需要page2ppn (Page*)找到ppn
2. 物理地址地址和PPN一一对应，所以page2pa和pa2page可以通过PPN实现pa和Page*的一一对应

重要收获！！！小白师兄牛逼阿。。

在inc/queue.h定义的链表模板里，为什么要这样写这个宏？

```
#define LIST_INIT(head) do {       \
 LIST_FIRST((head)) = NULL;      \
} while (0)
```

这个宏在外面包了一层do while 的意义在哪？直接加大括号不行么？

小白师兄的解释是，如果你写了这么个语句：

```
            if (some bool) LIST_INIT (some head); else XXXX
```

注意LIST_INIT后的分号是必须的，因为是作为一个函数调用来写的，那么GCC展开代码会变成：(使用gcc -E xx.c显示预处理以后的结果)

```
            if (some bool)
                 do {
                      LIST_FIRST (some head) = NULL;
                 } while (0);
           else
                XXXX
```

但是，如果你把while写成只剩大括号的话！！那么就变成了

```
            if (some bool)
                 {
                      LIST_FIRST (some head) = NULL;
                 } ;
           else
                XXXX
```

注意那个分号，这个显然是语法错误的。。。 所以do while的作用是把里面的代码块包装成一个语句的调用使得后面接分号时在任何情况下都符合语法

问题：

1. page_init 中，保留的已用页面的page[i]到底要不要用page_initpp ？
2. page_alloc中，它明明说不要清空物理内存中的值，那为什么又提示我们用page_initpp?

3. pgdir_walk中，在新建页表的时候，页目录相应的entry的权限位怎么设置？

答：通过翻看X86编程手册关于COMBINING PAGE AND SEGMENT PROTECTION (在Reference里的 Volume 3A: System Programming Guide, Part 1)，可以知道x86 mmu处理地址访问的时候，先查看segment的权限位，如果满足，则查看相应的页目录权限位，还满足，则查看对应的页表权限位，都满足以后就可以访问了。我们在创建新页表的时候，不能知道这个页表里所有的1024个页哪些可读可写，所以这个页表对应的在页目录里那项的权限位最好设置成用户态可读可写，即PTE_U | PTE_W | PTE_P

4. 在i386_vm_init ()里

```
 //////////////////////////////////////////////////////////////////////
 // Map 'pages' read-only by the user at linear address UPAGES
 // Permissions:
 //    - the new image at UPAGES -- kernel R, user R
 //      (ie. perm = PTE_U | PTE_P)
 //    - pages itself -- kernel RW, user NONE
 // Your code goes here:
```

在用户地址空间里从UPAGES开始分配了PTSIZE的空间出来，用来存在pages结构？？？pages结构给用户态程序有什么用？？？另外PTSIZE的空间够么？因为pages一共有npage * sizeof (struct Page)个空间

答：JOS的设计就是可以让用户可以看到这个数组，比如说用户想知道一个物理地址所在的页面被引用了多少次，就可以用pages[pa] -> pp_ref来得到了。至于够不够，PTSIZE = 1024 * PGSIZE = 1024 * 4KB, 一个(struct Page)只有12B，所以可以放下4096KB / 12B 个(struct Page) = 1024KB / 3B > 1024KB / 4B = 256K，就是说可以引用256K个物理页面，对应到真实内存就是 256K * 4KB = 1024k * kb = 1G, 足够了, 机器没那么多物理内存