lab2 物理内存管理

一.实验目的

- 理解基于段页式内存地址的转换机制
- 理解页表的建立和使用方法
- 理解物理内存的管理方法

二.Teasing Time

- 本次实验中涉及许多结构体相关内容，C++结构体掌握不太扎实，许多地方读起来有点费劲，边读边查。
- 自映射机制一开始一直被绕进去了

三.实验过程

通过meld工具比较lab1与lab2的代码，将lab1中修改过的部分合并到lab2代码中，

物理内存管理器初始化：

```
static void
default_init(void) {
    list_init(&free_list);//list_init，在lisi.h中，将前后向指针全部指向自己
    nr_free = 0;//空白页数量置0
}
```

操作系统启动后，bootloader先去执行kern_entry函数（lab2/kern/init/entry.S），为kern_init建立对应的堆栈结构和段映射关系，之后调用kerninit函数。在kern_init中调用了pmm_init函数，进行物理内存设置。pmm_init函数位于（lab2/kern/mm/pmm.c)中，其中包含一个pmm_manager结构体常量，初始未分配对象。在执行pmm_init的过程中，首先init_pmm_manager,然后page_init

物理内存的探测

物理内存的探测通过int 0x15中断调用实现，通过e820获取内存信息。e820 map位于（lab2/kern/mm/memlayout.h）

```
struct e820map {
    int nr_map;
    struct {
        uint64_t addr; //内存块基地址，8字节
        uint64_t size; //内存块大小，8字节
        uint32_t type; //内存类型，4字节
    } __attribute__((packed)) map[E820MAX];
    //__attribute__((packed))是指对齐方式，map[E820MAX]大概是指表项的数量上线是20项？（E820MAX=20）
};
/*type的类型：
01h    memory, available to OS
02h    reserved, not available (e.g. system ROM, memory-mapped device)
03h    ACPI Reclaim Memory (usable by OS after reading ACPI tables)
04h    ACPI NVS Memory (OS is required to save this memory between NVS sessions)
```

```
other  not defined yet -- treat as Reserved
*/
```

int 0x15中断调用

eax	e820h,中断调用参数
edx	534D4150h (即4个ASCII字符“SMAP”)
ebx	初始给成0，每次扫描结束时为上次调用后的计数值，当其为0时表明扫描结束
ecx	保存地址范围描述符的内存大小,应该大于等于20字节；
di	指向保存地址范围描述符结构的缓冲区，BIOS把信息写入这个结构的起始地址。

```
probe_memory:
    movl $0, 0x8000                                 #8000是e820的映射结构即
e820map，在此处将该地址清零。
    xorl %ebx, %ebx                                 #通过异或指令将ebx置零，是第一次调
用时提供的参数
    movw $0x8004, %di                               #给di赋成了8004
start_probe:
    movl $0xE820, %eax                              #e820的调用参数
    movl $20, %ecx                                  #保存地址范围描述符的内存大小,应该
大于等于20字节；
    movl $SMAP, %edx                                #534D4150h (即4个ASCII字
符“SMAP”) ，这只是一个签名而已；
    int $0x15                                       #触发中断
    jnc cont
    movw $12345, 0x8000
    jmp finish_probe
cont:
    addw $20, %di                                   #每次递增20字节
    incl 0x8000                                     #nr_map递增
    cmpl $0, %ebx                                   #比较搜索是否结束
    jnz start_probe                                 #继续搜索
finish_probe:
```

关于SMAP的签名：SMAP是管理模式访问保护，禁止内核CPU访问用户空间的代码的意思。

物理页的管理

物理页的数据结构体Page：

```
struct Page {
    int ref;           //被引用的次数，有多少虚拟页映射到该物理页上，
    uint32_t flags;    // 描述物理页帧的状态位
    unsigned int property;    // 在空闲块的第一个页内，记录空闲块中的页数
    list_entry_t page_link;  // 空闲物理内存块双向链表
};
//同一个块中会有许多页，在这个块的第一个页上启用property，用以记录该块中的页的数量。
```

双向链表结构：

```
struct list_entry {
    //前向与后向的指针。
    struct list_entry *prev, *next;
};
//重命名
typedef struct list_entry list_entry_t;
```

管理所有连续的空闲内存空间块的数据结构free_area_t:

```
typedef struct {
    list_entry_t free_list;         // 链表头
    unsigned int nr_free;           // 链表中空页的个数
} free_area_t;
```

page_init函数

```
/* pmm_init - initialize the physical memory management */
static void
page_init(void) {
    /*
    KERNBASE是0xC0000000
    3GB起始地址，0-3GB用于用户，3-4GB用于内核。
    */
    struct e820map *memmap = (struct e820map *)(0x8000 + KERNBASE);
    uint64_t maxpa = 0;
    //获取到了通过内存探测过程得到的物理内存结构，memmap
    cprintf("e820map:\n");
    int i;
    //memmap->nr_map即块的数量
    for (i = 0; i < memmap->nr_map; i ++) {
        //每一个块的起始地址与结束地址
        uint64_t begin = memmap->map[i].addr, end = begin + memmap->map[i].size;
        cprintf("  memory: %08llx, [%08llx, %08llx], type = %d.\n",
                memmap->map[i].size, begin, end - 1, memmap->map[i].type);
        //E820_ARM宏定义位于lab2/kern/mm/memlayout.h中，其值为1，对应type类型即OS可用内
存。
        if (memmap->map[i].type == E820_ARM) {
            //KMEMSIZE，物理内存量的最大值，具体含义.....?
            //不断增长maxpa
            if (maxpa < end && begin < KMEMSIZE) {
                maxpa = end;
            }
        }
    }
    if (maxpa > KMEMSIZE) {
        maxpa = KMEMSIZE;
    }
    //maxpa是探测到的空间的最高地址
    extern char end[];
    //PGSIZE=4096，按4KB分页，得到总的页数
    npage = maxpa / PGSIZE;
    //end是加载ucore结束后的地址，按页大小向上取整。pages即可以存储PAGE结构的起始地址，一个
page*的指针，
    pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);
```

```
    for (i = 0; i < npage; i ++) {
        SetPageReserved(pages + i);
    }
    //这里是不是把内存中所有的页都保留了？？？
    uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) * npage);
    //PADDR是减去了基址
    for (i = 0; i < memmap->nr_map; i ++) {
        uint64_t begin = memmap->map[i].addr, end = begin + memmap->map[i].size;
        if (memmap->map[i].type == E820_ARM) {
            //应该是探测到的空闲块已经有一部分被用于存储page数组结构了，因此目前还处在空闲阶段的块从freemem开始。
            if (begin < freemem) {
                begin = freemem;
            }
            if (end > KMEMSIZE) {
                end = KMEMSIZE;
            }
            if (begin < end) {
                begin = ROUNDUP(begin, PGSIZE);
                end = ROUNDDOWN(end, PGSIZE);
                if (begin < end) {
                    //begin是这个空闲块的起始地址，(end - begin) / PGSIZE是这个块中所包含的页的数量。
                    init_memmap(pa2page(begin), (end - begin) / PGSIZE);
                    //pa2page定义位于lab2/kern/mm/pmm.h中
                    /*
                        static inline struct Page *pa2page(uintptr_t pa) {
                        if (PPN(pa) >= npage)
                        {
                            panic("pa2page called with invalid pa");
                        }
                        return &pages[PPN(pa)];
                        }
                      涉及到了另一个函数，PPN，PPN宏定义位于lab2/kern/mm/mmu.h中，
                     page number field of address
                    #define PPN(la) (((uintptr_t)(la)) >> PTXSHIFT)
                    其中PTXSHIFT=12
                    pa就是这一块空间的起始地址
                    应该是地址的高位部分代表了页的编号，将其右移了12位获取到了这个页编号。因为按4kB对齐，12位
                    pa2page(begin)最终拿到的就是对应这个物理页的page结构的地址。
                    */
                }
            }
        }
    }
}
```

init_memmap最终拿到的是一个page结构（是一个空闲块所对应的第一个页）和空闲块中页的数量。再往后就涉及到pmm_manager了。

物理内存页管理器架构

pmm_manager(lab2/kern/mm/pmm.h)，就是一个函数指针列表

```
// pmm_manager is a physical memory management class. A special pmm manager - XXX_pmm_manager
// only needs to implement the methods in pmm_manager class, then XXX_pmm_manager can be used
// by ucore to manage the total physical memory space.
struct pmm_manager {
    const char *name;                                 // 物理内存页管理器的名字
    void (*init)(void);                               // 初始化内存页管理器
                                                      // (空闲块列表，空闲块数量)
    void (*init_memmap)(struct Page *base, size_t n); // 初始化管理空闲内存页的数据结构
                                                      // 根据空闲物理空间设置数据结构
    struct Page *(*alloc_pages)(size_t n);            // 依靠分配算法分配n个物理内存页
    void (*free_pages)(struct Page *base, size_t n);  // 释放n个物理内存页
    size_t (*nr_free_pages)(void);                    // 剩余空闲页数
    void (*check)(void);                              // 用于检测分配/释放实现是否正确的辅助函数
};
```

承接page_init的过程，需要补充函数default_init_memmap

```
static void
default_init_memmap(struct Page *base, size_t n) {
    //空闲页数大于0
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    list_add_before(&free_list, &(base->page_link));//首页的指针集合插入空闲页链表
}
```

分配一个页：

```
static struct Page *
default_alloc_pages(size_t n) {
     //分配n个页
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    //确保n的范围，要分配的页数n大于0且在当前总的空闲页范围内。
    //空页
    struct Page *page = NULL;
```

```
    //链表入口
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        //不断遍历空闲块，双向俩表，这里应该是环形
        //由le2page宏将链表元素转换为page指针。
        struct Page *p = le2page(le, page_link);
        //检查当前块的空闲页数量是否满足要求
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        for (struct Page *p = page; p != (page + n); ++p)
            ClearPageProperty(p); //将分配出去的内存页标记为非空闲
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            list_add(&free_list, &(p->page_link));
    }
        list_del(&(page->page_link));
        nr_free -= n;
    }
    return page;
}
```

释放n个页

```
/**
 * 释放掉自base起始的连续n个物理页,n必须为正整数
 * */
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;

    // 遍历这N个连续的Page页，将其相关属性设置为空闲
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }

    // 由于被释放了N个空闲物理页，base头Page的property设置为n
    base->property = n;
    SetPageProperty(base);

    // 下面进行空闲链表相关操作
    list_entry_t *le = list_next(&free_list);
    // 迭代空闲链表中的每一个节点
    while (le != &free_list) {
        // 获得节点对应的Page结构
        p = le2page(le, page_link);
        le = list_next(le);
        // TODO: optimize
```

```
        if (base + base->property == p) {
            // 如果当前base释放了N个物理页后，尾部正好能和Page p连上，则进行两个空闲块的合并
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            // 如果当前Page p能和base头连上，则进行两个空闲块的合并
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
    // 空闲链表整体空闲页数量自增n
    nr_free += n;
    le = list_next(&free_list);

    // 迭代空闲链表中的每一个节点
    while (le != &free_list) {
        // 转为Page结构
        p = le2page(le, page_link);
        if (base + base->property <= p) {
            // 进行空闲链表结构的校验，不能存在交叉覆盖的地方，能合并的必须合并。
            assert(base + base->property != p);
            break;
        }
        le = list_next(le);
    }
    // 将base加入到空闲链表之中
    list_add_before(le, &(base->page_link));
}
```

段页式映射启动后，二级页表的建立

指向页目录表的指针存储在boot_pgdir中，首个页表映射0-4M空间，该页表占一页。共管理1024个页。

通过boot_map_segment函数进一步完善映射关系。

```
//boot_map_segment - setup&enable the paging mechanism
// parameters
//  la:   linear address of this memory need to map (after x86 segment map)
//  size: memory size
//  pa:   physical address of this memory
//  perm: permission of this memory
static void
boot_map_segment(pde_t *pgdir, uintptr_t la, size_t size, uintptr_t pa, uint32_t
perm) {
    //PGOFF宏定义位于mmu.h中，取得是地址得低12位。这里是指偏移量相同，以页为单位？
    assert(PGOFF(la) == PGOFF(pa));
    //n是要映射的页的数量？
    size_t n = ROUNDUP(size + PGOFF(la), PGSIZE) / PGSIZE;
    la = ROUNDDOWN(la, PGSIZE);
    pa = ROUNDDOWN(pa, PGSIZE);
```

```
    for (; n > 0; n --, la += PGSIZE, pa += PGSIZE) {
        pte_t *ptep = get_pte(pgdir, la, 1);
        assert(ptep != NULL);
        *ptep = pa | PTE_P | perm;
    }
}
```

涉及到了get_pte函数。补充get_pte函数

```
//get_pte - get pte and return the kernel virtual address of this pte for la
//        - if the PT contians this pte didn't exist, alloc a page for PT
// parameter:
//  pgdir:  the kernel virtual base address of PDT
//  la:     the linear address need to map
//  create: a logical value to decide if alloc a page for PT
// return vaule: the kernel virtual address of this pte
pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    /* LAB2 EXERCISE 2: YOUR CODE
     *
     * If you need to visit a physical address, please use KADDR()
     * please read pmm.h for useful macros
     *
     * Maybe you want help comment, BELOW comments can help you finish the code
     *
     * Some Useful MACROs and DEFINEs, you can use them in below implementation.
     * MACROs or Functions:
     *   PDX(la) = the index of page directory entry of VIRTUAL ADDRESS la.
     *   KADDR(pa) : takes a physical address and returns the corresponding
kernel virtual address.
     *   set_page_ref(page,1) : means the page be referenced by one time
     *   page2pa(page): get the physical address of memory which this (struct
Page *) page  manages
     *   struct Page * alloc_page() : allocation a page
     *   memset(void *s, char c, size_t n) : sets the first n bytes of the memory
area pointed by s
     *                                       to the specified value c.
     * DEFINEs:
     *   PTE_P           0x001                   // page table/directory entry
flags bit : Present
     *   PTE_W           0x002                   // page table/directory entry
flags bit : Writeable
     *   PTE_U           0x004                   // page table/directory entry
flags bit : User can access
     */
#if 0
    pde_t *pdep = NULL;   // (1) find page directory entry
    if (0) {              // (2) check if entry is not present
                          // (3) check if creating is needed, then alloc page for
page table
                          // CAUTION: this page is used for page table, not for
common data page
                          // (4) set page reference
        uintptr_t pa = 0; // (5) get linear address of page
                          // (6) clear page content using memset
```

```
                        // (7) set page directory entry's permission
    }
    return NULL;          // (8) return page table entry
#endif
    /*
    关于PDX，定义位于kern/mm/mmu.h中，其中PDXSHIFT为22，即取了线性地址的高10位作为PDT的索引，获取到一个PDE
    #define PDX(la) ((((uintptr_t)(la)) >> PDXSHIFT) & 0x3FF)
    */
    pde_t *pdep = &pgdir[PDX(la)];
    if (!(*pdep & PTE_P)) {
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL) {
            return NULL;
        }
        /*此处涉及到两个函数，page2pa和KADDR
        page2pa与前面的一个pa2page对应
        pa2page是根据空间地址找到管理它的page结构
        page2pa是根据page结构找到对应空间的起始地址。
        二者联系在于空间地址的高20位就是page数组的索引位

        KADDR是在地址的基础上加了基址0xC0000000
        */
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);
        memset(KADDR(pa), 0, PGSIZE);
        //给pde设置对应的映射，地址，用户级，可写，存在
        *pdep = pa | PTE_U | PTE_W | PTE_P;
    }
    //PDE_ADDR是取高20位的PageTable的地址(低12位为0，省去了左移一步)，KADDR是加上基址，PTX是取线性地址的中间十位，最终得到一个PTE
    return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];
}
```

释放某一个页

page_remove_pte函数

```
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    /* LAB2 EXERCISE 3: YOUR CODE
     *
     * Please check if ptep is valid, and tlb must be manually updated if mapping is updated
     *
     * Maybe you want help comment, BELOW comments can help you finish the code
     *
     * Some Useful MACROs and DEFINEs, you can use them in below implementation.
     * MACROs or Functions:
     *   struct Page *page pte2page(*ptep): get the according page from the value of a ptep
     *   free_page : free a page
     *   page_ref_dec(page) : decrease page->ref. NOTICE: ff page->ref == 0 , then this page should be free.
```

```
     *   tlb_invalidate(pde_t *pgdir, uintptr_t la) : Invalidate a TLB entry, but
only if the page tables being
     *                        edited are the ones currently in use by the
processor.
     * DEFINEs:
     *   PTE_P           0x001                   // page table/directory entry
flags bit : Present
     */
#if 0
    if (0) {                      //(1) check if this page table entry is
present
        struct Page *page = NULL; //(2) find corresponding page to pte
                                  //(3) decrease page reference
                                  //(4) and free this page when page reference
reachs 0
                                  //(5) clear second page table entry
                                  //(6) flush tlb
    }
#endif
    /*
    给定的参数为页目录表项，线性地址和页表项
    */
    if (*ptep & PTE_P) {
        //检查PTE_P位，是否存在
        struct Page *page = pte2page(*ptep);
        //pte2page是根据页表项的高20位获得页的地址，在pa2page获得page结构
        if (page_ref_dec(page) == 0) {
            /*
            page_ref_dec(struct Page *page) {
                page->ref -= 1;
                return page->ref;
            }
            */
            //没有页在引用它，释放
            free_page(page);
        }
        //页表项清空
        *ptep = 0;
        //TLB清空，当修改的页表目前正在被进程使用时，使之无效。
        tlb_invalidate(pgdir, la);
        /*
        tlb_invalidate(pde_t *pgdir, uintptr_t la) {
            if (rcr3() == PADDR(pgdir)) {
                invlpg((void *)la);
            }
        }
        */
    }
}
```

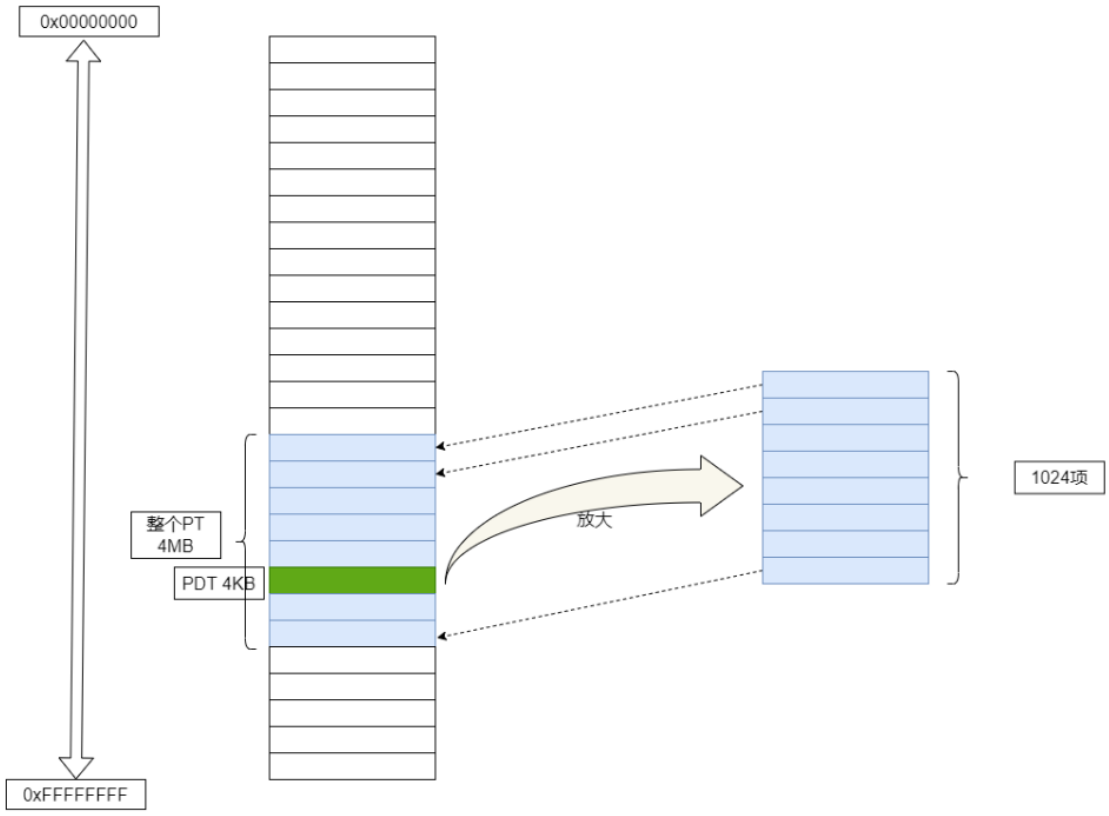
buddy system

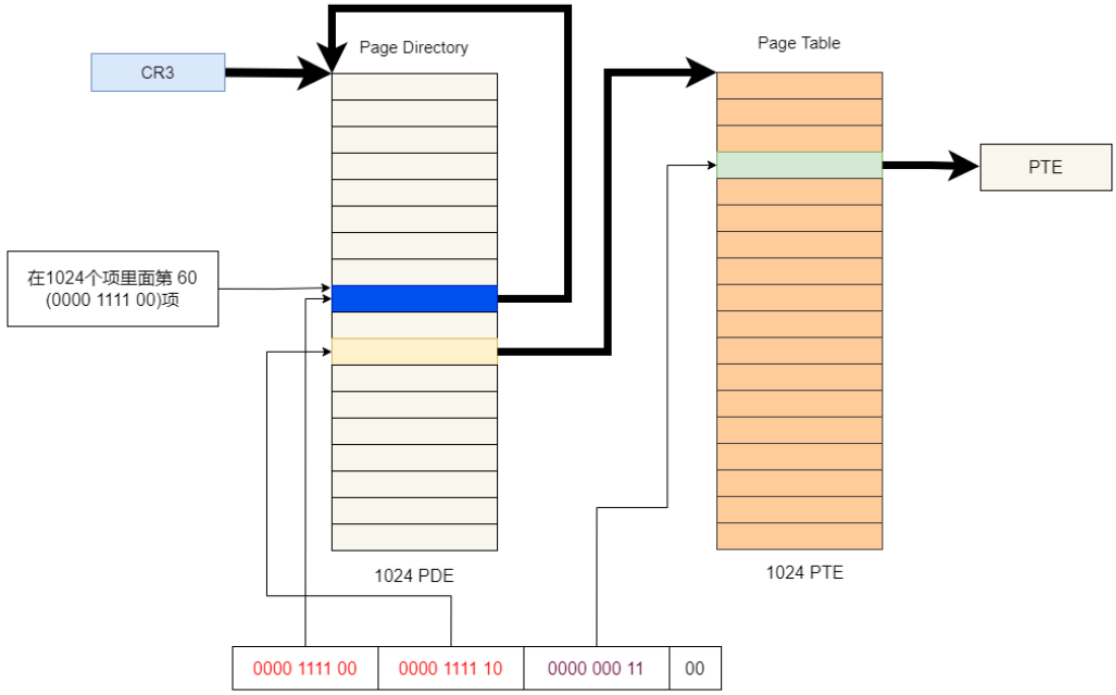
伙伴分配即将内存按2的幂进行划分，

相关宏定义

```
/*索引index指在节点树中由上至下的索引序号，从0开始*/
//left_leaf，寻找一个节点的左子节点
#define LEFT_LEAF(index) ((index) * 2 + 1)
#define RIGHT_LEAF(index) ((index) * 2 + 2)
#define PARENT(index) ( ((index) + 1) / 2 - 1)
```

自映射




CR3
Page Directory
Page Table
PTE
在1024个项里面第 60
(0000 1111 00)项
1024 PDE
1024 PTE
0000 1111 00
0000 1111 10
0000 000 11
00