uCore实验二报告

黄郭斌 计科1502班 201507010206

一. 内容

本次实验包含三个部分。首先了解如何发现系统中的物理内存；然后了解如何建立对物理内存的初步管理，即了解连续物理内存管理；最后了解页表相关的操作，即如何建立页表来实现虚拟内存到物理内存之间的映射，对段页式内存管理机制有一个比较全面的了解。本实验里面实现的内存管理还是非常基本的，并没有涉及到对实际机器的优化，比如针对 cache 的优化等。如果大家有余力，尝试完成扩展练习。

二. 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

三. 实验分析及结果

1. 练习一

首先查看实验中是用什么数据结构来保存物理页的信息; 可以看kern/mm/memlayout.h这个文件:

```
/* free_area_t - maintains a doubly linked list to record free
(unused) pages */
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // # of free pages in this
free list
} free_area_t;
```

结构体Page保存了物理页的信息,该结构四个成员变量意义如下:

1. ref表示这样页被页表的引用记数，应该就是映射此物理页的虚拟页个数。一旦某页表中有一个页表项设置了虚拟页到这个Page管理的物理页的映射关系，就会把Page的ref加一。反之，若是解除，那就减一。
   2.flags表示此物理页的状态标记，有两个标志位，第一个表示是否被保留，如果被保留了则设为1（比如内核代码占用的空间）。第二个表示此页是否是free的。如果设置为1，表示这页是free的，可以被分配；如果设置为0，表示这页已经被分配出去了，不能被再二次分配。
   3.property用来记录某连续内存空闲块的大小，这里需要注意的是用到此成员变量的这个Page一定是连续内存块的开始地址（第一页的地址）。
   4.page_link是便于把多个连续内存空闲块链接在一起的双向链表指针，连续内存空闲块利用这个页的成员变量page_link来链接比它地址小和大的其他连续内存空闲块。

下面这个结构是一个双向链表，负责管理所有的连续内存空闲块，便于分配和释放。

```
/* free_area_t - maintains a doubly linked list to record free
(unused) pages */
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // # of free pages in this
free list
} free_area_t;
```

可以看到这个结构体中包含着一个 list_entry_t 结构的双向链表指针用来保存链表头和记录当前空闲页的个数的无符号整型变量 nr_free。

- **具体的算法思想:**

First-fit 内存分配算法是物理内存页管理器顺着双向链表进行搜索空闲内存区域，直到找到一个足够大的空闲区域，这是一种速度很快的算法，因为它尽可能少地搜索链表。如果空闲区域的大小和申请分配的大小正好一样，则把这个空闲区域分配出去，成功返回;否则将该空闲区分为两部分，一部分区域与申请分配的大小相等，把它分配出去，剩下的一部分区域形成新的空闲区。其释放内存的设计思路很简单，只需把这块区域重新放回双向链表中即可。

- **具体实现:**

1. default_init_memmap，这个函数是用来初始化空闲页链表的,初始化每一个空闲页，然后计算空闲页的总数。 具体代码及注释如下:

```
68 static void
69 default_init_memmap(struct Page *base, size_t n) {
70     assert(n > 0);
71     struct Page *p = base;
72     for (; p != base + n; p ++) {
73         //检查此页是否是保留页
74         assert(PageReserved(p));
75         //设置标志位
76         p->flags = 0;
77         SetPageProperty(p);
78         p->property = 0;
79         //清零此页的引用计数
80         set_page_ref(p, 0);
81         //使用头插法将空闲页插入到链表
82         list_add_before(&free_list, &(p->page_link));
83     }
84     //说明连续有n个空闲块，计算空闲页总数
85     nr_free += n;
86     //连续内存空闲页大小为n
87     base->property = n;
88 }
```

注意:使用头插法的原因是地址从低地址往高地址增长;p->flags是将PG_reserved置0,代表不被内核占用或者不能使用。

2.default_alloc_pages(),此函数是用于为进程分配空闲页。 其分配的步骤如下：

> ①寻找足够大的空闲块 ，如果找到了，重新设置标志位; ②从空闲链表中删除此页 ; ③判断空闲块大小是否合适 ，如果不合适，分割页块 ，如果合适则不进行操作 ; ④ 计算剩余空闲页个数 ; ⑤ 返回分配的页块地址 。

具体代码及注释如下:

```
90 static struct Page *
91 default_alloc_pages(size_t n) {
92     assert(n > 0);
93     if (n > nr_free) {//如果所有的空闲页加起来的大小都不够，直接返回NULL
94         return NULL;
95     }
96     list_entry_t *le, *len;
97     le = &free_list;    //从空闲块链表的头指针开始
98
99     while((le=list_next(le)) != &free_list) { //依次往下寻找直到回到头
   指针处，即已经遍历一次
100       struct Page *p = le2page(le, page_link);//将地址转换成页的结构
101       //由于是first-fit，所以找到第一个大于等于n的块选中即可
102       if(p->property >= n){
103         int i;
104         for(i=0;i<n;i++){//把选中的空闲块链表的每一个页结构初始化
105           len = list_next(le);
106           //让pp指向分配的那一页
107           struct Page *pp = le2page(le, page_link);
108           //设置标志位
109           SetPageReserved(pp);
110           ClearPageProperty(pp);
111           list_del(le);//从空闲页链表中删除这个双向链表指针
112           le = len;
113         }
114         if(p->property>n){
115           //分割的页需要重新设置空闲大小
116           (le2page(le,page_link))->property = p->property - n;
117         }
118         ClearPageProperty(p);
119         SetPageReserved(p);
120         //空闲页大小-n
121         nr_free -= n;
122         return p;
123       }
124     }
125     return NULL;//没有大于等于n的连续空闲页块，返回空
126 }
```

3.default_free_pages(),这个函数的作用是释放已经使用完的页，把他们合并到free_list中。 具体步骤如下：

> ①在free_list中查找合适的位置以供插入; ②改变被释放页的标志位，以及头部的计数器；③尝试在free_list中向高地址或低地址合并。

具体代码及注释如下:

```
128 static void
129 default_free_pages(struct Page *base, size_t n) {
130     assert(n > 0);
131     assert(PageReserved(base));
132 
133     list_entry_t *le = &free_list;
134     struct Page * p;
135     //查找该插入的位置le
136     while((le=list_next(le)) != &free_list) {
137       p = le2page(le, page_link);
138       if(p>base){
139         break;
140       }
141     }
142     //向le之前插入n个空闲页，并设置标志位
143     for(p=base;p<base+n;p++){
144       list_add_before(le, &(p->page_link));
145     }
146     base->flags = 0;
147     set_page_ref(base, 0);
148     ClearPageProperty(base);
149     SetPageProperty(base);
150     //将页块信息记录在头部
151     base->property = n;
152     //判断是否需要合并
153     //向高地址合并
154     p = le2page(le,page_link) ;
155     if( base+n == p ){
156       base->property += p->property;
157       p->property = 0;
158     }
159     //向低地址合并
160     le = list_prev(&(base->page_link));
161     p = le2page(le, page_link);
162     //若低地址已分配则不需要合并
163     if(le!=&free_list && p==base-1){
164       while(le!=&free_list){
165         if(p->property){
166           p->property += base->property;
167           base->property = 0;
168           break;
169         }
170         le = list_prev(le);
171         p = le2page(le,page_link);
172       }
173     }
174     //释放已使用完的页后更新空闲页大小
175     nr_free += n;
176     return ;
177 }
178 
```

2. 练习二

- **设计思路**

1. pde_t全称为 page directory entry，也就是一级页表的表项（注意：pgdir实际不是表项，而是一级页表本身。实际上应该新定义一个类型pgd_t来表示一级页表本身）。pte_t全 称为 page table entry，表示二级页表的表项。uintptrt表示为线性地址，由于段式管理只做直接映射，所以它也是逻辑地址。
2. 有可能根本就没有对应的二级页表的情况，所以二级页表不必要一开始就分配，而是等到需要的时候再添加对应的二级页表。如果在查找二级页表项时，发现对应的二级页表不存在，则需要根据create参数的值来处理是否创建新的二级页表。如果create参数为0，则get_pte返回NULL；如果create参数不为0，则get_pte需要申请一个新的物理页（通过alloc_page来实现，可在mm/pmm.h中找到它的定义），

再在一级页表中添加页目录项指向表示二级页表的新物理页。注意，新申请的页必须全部设定为零，因为这个页所代表的虚拟地址都没有被映射。

3. 当建立从一级页表到二级页表的映射时，需要注意设置控制位。这里应该设置同时设置 上PTE_U、PTE_W和PTE_P（定义可在mm/mmu.h）。如果原来就有二级页表，或者新建立了页表，则只需返回对应项的地址即可。其中PTE_P 0x001 表示物理内存页存在; PTE_W 0x002 表示物理内存页内容可写; PTE_U 0x004 表示可以读取对应地址的物理内存页内容。

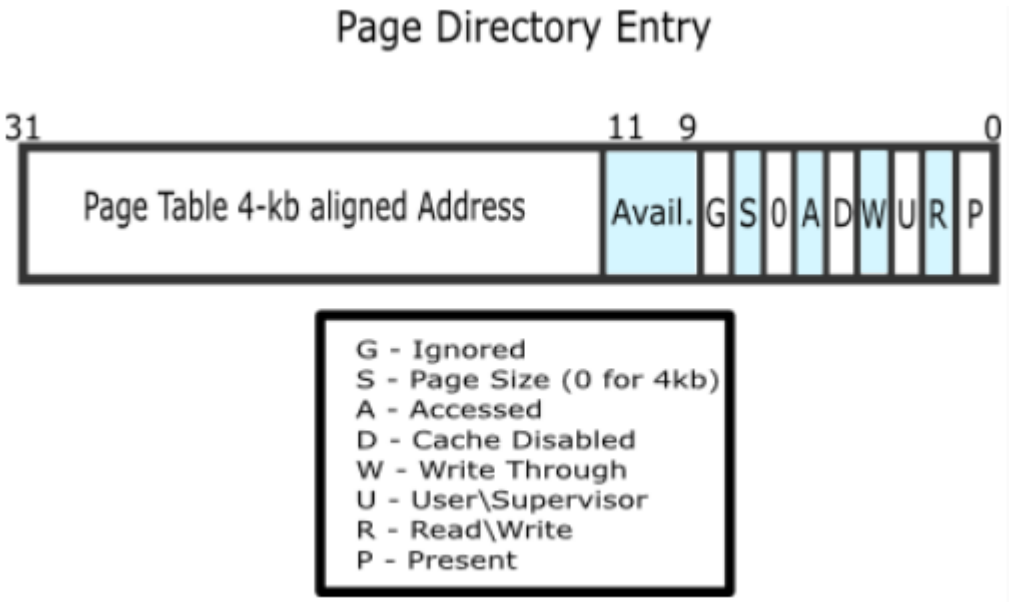
具体代码及注释如下：

```
383     //尝试获取页表
384     //线性地址la通过页目录索引，得到页表起始地址
385     pde_t *pdep = &pgdir[PDX(la)];
386     //若获取不成功
387     if (!(*pdep & PTE_P)) {
388         //申请一个物理页
389         struct Page *page;
390         if (!create || (page = alloc_page()) == NULL) {
391             return NULL;
392         }
393         //引用次数加1
394         set_page_ref(page, 1);
395         //获取物理页的线性地址
396         uintptr_t pa = page2pa(page);
397         //内存空间初始化
398         memset(KADDR(pa), 0, PGSIZE);
399         //设置页目录项内容
400         *pdep = pa | PTE_U | PTE_W | PTE_P;
401     }
402     //返回页表地址
403     return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];
404 }
```

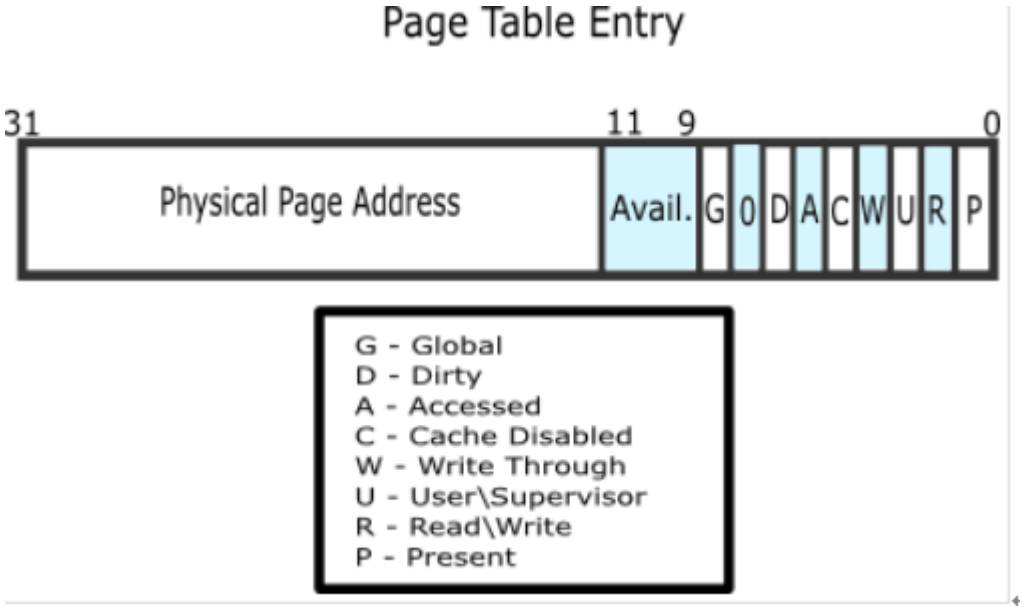
- **注意：**

1. 由于页目录和页表中存储的都是物理地址，所以当我们从页目录中获取页表的物理地址后，我们需要使用KADDR()将其转换成虚拟地址。之后就可以用这个虚拟地址直接访问对应的页表了。
2. alloc_page()获取的是物理page对应的Page结构体，而不是我们需要的物理page。通过一系列变化(page2pa())，我们可以根据获取的Page结构体得到与之对应的物理page的物理地址，之后我们就能获得它的虚拟地址。

- **问题1：**
  - 页目录表表项：



- 页表表项：



高20位表示下一级页表（或物理页表）的基址（4K对齐），后12位为标志位：从低位起第0～6位为ucore内核占用，表示指向的下一级页面的某些特性（是否存在、是否可写„„）；对于PDE，第7位 表 示如果pde的PTE_PS位为1，那么pde中所存的就不是下一级页表，而是一张4M页的起始地址；对于PTE，第7、8位强制为0；第9～11位供内核以外的软件使用。

对ucore的用处：在X86系统中，页目录表的起始物理地址存放在cr3 寄存器中， 这个地址必须是一个页对齐的地址，也就是低 12 位必须为0。在ucore 用boot_cr3（mm/pmm.c）记录这个值。

在ucore中，线性地址的的高10位作为页目录表的索引，之后的10位作为页表的的索引，所以页目录表和页表中各有1024个项，每个项占4B，所以页目录表和页表刚好可以用一个物理的页来存放。

- **问题2：**

硬件首先接受操作系统给出的中断，然后进行换页的操作。 具体中断发生时的事件顺序如下：

> 1) 硬件陷入内核，在内核堆栈中保存程序计数器。大多数机器将当前指令的各种状态信息保存在特殊的CPU寄存器中。 2) 启动一个汇编代码例程保存通用寄存器和其他易失的信息，以免被操作系统破坏。 3) 当操作系统发现一个缺页中断时，尝试发现需要哪个虚拟页面。通常一个硬件寄存器包含了这一信息，如果没有的话，操作系统必须检索程序计数器，取出这条指令，用软件分析这条指令，看看它在缺页中断时正在做什么。 4) 一旦知道了发生缺页中断的虚拟地址，操作系统检查这个地址是否有效，并检查存取与保护是否一致。如果不一致，向进程发出一个信号或杀掉该进程。如果地址有效且没有保护错误发生，系统则检查是否有空闲页框。如果没有空闲页框，执行页面置换算法寻找一个页面来淘汰。 5) 如果选择的页框"脏"了，安排该页写回磁盘，并发生一次上下文切换，挂起产生缺页中断的进程，让其他进程运行直至磁盘传输结束。无论如何，该页框被标记为忙，以免因为其他原因而被其他进程占用。 6) 一旦页框"干净"后（无论是立刻还是在写回磁盘后），操作系统查找所需页面在磁盘上的地址，通过磁盘操作将其装入。该页面被装入后，产生缺页中断的进程仍然被挂起，并且如果有其他可运行的用户进程，则选择另一个用户进程运行。 7) 当磁盘中断发生时，表明该页已经被装入，页表已经更新可以反映它的位置，页框也被标记为正常状态。 8) 恢复发生缺页中断指令以前的状态，程序计数器重新指向这条指令。 9) 调度引发缺页中断的进程，操作系统返回调用它的汇编语言。 10) 该例程恢复寄存器和其他状态信息。

3. 练习三

设计思路：使用pte2page宏即可得到从页表项得到对应的物理页面所对应的Page结构体。得到结构体后，判断此页被引用的次数，如果仅仅被引用一次，对页面9引用进行更改，降为零则这个页可以被释放否则，只能释放页表入口。 具体代码及注释如下：

```
    //判断页表是否存在
    if (*ptep & PTE_P) {
        //从页表项得到对应的物理页所对应的Page结构体
        struct Page *page = pte2page(*ptep);
        //判断此页是否被多次引用
        if (page_ref_dec(page) == 0) {
            free_page(page);
        }
        //该页目录项清零
        *ptep = 0;
        //使得TLB无效，清除二级页表项
        tlb_invalidate(pgdir, la);
    }
```

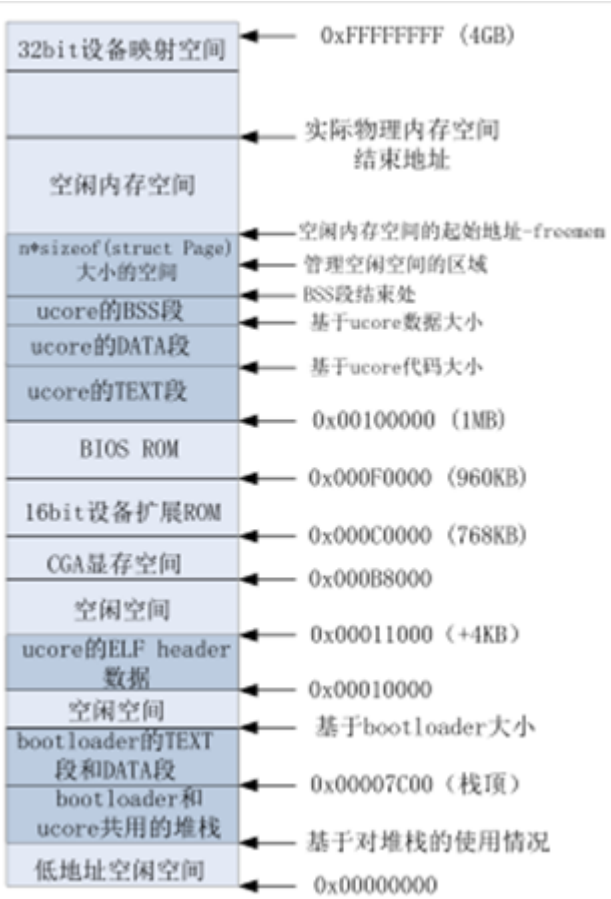
- **问题1**

  Page结构体与物理页一一对应，pte2page函数从页表项找到对应物理页的Page结构体：

```
105 static inline struct Page *
106 pte2page(pte_t pte) {
107     if (!(pte & PTE_P)) {
108         panic("pte2page called with invalid pte");
109     }
110     return pa2page(PTE_ADDR(pte));
111 }
```

  Page结构体和对应物理页的起始地址一一对应。

- **问题2**



通过上面这个图可以看到BootLoader把ucore放在了起始物理地址为0x00100000的物理内存空间内;而通过查看链接文件kernel.ld:

```
SECTIONS {
    /* Load the kernel at this address: "." means the current
address */
    . = 0xC0100000;
```

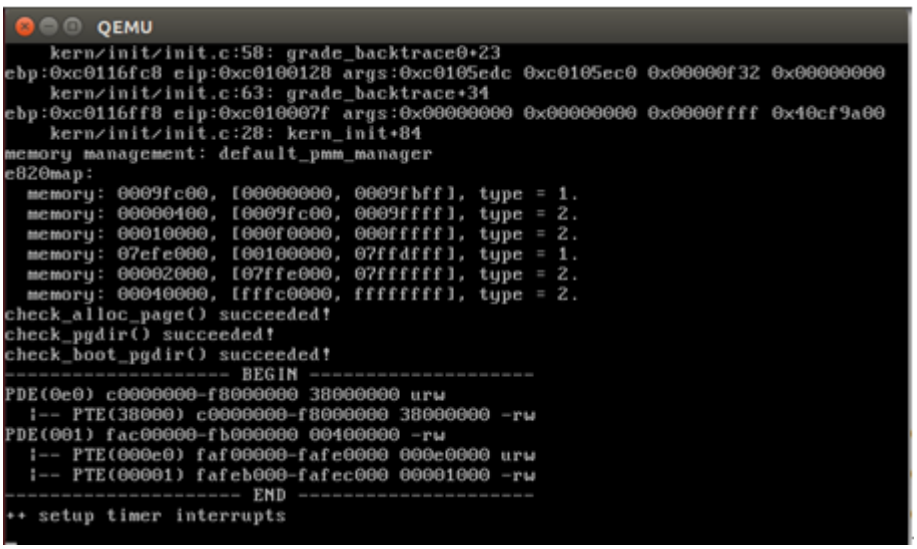
得出ucore的起始虚拟地址从0xC0100000开始。地址的映射关系为:

```
virt addr - 0xC0000000 = linear addr = phy addr
```

其中KERNBASE=0xC000000;为了使虚拟地址与物理地址相等,首先要使KERNBASE=0x0000000;同时为了满足下面的关系: PA（物理地址）=LA（线性地址）=VA（逻辑地址）-KERNBASE；虚拟地址的起始地址改成0x00100000。

- **运行结果**

```
memory management: default_pmm_manager
e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efe000, [00100000, 07ffdfff], type = 1.
  memory: 00002000, [07ffe000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ----------------------
```



四. 实验体会和思考题

通过这次关于物理内存管理的实验，我对分段、分页的机制和逻辑地址、线性地址、物理地址的转化有了更深入的理解。并理解了空闲物理页在真实操作系统是如何管理的（通过一个空闲链表)。还懂得了系统是如何通过通过中断获取IO服务的。