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

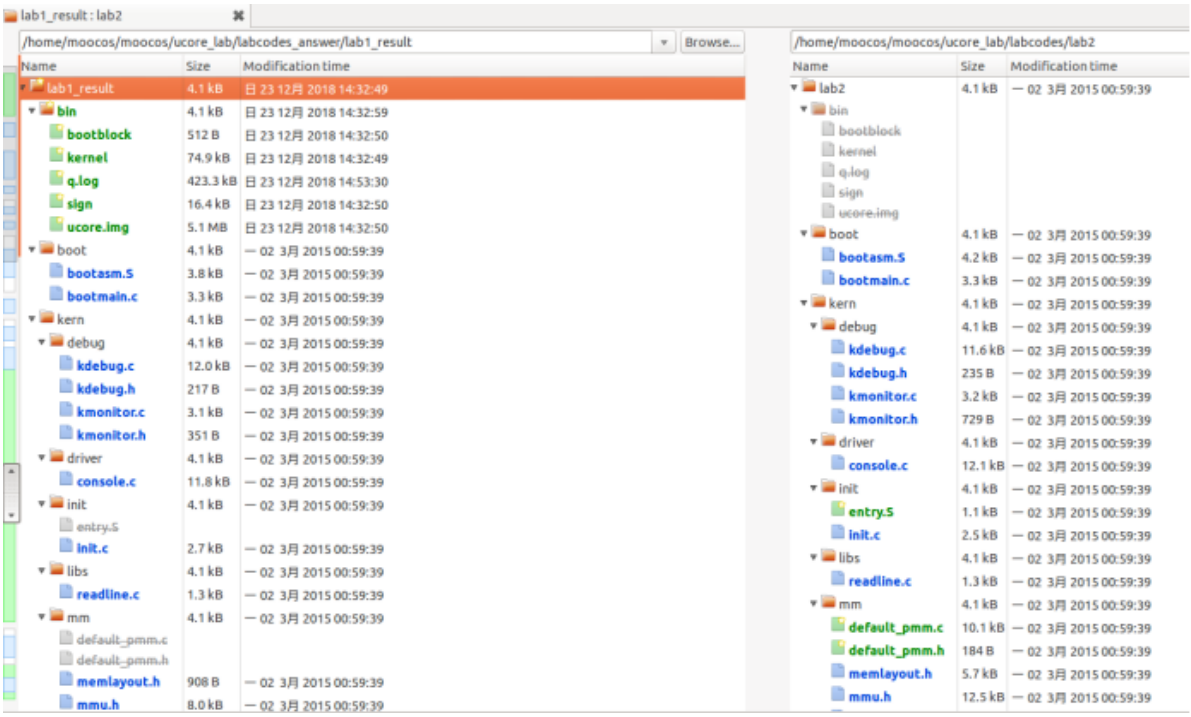
二、实验内容：

练习0：填写已有实验

对于练习0，只需要把实验1的代码填入本次实验的代码中即可，又由于实验一并不要求，因此在这里我直接将lab1_result与lab2利用Meld来比较，完成lab2的更新。

如图，其中蓝色文件为存在差异的文件，一个个排查找到lab2中标有"lab1 your code"注释的地方更新lab2的内容即可。

其中涉及到kern/debug/kdebug.c和kern/trap/trap.c两个文件



练习1：实现first-fit连续物理内存分配算法

1、实验内容：

根据题目要求，首先查看default_pmm.c文件，可以看到

```
you should rewrite functions: default_init,default_init_memmap,default_alloc_pages, default_free_pages.
```

即需要我们完善上述代码，且在该文件及实验参考书里面也详细说明了各个函数应该完成的功能，那么我们一一查看进行修改完善即可。

不过在此之前，我们需要先把涉及到的数据结构及其基本操作了解一下。

1、Page定义：

它包含了映射此物理页的虚拟页个数，描述物理页属性的flags和双向链接各个Page结构的page_link双向链表。

```
struct Page {
    int ref; // page frame's reference counter
    uint32_t flags; // array of flags that describe the status of the page frame
    unsigned int property;// the num of free block, used in first fit pm manager
    list_entry_t page_link;// free list link
};
```

其中各个成员变量的具体含义如下：

> **ref**：表示这样的页被页表引用的记数。如果这个页被页表引用了，即在某页表中有一个页表项设置了一个虚拟页到这个Page管理的物理页的映射关系，就会把Page的ref加一；反之，若页表项取消，即映射关系解除，就会把Page的ref减一。
>
> **flags**：表示此物理页的状态标记，有两种属性，bit 0表示是否被保留，如果被保留了则设为1，且不能放到空闲页链表中，即这样的页不是空闲页，不能动态分配与释放。比如内核代码占用的空间。bit 1表示此页是否是空闲的。如果设置为1，表示这页是空闲的，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。
>
> **property**：用来记录某连续内存空闲块的大小（即地址连续的空闲页的个数）。这里需要注意的是用到此成员变量的这个Page比较特殊，是连续内存空闲地址最小的一页（即第一页）。
>
> **page_link**：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。

2、free_area_t定义：

```
typedef struct {
    list_entry_t free_list; // the list header
    unsigned int nr_free; // # of free pages in this free list
} free_area_t;
```

free_area_t数据结构，包含了一个list_entry结构的双向链表指针和记录当前空闲页的个数的无符号整型变量nr_free。其中的链表指针指向了空闲的物理页表的表头。

了解了两个基本的数据结构（链表）的定义之后，接下来就是要了解其基本操作，这部分在libs/list.h里面，其实也就是常见的链表的初始化，元素的插入，删除等等操作，这里就不多说了。

实际看了上述内容后，很容易意识到实现first-fit算法，无非是让我们去完成一系列关于链表的操作，有了这个思路后开始看代码。

其中需要我们修改：

- default_init：对物理内存管理器的初始化；
- default_init_memmap：对管理的空闲页的数据进行初始化；
- default_alloc_pages：申请分配指定数量的物理页；
- default_free_pages: 申请释放若干指定物理页；

下面分别来看：

1. **default_init**：不用修改，直接使用即可，我就只是添加了注释：

```
//建立空闲页链表表头
default_init(void) {
    list_init(&free_list); //free_list 是空闲页表链表的表头
    nr_free = 0; //初始化时，链表中无空闲页
}
```

2. **default_init_memmap**：修改后的代码大体上与原来没什么区别，主要是原代码仅仅是将base->page_link插入了free_list，其余的p并没有加入链表（个人感觉好像是这样），以及原代码只设置了base的Property而没有设置每一个p的Property，所以将上述两条代码加入了for循环，并且传入参数改为p。

```
//初始化空闲页链表
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0); //空闲页个数应大于0
    struct Page *p = base; //起始地址
    for (; p != base + n; p ++) {
        assert(PageReserved(p)); //确认本页是否为保留页(若是则不分配)
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    base->property = n; //第一页记录连续空闲页数量
    SetPageProperty(base);
    nr_free += n; //空闲页数量增加n
    list_add(&free_list, &(base->page_link)); //插入空闲页链表
}

//修改后：
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0); //空闲页个数应大于0
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p)); //确认本页是否为保留页(若是则不分配)
        SetPageProperty(p);
        p->flags = p->property = 0;
        set_page_ref(p, 0); //初始化时无引用
        list_add(&free_list, &(p->page_link)); //将p插入空闲页链表
    }
    base->property = n; //第一页记录连续空闲页数量
    nr_free += n; //空闲页数量增加n
}
```

3. **default_alloc_pages**：

```
static struct Page * //分配
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) { //空间不够
        return NULL;
    }
    struct Page *page = NULL;
```

```
    struct Page *p = NULL;
    list_entry_t *le = &free_list; //从头开始找
    list_entry_t *len;

    while ((le = list_next(le)) != &free_list) {
        p = le2page(le, page_link);
        if (p->property >= n) { //找到适合的块
            page = p;
            break;
        }
    }
    if (page != NULL) {
        for(int i=0;i<n;i++){ //删除page往后的n个页块节点（被分配出去了），由于
list_del函数的定义保证了删除中间节点之后，被删除节点的前后节点会连接起来，因此没有必要再对
前后的节点进行操作
          len = list_next(le);
          struct Page *pp = le2page(le, page_link);
          SetPageReserved(pp);
          ClearPageProperty(pp);
          list_del(le);
          le = len;
        }
        if (page->property > n) { //空闲块大于n,设置已分配出去页块节点的后一个节点的
property（若==n，则已经分配完了，自然不需要这一步操作）
            (le2page(le,page_link))->property = p->property - n;
        }
        ClearPageProperty(p);
        SetPageReserved(p);
        nr_free -= n; //剩余空闲页数量-n
        return p;
    }
    return NULL;
}
```

4. **default_free_pages:**

   这部分自己写的代码一直运行不成功，于是搬来了答案...自己补了点注释

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    assert(PageReserved(base));

    list_entry_t *le = &free_list;
    struct Page * p;
    while((le=list_next(le)) != &free_list) {
      p = le2page(le, page_link);
      if(p>base){
        break; //在free_list里面查找一个插入的位置
      }
    }

    for(p=base;p<base+n;p++){
      list_add_before(le, &(p->page_link)); //插入n个空闲块
    }
    base->flags = 0;
```

```
    set_page_ref(base, 0);
    ClearPageProperty(base);
    SetPageProperty(base);
    base->property = n;

    p = le2page(le,page_link) ;
    if( base+n == p ){ //向高地址合并
      base->property += p->property;
      p->property = 0;
    }
    //向低地址合并
    le = list_prev(&(base->page_link));
    p = le2page(le, page_link);
    if(le!=&free_list && p==base-1){
      while(le!=&free_list){
        if(p->property){
          p->property += base->property;
          base->property = 0;
          break;
        }
        le = list_prev(le);
        p = le2page(le,page_link);
      }
 }

    nr_free += n;
    return ;
}
```

2、回答提问：

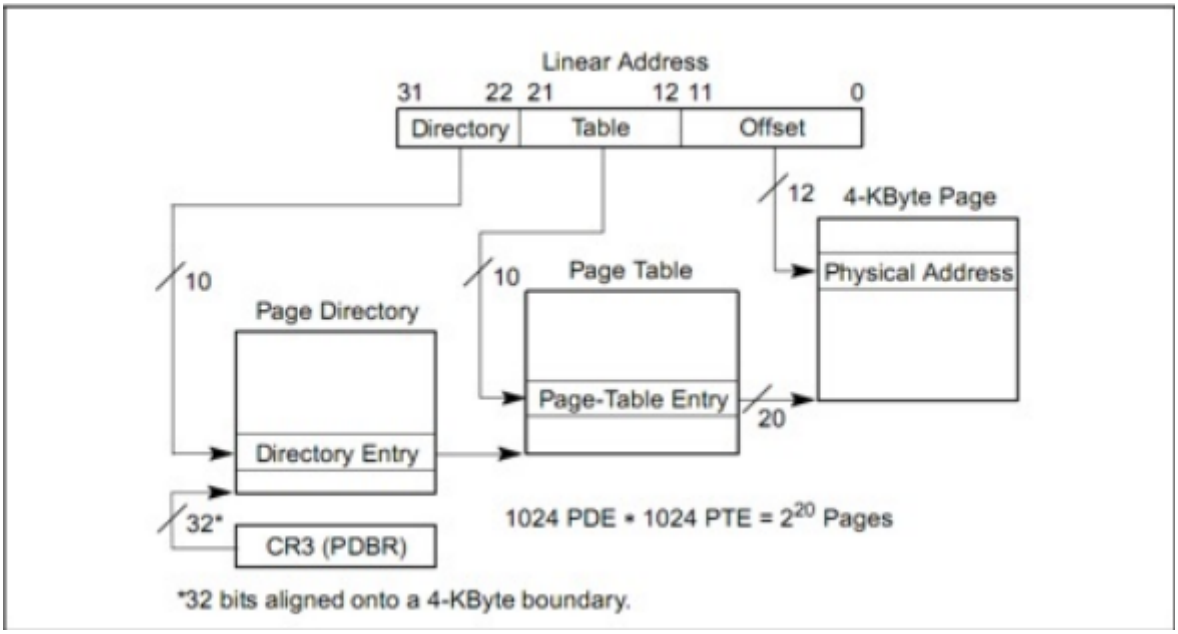
- 你的first-fit算法是否有进一步的改进空间?

  first-fit算法本质上就是对链表中节点的操作（增加、删除节点之类的），要说改进的话，感觉上也只能从查找的效率上入手。不过对于链表这个数据结构而言，好像确实也没有什么特别有效的查找算法，如果可能的话，以别的数据结构来组织空闲页链表想必查找性能上会有很大的提升，比如树。

练习2：实现寻找虚拟地址对应的页表项

1、实验内容：

练习2实际就是要求我们完成如图所示二级页表的建立：



可能看起来挺复杂的，不过我们需要完成的仅仅是get_pte()函数，也就是根据页目录表项给出的结果找到相应的页表项的地址。总之大致按照注释一条一条写就差不多了。

可能用到的MACROs or Functions：

- PDX(la) = the index of page directory entry of VIRTUAL ADDRESS la。即根据虚地址la找到相应页目录表项的索引；
- KADDR(pa) : takes a physical address and returns the corresponding kernel virtual address。根据物理地址找到对应的内核虚地址；
- set_page_ref(page,1) : means the page be referenced by one time。设置该页被引用一次；
- page2pa(page): get the physical address of memory which this (struct Page *) page manages。返回该Page管理的物理页地址；
- struct Page * alloc_page() : allocation a page。分配一页；
- 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。将s指向的地址开始往后n个bytes的数据设置为c；
- PTX(la)：la在页表项的偏移量。

代码如下：（保留了原注释，方便看每一条代码是干什么的）

```
pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    pde_t *pdep = &pgdir[PDX(la)];   // (1) find page directory entry。pgdir为一级页表的首地址，传入对应页目录表项的索引来得到对应的页目录表项
    if (!(*pdep & PTE_P)) {              // (2) check if entry is not present。判断pdep是否合法，以及存在标志位PTE_P是否为1。若不存在，则进入if（若不存在则直接执行最后一句返回NULL）
        struct Page *page;
        if(create)      // (3) check if creating is needed, then alloc page for page table
            page = alloc_page();
        if(!create || (page == NULL)) //不需要创建或创建失败则返回NULL
            return NULL;            // CAUTION: this page is used for page table, not for common data page
        set_page_ref(page, 1);                    // (4) set page reference
        uintptr_t pa = page2pa(page); // (5) get linear address of page（这个地方应该是先得到物理地址吧）
        memset(KADDR(pa), 0, PGSIZE);                    // (6) clear page content using memset
```

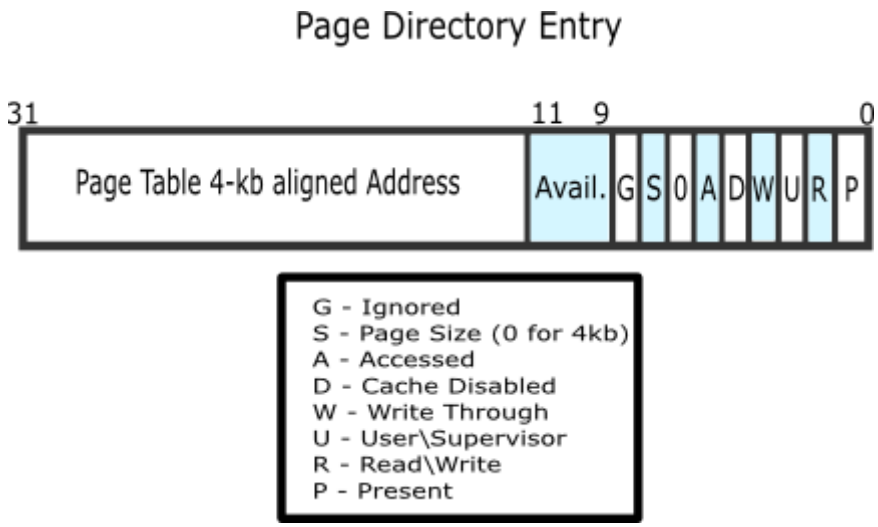
```
        *pdep = pa | PTE_U | PTE_W |PTE_P;                    // (7) set page
directory entry's permission。
    }
    return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];          // (8) return
page table entry
}
```

2、回答提问：

1. 请描述页目录项（Page Directory Entry）和页表（Page Table Entry）中每个组成部分的含义以及对ucore而言的潜在用处。

   下面是网上直接找的图：

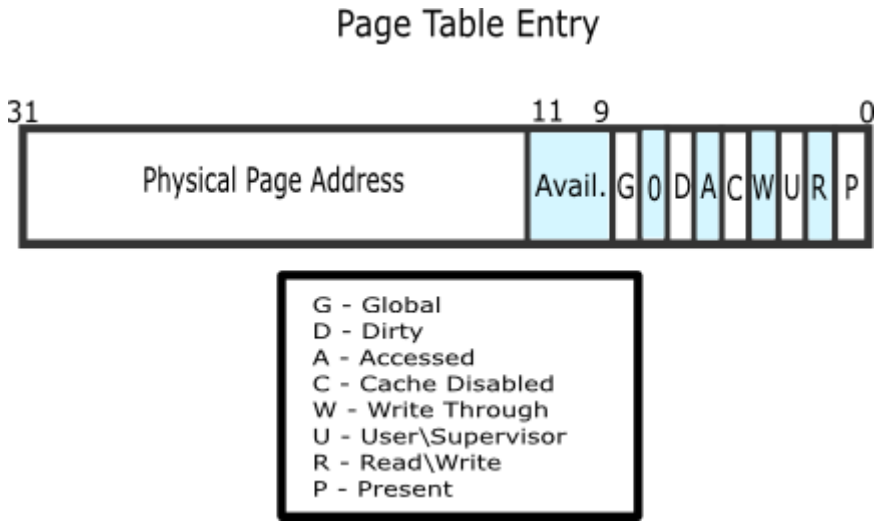
   PDE:



   从低到高，分别是：

   - *P* (Present) 位：表示该页保存在物理内存中。
   - *R* (Read/Write) 位：表示该页可读可写。
   - *U* (User) 位：表示该页可以被任何权限用户访问。
   - *W* (Write Through) 位：表示 CPU 可以直写回内存。
   - *D* (Cache Disable) 位：表示不需要被 CPU 缓存。
   - *A* (Access) 位：表示该页被写过。
   - *S* (Size) 位：表示一个页 4MB 。
   - 9-11 位保留给 OS 使用。
   - 12-31 位指明 PTE 基址地址。

   PTE:



从低到高，分别是：

- 0-3 位同 PDE。
- *C* (Cache Disable) 位：同 PDE *D* 位。
- *A* (Access) 位：同 PDE 。
- *D* (Dirty) 位：表示该页被写过。
- *G* (Global) 位：表示在 CR3 寄存器更新时无需刷新 TLB 中关于该页的地址。
- 9-11 位保留给 OS 使用。
- 12-31 位指明物理页基址。

对ucore的作用：

无论是PDE还是PTE，可以看到它们并没有将自己的32个bit全部用完，而是保留了几位供操作系统使用，那么ucore就可以利用这些保留位做一些记录之类的。

2. 如果ucore执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情？
   - 将引发页访问异常的地址被保存在cr2寄存器中；
   - 保护现场，各寄存器值进栈；
   - 设置错误代码，触发Page_Fault；
   - 查询到Page_Fault对应的ISR地址，跳转到ISR执行；
   - 执行完毕后恢复现场。

练习3：释放某虚地址所在的页并取消对应二级页表项的映射

1、实验内容：

根据题目，首先在pmm.c中找到page_remove_pte()函数，同样是对照着注释一步一步写下去。

会用到的MACROs or Functions：

- struct Page *page pte2page(*ptep): get the according page from the value of a ptep（由页表项得到相应的Page）
- free_page : free a page（释放Page）
- page_ref_dec(page) : decrease page->ref. NOTICE: if page->ref == 0 , then this page should be free.（释放掉某虚地址所在的页后，该物理页的引用次数-1，若-1后为0，则释放该页（也就是调用free_page））

- 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.

```
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    if (*ptep & PTE_P) {                      //(1) check if page directory is
present
        struct Page *page = pte2page(*ptep); //(2) find corresponding page to
pte
        if(page_ref_dec(page) == 0)                          //(3) decrease page
reference
            free_page(page);                        //(4) and free this page when
page reference reachs 0
        *ptep = 0;                             //(5) clear second page table entry
        tlb_invalidate(pgdir, la);                           //(6) flush tlb
    }
}
```

2、回答提问：

1. 数据结构Page的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系？如果有，其对应关系是啥？

   有对应关系。页目录项以及页表项中保存的物理地址就对应到Page数组中的某一项，也就是可以通过这个物理地址来获取到Page数组中对应的这一项。

2. 如果希望虚拟地址与物理地址相等，则需要如何修改lab2，完成此事？ 鼓励通过编程来具体完成这个问题

   参考实验指导书中系统执行中地址映射的四个阶段。

   由于在lab1中有：`virt addr = linear addr = phy addr`，因此考虑将lab2往lab1上靠。

   先修改tools/kernel.ld，将虚拟地址改为0x100000：

```
ENTRY(kern_entry)
SECTIONS {
    /* Load the kernel at this address: "." means the current address */
    . = 0x0100000;
    .text : {
        *(.text .stub .text.* .gnu.linkonce.t.*)
    }
```

然后把kernel基址改为0：

```
/* All physical memory mapped at this address */
#define KERNBASE            0x00000000
```

并注释掉取消0~4M区域内存页映射的代码。

三、运行结果：

make qemu编译运行：

```
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc0106043 (phys)
  edata  0xc0117a36 (phys)
  end    0xc01189c8 (phys)
Kernel executable memory footprint: 99KB
ebp:0xc0116f38 eip:0xc01009e0 args:0x00010094 0x00000000 0xc0116f68 0xc01000c1
    kern/debug/kdebug.c:308: print_stackframe+21
ebp:0xc0116f48 eip:0xc0100ccf args:0x00000000 0x00000000 0x00000000 0xc0116fb8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc0116f68 eip:0xc01000c1 args:0x00000000 0xc0116f90 0xffff0000 0xc0116f94
    kern/init/init.c:48: grade_backtrace2+33
ebp:0xc0116f88 eip:0xc01000ea args:0x00000000 0xffff0000 0xc0116fb4 0x00000029
    kern/init/init.c:53: grade_backtrace1+38
ebp:0xc0116fa8 eip:0xc0100108 args:0x00000000 0xc010002a 0xffff0000 0x0000001d
    kern/init/init.c:58: grade_backtrace0+23
ebp:0xc0116fc8 eip:0xc010012d args:0xc010607c 0xc0106060 0x00000f92 0x00000000
    kern/init/init.c:63: grade_backtrace+34
ebp:0xc0116ff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    kern/init/init.c:28: kern_init+84
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: 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 ----------------------
++ setup timer interrupts
0: @ring 0
0:  cs = 8
0:  ds = 10
0:  es = 10
0:  ss = 10
+++ switch to  user  mode +++
100 ticks
100 ticks
100 ticks
100 ticks
```

四、实验总结：

OS的第一次实验，首先就是感觉有点迷糊，一开始不知道该如何入手。其与别的实验的最大不同就是实验中大量的时间是在读，写代码的时间反而不多。或许是因为初次接触的原因，导致第一次实验花费了大量的时间才勉强将其完成，整体效果也不是很好，较大程度上参考了答案。虽然说最后通过参考答案以及网上一些代码使得最终运行成功，但其实我自己对于本次实验内容的认识并不深刻，特别是对于实验后需要回答的问题，我到现在其实还是一知半解。但是不管怎么说，通过这次实验，多多少少还是有点收获，一方面算是复习了链表的操作，一方面算是预习了理论课的内存管理部分，还有就是通过自己亲手试验，大致明白了操作系统是如何建立对物理内存的管理的，以及加深了对基于段页式的内存地址管理机制的认识，也希望下一次实验能够做得更好，收获更多。