实验项目：物理内存管理

姓名：张伟焜　学号：17343155　邮箱：zhangwk8@mail2.sysu.edu.cn
院系：数据科学与计算机学院 专业：17 级软件工程 指导教师：张永东

【实验题目】

物理内存管理

【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

【实验要求】

根据指导，完成练习 0~3。

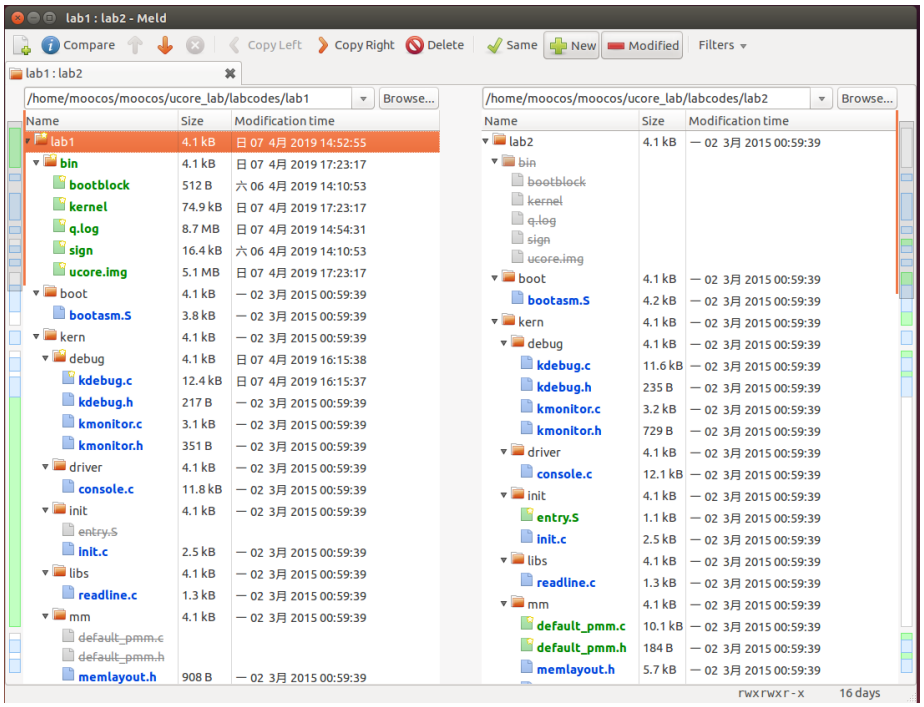
【实验方案】

实验环境：老师提供的虚拟机（Virtual box），无特殊硬件要求
实验思路：根据实验指导，先了解理论知识，再进行实验

【实验过程】

练习 0：填写已有实验。

使用 meld 软件将 ucore 启动实验的代码导入。



注意要点击标星文件进行对比，将上次实验完成的函数复制过来，不要将整个文件进行覆盖。上次实验修改的内容集中在 kdebug.c 和 trap.c

练习 1：实现 first-fit 连续物理内存分配算法。

首先，根据注释中的提示，完成准备工作：

```
16 * (1) Prepare: In order to implement the First-Fit Mem Alloc (FFMA), we should manage the free mem block
   use some list.
17 *              The struct free_area_t is used for the management of free mem blocks. At first you should
18 *              be familiar to the struct list in list.h. struct list is a simple doubly linked list
   implementation.
19 *              You should know howto USE: list_init, list_add(list_add_after), list_add_before, list_del,
   list_next, list_prev
20 *              Another tricky method is to transform a general list struct to a special struct (such as
   struct page):
21 *              you can find some MACRO: le2page (in memlayout.h), (in future labs: le2vma (in vmm.h),
   le2proc (in proc.h),etc.)
```

first-fit 连续物理内存分配算法涉及到的的函数有 default_init(), default_init_memmap(), default_alloc_pages()，default_free_pages()。

default_init()函数。根据提示该可以被重用，用于初始化 free_list 和置零 nr_free：

```
22 * (2) default_init: you can reuse the  demo default_init fun to init the free_list and set nr_free to 0.
23 *              free_list is used to record the free mem blocks. nr_free is the total number for free mem
   blocks.
```

```
74 static void
75 default_init(void) {
76     list_init(&free_list);
77     nr_free = 0;
78 }
```

default_init_memmap()函数。

```
24 * (3) default_init_memmap:  CALL GRAPH: kern_init --> pmm_init-->page_init-->init_memmap--> pmm_manager-
   >init_memmap
25 *              This fun is used to init a free block (with parameter: addr_base, page_number).
26 *              First you should init each page (in memlayout.h) in this free block, include:
27 *                  p->flags should be set bit PG_property (means this page is valid. In pmm_init fun (in
   pmm.c),
28 *                  the bit PG_reserved is setted in p->flags)
29 *                  if this page  is free and is not the first page of free block, p->property should be
   set to 0.
30 *                  if this page  is free and is the first page of free block, p->property should be set
   to total num of block.
31 *                  p->ref should be 0, because now p is free and no reference.
32 *                  We can use p->page_link to link this page to free_list, (such as: list_add_before
   (&free_list, &(p->page_link)); )
33 *              Finally, we should sum the number of free mem block: nr_free+=n
```

该函数用于初始化空闲块，将起始地址为 base 的 n 个连续页加入到内存中。首先，把每个空闲页的 flags 和 property 位置零，接着用函数 SetPageProperty()设置标志位。其中，PageReserved（p）函数是用来判断此页是否为保留页的。得到代码如图：

```
80 static void
81 default_init_memmap(struct Page *base, size_t n) {
82     assert(n > 0);
83     struct Page *p = base;
84     for (; p != base + n; p ++) {
85         assert(PageReserved(p));
86         p->flags = 0;
87         p->property = 0;
88         SetPageProperty(p);
89         set_page_ref(p, 0);
90         list_add_before(&free_list, &(p->page_link));
91     }
92     base->property = n;
93     nr_free += n;
94 }
```

default_alloc_pages()函数。该函数用于寻找合适的空闲块。

```
34 *| (4) default_alloc_pages: search find a first free block (block size >=n) in free list and reszie the
   free block, return the addr
35 *             of malloced block.
36 *             (4.1) So you should search freelist like this:
37 *                     list_entry_t le = &free_list;
38 *                     while((le=list_next(le)) != &free_list) {
39 *                     ....
40 *                 (4.1.1) In while loop, get the struct page and check the p->property (record the num of
   free block) >=n?
41 *                     struct Page *p = le2page(le, page_link);
42 *                     if(p->property >= n){ ...
43 *                 (4.1.2) If we find this p, then it' means we find a free block(block size >=n), and the
   first n pages can be malloced.
44 *                     Some flag bits of this page should be setted: PG_reserved =1, PG_property =0
45 *                     unlink the pages from free_list
46 *                     (4.1.2.1) If (p->property >n), we should re-caluclate number of the the rest of
   this free block,
47 *                           (such as: le2page(le,page_link))->property = p->property - n;)
48 *                 (4.1.3)  re-caluclate nr_free (number of the the rest of all free block)
49 *                 (4.1.4)  return p
50 *               (4.2) If we can not find a free block (block size >=n), then return NULL
```

修改前的函数已经完成了判断 n 的合法性以及最大能分配空间与 n 的关系；完成了空闲链表向页的转换；若页大小大于需要大小，记录页地址；若找到符合条件的页就从 free_list 删去该页；分割页块，从第 n+1 块把剩下的空闲部分重新加入 free_list。

我们需要在此基础上修改代码，找到合适的空闲块后设置标志位，把表头给 page，并在函数结束时返回 page。代码及注释如下:

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);     //判断n是否大于0
    if (n > nr_free) {//如果最大能分配空间小于n就返回
        return NULL;
    }
    list_entry_t *le, *len;
    le = &free_list;  //记录空闲链表的头部
    while((le=list_next(le)) != &free_list) {
      struct Page *p = le2page(le, page_link);   //转换为页结构
      if(p->property >= n){    //若页大小大于n,即满足需要
        int i;
        for(i=0;i<n;i++){
          len = list_next(le);
          struct Page *pp = le2page(le, page_link);
          SetPageReserved(pp); //设置每一页的标志位
          ClearPageProperty(pp);
          list_del(le);    //将此页从free_list中删除
          le = len;
        }
        if(p->property>n){//如果满足条件，分割页块
          (le2page(le,page_link))->property = p->property - n;
        }
        ClearPageProperty(p);
        SetPageReserved(p);
        nr_free -= n; //减去已经分配的页块大小
        return p;
      }
```

```
    }
    return NULL;
}
```

default_free_pages()函数。该函数释放已经使用完的页，把他们重新加入到 freelist 中。在 freelist 中查找合适的位置以供插入。参数为空闲块地址和大小。

```
51 * (5) default_free_pages: relink the pages into  free list, maybe merge small free blocks into big free
   blocks.
52 *               (5.1) according the base addr of withdrawed blocks, search free list, find the correct
   position
53 *                     (from low to high addr), and insert the pages. (may use list_next, le2page,
   list_add_before)
54 *               (5.2) reset the fields of pages, such as p->ref, p->flags (PageProperty)
55 *               (5.3) try to merge low addr or high addr blocks. Notice: should change some pages's p-
   >property correctly.
56 */
```

首先判断要释放的块是否已经被分配，然后设置标志位后遍历链表，找到前后对应的位置再进行合并。代码及注释如下：

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    assert(PageReserved(base));     //检查此页块（需要释放的页块）是否已经被分配
    list_entry_t *le = &free_list;
    struct Page * p;
    while((le=list_next(le)) != &free_list) {    //找位置
      p = le2page(le, page_link); //链表对应的page
      if(p>base){
        break;
      }
    }
    for(p=base;p<base+n;p++){
      list_add_before(le, &(p->page_link)); //将每一空闲块对应的链表插入空闲链表
    }
    base->flags = 0;  //标志位设为0
    set_page_ref(base, 0);
    ClearPageProperty(base);
    SetPageProperty(base);
    base->property = n;
    //向高地址合并
    p = le2page(le,page_link) ;
    if( base+n == p ){
      base->property += p->property;
      p->property = 0;
    }
    //向低地址合并
    //需要是低位且在范围内
    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 ;
}
```

你的 first fit 算法是否有进一步的改进空间?

分析代码，用双向链表进行查找时时间开销较大，default_alloc_pages 过程和 default_free_pages 过程都需要 O(n)的复杂度。

为降低复杂度，可以使用树形结构，default_alloc_pages 过程为深度优先遍历，复杂度还是是 O(n)，但是 default_free_pages 在查找插入位置时可以使用二分查找降低为 O(logn)的复杂度。

练习 2：实现寻找虚拟地址对应的页表项。

```
357     * Some Useful MACROs and DEFINEs, you can use them in below implementation.
358     * MACROs or Functions:
359     *   PDX(la) = the index of page directory entry of VIRTUAL ADDRESS la.
360     *   KADDR(pa) : takes a physical address and returns the corresponding kernel virtual address.
361     *   set_page_ref(page,1) : means the page be referenced by one time
362     *   page2pa(page): get the physical address of memory which this (struct Page *) page  manages
363     *   struct Page * alloc_page() : allocation a page
364     *   memset(void *s, char c, size_t n) : sets the first n bytes of the memory area pointed by s
365     *                                       to the specified value c.
366     * DEFINEs:
367     *   PTE_P           0x001                   // page table/directory entry flags bit : Present
368     *   PTE_W           0x002                   // page table/directory entry flags bit : Writeable
369     *   PTE_U           0x004                   // page table/directory entry flags bit : User can access
370     */
```

根据注释部分，我们得到一些宏定义的用途:

PDX(la): 返回虚拟地址 la 的页目录索引

KADDR(pa): 返回物理地址 pa 相关的内核虚拟地址

set_page_ref(page,1): 设置此页被引用一次

page2pa(page): 得到 page 管理的那一页的物理地址

struct Page * alloc_page() : 分配一页出来

memset(void * s, char c, size_t n) : 设置 s 指向地址的前面 n 个字节为字节'c'

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

```
371#if 0
372    pde_t *pdep = NULL;   // (1) find page directory entry
373    if (0) {              // (2) check if entry is not present
374                          // (3) check if creating is needed, then alloc page for page table
375                          // CAUTION: this page is used for page table, not for common data page
376                          // (4) set page reference
377        uintptr_t pa = 0; // (5) get linear address of page
378                          // (6) clear page content using memset
379                          // (7) set page directory entry's permission
380    }
381    return NULL;          // (8) return page table entry
382#endif
```

根据上述提示。首先找到页目录项，接着判断页表是否存在。若发现对应的二级页表不存在，则需要根据 create 参数的值来决定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，则需要申请一个新的物理页，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。接下来设置引用次数，获得页的线性地址并用 memset 清除页内容，然后设置控制位，最后返回页表元素。

代码如下：

```
383         pde_t *pdep = &pgdir[PDX(la)];
384         if (!(*pdep & PTE_P)) {
385                 struct Page *page;
386                 if (!create || (page = alloc_page()) == NULL) {
387                         return NULL;
388                 }
389                 set_page_ref(page, 1);
390                 uintptr_t pa = page2pa(page);
391                 memset(KADDR(pa), 0, PGSIZE);
392                 *pdep = pa | PTE_U | PTE_W | PTE_P;
393         }
394         return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];
395 }
```

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

页目录项共 32 位，其中：

31-12 位：页表 4KB 对齐地址，在 ucore 中为对应的页表地址；

11-9 位：Avail 位，该字段保留专供程序使用。处理器不会修改这几位。对应 Ucore 中的 PTE_AVAIL 位。

第 8 位：忽略位，无具体作用。

第 7 为：page size 位，记录页大小，对应 ucore 的 PTE_PS。

第 6 位：0 保留位，对应 ucore 的 PTE_MBZ 位。

第 5 位：Accessed 位，是已访问（Accessed）标志。处理器访问页目录表项映射的任何页面时，页目录表项的这个标志被置为 1。处理器负责设置该标志，操作系统可通过定期地复位该标志来统计页面的使用情况。对应 ucore 的 PTE_A 位。

第 4 位：Cache Disabled 位，当其为 1 时，物理页面是不能被 Cache 的；当其为 0 时允许 Cache，对应 ucore 的 PTE_PCD 位。

第 3 位：Write-Through 标志位，当其为 1 时，使用 Write-Through 的 Cache 类型；当其为 0 时，使用 Write-Back 的 Cache 类型。对应 ucore 的 PTE_PWT 位。

第 2 位：User/Supervisor 位，为 1 时，允许所有特权级别的程序访问；为 0 时，仅允许特权级为 0、1、2 的程序访问。对应 ucore 的 PTE_U 位。

第 1 位：Read/Write 位，读写标志。为 1 表示页面可以被读写，为 0 表示只读。当处理器运行在 0、1、2 特权级时，此位不起作用。页目录中的这个位对其所映射的所有页面起作用。对应 ucore 的 PTE_W 位。

第 0 位：Present 位，存在位。为 1 表示页表或者页位于内存中。否则，表示不在内存中，必须先予以创建或者从磁盘调入内存后方可使用。对应 ucore 的 PTE_P 位。

页表项共 32 位，其中：（与页目录项不同的用黄色标出）
31-12 位：20 位物理地址，在 ucore 中对应物理地址的高 20 位；
11-9 位：Avail 位，该字段保留专供程序使用。处理器不会修改这几位。对应 Ucore 中的 PTE_AVAIL 位。
第 8 位：全局(Global)位，如果页是全局的，那么它将在高速缓存中一直保存。Ucore 中没有这一位。
第 7 为：0 保留位，对应 ucore 的 PTE_MBZ 位。
第 6 位：Dirty 位。由处理器固件设置，用来表明此表项所指向的页是否进行过写操作。对应 ucore 的 PTE_D 位。
第 5 位：Accessed 位，是已访问（Accessed）标志。处理器访问页表项映射的页面时，这个标志就会被置为 1。处理器负责设置该标志，操作系统可通过定期地复位该标志来统计页面的使用情况。对应 ucore 的 PTE_A 位。
第 4 位：Cache Disabled 位，当其为 1 时，物理页面是不能被 Cache 的；当其为 0 时允许 Cache，对应 ucore 的 PTE_PCD 位。
第 3 位：Write-Through 标志位，为 1 时使用 Write-Through 的 Cache 类型；为 0 时使用 Write-Back 的 Cache 类型。对应 ucore 的 PTE_PWT 位。
第 2 位：User/Supervisor 位，为 1 时，允许所有特权级别的程序访问；为 0 时，仅允许特权级为 0、1、2 的程序访问。对应 ucore 的 PTE_U 位。
第 1 位：Read/Write 位，读写标志。为 1 表示页面可以被读写，为 0 表示只读。当处理器运行在 0、1、2 特权级时，此位不起作用。页目录中的这个位对其所映射的所有页面起作用。对应 ucore 的 PTE_W 位。
第 0 位：Present 位，存在位。为 1 表示页表或者页位于内存中。否则，表示不在内存中，必须先予以创建或者从磁盘调入内存后方可使用。对应 ucore 的 PTE_P 位。

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?

（1）将引发页访问异常的地址将被保存在 cr2 寄存器中
（2）设置错误代码
（3）引发 Page Fault

练习 3：释放某虚拟地址所在的页并取消对应二级页表项的映射。

首先根据注释了解需要用到的相关宏定义：

```
421     * Some Useful MACROs and DEFINEs, you can use them in below implementation.
422     * MACROs or Functions:
423     *   struct Page *page pte2page(*ptep): get the according page from the value of a ptep
424     *   free_page : free a page
425     *   page_ref_dec(page) : decrease page->ref. NOTICE: ff page->ref == 0 , then this page should be
   free.
426     *   tlb_invalidate(pde_t *pgdir, uintptr_t la) : Invalidate a TLB entry, but only if the page tables
   being
427     *                        edited are the ones currently in use by the processor.
428     * DEFINEs:
429     *   PTE_P           0x001                   // page table/directory entry flags bit : Present
430     */
```

```
#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
```

根据上述注释，我们首先判断页表项是否存在，接着找到对应的页表项，递减页引用，并且当页引用为 0 时把页释放，最后清零页目录项并使之无效。

对应的代码如下：

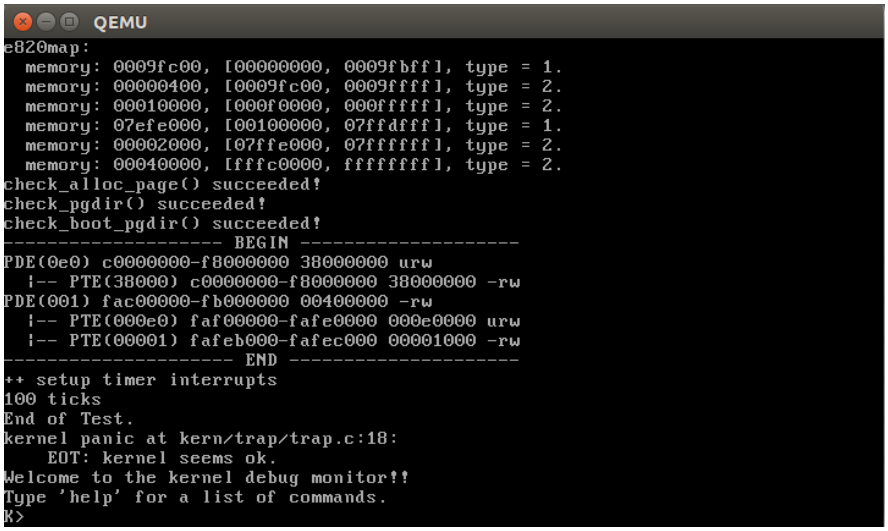
```
        if (*ptep & PTE_P) {
                struct Page *page = pte2page(*ptep);
                if (page_ref_dec(page) == 0) {
                        free_page(page);
                }
                *ptep = 0;
                tlb_invalidate(pgdir, la);
        }
}
```

【实验结果】

make grade 结果：

```
zhangweikun$>cd moocos
zhangweikun$>cd ucore_lab
zhangweikun$>cd labcodes
zhangweikun$>cd lab2
zhangweikun$>make grade
Check PMM:                (3.3s)
  -check pmm:                                 OK
  -check page table:                          OK
  -check ticks:                               OK
Total Score: 50/50
zhangweikun$>
```

make qemu 结果：



```
zhangweikun$>cd moocos
zhangweikun$>cd ucore_lab
zhangweikun$>cd labcodes
zhangweikun$>cd lab2
zhangweikun$>make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc0105f6b (phys)
  edata  0xc0117a36 (phys)
  end    0xc0118968 (phys)
Kernel executable memory footprint: 99KB
ebp:0xc0116f38 eip:0xc01009d1 args:0x00010094 0x00000000 0xc0116f68 0xc01000bc
    kern/debug/kdebug.c:309: print_stackframe+22
ebp:0xc0116f48 eip:0xc0100cc9 args:0x00000000 0x00000000 0x00000000 0xc0116fb8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc0116f68 eip:0xc01000bc args:0x00000000 0xc0116f90 0xffff0000 0xc0116f94
    kern/init/init.c:49: grade_backtrace2+33
ebp:0xc0116f88 eip:0xc01000e5 args:0x00000000 0xffff0000 0xc0116fb4 0x00000029
    kern/init/init.c:54: grade_backtrace1+38
ebp:0xc0116fa8 eip:0xc0100103 args:0x00000000 0xc010002a 0xffff0000 0x0000001d
    kern/init/init.c:59: grade_backtrace0+23
ebp:0xc0116fc8 eip:0xc0100128 args:0xc0105f9c 0xc0105f80 0x00000f32 0x00000000
    kern/init/init.c:64: grade_backtrace+34
ebp:0xc0116ff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    kern/init/init.c:29: kern_init+84
memory management: default_pmm_manager
e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
```

```
ebp:0xc0116fc8 eip:0xc0100128 args:0xc0105f9c 0xc0105f80 0x00000f32 0x00000000
    kern/init/init.c:64: grade_backtrace+34
ebp:0xc0116ff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    kern/init/init.c:29: 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: 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
100 ticks
End of Test.
kernel panic at kern/trap/trap.c:18:
    EOT: kernel seems ok.
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

【实验总结】

完成实验后，请分析 ucore_lab 中提供的参考答案，并请在实验报告中说明你的实现与参考答案的区别

由于一开始没有思路，只懂理论知识，所以只能结合注释打代码，注释中每一步都很详细，按照注释打下来几乎和答案一模一样，当然有些语句的顺序不一样，但最终结果是一样的。

同时，有些函数实在搞不懂，也参考了答案，我自己认真搞懂后，最后在写实验报告的时候也详细地加上了注释。

列出你认为本实验中重要的知识点，以及与对应的 OS 原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）

1.First-fit 算法

对应 OS 中的连续内存分配。在理论课中，除了 first-fit 算法，还有 best-fit 算法，worst-fit 算法。first-fit 算法的思想是分配第一个足够大的孔。其查找可以从头开始，也可以从上次结束处开始查找。一旦找到足够大的空闲孔，就可以停止。

在本次实验中实现 first-fit 算法时还要考虑设置标志位，链表和页结构的转化，以及如何分割页块。

2.段页式内存管理

练习 2、3 是页式管理。用到二级页表的知识，页式内存管理将的虚拟空间分成若干长度相等的页。页式内存把内存空间按页大小划分成片或者页面，然后把页式虚拟地址与内存地址建立一一对应页表。二级页表管理通过建立页目录表，每个表项对应一个页表项，以解决页表大小过大的问题。

3.PTE、PDE

练习二涉及的问题。PTE 和 PDE 是多级页式管理的重要部分，多级页表由于页表也不连续，所以页表页本身也需要地址索引，这种地址索引称为页目录。页目录中存放着进程页表的所有页表页的地址。

列出你认为 OS 原理中很重要，但在实验中没有对应上的知识点

1.按需调页。当进程需要的页在磁盘上存储时，如何将页调入调出磁盘

2.共享页。实验没有实现共享公共代码。

3.段页式内存管理。该实验只涉及了页式内存管理，没有涉及段式内存管理。

心得体会

本次实验 1~3 的内容全都涉及到了编程，这让我体会到了操作系统理论的实际应用。理论知识容易理解，但是打代码却并非想象的那么简单，或许这就是我们开设实验课的原因之一吧。

本次实验是操作系统的内存管理，在理论课上已经涉及到相关知识点，也了解了 first-fit 算法的原理，但编程时还是遇到了困难，我总结的经验就是，一定要认真阅读注释。

注释中会比较详细地说明该函数地主要内容，并且会介绍一些可能用到的函数或宏定义，能够为我们提供很大的帮助。同时，在写实验报告时，为自己的代码加上注释，也能够帮助自己加深对内存管理机制的了解并且起到了一定的检查作用。

总之，通过这次实验，我对操作系统内存管理相关知识掌握更加熟练了，我也意识到自己的操作系统编程能力有待提高。

【参考文献】

《操作系统实验指导(清华大学)陈渝、向勇编著》