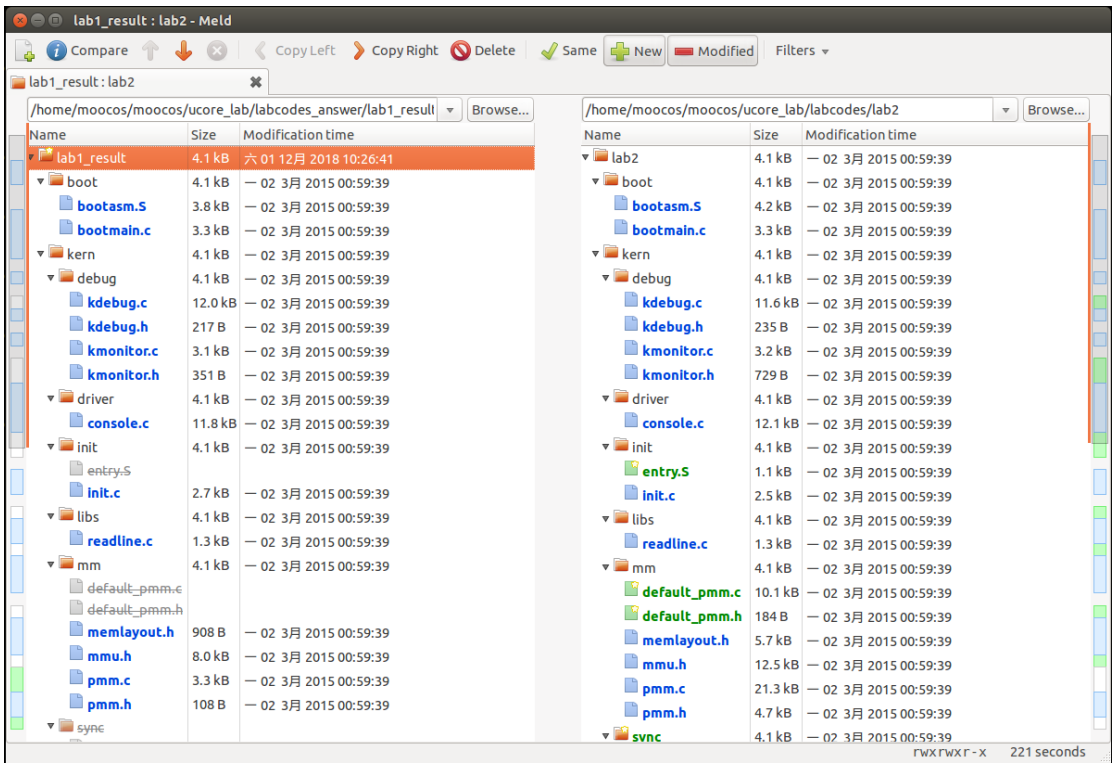
ucore lab4 实验报告

一、 实验目的

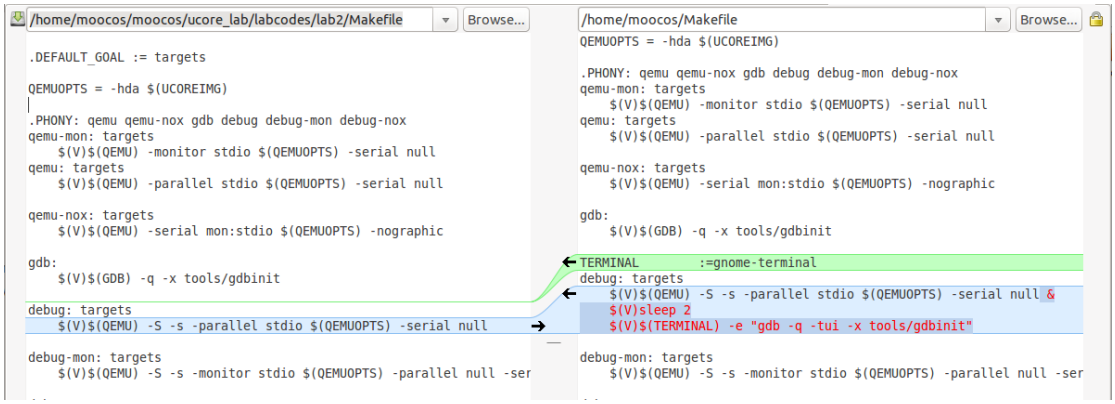
1） 理解物理内存的管理方法
2） 实现 first-fit 连续分配算法
3） 理解基于段页式内存地址的转换机制，实现页表的建立和使用

二、 实验内容

0）首先使用 meld 工具将 lab1_result 填入 lab2 中



lab2 中补全 lab1：debug/kdebug.c 里面的 print_stackframe()函数；kern/trap/trap.c 里面的 idt_init()函数和 trap_dispatch()函数
另外修改 gdbinit 和 Makefile



实验流程概述

ucore 总控函数 **kern_init()**

Getting Started × pmm.c × init.c ×

```
16
17  int
18  kern_init(void) {
19      extern char edata[], end[];
20      memset(edata, 0, end - edata);
21
22      cons_init();                // init the console
23
24      const char *message = "(THU.CST) os is loading ...";
25      cprintf("%s\n\n", message);
26
27      print_kerninfo();
28
29      grade_backtrace();
30
31      pmm_init();                 // init physical memory management
32
33      pic_init();                 // init interrupt controller
34      idt_init();                 // init interrupt descriptor table
35
36      clock_init();               // init clock interrupt
37      intr_enable();              // enable irq interrupt
38
39      //LAB1: CAHLLENGE 1 If you try to do it, uncomment lab1_switch_test()
40      // user/kernel mode switch test
41      //lab1_switch_test();
42
43      /* do nothing */
44      while (1);
45  }
```

其中 **pmm_init()函数前后**都是 lab1 的工作，lab2 有两方面改动：

① bootloader 中完成了对物理内存资源的探测工作（附录 A、B）

② bootloader 不像 lab1 直接调用 kern_init()函数，而是先调用 entry.S 中的 kern_entry()函数，主要任务是设置堆栈，并且临时建立了一个段映射关系，为之后建立分页机制做准备

entry.S ×

```
1 #include <mmu.h>
2 #include <memlayout.h>
3
4 #define REALLOC(x) (x - KERNBASE)
5
6 .text
7 .globl kern_entry
8 kern_entry:
9     # reload temperate gdt (second time) to remap all physical memory
10    # virtual_addr 0~4G=linear_addr&physical_addr -KERNBASE~4G-KERNBASE
11    lgdt REALLOC(__gdtdesc)
12    movl $KERNEL_DS, %eax
13    movw %ax, %ds
14    movw %ax, %es
15    movw %ax, %ss
16
17    ljmp $KERNEL_CS, $relocated
18
```

kern_init()函数首先完成一些输出并对 lab1 实验结果进行检查（grade_backtrace()），然后进入物理内存管理初始化工作，调用 pmm_init 函数完成物理内存的管理，然后初始化中断和异常

完成物理内存管理首先要探测可用物理内存资源（对应连续内存分配实现）；然后建立页，启动分页机制表（对应内存映射、多级页表具体实现）

1）探测系统物理内存布局

首先了解 lab1 中提到的**实模式**和**保护模式**:

在 bootloader 接手 BIOS 的工作后，系统处于**实模式**（16 位模式）运行状态，物理内存被看成分段的区域，程序代码和数据位于不同区域，操作系统和用户程序不加区分，每一个指针指向实际的物理地址。容易发生用户程序指针指向操作系统或其他用户程序区域并修改内容的错误，所以需要有一定的保护机制；在**保护模式**下，80386 的 32 位地址线才全部有效，才可采用段页式存储管理机制

ucore 启动后，在分配物理空间之前，必须获取物理内存空间的信息，比如哪些地址空间可以使用，哪些不能使用，还有多少内存可用等。**获取内存大小有两种方法：**

① BIOS 中断调用方法（实模式）

通过 BIOS 中断获取内存布局有三种方式，都是基于 INT15h 中断，分别为 88h、e801h、e820h。但是并非在所有情况下这三种方式都能工作。在 Linux kernel 里，采用的方法是依次尝试这三种方法。在本实验中通过向 INT 15h 中断传入 e820h 参数获取，因为 e820h 中断必须在实模式下使用，所以在 bootloader 进入保护模式之前调用该 BIOS 中断，并且把 e820 映射结构保存在物理地址 0x8000 处

② 直接探测（保护模式）

INT 15h 中断与 E820 参数

下面我们来看一下 ucore 中物理内存空间的信息:

```
e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07ee0000, [00100000, 07fdffff], type = 1.
  memory: 00020000, [07fe0000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
```

这里的 type 是物理内存空间的类型，1 是可以使用的物理内存空间， 2 是不能使用的物理内存空间。2 中的"不能使用"指的是这些地址不能映射到物理内存上， 但它们可以映射到 ROM 或者映射到其他设备，如各种外设等。除了这两种类型，还有几种其他类型，只是在这个实验中我们并没有使用:

```
type = 3: ACPI Reclaim Memory (usable by OS after reading ACPI tables)
type = 4: ACPI NVS Memory (OS is required to save this memory between NVS
 sessions)
type = other: not defined yet -- treat as Reserved
```

实现过程

BIOS 中断调用需要在实模式下进行，在 bootloader 进入保护模式之前完成

这种方法获取的物理内存空间的信息是用内存映射**地址描述符**(Address Range Descriptor)来表示的，一个内存映射地址描述符占 20B，其具体描述如下:

```
00h    8字节    base address            #系统内存块基地址
08h    8字节    length in bytes         #系统内存大小
10h    4字节    type of address range   #内存类型
```

每探测到一块物理内存空间，其对应的内存映射地址描述符就会被写入我们指定的内存空间(可以理解为是内存映射地址描述符表)。 完成物理内存空间的探测后，就可以通过这个表来了解物理内存空间的分布情况

下面来看 INT 15h 中断是如何进行物理内存空间的探测：

通过调用 INT 15h BIOS 中断，递增 di 的值（20 的倍数），让 BIOS 帮我们查找出一个一个的内存布局 entry，并放入到一个保存地址范围描述符结构的缓冲区中，供后续的 ucore 进一步进行物理内存管理。这个**缓冲区结构 e820map** 定义在 **memlayout.h** 中：

```
80  // some constants for bios interrupt 15h AX = 0xE820
81  #define E820MAX             20      // number of entries in E820MAP
82  #define E820_ARM            1       // address range memory
83  #define E820_ARR            2       // address range reserved
84
85  struct e820map {
86      int nr_map;
87      struct {
88          uint64_t addr;
89          uint64_t size;
90          uint32_t type;
91      } __attribute__((packed)) map[E820MAX];
92  };
```

bootasm.S 中从 probe_memory 到 finish probe 物理内存探测的实现：

```
probe_memory:
//对0x8000处的32位单元清零,即给位于0x8000处的
//struct e820map的成员变量nr_map清零
                    movl $0, 0x8000
                    xorl %ebx, %ebx
//表示设置调用INT 15h BIOS中断后，BIOS返回的映射地址描述符的起始地址
                    movw $0x8004, %di
start_probe:
                    movl $0xE820, %eax // INT 15的中断调用参数
//设置地址范围描述符的大小为20字节，其大小等于struct e820map的成员变量map的大小
                    movl $20, %ecx
//设置edx为534D4150h (即4个ASCII字符“SMAP”)，这是一个约定
                    movl $SMAP, %edx
//调用int 0x15中断，要求BIOS返回一个用地址范围描述符表示的内存段信息
                    int $0x15
//如果eflags的CF位为0，则表示还有内存段需要探测
                    jnc cont
//探测有问题，结束探测
                    movw $12345, 0x8000
                    jmp finish_probe
cont:
//设置下一个BIOS返回的映射地址描述符的起始地址
                    addw $20, %di
//递增struct e820map的成员变量nr_map
                    incl 0x8000
//如果INT0x15返回的ebx为零，表示探测结束，否则继续探测
                    cmpl $0, %ebx
                    jnz start_probe
finish_probe:
```

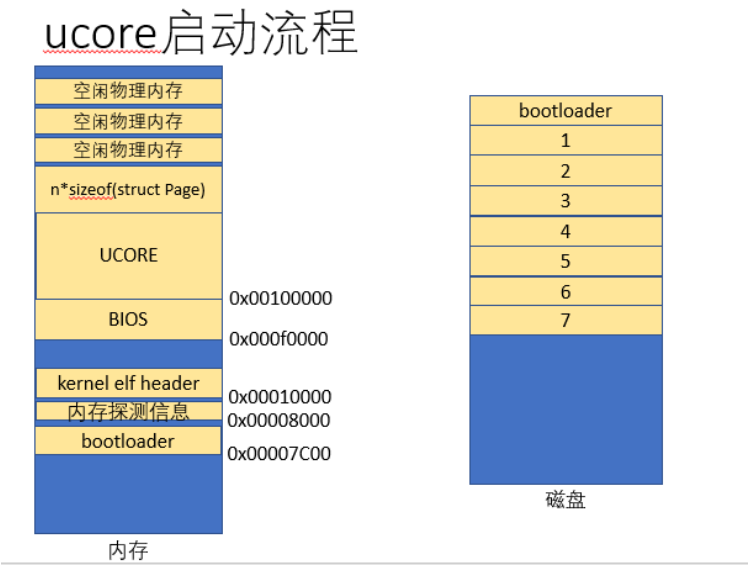
从上面代码可以看出，要实现物理内存空间的探测，大体上只需要 3 步：

1） 设置一个存放内存映射地址描述符的物理地址(这里是 0x8000)

2） 将 e820 作为参数传递给 INT 15h 中断

3） 通过检测 eflags 的 CF 位来判断探测是否结束。如果没有结束， 设置存放下一个内存映射地址描述符的物理地址，然后跳到步骤 2；如果结束，则程序结束

上述代码正常执行完毕后，在 0x8000 地址处保存了从 BIOS 中获得的内存分布信息，此信息按照 struct e820map 的设置来进行填充。这部分信息将在 bootloader 启动 ucore 后，由 ucore 的 page_init 函数来根据 struct e820map 的 memmap（定义了起始地址为 0x8000）来完成对整个机器中的物理内存的总体管理



2）以页为单位管理物理内存

获得可用范围后，系统需要建立数据结构来管理以物理页（大小为 4KB）为最小单位的整个物理内存，页的信息用 **Page 数据结构**表示

memlayout.h 中 **Page** 的定义：

Getting Started | init.c | pmm.c | memlayout.h | pmm.h

```
93
94  /* *
95   * struct Page - Page descriptor structures. Each Page describes one
96   * physical page. In kern/mm/pmm.h, you can find lots of useful functions
97   * that convert Page to other data types, such as phyical address.
98   * */
99  struct Page {
100     int ref;                        // page frame's reference counter
101     uint32_t flags;                 // array of flags that describe the status of the page frame
102     unsigned int property;          // the num of free block, used in first fit pm manager
103     list_entry_t page_link;         // free list link
104 };
105
106 /* Flags describing the status of a page frame */
107 #define PG_reserved                 0       // if this bit=1: the Page is reserved for kernel, cannot be us
108 #define PG_property                 1       // if this bit=1: the Page is the head page of a free memory bl
```

- **ref** 表示这页被页表引用的次数
- **flags** 标记此物理页的状态，第 0 位标记此页是否被保留，如果该位为 1，则不是空闲页，不能动态分配和释放；第 1 位如果为 1，则此页是空闲内存块的首页，可以被分配，如果为 0，则表示该页已经分配出去，不能进行二次分配，或者不是首页

- **property** 记录某连续内存空闲块大小，即地址连续的空闲页个数（用到此成员的是头页）
- **page_link** 是把多个连续内存空闲块链接在一起的双向链表指针

初始所有空闲物理页都是连续的，随着分配和释放，整个连续内存空闲块会分裂成一系列不连续的内存块，用双向链表管理这些小的连续内存空闲块，定义了 free_area_t 数据结构：

```
/* 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;
```

- **free_list** 是双向链表指针
- **nr_free** 记录当前空闲页个数

通过上面两个数据结构，ucore 可以管理整个以页为单位的物理内存空间。接下来需要考虑 Page 结构的起始位置和占用空间大小，初始化由 **page_init()函数**实现

Getting Started | memlayout.h | default_pmm.c | pmm.c | pmm.h | defs.h | list.h | init.c

```
static void
page_init(void) {
    struct e820map *memmap = (struct e820map *)(0x8000 + KERNBASE);
    uint64_t maxpa = 0;

    cprintf("e820map:\n");
    int i;
    for (i = 0; i < memmap->nr_map; i ++) {
        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);
        if (memmap->map[i].type == E820_ARM) {
            if (maxpa < end && begin < KMEMSIZE) {
                maxpa = end;
            }
        }
    }
    if (maxpa > KMEMSIZE) {
        maxpa = KMEMSIZE;
    }

    extern char end[];

    npage = maxpa / PGSIZE;//需要管理的物理页个数
    pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);//把end按页大小为边界取整，作为page内存空间

    for (i = 0; i < npage; i ++) {
        SetPageReserved(pages + i);
    }
    //空闲空间起始位置，之前的空间设定为已占用物理内存空间
    uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) * npage);

    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) {
            if (begin < freemem) {
                begin = freemem;
            }
            if (end > KMEMSIZE) {
                end = KMEMSIZE;
            }
            if (begin < end) {
                begin = ROUNDUP(begin, PGSIZE);
                end = ROUNDDOWN(end, PGSIZE);
                if (begin < end) {
                    //根据探测到的空间实现空闲标记
                    init_memmap(pa2page(begin), (end - begin) / PGSIZE);
                }
            }
        }
    }
}
```

首先根据 bootloader 给出的内存布局信息找出最大的物理内存地址 maxpa（定义在 page_init 函数中的局部变量），由于 x86 的起始物理内存地址为 0，所以可以得知**需要管理的物理页个数**为：npage = maxpa / PGSIZE；然后预估出管理页级物理内存空间所需的 **Page 结构的内存空间所需的内存大小**为：sizeof (struct Page) * npage；由于 bootloader 加载

ucore 的结束地址（用全局指针变量 end 记录）以上的空间没有被使用，所以把 end 按页大小为边界取整后，作为管理页级物理内存空间所需的 **Page 结构的内存空间**，记为：
pages = (struct Page*) ROUNDUP ((void *)end, PGSIZE)；为了简化起见，从地址 0 到地址 pages + sizeof(struct Page) * npage)结束的物理内存空间设定 为已占用物理内存空间（起始 0~640KB 的空间是空闲的），地址 pages+ sizeof(struct Page) * npage)以上的空间为空闲物理内存空间，这时的空闲空间起始地址为 freemem。然后对于所有物理空间使用 **SetPageReserved 实现占用标记**，只需把物理地址对应的 Page 结构中的 flags 标志设置为 PG_reserve，表示这些页已经被使用了，将来不能被用于分配；对于探测到的空闲物理空间，使用 **init_memmap()实现空闲标记**，即把空闲物理页对应的 Page 结构中的 flags 和引用计数 ref 清零，并加到 free_area.free_list 指向的双向列表中，为将来的空闲页管理做好初始化准备工作。

当完成物理内存页管理初始化工作后，计算机系统的内存布局如下图所示：

3）物理内存页分配算法实现

first-fit 内存分配算法

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。

关键数据结构和变量

使用双向链表（libs/list.h 中定义）维护一个查找有序（地址从小到大）的空闲块
前面提到过的 **free_area_t 数据结构**用来实现对空闲块的管理
default_pmm.c 中定义了该类型的变量 free_area

```
 *               (5.3) try to merge low addr or high addr blocks. Notice: should change some
 */
free_area_t free_area;
```

pmm_manager 数据结构是一个通用的分配算法的函数指针列表，建立了一个物理内存页管理器框架，封装了内存页管理器名字、初始化函数、分配释放内存页函数，以及返回当前剩余空闲页数、检测分配释放是否正确的辅助函数

```
// 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;                                 // XXX_pmm_manager's name
    void (*init)(void);                               // initialize internal description&management data st
                                                      // (free block list, number of free block) of XXX_pmm_
    void (*init_memmap)(struct Page *base, size_t n); // setup description&management data structcure accord
                                                      // the initial free physical memory space
    struct Page *(*alloc_pages)(size_t n);            // allocate >=n pages, depend on the allocation algori
    void (*free_pages)(struct Page *base, size_t n);  // free >=n pages with "base" addr of Page descriptor
    size_t (*nr_free_pages)(void);                    // return the number of free pages
    void (*check)(void);                              // check the correctness of XXX_pmm_manager
};
```

在 default_pmm.c 中定义的 pmm_manager 类型结构 **default_pmm_manager：**

```
const struct pmm_manager default_pmm_manager = {
    .name = "default_pmm_manager",
    .init = default_init,
    .init_memmap = default_init_memmap,
    .alloc_pages = default_alloc_pages,
    .free_pages = default_free_pages,
    .nr_free_pages = default_nr_free_pages,
    .check = default_check,
};
```

.init 直接重用 default_init 函数实现

.init_memmap 根据现有内存情况构建空闲块列表初始状态，从总控函数调用过程如下：

kern_init()中调用 pmm_init()

```
int
kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);

    cons_init();                // init the console

    const char *message = "(THU.CST) os is loading ...";
    cprintf("%s\n\n", message);

    print_kerninfo();

    grade_backtrace();

    pmm_init();                 // init physical memory management
```

pmm_init()调用 page_init()

```
//pmm_init - setup a pmm to manage physical memory, build PDT&PT to setup paging mechanism
//         - check the correctness of pmm & paging mechanism, print PDT&PT
void
pmm_init(void) {
    //We need to alloc/free the physical memory (granularity is 4KB or other size).
    //So a framework of physical memory manager (struct pmm_manager)is defined in pmm.h
    //First we should init a physical memory manager(pmm) based on the framework.
    //Then pmm can alloc/free the physical memory.
    //Now the first_fit/best_fit/worst_fit/buddy_system pmm are available.
    init_pmm_manager();

    // detect physical memory space, reserve already used memory,
    // then use pmm->init_memmap to create free page list
    page_init();
```

page_init()调用 init_memmap()

```
233                 if (begin < end) {
234                     init_memmap(pa2page(begin), (end - begin) / PGSIZE);
235                 }
```

init_memmap()调用 pmm_manager 中同名的成员函数 init_memmap()

```
144 //init_memmap - call pmm->init_memmap to build Page struct for free memory
145 static void
146 init_memmap(struct Page *base, size_t n) {
147     pmm_manager->init_memmap(base, n);
148 }
```

成员函数 init_memmap()

```
17     void (*init_memmap)(struct Page *base, size_t n); // setup description&management data structcure according to
```

fist fit 算法中对应 **default_init_memmap()函数**，需要实现！！

```
266     .init_memmap = default_init_memmap,
```

[TODO1] 设计实现

需要用到的一些宏定义和函数在 **memlayout.h** 中定义实现：

```
110 #define SetPageReserved(page)       set_bit(PG_reserved, &((page)->flags))
111 #define ClearPageReserved(page)     clear_bit(PG_reserved, &((page)->flags))
112 #define PageReserved(page)          test_bit(PG_reserved, &((page)->flags))
113 #define SetPageProperty(page)       set_bit(PG_property, &((page)->flags))
114 #define ClearPageProperty(page)     clear_bit(PG_property, &((page)->flags))
115 #define PageProperty(page)          test_bit(PG_property, &((page)->flags))
```

```
117 // convert list entry to page
118 #define le2page(le, member)                 \
119     to_struct((le), struct Page, member)
```

- **default_init_memmap()函数**

功能：根据每个物理页帧情况**建立空闲页链表**，且空闲页块根据地址高低形成一个有序链表

步骤：初始化空闲页链表，分为两步：初始化每一个空闲页；计算空闲页总数

其中使用头插法是因为地址是从低地址向高地址增长的

```
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 ++) {//循环遍历每一页
73         assert(PageReserved(p));//检查此页是否为保留页
74         p->flags = p->property = 0;//标志位和连续内存空闲块大小置零
75         SetPageProperty(p);//将p中flags的property位(第1位)置1
76         set_page_ref(p, 0);//引用计数清零
77         list_add_before(&free_list, &(p->page_link));//使用头插法将空闲页插入链表
78     }
79     base->property = n;
80     nr_free += n;//更新空闲页总数
81 }
```

- **default_alloc_pages()函数**

功能：用于为进程分配空闲页

步骤如下：

① 遍历链表寻找足够大的空闲块

② 找到后重新设置标志位，从链表中删除此页

③ 判断该空闲块大小是否合适，如果不合适，分割页块

④ 计算剩余空闲页个数

⑤ 返回分配的页块地址

```
83 static struct Page *
84 default_alloc_pages(size_t n) {
85     assert(n > 0);
86     if (n > nr_free) {
87         return NULL;
88     }
89     struct Page *page = NULL;
90     list_entry_t *le = &free_list;
91     while ((le = list_next(le)) != &free_list) {//在空闲块链表中寻找大小合适的页块
92         struct Page *p = le2page(le, page_link);
93         if (p->property >= n) {//找到大小合适的块
94             page = p;
95             int i;
96             struct Page *pp;
97             list_entry_t *temp_le;
98             temp_le=le;
99             for(i=0;i!=n;i++){
100                pp=le2page(temp_le,page_link);
101                //设置每一页的标志位
102                SetPageReserved(pp);
103                ClearPageProperty(pp);
104                //清除free_list中此页
105                list_del(temp_le);
106                temp_le=list_next(temp_le);
107            }
108            break;
109        }
110    }
111    if (page != NULL) {
112//          list_del(&(page->page_link));
113        if(page->property > n) {//如果找到的页块大小大于n，分割大页块
114            struct Page *p = page + n;
115            p->property = page->property - n;
116//              list_add(&free_list, &(p->page_link));
117        }
118        nr_free -= n;
119//          return page;
120    }
121//    return NULL;
122    return page;
123
124 }
```

- **default_free_pages()函数**

功能：default_alloc_pages 逆过程，释放已经使用完的页，合并到 free_list 中

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

```
156 static void
157 default_free_pages(struct Page *base, size_t n) {
158     assert(n > 0);
159     //查找该插入位置le
160     list_entry_t *le = &free_list;
161     struct Page *p;
162     while ((le = list_next(le))!= &free_list) {
163         p = le2page(le, page_link);
164         if(p>base)
165             break;
166     }
167     //向le之前插入n个页，并设置标志位
168     for(p=base;p!=base+n;p++){
169         list_add_before(le,&(p->page_link));
170         p->flags=0;
171         set_page_ref(p,0);
172         ClearPageProperty(p);
173         SetPageProperty(p);
174     }
175     //将页块信息记录在头部
176     base->property=n;
177
```

```
178     //合并
179     //向高地址合并
180     p = le2page(le,page_link) ;
181     if( base+n == p ){
182       base->property += p->property;
183       p->property = 0;
184     }
185     //向低地址合并
186     le = list_prev(&(base->page_link));
187     p = le2page(le, page_link);
188     if(le!=&free_list && p==base-1){//如果低地址已经分配则不需要合并
189       while(le!=&free_list){
190         if(p->property){
191           p->property += base->property;
192           base->property = 0;
193           break;
194         }
195         le = list_prev(le);
196         p = le2page(le,page_link);
197       }
198     }
199     nr_free += n;
200 //    list_add(&free_list, &(base->page_link));
201 }
```

4）实现分页机制

段页式管理基本概念

内存地址分为三类：**逻辑地址**、线性地址、**物理地址**

在 ucore 中段式管理只起到了一个过渡作用，它将逻辑地址不加转换直接映射成线性地址，所以在下面的讨论中对这两个地址不加区分

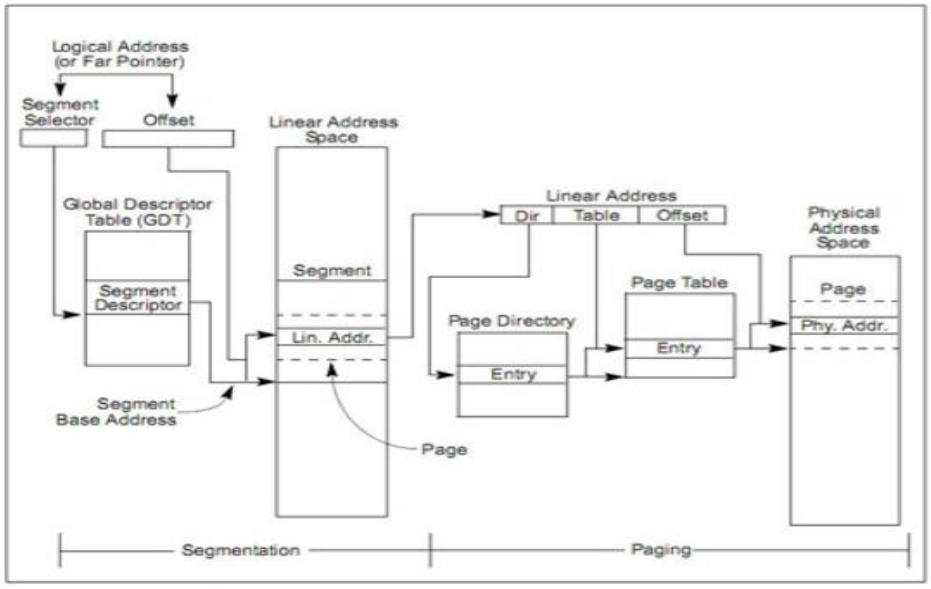


图 4 段页式管理总体框架图

系统执行中地址映射的四个阶段

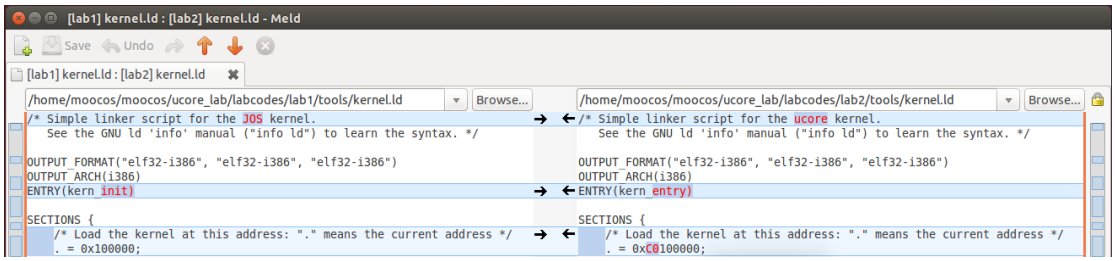
lab1：简单的段映射（对等映射关系），通过建立全局段描述符表，让每一个段的基址为 0，保证物理地址和虚拟地址相等

lab2：最终段页式映射关系：

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

具体实现：

tools/kernel.ld 链接脚本文件



lab1

通过 ld 工具形成的 ucore 的起始虚拟地址从 0x100000 开始，由于段地址映射为对等关系，所以 ucore 物理地址也是 0x100000，入口函数 kern_init 的起始地址

在 lab1 中虚拟地址，线性地址以及 物理地址之间的映射关系如下：

```
lab1： virt addr = linear addr = phy addr
```

lab2：

通过 ld 工具形成的 ucore 的起始虚拟地址从 0xC0100000 开始

入口函数为 kern_entry()函数

在 1 和 2 中 bootloader 把 ucore 都放在了起始物理地址为 0x1000000 的物理内存空间，说明两者是地址映射不同，**2 在不同阶段有不同的映射关系：**

第一阶段：从 bootloader 的 start()函数到执行 kern_entry()函数之前，三个地址一样

```
lab2 stage 1： virt addr = linear addr = phy addr
```

第二阶段：从 kern_entry()函数，到 enable_page()函数之前

（之前分析过，从总控函数 kern_init→pmm_init()→enable_paging()）

更新了段映射，还没有启动页映射

由于 gcc 编译出的虚拟起始地址从 0xC0100000 开始，ucore 被 bootloader 放置在从物理地址 0x100000 处开始的物理内存中。所以当 kern_entry 函数完成新的段映射关系后，且 ucore 在没有建立好页映射机制前，CPU 按照 ucore 中的虚拟地址执行，能够被分段机制映射到正确的物理地址上，确保 ucore 运行正确。

这时的虚拟地址，线性地址以及物理地址之间的映射关系为：

```
lab2 stage 2： virt addr - 0xC0000000 = linear addr = phy addr
```

此时 CPU 在寻址时还是只采用了分段机制，最后后并使能分页映射机制（lab2/kern/mm/pmm.c 中的 enable_paging 函数），一旦执行完 enable_paging 函数中的加载 cr0 指令（即让 CPU 使能分页机制），则接下来的访问是基于段页式的映射关系了。

第三阶段：从 enable_page()函数开始，到执行 gdt_init()函数之前

```
242 static void
243 enable_paging(void) {
244     lcr3(boot_cr3);
245
246     // turn on paging
247     uint32_t cr0 = rcr0();
248     cr0 |= CR0_PE | CR0_PG | CR0_AM | CR0_WP | CR0_NE | CR0_TS | CR0_EM | CR0_MP;
249     cr0 &= ~(CR0_TS | CR0_EM);
250     lcr0(cr0);
251 }
```

启动了页映射机制，没有第三次更新段映射

```
 lab2 stage 3:  virt addr - 0xC0000000 = linear addr  = phy addr + 0xC0000000 # 线性地址在0~4MB之外的三者映射关系
                virt addr - 0xC0000000 = linear addr  = phy addr # 线性地址在0~4MB之内的三者映射关系
```

```
320     //temporary map:
321     //virtual_addr 3G~3G+4M = linear_addr 0~4M = linear_addr 3G~3G+4M = phy_addr 0~4M
322     boot_pgdir[0] = boot_pgdir[PDX(KERNBASE)];
```

用来建立物理地址在 0~4MB 之内的三个地址间的临时映射关系

第四阶段：从 gdt_init()函数开始

第三次更新段映射，形成新的段页式映射机制，取消了临时映射关系，即 bootpgdir[0]=0，把 boot_pgdir[0]的第一个页目录表项（0-4MB）清零来取消临时的页映射关系，形成期望的映射关系：

```
lab2 stage 4：virt addr = linear addr = phy addr + 0xC0000000
```

建立虚拟页和物理页帧的地址映射关系

建立二级页表

ucore 的页式管理通过一个二级的页表实现。一级页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。目前，ucore 用 **boot_cr3**（mm/pmm.c）记录这个值。

页目录表（PDT） 页目录项（PDE）页表（PT） 页表项（PTR）

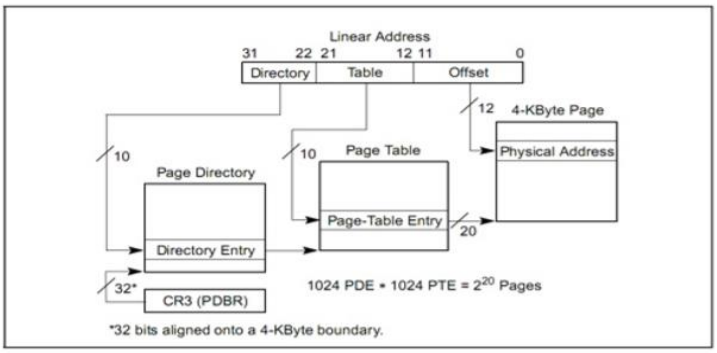


图 5 分页机制管理

ucore 设定实际物理内存不超过 KERNSIZE 值（0x38,000,000 字节），把 0~KERNSIZE 的物理地址一一映射到页目录项和页表项的内容，过程：

- **alloc_page()**获得一个空闲物理页，用于页目录表

这里实际调用的应该是 default_pmm.c 里面的 default_alloc_pages()

```
150 //alloc_pages - call pmm->alloc_pages to allocate a continuous n*PAGESIZE memory
151 struct Page *
152 alloc_pages(size_t n) {
153     struct Page *page=NULL;
154     bool intr_flag;
155     local_intr_save(intr_flag);
156     {
157         page = pmm_manager->alloc_pages(n);
158     }
159     local_intr_restore(intr_flag);
160     return page;
161 }
```

- **boot_map_segment()函数建立一一映射关系**

```
252 //boot_map_segment - setup&enable the paging mechanism
253 // parameters
254 //  la:   linear address of this memory need to map (after x86 segment map)
255 //  size: memory size
256 //  pa:   physical address of this memory
257 //  perm: permission of this memory
258 static void
259 boot_map_segment(pde_t *pgdir, uintptr_t la, size_t size, uintptr_t pa, uint32_t perm) {
260     assert(PGOFF(la) == PGOFF(pa));
261     size_t n = ROUNDUP(size + PGOFF(la), PGSIZE) / PGSIZE;
262     la = ROUNDDOWN(la, PGSIZE);
263     pa = ROUNDDOWN(pa, PGSIZE);
264     for (; n > 0; n --, la += PGSIZE, pa += PGSIZE) {
265         pte_t *ptep = get_pte(pgdir, la, 1);
266         assert(ptep != NULL);
267         *ptep = pa | PTE_P | perm;
268     }
269 }
```

- 宏定义都在 **mmu.h** 中

Getting Started | default_pmm.c | pmm.c | pmm.h | memlayout.h | defs.h | list.h | init.c | mmu.h

```
188 #endif /* !__ASSEMBLER__ */
189
190 // A linear address 'la' has a three-part structure as follows:
191 //
192 // +--------10------+-------10-------+---------12----------+
193 // | Page Directory |   Page Table   | Offset within Page  |
194 // |      Index     |     Index      |                     |
195 // +----------------+----------------+---------------------+
196 //  \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
197 //  \---------- PPN(la) -----------/
198 //
199 // The PDX, PTX, PGOFF, and PPN macros decompose linear addresses as shown.
200 // To construct a linear address la from PDX(la), PTX(la), and PGOFF(la),
201 // use PGADDR(PDX(la), PTX(la), PGOFF(la)).
202
203 // page directory index
204 #define PDX(la) ((((uintptr_t)(la)) >> PDXSHIFT) & 0x3FF)
205
206 // page table index
207 #define PTX(la) ((((uintptr_t)(la)) >> PTXSHIFT) & 0x3FF)
208
209 // page number field of address
210 #define PPN(la) (((uintptr_t)(la)) >> PTXSHIFT)
211
212 // offset in page
213 #define PGOFF(la) (((uintptr_t)(la)) & 0xFFF)
214
215 // construct linear address from indexes and offset
216 #define PGADDR(d, t, o) ((uintptr_t)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
```

PTE_U：位 3，表示用户态的软件可以读取对应地址的物理内存页内容
PTE_W：位 2，表示物理内存页内容可写
PTE_P：位 1，表示物理内存页存在

ROUNDUP 和 **ROUNDDOWN**（向下取整找到最接近的 2^n 的倍数）

```
39 /* *
40  * Rounding operations (efficient when n is a power of 2)
41  * Round down to the nearest multiple of n
42  * */
43 #define ROUNDDOWN(a, n) ({                                          \
44             size_t __a = (size_t)(a);                               \
45             (typeof(a))(__a - __a % (n));                           \
46         })
47
48 /* Round up to the nearest multiple of n */
49 #define ROUNDUP(a, n) ({                                            \
50             size_t __n = (size_t)(n);                               \
51             (typeof(a))(ROUNDDOWN((size_t)(a) + __n - 1, __n));     \
52         })
```

get_pte()函数用于查找页表，如果获取不到就新建一个页表
建立好二级页表结构之后就可以使能分页机制 **enable_paging()函数**

【TODO2】实现寻找虚拟地址对应的页表的函数 get_pte()

步骤如下：

① 尝试获取页表

② 如果获取不成功需要申请一页，引用次数+1

③ 获取页的线性地址，设置页目录入口的权限

④ 最后返回页表的地址

```
341 //get_pte - get pte and return the kernel virtual address of this pte for la
342 //        - if the PT contians this pte didn't exist, alloc a page for PT
343 // parameter:
344 //  pgdir:  the kernel virtual base address of PDT
345 //  la:     the linear address need to map
346 //  create: a logical value to decide if alloc a page for PT
347 // return vaule: the kernel virtual address of this pte
```

```
348 pte_t *
349 get_pte(pde_t *pgdir, uintptr_t la, bool create) {
350     /* LAB2 EXERCISE 2: YOUR CODE
351      *
352      * If you need to visit a physical address, please use KADDR()
353      * please read pmm.h for useful macros
354      *
355      * Maybe you want help comment, BELOW comments can help you finish the code
356      *
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      */
```

```
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
```

```
371 //#if 0
372     pde_t *pdep = &pgdir[PDX(la)];   // (1) find page directory entry
373     if (!(*pdep&PTE_P)) {            // (2) check if entry is not present
374         struct Page *page;           // (3) check if creating is needed, then alloc page for page t
375         // CAUTION: this page is used for page table, not for common data page
376         if(!create||page==alloc_page()==NULL){
377             return NULL;
378         }
379         set_page_ref(page,1)         // (4) set page reference，引用次数+1
380         uintptr_t pa = page2pa(page);// (5) get linear address of page
381         memset(KADDR(pa),0,PGSIZE);  // (6) clear page content using memset
382         *pdep=pa|PTE_U|PTE_W|PTE_P;  // (7) set page directory entry's permission
383     }
384     return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];// (8) return page table entry
385 //#endif
386 }
```

【TODO3】释放某虚拟地址所在页，并取消对应二级页表表项的映射 page_remove_pte()

步骤如下：

① 判断页表入口是否存在

② 判断此页被引用的次数，如果仅被引用一次，则这个页可以被释放；否则不能释放此页，只能释放页表入口 pte

```
401 //page_remove_pte - free an Page sturct which is related linear address la
402 //                - and clean(invalidate) pte which is related linear address la
403 //note: PT is changed, so the TLB need to be invalidate
404 static inline void
405 page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
406     /* LAB2 EXERCISE 3: YOUR CODE
407      *
408      * Please check if ptep is valid, and tlb must be manually updated if mapping is updated
409      *
410      * Maybe you want help comment, BELOW comments can help you finish the code
411      *
412      * Some Useful MACROs and DEFINEs, you can use them in below implementation.
413      * MACROs or Functions:
414      *   struct Page *page pte2page(*ptep): get the according page from the value of a ptep
415      *   free_page : free a page
416      *   page_ref_dec(page) : decrease page->ref. NOTICE: ff page->ref == 0 , then this page should be free.
417      *   tlb_invalidate(pde_t *pgdir, uintptr_t la) : Invalidate a TLB entry, but only if the page tables being
418      *                        edited are the ones currently in use by the processor.
419      * DEFINEs:
420      *   PTE_P           0x001                   // page table/directory entry flags bit : Present
421      */
```

```
422 //#if 0
423     if (*ptep&PTE_P) {                      //(1) check if this page table entry is present
424         struct Page *page = pte2page(*ptep); //(2) find corresponding page to pte
425         if(page_ref_dec(page)==0){                          //(3) decrease page reference
426             free_page(page);                          //(4) and free this page when page reference
427         }
428         *ptep=0;                            //(5) clear second page table entry
429         tlb_invalidate(pgdir,la);                           //(6) flush tlb
430     }
431 //#endif
432 }
```

make qemu

```
[~/moocos/ucore_lab/labcodes/lab2]
moocos-> make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc0105ee2 (phys)
  edata  0xc0117a36 (phys)
  end    0xc0118968 (phys)
Kernel executable memory footprint: 99KB
ebp:0xc0116f38 eip:0xc01009d0 args:0x00010094 0x00000000 0xc0116f68 0xc01000bc
    kern/debug/kdebug.c:308: print_stackframe+21
ebp:0xc0116f48 eip:0xc0100cbf 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:0xc0105f1c 0xc0105f00 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
100 ticks
100 ticks
100 ticks
100 ticks
qemu: terminating on signal 2
```

make grade

```
[~/moocos/ucore_lab/labcodes/lab2]
moocos-> make grade
Check PMM:                (2.4s)
  -check pmm:                                 OK
  -check page table:                          OK
  -check ticks:                               OK
Total Score: 50/50
```

Q：写出 ucore 启动流程

- **结合代码，描述 ucore 是如何从磁盘加载到内存的，以及之后内存是如何被管理起来的**
- **ucore 启动过程中，段页机制的改变**
- **描述最终 pm_init 执行完成后，内核的地址映射状态**

A：从磁盘加载到内存过程见 lab1：
计算机启动后，CPU 从存放了系统初始化软件的地址开始执行指令。系统初始化软件是在 OS 内核运行之前运行的一小段代码，负责完成计算机基本的 IO 初始化和引导加载功能。80386 体系结构中系统初始化软件由 BIOS（固化在主板上的基本 IO 系统）和位于软盘/硬盘引导扇区中的 OS bootloader 组成。BIOS 做完硬件自检和初始化后，会选择一个启动设备读取该设备的第一扇区到内存一个特定地址，CPU 控制权会转移到该地址继续执行，然后把工作交给 ucore 的 bootloader 执行
bootlader 完成的工作包括：

- 切换到保护模式，启用分段机制
- 读磁盘中 ELF 执行文件格式的 ucore 操作系统到内存
- 显示字符串信息
- 把控制权交给 ucore 操作系统

启动后内存管理机制和段页机制的实现见前面**“4）实现分页机制”**中结合代码的分析
最终的内存地址映射状态见下图：

[Challenge] buddy system

学习“伙伴分配器的一个极简实现”中的 cloudwu 的代码:

分配器的**整体思想**是，通过一个数组形式的完全二叉树来监控管理内存，二叉树的节点用于标记相应内存块的使用状态，高层节点对应大的块，低层节点对应小的块，在分配和释放中就通过这些节点的标记属性来进行块的分离合并

数据结构：

```
struct buddy2 {
  unsigned size;
  unsigned longest[1];
};
```

成员 size 是内存总单元数目（每个节点都一样）；成员 longest 对应某一个节点，方括号内索引是节点的标号，Longest 值为该节点对应内存块空闲大小

下面结合 ucore 中数据结构 Page、free_list 等实现 buddy system，由于时间原因，只参考讲义算法简单实现了初始化、分配和释放函数，没有写检验函数，也还没有调试

初始化形成二叉树

unfinished_buddy_system.c

```
#include <pmm.h>
#include <list.h>

free_area_t free_area;
#define free_list (free_area.free_list)
#define nr_free (free_area.nr_free)

#define is_pow_of_2(x) (!(x&(x-1)))
#define max(a,b) (a>=b)?a:b

struct buddy{
    size_t size;
    size_t *longest;
};

```

```c
static struct buddy* buddy_init(struct Page *base, size_t size){
    assert(size>0);
    assert(is_pow_of_2(size));

    //初始化链表和pages
    list_init(free_list);
    nr_free=size;
    struct Page *p;
    for(p=base;p!=base+size;p++){
        assert(PageReserved(p));
        p->flags=p->property=0;
    }

    struct buddy *root;
    root=(struct buddy*)base;
    root->size=size;

    int index;
    size_t node_size;
    node_size=size*2;
    size_t offset=0;
    for(index=0;index!=2*size-1;index++){
        if(is_pow_of_2(index+1))
            node_size/=2;
        root->longest[index]=node_size;
        //分配size页空间并加入到free_list中
        offset=(index+1)*node_size-root->size;
        struct Page *page=base+offset;
        page->property=node_size;
        set_page_ref(page,0);
        SetPageProperty(page);
        list_add(&(free_list),&(page->page_link));
    }
    return root;
}
```

分配时，首先将请求的页数向上取整为 2 的幂，然后从根节点进行深度优先搜索，搜索大小合适的块，标记该节点占用，并回溯将祖先节点取左右子树较大值，表明其最大空闲值

```c
static struct Page* buddy_alloc(struct buddy* root, size_t size){
        assert(root&&size>0);
        struct Page *page;
        size_t node_size;
        size_t index=0;

        if(!is_pow_of_2(size))
            size=roundup(size);
        if(size>root->longest[index])
            return NULL;

        //深度优先遍历，找到大小合适的节点
        for(node_size=root->size;size<node_size;node_size/=2){
            if(size<=root->longest[2*index+1])
                index=2*index+1;
            else
                index=2*index+2;
        }
        //分配空间
        root->longest[index]=0;
        page=&(buddy_alloc[buddy_begin(root)]);
        list_del(&page->page_link);
        nr_free-=size;
        //回溯更新祖先节点
        while(index){
            index=(index+1)/2-1;
            root->longest[index]=max(root->longest[2*index+1],root->longest[2*index+2]);
        }
        return page;
}
```

释放时，自底向上地将空间标记为可用，并回溯检查是否存在合并的块，即将左右子树的 longest 值相加是否等于原空闲块满状态大小，如果能够合并，将父节点 longest 值改为和

```
83 static struct Page* buddy_free(struct Page *base,struct buddy* root, size_t offset){
84         assert(root&&offset>0);
85         size_t node_size=1;
86         size_t index=0;
87         //找到分配块，恢复longest值
88         index=offset+root->size-1;
89         for(;root->longest[index];index=(index+1)/2-1){
90             node_size*=2;
91             if(index==0)
92                 return;
93         }
94         root->longest[index]=node_size;
95         //释放空间
96         struct Page *page=base+offset;
97         page->property=node_size;
98         struct Page *p=page;
99         for(;p!=base+offset+node_size;p++){
100            assert(!PageReserved(p));
101            p->flags=0;
102            set_page_ref(p,0);
103        }
104        list_add(&(free_list),&(page->page_link));
105        nr_free+=node_size;
106        //向上回溯，检查是否存在合并的块
107        while(index){
108            index=(index+1)/2-1;
109            node_size*=2;
110            left_longest=root->longest[2*index+1];
111            right_longest=root->longest[2*index+2];
112            if(left_longest+right_longest==node_size)
113                root->longest[index]=node_size;
114            else
115                root->longest[index]=max(left_longest,right_longest);
116        }
117 }
```

三、 实验感悟

本次实验加深了对物理内存管理的了解，结合 ucore 框架实现了 first-fit 连续内存分配算法；对 x86 体系结构段页式内存管理机制有了更深入的理解，尤其是系统执行中地址映射的四个阶段，虚地址到物理地址的映射发生了多次变化；伙伴系统算法思路比较简单，但是实现起来需要仔细结合 ucore 的数据结构，有时间继续尝试完成!