ucore Lab2

21307265 童齐嘉

实验要求

为了实现lab2的目标，lab2提供了3个基本练习和2个扩展练习，要求完成实验报告。

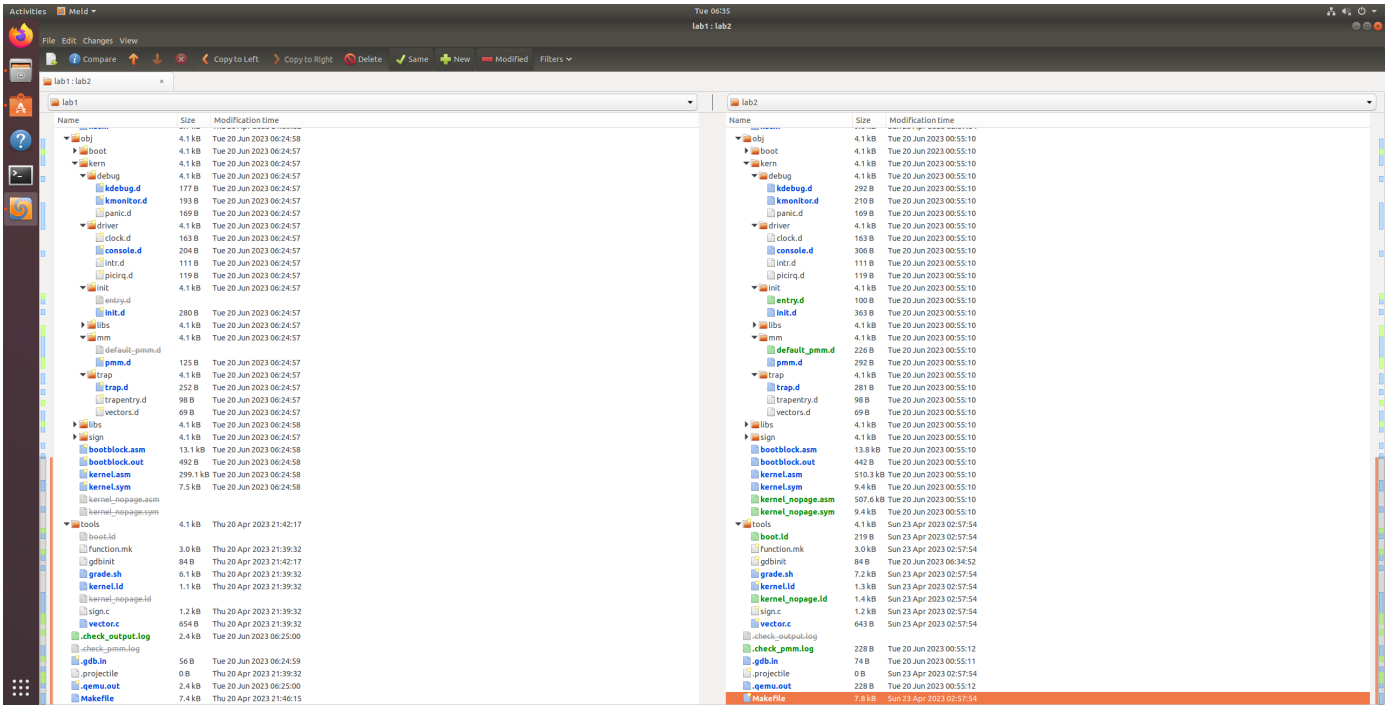
对实验报告的要求：

- 基于markdown格式来完成，以文本方式为主。
- 填写各个基本练习中要求完成的报告内容
- 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
- 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）
- 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

实验过程：

练习0：对比补全文件代码

我们使用了Meld软件进行了替换



练习1：实现 first-fit 连续物理内存分配算法

实现过程

由于要求**将空闲内存块按照地址从小到大的方式连起来**，所以在实现first fit 内存分配算法的回收函数时，要考虑地址连续的空闲块之间的合并操作。提示:在建立空闲页块链表时，需要按照空闲页块起始地址来排序，形成一个有序的链表。可能会修改default_pmm.c中的default_init，default_init_memmap，default_alloc_pages，default_free_pages等相关函数。请仔细查看和理解default_pmm.c中的注释。

我们根据“以页为单位管理物理内存”一节中的提示，找到了如下定义：

```
struct Page {
    int ref;        // 这页被页表的引用记数
    uint32_t flags; // 状态，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。
    unsigned int property;// 地址连续的空闲页的个数
    list_entry_t page_link;// 链接多个连续内存空闲块的双向链表指针
};
struct list_entry {
    struct list_entry *prev, *next;
};
```

我们根据提示，来到lab2/kernel/mm/default_pmm.c中：

```
// LAB2 EXERCISE 1: YOUR CODE
// you should rewrite functions: default_init,default_init_memmap,default_alloc_pages,
default_free_pages.
```

按照注释以及所给提示，我们首先修改**default_init_memmap**：

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        SetPageProperty(p);
        set_page_ref(p, 0);
        list_add_before(&free_list, &(p->page_link));
    }
    base->property = n;
    nr_free += n;
}
```

其缺陷主要为：

1. 遍历中间的空闲页没有被加入到空闲页表中
2. 中间页的 flags 处未被设为 PG_property

其次，修改**default_alloc_pages**：

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    list_entry_t *lenext;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL){
        int temp;
        //pick and unlink all pages selected
```

```
        struct Page *tempPage;
        for (temp = 0; temp < n; temp++){
            lenext = list_next(le);
            tempPage = le2page(le, page_link);
            SetPageReserved(tempPage);
            ClearPageProperty(tempPage);
            list_del(le);
            le = lenext;
        }
        if (page->property>n)
            (le2page(le,page_link))->property = page->property - n;
        ClearPageProperty(page);
        SetPageReserved(page);
        nr_free -= n;
    }
    return page;
}
```

其缺陷主要为：

1. 按顺序查找后，只取出了第一页
2. 除第一页外，其他页并未被设置为保留页

最后，我们修改**default_free_pages**:

```
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;
      }
    }
    for(p=base;p<base+n;p++){
      list_add_before(le, &(p->page_link));
    }
    base->flags = 0;
    set_page_ref(base, 0);
    ClearPageProperty(base);
    SetPageProperty(base);
    base->property = n;
    //如果是高位，则向高地址合并
    p = le2page(le,page_link) ;
    if( base+n == p ){
      base->property += p->property;
      p->property = 0;
    }
     //如果是低位且在范围内，则向低地址合并
    le = list_prev(&(base->page_link));
    p = le2page(le, page_link);
    if(le!=&free_list && p==base-1){ //未分配则合并
      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 ;
}
```

其缺陷主要为:

1. 插入链表是并没有找到 base 相对应的位置
2. 没有把页插入到空闲页表中
3. 合并过程需要在freelist中向高地址或低地址合并

实验截图

```
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: 07ee0000, [00100000, 07fdffff], type = 1.
  memory: 00020000, [07fe0000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
```

理论问答

请在实验报告中简要说明你的设计实现过程。请回答如下问题:

- 你的first fit算法是否有进一步的改进空间

答：空闲链表开头会产生许多小的空闲块，可以尝试优化，减少空间占用。

练习2：实现寻找虚拟地址对应的页表项

实现过程

通过设置页表和对应的页表项，可建立虚拟内存地址和物理内存地址的对应关系。其中的get_pte函数是设置页表项环节中的一个重要步骤。此函数找到一个虚地址对应的二级页表项的内核虚地址，如果此二级页表项不存在，则分配一个包含此项的二级页表。本练习需要补全get_pte函数 in kern/mm/pmm.c，实现其功能。请仔细查看和理解get_pte函数中的注释。get_pte函数的调用关系图如下所示:

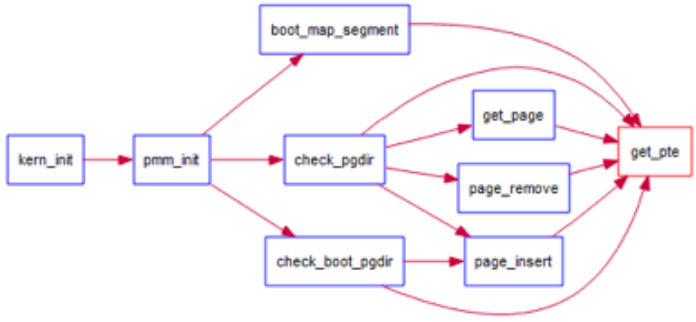


图1 get_pte函数的调用关系图

该部分练习主要是要完成 `get_pte` 函数

```
pte_t * get_pte(pde_t *pgdir, uintptr_t la, bool create) {
   // 获取传入的线性地址中所对应的页目录条目的物理地址
   pde_t *pdep = &pgdir[PDX(la)];
```

```
    // 如果该条目不可用(not present)
    if (!(*pdep & PTE_P)) {
        struct Page *page;
        // 如果分配页面失败，或者不允许分配，则返回NULL
        if (!create || (page = alloc_page()) == NULL)
            return NULL;
        // 设置该物理页面的引用次数为1
        set_page_ref(page, 1);
        // 获取当前物理页面所管理的物理地址
        uintptr_t pa = page2pa(page);
        // 清空该物理页面的数据。需要注意的是使用虚拟地址
        memset(KADDR(pa), 0, PGSIZE);
        // 将新分配的页面设置为当前缺失的页目录条目中
        // 之后该页面就是其中的一个二级页面
        *pdep = pa | PTE_U | PTE_W | PTE_P;
    }
    // 返回在pgdir中对应于la的二级页表项
    return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];
}
```

- 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

理论问答

- 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。

答:

PDE和PTE的大小都为4B，高20位用于保存索引，低12位用于保存属性

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)							Bits 39:32 of address[2]		PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

其中，前五位相同:

- bit 0§: Present，用来确认对应的页表是否存在。
- bit 1(R/W): read/write，若该位为0 ,则只读，否则可写。
- bit 2(U/S): user/supervisor，用来确认用户态下是否可以访问。
- bit 3(PWT): page-level write-through，表示是否使用write through缓存写策略。
- bit 4(PCD): page-level cache disable，表示是否不对该页进行缓存。
- bit 5(A): accessed，用来确认对应页表是否被访问过。

此外，对于PDE:

- bit 7(PS): Page size，这个位用来确定32位分页的页大小，当该位为1且CR4的PSE位为1时，页大小为4M，否则为4K。

而对于PTE:

- bit 6(D): 为脏位，判断是否有写入
- bit 7: 如果支持 PAT 分页，间接决定这项访问的页的内存类型，否则为0
- bit 8: Global 位。当 CR4.PGE 位为 1 时,该位为1则全局

- 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?

答:

- 将引发页访问异常的地址la将被保存在cr2寄存器中
- 设置错误代码
- 引发Page Fault，将外存的数据换到内存中
- 进行上下文切换，退出中断，返回到中断前的状态

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

实现过程

当释放一个包含某虚地址的物理内存页时，需要让对应此物理内存页的管理数据结构Page做相关的清除处理，使得此物理内存页成为空闲；另外还需把表示虚地址与物理地址对应关系的二级页表项清除。请仔细查看和理解page_remove_pte函数中的注释。为此，需要补全在 kern/mm/pmm.c中的page_remove_pte函数。page_remove_pte函数的调用关系图如下所示：

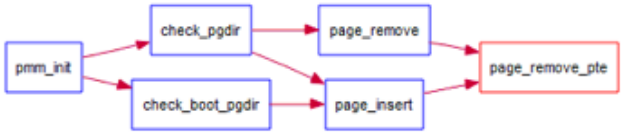


图2 page_remove_pte函数的调用关系图

思路：先判断该页被引用的次数，如果只被引用了一次，那么直接释放掉这页，否则就删掉二级页表的该表项，即该页的入口。

```
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    // 如果页表项存在
    if (*ptep & PTE_P) {
        // 获取该页表条目所对应的地址
        struct Page *page = pte2page(*ptep);
        // 如果该页的引用次数在减1后为0
        if (page_ref_dec(page) == 0)
            // 释放当前页
            free_page(page);
        // 清空PTE
        *ptep = 0;
        // 刷新TLB内的数据
        tlb_invalidate(pgdir, la);
    }
}
```

实验截图

```
Kernel executable memory footprint: 108KB
ebp:0xc0116f48 eip:0xc0100a72 args:0x00010094 0x00010094 0xc0116f78 0xc01000b5
    kern/debug/kdebug.c:308: print_stackframe+21
ebp:0xc0116f58 eip:0xc0100d6e args:0x00000000 0x00000000 0x00000000 0xc0116fc8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc0116f78 eip:0xc01000b5 args:0x00000000 0xc0116fa0 0xffff0000 0xc0116fa4
    kern/init/init.c:58: grade_backtrace2+19
ebp:0xc0116f98 eip:0xc01000d7 args:0x00000000 0xffff0000 0xc0116fc4 0x0000002a
    kern/init/init.c:63: grade_backtrace1+27
ebp:0xc0116fb8 eip:0xc01000f4 args:0x00000000 0xc0100036 0xffff0000 0xc0100079
    kern/init/init.c:68: grade_backtrace0+19
ebp:0xc0116fd8 eip:0xc0100115 args:0x00000000 0x00000000 0x00000000 0xc0105a20
    kern/init/init.c:73: grade_backtrace+26
ebp:0xc0116ff8 eip:0xc0100086 args:0xc0105c1c 0xc0105c24 0xc0100cf7 0xc0105c43
    kern/init/init.c:32: kern_init+79
```

理论问答

- 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？

答：当页目录项或页表项有效时，Page数组中的项与页目录项或页表项存在对应关系。

Page的每一项记录一个物理页的信息，而每个页目录项记录一个页表的信息，每个页表项则记录一个物理页的信息。

1. 可以通过 PTE 的地址计算其所在的页表的Page结构：

将虚拟地址向下对齐到页大小，换算成物理地址(减 KERNBASE), 再将其右移 PGSHIFT(12)位获得在pages数组中的索引PPN，&pages[PPN]就是所求的Page结构地址。

2. 可以通过 PTE 指向的物理地址计算出该物理页对应的Page结构：

PTE 按位与 0xFFF获得其指向页的物理地址，再右移 PGSHIFT(12)位获得在pages数组中的索引PPN，&pages[PPN]就 PTE 指向的地址对应的Page结构。

- 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**

相关背景知识：

系统执行中地址映射的四个阶段：

在lab1中，我们已经碰到到了简单的段映射，即对等映射关系，保证了物理地址和虚拟地址相等，也就是通过建立全局段描述符表，让每个段的基址为0，从而确定了对等映射关系。在lab2中，由于在段地址映射的基础上进一步引入了页地址映射，形成了组合式的段页式地址映射。从计算机加电，启动段式管理机制，启动段页式管理机制，在段页式管理机制下运行这整个过程中，虚地址到物理地址的映射产生了多次变化，实现了最终的段页式映射关系：

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

第一个阶段是bootloader阶段，这个阶段其虚拟地址，线性地址以及物理地址之间的映射关系与lab1的一样，即：

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

第二个阶段是从kern_\entry函数开始，到执行enable_page函数（在kern/mm/pmm.c中）之前再次更新了段映射，还没有启动页映射机制。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，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。

第三个阶段是从enable_page函数开始，到执行gdt_init函数（在kern/mm/pmm.c中）之前，启动了页映射机制，但没有第三次更新段映射。这是候映射关系是：

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

请注意pmm_init函数中的一条语句：

```
boot_pgdir[0] = boot_pgdir[PDX(KERNBASE)];
```

就是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

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

第四个阶段是从gdt_init函数开始，第三次更新了段映射，形成了新的段页式映射机制，并且取消了临时映射关系，即执行语句“boot_pgdir[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。这时形成了我们期望的虚拟地址，线性地址以及物理地址之间的映射关系： `lab2 stage 4： virt addr = linear addr = phy addr + 0xC0000000`

使能分页机制后的虚拟地址空间图：

```
/* *
 * Virtual memory map:                                          Permissions
 *                                                              kernel/user
 *
 *     4G ------------> +---------------------------------+
 *                      |                                 |
 *                      |         Empty Memory (*)        |
 *                      |                                 |
 *                      +---------------------------------+ 0xFB000000
 *                      |   Cur. Page Table (Kern, RW)    | RW/-- PTSIZE
 *     VPT ----------> +---------------------------------+ 0xFAC00000
 *                      |        Invalid Memory (*)       | --/--
 *     KERNTOP ------> +---------------------------------+ 0xF8000000
 *                      |                                 |
 *                      |    Remapped Physical Memory     | RW/-- KMEMSIZE
 *                      |                                 |
 *     KERNBASE -----> +---------------------------------+ 0xC0000000
 *                      |                                 |
 *                      |                                 |
 *                      |                                 |
 *                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * (*) Note: The kernel ensures that "Invalid Memory" is *never* mapped.
 *     "Empty Memory" is normally unmapped, but user programs may map pages
 *     there if desired.
 *
 * */
```

所以，我们需要：

1. 修改虚拟地址为0x100000 在tools/kernel.ld找到

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

对其进行修改

2. 保留临时映射 在kern/init/entry.S中,注释掉

```
# next:

    # unmap va 0 ~ 4M, it is temporary mapping
    # xorl %eax, %eax
    # 将__boot_pgdir的第一个页目录项清零，取消0~4M虚地址的映射
    # movl %eax, __boot_pgdir
```

3. 修改KERNBASE 在kern/mm/memlayout.h中修改KERNBASE

```
#define KERNBASE            0xc0000000 //改为0x00000000
```

并注释掉：

```
    # 因此需要REALLOC来对内核全局变量进行重定位，在开启分页模式前保证程序访问的物理地址的正确性

    # load pa of boot pgdir
    # 此时还没有开启页机制，__boot_pgdir(entry.S中的符号)需要通过REALLOC转换成正确的物理地址
    movl $REALLOC(__boot_pgdir), %eax
    # 设置eax的值到页表基址寄存器cr3中
    # movl %eax, %cr3
```

同时，到pmm_init中注释掉检查函数

```
    //use pmm->check to verify the correctness of the alloc/free function in a pmm
    //check_alloc_page();

    //check_pgdir();
```

4. 修改boot_map_segment函数

在boot_map_segment()中，先清除boot_pgdir[1]的 present 位，再进行其他操作。这是get_pte会分配一个物理页作为boot_pgdir[1]指向的页表。

```
static void
boot_map_segment(pde_t *pgdir, uintptr_t la, size_t size, uintptr_t pa, uint32_t perm) {
    boot_pgdir[1] &= ~PTE_P; // 添加
    assert(PGOFF(la) == PGOFF(pa));
    // 计算出一共有多少需要进行虚实映射的页面数
    size_t n = ROUNDUP(size + PGOFF(la), PGSIZE) / PGSIZE;
    // 按照物理页大小进行向下对齐
    la = ROUNDDOWN(la, PGSIZE);
    pa = ROUNDDOWN(pa, PGSIZE);
    // la线性地址，pa物理地址每次递增PGSIZE 在内核页表项中进行等位的映射
    for (; n > 0; n --, la += PGSIZE, pa += PGSIZE) {
        // 获取线性地址la，在pgdir页目录表下的二级页表项指针
        pte_t *ptep = get_pte(pgdir, la, 1);
        assert(ptep != NULL);
        // 为二级页表项赋值(共32位，pa中31~12位为对应的物理页框物理基地址，或PTE_P是设置第0位存在位为1，
或perm是对页表项进行权限属性的设置)
        *ptep = pa | PTE_P | perm;
    }
}
```

至此，我们实现了目标:

```
stack trackback:
ebp:0x00116f58 eip:0x00100a6b args:0x00118000 0x37fff000 0x00000001 0x00116f8c
    kern/debug/kdebug.c:308: print_stackframe+21
ebp:0x00116f78 eip:0x0010043e args:0x00106170 0x0000021c 0x0010615b 0x001064dc
    kern/debug/panic.c:27: __panic+107
ebp:0x00116fb8 eip:0x00103b4a args:0x00000000 0x0010002f 0x00118000 0x00118000
    kern/mm/pmm.c:540: check_boot_pgdir+286
ebp:0x00116fd8 eip:0x001031b5 args:0x00000000 0x00000000 0x00000000 0x00105960
    kern/mm/pmm.c:358: pmm_init+95
ebp:0x00116ff8 eip:0x00100084 args:0x00105b5c 0x00105b64 0x00100cf0 0x00105b83
    kern/init/init.c:35: kern_init+84
Welcome to the kernel debug monitor!!
```

运行截图

完成了基本的实验后，我们验证一下

```
summer@ubuntu:~/ulab2$ make grade
Check PMM:                (2.1s)
  -check pmm:                                 OK
  -check page table:                          OK
  -check ticks:                               OK
Total Score: 50/50
```

Challenge1&Challenge2

没有自己实现这一过程，只是去理解代码并参考了别人的实现过程

实验过程中遇到的问题

1. 练习0完成后，运行时有报错，疑似文件替换错误

解决方式： 尝试了很多方式，但运行过程中仍然会报出各种异常，这给我造成了很大的困难。

最终我们选择将答案文件中的todo部分删除重填以完成实验

2. 在尝试着解决练习3第二个问题后，我的文件没法通过 `make grade` 测试了

解决方式： 重新填一遍

参考资料

https://blog.csdn.net/sfadjlha/article/details/125089620

https://blog.csdn.net/CNRalap/article/details/124512925

https://github.com/Kiprey/Skr_Learning/tree/master/week9-19

https://blog.csdn.net/qq_19876131/article/details/51706978

https://blog.csdn.net/Aaron503/article/details/130189764