# 操作系统

# LAB2实验报告

班级：人工智能2103

学号：202107030125

姓名：姚丁钰

# 1. 知识点

## 1) 物理内存探测

- 操作系统需要知道了解整个计算机系统中的物理内存如何分布的，哪些被可用，哪些不可用。其基本方法是通过BIOS中断调用来帮助完成的。`bootasm.S`中新增了一段代码，使用BIOS中断检测物理内存总大小。
- 在讲解该部分代码前，先引入一个结构体

```
struct e820map {       // 该数据结构保存于物理地址0x8000
    int nr_map;        // map中的元素个数
    struct {
        uint64_t addr;     // 某块内存的起始地址
        uint64_t size;     // 某块内存的大小
        uint32_t type;     // 某块内存的属性。1标识可被使用内存块；2表示保留的内存块，不可映射。
    } __attribute__((packed)) map[E820MAX];
};
```

- 以下是bootasm.S中新增的代码，详细信息均以注释的形式写入代码中。

```
probe_memory:
    movl $0, 0x8000    # 初始化，向内存地址0x8000，即uCore结构e820map中的成员nr_map中写入0
    xorl %ebx, %ebx    # 初始化%ebx为0，这是int 0x15的其中一个参数
    movw $0x8004, %di # 初始化%di寄存器，使其指向结构e820map中的成员数组map
start_probe:
```

```
    movl $0xE820, %eax  # BIOS 0x15中断的子功能编号 %eax == 0xE820
    movl $20, %ecx      # 存放地址范围描述符的内存大小，至少20
    movl $SMAP, %edx    # 签名， %edx == 0x534D4150h("SMAP"字符串的ASCII码)
    int $0x15           # 调用0x15中断
    jnc cont            # 如果该中断执行失败，则CF标志位会置1，此时要通知UCore出错
    movw $12345, 0x8000 # 向结构e820map中的成员nr_map中写入特殊信息，报告当前错误
    jmp finish_probe    # 跳转至结束，不再探测内存
cont:
    addw $20, %di       # 如果中断执行正常，则目标写入地址就向后移动一个位置
    incl 0x8000         # e820::nr_map++
    cmpl $0, %ebx       # 执行中断后，返回的%ebx是原先的%ebx加一。如果%ebx为0，则说明当前内存探测完成
    jnz start_probe
finish_probe:
```

- 这部分代码执行完成后，从BIOS中获得的内存分布信息以结构体`e820map`的形式写入至物理`0x8000`地址处。稍后ucore的page_init函数会访问该地址并处理所有的内存信息。

## 2) 链接地址

- 审计`lab2/tools/kernel.ld`这个链接脚本，我们可以很容易的发现，链接器设置kernel的链接地址(link address)为`0xC0100000`，这是个虚拟地址。在uCore的bootloader中，bootloader使用如下语句来加载kernel：

```
readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
```

`0xC0010000 & 0xFFFFFF == 0x00100000`即kernel最终被装载入物理地址`0x10000`处，其相对偏移为`-0xc0000000`，与uCore中所设置的虚拟地址的偏移量相对应。

- 需要注意的是，在lab2的一些代码中会使用到如下两个变量，但这两个变量并没有被定义在任何C语言的源代码中：

```
extern char end[];
extern char edata[];
```

实际上，它们定义于`kernel.ld`这个链接脚本中

```
. = ALIGN(0x1000);
```

```
    .data.pgdir : {
        *(.data.pgdir)
    }

    PROVIDE(edata = .);

    .bss : {
        *(.bss)
    }

    PROVIDE(end = .);

    /DISCARD/ : {
        *(.eh_frame .note.GNU-stack)
    }
    // 脚本文件的结尾
```

`edata`表示`kernel`的`data`段结束地址；`end`表示`bss`段的结束地址（即整个`kernel`的结束地址）

`edata[]`和 `end[]`这些变量是ld根据kernel.ld链接脚本生成的全局变量，表示相应段的结束地址，它们不在任何一个.S、.c或.h文件中定义，但仍然可以在源码文件中使用。

## 3) uCore的内存空间布局

- 在uCore中，CPU先在bootasm.S（实模式）中通过调用BIOS中断，将物理内存的相关描述符写入特定位置`0x8000`，然后读入kernel至物理地址`0x10000`、虚拟地址`0xC0000000`。
- 而kernel在`page_init`函数中，读取物理内存地址`0x8000`处的内存，查找最大物理地址，并计算出所需的**页面数**。虚拟页表`VPT(Virtual Page Table)`的地址紧跟`kernel`，其地址为4k对齐。虚拟地址空间结构如下所示：

```
/* *
 * 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.
 *
 * */
```

完成**物理内存页管理初始化工作**后，其物理地址的分布空间如下

```
+----------------------+ <- 0xFFFFFFFF(4GB)        ------------------------  4GB
|   一些保留内存，例如用于|                                                保留空间
|   32bit设备映射空间等   |
+----------------------+ <- 实际物理内存空间结束地址 ----------------------
|                      |
|                      |
|      用于分配的          |                                                可用的空间
|     空闲内存区域         |
|                      |
|                      |
|                      |
```

```
+----------------------+ <- 空闲内存起始地址          ----------
-----------------
|      VPT页表存放位置        |
 VPT页表存放的空间    (4MB左右)
+----------------------+ <- bss段结束处              ---------
------------------
|uCore的text、data、bss |
 uCore各段的空间
+----------------------+ <- 0x00100000(1MB)          --------
-------------------- 1MB
|        BIOS ROM        |
+----------------------+ <- 0x000F0000(960KB)
|      16bit设备扩展ROM   |                                  显存与
其他ROM映射的空间
+----------------------+ <- 0x000C0000(768KB)
|      CGA显存空间        |
+----------------------+ <- 0x000B8000               --------
-------------------- 736KB
|         空闲内存         |
+----------------------+ <- 0x00011000(+4KB)
 uCore header的内存空间
| uCore的ELF header数据 |
+----------------------+ <-0x00010000                --------
-------------------- 64KB
|         空闲内存         |
+----------------------+ <- 基于bootloader的大小
 bootloader的
|       bootloader的     |
 内存空间
|      text段和data段      |
+----------------------+ <- 0x00007C00               --------
-------------------- 31KB
|    bootloader和uCore   |
|        共用的堆栈         |                                  堆
栈的内存空间
+----------------------+ <- 基于栈的使用情况
|      低地址空闲空间      |
+----------------------+ <-  0x00000000              --------
-------------------- 0KB
```

易知，其页表地址之上的物理内存空间是空闲的（除去保留的内存），故将该物理地址之下的物理空间对应的页表全部设置为保留(reserved)。并将这些空闲的内存全部添加进页表项中。

## 4) 段页式存储管理（重要）

> 在保护模式中，x86 体系结构将内存地址分成三种：**逻辑地址**（也称**虚拟地址**）、**线性地址**和**物理地址**。

- 段式存储在内存保护方面有优势，页式存储在内存利用和优化转移到后备存储方面有优势。
- 在段式存储管理基础上，给每个段加一级页表。同时，通过指向相同的页表基址，实现进程间的段共享。
- 在段页式管理中，操作系统弱化了段式管理中的功能，实现以分页为主的内存管理。段式管理只起到了一个过滤的作用，它将地址不加转换直接映射成线性地址。将虚拟地址转换为物理地址的过程如下：
  - 根据段寄存器中的段选择子，获取GDT中的特定基址并加上目标偏移来确定**线性地址**。由于GDT中所有的基址全为0（因为弱化了段式管理的功能，对等映射），所以此时的逻辑地址和线性地址是相同的。
  - 根据该线性地址，获取对应页表项，并根据该页表项来获取对应的物理地址。
- **一级页表（页目录表PageDirectoryTable, PDT）的起始地址存储于`%cr3`寄存器中。**

### a. 启动页机制（重要）

启动页机制的代码很简单，其对应的汇编代码为

```
# labcodes/lab2/kern/init/entry.S
kern_entry:
    # load pa of boot pgdir
    movl $REALLOC(__boot_pgdir), %eax
    movl %eax, %cr3
    # enable paging
    movl %cr0, %eax
    orl $(CR0_PE | CR0_PG | CR0_AM | CR0_WP | CR0_NE | CR0_TS | CR0_EM | CR0_MP), %eax
    andl $~(CR0_TS | CR0_EM), %eax
    movl %eax, %cr0

    # update eip
    # now, eip = 0x1xxxxx
    leal next, %eax
    # set eip = KERNBASE + 0x1xxxxx
    jmp *%eax
next:
  # .....省略剩余代码
```

```
19
20  # kernel builtin pgdir
21  # an initial page directory (Page Directory Table, PDT)
22  # These page directory table and page table can be reused!
23  .section .data.pgdir
24  .align PGSIZE
25  __boot_pgdir:
26  .globl __boot_pgdir
27      # map va 0 ~ 4M to pa 0 ~ 4M (temporary)
28      .long REALLOC(__boot_pt1) + (PTE_P | PTE_U | PTE_W)
29      .space (KERNBASE >> PGSHIFT >> 10 << 2) - (. -
    __boot_pgdir) # pad to PDE of KERNBASE
30      # map va KERNBASE + (0 ~ 4M) to pa 0 ~ 4M
31      .long REALLOC(__boot_pt1) + (PTE_P | PTE_U | PTE_W)
32      .space PGSIZE - (. - __boot_pgdir) # pad to PGSIZE
33
34  .set i, 0
35  __boot_pt1:
36  .rept 1024
37      .long i * PGSIZE + (PTE_P | PTE_W)
38      .set i, i + 1
39  .endr
```

- 首先，将一级页表 **__boot_pgdir** （页目录表PDT）的**物理**基地址加载进`%cr3`寄存器中。
    - 该一级页表**暂时**将虚拟地址 **0xC0000000 + (0~4M)** 以及虚拟地址 **(0~4M)** 设置为物理地址 **(0-4M)** 。

      之后会重新设置一级页表的映射关系。

      > 为什么要将两段虚拟内存映射到同一段物理地址呢？思考一下，答案就在下方。

- 之后，设置`%cr0`寄存器中**PE、PG、AM、WP、NE、MP**位，关闭**TS** 与**EM**位，以启动分页机制。

  > 先介绍一下`%cr0`寄存器主要3个标志位的功能：
  >
  > - **P**rotection **E**nable: 启动保护模式，默认只是打开分段。
  > - **Pag**ing: 设置分页标志。只有PE和PG位同时置位，才能启动分页机制。
  > - **W**rite **P**rotection: 当该位为1时，CPU会禁止ring0代码向read only page写入数据。这个标志位主要与**写时复制**有关系。
  >
  > 除**PE、PG**与**WP** 的**其他**标志位与分页机制关联不大，其设置或清除的原因盲猜可能是通过启动分页机制这个机会来**顺便做个初始化**。

> 当改变PE和PG位时，必须小心。只有**当执行程序至少有部分代码和数据在线性地址空间和物理地址空间中具有相同地址时，我们才能改变PG位的设置**。
>
> 因为当`%cr0`寄存器一旦设置，则**分页机制立即开启**。此时这部分具有相同地址的代码在分页和未分页之间起着桥梁的作用。无论是否开启分页机制，这部分代码都必须具有相同的地址。
>
> 而这一步的操作能否成功，关键就在于一级页表的设置。一级页表将虚拟内存中的两部分地址**KERNBASE+(0-4M)** 与 **(0-4M)** 暂时都映射至物理地址 **(0-4M)** 处，这样就可以满足上述的要求。

- 最后，**必须**来个简单的`jmp`指令，将`eip`从物理地址“修改”为虚拟地址。

> 在修改该了PE位之后程序必须立刻使用一条跳转指令，以刷新处理器执行管道中已经获取的不同模式下的任何指令。

## b. uCore段页机制启动过程

- bootloader在启动保护模式前，会设置一个临时GDT以便于进入CPU保护模式后的bootloader和uCore所使用。
- uCore被bootloader加载进内存后，在`kern_entry`中启动页机制。
- 在`pmm_init`中建立双向链表来管理物理内存，并设置一级页表（页目录表）与二级页表。
- 最后重新加载新的GDT。

> lab2相对于lab1，新增了页机制相关的处理过程，其他过程没有改变。

## c. uCore物理页结构

- uCore中用于管理物理页的结构如下所示

```
/* *
* struct Page - Page descriptor structures. Each Page describes one
* physical page. In kern/mm/pmm.h, you can find lots of useful functions
* that convert Page to other data types, such as phyical address.
* */
struct Page {
  int ref;                 // 当前页被引用的次数，与内存共享有关
  uint32_t flags;          // 标志位的集合，与eflags寄存器类似
  unsigned int property;   // 空闲的连续page数量。这个成员只会用在连续空闲page中的第一个page
  list_entry_t page_link; // 两个分别指向上一个和下一个非连续空闲页的指针。
};
```

目前在lab2中，flags可以设置的位只有`reserved`位和`Property`位。

> `reserved`位表示当前页是否被保留，一旦保留该页，则该页无法用于分配；
>
> `Property`位表示当前页是否已被分配，为1则表示已分配。

- 所有的数据结构Page都存放在**一维Page结构数组**中。但请注意，这并非虚拟页表（VPT），即该**一维Page结构数组并非分页机制用于*将虚拟地址转换为物理地址*这个过程所用到的一级与二级页表**，它们只是用于设置对应物理页表的相关属性，例如当前物理页表被二级页表的引用次数等等。
- 同时，该一维Page结构数组的存储位置与虚拟页表VPT的存储位置不同。前者的起始存储地址为kernel结尾地址向上4k对齐后的第一个物理页面，而后者则存储于指定虚拟地址`0xFAC00000`。
- 页目录表使用**线性地址**的首部(PDX, 10bit)作为索引，二级页表使用**线性地址**的中部(PTX, 10bit)作为索引，而Page结构数组使用**物理地址**的首部与中部(PPN, 20bit)作为索引（注意是**物理**地址）。
- 为了加快查找，所有连续空闲pages中的第一个Page结构都会构成一个双向链表。相互链接，其第一个结点是`free_area.free_list`

```
/* 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_area_t free_area;
```

### d. 虚拟页表结构

每个页表项（PTE）都由一个32位整数来存储数据，其结构如下

```
      31-12      9-11     8    7    6   5   4      3    2   1
  0
+--------------+-------+-----+----+---+---+-----+-----+---+---
+---+
|     Offset   | Avail | MBZ | PS | D | A | PCD | PWT | U | W
| P |
+--------------+-------+-----+----+---+---+-----+-----+---+---
+---+
```

- 0 - **P**resent: 表示当前PTE所指向的物理页面是否驻留在内存中
- 1 - **W**riteable: 表示是否允许读写
- 2 - **U**ser: 表示该页的访问所需要的特权级。即User(ring 3)是否允许访问
- 3 - **P**age**W**rite**T**hough:  表示是否使用write through缓存写策略
- 4 - **P**age**C**ache**D**isable: 表示是否**不对**该页进行缓存
- 5 - **A**ccess: 表示该页是否已被访问过
- 6 - **D**irty: 表示该页是否已被修改
- 7 - **P**age**S**ize: 表示该页的大小
- 8 - **M**ust**B**e**Z**ero: 该位必须保留为0
- 9-11 - **Avail**able: 第9-11这三位并没有被内核或中断所使用，可保留给OS使用。
- 12-31 - Offset: 目标地址的后20位。

> 因为目标地址以4k作为对齐标准，所以该地址的低12位永远为0，故这12位空间可用于设置标志位。

## e. 自映射

- 自映射的好处
  - 当页目录与页表建立完成后，如果需要按虚拟地址的地址**顺序显示整个页目录表和页表的内容**，则要查找页目录表的页目录表项内容，并根据页目录表项内容找到页表的物理地址，再转换成对应的虚地址，然后访问页表的虚地址，搜索整个页表的每个页目录项。这样的过程比较繁琐，而自映射可以改善这个过程。
  - 节省4KB空间
  - 方便用户态程序访问页表，可以用这种方式实现一个用户地址空间的映射
- 页表自映射的关键点
  - 把所有的页表（4KB * 1024个）放到**连续**的4MB **虚拟地址** 空间中，并且要求这段空间4MB对齐，这样，**就会有一张虚拟页的内容与页目录的内容完全相同。**
  - 页目录结构必须和页表结构相同。
  - 此时在页目录表中，会存在一个页目录条目，该页目录条目指向某个二级页表。而该二级页表的物理地址，正是页目录表所处于物理页的**物理**地址。

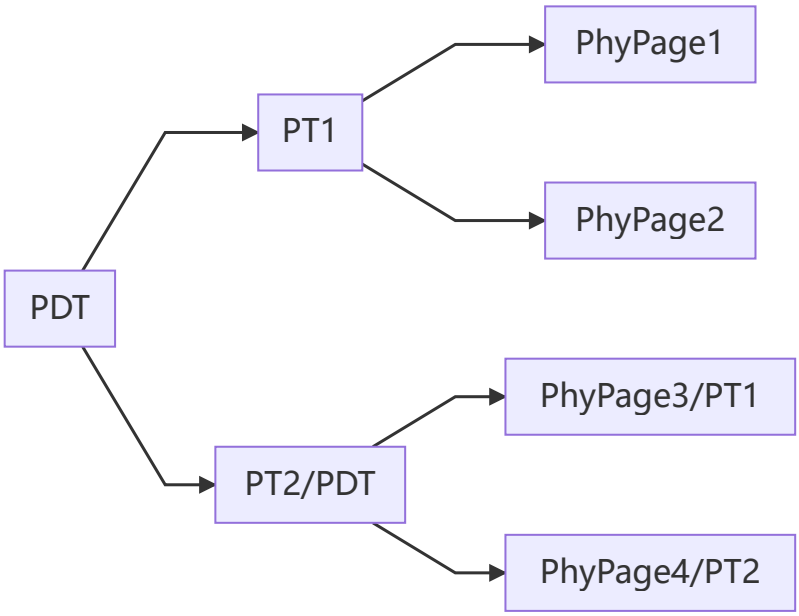
    即，页目录表中存在一个页目录条目，该条目内含的物理地址就是页目录表本身的物理地址。

    uCore中的这条代码证实了这个结论:

    ```
    // recursively insert boot_pgdir in itself
    // to form a virtual page table at virtual address VPT
    //
    // PDX(VPT)为4MB虚拟页表所对应的PageDirectoryIndex
    boot_pgdir[PDX(VPT)] = PADDR(boot_pgdir) | PTE_P | PTE_W;
    ```

    而下面这张图演示了其指向过程:

    > 注意页目录表此时存储于VPT的4MB范围中。




- 参考：页表自映射

### 5) uCore栈的迁移

- 在原先的lab1中，bootloader所设置的栈起始地址为`0x7c00`，之后的uCore的代码也沿用了这个栈，但仍然划分出了一个全局数组作为TSS上的ring0栈地址（该全局数组位于uCore的bss段）。

  > 注意此时的**两个**内核栈是不一样的，一个是中断外使用的栈，另一个是中断内使用的栈。

- 而在lab2中，栈稍微做了一些改变。bootloader所设置的栈起始地址仍然为`0x7c00`，但在将uCore加载进内存之后，在`kern_entry`中，该部分代码在启动页机制后将栈设置为uCore data段上的一个全局数组的末尾地址`bootstacktop`（8KB），并也在`gdt_init`将TSS ring0栈地址设置为了`bootstacktop`。

  > 与Lab1不同，之后内核可以使用的内核栈只有一个。

  > 中断处理程序可能会从高地址开始，向下覆盖ring3的栈数据。这个漏洞可能是因为未完全实现的内存管理机制所导致的。

## 2. 练习解答

### 1) 练习0

> **填写已有实验**，将完成的实验1中的代码添加进实验2中。

这个没什么好说的，一个个照搬就成。

## 2) 练习1

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。

> **实现 first-fit 连续物理内存分配算法。**

原先的uCore实验2代码中几乎已经完全实现了first-fit算法，但其中仍然存在一个问题，以至于无法通过check。什么问题呢：

> `first-fit`算法要求将空闲内存块**按照地址从小到大的方式**连起来。

但uCore中的代码没有实现这一点。所以要手动修改相关的代码。

- `default_init_memmap`
  - 该函数将新页面插入链表时，没有按照地址顺序插入

    ```
    1 list_add(&free_list, &(base->page_link));
    ```

  - 故需要修改该行代码，使其按地址顺序插入至双向链表中。

    ```
    1 list_add_before(&free_list, &(base->page_link));
    ```

- `default_alloc_pages`
  - 在原先的代码中，当获取到了一个大小足够大的页面地址时，程序会先将该页头从链表中断开，切割，并将剩余空间放回链表中。但将*剩余空间放回链表*时，并没有按照地址顺序插入链表。

    > 连续空闲页面中的第一个页称为页头，page header。

    ```
    1  if (page != NULL) {
    2      list_del(&(page->page_link));
    3      if (page->property > n) {
    4          struct Page *p = page + n;
    5          p->property = page->property - n;
    6          // 注意这一步
    7          list_add(&free_list, &(p->page_link));
    8      }
    9      nr_free -= n;
    10     ClearPageProperty(page);
    11 }
    ```

- 以下是修改后的代码

```
if (page != NULL) {
    if (page->property > n) {
        struct Page *p = page + n;
        p->property = page->property - n;
        SetPageProperty(p);
        // 注意这一步add after
        list_add_after(&(page->page_link), &(p->page_link));
    }
    list_del(&(page->page_link));
    nr_free -= n;
    ClearPageProperty(page);
}
```

- `default_free_pages`
  - 该函数默认会在函数末尾处，将待释放的页头插入至链表的第一个节点。

```
list_add(&free_list, &(base->page_link));
```

  - 所以我们需要修改这部分代码，使其按地址顺序插入至对应的链表结点处。

```
// 将空闲页面按地址大小插入至链表中
for(le = list_next(&free_list); le != &free_list; le = list_next(le))
{
    p = le2page(le, page_link);
    if (base + base->property <= p) {
        assert(base + base->property != p);
        break;
    }
}
list_add_before(le, &(base->page_link));
```

## 3) 练习1知识点

根据实验指导书中的实验执行流程概述，先了解分析ucore如何对物理内存进行管理，再完成实验练习。

在对物理内存进行管理之前，需要先进行物理内存布局的探测，探测得到的内存映射存放在e820map中。物理内存以页的形式进行管理，页的信息保存在Page结构中，而Page以链式结构保存在链表free_area_t中。对于物理内存的管理，ucore是通过定义一个pmm_manager实现的，其中包含了初始化需要管理的物理内存空间，初

始化链表，内存空间分配释放等功能的函数。在内核启动后，会初始化pmm_manager，调用page_init，使用init_memmap将e820map中的物理内存纳入物理内存页管理（将空闲页初始化并加入链表），接下来就可以使用pmm_manager中的空间分配和释放等函数进行内存管理了。

## 1.探测系统物理内存布局

对物理内存空间进行管理，首先要对物理内存布局进行探测。本实验中是在bootloader进入保护模式之前通过BIOS中断获取。基于INT 15h，通过e820h中断探测物理内存信息。并将探测到的物理内存对应的映射存放在物理地址0x8000处，定义e820map结构保存映射。在bootasm.S中有内存探测部分的代码，e820map的结构定义则在memlayout.h中。

```
//memlayout.h
struct e820map {
    int nr_map;
    struct {
        uint64_t addr;
        uint64_t size;
        uint32_t type;
    } __attribute__((packed)) map[E820MAX];
};
//bootasm.S
probe_memory:
    movl $0, 0x8000             #存放内存映射的位置
    xorl %ebx, %ebx
    movw $0x8004, %di           #0x8004开始存放map
start_probe:
    movl $0xE820, %eax          #设置int 15h的中断参数
    movl $20, %ecx              #内存映射地址描述符的大小
    movl $SMAP, %edx
    int $0x15
    jnc cont                    #CF为0探测成功
    movw $12345, 0x8000
    jmp finish_probe
cont:
    addw $20, %di               #下一个内存映射地址描述符的位置
    incl 0x8000                 #nr_map+1
    cmpl $0, %ebx               #ebx存放上次中断调用的计数值，判断是否继续进行探测
    jnz start_probe
```

## 2.以页为单位管理物理内存

物理内存是以页为单位进行管理的。探测可用物理内存空间的情况后，就可以建立相应的数据结构来管理物理内存页了。每个内存页使用一个Page结构来表示，Page的定义在memlayout.h中，如下：

```
//memlayout.h中的Page定义
struct Page {
    int ref;                        // 引用计数
    uint32_t flags;                 // 状态标记
    unsigned int property;          // 在first-fit中表示地址连续的空闲页的个数，空闲块头部才有该属性
    list_entry_t page_link;         // 双向链表
};
//状态
#define PG_reserved                 0       //表示是否被保留
#define PG_property                 1       //表示是否是空闲块第一页
```

初始情况下，空闲物理页可能都是连续的，随着物理页的分配与释放，大的连续内存空闲块会被分割为地址不连续的多个小连续内存空闲块。为了管理这些小连续内存空闲块，使用一个双向链表进行管理，定义free_area_t数据结构，包含一个list_entry结构的双向链表指针，记录当前空闲页个数的无符号整型：

```
typedef struct {
    list_entry_t free_list;         // 链表
    unsigned int nr_free;           // 空闲页个数
} free_area_t;
```

对以页为单位进行内存管理，还有两个问题需要解决。一个是找到管理页级物理内存空间所需的Page结构的内存空间的位置，另一个是要找到空闲空间开始的位置。

可以根据内存布局信息找到最大物理内存地址maxpa计算出页的个数，并计算出管理页的Page结构需要的空间。ucore的结束地址（即.bss段的结束地址，用全局变量end表示）以上的空间空闲，从这个位置开始存放Pages结构，而存放Pages结构的结束的位置以上就是空闲物理内存空间。将空闲的物理内存空间使用init_memmap()函数纳入物理内存管理器，部分代码如下：

```
//pmm.c中的page_init的部分代码
    npage = maxpa / PGSIZE;
 //页数
    pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);
  //Page结构的位置
```

```
    for (i = 0; i < npage; i ++) {
        SetPageReserved(pages + i);
 //每一个物理页默认标记为保留
    }
    //空闲空间起始
    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;
  //限制空闲地址最小值
            }
            if (end > KMEMSIZE) {
                end = KMEMSIZE;
 //限制空闲地址最大值
            }
            if (begin < end) {
                begin = ROUNDUP(begin, PGSIZE);
 //对齐地址
                end = ROUNDDOWN(end, PGSIZE);
                if (begin < end) {
                    //将空闲内存块映射纳入内存管理
                    init_memmap(pa2page(begin), (end - begin)
/ PGSIZE);
                }
            }
        }
    }
```

## 3.物理内存空间管理的初始化

ucore中建立了一个物理内存页管理框架对内存进行管理，定义如下：

```
//pmm.h中的pmm_manager定义
struct pmm_manager {
    const char *name;                                 // 管理器名称
    void (*init)(void);                               // 初始化管理器
    void (*init_memmap)(struct Page *base, size_t n); // 设置并初始化可管理的内存空间
    struct Page *(*alloc_pages)(size_t n);            // 分配n个连续物理页，返回首地址
    void (*free_pages)(struct Page *base, size_t n);  // 释放自Base起的连续n个物理页
    size_t (*nr_free_pages)(void);                    // 返回剩余空闲页数
    void (*check)(void);                              // 用于检测分配释放是否正确
};
```

其中，init_memmap用于对页内存管理的Page的初始化，在上面提到的page_init负责确定探查到的物理内存块与对应的struct Page之间的映射关系，Page的初始化工作由内存管理器的init_memmap()来完成。而init则用于初始化已定义好的free_area_t结构的free_area。

在内核初始化的kern_init中会调用pmm_init()，pmm_init()中会调用init_pmm_manager()进行初始化，使用默认的物理内存页管理函数，即使用default_pmm.c中定义的default_init等函数进行内存管理，本实验中需要实现的分配算法可以直接通过修改这些函数进行实现。

```
//pmm.c中的init_pmm_manager()定义，在pmm_init()中调用，在kernel_init()中初始化
static void
init_pmm_manager(void) {
    pmm_manager = &default_pmm_manager;
//pmm_manager指向default_pmm_manager
    cprintf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}
//default_pmm.c中的default_init()
static void
default_init(void) {
    list_init(&free_list);                            //初始化链表
    nr_free = 0;
}
```

```
//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));                    //检查是否为保留页
        //property设置为0，只有空闲块的第一页使用该变量,标志位清0
        p->flags = p->property = 0;
        set_page_ref(p, 0);                         //引用计数为0
    }
    base->property = n;                             //设置第一页的property
    SetPageProperty(base);
    nr_free += n;                                   //空闲页+n
    list_add(&free_list, &(base->page_link));       //将空闲块加入列表
}
```

## 4.first-fit算法的实现

在pmm_manager完成链表的初始化，page_init完成页的初始化后，就可以使用pmm_manager的函数进行内存空间管理了。该练习中使用默认的pmm_manager进行管理，其中的default_init和default_init_memmap可以直接使用，只需要修改default_alloc_pages，default_free_pages，实现first-fit算法以及内存空间释放。

### 使用的函数及宏定义

使用的链表相关操作和宏定义如下。ucore对空闲内存空间管理使用的链表与常见的链表不同，链表只包含了前后节点的信息，而链表包含在Page结构中，这样做是为了实现链表通用性（c语言中并不支持泛型功能，因此采用这种方式）。而通过链表节点获取节点数据是通过宏定义le2page实现的。链表定义和部分相关函数如下：

```
//双向链表的定义
struct list_entry {
    struct list_entry *prev, *next;
};
typedef struct list_entry list_entry_t;
 //返回下一个节点
static inline list_entry_t *
list_next(list_entry_t *listelm) {
    return listelm->next;
}
```

```
//删除当前节点
static inline void
list_del(list_entry_t *listelm) {
    __list_del(listelm->prev, listelm->next);
}
//前插节点，还有类似的list_add_after
static inline void
list_add_before(list_entry_t *listelm, list_entry_t *elm) {
    __list_add(elm, listelm->prev, listelm);
}
```

le2page通过page_link地址减去其相对于Page结构的偏移，实现从链表节点找到对应的数据。

```
#define le2page(le, member)
    to_struct((le), struct Page, member
#define to_struct(ptr, type, member)

    ((type *)((char *)(ptr) - offsetof(type, member))))
```

还有一些其他函数需要使用，以设置Page的信息。

```
//pmm.h中定义的设置引用计数的函数
static inline void
set_page_ref(struct Page *page, int val) {
    page->ref = val;
}
//memlayou.h中的关于页标志位设置的宏定义
#define SetPageReserved(page)       set_bit(PG_reserved, &((page)->flags))
#define ClearPageReserved(page)     clear_bit(PG_reserved, &((page)->flags))
#define PageReserved(page)          test_bit(PG_reserved, &((page)->flags))
#define SetPageProperty(page)       set_bit(PG_property, &((page)->flags))
#define ClearPageProperty(page)     clear_bit(PG_property, &((page)->flags))
#define PageProperty(page)          test_bit(PG_property, &((page)->flags))
```

**使用first-fit实现default_alloc_pages**

使用first-fit实现空闲空间分配，只需要遍历空闲链表，找到合适的块大小，重新设置标志位并从链表中删除该页，如果找到的块大于需要的大小，则需要进行分割，最后更新空闲页数，返回分配好的页块的地址。

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    //空闲页不够，直接返回
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    //寻找合适的空闲块
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        ClearPageProperty(page);                //page已被分配
        //如果空闲块过大则进行分割
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);                 //空闲块
            list_add_after(&(page->page_link), &(p->page_link));
        }
        list_del(&(page->page_link));
        nr_free -= n;
    }
    return page;
}
```

### default_free_pages

释放空间，首先确定需要释放的n都页是未被保留且非空闲的页，然后将这些页的标志位和引用计数清零，并将释放空间的第一页的property设置为n，即空闲块共有n页。接下来完成空闲块的合并，对于相邻的空间，将高地址的块合并到低地址的空闲块中，删除被合并的块，并重新设置空闲块的第一页的property，最后由于空闲链表按地址空间由低到高排列空闲块，还需要找到插入的位置。

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    //将需要释放的页设为空闲状态
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);                 //引用计数清0
    }
    SetPageProperty(base);
    base->property = n;                     //空闲块第一页，property=n
    //空闲块的合并
    list_entry_t *le = list_next(&free_list);
    while (le != &free_list) {
        p = le2page(le, page_link);
        le = list_next(le);
        if (base + base->property == p) {
            base->property += p->property;
            p->property=0;                  //不再是空闲块第一页，property清0
            SetPageProperty(base);          //设置为空闲块第一页
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            p->property += base->property;
            base->property=0;
            SetPageProperty(p);              //设置为空闲块第一页
            list_del(&(base->page_link));
            base=p;                          //更新空闲块第一页
        }
    }
    le = list_next(&free_list);
    //找到插入位置
     while (le != &free_list)
    {
        p = le2page(le, page_link);
        if (base + base->property <= p)
        {
            break;
        }
        le = list_next(le);
    }
```

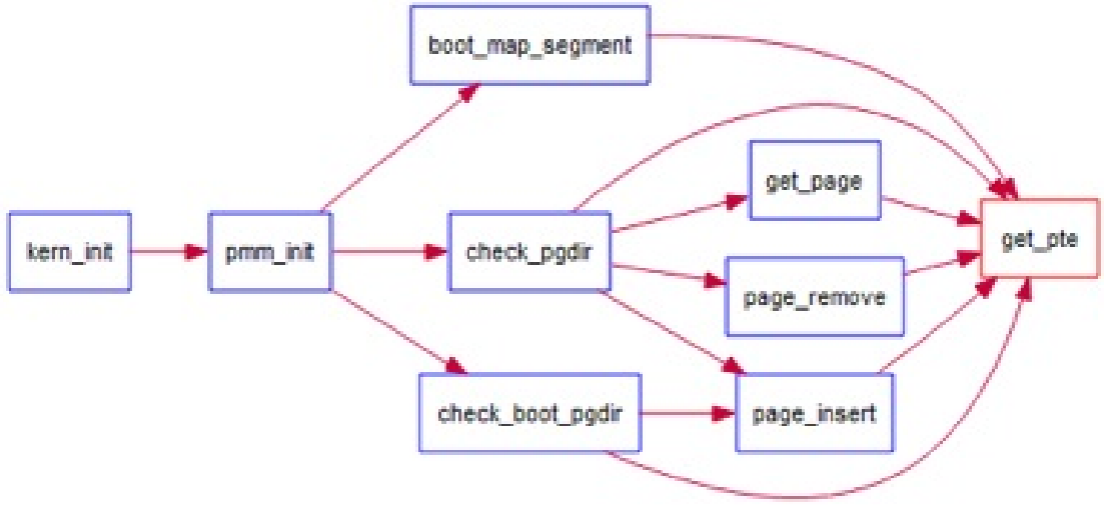
```
43      //将base插入到正确位置
44      list_add_before(le, &(base->page_link));
45      nr_free += n;
46  }
```

**改进空间**

每次查找链表都需要进行遍历，时间复杂度较高，且在空闲链表开头会产生许多小的空闲块，仍然有优化空间。

# 4) 练习2

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：




> **实现寻找虚拟地址对应的页表项**.

> 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。

以下为实现的代码

```
1  pte_t * get_pte(pde_t *pgdir, uintptr_t la, bool create) {
2      // 获取传入的线性地址中所对应的页目录条目的物理地址
3      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)];
}
```

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

> 请查看虚拟页表结构

- 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?

> 以下答案参考了其他blog，具体细节留待以后再来研究。

  - 将引发页访问异常的地址将被保存在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

# 5) 练习2知识点

## 1.段页式管理

在保护模式中，内存地址分成三种：逻辑地址、线性地址和物理地址。逻辑地址即是程序指令中使用的地址，物理地址是实际访问内存的地址。段式管理的映射将逻辑地址转换为线性地址，页式管理的映射将线性地址转换为物理地址。

ucore中段式管理仅为一个过渡，逻辑地址与线性地址相同。而页式管理是通过二级页表实现的，地址的高10位为页目录索引，中间10位为页表索引，低12位为偏移（页对齐，低12位为0）。一级页表的起始物理地址存放在 boot_cr3中。

## 2.页目录项和页表项的组成

问题一：请描述页目录项（Pag Director Entry）和页表（Page Table Entry）中每个组成部分的含义和以及对ucore而言的潜在用处

**页目录项的组成**

- 前20位表示该PDE对应的页表起始位置
- 第9-11位保留给OS使用
- 第8位可忽略
- 第7位用于设置Page大小，0表示4KB
- 第6位为0
- 第5位表示该页是否被写过
- 第4位表示是否需要进行缓存
- 第3位表示CPU是否可直接写回内存
- 第2位表示该页是否可被任何特权级访问
- 第1位表示是否允许读写
- 第0位为该PDE的存在位

**页表项的组成**

- 前20位表示该PTE指向的物理页的物理地址
- 第9-11位保留给OS使用
- 第8位表示在 CR3 寄存器更新时无需刷新 TLB 中关于该页的地址
- 第7位恒为0
- 第6位表示该页是否被写过
- 第5位表示是否可被访问
- 第4位表示是否需要进行缓存
- 第0-3位与页目录项的0-3位相同

## 3.页访问异常的处理

问题二：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?

当发生页访问异常时，硬件需要将发生错误的地址存放在cr2寄存器中，向栈中压入EFLAGS，CS，EIP等，如果异常发生在用户态，还需要进行特权级切换，最后根据中断描述符找到中断服务例程，接下来由中断服务例程处理该异常。

## 4.实现get_pte寻找页表项

练习二要求实现get_pte函数，使该函数能够找到传入的线性地址对应的页表项，返回页表项的地址。为了完成该函数，需要了解ucore中页式管理的相关函数及定义。

```
//PDX(la)：la为线性地址，该函数取出线性地址中的页目录项索引
#define PDX(la) ((((uintptr_t)(la)) >> PDXSHIFT) & 0x3FF)
//取出线性地址中的页表项索引
#define PTX(la) ((((uintptr_t)(la)) >> PTXSHIFT) & 0x3FF)
//KADDR(pa)：返回物理地址pa对应的虚拟地址
#define KADDR(pa) {...}
//set_page_ref(page,1)：设置该页引用次数为VAL
static inline void
set_page_ref(struct Page *page, int val) {
    page->ref = val;
}
//page2pa：找到page结构对应的页的物理地址
static inline uintptr_t
page2pa(struct Page *page) {
    return page2ppn(page) << PGSHIFT;
}
//alloc_page()：分配一页
#define alloc_page() alloc_pages(1)
//页目录项、页表项的标志位
#define PTE_P           0x001                   // 存在位
#define PTE_W           0x002                   // 是否可写
#define PTE_U           0x004                   // 用户是否可访
问
//页目录项和页表项类型
typedef uintptr_t pte_t;
typedef uintptr_t pde_t;
//以下两个函数用于取出页目录项中的页表地址，取出页表项中的页地址
#define PDE_ADDR(pde)   PTE_ADDR(pde)
#define PTE_ADDR(pte)   ((uintptr_t)(pte) & ~0xFFF)
```

需要完成的get_pte函数原型如下，其中pgdir是一级页目录的起始地址，la为线性地址，creat表示是否可以为页表分配新的页。

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create)
```

取出线性地址对应的页表项，首先在页目录中找到对应的页目录项，并判断是否有效（二级页目录是否存在）。如果不存在则根据create判断是否需要创建新的一页存放页表，如果需要则调用alloc_page分配新的一页，将这一页的地址结合标志位设置为页目录项。最后返回页表项的线性地址，使用PDE_ADDR取出页目录项中保存的页表地址，再加上使用PTX从线性地址取出的页表索引，就找到了页表项的位置，使用KADDR转换为线性地址返回（先将页表地址转换为线性地址再加索引同样可行），注意类型转换。

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
```

```
    pde_t *pdep = &pgdir[PDX(la)];                  //找到页目录项
    if (!*pdep&PTE_P) {                             //判断二级页表是否存在
        if(!create) return NULL;                    //create=0则直接返回
        else{
            struct Page* page=alloc_page();         //分配一页用于存放二级页表
            if(page==NULL) return NULL;
            set_page_ref(page,1);                   //引用计数设为1
            uintptr_t pte_pa=page2pa(page);         //分配的页物理地址
            memset(KADDR(pte_pa),0,PGSIZE);         //清除页面内容
            *pdep=pte_pa | PTE_P | PTE_W | PTE_U;
        }
    }
    return ((pte_t*)KADDR(PDE_ADDR(*pdep)))+PTX(la);      //返回页表项的地址
}
```

## 6) 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**.

> 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。

以下为实现的代码：

```
//page_remove_pte - free an Page sturct which is related linear address la
//                - and clean(invalidate) pte which is related linear address la
//note: PT is changed, so the TLB need to be invalidate
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    // 如果传入的页表条目是可用的
    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);
    }
}
```

- 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？
    - 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。
    - 页目录表中存放着数个页表条目PTE，这些页表条目中存放了某个二级页表所在物理页的信息，包括该二级页表的**物理地址**，但使用**线性地址**的头部PDX(Page Directory Index)来索引页目录表。

        > 总结一下，页目录表内存放二级页表的**物理地址**，但却使用**线性地址**索引页目录表中的条目。

    - 而页表（二级页表）与页目录（一级页表）具有类似的特性，页表中的页表项指向所管理的物理页的**物理地址**（不是数据结构Page的地址），使用线性地址的中部PTX(Page Table Index)来索引页表。
    - 当二级页表获取物理页时，需要对该物理页所对应的数据结构page来做一些操作。其操作包括但不限于设置引用次数，这样方便共享内存。

> 为什么页目录表中存放的是**物理**地址呢？可能是为了防止递归查找。
>
> 即原先查找页目录表的目的是想将某个线性地址转换为物理地址，但如果页目录表中存放的是二级页表的**线性**地址，则需要先查找该二级页表的物理地址，此时需要递归查找，这可能会出现永远也查找不到物理地址的情况。

- 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**
    - 将`labcodes/lab2/tools/kernel.ld`中的加载地址从`0xC0100000`修改为`0x0`

        ```
        // 修改前
        . = 0xC0100000;
        // 修改后
        . = 0x0;
        ```

    - 将`mm/`中的内核偏移地址修改为0

```
// 修改前
#define KERNBASE            0xC0000000
// 修改后
#define KERNBASE            0x0
```

- 最后一步，但也是必须要做的一步——**关闭页机制**。将开启页机制的那一段代码删除或注释掉最后一句即可。

```
# 修改后
movl %cr0, %eax
orl $(CR0_PE | CR0_PG | CR0_AM | CR0_WP | CR0_NE |
CR0_TS | CR0_EM | CR0_MP), %eax
andl $~(CR0_TS | CR0_EM), %eax
# 注释了最后一句
# movl %eax, %cr0
```

> 为什么要关闭页机制？只将偏移地址设置为0不够么？这是个值得探讨的问题。
>
> 注意到`kern/init.entry.S`中有这样一段代码
>
> ```
> next:
> # unmap va 0 ~ 4M, it's temporary mapping
> xorl %eax, %eax
> movl %eax, __boot_pgdir
> ```
>
> 当CPU完成了`eip`的地址更新后，这两条指令会删除页目录表中的一个**临时**映射（va 0 ~ 4M to pa 0 ~ 4M）
>
> 但一旦删除了这个临时映射后，CPU无法正常寻址，即便页目录表中还有一个映射（va KERNBASE + (0 ~ 4M) to pa 0 ~ 4M， 注意KERNBASE为0）
>
> 但只要基地址不为0，则不会出错。
>
> 具体的问题在哪呢？或许，需要查询一下intel 80386的相关手册。

# 7) 练习3知识点

## 1.Page与页目录项和页表项的关系

问题一：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？

当页目录项与页表项均有效时，有对应关系。每个页目录项记录一个页表的位置，每个页表项则记录一个物理页的位置，而Page变量保存的就是物理页的信息，因此每个有效的页目录项和页表项，都对应了一个page结构，即一个物理页的信息。

## 2.实现虚拟地址与物理地址相等

问题二：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?

由附录可知，在lab1中，虚拟地址=线性地址=物理地址，ucore的起始虚拟地址（也即物理地址）从0x100000开始。而在lab2中建立了从虚拟地址到物理地址的映射，ucore的物理地址仍为0x100000，但虚拟地址变为了0xC0100000，即最终建立的映射为：virt addr = linear addr = phy addr + 0xC0000000。

```
1  //memlayout.h给出了直观的映射关系
2  *     4G ------------------> +------------------------------
   ---+
3  *                            |
    |
4  *                            |          Empty Memory (*)
     |
5  *                            |
    |
6  *                           +------------------------------
   ---+ 0xFB000000
7  *                            |   Cur. Page Table (Kern, RW)
     | RW/-- PTSIZE
8  *     VPT ------------------> +------------------------------
   ---+ 0xFAC00000
9  *                            |        Invalid Memory (*)
    | --/--
10 *     KERNTOP -------------> +------------------------------
   ---+ 0xF8000000
11 *                            |
    |
12 *                            |    Remapped Physical Memory
    | RW/-- KMEMSIZE
13 *                            |
    |
14 *     KERNBASE ------------> +------------------------------
   ---+ 0xC0000000
15 *                            |
    |
```

```
16  *                               |
     |
17  *                               |
     |
18  *
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

只要取消这个映射，就可以实现虚拟地址和物理地址相等，将ld工具形成的ucore的虚拟地址修改为0x100000，就可以取消映射。

```
1  ENTRY(kern_entry)
2  SECTIONS {
3              /* Load the kernel at this address: "." means the
   current address */
4              . = 0xC0100000;         //修改为0x100000就可以实现虚拟地
   址=物理地址
5              .text : {
6                          *(.text .stub .text.*
   .gnu.linkonce.t.*)
7              }
```

还需要在memlayout.h中将KERNBASE即虚拟地址基址设置为0，并关闭entry.S中对页表机制的开启。

```
1  //memlayout.h中定义的KERNBASE
2  #define KERNBASE 0x0
3  //页表机制开启
4      # enable paging
5      movl %cr0, %eax
6      orl $(CR0_PE | CR0_PG | CR0_AM | CR0_WP | CR0_NE | CR0_TS
   | CR0_EM | CR0_MP), %eax
7      andl $~(CR0_TS | CR0_EM), %eax
8      movl %eax, %cr0         //将这句注释掉
```

pmm_init中的check_pgdir和check_boot_pgdir都假设kernbase不为0，对线性地址为0的地址进行页表查询等，因此会产生各种错误，可以将这两个函数注释掉。

## 3.实现page_remove_pte释放虚拟页并取消二级映射

本练习中需要完成的是page_remove_pte函数，该函数将传入的虚拟页释放，取消二级页表的映射，并且需清除TLB中对应的项。原型如下：

```
1  page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep)
```

使用的相关函数定义如下：

```
//pte2page：找到页表项对应的页
struct Page *page pte2page(*ptep)
//free_page：释放页
free_page(page)
//page_ref_dec：引用计数-1
page_ref_dec(page)
//tlb_invalidate：清除TLB
tlb_invalidate(pde_t *pgdir, uintptr_t la)
```

对于传入的页表项，首先要判断其是否有效，如果有效，将引用计数-1，当引用计数为0时释放该页，再清0二级页表映射，使用tlb_invalidate清除对应的TLB项。最终实现如下：

```
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {

    //判断页表项是否有效
    if (*ptep & PTE_P) {
        struct Page *page = pte2page(*ptep);          //找到对应的页
        page_ref_dec(page);                           //引用计数-1
        if(page->ref==0) free_page(page);             //引用计数为0释放该页
        *ptep=0;                                      //清除二级页表映射
        tlb_invalidate(pgdir,la);                     //修改TLB
        return;
    }
    else return;
}
```

# 8) 扩展练习

## Challenge1

> **buddy system（伙伴系统）分配算法**
>
> Buddy System算法把系统中的可用存储空间划分为存储块(Block)来进行管理, 每个存储块的大小必须是2的n次幂(Pow(2, n)), 即1, 2, 4, 8, 16, 32, 64, 128...

a. 前置准备

伙伴系统中每个存储块的大小都必须是2的n次幂，所以其中必须有个**可以将传入数转换为最接近该数的2的n次幂**的函数，相关代码如下

```
// 传入一个数，返回最接近该数的2的指数（包括该数为2的整数这种情况）
size_t getLessNearOfPower2(size_t x)
{
    size_t _i;
    for(_i = 0; _i < sizeof(size_t) * 8 - 1; _i++)
        if((1 << (_i+1)) > x)
            break;
    return (size_t)(1 << _i);
}
```

b. 初始化

初始时，程序会多次将一块尺寸很大的物理内存空间传入`init_memmap`函数，但该物理内存空间的大小却不一定是2的n次幂，所以需要对其进行分割。设定分割后的内存布局如下

```
/*
buddy system中的内存布局
      某块较大的物理空间
低地址                                    高地址
+-+--+----+--------+-------------------+
| |  |    |        |                   |
+-+--+----+--------+-------------------+
低地址的内存块较小              高地址的内存块较大

*/
```

同时，在双向链表`free_area.free_list`中，令空间较小的内存块在双向链表中靠前，空间较大的内存块在双向链表中靠后；低地址在前，高地址在后。故以下是最终的链表布局：

```
/*
free_area.free_list中的内存块顺序:

1. 一大块连续物理内存被切割后，free_area.free_list中的内存块顺序
    addr: 0x34       0x38           0x40
        +----+     +--------+     +---------------+
    <-> | 0x4| <-> | 0x8    | <-> |     0x10      | <->
        +----+     +--------+     +---------------+

```

```
2. 几大块物理内存（这几块之间可能不连续）被切割后，free_area.free_list中的内存块顺序
    addr: 0x34       0x104       0x38           0x108          0x40                  0x110
        +----+     +----+     +--------+     +--------+     +---------------+     +---------------+
    <-> | 0x4| <-> | 0x4| <-> | 0x8    | <-> | 0x8    | <-> |    0x10       | <-> |     0x10      | <->
        +----+     +----+     +--------+     +--------+     +---------------+     +---------------+
*/
```

根据上面的内存规划，可以得到`buddy_init_memmap`的代码

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

    // 设置当前页向后的curr_n个页
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    // 设置总共的空闲内存页面
    nr_free += n;
    // 设置base指向尚未处理内存的end地址
    base += n;
    while(n != 0)
    {
        size_t curr_n = getLessNearOfPower2(n);
        // 向前挪一块
        base -= curr_n;
        // 设置free pages的数量
        base->property = curr_n;
        // 设置当前页为可用
        SetPageProperty(base);
        // 按照块的大小来插入空闲块，从小到大排序
        // @note 这里必须使用搜索的方式来插入块而不是直接list_add_after(&free_list)，因为存在大的内存块不相邻的情况
        list_entry_t* le;
        for(le = list_next(&free_list); le != &free_list; le = list_next(le))
        {
```

```
            struct Page *p = le2page(le, page_link);
            // 排序方式以内存块大小优先，地址其次。
            if((p->property > base->property)
                 || (p->property ==  base->property && p > base))
                break;
        }
        list_add_before(le, &(base->page_link));
        n -= curr_n;
    }
}
```

c. 空间分配

分配空间时，遍历双向链表，查找大小合适的内存块。

- 若链表中不存在合适大小的内存块，则**对半切割**遍历过程中遇到的第一块大小大于所需空间的内存块。
- 如果切割后的两块内存块的大小还是太大，则继续切割**第一块**内存块。
- 循环该操作，直至切割出合适大小的内存块。
- 最终`buddy_alloc_pages`代码如下

```
static struct Page *
buddy_alloc_pages(size_t n) {
    assert(n > 0);
    // 向上取2的幂次方，如果当前数为2的幂次方则不变
    size_t lessOfPower2 = getLessNearOfPower2(n);
    if (lessOfPower2 < n)
        n = 2 * lessOfPower2;
    // 如果待分配的空闲页面数量小于所需的内存数量
    if (n > nr_free) {
        return NULL;
    }
    // 查找符合要求的连续页
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    // 如果需要切割内存块时，一定分配切割后的前面那块
```

```
    if (page != NULL) {
        // 如果内存块过大，则持续切割内存
        while(page->property > n)
        {
            page->property /= 2;
            // 切割出的右边那一半内存块不用于内存分配
            struct Page *p = page + page->property;
            p->property = page->property;
            SetPageProperty(p);
            list_add_after(&(page->page_link), &(p->page_link));
        }
        nr_free -= n;
        ClearPageProperty(page);
        assert(page->property == n);
        list_del(&(page->page_link));
    }
    return page;
}
```

d. 内存释放

释放内存时

- 先将该内存块按照**内存块大小从小到大与内存块地址从小到大**的顺序插入至双向链表（具体请看上面的链表布局）。
- 尝试向前合并，一次就够。如果向前合并成功，则一定不能再次向前合并。
- 之后循环向后合并，直至无法合并。

> 需要注意的是，在查找两块内存块能否合并时，若当前内存块合并过，则其大小会变为原来的2倍，此时需要遍历比原始大小（合并前内存块大小）更大的内存块。

- 判断当前内存块的位置是否正常，如果不正常，则需要断开链表并重新插入至新的位置。

> 如果当前内存块没有合并则肯定正常，如果合并过则**不一定异常**。

- 最终代码如下

```
static void
buddy_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    // 向上取2的幂次方，如果当前数为2的幂次方则不变
    size_t lessOfPower2 = getLessNearOfPower2(n);
    if (lessOfPower2 < n)
```

```
        n = 2 * lessOfPower2;
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    list_entry_t *le;
    // 先插入至链表中
    for(le = list_next(&free_list); le != &free_list; le = list_next(le))
    {
        p = le2page(le, page_link);
        if ((base->property <= p->property)
                || (p->property ==  base->property && p > base))) {
            break;
        }
    }
    list_add_before(le, &(base->page_link));
    // 先向左合并
    if(base->property == p->property && p + p->property == base) {
        p->property += base->property;
        ClearPageProperty(base);
        list_del(&(base->page_link));
        base = p;
        le = &(base->page_link);
    }

    // 之后循环向后合并
    // 此时的le指向插入块的下一个块
    while (le != &free_list) {
        p = le2page(le, page_link);
        // 如果可以合并(大小相等+地址相邻),则合并
        // 如果两个块的大小相同，则它们不一定内存相邻。
        // 也就是说，在一条链上，可能存在多个大小相等但却无法合并的块
        if (base->property == p->property && base + base->property == p)
        {
            // 向右合并
            base->property += p->property;
```

```
            ClearPageProperty(p);
            list_del(&(p->page_link));
            le = &(base->page_link);
        }
        // 如果遍历到的内存块一定无法合并，则退出
        else if(base->property < p->property)
        {
            // 如果合并不了，则需要修改base在链表中的位置，使大小相同的聚在一起
            list_entry_t* targetLe = list_next(&base->page_link);
            p = le2page(targetLe, page_link);
            while(p->property < base->property)
                 || (p->property ==  base->property && p > base))
                targetLe = list_next(targetLe);
            // 如果当前内存块的位置不正确，则重置位置
            if(targetLe != list_next(&base->page_link))
            {
                list_del(&(base->page_link));
                list_add_before(targetLe, &(base->page_link));
            }
            // 最后退出
            break;
        }
        le = list_next(le);
    }
}
```

e. 算法检查

`buddy_check`是一个不能忽视的检查函数，该函数可以帮助查找出程序内部隐藏的bug。笔者将其中原本用于检查`FIFO`算法的内容修改成检查`buddySystem`的内容。所修改的内容如下：

```
//.....................................................
// 先释放
free_pages(p0, 26);      // 32+  (-:已分配 +: 已释放)
// 首先检查是否对齐2
p0 = alloc_pages(6);     // 8- 8+ 16+
p1 = alloc_pages(10);    // 8- 8+ 16-
assert((p0 + 8)->property == 8);
free_pages(p1, 10);      // 8- 8+ 16+
assert((p0 + 8)->property == 8);
```

```
assert(p1->property == 16);
p1 = alloc_pages(16);   // 8- 8+ 16-
// 之后检查合并
free_pages(p0, 6);      // 16+ 16-
assert(p0->property == 16);
free_pages(p1, 16);     // 32+
assert(p0->property == 32);

p0 = alloc_pages(8);    // 8- 8+ 16+
p1 = alloc_pages(9);    // 8- 8+ 16-
free_pages(p1, 9);     // 8- 8+ 16+
assert(p1->property == 16);
assert((p0 + 8)->property == 8);
free_pages(p0, 8);      // 32+
assert(p0->property == 32);
// 检测链表顺序是否按照块的大小排序的
p0 = alloc_pages(5);
p1 = alloc_pages(16);
free_pages(p1, 16);
assert(list_next(&(free_list)) == &((p1 - 8)->page_link));
free_pages(p0, 5);
assert(list_next(&(free_list)) == &(p0->page_link));

p0 = alloc_pages(5);
p1 = alloc_pages(16);
free_pages(p0, 5);
assert(list_next(&(free_list)) == &(p0->page_link));
free_pages(p1, 16);
assert(list_next(&(free_list)) == &(p0->page_link));

// 还原
p0 = alloc_pages(26);
//.....................................................
```

f. 总结与完整代码

- `buddySystem`在**所分配的内存大小均为2的n次幂**这种环境下，使用效果极佳。
- 由于`buddySystem`的特性，最好使用二叉树而非普通双向链表来管理内存块，这样就可以避免一系列的bug。

  即便普通双向链表可以很好的实现`buddySystem`，但其中仍然存在一个较为麻烦的问题：

  > **当某个物理块释放，将其插入至双向链表后，如果该物理块既可以和上一个物理块合并，又可以和下一个物理块合并，那么此时该合并哪一个物理块？**

这个问题，双向链表无法很好的解决，该问题很可能会使一些物理块因为错误的合并顺序而最终导致内存的碎片化，降低内存的使用率。

## Challenge2

> **任意大小的内存单元slub分配算法**
>
> slub算法，实现两层架构的高效内存单元分配，第一层是基于页大小的内存分配，第二层是在第一层基础上实现基于任意大小的内存分配。可简化实现，能够体现其主体思想即可。

slub算法，实现两层架构的高效内存单元分配，第一层是基于页大小的内存分配，第二层是在第一层基础上实现基于任意大小的内存分配。可简化实现，能够体现其主体思想即可。

参考linux的slub分配算法/，在ucore中实现slub分配算法。要求有比较充分的测试用例说明实现的正确性，需要有设计文档。
SLAB 分配器为每种使用的内核对象建立单独的缓冲区。每种缓冲区由多个 slab 组成，每个 slab就是一组连续的物理内存页框，被划分成了固定数目的对象。根据对象大小的不同，缺省情况下一个 slab 最多可以由 1024 个物理内存页框构成。

内核使用 kmem_cache 数据结构管理缓冲区。由于 kmem_cache 自身也是一种内核对象，所以需要一个专门的缓冲区。所有缓冲区的 kmem_cache 控制结构被组织成以 cache_chain 为队列头的一个双向循环队列，同时 cache_cache 全局变量指向 kmem_cache 对象缓冲区的 kmem_cache 对象。每个 slab 都需要一个类型为 struct slab 的描述符数据结构管理其状态，同时还需要一个 kmem_bufctl_t（被定义为无符号整数）的结构数组来管理空闲对象。如果对象不超过 1/8 个物理内存页框的大小，那么这些 slab 管理结构直接存放在 slab 的内部，位于分配给 slab 的第一个物理内存页框的起始位置；否则的话，存放在 slab 外部，位于由 kmalloc 分配的通用对象缓冲区中。

slab 中的对象有 2 种状态：已分配或空闲。

为了有效地管理 slab，根据已分配对象的数目，slab 可以有 3 种状态，动态地处于缓冲区相应的队列中：

1. Full 队列，此时该 slab 中没有空闲对象。
2. Partial 队列，此时该 slab 中既有已分配的对象，也有空闲对象。
3. Empty 队列，此时该 slab 中全是空闲对象。

在 SLUB 分配器中，一个 slab 就是一组连续的物理内存页框，被划分成了固定数目的对象。slab 没有额外的空闲对象队列（这与 SLAB 不同），而是重用了空闲对象自身的空间。slab 也没有额外的描述结构，因为 SLUB 分配器在代表物理页框的 page 结构中加入 freelist，inuse 和 slab 的 union 字段，分别代表第一个空闲对象的指

针，已分配对象的数目和缓冲区 kmem_cache 结构的指针，所以 slab 的第一个物理页框的 page 结构就可以描述自己。

每个处理器都有一个本地的活动 slab，由 kmem_cache_cpu 结构描述。

以下是`slab_alloc`的实现

```
static __always_inline void *slab_alloc(struct kmem_cache *s,
gfp_t gfpflags, int node, void *addr)
{
    void **object;
    struct kmem_cache_cpu *c;
    unsigned long flags;
    local_irq_save(flags);
    c = get_cpu_slab(s, smp_processor_id()); //获取本处理器的
kmem_cache_cpu 数据结构
    if (unlikely(!c->freelist || !node_match(c, node)))
        object = __slab_alloc(s, gfpflags, node, addr, c); //
假如当前活动 slab 没有空闲对象，或本处理器所在节点与指定节点不一致，则调用
__slab_alloc 函数
    else {
        object = c->freelist; //获得第一个空闲对象的指针，然后更新指
针使其指向下一个空闲对象
          c->freelist = object[c->offset];
          stat(c, ALLOC_FASTPATH);
    }
    local_irq_restore(flags);
    if (unlikely((gfpflags & __GFP_ZERO) &&
object))memset(object, 0, c->objsize);
    return object; //返回对象地址
}

```

`static void *__slab_alloc(struct kmem_cache *s, gfp_t gfpflags, int node, void *addr, struct kmem_cache_cpu *c)`函数实现

```
static void *__slab_alloc(struct kmem_cache *s, gfp_t
gfpflags, int node, void *addr, struct kmem_cache_cpu *c)
{
    void **object;
    struct page *new;
    gfpflags &= ~__GFP_ZERO;
    if (!c->page) // (a)：如果没有本地活动 slab，转到 (f) 步骤获取
slab
        goto new_slab;
```

```
8     slab_lock(c->page);
9     if (unlikely(!node_match(c, node))) // (b)：检查处理器活动 slab 没有空闲对象，转到 (e) 步骤
10        goto another_slab;
11    stat(c, ALLOC_REFILL);
12    load_freelist:object = c->page->freelist;
13    if (unlikely(!object)) // (c)：此时活动 slab 尚有空闲对象，将 slab 的空闲对象队列指针复制到 kmem_cache_cpu 结构的 freelist 字段，把 slab 的空闲对象队列指针设置为空，从此以后只从 kmem_cache_cpu 结构的 freelist 字段获得空闲对象队列信息
14        goto another_slab;
15    if (unlikely(SlabDebug(c->page)))goto debug;
16    c->freelist = object[c->offset]; // (d)：此时活动 slab 尚有空闲对象，将 slab 的空闲对象队列指针复制到 kmem_cache_cpu 结构的 freelist 字段，把 slab 的空闲对象队列指针设置为空，从此以后只从 kmem_cache_cpu 结构的 freelist 字段获得空闲对象队列信息
17    c->page->inuse = s->objects;
18    c->page->freelist = NULL;
19    c->node = page_to_nid(c->page);
20    unlock_out:slab_unlock(c->page);
21    stat(c, ALLOC_SLOWPATH);
22    return object;
23    another_slab:deactivate_slab(s, c); // (e)：取消当前活动 slab，将其加入到所在 NUMA 节点的 Partial 队列中
24    new_slab:new = get_partial(s, gfpflags, node); // (f)：优先从指定 NUMA 节点上获得一个 Partial slab
25    if (new) {
26        c->page = new;
27        stat(c, ALLOC_FROM_PARTIAL);
28        goto load_freelist;
29    }
30    if (gfpflags & __GFP_WAIT) // (g)：加入 gfpflags 标志置有 __GFP_WAIT，开启中断，故后续创建 slab 操作可以睡眠
31     local_irq_enable();
32    new = new_slab(s, gfpflags, node); // (h)：创建一个 slab，并初始化所有对象
33    if (gfpflags & __GFP_WAIT)local_irq_disable();
34    if (new) {
35        c = get_cpu_slab(s, smp_processor_id());
36        stat(c, ALLOC_SLAB);
37        if (c->page) flush_slab(s, c);
38        slab_lock(new);
39        SetSlabFrozen(new);
40        c->page = new;
41        goto load_freelist;
```

```
    }
 if (!(gfpflags & __GFP_NORETRY) && (s->flags & __PAGE_ALLOC_FALLBACK))
 {
     if (gfpflags & __GFP_WAIT) local_irq_enable();
     object = kmalloc_large(s->objsize, gfpflags); // (i)：如果内存不足，无法创建 slab，调用 kmalloc_large（实际调用物理页框分配器）分配对象
     if (gfpflags & __GFP_WAIT)
         local_irq_disable();
     return object;
 }
 return NULL;

 debug:
    if (!alloc_debug_processing(s, c->page, object, addr))
     goto another_slab;
    c->page->inuse++;
    c->page->freelist = object[c->offset];
    c->node = -1;
    goto unlock_out;
}
```

以下是`slab_free`的实现

```
static __always_inline void slab_free(struct kmem_cache *s, struct page *page, void *x, void *addr)
{
    void **object = (void *)x;
    struct kmem_cache_cpu *c;
    unsigned long flags;
    local_irq_save(flags);
    c = get_cpu_slab(s, smp_processor_id());
    debug_check_no_locks_freed(object, c->objsize);
    if (likely(page == c->page && c->node >= 0))
    { // (a)如果对象属于处理器当前活动的 slab，或处理器所在 NUMA 节点号不为 -1（调试使用的值），将对象放回空闲对象队列
     object[c->offset] = c->freelist;
     c->freelist = object;
     stat(c, FREE_FASTPATH);
    }
    else__slab_free (s, page, x, addr, c->offset); // (b)否则调用 __slab_free 函数
    local_irq_restore(flags);
}
```

```
18
```

以下是`static void __slab_free(struct kmem_cache *s, struct page *page,void *x, void *addr, unsigned int offset)`函数的实现

```
1  static void __slab_free(struct kmem_cache *s, struct page
   *page,
2  void *x, void *addr, unsigned int offset)
3  {
4      void *prior;void **object = (void *)x;
5      struct kmem_cache_cpu *c;
6      c = get_cpu_slab(s, raw_smp_processor_id());
7      stat(c, FREE_SLOWPATH);
8      slab_lock(page);
9      if (unlikely(SlabDebug(page)))goto debug;
10     checks_ok:prior = object = page->freelist; // (a)执行本函数
   表明对象所属 slab 并不是某个活动 slab。保存空闲对象队列的指针，将对象放回
   此队列，最后把已分配对象数目减一。
11     page->freelist = object;page->inuse--;
12     if (unlikely(SlabFrozen(page)))
13     {
14         stat(c, FREE_FROZEN);
15         goto out_unlock;
16     }
17     if (unlikely(!page->inuse)) // (b)如果已分配对象数为 0，说明
   slab 处于 Empty 状态，转到 (d) 步骤。
18         goto slab_empty;
19     if (unlikely(!prior))
20     { // (c)1. 如果原空闲对象队列的指针为空，说明 slab 原来的状态为
   Full，那么现在的状态应该是 Partial，将该 slab 加到所在节点的 Partial
   队列中。
21          add_partial(get_node(s, page_to_nid(page)), page,
   1);
22          stat(c, FREE_ADD_PARTIAL);
23     }
24     out_unlock:slab_unlock(page);
25  return;
26
27  slab_empty:
28  if (prior) { // (d)如果 slab 状态转为 Empty，且先前位于节点的
   Partial 队列中，则将其剔出并释放所占内存空间。
29      remove_partial(s, page);
30      stat(c, FREE_REMOVE_PARTIAL);
31  }
32  slab_unlock(page);
```

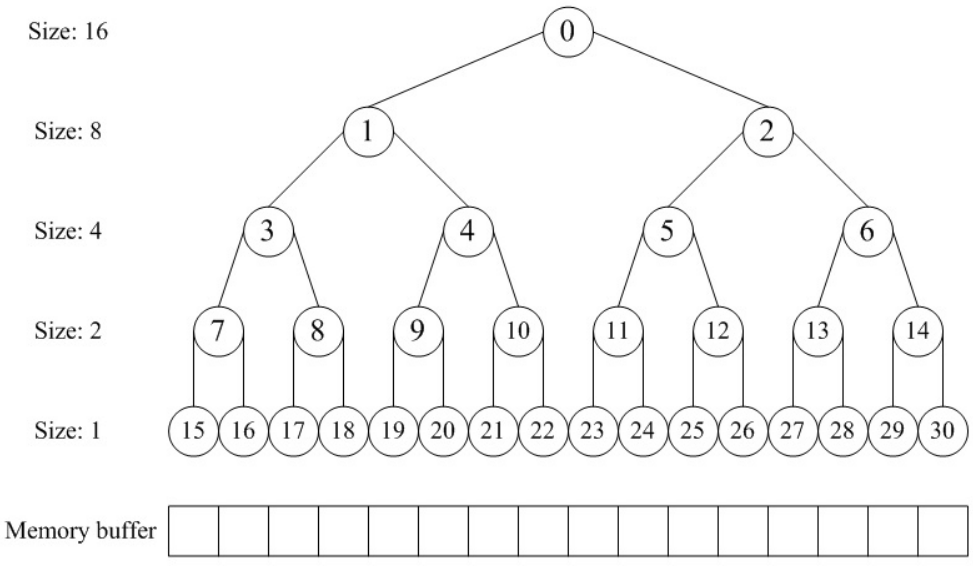
```
 stat(c, FREE_SLAB);
 discard_slab(s, page);
 return;

 debug:if (!free_debug_processing(s, page, x, addr))goto
out_unlock;
 goto checks_ok;
}
```

# 9) 扩展练习知识点

## 1.伙伴系统分配算法

伙伴系统是一种采用二分的方式分割和合并空闲空间的内存分配算法。空闲空间被视为2^n的空间，当有内存空间分配请求时，将空闲空间一分为二，直到刚好可以满足请求大小。在空间释放时，分配程序会检查该块同大小的伙伴块是否空闲，如果是则可以进行合并，并继续上溯，直到完成全部可能的合并。

分配器使用伙伴系统，是通过数组形式的二叉树实现的。二叉树的节点标记相应的内存块是否使用，通过这些标记进行块的分离与合并。二叉树的情况如下图（总大小为16），其中节点数字为在数组中的索引：




## 2.伙伴系统的实现

下面是对实验指导书中给出的伙伴系统的分析。

首先是数据结构和一些宏定义，主要是伙伴系统定义，计算节点等：

```
struct buddy2 {
  unsigned size;//表明物理内存的总单元数
  unsigned longest[1]; //二叉树的节点标记，表明对应内存块的空闲单位
};
#define LEFT_LEAF(index) ((index) * 2 + 1)//左子树节点的值
#define RIGHT_LEAF(index) ((index) * 2 + 2)//右子树节点的值
#define PARENT(index) ( ((index) + 1) / 2 - 1)//父节点的值
#define IS_POWER_OF_2(x) (!((x)&((x)-1)))//x是不是2的幂
#define MAX(a, b) ((a) > (b) ? (a) : (b))//判断a，b大小
#define ALLOC malloc//申请内存
#define FREE free//释放内存
```

在分配时，分配的空间大小必须满足需要的大小，且为2的幂次方，以下函数用于找到合适的大小：

```
static unsigned fixsize(unsigned size) {//找到大于等于所需内存的2的倍数
  size |= size >> 1;
  size |= size >> 2;
  size |= size >> 4;
  size |= size >> 8;
  size |= size >> 16;
  return size+1;
}
```

接下来是分配器的初始化及销毁。初始化传入的参数是需要管理的内存空间大小，且这个大小应该是2的幂次方。在函数中node_size用于计算节点的大小，每次除2，初始化每一个节点。

```
struct buddy2* buddy2_new( int size ) {//初始化分配器
  struct buddy2* self;
  unsigned node_size;//节点所拥有的内存大小
  int i;

  if (size < 1 || !IS_POWER_OF_2(size))
    return NULL;

  self = (struct buddy2*)ALLOC( 2 * size * sizeof(unsigned));
  self->size = size;
  node_size = size * 2;

  for (i = 0; i < 2 * size - 1; ++i) {
    if (IS_POWER_OF_2(i+1))
      node_size /= 2;
```

```
    self->longest[i] = node_size;
  }
  return self;
}

void buddy2_destroy( struct buddy2* self) {
  FREE(self);
}
```

内存分配的实现如下，传入分配器，需要分配的空间大小，首先判断是否可以进行分配，并将空间大小调整为2的幂次方，然后进行分配。分配的过程为遍历寻找合适大小的节点，将找到的节点大小清0表示以被占用，并且需要更新父节点的值，最后返回的值为所分配空间相对于起始位置的偏移。

```
int buddy2_alloc(struct buddy2* self, int size) {
  unsigned index = 0;         //节点在数组的索引
  unsigned node_size;
  unsigned offset = 0;

  if (self==NULL)             //无法分配
    return -1;
  if (size <= 0)
    size = 1;
  else if (!IS_POWER_OF_2(size))//调整大小为2的幂次方
    size = fixsize(size);
  if (self->longest[index] < size)//可分配内存不足
    return -1;
  //从根节点开始向下寻找合适的节点
  for(node_size = self->size; node_size != size; node_size /=
2 ) {
    if (self->longest[LEFT_LEAF(index)] >= size)
      index = LEFT_LEAF(index);
    else
      index = RIGHT_LEAF(index);//左子树不满足时选择右子树
  }
  self->longest[index] = 0;      //将节点标记为已使用
  offset = (index + 1) * node_size - self->size;//计算偏移量
    //更新父节点
  while (index) {
    index = PARENT(index);
    self->longest[index] =
      MAX(self->longest[LEFT_LEAF(index)], self-
>longest[RIGHT_LEAF(index)]);
  }
```

```
  return offset;
}
```

内存释放时，先自底向上寻找已被分配的空间，将这块空间的大小恢复，接下来就可以匹配其大小相同的空闲块，如果块都为空闲则进行合并。

```
void buddy2_free(struct buddy2* self, int offset) {
  unsigned node_size, index = 0;
  unsigned left_longest, right_longest;
  //判断请求是否出错
  assert(self && offset >= 0 && offset < self->size);
  node_size = 1;
  index = offset + self->size - 1;
  //寻找分配过的节点
  for (; self->longest[index] ; index = PARENT(index)) {
    node_size *= 2;
    if (index == 0)                  //如果节点不存在
      return;
  }
  self->longest[index] = node_size;//释放空间
  //合并
  while (index) {
    index = PARENT(index);
    node_size *= 2;

    left_longest = self->longest[LEFT_LEAF(index)];
    right_longest = self->longest[RIGHT_LEAF(index)];

    if (left_longest + right_longest == node_size) //如果可以则合并
      self->longest[index] = node_size;
    else
      self->longest[index] = MAX(left_longest, right_longest);
  }
}
```

以上就是参考资料中给出的伙伴系统的实现（另外还有两个函数分别用于返回当前节点大小和打印内存状态），在ucore中实现伙伴系统的原理相同，但需要对具体的页进行处理分配以及释放，完成对应的buddy.h头文件和buddy.c文件后，修改pmm.c中的init_pmm_manager，将默认使用的分配器修改为伙伴系统分配器就可以在ucore中实现伙伴系统了。

# 3. 实验总结

编译并运行代码的命令如下：

```
make

make qemu
```

则可以得到如下显示界面

```
chenyu$ make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002c (phys)
  etext  0xc010537f (phys)
  edata  0xc01169b8 (phys)
  end    0xc01178dc (phys)
Kernel executable memory footprint: 95KB
memory managment: default_pmm_manager
e820map:
  memory: 0009f400, [00000000, 0009f3ff], type = 1.
  memory: 00000c00, [0009f400, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efd000, [00100000, 07ffcfff], type = 1.
  memory: 00003000, [07ffd000, 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
……
```

通过上图，我们可以看到ucore在显示其entry（入口地址）、etext（代码段截止处地址）、edata（数据段截止处地址）、和end（ucore截止处地址）的值后，探测出计算机系统中的物理内存的布局（e820map下的显示内容）。接下来ucore会以页为最小分配单位实现一个简单的内存分配管理，完成二级页表的建立，进入分页模式，执行各种我们设置的检查，最后显示ucore建立好的二级页表内容，并在分页模式下响应时钟中断。

# 参考答案对比

- 练习1
  `default_init`函数实现未在原基础上做修改；
  `default_init_memmap`函数在原基础上做的修改只是按照注释要求使用了List_add_before而不是list_add，但由于初始构建空闲列表时只有两个结点、顺序无关，所以实质上效果一样。和参考答案一致。
  `default_alloc_pages`函数在原基础上，在找到第一个合适的空闲块并拆分新空闲块的实现上，虽然基本原理与参考一致，但执行思路不同；此外，似乎不如参考答案的实现更加简洁。
  `default_free_pages`函数在原基础上只修改了将空闲块添加到空闲列表时的方式，保证了空闲块按照地址从小到达分步在空闲列表中。采用的循环检查方式和比较方式均不同于答案，而答案中在对空闲块的大小进行比较时明显更加严谨一些，而且能够排除更多的潜在错误情况。
- 练习2
  与参考答案相比，使用的临时变量较多，不是十分简洁；判断是否分配页的时候，不如参考答案精简，对页目录项的赋值处理和返回指针的方式较为繁琐。
- 练习3
  释放部分原本没有考虑到对引用的处理和辨析，所以最后借鉴了参考报告的思路（详见后续参考文献），基本和参考答案一致。

# 重要知识点和对应原理

- 实验中的重要知识点
  连续内存管理机制
  物理内存分配算法具体实现
  实现双向链表的数据结构
  利用函数指针和结构体近似面向对象功能
  段机制与页机制相关数据结构和操作方法
  虚拟地址到物理地址的映射和转换
- 对应的OS原理知识点
  连续空间分配算法
  分段机制、分页机制和多级页表
  虚拟地址空间到物理地址空间的映射关系

## 二者关系

本实验设计的知识是对OS原理的具体实现，在细节上非常复杂。

## 未对应的知识点

页交换和页分配机制
TLB快速缓存实现方法
页中断处理的详细软件机制
虚存地址空间实现方法
操作系统代码的映射关系和内核栈的具体实现