[1]

操作系统通过BIOS中断调用了解整个计算机系统中的物理内存分布,而BIOS中断调用必须在实模式下进行，所以在bootloader进入保护模式之前完成这部分工作。

[2]

BIOS通过**系统内存映射地址描述符**（Address Range Descriptor）格式来表示系统物理内存布局

- 系统内存块基地址 8字节
- 系统内存大小 8字节
- 内存类型 4字节

[3]

INT 15h BIOS中断

调用的参数

```
eax: e820h int 15中断调用的参数
edx: SMAP 一个签名
ebx: 如果是第一次调用或内存区域扫描完毕，则为0；否则，存放上次调用之后的计数值
ecx: 保存地址范围描述符的内存大小，应该大于等于20字节
```

返回值

```
cflags的CF位：中断执行成功不置位，否则置位
eax: SMAP
es:di: 指向保存地址范围描述符的缓冲区，此时缓冲区内的数据已由BIOS填写完毕
ebx: 下一个地址范围描述符的计数地址
ecx: 返回BIOS向es:di处写的地址范围描述符的字节大小
ah: 失败时保存出错代码
```

[4]

保存**地址范围描述符结构**的缓冲区

```
struct e820map {
    int nr_map;

    //地址范围描述符
```

```
    struct {
                long long addr;
                long long size;
                long type;
            } map[E820MAX];
};
```

e820map 的起始地址 0x8000是不变的， 每次变化的是内部的 map索引，di每次加20指向下一个地址描述符结构，实现填充

[5]

`xorl %ebx,%ebx` 的位置放在start_probe之前进行初始化,之后随它翻滚不是0即可，若为0表示结束

```
int $0x15
jnc cont
返回的应该是本次探测是否成功，而不知道下次是否成功，
若此次成功，所以此时ebx必定不为0，进入cont后cmpl无用，
之后进入下一次探测，这一次探测失败，不进入cont
movw $12345 ,0x8000
jmp finish_probe
结束，这样看 cmpl $0,%ebx似乎没有用到啊
把如下两行改写
cmpl $0,%ebx   ---> #cmpl $0,%ebx 注释掉
jnz start_probe ---> jmp start_probe 直接跳转
结果，，，qemu死了,,,上面的推断有问题


好的，我们换一种方式
jnc cont  ---> jmp cont
不从标志位判断结束，而用ebx判断
正常执行，这说明程序每次都进入cont
jnc cont 通过标志位的判断存在冗余

问题应该出在这里：ebx 在内存区域扫描完毕之后自动置为0，探测实际上是一种扫描，即有记忆性
```

[6]

上述的代码执行完，在0x8000地址保存了内存分布信息，准确说是 0x8004，填充了e820map。这部分信息将交由ucore的`page_init`来完成对整个机器中物理内存的总体管理

[7] 链接地址与物理地址

OS 的链接地址在kernel.ld中设置好了，是一个虚地址；而加载地址在bootmain函数中指定,是一个物理地址

`ph->p_va & 0xFFFFFF`

[8] 两行代码引发的血案

- BSS段 存放程序中未初始化的全局变量，属于静态内存分配
- 数据段 存放程序中已初始化的全局变量，属于静态内存分配
- 代码段 存放代码的内存区域，程序运行前大小已经确定

- data 代码段结束，数据段开始
- edata 数据段结束
- bss 数据段结束，BSS段起始地址
- end BSS段的结束地址

```
extern char edata[],end[]
memset(edata,0,end-edata)
```

> memset的作用是将一块内存的内容全部设置为指定的值

所以这里的含义应该是将 BSS段清零

> 再来说说BSS，早期的计算机存储设备很贵，而很多时候数据段里的全局变量都是0或者没有初始值，存储这么多0到目标文件是没有必要的，所以为了节约空间，在生成目标文件的时候，就把没有初始值的数据段变量都放大BSS段里，这样缩小了目标文件的体积。当目标文件被载入时，加载器负责把BSS段清零。（似乎突然明白C++里未初始化的全局变量为0的原因）

[9]神奇的align

---from Stack Overflow

Input:

```
.byte 1
.align 16
sym: .byte 2
```

Output: `sym` was moved to byte 16

```
0000000000000000 <a-0x10>:
   0:   01 0f                   add    %ecx,(%rdi)
   2:   1f                      (bad)
   3:   44 00 00                add    %r8b,(%rax)
   6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
   d:   00 00 00

0000000000000010 <sym>:
  10:   02                      .byte 0x2
```

Input:

```
.skip 5
.align 4
sym: .byte 2
```

Output: `sym` was moved to byte 8

```
0000000000000000 <sym-0x8>:
   0:   00 00                   add    %al,(%rax)
   2:   00 00                   add    %al,(%rax)
   4:   00 0f                   add    %cl,(%rdi)
   6:   1f                      (bad)
    ...

0000000000000008 <sym>:
   8:   02                      .byte 0x2
```

[10]

`kern_entry` 函数的主要任务是为执行 `kern_init` 建立一个良好的C语言运行环境（**设置堆栈**），而且临时建立一个**段映射关系**，为之后建立分页机制的过程做一个准备

临时的段映射

```
lgdt REALLOC(__gdtdesc)
movl $KERNEL_DS,%eax
movw %ax,%ds
movw %ax,%es
movw %ax,%ss
```

设置堆栈

```
movl $0x0,%ebp
movl $bootstacktop,%esp
call kern_init
```

[11]

- 探测物理内存资源
- 以固定页面大小来划分整个物理内存空间
- 设定状态 free used reserved
- 建立页表，启动分页机制
- MMU把预先建立好的页表中的页表项读入TLB

- 根据页表项描述的虚拟页与物理页帧的对应关系完成CPU对内存的读、写和执行操作

[12] 翻译注释

```
void
pmm_init(void)
{
    //初始化物理内存管理器框架pmm_manager
    init_pmm_manager();

    //探测物理内存空间，使用 pmm->init_memmap 建立空闲的page链表
    page_init();

    //检查物理内存页分配算法
    check_alloc_page();

    //建立一个临时二级页表
    boot_pgdir=boot_alloc_page();
    memset(boot_pgdir,0,PGSIZE)''
    boot_cr3=PADDR(boot_pgsize);
    check_pgdir();
    static_assert(KERNBASE%PTSIZE==0&&KERNTOP%PTSIZE==0);

    //建立一一映射关系的二级页表
    boot_pgdir[PDX(VPT)]=PADDR(boot_pgdir)|PTE_P|PTE_W;
    boot_map_segment(boot_pgdir,KERNBASE,KMEMSIZE,0,PTE_W);//完成页表和页表项的建立
    boot_pgdir[0]=boot_pgdir[PDX(KERNBASE)];

    //使能分页机制
    enable_paging();

    //重新设置全局段描述符表（第三次,也是最后一次）
    //set kernel stack in TSS,setup TSS ingdt,load TSS
    gdt_init();

    //取消临时二级页表
    boot_pgdir[0]=0;

    //检查页表建立是否正确
    check_boot_pgdir();

    //通过自映射机制完成页表的打印输出
    print_pgdir();
}
```

get_pte 函数完成虚实映射

[13]

```
struct Page {
    int ref; //页帧被页表的引用计数
    uint32_t flags; //物理页状态
    unsigned int property;//连续内存空闲块的大小（空闲页个数）头一页使用这个变量
    list_entry_t page_link;//头一页使用，连接其他连续内存空闲块
};
```

PG_reserved 1表示被保留

PG_property

1： 该页是空闲块的头页，并且可以被分配 0： 如果该页是头页，表示已经被分配 **如果不是头页，那么就不是头页**

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

[14]

page_init()

- 0x8000+KERNBASE
- end-1 小学数学题
- E820_ARM
- 探测过程
- maxpa 为探测到的最大的结束地址，由于起始地址是0x00000000,所以实际上它表示的是大小，并且这一大块内存的最后一块一定是空闲的，将它和KMEMSIZE比较，取较小的
- ROUNDUP(a,n) 向上补a,使得a整除n end以上的空间没有被使用，可以向上取整
- set_bit(PG_reserved,&((page)->flags))
  - 将flags中的PG_resereved位 置1
  - 也就是说开始阶段探测到的所有内存，先置为reserved
- 看到

  ```
  uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) * npage);
  ```

  突然意识到Page这个结构也是需要空间存储的，那么进而的问题是，Page如何与物理页对应的呢？ 似乎在init_memmap()中

[15]

计算机系统内存布局与e820map比较

[16]

```
struct Page *p=le2page(le,page_link)
```

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

```
//成员变量的地址大于节后开始的地址，减去偏移量即得到结构开始的地址
#define to_struct(ptr,type,member) ((type*)((char *)(ptr)-offsetof(type,member)))
```

```
//得到成员变量member在结构中的偏移量
#define offsetof(type,member) ((size_t)(&((type*)0)->member))
```

这一通操作，就干了一件事，根据 le 这个在Page中的page_link找到对应的Page，追的我好辛苦

[17]

重写default_pmm为FFMA

需要考虑的就是 插入的位置根据地址大小确定

[18]

32位机器上

- intptr_t 表示 int
- uintptr_t 表示 unsigned int

64位机器上

- intptr_t 表示 long int
- uintptr_t 表示 unsigned long int

[19]

一级页表(Page Directory Table)的起始物理地址存放在 cr3 寄存器中，这个地址必须是页对齐的地址，低12位为0

```
boot_cr3=PADDR(boot_pgdir)
```

[20]

写到[21]发现需要一些准备工作

mmu.h

```
//段描述符结构
struct segdesc {
    unsigned sd_lim_15_0 : 16;      // low bits of segment limit
    unsigned sd_base_15_0 : 16;     // low bits of segment base address
    unsigned sd_base_23_16 : 8;     // middle bits of segment base address
    unsigned sd_type : 4;           // segment type (see STS_ constants)
    unsigned sd_s : 1;              // 0 = system, 1 = application
    unsigned sd_dpl : 2;            // descriptor Privilege Level
    unsigned sd_p : 1;              // present
    unsigned sd_lim_19_16 : 4;      // high bits of segment limit
    unsigned sd_avl : 1;            // unused (available for software use)
    unsigned sd_rsv1 : 1;           // reserved
    unsigned sd_db : 1;             // 0 = 16-bit segment, 1 = 32-bit segment
    unsigned sd_g : 1;              // granularity: limit scaled by 4K when set
    unsigned sd_base_31_24 : 8;     // high bits of segment base address
};

//以汇编的形式构造段描述符
#define SEG_ASM(type,base,lim)                                  \
    //开辟响应大小的空间，存放后面的变量
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);          \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)),             \
        (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)

//和上面的描述符结构对应
#define SEG(type, base, lim, dpl)                           \
    (struct segdesc) {                                      \
        ((lim) >> 12) & 0xffff, (base) & 0xffff,            \
        ((base) >> 16) & 0xff, type, 1, dpl, 1,             \
        (unsigned)(lim) >> 28, 0, 0, 1, 1,                  \
        (unsigned) (base) >> 24                             \
    }
```

[21]

两次实验bootloader都把 ucore放在了起始物理地址为0x100000的物理地址，而lab1中通过ld工具形成的ucore的起始地址从0x100000开始，lab2通过ld工具形成的ucore的起始地址从0xC0100000开始

第一阶段

bootloader阶段

```
virt addr = linear addr = phy addr
```

```
lgdt gdtdesc
...
gdt:
    SEG_NULLASM                                 # null seg
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)       # code seg for bootloader and kernel
    SEG_ASM(STA_W, 0x0, 0xffffffff)             # data seg for bootloader and kernel

gdtdesc:
    .word 0x17                                  # sizeof(gdt) - 1
    .long gdt                                   # address gdt
```

第二阶段

kern_entry 到 enable_page 更新了段映射，还没有启动页映射

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

```
lgdt REALLOC(__gdtdesc)
...
_gdt:
//可以看到与上一个相比 base发生了变化，从0x0变为 -KERNBASE
    SEG NULL
    SEG_ASM(STA_X|STA_R,-KERNBASE,0xFFFFFFFF) #code
    SEG_ASM(STA_W,-KERNBASE,0xFFFFFFFF) # data
_gdtdesc:
    .word 0x17
    .long REALLOC(__gdt)
```

第三阶段

从 enable_page 到 gdt_init 启动了页映射机制，但没有第三次更新段映射

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

第四阶段

从 gdt_init() 开始，形成新的段页式映射机制，取消临时映射关系

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

```
lgdt(&gdt_pd);

static struct pseudodesc gdt_pd =
{
    sizeof(gdt) - 1, (uintptr_t)gdt
};

static struct segdesc gdt[] =
{
    SEG_NULL,
    //这里 base又变为0，即对等映射
    [SEG_KTEXT] = SEG(STA_X | STA_R, 0x0, 0xFFFFFFFF, DPL_KERNEL),
    [SEG_KDATA] = SEG(STA_W, 0x0, 0xFFFFFFFF, DPL_KERNEL),
    [SEG_UTEXT] = SEG(STA_X | STA_R, 0x0, 0xFFFFFFFF, DPL_USER),
    [SEG_UDATA] = SEG(STA_W, 0x0, 0xFFFFFFFF, DPL_USER),
    [SEG_TSS]   = SEG_NULL,
};

static inline void
lgdt(struct pseudodesc *pd)
{
    asm volatile ("lgdt (%0)" :: "r" (pd));
    asm volatile ("movw %%ax, %%gs" :: "a" (USER_DS));
    asm volatile ("movw %%ax, %%fs" :: "a" (USER_DS));
    asm volatile ("movw %%ax, %%es" :: "a" (KERNEL_DS));
    asm volatile ("movw %%ax, %%ds" :: "a" (KERNEL_DS));
    asm volatile ("movw %%ax, %%ss" :: "a" (KERNEL_DS));
    // reload cs
    asm volatile ("ljmp %0, $1f\n 1:\n" :: "i" (KERNEL_CS));
}
```

总结一下，为了建立页映射，线性地址和物理地址相差base，需要借用段映射，先使虚拟地址和线性地址相差base，将利用这个映射填写页表，在过程中就会有相差出两个base的时刻，这是创建一个临时映射，最后将段映射还原为对等映射，取消临时映射，线性地址和物理地址相差一个base,物理地址等于线性地址

[22]

利用分配算法分配获得一个物理页

```
boot_pgdir=boot_alloc_page();
```

将物理页地址转化为 虚拟地址最终给到 boot_pgdir

```
 return page2kva(p)
```

pmm.h

page2kva(p)

```
 return KADDR(page2pa(page))
```

先将物理页转化为 物理地址

```
static inline uintptr_t
page2pa(struct Page* page)
{
    return page2ppn(page)<<PGSHIFT;

}
```

```
static inline uintptr_t
page2ppn(struct Page *page)
{
    return page-pages  //pages=(struct Page*)ROUNDUP((void*)end,PGSIZE)
}
```

pages应该是第一个page结构（不是物理页）的起始地址，并且是BSS段向上取整的结果，当前的page的物理地址减去起始地址，将这个偏移左移12位，即为物理页的地址

接下来需要将物理页的地址转换为虚拟地址

```
KADDR(pa)
简单说就是加上 KERNBASE
```

最后将 page2kva(p) 传给 boot_pgdir 注意到这个变量类型为 pde_t*(uintptr_t*) ,而不是Page*,所以上述对于page结构物理地址和物理页地址的区分是正确的,这样也解决了[14]中的疑问

[23]

```
boot_pgdir[PDX(VPT)]=PADDR(boot_pgdir)|PTE_P|PTE_W;

//反正boot_pgdir物理地址低12位一定是0，那么不如用来表示些其他信息
//mmu.h
//PTE_P  0x001 表示物理页存在
//PTE_W  0x002 表示物理内存页可写    权限设置问题

#define PDX(la)((((uintptr_t)(la))>>PDXSHIFT)&0x3FF)
右移22位 取高十位作为 页目录表boot_pgdir中的索引

现在的问题是VPT是个啥
#define VPT 0xFAC00000
```

好吧，又到了激动人心的翻译注释环节 ？？？
Virtual page table. Entry PDX[VPT] in the PD (Page Directory) contains
a pointer to the page directory itself, thereby turning the PD into a page
table, which maps all the PTEs (Page Table Entry) containing the page mappings
for the entire virtual address space into that 4 Meg region starting at VPT.

原来跟后面的自映射机制有关，将页目录表作为页表把物理地址写入到自己这张表中，
构造一个VPT，放到VPT对应索引的位置，为后面打印页目录项和页表项带来遍历，
不需要进行多次物理地址到虚拟地址的转换

[24]

完成线性地址和物理地址的映射，相差KERNBASE

```
boot_map_segment(boot_pgdir, KERNBASE, KMEMSIZE, 0, PTE_W);
```

由此进入 `get_pte` 得到 线性地址 对应页表表项的虚拟地址，便于之后对其进行设置 如果包含这个表项的页表不存在，那就分配一个页给他，这就涉及到了目录表的表项 看到 `help commont` 内心有些崩溃，这些解释为什么不放在前面，放这来解释，然而都用了八百遍了。。。

```
return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];
//get_pte中最重要的一句话了，确定线性地址在页表中应该放的位置，
//然而有可能还没有该页表，所以在return之前还有其他操作，
//PTX取到了线性地址中间相应位置的10位（21~12）是页表中的偏移
//前面根据页目录表 对应表项的内容 找到页表的基址，基址加偏移即为相应的页表表项 再取地址作为结果返回
```

[25]

```
static void
enable_paging(void)
{
    //通过 lcr3 指令把页目录表的起始地址存入 CR3 寄存器
    lcr3(boot_cr3);
    //通过lcr0指令 cr0 中的CR0_PG 标志位设置为上
    uint32_t cr0=rcr0(); //内联汇编，一个mov操作
    //这些都是个嘛啊?
    //muu.h给出了定义 控制寄存器的标志位
    // 之前遇到过 CR0_PE 保护模式使能
    // 这里主要关注 CR0_PG 分页使能
    cr0|=CR0_PE|CR0_PG|CR0_AM|CR0_WP|CR0_NE|CR0_TS|CR0_EM|CR0_MP;
    cr0&=~(CR0_TS|CR0_EM);//把这两位置0,那上面或操作加上这俩哥也没什么意义哇，
    //考虑到可能跟后面的实验有关，然而看了实验8的代码还是酱紫，那把上面或操作的俩哥们去掉试一试，也能正常执行
```

```
    lcr0(cr0);//取出来，整了一通，再放回去，还是个内联汇编的mov操作
}
```

Wiki上是这样说的

EM Emulation

> If set, no x87 floating-point unit present, if clear, x87 FPU present

TS Task switched

> Allows saving x87 task context upon a task switch only after x87 instruction used

所以 x87是个啥

> 8087 是由 Intel 所设计的第一个数学 辅助处理器，并且它是建造来与 Intel 8088 和 8086 微处理器成对工作。它是 x87 家族中的第一个，8087 的目的是用来加速应用程序有关 浮点 。

应该跟进行浮点运算有关

那么上述操作应该是说，存在x87FPU，然而保存任务上下文的操作不只适用于x87指令