Lab2.操作系统物理内存管理系统

遇到的错误

- 在lab2中执行make报错： 'for' loop initial declaration are only allowed in C99 mode。

```
pcy@ubuntu:~/Desktop/labcodes/lab2$ make
+ cc kern/trap/trap.c
kern/trap/trap.c: In function ‘idt_init’:
kern/trap/trap.c:51:9: error: ‘for’ loop initial declarations are only allowed i
n C99 mode
         for(int i=0;i<num;++i)
         ^
kern/trap/trap.c:51:9: note: use option -std=c99 or -std=gnu99 to compile your c
ode
Makefile:143: recipe for target 'obj/kern/trap/trap.o' failed
make: *** [obj/kern/trap/trap.o] Error 1
```

是因为在gcc中直接在for循环中初始化了增量：

```
for(int i=0; i<len; i++) { ... }
```

这语法在gcc中是错误的，必须先先定义i变量：

```
int i;
for(i=0;i<len;i++){ ... }
```

可是我在lab1里执行make就没报错啊.....是不是因为我做完lab1之后对gcc进行了降级，所以一些标准设置就改变了.....

- 写完所有函数之后执行make qemu，发现default_free_pages函数中的PageReserved(base)判断语句报错：

```
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.
kernel panic at kern/mm/default_pmm.c:127:
    assertion failed: !PageReserved(p) && !PageProperty(p)
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

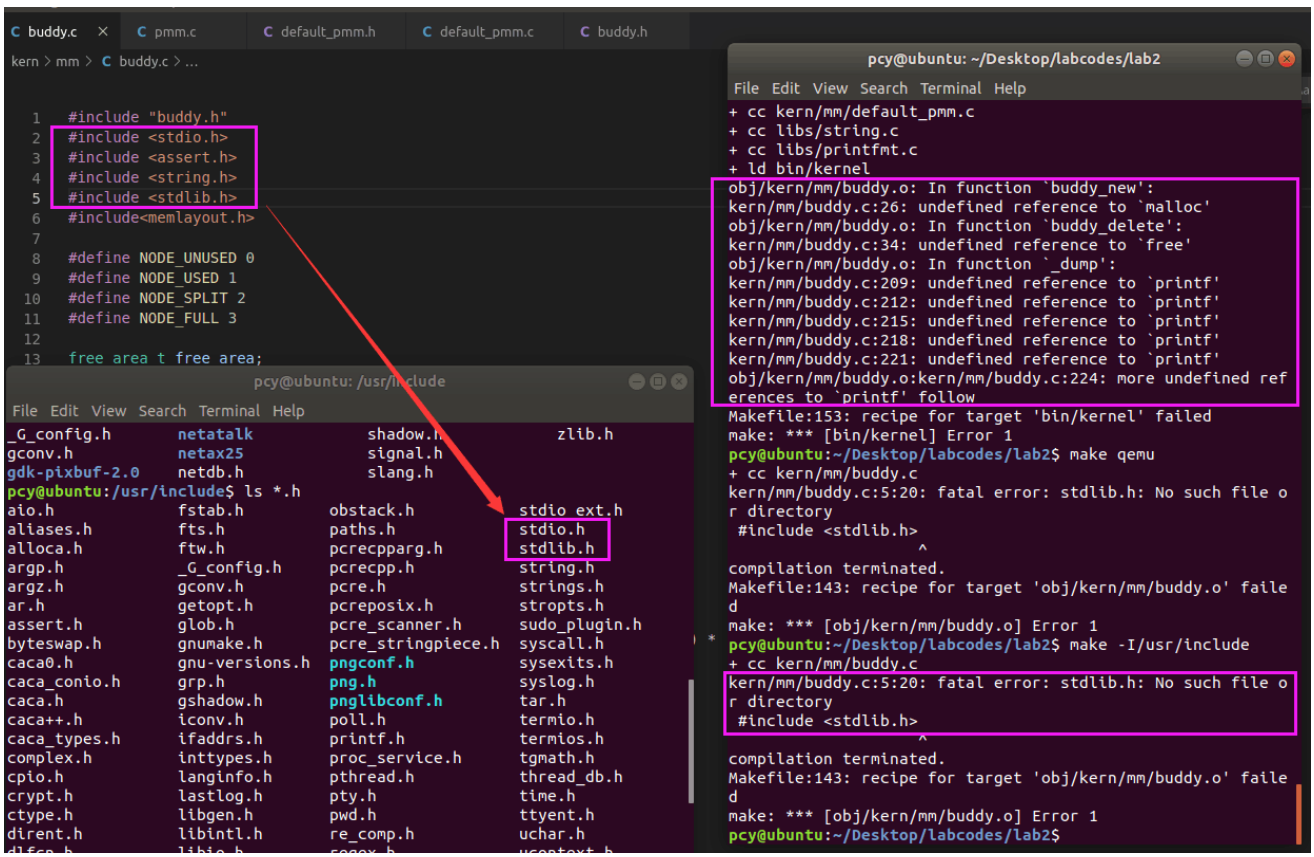
在default_alloc_pages函数中写过SetPageReserved(p)，在default_free_pages函数中写了assert(PageReserved(base))，查看PageReserved(base)相关的定义：

```
#define PG_reserved 0// if this bit=1: the Page is reserved for kernel, cannot be used
in alloc/free_pages; otherwise, this bit=0
#define PageReserved(page) test_bit(PG_reserved, &((page)->flags))
#define SetPageReserved(page) set_bit(PG_reserved, &((page)->flags))
```

SetPageReserved把物理地址对应的Page结构中的flags标志设置为PG_reserved ，表示这些页已经被使用了，将来不能被用于分配。

- 数据类型写错了，list_entry_t 和 Page 写混了，导致make时有warning，但当时忽略了，导致后来的错误。
- 犯的最大的一个错误，就是混淆了list_entry_t的意义，链表的每一个元素中包含的是一个块，而不是一个页，最开始迷迷糊糊的，没有真正弄清这一点，导致default_init_memmap、default_alloc_pages、default_free_pages全都写错了，虽然觉得哪里不太对，但是make qemu和make grade时居然完全没报出任何问题，所以就没在意……是在最后在对比答案时，才发现自己理解错了。
- 在写Buddy System的时候引用头文件stdlib时报错：stdlib.h: No such file or directory，查看了/usr/include下面的文件，stdlib.h是存在的，并且它上面的stdio.h就能正常使用，网上提到的方法都试过了，都没起作用，所以我也不清楚这是什么原因。

  用一个简单的调用stdlib.h的c文件测试了一下，g++可以正常编译，但是gcc无法正常编译，在链接时会报错，所以我想着是不是需要在makefile文件中将gcc的相关部分改成g++，但是我还没有尝试。换了另一种方法写的Buddy System。



（下面是实验过程中做的笔记，没有什么新的东西了）

实验目的

- 理解基于段页式内存地址的转换机制

- 理解页表的建立和使用方法
- 理解物理内存的管理方法

实验内容

1. 如何发现系统中的物理内存
2. 如何建立对物理内存的初步管理，即连续物理内存管理
3. 页表相关的操作，即如何建立页表来实现虚拟内存到物理内存之间的映射，对段页式内存管理机制有一个比较全面的了解。
4. 针对 cache 的优化等

练习0：填写已有实验

用meld进行代码合并还是很方便的。主要修改的文件有：kern/init/init.c、kern/trap/trap.c、kern/debug/kdebug.c、

lab2相比于lab1两个方面的扩展。首先，bootloader的工作有增加，在bootloader中，完成了对物理内存资源的探测工作，让ucore kernel在后续执行中能够基于bootloader探测出的物理内存情况进行物理内存管理初始化工作。其次，bootloader不像lab1那样，直接调用kern_init函数，而是先调用位于lab2/kern/init/entry.S中的kern_entry函数。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），而且临时建立了一个段映射关系，为之后建立分页机制的过程做一个准备（细节在3.5小节有进一步阐述）。完成这些工作后，才调用kern_init函数。

kern_init函数在完成一些输出并对lab1实验结果的检查后，将进入物理内存管理初始化的工作，即调用pmm_init函数完成物理内存的管理，这也是我们lab2的内容。接着是执行中断和异常相关的初始化工作，即调用pic_init函数和idt_init函数等，这些工作与lab1的中断异常初始化工作的内容是相同的。

练习1：实现 first-fit 连续物理内存分配算法

First-Fit Mem Alloc (FFMA)，在实现回收函数时，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。

需要修改default_pmm.c中的default_init_memmap，default_alloc_pages， default_free_pages

对物理页的定义在memlayout.h中：

```
struct Page {
    int ref;// 表示此物理页被页表的引用记数，即映射此物理页的虚拟页个数。一旦某页表中有一个页表项设置了虚拟页到这个Page管理的物理页的映射关系，就会把Page的ref加一。反之，若是解除，那就减一。
    uint32_t flags;// 表示此物理页的状态标记，有两个标志位，bit 0表示此页是否被保留（reserved），如果是被保留的页，则bit 0会设置为1，且不能放到空闲页链表中，即这样的页不是空闲页，不能动态分配与释放。比如目前内核代码占用的空间就属于这样“被保留”的页。bit 1表示此页是否是free的，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;// 记录某连续内存空闲块的大小，用到此成员变量的这个Page一定是连续内存块的开始地址（第一页的地址）。
    list_entry_t page_link;// 是便于把多个连续内存空闲块链接在一起的双向链表指针
};
```

在kern/mm/pmm.h中有许多将物理页转换为其它数据类型（如物理地址）的函数。

内存管理相关的总体控制函数是pmm_init函数，它完成的主要工作包括：

1. 初始化物理内存页管理器框架pmm_manager；
2. 建立空闲的page链表，这样就可以分配以页（4KB）为单位的空闲内存了；
3. 检查物理内存页分配算法；
4. 为确保切换到分页机制后，代码能够正常执行，先建立一个临时二级页表；
5. 建立一一映射关系的二级页表；
6. 使能分页机制；
7. 从新设置全局段描述符表；
8. 取消临时二级页表；
9. 检查页表建立是否正确；
10. 通过自映射机制完成页表的打印输出

在libs/list.h中有结构体列表的定义，是一个双链表，需要熟悉其中list_init, list_add(list_add_after), list_add_before, list_del, list_next, list_prev的使用。双向链表的定义：

```
struct list_entry {
    struct list_entry *prev, *next; //父节点，子节点
};
```

结构体free_area_t用于空闲内存块的管理，定义在memlayout.h中：

```
typedef struct {
    list_entry_t free_list;// 一个双向链表，负责管理所有的连续内存空闲块，便于分配和释放。
    unsigned int nr_free;// 空闲块的个数
} free_area_t;
```

default_init：创建一个空的双向链表，初始化空闲列表free_list，将空闲内存块的总数nr_free置为0，已经实现好了，不用再修改。

```
static void
default_init(void) {
    list_init(&free_list);
    nr_free = 0;
}
```

default_init_memmap(addr_base, page_number)：初始化空闲块。调用过程：kern_init --> pmm_init-->page_init-->init_memmap--> pmm_manager->init_memmap。函数流程：首先初始化这个空闲块中的每一个页，初始化包括：p->flags和PG_property置为0，表明当前页可用。如果当前页空闲，并且不是第一个空闲块的第一个页，那么p->property应该置为0。如果当前页空闲，并且是第一个空闲块的第一个页，那么p->property置为块的总数。p空闲且未被引用时p->ref应该为0。使用p->page_link将当前物理页连接到free_list(例如list_add_before(&free_list, &(p->page_link));)。最后汇总空闲内存块的个数nr_free+=n。

```
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);//加一些 assert函数，在有错误出现时，能够迅速发现。
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));
        p->flags = p->property = 0;
```

```
            set_page_ref(p, 0);
        }
        base->property = n;
        SetPageProperty(base);
        nr_free += n;
        list_add_before(&free_list, &(base->page_link));
    }
```

firstfit需要从空闲链表头开始查找最小的地址，通过list_next找到下一个空闲块元素，通过le2page宏可以由链表元素获得对应的Page指针p。通过p->property可以了解此空闲块的大小。如果>=n，这就找到了！如果<n，则list_next，继续查找。直到list_next== &free_list，这表示找完了一遍了。找到后，就要从新组织空闲块，然后把找到的page返回。

default_alloc_pages()：在free_list中查找第一个可用的空闲块，改变空闲块的大小，返回分配的块的地址。在while循环中检查物理页的p->property（块的大小）是否大于等于n，如果找到了这样的块，那么前n也就可以被分配，令PG_reserved =1, PG_property =0，从free_list中取出；如果p->property大于n，需要计算剩余块的大小。重新计算nr_free的数量（空闲块总数），返回物理页。如果找不到，返回NULL。

```
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;
    // TODO: optimize (next-fit)
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add_after(&(page->page_link), &(p->page_link));
        }
        list_del(&(page->page_link));
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

default_free_pages()：将page放入free_list中，可能需要合并成更大的块。首先根据基地址找到合适的位置，插入链表，重置页的参数，如p->ref, p->flags。最后合并块。

```
static void
```

```
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    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);
    list_entry_t *le = list_next(&free_list);
    while (le != &free_list) {
        p = le2page(le, page_link);
        le = list_next(le);
        // TODO: optimize
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
    nr_free += n;
    le = list_next(&free_list);
    while (le != &free_list) {
        p = le2page(le, page_link);
        if (base + base->property <= p) {
            assert(base + base->property != p);
            break;
        }
        le = list_next(le);
    }
    list_add_before(le, &(base->page_link));
}
```

问题：first fit算法是否有进一步的改进空间?

有序链表插入：一个刚被释放的内存块，如果它的邻接空间都是空闲的，那么就不需要进行线性时间复杂度的链表插入操作，而是直接并入邻接空间，时间复杂度为常数。为了判断邻接空间是否为空闲状态，空闲块的信息除了保存在第一个页面之外，还需要在最后一页保存信息，这样新的空闲块只需要检查邻接的两个页面就能判断邻接空间块的状态。

练习2：实现寻找虚拟地址对应的页表项

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。逻辑地址通过段式管理的地址映射可以得到线性地址，线性地址通过页式管理的地址映射得到物理地址。

页表是一个元素为页表条目（Page Table Entry, *PTE*）的集合，每个虚拟页在页表中一个固定偏移量的位置上都有一个PTE。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。

kern/mm/pmm.c中的get_pte(pgdir, la, create)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。

```
pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    /* LAB2 EXERCISE 2: YOUR CODE
     *   PDX(la)返回虚拟地址la的页目录索引
     *   KADDR(pa)返回物理地址pa相关的内核虚拟地址
     *   set_page_ref(page,1)设置此页被引用一次
     *   page2pa(page)得到page管理的那一页的物理地址
     *   struct Page * alloc_page()分配一页出来
     *   memset(void *s, char c, size_t n) :设置s指向地址的前面n个字节为字节‘c’
     * DEFINES:
     *   PTE_P           0x001                   // 表示物理内存页存在
     *   PTE_W           0x002                   // 表示物理内存页内容可写
     *   PTE_U           0x004                   // 表示可以读取对应地址的物理内存页内容
     *
     * typedef unsigned int uint32_t;
     * typedef uint32_t uintptr_t;  //uintptr_t表示为线性地址，由于段式管理只做直接映射，所以它也是逻辑地址。
     * typedef uintptr_t pte_t;
     * typedef uintptr_t pde_t;
     */
    //pgdir是一级页表本身,pde_t是一级页表的表项，pte_t表示二级页表的表项
    //pgdir给出页表起始地址。通过查找这个页表，我们可以得到一级页表项(二级页表的入口地址)。
    pde_t *pdep = &pgdir[PDX(la)];//根据虚地址的高十位查询页目录，找到页表项的pdep
    if (!(*pdep & PTE_P)) {
        //如果在查找二级页表项时，发现对应的二级页表不存在，则需要根据create参数的值来处理是否创建新的二级页表。
        struct  Page* page = alloc_page();
        if(!create||page==NULL){
            return NULL;
        }
        set_page_ref(page,1);//引用次数加一
        uintptr_t pa = page2pa(page); //获取页的线性地址
        memset(KADDR(pa),0,PGSIZE);//初始化,新申请的页全设为零，因为这个页所代表的虚拟地址都没有被映射。
        *pdep=pa|PTE_U|PTE_W|PTE_P;//设置控制位
    }
    //KADDR(PDE_ADDR(*pdep)):这部分是由页目录项地址得到关联的页表物理地址， 再转成虚拟地址。
    //PTX(la): 返回虚拟地址la的页表项索引
    //最后返回的是虚拟地址la对应的页表项入口地址
    return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];
}
```

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

页目录项包括一些几部分：

名称	地址	ucore中的对应
Page Table 4KB Aligned Address	31 downto 12	对应的页表地址
Avail	11 downto 9	PTE_AVAIL
Ignored	8	
Page Size	7	PTE_PS
0	6	PTE_MBZ
Accessed	5	PTE_A
Cache Disabled	4	PTE_PCD
Write Through	3	PTE_PWT
User/Supervisor	2	PTE_U
Read/Write	1	PTE_W
Present	0	PTE_P

页表项包括一些几部分：

名称	地址	ucore中的对应
Physical Page Address	31 downto 12	对应的物理地址高20位
Avail	11 downto 9	PTE_AVAIL
Global	8	
0	7	PTE_MBZ
Dirty	6	PTE_D
Accessed	5	PTE_A
Cache Disabled	4	PTE_PCD
Write Through	3	PTE_PWT
User/Supervisor	2	PTE_U
Read/Write	1	PTE_W
Present	0	PTE_P

问题：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?

进行换页操作。首先 CPU 将产生页访问异常的线性地址放到 cr2 寄存器中，然后和普通的中断一样，保护现场，将寄存器的值压入栈中，压入 error_code，中断服务例程将外存的数据换到内存中来，最后退出中断，回到进入中断前的状态。

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

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。取消页表映射过程如下：

- 将物理页的引用数目减一，如果变为零，那么释放页面；
- 将页目录项清零；
- 刷新TLB

kern/mm/pmm.c中的page_remove_pte函数:

```
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    /* LAB2 EXERCISE 3: YOUR CODE
     *   free_page : free a page
     *   page_ref_dec(page) : 减少该页的引用次数，返回剩下引用次数
     *   tlb_invalidate(pde_t *pgdir, uintptr_t la) : Invalidate a TLB entry, but only
if the page tables being edited are the ones currently in use by the processor.当修改的页表是进程正在使用的那些页表，使之无效。
     *   PTE_P           0x001 // page table/directory entry flags bit : Present
     */
    if (*ptep & PTE_P) {// 页表项存在
        struct Page *page = pte2page(*ptep); //找到页表项
        if(page_ref_dec(page)==0)
        {//判断此页被引用的次数，如果仅仅被引用一次，则这个页也可以被释放。否则，只能释放页表入口
            free_page(page);//释放页
        }
        *ptep=0;//该页目录项清零
        tlb_invalidate(pgdir,la);//flush tlb
    }
}
```

问题：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的对应关系？

页目录项或者页表项中都保存着一个物理页面的地址，对于页目录项，这个物理页面用于保存二级页表，对于页表来说，这个物理页面用于内核或者用户程序。页目录项保存的物理页面地址（即某个页表）以及页表项保存的物理页面地址都对应于Page数组中的某一页。

问题：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？

- 修改链接脚本，将内核起始虚拟地址修改为`0x100000`；

tools/kernel.ld

```
SECTIONS {
    /* Load the kernel at this address: "." means the current address */
    . = 0x100000;
```

- 修改虚拟内存空间起始地址为0；

kern/mm/memlayout.h

```
/* All physical memory mapped at this address */
#define KERNBASE            0x00000000
```

- 注释掉取消0~4M区域内存页映射的代码

kern/mm/pmm.c

```
//disable the map of virtual_addr 0~4M
// boot_pgdir[0] = 0;
```

```
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!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ----------------------
++ setup timer interrupts
100 ticks
End of Test.
kernel panic at kern/trap/trap.c:18:
    EOT: kernel seems ok.
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K> qemu-system-i386: terminating on signal 2

pcy@ubuntu:~/Desktop/labcodes/lab2$ make grade
Check PMM:                  (2.3s)
  -check pmm:                                 OK
  -check page table:                          OK
  -check ticks:                               OK
Total Score: 50/50
```

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，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这个 BIOS 中断，并且把 e820 映 射结构保存在物理地址0x8000处。）。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。

扩展练习Challenge：buddy system（伙伴系统）分配算法

伙伴分配的实质就是一种特殊的**“分离适配”**，即将内存按2的幂进行划分，相当于分离出若干个块大小一致的空闲链表，搜索该链表并给出同需求最佳匹配的大小。其优点是快速搜索合并（O(logN)时间复杂度）以及低外部碎片（最佳适配best-fit）

（我只是大概明白伙伴系统的原理了，但是我不太会写代码，所以在网上找了代码https://github.com/zhenghaoz/ucore/tree/master/lab2，然后修改了一些调用的地方，最后能正确地执行make qemu）

初始化

在Buddy System中，空间块之间的关系形成一颗完全二叉树，对于一颗有着n叶子的完全二叉树来说，所有节点的总数为$2n-1\approx 2n$。也就是说，如果Buddy System的可分配空间为n页的话，那么就需要额外保存2n-1个节点信息。

初始化空闲链表

Buddy System并不需要链表，但是为了在调式的时候方便访问所有空闲空间，还是将所有的空闲空间加入链表中。

确定分配空间大小

假设我们得到了大小为n的空间，我们需要在此基础上建立Buddy System，经过初始化后，Buddy System管理的页数为2^m，那么大小为n的实际空间可能分为两个或者三个部分。

节点信息区：节点信息区可以用来保存每个节点对应子树中可用空间的信息，用于在分配内存的时候便于检查子树中是否有足够的空间来满足请求大小。在32位操作系统中，最大页数不会超过4GB/4KB=1M，所有使用一个32位整数即可表示每个节点的信息。所以节点信息区的大小为$2^m\times 2\times 4=2^{m+3}$字节，每页大小为4KB，内存占用按照页面大小对齐，所以占用$max1,2^{m-9}$页。

虚拟分配区：占用2^m页。

实际分配区：显然实际可以得到的内存大小不大可能刚好等于节点信息区大小+分配空间区大小。如果节点信息区大小+分配空间区大小<=内存大小，那么实际可以分配的区域就等于2^m页。如果节点信息区大小+分配空间区大小>内存大小，那么实际可以分配的区域就等于$n-max1,2^{m-9}$页。

作为操作系统，自然希望实际使用的区域越大越好，不妨分类讨论。

当内存小于等于512页：此时无论如何节点信息都会占用一页，所以提高内存利率的方法就是将实际内存大小减一后向上取整（文中整数意为2的幂）。

当内存大于512页：不难证明，对于内存大小n来说，最佳虚拟分配区大小往往是n向下取整或者向上取整的数值，所以候选项也就是只有两个，所以可以先考虑向下取整。对于$[2^m,2^{m+1}-1]$中的数n，向下取整可以得到2^m：

- 当$n\leq 2^{m-9}+2^m$时，显然已经是最佳值；
- 当$2^{m-9}+2^m<n\leq 2^{m-8}+2^m$时，扩大虚拟分配区导致节点信息区增加却没有使得实际分配区增加，所以当期m还是最佳值；
- 当$n>2^{m-8}+2^m$时，$m+1$可以扩大实际分配区。

初始化节点信息

虚拟分配区可能会大于实际分配区，所以一开始需要将虚拟分配区中没有实际分配区对应的空间标记为已经分配进行屏蔽。另当前区块的虚拟空间大小为v，实际空间大小为r，屏蔽的过程如下：

- 如果$v=r$，将空间初始化为一个空闲空间，屏蔽过程结束；
- 如果$r=0$，将空间初始化为一个已分配空间，屏蔽过程结束；
- 如果$r\leq v/2$，将右半空间初始化为已分配空间，更新$r=r/2$后继续对左半空间进行操作；

- 如果$r > v/2$，将左半空间初始化为空闲空间，更新$r = r - v/2, v = v/2$后继续对左半空间进行操作。

分配过程

Buddy System要求分配空间为2的幂，所以首先将请求的页数向上对齐到2的幂。

接下来从二叉树的根节点（1号节点）开始查找满足要求的节点。对于每次检查的节点：

- 如果子树的最大可用空间小于请求空间，那么分配失败；
- 如果子树的最大可用空间大于等于请求空间，并且总空间大小等于请求空间，说明这个节点对应的空间没有被分割和分配，并且满足请求空间大小，那么分配这个空间；
- 如果子树的最大可用空间大于等于请求空间，并且总空间大小大于请求空间，那么在这个节点的子树中查找：
  - 如果这个节点对应的空间没有被分割过（最大可用空间等于总空间大小），那么分割空间，在左子树（左半部分）继续查找；
  - **如果左子树包含大小等于请求空间的可用空间，那么在左子树中继续查找；**
  - **如果右子树包含大小等于请求空间的可用空间，那么在右子树中继续查找；**
- 如果左子树的最大可用空间大于等于请求空间，那么在左子树中继续查找；
  - 如果右子树的最大可用空间大于等于请求空间，那么在右子树中继续查找。

当一个空间被分配之后，这个空间对应节点的所有父节点的可用空间表都会受到影响，需要自地向上重新更新可用空间信息。

释放过程

Buddy System要求分配空间为2的幂，所以同样首先将请求的页数向上对齐到2的幂。

在进行释放之前，需要确定要释放的空间对应的节点，然后将空间标记为可用。接下来进行自底向上的操作：

- 如果某节点的两个子节点对应的空间都未分割和分配，那么合并这两个空间，形成一个更大的空间；
- 否则，根据子节点的可用空间信息更新父节点的可用空间信息。

指导书中的问题

- 管理页级物理内存空间所需的Page结构的内存空间从哪里开始，占多大空间?

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
```

这样，我们就可以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，所以我们可以把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：

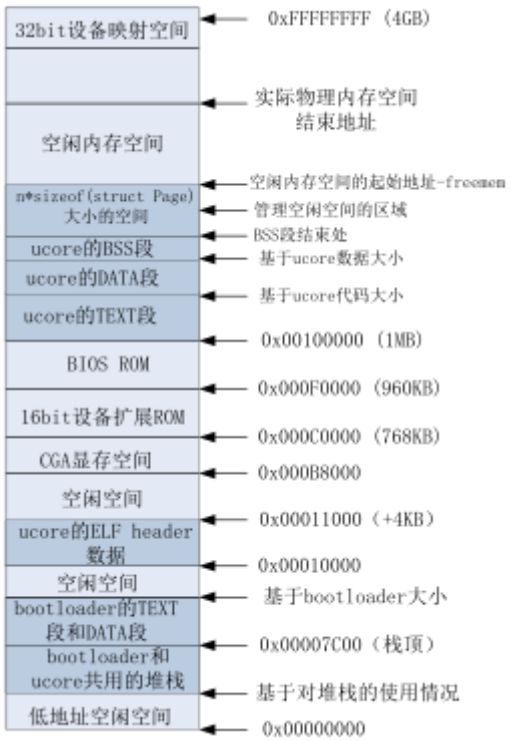
```
pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);
```

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

- 空闲内存空间的起始地址在哪里?

空闲空间起始地址为

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



- 具体的页映射关系是什么?

  在lab1和lab2中，bootloader把ucore都放在了起始物理地址为0x100000的物理内存空间。

  在lab1中，我们已经碰到了简单的段映射，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。

  lab2中，通过几条汇编指令（在kern/init/entry.S中）实现分页机制，主要做了两件事：

  1. 通过 `movl %eax, %cr3` 指令把页目录表的起始地址存入CR3寄存器中；
  2. 通过 `movl %eax, %cr0` 指令把cr0中的CR0_PG标志位设置上。

  执行完这几条指令后，计算机系统进入了分页模式，虚拟地址、线性地址以及物理地址之间的临时映射关系为：

  ```
  lab2 stage 2 before:
      virt addr = linear addr = phy addr # 线性地址在0~4MB之内三者的映射关系
      virt addr = linear addr = phy addr + 0xC0000000 # 线性地址在
  0xC0000000~0xC0000000+4MB之内三者的映射关系
  ```

  pmm_init函数将页目录表项补充完成（从0~4M扩充到0~KMEMSIZE）

- 页目录表的起始地址设置在哪里?

页目录表的虚地址为0xFAFEB000，存储在一个4K字节的物理页中，其中每一项是4个字节，保存了页表的地址。

- 页表的起始地址设置在哪里，需要多大空间?

  页表的起始地址在0xFAC00000，大小为4K。

pde_t全称为 page directory entry，页目录表，也就是一级页表的表项（注意：pgdir实际不是表项，而是一级页表本身。实际上应该新定义一个类型pgd_t来表示一级页表本身）。pte_t全称为 page table entry，表示二级页表的表项。uintptr_t表示为线性地址，由于段式管理只做直接映射，所以它也是逻辑地址。

pgdir给出页表起始地址。通过查找这个页表，我们需要给出二级页表中对应项的地址。

参考资料：

https://www.jianshu.com/p/216dc51294b9

https://github.com/zhenghaoz/ucore/tree/master/lab2