实验二：物理内存管理

1 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

2 实验内容

实验一过后大家做出来了一个可以启动的系统，实验二主要涉及操作系统的物理内存管理。操作系统为了使用内存，还需高效地管理内存资源。在实验二中大家会了解并且自己动手完成一个简单的物理内存管理系统。

本次实验包含三个部分。首先了解如何发现系统中的物理内存；然后了解如何建立对物理内存的初步管理，即了解连续物理内存管理；最后了解页表相关的操作，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。本实验里面实现的内存管理还是非常基本的，并没有涉及到对实际机器的优化，比如针对 cache 的优化等。实际操作系统（如Linux等）中的内存管理是相当复杂的。 如果大家有余力，尝试完成扩展练习。

2.1 练习

练习0：填写已有实验

本实验依赖实验1。请把你做的实验1的代码填入本实验中代码中有“LAB1”的注释相应部分。提示：可采用merge工具，比如kdiff3，eclipse中的diff/merge工具，understand中的diff/merge工具等。

练习 1：实现first-fit连续物理内存分配算法。（需要编程）

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: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。

练习 2：实现寻找虚拟地址对应的页表项。（需要编程）

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。get_pte函数的调用关系图如下所示：

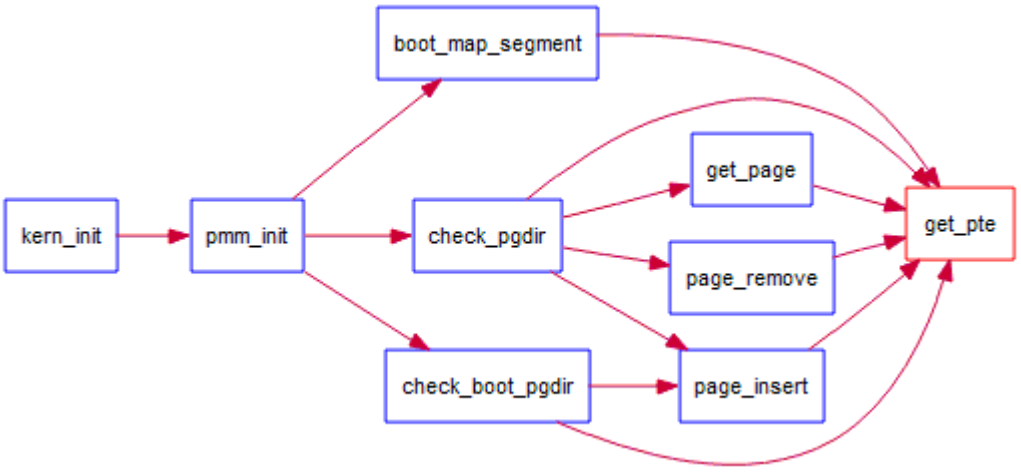

图1 get_pte函数的调用关系图

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（需要编程）

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。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：

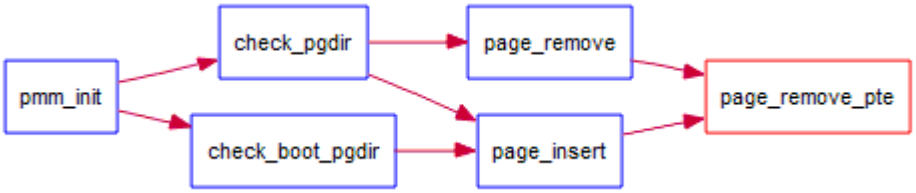

图2 page_remove_pte函数的调用关系图

扩展练习Challenge：任意大小的内存单元slub分配算法（需要编程）

如果觉得上诉练习难度不够，可考虑完成此扩展练习。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。比如，如果连续分配8个16字节的内存块，当分配完毕后，实际只消耗了一个空闲物理页。要求时空都高效，可参考slub算法来实现，可简化实现，能够体现其主体思想即可。要求有设计文档。slub相关网页在http://www.ibm.com/developerworks/cn/linux/l-cn-slub/ 。完成challenge的同学可单独提交challenge。完成得好的同学可获得最终考试成绩的加分。

2.2 项目组成

表1： 实验二文件列表

```
|-- boot
|   |-- asm.h
|   |-- bootasm.S
|   `-- bootmain.c
|-- kern
|   |-- init
|   |   |-- entry.S
|   |   `-- init.c
|   |-- mm
|   |   |-- default_pmm.c
|   |   |-- default_pmm.h
|   |   |-- memlayout.h
|   |   |-- mmu.h
|   |   |-- pmm.c
|   |   `-- pmm.h
|   |-- sync
|   |   `-- sync.h
|   `-- trap
|       |-- trap.c
|       |-- trapentry.S
|       |-- trap.h
|       `-- vectors.S
|-- libs
|   |-- atomic.h
|   |-- list.h
`-- tools
    |-- kernel.ld
```

相对与实验一，实验二主要增加和修改的文件如上表红色部分所示。主要改动如下：

- boot/bootasm.S：增加了对计算机系统中物理内存布局的探测功能；
- kern/init/entry.S：根据临时段表重新暂时建立好新的段空间，为进行分页做好准备。
- kern/mm/default_pmm.[ch]：提供基本的基于链表方法的物理内存管理（分配单位为页，即4096字节）；

- kern/mm/pmm.[ch]：pmm.h定义物理内存管理类框架struct pmm_manager，基于此通用框架可以实现不同的物理内存管理策略和算法(default_pmm.[ch] 实现了一个基于此框架的简单物理内存管理策略)；pmm.c包含了对此物理内存管理类框架的访问，以及与建立、修改、访问页表相关的各种函数实现。
- kern/sync/sync.h：为确保内存管理修改相关数据时不被中断打断，提供两个功能，一个是保存eflag寄存器中的中断屏蔽位信息并屏蔽中断的功能，另一个是根据保存的中断屏蔽位信息来使能中断的功能；（可不用细看）
- libs/list.h：定义了通用双向链表结构以及相关的查找、插入等基本操作，这是建立基于链表方法的物理内存管理（以及其他内核功能）的基础。其他有类似双向链表需求的内核功能模块可直接使用list.h中定义的函数。
- libs/atomic.h：定义了对一个变量进行读写的原子操作，确保相关操作不被中断打断。（可不用细看）
- tools/kernel.ld：修改了ucore的起始入口和代码段的起始地址。

编译方法

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（仅供参考）

```
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。

3 物理内存管理

3.1 探测系统物理内存布局

当 ucore 被启动之后，最重要的事情就是知道还有多少内存可用，一般来说，获取内存大小的方法由 BIOS 中断调用和直接探测两种。其中 ，BIOS 中断调用方法是在实模式下完成，而直接探测方法必须在保护模式下完成。通过 BIOS 中断获取内存布局有三种方式，都是基于 INT 15h中断，分别为88h e801h e820h。但是 并非在所有情况下这三种方式都能工作。在 Linux kernel 里,采用的方法是依次尝试这三 种方法。而在本实验中，我们通过e820h中断获取内存信息。因为e820h中断必须在实模式下使用，我们在 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。具体实现详见boot/bootasm.S。有关探测系统物理内存的 具体信息参见附录。

3.2 以页为单位管理物理内存

在获得可用物理内存范围后，系统需要建立相应的数据结构来管理以物理页（按4KB对齐，且大小为4KB的物理内存单元）为最小单位的整个物理内存，以配合后续涉及的分页管理机制。每个物理页可以用一个 Page数据结构来表示。由于一个物理页需要占用一个Page结构的空间，Page结构在设计时须尽可能小，以减少对内存的占用。Page的定义在mm/memlayout.h中。以页为单位的物理内存分配管理的实现在kern/default_pmm.[ch]。

为了与以后的分页机制配合，我们首先需要建立对整个计算机的每一个物理页的属性用结构Page来表示，它包含了映射此物理页的虚拟页个数，描述物理页属性的flags和双向链接各个Page结构的page_link双向链表。

```
struct Page {
        atomic_t ref;         // page frame's reference counter
        uint32_t flags;  // array of flags that describe the status of the page frame
        list_entry_t page_link; // free list link
};
```

所有空闲的物理页可用一个双向链表管理起来，便于分配和释放，为此定义了一个free_area_t数据结构，包含了一个list_entry结构的双向链表指针和记录当前空闲页的个数的无符号整型变量nr_free。其中的链表指针指向了空闲的物理页。

```
/* 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;
```

有了这两个数据结构，ucore就可以管理起来整个以页为单位的物理内存空间。接下来需要解决两个问题：

- 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？
- 空闲内存空间的起始地址在哪里？

对于这两个问题，我们首先根据bootloader给出的内存布局信息找出最大的物理内存地址

maxpa（定义在page_init函数中的局部变量），由于x86的起始物理内存地址为0，所以可以得知需要管理的物理页个数为

```
npage = maxpa / PGSIZE
```

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按页大小为边界去整后，作为管理页级物理内存空间所需的Page结构的内存空间，记为：

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

为了简化起见，从地址0到地址pages+ sizeof(struct Page) * npage)结束的物理内存空间设定为已占用物理内存空间（起始0~640KB的空间是空闲的），地址pages+ sizeof(struct Page) * npage)以上的空间为空闲物理内存空间，这时的空闲空间起始地址为

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

为此我们需要把这两部分空间给标识出来。首先，对于所有物理空间，通过如下语句即可实现占用标记：

```
for (i = 0; i < npage; i ++) {
        SetPageReserved(pages + i);
}
```

然后，根据探测到的空闲物理空间，通过如下语句即可实现空闲标记：

```
//获得空闲空间的起始地址 begin 和结束地址 end
……
init_memmap(pa2page(begin), (end - begin) / PGSIZE);
```

其实SetPageReserved只需把物理地址对应的Page结构中的flags标志设置为PG_reserved ，表示这些页已经被使用了。而init_memmap函数则是把空闲物理页对应的Page结构中的flags和引用计数ref清零，并加到free_area.free_list指向的双向列表中，为将来的空闲页管理做好初始化准备工作。

关于内存分配的操作系统原理方面的知识有很多，但在本实验中只实现了最简单的内存页分配算法。相应的实现在default_pmm.c中的default_alloc_pages函数和default_free_pages函数，相关实现很简单，这里就不具体分析了，直接看源码，应该很好理解。

其实实验二在内存分配和释放方面最主要的作用是建立了一个物理内存页管理器框架，这实际上是一个函数指针列表，定义如下：

```
struct pmm_manager {
        const char *name; //物理内存页管理器的名字
        void (*init)(void); //初始化内存管理器
        void (*init_memmap)(struct Page *base, size_t n); //初始化管理空闲内存页的数据结构
        struct Page *(*alloc_pages)(size_t n); //分配 n 个物理内存页
        void (*free_pages)(struct Page *base, size_t n); //释放 n 个物理内存页
        size_t (*nr_free_pages)(void); //返回当前剩余的空闲页数
        void (*check)(void); //用于检测分配/释放实现是否正确的辅助函数
};
```

重点是实现init_memmap/ alloc_pages/ free_pages这三个函数。当完成物理内存页管理初始化工作后，计算机系统的内存布局如下图所示：

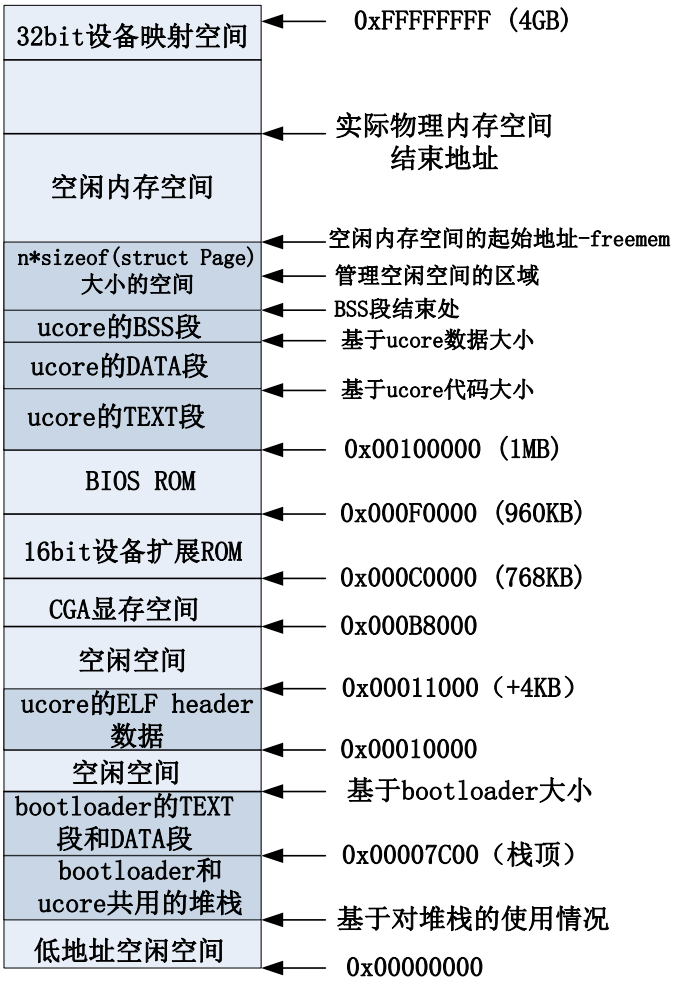

图 3 计算机系统的内存布局

3.3 实现分页机制

如图4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可以得到线性地址，线性地址通过页式管理得到物理地址。

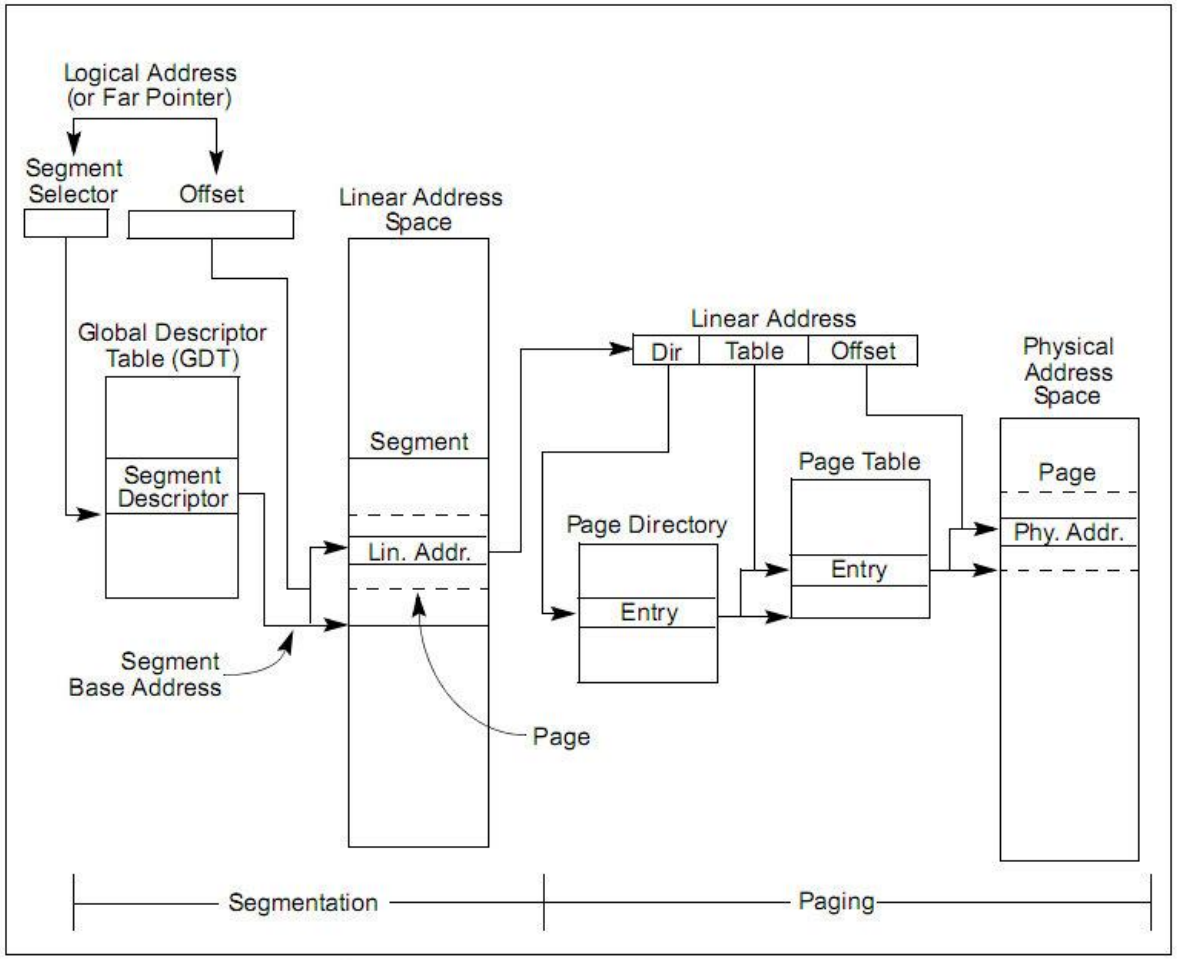

图 4 段页式管理

段式管理前一个实验已经讨论过。在 ucore 中段式管理只起到了一个过渡作用，它将逻辑地址不加转换直接映射成线性地址，所以我们在下面的讨论中可以对这两个地址不加区分（目前的 OS 实现也是不加区分的）。对段式管理有兴趣的同学可以参照《Intel® 64 and IA-32Architectures Software Developer ’s Manual – Volume 3A》3.2 节。

如图5所示，页式管理将线性地址分成三部分（图中的 Linear Address 的 Directory 部分、Table 部分和 Offset 部分）。ucore 的页式管理通过一个二级的页表实现。一级页表的起始物理地址存放在 cr3 寄存器中，这个地址必须是一个页对齐的地址，也就是低 12 位必须为 0。目前，ucore 用boot_cr3（mm/pmm.c）记录这个值。

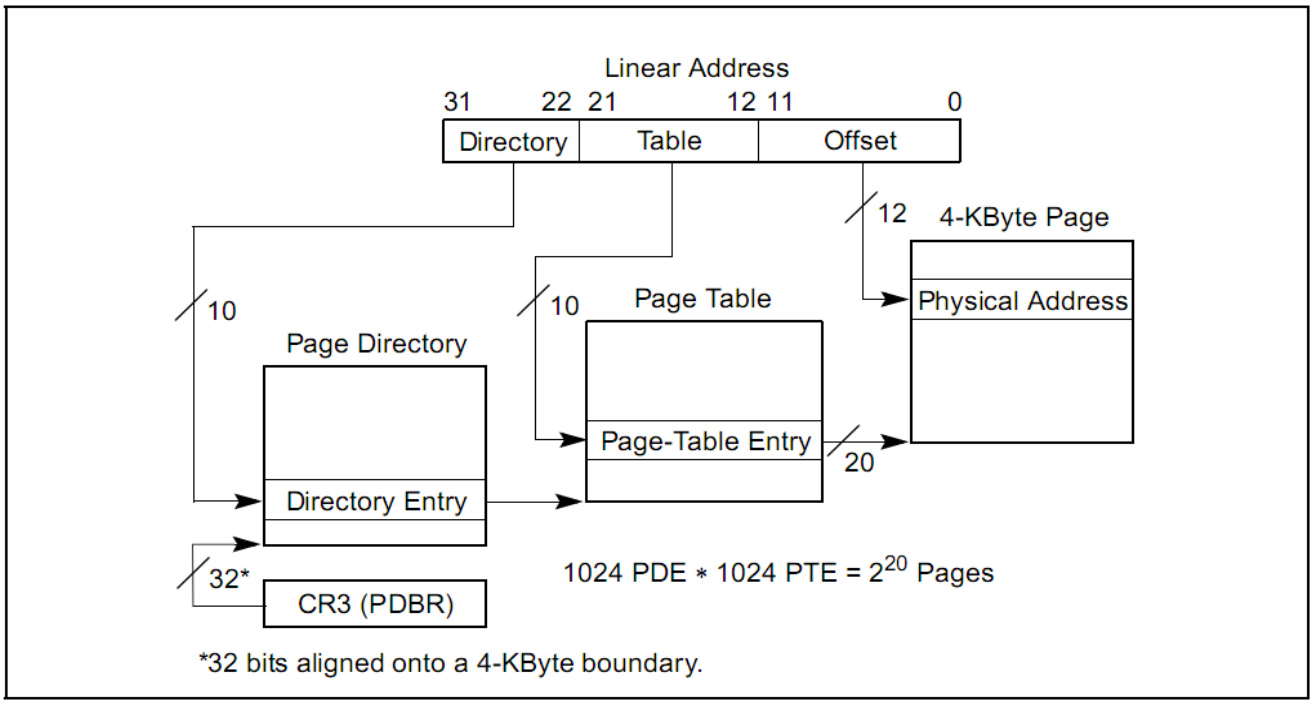

图5 分页机制管理

为了实现分页机制，需要建立好虚拟内存和物理内存的页映射关系，即建立二级页表。这需要思考如下问题：

- 对于哪些物理内存空间需要建立页映射关系？
- 具体的页映射关系是什么？
- 页目录表的起始地址设置在哪里？
- 页表的起始地址设置在哪里，需要多大空间？
- 如何设置页目录表项的内容？
- 如何设置页目录表项的内容？

由于物理内存页管理器管理了从 0 到实际可用物理内存大小的物理内存空间，所以对于这些物理内存空间都需要建立好页映射关系。由于目前 ucore 只运行在内核空间，所以可以建立一个一一映射关系。假定内核虚拟地址空间的起始地址为 0xC0000000，则虚拟内存和物理内存的具体页映射关系为：

```
Virtual Address=Physical Address+0xC0000000
```

由于我们已经具有了一个物理内存页管理器 default_pmm_manager，我们就可以用它来获得所需的空闲物理页。在二级页表结构中，页目录表占 4KB 空间，ucore 就可通过 default_pmm_manager 的 default_alloc_pages 函数获得一个空闲物理页，这个页的起始物理地址就是页目录表的起始地址。同理，ucore 也通过这种方式获得各个页表所需的空间。页表的空间大小取决与页表要管理的物理页数 n，一个页表项（32 位，即 4 字节）可管理一个物理页，页表需要占 n/256 个物理页空间。这样页目录表和页表所占的总大小为 4096+1024*n 字节。

为把 0~KERNSIZE（明确 ucore 设定实际物理内存不能超过 KERNSIZE 值，即 0x38000000 字节，896MB，3670016 个物理页）的物理地址一一映射到页目录表项和页表项的内容，其大致流程如下：

1. 先通过 default_pmm_manager 获得一个空闲物理页，用于页目录表；
2. 调用 boot_map_segment 函数建立一一映射关系，具体处理过程以页为单位进行设置，即

   Virtual Address=Physical Address+0xC0000000

   - 设一个逻辑地址 la（按页对齐，故低 12 位为零）对应的物理地址 pa（按页对齐，

故低 12 位为零），如果在页目录表项（la 的高 10 位为索引值）中的存在位（PTE_P）为 0，表示缺少对应的页表空间，则可通过 default_pmm_manager 获得一个空闲物理页给页表，页表起始物理地址是按 4096 字节对齐的，这样填写页目录表项的内容为

页目录表项内容 = 页表起始物理地址| PTE_U | PTE_W | PTE_P

进一步对于页表中对应页表项（la 的中 10 位为索引值）的内容为

页表项内容 = pa | PTE_P | PTE_W

其中：

- 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

建立好一一映射的二级页表结构后，接下来就要使能分页机制了，这主要是通过 enable_paging 函数实现的，这个函数主要做了两件事：

1. 通过 lcr3 指令把页目录表的起始地址存入 CR3 寄存器中；
2. 通过 lcr0 指令把 cr0 中的 CR0_PG 标志位设置上。

执行完 enable_paging 函数后，计算机系统进入了分页模式！但到这一步还不够，还记得 ucore 在最开始通过 kern_entry 函数设置了临时的新段映射机制吗？这个临时的新段映射机制不是最简单的对等映射，导致虚拟地址和线性地址不相等。而刚才建立的页映射关系是建立在简单的段对等映射，即虚拟地址=线性地址的假设基础之上的。所以我们需要进一步调整段映射关系，即重新设置新的 GDT，建立对等段映射。

这里需要注意：在进入分页模式到重新设置新 GDT 的过程是一个过渡过程。在这个过渡过程中，已经建立了页表机制，所以通过现在的段机制和页机制实现的地址映射关系为：

```
Virtual Address=Linear Address + 0xC0000000 = Physical Address +0xC0000000+0xC0000000
```

在这个特殊的阶段，如果不把段映射关系改为 Virtual Address = Linear Address，则通过段页式两次地址转换后，无法得到正确的物理地址。为此我们需要进一步调用 gdt_init 函数，根据新的 gdt 全局段描述符表内容（gdt 定义位于 pmm.c 中），恢复以前的段映射关系，即使得 Virtual Address = Linear Address。这样在执行完 gdt_init 后，通过的段机制和页机制实现的地址映射关系为：

```
Virtual Address=Linear Address = Physical Address +0xC0000000
```

这里存在的一个问题是，在调用 enable_page 函数使能分页机制后到执行完毕 gdt_init 函数重新建立好段页式映射机制的过程中，内核使用的还是旧的段表映射，也就是说，enable paging 之后，内核使用的是页表的低地址 entry。 如何保证此时内核依然能够正常工作呢？其实只需让低地址目录表项的内容等于以 KERNBASE 开始的高地址目录表项的内容即可。目前内核大小不超过 4M （实际上是 3M，因为内核从 0x100000 开始编址），这样就只需要让页表在 0~4MB 的线性地址与 KERNBASE ~ KERNBASE+4MB 的线性地址获得相同的映射即可，都映射到 0~4MB 的物理地址空间，具体实现在 pmm.c 中 pmm_init 函数的语句：

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

实际上这种映射也限制了内核的大小。当内核大小超过预期的 3MB 就可能导致打开分页之后内核 crash，在后面的试验中，也的确出现了这种情况。解决方法同样简单，就是拷贝更多的高地址项到低地址。

当执行完毕 gdt_init 函数后，新的段页式映射已经建立好了，上面的 0~4MB 的线性地址与 0~4MB 的物理地址一一映射关系已经没有用了。所以可以通过如下语句解除这个老的映射关系。

```
boot_pgdir[0] = 0;
```

3.4 自映射机制

上一小节讲述了通过 boot_map_segment 函数建立了基于一一映射关系的页目录表项和页表项，这里的映射关系为：

virtual addr (KERNBASE~KERNBASE+KMEMSIZE) = physical_addr (0~KMEMSIZE)

这样只要给出一个虚地址和一个物理地址，就可以设置相应 PDE 和 PTE，就可完成正确的映射关系。

如果我们这时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，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。这样过程比较繁琐。

我们需要有一个简洁的方法来实现这个查找。ucore 做了一个很巧妙的地址自映射设计，把页目录表和页表放在一个连续的 4MB 虚拟地址空间中，并设置页目录表自身的虚地址<-->物理地址映射关系。这样在已知页目录表起始虚地址的情况下，通过连续扫描这特定的 4MB 虚拟地址空间，就很容易访问每个页目录表项和页表项内容。

具体而言，ucore 是这样设计的，首先设置了一个常量（memlayout.h）：

VPT=0xFAC00000， 这个地址的二进制表示为：

1111 1010 1100 0000 0000 0000 0000 0000

高 10 位为 1111 1010 11，即 10 进制的 1003，中间 10 位为 0，低 12 位也为 0。在 pmm.c 中有两个全局初始化变量

```
pte_t * const vpt = (pte_t *)VPT;
pde_t * const vpd = (pde_t *)PGADDR(PDX(VPT), PDX(VPT), 0);
```

并在 pmm_init 函数执行了如下语句：

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

这些变量和语句有何特殊含义呢？其实 vpd 变量的值就是页目录表的起始虚地址 0xFAFEB000，且它的高 10 位和中 10 位是相等的，都是 10 进制的 1003。当执行了上述语句，就确保了 vpd 变量的值就是页目录表的起始虚地址，且 vpt 是页目录表中第一个目录表项指向的页表的起始虚地址。此时描述内核虚拟空间的页目录表的虚地址为 0xFAFEB000，大小为 4KB。页表的理论连续虚拟地址空间 0xFAC00000~0xFB000000，大小为 4MB。因为这个连续地址空间的大小为 4MB，可有 1M 个 PTE，即可映射 4GB 的地址空间。

但 ucore 实际上不会用完这么多项，在 memlayout.h 中定义了常量

```
#define KMEMSIZE        0x38000000
```

表示 ucore 只支持 896MB 的物理内存空间，这个 896MB 只是一个设定，可以根据情况改变。则最大的内核虚地址为常量

```
#define KERNTOP         (KERNBASE + KMEMSIZE)=0xF8000000
```

所以最大内核虚地址 KERNTOP 的页目录项虚地址为

vpd+0xF8000000/0x400000=0xFAFEB000+0x3E0=0xFAFEB3E0

最大内核虚地址 KERNTOP 的页表项虚地址为：

vpt+0xF8000000/0x1000=0xFAC00000+0xF8000=0xFACF8000

在 pmm.c 中的函数 print_pgdir 就是基于 ucore 的页表自映射方式完成了对整个页目录表和页表的内容扫描和打印。注意，这里不会出现某个页表的虚地址与页目录表虚地址相同的情况。

print pgdir 函数使得 ucore 具备和 qemu 的 info pg 相同的功能，即 print pgdir 能够从内存中，将当前页表内有效数据（PTE_P）印出来。拷贝出的格式如下所示:

```
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
```

上面中的数字包括括号里的，都是十六进制。

主要的功能是从页表中将具备相同权限的 PDE 和 PTE 项目组织起来。比如上表中：

```
PDE(0e0) c0000000-f8000000 38000000 urw
```

- PDE(0e0)：0e0表示 PDE 表中相邻的 224 项具有相同的权限；
- c0000000-f8000000：表示 PDE 表中,这相邻的两项所映射的线性地址的范围；
- 38000000：同样表示范围，即f8000000减去c0000000的结果；
- urw：PDE 表中所给出的权限位，u表示用户可读，即PTE_U，r表示PTE_P，w表示用户可写，即PTE_W。

```
PDE(001) fac00000-fb000000 00400000 -rw
```

表示仅1 条连续的PDE 表项具备相同的属性。相应的，在这条表项中遍历找到2 组PTE 表项，输出如下:

```
|-- PTE(000e0) faf00000-fafe0000 000e0000 urw
|-- PTE(00001) fafeb000-fafec000 00001000 -rw
```

注意：

1. PTE 中输出的权限是 PTE 表中的数据给出的，并没有和 PDE 表中权限做与运算。
2. 整个print_pgdir函数强调两点：第一是相同权限，第二是连续。
3. print_pgdir中用到了vpt和vpd两个变量。可以参 考VPT和PGADDR两个宏。

自映射机制还可方便用户态程序访问页表。因为页表是内核维护的，用户程序很难知道自己页表的映射结构。VPT 实际上在内核地址空间的，我们可以用同样的方式实现一个用户地址空间的映射（比如 pgdir[UVPT] = PADDR(pgdir) | PTE_P | PTE_U，注意，这里不能给写权限，并且 pgdir 是每个进程的 page table，不是 boot_pgdir），这样，用户程序就可以用和内核一样的 print_pgdir 函数遍历自己的页表结构了。

在 page_init 函数建立完实现物理内存一一映射和页目录表自映射的页目录表和页表后，一旦使能分页机制，则 ucore 看到的内核虚拟地址空间如下图所示：

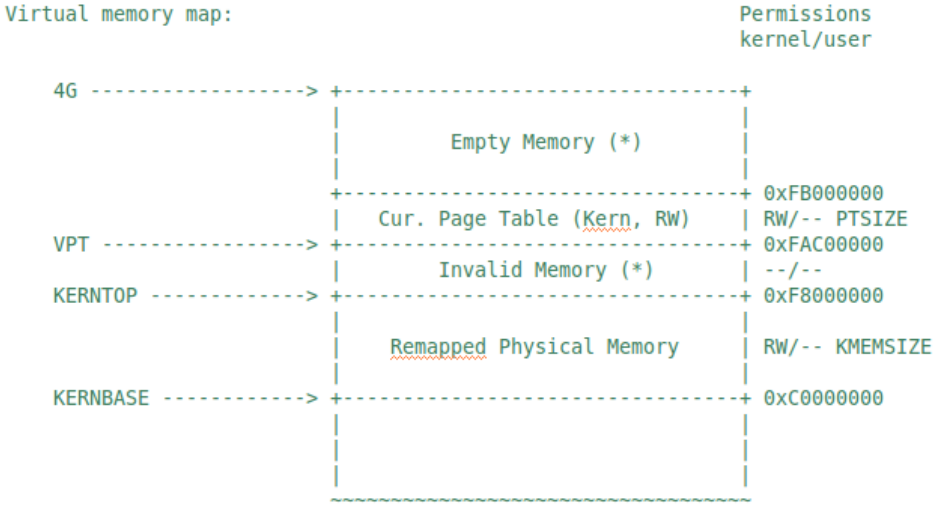

图 6 使能分页机制后的虚拟地址空间图

ucore 的内存管理经常需要查找页表：给定一个虚拟地址，找出这个虚拟地址在二级页表中对应的项。通过更改此项的值可以方便地将虚拟地址映射到另外的页上。这个函数叫 get_pte，在mm/pmm.c中。请根据代码中的提示完成。它的原型为

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

这里涉及到三个类型pte t、pde t和uintptr t。通过参见mm/mmlayout.h和libs/types.h，可知它们其实都是unsigned int类型。在此做区分，是为了分清概念。

pde_t全称为 page directory entry，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。虽然目前我们只有boot_pgdir一个页表，但是引入进程的概念之后每个进程都会有自己的页 表。

有可能根本就没有对应的二级页表的情况，所以二级页表不必要一开始就分配，而是等到需要的时候再添加对应的二级页表。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。如果create参数为0，则get_pte返回NULL；如果create参数不为0，则get_pte需要申请一个新的物理页（通过alloc_page来实现，可在mm/pmm.h中找到它的定义），再在一级页表中添加页目录表项指向表示二级页表的新物理页。注意，新申请的页必须全部设定为零，因为这个页所代表的虚拟地址都没有被映射。

当建立从一级页表到二级页表的映射时，需要注意设置控制位。这里应该设置同时设置上PTE_U、PTE_W和PTE_P（定义可在mm/mmu.h）。如果原来就有二级页表，或者新建立了页表，则只需返回对应项的地址即可。

虚拟地址只有映射上了物理页才可以正常的读写。在完成映射物理页的过程中，除了要象上面那样在页表的对应表项上填上相应的物理地址外，还要设置正确的控制位。有关 x86 中页表控制位的详细信息，请参照《Intel® 64 and IA-32 Architectures Software Developer ’s Manual – Volume 3A》4.11 节。

只有当一级二级页表的项都设置了用户写权限后，用户才能对对应的物理地址进行读写。所以我们可以在一级页表先给用户写权限，再在二级页表上面根据需要限制用户的权限，对物理页进行保护。由于一个物理页可能被映射到不同的虚拟地址上去（譬如一块内存在不同进程 间共享），当这个页需要在一个地址上解除映射时，操作系统不能直接把这个页回收，而是要先看看它还有没有映射到别的虚拟地址上。这是通过查找该物理页的ref变量实现的。参看page_insert函数，可以看到系统是如何维护这个变量的。

page_insert函数将物理页映射在了页表上。当不需要再访问这块虚拟地址时，可以把这块物理页回收，用在其他地方。取消映射由page_remove来做，这其实是page insert的 逆操作，这里需要你把page_remove_pte函数补全。请仔细阅读page insert函数了 解映射过程，以及函数上面的注释来完成这个练习。

4. 实验报告要求

从网站上下载lab2.zip后，解压得到本文档和代码目录 lab2，完成实验中的各个练习。完成代码编写并检查无误后，在对应目录下执行 make handin 任务，即会自动生成 lab2-handin.tar.gz。最后请一定提前或按时提交到网络学堂上。

注意有***“LAB2”***的注释，代码中所有需要完成的地方（challenge除外）都有***“LAB2”和“YOUR CODE”***的注释，请在提交时特别注意保持注释，并将***“YOUR CODE”***替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

附录

探测计算机系统中的物理内存分布和大小

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。其中 BIOS 中断调用必须在实模式下进行，所以在 bootloader 进入保护模式前完成这部分工作相对比较合适。这些部分由 boot/bootasm.S 中从 probe_memory 处到 finish_probe 处的代码部分完成完成。通过 BIOS 中断获取内存可调用参数为 e820h 的 INT 15h BIOS 中断。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，其具体表示如下：

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

看下面的(Values for System Memory Map address type)14

```
Values for System Memory Map address type:
01h    memory, available to OS
02h    reserved, not available (e.g. system ROM, memory-mapped device)
03h    ACPI Reclaim Memory (usable by OS after reading ACPI tables)
04h    ACPI NVS Memory (OS is required to save this memory between NVS sessions)
other  not defined yet -- treat as Reserved
```

INT15h BIOS 中断的详细调用参数:

```
eax：e820h：INT 15 的中断调用参数；
edx：534D4150h (即 4 个 ASCII 字符“SMAP”)，这只是一个签名而已；
ebx：如果是第一次调用或内存区域扫描完毕，则为 0。如果不是，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；
es:di：指向保存地址范围描述符结构的缓冲区，BIOS 把信息写入这个结构的起始地址。
```

此中断的返回值为:

```
cflags 的 CF 位：若 INT 15 中断执行成功，则不置位，否则置位；
eax：534D4150h ('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 往 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
```

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

```
struct e820map {
        int nr_map;
        struct {
                long long addr;
                long long size;
                long type;
        } map[E820MAX];
};
```

物理内存探测

物理内存探测是在 bootasm.S 中实现的，相关代码很短，如下所示：

```
probe_memory:
//对 0x8000 处的 32 位单元清零,即给位于 0x8000 处的
//struct e820map 的结构域 nr_map 清零
        movl $0, 0x8000
            xorl %ebx, %ebx
//表示设置调用 INT 15h BIOS 中断后，BIOS 返回的映射地址描述符的起始地址
```

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

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