实验二：内存管理

1 实验目的

- 理解内存地址的转换和保护
- 理解页表的建立和使用方法
- 了解物理内存的管理方法
- 了解常用的减少碎片的方法
- 了解虚拟内存的管理方法

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，但是仅仅启动是远远不够的，一个合格的系统至少可以运行一些其它的任务，可以对计算资源进行管理。除了 CPU 之外，最重要的计算资源就是内存。在实验二中大家会了解并且自己动手完成一个简单的内存管理系统。

在进行本次实验之前，大家需要将实验一中的代码填写到相应的位置，再开始本次实验，否则会出现运行时错误。本次实验包含 3 个部分。首先通过 proj 5 熟悉最低层的内存操作，即如何建立页表以及如何使用 Buddy System 管理物理页；然后通过 proj 6（代码在 proj 7 中）熟悉建立在 Buddy System 之上的 slab 分配器。前两个部分完成后，系统在内核态分配内存的请求都可以得到很好的处理。之后通过 proj 7 熟悉初步的虚拟内存管理，了解用户态内存有关知识，可以对之前经常遇到的“Segmentation fault”或者“Bad access”之类的错误有更深层次的认识。

ucore 里面实现的内存管理还是非常基本的内容，并没有涉及到对实际机器的优化，比如针对 cache 的优化等。实际系统中的内存管理是相当复杂的（助教对此表示非常痛苦……）。如果希望自己的系统支持更高级的内存管理，可以尝试完成扩展练习。

3 proj 5 物理内存管理

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，以及怎样使用它们。

内存管理（Memory Management，简称 mm）需要做的事情有两步：

1. 探测系统物理内存的分布和大小：一般来说，获取内存大小的办法由 BIOS 中断调用和直接探测。其中 BIOS 中断调用是在实模式下，而直接探测必须在保护模式下。通过 BIOS 中断获取内存有三种方式，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。详见`boot/bootasm.S`。有关探测系统物理内存的具体信息参见这里。

2. 建立物理页管理：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物理页。每个物理页可以用一个 Page 描述符来表示。由于一个物理页需要一个`Page`结构的空间，`Page`结构在设计时必须要尽可能的小，以减少操作系统内核的内存占用。`Page`的定义可以在`mm/memlayout.h`中找到。

3.1 页式管理

如图1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。

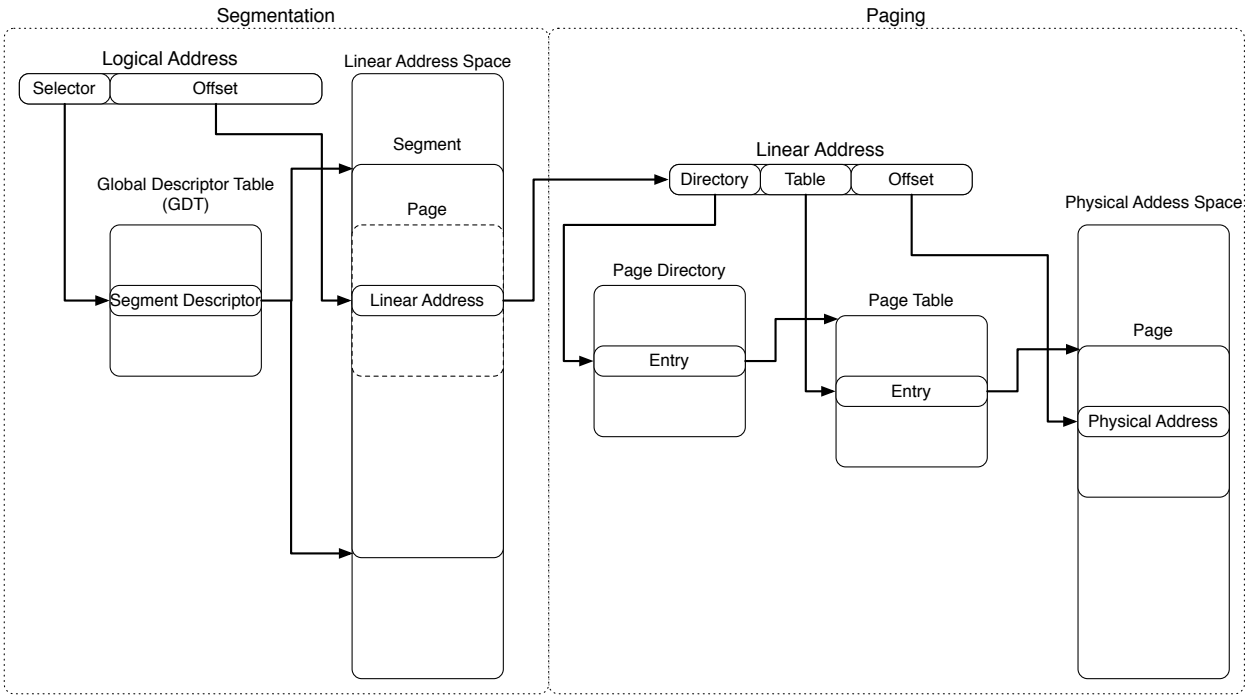

图 1: 段式管理和页式管理

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-32 Architectures Software Developer's Manual – Volume 3A》3.2 节。

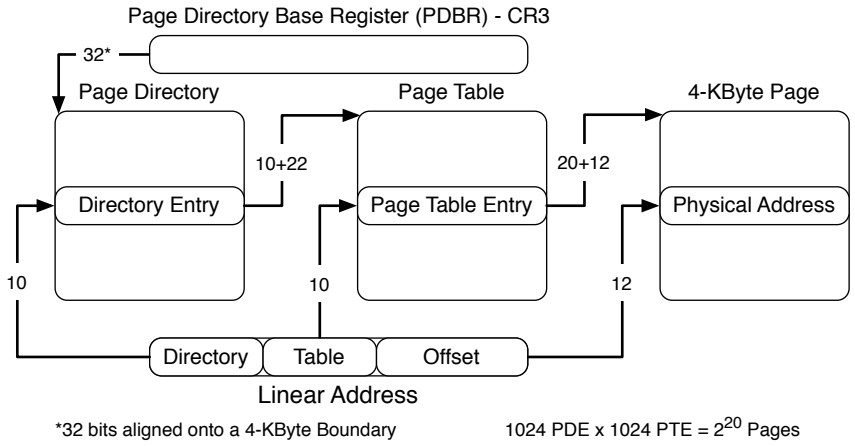

图 2: 页式管理

如图2，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。以后，我们将为每个进程维护一套页表，届时将会维护不同的页表的初始物理地址。

一级页表有一个物理页，共 4096B，每 4B 一个项，共 1024 项。线性地址的前 10 位可以用来表示这个虚拟地址在一级页表的哪一个项。每个项上高 20 位记录下一级页表的物理地址（因为页表地址的低 12 位必须为 0），另外 12 位作为控制位，可以储存各种信息，包括这个项是否有效（`PTE_P`），访问权限（`PTE_W`、`PTE_U`）等。找到二级页表的位置后,线性地址的中间 10 位可以用来寻找二级页表中的项。这个项中的高 20 位便是这个线性地址对应页的物理地址。同样的，项里的低 12 位也作为控制位存储了关于这个物理页的信息。这仅仅是一个简单的介绍，详情请参见《Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 3A》3.6、3.7 节。

如果每次访问内存都需要查找页表，则一次读写内存将至少导致两次内存读，极大的降低了运行速度。为了解决这个问题，CPU 里有 TLB（Translation Lookaside Buffer），缓存最近使用的虚拟页与物理页之间的映射关系。当修改了页表后，需要将 TLB 更新，使得 TLB 会去页表中取并缓存新的映射关系，而不是继续沿用旧的映射关系。刷 TLB 的方法有全部刷和刷特定项两种。具体参见《Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 3A》3.12 节。ucore 采取显式地重新载入 cr3 寄存器的方法（`tlb_invalidate`函数）。

3.1.1 练习 1：查找页表项

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。在此做这个区分，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。整套页表不必要一开始就全部展开，而是等到需要的时候再添加对应的二级页表。这时需要参看create参数，如果为0，则返回NULL；如果不为零，则申请一个新的物理页（通过alloc_page，可在mm/pmm.h中找到它的定义），再在一级页表中添加项指向新的页。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（它们的定义可在mm/mmu.h里找到）。详情可参看下节。

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。

3.1.2 练习 2：映射和取消映射

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，还要正确的设置控制位。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这个函数补全。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。

3.1.3 练习 3：打印页表

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）打印出来。

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。实现起来可能相对复杂。但是理解好页表结构,实现起来并不困难。
3. print_pgdir中用到了vpt和vpd两个变量，这两个变量的值需要自己指定。可以参考VPT和PGADDR两个宏。

VPT存放了页表实际物理页的地址。当 ucore 运行到print_pgdir中时，已经开启了页机制。也就是说现在你不能通过页表存放的物理地址来访问页表，而只能通过虚拟地址来访问。你需要了解页表的虚拟地址、cr3 寄存器存储数值以及页表结构的确切含义。

你可以使用PGADDR这个宏构造一个地址使它映射到页表项上。比如你现在知道pgdir[PDX(VPT)] = boot_cr3 | PTE_U | PTE_W | PTE_P。就是说一级页表中VPT项存放了实际一级页表的物理页。你现在需要访问 PDE 这张表，直接访问VPT行么？可以这样看，访问VPT的过程经过页表，实际上是分成了如下步骤：

1. 在一级页表中，访问PDX(VTP)项，得到存放二级页表的物理页；
2. 在对应的二级页表中，访问PTX(VPT)项,得到存储数据的实际的物理页；
3. 在实际的物理页中，访问PGOFF(VPT)，得到VPT映射的数据。

可以看到在第一步仍然访问的是一级页表本身，但是第二步已经飞掉了。你可以利用VPT构造出一个地址，使得在第二步和第三步中还可以保持一直访问一级页表本身。同样可以利用VPT构造一个地址访问二级页表。

你需要自己实现get_pgtable_items函数。函数原型为：

```
static int get_pgtable_items (int left, int right, int start,
   uint32_t *table, int *left_store, int *right_store)
```

其中table是需要查找的页表，left到right是需要查找的表项范围，start是开始查找的位置，而left_store和right_store是连续相同权限的表项的范围。具体用法请参考print_pgdir。

3.2 Buddy 系统

在内存管理中，有一个常见的问题：外部碎片。外部碎片指的是当进行频繁的内存分配和释放操作，而且每次请求的内存大小不相同时，会导致许多小的内存块分散在内存之中。这样当系统请求一块大的内存块时，会导致内存中总的的空间足够，但却找不到一块单独的连续内存能满足要求。要解决外碎片的问题，有两种方法：一种是通过页映射机制把不连续的物理页映射到连续的线性地址空间（linear address space），或者用合适的方法来避免外部碎片的出现。通常情况下我们更倾向于后者，因为在一些情况下，连续的物理页是必要的（比如 DMA），一种解决外碎片问题的方法是使用 Buddy System。

如果在分配存储块时经常会将一个大的存储块分裂成两个大小相等的小块，那么这两个小块就称为“伙伴”。在 Buddy System 进行合并时，只会将互为伙伴的两个存储块合并成大块，也就是说如果有两个存储块大小相同，地址也相邻，但是不是由同一个大块分裂出来的，也不会被合并起来。正常情况下，起始地址为 p，大小为 2^k 的存储块，其伙伴块的起始地址为 $p + 2^k$ 或 $p - 2^k$。

Buddy System 的基本原理是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（2^n），即 $1, 2, 4, 8, 16, 32, 64, 128 \ldots$。假设系统全部

可用空间为 2^k，那么在 Buddy System 初始化时将生成一个长度为 $k+1$ 的可用空间表，并将全部可用空间作为一个大小为 2^k 的块挂接在表的最后一个节点上。当用户申请 $size$ 大小的存储空间时，Buddy System 分配的块大小为 2^m，满足 $2^{m-1} < size \leq 2^m$。此时 Buddy System 将在可用空间表中的 m 位置寻找可用的块。显然表中这个位置为空，于是 Buddy System 就开始寻找向上查找 $m+1$、$m+2$，直到达到 k 为止。找到 k 后，便得到可用块。这个可用块将分裂成两个大小为 2^{k-1} 的块，并将其中一个插入到可用空间表中 $k-1$ 的位置，同时对另外一个继续进行分裂。如此以往直到得到两个大小为 2^m 的块为止。当一个存储块不再使用时，用户应该将该块归还给 Buddy System。此时系统将根据块的大小计算出其在可用空间表中的位置，然后插入到可用块链表的末尾。在这一步完成后，系统立即开始合并操作。该操作是将“伙伴”合并到一起，并放到可用空间表的下一个位置中，并继续对更大的块进行合并，直到无法合并为止。

在实验系统中，有关 Buddy System 的代码定义在`mm/buddy.h`和`mm/buddy.c`中。Buddy System 一共维护 11 种大小的页。`free_area`中为每种大小的页维护了一个列表。分配内存时，Buddy System 首先查看有没有相应大小的块存在。如要分配 8KB（2^1 个页），则从`free_area[1]`中查找有没有空闲块，如果没有则找更大的块，并将相应的大块分裂成小块，加入相应列表。释放内存时，Buddy System 先看对应块的“伙伴”是否也是空闲的，如果是则合并成更大的块继续重复释放的过程。

3.2.1 练习 4：分配和释放物理页

分配物理页的核心函数为`buddy_alloc_pages_sub`，注意里面的链表操作以及属性设置。注意到这里初步引入了 Zone 的概念，因此需要用特定的函数得到物理地址对应的`Page`结构。

仿照`buddy_alloc_pages_sub`,你需要实现对应的释放物理页的函数`buddy_free_pages_sub`。为了提高效率，某一个块所对应的“伙伴”的起始地址是可以直接计算的，但是仍然需要`page_is_buddy`检查权限。

4 proj 6 slab 缓存

Buddy System 虽然较好地解决了外碎片的问题，但是仍然没有解决内碎片的问题。首先，内核通常依赖于对小对象的分配，它们会在系统生命周期内进行无数次分配。如果每次都给这些对象分配一个页，那么内存的浪费是巨大的（所谓的内碎片）。同时内核中普通对象进行初始化所需的时间超过了对其进行分配和释放所需的时间。因此如果不将内存释放回一个全局的内存池，而是将内存保持为针对特定目而初始化的状态则会提高内存利用的效率。例如，如果内存被分配给了一个互斥锁，那么只需在为互斥锁首次分配内存时执行一次互斥锁初始化函数即可。后续的内存分配不需要执行这个初始化函数，因为从上次释放和调用析构之后，它已经处于所需的状态中了。

这就是 slab 缓存分配器的思想。与传统的内存管理模式相比，slab 缓存分配器提供了很多优点。slab 缓存分配器通过对类似大小的对象进行缓存而提供这种功能，从而避免了常见的碎片问题。slab 分配器还支持通用对象的初始化，从而避免了为同一目而对一个对象重复进行

初始化。最后，slab 分配器还可以支持硬件缓存对齐和着色，这允许不同缓存中的对象占用相同的缓存行，从而提高缓存的利用率并获得更好的性能。

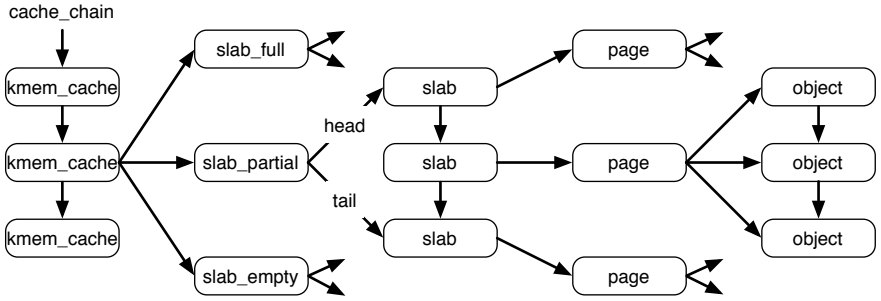

图 3: slab 结构

图3给出了 slab 结构的高层组织结构。在最高层是 cache_chain，这是一个 slab 缓存的链接列表。这对于 best-fit 算法非常有用，可以用来查找最适合所需要的分配大小的缓存（遍历列表）。cache_chain 的每个元素都是一个 kmem_cache 结构的引用（称为一个缓存）。它定义了一个要管理的给定大小的对象池。

每个缓存都包含了一个 slab 列表，这是一段连续的内存块（通常都是页）。存在 3 种 slab：

- slabs_full：完全分配的 slab
- slabs_partial：部分分配的 slab
- slabs_empty：空 slab，或者没有对象被分配

slabs_empty 列表中的 slab 是进行回收（reaping）的主要备选对象。正是通过此过程，slab 所使用的内存被返回给操作系统供其他用户使用。

slab 列表中的每个 slab 都是一个连续的内存块（一个或多个连续页），它们被划分成一个个对象。这些对象是从特定缓存中进行分配和释放的基本元素。注意 slab 是 slab 分配器进行操作的最小分配单位，因此如果需要对 slab 进行扩展，这也就是所扩展的最小值。通常来说，每个 slab 被分配为多个对象。

由于对象是从 slab 中进行分配和释放的，因此单个 slab 可以在 slab 列表之间进行移动。例如，当一个 slab 中的所有对象都被使用完时，就从 slabs_partial 列表中移动到 slabs_full 列表中。当一个 slab 完全被分配并且有对象被释放后，就从 slabs_full 列表中移动到 slabs_partial 列表中。当所有对象都被释放之后，就从 slabs_partial 列表移动到 slabs_empty 列表中。

原始的 slab 算法由 Jeff Bonwick 首先在 SunOS 中实现，后来被移植入 Linux。但是自从 2.6.22 以来，Linux 采用了简化版的 slab——slub。而 ucore 中的 slab 则仅实现了最简单的功能，即减少内碎片。

4.1 分配和释放对象

ucore 中采用静态的 kmem_cache 数组，每一项对应一种对象的大小。

分配对象的入口函数是`kmalloc`，而核心函数为`kmem_cache_alloc`。分配时检查特定缓存是否有空闲 slab，若有则直接取出其中一个空闲对象进行分配；若没有则需要分配新的空闲 slab，再分配对象。

分配 slab 的核心函数为`kmem_cache_grow`。除了设置页面的 slab 属性，还需要对控制结构（一个用数组实现的链表）进行分配（如果控制结构在 slab 外部）和初始化，以保证之后分配对象的正确性和高效率。之后需要将新分配的 slab 加入到缓存的 slab 列表之中。

释放对象的入口函数为`kfree`，而核心函数为`kmem_cache_free`。释放时先获得对象所在的 slab，检查该 slab 是否为满，或者释放后为空，并将 slab 移动到合适的列表中。当 slab 为空时需要释放 slab。

释放 slab 的核心函数为`kmem_cache_destroy`。除了清除页面属性以及释放页面，还需要释放外部控制结构，同时将 slab 移除列表。

5 proj 7 虚拟内存管理

我们现在知道物理内存管理器为对象分配唯一的一个物理内存地址，同时建立好页表的映射关系。但是建立这样一个映射关系除了需要知道物理地址，同时也需要知道虚拟地址。那虚拟地址又由谁分配呢?

内核态中这个问题非常直观，因为虚拟地址和物理地址有一个线性映射关系。但是用户态内存则是非线性映射关系。非线性映射的好处是灵活，坏处当然是麻烦。为了更好地分配虚拟地址空间，我们为每个进程分配一个内存控制结构管理整个虚拟地址空间，它将这个空间分区，叫做虚拟内存区域（Virtual Memory Area，VMA）。和大家熟悉的硬盘分区类似，VMA 可以增大和缩小,一个 VMA 可以分裂为两个更小的 VMA，而两个连续的 VMA 可以合并为一个大的 VMA。用户态内存需要在某个 VMA 中进行分配，而同一个进程下的两个 VMA 是不会重合的，所以保证了同一进程下虚拟地址的唯一性。

5.1 练习 5：处理 Page Fault

通常情况下，用户进程在请求内存时，不会真正得到一个物理页，因为很有可能请求的页根本不会被使用。因此操作系统采用了一种叫做“按需分配”的机制，只为需要使用的页分配并映射一个物理页。操作系统通过缺页中断（Page Fault）来实现这个机制。

Page Fault 是系统的重要中断。当访问没有映射物理地址的虚拟地址时，会触发 Page Fault。另外对没有写权限的虚拟地址进行写操作，以及其他种种原因也会触发 Page Fault（具体参见《Intel® 64 and IA-32 Architectures Software Developer’s Manual – Volume 3A》6-54 部分“Interrupt 14 – Page-Fault Exception (#PF)”）。

触发 Page Fault 后，系统进入中断处理例程。一个 Page Fault 会提供两个信息（除了触发 Page Fault 时的 CPU 上下文）供我们进行处理：存放在栈中的 Error Code 和 cr2 寄存器。Error Code 可以告诉我们是什么原因触发了 Page Fault，而 cr2 寄存器则告诉我们触发 Page Fault 的线性地址（目前也就是虚拟地址）。

实现 Page Fault 中断处理例程需要完成以下几点：

1. 在`trap_dispatch`（`trap/trap.c`）中添加处理 Page Fault 的 case（请参照`trap/trap.h`），调用`pgfault_handler`函数。
2. 完成`do_pgfault`（`mm/vmm.c`）函数，给未被映射的地址映射上物理页。设置访问权限的时候需要参考页面所在 VMA 的权限，同时需要注意映射物理页时需要操作内存控制结构所指定的页表，而不是内核的页表。

6 扩展练习（挑战）

挑战部分不是必做部分，不过在最后会酌情加分。

6.1 Challenge A：实现 Clock 页替换算法

实现 swap out 处理，把不常用的页写到硬盘的 swap 分区（可自己设定）；再次访问到被 swap out 出去的页，则通过 swap in 处理把相关页调入到内存中。（通过网络查询所需信息，可找老师咨询。如果完成，且有兴趣做代替考试的实验，可找老师商量）。需写出详细的设计和分析报告。完成出色的可获得适当加分。（2 周内完成，单独提交）

6.2 Challenge B：实现动态链接库的功能

使得在执行阶段进行 dynamic link（动态链接过程）。（通过网络查询所需信息，可找老师咨询。如果完成，且有兴趣做代替考试的实验，可找老师商量）。需写出详细的设计和分析报告。完成出色的可获得适当加分。（4 周内完成，单独提交）

7 实验报告要求

从网站上下载 lab2.tar.bz2 后，解压得到代码目录 lab2，完成实验。在实践中完成实验的各个练习。在报告中写出自己遇到的问题及解决方案。

实验报告文档命名为 lab2-学生 ID.odt 或者 lab2-学生 ID.txt。推荐用 txt 格式，即基本文本格式，保存时请用UTF-8 编码。对于 proj 5 和 proj 7 中的函数编写任务，完成编写之后，在对应 proj 目录下执行 make handin 任务，即会自动生成 proj5-handin.tar.gz 和 proj7-handin.tar.gz 文件。建立一个目录,名字为 lab2_result，将实验报告文档和之前生成的 handin 文件放在该目录下。然后用 tar 软件压缩打包此目录，并命名为 lab2-学生 ID.tar.bz2（在 lab2_result 的上层目录下执行“tar -cjvf lab2-学生 ID.tar.bz2 lab2_result”即可，请不要用其他软件例如 WinRAR 压缩）。最后请一定提前或按时提交到网络学堂上。

文件内容如下所示，请注意区分大小写

```
lab2-2007011350.tar.bz2
+-- lab2_result/
    |-- lab2-2007011350.txt
    |-- proj5-handin.tar.gz
    +-- proj7-handin.tar.gz
```

代码中所有需要完成的地方都有“LAB2 PROJ*: 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。