Linux 内存管理系统：初始化

作者：Joe Knapka

臭翻：colyli

内存管理系统的初始化处理流程分为三个基本阶段：

激活页内存管理

- 在 swapper_pg_dir 中初始化内核的页表
- 初始化一系列和内存管理相关的内核数据
- Turning On Paging (i386)

启动分页机制(i386)

Kernel 代码被加载到物理地址 0x100000(1MB)，在分页机制打开后被重新映射到 PAGE_OFFSET + 0x100000 的位置（PAGE_OFFSET 在 IA32 上为 3GB，即进程虚拟地址中用户空间与内核空间的分界处）。这是通过将物理地址映射到编译进来的页表 （在 arch/i386/kernel/head.S 中)的 0－8MB 以及 PAGE_OFFSET－PAGE_OFFSET+8MB 实现的。然后我们跳转到 init/main.c 中的 start_kernel，这个函数被定位到 PAGE_OFFSET+某一个地址。这看起来有些狡猾。要注意到在 head.S 中启动分页机制的代码是通过让它自己所执行的地址空间不再有效的方式来实现这一点的；因此 0－4MB 被映射（不明白：hence the 0-4MB identity mapping.）。在分页机制没有启动之前，start_kernel 是不会被调用的，我们假定他运行在 PAGE_OFFSET+某一个地方的位置。因此 head.S 中的页表必须同样映射内核代码所使用的地址，这样后继才能跳转到 staert_kernel 处；因此 PAGE_OFFSET 被映射（不明

白：hence the PAGE_OFFSET mapping.）。

下面在 head.S 中分页机制启动时的一些神奇的代码：

```
/*
 * Enable paging
 */
3:
    movl $swapper_pg_dir-__PAGE_OFFSET,%eax
    movl %eax,%cr3      /* set the page table pointer.. */
    movl %cr0,%eax
    orl $0x80000000,%eax
    movl %eax,%cr0      /* ..and set paging (PG) bit */
    jmp 1f          /* flush the prefetch-queue */
1:
    movl $1f,%eax
    jmp *%eax       /* make sure eip is relocated */
1:
```

在两个 1 的 label 之间的代码将第二个 label 1 的地址加载到 EAX 中，然后跳转到那里。这时，指令指针寄存器 EIP 指向 1MB＋某个数值的物理地址。而 label 都在内核的虚拟地址空间(PAGE_OFFSET+某个位置)，所以这段代码将 EIP 有效的从物理地址空间重新定位到了虚拟地址空间。

Start_kernel 函数初始化了所有的内核数据，然后启动了 init 内核线程。Start_kernel 中

最初的几件事情之一就是调用 setup_arch 函数，这是一个和具体的体系结构相关的设置函数，调用了更底层的初始化细节。对于 x86 平台而言，这些函数在 arch/i386/kernel/setup.c 中。

在 setup_arch 中和内存相关的第一件事就是计算低端内存(low-memory) 和高端内存(high-memory)的有效页的数目；每种内存类型(each memory type)最高端的页的数目分别保存在全局变量 highstart_pfn 和 highend_pfn 中。高端内存并不是直接映射到内核的虚拟内存(VM)中；这是后面要讨论的。

接下来，setup_arch 调用 init_bootmem 函数以初始化启动时的内存分配器(boot-time memory allocator)。Bootmem 内存分配器仅仅在系统 boot 的过程中使用，为永久的内核数据分配页。因此我们不会对它涉及太多。需要记住的就是 bootmem 分配器(bootmem allocator)在内核初始化时提供页，这些页为内核专门预留，就好像他们是从内核景象文件中载入的一样，他们在系统启动以后不参与任何的内存管理活动。

初始化内核页表

之后，setup_arch 调用在 arch/i386/mm/init.c 中的 paging_init 函数。这个函数做了一些事情。首先它调用 pagetable_init 函数去映射整个的物理内存，或者在 PAGE_OFFSET 到 4GB 之间的尽可能多的物理内存，这是从 PAGE_OFFSET 处开始。

在 pagetable_init 函数中，我们在 swapper_pg_dir 中精确的建立了内核的页表，映射到截至 PAGE_OFFSET 的整个物理内存。

这是一个将正确的数值填充到页目录和页表中去的简单的算术活。映射建立在 swapper_pg_dir 中，即 kernel 页目录；这也是初始化页机制时所使用的页目录。（当使用

4MB 的页表的时候，直到下一个 4MB 边界的虚拟地址才会被映射在这里，但这没有什么，因为我们不会使用这个内存所以没有什么问题）。如果有这里有剩下物理内存没有被映射，那就是大于 4GB－PAGE_OFFSET 范围的内存，这些内存只有 CONFIG_HIGHMEM 选项被设置后才能使用（即使用大于 4GB 的内存）。

在接近 pagetable_init 函数的尾部，我们调用了 fixrange_init 为编译时固定的虚拟内存映射预留页表。这些表将硬编码到 Kernel 中的虚拟地址进行映射，但是他们并不是已经加载的内核数据的一部分。Fixmap 表在运行时调用 set_fixmap 函数被映射到物理内存。

在初始化了 fixmap 之后，如果 CONFIG_HIGHMEM 被设置了，我们还要分配一些页表给 kmap 分配器。Kmap 允许 kernel 将物理地址的任何页映射到 kernel 的虚拟地址空间，以临时使用。这很有用，例如对在 pagetable_init 中不能直接映射的物理内存进行映射。

Fixmap 和 kmap 页表们占据了 kernel 虚拟空间顶部的一部分——因此这些地址不能在 PAGE_OFFSET 映射中被永久的映射到物理页上。由于这个原因，Kernel 虚拟内存的顶部的 128MB 就被预留了（vmalloc 分配器仍然是用这个范围内的地址）。（下面这句实在是不知道怎么翻译）Any physical pages that would otherwise be mapped into the PAGE_OFFSET mapping in the 4GB-128MB range are instead (if CONFIG_HIGHMEM is specified) included in the high memory zone, accessible to the kernel only via kmap()。如果没有设置 CONFIG_HIGMEM，这些页就完全是不可用的。这仅针对配置有大量内存的机器（900 多 MB 或者更多）。例如，如果 PAGE_OFFSET=3GB，并且机器有 2GB 的 RAM，那么只有开始的 1GB-128MB 的物理内存可以被映射到 PAGE_OFFSET 和 fixmap/kmap 地址范围之间。剩余的页是不可用的——实际上对于用户进程映射来说，他们是可以直接映射的页——但是内核不能够直接访问它们。

回到 paging_init，我们可以通过调用 kmap_init 函数来初始化 kmap 系统，kmap_init 简

单的缓存了最先的 kmap_pagetable[在 TLB?]。然后，我们通过计算 zone 的大小并调用 free_area_init 去建立 mem_map 和初始化 freelist，初始化了 zone 分配器。所有的 freelist 被初始化为空，并且所有的页都被标志为 reserved（不可被 VM 系统访问）；这种情况之后会被纠正。

当 paging_init 完成后，物理内存的分布如下[注意在 2.4 的内核中这不全对]：

0x00000000: 0-page

0x00100000: kernel-text

0x????????: kernel_data

0x???????? =_end: whole-mem pagetables

0x????????: fixmap pagetables

0x????????: zone data (mem_map, zone_structs, freelists &c)

0x???????? =start_mem: free pages

这块内存被 swapper_pg_dir 和 whole-mem-pagetables 映射以寻址 PAGE_OFFSET。

进一步的 VM 子系统初始化

现在我们回到 start_kernel。在 paging_init 完成后，我们为内核的其他子系统进行一些额外的配置工作，它们中的一些使用 bootmem 分配器分配额外的内核内存。从内存管理的观点来看，这其中最重要的是 kmem_cache_init，他初始化了 slab 分配器的数据。

在 kmem_cache_init 调用之后不久，我们调用了 mem_init。这个通过清除空闲物理页的 zone 数据中的 PG_RESERVED 位在 free_area_init 的开始完成了初始化 freelist 的工作；为不能被用为 DMA 的页清除 PG_DMA 位；然后释放所有可用的页到他们各自的 zone 中。最后

一步，在 bootmem.c 中的 free_all_bootmem 函数中完成，很有趣。他建立了伙伴位图和 freelist 描述了所有存在的没有预留的页，这是通过简单的释放他们并让 free_page_ok 做正确的事情。一旦 mem_init 被调用了，bootmem 分配器就不再使用了，所以它的所有的页也会被释放到 zone 分配器的世界中。

段

段用来将线性地址空间划分为专用的块。线性空间是被 VM 子系统管理的。X86 体系结构从硬件上支持段机制；你可以按照段+段内偏移量的方式指定一个地址，这里地址被描述为一定范围的线性（虚拟地址）并带有特定的属性（如保护属性）。实际上，在 x86 体系结构中你必须使用段机制。所以我们要设置 4 个段：

一个 kernel text 段：从 0 到 4GB

一个 kernel data 段：从 0 到 4GB

一个 user text 段：从 0 到 4GB

一个 user data 段：从 0 到 4GB

因此我们可以使用任何一个有效的段选择器(segment selector)访问整个虚拟地址空间。

问题：

段是在哪里被设置的?

答案：

全局描述符表（GDT）定义在 head.s 的 450 行。 GDT 寄存器在 250 行被加载。

问题：

为什么将内核段和用户端分离开。是否他们都有权限访问整个 4GB 的范围?

答案：

这是因为内核和用户段的保护机制有区别：

```
.quad 0x00cf9a000000ffff        /* 0x10 kernel 4GB code at 0x00000000 */

.quad 0x00cf92000000ffff        /* 0x18 kernel 4GB data at 0x00000000 */

.quad 0x00cffa000000ffff        /* 0x23 user   4GB code at 0x00000000 */

.quad 0x00cff2000000ffff        /* 0x2B user   4GB data at 0x00000000 */
```

段寄存器(CS，DS 等)包含有一个 13 位的描述符表的索引，索引指向的描述符告诉 CPU 所选择的段的属性。段选择器的低 3 位没有被用来索引描述符表，而是用来保存描述符类型（全局或局部）以及需要的特权级。因此内核段选择器 0x10 和 0x18 使用特权级 0（RPL0），用户选择器 0x23 和 0x2B 使用最特权级 RPL 3。

要注意到第三个高序字节的高位组对应内核和用户也是不同的：对内核，描述符特权级（DPL）为 0；对用户 DPL 为 3。如果你阅读了 Intel 的文档，你将看到确切的含义，但是由于 Linux 内核的 x86 段保护没有涉及太多，所以我就不再讨论太多了。