第 8 章 内存

内存是内核所管理的最重要的资源之一。某进程区别于其它进程的一个特征是两个进程存在于逻辑上相互独立的内存空间（与之相反，线程共享内存）。即使进程都是同一程序的实例，比如，两个 xterm 或两个 Emacse，内核都会为每个进程安排内存空间，使得它们看起来像是在系统之上运行的唯一进程。当一个进程不可能偶然或恶意的修改其它进程的执行空间时，系统的安全性和稳定性就会得到增强。

内核也生存在它自己的内存空间之中，即内核空间（kernel space）。与之对应的是用户空间（user space），它是所有非内核任务所处的内存空间的一个通用术语。

虚拟内存

计算机系统包括不同级别的存储器。图 8-1 说明了这些存储器中最重要的几项，并且以我自己原有的 Linux 机器（Linux box）为例标注了一些参数的估计值。当你从左向右观察该图时，会发现存储器容量越来越大而速度却越来越慢（而且每字节价格也会更低）。尤其令人注意的是，访问速度跨越了 3 个数量级（乘数因子为 1000），而容量竟跨越了超过 8 个数量级（乘数因子为 312500000）。（实际上有时速度的差异是可以被掩盖的，不过这些数字足以很好的说明这一部分讨论的目的。）最大的差距体现在最后两个：RAM 和磁盘上，它们又分别可被称作主存和辅存。

额外附加的存储器空间总是十分诱人的，即使它们也很慢。如果在 RAM 被用完时，通过暂时把不用的代码和数据转移到磁盘上以腾出更多空间的方法来使用磁盘代替 RAM 的话，那将是很好的一件事情。正如读者可能已经知道的，Linux 恰好能够做到这一点，这被称之为虚拟内存（virtual memory）。

虚拟内存是一种对 RAM 和磁盘（或称之为：主存和辅存）进行无缝混合访问的技术。所有这些（虚拟）内存对于应用程序来说就好像它真的存在一样。当然我们知道它并非真的内存，这正是为什么它被称为是“ 虚拟的 ”，但是多亏了内核使得应用程序无法分辨出它们的区别。对于应用程序来说，就好像真的有很大数量的 RAM，只不过有时候比较慢而已。

术语“ 虚拟内存 ”还有另外一层意思，从严格意义来讲是与前述的第一种意思没有关系的。这里的虚拟内存指的是对进程驻留地址进行欺骗的方法。每个进程都会有这样一种错觉，认为它的地址是从 0 开始并由此连续向上发展的。很明显，这一点同时对所有进程都成立是不可能的，但是在生成代码的时候这个假定（fiction）却能够带来很大方便，这是由于进程不必知道它们是否真正从 0 地址开始驻留，而且它们也不必去关心此事。

这两种意思也不必相关，因为一个操作系统从理论上可以给每个进程分配一个独有的逻辑地址空间而不用混合使用主存和辅存。然而在所有我已经知道的系统中（对这两种虚拟内存的实现方式）要么都采纳要么都不采纳，这一点可能会在开始时令人感到困惑。

为了避免这种意义上的分歧，有人倾向于术语“ 虚拟内存 ”代表逻辑地址空间（logical-address-space）的意义，同时使用“ 分页（paging）”或“ 交换 ”表示磁盘作为内存使用（disk-as-memory）的含义。尽管这种严格的区分具有充足的理由，但是我更喜欢普通的用法。除非上下文要求，否则我很少花费精力对它们进行区分。

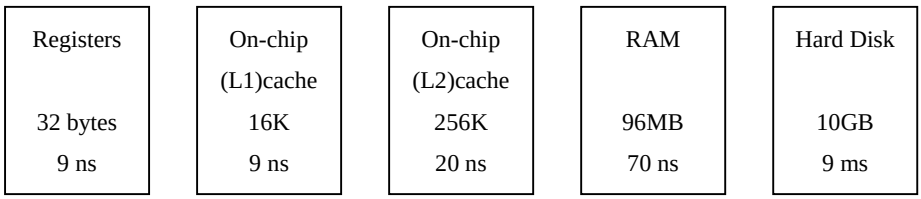

图 8-1 具有速度和容量的存储级别

交换和分页

早期的虚拟内存（VM）系统仅能够把整个应用程序代码和数据，即完整的进程从磁盘上移出或移入磁盘。这种技术被称为交换（swapping），因为它是把一个进程同另一个进程进行了对调。出于这个原因，磁盘上为 VM 所保留的区域通常被称为交换空间（swap space），或简称为交换区（swap），尽管如我们所见，现代的系统已不再使用这种最初意义上的交换技术。与此类似，读者通常会见到的术语是交换设备（swap device）和交换分区（swap partition），它是磁盘分区的同义词，但是被专门作为交换空间使用，以及术语交换文件（swap file），这是一个用于交换的规则的、有固定长度的文件。

交换是很有用的，当然要比根本没有 VM 好的多，但是它也有一定局限性。首先，交换需要把整个进程同时调入内存，所以当运行一个需要比系统所有 RAM 还要大的存储空间的进程时，交换便于事无补了，即使磁盘有大量空间可供补充。

其次，交换可能会很低效。交换就必须把整个进程同时调出，这就意味着为了 2K 的空间你不得不把一个 8MB 的进程整个调出。同样的道理，即使仅仅需要执行被调进的应用程序代码的一小部分，你也必须把整个进程同时调进。

分页（paging）是把系统的内存划分成很小的块，即页面，每个页面可以独立的从磁盘调入或调出磁盘。分页与交换技术相似，但它使用更加细小的粒度（granularity）。分页比交换有更多的登记（book-keeping）开销，这是因为页面数远比进程数要多，然而通过分页可以获得更多的灵活性。而且分页也更快一些，原因之一就是不再需要把整个进程调进调出，而只需要交换必要的页面就足够了。要记住前述的 1000 倍的速度差异，所以我们应该尽可能避免磁盘的 I/O 操作。

传统上特定平台上页面的大小是固定的，比如 x86 平台为 4K，这可以简化分页操作。不过，大多数 CPU 为可变大小的页面提供硬件支持，通常能够达到 4M 或者更大。可变大小页面可以使分页操作执行更快和更有效，不过要以复杂性为代价。标准发行的 Linux 内核不支持可变大小页面，所以我们仍然假定页面大小是 4K。（已经有支持 Cyrix 可变大小页面机制的补丁程序，但它们不是本书中官方发行版本的部分。而且据闻由此获得的性能增益也并不非常显著。）

因为分页可以完成交换所能完成的所有工作，而且更加有效，所以类似于 Linux 一样的现代操作系统已不再使用交换，严格的说是只使用分页技术。但是术语“交换”已得到了广泛使用，以至于实际应用中术语“交换”和“分页”已经几乎可以通用；由于内核使用分页技术，所以本书就遵从这种用法。

Linux 能够交换到一个专用磁盘分区、或一个文件，或是分区和文件的不同组合。Linux 甚至允许在系统运行时增加和移去交换空间，当你暂时需要额外大量的交换空间，或者假如你发现需要额外交换空间而又不想重启系统的时候，这就会很有用了。另外，与一些 Unix 的风格（flavors）不同，Linux 即使没有任何交换空间也能运行得很好。

地址空间

地址空间（address space）是一段表示内存位置的地址范围。地址空间有三种：

- 物理地址空间
- 线性地址空间
- 逻辑地址空间，也被称为虚拟地址空间

（需要指出的是，I/O 地址能够被看作是第四种地址空间，但是本书中对其不作讨论。）

物理地址是一个系统中可用的真实的硬件地址。假如一个系统有 64M 内存，它的合法地址范围是从 0 到 0x4000000（以十六进制表示）。每个地址都对应于所安装的 SIMMs 中的一组晶体管，而且对应于处理器地址总线上的一组特定信号。

分页可以在一个进程的生存期里，把它或它的片段移入或者移出不同的物理内存区域（或不同物理地址）。这正是进程被分配一个逻辑地址空间的原因之一。就任何特定的进程来说，从 0 开始扩展到十六进制地址 0xc0000000 共 3GB 的地址空间是绰绰有余的。即使每个进程有相同的逻辑地址空间，相应进程的物理地址也都是不同的，因此它们不会彼此重叠。

从内核的角度看来，逻辑和物理地址都被划分成页面。因此，就像我们所说的逻辑和物理地址一样，可以称它们为逻辑和物理页面 :每个合法的逻辑地址恰好处于一个逻辑页面中，物理地址也是这样的。

与之相反，线性地址通常不被认为是分页的。CPU（实际是下文中的 MMU）会以一种体系结构特有的方式把进程使用的逻辑地址转换成线性地址。在 x86 平台上，这种转换是简单地把虚拟地址与另一地址，即进程的段基址相加；因为每个任务的基址都被设置为 0，所以在这种体系结构中，逻辑地址和线性地址是相同的。得到的线性地址接着被转换成物理地址并与系统的 RAM 直接作用。

内存管理单元

在逻辑地址和物理地址之间相互转换的工作是由内核和硬件内存管理单元（MMU—memory management unit）共同完成的。MMU 是被集成进现代的 CPU 里的，它们都是同一块 CPU 芯片内的一个部分，但是把 MMU 当作一个独立的部分仍然非常有益。内核告诉 MMU 如何为每个进程把某逻辑页面映射到某特定物理页面，而 MMU 在进程提出内存请求时完成实际的转换工作。

当地址转换无法完成时，比如，由于给定的逻辑地址不合法或者由于逻辑页面没有对应的物理页面的时候，MMU 就给内核发出信号。这种情况称为页面错误（page fault），本章后面会对此进行详细论述。

MMU 也负责增强内存保护，比如当一个应用程序试图在它的内存中对一个已标明是只读的页面进行写操作时，MMU 就会通知 OS。

MMU 的主要好处在于速度。缺少 MMU 时为了获得同样的效果，OS 将不得不使用软件为每个进程的每一次内存引用进行校验，这种校验同时包括数据和指令在内，而这可能还包括要用为进程创建其生存所需的虚拟机。（Java 所进行的一些工作与此类似。）这样做的结果将使系统慢得令人无法忍受。但是一个以这种内存访问合法性检查方式集成在计算机硬件里的 MMU 却根本不会使系统变慢。在 MMU 建立起一个进程以后，内核就只是偶尔参与工作，例如在发生页面错误时，而这与全部内存引用数量相比是非常少的。

除此而外，MMU 还可以协助保护内存自身。没有 MMU，内核可能不能够防止一个进程非法侵入它自己的内存空间或者是其它进程的内存空间。但是如何避免内核也会作同样的

操作呢？在 Intel’s 80486 或更新的芯片上（不是 80386），MMU 的内存保护特性也适用于内核进程。

页目录和页表

在 x86 体系结构上，把线性地址（或者逻辑地址——记住在 Linux 上，这二者具有相同的值）解析（resolving）到物理地址分为两个步骤，整个过程如图 8-2 所示。提供给进程的线性地址被分为三个部分：一个页目录索引，一个页表索引和一个偏移量。页目录（page directory）是一个指向页表的指针数组，页表（page table）是一个指向页面的指针数组，因此地址解析就是一个跟踪指针链的过程。一个页目录使你能够确定一个页表，继而得到一个页面，然后页面中的偏移量（offset）能够指出该页面里的一个地址。

为了进行更详细因而也会更准确的描述：给定页目录索引中的页目录项保存着贮存在物理内存上的一个页表地址；给定页表索引中的页表项保存着物理内存上相应物理页面的基地址；然后线性地址的偏移量加到这个物理地址上形成最终物理页面内的目的地址。

其它 CPU 使用三级转换方法，如图 8-3 所示。这在 64 位体系中尤其有用，以 Alpha 为例，其更大的 64 位的地址空间意味着类似于 x86 体系的地址转换将要求大量的页目录、大量页表、大量偏移量，或三者兼有。对于这种情况，Alpha 的设计者们向线性地址模式中引入了另一层次，即 Linux 所称的页面中间目录（page middle directory），它位于页目录和页表之间。

这个方案与以前实际是一样的，只不过多增加了一级。这种三级转换方法同样具有页目录，页目录的每一项包含一个页面中间目录的入口地址，页面中间目录的每一项包含一个页表的入口地址，而页表也同以前一样每一项包含物理内存中一个页面的地址，这个地址再加上偏移量就得到了最终的地址。

而使情况更为复杂的是，通过进一步观察可知，三部分地址模式与两级地址转换是相关联的，而四部分地址模式则与三级地址转换相关联的，这是由于我们通常所说的“级（或层次 levels）”不包括索引到页目录的第一步（我想是因为这一步没有进行转换的缘故）。

令人奇怪的是内核开发者们决定只用其中一种模式来处理问题。绝大部分的内核代码对 MMU 一视同仁，就如同 MMU 都使用三级转换方法（也就是四部分地址模式）一样。在 x86 平台上，通过将页面中间目录定义为 1，页面相关的宏可以把三级分解过程完美地转换到二级分解过程上去。这些宏认为页面中间目录和页目录是几乎可以进行相互替换的等价品，以至于内核的主要代码可以认为其地址就是由四个部分组成的。

在 x86 系统中，32 位地址中 10 位是页目录索引，接下来 10 位是页表索引，剩下的 12 位用作偏移量，这就构成了 4K 大小的页面（2^{12} 等于 4096 个偏移量）。

用于创建和操作每一级项的函数和宏定义在头文件 include/asm-i386/page.h（第 10786 行）和 include/asm-i386/pgtable.h（第 10876 行）之中。在读者浏览这些函数和宏的时候，记住 PGD 通常代表“页目录项（page directory entry）”（不只是“页目录”），PMD 通常代表“页面中间目录项（page middle directory entry）”（不只是“页面中间目录”），同样 PTE 也通常代表“页表项”。而且，正如上面解释中限定词“通常”所暗示的那样，例外是存在的，例如下文将要提到的 **pte_alloc** 就分配页表而不是（如你所可能会认为的）页表项。非常遗憾的是，由于篇幅所限我们不能对全部例程进行讨论，我们将在后面对其中的一部分进行讨论。

页表项不仅记录了一个页面的基地址，而且记录了它的保护信息（protections），也就是一组指定该页为可读、可写，和/或可执行的标志（这容易让人联想到文件的保护位）。随着我们对页面保护信息的进一步剖析，读者会看到页表项所包括的其它页面特有的标志。

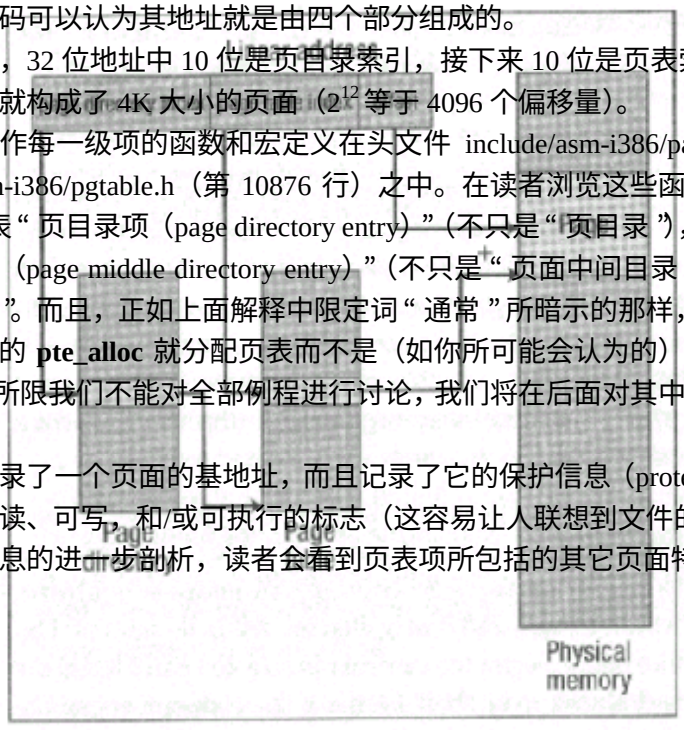

Figure 8.2 Paging on the x86.

转换后备缓存（Translation Lookaside Buffers：TLBs)

如果简单的执行从线性地址到物理地址的转换过程，在跟踪指针链时将会需要几个内存引用。RAM 虽然不像磁盘那么慢，但是仍然比 CPU 要慢的多，这样就容易形成性能的瓶颈。为了减少这种开销，最近被执行过的地址转换结果将被存储在 MMU 的转换后备缓存（translation lookaside buffers：TLBs）内。除了偶尔会通知 CPU，由于内核的某操作致使 TLBs 无效之外，Linux 不用明确管理 TLBs。

在作用于 TLB 的函数和宏中，我们只研究一下**__flush_tlb**，在 x86 平台上，它是其它大部分函数和宏的基础。

__flush_tlb

10917：CR3（控制寄存器 3）是 x86CPU 寄存器，它保存页目录的基地址。往这个寄存器送入一个值将会使 CPU 认为 TLBs 变成无效，甚至写入与 CR3 已有值相同的值也是这样。

因此，**__flush_tlb** 仅是两条汇编程序指令：它把 CR3 的值保存在临时变量 **tmpreg** 里，然后立刻把 **tmpreg** 的值拷贝回 CR3 中，整个过程就这么简单！

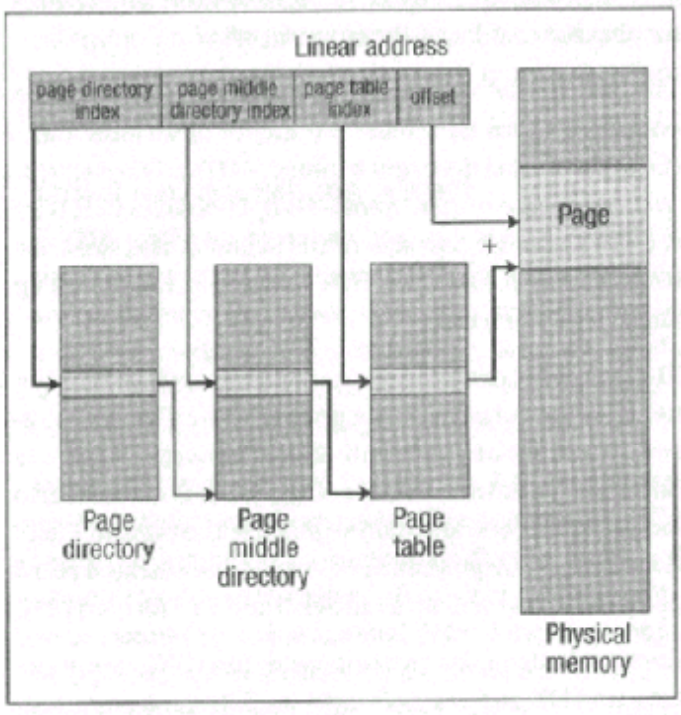

Figure 8.3 The kernel's generic view of paging.

注意 x86 系统也允许使某一个单独的 TLB 项无效，而并不一定非要使全部项，这种方法使用 **invipg** 指令——参见第 10926 行它的使用信息。

段

由于段不是在所有 CPU 中均可用，所以 Linux 内核中与体系结构无关的部分不能对段进行辨识。在不同的 CPU 体系中，段的（如果段在体系中是可用的）处理方式大相径庭，这一点是非常重要的。因此，我们在这个问题上不会花费太多时间，不过 x86 系统上内核使用段的方式还是值得大概讨论一下的。

段可以被简单的看作是定义内存区域的另一种机制，有些类似于页。这两种机制可以重叠：地址总是在页面之内，也可能处于段内。与页不同，段的大小可以变化，甚至在其生存期里能够增长和收缩。与页相同的是，段可以被保护，而且其保护可由硬件实施；当 x86 的段保护和同一地址的页保护发生冲突时，段保护优先。

X86 系统使用一些寄存器和全局描述表（GDT）和局部描述表（LDT）这两种表来对段进行跟踪。描述符（descriptor）是段的描述信息，它是用来保存段的大小和基址以及段的保护信息的 8 字节的对象。GDT 在系统中只有一个，而 Linux 可以为每个任务建立一个 LDT。

接下来我们将简单解释内核是如何使用这些表来建立段的。内核本身拥有分离的代码和数据段，它们被记述在 GDT 的第 2 和第 3 行项里。每个任务也有分离的代码和数据段。当前任务的代码段和数据段不仅在它自己的 LDT 的第 0 和第 1 行项被说明，而且还被记述在 GDT 的第 4 和第 5 行项里。

在 GDT 里，每个任务占两行项，一个用来定位它的 LDT，一个用来定位它的 TSS（前面章节曾简要提及的任务状态段）。因为 x86CPU 限制 GDT 的大小为 8192 个项，而且 Linux 为每个任务占用两行 GDT 项，因此显而易见的是我们不能拥有超过 4096 个任务，这也正是在第 7 章里提到的限制。事实上，任务的最大数目要稍小一点儿，不过仍有 4090 个，这是由于 GDT 的前 12 行项被保留用于其它目的。

富有经验的 x86 程序员可能已经注意到 Linux 所使用的 x86 分段机制是采用最低限度方式的；段的主要使用目的仅是为了避免用户代码出现在内核段中。Linux 更倾向于分页机制。从大的方面来看，对于处理器来说分页或多或少都是相同的，或者说总的事实就是这样，因此内核越是以分页方式工作，它的可移植性就越好。

最后要提及的是，如果读者对于 x86 的分段机制很感兴趣的话，不妨阅读一下 Intel 体系结构下的软件开发手册第 3 卷（Intel Architecture Software Developer’s Manual Volume 3），该书可以从 Intel 站点上免费得到（**developer.intel.com/design/pentiumii/manuals/243192.htm**）。

进程的内存组织

有三个重要的数据结构用于表示进程的内存使用：**struct vm_area_struct**（第 15113 行）、**struct vm_operations_struct**（第 15171 行），和 **struct mm_struct**（第 16270 行）。我们随后将对这三个数据结构进行逐一介绍。

Struct vm_area_struct

内核使用一个或更多的 **struct vm_area_struct** 来跟踪进程使用的内存区域，该结构体通常缩写为 VMAs。每个 VMA 代表进程地址空间的一块单独连续的区间。对于一个给定的进程，两个 VMAs 决不会重叠，一个地址最多被一个 VMA 所覆盖；进程从未访问过的的一个地址将不会在任何一个 VMA 中。

两个 VMA 之间的区别有两个特征：

- 两个 VMA 可以不连续（Two VMAs may be discontiguous）——换句话说，一个 VMA 的末尾不一定是另一个的开头。
- 两个 VMA 的保护模式可以不同（Two VMAs may have different protections）——例如，一个是可写的而另一个可能是不可写的。即使两个这样的 VMA 是连在一起的，它们也必须被分开管理，因为其不同的保护信息。

应注意的一个重点是，一个地址可以被一个 VMA 所覆盖，即使内核并没有分配一个页面来存贮这个地址。VMA 的一个主要应用就是在页面错误时决定如何作出反应。我们可以将 VMAS 看作是一个进程所占用的内存空间以及这些空间的保护模式的总体视图。内核能够反复重新计算从页表而来的 VMA 中的大部分信息，不过那样速度会相当慢。

进程的所有 VMA 是以一个排序的双向链表方式存储的，并且它们使用自己的指针来管理该列表。当一个进程有多于 **avl_min_map_count** 数目（在第 16286 行定义为 32）的 VAM 时，内核也会创建一个 AVL 数来存储它们，此时仍然是使用 VMAs 自己的指针对该树进行管理。AVL 树是一个平衡二叉树结构，因此这种方法在 VMA 数量巨大时查找效率十分高。不过，即使在 AVL 树被创建后，线性列表也会被保留以便内核即使不使用递归也能轻松的遍历一个进程的所有 VMA。

Struct_vm_area_struct 的两个最重要的元素是它的 **vm_start** 和 **vm_end** 成员（分别在第 15115 行和 15116 行），它们定义了 VMA 所覆盖的起止范围，其中 **vm_start** 是 VMA 中的最小地址，而 **vm_end** 是比 VMA 最大地址大一位的地址。在本章后面我们会反复提及这些成员。

注意，**vm_start** 和 **vm_end** 的类型是 **unsigned long**，而不是读者可能会认为的 **void***。由于这个原因，内核在所有表示地址的地方都使用 **unsigned long** 类型，而不用 **void***类型。采用这种方法的部分原因是可以避免因内核对诸如比特一级的地址进行计算操作时引起的编译警告，还可能避免由于它们而偶然引起的间接错误。在引用内核空间的一个数据结构的地址时，内核代码使用指针变量；在对用户空间的地址进行操作时，内核却频繁的使用 **unsigned long**——实际上，几乎只有本章中所涉及的代码才是这样。

这样就给用来编译内核的编译器提出了要求。使用 **unsigned long** 作为地址类型就意味着编译器必须使 **unsigned long** 的类型长度和 **void***的一样，尽管实践中对这一点的要求不是十分严格。对于 x86 寄存器上的 gcc 来说，两种类型很自然的都是 32 位长。在 64 位指针长度的体系中，比如 Alpha，gcc 的 **unsigned long** 类型长度通常也是 64 位。尽管如此，在将来的体系结构上，gcc 的一个端口可能提供与 **void***不同的 **unsigned long** 类型长度，这是需要内核的移植版本开发人员（kernel porters）注意的一点。

还要说明的是，除了 gcc 之外你不需要对编译器的性能有太多担心，因为其它大部分与 gcc 相关的特性都已经包括在代码之中了。假如读者试图用某个其它的编译器来编译内核的话，我想有关 **unsigned long** 和 **void***长度的错误将会占编译错误列表的绝大多数。

Struct vm_operations_struct

一个 VMA 可能代表一段平常的内存区间，就像是 **malloc** 函数所返回的那样。但是它也可以是对应于一个文件、共享内存、交换设备，或是其它特别的对象而建立的一块内存区域；这种对应关系是由本章后面将要涉及的称为 **mmap** 的系统调用所确定的。

我们不想牵扯太多关于 VMA 可以被映射的每一种对象的专门知识，这会使对内核代码的剖析变得凌乱不堪，因为那样就不得不反复决定是否要关闭一个文件、分离共享内存等等令人非常头疼的事情。与此不同，对象类型 **struct vm_operations_struct** 抽象了各种可能提供给被映射对象的操作，比如打开、关闭之类。一个 **struct vm_operations_struct** 结构体就

是一组函数指针，它们之中可能会是 **NULL** 用来表示一个操作对某个被映射对象是不可用的。举例来说，在一个共享内存没有被映射的情况下，把该共享内存对象的页面与磁盘进行同步是没有意义的，表示共享内存操作的 **struct vm_operations_struct** 里的 **sync** 成员就是 **NULL**。

总之，一旦 VMA 映射为一个对象，那么它的 **vm_op**s 成员就会是一个非空的指针，指向一个表示被映射对象所提供操作的 **struct vm_operations_struct** 结构体。对于 VMA 可以映射的每一种对象类型，都有一个该 VMA 可能会在某处指向的静态 **static struct vm _ operations_struct** 结构体。参见第 21809 行这样的一个例子。

Struct mm_struct

一个进程所保留的所有 VMA 都是由 **struct mm_struct** 结构体来管理的。指向这种结构类型的指针在 **struct task_struct** 中，确切的说，它就是后者的 **mm** 成员。这个成员被前一章中所讨论的 **goodness**（第 26388 行）应用，来判断是否两个任务是在同一个线程组中。两个具有相同 **mm** 成员（正如我们所见到的）的任务管理同一块全局内存区域，这也是线程的一个特点。

struct mm_struct 结构体的 **mmap** 成员（第 16271 行）就是前述的 VMA 的链接列表，而它的 **mmap_avl** 成员，如果非空，就是 VMA 的 AVL 树。读者可以浏览 **struct mm_struct** 的定义，会发现它还包括相当多的其它成员，它们中的几个会在本章中涉及到。

VMA 的操作

本小节介绍后面要用到的 **find_vma** 函数，并捎带简介它的同类函数 **find_vma_prev**。这将阐明 VMA 处理操作的一些方面，也为读者将要接触的代码做准备。

find_vma

33460：简单说来，**find_vma** 函数的工作就是找到包含某特定地址的第一个 VMA。更准确的说，它的工作是找到其 **vm_end** 比给定地址大的第一个 VMA，这个地址可能会在该 VMA 之外，因为它可以比 VMA 的 **vm_start** 要小。这个函数返回指向 VMA 的指针，如果没有满足要求的 VMA 就返回 **NULL**。

33468：首先，通过使用 **mm** 的 **mmap_cache** 成员，满足进程最近一次请求的同一 VMA 会被检查，而 **mmap_cache** 正是为此目的而设。我没有亲自测试过，不过这个函数的文档中说高速缓存的命中率可以达到 35%，考虑到高速缓存只由一个 VMA 组成，那么这个数字就相当好了。当然，著名的、被称之为“引用的局部性（locality of reference）”的特性一直在其中提供了很大帮助，这也是软件访问数据（和指令）时的一条原则，即访问最近使用过的数据（和指令）。由于 VMA 包含一块连续的地址区间，引用的局部性就使得所需的地址都在同一个 VMA 中变为可能，而这样的 VMA 就会满足前面的要求。

在修改 VMA 列表的其它几个地方，这个高速缓存的值被设为空，表明对 VMA 列表所做的修改可能会使它失效。至少在一个这种情况中，第 33953 行，使该高速缓存为空不总是必要的；这段代码如果能够再聪明一些的话，就可能从本质上改善高速缓存的命中率。

33471：高速缓存没有命中。假如没有 AVL 树，**find_vma** 只是搜索列表上的所有 VMA，然后返回第一个符合条件的 VMA。回想一下 VMA 的列表是保持顺序的，所以符合条件的 VMA 也就是所有符合条件的 VMA 中地址最小的一个。假如搜索到列表的末

尾都没有一个匹配，**vma** 就被置为 **NULL**，并被返回。

33476：若有大量 VMA，沿树遍历就比沿链表遍历要快；由于 AVL 树是平衡的，这就是一种对数时间操作而不是线性时间操作。

树的迭代遍历并不是十分少见的现象，不过一些特征也并不非常明显。首先注意第 33484 行的赋值；这个操作一直跟踪当前找到的最好节点，当不能找到更好的时，它就会被返回。接着的下一行中的 **if** 语句是一个最优测试，检测 **addr** 是否处于 VMA 中（我们已知的一点是 **addr** 小于 VMA 的 **vm_end**）。因为 VMA 绝对不会彼此覆盖，没有其它 VMA 将是一个较贴近的匹配结果，所以树的遍历可以早些结束。

33492：如果在树的遍历或列表搜索过程中找到一个 VMA，找到的值就被保存在高速缓存里以便下一次查找。

33496：在任何情况下，**vma** 都被返回；它的值或者是 **NULL**，或者是满足查找条件的第一个 VMA。

Find_vma_prev

如前所述，这个函数（从第 33501 行开始）和 **find_vma** 函数是一样的，不过它还会额外的返回一个指向前一个限定的 **addr**（如果有）的 VMA 的指针。这个函数不仅是因为它本身的缘故而令人感兴趣，更主要是由于它的出现会告诉我们一些关于内核程序设计，特别是关于 Linux 内核程序设计的信息。

应用程序员很可能已经在更加通用的 **find_vma_prev** 函数之上写出了 **find_vma** 函数，这只需简单的把指向 VMA 的指针去掉即可，代码如下：

p504.1

应用程序员这样做的原因是具有代表性的应用程序并不太拘于速度因素。这并非纯粹是在为铺张浪费找借口，而是由于 CPU 速度的不断增加使得应用能够更关注于其它方面，我们现在可以出于可维护性的充分理由而提供一个可以到处使用的额外函数调用。

与之相反，一个内核程序员可能不会随便增加多余的函数调用；试着减少几个 CPU 时钟周期会被认为比负责维护某个近乎是副本的函数要更胜一筹。即使没有其它原因，我们也可以说内核开发者所持有的这种态度就是为了让应用程序员能相对自由一些。

为什么这种重复对于源代码相对封闭的操作系统，Linux 而言不那么重要，这里是否有更深层次原因呢？尽管 Linux 内核必须限制它占用的 CPU 时间，Linux 内核的开发工作却不受程序员时间的限制。（明确的说，我必须要指出 Linux 的开发者不必把他们的时间浪费在会议上的，他们也不必被人工制订的时间表所拘束。）正是由于这众多的队伍，众人的智慧，才改变了软件开发的规则。

Linux 内核的源代码对任何人都是公开的，Linus 本人曾说过的一句名言是“ …只要眼睛够多，所有的臭虫（程序错误）都是浅薄的[1] ”。就算函数 find_vma 和 find_vma_prev 的执行会产生重大差异，在你能想到“ 重编译 ”之前，不知什么地方的某个 Linux 内核开发者就已经迅速发现并修复了这个问题。实际上，Linux 内核开发者比它的商业对手动作快得多，所得到的代码运行更快而且错误更少，尽管有时偶然出现的结构会被认为在任何其它环境中都不可维护。

当然，如果没有人负责对这些函数的改进进行维护的话，我认为这也是非常愚蠢的。内核的下一个发布版本就把它们合并了。但是我仍然对此持怀疑态度，而且即使我在这个具体问题上所持的态度并不正确，我依然在总体上保持原有态度。 不同的事还会继续不同，而不同正是 Linux 之所以为 Linux 的一方面。

[1] 原文为：”…given enough eyeballs, all bugs are shallow.”

分页

本章前面对分页已作了概要描述，现在我们进一步来研究 Linux 是如何处理分页的。

页面保护详述

正如早先提及的，页表项不仅保存了一个页面的基地址，还有其它一些标志信息，这些标志指出了该页面上所能进行的操作。现在是仔细研究一下这些标志的时候了。

如果页表项只保存一个页面的基地址，并且页面是页对准的（page-aligned），这个地址的低 12 位（x86 系统），即偏移量部分通常将总是为 0。取代这些位置 0 的作法是把它们编码作为与页面有关的标志，在获取地址时只需简单的把它们屏蔽掉就行了。以下就是这 12 位中的标志位：

- **_PAGE_PRESENT** 位（第 11092 行），若置位，当前页面物理存在于 RAM 中。
- **_PAGE_RW** 位（第 11093 行），置为 0 表示该页面是只读的，置为 1 表示可读可写。因此，没有只写的页面。
- **_PAGE_USER** 位（第 11094 行），置位表示某页面是用户空间页面，清空表示为内核空间页面。
- **_PAGE_WT** 位（第 11095 行），置为 1 表示页面高速缓存管理策略是透写，置为 0 表示管理策略是回写。透写（writethrough）会立刻把写入高速缓存的数据复制（拷贝）到主存储器内，即使保存在高速缓存的数据仍是读访问。与之相反，回写（writeback）具有更高的效率，写入高速缓存的数据仅当其必须为其它数据腾出空间，而必须移出时才被复制到主存储器内。（这是由硬件，而不是 Linux 完成的。）尽管直到本书写作之时，这个标志位在内核中的使用还并不非常普遍，不过这种情形有望很快改变。有时候，Intel 公司的处理器资料中把 WT 位更多的称为 PWT。
- **_PAGE_PCD** 位（第 11096 行），关闭页面高速缓存；本书中的代码不总是使用这个标志位。（缩写“ CD ”表示“ caching disabled ”。）如果我们恰好知道一个不经常使用的页面，那么就不必为它设置高速缓存，这可能会更有效率。这个标志位好像对于映射内存的 I/O 设备来说更有用处，尽管我们想确保对表示设备的内存进行的写操作不被高速缓存缓冲，但是取而代之的作法是立刻把数据直接拷贝到设备之中。
- **_PAGE_ACCESSED** 位（第 11097 行），若置位表示该页面最近曾被访问过。Linux 可以设置或清除这个标志，不过通常这是由硬件完成的。因为清除了该标志的页面已很久未被使用过，所以它们会在交换时被优先调出主存。
- **_PAGE_DIRTY** 位（第 11098 行），若置位，表明该页面的内容自从上次该位被清除后已发生改变。这就意味着它是一个内容没有保存的页面，就不能简单的为交换而被删除。当一个页面第一次写入内存时，该标志位由 MMU 或 Linux 设置；当这个页面调出内存时，Linux 要读取它的值。
- **_PAGE_PROTNONE** 位（第 11103 行），是一个以前的页表项没有使用过的标志位，用来跟踪当前页面。

_PAGE_4M 位和**_PAGE_GLOBAL** 位在同一个**#define** 定义块中出现，但是由于它们不像其它标志位那样用于页面级的保护，所以我们在此不予讨论。

随后的文件中，上述这些标志位被组合在一个高级宏内。

写拷贝（copy-on-write）

提高效率的一条路就是偷懒——只做最少量的必要工作，而且只在不得不做的时候才完成。现实生活中这可能是个坏习惯，至少它会导致拖拖拉拉。而在计算机的世界里，它可能更是一种优点。写拷贝（Copy-On-Write）就是 Linux 内核一种通过懒惰来获得效率的方法。其基本思想是把一个页面标记为只读，却把它所含的 VMA 标识为可写。任何对页面的写操作都会与页级保护相冲突，然后触发一个页面错误。页面错误处理程序会注意到这是由页级保护和 VMA 的保护不一致而导致的，然后它就会创建一个该页的可写拷贝作为代替。

写拷贝十分有用。进程经常 **fork** 并立刻 **exec**，这样为 **fork** 而复制的进程页面会造成浪费，因为 **exec** 之后它们会不得不被抛弃。正如读者所见，进程分配大量内存时也使用同样的机制。所有被分配的页面都与一个单独的空白页面相映射，这就是写拷贝的原意。向某页面的第一次写操作会触发页面错误，然后空白页面执行复制。用这种办法，只有页面分配不能再延期时，它才会被分配。

页面错误

到现在为止，本章已几次提到一个页面可以不在 RAM 里的可能性——毕竟，如果页面总是在内存里，虚拟内存就没什么必要了。但是我们还没有详细介绍过当某页面不在 RAM 中会怎样。当处理器试图访问一个当前不在 RAM 中的页面时，MMU 就会产生一次页面错误，而内核会尽力解决它。在进程违反页级保护时，页面错误也会产生，例如进程试图向只读内存区域写入。

因为任何无效内存访问都会导致页面错误，同样的机制支持请求分页。请求分页（Demand paging）的意思是只有在页面被引用的时候才从磁盘上读取它们——即按需分配——这是另一种通过懒惰来获得效率的方法。

特别地，请求分页用于实现被请求页面的可执行化。为了达到这个目的，应用程序第一次被装载时，只有一小部分可执行映象（image）被读入物理内存，然后内核就依靠页面错误来调入需要的（比如，进程首次跳转到一个子例程时）可执行页面。除了一些意外的情况，这样做总是要比把所有部分一次读入要快，这是因为磁盘较慢，而且并不是所有的程序都会用到的。事实上，因为一个大程序运行一次时，大部分功能特性都不会再用到，所以通常根本不需要全部都读入（这一点对大多数中小规模的程序也是成立的）。这对于按需分页（demand-paged）的可执行程序稍有不同——如果你对这种情况进行考虑的话，你就可以知道按需分页还需要二进制处理程序的支持，而且它是一个具有决定意义的部分。

Do_page_fault

6980： **do_page_fault** 是内核函数，产生页面错误时（在第 363 行）被调用。当页面错误产生时，CPU 调整进程的寄存器，当解决页面错误时，进程再从引起错误的指令处开始执行。通过这种方法，在内核使得冲突地内存访问操作完成后，会自动重试该操作。相反，如果页面错误仍然无法解决，内核就通知引起冲突的进程。当页面错误是由内核本身导致的时候，所采用的措施是近似的，但并不完全相同。

6992： 控制寄存器 2（CR2）是 Intel CPU 的寄存器，保存引起页面错误的线性地址。该寄存器内的地址会被直接读入局部变量 **address**。

7004： 函数 **find_vma**（第 33460 行）返回地址范围末尾在 **address** 之后的第一个 VMA。大家知道，这并不能够保证该地址位于 VMA 的范围内，而仅保证该地址比 VMA

的结束地址小，这样它就可能比 VMA 的初始地址还要小。因此这一点要被检查。假如通过判断，则 **address** 在 VMA 之内，控制就会向前跳转到标号 **good_area** 处（第 7023 行）；我们随后就会对这一点进行讨论。

7005： 如果 **find_vma** 返回空值 NULL，那么 **address** 就位于进程的所有 VMA 之后——换句话说就是超出了由进程引用的所有内存。

7009： **vma** 的开头和结尾都确实超过了 **address**；因此 **address** 在 VMA 低端地址以下。但是这并不会失去什么。如果 VMA 是向下扩展的类型——也可以说它是堆栈——这个堆栈可以简单的向下扩展来适应那个地址。

7011： CPU 提供的 **error_code** 的测试位 2。与监控（内核）模式相比，更多是在用户模式发生页面错误时设置此位。如果是在用户模式下，**do_page_fault** 函数会保证给定的地址在为进程建立的堆栈区域内，正如 ESP 寄存器所定义的那样。（例如，在代码溢出了被分配的堆栈矩阵时，就会产生这种情况。）如果是在监控（内核）模式下，就会跳过后一种判断，而简单的假定内核运行正常。

7019： 如果可能，使用 **expand_stack**（行 15480）将扩展到包含新的地址。如果成功，VMA 的 **vm_start** 成员将调整到包含 **address**。

7023： 到达 **good_area** 标记时，就意味着 VMA 包含 **address**，或者说要么它已经包括了 **address**，要么就是堆栈扩展后包括了该地址。

不管那一种方法，包括错误产生原因信息的 **error_code** 最低两位现在都可以被测试了。第 0 位是存在/保护位：0 表示该页不存在；1 表示该页存在，但试图的访问操作与页级保护位冲突。第 1 位是读/写位：0 表示读，1 表示写。

7025： **switch** 条件判断语句对于上述两个测试位所组合出的四种可能情况作出相应处理：

- case 2 或 3——检查包括的 VMA 是否可写。若可写，就是向一个写拷贝页面执行一次写操作；变量 **write** 被增加（设置到 1）以便接下来对 **hand_mm_fault** 的调用能够完成写拷贝过程。
- case 1——这意味着页面错误是由试图从一个存在但不可读的页面中读数据而导致的；这个尝试会被拒绝。
- case 0——表示页面错误是由试图从一个不存在的页面中读数据而导致的。如果涉及的 VMA 保护指出该区间既不可读也不可写，读页面只不过是浪费时间——如果再次尝试，将引起另一个页面错误，这样 **do_page_fault** 函数会以 case 1 的结果告终，即拒绝尝试。否则 **do_page_fault** 函数继续执行并从磁盘上读入页面。

7047： 请求 **handle_mm_fault** 函数（下面讨论）使该页面变为当前页面。如果失败，则发出一个 **SIGBUS** 错误。

7062： 大多数内核函数的清除代码都不太显著。**do_page_fault** 函数是一个例外；我们会比较详尽的研究它的清除代码。下列任何情况发生都会跳到 **bad_area** 标记处：

- 被引用的地址超过了为进程分配的（或保留的）所有内存。
- 被引用的地址位于所有 VMA 之外，而且可能由于比该地址小的 VMA 不是堆栈而无法扩展到这个地址。
- 违反了页面的读/写保护。

7066： 如果用户代码引起以上任何错误，那将发送致命的 **SIGSEGV** 信号——一个分段违例。（注意术语“ 分段 ”在这里是历史上的说法而不是字面所表达的意思——对 CPU 来说，从技术角度看它是分页违例，不一定是分段违例。）这个信号通常会像第 6 章中讨论的那样杀死一个进程。

7075： Intel Pentium CPU（以及它的一些兼容产品）具有一个所谓的 f00f 缺陷，它允许任

何进程用非法的 0xf00fc7C8 指令来冻结 CPU。Intel 所提议的弥补工作就是在这里实现的：中断描述表（见第 6 章）的一部分以前是被标识为只读的，因为这样会使非法指令执行时用产生页面错误代替冻结 CPU。在这里，**do_page_fault** 函数检查导致页面错误的地址是否位于 IDT 中由非法指令执行而产生的位置上。如果是这样的，处理器会试着执行“ Invalid Opcode ”服务中断——CPU 的缺陷会使得正确完成这一步失败，而代码却会通过直接调用 **do_invalid_op** 函数而产生正确的结果。否则，CPU 决不会对 IDT 进行写操作(即使没有标注为只读时也是如此),所以即使第 7080 行的检测失败，非法指令也是根本不会被执行的。

7086： 下列情况发生时，标记 **no_context** 会被执行：

- 在内核（不是用户）模式里到达 **bad_area**，而且 CPU 不执行触发 f00f 缺陷的非法指令。
- 在一个中断中或没有用户环境（用户任务没有处于正在执行状态）时发生的页面错误。
- **Handle_mm_fault** 函数错误并且系统处于内核模式中（我还从未遇到过这种情况）。

这里的任何一种情形都是内核错误（经常由驱动程序所导致），它不是因为任何用户代码而造成的页面错误。如果内核（或驱动程序）事先为这种可能准备了错误处理代码，那么这些错误处理代码一定位于本书讨论范围之外，并在错误发生时可以通过某种特殊技术跳转过去。

7097： 否则，内核试图访问一个坏页面，**do_page_fault** 函数将不知如何处理它。这可能也够被考虑到。内核启动代码检查是否 MMU 写保护工作正常；如果正常，那就不是一个真正的错误，**do_page_fault** 函数就可以简单的返回了。

7109： 内核访问了一个坏页面，并且 **do_page_fault** 函数无法修复这个错误。**do_page_fault** 函数会在第 7129 行显示出一些描述错误的信息，然后中止内核本身。这样整个系统就会被停止，很明显没有任何操作会被执行。不过，如果系统运行到了这一步，内核也别无选择了。

7134： 最后一个标记是 **do_sigbus**，只有当 **handle_mm_fault** 函数无法处理错误时才会执行到这里。这种情况相对简单；大体上是给违例的进程发送一个 **SIGBUS** 错误信号，如果这是在内核模式里发生的就再跳回到 **no_context** 标记处。

Handle_mm_fault

32725：调用者已经检测到了需要一个可用的页面。该页面正是包含 **address** 的页面，这个地址应归入 **vma** 中。**Handle_mm_fault** 函数本身相当简洁，但是它建立在其它几个处理冗长细节问题的函数和宏之上。我们介绍完此函数后将逐一研究那些底层函数。

32732：查找关联的页目录和页面中间目录入口项（如前所述，这两者在 x86 平台上实际是一样的）。

32735：从页面中间目录项得到或定位（如果可能的话）页表。

32737：调用 **handle_pte_fault** 函数把页面读入页表项（page table entry）；如果成功，就调用 **update_mmu _cache** 函数更新 MMU 的高速缓存。控制流程到此为止，一切顺利，**handle_mm_fault** 函数就可以返回一个非零值（1）表示成功了。如果此过程任何一步出错，控制就转向第 32744 行，函数返回 0 值表示失败。

Pgd_offset

11284：这个宏将 **address** 除以 2^{PGDIR_SHIFT}（第 11052 行#**define**d to 32），并对结果向下舍入，然后把最终结果（移位之前的高端 10 位）作为提供的 **struct mm_struct** 的 **pgd** 数组

的一个索引。因此，它的值就是页目录项，相应的页表 **address** 地址就位于该项中。这等价于代码

&((mm)->pgd[(address)>>PGDIR_SHIFT]);

而且可能会更高效。

Pmd_alloc

11454：因为 x86 平台上没有定义页面中间目录，这样就极其简单：它只需返回给定的 **pgd** 指针，并映射为一种不同类型。在其它平台上，该函数与 **pte_alloc** 类似，还要实现更多的工作。

Pte_alloc

11422：**Pte_alloc** 函数有两个参数：一个是指针，指向目标地址所位于的页面中间目录项，另一个是地址本身。如果我们暂时跳过一部分内容，那么对该函数经过变形的逻辑的理解就会更容易，所以让我们看一下随后的若干行代码。

11425：用一种几乎无法理解的方式把 **address** 转换成 PMD 内的一个偏移量。

这一行需要详细进行解释。首先，回忆一下 PMD 中的每项都是一个指针，在 x86 平台上它的长度是 4 个字节（这里的代码是与体系结构相关的，所以我们可以作出这样的假定）。用 C 语言来定义就是，

```
&pmd[middle_10_bits(address)]
```

（为清晰起见，我在这里引入了假定的 **pmd** 数组和 **middle_10_bits** 函数）该代码等价于

```
pmd+middle_10_bits(address)
```

这又与如下代码指向的地址相同

```
((char*)pmd)+middle_10_bits(address)*sizeof(pte_t*)
```

其技巧是在最后的公式中——或者更准确的说是+号后边的部分——最接近于第 11425 行所要计算的实际值。

为了使这一点更为明确，首先可知

```
4*(PTRS_PER_PTE-1)
```

就是 4092（第 11059 行 **PTRS_PER_PTE** 被定义为 1024）。用二进制表示，4092 只用占最低 12 位，甚至最后 2 位也用不上。它和只占最低 10 位的 1023 左移 2 位后的值相同。这样就有

```
(address>>(PAGE_SHIFT-2))
```

把 address 右移 10 位（第 10790 行 **PAGE_SHIFT** 被定义为 12）。这两个表达式结果再逐位进行与（AND）操作。最终的结果类似于：

((address>>PAGE_SHIFT)&(PTRS_PER_PTE-1))<<2

尽管这仍很复杂，不过它更简单明了：它把 **address** 右移 12 位（为了去掉页面偏移量部分），屏蔽掉除最低10位的其它位（去掉页目录索引部分，只保留最低10位的页面中间目录索引），接着把结果左移 2 位（相当于乘以 4，即指针长度的字节数 **sizeof(pte_t*)**）。更直接的方法可能会稍慢一些，但在内核里，我们终归是要尽力节省时间的。（虽然更直接的方法看来并非明显偏慢：同样版本内核进行两次移位、两次减法，以及按位与的操作，和进行两次移位、两次按位与的操作，就我的测试看来实际上是一样快。）

不管采用那一种方法，经过计算之后，把 **address** 和 PMD 的基地址相加（在第 11432 行和别的地方执行），就得到了指向与 **address** 初值关联的 PTE 的项指针。

11428：如果 PMD 项不指向任一个页表，函数向前跳到 **getnew** 处分配一个页表。

11435：通过调用 **get_pte_fast**（第 11357 行）尝试从 **pte_quicklist** 中申请一个页表。这个页表是页表的一个高速缓存，其思想是分配页表（它们本身就是独立的页面）慢，而从一系列近期释放的页表中指定一个却会稍快一些。所以，代码经常用 **free_pte_fast**（第 11369 行）来释放页表，这会把它们放在 **pte_quicklist** 里而不是确实把它们消除掉。

11439：**pte_quicklist** 能够提供一个页表页面。页表可以被送入页面中间目录，并且函数返回页表中这个页面的偏移值。

11438：**pte_quicklist** 缓存里没有剩下页面，因此 **pte_alloc** 需要调用 **get_pte_slow** 函数（第 7216 行）来分配一个缓慢页面。该函数用**__get_free_page** 来分配页面，执行过程和一个页面被找到时相似。

11430：如果 PMD 项不是 0，但是是无效的，**pte_alloc** 显示一个警告（通过调用第 7187 行的 **bad_pte**）并放弃尝试。

11432：所期待的正常情况：**pte_alloc** 函数返回一个指向包括 **address** 地址的 PTE 的指针。

Handle_pte_fault

32690：**Handle_pte_fault** 函数试图取回或者创建一个缺少的 PTE。

32702：给定的项与物理内存中的任何一个页面都无关联（32700 行），而且确实没有被设置（32701 行）。这样，**do_no_page**（32633 行）将被调用以创建一个新的页面映射。

32704：页面在内存中不存在，但是它有一个映射，所以它一定在交换空间里。函数 **do_swap_page**（32569 行）将被调用来把该页面读回内存。

32708：页面在内存里，所以情况可能是内核正在处理一个页面保护冲突。**Handle_pte_fault** 首先要用 **pte_mkyoung**（11252 行）来把该页面标识为已被访问。

32713：如果是一个写访问操作而页面又不是可写的，**Handle_pte_fault** 就调用 **do_wp_page** 函数（32401 行）。这个函数完成真正的写拷贝功能，因此我们要简单介绍一下。

32715：这是一次对可写页面的写访问。**Handle_pte_faul**t 设置该页面的“ dirty ”位，表示在它被丢弃之前必须被复制到交换空间。

32720：所需的页面现在可被调用者使用，所以 **Handle_pte_fault** 函数返回非零值（确定为 1）以示成功。

Update_mmu_cache

11506 在 x86 平台上，**update_mmu_cache** 函数是一个空操作。它是一种所谓的‘ 挂钩（hook）”函数——这种函数要在内核的平台无关部分中适当地点处保证被调用，以便不同的移植版本都能够在必要的情况下对它进行定义。

Do_wp_page

32401：如前所述，真正的写拷贝操作是在这里实现的，所以我们有必要介绍一下。**tsk** 试图写入 **address**，这个地址在给定的 **vma** 里并由所提供的 **page_table** 来控制。

32410：调用**__get_free_page**（15364 行，简单的转向第 34696 行调用**__get_free_pages** 函数）为进程提供一个新页面，此页面是写保护页面的一个新拷贝。注意这里可以允许一个任务转换。有趣的是，这里的代码不检查**__get_free_page** 分配新页面时是否成功——它实际上可能不需要新的页面，因此到必要时才会去进行检查。

32422：增加“次要（minor）”页面错误，这些错误无需访问磁盘就可被满足。

32438：只有两个页面用户存在，其中一个是交换高速缓存（swap cache），它是已被交换出但还未被回收的页面的临时缓冲池。该页面被移出交换高速缓存后（利用 37686 行的函数 **delete_from_swap_cache**），现在它就只有一个用户了。

32445：要么从交换高速缓存里回收该页面，要么它只有一个开始用户。这个页面会被标识为可写和“脏”dirty（因为它已被写过）。

32448：如果已分配了一个新页面，它就没有用了：由于该页面只有一个用户，所以没有必要进行复制。**do_wp_page** 函数释放这个新页面，并返回非零值表示成功。

32454：页面拥有不止一个用户，不能简单的从交换高速缓存里被收回。因此 **do_wp_page** 函数将需要复制一个新页面。如果先前的页面分配失败，现在就是该结果产生作用的时候了，**do_wp_page** 函数将不得不返回错误。

32459：利用 **copy_cow_page**（31814 行）复制页面内容。这通常是调用 **copy_page** 宏（32814 行），它是一个 **memcpy**。

32460：利用 **flush_page_to_ram**（10900 行）使 RAM 新旧页面拷贝同步。像 **update_mmu_cache** 函数一样，在 x86 平台上这是一个空操作。

32463：像以前一样，使得页面可写和“脏”，同时保留从封装的 VMA 而来的其它页面保护（比如可执行）。

32466：对函数 **free_page**（在 15386 行，它只是调用 34633 行的 **free_pages** 函数）的调用而不会真正释放旧的页面，因为该页面拥有多个用户——它只会减少旧页面被引用的次数。由于满足了调用者的请求，**do_wp_page** 就返回非零值表示成功。

页面调出

现在读者已经对交换页面调入有所了解，接下来看一看另一方面，交换页面的调出。

Try_to_swap_out

38863：**try_to_swap_out** 函数是最低一级交换调出函数，它由内核任务 **kswapd**（见 39272 行 **kswapd** 函数）周期性地调用（通过一系列其它函数调用）。这个函数用来写一个页面，该页面是由位于给定任务特定 VMA 中的一个单独页表项来控制的。

38873：如果内存中缺少该页面，它就不能从内存写回到磁盘，这样 **try_to_swap_out** 函数就返回失败。如果给定的地址明显是不合法的（**max_mapnr** 是当前系统中物理内存的页面数目；参见 7546 行），它也会丢弃尝试操作。

38880：如果页面被保留、锁住，或者被一个外设用于直接内存访问时，它就不能被调出。

38885：如果页面最近被访问过，把它调出可能是不明智的，因为引用的局部性可能会使该页面不久将再被引用。把该页面标识成“旧的”，这样将来的再一次尝试就可能把它调出内存——这可能很快就会发生，如果内核不顾一切要这么做的话。但事实是，

页面还没有被调出。

这一行之后的代码注释本身就含有大量信息，所以我们将跳过几段代码而不失完整性。

38965：减少任务的驻留段长度（注意 **vma->vm_mm** 是指向含有 **vma** 的 **struct mm_struct** 的指针）。驻留段长度是物理内存中的任务所占页面数目，而且很明显，这些页面中的一个现在已经不存在了。

38968：因为页面无效，所以 **try_to_swap_out** 函数必须通知所有 TLB 以无效化它们对该页面的引用。TLB 不应该再把地址解析到一个已经不存在了的页面。**try_to_swap_out** 函数接着把这个页面放入交换缓存。

38977：最后，**try_to_swap_out** 函数通过使用 **rw_swap_cache**（35186 行）把旧的页面写回磁盘，写操作是异步的，以便等待磁盘处理时系统也可以作其它工作。

38979：用**__free_page**（34621 行）来释放页面，并返回非零值表示成功。

交换设备

Linux 拥有一个按优先级排序的合法交换设备列表（以及文件，不过为了简单起见，这一部分通常用“设备”来代替这两者）。当需要分配一个交换页面时，Linux 会在仍然拥有空间的优先级最高的交换设备上来分配它。

Linux 也会在所有优先级相同的未满交换设备之间轮转使用，采用的是循环方式，通过这种在多个磁盘上分布分页请求的方法可以提高交换的性能。在等待第一个请求被满足时，另一个请求就可以分派到下一个磁盘上。最快的设置是把交换分区分布在几个相似的磁盘上，并给它们同样的优先级设置；而较慢的磁盘则有稍低一些的优先级。

不过循环也可能造成交换速度的降低。如果同一磁盘上的多个交换设备有同样的优先级，那么磁盘的读/写头将不得不在磁盘上来回的反复访问它们；在这种情况下，臭名卓著的 1000 倍的速度差异就不容忽视了。幸运的是，系统管理员会合理安排优先级以避免这种情况。Linux 继承了 Unix 的传统特性，既能让你陷入绝境，也能使你达成非常良好的目标。最简单的方案是给每个交换设备分配不同的优先级；这会有助于避免最坏的情况，但可能也不会最好。尽管如此，由于该方法简单且不会最坏，如果你不指定优先级设置，它将是缺省设置。

交换设备用 **struct swap_info_struct**（17554 行）结构体类型来表示。在 37834 行定义了这些结构体的一个数组 **swap_info**。好几个文件里的函数都操作和使用 **swap_info** 数组来进行交换管理；很快我们就会对它们进行分析。先来分析一下 **struct swap_info_struct** 的成员，这会使我们能够更清楚的了解这些函数。

- **swap_device**——发生交换的设备号；如果 **struct** 代表一个文件而不是分区，值是 0。
- **swap_file**——**struct** 代表的交换文件或分区
- **swap_map**——对交换空间里每个交换页面的用户数进行计数的数组；为 0 则表示页面空闲。
- **swap_lockmap**——用来跟踪基于磁盘的页面当前是否正被读出或写入磁盘，数组里的每一位对应一个页面。在 I/O 过程中页面将被锁定以防止内核同时对同一页面执行两次 I/O 操作，或者其它类似的愚蠢操作——需要记住的是，一旦有可能，其它进程就会与 I/O 操作相重叠，所以发生这种情况并非难事。
- **lowest_bit** 和 **highest_bit**——跟踪交换设备里第一个和最后一个可用的页面的位置。这可以有助于加快寻找空闲页面的循环。设备的第一个页面是一个不允许用于交换的头部，因此 **lowest_bit** 不会是 0。

- **cluster_next** 和 **cluster_nr**——用来对磁盘上的交换页面进行分组以获得更高的效率。
- **prio**——交换设备的优先级。
- **pages**——设备上可用的页面数目。
- **max**——内核在此设备中所允许的最大页面数目。
- **next**——把 **swap_info** 数组中的所有 **struct** 形成一个单独的链接列表（并保持优先级顺序）。这样，数组就被逻辑排序，而不是物理排序了。**next** 的值就是列表中逻辑指向下一个元素的索引，如果到达列表末尾它就是-1。

swap_list 在 37832 行定义，包括列表头（即 **head** 成员——参见 17627 行 **struct swap_list_t** 的定义）的索引；如果列表为空则此索引为-1。它还包括名字很令人迷惑的 **next** 成员，这个成员能够跟踪我们将要在其上尝试页面分配的下一个交换设备。因此 **next** 是一个迭代指针。如果列表为空或者当前没有交换，它的值就是-1。

Get_swap_page

37879：**get_swap_page** 函数从最高优先级的拥有空间的可用交换设备里获得一个页面；如果找到一个，它就返回一个非零代码描述该项，如果系统没有交换就返回 0。

37885：从上一次停止的地方继续进行迭代。如果列表是空的或没有剩余交换设备，函数即刻返回。

37891：否则的话，有理由确信存在交换空间，**get_swap_page** 函数恰恰需要找到它。这个循环过程一直迭代，直到函数找到一项（很可能的情形）或者扫描了每一个交换设备但没有一个还有剩余空间（不太可能的情形）为止。

37894：利用 **scan_swap_map**（37838 行）扫描当前交换设备的 **swap_map** 以寻找一个空闲单元，如果找到了一项，**lowest_bit** 和 **highest_bit** 成员也会被更新。要返回的 **offset** 是 0 或者是该项。

37897：当前的交换设备能够分配一个页面。**get_swap_page** 函数现在把 **swap_map** 的迭代游标向前推进以便请求能被正确的分布在交换设备上。

如果已经到达交换设备列表的末尾或是下一个交换设备的优先级低于当前设备，迭代过程就会从列表的头部重新开始。这产生两个重要作用：

- 如果较高优先级设备的交换空间又变得可用，**get_swap_page** 就会在下一次迭代时从那个设备开始分配交换。如果孤立的观察这些代码，读者会认为当高优先级设备可用的时候，这个函数仍可以从低优先级设备分配少数页面。然而事实并非如此，在我们对交换页面是如何被释放进行介绍的时候读者就会看到这一点。
- 如果优先级高的交换设备不可用，那么在下一次内核分配一个交换项时，**get_swap_page** 函数将沿列表进行迭代直到它找到当前优先级的第一个设备为止，并试着从那个设备分配交换。因此，在内核转向优先级较低的设备之前，内核会继续考虑优先级较高的设备直至它们全部耗尽。这就是先前讨论过的循环执行过程。

37910：当前设备没有可用的交换空间，或者当前设备是不可写的（这与我们所说的是同样的）。跳到下一个设备，这样如果它已经到达末尾但还未曾循环一整圈时，它就会再从头开始循环。

37916：如果 **get_swap_page** 函数到达列表的末尾而且已经循环了一遍，它就已经考虑了所有交换设备但是没有一个拥有空余的空间。因此，结论是再也没有可用的交换空间了，函数返回 0。

Swap_free

37923：**swap_free** 函数是与 **get_swap_page** 函数相对的，它释放一个单独的交换项。

37939：通过许多简单而又周密的测试后，**swap_free** 函数检查是否正在释放交换页面的设备具有比随后将被考虑的设备更高的优先级。如果是，它就把此作为一个线索以将 **swap_list** 的迭代器重新设置在列表头部。这样对 **get_swap_page** 函数的下一次调用就会从列表头部开始并能够检测到新近被释放的高优先级空间。

37944：假如最新被释放的页面处于 **lowest_bit** 和 **highest_bit** 成员所定义的范围之外，就要相应的对它们进行调整。你可以看到如果 **swap_free** 函数在一个以前已经耗尽了的设备中释放页面，这通常会引起对 **lowest_bit** 或者 **highest_bit** 的调整，但并非都要调整。这会使该区间比所需要的大，交换页面分配也会因此比所需要的要慢。不过这种情况很少发生。无论如何，交换范围都将调整自己以使更多的交换页面能够被分配和释放。

37950：对 **swap_map** 每一元素的使用计数只维护到一个最大值，即 **SWAP_MAP_MAX**（17551 行定义为 32767）。达到这个最大值之后，内核将无法知道真正的计数值有多大；由此它也无法安全的减少该值。否则的话，**swap_free** 函数将减少使用计数并增加空闲页面的总数。

Sys_swapoff

38161：**sys_swapoff** 函数在可能情况下从交换设备列表中移去被指明的交换设备。

38178：搜索 **swap_info_structs** 的列表以查找匹配的项，设置 **p** 指向这个数据项、**type** 指向该数据项的索引，以及 **prev** 指向前一项的索引。如果第一个元素被删除，**prev** 将是 -1。

38195：如果 **sys_swapoff** 函数搜索了整个列表但没有找到匹配项，那显然是给定了一个无效的名字。函数返回错误。

38198：如果 **prev** 是负值，**sys_swapoff** 函数将删除列表的第一个元素；它相应的适当更新 **swap_list.head**。可以证明，这等价于

```
swap_list.head=
  swap_info[swap_list.head].next
```

不过速度更快，因为其中所牵扯的间接转换更少。

38203：如果正被移去的设备是内核进行交换尝试的下一个设备，迭代游标会被重新设置在列表头部。这样下一次分配可能要稍慢一点儿，不过并不显著；无论如何，实际中这样的情况是相当少见的。

38209：由于设备仍在使用中而不能被释放时，它会被恢复到列表的适当地方。如果这是数个拥有同样优先级交换设备之中的一个，它可能不会回到同以前一样的相对位置了——它将是具有同样优先级的设备的第一个而不是最后一个——不过列表仍然是按照优先级进行排序的。

从交换设备列表上删除一个仍有可能被我们又放回原处的设备，这看起来就象是在做无用功——为什么不等到可以确信它可以被删除时再删除它呢？

答案在于经由一系列利用 **swap_list** 的函数调用后，在前面代码行对 **try_to_unuse**（38105 行）的调用能够结束。 如果正被删除的交换设备那时仍在 **swap_list** 里，那么终止这一切的代码将会给系统造成极大的混乱。

38223：若在一个分区上进行交换，**sys_swapoff** 函数将解除对它的引用。

38244：**sys_swapoff** 函数以使所有数据域无效和释放已分配的内存而告终。特别的，这行代码清除 **SWP_USED** 位，这样内核就会在它再次利用该交换设备时知道它已经是不可用的了。接下来，**sys_swapoff** 函数清除 **err** 指示符并返回成功。

Sys_swapon

38300：**sys_swapon** 函数是 **sys_swapoff** 的对应函数，它向系统列表里增加交换设备或交换文件。

38321：找寻未用的一个项。这里有一些微妙之处。读者可能会从 **nr_swapfiles** 的名字推断出它就是交换文件（或者设备）的数目，但是实际它不是。它是曾被使用过的 **swap_info** 的最大索引值，而且从不会被降低。（它记录着这个数组被使用的最高峰值。）因此，把 **swap_info** 中的这许多项循环一遍的结果是，要么发现未用的一个项，要么在最后一次循环增量后让 **p** 指向第 **nr_swapfiles** 项之后。在上述的后一种情况下，若 **nr_swapfiles** 比 **MAX_ SWAPFILES** 小，那么所有用过的项恰好会排在数组的左边，而循环就使得 **p** 指向它们右边的一个空位。这样，**nr_swapfiles** 就会被更新。

有趣的是，即使 **nr_swapfiles** 不是最高峰值而是活动交换设备的计数值，循环也能正确执行。不过若我们改变了 **nr_swapfiles** 的原意，文件里的其它代码就会有问题了。

38328：在 **swap_info** 里找到了一个未用的项；**sys_swapon** 函数开始对其进行填充。这里所提供的一些值将会发生变化。

38338：若 **SWAP_FLAGS_PREFER** 被置位，**swap_flags** 的低端 15 位就被编码为所需的优先级。（这里使用的常量和接着的几行代码在 17510 行进行定义。）否则，就不指定优先级。如前所述，在此情况下的缺省作法是给每一个设备分派一个逐渐降低的优先级，其目的是在无须人工干预时也能得到令人满意的交换性能。

38344：保证内核允许交换的文件或设备可以被打开。

38352：检查提供给 **sys_swapon** 函数的是一个文件还是一个分区。若 **S_ISBLK** 返回为真，它就是一个块设备，即磁盘分区。在此情形下，**sys_swapon** 函数继续确保能够打开该块设备而且内核此时没有同它进行交换。

38375：同样的，若给定的不是分区，**sys_swapon** 函数必须确保它是一个普通文件。若是文件，函数还要确保内核此时没有同该文件进行交换。

38384：如果两项测试均告失败，**sys_swapon** 函数就不会再被请求在磁盘分区或文件上进行交换；它已经拒绝了该尝试。

38396：从交换设备里把头页面读入 **swap_header**；这是一个在 17516 行定义的 **union swap_ header** 联合体类型。

38400：检查一串特征字节序列，该序列记述了交换头部的版本信息，它是由 mkswap 程序给出的。

38412：交换类型 1。此时，该头页面被当作一个大的位映射图，每一位代表设备中剩下的一个可用页面。同其它页面一样，头页面也是 4K 字节，即 32K 比特。由于每一位表示一个页面，设备就可以拥有 32768 个页面，也就是每个设备总计 128MB。（实际上要稍小一些，因为头页面的最后 10 位用于签名，这样我们就不能假定它们对应的 80 个页面也是可用的；另外头页面本身也不能用于交换。）如果实际设备比这个值小，那么头页面中的一些位就不起作用。在 38417 行，函数进入循环来检查哪些页面是可用的，并对它正在创建的 **swap_info_struct** 的 **lowest_bit**、**highest_bit** 以及 **max** 成员进行设置。

注意这个头页面位映射图不会永远被保持——当 **sys_swapon** 函数结束时它就会被

释放。内核利用交换映射表来跟踪正在使用的页面；该头页面位映射图仅被用来设置 **lowest_bit** 和其它 **swap_info_struct** 结构体的成员。

38427：分配交换映射表并把所有使用计数值设置为 0。

38440：交换类型为 2 的交换并没有减轻交换区容量的限制，不过它以一种更自然和有效的方式贮存头部的信息。在此情形之下，**swap_header** 的 **info** 成员就包含了 **sys_swapon** 函数所需的信息。

38451：新的交换头部版本不需要 **sys_swapon** 函数把头页面当作一个位映射图来计算 **lowest_bit**、**highest_bit**，和 **max** 的值——**lowest_bit** 总是 1，另外两个值可以从明确储存在头部的信息在定长时间内计算出来。这要比执行 32768 次位测试的循环快的多也简单的多，而且后者的定义语句甚至比前者要多出两倍以上！尽管如此，这部分以及余下的工作从概念上讲还是与以前十分相似的；**sys_swapon** 函数只不过是从交换头部直接获取了它所需要的大部分信息，而无须在计算它们而已。

读者现在可以看出我刚才撒的一个小谎；版本类型为 2 的交换实际上真正克服了交换区容量的限制。在这个版本中，文件末尾的 80 个页面不会由于交换头部签名而不可利用，因此单独一个设备有可以有 320K 用于交换。不过上限仍然是大概 128MB。

38491：**sys_swapon** 函数忽略读取头部。它把设备交换映射表的第一个元素设置为 **SWAP_MAP_BAD**（17552 行）以避免内核在头页面上进行交换。

38492：分配加锁映射表并清零。

38499：更新可用的交换页面总数，并对此结果显示一个消息。（在 38502 行，从移位计数器里减去 10 以便输出结果是千字节表示，2^{10} 就是 1K。）

38505：在交换设备的逻辑列表中插入新元素，仍遵循优先级排序的顺序。这里的代码从功能上是与 **sys_swapoff** 函数中相应的代码一样的，所以没理由把它们分离开来。一个能代替两者的内嵌函数就能简单的解决问题。

38519：进行清理工作，然后结束。

内存映射 mmap

mmap 是一个重要的系统调用，它允许为不同目的而设置专用的独享内存区域。该内存可能是一个文件或其它特别对象的代理，在这种情形中，内核将保持内存区域和潜在对象的一致，或者该内存可能是为一个应用程序所需要的简单的无格式内存。（应用程序通常不使用 **mmap** 来分配无格式内存区，因为此时 **malloc** 更符合其目的。）

mmap 最普遍的使用方法之一是为内核本身通过内存映射（memory-map）形成一个可执行文件（参见 8323 行的一个例子）。这是关于二进制处理程序如何同分页机制协同工作以提供所需要分页的可执行体，这正如本章早些时候所暗示的。可执行体通过 **mmap** 被映射为进程内存空间中的适当区域，然后 **do_page_fault** 函数调入执行体所需的剩余页面。

被 **mmap** 分配的内存可能被标识为可执行，其中充满了指令代码，随后系统跳入其中开始执行；这正是 Java Just-In-Time（JIT）编译器的工作方式。更简单的说，可执行文件能够被直接映射成一个正在运行的进程的内存空间；这项技术用于动态连接库的执行中。

执行 **mmap** 功能的内核函数是 **do_ mmap**。

do_ mmap

33240：**do_ mmap** 函数具有几个参数；它们共同定义应在内存中映射的文件或设备，并决定将被创建的内存区域的首选地址及其它特性。

33252：**TASK_SIZE** 和在 10867 行定义的 **PAGE_OFFSET** 值相同——即是 0xc0000000 或

3GB。这是用户进程所能拥有的最大内存，在此基础上代码才有意义：显然，如果要求 **do_ mmap** 函数分配大于 3GB 的内存，或者在 **addr** 之后的 3GB 内存空间没有足够的空间，分配请求就必须被放弃。

33275：如果 **file** 为 **NULL**，**do_ mmap** 函数将被请求去执行匿名映射（anonymous mapping）操作，这是一种并不与任何一个文件或其它特别对象连接的映射过程。否则，映射将被关联到一个文件，接着 **do_ mmap** 函数要继续检查为内存区域设置的标志位是否与用户在文件上允许执行的操作相兼容。举例来说，在 33278 行，函数要确保是否内存区可写，因为文件已经被打开并执行写操作了。省略这项判断将可能使文件打开时所作的检查发生混乱。

33307：允许调用程序强调 **do_ mmap** 函数应该或者在要求的地址上提供映射操作，或者根本没有什么也不做。如果提供地址，**do_ mmap** 函数只需保证提供的地址从一个页面的边界开始。否则，它将获得在 **addr** 处或之后的第一个可用地址（通过调用开始于 33432 行的 **get_unmapped_area** 函数），然后就使用这个地址。

33323：创建一个 VMA 并对其进行填写。

33333：如果内存映射着一个可读文件，则内存区域就被设为可读、可写和可执行。（**do_ mmap** 函数可以很快的取消写许可——这只是假定）另外，如果要求共享该内存区域，那么现在就可以满足该请求。

33347：若文件不可写，则内存区域也必须不可写。

33351：在此情形中，没有这样的文件，使得 **do_ mmap** 函数必须与该文件的打开模式和许可权限相一致——就允许函数自由运行。因此，函数把内存区域设为可读、可写和可执行的。

33361：在地址范围建立时，利用 **do_munmap**（很快就会被讨论到）来清除任何旧的内存映射。因为新的 VMA 还没有插入进程列表之中（只有 **do_ mmap** 函数当前知道它的存在），所以新 VMA 不会被此次调用影响。

33406：不会再有错误发生。**do_ mmap** 函数把新 VMA 插入进程的 VMA 列表（或是它的 AVL 树），合并所有新近相连的段片（接下来会对 **merge_segments** 函数进行讨论），更新一些统计数字，并返回新映射的地址。

Merge_segments

33892：**merge_segments** 函数是一个有趣的函数，它把相邻的 VMA 合并成单独的一个大范围的 VMA。换句话说，如果一个 VMA 所覆盖（有意这样设计）范围是从 0x100 到 0x200，而另一个 VMA 的覆盖范围是从 0x200 到 0x300，并且两者保护信息相同，那么 **merge_segments** 函数就会用一个覆盖范围从 0x100 到 0x300 的单独 VMA 来代替它们。（注意函数名中的“ segments ”并不暗示此时我们采用 CPU 分段机制。）**merge_segments** 函数的参数是结构体 **struct mm_struct**，它包含了我们该兴趣的 VMA 以及可能进行合并的开始地址和终止地址。

33897：**find_vma_prev** 函数将其 **vm_end** 定位在给定的 **start_addr** 之后的第一个 VMA 上——由此，第一个 VMA 可能会包括 **start_addr**。回忆一下 **find_vma_prev** 函数，它也返回一个指向前一个 VMA 的指针 **prevl**（如果第一个 VMA 满足条件则该返回值是 **NULL**）。

33911：进入处理所有覆盖给定区间的 VMA 的循环。在该循环过程中，**merge_segments** 函数将尝试把每一个段片都与其前一个段片进行合并，而前一个段片的值可以通过 **prev** 获得。

33921：绝大部分条件判断都是相对直截了当的，不过最后一个测试就不这么简捷了。它确保 **prev** 和 **mpnt** 是连续的——也就是在 **prev** 的结尾和 **mpnt** 的开头之间没有未被映

射的内存。即使检测结果是一个的 **vm_end** 和另一个的 **vm_start** 相等，这两块区域在这一点上也未必一定相互覆盖——回忆一下，**vm_end** 是要比 VMA 拥有的最后地址还要大一位的。从 33926 行到 33932 行的代码为被映射文件和共享内存坚持了同样的特性：一块区域的末尾要等于下一块的开头。

33937：**merge_segments** 函数找到了可以合并的 VMA。它把 **mpnt** 从 VMA 列表（还可能是 AVL 树）里移出，再将它存入 **prev**。要注意的是即使 VMA 的数目降到了 **MIN_MAP_COUNT** 以下，它都不会拆除 AVL 树。

33948：如果将要消失的 VMA 是一个被内存映射的文件的一部分，**merge_segments** 函数就删除它对该文件的引用。

do_munmap

33689：**do_munmap** 函数明显是 **do_mmap** 函数的反作用函数；它从一个进程的内存空间里废除虚拟内存映射。

33695：如果 **do_munmap** 函数被要求取消映射的地址不是页面对准的，或者地址区域位于进程的内存空间之外，那么很明显它就是无效的，因此请求就会被拒绝。

33699：如果连一个页面也没有被释放，就拒绝尝试。

33707：查找包括给定地址的 VMA。令人奇怪的是，**do_munmap** 函数返回的是 0——而不是错误——如果地址不在任何一个 VMA 之内的话。从某种意义上讲，这是正确无误的；**do_munmap** 函数被要求用来确保一个进程不再对特定内存区域进行映射，如果一开始就没有这种映射的话，那就很容易办到。不过这仍颇为奇怪；在调用者看来这是一个错误而且 **do_munmap** 函数也应该报告这个错误。然而，某些调用程序却希望它如 33361 行的示例那样执行工作。

33717：如果给出的内存区域整个在单独的一个 VMA 中，但又不在该 VMA 的一端，那么移去这段区域就会在封闭的 VMA 里生成一个空洞。内核是不会容忍这个空洞的，因为按照定义，VMA 应该是连续的一段内存。因此在这种情况之下，**do_munmap** 函数就需要创建另一个 VMA，使得空洞的两边各有一个 VMA。尽管如此，如果内核已经为该进程创建了所允许的所有 VMA，那么函数就不能这样做了，所以此时 **do_munmap** 函数不能满足请求。

33730：标识所有与该区域相交迭或在区域里的 VMA 为空闲状态，同时把每一个都放在本地堆栈 **free** 里。顺着这个过程，**do_munmap** 函数会把 VMA 从它们的 AVL 树中删除，如果有的话。

33743：**do_munmap** 函数已经建造了要释放的 VMA 堆栈，现在释放它们。

33748：计算要释放的准确范围，要牢牢记住的是这个范围可能不能以完整的 VMA 来度量。假如为 **min** 和 **max** 的定义适当，这三行可以被写成如下代码：

```
st  = max ( mpnt -> vm_start, addr );
end = min ( mpnt -> vm_end,   addr + len );
```

由此，**st** 是 **do_munmap** 函数实际开始释放区域的开头，**end** 是该区域的结尾。

33765：如果 VMA 是共享映射的一部分，**do_munmap** 函数通过调用 **remove_shared_vm_struct**（33140 行）来断开 **mpnt** 与共享 VMAs 列表的链接。

33759：更新 MMU 数据结构，它对应于这个 VMA 里当前被释放掉的子区域。

33765：调用 **unmap_fixup** 函数来修补映射，我们接下来就会对这个函数进行研究。

33773：**do_munmap** 函数已经释放了该范围内由 VMA 代表的所有映射；最后重要的一步就是要为同一区域释放页表，这是通过调用 **free_pgtables**（33645 行）实现的。

Unmap_fixup

33578：**unmap_fixup** 函数修复给定 VMA 的映射，这可以或者通过对一端进行调整，或者通过在中间制造一个空洞，再或者通过把 VMA 完全删除的方法来完成。

33590：第一种情况比较简单：去掉整个区间的映射。**do_munmap** 函数仅仅需要关闭底下的文件或其它对象即可，如果它们有的话。读者可以看到，这无须把 VMA 本身从 **current->mm** 里移出；它已经被调用者删除了。因为 VMA 的全部范围将被解除映射，没有什么要向后推移的，所以 **unmap_fixup** 函数就此返回。

33599：接下来的两种情况处理把 VMA 从开头到末尾一块区间移去的问题。这也是比较简单的；它们的主要工作是要调整 VMA 的 **vm_start** 或 **vm_end** 成员。

33608：这是四种情况中最有意思的一种——从一个 VMA 的中间移去一块区域，从而会产生一个空洞。函数先开始要复制一份额外生成的 VMA 的本地拷贝，然后通过将 ***extra** 设置为 **NULL** 来通知调用程序该附加 VMA 已被使用。

33611：图 8-4 表示了分裂 VMA 的过程。大部分信息被直接从旧 VMA 复制到了新 VMA，在此之后，**unmap_fixup** 函数对两个 VMA 的范围都作了调整以解决空洞问题。原先的 VMA，**area**，被缩小到了表示低于空洞的子区域，而 **mpnt** 则表示高于空洞的子区域。

33626：把全部新子区域插入 **current->mm**。

33629：在除了第一种的其它情况里，**unmap_fixup** 函数保持了旧的 VMA。它缩小了，但还未消失，因此它将被插回到 VMA 的 **current->mm** 集合中。

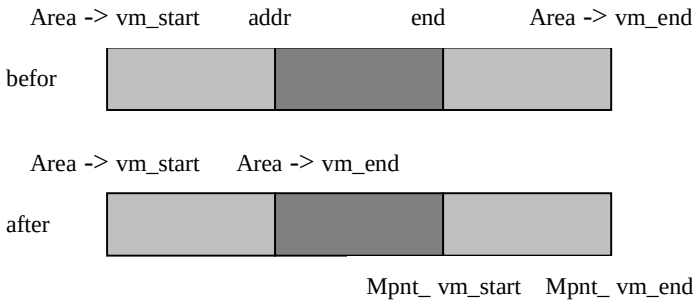

图 8.4 分裂 VMA

用户空间和内核空间

动态内存

用户任务和内核本身都经常需要快速分配内存。C 程序一般使用著名的 **malloc** 和 **free** 函数来完成这项工作；内核也有它自己类似的机制。当然，内核必须至少提供支持 C 语言的 **malloc** 和 **free** 函数的低级操作。

在 Linux 平台上，就像其它的 Unix 变种一样，一个进程的数据区分为两个便于使用的部分，即栈（stack）和堆（heap）。为了避免这两个部分冲突，栈从（准确的是接近）可用地址空间的顶端开始并向下扩展，而堆从紧靠代码段上方开始并向上扩展。虽然可以使用 **mmap** 在堆和栈之间分配内存，但是这部分空间通常是没有使用的内存的空白地带。

即使不去研究有关的内核代码（不过我们还是要继续这项工作），读者也能对这些地址区间所处位置有相当好的了解。下面的短程序显示了几个挑选出来的对象的地址，它们分处于三种不同内存区域之内。由于种种理由，我们不能保证它可以被移植到所有平台上，不过它可以在 Linux 的任何版本下工作，而且也应该可以被移植到你所尝试的大部分其它平台上。

P515-1 代码

在我的系统上，我得到了如下的数字。你的结果可能会稍有不同，除了所使用的编译器标志外，它还取决于你的内核及 gcc 的版本。即使不完全相同，它们也应该与下面结果相当接近。

P515-2 代码

从这里你不难看出，如果使用大概的数字的话，栈从接近 0xC0000000 处开始并向下生长，代码从 0x8000000 处开始，而堆则如前所述从临近代码上部的地方开始并向上扩展。

Brk

系统调用 **brk** 是一个在 C 库函数 **malloc** 和 **free** 底层的原语操作。进程的 **brk** 值是一个位于进程堆空间和它的堆、栈中间未映射区域之间的转折点。从另一个角度看，它就是进程的最高有效堆地址。

堆位于代码段顶端和 **brk** 之间。如果 **brk** 底下的可用自由空间不够满足请求，C 库函数 **malloc** 就抬高 **brk**；如果被释放的空间位于 **brk** 之下，就降低 **brk**。顺便说一句，Linux 是我所知道的唯一的在使用 **free** 函数时真正的减少进程内存空间的 Unix 变体；其它我所经历过的所有 Unix 商业版本实际上都是保留该进程的空间的——显然这是“ 以防万一 ”的作法。（其它 Unix 的自由版本可能同 Linux 一样，不过我没有使用过。）另外，对于大量的分配工作，GNU 的 C 库使用 **mmap** 和 **munmap** 系统调用来执行 **malloc** 和 **free**。

代码、数据，以及栈的关系如图 8-5 所示。

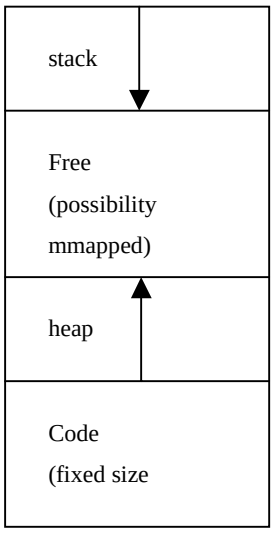

图 8.5 代码、数据和栈

Sys_brk

33155：实现 **brk** 的函数是 **sys_brk**。它可以修改进程的 **brk** 值，还可以返回一个新值。如果无法修改 **brk** 的值，返回的 **brk** 值就等于其原值。

33177：如果 **brk** 的新值位于代码区域之中，它就明显偏低而必须被抛弃。

33179：通过使用宏 **PAGE_ALIGN**（10842 行）把 **brk** 参数向上取整到地址更高的下一个页面。

33180：按页对准进程原有的 **brk** 值。这看起来有些多余，因为如果进程的 **brk** 只是在这里被设置，它就一定是按页排列的。但是在初始化一个进程的时候，进程的 **brk** 可以被设置在别的地方，代码并不会把它按页对准排列。不管进程的 **brk** 在哪里被设置，把它按页对准都可能会快一些；允许内核在这里跳过一次页对准操作，而且由于此处要比别的地方更频繁的对进程的 **brk** 进行设置，它应该不会降低执行效率而且还会少许提高。

33185：**brk** 被降低了，不过还没有进入代码区域，因此尝试被允许。

33192：如果堆的大小有限制，它就要被考虑。图 8-5 清楚的表明，**brk - mm->end_code** 是堆的大小。

33197：如果 **brk** 扩展到了已被一个 VMA 所内存映射的（**mmap**ped）区域，它就是不可利用的，因此这个新 **brk** 值要被舍弃。

33201：最后一项必要的检查是察看是否存在足够的自由页面用于空间分配。

33205：使用 **do_mmap** 函数（33240 行）为新区域分配空间。然后，**sys_brk** 函数更新进程的 **brk** 的位置并返回新值。

Vmalloc 和 vfree

内核编程中一个有趣的方面是并没有像应用程序编程人员通常所想当然的那样能够得到很多服务。就拿 **malloc** 和 **free** 作为例子，它们就是建立在一个内核原语 **brk** 之上的 C 库函数。

假使内核被修订以使其可以和标准 C 库连接，并使用它的函数 **malloc** 和 **free**，那么最终结果将是既笨拙又缓慢——这些函数被要求从用户模式调用，所以内核将不得不切换到用户模式去调用它们，然后它们又不得不掉转回到内核，还必须要对整个过程进行监控，等等。为了避免这一切，内核有许多十分熟悉的函数的自己的版本，它们包括 **malloc** 和 **free** 在内。

的确，内核提供了像 **malloc** 和 **free** 一样的两对独立的函数。第一对是 **kmalloc** 和 **kfree**，管理在内核段内分配的内存——这是真实地址已知的实际和物理内存块。第二对是 **vmalloc** 和 **vfree**，用于对内核使用的虚拟内存进行分配和释放。由 **kmalloc** 返回的内存更适合于类似设备驱动的程序来使用，因为它在物理内存里而且是物理连续的。不过，**kmalloc** 要比 **vmalloc** 所能使用的资源少，因为 **vmalloc** 还可以处理交换空间。

vmalloc 和 **vfree** 的一部分也是通过 **kmalloc** 和 **kfree** 来实现的，因为它们需要一部分不可交换的内存用于登记操作（bookkeeping）。**kmalloc** 和 **kfree** 又依次使用**__get_free_ pages**、**free_pages**，以及其它低级页面操作函数实现的。

在此我不对 **kmalloc** 和 **kfree** 进行解释，不过本书中提供了相关代码以供读者阅读（分别见 37043 和 37058 行）。我将要讨论的是更有意思的函数 **vmalloc** 和 **vfree**。

Vmalloc

38776：**vmalloc** 函数拥有一个参数，即要分配的内存区域的大小。函数返回指向分配区域的指针，如果无法分配就返回 **NULL**。

Vmalloc 可以分配内存的虚拟地址范围是由常量 **VMALLOC_START**（11081 行）

和 **VMALLOC_END**（11084 行）决定的。**VMALLOC_START** 从超过物理内存结束地址 8MB 的地方开始，以便对任何在这一区域错误的内核内存访问进行截获，**VMALLOC_END** 在接近可能的最大 32 位地址 4GB 的地址处。除非你的系统拥有比我的系统多得多的物理内存，否则这就意味着几乎整个 CPU 地址空间都潜在的可为 **Vmalloc** 所用。

38781：**vmalloc** 函数首先把要求的区域大小向上取整到地址更高的下一个页面边界，如果它不在一个页面的边界上的话。（**PAGE_ALIGN** 宏在 10842 行定义。）如果最终范围结果太小（0）或明显过大，则请求会被拒绝。

38784：利用 **get_vm_area** 来为 **size** 大小的块定位一段足够大的内存区域，这个函数接下来会进行介绍。

38788：通过调用 **vmalloc_area_pages**（38701 行）保证能够建立页表映射。

38792：返回被分配的区域。

get_vm_area

38727：**get_vm_area** 函数返回从 **VMALLOC_START** 到 **VMALLOC_END** 的一段自由内存区间。通常这就是 **vmalloc** 函数的工作；它还被用于我未曾提及的其它少数场合。调用程序有责任确保参数 **size** 是一个非零的页面大小的倍数值。

vmalloc 函数采用所谓的首次适应算法（first-fit algorithm），因为它返回一个指向定位区域的指针，该区域是它所能找到的第一个满足请求的区域。除此而外，还有最佳适应算法（best-fit algorithm），该算法选取足够满足需求的最小的一块可用自由区域进行分配，以及最坏适应算法（worst-fit algorithm），该算法总是分配最大的一块可用自由区间。每种分配方式都有优点和缺点，不过首次适应算法在这里对要达到的目的来讲，就已经非常简单、快捷而且足以满足要求了。

38732：分配一个 **struct vm_struct** 来代表新的区域。被分配的区域用一个有序链表，即 **vmlist**（38578 行）来维护，该链表是由 **struct vm_struct**s 构成的。包括 **struct vm_struct** 结构体的头文件被省略以节约空间，不过结构体的定义十分简单：

```
struct vm_struct {
  unsigned long flags;
  void* addr;
  unsigned long size;
  struct vm_struct* next;
};
```

如图 8-6 所示，链表的每一个元素都与单独一块已分配了的内存块相关联。形象的看起来，**get_vm_area** 函数的任务就是在已分配的区域之间找出足够宽的间隔。

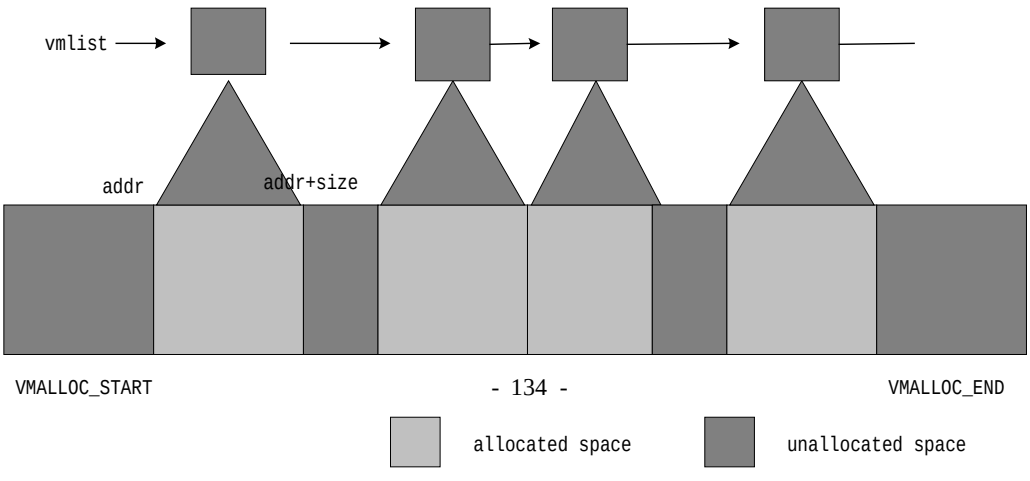

图 8.6 VMLIST 列表

38737：沿着链表进行循环。循环的结果要么是找到一个足够大的自由区间，要么是证明这样的区域不存在。它会先从 **VMALLOC_START** 开始尝试，然后挨个尝试紧随着每块被分配区域之后的地址。

38746：链表为空或者循环发现了一个足够大的新块；无论哪种情形，现在 **addr** 都是最小可用地址。填充新的 **struct vm_struct** 结构体，它将会被返回。

38747：给保留块增加一个页面的大小（x86 平台上是 4K），来捕获内核超出的内存——可能的话还包括下一个更高地址块下方的内存。因为在决定是否当前区域足够大的时候（38738 行）并没有把这些额外的空间算在内，那么保留区域可能会与接下来的一个相重叠，而且内核内存中超出而进入这个“额外”区域的部分也确实可能覆盖到被分配了的内存。事实不是如此吗？

事实不是这样。我们很容易证明 **addr** 总是页对准的，而且我们也已知道 **size** 总是页面大小的倍数。因此，**addr + size** 要小于接下来区域的开始地址，它至少是一整页。当然超出范围多于一页的内存会进入下一个区域，不过超出范围少于一页的内存就不会这样。

因为内核不会为额外内存建立页面映射，所以对它的错误访问将造成不可解决的页面错误（这在 Linux 的现代版本中几乎还未听说过！）。这将会带给内核一次痛苦的中断，不过那要比允许内核悄然无息地破坏自己的数据结构要好一些。至少你可以立刻知道这个折磨人的系统停机，它可以帮助你诊断问题所在；而后一种作法可能在内核已经破坏了你的磁盘之后，才能看出它的危害。

Vfree

38759：**vfree** 函数比 **vmalloc** 简单得多（要是把 **get_vm_area** 加进 **vmalloc** 至少是这样的），不过为了完整起见，我们还是要对 **vfree** 略为讨论。当然 **addr** 是要被释放的已分配区域的开头地址。

38763：在几项简洁而又完善的测试之后，函数沿着 **vmlist** 进行循环，搜索要释放的区域。这个线性查找过程使我想到一件有趣的事，假如采用一个如同 VMA 管理所用的 AVL 树那样的平衡树结构，也将会提高 **vmalloc** 和 **vfree** 函数的性能。

38764：当与 **addr** 相匹配的 **struct vm_struct** 被找到时，**vfree** 函数就把它从链表里分离出去，并释放该结构体和它所关联的页面，然后返回。每个 **struct vm_struct** 不仅记录它的初址还记录区域的大小，这一点对于 **get_vm_area** 是便利的，在这里同样也颇为便利，因此 **vfree** 函数是知道应该释放多大空间的。

38772：如果 **vfree** 函数在链表里找到了匹配项，它在此之前就应该已经返回了，所以没有找到匹配项。这是一个坏事，不过还未糟糕到不可收拾的地步。这样，**vfree** 函数以显示一个警告而结束。

转储内核（Dumping Core）

在一些情况之下，比如一个满是“臭虫”的程序试图去访问自己允许内存空间之外的内存时，进程可以转储内核。进行“转储内核”就是把一个进程的内存空间的映象（随同一些关于应用程序本身和其状态的识别信息一起）写入一个文件以备将来使用诸如 gdb 之类的调试器进行分析的过程（“内核”是一个差不多已经过时的内存术语）。

当然，或许你的代码从来不会犯这样的错误，但是这可能会发生在你隔壁不太聪明的程序员身上，而他可能在某一天会向你询问这件事，因此在此我要对此问题进行一些讨论。

不同的二进制处理程序完成转储内核的方式不同。（第 7 章里论述过二进制处理程序。）最常用的 Linux 二进制格式是 ELF，所以我们来看看 ELF 二进制处理程序是如何进行转储内核的。

Elf_core_dump

8748： **elf_core_dump** 函数由此开始。因为一个进程转储内存是由接受到一个信号而引起的（它也可能发送给自己，例如通过对 **about** 的调用），该信号编号在 **signr** 中被给出。**Signr** 对进程是否或者如何执行转储内存没有影响，但是在调试器里看内存文件的用户却想要知道是哪个信号导致内存转储的，它就像是一个关于出了什么错的提示一样。指向 **struct pt_regs**（11546 行）的 **regs** 参数包含一份对 CPU 寄存器的描述。**regs** 的重要性除了一些其它原因之外还在于它包含了 EIP 寄存器的内容，该寄存器是指令指针，它决定了收到信号时所执行的指令。

8771： 假如进程未通过一些基本检查则立即返回，这些检查中的第一个是确保 **dumpable** 标志被设置。进程的 **dumpable** 标志（16359 行）通常会被设置；它的清除主要是在进程改变其用户或组 ID 的时候。这似乎是一项安全措施。例如我们将不愿意创建一个被设定为 root 的不可读执行程序的可读内存文件——那会使得保证执行体不可读（出于安全考虑）的目的遭到失败。

elf_core_dump 函数此时也会返回，假如内存文件的大小限制使得连一个页面也无法转储，或者如果有其它线程要引用将要转储的内存。转储内核是和退出进程相关的，从用户的角度看来，只要进程任何一个线程还存在，它就没有消亡。

如果进程通过了这些测试，**elf_core_dump** 函数就继续运行并清除 **dumpable** 位以便它不会再次尝试转储进程的内存。（尽管这种情形不能会发生；我认为这只是预防式的编程设计。）

8785： 进入一个循环以对内存文件大小限制之内可以被转储的 VMA 个数进行计数。尽管 **elf_core_dump** 函数把计数植保存在叫做 **segs** 的变量里，它并不表明我们正对本章中所使用过的“内存段”进行计数。不要认为这个变量的名字有其它特别的附加涵义。

由于 **elf_core_dump** 函数在转储 VMA 之前要向内存文件写一些头部信息，而且这些头部的大小没有进行计算，因此输出结果可能会稍微超出内存文件的大小限制。这不难解决：一个简单的策略是在写入头部时递减 **limit**，并把循环计数移动到头部写入代码之后。实际解决方案要更麻烦一些，不过也并不是十分复杂。

8805： ELF 内存文件格式根据正式规范进行定义；第一个部分是描述文件的头部。结构体 **struct elfhdr** 类型（参见 14726 和 14541 行）定义了头部的格式，**elf_core_dump** 函数填写这个类型的一个局部变量 **elf**。

8827： 创建要转储到的文件名，并尝试打开这个文件。通过把 8828 行的**#if 0** 改变为**#if 1**，我们可以让内存文件名包括生成文件的执行程序的名字（或至少是名字的前 16 个字符——参见在 16406 行定义的 **struct task_struct** 的 **comm** 成员）。有的时候这是一个很有用的特性；能够一看到内存文件的名字就可以马上知道是什么应用生成的将是一件很好的事情。不过，这种行为并不标准，而且还有可能破坏已有代码——比如监视器脚本程序，它周期性地检查名为“code”的文件是否存在——所以缺省行为还是为遵守标准惯例而把文件命名为普通的“code”。尽管如此，发现这么一个可以调整的内核参数还是不错的。这个可选项也对 8756 行局部变量 **corefile** 那看似与众不同的定义方式进行了解释。

8853： 设置 **PF_DUMPCORE** 标识（16448 行），发出信号表明该进程正在转储内核。这个标识不在本书所涉及的任何代码中使用，它被用于读者将要了解的审计进程。审计进程（process accounting ）跟踪一个进程的资源使用情况和其它的一些相关信息——包括它是否在退出时转储内核——这些信息原本是用来帮助计算中心计算应向每个资源使用部门或用户收取多少费用的。这些日子都已经离我们远去了，难道我们不应该为此而感到高兴吗？

8855： 写入早先建立的 ELF 内存文件头部。这里要涉及一些隐含的流控制：定义在 8707 行的 **DUMP_WRITE** 宏使得 **elf_core_dump** 函数在写操作失败时关闭文件并返回。

8862：跟在 ELF 内存文件头部之后的是一系列节点（note）;它们中的每一个都有特殊目的，记录着有关进程的特定信息。我们将逐一对其论述。一个注解（数据类型是 **struct memelfnote**，8666 行）包括一个指向辅助数据（它的 **data** 成员）的指针和该数据的长度（它的 **datasz** 成员）；填写一个注解的大部分工作就是填充辅助数据结构，然后使该注解指向它。

有些信息被存储在若干个注解里。代码中没有对这种重复进行解释，但是其中至少有一部分原因是从 Unix 的变种中拷贝它们的行为方式。保持文件格式和其它平台一致有助于把诸如 gdb 这样的程序移植到 Linux 上来；少许重复要比延迟移植版本的进度和增加诸如此类的关键工具的维护复杂要好得多。

8865： 注解 0 在辅助数据结构体（类型 **struct_elf_prstatus**；参见 14774 行）里记录了进程的继承关系、信号量，以及 CPU 的使用情况。我们需要特别注意 8869 行的 **elf_core_dump**，它存储了引起进程转储内核的信号编号。所以当你（或者是你隔壁那个初级程序员）在一个内存文件上运行 gdb 而它显示”Program terminated with signal 11, Segmentation fault”的时候，你就会知道该信息是从哪里来的了。

8916： 注解 1 在辅助数据结构体 **psinfo**（属于类型 **struct_elf_prpsinfo**；参见 14813 行）里记录了进程的属主、状态，优先级等等信息。8922 行有一个虽然正确，但很不寻常的指向一个文字字符串常量的数组下标；被选择的字符是进程状态的一个记忆码。这与 ps 程序的 STAT 域报告的状态字是一样的（除非下标溢出）。更有意思的是 8945 行，代码把执行体的名字（如前所述，最多 16 个字符）复制进了注解。Gdb 和程序“文件”都用这个字段来报告是哪一个程序生成的内核转储。

8948： 节点 2 记录转储进程的 **struct task_struct**，这明显存储了关于该进程的大量必要信息。因为 **struct task_struct** 内的一些信息是由当调试器检查代码时便不再有效的指针组成的，**elf_core_dump** 函数随后还会分别转储一些指针所指向的信息——最紧要的，如进程的内存空间。

8954： 如果这个系统包含一个 FPU（浮点计算单元），那么就会据此而生成一个注解。否则，8957 行对所要存储的注解数目进行递减。

8968： 对于每个被创建的注解，都有一个描述该注解的头部；而注解本身会紧随其后。注解头是 **struct elf_phdr** 类型；参见 14727 和 14581 行它的定义。

8992： 这是写入进程内存空间的第一步。在这里，函数写入头部信息（又一次是 **phdr**），该头部描述了它将要写入的所有 VMA。

9016： 最后，**elf_core_dump** 函数才真正地写入它先前辛辛苦苦创建好的各个注解（内存文件）。

9022： 在文件里向前跳过 4K 到达下一个边界，内存文件真正的数据是从这里开始的。完成此项操作的 **DUMP_SEEK** 宏在 8710 行定义，像 **DUMP_WRITE** 宏一样，假如搜索失败它也会导致 **elf_core_dump** 函数的返回。

9024：在所有那些准备之后，这里的工作简直有些虎头蛇尾。不过，这才是转储内核的主要

部分：写入进程的每一个 VMA 直至先前计算出并保存在 **segs** 里的上限。接下来是少许收尾工作，然后 **elf_core_dump** 函数就完成了使命。