第2章 内存管理

内存管理子系统是操作系统最重要的部分之一。从早期计算开始，系统的内存大小就难以满足人们的需要。为了解决这个问题，可利用虚拟内存。虚拟内存通过当需要时在竞争的进程之间共享内存，使系统显得有比实际上更多的内存空间。

虚拟内存不仅仅使机器上的内存变多，内存管理子系统还提供以下功能：

- 大地址空间　操作系统使系统显得它有比实际上大得多的内存。虚拟内存可以比系统中的物理内存大许多倍。
- 保护　系统中每个进程有自己的虚拟地址空间。这些虚拟地址空间相互之间完全分离，所以运行一个应用的进程不能影响其他的进程。同样，硬件的虚拟内存机制允许内存区域被写保护。这样保护了代码和数据不被恶意应用重写。
- 内存映射　内存映射用来把映像和数据文件映像到一个进程的地址空间。在内存映射中，文件的内容被直接链接到进程的虚拟地址空间。
- 公平物理内存分配　内存管理子系统给予系统中运行的每个进程公平的一份系统物理内存。
- 共享虚拟内存　尽管虚拟内存允许进程拥有分隔的(虚拟)地址空间，有时你会需要进程共享内存。例如系统中可能会有几个进程运行命令解释 shell bash。最好是在物理内存中只有一份bash拷贝，所有运行bash的进程共享它；而不是有几份 bash拷贝，每个进程虚拟空间一个。动态库是另一个常见的几个进程共享执行代码的例子。

共享内存也可以被用作进程间通信(IPC)机制，两个或更多进程通过共有的内存交换信息。Linux支持Unix(tm) System V的共享内存IPC。

2.1 虚拟内存抽象模型

在考察Linux支持虚拟内存所使用的方法之前，考察一下抽象模型会有所帮助。

当处理器运行一个程序时，它从内存中读取一条指令并解码。在解码该指令过程中它可能需要取出或存放内存某个位置的内容。处理器然后执行该指令并移动到程序中下一条指令。这样处理器总是访问内存来取指令或取存数据。

在虚拟内存系统中以上所有的地址都是虚拟地址而不是物理地址。处理器基于由操作系统维护的一组表中的信息，将虚拟地址转换成物理地址。

为了使这种变换容易一些，虚拟内存和物理内存都被分为合适大小的块叫做“页(page)”。这些页都有同样的大小。它们可以不具有同样大小，但那样的话系统将很难管理。 Alpha AXP系统上Linux使用8KB字节大小的页，Intel x86系统上使用4KB字节大小的页。这些页中每一个都有一个唯一的号码：页帧号(Page Frame Number, PFN)。在这种分页模型中，一个虚拟地址由两部分组成：一个偏移和一个虚拟页帧号。如果页大小是 4KB字节，虚拟地址的11：0位包含偏移，12位及高位是虚拟页帧号。每当处理器面临一个虚拟地址时，它必须析取出偏移和虚拟页帧号。处理器必须将虚拟页帧号转换成物理的页帧号，然后在该物理页中正确的偏移

位置上进行访问。为了完成这些处理器要使用页表。

图1-2-1展示了两个进程的虚拟地址空间，进程 X和进程Y，每个都有自己的页表。

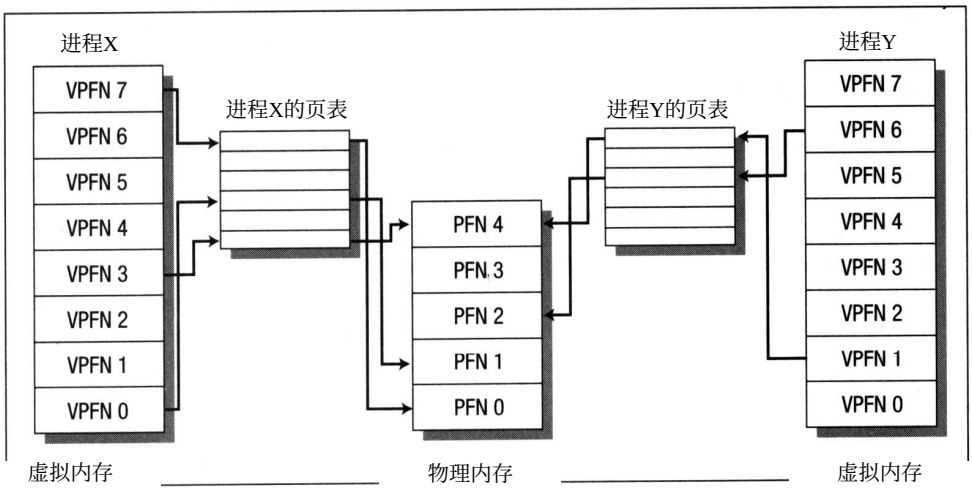

图1-2-1 虚拟地址到物理地址映射的抽象模型

这些页表将每个进程的虚拟页映射到内存中的物理页。图 1-2-1显示进程X的虚拟页帧号0映射到内存中物理页帧号 1，进程Y的虚拟页帧号1映射到物理页帧号4。理论上的页表中每一项包含下列信息：

- 有效标志，用来指示该页表项是否有效。
- 本项所描述的物理页帧号。
- 访问控制信息。它描述该页可以被怎样使用。它是否可以被写？是否包含可执行代码？

页表用虚拟页帧号作为偏移来访问。虚拟页帧号 5将是表中第6个元素(0是第1个)。

为了将一个虚拟地址转换成物理地址，处理器首先必须得到虚拟地址页帧号和在该虚拟页中的偏移。通过选取页大小为 2的幂，这可以很容易地屏蔽和移位得到。再看一下图 1-2-1，设页大小是0x2000字节(即十进制8192)，Y进程虚拟地址空间中一个地址为 0x2194，则处理器将把该地址转换为虚拟页帧号 1的偏移0x194。

处理器使用虚拟页帧号作为进程页表的索引来检索它的页表项。如果该偏移处页表项有效，处理器将从该项取出物理页帧号。如果该页表项无效，说明处理器访问了虚拟内存中不存在的区域。在这种情况下，处理器不能解析该地址，并且必须把控制传给操作系统来解决问题。

处理器如何通知操作系统一个正确的进程试图访问一个没有有效转换的虚拟地址，这是依处理器不同而不同的。无论如何处理器能够处理它，这被称作“ 页故障 (page fault) ”。操作系统被告知故障的虚拟地址和故障原因。

如果访问的是有效的页表项，处理器取出物理页帧号，并将它乘以页的大小以得到物理内存中该页的基地址。最后，处理器将偏移加到所需的指令或数据的地址。

再以上面的例子为例，进程 Y的虚拟页帧号 1映射到物理页帧号 4，起始于0x8000(4×0x2000)。加上0x194字节的偏移，最后得到物理地址 0x8194。

通过用这种方式映射虚拟地址和物理地址，虚拟内存能够以任何次序被映射到系统物理页。例如，在图1-2-1中进程X的虚拟页帧号0被映射到物理页帧号1，而虚拟页帧号7被映射到物理页帧号0，尽管在虚拟内存中它比虚拟页帧号 0要高。这说明了虚拟内存的一个有趣的副作用：虚拟内存页不必以任何特定次序出现在物理内存中。

2.1.1 请求调页

因为物理内存比虚拟内存小得多，操作系统必须小心以高效地利用物理内存。一种节约物理内存的方法是只装载被执行的程序当前正在使用的虚拟页。例如：一个数据库应用程序可能被运行来查询一个数据库。这种情况下，并非数据库的全部都需要被装入内存，而只是装入那些被检查的数据记录。如果数据库查询是一个搜索查询，那么把处理添加新记录的代码从数据库程序中装载进来是毫无意义的。这种只在被访问时把虚拟页装入内存的技巧叫请求调页。

当进程试图访问一个当前不在内存中的虚拟地址时，处理器将不能为被引用的虚拟页找到页表项。例如：图 1-2-1中，进程X的页表中没有虚拟页帧号 2的页表项，所以当 X试图从虚拟页帧号2中读一个地址时，处理器将不能把该地址转换成物理地址。这时处理器通知操作系统发生了页故障。

如果故障的虚拟地址无效，这意味着，该进程试图访问一个不该访问的地址。或许应用程序出了些错误，比如写内存中的随机地址。这种情况下操作系统将终止它，保护系统中其他进程不受此恶意进程破坏。

如果故障的虚拟地址有效，但它所引用的页当前不在内存中，操作系统就必须从磁盘上的映像中把适当的页取到内存中，相对来说，磁盘访问需要很长时间，所以进程必须等待一会儿直到该页被取出。如果还有其他可以运行的进程，操作系统将选择其中一个来运行。被取出的页会被写到一个空闲的物理页帧，该虚拟页帧号的页表项也被加入到进程页表中。然后进程从出现内存故障的机器指令处重新开始。这次进行虚拟内存访问时，处理器能够进行虚拟地址到物理地址的转换，进程将继续执行。

Linux使用请求调页把可执行映像装入进程虚拟内存中。每当一个命令被执行时，包含该命令的文件被打开，它的内容被映射到进程虚拟空间。这是通过修改描述进程内存映射的数据结构来完成的，被称作“ 内存映射 ”。然后，只有映像的开始部分被实际装入物理内存，映像其余部分留在磁盘上。随着进程的执行，它会产生页故障， Linux使用进程内存映射以决定映像的哪一部分被装入内存去执行。

2.1.2 交换

如果一个进程想将一个虚拟页装入物理内存，而又没有可使用的空闲物理页，操作系统就必须淘汰物理内存中的其他页来为此页腾出空间。

如果从物理内存中被淘汰的页来自于一个映像或数据文件，并且还没有被写过，则该页不必被保存，它可以被丢掉。如果有进程再需要该页时就可以把它从映像或数据文件中取回内存。

然而，如果该页被修改过，操作系统必须保留该页的内容以便晚些时候再被访问。这种页称为“ 脏(dirty) ”页，当它被从内存中删除时，将被保存在一个称为交换文件的特殊文件中。

相对于处理器和物理内存的速度，访问交换文件要很长时间，操作系统必须在将页写到磁盘以及再次使用时取回内存的问题上花费心机。

如果用来决定哪一页被淘汰或交换的算法(交换算法)不够高效的话，就可能出现称为“抖动”的情况。在这种情况下，页面总是被写到磁盘又读回来，操作系统忙于此而不能进行真正的工作。例如，如果图1-2-1中物理页帧号1经常被访问，它就不是一个交换到硬盘上的好的候选页。一个进程当前正使用的页的集合叫做“工作集 (working set)”。一个有效的交换算法将确保所有进程的工作集都在物理内存中。

Linux使用“最近最少使用(Least Recently Used, LRU)”页面调度技巧来公平地选择哪个页可以从系统中删除。这种设计中系统中每个页都有一个“年龄”，年龄随页面被访问而改变。页面被访问越多它越年轻；被访问越少越年老也就越陈旧。年老的页是用于交换的最佳候选页。

2.1.3 共享虚拟内存

虚拟内存使得几个进程共享内存变得容易。所有的内存访问都是通过页表进行，并且每个进程都有自己的独立的页表。为了使两个进程共享—物理页内存，该物理页帧号必须在它们两个的页表中都出现。

图1-2-1显示了共享物理页帧号4的两个进程。对进程X来说是虚拟页帧号4，而对进程Y来说是虚拟页帧号6。这说明了共享页有趣的一点：共享物理页不必存在于共享它的进程的虚拟内存的相同位置。

2.1.4 物理寻址模式和虚拟寻址模式

操作系统本身运行在虚拟内存中没有什么意义。操作系统必须维持自己的页表是件非常痛苦的事情。大多数多用途处理器在支持虚拟寻址模式的同时支持物理寻址模式。物理寻址模式不需要页表，在这种模式下处理器不会进行任何地址转换。 Linux内核就是要运行在物理寻址模式下。

Alpha AXP处理器没有特殊的物理寻址模式。相反，它把地址空间分成几个区，指定其中两个作为物理映射地址区。这种核心地址空间被称为 KSEG地址空间，它包括所有oxfffffc0000000000以上的地址。为了执行链接在KSEG中的代码(定义为内核代码)或存取其中的数据，代码必须执行于核心模式下。Alpha上的Linux内核被链接成从地址oxfffffc0000310000开始执行。

2.1.5 访问控制

页表项中也包含访问控制信息。因为处理器要使用页表项来把进程虚拟地址映射到物理地址，它可以方便地使用访问控制信息来检查并保证进程没有以其不应该采用的方式访问内存。

有许多原因导致限制访问内存区域。有些内存，比如那些包含可执行代码的，自然地就是只读内存；操作系统不应该允许进程写数据到它的可执行代码中。相反，包含数据的则可以被写，但是试图把它当作指令来执行就会失败。大部分处理器至少有两种执行模式：用户态(用户模式)和核心态(核心模式)。我们不希望核心代码被用户执行或核心数据结构被访问，除非处理器运行在核心态下。

访问控制信息保留在PTE(页表项)中，并且是处理器相关的。图1-2-2显示了Alpha AXP 的PTE。其各字段意义如下：

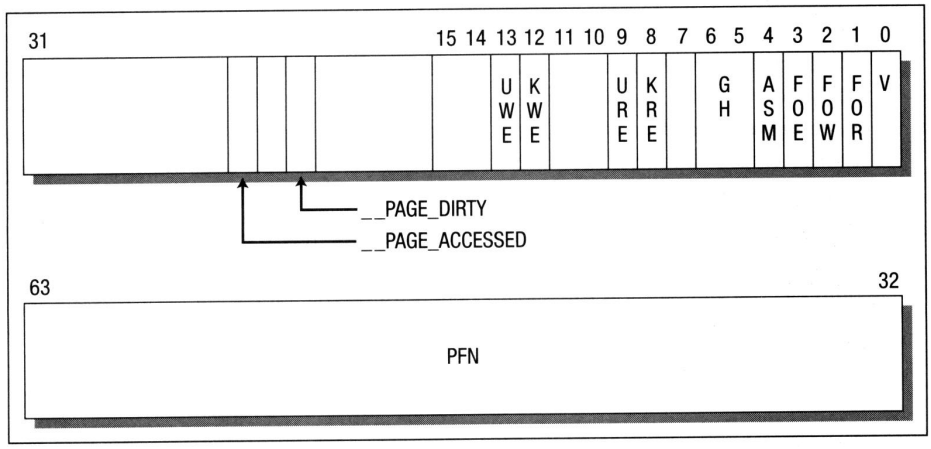

图1-2-2 Alpha AXP页表项

- V 有效性。如果置位则PTE有效。
- FOE 执行时故障，当试图执行本页中的指令时，处理器报告页故障并将控制传给操作系统。
- FOW 写时故障，当试图写本页时发出如上页故障。
- FOR 读时故障，当试图读本页时发出如上页故障。
- ASM 地址空间匹配，当操作系统想要仅清除转换缓冲区中一些项时用到。
- KRE 运行于核心态的代码可以读此页。
- URE 运行于用户态的代码可以读此页。
- GH 粒度暗示，当用一个而不是多个转换缓冲区项映射一整块时用到。
- KWE 运行于核心态的代码可以写此页。
- UWE 运行于用户态的代码可以写此页。
- PFN 页帧号。对于V位置位的PTE，本字段包括物理页帧号；对于无效PTE，如果本字段非0，则包含本页在交换文件中的位置的信息。

下面两个位在Linux中被定义并使用：

- _PAGE_DIRTY 如果置位，此页需要被写到交换文件中。
- _PAGE_ACCESSED Linux用来标识一个页已经被访问过。

2.2 高速缓存

如果你使用上面的理论模型实现一个系统，它可以工作但效率不高。操作系统设计者和处理器设计者都试图从系统中得到更高的性能。除了把处理器、内存等做得更快以外，最好的方法就是维持对有用信息和数据的高速缓存，以使一些操作更快。 Linux使用几种与内存管理有关的缓存：

- 缓冲区缓存 缓冲区缓存包含被块设备驱动程序使用的数据缓冲区。这些缓冲区是固定大小的(比如512字节)并包含从块设备中读出的或要写入块设备的信息块。块设备是只能通过读写固定大小的数据块来访问的设备。所有的硬盘都是块设备。

缓冲区缓存通过设备标识符和想要的块号来索引，并用来快速地寻找一块数据。块设备只能通过缓冲区缓存访问。如果数据能在缓冲区缓存中找到则不用从物理块设备 (如磁盘)中去读数据，从而可以很快地完成访问。

- 页缓存　用来加速对磁盘上的映像或数据的访问。它每次缓存一个文件中一页的逻辑内容并通过文件和文件内偏移来访问。所有的页都从磁盘读到内存，它们被缓存在页缓存中。
- 交换缓存　只有被修改的(脏的)页被存到交换文件中。只要这些页被写入交换文件以后没有被修改，那么下一次当该页被交换出去时就没有必要把它写到交换文件中，因为它已经在交换文件中了。该页可以简单地被淘汰掉。在交换负担严重的系统中这可以省去许多不必要的、费时的磁盘操作。
- 硬件缓存　一个普遍实现的硬件缓存，位于处理器内：页表项缓存。在这种情况下，当处理器需要时并不总是直接读取页表项，而是把页的转换信息缓存起来。这就是转换旁视缓冲器(TLB)，它们包含系统中一个或多个进程的页表项的缓存副本。

当引用虚拟空间地址时，处理器将试图找到一个匹配的 TLB项。如果能找到，它就可以直接将虚拟地址转换为物理地址，并对数据进行正确的操作。如果处理器找不到匹配的 TLB项，就必须要得到操作系统的帮助。它通过通知操作系统发生了 TLB失效来请求帮助。有一个系统相关的机制被用来将该异常传送到操作系统中完成相应操作的代码。操作系统为地址映射产生一个新的 TLB项。当异常清除后，处理器将再次尝试转换虚拟地址。这次将正常工作，因为现在TLB中有一个有效项为该地址进行转换。

使用缓存的缺点(无论是硬件的还是其他的)：为了节约开销Linux必须用更多的时间和空间来维护这些缓存，如果这些缓存崩溃，系统也将垮掉。

2.3 Linux页表

Linux假定系统中有三级页表。所访问的每级页表包含下一级页表的页帧号。图 1-2-3显示了一个虚拟地址如何被分解成几个字段，每个字段包含一个特定页表的偏移。为了将一个虚拟地址转换成一个物理地址，处理器必须取出每一个字段的内容，把它变换成包含页表的物理页的偏移并读出下一级页表的页帧号。这个过程被重复三次，直到包含该虚拟地址的物理页的页帧号被找到。然后虚拟地址的最后一个字段、字节偏移，被用来查找页中的数据。

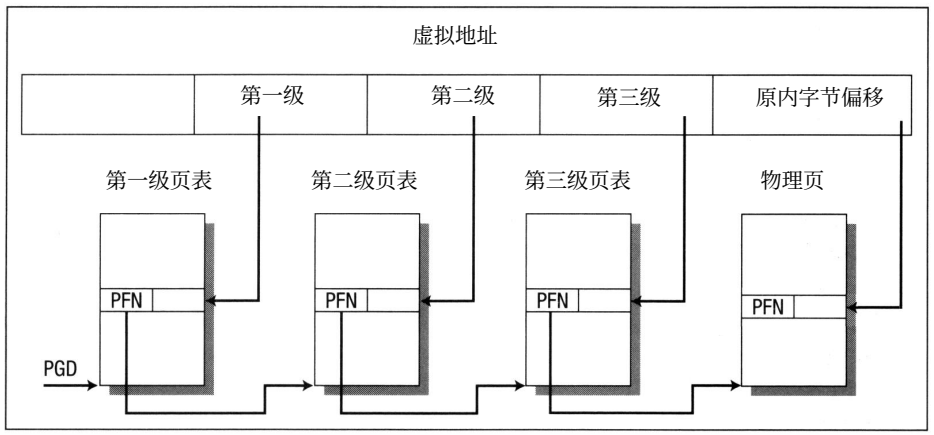

图1-2-3　三级页表

Linux运行的每一个平台都必须提供转换宏，使得内核可以遍历一个特定进程的页表。这样，内核不必知道页表项的格式以及它们如何组织。这是一种很成功的方法： Linux为Alpha处理器提供三级页表，而对Intel x86处理器，只提供两级页表，但使用相同的页表处理代码。

2.4 页分配和回收

系统中有许多对物理页的请求。例如，当一个映像被装载入内存时操作系统需要分配页。当映像结束执行并卸载时这些页又将被释放。物理页的另一个使用是保存内核专用的数据结构，比如页表本身。用来进行页分配和回收的机制和数据结构或许是维持虚拟内存子系统高效的最关键的一点。

系统中所有物理页都用 mem_map数据结构描述，它是一个在启动时被初始化的mem_map_t结构的列表。每个mem_map_t，描述系统中一个物理页。一些重要的字段有 (仅对内存管理而言)：

- count 本页使用者计数。当该页被许多进程共享时计数将大于 1。
- age 描述本页的年龄，用来判断该页是否为淘汰或交换的好的候选。
- map_nr men_map_t描述的物理页的页帧号。

Free_area向量被页分配代码用来寻找和释放页。整个缓冲区管理设计是基于这个机制的支持，并且对于代码来说，页大小和处理器所使用的物理分页机制是无关的。

Free_area的每个元素包含页块的信息。数组中第一个元素描述一页，下一个描述两页的一块，再下一个描述 4页的一块，依此以 2的幂增长。 List元素用作一个队列头并含有指向mem_map数组中page数据结构的指针，空闲的页块在这里排成队列。 map是指向一个位图的指针，该位图跟踪被分配的该大小的页组。如果第几个块空闲，则位图中第几位将被置位 N。

图1-2-4展示了free_area结构。元素0有一个空闲页(页帧号0)，而元素2有两个空闲的4页块，第一个开始于页帧号4而第二个开始于页帧号56。

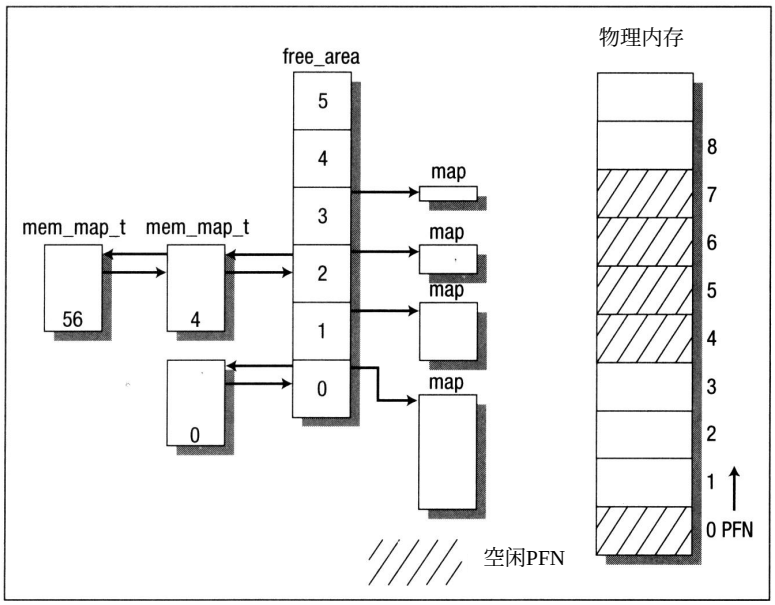

图1-2-4 free-area数据结构

2.4.1 页分配

Linux使用Buddy算法2 来高效地分配和回收页块。页分配代码试图分配一个或多个物理页的一块。页面分配是以 2的幂大小的块来进行，这意味着它可能分配 1页块、2页块、4页块等。只要系统中有足够的空间页来满足请求 (nr_free_pages > min_free_pages)，分配代码将搜索free_aree来寻找请求大小的一个页块。例如，数组中元素 2有一个内存映射来描述系统中 4页长的块的空闲和分配。

分配算法首先搜索请求大小的页块。它沿 free_area中list元素处排成队列的空闲页面链进行。如果没有所请求大小的页块是空闲的，下一个大小的块 (两倍于所请求的大小)将被查找，这个过程一直进行到 free_area整个被检查过或找到一个页块。若找到的页块比请求的大，则它必须被分开直到得到合适大小的一块。因为块大小都是 2的幂，所以分块的过程很容易，只需把块等分。空闲的块排到合适的队列中，被分配的块返回给调用者。

例如，图1-2-4中如果一个2页块被请求，第一个 4页块(开始于页帧号 4)将被分成两个2页块。第一个，也就是开始于页帧号 4的将被作为分配的页返回给调用者；而第二块，开始于页帧号6，将作为空闲的2页块排进free_area数组的元素1的队列中。

2.4.2 页回收

页块的分配可能会把内存分段，把大的空闲页块分开成小的。而页回收代码在可能的时候把页重新组合成大的空闲页块。事实上页块大小很重要，因为它使得可以容易地把块组合成更大的块。

当一个页块被释放时，将检查相邻的或称伙伴的同样大小的块是否空闲。如果是，它和新释放的页块被组合在一起形成一个新的下一个大小的空闲页块。每次两个页块被组合成更大的空闲页块时，页回收代码将试图再将该块组合成还要大的块。这样，空闲页块可以任意大，只要内存使用允许。

例如，在图1-2-4中，如果页帧号 1被释放，那它将和已经空闲的页帧号 0组合起来，并作为2页空闲块排进free_area的元素1的队列中。

2.5 内存映射

当一个映像被执行时，该可执行映像的内容必须被放到进程的虚拟地址空间。对于可执行映像链接到的共享库也是如此。可执行文件并非真正地被读到物理内存，而只是链接到进程的虚拟内存。然后，随着运行的应用对程序部分的引用，该映像被从可执行映像读到内存。这种将一个映像链到一个进程的虚拟地址空间的技术也被称为内存映射 (memery mapping)。

每个进程的虚拟内存都通过一个 mm_struct数据结构表示。它包含当前正执行的映像 (如bash)的信息以及一些指向 vm_area_struct数据结构的指针。每个 vm_area_struct数据结构描述虚拟内存区的开始和结束、进程对该内存的访问权以及对该内存的一组操作。这些操作是处理这个虚拟内存区时 Linux必须使用的一组例程。例如，当进程试图访问此虚拟内存，却发现(通过页故障)该内存并没有真正地在物理内存中时，虚拟内存操作之一将进行正确的动作。这就是nopage操作。nopage操作在Linux请求调一个可执行映像的页进内存时使用。

当一个可执行映像被映射到进程虚拟地址空间时，一组 vm_area_strnct数据结构将被产生。

每个vm_area_struct数据结构表示可执行映像的一部分；是可执行代码，或是初始化的数据 (变量)，以及未初始化数据等。 Linux支持一些标准虚拟内存操作并且当 vm_area_struct数据结构被创建时，相应的虚拟内存操作集将和它们关联起来。

2.6 请求调页

当一个可执行映像被内存映射到进程的虚拟内存时，它就可以开始执行。就在刚开始将该映像装入物理内存时，它就很快会访问一个还未在物理内存中的虚拟内存区 (图1-2-5)。当进程访问一个没有有效页表项的虚拟地址时，处理器将向 Linux报告页故障。页故障描述了发生页故障的虚拟地址以及引起页故障的内存访问类型。

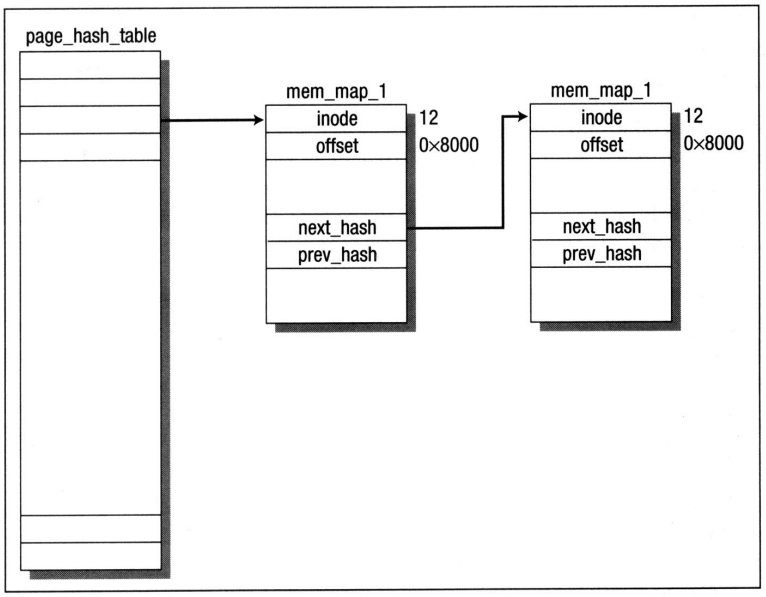

图1-2-5 虚拟内存区

Linux必须找到表示发生页故障的内存区域的 vm_area_struct数据结构。因为查找vm_area_struct数据结构是高效处理页故障的关键，它们被一起链接到一个 AVL(Addsen_Vebskii和Landis)树结构中，如果不存在发生故障的虚拟地址的vm_area_struct数据结构，那么该进程就访问了一个非法的虚拟地址。Linux将通知该进程，发送一个SIGSEGV信号，并且如果该进程没有此信号的处理器就会被终止。

Linux然后检查发生的页故障的类型，与本虚拟内存区所允许的访问类型比较。如果进程以非法方式访问内存，比如写一个只读区，它同样将会被通知发生了内存错误。

现在Linux判定页故障是合法的，则必须处理。 Linux必须区分在交换文件中的页和在磁盘上其他地方并且是可执行映像的一部分的页。它通过查看故障虚拟地址的页表项来区分。

如果该页的页表项无效但不空，则页故障是当前保存在交换文件中的一页。对于 Alpha AXP页表项来说，这些就是有效位没有置位但 PFN字段值又非0的页表项。这种情况下， PFN字段包含一些信息指明该页在交换文件中哪里 (以及哪个交换文件)被保存。交换文件中的页如何处理将在本章后面描述。

并非所有的vm_area_struct都有一个虚拟内存操作集，即使有，有的也可能没有 nopage操

作。这是因为默认(缺省)情况下Linux将通过分配一个新物理页并创建一个有效页表项来完成访问。如果一个虚拟内存区没有nopage操作，Linux将使用默认方式。

一般的Linux nopage操作用在内存映射的可执行映像上，并使用页缓存把请求的页装入物理内存。

不管怎样，所请求的页都将被装入物理内存，进程页表被更新。或许需要一些硬件相关的动作来更新这些项，特别是当处理器使用了转换旁视缓冲器时。在此页故障被处理之后，它可以被清除掉，进程在引起故障虚存访问的指令处重启动。

2.7 Linux页缓存

Linux页缓存的作用是加速对磁盘上文件的访问。内存映射的文件每次读取一页，并且这些页就保存在页缓存中。图1-2-6显示了由page_hash_table(一个指向mem_map_t数据结构的指针)组成的页缓存。Linux中每个文件由一个VFS inode数据结构标识(在第7章中描述)，并且每个VFS inode都是唯一的，完全描述一个且唯一的一个文件。页表的索引就由文件的 VFS inode和文件内的偏移导出。

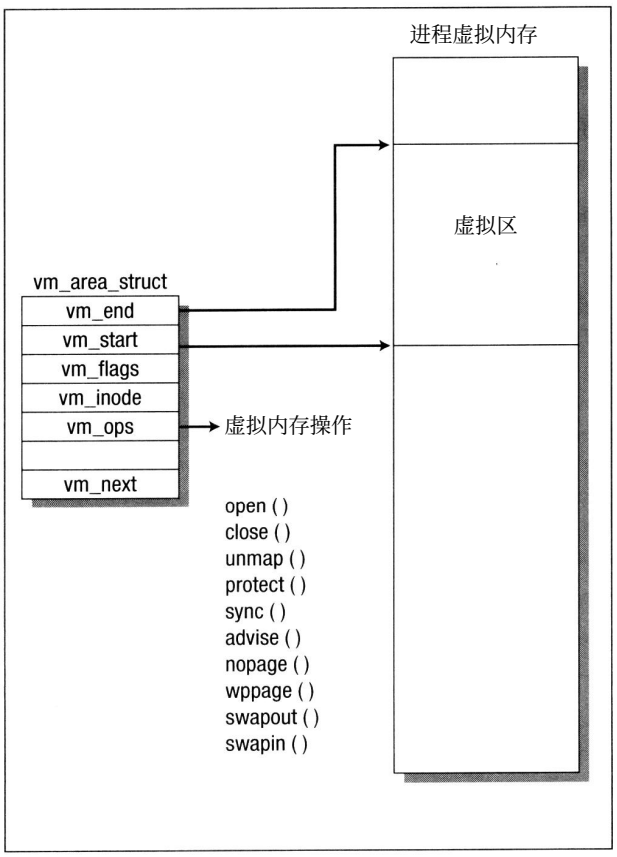

图1-2-6 Linux页缓存

每当一页要从内存映射文件读出时(比如当在请求调页时需要被读回内存)，该页通过页缓存被读出。如果该页在缓存中，一个代表该页的 mem_map_t数据结构指针被返回给页故障处

理代码，否则该页必须从保存相应映像的文件系统中被读到内存。 Linux分配一个物理页然后从磁盘文件中读取该页。

随着映像的读入和执行页缓存不断增长，当页不再被需要时会被移出缓存，比如说一个映像不再被任何进程使用时。随着 Linux不断使用内存，它可能开始缺少物理页，这时 Linux将减少页缓存的大小。

2.8 页换出和淘汰

当物理内存变得短缺时， Linux内存管理子系统必须试着释放物理页。这项任务落在内核交换守护进程(kswapd)头上。内核交换守护进程是一种特殊类型的进程 —— 一个内核线程。内核线程是没有虚拟内存的进程，它们在核态下的物理地址空间中运行。内核交换守护进程的命名有一点不准确，因为它并不仅是把页面交换到系统交换文件中。它的作用是确保系统中有足够空闲页来保持内存管理系统高效运行。

内核交换守护进程 (kswapd)在启动时由内核初始化进程启动，并等待内核交换定时器周期性超时。每当定时器超时，交换守护进程就去看系统中空闲页的数目是否太少了。它使用两个变量， free_pages_high 和free_pages_low来决定是否应释放一些页。只要系统中空闲页数目多于free_page_high，内核交换守护进程就什么也不做，它再次“睡眠”直到其定时器再次超时。在进行这次检查时，内核交换守护进程把当前正被写出到交换文件的页也计算在内。它保持一个这种页的计数于nr_async_pages中，每当有一页排队等待被写出到交换文件时，则递增；而当交换设备的写完成后递减。 Free_pages_low和free_pages_high在系统启动时被置值并与系统中物理页数有关。如果系统中的空闲页数少于 free_pages_high，甚至少于free_pages_low，则内核交换守护进程将试用三种方法来减少系统中使用的物理页数：

- 减少缓冲区和页缓存的大小。
- 交换出System V共享内存页。
- 换出及淘汰页。

如果系统中空闲页数低于 free_pages_low，内核交换守护进程将在下次运行前试着释放 6页，否则将试着释放3页。上面三种方法将依次被使用直到足够的页被释放。内核交换守护进程会记住上次它是用哪种方法来释放物理页。每当运行时它将开始试着用上次成功的方法释放页。

当释放完足够页后，交换守护进程再次睡眠直到定时器超时。如果内核交换守护进程释放页的原因是系统中空闲页数少于 free_pages_low，那么它仅睡眠通常一半的时间。一旦系统中空闲页多于free_pages_low，内核交换守护进程将在检查前睡眠长些。

2.8.1 减少缓冲区和页缓存大小

保存在页缓存和缓冲区缓存中的页是被释放到 free_area 向量中的很好的候选。页缓存 (包含内存映射文件)，可能包含着不必要的占据系统内存的页。缓冲区缓存与此类似，它包含将从物理设备中读出或写入的缓冲区，也可能包含不需要的缓冲区。当系统中物理页用完后，从这些缓存中淘汰页相对要容易些，因为它不需要写物理设备 (不像把页交换出内存)。淘汰这些页除了使对物理设备和内存映射文件的访问变慢外，没有其他太多不良作用。然而，如果从这些缓存中进行页淘汰是公平进行的，那么所有的进程将受到相同影响。

每次内核交换守护进程试图压缩这些缓存时，就检查 mem_page向量中的一块页面来看是否有页面能被淘汰出物理内存。各内核交换守护进程在强制交换时，所检查的那块页面将很大，也就是系统中的空闲页数已经降到非常低了。页面块用循环方式检查，每次试图压缩缓存时都将检查不同的一块页面。这也被称为时钟算法，因为就像时钟的分针一样，每次检查整个mem_map页向量的几页。

所检查的每一页都要看一下它是否被缓存在页缓冲区或缓冲区缓存中。需要注意，这时不考虑淘汰共享页并且一个页不能同时位于这两种缓存中。如果该页不在任何这两种缓存中则检查mem_map页向量的下一页。

页面被缓存在缓冲区缓存中 (或说缓冲区所在的页被缓存)来使缓冲区分配和回收更高效。内存映射压缩代码试着释放包含在正在检查的页面中的缓冲区。如果所有的缓冲区都被释放了，那么包含它们的页就被释放了。如果被检查的页位于 Linux页缓存中，它将被移出页缓存并释放。

当这次尝试释放了足够多的页后，内核交换守护进程将等待直到它下一次定期被唤醒。因为被释放的页不是任何进程虚拟内存的一部分 (它们是被缓存的页)，所以不需要更新任何页表。如果没有足够的缓存页被淘汰，则交换守护进程将试着换出一些共享页。

2.8.2 换出System V共享内存页

System V共享内存是一种进程间通信机制，它使得两个或多个进程可以共享虚拟内存用以在它们之间传递信息。进程如何以这种方式共享内存将在第 4章中更详细描述，现在只需要知道 每个 System V共享内存区通过一个 shmid_ds数据结构描述。它包含一个指向vm_area_struct数据结构列表的指针，每个数据结构被一个共享虚拟内存区的进程所用。Vm_area_struct数据结构描述此System V共享内存区位于每个进程的虚拟内存的什么地方。该System V共享内存区的所有vm_area_struct数据结构通过vm_next_shared和vm_prev_shared指针链接在一起。 shared_ds数据结构还包含一个页表项列表，它们中的每个描述一个虚拟共享页映射到的物理页。

内核交换守护进程在换出 System V共享内存页时同样使用时针算法。每次运行时它都记住最近换出了哪个共享虚拟内存区中的哪一页。它通过两个索引完成这种功能，一个是shmid_ds数据结构集的索引，另一个是这个 System V共享虚拟内存区中页表项列表的索引。这将保证它公平地牺牲System V共享虚拟内存区。

因为对于一个给出的System V共享内存页来说，其物理页帧号包含在所有共享了虚拟内存区的进程的页表中，内核交换守护进程必须修改所有这些页表来表明现在该页不再位于内存中而是保存在交换文件中。对于每个交换出去的共享页，内核交换守护进程找到每一个共享进程的页表中的页表项 (通过vm_area_struct数据结构中的一个指针)。如果此System V共享内存页的这个进程页表项有效，则把它变成无效且换出该页表项并把该 (共享)页的使用者计数减1。一个交换的系统V共享页表项格式包含一个 shmid_ds数据结构集的索引和一个此 System V共享内存区页表项的索引。

如果所有共享进程的页表被修改后页的使用计数为 0，则共享页就可以被写到交换文件中。该System V共享内存区的shmid_ds数据结构所指向的页表项列表将被一个换出页表项替代。一个换出页表项是无效的但包含一个打开的交换文件集的索引和一个该文件中的偏移，从偏

移处可以找到换出页，在此页需要被读回物理内存时将使用这些信息。

2.8.3 换出和淘汰页

交换守护进程依次查看系统中每个进程来看它是否是交换出的好的候选。好的候选是指那些可以被换出(有些不能)的进程，并且它有一个或多个页可以被换出或从内存中淘汰。仅当页中的内容不能从另一种方法获得时，页面才会被从物理内存交换出到系统的交换文件中。

一个可执行映像的许多内容来自于映像的文件并且可以容易地从文件中重读出来。例如，一个映像的可执行指令永远不会被映像修改，所以从来也不会被写到交换文件中。这些页可以简单地被扔掉；当再次被进程引用时，它们将从可执行映像中被读入内存。

一旦将被交换的进程被定位，交换守护进程查看它所有的共享内存区来寻找未共享并且锁住的区域。Linux并不是把所选中进程的全部可交换页换出，而是只移出少量的页。页面如果被锁住就不能被换出或淘汰。

Linux交换算法使用页年龄。每一个页有一个计数器 (保存在mem_map_t数据结构中)以提供给内核交换守护进程一些该页是否值得换出的信息。页面在不用时会变老而被访问时将重新年轻；交换守护进程只换出老的页面。一个页被分配时的缺省动作是给它一个初始年龄 3。每当它被访问时，它的年龄被加 3直到最大值20。每次内核交换守护进程运行时，要使页面变老：把它们的年龄减 1。这些缺省动作可以被改变，为此它们 (及其他有关交换的信息)被保存在swap_control数据结构中。

如果页面是老的 (age=0)，交换守护进程将进一步处理之。“ 脏 ”页是可以被换出的页。Linux使用PTE一个体系结构相关的位来用这种方式描述页 (见图1-2-2)。然而，并非所有“ 脏 ”页需要被写到交换文件中。进程的每个虚拟内存区可以有自己的交换操作 (由vm_area_struct中vm_ops指针所指向)并被使用。否则，交换守护进程将在交换文件中分配一页并把该页写出到设备上。

该页的页表项将被一个标记为无效但包含该页在交换文件中位置信息的节点所代替。这些信息是交换文件中该页被保存处的偏移和使用哪个交换文件的指示信息。无论使用什么交换方法，原先的物理页将通过放回到 free_area中而释放。干净(或说不“ 脏 ”)的页可以被扔掉并放回free_area中来重用。

如果足够的可交换进程页被换出或淘汰了，交换守护进程将再次睡眠。下次它醒来时将考虑系统中下一个进程。这样，交换守护进程一点一点地移走每个进程的物理页直到系统再次达到平衡。这比交换出整个进程要公平得多。

2.9 交换缓存

在把页面换出到交换文件时， Linux尽量避免写页。有的时候一页既在交换文件中又在物理内存中。当一个页面被换出内存并在又一次进程访问时被读进内存，则会出现这种情况。只要内存中的页没有被写，交换文件中的副本就保持有效。

Linux使用交换缓存来跟踪这些页。交换缓存是一个页表项列表，每个代表系统中一物理页。这是一个被换出的页的页表项并描述该页保存在哪个交换文件中以及在交换文件中的位置。如果一个交换缓存项非空，它代表保存在交换文件中并且未被修改的一页。若该页在以后被修改(通过写入)，该项将被从交换缓存中移去。

当Linux需要换出一物理页到交换文件中时，它将参考交换缓存，并且若有该页的一个有效项，就不需要写该页到交换文件中。这是因为内存中这一页在最近一次由交换文件中读出后还没被修改过。

交换缓存中的项是换出的页的页表项，它们被标识为无效但包含一些信息使 Linux可以找到正确的交换文件及该文件中正确的页。

2.10 页换入

存入到交换文件中的脏页可能再次被需要，例如当一个应用程序要写一段内容保存在换出的物理页的虚拟内存区时。访问一个不在物理内存中的虚拟内存页将导致页故障的发生。页故障是处理器在通知操作系统，它不能将一个虚拟地址转换到物理地址。在这里是因为当页被换出时描述该页虚拟内存的页表项被标识成无效。处理器不能处理虚拟地址到物理地址的转换，所以它把控制传回操作系统，并同时描述发生页故障的虚拟地址和故障原因。这些信息的格式以及处理器如何将控制传给操作系统是处理器相关的。处理器相关的页故障处理代码必须找到vm_area_struct数据结构，该数据结构描述发生页故障处的虚拟内存区。它通过查找进程的vm_area_struct数据结构来进行，直到找到包含故障地址的那个。这是一段对时间要求很严格的代码，并且进程的 vm_area_struct数据结构被组织成使这种查找耗费尽量少的时间。

执行完处理器相关动作并发现故障虚拟地址是位于合法的虚拟内存区中，页故障处理过程对于可运行Linux的所有处理器，就成为通用并且适用的。通用的页故障处理代码查找故障虚拟地址的页表项。如果找到的页表项从属于一个换出的页， Linux必须把该页换回物理内存。被换出的页表项格式是处理器相关的，但所有处理器都把这些页标识为无效，并把在交换文件中定位该页所需的信息放进页表项中，为了把该页换入物理内存， Linux需要使用这些信息。

这时， Linux知道了故障的虚拟地址，并有一个页表项包含该页被交换出到何处的信息。vm_area_struct数据结构可能含有一个指针指向一个例程，用于把它描述的虚拟内存区中任何页换回物理内存，这是它的swapin操作。如果有此虚拟内存区的swapin操作， Linux将使用它。事实上，它正是 System V共享内存页的处理方式：因为它需要特殊处理，来适应换出的 System V共享页的格式与普通换出页格式的差异。也可能没有 swapin操作，这时Linux将认为它是一个普通页，不需要特殊处理。它将分配一个空闲物理页并从交换文件中读回被换出的页。该页在交换文件中的位置信息(及哪个交换文件)将从相应无效页表项中获得。

如果引起页故障的访问不是写访问，则该页被留在交换缓存中，并且其页表项不被标识成可写。如果该页以后被写，将发生另一个页故障，并且那时该页将被标识为“脏”，其页表项从交换缓存中被移去。若该页没被写并且它需要再次被换出时， Linux可以免去把该页写到交换文件，因为它已经在交换文件中了。

如果引起把页从交换文件读回的访问是写操作，则此页被从交换缓存中移出，并且其页表项被标识为“脏”和“可写”。