第 13 章　虚　　存

13.1　简　　介

操作系统的主要功能之一就是高效地管理内存资源。每一部计算机系统都有一个高速的，随机访问的主(primary)存，或称为主(main)存、物理内存，或更简单地称为内存。它的访问时间一般均为 CPU 周期的几倍①，程序可以直接访问内存里的代码或数据。这样的存储设备相对比较昂贵，所有一般都是有限的。系统通常都使用二级存储设备(通常有磁盘，网络上的其他机器)来存储内存容纳不了的信息。访问这些设备的时间要比访问内存的时间多若干个数量级，而且需要操作系统的显式参与，内核中的内存管理子系统负责完成内存和二级存储器之间的数据分布。它直接操作称为存储管理单元(MMU)的部件，该部件完成 CPU 与主存间的数据传输。

如果没有了内存管理，操作系统的生命周期要简单的多，系统每次只将一个程序加载到地址固定的连续空间中，这样就大大简化厂链接和加载的任务，而将硬件从繁锁的地址转换中解脱了出来。所有的寻址操作都直接发给物理内存，而程序则独占整个系统(当然是与操作系统共享)。这是运行单个程序最快最高效的方式。

上面说的那种情形经常出现于小型实时嵌入微处理系统中。但对通用系统而言，缺点是很明显的。首先，程序的尺寸受限于内存的大小，不可能运行大程序；其次，程序被加载到内存后，一旦它等待 I/O 操作完成，整个系统就进入空闲状态。尽管系统非常适于小的单独的程序，但绝对不可能提供多道程序环境。

现在暂时认为某种形式的内存管理是必须的，让我们罗列一下我们希望它能做些什么：

· 运行比内存还要大的程序。理想情况下，应该可以运行任意大小的程序。

· 可以运行只加载厂部分的程序，缩短程序的启动时间。

· 可以使多个程序同时驻留在内存中，提高 CPU 的利用率。

· 可以运行重定位程序。即程序可以放于内存中的任何一处，而且可以在执行过程中移动。

· 写机器无关的代码。程序不必事先约定机器的配置情况。

· 减轻程序员分配和管理内存资源的负担。

· 可以进行共享——例如，如果两个进程运行同一个程序，它们应该可以共享程序代码的同一个副本。

这些目标都可以通过虚存实现[Denn 70]。从应用程序角度看，它看到大片可由其任意使用的主存，而系统可能只有很少的内存。这里需要明确地址空间与存储单元的区别。程序在自己的地址空间中访问代码和数据，这些地址必须被转换为对应的主存单元。当程序需要处理数据时，硬件和软件必须配合完成向主存加载数据和对每次访问的地址转换。

虚存系统并不是没有代价的。内存管理需要地址转换表和其他一些数据结构，留给程序的内存减少了。地址转换增加了每条指令的执行时间，而对于有额外内存操作的指令会更严重。当进程访问不在内存的页面时，系统发生失效。系统处理该失效，并将页面加载到内存中，这需要极耗时间的磁盘 I/O 操作。总之，内存管理活动占用了相当一部分 CPU 时间(在较忙的系统中大约占 10%)。而碎片又进一步减少了可用内存——例如，在基于页而的系统中，如果页面中只有一部分是有用的数据，剩下的内存就浪费了。所有这些因素都说明，一个既重视性能又重视功能的设计的重要性。

13.1.1　内存管理的石器时代

早期的 UNIX 实现(版本 7 或更早)运行在 PDP-11 上，它有一个只有 64KB 地址空间的 16

位体系结构，某些型号还支持独立的指令和数据空间。但既便如此，进程大小仍限制在 128KB 以下。这种限制造成各种用于用户程序和内核的软件覆盖技术的产生[Coll 91]。这种方法通过覆盖那些程序暂时不再使用的地址空间来复用内存，例如，一旦系统被加载并运行后，系统初始化代码就不再使用了，就可以把它释放掉，供程序的其他部分使用。这种覆盖技术需要应用程序开发人员的显式参与，他们必须清楚程序及运行它的机器的许多细节。由于覆盖技术依赖于物理内存的配置信息，使用覆盖技术的程序天生就是不可移植的。既便系统仅仅是增加了内存，程序员也需要改写程序。

早期 UNIX 的内存管理机制只有交换机制(图 13-1)。进程被整个加载到一片连续物理内存中。物理内存只能同时容纳很少几个进程。它们分时运行，如果又有一个进程要运行，就必须对换出去一个现有的进程。该进程被复制到磁盘上预定义好的交换区中，每个进程在其创建时都在该区上分配一片交换空间，保证进程有足够交换空间可用。

分页系统随着 1978 年 VAX-11/780 的推出而出现在 UNIX 系统中。VAX-11/780 有一个 32 位体系结构，4GB 地址空间，提供分页硬件支持[DEC 80]，3BSD 是第一个支持分页系统的 UNIX 版本[Baba 79，Baba 81]。在 20 世纪 80 年代中期，所有的 UNIX 系统都使用分页作为主要的内存管理机制，而交换只起辅助作用。

在分页系统中，内存和进程的地址空间都被分成固定大小的页。这些页面在需要时被换进或换出内存，内存的页通常称为页面(或物理页)。在任何时刻可以有若干个活动进程，而每个进程在物理内存中只有一部分页(图 13-2)，每个进程都认为自己是系统中的唯一一个程序。程序的地址是虚拟的，由硬件划分为页号和页内偏移。硬件和操作系统一起将程序地址中间的虚页号转换为物理页面号，并访问相应的内存单元。如果所需的页面不在内存中，就必须将其换入内存。对于纯分页系统来说，直到页面需要(被访问)时，才将其换入内存。大部分现代 UNIX 系统都进行一定程度的预分页，即换入那些马上可能使用的页面。

分页可以同交换一起或单独使用，它有如下优点：

· 程序尺寸受限于虚存。对于 32 位系统，最多可达 4GB。

· 运行程序时不必将其全部载入内存，启动时间快。

· 在任意时刻，每个程序在内存中只需少量页面，系统可以同时加载很多程序。

· 换入换出单个页面的代价比交换整个进程或内存段要小得多。

图 13-2　物理内存保存每个进程的若干页面

在谈论内存管理时不提及分段是不完全的。分段技术将进程地址空间分成若干段。程序中地址由一个段 ID 和段内偏移组成。每个段都独立的保护权限(读/写/执行)。段在物理中存中是连续的。每个段由一个描述子描述。该描述子包含段的物理基址，尺寸和保护权限。每次内存访问硬件都检查段边界，防止进程破坏邻近的段。程序的加载和交换都是以段为单位的，不再是整个程序。

分段也可以与分页结合起来[Bach 86]，构成一种灵活的内存管理机制，在这样的系统中，段不必在物理内存中连续存放，每个段有自己的地址映射图将段内偏移转换为物理内存单元。Intel 80x86②结构就提供这种模型。

程序员一般都认为进程地址空间由正文、数据和堆栈区组成，分段机制很适合这样看待进程。尽管许多 UNIX 明确定义了这 3 个区域，但它们不是硬件可识别的段，它们仅作为一种描述连续虚页的高级抽象表示。分段在主流的 UNIX 系统中并不流行，今后我们将不再讨论分段机制。

13.2　分　　页

分页系统负责分配和管理全体进程的地址空间。它对物理资源的使用进行优化，这当然包括主存和二级存储设备，以最小的代价实现需要的功能。本章连同第 14，15 章将对几种不

同的虚存体系结构进行探讨。我们首先来看看所有的分页系统中共同关注的基本问题。

13.2.1 功能需求

既然需要分页结构，我们就先来考虑一下它要有哪些功能。最主要的目的应该是让程序在虚拟地址空间中运行，并相对于进程透明地完成虚实地址转换。实现该目的的同时，必须做到对系统资源影响最小。几乎所有的其他需求都直接或间接地来源于这一目标。从这个角度出发，我们发展出一组详细的功能需求：

·地址空间管理　在 fork 时内核为进程分配地址空间，在进程 exit 时释放它们。如果进程调用了 exec 系统调用，内核释放原来的地址空间，为新程序分配一个新的地址空间。地址空间上的其他主要操作包括，改变数据区、堆栈区的大小，增加新的可访问内存区(如共享内存)。

·地址转换　对访问内存的每一条指令，MMU 都要将进程产生的虚地址转换为主存的物理地址。页面是分页系统中内存分配，保护和地址转换的基本单位。一个虚地址首先转换为虚页号和页内偏移。然后虚页号再通过某种地址转换映射图转换为物理页号。如果访问的页面不在物理内存中(没有驻留)，就会引起一个页面失效异常。内核的异常处理函数处理这个异常，将页面加载到内存中。

·物理内存管理　物理内存是内存管理子系统控制的最重要的资源，内核和用户进程都争夺内存资源，而且这些请求都必须快速响应。所有活动进程加在一起的尺寸通常远远大于物理内存，它(指物理内存)只能保留这些数据中的一个子集。系统仅将内存作为有用数据的缓存。内核对该缓存的使用进行优化，保证数据的一致性和并发性。

·内存保护　进程不应该以非授权方式访问或修改页面。内核保护它的数据和代码，以防用户进程修改它们。否则，用户程序就会偶然地(或恶意的)破坏了内核。出于安全性的考虑，内核也必须防止进程读取它的代码或数据。进程访问其他进程的页也是不允许的。进程的某些地址空间对进程本身也应该是受保护的。例如，进程的正文区通常是写保护的，防止进程偶然破坏其正文区。内核通过底层的硬件机制实现系统的内存保护机制。当内核发现存在对非法地址访问时，就向那个进程发送一个段违例信号(SIGSEGV)③。

·内存共享　UNIX 进程的特点及其交互方式很自然地要求共享它们地址空间中的某一部分。例如，运行同一程序的所有进程共享程序的同一份正文区。进程也可显式地要求与协同进程共享一段内存。标准库的正文区也可以用同样的方式共享。这些都是高层次共享的例子，还有一些低级共享的例子。低级共享可以共享个别的页面。比如，fork 后，父进程与子进程只要不修改数据区或堆栈区，它们就共享那些页面。

这些以及其他一些共享的方式大大提高了性能。它减少了对物理内存的竞争，去除了内存复制，以及为维护同一数据的多个副本而进行的磁盘 I/O。内存管理子系统决定支持哪种共享方式，以及如何实现它们。

·监视系统负荷　通常情况下，分页系统可以应付活动进程的请求。然而，有时系统会进入超负荷状态。当这种情况发生时，进程得不到足够的内存来容纳它的活动页面，不能继续运行。分页系统的负荷依赖于活动进程的数目及其尺寸，以及内存访问模式。操作系统监视分页系统，发现上述情况后，采取相应措施。这些措施包括停止启动新进程，不再继续某些进程的执行，等等。

·其他服务　内存管理系统还支持其他一些功能，如文件映射，动态链接库，以及执行存储在远程节点上的程序。

内存管理结构对整个系统性能有很大的影响，设计时尤为注重性能和可扩缩性。如果要求系统可以运行于不同类型的机器上，可移植性也是很重要的。最后，内存子系统应该对用户是透明的，用户在写代码时无需考虑低层的内存结构。

13.2.2 虚拟地址空间

进程的地址空间包括程序访问或引用的全部(虚)内存单元。任何时刻，地址空间连同进程的寄存器上下文就是进程的当前状态。进程调用 exec 启动新程序后，内核依据新程序的映像建立地址空间。分页结构中将地址空间分成许多固定尺寸的页面，程序的这些页面包含如下几种类型的信息：

- 正文
- 初始化数据
- 未初始化数据
- 修改的数据
- 栈
- 堆
- 共享内存
- 共享库

这些页面类型有不同的保护权限，初始化方法，以及进程间的共享方式，正文页通常是只读的，而数据，堆和栈页面是可读可写的。共享内存页面上的保护权限通常在其被创建时设置。

正文页面通常由运行同一程序的进程共享。进程将共享内存映射到自己的地址空间后，就可以和其他进程共享其中的页面了。共享库包含正文和数据两部分页面，访问库的所有进程共享所有的正文页。库的数据页是私有的，每个进程有自己的副本(在某些实现中，在数据页未修改前，也可以共享)。

13.2.3 页面初始访问

即使没有程序页面在内存中，进程照样可以开始运行一个程序。随着它不断访问那些不在内存中的页面，系统产生页面失效，内核为其分配空闲页面，用合适的数据进行初始化。

首次对页面访问的初始化方法不同于之后对页面访问的初始化方法，本节我们介绍首次访问。正文和初始化数据页面直接从执行文件中读取。未初始化数据页面置为 0，这样那些全局未初始化变量的初值自动置为 0。共享库页面从库文件中读取，u 区和内核栈在进程创建时通过复制父进程页面建立。

如果某个进程执行的程序已由另一个已存在的进程执行或刚刚执行完。此时，内存或交换区(见 13.2.4 节)等高速设备中尚有该程序的全部或部分页面。在这种情况下，系统可以避免从执行文件中再次访问页面。这个问题将在 13.4.4 节和 14.6 节中进一步讨论。

13.2.4 交换区

所有活动进程的全部尺寸一般远大于物理内存，因此内存只能为每一个进程保存某些页面。如果进程需要的页面不在内存内核就必须挪用另外一个页面，丢弃原来的内容。

理想情况下，我们应只替换那些不再需要的页面，比如属于终止进程的页面。通常这是不可能的，内核只能借用一个未来可能使用的页面。因此，内核必须将该页面的内容保存在二级存储设备上。UNIX 使用交换区保存临时页面，它一般由一个或多个磁盘分区组成。当系统初始配置时，保留一定量的未格式化磁盘用于交换区。这些空间不能由文件系统使用。

图 13-3 说明了一个页面是如何在物理内存和不同的二级存储设备间移动的,如果某个保存在交换区的页面再次被访问，内核的页面失效处理函数就从交换区中读出它们。为了这样做，必须维护某种形式的交换区映射图，记录所有被换出的页面在交换区上的位置。如果页面又需要被换出内存，仅当其内容与已保存的副本不同时才进行复制。仅当页面为已修改的时才保存它，也即页面自上次从交换区读出后是否被修改过。如果系统的硬件支持页表项上的已修改位，实现这一功能是很容易的。但如果没有硬件支持，内核就必须通过其他方式获取信息。13.5.3 节将讨论这一内容。

正文页不必保存到交换区中，它们可以从执行文件中获得。某些实现出于性能考虑也同

样换出正文页。交换区映射图一般都设计非常高效的数据结构，内核只需简单地索引一张驻留在内存中的表就可以查到页面在交换区上的位置。如果是定位执行文件的页面，内核就必须通过文件系统.文件系统访问 i 节点可能还要检查某些间接块(见 9.2.2 节)。这是比较慢的操作，特别是当读取间接块时还需要额卜的磁盘访问。然而，将正文页换出也需要额外的磁盘 I/O，这可能抵消了快速的换入操作带来的好处。大多数现代 UNIX 不换出正文页，直接从文件读回页面。

图 13-3　页面在土存中的移入移出

13.2.5　转换映射图

分页系统为实现虚存使用叫种不同类型的转换图。如图 13-4 所示：

图 13-4　地址转换

硬件地址转换　对每一条访问内存的指令，硬件都需要将程序的虚地址转换为物理内存单元。每种机器都提供某种硬件地址转换机制，以便操作系统不必参加到每次地址转换中。12.3 节将考察 3 种内存体系结构的例子。这些例子几乎都包括了快表(TLB)和页表。尽管硬件指定了它们的数据结构，仍须由操作系统来设置和维护它们。

硬件地址转换图是唯一需 MMU 了解的数据结构。本节中描述的其他映射图均只须由操作系统了解。

地址空间映射图　当硬件不能转换某个地址时，它产生一个页面失效。这可能是因为页面不在物理内存中或没有有效硬件转换图表项转换该地址。内核的失效处理函数针对这个失效在必要时装载页面，并加载一个有效的硬件转换图项。

硬件可识别的映射图可能无法提供进程地址空间的完整信息。例如，在 MIPS R3000 上，硬件只使用一个很小的 TLB，操作系统要维护额外一张映射图，它提供完整的地址空间信息。

物理内存映射图　内核经常需要查询反向映射，通过给定的物理页面获取其所属的进程以及虚页号。例如，当内核从内存中删除一个活动页面时，它需要清除所有关于该页面的映射。要这样做，就必须定位页面的页表项或 TLB 表项，否则，硬件可能仍然认为该页面在物理内存中。因此，内核需要维护一张物理内存映射图，随时跟踪物理页面中存储了什么数据。

后备存储映射图　当失效处理函数在物理内存中找不到页面时，它分配一个新页面，并用两种可能的方式对其进行初始化——全部置 0 或从二级存储设备中读取。对于后一种情况，页面可以从可执行文件，共享库目标文件，交换区上的副本中读取。这些对象就构成了进程页面的后备存储，内核需要维护一张映射图来定位后备存储上的页面。

13.2.6　页面替换策略

为了创建新页面，内校必须回收一些当前在内存中的页面。页面替换策略决定内核回收哪个页面[Bela 66]。理想的回收页面是僵页，即那些不再需要的页面(例如，属于终止进程的页面)。如果没有僵页(或者僵页不足)，内核可以选择局部或全局替换策略。局部策略只回收进程或与进程相关的进程组中的页面。如果某进程需要页面，内核替换其自身的页面。如果内核使用全局策略，它就从任一进程中挪用一个页面。这种选择依据全局选择标准，比如使用模式等。

为了确保某些进程有足够的贤源，需要使用局部策略。例如，系统管理员可能为重要的进程分配很多的页面。然而全局策略易于实现，更适合于通用分时系统。大部分 UNIX 变体都实现了全局替换策略，但为每个活动进程保留一个页面最小集。

对于全局替换策略来说，内核要选择合适的标准来决定哪些页面应该留在内存中。理想情况下，我们需要保留那些最近要使用的页面，我们称这些页面是进程的工作集。如果进程的页面访问行为预先知道的话，至少从理论上可以精确地确定工作集。但实际情况中我们对进程的访问模式知之甚少，只能依赖一般进程的实验研究来指导我们的实现。

这些研究表明进程通常具有“引用局域性”,即一个进程总是趋于将其引用的页面局域化

到一个很小的子集，而且这个子集变化很慢。例如，当执行一个函数时，所有指令都在包含那个函数的页面上。然后过一会，进程又转移到另一个函数，其工作集发生了变化。数据访问也很相似，数组上的循环操作以及在某个结构执行若干操作的函数都是体现访问局域性的例子。

根据常识可以推出，最近访问的页面很可能在最近再次访问。这样，对工作集的一个比较好的近似就是最近访问的页面集,这就是页面替换的最近最少使用策略(LRU)——丢弃那些很长时间内不访问的页面。文件访问也具有相似的局域性，这种 LRU 策略也同样适用于文件系统的块缓存。对于内存管理而言，由于一些实际限制必须对 LRU 策略进行一些修改，详见 13.5.3 节。

最后，内核还须决定何时释放活动页面。一种可能就是仅当进程真的需要页面时才选择页面回收，加载新页面。这种方法很低效，往往导致系统性能下降。更好的方案是维护一个空闲页面池，定期向池中增加一些页面。这样就可以将分页系统的负担均匀分布到整个时间段中了。

13.3 硬件需求

内存管理子系统依赖低层的硬件完成若干任务。这些任务是由一个位于 CPU 和主存之间的硬件部件完成的，该部件称为内存管理单元(MMU)。MMU 的结构对内核中的内存管理子系统的设计有着重大的影响。根据这一点，我们首先从抽象层上探讨 MMU 的功能，然后通过 3 个实例——Intel x86，IBM RS/6000 和 MIPS R3000 的研究来分析体系结构的特征是如何影响内核设计的。

MMU 的主要任务是完成虚地址转换。大多数系统使用页表，快表或两者兼备实现地址转换映射图。本节讲述页表，下一节是快表。在一般的系统上，内核地址空间对应一个页表，每个进程的用户地址空间对应一个或多个页表。页表是一个数组，每一项对应进程中的一个虚页，页表项(PTE)的索引对应其描述的虚页。例如，正文区页表中的 PTE 3 ，对应正文区中的虚页 3。

页表项大小通常为 32 位，分为若干域。这些域包括物理页面号，保护权限，有效位，修改位，以及可选的引用位。页表项的格式由硬件预先定义。除此之外，页表是位于内年中的非常简单的数据结构。MMU 仅使用那些加载到硬件页表寄存器中的活动表。通常，在单处理机中，有两个活动页表　　一个是内核的，另一个是当前运行的进程的。

MMU 将虚地址分为虚页号和页内偏移，然后在页表中定位该虚页项，抽取出物理页面号，再加上页内偏移计算出物理地址。

地址转换可能因以下 3 个原因失败：

·边界错误　地址不在进程有效地址范围内，没有虚页对应的页表项。

·有效性错误　页表项被标记为无效。这通常是说页面不在内存中，有几种情况，即使页面是有效的且在内存中，有效位同样会被置无效，13.5.3 节中将介绍。

·保护错误 页面不允许某些极限的访问(如，对只读页面写，用户过程访问内核页等)。

在这些情况下 MMU 发出一个异常，并将控制权较交给内核的处理函数④，这个异常称为页面失效。失效地址以及失效类型(有效性失效或保护失效——边界错误也导致有效性失效)一同传递给失效处理函数。处理函数完成页面的加载或利用信号(SIGSEGV)通知进程。

如果成功地处理了失效，进程(当它再次运行时)必须重新启动引起失效的指令。这需要硬件在失效前保存足以启动失效指令的正确信息。

每次页面被写入，硬件设置其 PTE 上的修改位。如果操作系统发现该位被置，在其重新利用页面时要将该页面保存在存储设备上。如果硬件支持 PTE 上的引用位，可以在每次访问页面时设置该位。这就可以使操作系统监视驻留页面的使用情况，重新利用那些看起来不再

使用的页面。

如果进程有一个非常大的地址空间，它的页表也会非常大。比如，如果-个进程的地址空间是 2GB，页面尺寸是 1KB，则页表有 200 万项，总共需要 8Mb 的内存保存页表。在内存中保存这么大的页表是不可能的，而且大部分地址空间可能根本不使用——一般的进程地址空间只包括几个区(正文、数据、栈等)，分散在地址空间不同的地方。我们需要一种更紧凑的地址空间描述方法。

该问题可以通过采用分段表或分页页表本身的方法解决。第一种方法在有显式分段支持的系统中很有效。进程的每个段有一个页表，刚好可以保存段的有效地址范围。第二种方法中页表本身被分页，这就是说用一个高级页表来映射低级页表。使用这样一种多级页表层次，只需为那些映射有效地址的低级页表分配页面。二级页表使用非常广泛，还有些体系结构支持三级页表，比如 SPARC 参考规范中 MMU[SPARC 91]。

页表将 MMU 与内核联系在一起，两者都可以访问使用和修改 PTE。以外，硬件还有一组指向页表基址的寄存器。MMU 负责使用 PTE 转换虚地址，检查进程中的有效位和保护权限，设置引用位和修改位等等，而内核则要建立页表，在 PTE 中填写正确的数据，设置指向页表的 MMU 寄存器等。这些寄存器通常在上下文切换时需要重新设置。

13.3.1 MMU 缓存

仅用页面还不能提供有效的地址转换。每条指令都需要访问若干次内存——一个用于转换程序计数器指定的虚地址，一个是读取指令。与此相似，还有每个内存操作数的两次防问。如果页表本身是分页的或是多级的，访问内存的次数会更多。由于每次内存访问至少需一个 CPU 周期，如此多的内存访问将使内存带宽趋于饱和，指令的执行时间增加到不可接受的地步。

这个问题有两个解决方案，第一个是增加高速缓存。许多机器都有分离的数据缓存和指令缓存，或者仅使用一个统一的缓存。从缓存中读取数据比从内存要快。许多机器中(尤其比较早的)是通过物理地址访问缓存。它完全由硬件管理，对软件是透明的。缓存访问发生在地址转换之后，所以这种方法的收效甚微。许多较新的体系结构，如惠普的 PA-RISC[Lee89]使用虚地址访问缓存，可以使缓存搜索与地址转换同时进行。这些方法大大提高了性能，但同时又带来缓存一致性问题，而这个问题必须由内核解决(见 15.13 节)。

第二种减少内存访问的方法是使用片上转换缓存，称为快表(TLB)。TLB 是一个关联缓存，保存最近使用的地址转换数据。快表项与页表项很相似，也包含地址转换信息和保护权限等，可能还有一些标记用来标识该地址属于哪个进程。缓存是关联的，也就是说查找不是基于索引的，而是基于内容的——在快表项中同时查找虚页号。

MMU 通常控制 TLB 的全部操作。当其在 TLB 中查找失败后，就查找软件地址映射图(例如页表)、并且加载一个快表项。在某些情况下也需要操作系统的协同。如果内核更改了某个页表项，该变化不会自动地更新 TLB 并标记其为无效，内核必须显式地清除所有快表项，这样，MMU 就会在页面下二次访问时从内存中重新加载快表项。比如，内核可能根据某个显式的用户请求(通过 mprotect 系统调用)标记页面为写保护的。这需要清除该页原有的快表项，否则，进程仍然可以写页面(因原映射允许写操作)。

硬件规定了内核以何种方式操作 TLB，或者提供显式的加载相清除 TLB 表项的指令，或者是某指令的副作用产生相同的功能。在某些系统中(如 MIPS R3000)，硬件仅支持 TLB，任何的页表或其他映射图均由内核单独维护。在这种系统中，操作系统处理每一个 TLB 失效，加载正确的表项。

尽管 MMU 必须具有相同的基本功能，但使用的方法却差别很大。MMU 的体系结构确定了虚页和物理页面的大小，可用的保护权限类型，地址转换表项的格式等。在硬件支持页表的机器中，MMU 还定义厂页表的层次，映射页表的寄存器等。它还定义了自己与内核间的分工，

以及内核在地址处理与地址转换缓存等上咋中所起的作用。根据这些，现在让我们看看 3 种不同的体系结构，着重看看它们是如何影响 UNIX 内存管理实现的。

13.3.2 Intel 80x86

Intel 80x86 是基于 SystemV 的 UNIX 的主要平台之一,它有 4GB 的地址空间(32 位地址)，页面大小为 4096 字节，同时支持分段和分页机制[intel 86]。图 13-5 给出了地址转换的若干步骤。每一个虚地址包括一个 16 位的段选择符和一个 32 位偏移。分段层将虚地址转换为 32 位的线性地址，然后再由分页层转换为物理地址。可以通过清除 CR0 寄存器的最高位关闭分页机制，此时线性地址就是物理地址。

图 13-5 Intel 80x86 上的地址转换

进程地址空间可使用 8192 个段。每个段由一个段描述符定义，其中包括基址，尺寸，保护权限。每个进程有一个自己的局部描述符表(LDT)，每一项对应一个段。还有一个全局范围的全局描述符表(GDT)，其中包含内核正文区、数据区、堆栈区，外加一些特殊的段，如每个进程的 LDT 等。当转换一个虚地址时，MMU 通过段选择符从 GDT 或 LDT 中(依选择符中的一位来判断)选择 j 正确的段描述符，检查偏移是否在段的范围内，加上段基址得到线性地址。

在 x86 上实现的 UNIX 仅在内存保护，内核陷入，上下文切换等情况中使用段[Robb87]，它对用户进程隐藏段的存在，进程看到一个平坦的地址空间。为达到这个目的，每个内核建立的段都使用基址 0，并且有足够大的尺寸，当然不包含高端的虚存，这部分(指虚存)用于内核正文和数据。正文区是只读的，而数据区则是可读写的，但两者都指向同一位置。每一个 LDT 还包括一些特殊的段——用于系统调用项的调用门段，以及在上下文交换时保存寄存器上下文的任务状态段(TSS)。

x86 使用二级页表(图 13-6)。4KB 的大小，意味着进程最多可以使用多达 100 多万个页面，x86 用许多小页表代替单独一个巨大的页表。每个页表的大小均为页面大小，有 1024 个 PTE。它映射 4MB 大小，在 4MB 边界上对齐的连续空间。固此，一个进程可以有 1024 个页表。但大多数进程具有稀疏的地址空间，仅使用少量页表。

图 13-6 Intel x86 上的地址转换

每一个进程有一个页目录，它的每个 PTE 映射一个页表。页目录也是页面大小，有 1024 个 PTE，每一项 PTE 对应一个页表。页目录就是一级页表，而页表本身就是二级页表，在后面的叙述中将不再使用一、二级页表，一律使用页目录和页表。

控制寄存器 CR3 在高 20 位中保存当前页目录的物理页号，因此也称其为 PDBR(页目录基址寄存器)。80x86 上的虚地址可以分为 3 部分，高 10 位是 DIR 域，是页目录的索引，其与 CR3 中的页面号一起计算得到页目录表项的物理地址，通过它可以访问相应的页表。中间 10 位是 PAGE 域，保存相对于该内存区起始位置的虚页号，通过它索引页表得到需要的 PTE，该 PTE 又包含所需页面的物理页面号。将物理页面号与低 12 位页内偏移组合得到物理地址。

每一页表项包含物理页面号，保护域，有效位，引用位和修改位(图 13-7)。保护域有两位——一位指定是只读(置 0)或呈可读可写(置 1)，另一位说明页面是用户页面(置 0)还是内核页面(置 1)。当进程运行于内核态时，所有的页面都是可读可写的(忽略写保护)。在用户态时，不管读写位是如何设置的，内核页面都不能访问，读写位仅对用户页面有效。由于页目录表项和页表项都有保护域，必须两个域同时允许才能访问页面。

x86 支持引用位，这大大简化了设计，减轻了内核监视页面的负担。在上下文切换时，需要重新设置 CR3 寄存器，使其指向新进程的页目录。

图 13-7 Intel x86 页表项

内核不能直接操作 x86 结构中的 TLB。但无论何时 PDBR 被重写后，整个 TLB 都被自动刷新。可以通过显式的移动指令或上下文切换间接更改 PDBR。当 UNIX 内核清除某个页表项时(例如，在复用一个页时)，都需要刷新 TLB。

x86 支持四级优先级或保护环，而 UNIX 仅使用 2 个。内核运行于最内层的保护环，特权级最高。该层允许执行特权指令(比如修改 MMU 寄存器)，可以访问所有的段和所有的页面(用户的和内核的)。用户代码运行于最外层，特权级最小。该层只能运行非特权指令，只能访问进程私有段中的页面。调用门段用于用户进行系统调用，该门的内容由内核填写。调用时系统转移至最内层，并将控制转交给调用门中指定的入口。

13.3.3 IBM RS/6000

IBM RS/6000[Bako 90]是一部精简指令集(RISC)机器，其上运行 IBM 的基于 SystemV 的操作乐统 AIX。它的内存体系有两个特点——一个是它使用单一的，平坦的系统地址空间。另一个是使用反向页表进行地址转换。惠普的 PA-RISC[Lee 89]也使用这种机制。

利用虚地址进行索引的常规页表有如下问题。其尺寸与虚地址空间成正比增长，对于某些进程页表会变得极大。许多现代系统采用 64 位地址获取更大的地址空间。这样页表也变得异常庞大，以致于在物理内存中无法容纳。一种方法是，就像 x86 这类机器那样，采用多层页表。尽管如此，对大进程来说，内核必须支持对页表本身的分页机制。

反向页表给出了另一种机制,通过它可以更有效地限定维护地址转换信息所需的物理 内存。该表中每一项对应物理内存中的一页，完成物理页面号到虚地址的映射。由于物理内存远远大于系统中所有进程的地址空间，反向页表非常紧凑。然而 MMU 需要转换虚地址，反向页表却不能直接完成这样的转换。因此系统中需维护虚实地址转换所需的其他数据结构，下段将开始介绍这些结构。

RS/6000 有两种类型的虚地址。其中一种是单一的，平坦的系统虚地址空间，采用 52 位地址，空间尺寸是 2^52，约 4×10^15 字节。每个进程使用 22 位地址，它们的地址空间都映射到系统空间中的一部分，如图 13-8 所示。虚实页面均为 4096 字节，与磁盘块的缺省大小相同。32 位的虚地址分为 3 部分——4 位的段标识，16 位的页索引和 12 位的页内偏移。整个地址空间有 16 个段，每个段的大小都是 256MB。

图 13-8 RS/6000 地址空间

RS/6000 有 16 个段寄存器，由当前进程的段描述符加载。每个段都有某种特殊的用途。段 l 保存用户程序正文。段 2 是进程的私有数据，保存用户数据，堆和栈，以及内核栈和 u 区(见 14.2 节)。段 3 到段 10 是共享段，用于共享内存和文件映射(见 14.2 节)。段 13 保存共享代码，如从共享内存加载的代码。其余的段供内核使用，由所有进程共享，但只能在内核态中访问。段 0 保存内核正文。段 11，12 保存内存管理数据结构。段 14 保存内核数据结构。段 15 用于 I/0 地址空间操作。

图 13-9 所示是从进程空间到系统虚地址空间的转换。段标识指定段寄存器，大小为 32 位，它包含一个 24 位的段索引，组成系统虚地址空间的高 24 位，其与进程虚地址中的 16 位虚页索引形成系统地址空间的虚页号，虚页号需要进一步转换得到物理页号。

如前面所描述的那样，RS/6000 没有直接的虚实地址映射图。它维护一称为页面表(PFT)的反向页表，每一页对应一个物理页面，系统采用哈希技术完成虚地址的转换。如图 13-10 所示。有一个称为哈希定位表(HAT)的数据结构，系统利用 HAT 将一个虚地址换算为一个哈希值，就是一个指向 PFT 表项链表的指针。每一个 PFT 表项有如下信息：

- 其映射的虚页号。
- 指向哈希链中下一项的指针。
- 失效位，引用位、修改位。
- 保护和加锁信息。

RS/6000 使用 HAT 定位哈希链，再遍历哈希链查找到所需的虚页号。PFT 中每一项的索引就是物理页面号，共 20 位，其与虚地址中的 12 位页内偏移组成物理地址。

地址转换过程很慢，代价很高，应该在每次内存访问中避免这一过程。RS/6000 有两个

图 13-10　RS/6000 地址转换——部分 2

分离的 TLB——一个是有 32 个表项的指令 TLB，另一个是有 128 个表项的数据 TLB。在正常操作中,大多数地址转换由这两个缓存完成。当有 TLB 失效时,才进行哈希搜索。此外,RS/5000 还有分离的数据和指令缓存。数据缓存大小为 32KB 或 64KB，指令缓存大小为 8KB 或 32KB，这取决于系统的型号[Chak 94]。这些缓存都是用虚地址访问的，因此当缓存命中时就无须地址转换。虚址缓存需要内核参与某些一致性维护，这些问题将在 15.13 节中讨论。

13.3.4　MIPS R3000

MIPS R3000 是一个 RISC 系统,是 SVR4 UNIX 和 DEC 的 ULTRIX(基于 4.2BSD 的 UNIX 系统)的平台。它有一个非常规的 MMU 结构[Kane 88]，该结构中没有硬件支持的页表。硬件仅使用片上的 TLB 完成地址转换。

该结构对内存管理分上以及内核与硬什的接口有着深远影响。比如在 Intel x86 结构中，TLB 项的结构对内核来说是不可见的，硬件允许的仅有的操作是清空整个 TLB 和由某个虚地址对应的 TLB 表项。相反，在 MIPS 的结构中，TLB 的结构和内容对内核是公开的，允许内核对指定的表项进行读、修改以及加载等操作。

如图 13-11 所示，虚地址空间分为四个段。kuseg 覆盖前 2GB，是用户地址空间。其他三个段只能在内核态下访问，kseg0 和 ksegl 分别映射 512MB 的物理内存，无需 TLB 映射，其中 kseg0 使用数据/指令缓存，而 kseg1 不使用。最后 1GB 是 kseg2，映射可缓存的内核段。kseg2 中的地址可以映射到物理内存中的任何一处。

图 13-11　MIPS R3000 虚地址空间

图 13-12 给出了 MMU 寄存器以及 TLB 表项的格式。MIPS 的页面尺寸固定使用 4KB。因此虚地址分为 20 位虚页号和 12 位偏移。TLB 包含 64 个表项，每一项大小为 64 位。En-tryHi 和 EntryLo 寄存器分别与 TLB 表项的高 32 位和低 32 位相同，这两个寄存器用于读写 TLB 表项。VPN(虚页号)和 PFN(实页号)域实现虚实页号的转换。PID 域作为一个标记使用，将 TLB 表项与进程联系起来。这个 PID 域大小为 6 位，可以从 0～63，但不同于传统的进程号。所有活动 TLB 表项的进程都赋予 0～63 中的一个值，称为 tlbpid。内核将当前进程的 tlbpid 写入 entryhi 的 PID 域，硬件将其与 TLB 表项中对应域进行比较，不匹配的不进行地址转换。这咩 TLB 中就可以同时存放不同进程的具有相同虚页号的 TLB 表项了。

如果设置了 N(不进行缓存)位，页面访问不再遍历数据或指令缓存。G(全局)位说明是否忽略 TLB 表项中的 PID 域。如果 V(有效)位清 0，TLB 表项无效。如 D(可写)位清 0，该表项是写保护的。注意，这里没有引用位和修改位。

在转换 kuseg 和 kseg2 的地址时，虚页号同时与 TLB 所有表项进行比较。如果匹配且 G 位清 0，就比较表项的 PID 和保存在 entryhi 牛的当前进程 tlbpid。如果相等(或者 G 位置 1)并且 V 位置 1，PFN 域就是有效的物理页面号。否则，就产生一个 TLBmiss 异常，对于写操作，D 位必须是置 1 的，否则会产生一个 TLBmod 异常。

图 13-12　MIPS R3000 的地址转换

硬件至此不再提供其他设施(比如页表)。内核必须处理这些异常。它搜索自己的地址映射图，或者查到一个有效的地址转换或者向进程发信号。对于前者，内核须加载一个有效的 TLB 表项，重新启动失效指令。硬件不对内核实现地址映射有更多的限制，内核可以采用或不采用页表方式，页表项是什么样子的也无所谓。实际使用中，MIPS 上的 UNIX 实现采用页表，以便维持基本的内存管理设计。EntryLo 寄存器的格式很自然就是 PTE 的格式，其低 8 位硬件不用，软件可以任意使用。

由于缺少引用位相修改位，内核需要增加额外的功能支持它们，内核必须知道哪些页面是修改过的，以便在复用前保留其内容。这一功能可以通过对未修改过的页面进行写保护实现(清除 TLB 中的 D 位)。这样会在对这些页面写操作时产生一个 TLBmod 异常。异常处理函数

设置 TLB 中的 D 位，并且设置软件 PTE 中的某位，标记页面是修改过的。页面的引用信息也可以间接收集，见 13.5.3 节。

在这种结构中，每次 TLB 失效均要由软件处理，从而产生大量的页面失效。对页面修改信息以及页面引用信息的采集使页面失效更多。这一缺陷可以通过简单的存储结构补偿。在这样的结构下，TLB 命中时可以进行非常快的地址转换。此外，不断增快的 CPU 速度也降低了页面失效的代价。最后，内核的静态正文和数据存放于不被映射的 kseg0 中，无须地址转换，这加速了内核代码的执行。同时也减弱了对 TLB 的竞争，只由用户地址和某些动态分配的内核数据结构使用 TLB。

13.4 4.3BSD 实例研究

到目前为止我们已介绍了分页的基本概念，以及硬件特征是如何影响设计的。为了更清晰地理解这些问题，我们使用 4.3BSD 的内存管理作为一个实例研究。第 -个支持虚存的 UNIX 系统是 3BSD。随着其版本的不断发行，它的内存结构也逐渐变化。4.3BSD 是最后一个基于这种内存模型的伯克利发行。4.4BSD 采用一种全新的基于 Mach 系统的内存结构，15.8 节将介绍该系统。[Leff 89]对 4.3BSD 的内存管理进行全面的讲解。本章中，我们先总结其重要特征，并评估其优点与缺点，后面章节中更复杂的方案都是基于这些分析设计的。

虽然 BSD 系统的目标平台是 VAX-11，它已经成功地移植到其他几个平台上。内核算法受底层硬件特征的影响，尤其是处理页表及地址转换缓存的底层函数。移植 BSD 的内存管理子系统并不十分容易，硬件相关性遍布整个系统的各个部分。几个基于 BSD 的实现不得不在软件上仿真 VAX 的内存结构，包括 VAX 的地址空间结构以及页表项的格式等等。由于 VAX 已经过时，我们不准备对它的内存结构进行详细介绍，仅将它的某些重要特征作为 BSD 的一部分进行介绍。

4.3BSD 中有一些重要数据结构——内存映射图描述物理内存，页表描述虚存，磁盘映射图描述交换区，还有资源映射图，用于管理诸如页表，交换区等资源的分配。最后，某些重要的信息保存在每个进程的 proc 结构和 u 区中。

13.4.1 物理内存

物理内存可视作一个从 0～n 的线性数组，其中 n 是系统中的内存总量。系统将内存分成若干逻辑页面，页面的大小取决于机器的结构。如图 13-13 所示，整个内存分为 3 段，在低端的是非换页内存池，保存内核正文，静态分配或启动时分配的部分内核数据。由于页面失效可能在不很合适的地方将内核中的进程阻塞住,大多数 UNIX 实现需要将内核页面标记为不可对换的。在最高端的内存保存系统崩溃时产生的错误信息。在这两者之间是占用了大部分内存的换页内存池，它包含了所有用户进程的页面，以及内核动态分配的页面。后者尽管是换页内存池的一部分，但它被标记为不可对换的，不参加对换。

图 13-13 物理内存布局

物理页称为页面。页面保存进程页的内容。保存在页面中的页随时可能被另一个页代替，为此我们需要维护每个页面内容的信息，这是通过内存映射图实现的。⑤内存映射图是元素为 struct cmap 的数组，每一项对应换页内存池中的一个页面。内存映射图的每一项包含如下信息：

·名字 保存在页面中的页的名字或标识。进程页用其属主进程号，类型(数据、堆栈)以及虚页号描述。正文页可能由若干进程共享，它们的属主是程序的正文数据结构。内存映射图的每一项都保存一个进程表或正文表的索引。如图 13-14 所示，名字(type，owner，virtual page number>可以供内核完成反向地址转换，定位页面对应的 PTE。

·空闲表 链接空闲页面的空闲表的前、后两个指针。该链按近似最近最少使用策略排序，用于内存分配例程分配或释放物理内存。

·正文页缓存　一般页名字仅在属主进程存活时有意义。由于数据页或堆栈页在进程退出不再使用，页名字这样定义是没有问题的。但对于正文页来说，有可能另一进程很快就要运行同一个程序。如果这种情况发生了而且仍有正文页驻留在内存中，此时可以不必从磁盘中重新读入这些页面，直接复用这些内存中的页面就可以了。为了在其属主终止后仍能标识这些页面，内存映射图表项中保存正文页的磁盘位置 (设备后和块号)，这些页面还通过哈希算法组织为一组哈希队列(基于设备号和块号)，方便快速访问。磁盘位置既可标识可执行文件中的页面，也可以标识交换设备上的页面。

·同步　有一组标记用于同步对页面的访问。页面在读出或写入磁盘时被锁住。

图 13-14　物理地址到虚地址的地址转换

13.4.2　地址空间

BSD 虚存使用 VAX-11 的地址空间模型。VAX-11 是一台 32 位机，页面大小为 512 字节，其 4GB 的地址空间分为相等的 4 个区。第一个 1GB 是 P0(程序)区，保存进程的正文区和数据区。然后是 P1(控制)区，保存用户栈，u 区，以及内核栈。接下来是 S0(系统)区，保存内核正文和数据。第 4 个区是保留区，当前的 VAX 硬件不支持。在这种模式下，每个区可以随意增长而不会在地址空间中留下空白。

VAX 硬件支持页表，并直接使用它们转换虚地址。页表用于几种功能。内核用它们描述进程的地址空间(proc 结构有一个简要描述，包含页表的位置和大小)，同时也保存页面如何初始化的信息(见 13.4.3 小节)。为此内核利用 PTE 的硬件不用位实现这些功能。

有一个系统页表映射内核正文和数据，每一个进程有两个页表映射 P0(正文和数据)区和 P1(用户栈，u 区等等)区。系统页表在物理内存中是连续的。每个用户进程页表在系统虚存中是连续的，通过系统页表 Userptmap 区中若干连续 PTE 映射，内核中用一组<base，size>偶对组成的资源映射图管理 Userptmap 的空闲区。该资源图的分配采用最先匹配算法，在重负荷情况下，资源映射图变得非常细碎，进程可能找不到连续的 PTE 映射页表。在这种情况下，内核调用 Swapper 进程，将整个进程换出，释放出其在 Userptmap 中的空间。

页表可以共享吗?特别是当两个进程运行同一个程序时，它们可以共享正文区的页表吗?通常足可以的，许多 UNIX 变体都允许这种共享。然而 BSD UNIX 用这种方法会有些小问题。每个进程都有一个 P0 区的页表，它在系统虚空间中必须是连续的，由 Userptmap 中连续的系统 PTE 描述。由于数据区是不共享的，只有部分的 P0 页表是共享的。又因为每个进程都有私有的 Userptmap 表项，正文区页表的 PTE 须指向同一组页面。这就是说数据区页表必须起始于一个新页面,并由 Userptmap 的新 PTE 描述。这样一来数据区就必须对齐于 64KB 边界上(页面 512 字节，PTE 32 位，占 4 个字节，页面上可有 512/4＝128 个 PTE，每个 PTE 对应 512 页面，共 128X512＝64KB)。

这一限制会给内核带来不兼容的、用户可见的变化。为了避免这样，BSD 要求每个进程有自己的正文页表。如果多个进程共享同一正文区，它们的正义页表项需保持同步。例如，如果一个进程加载了某个页面，相应的变化要传播到所有共享该正文区进程相应的 PTE 上。图 13-15 所示是内核如何定位映射同一正文区的所有页表。

图 13-15　正文页的多个映射

13.4.3　页面在哪里

在任意时刻，进程的某个页而可能是如下状态之一：

·驻留　页在实存中，页表项包含其物理页面号。

·请求填入　页尚未被进程引用，必须在初次访问时载入内存，有两种请求填入类型：

·正文填入　正文和初始化数据页在首次访问时从可执行文件中加载。

·填 0　未初始化数据，堆，栈页在初次访问时填写 0。

·换出 这些页已被载入内存，后来因空间紧张而被换出，这些页可以从交换区中恢复。

内核必须为不在内存中的页面维护足够的信息，这样在需要时可以恢复它们。对于换出的页面，必须保留其在交换设备的位置。对于填 0 页面，内核仅仅识别它们就可以了。对于正文填入页面，必须确定其在文件系统中的位置。可以通过文件系统例程读入 i 节点的磁盘块数组。然而，由于这样会频繁地访问其他磁盘块(间接块)定位页面，十分低效。

一个更好的方法是在程序启动时在内存管理数据结构中保存这些转换信息。这样只需对 i 节点的磁盘块数组和间接块做一次扫描就可以定位正文和初始化数据厂。可以使用另一个映射非驻留页面的表格实现这一功能。SVR3 UNIX 中的磁盘块描述符表就起这样的作用。但这种方法因需要一个与页表一样大的附加页表，带来了很大的内存开销。

4.3BSD 的方法依赖于这样的条件：在有效位置 0 时，除硬件除保护位和有效位外，不检查其他域。由于所有非驻留页面的有效位均被置 0，这样就可以利用其他域保存这些页的这些信息。图 13-16(a)是 VAX-11 硬件定义的页表项格式，4.3BSD 使用硬件不用的位 25 定义请求填人类型。

对于普通页表项(图 13-16(b))，请求填入位置 0。当该位置 1 时，说明该页是请求填入页(有效位一定已被置 0)。当页表项是请求填入表项时，它有不同的域定义(图 13-16(c))。此时页表项不保存页面号和修改位，而是保存文件系统块号，并利用其中一位定义页面是正文填入(位置 1)还是填 0(位置 0)的。对于一个正文填入页，可从进程的正文结构中获取设备号。

图 13-16　4.3BSD 页面项格式

对于换出页面有不同的处理方式。这些页面的 PTE 将有效位，请求填入位置 o，页面号也置 0。内核维护另一个独立的交换映射图，在交换设备上定位这些页面。13.4.4 节将对此进行详细讲述。

13.4.4　交换区

交换空间的存在有两个原因。一是在需要换出整个进程时，进程的所有页面都可以保存到磁盘上。二是当单独某个页面需要换出时，我们也需将其保存到磁盘上。交换空间使用一个或多个逻辑盘或分区。这些分区的格式均为原始的，也就说它上面没有文件系统。4BSD 可以将交换分区分布在多个磁盘上，这样可以提高分页的性能。这些分区在逻辑上被组织成一个统一的交换区，这有助于平衡所有文换区上的负荷。页面在交换空间的位置通过-个伪设备号表示，该设备号表尔逻辑交换分区及其上的偏移。然后在内部转换为相应的物理分区及其上的偏移。

严格地说，只需为那些需要换出的页面分配交换空间。这种激进的策略会造成内存泄漏(memory overeommit)，某个进程会在某个时刻用光交换空间。如果在程序正常执行时发生这种情况，就会导致进程异常挂起或中止。为了避免这种情况，4.3BSD 使用一种相对保守的交换区分配策略。当启动-个进程时，内核为其所需的数据和堆栈区分配交换区。为了在必要时可以扩展，交换区按大块方式分配。如果内存区增长超过了已分配的交换区，就需分配更多的交换空间。这样就可以保证交换区创建或扩展时仅在这些严格定义的点上发生交换区耗尽的情况。

正文页(和未修改的数据页)不必换出，它们可以从可执行文件中读取。但在 BSD 实现中，这会导致一些问题。由于文件块号保存在请求填入 PTE 中，一旦页面被加载到内存后，页面号就覆盖了该信息。从文件读取页面时需要重新计算它的位置，可能还要访问一次或多次间接块。这种操作十分耗时，是该绝对避免的，于是这些页面也同样换到交换区上。如果多个进程运行同一个程序，只需换出正文区的一个副本。正文区的大小固定。它的交换区可以分配为连续的磁盘块，可以在一次磁盘操作中读取其附近的若干相邻页面。

内核在每个区的 dmap 结构中记录交换空间分配情况。分配给内存区的第一块长度为 dmmin(通常为 16KB)，其后的块大小均是前一次的 2 倍，直到达到 dmmax(通常从 0.5MB～

2MB)，以后的块大小均为 dmmax。dmap 结构是一个固定大小的数组(见图 13-17)，每一项都包含块在交换设备上的起始地址。由于 dmap 的大小固定，用户进程的数据和堆栈区有一个最大的上界。u 区保存数据和堆栈区的映射图。正文区大小也是固定的，它有不同的映射方式。无边共享同一正文区的进程有多少，只须在交换区上保存一个副本。交换区映射图是正文结构的一部分，正文在交换区上都分配为若干个 dmtext 大小(通常为 512KB)的块，最后一块可能仅使用部分空间。

13.5 4.3BSD 内存管理操作

现在来看看 4.3BSD 内存管理中的一些重要操作——创建进程，页面失效处理，页面替换以及对换。

13.5.1 创建进程

调用系统调用 fork 创建一个新(子)进程，并复制其父进程的地址空间。这包括如下内存管理操作：

·交换空间　首先是为子进程的数据和堆栈区分配交换空间。此时分配给子进程的交换空间与其父进程相同。如果交换分配失败，fork 返回一个错误信息。

·页表　内核须在 Userptmap 中分配一片连续的 PTE。如果此操作失败，系统就需对换另一个进程，以便在 Userptmap 中腾出空间。系统 PTE 只从空闲内存列表中分配图 13-17 在 dmap 中记录交换空间

·u 区　系统为进程分配一个 u 区，并复制其父进程的 u 区完成对子进程 u 区的初始化。为了从内核中直接对于进程的 u 区进行操作，内核将其映射到一片称为 Forkmap 的系统空间上直接操作。

·正文区　父子进程共享正文区，系统将子进程加到共享该正文区的进程表中。正文区的页表项从父进程拷贝，

·数据和堆栈　数据和堆栈必须一个页面一个页面地处理。对于那些仍是请求填入的页面，只需复制 PTE。其余的页面需在内存中分配足够的物理页面，逐一从父进程中复制它们。如果某个父进程的页面已被换出，就必须先从交换区中读出，然后再复制它们。相应地，子进程的 PTE 指向这些新的副本。所有新生成的页面都标记为已修改，系统在回收这些页面前会将它们保存到交换区中。

fork 操作代价极高，这很大程度上是因为最后一步的复制过程。考虑到大多数进程在 fork 后马上退出或调用 exec 运行一个新进程，丢弃原来的地址空间，复制整个数据和堆栈很浪费。

有两种途径来减轻这一代价。第一种称为 copy-on-write，System V UNIX 采用了这种方法。在这种方法中，父子进程引用数据和堆栈的同一个副本，其保护权限被设置为只读。如果父进程或子进程修改某个页面，系统就产生一个保护失效。失效处理函数识别出这种情况，创建一个新的页面副本。在这种方法中，只需父进程或子进程复制那些修改的页面，降低了进程创建的代价。

实现 copy-on-write 需要为每个页面维护一个引用计数，这就是 BSD UNIX 没有采用这种方法的原因。BSD 采用 fork 的替代系统调用 vfork(虚 fork)提供另一种解决方案。

vfork 用于在 fork 之后马上使用 exit 或 exec 的情况。vfork 不复制地址空间，父进程将自己的地址空间传递给子进程，并在于进程调用 execs 和 exit 前睡眠。到那时，内核唤醒父进程，它重新控制地址空间。为创建子进程所需的唯一资源是 u 区和 proc 结构。通过复制父进程的页表控制器简单地完成地址空间的传递,不需要复制页表,仅需修改映射 u 区的 PTE。

vfork 代价极小，而且比 copy-on-write 快得多。它的缺点是子进程可以修改父进程地址空间的内容或大小，要求程序员住意正确地使用 vfork。

13.5.2 页面失效处理

有两种页面失效类型——有效性失效和保护失效。有效性失效发生在没有页面对应的PTE 或 PTE 被标为无效时。保护失效发生在对保护页面的非法访问中。如果用户试图访问无效且有保护权限的页面时，只产生一个保护失效(如果读/写访问都不允许，当作有效性失效处理没有意义)。

当这两种失效发生时，系统为重新启动指令保存足够的信息，然后将控制转移给内核的失效处理例程。对于边界错误且不是因堆栈溢出而造成的，进程通常会被一个信号中断。对于堆栈溢出，内核调用一个例程自动增长堆栈。保护错误时内核也会向进程发送一个信号。实现 copy-on-write 的系统必须增加对保护失效的处理，对 copy-on-write 的页面生成一个新的可写的副本。对其他情况，调用 pagein()例程处理失效。

系统传递给 pagein()失效虚地址，再由它获得 PTE。如果页面在内存中(PTE 不是请求填入的且页面号不为 0)，内核可进一步获得页面的 cmap 表项。所有这些合在一起就构成了页面的状态信息，控制 pagein()的行为。图 13-18 给出了 pagein()的基本算法，共有 7 种情况：

1.模拟引用位而将 PTE 标记为无效，见 13.5.3 节。这时页面驻留在内存中且 emap 项未标记为空闲，pagein()简单地设置有效位然后返回。

2.页面存在且在空闲列表中。除了 cmap 表项被标记为空间外，与第 1 种情况相似。pagein()重置有效位，并将 cmap 表项从空闲列表中删除。

3.对于一个正文页，另一个进程可能已开始读取该页面。这种情况发生在有两个进程共事同一个正文区且在同一个页面上发生失效时。第二个进程发现页面号非 0，但内存映射表表项被标记为锁住和数据传送中，pagein()设置 wanted 标志并将第二个进程阻塞住，让其在该页面所在的代码结构上睡眠。当第一个进程读出页面并对其解锁时，它唤醒第二个进程。由于第二个进程在被唤醒时可能不能马上执行，此时不能假定页面仍在内存中，它必须重新对页面进行搜索。

4.正文页在内存中，但 PTE 没有这些页面的页面号，这种情况发生在一个进程结束了一段时间以后。可以以<device，block number>偶对为键值，搜索相应的哈希队列定位这些页面。如果发现了，就从空闲队列中删除并复用这些页面。

对于其余的情况，页面不在内存中。在定位该页后，pagein()首先从空闲页面链表中分配页面，然后按如下情况读取页面：

5.页面在交换设备上，请求填入位为 0，页号为 0，并且第 4 种情况不成立。通过交换映射表定位页面，然后从交换设备上读取页面。

6.对于填 0 页，分配页面且填 0。

7.页是正文填入页，并且不在哈希队列中(第 4 种情况)，从可执行文件读取页面。这里数据传送是直接从可执行文件到内存，不经过缓冲。在磁盘上的数据比磁盘块缓存的内容旧时，这可能造成一致性问题。为此内核要检查缓存，如果发现该页面就刷新缓存，然后再读取页面。这种方法需两次磁盘操作，很低效，但由于历史原因而保留下来。如果在缓存中发现页面，直接从缓存中复制页面会更好。

图 13-18 pagein()算法

在第 5 和第 6 种情况中，新页面都标记为已修改，在复用它们时将保留在交换区上。

13.5.3 空闲页面链表

内核频繁地向内存中装入新页面，一旦内存变满，就需要替换当前内存中的页面。4.3BSD 使用全局替换策略选择要清除的页面。

很明显，最佳的候选重用页面呈不再使用的页面，比如终止进程的数据和堆栈页面。一般应该尽量使用这种不再使用的页面。如果这样的页面不存在，局域性原则要求按照 LRU 的顺序回收页面。相对于那些一段时间内不再使用的页面来说，那些最近使用的页面很可能马

上还要使用。

在这样的标准下，仅在我们需要页面时才查找一个合适的候选页面显然是不可行的。更好的方法是维护一个可用页面列表，在需要时从表中索取页面。不断地向表中加入可用页面，且将最佳候选页面排列在表头。理想情况下，将所有垃圾页排序在空闲表的头部，其后按 LRU 顺序排列有用页面。BSD 内核就维护这样一个空闲队列。系统参数 minfree 和 maxfree 规定了该表的最小和最大尺寸，它的当前尺寸由变量 freemem 记录。

是否严格按 LRU 顺序是个实际操作问题。它需要在每次对页面访问时都要重排表，而这可能在每次访问用户地址空间时都要进行一次，这会造成不能容忍的高消耗。4.3BSD 采用了一种有效的折衷方案，它用非最近使用策略[Baba 81]替代最近最少使用策略。如果最近没有访问过某页面，页面就可以替换。

这种策略可以这样实现。间隔一段时间对所有页面扫描两遍。第一遍清除相应页面 PTE 的引用位，第二遍检查这些引用位。如果它仍是清除的，页面就可以释放，因为这表明其在两遍扫描这段时间中页面没有被访问过。该算法称为双表针时钟算法。cmap 可以看做是一个圈(首尾相接)，有两个指针(表针)维护固定的间距(Cmap 表项数目)(见图 13-19)，两个表针一起前进。前面的表针清除引用位，后面的表针检查引用位，如果引用位仍为 0，则页面在前表针将其引用位置 0 后一直未访问过，可以释放。如果页面已修改过，必须在释放前先将其保存在交换区上。

图 13-19　双表针时钟

对于像 VAX-11 和 MIPS R3000 这样的某些体系结构，硬件不支持引用位。此时必须通过软件来摸拟引用位。这时，前表针清除 PTE 的有效位。页面被访问时将产生失效，失效处理函数识别出这种情况(上一节中的第 1 种情况)，简单地设置有效位。这样，如果后表针发现有效位仍然无效，就意味着页面在其间未被访问后，可以释放。为追踪引用信息会产生许多额外的页面失效，会带来一些负担。

一个称为 pagedaemon 的独立进程(进程 2)负责页而替换。这样将页面写到交换区不会阻塞那些无关的进程。此外，因 pagedaemon 需要写出其他进程的页面，它首先将那些页面映射到自己的地址空间，然后不通过缓存而是利用一组特殊的交换缓冲区头结构直接写回交换区。写操作是异步执行的，pagedaemon 可以同时继续检查其他页面。当写操作结束时，pagedaemon 将这些页面加入一个未修改页面表，然后调用 cleanup()例程从该表移至空闲内存表中。

13.5.4　交换

尽管分页系统在大部分时间内运转良好，但在重负荷下就不行了。这主要是由于一个称为颠簸的问题造成的，此时没有足够的内存容纳活动进程的工作集。在系统中有太多的活动进程或它们的内存访问模式太随机的情况(就是说它们的工作集太大)下就会出现这种现象，这种问题的表现就是页面失效率激增。当页画被换入时，它们替换了其他活动进程工作集的页面，而它们会使问题逐步升级，这种情况愈演愈烈，直到系统占用大部分时间来处理页面失效，造成进程运行缓慢。

这个问题可以通过减少活动进程的数日，控制系统负载来解决。被标记为“失活”的进程，将不再被调度和运行。必要时还要向交换区复制数据来尽可能多地释放出内存。这种操作就是所谓 的换出整个进程。在系统负荷减轻后，进程可以被再次换入。

一个称为 swapper 的特殊进程监视系统负荷，在必要时换入换出整个进程，在系统初始化时创建一个 PID 为 0 的进程，它最后调用 swapper 的基本函数 sched()，这样进程 0 就成了 swapper，它大部分的时间都处于睡眠状态，但被周期性地唤醒检查系统状态，并在需要时采取相应的措施。swapper 在如下情况换出进程：

·Userptmap 碎片化 如果 Userptmap 太满或碎片太多，某个进程就不可能在 Userptmap

中分配连续的 PTE 映射它的页表。这种情况发生在进行 fork，exec 或扩展内存区时。此时，swapper 将换出某个已有的进程，释放出更多的 Userptmap 空间。

·内存短缺　变量 freemem 保存空闲链表中的簇数。当 freemem 在一段时间中始终比期望值小的时候，说明系统超负荷了，此时系统调用 swapper。

·失活进程 如果某个进程很长一段时间(大于 20 秒)处于失活状态。它很可能仍然会保持这样的状态，于是将其换出。例如，用户可能没有退出就回家了，留下一个失活的 shell 进程。尽管它的驻留页面最终会被对换到交换区上，换出整个进程可释放其他重要资源，比如 Userptmap 项。

swapper 如何选择要换出的进程呢?理想的换出进程是那些已睡了超过 20 秒的进程，如果没有，swapper 挑选 4 个最大的进程，并换出其中驻留内存时间最长的一个。在需要更多的内存时，其他 3 个进程也被陆续换出。

在换出一个进程时，swapper 完成如下任务：

1.为 u 区，内核栈，页表分配交换空间。

2，去掉进程对其正文区的引用。如果没有进程共享此正文区，正文区也要换出。

3.在交换区上保存驻留在内存中的数据页，堆栈页，然后是页表，最后是 u 区和内核栈。

4.在 Userptmap 中释放映射进程页表的系统 PTE。

5.在 proc 结构中记录 u 区在交换区上的位置。

当一个或多个进程换出后，swapper 周期性地检查是否能够将它们换入，这取决于 Userptmap 中是否有足够的空间。如果有若干个进程已被换出,swapper 基于它们的尺寸,nice 值(见 5.4.1 节)，它们被换出时间，以及它们睡眠的时间等为它们计算一个换入优先级。

换入过程基本上是换出的逆过程。进程重新引用正文区，在 Userptmap 中分配 PTE 映射页表，为 u 区，内核栈和页表分配物理内存，从交换区中读同这些信息，交换区上的这些空间随后释放，之后，进程被标记为就绪，加入调度队列。随着进程的运行，数据和堆栈页面也逐渐被换入。

13.6 分　析

BSD 的内存管理设计使用少量原语实现了强大的功能。唯一的硬件需求就是支持分页(BSD 没有使用分段)。但是，BSD 有几个不容忽视的重要缺陷：

·不支持执行远程程序(通过网络)。这是因为 BSD 的文件系统不支持远程文件。如果文件系统支持这种功能，在内存子系统中加入这种扩充是很简单的，

·不支持除了共读正文区外的内存共享。特别是没有与 System V 中等价的共享内存设施。

·vfork 不是一个真正的 fork 替换接口。缺少 copy-on-write 机制严重影响了那些非常依赖于 fork 的进程的性能。特别是 daemon 和其他几个应用，它们在每次接收到一个请求时都 fork 一个子进程。

·每个进程必须有自己的页表来共享正文区。这不仅浪费了空间而且还需维护页表的同步，将某个进程在 PTE 上的修改传播到其他共享同一正文的进程。

·不支持内存映射文件，14.2 节将详细讨论内存映射文件的细节。

·不支持共享库。

·在调试某个已有多个进程运行的程序时，存在一些问题。如果调试器在程序中设置了断点，它将修改相应的正文页。该改动对运行该程序的进程来说都是可见的，这会给它们带来不可预期的结果。为了避免这种情况，系统不允许在共享的代码中加断点，不允许新进程运行正在调试程序。这种方案显然不能令人满意。

·在 BSD 实现中，系统在进程换出其地址空间中的第一个页面之前预留交换区。这种策略保证进程仅当其增长(或者 fork，exec)而不是执行中的某个时刻用光交换区。这种较为保守

的方法需要系统有较大的分配空间。从另一方面考虑，就是系统交换区限制了可以运行的程序的最大尺寸。

·不支持使用远程节点上的交换区，诸如无盘操作之类工具需要这种功能。

·该设计在很大程度上受 VAX-11 影响，并针对其结构进行了优化。此设计对于许多移植 UNIX 的其他类型机器是不适用的。此外，这种机器相关性遍布于系统各处，使移植需付出很大的努力。

代码不是模块化的,增加功能,改变某个部分或策略比较困难。例如,在无效(请求填入)PTE 中保存文件系统块号，这使得地址转换与页面获取任务很难分离。

尽管有这些缺点，4.3BSD 的设计还是为现代内存体系建立了坚实的基础。在后面的章节中，将介绍 SVR4，4.4BSD，以及 Mach 等系统。这些体系都保留了许多 BSD 的方法，但它们为了提供更多的功能以及解决 BSD 方法的局限性，修改了低层的设计。

4.3BSD 体系对 20 世纪 80 年代的系统是很适用的。那时的系统一般都有较慢的 CPU 和少量的内存，但却有相对大容量的磁盘。因此算法尽量减少内存使用，增加 I/O 负担。进入 20 世纪 90 年代，一般的桌面系统都有较大的内存利较快的 CPU，但相对而言磁盘较小，许多用户文件都存放在某个文件服务器上。4.3BSD 的内存管理模型不适合于这样的系统，4.4BSD 采用了基于 Mach 的崭新的内存体系，这将在 15.8 节中介绍。

13.7 练　习

1.在 13.2.1 节所罗列的目标中，哪一个适合于仅使用交换作为内存管理机制的系统?

2.相对于分段，分页系统有哪些优点?

3.为什么 UNIX 系统使用预分页?它有哪些缺点?

4.在交换区中复制正文页有哪些优缺点?

5.假定一个可执行程序在远程节点上.在执行前，先将它复制到本地交换区是否更好?

6.硬件与操作系统协同完成地址转换。它们是如何划分任务的?仔细阅读 13.3 节中所述的三种体系结构，并找出答案。

7，全局替换策略与局域替换策略相比?有哪些优缺点?

8.程序员可以采取哪些步骤减小应用的工作集。

9.反向页表的优点是什么?

10.为什么 MIPS 3000 会产生大量的页面失效?其结构中的哪些优点可以弥补处理这些失效的代价?

11.假如一个 4.3BSD 的进程在这样一个页上失效，它不在内存中，而且是受保护的（不允许所要的访问类型)，失效处理函数先检查哪种情况?处理函数怎样处理?

12.为什么内存映射图仅管理换页内存池?

13.页的名字指什么?一个页仅有一个名字吗?4.3BSD 页名字空间有什么不同?

14.4.3BSD 系统最少要有多少交换空间?有一个很大的的交换区有哪些优点?

15.哪些因素限制了进程可以有的最大虚空间?为什么进程必须减少其虚空间的浪费?

16.交换区分布在多个磁盘上更好吗?为什么或为什么不。

17.为什么纯 LRU 策略不适用于页面替换?

18.早期 BSD 发行[Baba 81，Leff 89]使用单表针时针算法。它在第一遍扫描中清除引用位，然后在第二遍扫描中换出那些引用位仍是清除的页面。为什么这个算法没有双表针时针算法好?

① 例如，1995 年，一部典型的桌面系统备育 75MHz 的处理器和 70ns 的内存访问时间，这相当于 5.25 个 CPU 周期。
② 本书使用 80x86(或更简单，x86)指代 Intel 80386，80486 和奔腾体系结构的通用特性。
③ 在某些情况下内核发送信号 SIGBUS(总线错误)。
④ 某些体系结构不支持保护性失效。在这样的系统中，内核必须通过去映射页面强制产中有效性失效，然后根据页面是否在内存中以及访问类型来采取相应的措施。
⑤ 事实上，帧中的每一簇都有一个 cmap 项。簇为由若干(固定)数目物理页面构成的逻辑页面提供-种标识方法。它通过增加子若干操作的粒度相减少了像内核映射图之类的数据结构而提高了性能。