第5章 内存管理

要执行一个进程，首先必须把进程的结构加载到物理内存空间中。内存管理就是围绕如何为一个进程分配物理内存空间这一核心问题展开的。进程是资源分配和管理的基本单位，因此内存管理也是以进程为单位的。从系统角度来看，内存管理本质上是一个最优化问题，不仅要考虑单个进程的内存需求，还要考虑整个系统对有限内存资源的需求，使得系统的一个或多个目标函数达到最优。

本章内容和基本要求如下。

（1）明确内存管理的基本需求，即重定位、共享、隔离及存储器扩充的需求，了解早期操作系统内存管理的方法和存在的不足。

（2）重点掌握虚拟内存的概念、分页式管理的基本原理、虚拟地址到物理地址的转换方法、缺页中断的处理流程，了解地址转换过程的硬件加速实现。

（3）重点掌握分页管理的读取策略和置换策略。理解分页管理是如何支持进程可重定位、保护、共享及存储器扩充的。

（4）掌握分段式内存管理的基本原理，理解段的动态链接和段共享的方法。

总之，本章介绍的内存管理与前面介绍的进程管理是操作系统极为重要的内容。在学习时需要加强与进程概念的联系，特别需要注意，当进程发生切换时与进程相关联的内存空间是如何随之发生转换的。只有把片段的知识联系起来，才能形成对操作系统工作过程的完整理解，最终达到深刻认识的目的。

5.1 内存管理的需求

5.1.1 内存管理的4个基本要求

为了有效地为一个进程分配物理内存空间，内存管理必须满足4个基本要求，即**重定位**、**共享**、**隔离**及**存储器扩充**。

（1）**重定位**需求。一个程序在不同的运行实例中，操作系统为它分配的物理内存空间可以不同，甚至在一个程序运行期间，它所占据的物理内存空间也可以上下浮动。之所以提出该需求是基于以下两个原因：第一，程序员在编写程序时通常无法确定，而且也没有必要确定程序所在的物理内存空间；第二，当一个进程被换出、下一次又被换入时，如果

要求必须换入先前的内存区域，那么这将是一个很大的限制。为此，要求进程可以重定位到内存的不同区域。

(2) **共享**需求。共享需求有两方面含义：第一，在多道程序设计中，内存中将会驻留多个进程，这些进程共享同一物理内存，为此需要对内存空间进行合理有效的划分，使得每个进程独占一定的内存区域，而且这些区域不相互冲突；第二，允许多个进程访问同一内存区域。例如，当多个进程需要调用 C 语言库函数时，如果每个进程都拥有 C 语言库的一个副本，那么这将浪费宝贵的内存空间；如果让 C 语言库的一个副本为多个进程所共享，那么就会节约有限的内存资源。再如，合作完成同一任务的多个进程可能需要访问相同的数据结构(如信号量、信箱、共享内存等)，这时也要求这些数据结构所在的内存区域被多个进程所共享。

(3) **隔离**需求。隔离需求与共享需求是一个问题的两个方面。由于多个进程驻留在内存中，因此必须保护一个进程的内存空间不受其他进程有意或无意的干扰。一个进程不能未经授权访问另一个进程的内存单元。处理器在执行时必须终止这样的指令，或产生相应的异常。

可重定位需求增加了隔离的难度。由于进程在内存中的位置是可以变动的，因而在编译时不可能通过检查绝对地址来进行保护，并且大多数程序设计语言允许在运行时进行地址解析(例如计算数组下标或数据结构中的指针等)，因此必须在**运行时**检查进程产生的所有内存访问，以确保它们只能访问分配的内存空间。

注意，内存保护必须由处理器(硬件)来实现，而不是由操作系统(软件)来实现。这是因为操作系统不能预测程序产生的所有内存访问；即使可以预测，提前审查每个进程中可能存在的非法内存访问也是非常耗时的。因此，只能在指令访问内存时(例如，存取数据或跳转时)来判断这个内存访问是否非法。这一点必须由处理器硬件来保证。

(4) **存储器扩充**。充分使用有限的内存空间，运行更多的进程，这些进程的内存需求总量可能已经超过了存储器所能提供的存储容量。这种扩充不是通过增加物理内存容量来实现的，而是通过内存管理算法来实现的，是虚拟的扩充。该需求要求进程的结构不能一次全部载入内存，而是边运行边加载(on-the-fly)。当物理内存不足时，需要采用一定的交换(swap)策略，将部分进程换出以便为其他进程腾出足够的内存空间。

5.1.2 地址定位

所谓**地址定位**(或称为**地址绑定**，address binding)指为了执行一个程序，必须确定程序指令和数据所在物理内存地址的过程。在程序还没有加载之前，程序以文件的形式存在于磁盘空间中，程序中的指令和数据使用相对地址(或逻辑地址)来编址；当准备运行程序时，程序被加载到内存中，这时需要定位程序指令和数据的物理内存地址，即在物理内存中确定指令和数据的位置。因此，地址定位过程就是逻辑地址到物理地址的变换过程。

在程序生命周期的时间轴上，我们将程序地址定位发生的时态总结为图 5-1。

在程序设计时进行地址定位，要求程序员在写程序时就能将指令和数据的物理地址

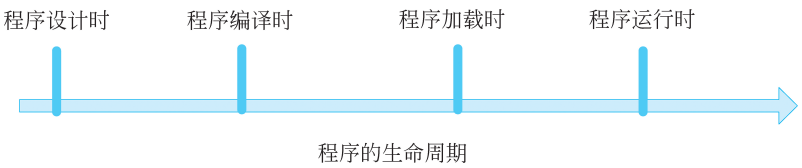


图 5-1　程序地址定位的时态

确定下来；在程序编译时进行地址定位，是将地址定位的时间推迟到程序编译时，使用编译器把指令和数据的物理地址确定下来。显然，这两种方式需要在编写程序时或编译时预知程序执行时的内存环境，这对于大多数计算机系统来说几乎是不可能的。

高级语言程序大多在程序加载时或程序运行时进行地址定位，但这并不意味着设计时和编译时地址定位没有意义；相反，在很多专用计算机系统（如简单嵌入式系统）或操作系统的底层程序中，通常需要在设计时或编译时将程序和数据安排在内存的特定位置，以适应或利用特定的计算机硬件结构。

1. 静态地址定位

在程序加载时进行地址定位，也称为**静态地址定位**，指由加载器（loader）在加载程序过程中将程序指令和数据的物理地址确定下来，如图 5-2 所示。

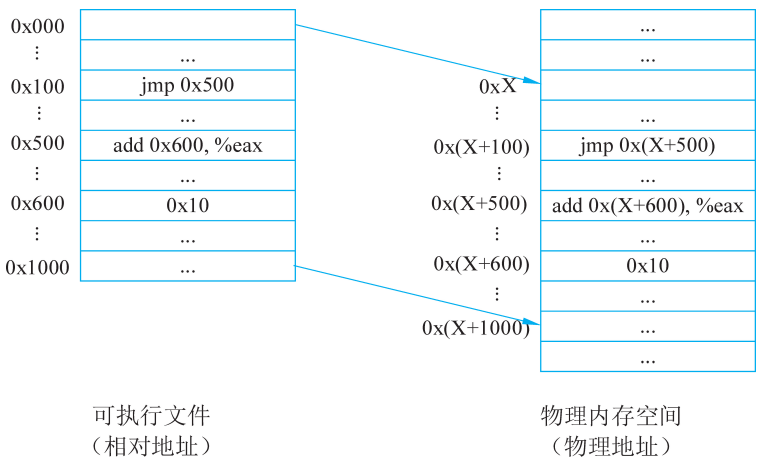


图 5-2　静态地址定位

静态地址定位把可执行文件中相对地址 0x000～0x1000 的一段程序映射到从 0xX 到 0x(X＋1000)的一段物理地址。程序中指令和数据的逻辑地址（用 LA 表示）及指令所引用的逻辑地址都在加载时被定位到物理地址（用 PA 表示）。地址定位的算法为

$$PA = LA + X$$

其中 X 是加载的起始内存地址。

静态地址定位的优点是容易实现，无须硬件支持，它只要求程序本身是可重定位的，

即对那些要修改的地址部分具有某种标识。早期的操作系统大多采用这种方法进行地址定位。其主要缺点如下。

(1) 程序经地址定位之后,在运行过程中就不能再移动,因而不能重新分配内存,不利于内存的有效利用。

(2) 程序在内存空间中只能连续分配,不能离散分布在内存的不同区域。

(3) 不利于多个进程共享内存中的同一程序或数据。

2. 动态地址重定位

在程序运行时进行地址定位,也称为**动态地址重定位**,指把地址定位从程序加载时推迟到程序运行时。在程序加载时,仍然保持程序的相对地址不变,只有在访问一条指令或数据时,才计算它们的物理地址,如图 5-3 所示。

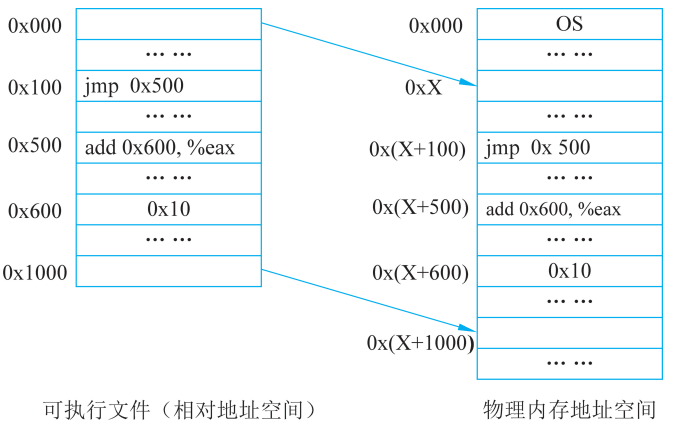


图 5-3 动态地址重定位

动态地址重定位使用硬件机构 MMU(内存管理单元)来实现,如图 5-4 所示。基址寄存器记录程序在内存中的起始地址,界限寄存器记录程序的最大逻辑地址(即程序的字节数)。当程序被加载入内存或当该进程被换入时,必须设置这两个寄存器。

在程序指令执行过程中会遇到逻辑地址,这些逻辑地址可能来自程序计数器 PC、跳转指令(jmp)或调用指令(call)中的指令地址,以及 load 指令和 store 指令中的数据地址等。逻辑地址被映射为物理地址的方法是:首先将逻辑地址与界限寄存器中的值相比较,判断逻辑地址是否越界,如果越界,则 MMU 产生一个地址错误异常,操作系统必须以某种方式对该异常作出响应;如果不越界,则把逻辑地址与基址寄存器的内容相加产生一个物理地址。简单地说,运行时地址定位的算法为

$$PA = LA + (\text{基址寄存器}) \quad \text{并且} \quad LA < (\text{限界寄存器})$$

其中"(基址寄存器)"和"(限界寄存器)"分别表示基址寄存器和限界寄存器中的内容。

动态地址重定位具有如下优点。

(1) 程序在加载后,起始地址可以上下浮动,有利于实现进程地址空间的换入和换出

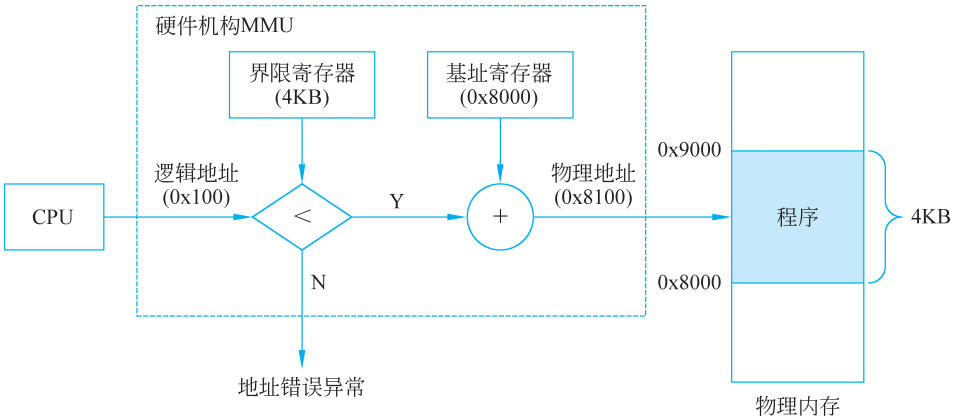


图 5-4　动态地址重定位的实现

策略，也有利于内存的充分利用。

（2）程序不必连续存放在内存中，可以分散在内存中的不同区域，只须增加几对基址-界限寄存器，每对寄存器对应一个区域。

（3）有利于实现多个进程共享同一程序。

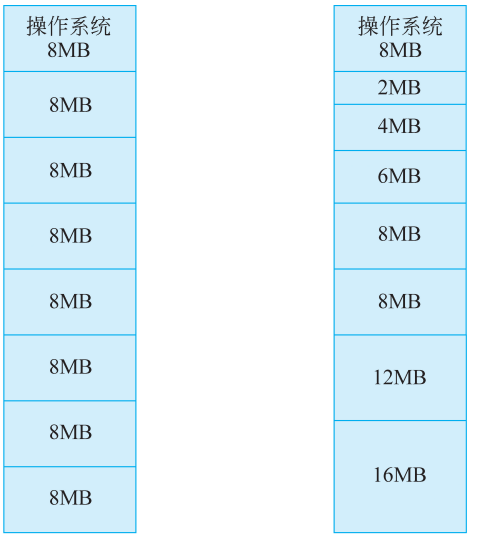
需要注意的是，动态地址重定位发生在运行时，因此不可能由操作系统来完成地址定位（这是因为当用户进程正在占用 CPU 运行时，操作系统是不可能运行的），只能由 CPU 的硬件机构 MMU 来完成地址变换。由于在运行时使用附加的硬件进行地址映射，因此处理器指令循环的一部分周期花费在了地址映射上，在一定程度上影响了程序的执行效率。

5.2　早期操作系统的内存管理

5.2.1　固定分区管理

分区内存管理将内存划分为若干个区域，每个区域只能被一个进程所独占。为进程分配内存时，只要有一个大小合适的分区，就可以分配给它；如果所有分区都已经被分配，那么需要按照某种调度策略换出一个进程，为新进程让出空间。具体哪个进程被换出或换入内存是由中程调度（见第 3 章）来决定的。一个进程被换出内存，它的状态就由活跃态变为挂起态；被换入内存时，它的状态就由挂起态变为活跃态。一般情况下，总是将处于阻塞态的进程换出。

按照分区大小是否可变，可以把分区管理分为**固定分区**和**可变分区**。固定分区又分为两种：每个分区大小相等和分区大小不等。图 5-5 显示了一个 64MB 内存的固定分区方法。操作系统占据了内存的某些固定部分，其余分区可供多个进程使用。



(a) 大小相等的分区　　(b) 大小不等的分区

图 5-5　固定分区

对于大小相等的固定分区，当为一个进程分配内存时，只要进程所需要的内存小于或等于分区大小，就可以选择任意一个分区分配给进程。对于大小不相等的固定分区，如果有多个分区都可以满足进程的内存需求，那么需要选择一个合适的分区来分配。选择的方法主要有以下 3 种。

(1) **最小适配**。在所有满足进程内存需求的分区中选择一个最小的分区来分配，即 $\min\{x \mid x \geqslant R\}$，其中 x 是分区的大小，R 是进程的内存需求大小。

(2) **首次适配**。即按照某种顺序，选择找到的第一个满足进程内存需求的分区分配给该进程。

(3) **最大适配**。在所有满足进程内存需求的分区中选择一个最大的分区来分配，即 $\max\{x \mid x \geqslant R\}$。

固定分区方法的优点是分区的大小和数目固定，因此内存管理算法相对简单，只需要很小的操作系统软件和处理开销。但是它也存在以下 3 个缺点。

(1) 分区的数目是固定的，限制了系统中活跃进程的数目。

(2) 由于分区的大小是事先设置的，因而小进程不能有效地利用分区空间，容易产生分区内部空间碎片(fragmentation)。

(3) 程序大小超过最大分区时，不能放到一个分区中。这种情况下，程序员必须使用覆盖技术(overlaying)设计程序，使得任何时候只有一部分程序驻留内存中，而且不影响程序的运行。

5.2.2　覆盖技术

覆盖技术和后面将要介绍的请求分页技术(paging)是为了解决进程所需要的内存大于计算机系统所能提供的内存而提出的两种解决方案。它们的基本思想都是利用了程序局部性原理,将进程的部分结构先载入内存,让进程运行起来。当进程在执行过程中用到某个模块时,才把该模块加载进内存。这两种方法都实现了存储器扩充的目的。

覆盖技术对于程序员是不透明的,必须通过程序员编写代码来实现。程序员在编写程序时,必须手工将程序分成若干块,然后编写一个小的辅助代码来管理这些模块何时驻留在内存、何时被替换掉。这个小的辅助代码就是所谓的**覆盖管理器**(overlay manager)。

例如,一个程序有主模块 main,它分别会调用模块 A 和模块 B,但是 A 和 B 之间不会相互调用。假定这 3 个模块的大小分别是 1024 字节、512 字节和 256 字节,理论上它们需要占用 1792 字节的内存。由于模块 A 和 B 相互不调用,因此它们可以相互覆盖,共享同一块内存区域,使得实际内存占用减小到 1536 字节,如图 5-6 所示。

当模块 main 调用模块 A 时,覆盖管理器保证将模块 A 从文件中读入内存;当模块 main 调用模块 B 时,则覆盖管理器将模块 B 从文件中读入内存,由于这时模块 A 不会被使用,因此模块 B 可以载入原来模块 A 所占用的内存空间。

事实上,程序往往不止两个模块,而模块之间的调用关系也比上面的例子复杂得多。在多个模块的情况下,程序员需要手工将模块按照它们之间的调用依赖关系组织成树状结构图。在这个树状结构图中,从任何一个模块到树的根模块(即 main 模块)的路径叫作**调用路径**。当一个模块被调用时,整个调用路径上的模块都必须在内存中,这一点由程序员来保证。图 5-7 是一个复杂覆盖的例子。其中 main 调用模块 A 和 B,A 又调用模块 C 和 D,B 调用模块 E 和 F。

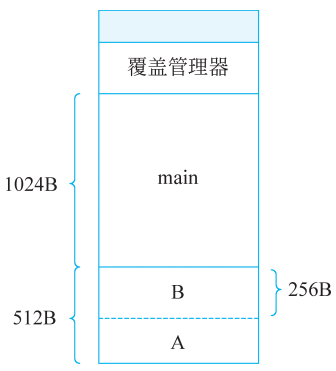


图 5-6　模块 A 和 B 相互覆盖

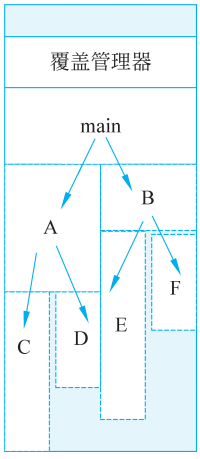


图 5-7　覆盖技术

值得注意的是，覆盖管理器必须保证满足以下两个要求。

（1）如果一个模块在内存中，那么从该模块开始的整个调用路径上的模块都必须在内存中。例如当模块 C 被调用时，那么 C、A、main 都必须被加载到内存中，否则当这个模块调用结束时，就不能正确地返回到调用模块。

（2）分支树上的模块之间不能存在相互调用。如果分支树上的模块之间存在相互调用，那么就会破坏树状调用依赖关系图。也就是说，当程序运行时，就需要将更多的模块加载到内存中，极端情况下，需要将所有模块都加载到内存中，这就失去了覆盖技术的意义。

5.2.3 可变分区管理

可变分区管理的分区长度和分区数目都是不固定的。当进程被载入内存时，系统会给它分配一块和它所需内存大小完全相等的内存空间。图 5-8 是一个可变分区的例子。开始时，64MB 的内存中只有操作系统，被装入的前 3 个进程从操作系统结束处开始，分别占据了它们所需要的空间大小。假定第 4 个进程需要 8MB 内存，这时剩下的 4MB 内存不够分配，只能等待前 3 个进程中的一个退出或者被换出内存。如果内存中的进程都是阻塞的，那么就可以换出一个进程腾出空间。假定进程 2 被换出，那么进程 4 就可以被载入内存，但是会产生 6MB 的内存碎片。在接下来的运行中，进程 1 退出，其内存空间被释放，这时进程 2 可以被换入内存，状态从挂起变为活跃。可以看到，这时的内存布局产生了 3 个内存碎片。可以想象，当内存中的进程数目很多时，经过较长运行时间之后，内存中的碎片变得很小、很多，使得它们难以再被分配。

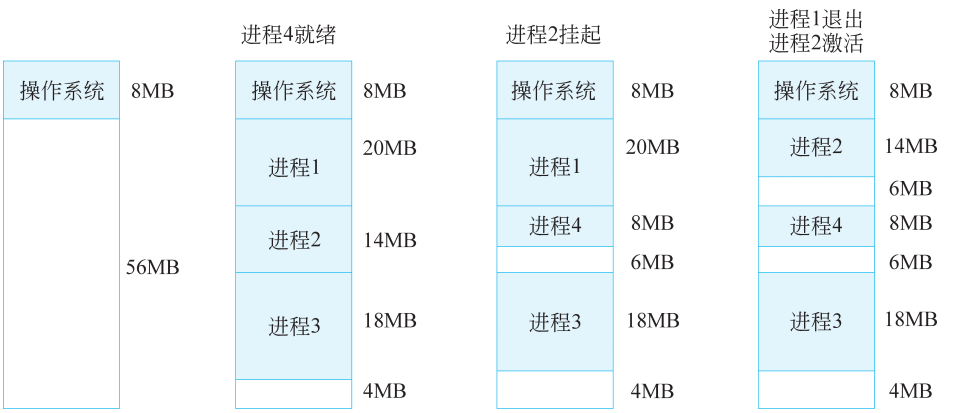


图 5-8 可变分区的例子

克服内存碎片的一个办法是将进程占据的内存上下浮动，使得若干细小、零散的碎片被整合成一整块大的内存，以便用于其他进程的内存分配，但是这样做需要两个机制的支持：一是动态地址重定位，二是内存复制。在进程的生命周期内，其内存空间要上下浮动，如果采用静态地址定位，当程序和数据部分浮动之后，内存引用就会出现错位。另外，

内存复制浪费处理器时间，效率不高。

下面对分区内存管理进行总结。分区管理具有下面几个特点。

(1) 分区管理能够实现多个进程共享同一内存空间，而且便于实现进程的保护。另外，分区管理相对比较简单，容易实现。

(2) 一个进程的地址空间要么全部交换到内存空间，要么全部交换出内存空间。使用分区管理无法实现真正意义上的存储器扩充。尽管可以通过把某些进程换出内存的方式腾出部分内存空间供其他进程使用，但是由于需要将整个进程的地址空间都加载到内存中，因此内存中活跃进程的数目仍然非常有限。

(3) 一个进程需要被映射到连续的内存区域。无论是静态还是动态地址定位，都以连续内存分配为前提。如果进程的不同部分被映射到离散的内存区域，那么逻辑地址变换和进程保护就将变得非常复杂。

5.2.4　伙伴系统

固定分区方案限制了活动进程的数目，而且如果可用分区的大小与进程大小完全不匹配，则内存空间的利用率非常低。可变分区的维护较复杂，并且整理内存碎片需要额外开销。**伙伴系统**(buddy system)是二者的一种折中方案。

在伙伴系统中，可用内存块的大小为 2^K 字节，$L \leqslant K \leqslant U$，其中，$2^L$ 表示最小内存块的大小，2^U 表示最大内存块的大小。通常 2^U 是可供分配的整个内存的大小。

开始时，可用于分配的整个空间被看作一个大小为 2^U 的块。如果请求的大小 s 满足 $2^{U-1} < s \leqslant 2^U$，则分配整个空间。否则该块被分成两个大小相等的伙伴，大小均为 2^{U-1}。如果 $2^{U-2} < s \leqslant 2^{U-1}$，则分配两个伙伴中的任何一个；否则，其中的一个伙伴又被分成大小相等的两半。这个过程一直继续下去，直到产生大于或等于 s 的最小块，并分配给该请求。

通常使用一个页面的大小(如 4KB)作为最小内存块的大小，伙伴内存块的大小为页面的 2^n 倍。例如，如果页面大小为 4KB，那么划分出的伙伴为 4KB($1=2^0$ 个页面)，8KB($2=2^1$ 个页面)，16KB($4=2^2$ 个页面)，32KB($8=2^3$ 个页面)，64KB($16=2^4$ 个页面)、……

通常使用空闲链表数组 free_area 实现伙伴系统，该数组包含 11 个元素，如图 5-9 所示。数组第 1 个元素 free_area[0]保存 4KB 空闲内存块(即 1 个页面)的链表头指针，数组第 2 个元素 free_area[1]保存 8KB 空闲内存块(即 2 个页面)的链表指针，以此类推，数组第 11 个元素 free_area[10]保存 4MB 空闲内存块(即 1024 个页面)的链表指针。

假如一个进程的内存请求是 15KB，那么合适的内存大小应当为 16KB，因此首先在数组元素 free_area[2]指示的空闲链表中去找。如果链表不为空，则直接从链表头取出空闲块进行分配；如果链表为空，则依次查找存储更大块的链表。假如能够在 free_area[3]指示的链表中找到一个 32KB 的空闲块，则把该块分为两个 16KB 大小的伙伴块，其中一个分配给该进程，另一个则加入 free_area[2]指示的链表中。

释放过程与分配过程相反。首先找到被释放的块的伙伴块，如果伙伴块处于非空闲

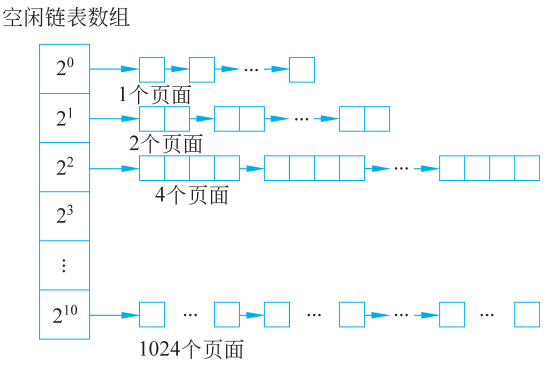


图 5-9 伙伴系统的空闲链表数组

态，则将被释放的块直接插入对应大小的空闲链表中，即完成释放；如果伙伴块处于空闲态，即在空闲链表中，则将它们进行合并，将合并后的块当成一个完整的块释放，并重复该过程。

在释放过程中需要注意，只有伙伴块才可以合并，因此需要快速找到一个块的伙伴块。可以观察到，互为伙伴的两个块的物理地址仅有一位有所不同，且该位由块的大小决定。例如，块 A、B 互为伙伴，且大小为 8KB。如果 A 的地址为 0x0000，那么 B 的地址一定为 0x2000，这两个地址只有第 13 位（从第 0 位开始计）不同。利用这一特点，可以快速找到一个块的伙伴块。

从上面的描述可以看出，伙伴系统的工作过程很适合用二叉树来进行描述。

【例 5-1】 设内存大小为 1MB，初始时未分配。第 1 个进程 A 需要 100KB，第 2 个进程 B 需要 256KB，第 3 个进程 C 需要 500KB。这些进程按照 A、B、C 的顺序依次进入并退出系统。伙伴系统管理内存的过程如下。

（1）将 1MB 内存折半分成两个伙伴，直到产生一个 128KB 大小的伙伴，分配给进程 A。这时伙伴系统的状态如图 5-10(a)所示。

（2）为进程 B 分配 256KB 大小。这时由于恰好存在一个大小为 256KB 的空闲分区，于是直接分配给进程 B。按照类似的方法，可以为进程 C 分配内存。此时伙伴系统的状态如图 5-10(b)所示。

下面再来看内存的释放过程。

（1）释放进程 A 的内存之后，两个大小为 128KB 的伙伴都变为空闲状态，于是它们可以合并成一个大小为 256KB 的伙伴。由于进程 B 占据着另外一个伙伴，因此不能再继续合并。此时伙伴系统的状态如图 5-10(c)所示。

（2）进程 B 的退出释放了另一个 256KB 的伙伴，于是这两个空闲伙伴又合并为一个 512KB 的空闲伙伴；进程 C 的退出释放了另一个 512KB 的伙伴，这两个伙伴又合并为 1MB 大小的内存。至此，内存释放过程结束，如图 5-10(d)所示。