分配单元的大小是一个重要的设计因素。分配单元越小，位图越大。然而即使只有4个字节大小的分配单元，32位的内存也只需要位图中的1位；32n位的内存需要n位的位图，所以位图只占用了1/33的内存。若选择比较大的分配单元，则位图更小。但若进程的大小不是分配单元的整数倍，那么在最后一个分配单元中就会有一定数量的内存被浪费了。

因为内存的大小和分配单元的大小决定了位图的大小，所以它提供了一种简单的利用一块固定大小的内存区就能对内存使用情况进行记录的方法。这种方法的主要问题是，在决定把一个占k个分配单元的进程调入内存时，存储管理器必须搜索位图，在位图中找出有k个连续0的串。查找位图中指定长度的连续0串是耗时的操作（因为在位图中该串可能跨越字的边界），这是位图的缺点。

2. 使用链表的存储管理

另一种记录内存使用情况的方法是，维护一个记录已分配内存段和空闲内存段的链表。其中链表中的一个结点或者包含一个进程，或者是两个进程间的一个空的空闲区。可用图3-6c所示的段链表来表示图3-6a所示的内存布局。链表中的每一个结点都包含以下域：空闲区（H）或进程（P）的指示标志、起始地址、长度和指向下一结点的指针。

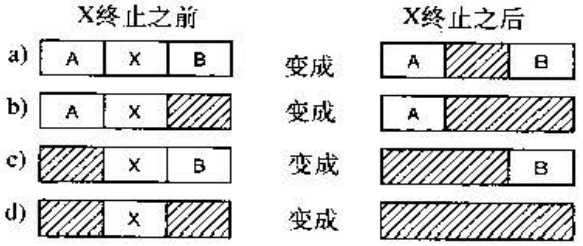

图3-7 结束进程X时与相邻区域的四种组合

在本例中，段链表是按照地址排序的，其好处是当进程终止或被换出时链表的更新非常直接。一个要终止的进程一般有两个邻居（除非它是在内存的最底端或最顶端），它们可能是进程也可能是空闲区，这就导致了图3-7所示的四种组合。在图3-7a中更新链表需要把P替换为H；在图3-7b和图3-7c中两个结点被合并成为一个，链表少了一个结点；在图3-7d中三个结点被合并为一个，从链表中删除了两个结点。

因为进程表中表示终止进程的结点中通常含有指向对应于其段链表结点的指针，因此段链表使用双链表可能要比图3-6c所示的单链表更方便。这样的结构更易于找到上一个结点，并检查是否可以合并。

当按照地址顺序在链表中存放进程和空闲区时，有几种算法可以用来为创建的进程（或从磁盘换入的已存在的进程）分配内存。这里，假设存储管理器知道要为进程分配的多大的内存。最简单的算法是*首次适配*（first fit）算法。存储管理器沿着段链表进行搜索，直到找到一个足够大的空闲区，除非空闲区大小和要分配的空间大小正好一样，否则将该空闲区分为两部分，一部分供进程使用，另一部分形成新的空闲区。首次适配算法是一种速度很快的算法，因为它尽可能少地搜索链表结点。

对首次适配算法进行很小的修改就可以得到*下次适配*（next fit）算法。它的工作方式和首次适配算法相同，不同点是每次找到合适的空闲区时都记录当时的位置。以便在下次寻找空闲区时从上次结束的地方开始搜索，而不是像首次适配算法那样每次都从头开始。Bays（1977）的仿真程序证明下次适配算法的性能略低于首次适配算法。

另一个著名的并广泛应用的算法是*最佳适配*（best fit）算法。最佳适配算法搜索整个链表（从开始到结束），找出能够容纳进程的最小的空闲区。最佳适配算法试图找出最接近实际需要的空闲区，以最好地匹配请求和可用空闲区，而不是先拆分一个以后可能会用到的大的空闲区。

以图3-6为例来考察首次适配算法和最佳适配算法。假如需要一个大小为2的块，首次适配算法将分配在位置5的空闲区，而最佳适配算法将分配在位置18的空闲区。

因为每次调用最佳适配算法时都要搜索整个链表，所以它要比首次适配算法慢。让人感到有点意外的是它比首次适配算法或下次适配算法浪费更多的内存，因为它会产生大量无用的小空闲区。一般情况下，首次适配算法生成的空闲区更大一些。

最佳适配的空闲区会分裂出很多非常小的空闲区，为了避免这一问题，可以考虑*最差适配*（worst fit）算法，即总是分配最大的可用空闲区，使新的空闲区比较大从而可以继续使用。仿真程序表明最差适配算法也不是一个好主意。

如果为进程和空闲区维护各自独立的链表，那么这四个算法的速度都能得到提高。这样就能集中精力只检查空闲区而不是进程。但这种分配速度的提高的一个不可避免的代价就是增加复杂度和内存释放