伙伴算法剖析：

**1、什么是伙伴**

a.两个块大小相同

b.两个块地址连续

c.同属于一个大块，第0块和第1块是伙伴，第2块和第3块是伙伴，但是第1块和第2块不是伙伴）

**2、伙伴位图**

  伙伴算法通过位图进行操作，用一位描述相邻的两块（第0块和第1块是伙伴，第2块和第3块是伙伴，但是第1块和第2块不是伙伴）的状态。这个位码叫伙伴位码。

为0的时候：说明这两块都为空闲

为1的时候：说明这两块有一块是忙（两块都忙是属于1状态吗？）

 注意：这里的块就是本级的单位块。假设现在所属的组是64页，那它的伙伴块就是64页。即两个连续的64页——128页需要一个bit

bitmap\_size=内存页区页数/内存页区页数（2^order）/8/2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

**3、基于伙伴算法的内存分配**

伙伴算法每次只能分配2的幂次页的空间。

linux以4k页作为页管理的基本单位，一个page结构对应一个页。因为伙伴算法是根据每个zone来的，因此在struct zone的结构体中有一个struct free\_area  free\_area[MAX\_ORDER]成员;MAX\_ORDER默认是11,也就是每个zone中有11个free\_area结构体，分别是2^0,2^1,……2^10，这就是内存块链表，也就是最大分配的内存卡是从free\_area[10]中能够获得，大小是2^10页，即4M。（启动之后能获得的最大连续物理空间是4M，要超过4M的连续空间，要么是该order的值，要么是在启动之前就获得，就是在mem\_init之前）

struct free\_area {

struct list\_head  free\_list;

unsigned long   nr\_free;

};

可以看出free\_area结构体仅仅包含了一个双向链表，指向了空闲块的前后页，还有一个里面的free的页数。

  举个例子：在分配4（2^2）页的时候，系统会先从free\_area[2]中查看其成员nr\_free是否空，如果空则从中分配，如果free\_area[2]中没有空的，就从它的上一级free\_area[3]中分配，并将多余的内存块加入到free\_area[2]中。如果free\_area[3]的nr\_free也没有空闲，则从更上一级，以此类推直至free\_area[max\_order]，如果顶级都没有空间，就报告分配失败。

**4、基于伙伴算法的内存释放**

内存的释放是分配的逆过程，也可以看做是伙伴的合并。当释放一个块时，先在其对应的free\_area链表中查找是否有伙伴存在，如果没有伙伴块，直接将要释放的块插入链表头；如果有，则将其从链表下摘下，合并成一个大块，然后继续查找在合并后的块在更大一级链表中是否有伙伴存在，直至不能合并或者已经合并至最大的块2^10为止。

**5、伙伴算法的不足**

a.合并的要求太过严格：只能是满足伙伴关系的块。比如第1,2块就不能合并。

b.碎片问题：如何内存管理的算法都存在这个问题，一个连续的内存中仅仅一个页面被占用，导致这整个内存区都不具备合并的条件。

c.算法中的浪费现象：伙伴算法是按2的幂次方分配内存区的，当需要257（2^8+1）个页面时，不得不申请2^9的页面。于是就有255个页面被浪费掉了。

d.算法的效率问题：伙伴算法涉及了比较多的计算还有链表和位图的操作，开销还是比较大的，如果每次2^n大小的伙伴块就会合并到2^(n+1)的链表队列中，那么2^n大小链表中的块就会因为合并操作而减少，但系统随后立即有可能又有对该大小块的需求，为此必须再从2^(n+1)大小的链表中拆分，这样的合并又立即拆分的过程是无效率的。