第8章 动态存储管理

8.1 概述

程序执行过程中，(数据)结构中的每一个数据元素都对应一定的存储空间，数据元素的访问都是通过对应的存储单元来进行的。存储空间的分配与管理是由操作系统或编译程序负责实现的，是一个复杂而又重要的问题，现代的存储管理往往采用动态存储管理思想。

动态存储管理：如何根据“存储请求”分配内存空间？如何回收被释放的(或不再使用的)内存空间？

对于允许进行动态存储分配的程序设计语言，操作系统在内存中划出一块地址连续的大区域(称为堆)，由设计者在程序中利用语言提供的内存动态分配函数(如C的malloc()，calloc()，free()函数，C++的new，delete函数等)来实现对堆的使用。

1 两个基本概念

- 占用块：已分配给用户使用的一块地址连续的内存区域；
- 空闲块：未曾分配的地址连续的内存区域；

2 用户请求分配内存，系统的处理方式

当有用户程序进入系统请求分配内存时，系统有两种处理方式：

(1) 系统从高地址空闲块中进行分配，直到分配无法进行时，才回收所有用户不再使用的空闲块，重新组织一个大的空闲块来再分配；

(2) 用户程序一旦运行结束，便将它所占内存区释放成为空闲块，同时，每当新用户请求分配内存时，系统需要巡视整个内存区中所有空闲块，并从中找出一个“合适”的空闲块分配之。

对于(2)的情况，系统需建立一张“可利用空间表”。

程序运行过程中，不断地对堆中的部分区域进行分配和释放，堆中会出现占用块和空闲块交错的状态，如图8-1所示。

3 动态存储分配的基本问题

(1) 当某一时刻用户程序请求分配400个字节的存储空间，如何分配?

- 将块A分配给用户程序?
- 从大块C中划出一部分分配给用户程序?

(2) 当某一时刻分配B块的用户程序运行结束，B块要进行回收，如何回收?

- B块直接回收并成为一个独立的空闲块?
- B块回收并和前、后的空闲块A、C合并后形成一个更大的空闲块?

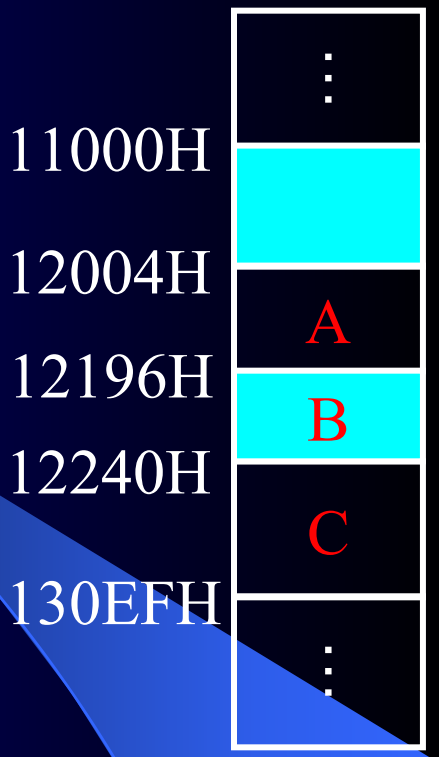


图8-1 堆的状态

8.2 可利用空间表及分配方法

可利用空间表中包含所有可分配的空闲块，当用户请求分配时，系统从可利用空间表中删除一个结点分配之；当用户释放其所占内存时，系统即回收并将它插入到可利用空间表中。因此，可利用空间表亦称做“存储池”。

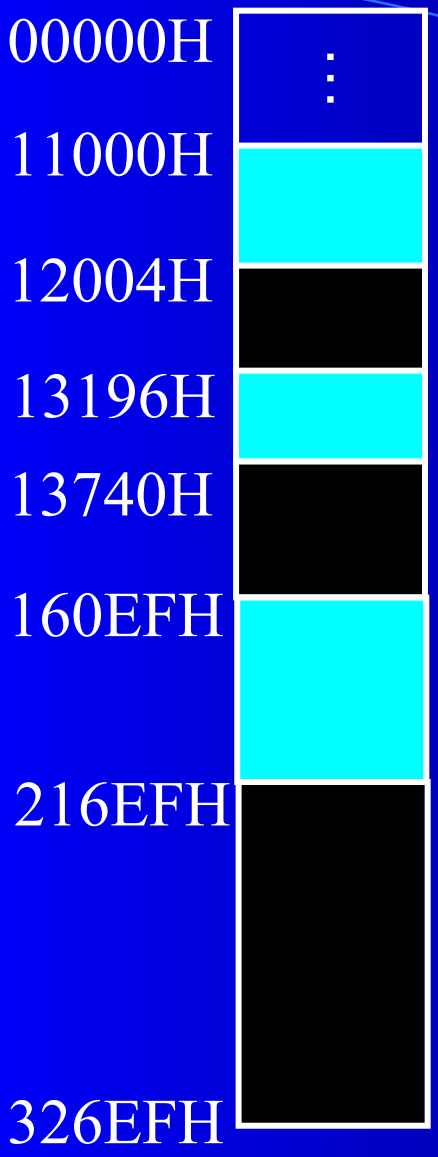
8.2.1 可利用空间表的组织

可用空间表的组织有两种方式：目录表方式和链表方式，如图8-2所示 。动态存储管理中需要不断地进行空闲块的分配和释放，对目录表来说管理复杂，因此，可利用空间表通常以链表方式组织 。

当可利用空间表以链表方式组织时，每个空闲块就是链表中的一个结点。

- 分配时：从链表中找到一个合适的结点加以分配，然后将该结点删除之；
- 回收时：将空闲块插入到链表中。

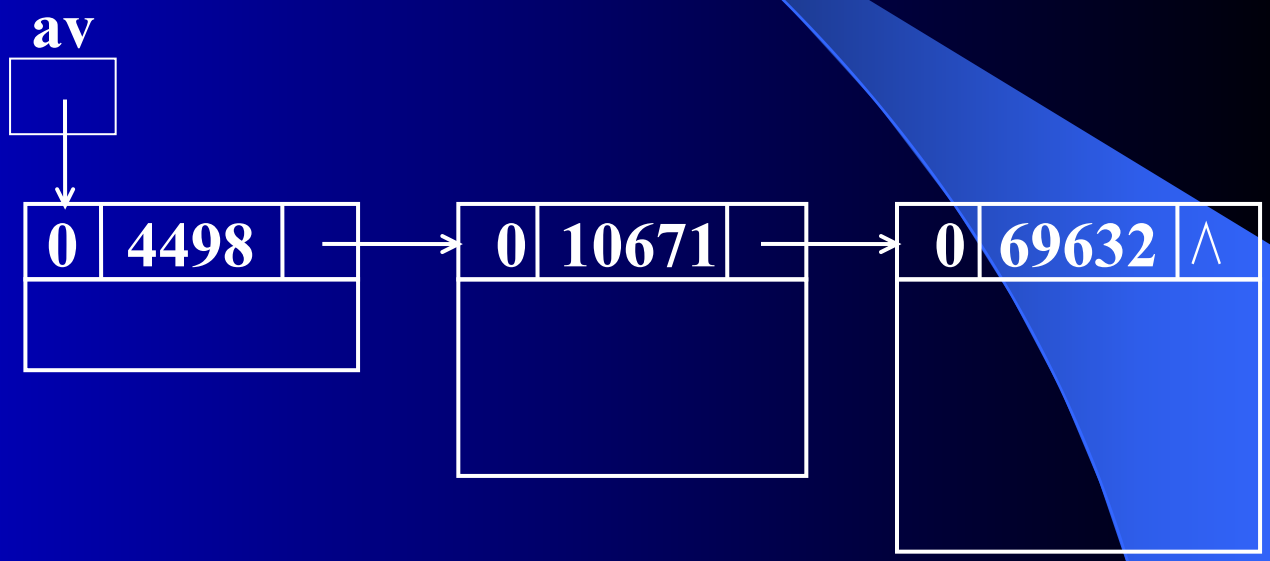
实际的动态存储管理实施时，具体的分配和释放的策略取决于结点(空闲块)的结构。



(a) 堆的状态

起始地址	空闲块大小	使用情况
12004H	4498	空闲
13740H	10671	空闲
216EFH	69632	空闲

(b) 目录表方式



(c) 链表方式

图8-2 动态存储管理过程中的内存状态和空闲表结构

8.2.2 结点结构方式与分配策略

1 请求分配的块大小相同

将进行动态存储分配的整个内存区域(堆)按所需大小分割成若干大小相同的块，然后用指针链接成一个可利用空间表。

- 分配时：从表的首结点分配，然后删除该结点；
- 回收时：将释放的空闲块插入表头。

2 请求分配的块大小只有几种规格

根据统计概率事先对动态分配的堆建立若干个可利用空间链表，同一链表中的结点(块)大小都相同。

- 分配时：根据请求的大小，将最接近该大小的某个链表的首结点分配给用户。若剩余部分正好差不多是另一种规格大小，则将剩余部分插入到另一种规格的链表中，然后删除该结点；
- 回收时：只要将所释放的空闲块插入到相应大小的表头。

存在的问题：

当请求分配的块空间大小比最大规格的结点还大时，分配不能进行。而实际上内存空间却可能存在比所需大小还要大的的连续空间，应该能够分配。

3　请求分配的块大小不确定

系统开始时，整个堆空间是一个空闲块，链表中只有一个大小为整个堆的结点，随着分配和回收的进行，链表中的结点大小和个数动态变化。

由于链表中结点大小不同，结点中除标志域和链域之外，尚需有一个结点大小域(size)，以保存空闲块的大小，如图8-2(b)。

问题： 若用户请求分配大小为n(kB)的内存，而链表中有若干大小不小于n的空闲块时，如何分配?有3种分配策略。

(1) 首次拟合法(First fit)

- 分配时：从表头指针开始查找可利用空间表，将找到的第一个不小于n的空闲块的部分(所需要大小)分配给用户，剩下部分仍然是一个空闲块结点；
- 回收时：将释放的空闲块插入在链表的表头。

特点：分配时随机的；回收时仅需插入到表头。

(2) 最佳拟合法(Best fit)

- 分配时：扫描整个可利用空间链表，找到一个大小满足要求且最接近n空闲块，将其中的一部分(所需要大小)分配给用户，剩下部分仍然是一个空闲块结点；

◆ 回收时：只要将释放的空闲块插入到链表的合适位置。

为了使分配时不需要扫描整个可利用空间链表，链表组织(块回收时)成按从小到大排序(升序）。

优点：适用于请求分配的内存块大小范围较广的系统；

缺点：系统容易产生无法分配的内存碎片；无论分配与回收，都需要查找表，最费时；

(3) 最差拟合法(Worst fit)

◆ 分配时：扫描整个可利用空间链表，找到一个大小最大的空闲块，将其中的一部分(所需要大小)分配给用户，剩下部分仍然是一个空闲块结点；

◆ 回收时：只要将释放的空闲块插入到链表的合适位置。

为了使分配时不需要扫描整个可利用空间链表，链表组织(块回收时)成按从大到小排序(降序）。

特点：适用于请求分配的内存块的大小范围较窄的系统；分配无需查找，回收需要查找适当的位置。

4 选择分配策略需考虑的因素

用户的逻辑要求、请求分配量的大小分布、分配和释放的频率以及效率对系统的重要性。

8.3 边界标识法

边界标识法(Boundary Tag Method)是操作系统中一种常用的进行动态分配的存储管理方法。

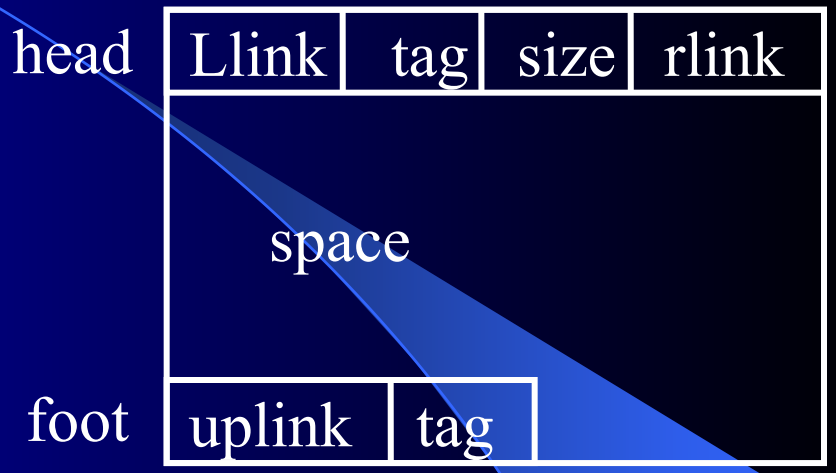
系统将所有的空闲块链接成一个双重循环链表，分配可采用几种方法(前述) 。

系统的特点

每个内存区域的头部和底部两个边界上分别设置标识，以标识该区域为占用块或空闲块，在回收块时易于判别在物理位置上与其相邻的内存区域是否为空闲块，以便于将所有地址连续的空闲存储区合并成一个尽可能大的空闲块。

8.3.1 可利用空闲表结点结构

```
typedef  struct word
   {  Union
      {  struct word *llink;
         struct word *uplink;
      };
      int tag;
      int size;
      struct word  *rlink;
      OtherType other;
   }WORD, head, foot, *Space;
#define FootLoc(p) p+p->size-1
```



8.3.2 分配算法

分配算法比较简单，可采用前述三种方法中的任一种进行分配。设采用首次拟合法，为了使系统更有效地运行，在边界标识法中还做了两条约定：

① 选定适当常量e，设待分配空闲块、请求分配空间的大小分别为m 、 n 。

- 当m-n≤e时：将整个空闲块分配给用户；
- 当m-n>e时：则只分配请求的大小n给用户；

作用：尽量减少空闲块链表中出现小碎片(容量≤e)，提高分配效率；减少对空闲块链表的维护工作量。为了避免修改指针，约定将高地址部分分配给用户。

② 空闲块链表中的结点数可能很多，为了提高查找空闲块的速度和防止小容量结点密集，每次查找时从不同的结点开始——上次刚分配结点的后继结点开始。

```
Space AllocBoundTag( Space  *pav, int n )
  {  p = pav ;
     for  ( ; p &&p->size<n && p ->rlink!=pav; p=p->rlink )
        if ( !p || p->size<n ) return NULL ;
        else
           {  f=FootLoc( p ) ; Pav=p->rlink ;
              if  ( p->size–n<=e )
```

```
            {  if  ( pav==p )  pav=NULL ;
               else
                  {  pav->llink=p->link ;
                     p->llink->rlink=pav ;   }
               p->tag=f->tag=1 ;
            }
         else
           {  f->tag=1 ; p->size-=n ; f= FootLoc( p ) ;
              f->tag= 0 ; f->uplink=p ; p=f+1;
             p->tag=1 ; p->size=n ;
           }
        return p ;
      }
}
```

8.3.3 回收算法

当用户释放占用块，系统需立即回收以备新的请求产生时进行再分配。关键的是使物理地址毗邻的空闲块合并成一个尽可能大的结点，则需检查刚释放的占用块的左、右紧邻是否为空闲块。

假设所释放的块的头地址为p，则与其低地址紧邻的块的底部地址为p-1；与其高地址紧邻的块的头地址为p+p->size，它们中的标志域就表明了两个相邻块的使用状况：

- 若(p-1)->tag=0 ： 则左邻块为空闲块；
- 若(p+p->size)->tag=0 ： 则右邻块为空闲块；

回收算法需要考虑的4种情况：

(1) 释放块的左、右邻块均为占用块

将被释放块简单地插入到空闲块链表中即可。

```
p->tag=0 ; FootLoc(p)->uplink=p ;
FootLoc(p)->tag=0 ;
if ( !pav ) pav=p->llink=p->rrlink=p ;
else
    {   q=pav->llink ;  P->rlink=pav ;
        p->llink=q ; q->rlink=pav->llink=p ;
        Pav=p ;
    }
```

(2) 释放块的左邻块空闲而右邻块为占用

和左邻块合并成一个大的空闲块结点，改变左邻块的size域及重新设置(合并后)结点的底部。

n=p->size ; s=(p-1)->uplink ; s->size+=n;

f=p+n–1 ; f->uplink=s ; f->tag=0 ;

(3) 释放块的左邻占用而右邻空闲

和右邻块合并成一个大的空闲块结点，改变右邻块的size域及重新设置(合并后)结点的头部。

t=p+p->size ; p->tag=0 ; q=t->llink ; p->llink=q ;

q->rlink=p ; q1=t->rlink ; p->rlink=q1 ;

q1->llink=p ; p->size+=t->size ; FootLoc(t)->uplink=p ;

(4) 释放块的左、右邻块均为空闲块

和左、右邻块合并成一个大的空闲块结点，改变左邻块的size域及重新设置(合并后)结点的底部。

```
n=p->size ; s=(p-1)->uplink ; t=p+p->size ;
s->size+=n+t->size ;  q=t->llink ; q1=t->rlink ;
q->rlink=q1 ; q1->llink=q ;
FootLoc(t)->uplink=s;
```

8.4 伙伴系统

伙伴系统是一种非顺序内存管理方法，不是以顺序片段来分配内存，是把内存分为两个部分，只要有可能，这两部分就可以合并在一起; 且这两部分从来不是自由的，程序可以使用伙伴系统中的一部分或者两部分都不使用。与边界标识法类似，所不同是：无论占用块或空闲块，其大小均为2的k次幂。

8.4.1 可利用空间表的结构

为了再分配时查找方便起见，我们将所有大小相同的空闲块建于一张子表中。每个子表是一个双重链表，这样的链表可能有m+1个，将这m+1个表头指针用向量结构组织成一个表，这就是伙伴系统的可利用空间表。

可利用空间表的数据类型描述如下：

```
#define  M  16
typedef struct WORD_b
   {   WORD_b  * llink;     /*   前驱结点   */
       int    tag;         /*   使用标识    */
       int    kval;        /*   块的大小,是2的幂次   */
       WORD_b  *rlink;     /*   后继结点   */
       OtherType    other;
   } WORD_b, head;
typedef  struct HeadNode
   {   int    nodesize;
       WORD_b * first;
   }FreeList[M+1];
```

8.4.2 分配算法

当程序提出大小为n的内存分配请求时，首先在可利用表中查找大小与n相匹配的子表.

1 算法思想

- 若存在$2^{k-1}<n\leq 2^k-1$的空闲子表结点：则将子表中的任意一个结点分配之；
- 若不存在$2^{k-1}<n\leq 2^k-1$的空闲子表结点：则从结点大小为2^k的子表中找到一个空闲结点，将其中一半分配给程序，剩余的一半插入到结点大小为2^{k-1}的子表中。

2 说明

在进行大小为$n(2^{k-i-1}<n\leq 2^{k-i}-1，\ i=1,2,\ldots,k-1)$的内存分配请求时，若所有小于$2^k$的子表均为空(没有空闲结点)，则同样需要从大小为$2^k$的子表中找到一个空闲结点，将其中$2^{k-i}$一小部分分配给用户，而将剩余部分分割成若干个结点分别插入对应的子表。

8.4.3 回收算法

当程序释放所占用的块时，系统将该新的空闲块插入到可利用空闲表中，需要考虑合并成大块问题。在伙伴系统中，只有“互为伙伴”的两个子块均空闲时才合并；即使有两个相邻且大小相同的空闲块，如果不是“互为伙伴”(从同一个大块中分裂出来的)也不合并。

1 伙伴空闲块的确定

设p是大小为2^k的空闲块的首地址，且p MOD 2^{k+1} =0，则首地址为p和$p+2^k$的两个空闲块“互为伙伴”。

首地址为p大小为2^k的内存块的，其伙伴的首地址为：

$$buddy(p,k)=\begin{cases} p+2^k & \text{若 } p \text{ MOD } 2^{k+1}=0 \\ P-2^k & \text{若 } p \text{ MOD } 2^{k+1}=2^k \end{cases}$$

2 回收算法

设要回收的空闲块的首地址是p，其大小为2^k的，算法思想是：

(1) 判断其“互为伙伴”的两个空闲块是否为空：

若不为空，仅将要回收的空闲块直接插入到相应的子表中；否则转(2)；

(2) 按以下步骤进行空闲块的合并：

- 在相应子表中找到其伙伴并删除之；
- 合并两个空闲块；

(3) 重复(2)，直到合并后的空闲块的伙伴不是空闲块为止。

系统的特点：算法简单；速度快；但容易产生碎片。