第35卷 第22期 计 算 机 工 程 2009年11月
Vol.35 No.22 Computer Engineering November 2009
·软件技术与数据库· 文章编号：1000—3428(2009)22—0077—02 文献标识码：A 中图分类号：TP314

通用三维内存管理器的设计

查 峰
(华为技术有限公司南京研究所，南京 210012)

摘 要：malloc和free是C语言中动态存储管理标准函数，在一个运行繁忙的系统中大量使用这2个函数容易造成内存碎片和系统颠簸。针对上述问题，设计一个内存管理器，采用按最大需求缓存的策略，优化空间复杂度和时间复杂度，进行多线程保护，构成一个通用的内存适配器，特别适用于数据仓库及XML信息处理，可在各类平台类服务器和网关中使用。

关键词：多线程；内存管理；缓存；双向链表

Design of 3D Universal Memory Manager

ZHA Feng
(Nanjing Institute of Huawei Technologies Co., Ltd., Nanjing 210012)

【**Abstract**】malloc and free are standard functions in C program language for dynamic memory management. If the two functions are used too much frequently in a busy system, it is easy to produce memory fragment and degree the stability of the system. Aiming at this problem, this paper designs a memory manager, adopts most demand cache strategy, optimizes it both on space and time complexity, and provides the protection for multithread, which makes it a universal memory allocator. It is suitable to data warehouse, XML information processing and so on, and can be used in various platform servers and gateways.

【**Key words**】multithread; memory management; cache; double link

1 概述

动态内存分配是指在程序执行过程中动态分配或回收存储空间的内存分配方法，系统根据程序运行过程中的实际需要即时分配，且分配的大小就是程序要求的大小。

malloc(unsigned int size)是C语言中动态存储管理的一组标准库函数之一，其作用是在内存的动态存储区中分配一个长度为size的连续空间。其参数是一个无符号整型数，返回值是一个指向所分配的连续存储域起始地址的指针malloc函数的实质体现，它有一个将可用的内存块连接为一个长列表的空闲链表。

在调用malloc函数时，它沿连接表寻找一个大到足以满足用户请求所需要的内存块。然后，将该内存块一分为二(一块的大小与用户请求的大小相等，另一块的大小就是剩下的字节)。接下来，将分配给用户的那块内存传给用户，并将剩下的那块(如果有的话)返回到连接表上。

在调用free函数时，它将用户释放的内存块连接到空闲链上。最后，空闲链会被切成很多小内存片段，如果这时用户申请一个大的内存片段，那么空闲链上可能没有可以满足用户要求的片段。于是，malloc函数请求延时，并开始在空闲链上遍历地检查各内存片段，对它们进行归整处理，将相邻的小空闲块合并成较大的内存块。

内存分配算法很容易引起堆的碎片，申请和回收内存时的随机性使回收的内存很难合并起来容纳大的分配块。分配空间所需的时间开销不是常数时间，一般，分配的存储块越大，所需搜索的时间越长。内存资源是临界资源，对其操作一般都加了锁操作，在多个线程并发动态使用内存资源时，锁操作的等待率会大大增加，这将延长系统的等待时间，很容易造成系统颠簸。

2 现有内存管理器的应用

内存池化管理是最常用的方法。静态或动态申请块数组作为内存池，将数组下标从0开始编号，直至最大值 n-1，将这些数存入一个队列L，申请内存时，从队列中取出一个数，以此数为下标在块数组中定位到块地址；如果申请成功，将此数从队列中删除。归还内存时，将块下标插入队列L。若数组是动态申请的，则在所有块都用完时还可以动态申请更大的新块，并将旧存储块的全部内容移到新块中，内存读取数据时，只要按块下标和块首址2个参数就能定位到数据块。这种方法管理内存简单，只要维护好块下标，就能循环地使用内存池。但是缺点也很明显：对不同尺寸的内存块需要维护不同的缓冲池；数据块是连续的，如果需要的同类数据块较多，系统不一定能提供大块的内存区供池使用。文献[1]介绍了一种基于开源项目OCR的存储管理方法，其原理是存储池通过malloc以块为单位从堆区中分配空间，再从存储池的块中二次分配空间，每分配一次，内置指针移动一次，一个块可能被二次分配几次才被释放。这种池不通用，只是减少了malloc的次数，且空间利用率很低。文献[2]设计了可滑动动态内存缓冲池，为所有线程申请一个整块空间作为存放数据的缓冲池，按需要建立一定数量单向指针来管理这个缓冲池，每个指针链表代表一个数据缓冲区。这些系统都采用预分配固定尺寸内存块，空间效率不高。文献[3]为每个应用程序模块保留自己所需的一块较大内存区，再将该内存区划分为若干内存池。每个内存池由若干尺寸相同的内存

作者简介：查 峰(1971－)，男，高级工程师，主研方向：计算机算法，通信协议

收稿日期：2009-04-16 **E-mail：**fengzha@china.com

块组成，但是该方法对内存块尺寸大小通用性未作规定。

3　通用内存管理器

本文设计通用内存管理器的基本思想是，在系统中为每一特定大小的内存块均构造一个缓冲池头节点，这些节点都用单向链表连接在一起；每个缓冲池中均有零个或者多个数据块(图 1)；数据块起始地址开始为块头结构，随后是紧跟块的内容(图 2、图 3)，内容最大长度存储在缓冲池头节点 blocksize 字段中，返回给用户的内存指针是块内容的首指针。块内容的大小为 2 的整数次幂。设有 n 个缓冲池头节点，第 $i(1<i\leq n)$个缓冲池头节点的 blocksiz 是第 i-1 个节点的 2 倍。这样定位缓冲池算法复杂度非常小，只要从头遍历此链表，找到第 1 个大于等于申请内存的缓冲池头节点。然后在此节点的缓冲池内寻找预分配的内存块。每个缓冲节点包含块内容大小指示、一个线程锁和 2 个双向链表(可用块链表和已用块链表)。

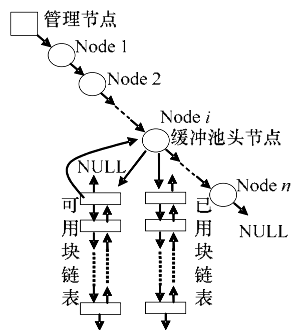


图 1　内存管理器结构

pre	next	m_pool	块内容
块内容			

图 2　块结构

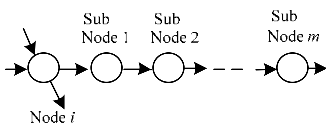


图 3　二次拉链升维优化

3.1　数据结构

管理节点结构如下：

```
typedef struct memMgr{
unsigned int MinBlockSize; //最小缓冲池尺寸
unsigned int MaxBlockSize; //最大缓冲池尺寸
struct mem_pool *pools;
}MemMgr
```

缓冲池头节点结构如下：

```
struct mem_pool{
unsigned int blocksize; //缓冲块尺寸
struct mem_pool *next; //下一个缓冲池头节点
unsigned int free_count; //可用块数量
struct mem_block *used_link; //已用块队列
struct mem_block *free_link; //可用块队列
pthread_mutex_t mutex; //锁变量};
```

数据块结构如下：

```
struct mem_block{
        struct mem_block *pre; //下一个块指针
  struct mem_block *next; //前一块指针
     struct mem_pool *pool; //缓冲池指针};
```

3.2　算法实现

在使用结构前，初始化 MemMgr 型管理节点，设立最小和最大缓冲块界限。建立单向链表，每个链表节点为 mem_pool 型，对每个节点初始化，将可用队列、已用队列 2 个双向链表都置为空，对其中代表互斥锁的字段进行初始化，此时缓冲池已随着链表被排序。假设最小块尺寸为 2^m，最大块尺寸为 2^n, $n>m$，链表中共有 n-m+1 个该类节点。

在申请内存 M 时，进入管理节点，从第 1 个缓冲池节点 pools 开始，如果该池的块尺寸大于等于 M，就定位到缓冲池，否则继续遍历到下一节点，直至找到符合条件的节点。如果整个链表遍历完均不满足条件，则可判定 $M>2^n$，直接调用系统 malloc 函数。

定位到缓冲池头节点 i 后，按如下步骤操作：

加锁，如果可用队列中 free_link 为空，则 malloc 有若干块，每个块大小等于缓冲池节点 blocksize 字段值与 mem_block 型节点大小之和，在生成块时，将块起始指针强制转换为 mem_block 指针型并加入到 free_link 队列中。

从双向队列头中取出并删除一个节点指针，将此指针位移 mem_block 型尺寸，即为返回给用户的内存指针。解锁，内存分配完毕。

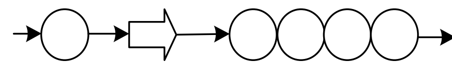
释放内存时，将用户传入的指针向前位移 mem_block 类型尺寸，将此值强制转换为 mem_block*块指针；加锁，将块指针插入到其 pool 指向缓冲池节点的可用队列中并从可用队列中删除；解锁，内存释放完毕。

对于那些不经常使用的内存申请，依然可以直接用 malloc 和 free 函数，但是实际申请的节点还是实际尺寸的 mem_block。

3.3　结构的优化

在此算法中，假设定位在第 i 个缓冲池节点，池中每个块的内容大小是 $2^{m=i-1}$，块实际平均空间的期望值为 $3/4\times2^{m=i-1}$，整个管理系统有近 1/4 空间被浪费。假设在每个缓冲池再拉一个同类型节点链表，链表个数是 2 的幂次方数减 1，假定是 2^k，链表中节点的 blocksize 按节点序号为升序，二次链表第 j 个节点的 blocksize 为 $1/2\times2^{m=i-1}\times(1+j/(2^k))$。定位缓冲池时，先定位到二次链表中第 1 个 blocksize 大于请求尺寸的节点。在可用队列中取块的操作与前面一致，见图 3。

除去管理字段的开销，优化后对每个内存块浪费的期望值仅为 $1/(2^{k+1})$。另一个重要的特征是在多线程运行时，由于不同线程申请的内存块大小是随机不同的，因此申请操作往往会被定位到不同的缓冲池，很大程度上避免了锁冲突。这样，二次拉链法在定位缓冲池时没有增大多少开销，其查询期望值却仅为$(n$-m+1)/2+k/2，见图 4。



一个缓冲池头节点分裂为多个

图 4　二维到三维立方的优化

进一步优化，将每个缓冲池头节点分裂为多个同构节点组，实际定位时，用轮询或负载均衡原则定位一个节点，从而进一步降低了多线程锁阻塞的概率。此优化在多个线程并发申请同一尺寸内存时性能提升效果很明显。

4　结束语

采用按内存尺寸分类的原则，建立三维立方索引结构，

(下转第 81 页)