设计文档

目录

1.概要

1.1. 动态内存管理

存储管理——每一种数据结构都必须研究该结构的存储结构，但它是借助于某一高级语

言中的变量说明来加以描述的，并没有涉及到具体的存储分配。

实际上，结构中的每个数据元素都占有一定的内存位置，在程序执行的过程中，数据元素的存取是通过对应的存储单元来进行的。研究数据存储与内存单元对应问题，就是存储管理问题。

1.1.2.动态存储管理的基本问题

- 1、如何根据用户提出的“请求”来分配内存。

  根据申请内存大小利用相应分配策略分配给用户所需空间；若分配块大小与申请大小相差不多，则将此块全部分给用户；否则，将分配块分为两部分，一部分给用户使用，另一部分继续留在可利用空闲表中。
- 2、如何收回被用户“释放”的内存，以备新的“请求”产生时重新进行分配。

  测试回收块前后相邻内存块是否空闲；若是则需将回收块与相邻空闲块合并成较大的空闲块，再链入可利用空闲表中。

1.1.3.各种内存分配方法优势及不足

- **1、首次拟合法**

  方案：按分区的先后次序，从头查找，找到符合要求的第一个分区

  优点：该算法的分配和释放的时间性能较好，较大的空闲分区可以被保留在内存高端。操作方便，查找快捷；

  缺点：但随着低端分区不断划分而产生较多小分区，每次分配时查找时间开销会增大。
- **2、最佳拟合法**

  方案：分配不小于 n 且最接近 n 的空闲块的一部分。

  优势：尽可能将大的空闲块留给大程序使用；

  缺点：从个别来看，外碎片较小，但从整体来看，会形成较多外碎片。较大的空闲分区可以被保留。
- **3、最坏拟合法**

  方案：分配不小于 n 且是最大的空闲块的一部分。

  优势：尽可能减少分配后无用碎片；

  缺点：基本不留下小空闲分区，但较大的空闲分区不被保留。
- **4、循环首次拟合法（又称下次拟合法）**

  方案：按分区的先后次序，从上次分配的分区起查找（到最后分区时再回到开头），找到符合要求的第一个分区

  优势：该算法的分配和释放的时间性能较好，使空闲分区分布得更均匀

  缺点：较大的空闲分区不易保留

1.2. 边界标识法

边界标识法（boundary tag method）是操作系统中用以进行动态分区分配的一种存储管理方法。系统将所有的空闲块链接在一个双重循环链表结构的可利用空间表中；分配可按首

次拟合进行，也可按最佳拟合进行。

采用边界标识法系统的特点：在每个内存区的头部和底部两个边界上分别设有标识，以标识该区域为占用块或空闲块，使得在回收用户释放的空闲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，以便将所有地址连续的空闲存储区组合成一个尽可能大的空闲块。

1.2.1.结点结构（C 语言描述）

下图是 C 语言描述的边界标识法节点结构，结点由 3 部分组成：头部 head 域、space 域和底部 foot 域。

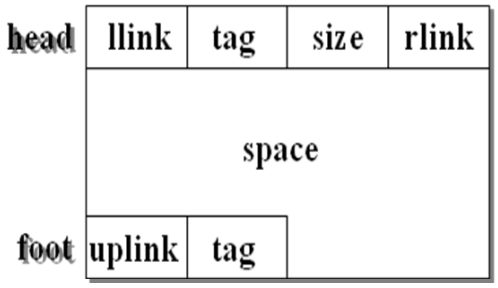


图. 节点结构

space：为一组地址连续的存储单元，是可以分配给用户使用的内存区域。其大小有 head 中的 size 域指示,并以头部 head 和底部 foot 作为它的两个边界。

tag：标志域,在 head 和 foot 中都设有该域,值为“0”表示空闲块,值为“1”表示占用块。

size：整个结点的大小，包括头部 head 和底部 foot 所占空间。

llink：链域，位于头部 head 域，指向前驱结点。

rlink：链域，位于头部 head 域，指向后继结点。

foot：位于结点底部,它的地址是随结点中 space 空间的大小而变的.

uplink：链域，位于尾部 foot 域，指向本结点头部，其值即为该空闲块的首地址。

1.2.2.分配算法

找到一个空闲的内存块，作为起始搜索内存块，搜索算法可以采用首次拟合法和最佳拟合法。下面针对首次拟合的分配算法进行一下介绍。设申请空间的大小为 size_n，搜索内存块的指针为 pointer_pav。首先获取当前内存块头部信息中内存块的大小 size，如果内存块的大 size 大于或者等于申请空间的大小 size_n，则将此内存块的一部分或整个分配给用户。如果内存块的大小 size 小于申请空间的大小 size_n，则将搜索内存块的指针 pointer_pav 指向下一个空闲的内存块。继续比较，直到找到或搜索完所有的空闲内存块为止。 为了避免产生无法使用的内存碎片，我们定义一个常量 size_e。如果分配给用户后内存块剩余空间的大小小于常量 size_e，则将整个内存块全部分配给用户。否则，只分配申请空间的大小。

另外，如果搜索指针的位置固定从某一个内存块开始，势必产生内存分布不均，一些地方密集，而一些地方稀疏。为了解决这个问题，我们将搜索指针指向每次释放后的内存块。

时间复杂度的讨论：

该算法的时间主要花费在查找第一个满足条件的内存块（简称命中块）上。而查找时间

的长短又由链表的长度和命中块在链表中的位置决定。设链表的长度为 n，命中块在链表中的位置为 i：

1、当 i=0 时，其时间复杂度为 O(1)，这是最好情况 ；

2、当 i=n-1 时，其时间复杂度为 O(n)， 这是最差情况 ；

因为命中块的位置是任意的，所以我们分析一下平均时间复杂度。假设每一个内存块满足条件的概率是均等的，都为 1/n，那么 n 的值越大花费的时间越长。因此查找算法的平均时间复杂度为 O(n)。

1.2.3.回收算法

回收算法是将用户不用的内存块回收，以便再次分配。 根据内存块物理上左右邻区的空闲占用情况分为四种情况：

1、左右邻区都为空闲的情况；

2、左右邻区都为占用的情况；

3、左邻区为空闲，右邻区为占用的情况；

4、左邻区为占用，右邻区为空闲的情况。

每一种情况对应一种回收算法。下面对每一种回收算法分别进行一下介绍：

第一种：左右邻区都为空闲的情况

为了让三个空闲块合并成一个空闲块，可以修改左邻块的空间大小，然后，将右邻块从链表中删除，再修改合并后内存块的底部信息。

第二种：左右邻区都为占用的情况

这种情况比较简单，只要将释放块插入到空闲块链表中即可。因为链表是无序的，所以插入位置是任意的。

第三种：左邻区为空闲，右邻区为占用的情况

修改左邻块的空间大小，然后修改底部信息，将此释放块和左邻块合并成一个内存块。

第四种：左邻区为占用，右邻区为空闲的请况

修改释放块的头部信息，然后修改释放块的底部信息。

时间复杂度的讨论：

由于在每个内存块的头部和底部都设立标识，所以不需要进行查找来找到与它毗邻的空闲块进行合并；其次，链表中内存块的位置是无序的，所以不需要查找链表进行插入。总之，不管哪种情况释放算法的时间复杂度都为常量。

2.设计思想

本程序所实现的内存分配器采用了上面所描述的边界标识算法，采用边界标识法内存系统的特点：在每个内存区的头部和底部两个边界上分别设有标识，以标识该区域为占用块或空闲块，使得在回收用户释放的空闲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，以便将所有地址连续的空闲存储区组合成一个尽可能大的空闲块。

2.1.分配方案

分配方案采用了首次拟合法（first-fit）和和循环首次拟合法（next-fit），如下程序段所描述：

```
#ifdef NEXT_FIT
static char *g_pNextFit; /* 循环首次适应算法的指针 g_pNextFit */
#endif
```

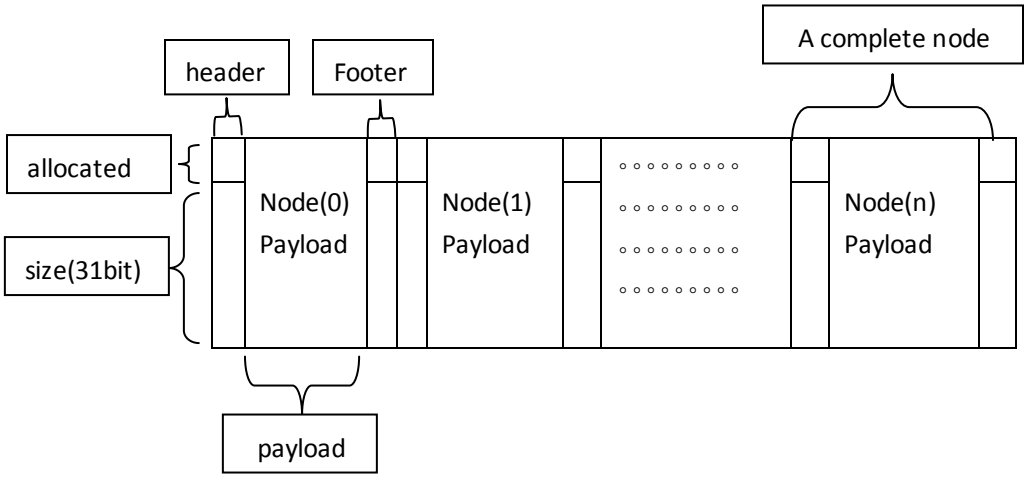
如果定义了宏 NEXT_FIT，则采用了循环首次拟合法，否则默认为首次拟合法。

2.2.回收方案

回收方案则采用上面 1.2.3 节中所示的回收算法，由于在每个内存块的头部和底部都设立标识，所以不需要进行查找来找到与它毗邻的空闲块进行合并；其次，链表中内存块的位置是无序的，所以不需要查找链表进行插入。所以，不管哪种情况释放算法的时间复杂度都为常量。

3.宏定义及数据结构描述

3.1.内存节点定义



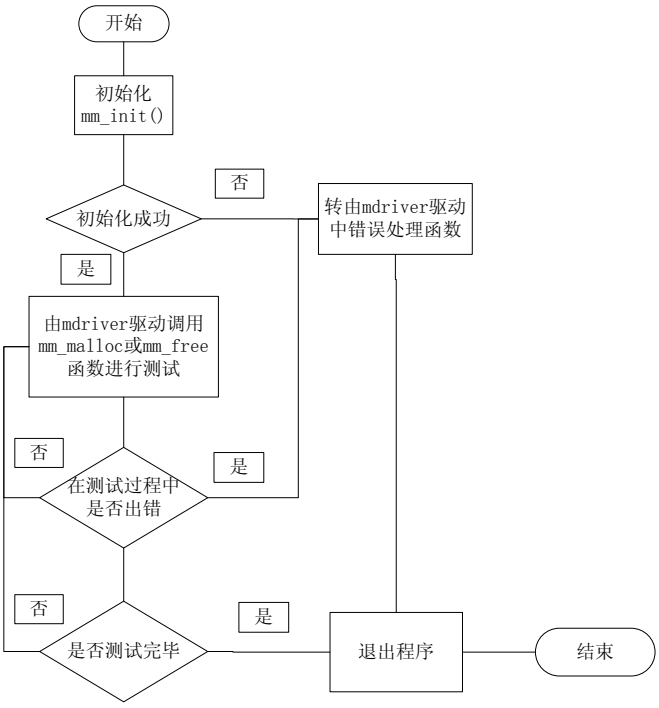
在上面的节点定义中，每个节点有三部分组成，首部 header，占用 4 个字节；尾部 footer，也是占用 4 个字节，和 payload 部分，大小由 void * mm_malloc(size_t size);中的 size 参数指定。其中首部和尾部中的各个字段相同，首部中的 size 字段（31bits）为在内存块的大小，allocated 字段（1bit）指示该内存块是否被占用，0 表示空闲，1 表示被占用。

3.2.常见的宏定义

```
/* 将size和allocated字段压缩成一个字*/
#define COMPRESS(size, alloc)  ((size) | (alloc))
/* 从地址 p 所示的内存单元读取一个数据 */
#define READ(p) (*(unsigned int *)(p))
/* 从地址 p 所示的内存单元写入一个数据 */
#define WRITE(p, val)  (*(unsigned int *)(p) = (val))
/* 从地址 P 指定的内存单元读取size字段，以获取当前内存块的大小*/
#define GET_BLKSIZE(p)  (READ(p) & ~(ALIGNMENT-1))
/* 从地址 P 指定的内存单元读取allocated字段，以判断内存块是否被占用*/
#define IS_ALLOCATED(p) (READ(p) & 0x1)
/* 如果给定指向某一内存块的指针，则计算其头部字段的地址*/
#define CMPT_HEADPTR(bp)       ((char *)(bp) - WORD)
/* 如果给定指向某一内存块的指针，则计算其尾部字段的地址*/
#define CMPT_FOOTPTR(bp)       ((char *)(bp) +
GET_BLKSIZE(CMPT_HEADPTR(bp)) - DWORD)
/* 如果给定指向内存块的指针，则计算下一个内存块的地址*/
#define GET_NEXT_BLKPTR(bp)  ((char *)(bp) + GET_BLKSIZE(((char *)(bp)
- WORD)))
/* 如果给定指向内存块的指针，则计算前一个内存块的地址*/
#define GET_PREV_BLKPTR(bp)  ((char *)(bp) - GET_BLKSIZE(((char *)(bp)
- DWORD)))
/* 全局变量 */
static char *g_pHeadList = 0; /* 指向第一个内存块的指针 */
```

4.程序流程

程序的流程比较简单，如下图所示，首先在 mdriver 驱动初始化模拟内存空间后在调用 mm_init()函数进行初始化，若初始化成功，则转入 mm_malloc 和 mm_free 函数的测试，如果中间没有出错，则 mdriver 驱动进行性能测评，如果中间任何一步出错，则转由 mdriver 驱动中的错误处理函数，如此处理直至程序结束退出。



5.程序性能分析

5.1.实验平台

硬件平台：

处理器：Intel(R) Core ™2 Duo CPU T5250 @1.5GHZ 1.5GHZ

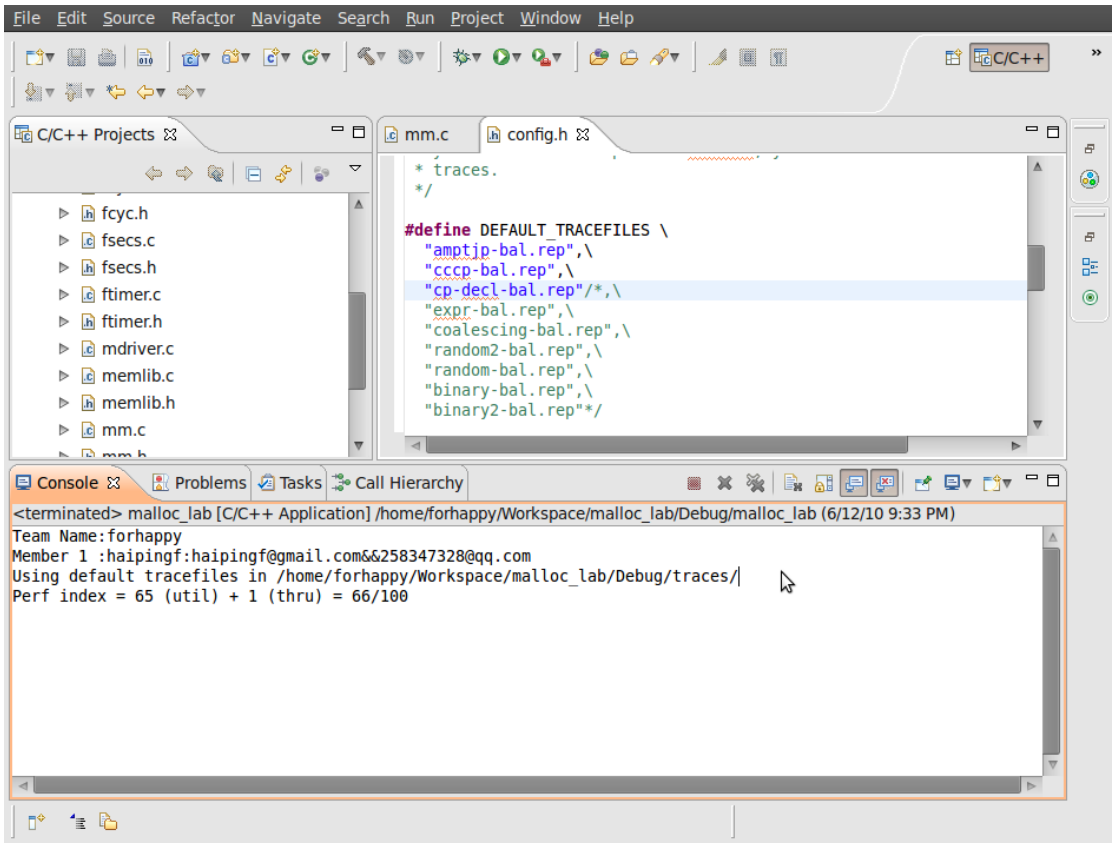
内存：2GB　DRAM

软件平台：

操作系统：Ubuntu 10.04LTS

Linux Kernel： 2.6.32-24+gcc4.4

5.2.程序运行截图



5.3.性能分析

5.3.1.指标

- *Space utilization*: The peak ratio between the aggregate amount of memory used by the driver (i.e., allocated via `mm_malloc` or `mm_realloc` but not yet freed via `mm_free`) and the size of the heap used by your allocator. The optimal ratio equals to 1. You should find good policies to minimize fragmentation in order to make this ratio as close as possible to the optimal.
- *Throughput*: The average number of operations completed per second.

按照上面的描述，性能测试分为两个部分，1、[虚拟]内存空间利用率，2、吞吐量。其中内存空间利用率表述为：由自己实现的内存分配器所利用的堆大小与驱动程序所分配的总的“虚拟”内存的最高比率；吞吐量为：每秒钟平均的操作“虚拟”内存操作次数，即（mm_malloc，mm_realloc 以及 mm_free 的次数）。

The driver program summarizes the performance of your allocator by computing a *performance index*, P, which is a weighted sum of the space utilization and throughput

$$P = wU + (1 - w)\min\left(1, \frac{T}{T_{libc}}\right)$$

where U is your space utilization, T is your throughput, and T_{libc} is the estimated throughput of `libc` malloc on your system on the default traces.1 The performance index favors space utilization over throughput, with a default of w = 0.65.（Tlibc=2300Kops/s）。

所以最后的评估指数可以说是内存空间利用率和吞吐量的加权平均，内存空间利用率默认权值为0.65，吞吐量的权值为1-w，但此时“吞吐量”间接的表述为：（函数模拟的吞吐量/Tlibc函数库的吞吐量）与1中的较小值，

5.3.2.具体分析

测试截图一：（所有的测试文件，分数不高~~）

```
Testing mm malloc
Reading tracefile: amptjp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cccp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cp-decl-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: expr-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.016228   351
 1       yes   99%    5851  0.014187   412
 2       yes   99%    6651  0.027563   241
 3       yes  100%    5383  0.020013   269
 4       yes   66%   14403  0.000391 36808
 5       yes   92%    4803  0.016847   285
 6       yes   93%    4803  0.017719   271
 7       yes   55%   12003  0.216503    55
 8       yes   51%   24003  0.643756    37
Total          84%   83594  0.973208    86

Perf index = 55 (util) + 1 (thru) = 56/100
forhappy@ubuntu:~/Workspace/testing$
```

测试截图二：

```
File  Edit  View  Terminal  Help
 4       yes    66%    14403  0.000391 36808
 5       yes    92%     4803  0.016847   285
 6       yes    93%     4803  0.017719   271
 7       yes    55%    12003  0.216503    55
 8       yes    51%    24003  0.643756    37
Total           84%    83594  0.973208    86

Perf index = 55 (util) + 1 (thru) = 56/100
forhappy@ubuntu:~/Workspace/testing$ ./mdriver -Vf amptjp-bal.rep
Team Name:forhappy
Member 1 :haipingf:haipingf@gmail.com&&258347328@qq.com
Measuring performance with gettimeofday().

Testing mm malloc
Reading tracefile: amptjp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.016249   350
Total           99%    5694  0.016249   350

Perf index = 64 (util) + 5 (thru) = 70/100
forhappy@ubuntu:~/Workspace/testing$
```

测试截图三：（binary2-bal.rep 文件，小块分配空间利用率很低）

```
File  Edit  View  Terminal  Help
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   55%   12003  0.232512    52
Total           55%   12003  0.232512    52

Perf index = 36 (util) + 1 (thru) = 36/100
forhappy@ubuntu:~/Workspace/testing$ ./mdriver -Vf binary2-bal.rep
Team Name:forhappy
Member 1 :haipingf:haipingf@gmail.com&&258347328@qq.com
Measuring performance with gettimeofday().

Testing mm malloc
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   51%   24003  0.682426    35
Total           51%   24003  0.682426    35

Perf index = 33 (util) + 1 (thru) = 34/100
forhappy@ubuntu:~/Workspace/testing$
```

6.不足及改进方案

6.1.边界标志法的缺陷

程序采用了边界标志算法，这个算法的优势是回收合并内存只需要常量的时间，而分配策略采用了首次拟合算法和循环首次拟合算法，如果采取这样方案，它的致命弱点就是需要线性时间进行内存分配，随着空闲块的链表长度越来越长，分配的时间开销将变大。这对于对时间延迟非常敏感内存你分配操作是不可忍受的，因为内存分配操作的时间越短越好，最好能在常数时间下完成操作。所以应该改用其他的分配方案。

6.2.方案的改进

针对上面提出的缺陷，应该对分配策略进行改进或采用额外的存储空间存储空闲链表的信息，这样在进行分配时就可以尽量做到在常数时间之类完成。具体的操作见另外一个设计方案。

6.3.改进后的方案效果

测试截图四：（评估指数还可以）

```
Team Name:forhappy
Member 1 :haipingf:haipingf@gmail.com&&258347328@qq.com
Using default tracefiles in ./
Measuring performance with gettimeofday().

Testing mm malloc
Reading tracefile: amptjp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cccp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cp-decl-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: expr-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   97%    5694  0.001018  5596
 1       yes   98%    5851  0.001115  5250
 2       yes   98%    6651  0.001187  5606
 3       yes   99%    5383  0.000953  5647
 4       yes   98%   14403  0.001428 10087
 5       yes   90%    4803  0.001385  3467
 6       yes   92%    4803  0.001392  3451
 7       yes   54%   12003  0.001728  6945
 8       yes   47%   24003  0.003275  7329
Total          86%   83594  0.013480  6201

Perf index = 56 (util) + 35 (thru) = 91/100
forhappy@ubuntu:~/Workspace/testing$
```

7.参考文献

【1】 严蔚敏，吴伟民著. 数据结（C 语言版）北京：清华大学出版社，2007

【2】 Donald.Knuth 著 The Art Of Computer Programming 3rd Vol1(Addison Wesley,1997)

【3】 http://blog.csdn.net/tommy_lgj/archive/2006/08/23/1108456.aspx

【4】 http://book.51cto.com/art/200907/138175.htm

【5】 http://book.csdn.net/bookfiles/1005/100100530763.shtml

【6】 http://g.oswego.edu/dl/html/malloc.html

【7】 http://www.malloc.de/en/

【8】 http://goog-perftools.sourceforge.net/doc/tcmalloc.html

8.附源代码

```
/*
 * mm-naive.c - The fastest, least memory-efficient malloc package.
 *
 * In this naive approach, a block is allocated by simply incrementing
 * the brk pointer.  A block is pure payload. There are no headers or
 * footers.  Blocks are never coalesced or reused. Realloc is
 * implemented directly using mm_malloc and mm_free.
 *
 * NOTE TO STUDENTS: Replace this header comment with your own header
 * comment that gives a high level description of your solution.
 */


#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "mm.h"
#include "memlib.h"

/*
 * 定义两种内存分配算法，首次分配算法和循环首次分配算法；
 * 如果定义了NEXT_FIT,则采用循环首次适应算法，将下面的注释去掉即采用首次适应算法;
 */
//#define NEXT_FIT


/* single word (4) or double word (8) alignment */
#define ALIGNMENT 8
/* rounds up to the nearest multiple of ALIGNMENT */
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~(ALIGNMENT-1))
#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))

#define WORD      4       /* 内存字的大小，也代表一个内存块节点头部和尾部标识字段长度（4个字节）*/
#define DWORD      8       /* 双字大小（8个字节）*/
#define INCREMENTSIZE  (1<<12)  /* 每次将堆扩大INCREMENTSIZE所标示的字节数*/

```

```
46. #define MAX(x, y) ((x) > (y)? (x) : (y))
47.
48. /* 将size和allocated字段压缩成一个字*/
49. #define COMPRESS(size, alloc)  ((size) | (alloc))
50.
51. /* 从地址 p 所示的内存单元读取一个数据 */
52. #define READ(p) (*(unsigned int *)(p))
53.
54. /* 从地址 p 所示的内存单元写入一个数据 */
55. #define WRITE(p, val)  (*(unsigned int *)(p) = (val))
56.
57. /* 从地址 P 指定的内存单元读取size字段，以获取当前内存块的大小*/
58. #define GET_BLKSIZE(p)  (READ(p) & ~(ALIGNMENT-1))
59.
60. /* 从地址 P 指定的内存单元读取allocated字段，以判断内存块是否被占用*/
61. #define IS_ALLOCATED(p) (READ(p) & 0x1)
62.
63. /* 如果给定指向某一内存块的指针，则计算其头部字段的地址*/
64. #define CMPT_HEADPTR(bp)       ((char *)(bp) - WORD)
65.
66. /* 如果给定指向某一内存块的指针，则计算其尾部字段的地址*/
67. #define CMPT_FOOTPTR(bp)       ((char *)(bp) +
    GET_BLKSIZE(CMPT_HEADPTR(bp)) - DWORD)
68.
69. /* 如果给定指向内存块的指针，则计算下一个内存块的地址*/
70. #define GET_NEXT_BLKPTR(bp)  ((char *)(bp) + GET_BLKSIZE(((char
    *)(bp) - WORD)))
71.
72. /* 如果给定指向内存块的指针，则计算前一个内存块的地址*/
73. #define GET_PREV_BLKPTR(bp)  ((char *)(bp) - GET_BLKSIZE(((char
    *)(bp) - DWORD)))
74.
75. /* 全局变量 */
76. static char *g_pHeadList = 0; /* 指向第一个内存块的指针 */
77.
78.
79. #ifdef NEXT_FIT
80. static char *g_pNextFit; /* 循环首次适应算法的指针 g_pNextFit */
81. #endif
82.
83. team_t team = {
84. /* Team name */
85. "forhappy",
86. /* First member's full name */
```

```
87. "haipingf",
88. /* First member's email address */
89. "haipingf@gmail.com&&258347328@qq.com",
90. /* Second member's full name (leave blank if none) */
91. "",
92. /* Second member's email address (leave blank if none) */
93. "" }; /* The global condition sign */
94.
95. /* 自定义函数 */
96. static void *extend_heap(size_t words);
97. static void split_blk(void *bp, size_t adjustedsize);
98. static void *find_fit(size_t adjustedsize);
99. static void *combine(void *bp);
100.
101.    /*
102.     * mm_init - 初始化内存块;
103.     */
104.    int mm_init(void) {
105.        /* 创建初始的空堆; */
106.        if ((g_pHeadList = mem_sbrk(4 * WORD)) == (void *) -1)
107.            //最小的堆，大小为4*WORD=16字节，返回堆的首址;
108.            return -1;//如创建失败，则返回-1;
109.        WRITE(g_pHeadList, 0); /* 字段对准填充 */
110.        WRITE(g_pHeadList + (1*WORD), COMPRESS(DWORD, 1)); /* 第一个
    内存块的首部字段*/
111.        WRITE(g_pHeadList + (2*WORD), COMPRESS(DWORD, 1)); /* 第一个
    内存块的尾部字段*/
112.        WRITE(g_pHeadList + (3*WORD), COMPRESS(0, 1)); /* 最后一个内
    存块的首部字段*/
113.        g_pHeadList += (2 * WORD);
114.
115.    #ifdef NEXT_FIT
116.        g_pNextFit = g_pHeadList;
117.    #endif
118.
119.        /* 将空堆扩大至INCREMENTSIZE字节大小 */
120.        if (extend_heap(INCREMENTSIZE / WORD) == NULL)
121.            return -1;
122.        return 0;
123.    }
124.
125.    /*
126.     * mm_malloc - 分配一个大小由参数size所指定的内存块;
127.     */
```

```
128.
129.    void *mm_malloc(size_t size) {
130.        size_t adjustedsize;
131.        size_t extendsize; /* 如果存储空间不够，则开辟大小由extendsize所
    指示的新堆*/
132.        char *bp;
133.
134.        if (g_pHeadList == 0) {
135.            mm_init();
136.        }
137.
138.        /* 忽略用户请求不合法的数值*/
139.        if (size == 0)
140.            return NULL;
141.
142.        /* 调整块的大小，包括首尾字段开销和字节对齐开销*/
143.        if (size <= DWORD)
144.            adjustedsize = 2 * DWORD;
145.        else
146.            adjustedsize = DWORD * ((size + (DWORD) + (DWORD - 1)) /
    DWORD);
147.
148.        /* 搜索空闲链表以查找合适的位置*/
149.        if ((bp = find_fit(adjustedsize)) != NULL) {
150.            split_blk(bp, adjustedsize);
151.            return bp;
152.        }
153.
154.        /* 如果没有找到，则开辟新的堆来存放将要分配的块*/
155.        extendsize = MAX(adjustedsize,INCREMENTSIZE);
156.        if ((bp = extend_heap(extendsize / WORD)) == NULL)
157.            return NULL;
158.        split_blk(bp, adjustedsize);
159.        return bp;
160.    }
161.
162.    /*
163.     * mm_free - 释放内存块
164.     */
165.    void mm_free(void *bp) {
166.
167.        if (bp == 0)
168.            return;
169.
```

```
170.        size_t size = GET_BLKSIZE(CMPT_HEADPTR(bp));//找到头部位置;
171.
172.        if (g_pHeadList == 0) {
173.            mm_init();//如果堆本身为空，则该函数相当于mm_init();
174.        }
175.        WRITE(CMPT_HEADPTR(bp), COMPRESS(size, 0));
176.        WRITE(CMPT_FOOTPTR(bp), COMPRESS(size, 0));
177.        combine(bp);
178.    }
179.
180.    /*
181.     * combine - 边界标志算法的合并. 返回指向合并后的内存块指针
182.     */
183.
184.    static void *combine(void *bp) {
185.        size_t prev_alloc =
    IS_ALLOCATED(CMPT_FOOTPTR(GET_PREV_BLKPTR(bp)));//找到由bp所给定的内
    存块的前一内存块的尾部中的allocated字段;
186.        size_t next_alloc =
    IS_ALLOCATED(CMPT_HEADPTR(GET_NEXT_BLKPTR(bp)));//找到由bp所给定的内
    存块的后一内存块的首部中的allocated字段;
187.        size_t size = GET_BLKSIZE(CMPT_HEADPTR(bp));//获得bp所指定块
    的大小;
188.
189.        if (prev_alloc && next_alloc) { /* 情形一：左右内存块都被占用，
    则直接返回，不用任何操作 */
190.            return bp;
191.        }
192.
193.        else if (prev_alloc && !next_alloc) { /* 情形二：左边内存块被占
    用，而右边的内存块空闲，则把将要被释放的块与右边的块合并 */
194.            size += GET_BLKSIZE(CMPT_HEADPTR(GET_NEXT_BLKPTR(bp)));
195.            WRITE(CMPT_HEADPTR(bp), COMPRESS(size, 0));//更新首部中的
    size和callocated字段;
196.            WRITE(CMPT_FOOTPTR(bp), COMPRESS(size,0));//更新尾部中的
    size和callocated字段;
197.        }
198.
199.        else if (!prev_alloc && next_alloc) { /* 情形三：右边内存块被占
    用，而左边的内存块空闲，则把将要被释放的块与右边的块合并 */
200.            size +=
    GET_BLKSIZE(CMPT_HEADPTR(GET_PREV_BLKPTR(bp)));//size被更新;
201.            WRITE(CMPT_FOOTPTR(bp), COMPRESS(size, 0));//更新尾部中的
    size和callocated字段;
```

```
202.            WRITE(CMPT_HEADPTR(GET_PREV_BLKPTR(bp)), COMPRESS(size,
    0));//更新首部中的size和callocated字段；
203.            bp = GET_PREV_BLKPTR(bp);//bp指向左部内存块；
204.        }
205.
206.        else { /*情形四：左右内存块均为空闲，则把将要被释放的块与左右内存块合
    并 */
207.            size += GET_BLKSIZE(CMPT_HEADPTR(GET_PREV_BLKPTR(bp))) +
    GET_BLKSIZE(CMPT_FOOTPTR(GET_NEXT_BLKPTR(bp)));
208.            WRITE(CMPT_HEADPTR(GET_PREV_BLKPTR(bp)), COMPRESS(size,
    0));//更新首部中的size和callocated字段；
209.            WRITE(CMPT_FOOTPTR(GET_NEXT_BLKPTR(bp)), COMPRESS(size,
    0));//更新尾部中的size和callocated字段；
210.            bp = GET_PREV_BLKPTR(bp);//bp指向左部内存块；
211.        }
212.
213.    #ifdef NEXT_FIT
214.        /* 确保rover没有指向刚刚合并的空闲块*/
215.        if ((g_pNextFit > (char *)bp) && (g_pNextFit <
    GET_NEXT_BLKPTR(bp)))
216.        g_pNextFit = bp;
217.    #endif
218.        return bp;//成功返回；
219.    }
220.
221.    /*
222.     * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
223.     */
224.    void *mm_realloc(void *ptr, size_t size) {
225.        void *oldptr = ptr;
226.        void *newptr;
227.        size_t copySize;
228.
229.        newptr = mm_malloc(size);
230.        if (newptr == NULL)
231.            return NULL;
232.        copySize = *(size_t *) ((char *) oldptr - SIZE_T_SIZE);
233.        if (size < copySize)
234.            copySize = size;
235.        memcpy(newptr, oldptr, copySize);
236.        mm_free(oldptr);
237.        return newptr;
238.    }
239.
```

```
240.    /*
241.     * checkheap - We don't check anything right now.
242.     */
243.    void mm_checkheap(int verbose) {
244.    }
245.
246.    /*
247.     * extend_heap - 扩大堆的空间;
248.     */
249.
250.    static void *extend_heap(size_t words) {
251.        char *bp;
252.        size_t size;
253.
254.        /* 为了保持字节对齐，应分配偶数倍的字大小 */
255.        size = (words % 2) ? (words + 1) * WORD : words * WORD;
256.        if ((long) (bp = mem_sbrk(size)) == -1)
257.            return NULL;
258.
259.        /* 初始化该空闲块的首部/尾部以及最后一个块的首部*/
260.        WRITE(CMPT_HEADPTR(bp), COMPRESS(size, 0)); /* //设置空闲块首部的size（置为0，表示空闲）和allocated的字段； */
261.        WRITE(CMPT_FOOTPTR(bp), COMPRESS(size, 0)); //设置空闲块尾部的size（置为0，表示空闲）和allocated的字段;
262.        WRITE(CMPT_HEADPTR(GET_NEXT_BLKPTR(bp)), COMPRESS(0, 1)); /* 新的最后一个块的首部 */
263.
264.        /* 如果前一个块为空闲的，则合并两个空闲块*/
265.        return combine(bp);
266.    }
267.
268.    /*
269.     * split_blk -  分配一个空闲块
270.     */
271.    static void split_blk(void *bp, size_t adjustedsize) {
272.        size_t csize = GET_BLKSIZE(CMPT_HEADPTR(bp));//获取bp块的大小;
273.
274.        if ((csize - adjustedsize) >= (2 * DWORD)) {//如果块足够大;
275.            WRITE(CMPT_HEADPTR(bp), COMPRESS(adjustedsize, 1));//更新bp块首部的size（置为1，表示被占用）和allocated的字段;
276.            WRITE(CMPT_FOOTPTR(bp), COMPRESS(adjustedsize, 1));//更新bp块尾部的size（置为1，表示被占用）和allocated的字段;
277.            bp = GET_NEXT_BLKPTR(bp);//分配当前的块
278.            WRITE(CMPT_HEADPTR(bp), COMPRESS(csize-adjustedsize,
```

```
    0));//更新剩余块首部的size（置为0，表示空闲）和allocated的字段；
279.            WRITE(CMPT_FOOTPTR(bp), COMPRESS(csize-adjustedsize,
    0));//更新剩余块尾部的size（置为0，表示空闲）和allocated的字段；
280.        } else {//否则，将整个块分配出去；
281.            WRITE(CMPT_HEADPTR(bp), COMPRESS(csize, 1));//将首部的
    allocated字段置为1，表示被占用；
282.            WRITE(CMPT_FOOTPTR(bp), COMPRESS(csize, 1));//将尾部的
    allocated字段置为1，表示被占用；
283.        }
284.    }
285.
286.    /*
287.     * find_fit - 查找具有adjustedsize字节的内存块
288.     */
289.
290.    static void *find_fit(size_t adjustedsize)
291.
292.    {
293.
294.    #ifdef NEXT_FIT
295.        /* 循环首次适应算法 */
296.        char *oldrover = g_pNextFit;
297.
298.        /* 从g_pNextFit所示的地方一直搜索的链表的最末尾*/
299.        for (; GET_BLKSIZE(CMPT_HEADPTR(g_pNextFit)) > 0; g_pNextFit
    = GET_NEXT_BLKPTR(g_pNextFit))
300.        if (!IS_ALLOCATED(CMPT_HEADPTR(g_pNextFit)) && (adjustedsize
    <= GET_BLKSIZE(CMPT_HEADPTR(g_pNextFit))))
301.        return g_pNextFit;
302.
303.        /* 从链表头开始搜索一直到g_pNextFit */
304.        for (g_pNextFit = g_pHeadList; g_pNextFit < oldrover;
    g_pNextFit = GET_NEXT_BLKPTR(g_pNextFit))
305.        if (!IS_ALLOCATED(CMPT_HEADPTR(g_pNextFit)) && (adjustedsize
    <= GET_BLKSIZE(CMPT_HEADPTR(g_pNextFit))))
306.        return g_pNextFit;
307.
308.        return NULL; /* 没有找到 */
309.    #else
310.        /* 首次适应算法 */
311.        void *bp;
312.
313.        for (bp = g_pHeadList; GET_BLKSIZE(CMPT_HEADPTR(bp)) > 0; bp
    = GET_NEXT_BLKPTR(bp)) {
```

```
314.          if (!IS_ALLOCATED(CMPT_HEADPTR(bp)) && (adjustedsize <=
   GET_BLKSIZE(CMPT_HEADPTR(bp)))) {
315.              return bp;
316.          }
317.      }
318.      return NULL; /*没有找到 */
319.  #endif
320.  }
321.
```