哈爾濱工業大學

实验报告

实 验（八）

题　　目 Dynamic Storage Allocator

动态内存分配器

专　　业 计算机科学与技术

学　　号 L170300601

班　　级 1703006

学 生 姓 名 高耶临

指 导 教 师 吴锐

实 验 地 点 G712

实 验 日 期 2018/12/16

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

理解现代计算机系统虚拟存储的基本知识
掌握 C 语言指针相关的基本操作
深入理解动态存储申请、释放的基本原理和相关系统函数
用 C 语言实现动态存储分配器，并进行测试分析
培养 Linux 下的软件系统开发与测试能力

1.2 实验环境与工具

1.2.1 硬件环境

X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

Windows7 64 位以上； VirtualBox/Vmware 11 以上； Ubuntu 16.04 LTS 64 位/优麒麟 64 位

1.2.3 开发工具

Gcc, profiler，工具 gprof

1.3 实验预习

第 2 章 实验预习

总分 20 分

2.1 动态内存分配器的基本原理（5 分）

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长。对于每个进程，内核维护着一个变量 brk 它指向堆的顶部。

分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可以用来分配。空闲块保持空闲，直到它显示地被应用分配。一个已分配的块保持已分配状态，直到它被释放。

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

在这种情况中，一个块是由一个字的头部、有效载荷，以及可能的一些额外的填充组成的。头部编码了这个块的大小，以及这个快是已分配的还是空闲的。头部后面就是调用 malloc 时请求的有效载荷。有效载荷后面时一片不使用的填充块，其大小是任意的。利用了系统对齐要求和分配器对块格式的选择会对分配器上的最小块有强制要求比如是双字对齐块大小就是 8 的整数倍，底三位就可以作为标志。

隐式空闲链表的空闲块是通过头部中的大小字段隐含地连接着的。分配器通过遍历堆中的所有块，从而间接地遍历整个空闲块的集合。

2.3 显示空间链表的基本原理（5 分）

根据定义，程序不需要一个空闲块的主体，所以实现这个数据结构的指针可以存放在这些空闲块的主体里面。例如，堆可以组织成一个双向空闲链表，在每个空闲块中都包含一个前驱和后继。使用双向链表不是隐式空闲链表，使首次适配的分配时间从块的总数的线性时间减少到了空闲块数量的线性时间。不过，放置一个块的时间可以是线性的，也可能是个常数，这取决于我们选择的空闲链表中块的排序策略。

2.4 红黑树的结构、查找、更新算法（5 分）

红黑树的定义是满足下列条件的二叉查找树：

⑴红链接均为左链接。

⑵没有任何一个结点同时和两条红链接相连。

⑶该树是完美黑色平衡的，即任意空链接到根结点的路径上的黑链接数量相同。

查找：

```
RBNode* rb_search(RBNode* node, int value) {
  while(node != NULL) {
    if(value < node->value) // 向左
      node = node->left;
    else if(value > node->value) // 向右
      node = node->right;
    else // 找到
      return node;
  }
  return NULL; // 失败
}
```

插入：

```
  #include<iostream>
#include<stack>
using namespace std;

#define RED 0
#define BLACK 1

struct RBNode {
  RBNode* left;
  RBNode* right;
  RBNode* parent;
  int color;
  int value;
};

// 左左情况右旋，返回子树旋转后的根
RBNode* LL(RBNode* A) {
  assert(NULL != A);
  RBNode* B = A->left;
  A->left = B->right;
  B->right = A;
  A->parent = B;
  return B;
}
// 右右情况左旋，返回子树旋转后的根
RBNode* RR(RBNode* A) {
  assert(NULL != A);
  RBNode* B = A->right;
  A->right = B->left;
  B->left = A;
```

```
  A->parent = B;
  return B;
}

bool rb_insert(RBNode*& root, int value) { // root 要求是指针的引用或者指针的指针
  RBNode *p, *pre, *tmp;
  tmp = root;
  // 寻找插入位置，并且保证路径
  pre = NULL;
  while(tmp != NULL) {
    pre = tmp;
    if(value < tmp->value)
      tmp = tmp->left;
    else if(value > tmp->value)
      tmp = tmp->right;
    else // 插入识别
      return false;
  }
  // 建立新结点
  p = new RBNode;
  p->left = p->right = NULL;
  p->color = RED;
  p->value = value;
  // 插入新结点
  if(pre == NULL) { // 根结点为空
    p->parent = NULL;
    p->color = BLACK;
    root = p;
    return true;
  }
  // 根结点不为空
  p->value<pre->value ? (pre->left=p):(pre->right=p);
  p->parent = pre;
  // 五种情况
  while(NULL != p) {
    if(p->parent == NULL) { // 情况一：P 为根结点
      p->color = BLACK;
      return true;
    }
    else if(p->parent->color == BLACK) { // 情况二：P 的父结点是黑色的
      return true;
    }
    else { // P 的父结点是红色
      RBNode* father = p->parent;
      RBNode* grandfather = p->parent->parent;
      RBNode* uncle = father->value<grandfather->value?grandfather->right:grandfather->left;
      if(uncle!=NULL && uncle->color==RED) { // 情况三：P 的父结点和叔父结点都是红色的
        grandfather->color = RED;
        father->color = BLACK;
        uncle->color = BLACK;
        p = grandfather; // 进入循环
      }
      else if( (father->value<grandfather->value && p->value>father->value)
        || (father->value>grandfather->value && p->value<father->value) ) {
        // 情况四：p 与其祖父结点关系为 LR 型或者 RL 型
        if(p->value>father->value) { // LR，左旋
          tmp = RR(father);
```

```
88.           grandfather->left = tmp;
89.           tmp->parent = grandfather;
90.         }
91.         else { // RL，右旋
92.           tmp = LL(father);
93.           grandfather->right = tmp;
94.           tmp->parent = grandfather;
95.         }
96.         // 进入循环，此时 p 与其祖父结点关系为 LL 或者 RR 型，肯定会进入情况五
97.       }
98.       else { // 情况五：p 与其祖父结点关系为 LL 型或者 RR 型
99.         grandfather->color = RED;
100.         father->color = BLACK;
101.         RBNode* tmp_father = grandfather->parent;
102.         if(p->value<father->value) { // LL
103.           tmp = LL(grandfather);
104.           tmp->parent = tmp_father;
105.         }
106.         else { // RR
107.           tmp = RR(grandfather);
108.           tmp->parent = tmp_father;
109.         }
110.         if(tmp_father != NULL)
111.           tmp->value<tmp_father->value?tmp_father->left=tmp:tmp_father->right
    =tmp;
112.         else
113.           root = tmp;
114.         return true;
115.       }
116.     }
117.   }
118. }
119.
120. void print_father_child(RBNode* root) {
121.   stack<RBNode*> store;
122.   store.push(root);
123.   while(!store.empty()) {
124.     root = store.top();
125.     store.pop();
126.     cout << root->value << " : ";
127.     if(root->color == RED)
128.       cout << "  RED, ";
129.     else
130.       cout << "  BLACK, ";
131.     if(root->left==NULL)
132.       cout << "NULL";
133.     else
134.       cout << root->left->value;
135.     cout << " , ";
136.     if(root->right==NULL)
137.       cout << "NULL";
138.     else {
139.       cout << root->right->value;
140.     }
141.     cout << endl;
142.     if(root->right)
143.       store.push(root->right);
144.     if(root->left)
145.       store.push(root->left);
146.   }
147. }
```

```
148.
149. int main() {
150.   RBNode* root = NULL;
151.   for(int i=0; i<10; i++)
152.     rb_insert(root, i);
153.   print_father_child(root);
154.   system("PAUSE");
155.   return 0;
156. }
```

第 3 章 分配器的设计与实现

总分 50 分

3.1 总体设计（10 分）

介绍堆、堆中内存块的组织结构，采用的空闲块、分配块链表/树结构和相应算法等内容。

1. 堆的整体结构：

18 个空闲链表头	填充块	序言块	序言块	中间都是普通块	结尾块

2. 普通内存块的结构

Head
Next
Prev
有效载荷
填充
Foot

在这种情况中，一个块是由一个字的头部、有效载荷，以及可能的一些额外的填充组成的。头部编码了这个块的大小，以及这个快是已分配的还是空闲的。头部后面就是调用 malloc 时请求的有效载荷。有效载荷后面时一片不使用的填充块，其大小是任意的。利用了系统对齐要求和分配器对块格式的选择会对分配器上的最小块有强制要求比如是双字对齐块大小就是 8 的整数倍，底三位就可以作为标志。

3. 策略：采用了隐式空闲链表+首次适配

重要函数的整体流程：

(1)int mm_init(void)

包括调用了 mem_sbrk -> void *extend_heap(size_t words) -> void *coalesce(void *bp)包括调用了 void insert_list(void *bp)和 void delete_list(void *bp)

(2) void mm_free(void *ptr)

设置 head 和 foot -> void *coalesce(void *bp)

(3)void *mm_malloc(size_t size)

至少申请 2DSIZE 堆空间 -> void *find_fit(size_t asize)和 void place(void *bp, size_t asize)

或者 -> void *extend_heap(size_t words)之后 void place(void *bp, size_t asize)

(4)void *mm_realloc(void *ptr, size_t size)
调用 mm_realloc 是为了将 ptr 所指向内存块（旧块)的大小变为 size，并返回新内存块的地址。
(5) **static void *coalesce(void *bp)**
按照书中的四种情况合并块，当前块和前后合并而所谓合并，就是修改上一个或者下一个块的头和尾中的 size。使之变成合并后的 size。

3.2 关键函数设计（40 分）

3.2.1 int mm_init(void)函数（5 分）

函数功能：初始化堆栈
处理流程：
首先调用 mem_sbrk 申请 4 个字的空间，首先第一个字填充为 0。接着创建 prologue block，因为 prologue block 是双字，也就是 8-byte。接着创建 epilogue block。接着将 head_listp 指针指向 prologue block 的结尾。
要点分析：
隐式空闲链表第一个字并没有用，这作为双字边界对齐的填充字。接着是 prologue block，这和正常块的区别在于它没有 payload，所以算是作为堆开始的标志，同样，epilogue block 放在堆的结尾处。。虽然隐式空闲链表的头和尾都构建好了，但是没有分配空间给它，它限制并不能存任何东西。所以我们要对刚才的空闲链表进行扩展堆。默认扩展一个空堆 CHUNKSIZE，这里定义的是 1<<12 也就是 4K，4K 也就是 1000 个字。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放块
参　　数：释放参数“ptr”指向的已分配内存块，没有返回值。指针值 ptr 应该是之前调用 mm_malloc 或 mm_realloc 返回的值，并且没有释放过。
处理流程：
应用通过调用 mm_free 函数，来释放一个以前分配的块，这个很熟释放所请求的块，然后使用边界合并技术将之与邻接的空闲块合并起来。
要点分析：其实就是将头和尾的标志位修改为 0，然后将这个块与前一个块或者后一个块合并。

3.2.3 void *mm_realloc(void *ptr, size_t size)函数（5 分）

函数功能：调用 mm_realloc 是为了将 ptr 所指向内存块（旧块)的大小变为 size，并返回新内存块的地址。

参　　数：需要处理的块指针 ptr,新块大小 size
处理流程：如 ptr 是空指针 NULL,等价于 mm_malloc(size),如果参数 size 为 0，等价于 mm_free(ptr),如 ptr 非空, 它应该是之前调用 mm_malloc 或 mm_realloc 返回的数值，指向一个已分配的内存块。
要点分析：
直接用 malloc 和 free 组合实现，其中 size 为 0，相当于 free，ptr 为 NULL，相当于 malloc，memmove 实现旧数据移动，包括缩小空间和扩大空间两种情况。注意：返回的地址与原地址可能相同，也可能不同，这依赖于算法的实现、旧块内部碎片大小、参数 size 的数值。新内存块中，前 min(旧块 size，新块 size)个字节的内容与旧块相同，其他字节未做初始化。

3.2.4 int mm_check(void)函数（5 分）

函数功能：检查重要的不变量和一致性条件
处理流程：依此检查堆中各个块，包括空闲链表，分配的块，当且仅当堆是一致的，才能返回非 0 值
要点分析：
需要关注的方面：空闲列表中的每个块是否都标识为 free（空闲），是否有连续的空闲块没有被合并，是否每个空闲块都在空闲链表中，空闲链表中的指针是否均指向有效的空闲块，分配的块是否有重叠，堆块中的指针是否指向有效的堆地址

3.2.5 void *mm_malloc(size_t size)函数（10 分）

函数功能：最主要的内存分配部分
参　　数：向内存申请块的大小 size
处理流程：1.对齐 2.最适搜索(这里是用首次适配搜索。从隐式空闲链表的开头开始，寻找首个合适的未分配的块) 3.扩展堆，在扩展完块以后，需要将这个块和之前的块进行合并。如果之前的块为空闲，则合并成更大的块来满足空间申请的请求。
要点分析：在检查完请求的真假之后，分配器必须调整请求块的大小，从而头部和脚部都留有空间，满足双字对齐的要求。这里强制了最小块的大小试 16 字节：8 字节用来满足对齐要求，另外 8 字节用来放头部和尾部。对于超过 8 字节的请求，加上开销字节，向上舍入到 8 的整数倍。之后搜索空闲链表寻找合适的空闲块，如果合适，放置这个请求块，并且可选地割出多余的部分，然后返回新分配块的地址。 如果不能发现匹配块，就割一个新的块来扩展堆，把这个块放置在新的空闲块里面，可选地分割这个块。

3.2.6 static void *coalesce(void *bp)函数（10 分）

函数功能：合并块

处理流程：合并块有 4 种可能的情况，情况 1：前后都已分配。此时不可能合并，就是单独从已分配编程空闲。情况 2：前面分配后面没有分配。那么当前块与后面合并。情况 3：前面空闲，后面已经分配。当前块和前面合并。情况 4：前面后面都已经分配。当前块和前后合并而所谓合并，就是修改上一个或者下一个块的头和尾中的 size。使之变成合并后的 size。

要点分析：按照书中图 9-40 中勾画的四种情况分别合并。但是这里有注意，空闲链表格式允许忽略潜在的边界情况，请求块的堆的起始处或者是在堆的结尾处。所以每次释放请求时，需要检查这些不常见的边界情况。

第 4 章测试

总分 10 分

4.1 测试方法

利用轨迹文件执行./mdriver [-hvVa] [-f <file>]

4.2 测试结果评价

```
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ make
gcc -Wall -O2 -m32   -c -o mm.o mm.c
gcc -Wall -O2 -m32 -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ft
imer.o
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -v -t traces
/
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.033889   168
 1       yes   99%    5848  0.027201   215
 2       yes   99%    6648  0.066738   100
 3       yes  100%    5380  0.054059   100
 4       yes   66%   14400  0.000494 29168
 5       yes   92%    4800  0.047625   101
 6       yes   92%    4800  0.045372   106
 7       yes   55%   12000  0.242111    50
 8       yes   51%   24000  1.170134    21
 9       yes   27%   14401  0.817246    18
10       yes   30%   14401  0.007525  1914
Total          74%  112372  2.512393    45

Perf index = 44 (util) + 3 (thru) = 47/100
```

```
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f short1-ba
l.rep
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Perf index = 40 (util) + 40 (thru) = 80/100
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f short2-ba
l.rep
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Perf index = 54 (util) + 40 (thru) = 94/100
```

结果的峰值利用率挺高，但是吞吐量很小，因为隐式空闲链表每次 malloc 的时候需要从头遍历所有的块，以及立即合并可能会产生抖动。而且因为块分配与堆块的总数呈线性关系，所以对于通用的分配器，隐式空闲链表时不合适的，。但是用另外两个测试文件 short1-bal.rep 和 short2.bal.rep 测试结果已经较好了，说明对于预先就知道堆块数量是很小的特殊分配器来说时合适的。但是总统还是很多需要优化的地方。

4.3 自测试结果

```
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f amptjp-ba
l.rep
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Perf index = 60 (util) + 11 (thru) = 71/100
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f binary2-b
al.rep
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Perf index = 31 (util) + 2 (thru) = 32/100
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f cccp-bal.
rep
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Perf index = 60 (util) + 13 (thru) = 73/100
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f coalescin
g-bal.rep
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Perf index = 40 (util) + 40 (thru) = 80/100
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f cp-decl-b
al.rep
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Perf index = 60 (util) + 6 (thru) = 66/100
```

```
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f expr-b
rep
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Perf index = 60 (util) + 7 (thru) = 67/100
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f randon
al.rep
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Perf index = 55 (util) + 7 (thru) = 62/100
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f randon
l_rep
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Could not open ./random-bal in read_trace: No such file or directory
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f randon
l.rep
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Perf index = 55 (util) + 6 (thru) = 61/100
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ make
make: 'mdriver' is up to date.
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f binary
l.rep
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Perf index = 33 (util) + 3 (thru) = 36/100
```

```
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f realloc
al.re
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Could not open ./realloc-bal.re in read_trace: No such file or directory
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -f realloc
bal.rep
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Perf index = 18 (util) + 40 (thru) = 58/100
```

```
l170300601@ubuntu:~/malloclab-handout/malloclab-handout$ ./mdriver -t traces
Team Name:ateam
Member 1 :gaoyelin:bovik@cs.cmu.edu
Using default tracefiles in traces/
Perf index = 44 (util) + 3 (thru) = 47/100
```

第 5 章 总结

5.1 请总结本次实验的收获

通过本次实验理解现代计算机系统虚拟存储的基本知识，深入理解动态存储申请、释放的基本原理和相关系统函数，用 C 语言实现动态存储分配器，并进行测试分析而且这个实验总体来说还是很难的，不过多亏了书上已经很详细了。

5.2 请给出对本次实验内容的建议

PPT 给出了要利用优化的要求 1.显示空闲链表 + 基于边界标签的空闲块合并 + 首次适配 2：使用红黑树（最优的方法）但是又给出了不能修改 mm.c 中的接口函数（函数声明不能改），不可以定义任何全局或静态的复合数据结构，例如：数据、结构体、树或链表的要求，没有说清楚，自相矛盾。

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.