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 计算机类

学　　号 1190201215

班　　级 1903007

学 生 姓 名 冯开来

指 导 教 师 吴锐

实 验 地 点 G709

实 验 日 期 2021.6.7

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

Visual Studio 2010 64 位以上；TestStudio；Gprof;Valgrind 等

1.3 实验预习

- ✓ 上实验课前，必须认真预习实验指导书（PPT 或 PDF）
- ✓ 了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。
- ✓ 熟知 C 语言指针的概念、原理和使用方法
- ✓ 了解虚拟存储的基本原理
- ✓ 熟知动态内存申请、释放的方法和相关函数
- ✓ 熟知动态内存申请的内部实现机制：分配算法、释放合并算法等

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者一共进程的虚拟内存区域，称为堆。系统之间细节不同，但是不失通用性，假设堆是一共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

分配器将堆是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篇，要么已是分配的，要么是空闲的。已分配的块显式地保留位供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配地块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器有两种风格。两种风格都要求应用显式地分配块。它们的不同之处在于由哪个实体来负责释放已分配的块。

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

在这种情况下，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（包括头部和所有的填充），以及这个块是已分配的还是空闲的。如果我们强加一个双字的对齐约束条件，那么块大小就总是 8 的倍数，且块大小的最低 3 位总是零。因此，我们只需要内存大小的 29 个高位，释放剩余的 3 位来编码其他的信息。在这种情况下，我们用其中的最低位（已分配位）来指明这个块是已分配的还是空闲的。

头部后面是应用调用 malloc 时请求的有效载荷。有效载荷后面是一片不使用的填充块，其大小可以是任意的。需要填充有很多原因。比如，填充可能是分配器策略的一部分，用来对付外部碎片。或者也需要用它来满足对齐要求。

空闲块是通过头部中的大小字段隐含地连接着的。分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合。注意，我们需要某种特殊标记的结束块，在这个示例中，就是一个设置了已分配位而大小为零的终止头部。

2.3 显示空间链表的基本原理（5 分）

程序不需要一个空闲的块的主体，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 pred 和 succ 指针。

使用双向链表使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。不过，释放一个块的时间可以是线性的，也可能是一个常数，这取决于空闲链表中块的排序策略。

一种方法是用后进先出的顺序维护链表，将新释放的块放置在链表的开始处。另一种方法是按照地址顺序来维护链表，其中链表中每个块的地址都小于它后继的地址。

2.4 红黑树的结构、查找、更新算法（5 分）

结构：

红黑树本质是一颗二叉搜索树，它满足二叉搜索树的基本性质——即树中的任何节点的值大于它的左子节点，且小于它的右子节点。

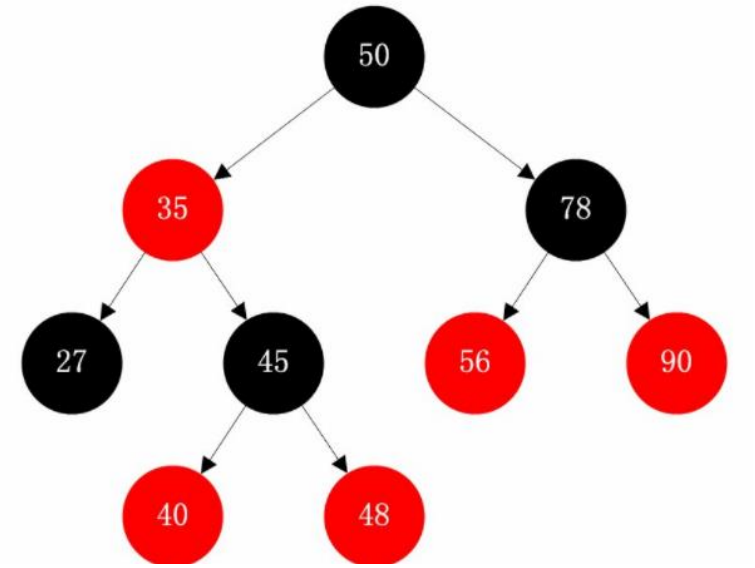
一棵红黑树必须满足以下几点条件：

规则 1：根节点必须是黑色。

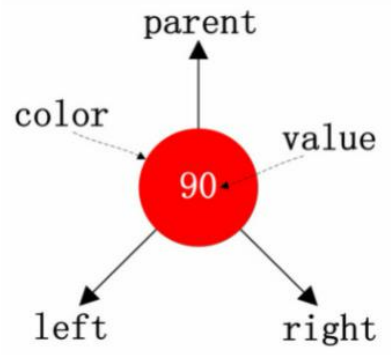
规则 2：任意从根到叶子的路径不包含连续的红色节点

规则 3：任意从根到叶子的路径的黑色节点总数相同。

如图，就是一棵合法的红黑树，构造如下红黑树的数据序列为：（50，35，78，27，56，90，45，40，48）。



作为红黑树节点，基本属性有：节点的颜色，左子节点指针，右子节点指针，父节点指针，节点的值。



查找：

红黑树的 get()方法不会检查节点的颜色，因此实现和二叉查找树一样，但由于树是平衡的，所以查找比二叉查找树更快。

```
public Value get(Key, key){
    return get(root, key);
}
private Value get(Node x, Key key) {
    while (x != null) {
        int cmp = key.compareTo(x.key);
        if (cmp < 0)
            x = x.left;
        else if (cmp > 0)
            x = x.right;
        else
            return x.val;
    }
    return null;
}
```

更新：

删除操作：

在查找路径上进行和删除最小键相同的变换同样可以保证在查找过程中任意当前节点均不是 2-节点。如果被查找的键在树的底部，我们可以直接删除它。如果不在，我们需要将它和它的后继节点交换，就和二叉查找树一样。因为当前节点必然不是 2-节点，问题已经转化为在一棵根节点不是 2-节点的子树中删除最小的键，可以直接使用上文的算法。删除之后我们需要向上回溯并分解余下的 4-节点。

新增操作：

先将一个节点按二叉查找树的方法插入到正确位置，然后在沿着插入点到根节点的路径向上移动时，在所经过的每个节点中顺序完成如下操作：

1. 如果右子节点是红色的而左子节点是黑色的，进行左旋转；
2. 如果左子节点是红色的，而且左子节点的左子节点也是红色的，进行右旋转；
3. 如果左右子节点均为红色的，进行颜色转换，将红链接在树中向上传递。

颜色转换会使根节点变为红色，但根节点并没有父节点，依次在每次插入后都将根节点设为黑色。注意，每当根节点由红变黑时树的黑链接高度就会加一，因为这意味着它由一个 4-节点分裂出去成为 2-节点了。

```
public void put(Key key, Value val) {
    //查找 key，找到则更新其值，否则为它新建一个节点
     root = put(root, key, val);
     root.color = BLACK;
}

private Node put(Node h, Key key, Value val) {
     if (h == null)  //标准的插入操作，和父节点用红链接相连
         return new Node(key, val, RED, 1);

     int cmp = key.compareTo(h.key);
     if          (cmp < 0) h.left  = put(h.left,  key, val);
     else if (cmp > 0) h.right = put(h.right, key, val);
     else                      h.val   = val;

     if (isRed(h.right) && !isRed(h.left))          h = rotateLeft(h);
     if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
     if (isRed(h.left)  &&  isRed(h.right))       flipColors(h);
     h.size = size(h.left) + size(h.right) + 1;

     return h;
}
```

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：

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。系统之间细节不同，但是不失通用性，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。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

2.堆中内存块的组织结构：

一个块是由一个字的头部、有效载荷、可能的一些额外的填充，以及在块的结尾处的一个字的脚部组成的。头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。如果我们强加一个双字的对齐约束条件，那么块大小就总是 8 的倍数，且块大小的最低 3 位总是 0。因此，我们只需要内存大小的 29 个高位，释放剩余的 3 位来编码其他信息。在这种情况中，我们用其中的最低位（已分配位）来指明这个块是已分配的还是空闲的。

3.采用的空闲块、分配块链表：

使用隐式的空闲链表，使用立即边界标记合并方式，最大的块为2^{32} =4GB，代码是64位干净的，即代码能不加修改地运行在32位或64位的进程中。

4.相关算法：

int mm_init(void)函数；
void mm_free(void *ptr)函数；
void *mm_realloc(void *ptr, size_t size)函数；
int mm_check(void)函数；
void *mm_malloc(size_t size)函数。

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：创建一个带初始空闲块的堆
处理流程：
1. 初始化分离空闲链表，将每个链表设置为 NULL
2. 设置开始 Block，结束 Block：申请堆空间，设置开始 Block 的 Header，Footer，设置结束 Block 的 Footer。
3. 拓展初始大小：拓展堆空间，拓展大小为 INITIALSIZE（48B）。
4. mm_check：堆的一致性检查。

要点分析：
1. 注意初始化分离空闲链表和初始化堆要知道分配器使用最小块的大小为 16 字节。创建空闲链表之后需要使用 extend_heap 来扩展堆。
2. 为保持对齐，extend_heap 将请求最接近 8 字节的倍数（向上舍入），然后向内存系统请求额外的堆空间。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放所请求的块
参　　数：void *ptr 代表指向需要释放的 block 有效负载的指针。
处理流程：
1. 设置隐式链表信息：将 block 的 HDR 和 FTR 都设置为空闲状态(size,0)。
2. 调用 GET_SIZE(HDRP(ptr))来获得请求块的大小。
3. 插入显示空闲链表：调用 insert_node 函数，将 ptr 指向的 block 插入到分 离空闲链表之中。
4. 调用 coalesce(bp);将释放的块 bp 与相邻的空闲块合并起来。
5. 检查堆的一致性。

要点分析：
1. 在释放分配块之后则该 block 已经空闲下来了，需要将该 block 加入 到显式分离空闲链表之中，添加之后因为位于堆之中，地址前后的块可能存在空闲块。
2. 注意 free 块需要和与之相邻的空闲块使用边界标记合并。

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

函数功能：给 ptr 所指向的块分配一个 size 大小的有效负载块
参　　数：void *ptr 代表指向需要释放的块的指针；size_t size 代表新的块的大小。
处理流程：
1. 首先检查请求的真假，在检查完请求的真假后，分配器必须调整请求块的大小。从而为头部和脚部留有空间，并满足双字对齐的要求。
2. 将 pre_size 对齐：如果 pre_size<=DSIZE，则手动对齐，否则调用 ALIGN 函数进行对齐至 8B 的倍数。

3. 执行 copySize = GET_SIZE(HDRP(ptr))，得到块的大小，比较 pre_size 和 size，判断是否需要拓展。
4. 如果 size 比 new_size 小，更新 pre_size 为 size，释放 ptr。
5. 堆的一致性检查

要点分析：

判断第一个块：使第一个 Block 位于堆空间的最后，这样的话，我们可以直接拓展的堆空间就正好位于第一个 Block 之后，拓展堆空间之后直接改变 Block 的大小即可完成空间拓展。

3.2.4 int mm_check(void)函数（5 分）

函数功能：堆的一致性检查

处理流程：

1. 定义指针 bp，初始化为指向堆地址开始的指针 heap_start 和获取指向所有链表的指针 segregated_free_lists。如果开始的块不是 8 字节已分配块，输出错误
2. 扫描所有的显式分离空闲链表，对所有链表，进行遍历，对于每个节点， 获取前一个和后一个 block，1.检查是否有连续的空闲块没有被合并。
3. 检查结尾块。当结尾块不是一个大小为 0 的已分配块，输出错误。

要点分析：

1. 空闲链表中的指针是否均指向有效的空闲块
2. 是否能够完整遍历整个链表来检查是否成立

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

函数功能：请求分配 size 大小的块

参　　数：size_t size

处理流程：

1. 首先检查请求的真假，在检查完请求的真假后，分配器必须调整请求块的大小。从而为头部和脚部留有空间，并满足双字对齐的要求。
2. 寻找合适的链表：通过要求的块的大小，在分离空闲链表之中进行查找，查 找到包含块大小的链表。
3. 分割出多余的部分，然后返回新分配的地址。

要点分析：

1. 链表代表的块空间的大小，所以有巧妙的索引方式：对 searchsize 进行循环/2，最终<=1 时就找到了正确链表。
2. mm_malloc 函数是为了更新 size 来满足要求的大小，然后在分离空闲链表数组里面找到合适的请求块，找不到的话就使用一个新的空闲块来扩展堆。

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

函数功能：使用边界标记合并将与之相邻的空闲块合并

处理流程：

1. 若前后均为 allocated，直接返回
2. 前 allocated，后 free，删除后面的块和当前的块，将两个 free 合并
3. 前 free，后 allocated，删除前面的块和当前的块，将两个 free 合并
4. 前后都是 free，删除前后当前三个块，将三个 free 合并
5. 更新的 bp 所指向的块插入分离空闲链表

要点分析：

1. 检查边界情况
2. 合并 free 时：合并空闲块成为一个大的空闲块时改变天猫的 header 和后面的 footer
3. 删除原有空闲链表节点：删除节点后合成空闲块。

第 4 章测试

总分 10 分

4.1 测试方法与测试结果

生成可执行评测程序文件的方法
make
评测方法：
./mdriver -av -t traces

```
fkl1190201215@ubuntu:~$ ./mdriver -av -t traces
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   98%    5694  0.000264 21560
 1       yes   97%    5848  0.000279 20968
 2       yes   99%    6648  0.000332 20000
 3       yes   99%    5380  0.000231 23341
 4       yes   99%   14400  0.000416 34607
 5       yes   93%    4800  0.000368 13054
 6       yes   90%    4800  0.000384 12493
 7       yes   55%   12000  0.000347 34542
 8       yes   51%   24000  0.001019 23553
 9       yes   99%   14401  0.000200 72005
10       yes   86%   14401  0.000271 53121
Total           88%  112372  0.004111 27332

Perf index = 53 (util) + 40 (thru) = 93/100
```

4.2 测试结果分析与评价

开始的时候没有使用分离的空闲链表，所以测试结果只有 55。
后来改进之后 93 分
对于前面 7 个数据而言，使用显式分离链表可以达到较好的效果。
对于后两个程序而言是不够的，此时需要应用上述专门针对数据的优化方法。

4.3 性能瓶颈与改进方法分析

对于前面 7 个数据而言，使分离的空闲链表、维护大小递增、首次适配算法可以达到较好的效果。但是这对于程序 7、8、10 还是不够的，此时需要应用上述专门针对数据的优化方法。其中后两个的测试数据只是符合程序的一个特殊情况罢了。有一个思路是对于不符合此特殊情况的 realloc 程序同样合并后一个空闲块。（仅对于测试数据）

第 5 章 总结

5.1 请总结本次实验的收获

明白动态内存分配原理
了解堆的运行规则

5.2 请给出对本次实验内容的建议

希望再难一点

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.