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 计算学部

学　　号 1190201008

班　　级 1903011

学　　生 周凡

指 导 教 师 史先俊

实 验 地 点 G712

实 验 日 期 2021.6.9

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;

2.60GHz;

8.00 GB RAM;

512 G HDD

1.2.2 软件环境

Windows10 64 位;

VMware 15.5.0;

Ubuntu Desktop 18.04

1.2.3 开发工具

Visual Studio 2019; Valgrind; gprof 等

1.3 实验预习

（1）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；

（2）熟知 C 语言指针的概念、原理和使用方法；

（3）了解虚拟存储的基本原理；

（4）熟知动态内存申请、释放的方法和相关函数；

（5）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。

分配器有两种基本风格，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。

显式分配器：要求应用显式地释放任何已分配的块。例如，C 标准库提供一种叫做 malloc 程序包的显式分配器。C 程序通过调用 malloc 函数来分配一个块，并通过调用 free 函数来释放一个块。C++中的 new 和 delete 操作符与 C 中的 malloc 和 free 相当。

隐式分配器：另一方面，要求分配器检测一个已分配块何时不再被程序所使用，那么就释放这个块。隐式分配器也叫做垃圾收集器，而自动释放未使用的已分配的块的过程叫做垃圾收集，例如 Lisp、ML 以及 Java 之类的高级语言就依赖垃圾收集来释放已分配的块。

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

对于隐式空闲链表而言，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 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。

头部后面就是应用调用 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。

我们称这种结构称为隐式空闲链表，是因为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：此时我们需要某种特殊标记的结束块，可以是一个设置了已分配位而大小为零的终止头部。

Knuth 提出了一种边界标记技术，允许在常数时间内进行对前面快的合并。这种思想是在每个块的结尾处添加一个脚部（边界标记），其中脚部就是头部的一个副本。如果每个块包括这样一个脚部，那么分配器就可以通过检查它的脚部，判断前面一个块的起始位置和状态，这个脚部总是在距当前块开始位置一个字的距离。

2.3 显式空间链表的基本原理（5 分）

因为根据定义，程序不需要一个空闲块的主体，所以实现空闲链表数据结构的指针可以存放在这些空闲块的主体里面。

显式空闲链表结构将堆组织成一个双向空闲链表，在每个空闲块的主体中，都包含一个 pred（前驱）和 succ（后继）指针。

使用双向链表而不是隐式空闲链表，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（LIFO）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）

1、结构

红黑树是一种特殊的二叉查找树，其每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

除了二叉查找树的基本性质，红黑树满足一下五个性质：

（1）每个节点或者是黑色，或者是红色；

（2）根节点是黑色；

（3）每个叶子节点是黑色（这里叶子节点，是指为空的叶子节点）；

（4）如果一个节点是红色的，则它的子节点必须是黑色的；

（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

其中性质（5）确保了没有一条路径会比其他路径长出俩倍。因而，红黑树是相对接近平衡的二叉树。

2、查找

红黑树是一种特殊的二叉查找树，其查找方法和二叉查找树一样，但是由于红黑树比一般的二叉查找树具有更好的平衡性，所以查找起来更快。

```
//查找获取指定的值
public override TValue Get(TKey key)
{
    return GetValue(root, key);
}

private TValue GetValue(Node node, TKey key)
{
    if (node == null) return default(TValue);
    int cmp = key.CompareTo(node.Key);
    if (cmp == 0) return node.Value;
    else if (cmp > 0) return GetValue(node.Right, key);
    else return GetValue(node.Left, key);
}
```

3、更新

（1）旋转

旋转又分为左旋和右旋。通常左旋操作用于将一个向右倾斜的红色链接旋转为向左链接。该操作实际上是将红线链接的两个节点中的一个较大的节点移动到根节点上。而右旋转则是与左旋转完全对称的操作。旋转操作伪代码实现如下：

```
//左旋转
private Node RotateLeft(Node h)
{
    Node x = h.Right;
    //将 x 的左节点复制给 h 右节点
    h.Right = x.Left;
    //将 h 复制给 x 右节点
    x.Left = h;
    x.Color = h.Color;
```

```
        h.Color = RED;
        return x;
    }
    //右旋转
    private Node RotateRight(Node h)
    {
        Node x = h.Left;
        h.Left = x.Right;
        x.Right = h;

        x.Color = h.Color;
        h.Color = RED;
        return x;
    }
```

（2）插入结点

在插入新的键时，我们可以使用旋转操作帮助我们保证 2-3 树和红黑树之间的一一对应关系，因为旋转操作可以保持红黑树的两个重要性质：有序性和完美平衡性。也就是说，我们在红黑树中进行旋转时无需为树的有序性或者完美平衡性担心。下面我们来看看应该如何使用旋转操作来保持红黑树的另外两个重要性质：不存在两条连续的红链接和不存在红色的右链接。以下是一些简单的情况。

A. 向树底部的 2-结点插入新键

一棵只含有一个键的红黑树只含有一个 2-结点。插入另一个键之后，我们马上就需要将他们旋转。如果新键小于老键，我们只需要新增一个红色的节点即可，新的红黑树和单个 3-结点完全等价。如果新键大于老键，那么新增的红色节点将会产生一条红色的右链接。我们需要使用 parent = rotateLeft(parent);来将其旋转为红色左链接并修正根结点的链接，插入才算完成。两种情况均把一个 2-结点转换为一个 3-结点，树的黑链接高度不变。

B. 向一棵双键树（即一个 3-结点）中插入新键

这种情况又可分为三种子情况：新键小于树中的两个键，在两者之间，或是大于树中的两个键。每种情况中都会产生一个同时链接到两条红链接的结点，而我们的目标就是修正这一点。

三者中最简单的情况是新键大于原树中的两个键，因此它被链接到 3-结点的右链接。此时树是平衡的，根结点为中间大小的键，它有两条红链接分别和较小和较大的结点相连。如果我们将两条链接的颜色都由红变黑，那么我们就得到了一棵由三个结点组成，高为 2 的平衡树。它正好能够对应一棵 2-3 树。其他两种情

况最终也会转化为这两种情况。

如果新键小于原书中的两个键，它会被链接到最左边的空链接，这样就产生了两条连续的红链接。此时我们只需要将上层的红链接右旋转即可得到第一种情况。

如果新键介于原书中的两个键之间，这又会产生两条连续的红链接，一条红色左链接接一条红色右链接。此时我们只需要将下层的红链接左旋即可看得到第二种情况。

C. 根结点总是黑色

颜色转换会使根结点变为红色，我们在每次插入操作后都会将根结点设为黑色。

D. 向树底部的 3-结点插入新键

现在假设我们需要在树的底部的一个 3-结点下加入一个新结点。前面讨论过的三种情况都会出现。颜色转换会使指向中结点的链接变红，相当于将它送入了父结点。这意味着在父结点中继续插入一个新键，我们也会继续用相同的办法解决这个问题。

E. 将红链接在树中向上传递

2-3 树中的插入算法需要我们分解 3-结点，将中间键插入父结点，如此这般知道遇到一个 2-结点或是根结点。总之，只要谨慎地使用左旋，右旋，颜色转换这三种简单的操作，我们就能保证插入操作后红黑树和 2-3 树的一一对应关系。在沿着插入点到根结点的路径向上移动时在所经过的每个结点中顺序完成以下操作：

a. 如果右子结点是红色的而左子结点是黑色的，进行左旋转

b. 如果左子结点是红色的且她的左子结点也是红色的，进行右旋

c. 如果左右子结点均为红色，进行颜色转换。

（3）删除操作

要描述删除算法，首先要回到 2-3 树。和插入操作一样，我们也可以定义一系列局部变换来在删除一个结点的同时保持树的完美平衡性。这个过程比插入一个结点更加复杂，因为我们不仅要在（为了删除一个结点而）构造临时 4-结点时沿着查找路径向下进行变换，还要在分解遗留的 4-结点时沿着查找路径向上进行变换（同插入操作）。

A. 自顶向下的 2-3-4 树

先说明一个沿着查找路径既能向上也能向下进行变换的稍简单的算法：2-3-4 树的插入算法，2-3-4 树中允许存在我们以前见过的 4-结点。它的插入算法沿着查找路径向下进行变换是为了保证当前结点不是 4-结点（这样树底才有空间来插入

新的键），沿着查找路径向上进行变换是为了将之前创建的 4-结点配平。

向下的变换和我们在 2-3 树中分解 4-结点所进行的变换完全相同。如果根结点是 4-结点，我们就将它分解成三个 2-结点，使得树高加 1。在向下查找的过程中，如果遇到一个父结点为 2-结点的 4-结点，我们将 4-结点分解为两个 2-结点并将中间键传递给他的父结点，使得父结点变为一个 3-结点；如果遇到一个父结点为 3-结点的 4-结点，我们将 4-结点分解为两个 2-结点并将中间键传递给它的父结点，使得父结点变为一个 4-结点；我们不必担心会遇到父结点为 4-结点的 4-结点，因为插入算法本身就保证了这种情况不会出现。到达树的底部之后，我们也只会遇到 2-结点或者 3-结点，所以我们可以插入新的键。要用红黑树实现这个算法，我们需要：

将 4-结点表示为由三个 2-结点组成的一颗平衡的子树，根结点和两个子结点都用红链接相连；

在向下的过程中分解所有 4-结点并进行颜色转换；

和插入操作一样，在向上的过程中用旋转将 4-结点配平。（因为 4-结点可以存在，所以可以允许一个结点同时链接两条红链接）。

令人惊讶的是，你只需要移动上面算法的 put()方法中的一行代码就能实现 2-3-4 树中的插入操作：将 colorFlip()语句（及其 if 语句）移动到递归调用之前（null 测试和比较操作之间）。在多个进程可以同时访问同一棵树的应用中这个算法优于 2-3 树。

B. 删除最小键

接下来是 2-3 树中删除最小键的操作。我们注意到从树底部的 3-结点中删除键是很简单的，但 2-结点则不然。从 2-结点中删除一个键会留下一个空结点，一般我们会将它替换为一个空链接，但这样会破坏树的完美平衡。所以我们需要这样做：为了保证我们不会删除一个 2-结点，我们沿着左链接向下进行变换，确保当前结点不是 2-结点（可能是 3-结点，也可能是临时的 4-结点）。首先根结点可能有两种情况。如果根是 2-结点且它的两个子结点都是 2-结点，我们可以直接将这三个结点变为一个 4-结点；否则我们需要保证根结点的左子结点不是 2-结点，如有必要可以从它右侧的兄弟结点“借”一个键来。

在沿着左链接向下的过程中，保证以下情况之一成立：

如果当前结点的左子结点不是 2-结点，完成；

如果当前结点的左子结点是 2-结点而它的亲兄弟结点不是 2-结点，将左子结点的兄弟结点中的一个键移动到左子结点中；

如果当前结点的左子结点和它的亲兄弟结点都是 2-结点，将左子结点，父结

点中的最小键和左子结点最近的兄弟结点合并为一个 4-结点，使父结点由 3-结点变为 2-结点或由 4-结点变为 3-结点。

C. 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结点均不是 2-结点。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结点交换，就和二叉树一样。因为当前结点必然不是 2-结点，问题已经转化为在一颗根结点不是 2-结点子树中删除最小键，我们可以在这个子树中使用前问所述的算法。和以前一样，删除之后我们需要向上回溯并分解余下的 4-结点。

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。简单来说，动态分配器就是我们平时在 C 语言上用的 malloc 和 free,realloc，通过分配堆上的内存给程序，我们通过向堆申请一块连续的内存，然后将堆中连续的内存按 malloc 所需要的块来分配，不够了，就继续向堆申请新的内存，也就是扩展堆，这里设定，堆顶指针想上伸展（堆的大小变大）。

2、 堆中内存块的组织结构

用隐式空闲链表来组织堆。对于带边界标签的隐式空闲链表分配器，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、 所采用的空闲块和分配块链表

采用分离的空闲链表，其中全局变量 void *Lists[MAX_LEN]用于存储每个链表的链尾。因为一个使用单向空闲块链表的分配器需要与空闲块数量呈线性关系的时间来分配块，而此堆的设计采用分离存储减少了需要查找的空闲块数量，进而减少了分配的时间。每个链表中的块有大致相等的大小，将所有可能的块大小根据 2 的幂进行划分组成相应的链表。

4、 适配方式

对分离的空闲链表采取首次适配的策略。为了分配一个块，先确定请求的大小类，并且对适当的空闲链表做首次适配，查找一个合适的块。如果找到了一个，那么就（可选地）分割它，并将剩余的部分插入到适当的空闲链表中。如果找不到合适的块，那么就搜索下一个更大的大小类的空闲链表。如此重复，直到找到

一个合适的块。如果空闲链表没有合适的块，那么就向操作系统请求额外的堆内存，从新的堆内存中分配一个块，将剩部分放置在适当的大小类中。要释放的块，我们执行合并，并将结果放置在相应的空闲链表中。对分离空闲链表的简单的首次适配搜索，其内存利用率近似于对整个堆的最佳适配搜索的内存利用率。

5、 相应算法

此处仅介绍扩展堆、插入节点、删除节点以及分配函数的算法，其余算法见下文“关键函数设计”。

（1） extend_heap 扩展堆：将堆扩展 size 大小。首先将 size 对齐， 然后调用 mem_sbrk 申请空间，设置块的 head 和 foot，重新设置结尾块，插入节点，最后调用 coalesce 函数进行必要的合并空闲块操作。

（2） insert_node 插入节点：向空闲链表中添加大小为 size 的空闲块，指针为 ptr。首先找到合适的链表，再在链表中找到合适的位置进行插入，最后需要注意插入时前驱和后继是否为 NULL 的情况，针对不同情况需要对 head、foot 以及 Lists 进行相应的修改。

（3） delete_node 删除节点：将空闲链表中指针 ptr 指向的空闲块删除。首先找到合适的链表（以便对 Lists 进行修改），再将对应的空闲块删除，最后需要注意插入时前驱和后继是否为 NULL 的情况，针对不同情况需要对 head、foot 以及 Lists 进行相应的修改。

（4） place 分配函数：将 ptr 指向的空闲块分配出 asize 大小的空间。首先将 ptr 指向的块从链表中删除，然后根据剩余空间大小判断是否需要切割块，再比较 asize 和阈值决定分配块的前部分或后部分。

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：初始化内存分配器。

处理流程：

（1） 初始化每个分离的空闲链表，将其设置为 NULL；

（2） 先对齐，再初始化一个序言块和一个结尾块。序言块是一个 8 字节的已分配块，仅由一个 head 和一个 foot 组成。结尾块是一个大小为 0 的已分配块；

（3） 调用 extend_heap 函数将初始化的堆进行扩展。

要点分析：

初始化分配器时，分配器使用的最小块为 16 字节，空闲链表采取隐式空闲链表的方式进行组织，形式为不使用的双字边界对齐填充字接 8 字节大小的序言块

以及最后 4 字节的结尾块，且空闲链表创建后利用 extend_heap 函数来扩展堆。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放指定块。

参　　数：void *ptr; //需要被释放的块对应的指针

处理流程：

（1） 通过 GET_SIZE(HDRP(ptr))语句获得释放块的大小，并使用 PUT 将块的 head 和 foot 的已分配位设置为 0，表示该块已经被 free；

（2） 调用 insert_node 函数将释放的块插入到空闲链表中；

（3） 调用 coalesce 函数，将刚插入的空闲块与周围可能存在的空闲块进行合并。

要点分析：

需要将释放块的 head 和 foot 的已分配位都设置为 0 以示释放，插入空闲链表后需要在添加一步合并空闲块的操作以减少外部碎片。

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

函数功能：为 ptr 指向的内存块重新分配一个大小至少为 size 的块。

参　　数：void *ptr, size_t size; //ptr 为需要重新分配的块对应的指针，size 为需要分配的大小

处理流程：

（1） 将需要分配的大小 copySize 进行对齐；

（2） 比较 copySize 和原来块的大小，若后者更大则不需要扩展；

（3） 若需要扩展，则再分以下 3 种情况讨论：

a. 若待处理块的下一块是结束块，则删除该块，在内存中重新申请一个 copySize 的块存储该块的内容；

b. 若待处理块的下一块是空闲块，则判断这两块合并后的大小是否足够，若足够则删除后面那块空闲块并更改待处理块大小（相当于将两块合并）；若不够，则删除该块，在内存中重新申请一个 copySize 的块存储该块的内容；

c. 否则，直接删除该块，在内存中重新申请一个 copySize 的块存储该块的内容；

（4） 检查堆的一致性。

要点分析：

（1） 要注意 copySize 的对齐，若 copySize<=DSIZE 的情况还需要额外考虑，其

他情况直接调用 ALIGN 进行对齐；

（2） 对重新分配的内存比原来小的情况只需修改块的大小即可；

（3） 为了减少外部碎片，需要尽可能利用相邻的空闲块。

3.2.4 int mm_check(void)函数（5 分）

函数功能：进行堆的一致性检查。

处理流程：

（1） 遍历所有空闲链表，检查是否有未合并的相邻空闲块以及是否所有链表中的块都是空闲的；

（2） 若上述遍历正常结束，则说明链表中所有的指针均指向有效的空闲块；

（3） 遍历所有空闲块，若遍历顺利结束，则说明每个堆块的指针都指向有效的堆地址；

（4） 通过比较空闲块的个数和空闲链表中块的个数判断是否所有空闲块都在链表中。

要点分析：

通过 Lists 数组对空闲链表进行遍历，通过 heap_listp 获得堆地址的开始指针对所有块进行遍历。可以通过判断遍历是否顺利完成判断链表中所有的指针均指向有效的空闲块以及每个堆块的指针都指向有效的堆地址，通过查看块的已分配位判断是否是空闲块，通过比较空闲链表中的块个数和空闲块的总个数判断是否所有空闲块都在链表中。

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

函数功能：分配大小为 size 的内存。

参　　数：size_t size; //需要分配的内存的大小

处理流程：

（1） 对 size 进行对齐；

（2） 根据 size 的大小寻找合适的链表；

（3） 遍历目标链表，链表中块大小是递增的，采取首次适配的策略，在寻找到首个适合的空闲块时就分配返回；

（4） 若没有找到合适的块，则扩展堆；

（5） 最后调用 place 函数，在 ptr 指向的空闲块中分配指定大小的内存，若剩余内存大于最小块则会对块进行分割；

（6） 检查堆的一致性。

要点分析：

（1） 寻找合适链表时，由于每个链表都和一个内存大小 size 唯一对应，故通过内存大小进行遍历，每次都对大小进行>>=2 操作；

（2） Lists 数组中存储的是每个链表的链尾，故在链表中查找适合块需要每次查找上一块的前驱；

（3） 最后需要调用 place 函数进行块大小的分配，必要时需要对块进行分割。

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

函数功能：合并相邻空闲块。

参　　数：void *bp; //需要进行相邻空闲块合并的块对应的指针

处理流程：

（1） 若该块的前驱和后继都是已分配的，则无需合并操作，直接返回；

（2） 若前驱已分配，后继未分配，则在空闲链表中删除该块和后继，将两者合并后再插入链表；

（3） 若前驱未分配，后继已分配，则在空闲链表中删除该块和前驱，将两者合并后再插入链表；

（4） 若前驱和后继都未分配，则在空闲链表中删除该块、前驱和后继，将三者合并后再插入链表。

要点分析：

（1） 合并空闲块时，只需要修改最前面空闲块的 head 和最后面空闲块的 foot 即可；

（2） 修改链表中的空闲块位置，需要先删除需要合并的所有空闲块，再将新合并的块插入到链表中；

（3） 返回的指针需要时指向合并后空闲块的首地址的，即若有前驱未分配则返回前驱的指针，否则返回原指针。

第 4 章测试

总分 10 分

4.1 测试方法

1、生成可执行评测程序文件的方法：linux>make
2、评测方法:mdriver [-hvVa] [-f <file>]
选项：

-a	不检查分组信息
-f <file>	使用 <file>作为单个的测试轨迹文件
-h	显示帮助信息
-l	也运行 C 库的 malloc
-v	输出每个轨迹文件性能
-V	输出额外的调试信息

3、轨迹文件：指示测试驱动程序 mdriver 以一定顺序调用
4、性能分 pindex 是空间利用率和吞吐率的线性组合
5、获得测试总：linux>./mdriver -av -t traces/

4.2 测试结果评价

由于通过分离空闲链表的有序性以及 place、coalesce 等函数和算法尽可能地减少了内部和外部碎片，即使是首次适配也能几乎达到最佳适配的效果，在很大程度上提高了内存率和吞吐率。

4.3 自测试结果

```
zf1190201008@ubuntu:~/Lab678/malloclab-handout$ ./mdriver -t traces/ -v
Team Name:1190201008
Member 1 :Zhou Fan:434696317@qq.com
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   98%    5694  0.000502 11352
 1       yes   97%    5848  0.000257 22781
 2       yes   98%    6648  0.000290 22893
 3       yes   99%    5380  0.000229 23473
 4       yes   99%   14400  0.000346 41631
 5       yes   93%    4800  0.000361 13289
 6       yes   91%    4800  0.000321 14953
 7       yes   55%   12000  0.000361 33213
 8       yes   51%   24000  0.000706 33975
 9       yes  100%   14401  0.000271 53219
10       yes   87%   14401  0.000273 52732
Total          88%  112372  0.003917 28685
```

第 5 章 总结

5.1 请总结本次实验的收获

1、对动态内存分配的实现原理有了更加深入的理解；
2、了解了多种动态内存的分配策略以及各自的特点；
3、了解了红黑树的结构、特点以及算法。

5.2 请给出对本次实验内容的建议

无

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.