

**实验报告**

**实 验（八）**

题 目 Dynamic Storage Allocator

动态内存分配器

专 业 计算机

学　　 号 1180300303

班　　 级 1836101

学 生 宿梓航

指 导 教 师 刘宏伟

实 验 地 点 G712

实 验 日 期 12.12.2019

**计算机科学与技术学院**

**目 录**

[第1章 实验基本信息 - 3 -](#_Toc500449079)

[1.1 实验目的 - 3 -](#_Toc500449080)

[1.2 实验环境与工具 - 3 -](#_Toc500449081)

[1.2.1 硬件环境 - 3 -](#_Toc500449082)

[1.2.2 软件环境 - 3 -](#_Toc500449083)

[1.2.3 开发工具 - 3 -](#_Toc500449084)

[1.3 实验预习 - 3 -](#_Toc500449085)

[第2章 实验预习 - 4 -](#_Toc500449086)

[2.1 进程的概念、创建和回收方法（5分） - 4 -](#_Toc500449087)

[2.2信号的机制、种类（5分） - 4 -](#_Toc500449088)

[2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5分） - 4 -](#_Toc500449089)

[2.4 什么是shell，功能和处理流程（5分） - 4 -](#_Toc500449090)

[第3章 TinyShell测试 - 5 -](#_Toc500449091)

[3.1 TinyShell设计 - 5 -](#_Toc500449092)

[第4章 总结 - 5 -](#_Toc500449093)

[4.1 请总结本次实验的收获 - 5 -](#_Toc500449094)

[4.2 请给出对本次实验内容的建议 - 5 -](#_Toc500449095)

[参考文献 - 7 -](#_Toc500449096)

# 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

Intel Core i7-8750H, 2.2GHz

16G RAM

Samsung SSD 970 PRO 1TB

### 1.2.2 软件环境

Windows 10 1903 x64

Ubuntu 19.04

Vmware Workstation 15.1

### 1.2.3 开发工具

Visual Studio 2019

GCC 8.3.0，8.1.0

GNU nano 3.2

## 1.3 实验预习

填写

# 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一个进程的虚拟内存区域，称为堆。对于每个进程，内核维护一个变量brk，指向堆顶。

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：显式分配器和隐式分配器。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。

1. 显式分配器：要求应用显式地释放任何已分配的块。例如，C标准库提一种叫做malloc程序包的显示分配器。C程序通过调用malloc函数来分配一个块，通过调用free函数来释放一个块。
2. 隐式分配器：也叫做垃圾收集器，例如，诸如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

对于malloc，其原理是：当堆内有足够的空闲时，则分配该空闲；否则就扩张堆。

## 2.2 带边界标签的隐式空闲链表分配器原理（5分）

带边界标签的隐式空闲链表分配器中，一个块是由一个字的头部(包含大小和是否被占用的信息)、有效载荷、可能的一些额外的填充，以及一个字大小、和头部一样的尾部。

因为采用八字节对齐约束条件，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，我们只需要内高位来储存块大小，最低位来记录占用状态。

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，用于满足对齐要求。

因为空闲块是通过头部中的大小字段隐含地连接着的，可以视为一个隐式空闲链表。分配器可以通过遍历堆中所有的块，间接地遍历整个空闲块的集合。

通过遍历，我们可以对块进行回收、合并、分配

## 2.3 显示空间链表的基本原理（5分）

由于空闲块并不被程序使用，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“有效载荷”里。

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，都包含上一项和下一项的指针。

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）

红黑树是每个节点都带有颜色属性的二叉查找树，颜色为红色或黑色。在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求：

* 节点是红色或黑色。
* 根是黑色。
* 所有叶子都是黑色（叶子是NIL节点）。
* 每个红色节点必须有两个黑色的子节点。（从每个叶子到根的所有路径上不能有两个连续的红色节点。）
* 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

**查找操作**

红黑树仍是二叉查找树。假设对于每个节点的值有LChild<This<RChild，查找只需：

1. 访问根结点
2. 对被访问的结点：查找值等于当前值，则查找完毕，返回；

查找值小于当前值，则继续访问LChild；

否则，访问RChild

1. 对于被访问的结点执行2.

**更新操作**

***插入***

我们首先以二叉查找树的方法增加节点并标记它为红色。（如果设为黑色，就会导致根到叶子的路径上有一条路上，多一个额外的黑节点，这个是很难调整的。但是设为红色节点后，可能会导致出现两个连续红色节点的冲突，那么可以通过颜色调换（color flips）和树旋转来调整。）下面要进行什么操作取决于其他临近节点的颜色。同人类的家族树中一样，我们将使用术语叔父节点来指一个节点的父节点的兄弟节点。注意：

性质1和性质3总是保持着。

性质4只在增加红色节点、重绘黑色节点为红色，或做旋转时受到威胁。

性质5只在增加黑色节点、重绘红色节点为黑色，或做旋转时受到威胁。

在下面的示意图中，将要插入的节点标为N，N的父节点标为P，N的祖父节点标为G，N的叔父节点标为U。在图中展示的任何颜色要么是由它所处情形这些所作的假定，要么是假定所暗含（imply）的。

对于每一种情形，我们将使用C示例代码来展示。通过下列函数，可以找到一个节点的叔父和祖父节点：

node\* grandparent(node \*n){

return n->parent->parent;

}

node\* uncle(node \*n){

if(n->parent == grandparent(n)->left)

return grandparent (n)->right;

else

return grandparent (n)->left;

}

情形1:新节点N位于树的根上，没有父节点。在这种情形下，我们把它重绘为黑色以满足性质2。因为它在每个路径上对黑节点数目增加一，性质5符合。

void insert\_case1(node \*n){

if(n->parent == NULL)

n->color = BLACK;

else

insert\_case2 (n);

}

情形2:新节点的父节点P是黑色，所以性质4没有失效（新节点是红色的）。在这种情形下，树仍是有效的。性质5也未受到威胁，尽管新节点N有两个黑色叶子子节点；但由于新节点N是红色，通过它的每个子节点的路径就都有同通过它所取代的黑色的叶子的路径同样数目的黑色节点，所以依然满足这个性质。

void insert\_case2(node \*n){

if(n->parent->color == BLACK)

return; /\* 树仍旧有效\*/

else

insert\_case3 (n);

}

注意：在下列情形下我们假定新节点的父节点为红色，所以它有祖父节点；因为如果父节点是根节点，那父节点就应当是黑色。所以新节点总有一个叔父节点，尽管在情形4和5下它可能是叶子节点。

情形3示意图

情形3:如果父节点P和叔父节点U二者都是红色，（此时新插入节点N做为P的左子节点或右子节点都属于情形3，这里右图仅显示N做为P左子的情形）则我们可以将它们两个重绘为黑色并重绘祖父节点G为红色（用来保持性质5）。现在我们的新节点N有了一个黑色的父节点P。因为通过父节点P或叔父节点U的任何路径都必定通过祖父节点G，在这些路径上的黑节点数目没有改变。但是，红色的祖父节点G可能是根节点，这就违反了性质2，也有可能祖父节点G的父节点是红色的，这就违反了性质4。为了解决这个问题，我们在祖父节点G上递归地进行情形1的整个过程。（把G当成是新加入的节点进行各种情形的检查）

void insert\_case3(node \*n){

if(uncle(n) != NULL && uncle (n)->color == RED) {

n->parent->color = BLACK;

uncle (n)->color = BLACK;

grandparent (n)->color = RED;

insert\_case1(grandparent(n));

}

else

insert\_case4 (n);

}

注意：在余下的情形下，我们假定父节点P是其祖父G的左子节点。如果它是右子节点，情形4和情形5中的左和右应当对调。

情形4示意图

情形4:父节点P是红色而叔父节点U是黑色或缺少，并且新节点N是其父节点P的右子节点而父节点P又是其父节点的左子节点。在这种情形下，我们进行一次左旋转调换新节点和其父节点的角色;接着，我们按情形5处理以前的父节点P以解决仍然失效的性质4。注意这个改变会导致某些路径通过它们以前不通过的新节点N（比如图中1号叶子节点）或不通过节点P（比如图中3号叶子节点），但由于这两个节点都是红色的，所以性质5仍有效。

void insert\_case4(node \*n){

if(n == n->parent->right && n->parent == grandparent(n)->left) {

rotate\_left(n->parent);

n = n->left;

} else if(n == n->parent->left && n->parent == grandparent(n)->right) {

rotate\_right(n->parent);

n = n->right;

}

insert\_case5 (n);

}

情形5示意图

情形5：父节点P是红色而叔父节点U是黑色或缺少，新节点N是其父节点的左子节点，而父节点P又是其父节点G的左子节点。在这种情形下，我们进行针对祖父节点G的一次右旋转；在旋转产生的树中，以前的父节点P现在是新节点N和以前的祖父节点G的父节点。我们知道以前的祖父节点G是黑色，否则父节点P就不可能是红色（如果P和G都是红色就违反了性质4，所以G必须是黑色）。我们切换以前的父节点P和祖父节点G的颜色，结果的树满足性质4。性质5也仍然保持满足，因为通过这三个节点中任何一个的所有路径以前都通过祖父节点G，现在它们都通过以前的父节点P。在各自的情形下，这都是三个节点中唯一的黑色节点。

void insert\_case5(node \*n){

n->parent->color = BLACK;

grandparent (n)->color = RED;

if(n == n->parent->left && n->parent == grandparent(n)->left) {

rotate\_right(grandparent(n));

} else {

/\* Here, n == n->parent->right && n->parent == grandparent (n)->right \*/

rotate\_left(grandparent(n));

}

}

注意插入实际上是原地算法，因为上述所有调用都使用了尾部递归。

***删除***

如果需要删除的节点有两个儿子，那么问题可以被转化成删除另一个只有一个儿子的节点的问题（为了表述方便，这里所指的儿子，为非叶子节点的儿子）。对于二叉查找树，在删除带有两个非叶子儿子的节点的时候，我们要么找到它左子树中的最大元素、要么找到它右子树中的最小元素，并把它的值转移到要删除的节点中（如在这里所展示的那样）。我们接着删除我们从中复制出值的那个节点，它必定有少于两个非叶子的儿子。因为只是复制了一个值（没有复制颜色），不违反任何性质，这就把问题简化为如何删除最多有一个儿子的节点的问题。它不关心这个节点是最初要删除的节点还是我们从中复制出值的那个节点。

在本文余下的部分中，我们只需要讨论删除只有一个儿子的节点（如果它两个儿子都为空，即均为叶子，我们任意将其中一个看作它的儿子）。如果我们删除一个红色节点（此时该节点的儿子将都为叶子节点），它的父亲和儿子一定是黑色的。所以我们可以简单的用它的黑色儿子替换它，并不会破坏性质3和性质4。通过被删除节点的所有路径只是少了一个红色节点，这样可以继续保证性质5。另一种简单情况是在被删除节点是黑色而它的儿子是红色的时候。如果只是去除这个黑色节点，用它的红色儿子顶替上来的话，会破坏性质5，但是如果我们重绘它的儿子为黑色，则曾经通过它的所有路径将通过它的黑色儿子，这样可以继续保持性质5。

需要进一步讨论的是在要删除的节点和它的儿子二者都是黑色的时候，这是一种复杂的情况（这种情况下该结点的两个儿子都是叶子结点，否则若其中一个儿子是黑色非叶子结点，另一个儿子是叶子结点，那么从该结点通过非叶子结点儿子的路径上的黑色结点数最小为2，而从该结点到另一个叶子结点儿子的路径上的黑色结点数为1，违反了性质5）。我们首先把要删除的节点替换为它的儿子。出于方便，称呼这个儿子为N（在新的位置上），称呼它的兄弟（它父亲的另一个儿子）为S。在下面的示意图中，我们还是使用P称呼N的父亲，SL称呼S的左儿子，SR称呼S的右儿子。我们将使用下述函数找到兄弟节点：

struct node \*

sibling(struct node \*n)

{

if(n == n->parent->left)

return n->parent->right;

else

return n->parent->left;

}

我们可以使用下列代码进行上述的概要步骤，这里的函数replace\_node替换child到n在树中的位置。出于方便，在本章节中的代码将假定空叶子被用不是NULL的实际节点对象来表示（在插入章节中的代码可以同任何一种表示一起工作）。

void

delete\_one\_child(struct node \*n)

{

/\*

\* Precondition: n has at most one non-null child.

\*/

struct node \*child = is\_leaf(n->right)? n->left : n->right;

replace\_node(n, child);

if(n->color == BLACK){

if(child->color == RED)

child->color = BLACK;

else

delete\_case1 (child);

}

free (n);

}

如果N和它初始的父亲是黑色，则删除它的父亲导致通过N的路径都比不通过它的路径少了一个黑色节点。因为这违反了性质5，树需要被重新平衡。有几种情形需要考虑：

情形1: N是新的根。在这种情形下，我们就做完了。我们从所有路径去除了一个黑色节点，而新根是黑色的，所以性质都保持着。

void

delete\_case1(struct node \*n)

{

if(n->parent != NULL)

delete\_case2 (n);

}

注意：在情形2、5和6下，我们假定N是它父亲的左儿子。如果它是右儿子，则在这些情形下的左和右应当对调。

情形2示意图

情形2： S是红色。在这种情形下我们在N的父亲上做左旋转，把红色兄弟转换成N的祖父，我们接着对调N的父亲和祖父的颜色。完成这两个操作后，尽管所有路径上黑色节点的数目没有改变，但现在N有了一个黑色的兄弟和一个红色的父亲（它的新兄弟是黑色因为它是红色S的一个儿子），所以我们可以接下去按情形4、情形5或情形6来处理。

（注意：这里的图中没有显示出来，N是删除了黑色节点后替换上来的子节点，所以这个过程中由P->X->N变成了P->N，实际上是少了一个黑色节点，也可以理解为Parent(Black)和Silbing(Red)那么他们的孩子黑色节点的数目肯定不等，让他们做新兄弟肯定是不平衡的，还需后面继续处理。这里看英文版本的[1]比较的明了）

void

delete\_case2(struct node \*n)

{

struct node \*s = sibling (n);

if(s->color == RED){

n->parent->color = RED;

s->color = BLACK;

if(n == n->parent->left)

rotate\_left(n->parent);

else

rotate\_right(n->parent);

}

delete\_case3 (n);

}

情形3示意图

情形3： N的父亲、S和S的儿子都是黑色的。在这种情形下，我们简单的重绘S为红色。结果是通过S的所有路径，它们就是以前不通过N的那些路径，都少了一个黑色节点。因为删除N的初始的父亲使通过N的所有路径少了一个黑色节点，这使事情都平衡了起来。但是，通过P的所有路径现在比不通过P的路径少了一个黑色节点，所以仍然违反性质5。要修正这个问题，我们要从情形1开始，在P上做重新平衡处理。

void

delete\_case3(struct node \*n)

{

struct node \*s = sibling (n);

if((n->parent->color == BLACK)&&

(s->color == BLACK)&&

(s->left->color == BLACK)&&

(s->right->color == BLACK)) {

s->color = RED;

delete\_case1(n->parent);

} else

delete\_case4 (n);

}

情形4示意图

情形4： S和S的儿子都是黑色，但是N的父亲是红色。在这种情形下，我们简单的交换N的兄弟和父亲的颜色。这不影响不通过N的路径的黑色节点的数目，但是它在通过N的路径上对黑色节点数目增加了一，添补了在这些路径上删除的黑色节点。

void

delete\_case4(struct node \*n)

{

struct node \*s = sibling (n);

if（(n->parent->color == RED)&&

(s->color == BLACK)&&

(s->left->color == BLACK)&&

(s->right->color == BLACK)) {

s->color = RED;

n->parent->color = BLACK;

} else

delete\_case5 (n);

}

情形5示意图

情形5： S是黑色，S的左儿子是红色，S的右儿子是黑色，而N是它父亲的左儿子。在这种情形下我们在S上做右旋转，这样S的左儿子成为S的父亲和N的新兄弟。我们接着交换S和它的新父亲的颜色。所有路径仍有同样数目的黑色节点，但是现在N有了一个黑色兄弟，他的右儿子是红色的，所以我们进入了情形6。N和它的父亲都不受这个变换的影响。

void

delete\_case5(struct node \*n)

{

struct node \*s = sibling (n);

if（s->color == BLACK){ /\* this if statement is trivial,

due to Case 2(even though Case two changed the sibling to a sibling's child,

the sibling's child can't be red, since no red parent can have a red child). \*/

// the following statements just force the red to be on the left of the left of the parent,

// or right of the right, so case six will rotate correctly.

if((n == n->parent->left)&&

(s->right->color == BLACK)&&

(s->left->color == RED)) { // this last test is trivial too due to cases 2-4.

s->color = RED;

s->left->color = BLACK;

rotate\_right (s);

} else if((n == n->parent->right)&&

(s->left->color == BLACK)&&

(s->right->color == RED)) {// this last test is trivial too due to cases 2-4.

s->color = RED;

s->right->color = BLACK;

rotate\_left (s);

}

}

delete\_case6 (n);

}

情形6示意图

情形6： S是黑色，S的右儿子是红色，而N是它父亲的左儿子。在这种情形下我们在N的父亲上做左旋转，这样S成为N的父亲（P）和S的右儿子的父亲。我们接着交换N的父亲和S的颜色，并使S的右儿子为黑色。子树在它的根上的仍是同样的颜色，所以性质3没有被违反。但是，N现在增加了一个黑色祖先：要么N的父亲变成黑色，要么它是黑色而S被增加为一个黑色祖父。所以，通过N的路径都增加了一个黑色节点。

此时，如果一个路径不通过N，则有两种可能性：

它通过N的新兄弟。那么它以前和现在都必定通过S和N的父亲，而它们只是交换了颜色。所以路径保持了同样数目的黑色节点。

它通过N的新叔父，S的右儿子。那么它以前通过S、S的父亲和S的右儿子，但是现在只通过S，它被假定为它以前的父亲的颜色，和S的右儿子，它被从红色改变为黑色。合成效果是这个路径通过了同样数目的黑色节点。

在任何情况下，在这些路径上的黑色节点数目都没有改变。所以我们恢复了性质4。在示意图中的白色节点可以是红色或黑色，但是在变换前后都必须指定相同的颜色。

void

delete\_case6(struct node \*n)

{

struct node \*s = sibling (n);

s->color = n->parent->color;

n->parent->color = BLACK;

if(n == n->parent->left){

s->right->color = BLACK;

rotate\_left(n->parent);

} else {

s->left->color = BLACK;

rotate\_right(n->parent);

}

}

同样的，函数调用都使用了尾部递归，所以算法是原地算法。此外，在旋转之后不再做递归调用，所以进行了恒定数目（最多3次）的旋转。

# 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。**

**堆**

动态内存分配器维护一个进程的虚拟内存区域，称为堆。对于每个进程，内核维护一个变量brk，指向堆顶。

**堆中内存块的组织结构**

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。

**采用的空闲块、分配块链表/树结构和相应算法**

带边界标签的隐式空闲链表分配器中，一个块是由一个字的头部(包含大小和是否被占用的信息)、有效载荷、可能的一些额外的填充，以及一个字大小、和头部一样的尾部。

因为采用八字节对齐约束条件，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，我们只需要内高位来储存块大小，最低位来记录占用状态。

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，用于满足对齐要求。

因为空闲块是通过头部中的大小字段隐含地连接着的，可以视为一个隐式空闲链表。分配器可以通过遍历堆中所有的块，间接地遍历整个空闲块的集合。

通过遍历，我们可以对块进行回收、合并、分配。

**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：

初始化内存系统，即初始化隐式空闲链表和堆。

处理流程：

1. 将堆扩充4个字
2. 将这四个字分别如下初始化：
   1. 第一个字作为对齐填充，设为0
   2. 第二个字是序言块的头标记(含有大小和分配状态)
   3. 第三个字是序言块的尾标记
   4. 第四个字是结尾标记(大小为0的已分配块)
3. 更新指向第一块的指针到序言块的尾处。
4. 将堆扩充CHUNKSIZE / WSIZE个字

要点分析：

1. 初始化隐式空闲链表
2. 使用extend\_heap函数来扩展堆

## 3.2.2 void mm\_free(void \*ptr)函数（5分）

函数功能：

释放一个块。

参 数：

bp表示指向该块的指针

处理流程：

1. 获取该块的大小
2. 更新块首尾的占用标记
3. 立即合并相邻空闲

要点分析：

1. 使用宏函数获取块大小和更新标记。
2. 更新标记后立即合并相邻空闲。

## 3.2.3 void \*mm\_realloc(void \*ptr, size\_t size)函数（5分）

函数功能：

对ptr所指的块扩充为，或者重新分配出一个具有size大小的块。

参 数：

ptr指向待扩充/重新分配的块，size表示期待大小。

处理流程：

1. 申请一个具有Size大小的块。
2. 获取原块大小copySize，和Size之间的最小值作为待复制数据的大小
3. 复制数据，然后释放ptr所在的块。

要点分析：

1. 为了简化逻辑，mm\_realloc总是重新申请一块新的内存，而不是扩充原有的块
2. 复制数据时，选取当前块和新块大小中的小者作为复制大小。

## 3.2.4 int mm\_check(void)函数（5分）

(是void mm\_checkheap(int verbose)吗。。。)

函数功能：

检查堆的一致性。

参 数：

Verbose：表示是否显示详细信息

处理流程：

1. 首先(可能输出)堆的地址
2. 检查序言块的大小和分配情况，视情况输出Bad prologue header
3. 遍历链表，对堆中的每一块检查对齐(使用checkblock)，可能还printblock
4. 如果结尾快的大小非0或者未占用，则输出Bad epilogue header

要点分析：

1. 检查了序言块和结尾块
2. 对每一块进行了对其检查

## 3.2.5 void \*mm\_malloc(size\_t size)函数（10分）

函数功能：

求分配一个具有size大小的块。

参 数：

Size表示申请的块的大小。

处理流程：

1. 检查申请的大小是否是个整数，否则返回null
2. 根据size的大小获取实际需要的块大小asize
3. 遍历整个链表寻找适配(使用find\_fit)
4. 如果找到适配，则更新找到的位置的块信息(使用place)，然后返回找到的适配
5. 否则扩充堆(大小取决于asize和CHUNKSIZE)之间的大值。
6. 如果成功扩充堆，则更新块信息(使用place)，然后返回对应的指针，否则返回直接null

要点分析：

1. 优先在链表里查找空闲，减少外部碎片。
2. 如果当前堆里没有足够大的空间，才扩充堆。
3. 如果成功分配，必然会调用place函数。

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

函数功能：

合并bp指向的块的附近的可能存在的空闲块。

处理流程：

1. 首先获取前后块的占用状态prev\_alloc, next\_alloc
2. 获取当前块的size
3. 分四种情况：
   1. 前后的块都被占用，则直接返回bp
   2. 仅后方的块被占用，则先更新size为当前和前块之和，然后更新前块的头标记和本块的尾标记；最后返回前一块的指针。
   3. 仅前方的块被占用，则先更新size为当前和后块之和，然后更新后块的尾标记和本块的头标记；最后返回本块的指针。
   4. 前后块均为被占用，则先更新size为当前块、后快和前块之和，然后更新前块的头标记和后块的尾标记；最后返回前一块的指针。

要点分析：

分四种情况讨论，分别实现。注意标记更新的位置和返回值。

# 第4章测试

**总分10分**

## 4.1 测试方法

**生成可执行评测程序文件的方法**

linux>make

**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

**我使用以下python程序进行测试：**

*#!/usr/bin/python3*

*import* os

traces=os.listdir('./traces/')

*for* trace in traces:

    print('for '+trace+':')

    os.system('./mdriver -v -f ./traces/'+trace)

    print()

    print()

## 4.2 测试结果评价

性能分pindex是空间利用率和吞吐率的线性组合：

p1 = UTIL\_WEIGHT \* avg\_mm\_util;

if (avg\_mm\_throughput > AVG\_LIBC\_THRUPUT) {

p2 = (double)(1.0 - UTIL\_WEIGHT);

}

else {

p2 = ((double) (1.0 - UTIL\_WEIGHT)) \*

(avg\_mm\_throughput/AVG\_LIBC\_THRUPUT);

}

pindex = (p1 + p2)\*100.0;

- UTIL\_WEIGHT = 0.6

- AVG\_LIBC\_THRUPUT=600000

- avg\_mm\_util 测试实验代码malloc得到的平均空间利用率

avg\_mm\_util = hwm/heapsize = brk/heapsize //hwm is high water mark

- avg\_mm\_throughput 测试实验代码malloc得到的平均吞吐率(次/秒, ops/sec)

每个人用不同的机器，速度不同，评分仅供参考。统一评测才有意义。

## 4.3 自测试结果

**for binary-bal.rep:**

Team Name:implicit first fit

Member 1 :Dave OHallaron:droh

Measuring performance with gettimeofday().

Results for mm malloc:

trace valid util ops secs Kops

0 yes 55% 12000 0.146755 82

Total 55% 12000 0.146755 82

Perf index = 33 (util) + 5 (thru) = 38/100

**for amptjp-bal.rep:**

Team Name:implicit first fit

Member 1 :Dave OHallaron:droh

Measuring performance with gettimeofday().

Results for mm malloc:

trace valid util ops secs Kops

0 yes 99% 5694 0.007756 734

Total 99% 5694 0.007756 734

Perf index = 60 (util) + 40 (thru) = 100/100

**for expr-bal.rep:**

Team Name:implicit first fit

Member 1 :Dave OHallaron:droh

Measuring performance with gettimeofday().

Results for mm malloc:

trace valid util ops secs Kops

0 yes 100% 5380 0.008717 617

Total 100% 5380 0.008717 617

Perf index = 60 (util) + 40 (thru) = 100/100

**for random2-bal.rep:**

Team Name:implicit first fit

Member 1 :Dave OHallaron:droh

Measuring performance with gettimeofday().

Results for mm malloc:

trace valid util ops secs Kops

0 yes 92% 4800 0.006949 691

Total 92% 4800 0.006949 691

Perf index = 55 (util) + 40 (thru) = 95/100

**for realloc-bal.rep:**

Team Name:implicit first fit

Member 1 :Dave OHallaron:droh

Measuring performance with gettimeofday().

Results for mm malloc:

trace valid util ops secs Kops

0 yes 27% 14401 0.061736 233

Total 27% 14401 0.061736 233

Perf index = 16 (util) + 16 (thru) = 31/100

**for cccp-bal.rep:**

Team Name:implicit first fit

Member 1 :Dave OHallaron:droh

Measuring performance with gettimeofday().

Results for mm malloc:

trace valid util ops secs Kops

0 yes 99% 5848 0.006385 916

Total 99% 5848 0.006385 916

Perf index = 60 (util) + 40 (thru) = 100/100

**for binary2-bal.rep:**

Team Name:implicit first fit

Member 1 :Dave OHallaron:droh

Measuring performance with gettimeofday().

Results for mm malloc:

trace valid util ops secs Kops

0 yes 51% 24000 0.272490 88

Total 51% 24000 0.272490 88

Perf index = 31 (util) + 6 (thru) = 37/100

**for realloc2-bal.rep:**

Team Name:implicit first fit

Member 1 :Dave OHallaron:droh

Measuring performance with gettimeofday().

Results for mm malloc:

trace valid util ops secs Kops

0 yes 34% 14401 0.002388 6030

Total 34% 14401 0.002388 6030

Perf index = 20 (util) + 40 (thru) = 60/100

**for random-bal.rep:**

Team Name:implicit first fit

Member 1 :Dave OHallaron:droh

Measuring performance with gettimeofday().

Results for mm malloc:

trace valid util ops secs Kops

0 yes 92% 4800 0.007122 674

Total 92% 4800 0.007122 674

Perf index = 55 (util) + 40 (thru) = 95/100

**for cp-decl-bal.rep:**

Team Name:implicit first fit

Member 1 :Dave OHallaron:droh

Measuring performance with gettimeofday().

Results for mm malloc:

trace valid util ops secs Kops

0 yes 99% 6648 0.010842 613

Total 99% 6648 0.010842 613

Perf index = 60 (util) + 40 (thru) = 100/100

**for coalescing-bal.rep:**

Team Name:implicit first fit

Member 1 :Dave OHallaron:droh

Measuring performance with gettimeofday().

Results for mm malloc:

trace valid util ops secs Kops

0 yes 66% 14400 0.000111129847

Total 66% 14400 0.000111129847

Perf index = 40 (util) + 40 (thru) = 80/100

# 第5章 总结

## 5.1 请总结本次实验的收获

## 5.2 请给出对本次实验内容的建议

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.