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 1190200717

班　　级 1903008

学 生 姓 名 梁浩

指 导 教 师 吴锐

实 验 地 点 G709

实 验 日 期 2021/06/15

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

处理器：Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40GHz
已安装的内存(RAM)：8.00GB(7.81GB 可用)
系统类型：64 位操作系统，基于 x64 的处理器

1.2.2 软件环境

Windows 10 家庭中文版; VirtualBox 6.1; Ubuntu 20.04

1.2.3 开发工具

visual studio

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着一个进程的虚拟内存区域，称为堆。系统之间细节不同但不失通用性，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（向更高的地址）。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程序所分配。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，要么是内存分配器自身隐式执行的。

分配器分为两种基本风格：显式分配器、隐式分配器。

显式分配器：要求应用显式地释放任何已分配的块。

显示分配器的约束条件:

①处理任意的请求序列

②立即相应请求

③只使用堆

④对其块(对齐要求)

⑤不修改已分配的块

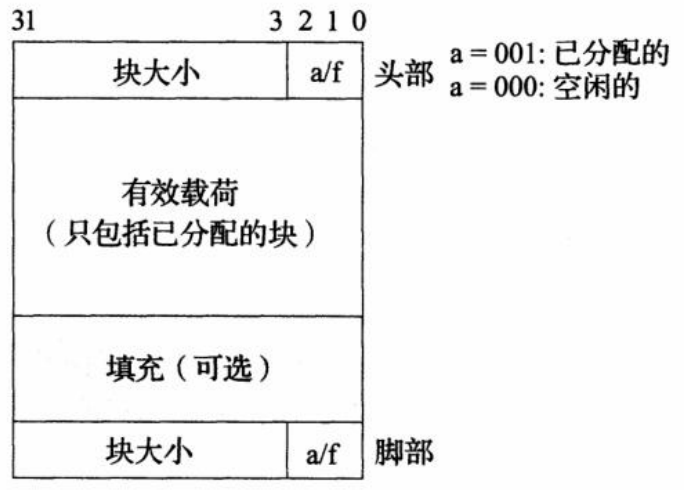
隐式分配器：要求分配器检测一个已分配块何时不再使用，那么就释放这个块，自动释放未使用的已经分配的块的过程叫做垃圾收集。

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

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。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。

使用边界标记的堆块的格式：



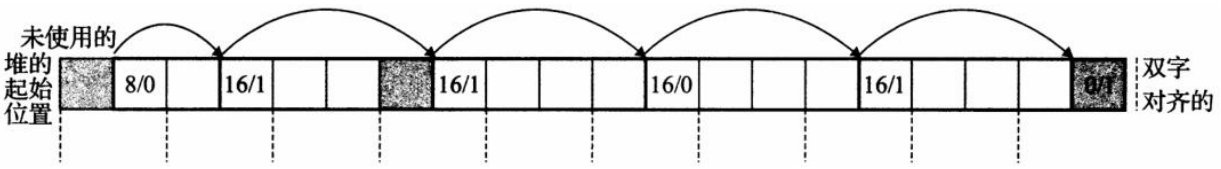
（1）放置空闲块

当一个应用请求一个 k 字节的块时，分配器搜索空闲链表。查找一个足够大可以放置所请求的空闲块。分配器搜索方式的常见策略是首次适配、下一次适配和最佳适配。

首次适配从头开始搜索空闲链表，选择第一个合适的空闲块。下一次适配和首次适配很相似，只不过不是从链表的起始处开始每次搜索，而是从上一次查询结束的地方开始。最佳适配检查每个空闲块，选择适合所需请求大小的最小空闲块。首次适配的优点是它趋向于将大的空闲块保留在链表的后面。缺点是它趋向于在靠近链表起始处留下小空闲块的“碎片”，这就增加了对较大块的搜索时间。

（2）分割空闲块

一旦分配器找到一个匹配的空闲块，它就必须做另一个策略决定，那就是分配这个空闲块中多少空间。一个选择是用整个空闲块。虽然这种方式简单而快捷，但是主要的缺点就是它会造成内部碎片。如果放置策略趋向于产生好的匹配，那么额外的内部碎片也是可以接受的。然而，如果匹配不太好，那么分配器通常会选择将这个空闲块分割为两部分。第一部分变成分配块，而剩下的变成一个新的空闲块。



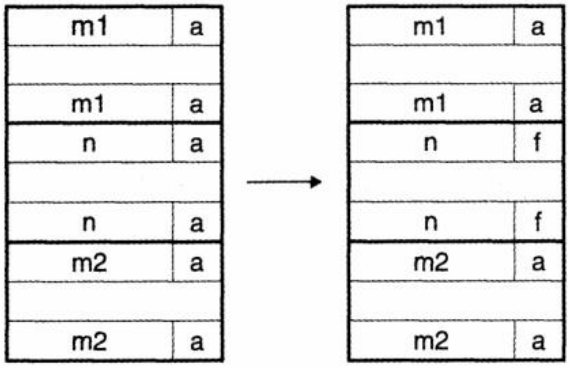
（3）获取额外堆内存

如果分配器不能为请求块找到合适的空闲块，一个选择是通过合并那些在内存中物理上相邻的空闲块来创建一些更大的空闲块，如果这样还是不能生成一个足够大的块，或者如果空闲块已经最大程度地合并了，那么分配器就会通过调用 sbrk 函数，向内核请求额外地堆内存。分配器将额外的内存转化成一个大的空闲块，将这个块插入到空闲链表中，然后将被请求的块放置在这个新的空闲块中。

（4）合并空闲块

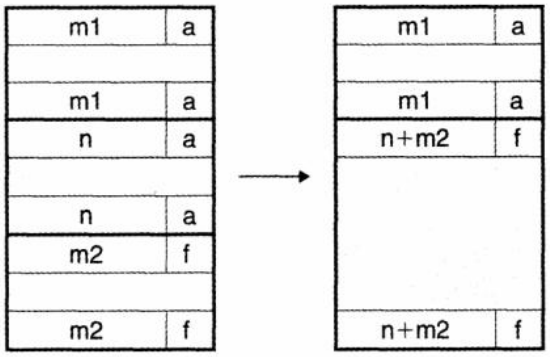
考虑当分配器释放当前块时所有可能存在的情况:

1、前面的块和后面的块都是已分配的。



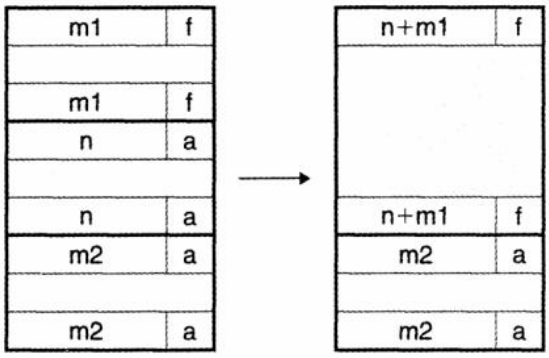
情况 1

2、前面的块是已分配的，后面的块是空闲的。



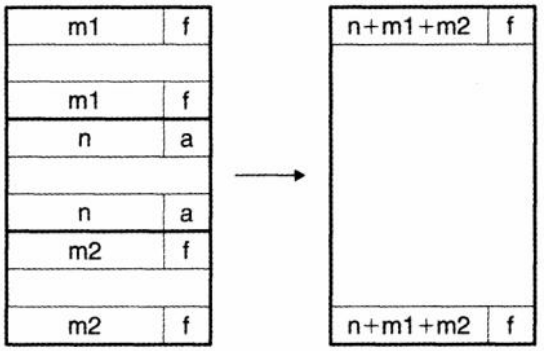
情况 2

3、前面的块是空闲的，而后面的块是已分配的。



情况 3

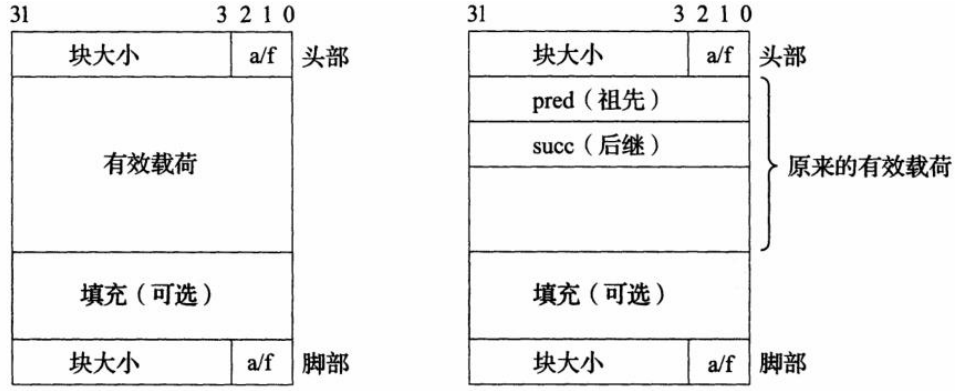
4、前面的和后面的块都是空闲的。



情况 4

2.3 显示空间链表的基本原理（5 分）

将空闲块组织为某种形式的显式数据结构，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 pred（前驱）和 succ（后继）指针，如下图所示：



a）分配块　　b）空闲块

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，这取决于我们所选择的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。使用 LIFO 的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块。在这种情况下，释放一个块可以在常数时间内完成。如果使用了边界标记，那么合并也可以在常数时间内完成。

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。在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于，按照地址排序的首次适配比 LIFO 排序的首次适配有更高的内存利用率，接近最佳适配的利用率。

一般而言，显式链表的缺点是空闲块必须足够大，以包含所有需要的指针，以及头部和可能的脚部。这就导致了更大的最小块大小，也潜在地提高了内部碎片的程度。

2.4 红黑树的结构、查找、更新算法（5 分）

红黑树的结构：

定义：

红黑树是一种自平衡二叉查找树，它在二叉查找树的基础上增加了着色和相关的性质使得红黑树相对平衡，红黑树从根到叶子的最长路径不会超过最短路径的 2 倍，这些特征保证了红黑树的查找、插入、删除的时间复杂度最坏为 O(log n)。

性质：

1、节点是红色或黑色。

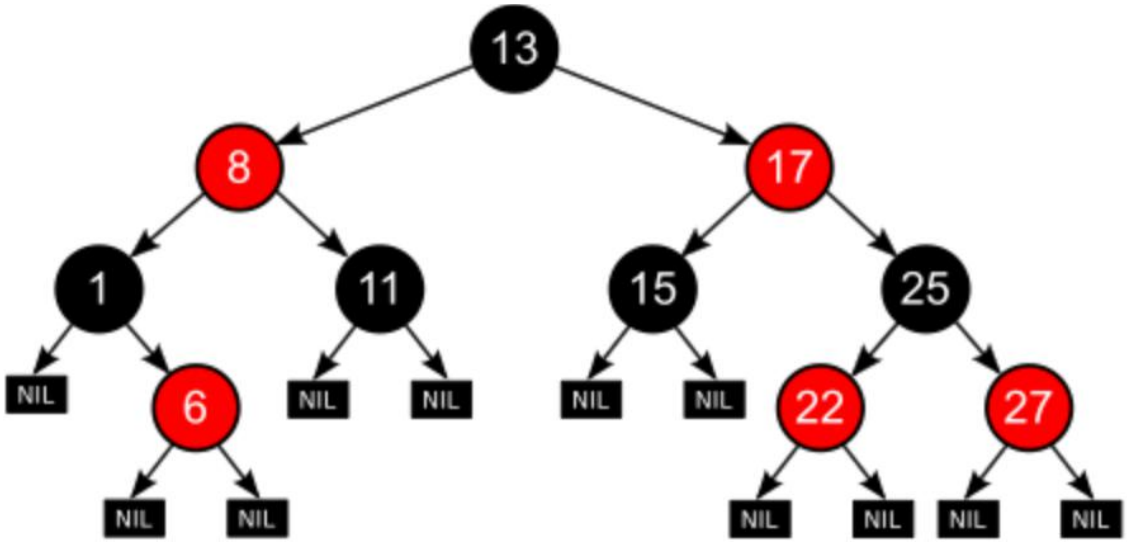
2、根是黑色。

3、所有叶子都是黑色（叶子是 NIL 节点）。

4、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。）

5、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。

举个红黑树的例子：



红黑树的查找：

思路如下：

1）从根结点开始查找，把根结点设置为当前结点

2）若当前结点不为空，用当前结点的 key 和要查找的 key 作比较

3）若当前结点的 key 等于要查找的 key，说明找到了

4）若当前结点的 key 大于要查找的 key，往当前节点的左支走

5）若当前结点的 key 小于要查找的 key，往当前节点的右支走

简单写了些思路：

```
/*
* (非递归实现)查找"红黑树x"中键值为key的节点
*/
template < class T>
RBTNode<T> * RBTree<T>::iterativeSearch(RBTNode<T> * x, T key) const
{
    while ((x != NULL) && (x->key != key))
    {
        if (key < x->key)
            x = x->left;
        else
        x = x->right;
    }
    return x;
}
```

红黑树的更新：

红黑树节点的插入和删除可能会破坏上述红黑树的性质并打破它的平衡，因此需要进行调整从而让红黑树重新恢复平衡。

调整分两种方式：旋转以及变色，旋转又分为左旋转和右旋转两种形式。

变色就是由黑色节点变成红色节点或者红色节点变成黑色节点，如图所示：

左旋转

示意图如图所示：以 P 为旋转支点，旋转支点 P 的右子节点 R 变为父节点，其右子节点 R 的左子节点 RL 变为旋转支点 P 的右子节点；左旋之后左子树的节点相对旋转之前要多出一个节点，也就是左子树从右子树借用一个节点。

代码实现如下：

```
/*************************************************************************/
/* 函数功能：对当前节点进行左旋操作                                      */
// 入口参数：左旋的当前节点
// 返回值：无
/*************************************************************************/
template <class T>
void RB_Tree<T>::Left_Rotate(RB_Tree_Node<T>* current_Node)
{
    RB_Tree_Node<T>* Right_child = current_Node->Right_child;
    RB_Tree_Node<T>* father_Node = current_Node->Father_Node;
    current_Node->Right_child = Right_child->Left_child;
    Right_child->Father_Node = father_Node;
    if (father_Node == NULL)
    {
        Root_Node = Right_child;
    }
    else if(current_Node == father_Node->Left_child)
    {
        father_Node->Left_child = Right_child;
    }
    else
    {
        father_Node->Right_child = Right_child;
    }

    Right_child->Left_child = current_Node;
    current_Node->Father_Node = Right_child;

}
```

右旋转

示意图如图所示：以 R 为旋转支点，旋转支点 R 的左子节点 P 变为父节点，而左子节点 P 的右子节点 RL 变为旋转支点 R 的左子节点；右旋之后右子树的节点相对旋转之前要多出一个节点，也就是右子树从左子树借用一个节点。

代码实现如下：

```
/*************************************************************************/
/* 函数功能：对当前节点进行右旋操作                                      */
// 入口参数：右旋的当前节点
// 返回值：无
/*************************************************************************/
template <class T>
void RB_Tree<T>::Right_Rotate(RB_Tree_Node<T>* current_Node)
{
    RB_Tree_Node<T>* left_Node = current_Node->Left_child;
    RB_Tree_Node<T>* father_Node = current_Node->Father_Node;
    current_Node->Left_child = left_Node->Right_child;
    left_Node->Right_child = current_Node;
    if (father_Node == NULL)
    {
        Root_Node = left_Node;
    }
    else if (current_Node = father_Node->Left_child)
    {
        father_Node->Left_child = left_Node;
    }
    else
    {
        father_Node->Right_child = left_Node;
    }
    current_Node->Father_Node = left_Node;
    left_Node->Father_Node = father_Node;
}
```

向红黑树中插入一个节点：

```
void RB_Tree<T>::Insert_Node(T insert_data)
{

    RB_Tree_Node<T>* temp_Node = Root_Node;
    while(temp_Node != NULL)
    {
        if (insert_data > temp_Node->data)
        {
            if (temp_Node->Right_child == NULL)
            {
                temp_Node->Right_child = new RB_Tree_Node<T>(insert_data);
                temp_Node->Right_child->color_tag = 1;
                temp_Node->Right_child->Father_Node = temp_Node;
                if (temp_Node->color_tag == 1)
                {
                    Fix_Tree(temp_Node->Right_child);
                }
                break;
            }
            else
            {
                temp_Node = temp_Node->Right_child;
            }
        }
        else
        {
            if (temp_Node->Left_child == NULL)
            {
                temp_Node->Left_child  = new RB_Tree_Node<T>(insert_data);
                temp_Node->Left_child->color_tag = 1;
                temp_Node->Left_child->Father_Node = temp_Node;
                if (temp_Node->color_tag == 1)
                {
                    Fix_Tree(temp_Node->Left_child);
                }
                break;
            }
            else
            {
                temp_Node = temp_Node->Left_child;
            }
        }
    }
}
```

向红黑树中删除一个顶点，需要分情况讨论，只截取部分代码：

```
}
if (temp_Node)
//找到返回的数据
{
    int color_tag = temp_Node->color_tag;

    if (temp_Node->Left_child == NULL && temp_Node->Right_child == NULL)
    //左右子树为空则直接删除
    {
        //
        delete temp_Node;
    }
    else
    if (temp_Node->Left_child == NULL && temp_Node->Right_child != NULL)
    //左子树为空,右子树不为空
    {
        if (temp_Node != Root_Node)
        //不为根节点
        {
            if (temp_Node->Father_Node->Left_child == temp_Node)
            {
                temp_Node->Father_Node->Left_child = temp_Node->Right_child;
                temp_Node->Right_child->Father_Node = temp_Node->Father_Node;
```

调整红黑树，使其在插入和删除后仍然保持红黑树的性质，需要分情况讨论，只截取部分代码：

```
    uncle_Node = grandfa_Node->Right_child;
    //如果有叔叔节点时
    if (uncle_Node)
    {
        //情况1 叔叔为红色  将父亲节点和叔叔节点设置为黑色
        //祖父节点设置为红色 将祖父节点设置为当前节点
        if (uncle_Node->color_tag == 1)
        {
            uncle_Node->color_tag = 0;
            father_Node->color_tag = 0;
            grandfa_Node->color_tag = 1;
            temp_current_Node = grandfa_Node;
        }
        //情况2：叔叔是黑色 且当前节点为右孩子 将父节点作为当前节点 对父节点进行左旋
        else if (temp_current_Node == father_Node->Right_child)
        {
            temp_current_Node = temp_current_Node->Father_Node;
            Left_Rotate(temp_current_Node);
        }
        //情况3：叔叔是黑色 且当前节点为左孩子 将父节点设为黑色 祖父节点设为红色 对祖父节点右旋
        else
        {
            father_Node->color_tag = 0;
            grandfa_Node->color_tag = 1;
            Right_Rotate(grandfa_Node);
        }
    }
    //没有叔叔节点时
    else
```

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。

4. 相关函数

int mm_init(void)

void mm_free(void *ptr)函数

void *mm_realloc(void *ptr, size_t size)
int mm_check(void)
void *mm_malloc(size_t 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：
创建一个带初始空闲块的堆

处理流程：
1）mm_init 函数从内存中得到四个字，第一个字是一个双字边界对齐的不使用的填充字。如下图所示：

```
/* Create the initial empty heap */
if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void*)-1)
    return -1;
PUT(heap_listp, 0);                          /* Alignment padding */
```

2）填充字后面紧跟着一个特殊的序言块，它是个 8 字节的已分配块，只由一个头部和一个脚部组成。序言块是在初始化时候创建的，并且永不释放。

```
PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1));  /* Prologue header */
PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1));  /* Prologue footer */
```

3）初始化时不创建普通块
4）堆以一个特殊的结尾块来结束，这块是一个大小为零的已分配块，只由一个头部组成。

```
PUT(heap_listp + (3 * WSIZE), PACK(0, 1));   /* Epilogue header */
```

要点分析：
1. 创建空闲链表之后需要使用 extend_heap 来扩展堆。

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：
指向请求块首字的指针 ptr

处理流程：
1）调用 GET_SIZE(HDRP(bp))来获得请求块的大小

```
size_t size = GET_SIZE(HDRP(bp));
```

2）将请求块的头部和脚部的已分配位置改为 0，表示为释放

```
PUT(HDRP(bp), PACK(size, 0));
PUT(FTRP(bp), PACK(size, 0));
```

3）调用 coalesce(bp)，将释放的块 bp 与相邻的空闲块合并起来

```
coalesce(bp);
```

要点分析：
将请求块的 bp 标记位改为 0 之后，需要调用 coalesce 函数使它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：
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。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。

参　　数：
指向请求块首字的指针 ptr，需要分配的字节 size

处理流程：
1）先调用 mm_malloc 函数申请空闲块，中间会涉及到双字对齐的问题

```
if ((newptr = mm_malloc(size)) == NULL) {
    printf("ERROR: mm_malloc failed in mm_realloc\n");
    exit(1);
}
```

2）获取 ptr 指向的块的大小 oldSize，如果现在的 size < oldSize，那么赋值的时候只需要赋值前 size 个字节

```
oldSize = GET_SIZE(HDRP(ptr));
if (size < oldSize)//如果现在申请的大小<原来的大小
    oldSize = size;//赋值的时候只需要赋值现在的大小那么多
memcpy(newptr, ptr, oldSize);
```

3）释放 ptr，返回 newptr

```
mm_free(ptr);
return newptr;
```

要点分析：
1. 当需要重新分配的大小 size 小于原来的 ptr 指向的块的大小时，赋值时只需要前新块 size 个字节的内容与旧块相同

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 指向序言块，最开始检查序言块，当序言块不是 8 字节的已分配块，就会打印”Bad prologue header”。

```
if ((GET_SIZE(HDRP(heap_listp)) != DSIZE) || !GET_ALLOC(HDRP(heap_listp))) {
    printf("Bad prologue header\n");//当序言块不是8字节的已分配块，就会打印Bad prologue header
}
```

2）检查每块是否双字对齐，头部和脚部是否匹配

```
//1、检查是否双字对齐
if ((size_t)bp % 8)
    printf("Error: %p is not doubleword aligned\n", bp);

//2、获得p所指块的头部和脚部指针，判断两者是否匹配，不匹配的话就返回错误信息
if (GET(HDRP(bp)) != GET(FTRP(bp)))
    printf("Error: header does not match footer\n");
```

3）检查所有 size 大于 0 的块

```
for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {//检查所有size > 0的块
```

4）检查结尾块，当结尾块不是一个大小为 0 的已分配块，就会打印”Bad epilogue header”

要点分析：
1. 主要检查了堆序言块、结尾块以及所有 size 大于 0 的块，检查它们是否双字对齐，头部和脚部是否匹配。

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

函数功能：向内存请求大小为 size 字节的块

参　　数：块大小 size

处理流程：
1）先检测 size 是否为 0，如果为 0，直接返回 NULL

```
if (size == 0)
    return NULL;
```

2）按照格式调整块大小

```
/* Adjust block size to include overhead and alignment reqs. */
if (size <= DSIZE)
    asize = 2 * DSIZE;
else
    asize = DSIZE * ((size + (DSIZE)+(DSIZE - 1)) / DSIZE);
```

3）如果在空闲链表中能找到合适的，直接放置

```
/* Search the free list for a fit*/
if ((bp = find_fit(asize)) != NULL) {
    place(bp, asize);
    return bp;
}
```

4）如果空闲链表中找不到合适的，调用 extend_heap 函数申请更多的空闲块

```
/* No fit found. Get more memory and place the block */
extendsize = MAX(asize, CHUNKSIZE);
if ((bp = extend_heap(extendsize / WSIZE)) == NULL)
    return NULL;
place(bp, asize);
return bp;
```

要点分析：

1. 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空闲块，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：

使用边界标记合并技术使邻接的空闲块合并起来

处理流程：

1）获得前、后块的已分配标记位和 bp 所指向的块的大小

```
size_t  prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));//获得前一个块的已分配标记位

size_t  next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));//获得后一个块的已分配标记位

size_t size = GET_SIZE(HDRP(bp));//获得bp所指向的块的大小
```

2）根据相邻块的分配情况，可以分为以下四种情形：

1. 前后均为 allocated 块，不做合并，直接返回

```
if (prev_alloc && next_alloc)
{
    return bp;
}
```

2. 前面的块是 allocated，但是后面的块是 free 的，这时将两个 free 块合并

```
else if (prev_alloc && !next_alloc)
{
    size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
}
```

3. 后面的块是 allocated，但是前面的块是 free 的，这时将两个 free 块合并

```
else if (!prev_alloc && next_alloc)
{
    size += GET_SIZE(HDRP(PREV_BLKP(bp)));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
    bp = PREV_BLKP(bp);
}
```

4. 前后两个块都是 free 块，这时将三个块同时合并

```
size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(HDRP(NEXT_BLKP(bp)));
PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
bp = PREV_BLKP(bp);
```

3）将更新的 bp 所指向的块插入空闲链表

要点分析：

1. 使用的空闲链表格式允许我们忽略潜在的麻烦边界情况，也就是请求块 bp 在堆的起始处或者堆的结尾处，如果没有这些特殊块，代码将混乱的多，更加容易出错，并且更慢。

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

轨迹文件：指示测试驱动程序 mdriver 以一定顺序调用 mm_malloc, mm_realloc 和 mm_free

性能分 pindex 是空间利用率和吞吐率的线性组合

获得测试总分 linux>./mdriver -av -t traces/

4.2 测试结果评价

由于时间问题并没有进行太多的优化，后面的几个轨迹文件的测试效果较差，导致整体的性能不高，得分较低。

4.3 自测试结果

```
1190200717lh@Graham:~/lab8/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 ftimer.o
1190200717lh@Graham:~/lab8/malloclab-handout$ ./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   99%    5694  0.007022   811
 1       yes   99%    5848  0.005772  1013
 2       yes   99%    6648  0.010346   643
 3       yes  100%    5380  0.007219   745
 4       yes   66%   14400  0.000138104348
 5       yes   92%    4800  0.006274   765
 6       yes   92%    4800  0.005894   814
 7       yes   55%   12000  0.138812    86
 8       yes   51%   24000  0.265781    90
 9       yes   27%   14401  0.061527   234
10       yes   34%   14401  0.002134  6750
Total          74%  112372  0.510919   220

Perf index = 44 (util) + 15 (thru) = 59/100
```

第 5 章 总结

5.1 请总结本次实验的收获

1）了解了红黑树这种数据结构
2）明白了动态内存分配的原理
3）了解了隐式空闲链表和显示空闲链表的差别，它们的优缺点

5.2 请给出对本次实验内容的建议

暂无

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] 深入理解计算机系统