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 计算机系

学　　号 1180300811

班　　级 1803008

学　　生 孙骁

指 导 教 师 吴锐

实 验 地 点 G712

实 验 日 期 2019/12/13

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

1. 理解现代计算机系统虚拟存储的基本知识
2. 掌握 C 语言指针相关的基本操作
3. 深入理解动态存储申请、释放的基本原理和相关系统函数
4. 用 C 语言实现动态存储分配器，并进行测试分析
5. 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

i7-8550U X64 CPU；1.80GHz；16G RAM；1T SSD

1.2.2 软件环境

Windows10 64 位；Vmware 15.1.0；Ubuntu 18.04 LTS

1.2.3 开发工具

Visual Studio 2019 ；CodeBlocks；gcc

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）
2. 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。
3. 熟知 C 语言指针的概念、原理和使用方法
4. 了解虚拟存储的基本原理
5. 熟知动态内存申请、释放的方法和相关函数
6. 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。分配器将堆视为一组不同大小的块的集合维护，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：隐式分配器和显式分配器。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. 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。

2. 显式分配器：要求应用显式地释放任何已分配的块。例如 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。其中 malloc 采用的总体策略是：先系统调用 sbrk 一次，会得到一段较大的并且是连续的空间。进程会在接下来执行的过程中对这段空间进行使用。之后调用 malloc 时就从这段空间中分配，free 回收时就再还回来（而不是还给系统内核）。只有当这段空间全部被分配掉时还不够用时，才再次系统调用 sbrk。

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，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。如果采用一个双字的对齐约束条件，块大小就总是 8 的倍数，且块大小的最低 3 位总是 0。因此，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，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。

将对组织为一个连续的已分配块和空闲块的序列，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。

2.3 显示空间链表的基本原理（5 分）

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。

2.4 红黑树的结构、查找、更新算法（5 分）

一、 红黑树的结构

红黑树是一种近似平衡的二叉查找树，它能够确保任何一个节点的左右子树的高度差不会超过二者中较低那个的一陪。具体来说，红黑树是满足如下条件的二叉查找树（binary search tree）：

1. 每个节点要么是红色，要么是黑色。
2. 根节点必须是黑色
3. 红色节点不能连续（也即是，红色节点的孩子和父亲都不能是红色）。
4. 对于每个节点，从该点至 null（树尾端）的任何路径，都含有相同个数的黑色节点。

在树的结构发生改变时（插入或者删除操作），往往会破坏上述条件 3 或条件 4，需要通过调整使得查找树重新满足红黑树的条件。

二、 红黑树的查找

同二叉查找树 BST

三、 红黑树的插入

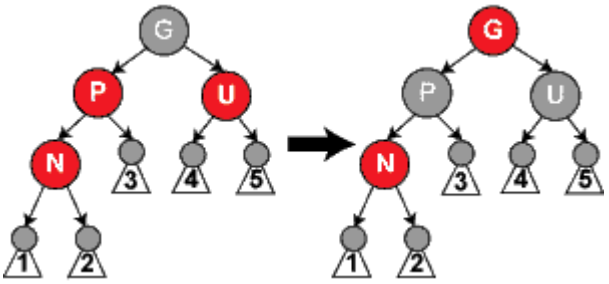
由于性质的约束：插入点不能为黑节点，应插入红节点。因为你插入黑节点将破坏性质 5，所以每次插入的点都是红结点，但是若他的父节点也为红，

那岂不是破坏了性质 4？对啊，所以要做一些“旋转”和一些节点的变色！另为叙述方便我们给要插入的节点标为 N（红色），父节点为 P，祖父节点为 G，叔节点为 U。下边将一一列出所有插入时遇到的情况：

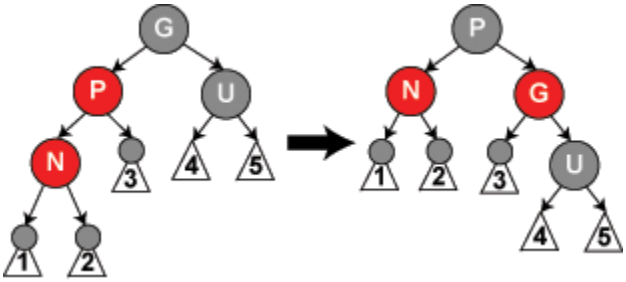
情形 1：该树为空树，直接插入根结点的位置，违反性质 1，把节点颜色有红改为黑即可。

情形 2：插入节点 N 的父节点 P 为黑色，不违反任何性质，无需做任何修改。

情形 3：N 为红，P 为红，（祖节点一定存在，且为黑，下边同理）U 也为红，这里不论 P 是 G 的左孩子，还是右孩子；不论 N 是 P 的左孩子，还是右孩子。



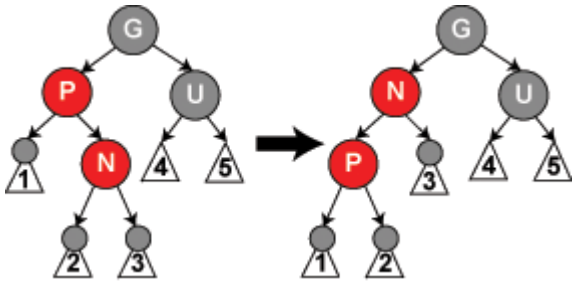
情形 4：N 为红，P 为红，U 为黑，P 为 G 的左孩子，N 为 P 的左孩子（或者 P 为 G 的右孩子，N 为 P 的左孩子；反正就是同向的）。



操作：如图 P、G 变色，P、G 变换即左左单旋（或者右右单旋），结束。

解析：要知道经过 P、G 变换（旋转），变换后 P 的位置就是当年 G 的位置，所以红 P 变为黑，而黑 G 变为红都是为了不违反性质 5，而维持到达叶节点所包含的黑节点的数目不变！还可以理解为：也就是相当于（只是相当于，并不是实事，只是为了更好理解；）把红 N 头上的红节点移到对面黑 U 的头上；这样即符合了性质 4 也不违反性质 5，这样就结束了。

情形 5：N 为红，P 为红，U 为黑，P 为 G 的左孩子，N 为 P 的右孩子（或者 P 为 G 的右孩子，N 为 P 的左孩子；反正两方向相反）。

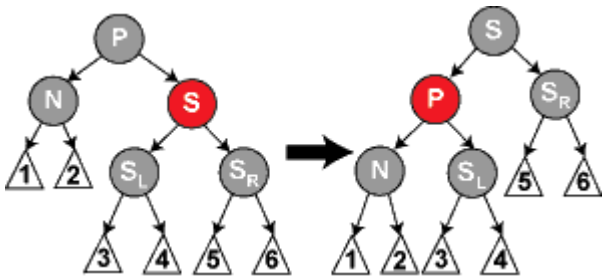


操作：需要进行两次变换（旋转），图中只显示了一次变换-----首先 P、N 变换，颜色不变；然后就变成了情形 4 的情况，按照情况 4 操作，即结束。

解析：由于 P、N 都为红，经变换，不违反性质 5；然后就变成 4 的情形，此时 G 与 G 现在的左孩子变色，并变换，结束。

四、　红黑树的删除

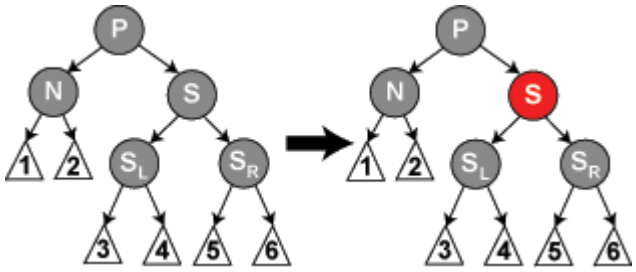
情形 1：S 为红色（那么父节点 P 一定是黑，子节点一定是黑），N 是 P 的左孩子（或者 N 是 P 的右孩子）。



操作：P、S 变色，并交换----相当于 AVL 中的右右中旋转即以 P 为中心 S 向左旋（或者是 AVL 中的左左中的旋转），未结束。

解析：我们知道 P 的左边少了一个黑节点，这样操作相当于在 N 头上又加了一个红节点----不违反任何性质，但是到通过 N 的路径仍少了一个黑节点，需要再把对 N 进行一次检索，并作相应的操作才可以平衡（暂且不管往下看）。

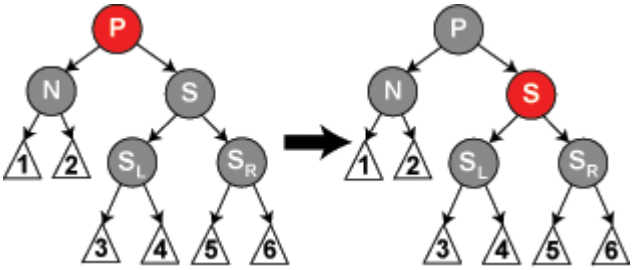
情形 2：P、S 及 S 的孩子们都为黑。



操作：S 改为红色，未结束。

解析：S 变为红色后经过 S 节点的路径的黑节点数目也减少了 1，那个从 P 出发到其叶子节点到所有路径所包含的黑节点数目（记为 num）相等了。但是这个 num 比之前少了 1，因为左右子树中的黑节点数目都减少了！一般地，P 是他父节点 G 的一个孩子，那么由 G 到其叶子节点的黑节点数目就不相等了，所以说没有结束，需把 P 当做新的起始点开始向上检索。

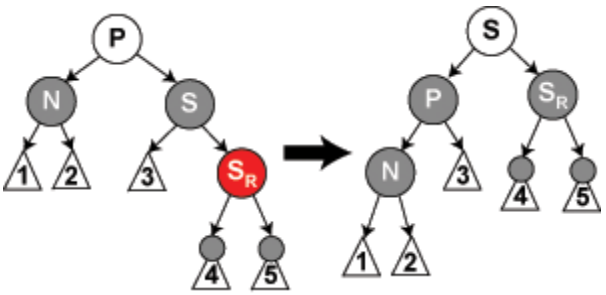
情形 3：P 为红（S 一定为黑），S 的孩子们都为黑。



操作：P 该为黑，S 改为红，结束。

解析：这种情况最简单了，既然 N 这边少了一个黑节点，那么 S 这边就拿出了一个黑节点来共享一下，这样一来，S 这边没少一个黑节点，而 N 这边便多了一个黑节点，这样就恢复了平衡，多么美好的事情哈！

情形 4：P 任意色，S 为黑，N 是 P 的左孩子，S 的右孩子 SR 为红，S 的左孩子任意（或者是 N 是 P 的右孩子，S 的左孩子为红，S 的右孩子任意）。

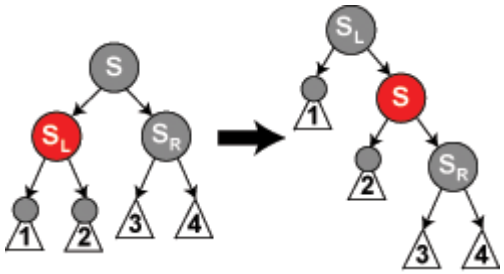


操作：SR（SL）改为黑，P 改为黑，S 改为 P 的颜色，P、S 变换--这里相对应于 AVL 中的右右中的旋转（或者是 AVL 中的左左旋转），结束。

解析：P、S 旋转有变色，等于给 N 这边加了一个黑节点，P 位置（是位置而不是 P）的颜色不变，S 这边少了一个黑节点；SR 有红变黑，S 这边又增加了一个黑节点；这样一来又恢复了平衡，结束。

情形 5：P 任意色，S 为黑，N 是 P 的左孩子，S 的左孩子 SL 为红，S 的

右孩子 SR 为黑（或者 N 是 P 的有孩子，S 的右孩子为红，S 的左孩子为黑）。



操作：SL（或 SR）改为黑，S 改为红，SL（SR）、S 变换；此时就回到了情形 4，SL（SR）变成了黑 S，S 变成了红 SR（SL），做情形 4 的操作即可，这两次变换，其实就是对应 AVL 的右左的两次旋转（或者是 AVL 的左右的两次旋转）。

解析：这种情况如果你按情形 4 的操作的话，由于 SR 本来就是黑色，你无法弥补由于 P、S 的变换（旋转）给 S 这边造成的损失！所以我没先对 S、SL 进行变换之后就变为情形 4 的情况了，何乐而不为呢？

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 C 语言上用的 malloc 和 free,realloc，通过分配堆上的内存给程序，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。
2. 堆中内存块的组织结构：用隐式空闲链表来组织堆，具体组织的算法在 mm_init 函数中。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，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，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. 对于空闲块和分配块链表：采用分离的空闲链表。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。空闲块头部的最后标记三位是 000，每次释放完已占用块之后，对释放的块进行合并，重新组合成更大的空闲块。
4. 放置策略（适配方式）：首次适配（其实有着最佳适配的效果）。malloc 搜索块的时间从所有空的空闲块降低到局部链表的空闲块中，当分到对应的大小类链表的时候，它的空间也会在大小类链表的范围里面，这样使得即使是首次适配也可以是空间利用率接近最佳适配。即：当空闲链表按照块大小递增的顺序排序时，首次适配是选择第一个合适的空闲块，最佳适配是选择所需请求大小最小的空闲块，也是会选择第一个合适的空闲块，后面的块大小递增，不再选择。因此两种适配算法效率近似。

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：初始化内存系统模型（包括初始化分离空闲链表和初始化堆）
处理流程：

```
/*
 * mm_init - initialize the malloc package.
 */
int mm_init(void)
{
    /* create the initial empty heap */
    if ((heap_listp = mem_sbrk(4*WSIZE)) == NULL)
    return -1;
    PUT(heap_listp, 0);                          /* alignment padding */
    PUT(heap_listp+WSIZE, PACK(OVERHEAD, 1));  /* prologue header */
    PUT(heap_listp+DSIZE, PACK(OVERHEAD, 1));  /* prologue footer */
    PUT(heap_listp+WSIZE+DSIZE, PACK(0, 1));   /* epilogue header */
    heap_listp += DSIZE;

    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
    return -1;
    return 0;
}
```

1. 从内存中得到四个字，并且将堆初始化，创建一个空的空闲链表。其中创建一个空的空闲链表分为以下 3 步：
   a) 1.第一个字是一个双字边界对齐不使用的填充字。
   b) 2.填充后面紧跟着一个特殊的序言块，，这是一个 8 字节的已分配块，只有一个头部和一个脚部组成，创建的时候使用 PUT(heap_listp + WSIZE, PACK(OVERHEAD, 1)); PUT(heap_listp + DSIZE, PACK(OVERHEAD, 1)); 两个语句将头部和脚部指向的字中，填充大小并且标记为已分配。
   c) 3.堆的结尾以一个特殊的结尾块来结束，使用 PUT(heap_listp + WSIZE + DSIZE, PACK(0, 1)); 语句来表示这个块是一个大小为零的已分配块。
2. 调用 extend_heap 函数，这个函数将堆扩展 CHUNKSIZE/WSIZE 字节，并且创建初始的空闲块。

要点分析：

初始化分离空闲链表和初始化堆的过程主要是要了解 mm_init 函数初始化分配器时，分配器使用最小块的大小是 16 字节，空闲链表组织成一个隐式空闲链表，固定方法是一个双字边界对齐不使用的填充字+8 字节的序言块+4 字节的结尾块。另外空闲链表创建之后需要使用 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：指向请求块首字的指针 ptr
处理流程：

```
/*
 * mm_free - Freeing a block does nothing.
 */
void mm_free(void *ptr)
{
    size_t size = GET_SIZE(HDRP(ptr));

    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));
    coalesce(ptr);
}
```

1. 通过 GET_SIZE(HDRP(ptr))来获得请求块的大小。并且使用 PUT(HDRP(ptr), PACK(size, 0)); PUT(FTRP(ptr), PACK(size, 0)); 将请求块的头部和脚部的已分配位置为 0，表示为 free 。
2. 调用 coalesce(bp); 将释放的块 bp 与相邻的空闲块合并起来。

要点分析：

注意将 free 后的块与链表前后的块进行合并。

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

函数功能：向 ptr 所指的块重新分配一个具有至少 size 字节的有效负载的块
参　　数：待处理的块第一个字的指针 ptr，需要分配的字节 size
处理流程：

```
/*
 * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
 */
void *mm_realloc(void *ptr, size_t size)
{
    void *newp;
    size_t copySize;

    if ((newp = mm_malloc(size)) == NULL) {
    printf("ERROR: mm_malloc failed in mm_realloc\n");
    exit(1);
    }
    copySize = GET_SIZE(HDRP(ptr));
    if (size < copySize)
      copySize = size;
    memcpy(newp, ptr, copySize);
    mm_free(ptr);
    return newp;
}
```

1. 首先通过已经定义的 mm_malloc 函数，分配 size 大小的新块。
2. 调用 GET_SIZE(HDRP(ptr))函数获得需要重新分配块的大小，并与新需要分配的块的大小进行比较，如果新分配块的大小小于已有块，就更新新的块大小为较小的（即新块的大小）
3. 调用 memcpy 函数，将旧快前 copySize 大小的内容复制到新块中。
4. 返回新块的指针。
5.

要点分析：

在原有块基础上对新块进行重新分配，但是可能存在链表中剩余块的长度小于新块的长度，故设计有待优化。

3.2.4 int mm_check(void)函数（5 分）

函数功能：检查堆的一致性
处理流程：

```
/*
 * mm_checkheap - Check the heap for consistency
 */
void mm_checkheap(int verbose)
{
    char *bp = heap_listp;

    if (verbose)
    printf("Heap (%p):\n", heap_listp);

    if ((GET_SIZE(HDRP(heap_listp)) != DSIZE) || !GET_ALLOC(HDRP(heap_listp)))
    printf("Bad prologue header\n");
    checkblock(heap_listp);

    for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
    if (verbose)
        printblock(bp);
    checkblock(bp);
    }

    if (verbose)
    printblock(bp);
    if ((GET_SIZE(HDRP(bp)) != 0) || !(GET_ALLOC(HDRP(bp))))
    printf("Bad epilogue header\n");
}
```

1. 定义指针 bp，初始化为指向序言块的全局变量 heap_listp。后面的操作大多数都是在 verbose 不为零时执行的。
2. 最初是检查序言块，如果序言块不是 8 字节的已分配块，则会打印 Bad prologue header。
3. 然后是 checkblock 函数

```
static void checkblock(void *bp)
{
    if ((size_t)bp % 8)
    printf("Error: %p is not doubleword aligned\n", bp);
    if (GET(HDRP(bp)) != GET(FTRP(bp)))
    printf("Error: header does not match footer\n");
}
```

检查是否双字对齐，并且通过获得 bp 所指块的头头部和脚部指针，判断二者是否匹配，如果不匹配，则返回错误信息。

4. 检查所有 size 大于 0 的块，如果 verbose 部位零，则执行 printblock 函数，对于 printblock 函数，先获得从 bp 所指的块的头部和脚部分别返回的大小和已分配位，然后打印信息，如果头部返回的大小为 0，则 printf("%p: EOL\n", bp); 之后再分别打印头部和脚部的信息，其中'a' 和 'f' 分别表示 allocated 和 free，对应的是已分配位的信息。

```
static void printblock(void *bp)
{
    size_t hsize, halloc, fsize, falloc;

    hsize = GET_SIZE(HDRP(bp));
    halloc = GET_ALLOC(HDRP(bp));
    fsize = GET_SIZE(FTRP(bp));
    falloc = GET_ALLOC(FTRP(bp));

    if (hsize == 0) {
    printf("%p: EOL\n", bp);
    return;
    }

    printf("%p: header: [%d:%c] footer: [%d:%c]\n", bp,
       hsize, (halloc ? 'a' : 'f'),
       fsize, (falloc ? 'a' : 'f'));
}
```

5. 最后检查结尾块。如果结尾块不是一个大小位零的已分配块，则会打印出 Bad epilogue header 。

要点分析：checkheap 函数主要检查了堆序言块和结尾块，每个 size 大于 0 的块是否双字对齐和头部脚部 match，并且打印了块的头部和脚部的信息。事实上 checkheap 函数只是对堆一致性的简单检查，如空闲块是否都在空闲链表等方面并没有展开检查。

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

函数功能：向内存请求分配一个具有至少 size 字节的有效负载的块。
参　　数：向内存请求块大小 size 字节
处理流程：

```
/*
 * mm_malloc - Allocate a block by incrementing the brk pointer.
 *     Always allocate a block whose size is a multiple of the alignment.
 */
void *mm_malloc(size_t size)
{
    size_t asize;      /* adjusted block size */
    size_t extendsize; /* amount to extend heap if no fit */
    char *bp;

    /* Ignore spurious requests */
    if (size <= 0)
    return NULL;

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

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

    /* 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;
}
```

1. 首先检查需要分配块的大小，负数返回空指针。
2. 如果 size 小于 8，分配 8 加上头部脚部大小，为 16 个字节；超过 8 个字节，头部脚部加上分配的字节，向上舍入到最近的 8 的倍数。

3. 确定分配大小，搜索空闲链表，寻找合适块申请。
4. 如果有合适大小，调用 place 函数放置请求块，并且分割出多余的部分，然后返回新分配块的地址。如果分配器不能够发现一个匹配的块，那么就用一个新的空闲块来扩展堆，同样把请求块放置在这个新的空闲块里，可选地分割这个块，然后返回一个指向这个新分配块的指针。

要点分析：mm_malloc 函数主要是根据 size 大小在分离空闲链表数组中找合适的请求块，如果找不到则用一个新的空闲块来扩展堆。每次都要使用 place 函数放置请求块，并可选地分割出多余的部分。

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

函数功能：边界标记合并。将指针返回到合并块。
处理流程：指向请求块首字的指针 bp

```
/*
 * coalesce - boundary tag coalescing. Return ptr to coalesced block
 */
static void *coalesce(void *bp)
{
   /**/
    size_t  prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
    size_t  next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));
    if (prev_alloc && next_alloc) {
        return bp;
    }
    else if (prev_alloc && next_alloc == NULL) {
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }
    else if (prev_alloc == NULL && 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);
    }
    else {
        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);
    }
    return bp;
}
```

1. size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
   size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
   size_t size = GET_SIZE(HDRP(bp));
   获得前一块和后一块的已分配位，并且获得 bp 所指块的大小。
2. 根据 bp 所指块相邻块的情况，可以得到以下四种可能性：

a) 前面的和后面的块都已分配。此时不进行合并，所以当前块直接返回就可以了。

b) 前面块已分配，后面块空闲。先把当前块和后面块从分离空闲链表中删除。然后此时当前块和后面块合并，用当前块和后面块的大小的和来更新当前块的头部和脚部。具体操作为

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

c) 前面块空闲，后面块已分配。这时和上面类似，先把当前块和前面块从分离链表中删除。然后此时将前面块和当前块合并，用两个块大小的和来更新前面块的头部和当前块的脚部。

```
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);
```

此时 bp 指向前面块

d) 前面块和后面块都空闲。先把前面块、当前块和后面块从分离链表中删除。然后合并所有的三个块形成一个单独的空闲块，用三个块大小的和来更新前面块的头部和后面块的脚部。

```
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);
```

此时 bp 之前前面块

3. 将上述四种操作后更新的 bp 所指的块插入分离空闲链表。

要点分析：

获得了当前块相邻块的情况之后，主要是处理不同的四种情况。合并前首先要把待合并的块从分离空闲链表中删除，合并后注意更新总合并块的头部和脚部，大小为总合并块之和。

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

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

在未调用 realloc 函数的几个测试文件中表现较好，仅是 malloc 和 free，隐式链表和首次适配策略还是比较好。

但是调用 realloc 后，因为在已经 malloc 的块中继续加长，效率很低。采用显示链表或者红黑树可以更加优化，临近期末，还是复习为主了，没有再弄更深的。

4.3 自测试结果

```
demerzel@demerzel-virtual-machine:~/ccode/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  0.008385   679
 1       yes   99%    5848  0.006428   910
 2       yes   99%    6648  0.010991   605
 3       yes  100%    5380  0.009136   589
 4       yes   66%   14400  0.000102140762
 5       yes   92%    4800  0.008420   570
 6       yes   92%    4800  0.007046   681
 7       yes   55%   12000  0.161501    74
 8       yes   51%   24000  0.281623    85
 9       yes   27%   14401  0.063855   226
10       yes   34%   14401  0.002407  5983
Total          74%  112372  0.559894   201

Perf index = 44 (util) + 13 (thru) = 58/100
```

第 5 章 总结

5.1 请总结本次实验的收获

了解了动态内存的分配方式，了解了不同的分配方式，比较清楚地知道了动态内存分配的过程。

5.2 请给出对本次实验内容的建议

无

参考文献

[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.