CSAPP lab5 malloclab实验报告

实验日期：2025.May.28-2025.June.4

> 由于报告在解释思路的同时附上了相应的代码，所以看起来可能页数较多。
> **实验解题思路**部分结构如下：
> **1.1 需要明确的理论**
> **1.2 Version1-隐式空闲链表**
> 空闲块组织、初始与拓展堆、释放与合并块、放置策略与分隔、分配块
> *（对应代码在文件mm1.c）*
> **1.3 Version2-分离空闲链表**
> 空闲块组织、初始与拓展堆、**维护链表的操作**、释放与合并块、放置策略与分隔、分配块
> *（对应代码在文件mm2.c）*
> **1.4 Version3-对realloc的优化**
> 特殊情况处理、原地缩小、原地拓展、最后再分配
> *（该版本为最终代码 在文件mm.c）*

1实验解题思路

1.1 需要明确的理论

1.1.1 性能衡量

实验对于性能的考察有两点：**空间利用率(util)**和**吞吐率(thru)**

- 空间利用率：mm_malloc 或 mm_realloc 函数分配且未被 mm_free 释放的内存与堆的大小的比值。应该找到好的策略使**碎片最小化**，以使该比率尽可能接近 1
- 吞吐率：每**单位时间完成的最大请求数**，即要使时间复杂度尽可能小

1.1.2 空闲块组织结构【store & find】

书上介绍的分配器的空闲块组织结构有三种：**隐式空闲链表、显式空闲链表和分离空闲链表**

- 隐式空闲链表(`implicit free list`)：空闲块和分配块交错存放，没有额外的链表结构来供快速定位空闲块，每次分配都需要**遍历整个堆。**
- 显式空闲链表(`explicit free list`)：在隐式链表的结构基础上，利用空闲块释放后，**额外维护一个由空闲块组成的链表**，每次分配只需要**遍历所有空闲块。**
- 分离空闲链表(`segregated free list`)：在显式空闲链表的基础上，**将空闲块按照大小分成不同的链表**，每次分配时，只需要**遍历大小合适的空闲块链表**（如果没找到的话，继续遍历分类上 size 更大的链表）。

我在**Version1**中实现了**隐式空闲链表**，**Version2**中实现了**分离空闲链表。**

1.1.3 查找空闲块以放置的策略【place & cut】

对于如何选择合适的空闲块分配，书上给出了三种策略：

- **首次适配**(`first_fit`)：从头开始遍历空闲链表，直到找到一个大小合适的空闲块。
- **下一次适配**(`next_fit`)：从上一次查询结束的地方开始搜索选择第一个合适的空闲块
- **最佳适配**(`best_fit`)：从头开始遍历空闲链表，直到找到一个大小最合适的空闲块，即其大小和需要分配的大小差距最小。
  在找到准备放置的空闲块后，为了增大空间利用率（util），还需要考虑剩余部分是否能**分隔**为一个新的空闲块

1.1.4 释放与合并【free & coalesce】

释放某个块后，要让它与相邻的空闲块合并，**何时合并**也是一个重要的决策，书上提供了两种策略：

- **立即合并**(`immediate coalescing`)：在每个块**被释放时**，就立即合并所有相邻的块
- **推迟合并**(`deferred coalescng`)：等到某个稍晚的时候再合并，比如某个分配请求失败时
  对于合并的具体实现方式，书上提出了一种有效的方式，即**边界标记**(`boundary tag`)，并展现了**四种**合并的情况，如图：

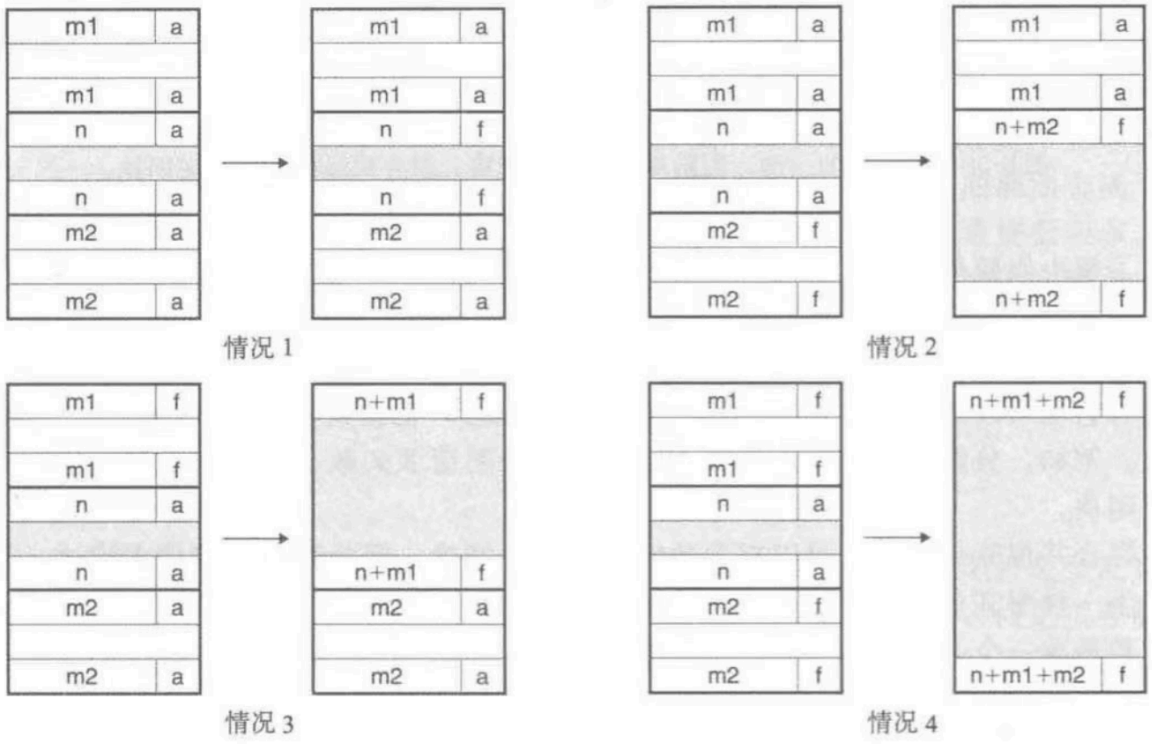


图 9-40　使用边界标记的合并(情况 1：前面的和后面块都已分配。情况 2：前面块已分配，后面块空闲。情况 3：前面块空闲，后面块已分配。情况 4：后面块和前面块都空闲)

我在实验中采取的都是**立即合并**的方式

1.2 Version1-隐式空闲链表

CSAPP书中详细介绍了基于**隐式空闲链表**，使用**立即边界合并**方式的实现，我参考书上的内容又实际实现了一遍。

1.2.1 空闲块组织

单个块结构

- 单个空闲块的结构示意图（截屏自书本）：

图 9-39　使用边界标记的堆块的格式

- 因为需要通过**边界标记**来进行**常数时间**的块合并，所以每个块结尾处都有一个**脚部**，是头部的副本，方便判断前一个块的起始位置和状态
- 脚部与头部均为 **4 个字节（单字）**，用来存储**块的大小**，以及表明这个块是**已分配还是空闲块**
- **有效载荷不为0**且需要考虑**对齐**的情况下，**块的最小大小为4个单字（16字节）**
- 根据单个块的结构可以写出以下**宏定义**来获取块信息

```
/*从地址p读取一个无符号整数*/
#define GET(p) (*(unsigned int *)(p))
/*将val写入地址p*/
#define PUT(p, val) ((*(unsigned int *)(p)) = (val))
/*设置块大小+分配位*/
#define PACK(size, alloc) ((size) | (alloc))
/*获取内存块的大小*/
#define GET_SIZE(p) (GET(p) & ~0x7)
```

```
/*读取p处值的最低位 检查该内存块是否已分配*/
#define GET_ALLOC(p) (GET(p) & 0x1)
/*从指向有效载荷的指针bp，得到指向该内存块的头部指针*/
#define HDRP(bp) ((char *)(bp) - WSIZE)
/*从指向有效载荷的指针bp，得到指向该内存块的脚部指针*/
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
/* 给定有效载荷指针，找到前一块或下一块 */
#define NEXT_BLKP(bp) ((char*)(bp) + GET_SIZE(((char*)(bp) - WSIZE)))
#define PREV_BLKP(bp) ((char*)(bp) - GET_SIZE(((char*)(bp) - DSIZE)))
```

整体隐式空闲链表的堆结构

- 堆结构示意图（截屏自书本）：

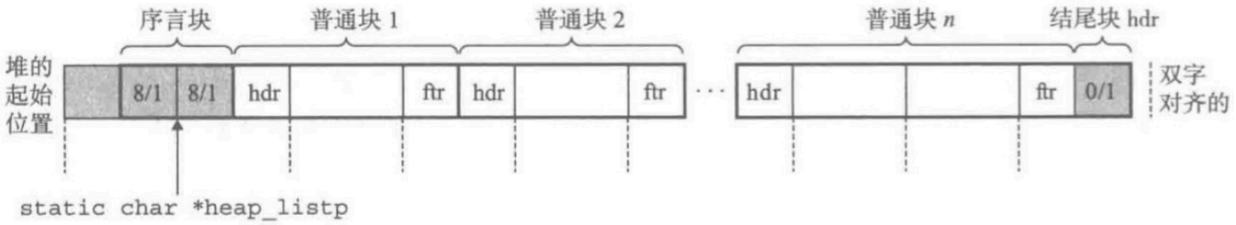


图 9-42　隐式空闲链表的恒定形式

- 堆有两个特殊的标记：
  - **序言块**：8 个字节，由一个头部和一个脚部组成
  - **结尾块**：大小为 0 的头部
- 为了消除合并空闲块时边界的考虑，将序言块和结尾块的分配位均设置为**已分配**。为了保证双字对齐，在序言块的前面还设置了 **4 个字节作为填充。**
- **指针 `heap_listp` 是一个私有的(`static`)全局变量，始终指向序言块之后**

1.2.2 初始与拓展堆

初始化

根据以上**空闲块结构的分析**加上**对系统空间的申请**可以写出**分配器初始化**的代码

- `mem_sbrk` 函数是自定义在 `memlib.c` 中的对于 `sbrk` 函数的模拟，以**字节**为单位**拓展堆的大小（不能收缩）**，并返回**新堆的开头**的地址
- 结合以上对**空闲块组织**的分析以及申请空间的方法，可以写出如下**堆初始化**代码：
  - 首先申请**填充块、序言块和结束块**的空间（四个单字）并对其进行初始化
  - 移动全局变量 `heap_listp`，使其指向序言块之后
  - 拓展空堆，默认大小为 `CHUNKSIZE`，在宏中定义为 `1<<12`

```
int mm_init(void){
    /*填充+序言+结尾*/
    if((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
        return -1; // 申请四个单字的空间（填充+序言+结尾）
```

```
    PUT(heap_listp, 0);                             // 对齐填充
    PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1));    // 序言块头部
    PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1));    // 序言块脚部
    PUT(heap_listp + (3*WSIZE), PACK(0, 1));        // 结尾块头部

    /*移动指针到序言块之后*/
    heap_listp += (2*WSIZE);

    /* 扩展空堆，创建初始空闲块*/
    if((extend_heap(CHUNKSIZE/WSIZE)) == NULL)
        return -1;
    return 0;
}
```

拓展堆

- **拓展堆**会在两种情况被调用：
  - 堆初始化时
  - 当`mm_malloc`不能找到合适的匹配块时，申请更多的内存
- 扩展堆每次在原始堆的尾部申请空间，`mem_sbrk`函数返回指向旧堆尾部的指针，因此，可以**直接将原始堆的尾部位置设置新空闲块的头部。**
- 由于可能出现拓展堆前的块为空闲块，于是调用`coalesce`函数来**合并空闲块**，并指向合并后的块的有效载荷

```
void *extend_heap(size_t words){
    char *bp;//block poiter指向有效载荷
    size_t size = ALIGN(words * WSIZE);//八字节对齐
    if((long)(bp = mem_sbrk(size)) ==-1)//向系统申请size字节大小的内存
        return NULL;

    PUT(HDRP(bp), PACK(size, 0));//写头部
    PUT(FTRP(bp), PACK(size, 0));//写脚部
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));//新的结尾块

    return coalesce(bp);//合并空闲块
}
```

1.2.3 释放与合并块

- 释放块只需要设置一下**头部**和**脚部**即可，把**分配位改成0**
- 由于我们采用**立即合并**的策略，释放后要**合并空闲块**

```
void mm_free(void *ptr){
    if(ptr==0)
        return;
```

```
    size_t size = GET_SIZE(HDRP(ptr));//获取内存块大小
    PUT(HDRP(ptr), PACK(size, 0));//写头部
    PUT(FTRP(ptr), PACK(size, 0));//写脚部
    coalesce(ptr);//合并空闲块 立即合并
}
```

- 合并块要考虑在`1.1.4`中提到的**四种情况**
- 注意需要合并的情况下，**不需要显式地清零原先块的头部和脚部**，因为在**隐式空闲链表**中是靠各个块的**头部信息**来隐式的显示块的排布的，修改了合并后块的头部后，它们已经被合并到一个大的块中，会**被解释为有效载荷**

```
void *coalesce(void *bp){
    /*读取前两个块的分配位*/
    size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    /*当前块大小*/
    size_t size = GET_SIZE(HDRP(bp));

    //四种情况：前后都不空，前不空后空，前空后不空，前后都空
    /* 前后都不空 */
    if(prev_alloc && next_alloc){
        return bp;//直接返回
    }
    /* 前不空后空 */
    else if(prev_alloc && !next_alloc){
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));  //增加当前块大小
        PUT(HDRP(bp), PACK(size, 0));           //先修改头
        PUT(FTRP(bp), PACK(size, 0));           //根据头部中的大小来定位与修改尾
部
    }
    /* 前空后不空 */
    else if(!prev_alloc && next_alloc){
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));  //增加当前块大小
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);                     //注意指针要变
    }
    /* 都空 */
    else{
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    return bp;
}
```

1.2.4 放置策略与分隔

- 为了提高**空间利用率**，放置块时，需要考虑能否讲空闲块进行**分隔**。若是分配后剩余大小大于等于最小快的大小，则可以分隔。
  - 头加脚是8字节,有效载荷不为零,**最小块大小**为16字节(`2*DSIZE`)

```
void place(void *bp, size_t asize) //asize为需分配的大小（已对齐）
{
    size_t csize = GET_SIZE(HDRP(bp));//获取当前空闲块总大小

    /* 判断是否能够分隔空闲块 */
    if((csize - asize) >= 2*DSIZE) { //能分隔
        PUT(HDRP(bp), PACK(asize, 1));//写头部 修改块大小为需分配的大小
        PUT(FTRP(bp), PACK(asize, 1));//写脚部
        bp = NEXT_BLKP(bp);//移动到分隔出的空闲块
        PUT(HDRP(bp), PACK(csize - asize, 0));//写头部 修改块大小为剩余的大小
        PUT(FTRP(bp), PACK(csize - asize, 0));//写脚部
    }
    else{//不能分隔
        PUT(HDRP(bp), PACK(csize, 1));//整个空闲块直接改为已分配
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

- **如何选择放置块**有以下两种
  - **首次适配**：找到的第一给能够放置的块
  - **最佳适配**：遍历所有空闲块，找到能放置的块中最小的

```
void* find_first_fit(size_t asize){//首次适配
    void *bp;
    for(bp=heap_listp;GET_SIZE(HDRP(bp))>0;bp=NEXT_BLKP(bp)){
        if(GET_ALLOC(HDRP(bp))==0&&GET_SIZE(HDRP(bp))>=asize){
            return bp;
        }
    }
    return NULL;
}
```

```
void* find_best_fit(size_t asize){//最佳适配
    void *bp;
    void *best_bp = NULL;
    for(bp=heap_listp;GET_SIZE(HDRP(bp))>0;bp=NEXT_BLKP(bp)){
        if(GET_ALLOC(HDRP(bp))==0&&GET_SIZE(HDRP(bp))>=asize){
            if(best_bp==NULL||GET_SIZE(HDRP(bp))<GET_SIZE(HDRP(best_bp))){
                best_bp = bp;
```

```
            }
        }
    }
    return best_bp;
}
```

- 利用宏来选择的放置策略，方便切换与调试

```
#define find_fit find_first_fit
```

- 运行结果发现其他代码不变的情况下，`find_first_fit`（45+19）比`find_best_fit`（45+18）在空间吞吐量上高了一分

```
Perf index = 45 (util) + 19 (thru) = 64/100
```

```
Perf index = 45 (util) + 18 (thru) = 63/100
```

1.2.5 分配块

最后是主体部分 mm_malloc 函数，对申请的空间大小按进行8字节**对齐**，然后**根据放置策略查找有无合适的空闲块**，如果没有则申请**扩展堆**，然后在对应的空闲块进行**放置**

```
void *mm_malloc(size_t size){
    if(size == 0) return NULL;
    size_t asize = ALIGN(size + DSIZE);//对齐
    size_t extendsize;
    char *bp;

    /* 寻找合适的空闲块 */
    if((bp = find_fit(asize)) == NULL){//找不到则拓展堆
        //!extendsize = MAX(asize, CHUNKSIZE); //拓展chunksize
        extendsize=asize;//直接拓展相应大小也可以
        if((bp = extend_heap(extendsize/WSIZE)) == NULL)
            return NULL;
    }
    /*放置*/
    place(bp, asize);
    return bp;
}
```

- 注意拓展堆时可以只**拓展相应的大小**，也可**拓展默认的**`CHUNKSIZE`，测试下来在所给的`traces`文件上性能没什么差异

1.2.6 Version1 测试结果

- 对应的代码在文件`mm1.c`
- 采用**首次适配**，且在找不到空闲块时**只拓展所需块大小**
- `realloc`采取的就是原先代码的方式，分配一块新的内存，然后将原先内容复制到新内存，然后释放原先块

```
wcx@LAPTOP-OMTCB5PG:~/lab5/malloclab-stu/malloclab-handout$ ./mdriver -v
Team Name:IMSB
Member 1 :Harry Bovik:bovik@cs.cmu.edu
Using default tracefiles in /home/wcx/lab5/malloclab-stu/traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.005719   996
 1       yes  100%    5848  0.006444   908
 2       yes   99%    6648  0.008977   741
 3       yes  100%    5380  0.006080   885
 4       yes   66%   14400  0.000128112237
 5       yes   92%    4800  0.006893   696
 6       yes   92%    4800  0.005934   809
 7       yes   55%   12000  0.075894   158
 8       yes   51%   24000  0.278447    86
 9       yes   32%   14401  0.031596   456
10       yes   34%   14401  0.001210 11898
Total          75%  112372  0.427322   263

Perf index = 45 (util) + 18 (thru) = 62/100
```

只有刚及格的分数，需要使用更好的空闲块组织方式！

1.3 Version2-分离空闲链表

1.3.1 空闲块组织

- 分离空闲链表相比显示空闲链表（只维护一个双向空闲链表）需要维护**多个空闲链表**，每个链表大小大致相等，叫做**等价类(size class)**
- 在空闲块的主体里面加入指向前一个空闲块(`pred`)和后一个(`succ`)空闲块的指针，实现一个**双向链表**，分配块与空闲块的结构如下：

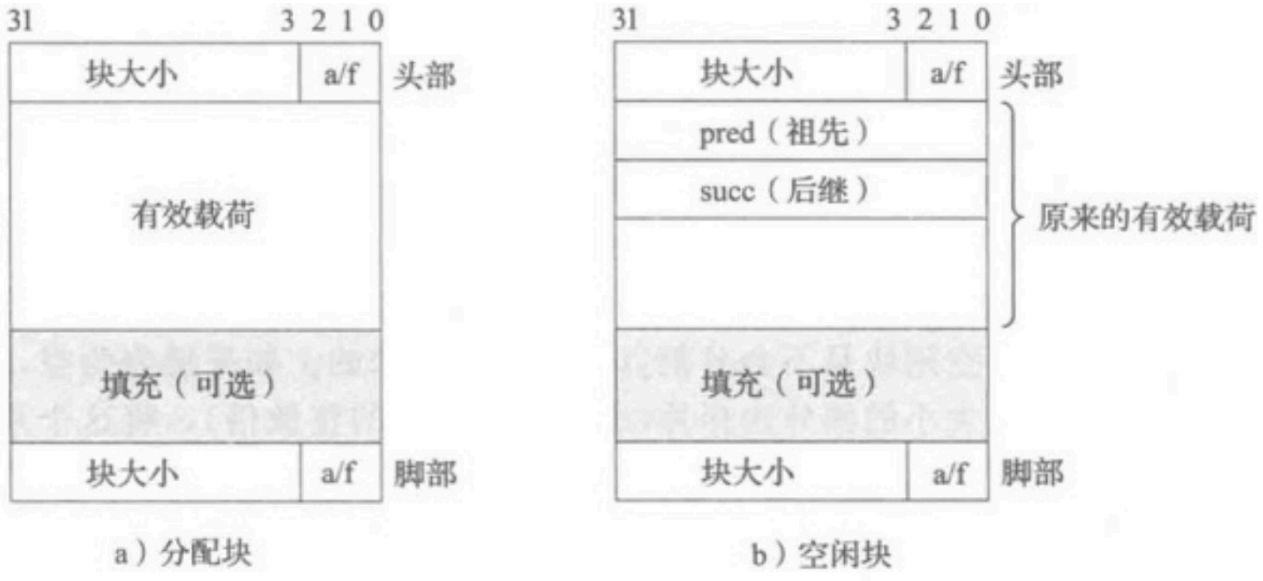
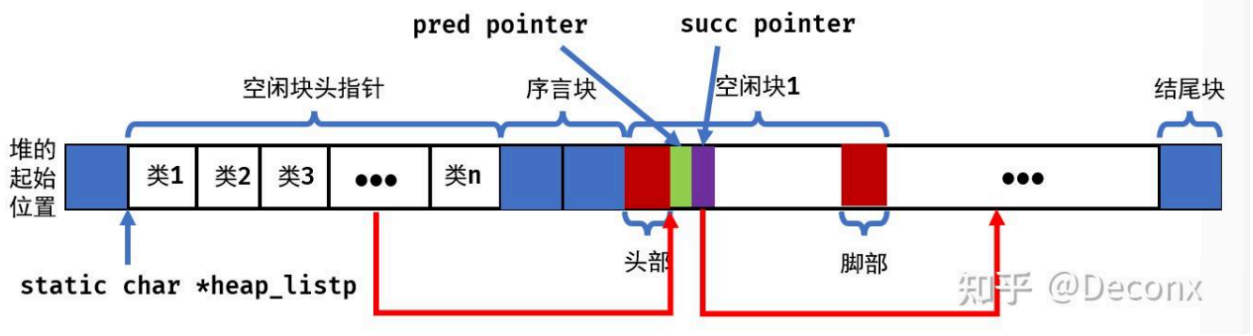


图 9-48　使用双向空闲链表的堆块的格式

- 分离存储方式有很多，书中主要介绍了以下两种：
  - **简单分离存储**：每个空闲链表包含**大小相等**的块，不分隔，不合并
  - **分离适配**：每个空闲链表包含**大小不同**的块

  我在Version2中采用**分离适配**的方式，堆的结构如图所示，在序言块之前放置了不同等价类空闲块的头指针。



- **注意 `heap_listp` 的位置指向链表数组的开头**，不是序言块之后！
- 根据链表的结构特点，增加了几个操作宏：

```
/* 给定序号，找到链表头节点位置 */
#define GET_HEAD(num) ((unsigned int *)(long)(GET(heap_listp + WSIZE *
num)))
/* 给定bp,找到前驱和后继 */
#define GET_PRE(bp) ((unsigned int *)(long)(GET(bp)))
#define GET_SUC(bp) ((unsigned int *)(long)(GET((unsigned int *)bp + 1)))
/* 大小类的数量 */
#define CLASS_SIZE 20
```

1.3.2 初始与拓展堆

初始化

- 初始化函数与Version1不同的地方在于增加了空闲链表数组中20个**大小类头指针的初始化**

- 同时**填充块、序言块、结尾块**的位置向后移了`CLASS_SIZE*WSIZE`

```
/* 初始化大小类数组 */
    for(int i=0;i<CLASS_SIZE;i++){
        PUT(heap_listp+i*WSIZE,NULL);//定义为null比定义为0安全一点
    }

/* 填充+序言+结尾 */
    PUT(heap_listp+CLASS_SIZE*WSIZE, 0);// 填充
    PUT(heap_listp + ((1+CLASS_SIZE)*WSIZE), PACK(DSIZE, 1));//序言块头部
    PUT(heap_listp + ((2+CLASS_SIZE)*WSIZE), PACK(DSIZE, 1));//序言块脚部
    PUT(heap_listp + ((3+CLASS_SIZE)*WSIZE), PACK(0, 1));//结尾块
```

拓展堆

拓展堆函数与Version1中一致

1.3.3 维护链表的操作

根据请求块大小找到对应大小类的头节点(search)

- 因为块**最小大小为16字节**，而头结点位置下标是从 0 开始，所以返回 `i-4`
- `i`遍历到24(`i+CLASS_SIZE`)其实没有必要，只需遍历到22即可，因为**大多数内存分配器只需要处理到 2^22 字节（4MB）的块大小**

```
int search(size_t size){
    int i;
    for(i=4;i<24;i++){
        if(size<=(1<<(i))){//因为块最小大小为16字节
            return i-4;
        }
    }
    return i-4;
}
```

向双向链表中插入空闲块(insert)

- 注意 `heap_list + WSIZE * num` 对应大小类**头节点**在堆中的位置，而 `GET_HEAD(num)` 是大小类头结点**存放的第一个块的地址**
- 具体流程为：
  - 根据空闲块的大小找到对应的大小类
  - 判断该大小类是否为空
    - 若**为空**：设置为链表头
    - 若**不为空**：插入到链表头

```
void insert(void* bp){
    if (bp == NULL) return;
    size_t size=GET_SIZE(HDRP(bp));
    int class=search(size);//找到属于的类

    if(GET_HEAD(class)==NULL){//为头
        PUT(heap_listp+class*WSIZE,bp);//设置为链表头
        PUT(bp,NULL);//前驱
        PUT((unsigned int*)bp+1,NULL);//后继
    }else{//否则插入到头节点
        PUT(bp,NULL);//bp前驱
        PUT((unsigned int*)bp+1,GET_HEAD(class));//bp后继为原链表头
        PUT(GET_HEAD(class),bp);//原来头的前驱设为bp
        PUT(heap_listp+class*WSIZE,bp);//将bp设为链表头
    }
}
```

删除链表中的块(delete)

- 我原先一直段错误，参考了网上做法后，决定还是**分为四种来处理**会清晰一点：
  - 后继为null,前驱为null（**唯一节点**）
  - 后继为null,前驱不为null（**最后一个节点**）
  - 后继不为null,前驱为null（**第一节点**）
  - 后继前驱均不为null（**中间节点**）
- 注意指针的问题。比如：`GET_PRE(bp) + 1`是`bp`指向的块的前驱的后继的**位置**；而`GET_PRE(bp+1)`是`bp`指向的块的后继

```
void delete(void* bp){
    if (bp == NULL) return;
    size_t size=GET_SIZE(HDRP(bp));
    int class=search(size);
    /*
     * 唯一节点,后继为null,前驱为null
     * 头节点设为null
     */
    if (GET_PRE(bp) == NULL && GET_SUC(bp) == NULL) {
        PUT(heap_listp + WSIZE * class, NULL);
    }
    /*
     * 最后一个节点
     * 前驱的后继设为null
     */
    else if (GET_PRE(bp) != NULL && GET_SUC(bp) == NULL) {
        PUT(GET_PRE(bp) + 1, NULL);
    }
    /*
```

```
     * 第一个结点
     * 头节点设为bp的后继
     */
    else if (GET_SUC(bp) != NULL && GET_PRE(bp) == NULL){
        PUT(heap_listp + WSIZE * class, GET_SUC(bp));
        PUT(GET_SUC(bp), NULL);
    }
    /*
     * 中间结点
     * 前驱的后继设为后继
     * 后继的前驱设为前驱
     */
    else if (GET_SUC(bp) != NULL && GET_PRE(bp) != NULL) {
        PUT(GET_PRE(bp) + 1, GET_SUC(bp));
        PUT(GET_SUC(bp), GET_PRE(bp));
    }
```

1.3.4 释放与合并块

释放块

释放块的函数与Version1中一致，注意**释放完立即合并**(`coalesce`)，将新释放出来的空闲块 `insert` 进入空闲链表的操作就在 `coalesce` 里。

合并块

- 同样是考虑四种情况：前后都不空, 前不空后空, 前空后不空, 前后都空
  - 四种情况最后都要 `insert` 进空闲链表
    - 注意**前后都不空的情况**也不要忘了 `insert` !
    - 注意**前空**的情况要修改 `bp` ，方便正确 `insert`
  - 其他情况的操作均需要合并，相较Version1多了一个**把原先空闲块从链表中** `delete` **的操作**

```
void *coalesce(void *bp){
    /*读取前两个块的分配位*/
    size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    /*当前块大小*/
    size_t size = GET_SIZE(HDRP(bp));//当前块大小

    //四种情况：前后都不空，前不空后空，前空后不空，前后都空
    /* 前后都不空 */
    if(prev_alloc && next_alloc){
        insert(bp);
        return bp;
    }
    /* 前不空后空 */
```

```
    else if(prev_alloc && !next_alloc){
        delete(NEXT_BLKP(bp));//相比较隐式多了从链表中删除
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));//增加当前块大小
        PUT(HDRP(bp), PACK(size, 0));//修改头部
        PUT(FTRP(bp), PACK(size, 0));//根据头部中的大小来定位并修改尾部
    }
    /* 前空后不空 */
    else if(!prev_alloc && next_alloc){
        delete(PREV_BLKP(bp));
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));  //增加当前块大小
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);//注意bp要变 方便后面insert
    }
    /* 都空 */
    else{
        delete(PREV_BLKP(bp));
        delete(NEXT_BLKP(bp));
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
GET_SIZE(HDRP(NEXT_BLKP(bp)));  //增加当前块大小
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    insert(bp);
    return bp;
}
```

1.3.5 放置策略与分隔

放置策略

- CSAPP的书上写道：”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“，因此我们就采用简单的**首次适配**搜索
- 具体流程如下：
  - 先从**对应的大小类的空闲链表**中查找
  - 如果找不到，则到**下一个更大的大小类**查找
  - 如果都找不到，放回 `NULL` ，会在 `mm_malloc` 中**扩展堆**

```
void* find_fit(size_t size){
    int class=search(size);//先找对应的大小类
    while(class < CLASS_SIZE){
        unsigned int *cur = GET_HEAD(class);
        while (cur != NULL){
            if(GET_SIZE(HDRP(cur)) >= size){
                return cur;//找到了
            }else{
```

```
                cur = GET_SUC(cur);//找下一个空闲块
            }
        }
        class++;//向下一个大小类找
    }
    return NULL;//所有大小类链表中都找不到
}
```

分隔

- 操作与Version1几乎相同，额外的操作就是
  - 把原空闲块从链表中**删除**
  - 把新空闲块**插入**进链表

```
void place(void *bp, size_t asize) //asize为请求分配的大小
{
    size_t csize = GET_SIZE(HDRP(bp));//获取当前空闲块的总大小
    delete(bp);//当前空闲块不再空闲

    /* 判断是否能够分隔空闲块 */
    if((csize - asize) >= 2*DSIZE) { //能分隔
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        bp = NEXT_BLKP(bp);//移动到下一个内存块
        PUT(HDRP(bp), PACK(csize - asize, 0));
        PUT(FTRP(bp), PACK(csize - asize, 0));
        insert(bp);//将分离出的空闲块插入到空闲链表
    }
    else{//不能分隔
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

1.3.6 分配块

分配块函数与Version1中完全一致，就是先**对齐**再**寻找合适的空闲块**，寻找不到就**拓展堆**，然后进行**放置**

1.3.7 Version2 测试结果

- 对应的代码在文件`mm2.c`

- `realloc` 采取的还是原先代码的方式

```
wcx@LAPTOP-OMTCB5PG:~/lab5/malloclab-stu/malloclab-handout$ ./mdriver -v
Team Name:IMSB
Member 1 :Harry Bovik:bovik@cs.cmu.edu
Using default tracefiles in /home/wcx/lab5/malloclab-stu/traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   97%    5694  0.000626  9091
 1       yes   98%    5848  0.000344 16995
 2       yes   98%    6648  0.000438 15195
 3       yes   99%    5380  0.000348 15446
 4       yes   66%   14400  0.000892 16149
 5       yes   93%    4800  0.000471 10195
 6       yes   90%    4800  0.000458 10476
 7       yes   55%   12000  0.000794 15117
 8       yes   51%   24000  0.001228 19549
 9       yes   23%   14401  0.034723   415
10       yes   29%   14401  0.002040  7059
Total          73%  112372  0.042361  2653

Perf index = 44 (util) + 40 (thru) = 84/100
```

- 发现分数提升了不少！吞吐量已经达到了满分！但是后两个对于 `realloc` 的 `trace` 得分还是较低

1.4 Version3-对realloc的优化

在Version3中我试图对realloc进行优化，原来的 `mm_realloc` 实现非常简单，但效率低下：

```
void *mm_realloc(void *ptr, size_t size) {
    void *newptr = mm_malloc(size); //直接重新申请新内存
    if (newptr == NULL) return NULL;
    size_t copysize = GET_SIZE(HDRP(ptr));
    if (size < copysize) copysize = size;
    memcpy(newptr, ptr, copysize); //将原地址的内容复制到新内存
    mm_free(ptr); //释放原分配块
    return newptr;
    }
```

这种实现存在以下问题：

1. **总是分配新内存块**，即使原内存块可以直接使用
2. 总是进行内存**复制**，即使不必要
3. 不考虑**相邻空闲块合并**的可能
4. 内存利用率低，容易产生**碎片**

1.4.1 特殊情况处理

首先对于**原先指针为空**(`realloc`相当于`malloc`)及**重分配大小为0**(相当于`free`)的情况进行特殊处理

```
/* 特殊情况处理 */
if (ptr == NULL)
    return mm_malloc(size);
if (size == 0) {
    mm_free(ptr);
    return NULL;
}
```

1.4.2 原地缩小

- 当请求的新大小小于或等于原先块大小时，可以直接使用原先块
  - 如果剩余空间足够大，可以进行**分隔**
  - 剩余空间不够大就**直接返回原地址**
- 这样可以避免不必要的内存复制，并且当剩余空间足够大时，将多余部分分割成新的空闲块，可以提高内存利用率

```
/* 如果新大小小于等于旧大小，可以直接使用当前块 */
if (newsize <= oldsize) {
    /* 如果剩余空间足够大，可以分割 */
    if (oldsize - newsize >= 2*DSIZE) {
        PUT(HDRP(ptr), PACK(newsize, 1));
        PUT(FTRP(ptr), PACK(newsize, 1));
        void *next_block = NEXT_BLKP(ptr);
        PUT(HDRP(next_block), PACK(oldsize - newsize, 0));
        PUT(FTRP(next_block), PACK(oldsize - newsize, 0));
        insert(next_block);
    }
    return ptr;
}
```

1.4.3 原地拓展

向前拓展

- 如果当前块的**前一个块是空闲的**，且**合并后大小足够**，可以向前扩展
- 这一策略的优点是：
  - 利用了前面的空闲块，减少了内存碎片
  - 使用`memmove`而不是`memcpy`，避免了内存重叠问题`memmove(prev_block, ptr, oldsize - DSIZE)`表示从地址`ptr`开始的`oldsize - DSIZE`字节的数据复制到地址`prev_block`

```
size_t prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(ptr)));//读取前一个块是否分配
size_t prev_size = GET_SIZE(HDRP(PREV_BLKP(ptr)));//读取前一个块大小

if (!prev_alloc && (prev_size + oldsize >= newsize)) {
    void *prev_block = PREV_BLKP(ptr);
    delete(prev_block);
    /* 复制数据到新位置 - 使用memmove避免内存重叠问题 */
    memmove(prev_block, ptr, oldsize - DSIZE);
    PUT(HDRP(prev_block), PACK(prev_size + oldsize, 1));
    PUT(FTRP(prev_block), PACK(prev_size + oldsize, 1));
    /* 如果合并后空间足够大，可以分割 */
    if (prev_size + oldsize - newsize >= 2*DSIZE) {
        PUT(HDRP(prev_block), PACK(newsize, 1));
        PUT(FTRP(prev_block), PACK(newsize, 1));
        void *next_block = NEXT_BLKP(prev_block);
        PUT(HDRP(next_block), PACK(prev_size + oldsize - newsize, 0));
        PUT(FTRP(next_block), PACK(prev_size + oldsize - newsize, 0));
        insert(next_block);
    }
    return prev_block;
    }
```

向后拓展

- 与**向前拓展**同理，如果当前块的**后一个块是空闲的**，且**合并后大小足够**，可以直接合并这两个块
- 向后拓展**不需要移动原数据**

```
/* 检查下一个块是否空闲且合并后大小足够 */
if (!next_alloc && (oldsize + next_size >= newsize))
{
    delete(NEXT_BLKP(ptr));
    PUT(HDRP(ptr), PACK(oldsize + next_size, 1));
    PUT(FTRP(ptr), PACK(oldsize + next_size, 1));
    /* 如果合并后空间足够大，可以分割 */
    if (oldsize + next_size - newsize >= 2*DSIZE) {
        PUT(HDRP(ptr), PACK(newsize, 1));
        PUT(FTRP(ptr), PACK(newsize, 1));
        void *next_block = NEXT_BLKP(ptr);
        PUT(HDRP(next_block), PACK(oldsize + next_size - newsize, 0));
        PUT(FTRP(next_block), PACK(oldsize + next_size - newsize, 0));
        insert(next_block);
    }
    return ptr;
}
```

双向拓展

- 如果当前块的**前后两个块都是空闲的**，且**三者合并后大小足够**，可以同时合并这三个块
  - **检查**是否前后快空闲且合并后大小足够
  - **删除**前后原空闲块
  - **复制**数据到新位置(使用 `memmove`)
  - 若合并后剩余空间足够，可以**分隔**，将剩余块**插入**为新空闲块

```
/* 检查前后块都空闲且合并后大小足够 */
if (!prev_alloc && !next_alloc && (prev_size + oldsize + next_size >=
newsize))
{
    void *prev_block = PREV_BLKP(ptr);
    delete(prev_block);
    delete(NEXT_BLKP(ptr));
    /* 复制数据到新位置 */
    memmove(prev_block, ptr, oldsize - DSIZE);
    PUT(HDRP(prev_block), PACK(prev_size + oldsize + next_size, 1));
    PUT(FTRP(prev_block), PACK(prev_size + oldsize + next_size, 1));
    /* 如果合并后空间足够大，可以分割 */
    if (prev_size + oldsize + next_size - newsize >= 2*DSIZE) {
        PUT(HDRP(prev_block), PACK(newsize, 1));
        PUT(FTRP(prev_block), PACK(newsize, 1));
        void *next_block = NEXT_BLKP(prev_block);
        PUT(HDRP(next_block), PACK(prev_size + oldsize + next_size -
newsize, 0));
        PUT(FTRP(next_block), PACK(prev_size + oldsize + next_size -
newsize, 0));
        insert(next_block);
    }
    return prev_block;
}
```

1.4.4 最后再分配新块

- 如果以上所有策略都不可行，才分配新块并复制数据
- 复制数据的时候**减去头部和脚部**，只复制实际需要的数据量

```
/* 如果以上策略都不可行，则分配新块并复制数据 */
newptr = mm_malloc(size);
if (newptr == NULL)
    return NULL;
size_t copySize = oldsize - DSIZE; // 减去头部和脚部 只复制实际需要的数据量
if (size < copySize) copySize = size;
memcpy(newptr, ptr, copySize);
```

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

1.4.5 Version3 测试结果

- 对应代码在 `mm.c`
- 除了 `realloc` 以外其余部分与Version2一致

```
wcx@LAPTOP-OMTCB5PG:~/lab5/malloclab-stu/malloclab-handout$ ./mdriver -v
Team Name:IMSB
Member 1 :Harry Bovik:bovik@cs.cmu.edu
Using default tracefiles in /home/wcx/lab5/malloclab-stu/traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   97%    5694  0.000852  6685
 1       yes   98%    5848  0.000768  7619
 2       yes   98%    6648  0.000803  8282
 3       yes   99%    5380  0.000667  8062
 4       yes   66%   14400  0.001288 11183
 5       yes   93%    4800  0.001040  4615
 6       yes   90%    4800  0.000989  4854
 7       yes   55%   12000  0.001235  9717
 8       yes   51%   24000  0.001832 13103
 9       yes   49%   14401  0.003217  4476
10       yes   45%   14401  0.001036 13902
Total           77%  112372  0.013726  8187

Perf index = 46 (util) + 40 (thru) = 86/100
```

发现重写 `realloc` 后分数确实提高了（虽然不多）！就优化到这里吧，**完工**!

2实验中遇到和解决的问题

2.1 可恶的段错误

问题

写代码时觉得思路都没有问题，结果一运行就是 `segmentation fault`，并且对于问题出在哪毫无头绪

解决

方法1：利用gdb定位发生段错误的具体位置

- 我按照网上的说法把 `makefile` 中的 `CFLAGS` 中的 `-O2` 删掉，并且加上了 `-g` 的选项，然后用 `gdb` 运行 `mdriver` 就可以定位到段错误，如图：

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from mdriver...
(gdb) r
Starting program: /home/wcx/lab5/malloclab-stu/malloclab-handout/mdriver
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Team Name:IMSB
Member 1 :Harry Bovik:bovik@cs.cmu.edu
Using default tracefiles in /home/wcx/lab5/malloclab-stu/traces/

Program received signal SIGSEGV, Segmentation fault.
0x56558841 in delete (bp=0xf6a56ff8) at mm.c:189
189             PUT(GET_PRE(bp) + 1, GET_SUC(bp));
(gdb)
```

方法2：利用 `printf` 打印地址信息

当时在写Version2的 `delete` 函数时，由于指针操作复杂发生了段错误，于是我像下面这样添加了打印信息，来发现错误

```
void delete(void* bp) {
    if (bp == NULL) return;
    size_t size = GET_SIZE(HDRP(bp));
    int class = search(size);
    void* pre = GET_PRE(bp);
    void* suc = GET_SUC(bp);
    printf("Deleting block at %p: pre=%p, suc=%p, class=%d\n", bp, pre, suc,
class);
    // 检查类别索引是否有效
    if (class < 0 || class >= NUM_CLASSES) {
        fprintf(stderr, "Error: Invalid class index %d\n", class);
        exit(EXIT_FAILURE);
    }
    // 更新前驱节点的后继
    if (pre != NULL) {
        if (!is_valid_pointer(pre)) {
            fprintf(stderr, "Error: Invalid predecessor pointer %p\n", pre);
            exit(EXIT_FAILURE);
        }
        printf("Updating predecessor's successor: pre=%p, suc=%p\n", pre,
suc);
        PUT(GET_SUC(pre), suc);
    } else {
        printf("Updating head of free list: heap_listp[%d]=%p\n", class,
suc);
        PUT(heap_listp + class * WSIZE, suc);
    }
    // 更新后继节点的前驱
    if (suc != NULL) {
        if (!is_valid_pointer(suc)) {
```

```
            fprintf(stderr, "Error: Invalid successor pointer %p\n", suc);
            exit(EXIT_FAILURE);
        }
        printf("Updating successor's predecessor: suc=%p, pre=%p\n", suc,
pre);
        PUT(GET_PRE(suc), pre);
    }
    // 清理 bp 的元数据
    PUT(GET_PRE(bp), NULL);
    PUT(GET_SUC(bp), NULL);
}
```

然后打印出来以后我的实际做法其实是把打印的内容输给了ai，然后让ai来帮我根据打印信息分析错误在哪哈哈

方法3：逐行人工检查

- 很痛苦的解决问题的方法，但鉴于我debug水平不够，这是我用的最多的方法。
- 主要注意：
  - 检查**宏定义**！
  - 检查**指针类型转换**是否统一！
  - 检查分配的**流程是否完整且正确**（比如有没有分隔出空闲块却没有 `insert` /空闲块被分配后却没有 `delete` 的情况）！

3实验最终结果截图

最终Version3的结果截图：

```
wcx@LAPTOP-OMTCB5PG:~/lab5/malloclab-stu/malloclab-handout$ ./mdriver -v
Team Name:IMSB
Member 1 :wcxx57:10242150443@stu.ecnu.edu.cn
Using default tracefiles in /home/wcx/lab5/malloclab-stu/traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   97%    5694  0.000805  7073
 1       yes   98%    5848  0.000809  7230
 2       yes   98%    6648  0.001075  6182
 3       yes   99%    5380  0.000624  8620
 4       yes   66%   14400  0.001120 12862
 5       yes   93%    4800  0.000757  6342
 6       yes   90%    4800  0.000708  6776
 7       yes   55%   12000  0.001155 10391
 8       yes   51%   24000  0.002056 11675
 9       yes   49%   14401  0.004273  3370
10       yes   45%   14401  0.001040 13851
Total          77%  112372  0.014421  7792

Perf index = 46 (util) + 40 (thru) = 86/100
```