

Malloc 实验报告

张澍民 10300720096

一、实验目的

通过 malloc 实验，提升对于动态存储器分配的理解和掌握，增强优化代码的能力，加深对于 malloc 等动态分配函数的理解。

二、实验内容

本实验要求编写一个 C 程序的动态存储分配器，即重新编写 malloc 函数，以实现 mm_init 初始化函数、mm_malloc 动态分配函数、mm_free 释放块函数、mm_realloc 重分配函数这四个函数的功能。实验可以声明整型、浮点型数据和指针，但是不允许定义额外的复合数据结构。

三、实验分析

1、内存分配器的组织策略

对于隐式空闲链表来讲，它是以物理地址的递增作为它本身的寻址方式，它的一个弊端在于 malloc、free、realloc 函数中每一步对块的操作都需要从堆的开头寻址到相应的位置再进行操作，比如要重定义一个模块，它的头地址可以计算出来，而在重定义的时候必须从堆的开头开始往后依次寻址来找到它，比如要 malloc 一个 16 字节的块并定义，那就必须从堆的开头往后依次寻找一个合适的 16 字节的空块，然后对其进行分配定义，而且这样的操作也会造成很多小的低能空闲块和碎片的产生，这将大大的降低动态分配的速度和空间利用率。

由于隐式空闲链表的块分配与堆块的总数呈线性关系，所以对于通用的分配器来说，隐式空闲链表是不合适的。

一种更好的方法是将空闲块组织为某种形式的显示数据结构，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。使用双向链表而不是隐式空闲链表，使得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，释放一个块的时间取决于所选择的空闲链表中块的排序策略。

此外，我们可以通过称为分离存储的方法来减少时间的分配。也就是维护多

个空闲链表，其中每个链表中的块有大致相等的大小。一般是将所有的块大小分成一些等价类，即大小类。分配器维护者一个空闲链表数组，每个大小类一个空闲链表，按照大小的升序排列，当分配器需要一个大小为 n 的块时，就搜索相应的空闲链表，如果找不到合适的块与之适配，就搜索下一个链表，以此类推。

分离存储的方法有很多，基本的方法主要有简单分离存储和分离适配。

使用简单分离适配，每个大小类的空闲链表包含大小相等的块，每个块的大小就是这个大小类中最大元素的大小。为了分配一个给定大小的块，会检查相应的空闲链表，如果链表非空，就简单地分配其中第一块的全部。空闲块不会分割以满足分配请求。如果链表为空，分配器就向操作系统请求一个固定大小的额外存储器片，将这个片分成大小相等的块，并将这些块链接起来形成新的空闲链表。要释放一个块，分配器只要简单地将这个块插入到相应的空闲链表的前部。这种方法的优点在于，分配和释放块都是很快的常数时间操作，每个片中都是大小相等的块，不分割不合并，这意味着每个块只有很少的存储器开销。显著的缺点在于，简单分离存储很容易造成内部和外部碎片。

使用分离适配的方法，分配器维护着一个空闲链表的数组。每个空闲链表是和一个小类相关联的，并且被组织成某种类型的显式或隐式链表。每个链表包含潜在的大小不同的块，这些块的大小是大小类的成员。分离适配方法十分快速，对于存储器的使用也很有效率。搜索时间减少了，因为搜索被限制在堆的某个部分，而不是整个堆。存储器利用率得到了改善，对于分离空闲链表的简单的首次适配搜索，其存储器利用率近似于对整个堆的最佳适配搜索的存储器利用率。

2、内存分配器的放置策略

常见的放置策略有首次适配，下次适配，最佳适配等。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往往将大的空闲块保留在链表的后面，缺点是它往往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。下一次适配的存储器利用率比首次适配低得多，最佳适配比首次适配和下一次适配

的存储器利用率要高。

3、内存分配器的合并策略

分配器可以选择立即合并，也可以选择推迟合并。立即合并也就是在每次一个块被释放时，就合并所有的相邻块；推迟合并就是等到某个稍晚的时候再合并空闲块。比如分配器可以推迟合并，直到某个分配请求失败，然后扫描整个堆，合并所有的空闲块。

立即合并可以在常数时间内执行完成，但是对于某些请求模式，这种方式会产生一种形式的抖动，块会反复地合并，然后马上分割。

四、初始 naïve 代码分析

初始版本是利用的隐式空闲链表方式所写。对于隐式空闲链表来讲，它是以物理地址的递增作为它本身的寻址方式，它的一个弊端在于 `malloc`、`free`、`realloc` 函数中每一步对块的操作都需要从堆的开头寻址到相应的位置再进行操作，就比如说：想要重定义一个模块，它的头地址可以计算出来，而在重定义的时候必须从堆的开头开始往后依次寻址来找到它，还有比如 `malloc` 一个 48 字节的块并定义，也必须从堆的开头往后依次寻找一个合适的 48 字节的空块，然后对其进行分配定义，而且这样的操作也会造成很多小的低能空闲块和碎片的产生，这大大的降低动态分配的速度和空间利用率。

五、实验设计

1、实验设计

本文设计的内存分配是一个简单分离空闲链表的变体，其中包含一些来自分离最佳适配的元素。我们使用 `NUM_FREE_LISTS` 的分离链表来存储空闲块。每个空闲链表的大小与 2 的幂有关。举例来说，第一个链表容纳了大小为 32B 的块，那么下一个链表就会容纳 40B 大小的块，再下一个为 72B，依次类推。最后的链表容纳了所有不能容纳在其他链表中的块。

当调用 `malloc` 函数时，就会从大小适当的链表中选出第一个合适的块。如果这个链表为空的话，就会使用下一个最大的链表。动态存储器的分配只会以 2

的幂来完成，除非需求足够大的空间来使用最后一个链表的块。空闲块的分离只会在最大的链表中操作，并且只有当结果还属于最大的链表中。

空闲链表以 LIFO 队列的形式进行操作。存储器的重分配操作是调用一个简单的 malloc 函数和 free 函数，除非等待重新分配的块在堆的末尾。在这种情况下，堆将会被扩展，并且额外的空间将会被添加给相应的块。没有存储器需要被复制。

存储器中的空闲块都有一个 4B 大小的头部和一个 4B 大小的脚部，其中头部和脚部是完全相同的。头部和脚部的信息包含了块的大小，块是否已被分配，以及前一个块是否已被分配。

已被分配的块包含了与空闲块相同的头部，但是已分配的块没有脚部。此外，额外的空间很少会使用到。

2、代码说明

（1）宏定义模块

宏定义模块的主要作用就是将一些常用的写起来又比较复杂的公式、代码、功能段等用简单明了、按照功能说明命名的宏，这样既能够减少程序写起来的复杂度，又能够很大程度上的避免一些细节错误，由于宏是按照功能来命名的，因此在用起来的时候也更加方便明朗，因此虽然是很简单的工作，但却有着至关重要的作用。

（2）变量和函数定义

对函数和变量的类型进行说明和陈列

（3）mm_init 函数

mm_init 函数是初始化函数，它将堆可用的虚拟存储器模型化为一个大的、双字对其的字节数组。它需要完成的工作是新建一个堆，并写入填充字和序言块，并通过 extend_heap 函数将堆扩展为 CHUNKSIZE 大小的字节。第一个字是一个双字边界对齐的不适用的填充字，填充字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，只由一个头部和一个脚部构成。

（4）mm_malloc 函数

mm_malloc 函数实现的分配一个用户请求的字节 size 在考虑头字节和对齐填

充字之后分配一个调整大小的空闲块并返回其头指针。该函数中需要考虑以下几点功能：

- a、排除无效请求；
- b、对 size 进行扩充（对于头指针字节的考虑和对齐填充字的考虑）；
- c、查询最佳适配；
- d、将相应字节的块放入指针处并返回指针。

（5）mm_free 函数

mm_free 函数完成的功能是将某个地址下的已分配块释放为未分配块。对它的操作很简单，只需要将该指针下的块的头尾标记为未分配块，并将其连接在显示空闲链表的合适位置即可。

（6）mm_realloc 函数

mm_realloc 函数实现的功能是再分配，即给特定指针下的已分配块赋予另外一个新的字节大小，并返回块的指针，也就是对已分配块进行修改和再分配。再分配的处理可以分为几种情况：

- a、当前块大小大于请求块大小；
- b、当前块大小小于请求块大小，但其后面块为未分配块，且二者大小之和大于请求块大小；
- c、当前块大小小于请求块大小，但其前面块为未分配块，且二者大小之和大于请求块大小；
- d、上面情况之外，需要扩充堆。

（7）coalesce 函数

Coalesce 函数实现的功能是将一个空闲块与其临近的空闲块进行合并组合成一个新的空闲块，以便于及时的对空闲链表进行更新。共有四种可能的情况：

- a、前后都是已分配块；
- b、前面为空闲块，后面为已分配块；
- c、前面为已分配块，后面为空闲块；
- d、前后都为空闲块。

（8）place 函数

Place 函数是将请求大小的内容块放入相应的指针入口处，在实际的物理空

间上完成对于块的分配作用。

(9) find_fit 函数

Find_fit 函数实现的功能很明确，就是在分配的时候寻找一个最佳适配的块作为待分配的目标，在这里我们要求它需要遵循以下三个原则：

- a、每次搜索后都必须返回一个合适大小的有效的空闲块指针；
- b、必须在满足需求的情况下尽可能选择小的空闲块进行适配；
- c、如果请求块太大而没有任何可以 STEM 点可以适配，那么就要延展堆的大小然后将块放在树干的最右端，并返回头指针。

由于延展堆的操作一般都在计算和判定最终请求大小的时候进行操作，因此在我们调用 find_fit 函数的时候就默认了第三项操作已经在函数外部完成，find_fit 函数在调用的时候肯定能从表里面选择到合适的块并返回指针，因此我们在 find_fit 函数中只需要进行前两个部分的功能进行编写。

(10) add_to_free_list 函数

该函数将 bp 指针指向的块添加到正确的空闲链表的头部。假设这个块已经被标记为空闲，并且头部和脚部都被正确设置，下一个物理块的前一位已经在其他地方被处理。

(11) remove_from_free_list 函数

该函数通过改变前一个和下一个块指针来将 bp 指针指向的块从它的空闲链表中去除。它不会将块标记为已分配，也不会改变下一个物理块的前一位。

(12) extend_heap 扩展堆函数

该函数以字将堆扩展来保持对齐要求。它将释放之前的结尾块，并且重新分配新的头部。

(13) split 函数

该函数将一个空闲块分成两个部分，并且返回第二个部分的指针。

(14) mark_free 函数

该函数将块标记为空闲，并且设置头部和脚部。更新下一个物理块的前一位，并且不会操作当前的空闲链表。

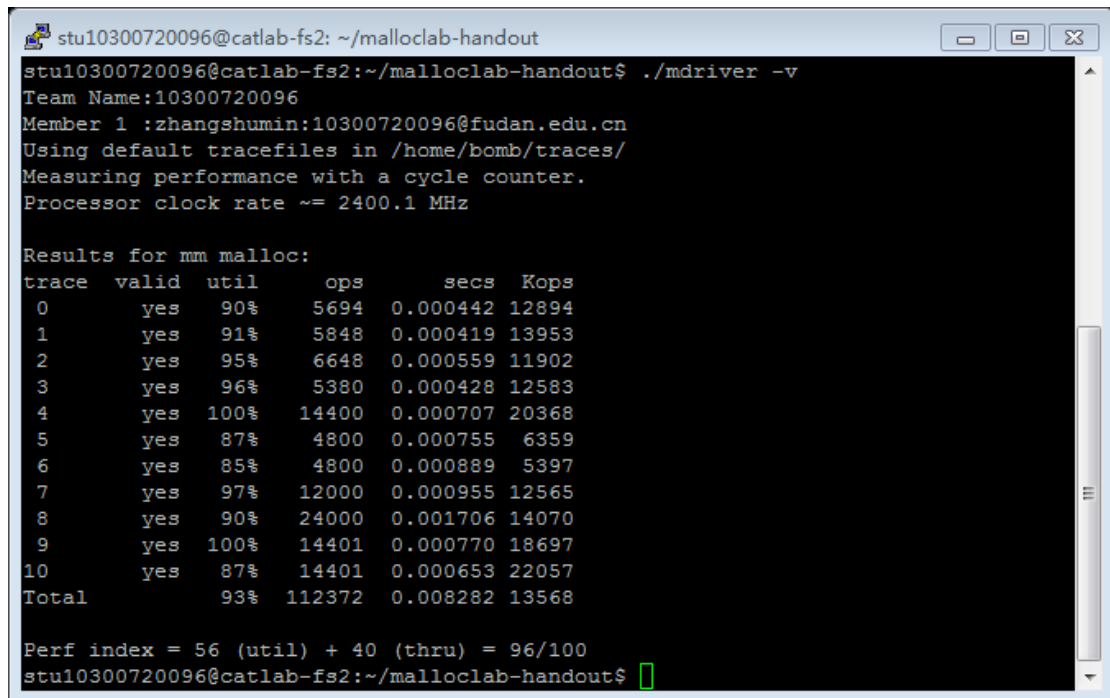
此外还有一些针对堆的检查函数和调试程序时的调试函数，见代码部分。

3、实验代码

代码部分见后附。

六、实验结果

经过多次的调试和修改程序代码，得到了如下的调试结果：



```
stu10300720096@catlab-fs2: ~/malloclab-handout
stu10300720096@catlab-fs2:~/malloclab-handout$ ./mdriver -v
Team Name:10300720096
Member 1 :zhangshumin:10300720096@fudan.edu.cn
Using default tracefiles in /home/bomb/traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 2400.1 MHz

Results for mm malloc:
trace  valid  util    ops      secs  Kops
0      yes   90%    5694  0.000442 12894
1      yes   91%    5848  0.000419 13953
2      yes   95%    6648  0.000559 11902
3      yes   96%    5380  0.000428 12583
4      yes  100%   14400  0.000707 20368
5      yes   87%    4800  0.000755  6359
6      yes   85%    4800  0.000889  5397
7      yes   97%   12000  0.000955 12565
8      yes   90%   24000  0.001706 14070
9      yes  100%   14401  0.000770 18697
10     yes   87%   14401  0.000653 22057
Total                93%  112372  0.008282 13568

Perf index = 56 (util) + 40 (thru) = 96/100
stu10300720096@catlab-fs2:~/malloclab-handout$
```

从图中可以看到，调试的最佳成绩为 96/100。

在执行 make 和 ./mdriver -v 指令后，看到出现了这样一条语句：

```
Using default tracefiles in /home/bomb/traces/
```

猜测可能是测试程序所要用到的测试 trace 文件。我们可以找到 trace 文件所在的位置去查看某个特定的 trace 文件中所包含的测试内容，下面就是其中一个 trace 文件中内容的部分截图：

```
2048100
6000
12000
1
a 0 64
a 1 448
f 3302
f 3304
```

我们可以看到，第一列的字母 a、r、f 分别代表的意义是 malloc、realloc、free；第二列的数字代表的意义是待操作的块标号，例如“f 1”的意义就是将标号

为 1 的已分配块释放；第三列数字代表的意义是分配或再分配时申请的块大小，单位为字节。

查看结果可以发现，最终只有第 5、6、10 三个 trace 文件的测试结果低于 90%，剩余的都在 90% 以上。尝试针对这三个 trace 文件进行调整程序，最后发现如果提高这三个文件的结果，则其他的可能会有所降低。综合各种测试结果，96/100 是最优的结果。

七、实验总结

1、通过这个 malloc 实验，更加深刻地了解到了 malloc 动态存储器分配的运行机制，理解到隐式空闲链表和显示空闲链表的优缺点以及各种选择策略和合并策略的方法，如简单分离存储和分离适配等等。

2、在实验过程中可以看到，不同的方法对于同一个 trace 测试文件运行的结果都不一定相同。某一种方法可能使得其中一个 trace 结果很高，但是另一些 trace 结果不是很高，而使用另一种方法之后，结果可能发生改变。也就是说，每一种方法都有其突出的特点，但是不同方法的结合有时候会比较困难，这是由于所用的思想或者体系可能不同。这就需要在各种方法之中做出一定的取舍，最终取综合来看最好的结果。

3、实验中为了更好地对各个函数进行调试，设置了一些调试的函数和检查堆的函数，通过这些函数，可以比较清楚地反映出动态分配过程中的一些状态，在遇到一些不容易发现的问题时，通过这些检查的函数返回的状态，可以比较容易发现问题所在，这也是设置这些函数的优势和方便所在。

4、类似于上一个优化实验一样，这一次实验的过程也是一步步不断地调试程序，修改代码，最后才能得出比较满意的结果，这就需要有足够的耐心，需要清晰的思维来分析代码，才能在多次的一步一步的不断调整修改中最终得到最优的结果。

八、实验完整代码


```

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>
#include <string.h>

#include "mm.h"
#include "memlib.h"

/*常量定义和宏定义*/
#define WSIZE      4           /*字大小（字节）*/
#define DSIZE      8           /*双字大小（字节）*/
#define CHUNKSIZE  16         /*初始堆大小（字节）*/
#define FREE_OVERHEAD  16      /*空闲块头部和脚部的开销（字节）*/
#define ALLOC_OVERHEAD  4      /*分配块头部和脚部的开销（字节）*/

#define MAX(x,y) ((x) > (y) ? (x) : (y))

/* 将块头部信息和分配位合并 */
#define PACK(size, alloc, prev) ((size) | (alloc) | (prev << 1))

/* 在地址 p 处读或写一个字 */
#define GET(p)      (*(size_t *) (p))
#define PUT(p, val) (*(size_t *) (p) = (val))

/* 在地址 p 处读出有效载荷的大小和分配位的信息 */
#define GET_SIZE(p)  (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)
#define GET_PREV(p)  ((GET(p) & 0x2) >> 1)

/* 根据给定的块指针，计算其头部地址和脚部地址*/
#define HDRP(bp)      ((char *) (bp) - WSIZE)
#define FTRP(bp)      ((char *) (bp) + GET_SIZE(HDRP(bp)) - DSIZE)

/* 根据给定的块指针，计算前一个和下一个的块地址*/
#define NEXT_BLKP(bp) ((char *) (bp) + GET_SIZE(((char *) (bp) - WSIZE)))
#define PREV_BLKP(bp) ((char *) (bp) - GET_SIZE(((char *) (bp) - DSIZE)))

/* 对齐 */
#define ALIGNMENT 8

```

```

/*近似到最近的对齐整数倍 */
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)

#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))

/*下一个和前一个指针地址*/
#define NEXTP(bp) HDRP(bp)+4
#define PREVP(bp) HDRP(bp)+8

/*获取和设置下一个和前一个链表的值*/
#define GetNext(bp) (void *) (GET(NEXTP(bp)))
#define GetPrev(bp) (void *) (GET(PREVP(bp)))
#define SetNext(bp,val) PUT(NEXTP(bp), (size_t)val)
#define SetPrev(bp,val) PUT(PREVP(bp), (size_t)val)

/* 分离空闲链表的数量*/
#define NUM_FREE_LISTS 7

/* 调试相关的各种常量 */
#define DEBUG 0
#define RUN_ON_INSN 10
#define SANITY 0
#define HEAP_CHECK_PERIOD 1000

/*全局变量定义*/
void* epilogue = NULL;
void* fls[NUM_FREE_LISTS];
size_t limits[NUM_FREE_LISTS];
int counter;
int itr=0;
int number_of_items[NUM_FREE_LISTS];

/* 函数声明*/
void F(char*c,int b);
void heapChekka(void* l);
int coalescingCheck(void* l);
void printCheck(char*c,int b);
int flValidPointersCheck(void *l);
int searchmem_check(void*bp);
int flCorrectnessCheck(void **l);
int searchlist_check(void*p,void **l);
int flAllFreeCheck(void* l);
int flPointerBoundsCheck(void *l);
int flSizeRangeCheck(void *l, int min, int max);

```

```

int* flCountsCheck(void ** l);

/*****
主函数
*****/

/*mm_init 函数，初始化堆，包括序言块、结尾块，同时初始化空闲链表的指针和空闲链表的范围*/
int mm_init(void)
{
    int list;
    size_t limit;
    void* heap_listp = NULL;

    if ((heap_listp = mem_sbrk(4*WSIZE)) == NULL)
        return -1;
    PUT(heap_listp, 0); //填充对其要求
    // 序言块头部，其不需要脚部（因为其为已分配的块），但是仍需要对齐
    PUT(heap_listp+WSIZE, PACK(ALIGNMENT, 1, 1)); // 序言块头部
    PUT(heap_listp+DSIZE+WSIZE, PACK(0, 1, 1)); // 结尾块头部
    epilogue = heap_listp+DSIZE+DSIZE;
    //初始化空闲链表指针和范围
    limit = 16;
    for (list = 0; list < NUM_FREE_LISTS; ++list)
    {
        fls[list] = NULL;
        limits[list] = ALIGNMENT * (limit + ALLOC_OVERHEAD + ALIGNMENT
- 1)/ALIGNMENT;
        limit = limit << 1;
    }
    //设置最后的极限为-1，以示没有上限
    limits[NUM_FREE_LISTS - 1] = -1;
    counter=0;
    itr++;
    return 0;
}

/* mm_free 释放函数*/
void mm_free(void *bp)
{
    mark_free(bp);
    //调用合并函数
    coalesce(bp);
}

```

```

}

/* mm_malloc 动态存储器分配函数*/
void *mm_malloc(size_t size)
{
    size_t asize; /* 调整块的大小 */
    size_t extendsize; /* 如果不合适则调整大小*/
    char * bp;
    if (size <= 0)
        return NULL;

    if (size <= limits[NUM_FREE_LISTS - 2])
    {
        size = round_up_to_next_power_of_two(size);
    }

    /* 调整块的大小以包括开销和对齐要求 */
    if (size <= DSIZE)
        asize = DSIZE + ALLOC_OVERHEAD;
    else
        asize = ALIGNMENT * ((size + (ALLOC_OVERHEAD) + (ALIGNMENT-1))/
ALIGNMENT);

    /* 查询空闲链表以找到合适的匹配 */
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }

    /* 没有匹配被找到，则申请更大的内存来放置块 */
    extendsize = MAX(asize, CHUNKSIZE);
    // 扩展堆函数添加新空间至空闲链表
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
    {
        return NULL;
    }
    place(bp, asize);
    return bp;
}

/* mm_realloc 重分配函数*/
void *mm_realloc(void *ptr, size_t size)
{

```

```

size_t difference, extendsize;
void * new_bp;

//检查块的大小
size_t block_size = GET_SIZE(HDRP(ptr));
// 确定所需要的大小
size_t new_size = (size + ALLOC_OVERHEAD + ALIGNMENT - 1) / ALIGNMENT
* ALIGNMENT;
//是否已分配的大小是否做够容纳需求的大小
if(new_size <= block_size)
    return ptr;//可使用同样的内存
void *newptr;

// 若增长的块时最后的块，则不需要复制，只需要扩展堆的大小来给这个块足够的空间
if (NEXT_BLKPTR(ptr) == epilogue)
{
    difference = new_size - block_size;
    // 扩展堆大小
    extendsize = MAX(difference, CHUNKSIZE);
    if ((new_bp = extend_heap(extendsize/WSIZE)) == NULL)
    {
        return NULL;
    }
    remove_from_free_list(new_bp);
    // 将块结合
    PUT(HDRP(ptr), PACK(block_size+extendsize, 1,
GET_PREV(HDRP(ptr))));
    // 更新结尾块
    PUT(HDRP(epilogue), PACK(0, 1, 1));
    newptr = ptr;
}
else
{
    //分配更多的内存
    newptr=mm_malloc(new_size);

    //若分配失败，则返回空 NULL
    if (newptr == NULL)
        return NULL;

    //将 stuff 从旧指针复制到新指针
    memcpy(newptr,ptr,new_size);
    mm_free(ptr);
}

```

```

    }
    return newptr;
}

// 将指针 bp 指向的块添加到正确的空闲链表中
void add_to_free_list(void *bp)
{
    int list;
    size_t size = GET_SIZE(HDRP(bp));
    //确定这个块属于哪个空闲链表
    for (list = 0; list < NUM_FREE_LISTS; ++list)
    {
        if (size <= limits[list])
            break;
    }
    if (list == NUM_FREE_LISTS)
        --list;

    if (fls[list]==NULL) //如果空闲链表为空
    {
        SetNext(bp, NULL); //把下一个设置为空
        SetPrev(bp, NULL); //把前一个设置为空
    }
    else //如果链表不为空
    {
        //将该块添加到指向链表头部的的位置
        SetNext(bp, fls[list]);
        SetPrev(bp, NULL);
        SetPrev(fls[list], bp);
    }
    fls[list]=bp; //将块添加到空闲链表头部
}

//将指针 bp 指向的块从空闲链表中去除
void remove_from_free_list(void* bp)
{
    //在空闲链表之前和之后立刻得到该块
    void * y = GetPrev(bp);
    void * z = GetNext(bp);

    if (y == NULL)
    {
        int i, list;
        list = -1;
    }

```

```

    for (i = 0; i < NUM_FREE_LISTS; ++i)
    {
        if (fls[i] == bp)
        {
            list = i;
        }
    }
    assert(list != -1);
    // 使下一个块作为链表的新的头部
    fls[list] = z;
}
else
{
    SetNext(y, z);
}
if (z != NULL)
{
    SetPrev(z, y);
}
}

/* coalesce 合并函数*/
void *coalesce(void *bp)
{
    unsigned int prev_alloc = GET_PREV(HDRP(bp));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    if (prev_alloc &&!next_alloc)
    {
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        void * nbp=NEXT_BLKP(bp);
        PUT(HDRP(bp), PACK(size, 0, 1));
        PUT(FTRP(bp), PACK(size, 0, 1));

        remove_from_free_list(nbp);
    }
    else if (next_alloc&&!prev_alloc)
    {
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0, 1));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0, 1));

        remove_from_free_list(PREV_BLKP(bp));
    }
}

```

```

        bp = PREV_BLK (bp);
    }
    else if (!next_alloc && !prev_alloc)
    {
        size += GET_SIZE (HDRP (PREV_BLK (bp))) +
            GET_SIZE (FTRP (NEXT_BLK (bp))) ;
        void * nbp = NEXT_BLK (bp);
        void * pbp = PREV_BLK (bp);
        PUT (HDRP (pbp), PACK (size, 0, 1));
        PUT (FTRP (nbp), PACK (size, 0, 1));

        remove_from_free_list (nbp);
        remove_from_free_list (pbp);
        bp = pbp;
    }

    add_to_free_list (bp);
    return bp;
}

unsigned int round_up_to_next_power_of_two (unsigned int size)
{
    size--;
    size |= size >> 1;
    size |= size >> 2;
    size |= size >> 4;
    size |= size >> 8;
    size |= size >> 16;
    size++;
    return size;
}

/*extend_heap 扩展堆函数*/
void *extend_heap (size_t words)
{
    char *bp;
    size_t size;

    /* 分配一个偶数字长度来保持对齐 */
    size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
    if ( (bp = mem_sbrk (size)) == NULL )
        return NULL;

    /* 初始化空闲块的头部脚部和结尾块头部*/

```



```

    unsigned int prev = GET_PREV(HDRP(epilogue));
    PUT(HDRP(bp), PACK(size, 0, prev));           // 释放块头部
    PUT(FTRP(bp), PACK(size, 0, prev));           // 释放块脚部
    PUT(HDRP(NEXT_BLK_P(bp)), PACK(0, 1, 0));     // 新的结尾块头部

    // 更新结尾块指针
    epilogue = NEXT_BLK_P(bp);

    /*若前一个块是空闲的则合并 */
    return coalesce(bp);
}

/* find_fit函数*/
void * find_fit(size_t asize)
{
    void *bp;
    int list;

    // 确定检查哪一个空闲链表
    for (list = 0; list < NUM_FREE_LISTS; ++list)
    {
        if (asize <= limits[list])
            break;
    }
    if (list == NUM_FREE_LISTS)
        --list;

    //在所有空闲链表中查找上一个确定的开始处
    while (list < NUM_FREE_LISTS)
    {
        for (bp = fls[list]; bp!=NULL; bp = GetNext(bp))//遍历空闲链表
        {
            //如果有块适合
            if ((asize <= GET_SIZE(HDRP(bp))))
            {
                assert(!GET_ALLOC(HDRP(bp))); //若已分配则不应该出现在空
闲链表中
                return bp;
            }
        }
        ++list;
    }
}

```

```

    return NULL;
}

// Splits 分离函数
void *split(void *bp, size_t size)
{
    void * new_block;
    size_t old_size;

    old_size = GET_SIZE(HDRP(bp));
    // 需要保持对齐要求
    assert(((old_size - size) % ALIGNMENT) == 0);

    // 调整旧的块大小
    PUT(HDRP(bp), PACK(size, 0, GET_PREV(HDRP(bp))));
    PUT(FTRP(bp), PACK(size, 0, GET_PREV(HDRP(bp))));
    // 创建一个新的块
    new_block = NEXT_BLK(P(bp));
    PUT(HDRP(new_block), PACK(old_size - size, 0, 0));
    PUT(FTRP(new_block), PACK(old_size - size, 0, 0));

    return new_block;
}

/* place 放置函数 */
void place(void* bp, size_t asize)
{
    /* Get the current block size */
    size_t bsize = GET_SIZE(HDRP(bp));

    remove_from_free_list(bp);

    if (bsize > limits[NUM_FREE_LISTS - 2] && (bsize - asize) >
limits[NUM_FREE_LISTS - 2])
    {
        // 将块的剩余放回到一个空闲链表中
        add_to_free_list(split(bp, asize));
        bsize = asize;
    }

    // 更新下一个物理块的前一位
    void * next_physical = NEXT_BLK(P(bp));
    unsigned int alloc = GET_ALLOC(HDRP(next_physical));
    size_t next_size = GET_SIZE(HDRP(next_physical));

```

```

    PUT(HDRP(next_physical), PACK(next_size, alloc, 1));
    if (alloc == 0 && next_size>0)
        PUT(FTRP(next_physical), PACK(next_size, alloc, 1));

    // 自动更新块
    // 注：我们不操作脚部是因为已分配的块没有脚部
    PUT(HDRP(bp), PACK(bsize, 1, GET_PREV(HDRP(bp))));
}

```

// Mark 标记函数

```

void mark_free(void *bp)
{
    // 更新块
    size_t size = GET_SIZE(HDRP(bp));
    unsigned int prev = GET_PREV(HDRP(bp));
    PUT(HDRP(bp), PACK(size, 0, prev));
    PUT(FTRP(bp), PACK(size, 0, prev));

    // 更新下一个块
    void * next = NEXT_BLKP(bp);
    size_t next_size = GET_SIZE(HDRP(next));
    unsigned int next_alloc = GET_ALLOC(HDRP(next));
    PUT(HDRP(next), PACK(next_size, next_alloc, 0));
    //如果该块有脚部则更新
    if (next_alloc == 0 && next_size > 0)
        PUT(FTRP(next),
PACK(GET_SIZE(HDRP(next)), GET_ALLOC(HDRP(next)), 0));
}

```

/*以下为堆检查函数*/

//空闲链表的指针是否指向有效的堆地址?

```

int flPointerBoundsCheck(void *l)
{
    void * bp;
    void *start = mem_heap_lo() ;
    void *end = mem_heap_hi()-3;
    for(bp=l;bp!=NULL;bp=GetNext(bp))
    {
        // 超过边界
        if((bp>=start && bp <=end)==0)
            return 0;
    }
}

```

```

    }
    return 1;
}
//空闲链表中的每个块是否被标为空闲?
int flAllFreeCheck(void* l)
{
    void * bp;
    for(bp=l;bp!=NULL;bp=GetNext(bp))
    {
        //如果块已被分配
        if(GET_ALLOC(HDRP(bp))==1)
            return 0;
    }
    return 1;
}

//块是否被正确合并
//假设空闲链表完全正确
int coalescingCheck(void* l)
{
    void * bp;
    //遍历空闲链表
    for(bp=l;bp!=NULL;bp=GetNext(bp))
    {
        if(GET_ALLOC(HDRP(NEXT_BLKP(bp)))== 0 || GET_PREV(HDRP(bp))
== 0)
        {
            return 0;
        }
    }
    return 1;
}

//查找该指针是否在空闲链表中
int searchlist_check(void*p,void **l)
{
    int i;
    void * bp;
    for(i=0;i<NUM_FREE_LISTS;i++)/*NUM_FREE_LISTS==8*/
    {
        for(bp=l[i];bp!=NULL;bp=GetNext(bp))
        {
            if(bp==p)
                return 1;
        }
    }
}

```

```

    }
}

//遍历链表
return 0;
}

//每个空闲块是否在空闲链表中
int flCorrectnessCheck(void **flp)
{
    return 1;
    void *bp;
    void *end = mem_heap_hi() - WSIZE + 1; //points to last word
    //遍历内存
    for(bp = mem_heap_lo() + DSIZE; bp < end; bp += GET_SIZE(HDRP(bp)))
    {
        if(GET_ALLOC(HDRP(bp)) == 0 && searchlist_check(bp, flp) == 0)
            return 0;
    }
    return 1;
}

//查找指针是否在内存中
int searchmem_check(void *bp)
{
    void *p;
    void *end = mem_heap_hi() - 3;
    //从内存的开始到结束
    for(p = mem_heap_lo() + DSIZE; p < end; p += GET_SIZE(HDRP(p)))
    {
        //若找到这个块，则是有效的
        if(p == bp)
            return 1;
    }
    return 0;
}

//检查是否所有空闲链表的指针都在内存中
int flValidPointersCheck(void *l)
{
    void *bp;
    //遍历空闲链表中的每个元素
    for(bp = l; bp != NULL; bp = GetNext(bp))
    {

```

```

        //若不能找到 bp
        //则空闲链表指针是错误的, 返回 fail
        if (searchmem_check(bp) == 0)
            return 0;
    }
    return 1;
}

//检查所有在空闲链表中的大小是否在最小值和最大值之间
//max=-1 代表没有最大值
int flSizeRangeCheck(void *l, int min, int max)
{
    void*bp;
    int size;
    //遍历链表
    for (bp=l; bp!=NULL; bp=GetNext(bp))
    {
        size=GET_SIZE(HDRP(bp));
        if (size<min || ( (max!=-1) && (size > max) ) )
            return 0;
    }
    return 1;
}

//计算空闲链表中多少元素可以在一个数组中
int* flCountsCheck(void ** l)
{
    int i;
    void * bp;
    for (i=0; i<NUM_FREE_LISTS; i++)
    {
        number_of_items[i]=0;
        //遍历链表
        for (bp=l[i]; bp!=NULL; bp=GetNext(bp))
        {
            //计数
            number_of_items[i]++;
        }
    }
    return number_of_items;
}

//显示一个检查
void printCheck(char*c, int b)

```

```

{
    printf("\n%i CHEKKA: %s?",b,c); printfFlush("");
}
//显示检查结果
void heapChekka(void* l)
{
    printCheck("1. Do pointers in heap block point to valid heap
addresses", flPointerBoundsCheck(l));
    printCheck("2. Is every block in the free list marked as free",
flAllFreeCheck(l));
    printCheck("3. Do pointers in free list point to valid free blocks",
flValidPointersCheck(l));
    printCheck("4. Are blocks are coalesced
properly",coalescingCheck(l));
    printCheck("5. Is every free block actually in the free
list" ,flCorrectnessCheck(l));
    printf("\n\n\n");printfFlush("");
}

/*以下部分为调试函数，来帮助更好地调试程序*/
void printfFlush(char *c)
{
    printf("%s",c);
    fflush(stdout);
}

void run_check(void* l, size_t max, size_t min)
{
    assert(flSizeRangeCheck(l,min,max));
    assert(flPointerBoundsCheck(l));
    assert(flAllFreeCheck(l));
    assert(flValidPointersCheck(l));
    assert(coalescingCheck(l));
}

void mm_check()
{
    int i;
    for (i = 0; i < NUM_FREE_LISTS; ++i)
    {
        run_check(fls[i], limits[i], i>0?limits[i-1]:0);
    }
    assert(flCorrectnessCheck(fls));
}

```

```

}

void heapCheckCounter(char* c)
{
    counter++;
    printf("(%s%i)", c, counter); printfFlush("");
    if(counter==RUN_ON_INSN)
    {
        if(itr%12==1)
            mm_check();
    }

    if(counter%HEAP_CHECK_PERIOD==0)
    {
        if(itr%12==1)
            mm_check();
    }
}

```