

华东师范大学软件学院实验报告

实验课程：计算机系统	年级：2023 级本科	实验成绩：
实验名称：Lab5 – Malloc Lab	姓名：张梓卫	
实验编号：(5)	学号：10235101526	实验日期：2024/06/03
指导老师：肖波	组号：	

目录

一 实验简介	1
1 实验目的	1
2 实验前置准备	1
3 实验要求	1
二 Part A 实验内容	2
1 堆栈分配	2
2 代码分析及填充	3
3 实验结果	7
三 实验总结	8
四 附录	8

一 实验简介

1 实验目的

本实验是 CSAPP 的 Malloc Lab，目的是通过实现一个动态内存分配器，加深对内存分配和管理理解。

2 实验前置准备

3 实验要求

- 不允许在 `mm.c` 程序中定义任何全局或静态复合数据结构，例如数组、结构、树或列表等规则。
- 在本实验中，我们将为 C 程序编写一个动态存储分配器（`malloc`、`free` 和 `realloc` 例程，实现一个正确，高效和快速的分配器。
- 本实验性能指标有两个方面，内存利用率和吞吐量，我们定义的分配器需要平衡这两个指标，以获取更高的分数。
- 我们需要完善以下函数：

```
1 int mm_init(void);
2 void *mm_malloc(size_t size);
3 void mm_free(void *ptr);
4 void *mm_realloc(void *ptr, size_t size);
5
```

Function List

- 实验工具：*mdriver* 工具，可以检测正确性和性能。

二 Part A 实验内容

1 堆栈分配

堆的地址是连续的，一整块内存区域都是堆，那么如何管理这一整块内存区域呢。有三种常见的方式

- 隐式空闲列表 (Implicit Free List):

把整块堆切分为许多紧邻的块，每个块的头部包含了这个块的大小以及它是否被分配的信息，通过这个块的大小我们就能找到与它相连的下一块的起始地址。

- 好处：简单
- 坏处：每次申请时都需要从头开始遍历，吞吐量低下，且容易产生许多空间碎片

- 显式空闲列表 (Explicit Free Lists):

对隐式空闲列表的改进，在隐式空闲列表基础上，每个空闲块除了存储它的大小和是否被分配的信息外，还存储了指向下一个和/或上一个空闲块的指针，这样查找空闲块的时候只需要遍历空闲列表

- 好处：提高了内存分配效率，降低空间碎片
- 坏处：需要多的空间存储指针，维护困难

- 分离空闲列表 (Segregated Free Lists):

维护多个空闲链表，每个链表中的块有大致相等的大小，分配器维护着一个空闲链表数组

- 好处：更加提高了内存分配效率，降低空间碎片
- 坏处：维护困难，可能导致空间利用率不高

我们的目的：平衡好吞吐率和空间利用率（这两个其实是冲突的，不可能吞吐率高又空间利用率高，因为高的吞吐率必然采用诸如链表，哈希表，树等结构，这些结构必然导致空间利用率降低，所以得平衡）

原文链接：[堆栈分配](#)

堆栈相关的知识如下所示，我们要做的就是用以上的方法实现一个动态内存分配器，加深对内存分配和管理的理解。

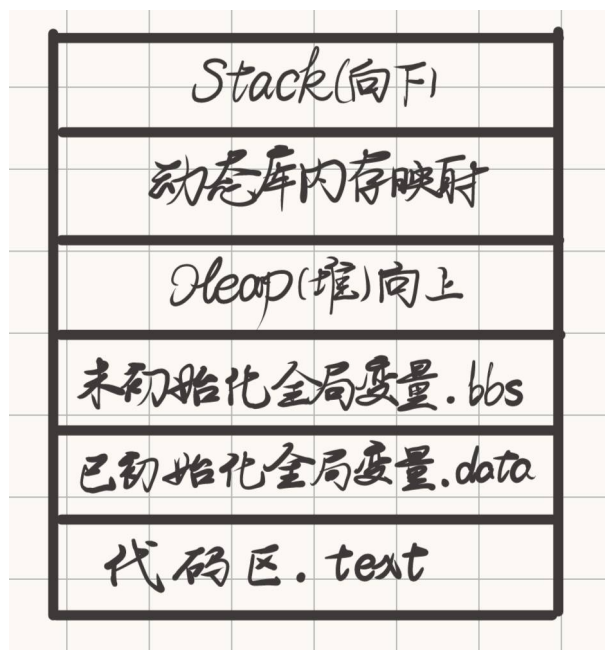


图 1: 堆栈分配

查找空闲块的三个方法：

- 1. 首次适配：选择第一个合适的块。
- 2. 下次适配：每次搜索从上次结束的地方开始。
- 3. 最佳适配：选择大小合适的最小块。

2 代码分析及填充

对于速度 (*thru*) 而言，我们需要关注 *malloc*、*free*、*realloc* 每次操作的复杂度。

对于内存利用率 (*util*) 而言，我们需要关注 *Internal Fragmentation* (块内损失) 和 *External Fragmentation* (块是分散不连续的，无法整体利用)。

即我们 *Free* 和 *Malloc* 的时候要注意整体大块利用 (例如合并 *free* 块、*realloc* 的时候判断下一个块是否空闲)。

按照老师的要求，我首先选择实现双字对齐，隐式链表，采用首次适配。

按照 *mm.c* 中的内容补充一些宏定义以及函数定义，如下所示：

```

1  /* Basic constants and macros */
2
3  #define WSIZE 4 // Word and header/footer size (bytes)
4  #define DSIZE 8 // Double word size (bytes)
5  #define CHUNKSIZE (1<12) // Extend heap by this amount (bytes)
6
7  #define MAX(x,y) ((x)>(y)?(x):(y)) // Maximum of two values
8
9  #define PACK(size,alloc) ((size)|(alloc)) // Pack a size and allocated bit into a word
10
11 #define GET(p) (*(unsigned int *) (p)) // Read a word at address p
12 #define PUT(p,val) (*(unsigned int *) (p) = (val)) // Write a word at address p
13 #define GET_SIZE(p) (GET(p) & ~0x7) // Read the size and allocated fields from address p
14 #define GET_ALLOC(p) (GET(p) & 0x1) // Read the allocated field from address p
15
16 // Given block ptr bp, compute address of its header and footer
17 #define HDRP(bp) ((char *) (bp) - WSIZE)
18 #define FTRP(bp) ((char *) (bp) + GET_SIZE(HDRP(bp)) - DSIZE)
19
20 // Given block ptr bp, compute address of next and previous blocks
21 #define NEXT_BLKP(bp) ((char *) (bp) + GET_SIZE(((char *) (bp) - WSIZE)))
22 #define PREV_BLKP(bp) ((char *) (bp) - GET_SIZE(((char *) (bp) - DSIZE)))

```

Define List

接下来是需要定义的函数和辅助函数：

```

1  /* 内部编程的函数原型 */
2  extern void *extend_heap(size_t words); // 扩展堆
3  extern void *coalesce(void *bp); // 合并空闲块
4  extern void *find_fit(size_t size); // 查找合适的空闲块
5  extern void place(void *bp, size_t asize); // 分配空闲块
6  extern int mm_init (void); // 初始化
7  extern void *mm_malloc (size_t size); // 分配
8  extern void mm_free (void *ptr); // 释放
9  extern void *mm_realloc(void *ptr, size_t size); // 重新分配
10 static char *heap_listp = 0; // 指向块首的指针

```

Function List

接下来，我们对函数进行填充：

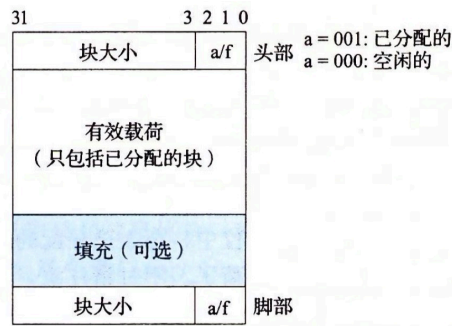


图 2: 块大小解释

```

1  int mm_init(void)
2  {
3      /* Create the initial empty heap */
4      if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1) // 申请4个字节
5          return -1;
6      PUT(heap_listp, 0); // Alignment padding
7      PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); // Prologue header
8      PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); // Prologue footer
9      PUT(heap_listp + (3*WSIZE), PACK(0, 1)); // Epilogue header
10     heap_listp += (2*WSIZE); // 指向序言块的头部
11
12     /* Extend the empty heap with a free block of CHUNKSIZE bytes */
13     if (extend_heap(CHUNKSIZE/WSIZE) == NULL) // 扩展堆
14         return -1; // 扩展失败
15     return 0; // 成功
16 }

```

mm_init.c

- 以堆底为起始位置，新建四个 *WSIZE 4 Bytes Or 32 Bits* 大小的块
- 第一块什么都不放作为填充字
- 第二、三块分别作为序言块的头和脚
- 第四块作为结尾块
- 然后将指针后移两块，指向序言块脚部起始位置，随后调用 *extend_heap()*，申请 *CHUNKSIZE* 大小的空间，以备使用。

```

1  static void *extend_heap(size_t words){
2      char *bp; // 块指针
3      size_t size; // 扩展大小
4      size = (words%2) ? (words+1)*WSIZE : words*WSIZE; // 8字节对齐
5      if((long)(bp=mem_sbrk(size))==(void *)-1)
6          return NULL; // 申请失败
7      PUT(HDRP(bp),PACK(size,0)); // 更新头部
8      PUT(FTRP(bp),PACK(size,0)); // 更新尾部
9      PUT(HDRP(NEXT_BLKP(bp)),PACK(0,1)); // 新结尾块
10     return coalesce(bp); // 合并空闲块
11 }

```

Extend_heap.c

这个函数将堆扩容指定 *byte* 大小。如果指定 *words* 大小不为 8 的倍数则向上取整使得每次扩容都是八字节对齐，最后将头、脚内容补齐并将下一个块置为结尾块，最后调用 *coalesce()* 函数堆 *bp* 进行合并操作后返回。

接下来，是 *mm_malloc()* 函数：

此函数申请大小为 *size* 的空间。*asize* 为对 *size* 进行 8 字节对齐检查后的大小，*extendsize* 为取 *CHUNKSIZE* 和 *asize* 中较大的一个。

先用 *find_fit()* 在现有的块中进行搜索，如果搜索到了，即用 *place()* 函数将 *asize* 大小的空间放到 *bp* 中。由于有可能全部分也可能分割成一个占用块和一个空闲块，所以不能粗暴的完全占用，要写一个 *place()* 函数来分配。如果找不到合适块，就向堆申请新空间，新空间的大小为 *extendsize*，然后再用 *place()* 函数放入。

```

1 void *mm_malloc(size_t size)
2 {
3     size_t asize; // 对齐大小
4     size_t extendsize; // 扩展大小
5     char *bp; // 块指针
6     if (size == 0) return NULL; // 无效大小
7     if (size <= DSIZE){
8         asize = 2*(DSIZE); // 8字节对齐
9     }else{
10        asize = (DSIZE)*((size+(DSIZE)+(DSIZE-1)) / (DSIZE)); // 8字节对齐
11    }
12    if ((bp = find_fit(asize)) != NULL){ // 查找合适的块
13        place(bp, asize); // 分配
14        return bp; // 返回块指针
15    }
16    extendsize = MAX(asize, CHUNKSIZE); // 申请大小
17    if ((bp = extend_heap(extendsize/WSIZE)) == NULL){ // 扩展堆
18        return NULL; // 申请失败
19    }
20    place(bp, asize); // 分配
21    return bp; // 返回块指针
22 }

```

mm_malloc.c

接下来是 *mm_free()* 函数：

此函数释放指针 *ptr* 所指向的块，将其标记为未分配状态，然后调用 *coalesce()* 函数合并空闲块。

```

1 void mm_free(void *bp)
2 {
3     if (bp == 0)
4         return; // 无效指针
5     size_t size = GET_SIZE(HDRP(bp)); // 获取块大小
6     PUT(HDRP(bp), PACK(size, 0)); // 标记为未分配
7     PUT(FTRP(bp), PACK(size, 0)); // 标记为未分配
8     coalesce(bp); // 合并空闲块
9 }

```

mm_free.c

接下来是 *mm_realloc()* 函数：

此函数重新分配指针 *ptr* 所指向的块，将其大小调整为 *size*，如果 *size* 为 0，则释放指针 *ptr* 所指向的块。

```

1 void *mm_realloc(void *ptr, size_t size) {
2     size_t oldsize; // 旧块大小
3     void *newptr; // 新块指针
4     /* If size == 0 then this is just free, and we return NULL. */
5     if (size == 0) { // 释放
6         mm_free(ptr); // 释放
7         return 0; // 返回NULL
8     }
9     /* If oldptr is NULL, then this is just malloc. */
10    if (ptr == NULL) {
11        return mm_malloc(size); // 申请
12    }
13    newptr = mm_malloc(size); // 申请
14    /* If realloc() fails the original block is left untouched */
15    if (!newptr) {

```

```

16     return 0;
17 }
18 /* Copy the old data. */
19 oldsize = GET_SIZE(HDRP(ptr)); // 旧块大小
20 if(size < oldsize) oldsize = size; // 复制旧数据
21 memcpy(newptr, ptr, oldsize); // 复制
22 /* Free the old block. */
23 mm_free(ptr); // 释放
24 return newptr; // 返回新块
25 }

```

mm_realloc.c

接下来是 *place()* 函数：

此函数将 *asize* 大小的块放入 *bp* 中，如果剩余空间大于 *DSIZE*，则将剩余空间分割出来，否则将整个块分配出去。

```

1 static void place(void *bp, size_t asize){
2     size_t csize = GET_SIZE(HDRP(bp)); // 当前块大小
3     if((csize-asize)>=(2*DSIZE)){ // 剩余空间大于DSIZE
4         PUT(HDRP(bp),PACK(asize,1)); // 更新头部
5         PUT(FTRP(bp),PACK(asize,1)); // 更新尾部
6         bp = NEXT_BLK(bp); // 下一个块
7         PUT(HDRP(bp),PACK(csize-asize,0)); // 更新头部
8         PUT(FTRP(bp),PACK(csize-asize,0)); // 更新尾部
9     }else{
10        PUT(HDRP(bp),PACK(csize,1)); // 更新头部
11        PUT(FTRP(bp),PACK(csize,1)); // 更新尾部
12    }
13 }

```

place.c

coalesce() 函数：

此函数合并空闲块，如果前后块都是空闲块，则将其合并为一个块。

```

1 static void *coalesce(void *bp){
2     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLK(bp))); // 前块是否空闲
3     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLK(bp))); // 后块是否空闲
4     size_t size = GET_SIZE(HDRP(bp)); // 当前块大小
5     if(prev_alloc && next_alloc) { // 前后块都忙碌
6         return bp; // 返回当前块
7     }else if(prev_alloc && !next_alloc){ // 前忙碌，后空闲
8         size += GET_SIZE(HDRP(NEXT_BLK(bp))); // 合并
9         PUT(HDRP(bp),PACK(size,0)); // 更新头部
10        PUT(FTRP(bp),PACK(size,0)); // 更新尾部
11    }else if(!prev_alloc && next_alloc){ // 前空闲，后忙碌
12        size += GET_SIZE(HDRP(PREV_BLK(bp))); // 合并
13        PUT(FTRP(bp),PACK(size,0)); // 更新尾部
14        PUT(HDRP(PREV_BLK(bp)),PACK(size,0)); // 更新头部
15        bp = PREV_BLK(bp); // 返回前块
16    }else {
17        size +=GET_SIZE(FTRP(NEXT_BLK(bp)))+ GET_SIZE(HDRP(PREV_BLK(bp))); // 前后都空闲
18        PUT(FTRP(NEXT_BLK(bp)),PACK(size,0)); // 更新尾部
19        PUT(HDRP(PREV_BLK(bp)),PACK(size,0)); // 更新头部
20        bp = PREV_BLK(bp); // 返回前块
21    }
22    return bp;
23 }

```

calesec.c

此函数将对传入的块指针进行前后检查，如果前面块或者后面块同样为空闲块，就进行合并。首先获取前后块的空闲状态，然后进行条件判断，会出现四种情况：

- 前后都忙碌

- 前忙碌，后空闲
- 前空闲，后忙碌
- 前后都空闲

find_fit() 函数:

此函数在空闲块中查找合适的块，如果找到了就返回块指针，否则返回 *NULL*。

```

1 static void *find_fit(size_t asize){
2     void *bp; // 遍历指针
3     for(bp = heap_listp; GET_SIZE(HDRP(bp))>0; bp = NEXT_BLKP(bp)){ // 从堆底开始遍历
4         if(!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))){ // 找到合适的块
5             return bp; // 返回块指针
6         }
7     }
8     return NULL;
9 }

```

find_fit.c

这个函数从堆底开始遍历，如果找到了一个空闲块且大小大于等于 *asize*，就返回这个块的指针，否则返回 *NULL*。

3 实验结果

将补充好的 *mm.c* 文件导入 Linux 系统，使用 *make* 命令编译，然后使用 *mdriver* 工具进行测试，得到如下结果：

- make clean
- make
- ./mdriver -av -t traces/

```

deralve@10235101526: ~/Lab5/malloclab-handout
|
gcc -Wall -O2 -m64 -c -o memlib.o memlib.c
gcc -Wall -O2 -m64 -c -o fsecs.o fsecs.c
gcc -Wall -O2 -m64 -c -o fcyc.o fcyc.c
gcc -Wall -O2 -m64 -c -o clock.o clock.c
gcc -Wall -O2 -m64 -c -o ftimer.o ftimer.c
gcc -Wall -O2 -m64 -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ftimer.o
deralve@10235101526:~/Lab5/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 0.006022 945
1 yes 99% 5848 0.004261 1373
2 yes 99% 6648 0.007798 853
3 yes 100% 5380 0.005645 953
4 yes 66% 14400 0.00063230400
5 yes 92% 4800 0.007927 606
6 yes 92% 4800 0.007427 646
7 yes 55% 12000 0.111410 108
8 yes 51% 24000 0.381008 63
9 yes 27% 14401 0.046951 307
10 yes 34% 14401 0.001909 7544
Total 74% 112372 0.580422 194

Perf index = 44 (util) + 13 (thru) = 57/100
deralve@10235101526:~/Lab5/malloclab-handout$

```

图 3: Result

可以看到，我们获得了 44 的 *Util* 分，13 的 *Thru* 分，总分为 57。

这个分数是一个看起来尚未及格的分，我后续将考虑使用其他方法来优化。

三 实验总结

在查阅资料的过程中，我看到来自北京大学的学生得到了满分：

原文链接：[更适合北大宝宝体质的 Malloc Lab](#)

这个实验可谓是最困难的实验之一了，显然，它需要我对整个课本有体系化的了解，但上面的链接里写的很大部分的内容是我暂时还没办法看懂的，光是查看前人的代码，通过理解并加以注释的过程，都已经消耗了我很多精力，就不把他们的代码复制过来跑分了，等到以后学有余力的情况下，再回来重新分析，毕竟 CSAPP 是公认的神书。

四 附录

实验的源代码如下所示：

```

1  /* single word (4) or double word (8) alignment */
2  #define ALIGNMENT 8
3
4  #define WSIZE 4          // Word and header/footer size (bytes)
5  #define DSIZ 8          // Double word size (bytes)
6  #define CHUNKSIZE (1 << 12) // Extend heap by this amount (bytes)
7
8  #define MAX(x, y) ((x) > (y) ? (x) : (y)) // Maximum of two values
9
10 #define PACK(size, alloc) ((size) | (alloc)) // Pack a size and allocated bit into a word
11
12 #define GET(p) (*(unsigned int *) (p)) // Read a word at address p
13 #define PUT(p, val) (*(unsigned int *) (p) = (val)) // Write a word at address p
14 #define GET_SIZE(p) (GET(p) & ~0x7) // Read the size and allocated fields from address p
15 #define GET_ALLOC(p) (GET(p) & 0x1) // Read the allocated field from address p
16
17 // Given block ptr bp, compute address of its header and footer
18 #define HDRP(bp) ((char *) (bp) - WSIZE)
19 #define FTRP(bp) ((char *) (bp) + GET_SIZE(HDRP(bp)) - DSIZ)
20
21 // Given block ptr bp, compute address of next and previous blocks
22 #define NEXT_BLK(bp) ((char *) (bp) + GET_SIZE(((char *) (bp) - WSIZE)))
23 #define PREV_BLK(bp) ((char *) (bp) - GET_SIZE(((char *) (bp) - DSIZ)))
24
25 /* rounds up to the nearest multiple of ALIGNMENT */
26 #define ALIGN(size) (((size) + (ALIGNMENT - 1)) & ~0x7)
27
28 #define SIZE_T_SIZE (ALIGN(sizeof(size_t)))
29
30 /* 内部编程的函数原型 */
31 static void *extend_heap(size_t words); // 扩展堆
32 static void *coalesce(void *bp); // 合并空闲块
33 static void *find_fit(size_t size); // 查找合适的空闲块
34 static void place(void *bp, size_t asize); // 分配空闲块
35
36 static char *heap_listp = 0; // 指向块首的指针
37
38 /*
39  * mm_init - initialize the malloc package.
40  */
41 int mm_init(void) {
42     /* Create the initial empty heap */
43     if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *) -1) // 申请4个字节
44         return -1;
45     PUT(heap_listp, 0); // Alignment padding
46     PUT(heap_listp + (1 * WSIZE), PACK(DSIZ, 1)); // Prologue header
47     PUT(heap_listp + (2 * WSIZE), PACK(DSIZ, 1)); // Prologue footer
48     PUT(heap_listp + (3 * WSIZE), PACK(0, 1)); // Epilogue header
49     heap_listp += (2 * WSIZE); // 指向序言块的头部

```



```

50
51 /* Extend the empty heap with a free block of CHUNKSIZE bytes */
52 if (extend_heap(CHUNKSIZE / WSIZE) == NULL) // 扩展堆
53     return -1;                               // 扩展失败
54 return 0;                                   // 成功
55 }
56
57 /*
58  * mm_malloc - Allocate a block by incrementing the brk pointer.
59  *   Always allocate a block whose size is a multiple of the alignment.
60  */
61 void *mm_malloc(size_t size) {
62     size_t asize;
63     size_t extendsize;
64     char *bp;
65     if (size == 0)
66         return NULL;
67     if (size <= DSIZE) {
68         asize = 2 * (DSIZE);
69     } else {
70         asize = (DSIZE) * ((size + (DSIZE) + (DSIZE - 1)) / (DSIZE));
71     }
72     if ((bp = find_fit(asize)) != NULL) {
73         place(bp, asize);
74         return bp;
75     }
76     extendsize = MAX(asize, CHUNKSIZE);
77     if ((bp = extend_heap(extendsize / WSIZE)) == NULL) {
78         return NULL;
79     }
80     place(bp, asize);
81     return bp;
82 }
83
84 static void place(void *bp, size_t asize) {
85     size_t csize = GET_SIZE(HDRP(bp));
86     if ((csize - asize) >= (2 * DSIZE)) {
87         PUT(HDRP(bp), PACK(asize, 1));
88         PUT(FTRP(bp), PACK(asize, 1));
89         bp = NEXT_BLK(P(bp));
90         PUT(HDRP(bp), PACK(csize - asize, 0));
91         PUT(FTRP(bp), PACK(csize - asize, 0));
92     } else {
93         PUT(HDRP(bp), PACK(csize, 1));
94         PUT(FTRP(bp), PACK(csize, 1));
95     }
96 }
97
98 static void *coalesce(void *bp) {
99     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLK(P(bp))));
100    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLK(P(bp))));
101    size_t size = GET_SIZE(HDRP(bp));
102    if (prev_alloc && next_alloc) {
103        return bp;
104    } else if (prev_alloc && !next_alloc) {
105        size += GET_SIZE(HDRP(NEXT_BLK(P(bp))));
106        PUT(HDRP(bp), PACK(size, 0));
107        PUT(FTRP(bp), PACK(size, 0));
108    } else if (!prev_alloc && next_alloc) {
109        size += GET_SIZE(HDRP(PREV_BLK(P(bp))));
110        PUT(FTRP(bp), PACK(size, 0));
111        PUT(HDRP(PREV_BLK(P(bp))), PACK(size, 0));
112        bp = PREV_BLK(P(bp));
113    } else {

```

```

114     size += GET_SIZE(FTRP(NEXT_BLK(bp))) + GET_SIZE(HDRP(PREV_BLK(bp)));
115     PUT(FTRP(NEXT_BLK(bp)), PACK(size, 0));
116     PUT(HDRP(PREV_BLK(bp)), PACK(size, 0));
117     bp = PREV_BLK(bp);
118 }
119 return bp;
120 }
121
122 static void *extend_heap(size_t words) {
123     char *bp;
124     size_t size;
125     size = (words % 2) ? (words + 1) * WSIZE : words * WSIZE;
126     if ((long) (bp = mem_sbrk(size)) == (void *) -1)
127         return NULL;
128     PUT(HDRP(bp), PACK(size, 0));
129     PUT(FTRP(bp), PACK(size, 0));
130     PUT(HDRP(NEXT_BLK(bp)), PACK(0, 1));
131     return coalesce(bp);
132 }
133
134 static void *find_fit(size_t asize) {
135     void *bp;
136     for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLK(bp)) {
137         if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
138             return bp;
139         }
140     }
141     return NULL;
142 }
143
144 /*
145  * mm_free — Freeing a block does nothing.
146  */
147 void mm_free(void *bp) {
148     if (bp == 0)
149         return; // 无效指针
150     size_t size = GET_SIZE(HDRP(bp)); // 获取块大小
151     PUT(HDRP(bp), PACK(size, 0)); // 标记为未分配
152     PUT(FTRP(bp), PACK(size, 0)); // 标记为未分配
153     coalesce(bp); // 合并空闲块
154 }
155
156 /*
157  * mm_realloc — Implemented simply in terms of mm_malloc and mm_free
158  */
159 void *mm_realloc(void *ptr, size_t size) {
160     size_t oldsize;
161     void *newptr;
162     /* If size == 0 then this is just free, and we return NULL. */
163     if (size == 0) {
164         mm_free(ptr);
165         return 0;
166     }
167     /* If oldptr is NULL, then this is just malloc. */
168     if (ptr == NULL) {
169         return mm_malloc(size);
170     }
171     newptr = mm_malloc(size);
172     /* If realloc() fails the original block is left untouched */
173     if (!newptr) {
174         return 0;
175     }
176     /* Copy the old data. */
177     oldsize = GET_SIZE(HDRP(ptr));

```

```
178     if (size < oldsize)
179         oldsize = size;
180     memcpy(newptr, ptr, oldsize);
181     /* Free the old block. */
182     mm_free(ptr);
183     return newptr;
184 }
```

Appendix.c