

华东师范大学软件工程学院实验报告

姓 名:	李鹏达	学 号:	10225101460
实验编号:	Lab 05	实验名称:	Writing Your Own Malloc Package

1 实验目的

- 1) 深入了解动态内存分配
- 2) 实现一个动态内存分配器

2 实验内容与实验步骤

2.1 实验内容

在本实验中，您将为 C 程序编写一个动态存储分配器，即您自己的 `malloc`，`free` 和 `realloc` 函数。我们鼓励您创造性地探索设计空间，实施正确、高效和快速的分配器。

我们将从吞吐量（单位时间可执行次数）和空间利用率两个方面对您的分配器进行评估。

2.1.1 隐式空闲链表

参考课本 9.9.12 的内容，我们先考虑使用隐式空闲链表来实现。

首先，我们先定义在分配器编码中所需的基本常数和宏。

其中，在第 1 至 3 行，我们定义了所使用的基本常数——字的大小 `WSIZE` 和双字的大小 `DSIZE`，以及初始空闲块的大小和扩展时堆的默认大小 `CHUNKSIZE`。

在第 7 行中，我们使用 `PACK` 宏来将大小和已分配位结合起来返回一个值，用于存放在头部或者脚部中。9 至 10 行中，我们定义了 `GET` 和 `PUT` 宏来对参数 `p` 指向的字进行读取或写入。12 至 13 行中，宏 `GET_SIZE` 和 `GET_ALLOC` 可以从地址 `p` 的头部或脚部分别返回大小和已分配位。15 至 16 行中，宏 `HDRP` 和 `FTRP` 分别返回指向块 `bp` 的头部和脚部的指针。18 至 19 行中，`NEXT_BLK` 和 `PREV_BLK` 则分别返回指向后面的块和指向前面的块的指针。

```
1 #define WSIZE 4
2 #define DSIZE 8
3 #define CHUNKSIZE (1 << 12)
4
5 #define MAX(x, y) ((x) > (y) ? (x) : (y))
6
7 #define PACK(size, alloc) ((size) | (alloc))
8
```

```

9 #define GET(p) (*(unsigned int *)(p))
10 #define PUT(p, val) (*(unsigned int *)(p) = (val))
11
12 #define GET_SIZE(p) (GET(p) & ~0x7)
13 #define GET_ALLOC(P) (GET(P) & 0x1)
14
15 #define HDRP(bp) ((char *)(bp) - WSIZE)
16 #define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
17
18 #define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
19 #define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))

```

接下来，我们需要将堆初始化。

首先我们申请四个字的空间，然后对这四个字进行初始化，第一个字是不使用的填充字，后两个字是序言块，最后一个字是结尾块。然后我们创建一个初始的空闲块。

```

1 int mm_init(void) {
2     if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1) {
3         return -1;
4     }
5     PUT(heap_listp, 0);
6     PUT(heap_listp + 1 * WSIZE, PACK(DSIZE, 1));
7     PUT(heap_listp + 2 * WSIZE, PACK(DSIZE, 1));
8     PUT(heap_listp + 3 * WSIZE, PACK(0, 1));
9     heap_listp += 2 * WSIZE;
10    if (extend_heap(CHUNKSIZE / WSIZE) == NULL) {
11        return -1;
12    }
13    return 0;
14 }

```

在初始化堆的过程中，我们使用了 `extend_heap` 函数，接下来我们需要编写这个函数来增长堆。首先我们需要将字节数转化为大小的时进行一个对齐操作。然后进行分配，以及设置头部和脚部。最后的时候还要调用合并空闲块函数，将附近空闲的块进行合并，将合并后的头指针返回。

```

1 static void *extend_heap(size_t words) {
2     char* bp;
3     size_t size;
4     size = (words & 1) ? (words + 1) * WSIZE : words * WSIZE;
5     if ((bp = mem_sbrk(size)) == (void *)-1) {
6         return NULL;
7     }
8     PUT(HDRP(bp), PACK(size, 0));
9     PUT(FTRP(bp), PACK(size, 0));
10    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));

```

```

11     return coalesce(bp);
12 }

```

在这个过程中，我们调用了 `coalesce` 函数，我们接下来需要编写这个函数来完成空闲块的合并。

合并空闲块时，我们需要先获取当前块的上一个块，以及下一个块，然后对四种情况进行讨论。情况 1：前后都有填充，那么不进行任何操作。情况 2：前面有填充，后面没有填充，那么指针不变，将空闲块合并即可。情况 3：前面有没有填充，后面有填充，那么需要将指针先前移动一个块，然后将空闲块合并。情况 4：前后都没有填充，那么将指针向前移动一个块的同时，将前后空闲块进行合并。

```

1 static void *coalesce(void *bp) {
2     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKPTR(bp)));
3     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKPTR(bp)));
4     size_t size = GET_SIZE(HDRP(bp));
5     if (prev_alloc && next_alloc) {
6         return bp;
7     } else if (prev_alloc && !next_alloc) {
8         size += GET_SIZE(HDRP(NEXT_BLKPTR(bp)));
9         PUT(HDRP(bp), PACK(size, 0));
10        PUT(FTRP(bp), PACK(size, 0));
11    } else if (!prev_alloc && next_alloc) {
12        size += GET_SIZE(HDRP(PREV_BLKPTR(bp)));
13        PUT(FTRP(bp), PACK(size, 0));
14        PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
15        bp = PREV_BLKPTR(bp);
16    } else {
17        size += GET_SIZE(HDRP(NEXT_BLKPTR(bp))) + GET_SIZE(HDRP(PREV_BLKPTR(bp)));
18        PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
19        PUT(FTRP(NEXT_BLKPTR(bp)), PACK(size, 0));
20        bp = PREV_BLKPTR(bp);
21    }
22    return bp;
23 }

```

同时，我们还需要一个函数 `place` 来实现对空闲块的分割，来提高空闲块的利用率，减少碎片的产生。如果当前空闲块的大小要比我们要分配的内容大且超过两个字，那么空闲块将被分成两部分，一部分是用来分配的，另一部分仍然是空闲的。

```

1 static void place(void *bp, size_t asize) {
2     size_t size = GET_SIZE(HDRP(bp));
3     if (size - asize >= 2 * DSIZ) {
4         PUT(HDRP(bp), PACK(asize, 1));
5         PUT(FTRP(bp), PACK(asize, 1));
6         size = size - asize;
7         bp = NEXT_BLKPTR(bp);

```

```
8      PUT(HDRP(bp), PACK(size, 0));
9      PUT(FTRP(bp), PACK(size, 0));
10     coalesce(bp);
11 } else {
12     PUT(HDRP(bp), PACK(size, 1));
13     PUT(FTRP(bp), PACK(size, 1));
14 }
15 }
```

下一步，我们来实现查找空闲块的策略。我们使用**首次适配策略**——从头往后遍历空闲链表，直至找到第一个可以放置空闲块的地方。

```
1 static void *find_fit(size_t asize) {
2     for (void *bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLK(bp)) {
3         if (!GET_ALLOC(HDRP(bp)) && GET_SIZE(HDRP(bp)) >= asize) {
4             return bp;
5         }
6     }
7     return NULL;
8 }
```

然后，我们来实现 `malloc` 函数。我们对 `size` 进行非 0 判断，然后对 `size` 进行按 8 对齐舍入，接着用首次适应策略去寻找空闲块，如果存在直接存入，并返回指针，如果不存在这么大的空闲块，那么对堆进行扩展。

```
1 void *mm_malloc(size_t size) {
2     size_t asize;
3     size_t extendsize;
4     char *bp;
5     if (size == 0) {
6         return NULL;
7     }
8     if (size < DSIZ) {
9         asize = 2 * DSIZ;
10    } else {
11        asize = DSIZ * ((size + DSIZ + DSIZ - 1) / DSIZ);
12    }
13    if ((bp = find_fit(asize)) != NULL) {
14        place(bp, asize);
15        return bp;
16    }
17    extendsize = MAX(asize, CHUNKSIZE);
18    if ((bp = extend_heap(extendsize / WSIZ)) == NULL) {
19        return NULL;
20    }
21    place(bp, asize);
22 }
```

```
22     return bp;
23 }
```

接下来我们实现 `free` 函数，将一个分配块转为空闲块。这需要我们修改它的头部和脚部，然后进行前后空闲块的合并。

```
1 void mm_free(void *ptr) {
2     if (ptr == NULL) {
3         return;
4     }
5     size_t size = GET_SIZE(HDRP(ptr));
6     PUT(HDRP(ptr), PACK(size, 0));
7     PUT(FTRP(ptr), PACK(size, 0));
8     coalesce(ptr);
9 }
```

`realloc` 函数则需要我们对已分配的块进行一个重新分配。可以根据原本已经写好的代码，只需要将内存分配的函数改为自己写的内存分配函数即可。

```
1 void *mm_realloc(void *ptr, size_t size) {
2     void *oldptr = ptr;
3     void *newptr;
4     size_t copySize;
5     newptr = mm_malloc(size);
6     if (newptr == NULL) return NULL;
7     copySize = GET_SIZE(HDRP(ptr));
8     if (size < copySize) copySize = size;
9     memcpy(newptr, oldptr, copySize);
10    mm_free(oldptr);
11    return newptr;
12 }
```

2.1.2 显式空闲链表

一种更好的方法是将空闲块组织为某种形式的显式数据结构。因为根据定义，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。

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。

我们使用**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。

我们更改空闲块的结构，在其有效载荷的开始处存放其前驱和后继指针。我们额外定义一些宏来完成其前驱和后继的读取与写入。

```
1 #define GET_PREV(p) (*(unsigned int *)(p))
2 #define SET_PREV(p, val) (*(unsigned int *)(p) = (val))
3 #define GET_NEXT(p) (*(unsigned int *)(p) + 1)
4 #define SET_NEXT(p, val) (*(unsigned int *)(p) + 1) = (val))
```

然后，我们编写函数 `list_insert` 和 `list_remove` 来用于链表中节点的插入与删除。我们使用后进先出的顺序维护链表，即始终在链表的头部插入节点。

```
1 static void list_insert(void *bp) {
2     if (bp == NULL) return;
3     if (list_head == NULL) {
4         list_head = bp;
5         return;
6     }
7     SET_NEXT(bp, (list_head));
8     SET_PREV(list_head, (bp));
9     list_head = bp;
10 }
11
12 static void list_remove(void *bp) {
13     if (bp == NULL || GET_ALLOC(HDRP(bp))) return;
14     void* prev = GET_PREV(bp);
15     void* next = GET_NEXT(bp);
16     SET_PREV(bp, 0);
17     SET_NEXT(bp, 0);
18     if (prev == NULL && next == NULL) {
19         list_head = NULL;
20     } else if (prev == NULL) {
21         SET_PREV(next, 0);
22         list_head = next;
23     } else if (next == NULL) {
24         SET_NEXT(prev, 0);
25     } else {
26         SET_NEXT(prev, (next));
27         SET_PREV(next, (prev));
28     }
29 }
```

然后，我们对 `coalesce` 函数、`extend_heap` 函数、`place` 函数和 `mm_free` 函数进行修改，使其在合并、增长堆、拆分和释放时均能够维护空闲链表。

```
1 static void *coalesce(void *bp) {
```

```
2     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLK(bp)));
3     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLK(bp)));
4     size_t size = GET_SIZE(HDRP(bp));
5     if (prev_alloc && next_alloc) {
6         /*****
7             list_insert(bp);
8         *****/
9         return bp;
10    } else if (prev_alloc && !next_alloc) {
11        /*****
12            list_remove(NEXT_BLK(bp));
13        *****/
14        size += GET_SIZE(HDRP(NEXT_BLK(bp)));
15        PUT(HDRP(bp), PACK(size, 0));
16        PUT(FTRP(bp), PACK(size, 0));
17    } else if (!prev_alloc && next_alloc) {
18        /*****
19            list_remove(PREV_BLK(bp));
20        *****/
21        size += GET_SIZE(HDRP(PREV_BLK(bp)));
22        PUT(FTRP(bp), PACK(size, 0));
23        PUT(HDRP(PREV_BLK(bp)), PACK(size, 0));
24        bp = PREV_BLK(bp);
25    } else {
26        /*****
27            list_remove(NEXT_BLK(bp));
28            list_remove(PREV_BLK(bp));
29        *****/
30        size += GET_SIZE(HDRP(NEXT_BLK(bp))) + GET_SIZE(HDRP(PREV_BLK(bp)));
31        PUT(HDRP(PREV_BLK(bp)), PACK(size, 0));
32        PUT(FTRP(NEXT_BLK(bp)), PACK(size, 0));
33        bp = PREV_BLK(bp);
34    }
35    /*****
36        list_insert(bp);
37    *****/
38    return bp;
39 }
40
41 static void *extend_heap(size_t words) {
42     char* bp;
43     size_t size;
44     size = (words & 1) ? (words + 1) * WSIZE : words * WSIZE;
45     if ((bp = mem_sbrk(size)) == (void *)-1) {
```

```

46         return NULL;
47     }
48     PUT(HDRP(bp), PACK(size, 0));
49     PUT(FTRP(bp), PACK(size, 0));
50     /*****
51     SET_NEXT(bp, 0);
52     SET_PREV(bp, 0);
53     /*****
54     PUT(HDRP(NEXT_BLKp(bp)), PACK(0, 1));
55     return coalesce(bp);
56 }
57
58 static void place(void *bp, size_t asize) {
59     size_t size = GET_SIZE(HDRP(bp));
60     list_remove(bp);
61     if (size - asize >= 2 * DSIZe) {
62         PUT(HDRP(bp), PACK(asize, 1));
63         PUT(FTRP(bp), PACK(asize, 1));
64         size = size - asize;
65         bp = NEXT_BLKp(bp);
66     /*****
67         SET_NEXT(bp, 0);
68         SET_PREV(bp, 0);
69     /*****
70         PUT(HDRP(bp), PACK(size, 0));
71         PUT(FTRP(bp), PACK(size, 0));
72         coalesce(bp);
73     } else {
74         PUT(HDRP(bp), PACK(size, 1));
75         PUT(FTRP(bp), PACK(size, 1));
76     }
77 }
78
79 void mm_free(void *ptr) {
80     if (ptr == NULL) {
81         return;
82     }
83     size_t size = GET_SIZE(HDRP(ptr));
84     PUT(HDRP(ptr), PACK(size, 0));
85     PUT(FTRP(ptr), PACK(size, 0));
86     /*****
87     SET_NEXT(ptr, 0);
88     SET_PREV(ptr, 0);
89     /*****

```



```
90     coalesce(ptr);  
91 }
```

最后，我们重新编写 `find_fit` 函数，进而实现在显式空闲链表中的查找。

```
1 static void *find_fit(size_t asize) {  
2     for (void *bp = list_head; bp; bp = GET_NEXT(bp)) {  
3         if (GET_SIZE(HDRP(bp)) >= asize) {  
4             return bp;  
5         }  
6     }  
7     return NULL;  
8 }
```

2.2 实验步骤

- 1) 解打包 `malloc-handout.tar`

```
1 linux> tar -xvf malloc-handout.tar
```

- 2) 阅读要求，编写 `mm.c`
- 3) 编译

```
1 linux> make
```

- 4) 评测

```
1 linux> ./mdriver -av -t traces
```

3 实验过程与分析

3.1 隐式空闲链表

实验的运行结果如下：

```
~/De/lab5/malloclab-handout ./mdriver -av -t traces at 04:21:58
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace valid util ops secs Kops
0 yes 93% 5694 0.005181 1099
1 yes 91% 5848 0.005054 1157
2 yes 91% 6648 0.007662 868
3 yes 90% 5380 0.005887 914
4 yes 66% 14400 0.000066218845
5 yes 93% 4800 0.004593 1045
6 yes 91% 4800 0.004413 1088
7 yes 55% 12000 0.063900 188
8 yes 51% 24000 0.216020 111
9 yes 26% 14401 0.033895 425
10 yes 34% 14401 0.001454 9908
Total 71% 112372 0.348123 323

Perf index = 43 (util) + 22 (thru) = 64/100
```

图 1: 运行结果

3.2 显式空闲链表

实验的运行结果如下:

```
~/De/lab5/malloclab-handout ./mdriver -av -t traces at 04:24:15
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace valid util ops secs Kops
0 yes 89% 5694 0.000167 34198
1 yes 92% 5848 0.000102 57616
2 yes 94% 6648 0.000253 26266
3 yes 96% 5380 0.000211 25498
4 yes 66% 14400 0.000106135338
5 yes 88% 4800 0.000389 12339
6 yes 85% 4800 0.000420 11431
7 yes 55% 12000 0.001668 7194
8 yes 51% 24000 0.001647 14573
9 yes 26% 14401 0.034568 417
10 yes 34% 14401 0.001485 9696
Total 71% 112372 0.041016 2740

Perf index = 42 (util) + 40 (thru) = 82/100
```

图 2: 运行结果

4 实验结果总结

通过本次实验,我对动态分配内存的内容有了一个深刻的印象,对概念的掌握更加清晰,也充分明白了该如何去利用 C 语言对分配器进行模拟,对指针的运用也有了一定的能力提升。

实验中也存在一些不足,例如可以使用分离空闲链表来进一步优化, `realloc` 函数的效率也有待提高。

5 附录（源代码）

5.1 隐式空闲链表

```
1  /*
2   * mm-naive.c - The fastest, least memory-efficient malloc package.
3   *
4   * In this naive approach, a block is allocated by simply incrementing
5   * the brk pointer.  A block is pure payload. There are no headers or
6   * footers.  Blocks are never coalesced or reused. Realloc is
7   * implemented directly using mm_malloc and mm_free.
8   *
9   * NOTE TO STUDENTS: Replace this header comment with your own header
10  * comment that gives a high level description of your solution.
11  */
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <assert.h>
15 #include <unistd.h>
16 #include <string.h>
17
18 #include "mm.h"
19 #include "memlib.h"
20
21 /*****
22  * NOTE TO STUDENTS: Before you do anything else, please
23  * provide your team information in the following struct.
24  *****/
25 team_t team = {
26     /* Team name */
27     "team",
28     /* First member's full name */
29     "Pengda Li",
30     /* First member's email address */
31     "10225101460@stu.ecnu.edu.cn",
32     /* Second member's full name (leave blank if none) */
33     "",
34     /* Second member's email address (leave blank if none) */
35     ""
36 };
37
38 /* single word (4) or double word (8) alignment */
39 #define ALIGNMENT 8
40
```

```
41 /* rounds up to the nearest multiple of ALIGNMENT */
42 #define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)
43
44 #define SIZE_T_SIZE (ALIGN(sizeof(size_t)))
45
46 // Basic constants and macros
47 #define WSIZE 4
48 #define DSIZE 8
49 #define CHUNKSIZE (1 << 12)
50
51 #define MAX(x, y) ((x) > (y) ? (x) : (y))
52
53 #define PACK(size, alloc) ((size) | (alloc))
54
55 #define GET(p) (*(unsigned int *)(p))
56 #define PUT(p, val) (*(unsigned int *)(p) = (val))
57
58 #define GET_SIZE(p) (GET(p) & ~0x7)
59 #define GET_ALLOC(P) (GET(P) & 0x1)
60
61 #define HDRP(bp) ((char *)(bp) - WSIZE)
62 #define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
63
64 #define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
65 #define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
66
67 static char * heap_listp;
68
69 static void *coalesce(void *bp) {
70     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
71     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
72     size_t size = GET_SIZE(HDRP(bp));
73     if (prev_alloc && next_alloc) {
74         return bp;
75     } else if (prev_alloc && !next_alloc) {
76         size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
77         PUT(HDRP(bp), PACK(size, 0));
78         PUT(FTRP(bp), PACK(size, 0));
79     } else if (!prev_alloc && next_alloc) {
80         size += GET_SIZE(HDRP(PREV_BLKP(bp)));
81         PUT(FTRP(bp), PACK(size, 0));
82         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
83         bp = PREV_BLKP(bp);
84     } else {
```

```
85     size += GET_SIZE(HDRP(NEXT_BLK(b))) + GET_SIZE(HDRP(PREV_BLK(b)));
86     PUT(HDRP(PREV_BLK(b)), PACK(size, 0));
87     PUT(FTRP(NEXT_BLK(b)), PACK(size, 0));
88     bp = PREV_BLK(b);
89 }
90 return bp;
91 }
92
93 static void *extend_heap(size_t words) {
94     char* bp;
95     size_t size;
96     size = (words & 1) ? (words + 1) * WSIZE : words * WSIZE;
97     if ((bp = mem_sbrk(size)) == (void *)-1) {
98         return NULL;
99     }
100     PUT(HDRP(bp), PACK(size, 0));
101     PUT(FTRP(bp), PACK(size, 0));
102     PUT(HDRP(NEXT_BLK(bp)), PACK(0, 1));
103     return coalesce(bp);
104 }
105
106 static void *find_fit(size_t asize) {
107     for (void *bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLK(bp)) {
108         if (!GET_ALLOC(HDRP(bp)) && GET_SIZE(HDRP(bp)) > asize) {
109             return bp;
110         }
111     }
112     return NULL;
113 }
114
115 static void place(void *bp, size_t asize) {
116     size_t size = GET_SIZE(HDRP(bp));
117     if (size - asize >= 2 * DSIZE) {
118         PUT(HDRP(bp), PACK(asize, 1));
119         PUT(FTRP(bp), PACK(asize, 1));
120         size = size - asize;
121         bp = NEXT_BLK(bp);
122         PUT(HDRP(bp), PACK(size, 0));
123         PUT(FTRP(bp), PACK(size, 0));
124     } else {
125         PUT(HDRP(bp), PACK(size, 1));
126         PUT(FTRP(bp), PACK(size, 1));
127     }
128 }
```

```
129
130
131 /*
132 * mm_init - initialize the malloc package.
133 */
134 int mm_init(void) {
135     if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1) {
136         return -1;
137     }
138     PUT(heap_listp, 0);
139     PUT(heap_listp + 1 * WSIZE, PACK(DSIZE, 1));
140     PUT(heap_listp + 2 * WSIZE, PACK(DSIZE, 1));
141     PUT(heap_listp + 3 * WSIZE, PACK(0, 1));
142     heap_listp += 2 * WSIZE;
143     if (extend_heap(CHUNKSIZE / WSIZE) == NULL) {
144         return -1;
145     }
146     return 0;
147 }
148
149 /*
150 * mm_malloc - Allocate a block by incrementing the brk pointer.
151 *     Always allocate a block whose size is a multiple of the alignment.
152 */
153 void *mm_malloc(size_t size) {
154     size_t asize;
155     size_t extendsize;
156     char *bp;
157     if (size == 0) {
158         return NULL;
159     }
160     if (size < DSIZE) {
161         asize = 2 * DSIZE;
162     } else {
163         asize = DSIZE * ((size + DSIZE + DSIZE - 1) / DSIZE);
164     }
165     if ((bp = find_fit(asize)) != NULL) {
166         place(bp, asize);
167         return bp;
168     }
169     extendsize = MAX(asize, CHUNKSIZE);
170     if ((bp = extend_heap(extendsize / WSIZE)) == NULL) {
171         return NULL;
172     }
```

```
173     place(bp, asize);
174     return bp;
175 }
176
177 /*
178  * mm_free - Freeing a block does nothing.
179  */
180 void mm_free(void *ptr) {
181     if (ptr == NULL) {
182         return;
183     }
184     size_t size = GET_SIZE(HDRP(ptr));
185     PUT(HDRP(ptr), PACK(size, 0));
186     PUT(FTRP(ptr), PACK(size, 0));
187     coalesce(ptr);
188 }
189
190 /*
191  * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
192  */
193 void *mm_realloc(void *ptr, size_t size) {
194     void *oldptr = ptr;
195     void *newptr;
196     size_t copySize;
197     newptr = mm_malloc(size);
198     if (newptr == NULL) return NULL;
199     copySize = GET_SIZE(HDRP(ptr));
200     if (size < copySize) copySize = size;
201     memcpy(newptr, oldptr, copySize);
202     mm_free(oldptr);
203     return newptr;
204 }
```

5.2 显式空闲链表

```
1  /*
2  * mm-naive.c - The fastest, least memory-efficient malloc package.
3  *
4  * In this naive approach, a block is allocated by simply incrementing
5  * the brk pointer. A block is pure payload. There are no headers or
6  * footers. Blocks are never coalesced or reused. Realloc is
7  * implemented directly using mm_malloc and mm_free.
8  *
```

```
9  * NOTE TO STUDENTS: Replace this header comment with your own header
10 * comment that gives a high level description of your solution.
11 */
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <assert.h>
15 #include <unistd.h>
16 #include <string.h>
17
18 #include "mm.h"
19 #include "memlib.h"
20
21 /*****
22  * NOTE TO STUDENTS: Before you do anything else, please
23  * provide your team information in the following struct.
24  *****/
25 team_t team = {
26     /* Team name */
27     "team",
28     /* First member's full name */
29     "Pengda Li",
30     /* First member's email address */
31     "10225101460@stu.ecnu.edu.cn",
32     /* Second member's full name (leave blank if none) */
33     "",
34     /* Second member's email address (leave blank if none) */
35     ""
36 };
37
38 /* single word (4) or double word (8) alignment */
39 #define ALIGNMENT 8
40
41 /* rounds up to the nearest multiple of ALIGNMENT */
42 #define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)
43
44 #define SIZE_T_SIZE (ALIGN(sizeof(size_t)))
45
46 // Basic constants and macros
47 #define WSIZE 4
48 #define DSIZE 8
49 #define CHUNKSIZE (1 << 12)
50
51 #define MAX(x, y) ((x) > (y) ? (x) : (y))
52
```



```
53 #define PACK(size, alloc) ((size) | (alloc))
54
55 #define GET(p) (*(unsigned int *)(p))
56 #define PUT(p, val) (*(unsigned int *)(p) = (val))
57
58 #define GET_SIZE(p) (GET(p) & ~0x7)
59 #define GET_ALLOC(P) (GET(P) & 0x1)
60
61 #define HDRP(bp) ((char *)(bp) - WSIZE)
62 #define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
63
64 #define NEXT_BLK(p) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
65 #define PREV_BLK(p) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
66
67 #define GET_PREV(p) (*(unsigned int *)(p))
68 #define SET_PREV(p, val) (*(unsigned int *)(p) = (val))
69 #define GET_NEXT(p) (*(unsigned int *)(p) + 1)
70 #define SET_NEXT(p, val) (*(unsigned int *)(p) + 1) = (val))
71
72 static char * heap_listp;
73 static char * head;
74
75 static void list_insert(void *bp) {
76     if (bp == NULL) return;
77     if (head == NULL) {
78         head = bp;
79         return;
80     }
81     SET_NEXT(bp, (head));
82     SET_PREV(head, (bp));
83     head = bp;
84 }
85
86 static void list_remove(void *bp) {
87     if (bp == NULL || GET_ALLOC(HDRP(bp))) return;
88     void* prev = GET_PREV(bp);
89     void* next = GET_NEXT(bp);
90     SET_PREV(bp, 0);
91     SET_NEXT(bp, 0);
92     if (prev == NULL && next == NULL) {
93         head = NULL;
94     } else if (prev == NULL) {
95         SET_PREV(next, 0);
96         head = next;
```

```
97     } else if (next == NULL) {
98         SET_NEXT(prev, 0);
99     } else {
100         SET_NEXT(prev, (next));
101         SET_PREV(next, (prev));
102     }
103 }
104
105 static void *coalesce(void *bp) {
106     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKPTR(bp)));
107     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKPTR(bp)));
108     size_t size = GET_SIZE(HDRP(bp));
109     if (prev_alloc && next_alloc) {
110         list_insert(bp);
111         return bp;
112     } else if (prev_alloc && !next_alloc) {
113         list_remove(NEXT_BLKPTR(bp));
114         size += GET_SIZE(HDRP(NEXT_BLKPTR(bp)));
115         PUT(HDRP(bp), PACK(size, 0));
116         PUT(FTRP(bp), PACK(size, 0));
117     } else if (!prev_alloc && next_alloc) {
118         list_remove(PREV_BLKPTR(bp));
119         size += GET_SIZE(HDRP(PREV_BLKPTR(bp)));
120         PUT(FTRP(bp), PACK(size, 0));
121         PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
122         bp = PREV_BLKPTR(bp);
123     } else {
124         list_remove(NEXT_BLKPTR(bp));
125         list_remove(PREV_BLKPTR(bp));
126         size += GET_SIZE(HDRP(NEXT_BLKPTR(bp))) + GET_SIZE(HDRP(PREV_BLKPTR(bp)));
127         PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
128         PUT(FTRP(NEXT_BLKPTR(bp)), PACK(size, 0));
129         bp = PREV_BLKPTR(bp);
130     }
131     list_insert(bp);
132     return bp;
133 }
134
135 static void *extend_heap(size_t words) {
136     char* bp;
137     size_t size;
138     size = (words & 1) ? (words + 1) * WSIZE : words * WSIZE;
139     if ((bp = mem_sbrk(size)) == (void *)-1) {
140         return NULL;
141     }
```

```
141     }
142     PUT(HDRP(bp), PACK(size, 0));
143     PUT(FTRP(bp), PACK(size, 0));
144     SET_NEXT(bp, 0);
145     SET_PREV(bp, 0);
146     PUT(HDRP(NEXT_BLKp(bp)), PACK(0, 1));
147     return coalesce(bp);
148 }
149
150 static void *find_fit(size_t asize) {
151     for (void *bp = head; bp; bp = GET_NEXT(bp)) {
152         if (GET_SIZE(HDRP(bp)) >= asize) {
153             return bp;
154         }
155     }
156     return NULL;
157 }
158
159 static void place(void *bp, size_t asize) {
160     size_t size = GET_SIZE(HDRP(bp));
161     list_remove(bp);
162     if (size - asize >= 2 * DSIZE) {
163         PUT(HDRP(bp), PACK(asize, 1));
164         PUT(FTRP(bp), PACK(asize, 1));
165         size = size - asize;
166         bp = NEXT_BLKp(bp);
167         SET_NEXT(bp, 0);
168         SET_PREV(bp, 0);
169         PUT(HDRP(bp), PACK(size, 0));
170         PUT(FTRP(bp), PACK(size, 0));
171         coalesce(bp);
172     } else {
173         PUT(HDRP(bp), PACK(size, 1));
174         PUT(FTRP(bp), PACK(size, 1));
175     }
176 }
177
178
179 /*
180  * mm_init - initialize the malloc package.
181  */
182 int mm_init(void) {
183     if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1) {
184         return -1;
```

```
185     }
186     head = NULL;
187     PUT(heap_listp, 0);
188     PUT(heap_listp + 1 * WSIZE, PACK(DSIZE, 1));
189     PUT(heap_listp + 2 * WSIZE, PACK(DSIZE, 1));
190     PUT(heap_listp + 3 * WSIZE, PACK(0, 1));
191     heap_listp += 2 * WSIZE;
192     if (extend_heap(CHUNKSIZE / WSIZE) == NULL) {
193         return -1;
194     }
195     return 0;
196 }
197
198 /*
199  * mm_malloc - Allocate a block by incrementing the brk pointer.
200  *     Always allocate a block whose size is a multiple of the alignment.
201  */
202 void *mm_malloc(size_t size) {
203     size_t asize;
204     size_t extendsize;
205     char *bp;
206     if (size == 0) {
207         return NULL;
208     }
209     if (size < DSIZE) {
210         asize = 2 * DSIZE;
211     } else {
212         asize = DSIZE * ((size + DSIZE + DSIZE - 1) / DSIZE);
213     }
214     if ((bp = find_fit(asize)) != NULL) {
215         place(bp, asize);
216         return bp;
217     }
218     extendsize = MAX(asize, CHUNKSIZE);
219     if ((bp = extend_heap(extendsize / WSIZE)) == NULL) {
220         return NULL;
221     }
222     place(bp, asize);
223     return bp;
224 }
225
226 /*
227  * mm_free - Freeing a block does nothing.
228  */
```

```
229 void mm_free(void *ptr) {
230     if (ptr == NULL) {
231         return;
232     }
233     size_t size = GET_SIZE(HDRP(ptr));
234     PUT(HDRP(ptr), PACK(size, 0));
235     PUT(FTRP(ptr), PACK(size, 0));
236     SET_NEXT(ptr, 0);
237     SET_PREV(ptr, 0);
238     coalesce(ptr);
239 }
240
241 /*
242  * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
243  */
244 void *mm_realloc(void *ptr, size_t size) {
245     void *oldptr = ptr;
246     void *newptr;
247     size_t copySize;
248     newptr = mm_malloc(size);
249     if (newptr == NULL) return NULL;
250     copySize = GET_SIZE(HDRP(ptr));
251     if (size < copySize) copySize = size;
252     memcpy(newptr, oldptr, copySize);
253     mm_free(oldptr);
254     return newptr;
255 }
```