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要的基本常数和宏。

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_BLKP` 和 `PREV_BLKP` 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
```

```c
#define GET(p) (*(unsigned int *)(p))
#define PUT(p, val) (*(unsigned int *)(p) = (val))

#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(P) (GET(P) & 0x1)

#define HDRP(bp) ((char *)(bp) - WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#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。

```c
int mm_init(void) {
    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1) {
        return -1;
    }
    PUT(heap_listp, 0);
    PUT(heap_listp + 1 * WSIZE, PACK(DSIZE, 1));
    PUT(heap_listp + 2 * WSIZE, PACK(DSIZE, 1));
    PUT(heap_listp + 3 * WSIZE, PACK(0, 1));
    heap_listp += 2 * WSIZE;
    if (extend_heap(CHUNKSIZE / WSIZE) == NULL) {
        return -1;
    }
    return 0;
}
```

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。

```c
static void *extend_heap(size_t words) {
    char* bp;
    size_t size;
    size = (words & 1) ? (words + 1) * WSIZE : words * WSIZE;
    if ((bp = mem_sbrk(size)) == (void *)-1) {
        return NULL;
    }
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));
```

```c
    return coalesce(bp);
}
```

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。

```c
static void *coalesce(void *bp) {
    size_t prev_alloc = GET_ALLOC(FTRP(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) {
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    } else if (!prev_alloc && 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(NEXT_BLKP(bp))) + GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    return bp;
}
```

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。

```c
static void place(void *bp, size_t asize) {
    size_t size = GET_SIZE(HDRP(bp));
    if (size - asize >= 2 * DSIZE) {
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        size = size - asize;
        bp = NEXT_BLKP(bp);
```

```
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
        coalesce(bp);
    } else {
        PUT(HDRP(bp), PACK(size, 1));
        PUT(FTRP(bp), PACK(size, 1));
    }
}
```

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。

```
static void *find_fit(size_t asize) {
    for (void *bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
        if (!GET_ALLOC(HDRP(bp)) && GET_SIZE(HDRP(bp)) >= asize) {
            return bp;
        }
    }
    return NULL;
}
```

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。

```
void *mm_malloc(size_t size) {
    size_t asize;
    size_t extendsize;
    char *bp;
    if (size == 0) {
        return NULL;
    }
    if (size < DSIZE) {
        asize = 2 * DSIZE;
    } else {
        asize = DSIZE * ((size + DSIZE + DSIZE - 1) / DSIZE);
    }
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize / WSIZE)) == NULL) {
        return NULL;
    }
    place(bp, asize);
```

```
    return bp;
}
```

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。

```
void mm_free(void *ptr) {
    if (ptr == NULL) {
        return;
    }
    size_t size = GET_SIZE(HDRP(ptr));
    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));
    coalesce(ptr);
}
```

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。

```
void *mm_realloc(void *ptr, size_t size) {
    void *oldptr = ptr;
    void *newptr;
    size_t copySize;
    newptr = mm_malloc(size);
    if (newptr == NULL) return NULL;
    copySize = GET_SIZE(HDRP(ptr));
    if (size < copySize) copySize = size;
    memcpy(newptr, oldptr, copySize);
    mm_free(oldptr);
    return newptr;
}
```

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。

```
#define GET_PREV(p) (*(unsigned int *)(p))
#define SET_PREV(p, val) (*(unsigned int *)(p) = (val))
#define GET_NEXT(p) (*((unsigned int *)(p) + 1))
#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。

```
static void list_insert(void *bp) {
    if (bp == NULL) return;
    if (list_head == NULL) {
      list_head = bp;
      return;
    }
    SET_NEXT(bp, (list_head));
    SET_PREV(list_head, (bp));
    list_head = bp;
}

static void list_remove(void *bp) {
    if (bp == NULL || GET_ALLOC(HDRP(bp))) return;
    void* prev = GET_PREV(bp);
    void* next = GET_NEXT(bp);
    SET_PREV(bp, 0);
    SET_NEXT(bp, 0);
    if (prev == NULL && next == NULL) {
      list_head = NULL;
    } else if (prev == NULL) {
        SET_PREV(next, 0);
        list_head = next;
    } else if (next == NULL) {
        SET_NEXT(prev, 0);
    } else {
        SET_NEXT(prev, (next));
        SET_PREV(next, (prev));
    }
}
```

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。

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

```
    size_t prev_alloc = GET_ALLOC(FTRP(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) {
/*****************************************************************************/
        list_insert(bp);
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
        return bp;
    } else if (prev_alloc && !next_alloc) {
/*****************************************************************************/
        list_remove(NEXT_BLKP(bp));
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    } else if (!prev_alloc && next_alloc) {
/*****************************************************************************/
        list_remove(PREV_BLKP(bp));
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
        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 {
/*****************************************************************************/
        list_remove(NEXT_BLKP(bp));
        list_remove(PREV_BLKP(bp));
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
        size += GET_SIZE(HDRP(NEXT_BLKP(bp))) + GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
/*****************************************************************************/
    list_insert(bp);
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
    return bp;
}

static void *extend_heap(size_t words) {
    char* bp;
    size_t size;
    size = (words & 1) ? (words + 1) * WSIZE : words * WSIZE;
    if ((bp = mem_sbrk(size)) == (void *)-1) {
```

```c
        return NULL;
    }
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
/**********************************************************************/
    SET_NEXT(bp, 0);
    SET_PREV(bp, 0);
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));
    return coalesce(bp);
}

static void place(void *bp, size_t asize) {
    size_t size = GET_SIZE(HDRP(bp));
    list_remove(bp);
    if (size - asize >= 2 * DSIZE) {
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        size = size - asize;
        bp = NEXT_BLKP(bp);
/**********************************************************************/
        SET_NEXT(bp, 0);
        SET_PREV(bp, 0);
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
        coalesce(bp);
    } else {
        PUT(HDRP(bp), PACK(size, 1));
        PUT(FTRP(bp), PACK(size, 1));
    }
}

void mm_free(void *ptr) {
    if (ptr == NULL) {
        return;
    }
    size_t size = GET_SIZE(HDRP(ptr));
    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));
/**********************************************************************/
    SET_NEXT(ptr, 0);
    SET_PREV(ptr, 0);
/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
```

```
    coalesce(ptr);
}
```

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。

```c
static void *find_fit(size_t asize) {
    for (void *bp = list_head; bp; bp = GET_NEXT(bp)) {
        if (GET_SIZE(HDRP(bp)) >= asize) {
            return bp;
        }
    }
    return NULL;
}
```

2.2 实验步骤

1) 解打包 malloc-handout.tar

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

2) 阅读要求，编写 mm.c
3) 编译

```
linux> make
```

4) 评测

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

3 实验过程与分析

3.1 隐式空闲链表

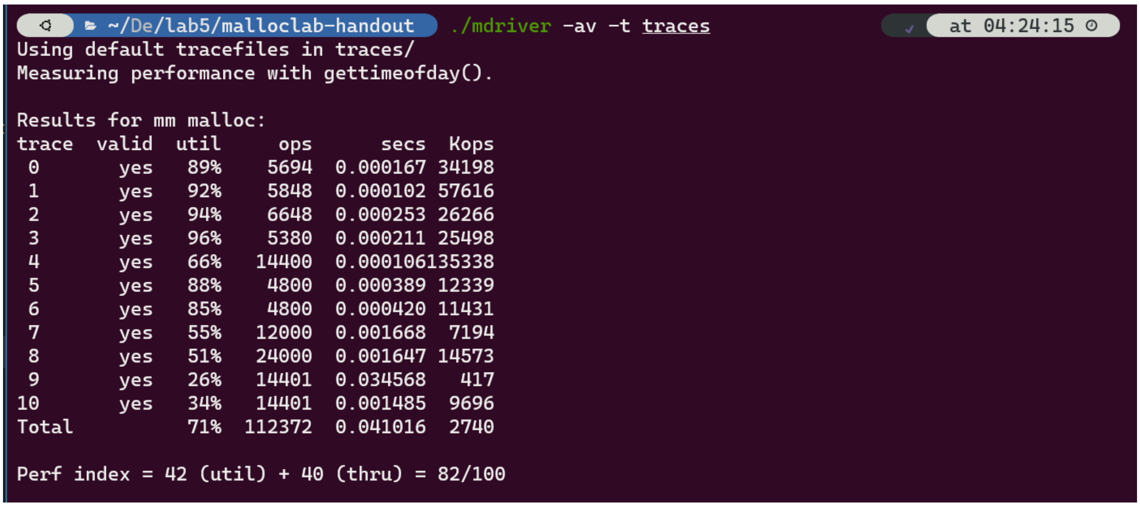
实验的运行结果如下：

```
~/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

```c
/*
 * mm-naive.c - The fastest, least memory-efficient malloc package.
 *
 * In this naive approach, a block is allocated by simply incrementing
 * the brk pointer.  A block is pure payload. There are no headers or
 * footers.  Blocks are never coalesced or reused. Realloc is
 * implemented directly using mm_malloc and mm_free.
 *
 * NOTE TO STUDENTS: Replace this header comment with your own header
 * comment that gives a high level description of your solution.
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>
#include <string.h>

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

/*********************************************************
 * NOTE TO STUDENTS: Before you do anything else, please
 * provide your team information in the following struct.
 ********************************************************/
team_t team = {
    /* Team name */
    "team",
    /* First member's full name */
    "Pengda Li",
    /* First member's email address */
    "10225101460@stu.ecnu.edu.cn",
    /* Second member's full name (leave blank if none) */
    "",
    /* Second member's email address (leave blank if none) */
    ""
};

/* single word (4) or double word (8) alignment */
#define ALIGNMENT 8

```

```c
/* rounds up to the nearest multiple of ALIGNMENT */
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)

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

// Basic constants and macros
#define WSIZE 4
#define DSIZE 8
#define CHUNKSIZE (1 << 12)

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

#define PACK(size, alloc) ((size) | (alloc))

#define GET(p) (*(unsigned int *)(p))
#define PUT(p, val) (*(unsigned int *)(p) = (val))

#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(P) (GET(P) & 0x1)

#define HDRP(bp) ((char *)(bp) - WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))

static char * heap_listp;

static void *coalesce(void *bp) {
    size_t prev_alloc = GET_ALLOC(FTRP(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) {
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    } else if (!prev_alloc && 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(NEXT_BLKP(bp))) + GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    return bp;
}

static void *extend_heap(size_t words) {
    char* bp;
    size_t size;
    size = (words & 1) ? (words + 1) * WSIZE : words * WSIZE;
    if ((bp = mem_sbrk(size)) == (void *)-1) {
        return NULL;
    }
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));
    return coalesce(bp);
}

static void *find_fit(size_t asize) {
    for (void *bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
        if (!GET_ALLOC(HDRP(bp)) && GET_SIZE(HDRP(bp)) > asize) {
            return bp;
        }
    }
    return NULL;
}

static void place(void *bp, size_t asize) {
    size_t size = GET_SIZE(HDRP(bp));
    if (size - asize >= 2 * DSIZE) {
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        size = size - asize;
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    } else {
        PUT(HDRP(bp), PACK(size, 1));
        PUT(FTRP(bp), PACK(size, 1));
    }
}
```

```c


/*
 * mm_init - initialize the malloc package.
 */
int mm_init(void) {
    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1) {
        return -1;
    }
    PUT(heap_listp, 0);
    PUT(heap_listp + 1 * WSIZE, PACK(DSIZE, 1));
    PUT(heap_listp + 2 * WSIZE, PACK(DSIZE, 1));
    PUT(heap_listp + 3 * WSIZE, PACK(0, 1));
    heap_listp += 2 * WSIZE;
    if (extend_heap(CHUNKSIZE / WSIZE) == NULL) {
        return -1;
    }
    return 0;
}

/*
 * 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;
    size_t extendsize;
    char *bp;
    if (size == 0) {
        return NULL;
    }
    if (size < DSIZE) {
        asize = 2 * DSIZE;
    } else {
        asize = DSIZE * ((size + DSIZE + DSIZE - 1) / DSIZE);
    }
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize / WSIZE)) == NULL) {
        return NULL;
    }
```

```c
    place(bp, asize);
    return bp;
}

/*
* mm_free - Freeing a block does nothing.
*/
void mm_free(void *ptr) {
    if (ptr == NULL) {
        return;
    }
    size_t size = GET_SIZE(HDRP(ptr));
    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));
    coalesce(ptr);
}

/*
* mm_realloc - Implemented simply in terms of mm_malloc and mm_free
*/
void *mm_realloc(void *ptr, size_t size) {
    void *oldptr = ptr;
    void *newptr;
    size_t copySize;
    newptr = mm_malloc(size);
    if (newptr == NULL) return NULL;
    copySize = GET_SIZE(HDRP(ptr));
    if (size < copySize) copySize = size;
    memcpy(newptr, oldptr, copySize);
    mm_free(oldptr);
    return newptr;
}
```

5.2 显式空闲链表

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

```c
 * NOTE TO STUDENTS: Replace this header comment with your own header
 * comment that gives a high level description of your solution.
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>
#include <string.h>

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

/*********************************************************
* NOTE TO STUDENTS: Before you do anything else, please
* provide your team information in the following struct.
********************************************************/
team_t team = {
    /* Team name */
    "team",
    /* First member's full name */
    "Pengda Li",
    /* First member's email address */
    "10225101460@stu.ecnu.edu.cn",
    /* Second member's full name (leave blank if none) */
    "",
    /* Second member's email address (leave blank if none) */
    ""
};

/* single word (4) or double word (8) alignment */
#define ALIGNMENT 8

/* rounds up to the nearest multiple of ALIGNMENT */
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)

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

// Basic constants and macros
#define WSIZE 4
#define DSIZE 8
#define CHUNKSIZE (1 << 12)

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

```

```c
#define PACK(size, alloc) ((size) | (alloc))

#define GET(p) (*(unsigned int *)(p))
#define PUT(p, val) (*(unsigned int *)(p) = (val))

#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(P) (GET(P) & 0x1)

#define HDRP(bp) ((char *)(bp) - WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))

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

static char * heap_listp;
static char * head;

static void list_insert(void *bp) {
    if (bp == NULL) return;
    if (head == NULL) {
        head = bp;
        return;
    }
    SET_NEXT(bp, (head));
    SET_PREV(head, (bp));
    head = bp;
}

static void list_remove(void *bp) {
    if (bp == NULL || GET_ALLOC(HDRP(bp))) return;
    void* prev = GET_PREV(bp);
    void* next = GET_NEXT(bp);
    SET_PREV(bp, 0);
    SET_NEXT(bp, 0);
    if (prev == NULL && next == NULL) {
        head = NULL;
    } else if (prev == NULL) {
        SET_PREV(next, 0);
        head = next;
```

```
    } else if (next == NULL) {
        SET_NEXT(prev, 0);
    } else {
        SET_NEXT(prev, (next));
        SET_PREV(next, (prev));
    }
}

static void *coalesce(void *bp) {
    size_t prev_alloc = GET_ALLOC(FTRP(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) {
        list_insert(bp);
        return bp;
    } else if (prev_alloc && !next_alloc) {
        list_remove(NEXT_BLKP(bp));
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    } else if (!prev_alloc && next_alloc) {
        list_remove(PREV_BLKP(bp));
        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 {
        list_remove(NEXT_BLKP(bp));
        list_remove(PREV_BLKP(bp));
        size += GET_SIZE(HDRP(NEXT_BLKP(bp))) + GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    list_insert(bp);
    return bp;
}

static void *extend_heap(size_t words) {
    char* bp;
    size_t size;
    size = (words & 1) ? (words + 1) * WSIZE : words * WSIZE;
    if ((bp = mem_sbrk(size)) == (void *)-1) {
        return NULL;
```

```c
    }
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    SET_NEXT(bp, 0);
    SET_PREV(bp, 0);
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));
    return coalesce(bp);
}

static void *find_fit(size_t asize) {
    for (void *bp = head; bp; bp = GET_NEXT(bp)) {
        if (GET_SIZE(HDRP(bp)) >= asize) {
            return bp;
        }
    }
    return NULL;
}

static void place(void *bp, size_t asize) {
    size_t size = GET_SIZE(HDRP(bp));
    list_remove(bp);
    if (size - asize >= 2 * DSIZE) {
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        size = size - asize;
        bp = NEXT_BLKP(bp);
        SET_NEXT(bp, 0);
        SET_PREV(bp, 0);
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
        coalesce(bp);
    } else {
        PUT(HDRP(bp), PACK(size, 1));
        PUT(FTRP(bp), PACK(size, 1));
    }
}


/*
* mm_init - initialize the malloc package.
*/
int mm_init(void) {
    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1) {
        return -1;
```

```
    }
    head = NULL;
    PUT(heap_listp, 0);
    PUT(heap_listp + 1 * WSIZE, PACK(DSIZE, 1));
    PUT(heap_listp + 2 * WSIZE, PACK(DSIZE, 1));
    PUT(heap_listp + 3 * WSIZE, PACK(0, 1));
    heap_listp += 2 * WSIZE;
    if (extend_heap(CHUNKSIZE / WSIZE) == NULL) {
        return -1;
    }
    return 0;
}

/*
* 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;
    size_t extendsize;
    char *bp;
    if (size == 0) {
        return NULL;
    }
    if (size < DSIZE) {
        asize = 2 * DSIZE;
    } else {
        asize = DSIZE * ((size + DSIZE + DSIZE - 1) / DSIZE);
    }
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize / WSIZE)) == NULL) {
        return NULL;
    }
    place(bp, asize);
    return bp;
}

/*
* mm_free - Freeing a block does nothing.
*/
```

```c
void mm_free(void *ptr) {
    if (ptr == NULL) {
        return;
    }
    size_t size = GET_SIZE(HDRP(ptr));
    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));
    SET_NEXT(ptr, 0);
    SET_PREV(ptr, 0);
    coalesce(ptr);
}

/*
* mm_realloc - Implemented simply in terms of mm_malloc and mm_free
*/
void *mm_realloc(void *ptr, size_t size) {
    void *oldptr = ptr;
    void *newptr;
    size_t copySize;
    newptr = mm_malloc(size);
    if (newptr == NULL) return NULL;
    copySize = GET_SIZE(HDRP(ptr));
    if (size < copySize) copySize = size;
    memcpy(newptr, oldptr, copySize);
    mm_free(oldptr);
    return newptr;
}
```