实验报告

第一版：隐式链表

这一版是仿照csapp上的样例写的。用隐式空闲链表来组织空闲块，每个空闲块都同时具有头部和脚部，其中存储块的大小和是否分配的信息。合并采用立即边界合并的方式，尝试了first-fit和best-fit方法，并且分配的时候有尝试分割的操作。并且按照提示设置了堆的序言块和结尾块。

宏定义

```
/* 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)  //根据对齐规则舍入
#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))  //舍入后的大小

#define WSIZE 4  //单字
#define DSIZE 8  //双字
#define CHUNKSIZE (1<<12)  //每次扩展堆的大小时申请的内存

#define MAX(x,y) ((x)>(y)? (x):(y))
//块头部和脚部存储的值，包括块大小和是否分配
#define PACK(size, alloc) ((size) | (alloc))

#define GET(p) (*(unsigned int *)(p))  //得到p地址处的内容
#define PUT(p, val) (*(unsigned int *)(p) = (val))  //向p地址存入值val
#define GET_SIZE(p) (GET(p) & ~0x7)  //从头部或脚部得到块的大小
#define GET_ALLOC(p) (GET(p) & 0x1)  //从头部或脚部得到块是否已分配

//bp指向块的payload的开始
#define HDRP(bp) ((char*)(bp) - WSIZE) //得到块头部的地址
#define FTRP(bp) ((char*)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)//得到块脚部的地址
//得到下一个块的payload起始地址
#define NEXT_BLKP(bp) ((char*)(bp) + GET_SIZE(((char*)(bp) - WSIZE)))
//得到上一个块的地址
#define PREV_BLKP(bp) ((char*)(bp) - GET_SIZE(((char*)(bp) - DSIZE)))
```

其他函数

`int mm_init(void)`函数进行初始化，设置堆的序言块和结尾块并且将其设置为已分配，还要调用extend_heap函数申请空间。

`void* extend_heap(size_t words)`函数通过调用mem_sbrk申请空间并设置其头部和脚部，返回指向这段空间的有效载荷的指针。并且要调用coalesce函数尝试进行合并。

`void* coalesce(void* bp)`函数检查当前块的前后块，将当前块与其中的空闲块进行合并，修改相应的头部和脚部，返回指向合并后的有效载荷的指针

`void* first_fit(size_t asize)`函数从堆头开始，返回搜索到的第一个足够大的空闲块

`void* best_fit(size_t asize)`函数遍历全部的块，找到符合要求的最小的空闲块

`void place(void* bp, size_t asize)`函数会在当前块足够大（比需要的size至少大8字节）时进行分割

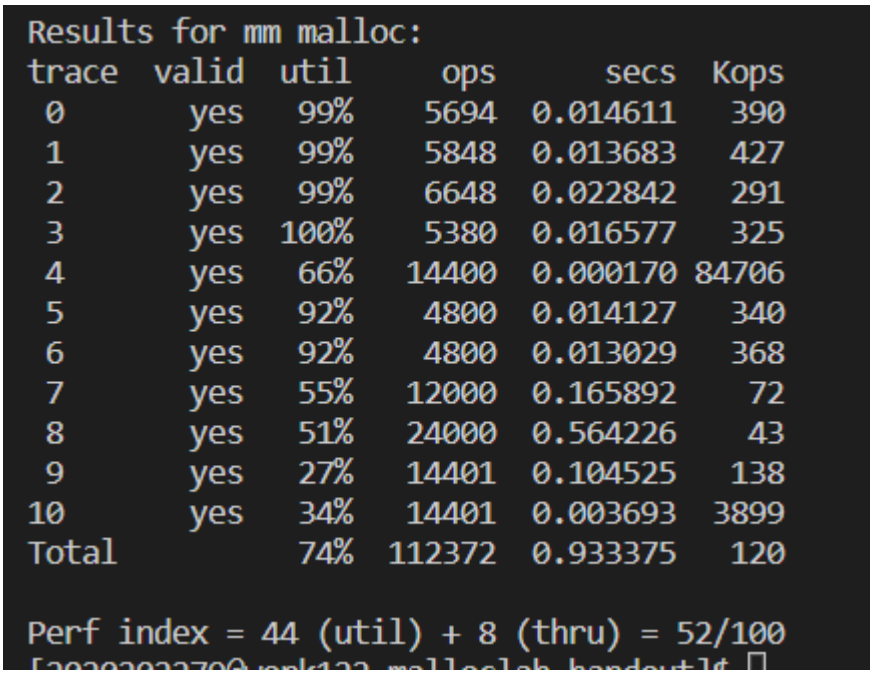
`void *mm_malloc(size_t size)`函数将size按8字节对齐处理，然后调用first-fit或best-bit算法找可用的空闲块，若找不到则调用extend_heap函数申请空间

`void mm_free(void *ptr)`函数释放已分配块，修改其头部和脚部的标记为未分配并尝试合并

`void *mm_realloc(void *ptr, size_t size)`函数根据size大小重新malloc分配空间然后memcpy复制旧内容

由于书上已经提供大部分代码，不再赘述

得分情况



第二版：隐式链表+next-fit 85

这一版中仍然沿用隐式链表，只改变为next-fit算法

宏定义部分以及大部分函数都与之前完全相同，只有如下少量改动：

- 声明一个全局变量`static char* pre_list;`用于指示上一次查找到的位置，每次查找匹配块时从它开始找起。
- `void* coalesce(void* bp)`函数、`void place(void* bp, size_t asize)`函数、`int mm_init(void)`函数中需要对`pre_list`进行相应处理
- 另外，查找匹配块的`void* find_fit(size_t asize)`函数设置如下：

```
void* find_fit(size_t asize){
    char* bp = pre_list;
    size_t isalloc;
    size_t size;
    while (GET_SIZE(HDRP(NEXT_BLKP(bp))) > 0) {
        bp = NEXT_BLKP(bp);
        isalloc = GET_ALLOC(HDRP(bp));
        if (isalloc) continue;
        size = GET_SIZE(HDRP(bp));
        if (size < asize) continue;
        return bp;
    }
    bp = heap_list;
    while (bp != pre_list) {
        bp = NEXT_BLKP(bp);
        isalloc = GET_ALLOC(HDRP(bp));
        if (isalloc) continue;
        size = GET_SIZE(HDRP(bp));
        if (size < asize) continue;
        return bp;
    }
    return NULL;
}
```

得分情况

第一次尝试后，得分83。尝试进行了一个优化，缩小初始化堆时用extend_heap申请的空间的大小为1<<6，提升到85分

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%    5694  0.003423  1663
 1       yes   91%    5848  0.002182  2680
 2       yes   96%    6648  0.006893   964
 3       yes   97%    5380  0.006854   785
 4       yes   99%   14400  0.000165 87167
 5       yes   92%    4800  0.008039   597
 6       yes   91%    4800  0.007315   656
 7       yes   55%   12000  0.015723   763
 8       yes   51%   24000  0.014157  1695
 9       yes   27%   14401  0.102116   141
10       yes   30%   14401  0.004348  3312
Total          74%  112372  0.171215   656

Perf index = 45 (util) + 40 (thru) = 85/100
```

第三版：显式链表

显示链表的各种函数基本逻辑与之前大概相同，只不过使用链表串联了所有的空闲块，使我们在搜索可分配块时可以跳过已分配的块，直接搜索空闲块，节省时间，但是它需要每个空闲块中有额外的位置存放指针。

宏定义

宏定义中在原来的基础上，增加了以下几条：

```
#define GET_PTR(p) (*(unsigned long *)(p))
#define PUT_PTR(p, ptr)    (*(unsigned long *)(p) = (unsigned long)(ptr))

//得到指向前一个空闲块的指针
#define GET_PREV(bp) ((char*)(*(unsigned long *)(bp)))
//得到指向后一个空闲块的指针
#define GET_NEXT(bp) ((char*)(*(unsigned long *)(bp + DSIZE)))
//设置指向前一个空闲块的指针
#define SET_PREV(p, ptr) (PUT_PTR((char*)(p),ptr))
//设置指向后一个空闲块的指针
#define SET_NEXT(p, ptr) (PUT_PTR((char*)(p) + DSIZE,ptr))
```

全局变量

由于是双向链表，所以有`static char* head`指示空闲链表的开始，`static char* tail`指示空闲链表的结尾

其他函数

函数的实现与之前差异不算太大，只是要增加从链表中插入和删除的操作。

`void* extend_heap(size_t words)`函数中，除了设置新分配块的头部和脚部，还要初始化其指向链表中前后块的指针。

`void* coalesce(void* bp)`函数在之前的基础上，要把可合并块从空闲链表中移除，把合并后的新块插入空闲链表中。

`void* find_fit(size_t asize)`函数采用first-fit算法，与第一版相同

`void place(void* bp, size_t asize)`函数中，要将被分割块从空闲链表中移除，并且在成功分割后将分割得到的空闲块进行初始化，然后插入到空闲链表中

`int mm_init(void)`函数思路与之前相同，只是初始化时要留出存放指针的空间。

`void *mm_malloc(size_t size)`函数与之前一样

`void mm_free(void *ptr)`函数要注意将free掉的块插入到空闲链表中的合适位置

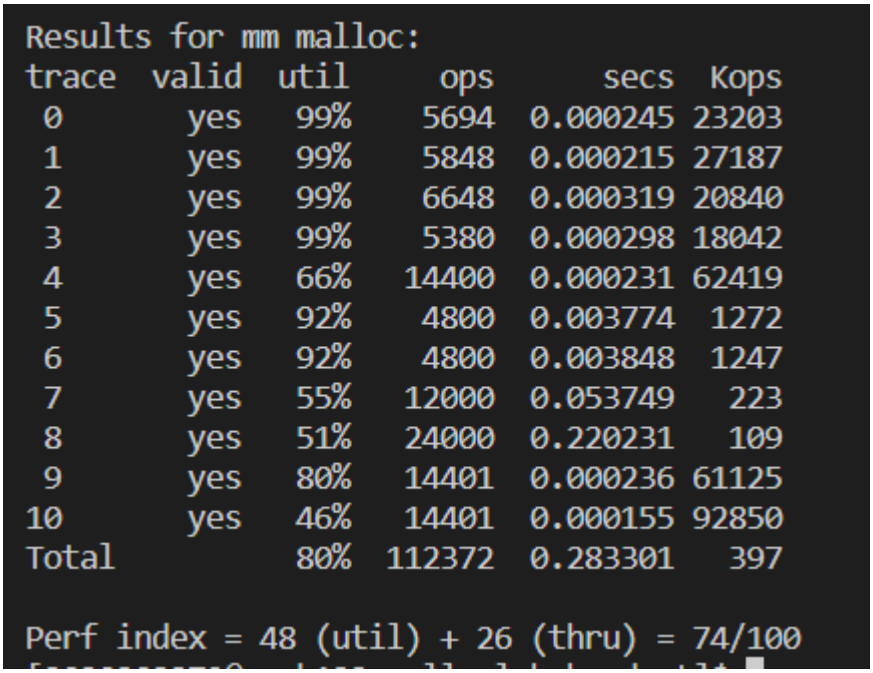
`void *mm_realloc(void *ptr, size_t size)`函数则进行了如下的分情况讨论：

```
void *mm_realloc(void *ptr, size_t size)
{
    //两种特殊情况
    if(size == 0){
        mm_free(ptr);
        return NULL;
    }
    if(ptr == NULL){
        return mm_malloc(size);
    }

    size_t extendsize;

    //按照规则对齐
    if(size <= 2*DSIZE)
        size = 3*DSIZE;
    else
        size = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);

    void* next_ptr;
    void* newptr;
    void* oldptr = ptr;
    char* p;
    size_t copySize = GET_SIZE(HDRP(ptr));

    if(copySize == size) return ptr;  //大小相等，直接返回
    else if(copySize > size){ //如果新块的大小比原来的小
        if(copySize >= size + (3 * DSIZE)){//足够分割出新的空闲块
            PUT(HDRP(ptr), PACK(size, 1));
            PUT(FTRP(ptr), PACK(size, 1));
            next_ptr = NEXT_BLKP(ptr);
            PUT(HDRP(next_ptr), PACK(copySize-size, 0));
            PUT(FTRP(next_ptr), PACK(copySize-size, 0));

            //新的空闲块插入空闲列表
            for(p = GET_NEXT(head); ; p = GET_NEXT(p)){
                if(next_ptr < (void*)p){
                    void* prev_list = GET_PREV(p);
                    void* next_list = p;
                    SET_NEXT(next_ptr, prev_list);
                    SET_PREV(next_ptr, next_list);
                    SET_NEXT(prev_list, next_ptr);
                    SET_PREV(p, next_ptr);
                    break;
                }
            }

        }
        return ptr;
    }else{ //如果新块的大小更大
```

```
        next_ptr = NEXT_BLKP(ptr);
        size_t newsize = copySize + GET_SIZE(HDRP(next_ptr));
        if(!GET_ALLOC(HDRP(next_ptr)) && newsize >= size){
        //如果后面的块未分配且够大，则不需要移动
                void* prev_list = GET_PREV(next_ptr);
                void* next_list = GET_NEXT(next_ptr);
                if(newsize - size >= DSIZE * 3){
                    PUT(HDRP(ptr), PACK(size, 1));
                    PUT(FTRP(ptr), PACK(size, 1));
                    next_ptr = NEXT_BLKP(ptr);
                    PUT(HDRP(next_ptr), PACK(newsize-size, 0));
                    PUT(FTRP(next_ptr), PACK(newsize-size, 0));

                    SET_NEXT(next_ptr, next_list);
                    SET_PREV(next_ptr, prev_list);
                    SET_NEXT(prev_list, next_ptr);
                    SET_PREV(next_list, next_ptr);
                }else{
                    PUT(HDRP(ptr), PACK(newsize, 1));
                    PUT(FTRP(ptr), PACK(newsize, 1));
                    SET_NEXT(prev_list, next_list);
                    SET_PREV(next_list, prev_list);
                }
                return ptr;
        }else{//否则则要另找一块空间
            newptr = find_fit(size);
            if (newptr == NULL){
                extendsize = MAX(size, CHUNKSIZE);
                if((newptr = extend_heap(extendsize/WSIZE)) == NULL){
                    return NULL;
                }
            }
            place(newptr, size);
            memcpy(newptr, oldptr, copySize - 2 * WSIZE);
            mm_free(oldptr);
            return newptr;
        }
    }
    return ptr;
}
```

得分情况

得分74



第四版：分离适配算法

这一版虽然写了，但始终segmentation fault没有调试通过，所以综合以上情况，选择第二版作为最终提交。