mallocLab实验报告

代码参考

本次实验我一开始是基于书上的代码完成的，所以有很多宏定义直接来源于书上。~~(说不能抄网上的代码没说不能看书吧)~~

隐式列表

为了完成困难的mallocLab，我听取了实验指导的建议，认真地理解~~(复制)~~了书上malloc实现的每一行。书上完成了一个简单的隐式列表，并留了find和place函数让我们自己完成。

find策略

对于find函数，我使用了三种查找策略，分别是first-fit，next-fit和best-fit。具体得分如下：

```
Team Name:Genshin start
Member 1 :MeiYihang:2022201459@ruc.edu.cn
Using default tracefiles in ../traces/
Perf index = 44 (util) + 16 (thru) = 60/100
```

```
Team Name:Genshin start
Member 1 :MeiYihang:2022201459@ruc.edu.cn
Using default tracefiles in ../traces/
Perf index = 43 (util) + 40 (thru) = 83/100
```

```
Team Name:Genshin start
Member 1 :MeiYihang:2022201459@ruc.edu.cn
Using default tracefiles in ../traces/
Perf index = 45 (util) + 10 (thru) = 55/100
```

可以看出3者虽然对util上也有差异，但是thru上的差异更为明显，木桶效应显著，因此，我选择了next-fit查找策略，具体函数如下：

```
static void *find_fit(size_t asize)
{
    void *bp;

    for (bp = last_search_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
        if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
```

```
            last_search_listp = bp;
            return bp;
        }
    }

    for (bp = heap_listp; bp != last_search_listp;bp = NEXT_BLKP(bp)) {
        if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
            last_search_listp = bp;
            return bp;
        }
    }
    return NULL; /* No fit */
}
```

其中last_search_listp指向上次查找的空闲块，heap_listp指向首个内存块。值得一提的是，last_search_listp不仅要在find_fit()函数中进行更新，同时也要在coalesce()这个合并空闲块的函数中更新，避免某次合并空闲块后last_search_listp指向了空闲块内部而非空闲块头部。

```
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) {            /* Case 1 */
        return bp;
    }

    else if (prev_alloc && !next_alloc) {      /* Case 2 */
        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) {      /* Case 3 */
        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 {                                     /* Case 4 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
        GET_SIZE(FTRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }

    last_search_listp =bp;          //更新last_search_listp
    return bp;
}
```

realloc重实现

我原本使用的是已有代码，但是后来发现一个可改进的点。就是当ptr指向的内存块的下一个内存块是空闲的，且ptr指向的内存块和其下一个内存块的空间能容下realloc的size时，可以直接分配内存。而不用像之前那样malloc，mmcopy再free。

代码如下：

```
void *mm_realloc(void *ptr, size_t size)
{
    if (ptr == NULL) {
        return mm_malloc(size);
    }
    if (size == 0) {
        mm_free(ptr);
        return NULL;
    }

    size_t old_size = GET_SIZE(HDRP(ptr));
    size_t new_size = MAX(ALIGN(size) + DSIZE, 2*DSIZE);
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(ptr)));
    size_t next_size = GET_SIZE(HDRP(NEXT_BLKP(ptr)));

    if (!next_alloc && (old_size + next_size >= new_size)) {
        size_t combined_size = old_size + next_size;
        if (combined_size - new_size >= 2*DSIZE) {
            PUT(HDRP(ptr), PACK(new_size, 1));
            PUT(FTRP(ptr), PACK(new_size, 1));
            void *next_bp = NEXT_BLKP(ptr);
            PUT(HDRP(next_bp), PACK(combined_size - new_size, 0));
            PUT(FTRP(next_bp), PACK(combined_size - new_size, 0));
            coalesce(next_bp);
        } else {
            PUT(HDRP(ptr), PACK(combined_size, 1));
            PUT(FTRP(ptr), PACK(combined_size, 1));
        }
        return ptr;
    }

    void *new_ptr = mm_malloc(size);
    if (new_ptr == NULL) {
        return NULL;
    }
    size_t copy_size = MIN(old_size - DSIZE, size);
    memcpy(new_ptr, ptr, copy_size);
    mm_free(ptr);
    return new_ptr;
}
```

经过这个改进，空间利用率进一步上升，我的分数来到了85分。

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   91%    5694  0.001465  3887
 1       yes   91%    5848  0.000949  6164
 2       yes   95%    6648  0.002837  2343
 3       yes   97%    5380  0.003203  1680
 4       yes   66%   14400  0.000067216541
 5       yes   90%    4800  0.003251  1477
 6       yes   88%    4800  0.002980  1611
 7       yes   55%   12000  0.013648   879
 8       yes   51%   24000  0.006378  3763
 9       yes   42%   14401  0.000182 78953
10       yes   57%   14401  0.000089161265
Total          75%  112372  0.035050  3206

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

本来还想试试Segregated Fit的，~~不过春假来子~~，不过没有在预期的时间内完成，所以就这样吧。