malloclab实验报告

罗思佳202121679

一、实验目的

模仿实现一个动态内存分配器（即自己实现库函数 malloc 、 free 和 realloc ），需要修改下发文件中的 mm.c 的相关代码，最终完成一个正确且高效的分配器。

需要实现的函数：

```
int mm_init(void);
void * mm_malloc(size_t size);
void mm_free(void * ptr);
void * mm_realloc(void * ptr, size_t size);
```

每个函数的说明（摘自实验指导）

- `mm_init`：用来评估你的分配器的进程预先进行初始化，例如分配初始堆空间。如果初始化过程出现问题，返回值应当设置为-1，否则请设置为0。
- `mm_malloc`：类似`libc`中的`malloc`，请返回一个指向大小至少为`size`的已分配块的指针，且该指针应当是8字节对齐的。也就是说$ret \ \& \ 0\text{x}7 = 0$。
- `mm_free`：释放由`mm_malloc`或者`mm_realloc`分配的空间。
- `mm_realloc`
  - 如果`ptr==NULL`，等价于`mm_malloc(size)`
  - 如果`size==0`，等价于`mm_free(ptr)`
  - 否则，返回指向新块的指针，新块的前$\min(size_{new}, size_{old})$字节应当与旧的块保持一致，剩下的内容保持未初始化状态。注意，新块和旧块的地址可以一样，这取决于你的实现。

模拟内存系统：

```
void * mem_sbrk(int incr); // 使堆增加incr字节，返回新分配堆区域的第一个字节地址
void * mem_heap_lo(void); // 返回堆的第一个字节地址
void * mem_heap_hi(void); // 返回堆的最后字节地址
size_t mem_heapsize(void); // 返回当前堆的大小
size_t mem_pagesize(void); // 返回系统的Page大小，Linux上为4096
```

堆一致性检查器：帮助调试和发现问题

- 空闲链表中的块都标记为空闲了吗?
- 是否有连续的空闲块未被合并?
- 每一个空闲块都在空闲链表中吗?
- 空闲链表的指针指向的都是有效的空闲块吗?
- 有相交的或者大小不对的分配块吗?
- 返回的指针是有效的吗?
- ……

二、前期准备

1.理解课本上的代码

课本上设计了一个基于隐式空闲链表的分配器，先介绍了堆和链表的形式，什么是序言块和结尾块，展示了需要经常用到的一些基本常数和宏，然后依次设计了`mm_init`函数、`extend_heap`函数、`mm_free`函数、`coalesce`函数、`mm_malloc`函数，练习题中给出了`find_fit`函数和`place`函数，使用的是首次适配。

2.阅读CMU的Slides

两份Slides主要介绍了内存分配的几种策略、可能遇到的困难（Garbled bytes、Overlapping payloads、Segmentation fault）、如何使用GDB进行调试、常用的GDB命令等等。

GDB调试可以在Makefile文件中将`CC = gcc`改为`CC = gcc -g`以增加调试信息。

```
> gdb mdriver
(gdb) b mm_init
(gdb) b mm_malloc
(gdb) b mm_free
(gdb) run
```

三、实验过程

空闲块组织策略：使用**隐式空闲链表**

放置策略：先尝试首次适配，再尝试下一次适配，再尝试最佳适配。

分割策略：将请求块放置在空闲块的起始位置

合并策略：立即合并

(一) 代码设计

1.常数、宏和自定义函数

参考了课本上的对应部分，定义了一些大小常数、访问和遍历空闲链表的宏等。

```
/* 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 WSIZE 4 //字
#define DSIZE 8 //双字
#define CHUNKSIZE (1<<12) //每次增大堆的大小

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

/* Pack a size and allocated bit into a word */
#define PACK(size, alloc) ((size)|(alloc))
/* Read and write a word at address p */
#define GET(p)       (*(int *)(p))
#define PUT(p, val)  (*(int *)(p) = (val))
```

```
/* Read the size and allocated fields from address p */
#define GET_SIZE(p)  (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)
/* Given block ptr bp, compute address of its header and footer */
#define HDRP(bp)       ((char *)(bp) - WSIZE)
#define FTRP(bp)       ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp)  ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp)  ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))

static void* heap_listp;  /* pointer to first block */

static void *extend_heap(size_t size); //扩大堆
static void place(void *bp, size_t asize); //放置并分割
static void *find_fit(size_t asize); //寻找空闲块
static void *coalesceFreeBlock(void *bp); //合并

static void* next_fitptr;     //下一次匹配指向的指针
static int mm_check(int verbose, const char* func);//检查
```

2.mm_init

作用是初始化堆的 序言块 和 结尾块，并调用 extend_heap() 函数扩展堆的大小。函数首先调用 mem_sbrk() 申请一个初始的堆空间，大小为 4 * WSIZE 字节。接着，设置堆的**序言块**和**结尾块**，将 prologue header 和 prologue footer 设为同样的值；将 epilogue header 设为 0，表示堆的结尾。然后，移动 heap_listp ，指向可用空间的起始位置。next_fitptr 初始化为 heap_listp（next fit中加这一句）。最后调用 extend_heap() 函数来扩展堆的大小，分配 CHUNKSIZE 字节的内存。

```
/*
 * mm_init - initialize the malloc package.
 */
int mm_init(void)
{
    if((heap_listp = mem_sbrk(4*WSIZE)) == (void*)-1)//分配失败
    {
        printf("ERROR: mem_sbrk failed in mm_init\n");
        return -1;
    }

    PUT(heap_listp, 0);
    PUT(heap_listp+WSIZE, PACK(DSIZE, 1));  // prologue header
    PUT(heap_listp+DSIZE, PACK(DSIZE, 1));  // prologue footer
    PUT(heap_listp+3*WSIZE, PACK(0,1));    // epilogue header

    heap_listp += DSIZE;
    next_fitptr = heap_listp;//初始化
    if (extend_heap(CHUNKSIZE) == NULL)
        return -1;


    //if(mm_check(VERBOSE, __func__) == 0)
    //    printf("==================================\n");
    return 0;
}
```

3.extend_heap

用于分配一段指定大小的内存，并将其扩展到当前堆的末尾。函数首先将要扩展的内存大小 size 对齐为 ALIGN 的倍数，然后调用 mem_sbrk() 函数来分配 size 大小的内存，如果分配失败，则返回 NULL。接着，函数设置新分配块的头部和脚部，将它们标记为空闲块，并设置新的结尾块的头部，将它标记为已分配块。最后，调用 coalesceFreeBlock() 函数合并相邻的空闲块，返回新分配块的地址。

ps：课本上传的函数参数是字words，然后再乘以WSIZE，个人觉得略微麻烦，所以改为字节数size。

```
static void *extend_heap(size_t size)
{
    size_t asize;
    void *bp;

    asize = ALIGN(size);
    if ((long)(bp = mem_sbrk(asize)) == -1)
        return NULL;

    PUT(HDRP(bp), PACK(size,0)); //空闲块的头部
    PUT(FTRP(bp), PACK(size,0)); //空闲块的脚部
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0,1));//新的结尾块

    return coalesceFreeBlock(bp);

}
```

4.mm_malloc

首先将要分配的内存大小 size 转换成适当的块大小 asize，然后调用 find_fit() 函数查找合适的空闲块，如果找到了，则调用 place() 函数进行放置和分割。如果没找到合适的空闲块，则调用 extend_heap() 函数扩展堆的大小，并调用 place() 函数将新分配块划分成两部分。最后，函数返回新分配块的地址。

```
/*
 * 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 addSize;
    char* bp = NULL;

    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);

        //if(mm_check(VERBOSE, __func__) == 0)
        //    printf("==================================\n");
```

```
        return bp;
    }

    //没找到合适的块
    addSize =  MAX(asize, CHUNKSIZE);
    if((bp = extend_heap(addSize))==NULL)
        return NULL;
    place(bp, asize);


    //if(mm_check(VERBOSE, __func__) == 0)
    //    printf("==================================\n");
    return bp;

}
```

5.mm_free

该函数释放所请求的块bp，然后使用边界标记合并将其与邻接的空闲块合并。

```
/*
 * mm_free - Freeing a block does nothing.
 */
void mm_free(void *ptr)
{
    size_t size = GET_SIZE(HDRP(ptr));
    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));
    coalesceFreeBlock(ptr);//合并


    //if(mm_check(VERBOSE, __func__) == 0)
    //    printf("==================================\n");
}
```

6.mm_realloc

首先检查传递给它的指针是否为NULL，如果是，则返回调用mm_malloc函数分配新的内存。如果size为0，则调用mm_free释放传递的指针并返回NULL。然后对size进行对齐，**检查旧块的大小是否小于新大小**。如果是，则调用find_fit函数寻找适合的空闲块，如果找不到，则调用extend_heap函数扩展堆，以获得更多空间。然后它使用place函数将块放置在新块中，并使用memcpy函数将旧数据复制到新块中，最后释放旧块。如果新大小小于旧大小，则**检查旧块是否需要分割**，如果可以，则使用place函数将其分配到新块中。函数返回分配的新块或NULL。

```
/*
 * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
 */
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 newSize = ALIGN(size + 2*WSIZE);//对齐
    size_t oldSize = GET_SIZE(HDRP(ptr));
    void* new_ptr = NULL;
    if(newSize > oldSize)
    {
        new_ptr = find_fit(newSize);//寻找适合的空闲块
        if(!new_ptr)//没有合适的空闲块
        {
            size_t addSize = MAX(newSize, CHUNKSIZE);
            new_ptr = extend_heap(addSize);
            if(new_ptr == NULL) return NULL;
        }
        place(new_ptr, newSize);
        memcpy(new_ptr, ptr, oldSize-2*WSIZE);//拷贝过去
        mm_free(ptr);//释放原来的块

        //if(mm_check(VERBOSE, __func__) == 0)
        //    printf("==================================\n");
        return new_ptr;

    }
    else//将之前的size缩小
    {
        if(oldSize - newSize < 2*DSIZE)//小于最小块，不分割
        {
            return ptr;
        }
        else
        {
            place(ptr, newSize);
            //if(mm_check(VERBOSE, __func__) == 0)
            //    printf("==================================\n");
            return ptr;
        }
    }
    return NULL;

}
```

7. place

函数`place`用于将一个空闲块分配出去，并在需要的情况下将其分割成一个已分配块和一个剩余的空闲块。

首先，获取原来的空闲块的大小，然后计算剩余的块的大小。如果剩余的块小于最小块的大小，那么不分割，将整个块标记为已分配。否则，将要分配的块标记为已分配，将剩余的块标记为未分配，并将指针移动到剩余块的头部。

```
static void place(void *bp, size_t asize)//分配并分割
```

```
{
    size_t old_size = GET_SIZE(HDRP(bp));//原来的大小
    size_t unalloc_size = old_size - asize;//分配后剩余的大小

    if(unalloc_size < 2*DSIZE)//如果剩余的块小于最小块，不分割
    {
        PUT(HDRP(bp), PACK(old_size, 1));
        PUT(FTRP(bp), PACK(old_size, 1));
    }
    else//剩余的块大于最小块，分割
    {
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        bp = NEXT_BLKP(bp);//指到剩余的空闲块
        PUT(HDRP(bp), PACK(unalloc_size, 0));
        PUT(FTRP(bp), PACK(unalloc_size, 0));
    }
}
```

8.find_fit

首次适配

从heap_listp开始遍历，直到找到第一个大小合适的空闲块，返回该块的地址。

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

下一次适配

在开始搜索的位置上进行连续搜索，直到找到一个足够大的未分配空闲块。如果从开始搜索的位置到堆顶都没有找到，则从堆底开始重新搜索。如果找到了符合要求的空闲块，则返回该空闲块的指针，否则返回NULL。

```
static void *find_fit(size_t asize)// next fit
{
    char *hi_ptr = (char*)mem_heap_hi()+1;
    next_fitptr = NEXT_BLKP(next_fitptr);    //此次开始搜索的位置
    void *start_ptr = next_fitptr;

    while(next_fitptr != hi_ptr)//找start_ptr和hi_ptr之间的空闲块
    {
        if (!GET_ALLOC(HDRP(next_fitptr)) && (GET_SIZE(HDRP(next_fitptr)) >=
asize))
            return next_fitptr;
        next_fitptr = NEXT_BLKP(next_fitptr);
```

```
    }
    //如果没找到，就从头开始找
    next_fitptr = NEXT_BLKP(heap_listp);
    while(next_fitptr != start_ptr)
    {
        if (!GET_ALLOC(HDRP(next_fitptr)) && (GET_SIZE(HDRP(next_fitptr)) >=
asize))
            return next_fitptr;
        next_fitptr = NEXT_BLKP(next_fitptr);
    }

    return NULL;
}
```

最佳适配

遍历空闲链表，找到空闲链表中大小合适并且最小的空闲块。

```
static void *find_fit(size_t asize){
    size_t minSize, size;
    void *ptr;
    void *best_ptr = NULL;
    bool flag = true;
    for(ptr = heap_listp; (size = GET_SIZE(HDRP(ptr))) > 0; ptr =
NEXT_BLKP(ptr)){
        if((!GET_ALLOC(HDRP(ptr))) && (size >= asize)){
            if(flag){//先找第一个合适的，作为基准
                best_ptr = ptr;
                minSize = size;
                flag = false;
                continue;
            }
            if(size < minSize){//找到了更小的合适的
                best_ptr = ptr;
                minSize = size;
            }
        }
    }
    return best_ptr;
}
```

9.coalesceFreeBlock

合并空闲块

首先通过调用`GET_ALLOC`来获取前一个和后一个块的分配情况，然后获取当前块的大小。如果前一个块是空闲的，就将`bp`指向前一个块，将当前块和前一个块合并成一个块，并更新该块的头和尾部。如果后一个块是空闲的，就将当前块和后一个块合并成一个块，并更新该块的头和尾部。最后，函数返回指向合并后块的指针`bp`。在**下一次适配策略**中，如果前一个块是空闲的，并且`next_fitptr`指向该块，那么在合并后需要将`next_fitptr`更新为前一个块的指针，以保证下一次空闲块分配的时候从正确的位置开始搜索。

```
static void *coalesceFreeBlock(void *bp)//合并空闲块
{
```

```
    int next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    int prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    if(!prev_alloc)//前一个是空闲的
    {
        if(bp == next_fitptr)//如果next_fitptr与bp相同，也要更新next_ptr，防止被合并掉
        {
            next_fitptr = PREV_BLKP(bp);
        }
        bp = PREV_BLKP(bp);
        size += GET_SIZE(HDRP(bp));

        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }

    if(!next_alloc)//后一个是空闲的
    {
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }

    return bp;
}
```

(二) 堆一致性检查器

编写了一个堆一致性检查器，以在程序出错时打印相关信息

主要检查空闲块是否都合并了、头部和脚部是否相同、payload是否对齐、块大小是否对齐

```
static int mm_check(int verbose, const char* func)
{
    if(!verbose) return 1;
    void *p;
    for (p = heap_listp; GET_SIZE(HDRP(p)) > 0; p = NEXT_BLKP(p))
    {
        if(GET_ALLOC(HDRP(p)) == 0 && GET_ALLOC(HDRP(NEXT_BLKP(p))) == 0)//空闲块
未合并
        {
            printf("ERROR:%p and the next free block are not coalesced.\n", p);
            printf("hsize = %d, fsize = %d\n", GET_SIZE(HDRP(p)),
GET_SIZE(FTRP(p)));
            printf("halloc = %d, falloc = %d\n", GET_ALLOC(HDRP(p)),
GET_ALLOC(FTRP(p)));
            printf("next_head_alloc = %d, next_footer_alloc = %d\n",
GET_ALLOC(HDRP(NEXT_BLKP(p))), GET_ALLOC(FTRP(NEXT_BLKP(p))));
            return 0;
        }
        if (GET(HDRP(p)) != GET(FTRP(p)))//头部和脚部不对应
        {
            printf("ERROR: %p's header and footer are not matched.\n", p);
            printf("hsize = %d, fsize = %d\n", GET_SIZE(HDRP(p)),
GET_SIZE(FTRP(p)));
```

```
            printf("halloc = %d, falloc = %d\n", GET_ALLOC(HDRP(p)), GET_ALLOC(FTRP(p)));
            return 0;
        }
        if ((int)p % ALIGNMENT != 0)//payload没对齐
        {
            printf("ERROR: %p's Payload area is not aligned.\n", p);
            return 0;
        }
        if (GET_SIZE(HDRP(p)) % ALIGNMENT != 0)//大小没对齐
        {
            printf("ERROR: %p payload size is not doubleword aligned.\n", p);
            return 0;
        }

    }
    return 1;
}
```

例如在写`mm_free`时，因为粗心忘记加上`coalesceFreeBlock`函数，把当前释放掉的块的前后空闲块合并了，输出调试信息：

```
Testing mm malloc
Reading tracefile: ./traces/short1-bal.rep
Checking mm_malloc for correctness, ERROR:0x7f383bd32858 and the next free block are not coalesce.
hsize = 4080, fsize = 4080
halloc = 0, falloc = 0
next_head_alloc = 0, next_footer_alloc = 0
==================================
ERROR:0x7f383bd32020 and the next free block are not coalesce.
hsize = 2048, fsize = 2048
halloc = 0, falloc = 0
next_head_alloc = 0, next_footer_alloc = 0
==================================
ERROR:0x7f383bd32020 and the next free block are not coalesce.
hsize = 2048, fsize = 2048
halloc = 0, falloc = 0
next_head_alloc = 0, next_footer_alloc = 0
==================================
ERROR:0x7f383bd32020 and the next free block are not coalesce.
hsize = 2048, fsize = 2048
halloc = 0, falloc = 0
next_head_alloc = 0, next_footer_alloc = 0
==================================
ERROR:0x7f383bd32020 and the next free block are not coalesce.
hsize = 2048, fsize = 2048
halloc = 0, falloc = 0
```

表示一直有未合并的空闲块，再去检查是不是合并函数和释放函数的地方出了问题。

四、实验结果

首次适配只得了52分

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.014332   397
 1       yes   99%    5848  0.013312   439
 2       yes   99%    6648  0.022111   301
 3       yes  100%    5380  0.016096   334
 4       yes   66%   14400  0.000166 86852
 5       yes   92%    4800  0.009746   492
 6       yes   92%    4800  0.009438   509
 7       yes   55%   12000  0.165805    72
 8       yes   51%   24000  0.561742    43
 9       yes   27%   14401  0.103540   139
10       yes   30%   14401  0.002336  6165
Total          74%  112372  0.918624   122

Perf index = 44 (util) + 8 (thru) = 52/100
```

下一次适配，**83分**，主要是吞吐量明显好于首次适配

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   90%    5694  0.003317  1716
 1       yes   91%    5848  0.002155  2714
 2       yes   95%    6648  0.006310  1054
 3       yes   96%    5380  0.006143   876
 4       yes   66%   14400  0.000185 77796
 5       yes   91%    4800  0.005276   910
 6       yes   89%    4800  0.005071   947
 7       yes   55%   12000  0.016959   708
 8       yes   51%   24000  0.015178  1581
 9       yes   27%   14401  0.102720   140
10       yes   45%   14401  0.001982  7267
Total          72%  112372  0.165296   680

Perf index = 43 (util) + 40 (thru) = 83/100
```

最佳适配，53分，只比首次适配高一点点，与下一次适配相比主要输在吞吐率上，因为每次都要对堆进行彻底的搜索。

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.015858   359
 1       yes   99%    5848  0.015041   389
 2       yes   99%    6648  0.023113   288
 3       yes  100%    5380  0.017117   314
 4       yes   66%   14400  0.000184 78133
 5       yes   96%    4800  0.021824   220
 6       yes   95%    4800  0.020820   231
 7       yes   55%   12000  0.165693    72
 8       yes   51%   24000  0.562657    43
 9       yes   31%   14401  0.103896   139
10       yes   30%   14401  0.002919  4934
Total          75%  112372  0.949122   118

Perf index = 45 (util) + 8 (thru) = 53/100
```

因此最终选择采用下一次适配的策略。

五、遇到的困难

1.在`extend_heap`函数中，扩展完堆大小之后忘记合并，因为如果堆的最后一个是空闲块的话，扩展出来的新空闲块与之是邻接的，因此需要进行一次合并操作。

2.使用next fit策略时，在`mm_init`函数中需要对指针`next_fitptr`进行初始化，否则之后在`find_fit`函数中会因为野指针而出错。

```
next_fitptr = heap_listp;//初始化
```

3.在`mm_realloc`函数中，因为参数size不包括头部和脚部，所以需要加上2*WSIZE再对齐，

```
size_t newSize = ALIGN(size + 2*WSIZE);//对齐
```

同样之后使用`memcpy`拷贝过去时，用的是`oldSize-2*WSIZE`而不是`oldSize`，即减去头部脚部的大小。

```
memcpy(new_ptr, ptr, oldSize-2*WSIZE);//拷贝过去
```

4.使用next fit策略时，因为额外增加了一个全局的`next_fitptr`指针，在`coalesceFreeBlock`函数中，需要考虑`next_fitptr`被合并掉的可能，因此需要增加一段更新`next_fitptr`的代码

```
if(bp == next_fitptr)//如果next_fitptr与bp相同，也要更新next_ptr，防止被合并掉
{
  next_fitptr = PREV_BLKP(bp);
}
```

六、实验心得

本次实验让我更加清楚了堆的结构、动态内存分配的组织、放置和合并策略，

如何用代码来实现这些策略，以及写代码时的一些细碎小细节，比如对齐问题、指针初始化等，

还有如何使用一致性检查器和GDB进行调试，这些都是在以后的实验中不可或缺的工具和经验。