实验准备

- 搭建环境参考： https://zhuanlan.zhihu.com/p/593904447
- 请认真阅读教材第9章。
- 请确保自己非常熟悉指针，在官方对这个实验的介绍中有这么一句话

> When students finish this one, they really understand pointers!

实验要求

我们需要修改文件`mm.c`, 以实现四个函数:

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

官方提供了几个现成的函数:

- `void *mem_sbrk(int incr)` :

  Expands the heap by incr bytes, where incr is a positive
  non-zero integer and returns a generic pointer to the first byte of the newly allocated heap area. The
  semantics are identical to the Unix sbrk function, except that mem_sbrk accepts only a positive
  non-zero integer argument.
- `void *mem_heap lo(void)` : Returns a generic pointer to the first byte in the heap.
- `void *mem_heap hi(void)` : Returns a generic pointer to the last byte in the heap.
- `size t mem_heapsize(void)` : Returns the current size of the heap in bytes.
- `size t mem_pagesize(void)` : Returns the system's page size in bytes (4K on Linux systems).

实验开始

基础模型

我们先把教材上提供的一些宏抄过来, 我均附上了详细的注释:

```
#define WSIZE 4 //单字大小
#define DSIZE 8 //双字大小
#define CHUNKSIZE (1<<12) //扩展堆时的默认大小(bytes)，以及初始时的free block的大小。
#define PACK(size,alloc) ((size)|(alloc)) //计算header和footer的值，教材p848.
#define GET(p) (*(unsigned int *)(p)) //把p强制转换为无符号int指针，然后拿到它所在位置的一个word。
#define PUT(p,val) (*(unsigned int *)(p) = (val)) //在p处赋一个word的值
#define GET_SIZE(p) ((GET(p)) & ~0x7) //拿到header里的size
#define GET_ALLOC(p) (GET(p) & 0x1) // 拿到header里的allocated fields
#define HDRP(bp) ((char *)(bp) - WSIZE) //bp指向一个block真正有效的位置，找到其header的位置
#define FTRP(bp) ((char *)(bp)+GET_SIZE(HDRP(bP)) - DSIZE) // GET_SIZE(HDRP(P))计算出该block有多少分bytes(包括header和footer的8 bytes)
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp))) //计算下一块的header位置
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE))) // 计算上一块的header的位置，((char *)(bp) - DSIZE))为上一块的footer
```

mm_init

```
int mm_init(void){

    if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1) return -1; //开出四个word的空间，教材p857
    /*heap_list 现在指向heap上的第一个byte*/
    PUT(heap_listp, 0); //栈上第一个block，
    PUT(heap_listp+(1*WSIZE), PACK(DSIZE,1)); //prologue block的header(一个word)
    PUT(heap_listp+(2*WSIZE), PACK(DSIZE,1); // prologue block的footer(一个word)
    PUT(heap_listp+(3*WSIZE), PACK(0,1)); //epilogue header
    heap_listp += 2*WSIZE; //当前堆顶的位置,刚好在prologue block的header的后面。
    if (extend_heap(CHUNKSIZE/WSIZE) == NULL) return -1;//把堆扩展CHUNKSIZE.
    return 0;

}
```

extend_heap

```
/*扩展堆*/
void *extend_heap(size t words){
    /*words 表示需要拓展的word数*/
    char *bp;
    size_t size;
    /*size是需要扩展的字节数，注意assignment的问题*/
    size = (words % 2) ? (words+1)*WSIZE : words * WSIZE;
    if ((long)(bp = mem_brk(size)) ==-1 ) return NULL; //bp == mem_brk(size) == (void *) -1;
    /*bp现在指向新开出的block的第一个byte，它的前面是prologue header*/
    PUT(HDRP(bp),PACK(size,0)); //初始化header
    PUT(FTRP(bp), PACK(size,0)); //初始化footer
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0,1)); //之前的epiogue block被覆盖了，需要创建个新的。
```

```
13
14      return coalesce(bp); //合并，往下看。
15
16  }
```

mm_free

```
1  /*释放块*/
2  void mm_free(void *ptr){
3      /*free一个block就只需修改其header和loader的alloc位，并且将其与其他free block合并*/
4      if (ptr == 0) return;
5      size_t size = GET_SIZE(HDRP(ptr));
6      PUT(HDRP(ptr), PACK(size,0)); //修改header
7      PUT(FTRP(ptr),PACK(size,0)); // 修改footer
8      coalesce(ptr); //合并，往下看
9  }
```

coalesce

```
1  /*合并*/
2  void *coalesce(void *bp){
3      size_t pre_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp))); //前一块是否被分配
4      size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp))); //后一块是否被分配
5      size_t size = GET_SIZE(HDRP(bp)); //当前块的大小
6      /*分四种情况：
7      - 前后均被分配
8      - 前面被分配后面没被分配
9      - 前面没被分配后面被分配
10     - 前后都没被分配     */
11     if (pre_alloc&&next_alloc){ //前后都被分配
12         return bp;
13     }
14     else if (pre_alloc &&!next_alloc){ //前面被分配，后面空闲
15         size += GET_SIZE(HDRP(NEXT_BLKP(bp))); //当前块的大小+后面空闲块的大小
16         PUT(HDRP(bp), PACK(size, 0)); //修改header
17         PUT(FTRP(bp), PACK(size, 0)); //修改footer，必须先修改header
18     }
19     else if(!prev_alloc && next_alloc){ //前面没被分配，后面被分配
20         size += GET_SIZE(HDRP(PREV_BLKP(bp))); //当前块大小+前面块大小
21         PUT(FTRP(bp), PACK(size, 0)); //修改当前块的footer，当前块的footer就是合并后
   的footer
22         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0)); //修改前一块的header，前一块的
   header是合并后的header
23         bp = PREV_BLKP(bp); //把bp拿到前一块的bp去，也就是合并后的bp所在位置。
24     }
25     else{ //前后都没被分配
26         size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
   //当前块的大小加前后两块的大小
27         PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0)); //后一块的footer为合并后的footer
28         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0)); // 前一块的header为合并后的header
29         bp = PREV_BLKP(bp);//把bp拿到前一块的bp去，也就是合并后的bp所在位置。
```

```
    }
    return bp;
}
```

frist_fit

`课本practice problem 9.8`

```
/*first_fit:返回能放下的那块的bp*/
void *first_fit(size_t asize){ //first_fit策略：前第一块开始查起，如果找到能放下的那块，直接就分配了。
    void *bp;
    for(bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)){ //看到这里你应该明白为什么要有一个epilogue header了吧。
        if (!GET_SIZE(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))){
            return bp;
        }
    }
    return NULL;
}
```

best_fit

```
void *best_fit(size_t asize){ //选择一个能放下当前块的并且最小的。
    void *bp;
    void *best_bp;
    size_t min_size = 0;
    for(bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)){
        if((GET_SIZE(HDRP(bp)) >= asize) && (!GET_ALLOC(HDRP(bp)))){
            if(min_size ==0 || min_size > GET_SIZE(HDRP(bp))){
                min_size = GET_SIZE(HDRP(bp));
                best_bp = bp;
            }
        }
    }
    return best_bp;
}
```

place

`课本practice problem 9.9`

我们在把新的一块放到某个`free block`上时，如果当前块的大小小于`free block`的大小，我们需要把这个`free block`拆成两个部分，第一个部分用来存放当前块，第二部分形成一个新的`free block`。

```
void place(void *bp,size_t asize){
    size_t csize = GET_SIZE(HDRP(bp));
    if ((csize - asize) >= (2*DSIZE)){ //我们规定最小的block为16bytes，否则不能自成一块
        PUT(HDRP(bp),PACK(asize ,1));
```

```
        PUT(FTRP(bp), PACK(asize,1));
        bp = NEXT_BLKP(bp);
        /*把下一块设置为长度为csize-asize的free block*/
        PUT(HDRP(bp), PACK(csize-asize, 0));
        PUT(FTRP(bp), PACK(csize-asize, 0));
    }
    else{ // 否则分配这一整块
        PUT(HDRP(bp),PACK(csize,1));
        PUT(FTRP(bp),PACK(csize,1));
    }
}
```

mm_malloc

```
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 = first_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;
}
```

测试

```
make
./mdriver -t ./traces -V
```

优化

explicit free lists

在我们的朴素模型中，我们发现如果我们想去寻找`free block`，我们只能线性地搜索整个`heap`。

而`explicit free lists`的想法是我们在每个`free block`里面再额外维护两个指针`pred`，`succ`，`pred`指向其前一个`free block`，而`succ`指向其后一个`free block`. 由于`free block`内部的空间是不被使用的，因此我们可以在其内部存储这两个指针。

Segregated free lists

我们还可以把`free block`按照它们的大小分组，每次需要分配一个新的块时，我们就从其对应的组中寻找它的位置。我们把这些组做成一个个链表，每次需要向这些组里添加新的`free block`时都用头插法。

我们把这些链表的头节点都放到哪里呢? 放到`prologue block`的前面。

```
#define GET_HEAD(num) ((unsigned int *)(long)(GET(heap_list + WSIZE * num))) //求第i组的头节点
#define GET_PRE(bp) ((unsigned int *)(long)(GET(bp))) //求前驱的bp
#define GET_SUC(bp) ((unsigned int *)(long)(GET((unsigned int *)bp + 1))) //求后驱的bp
```

我们来想想我们还需要另外实现哪些函数?

- `mm_init`: 初始化时需要初始化每个组的头节点
- `insert`: 当我们得到了一块新的`free block`， 我们要把它插入到对应的链表中去
- `delet`e: 我们重新分配了一块新的`block`后， 我们需要把其从对应的链表中删除
- `place`: 我们放入一块后，多出来的形成一个新的`free block`, 我们需要将其插入到对应的链表中去
- `find_fit`: 当需要放入新的一块时，我们需要去寻找装它的那块`free block`
- `coalesce`: 合并后，需要在对应链表中删除被合并的块