Malloclab

曹菡雯 2020010919

mm_init

`mm_init` 负责初始化堆内存。

- 首先推入一个 8 字节的空闲字用于对齐，然后依次推入多个空闲链表的表头。假设空闲链表包含 `FREE_LIST_LEN` 个链表，那么需要分配 `FREE_LIST_LEN * 8` 字节的空间。
- 接着推入序言块（16 字节），包含一个 8 字节的 `header` 和一个 8 字节的 `footer`，表示该块已分配，大小为 16 字节。
- 最后，推入一个大小为 0 且已分配的结尾块 `header`，用于简化分配过程并标识堆的结尾。

```
int mm_init(void) {
  if ((heap_listp = mem_sbrk(4 * DSIZE + FREE_LIST_LEN * DSIZE)) == (void *) - 1)
    return -1; // error
  // initialize the heap
  PUT(heap_listp, 0);
  memset(heap_listp + (1 * DSIZE), 0, FREE_LIST_LEN * DSIZE);

  free_listp = heap_listp + (1*DSIZE);
  heap_listp += (FREE_LIST_LEN * DSIZE);
  // initialize the prologue block
  PUT(heap_listp + (1*DSIZE), PACK(QSIZE, 1));
  PUT(heap_listp + (2*DSIZE), PACK(QSIZE, 1));
  PUT(heap_listp + (3*DSIZE), PACK(0, 1));
  heap_listp += (2 * DSIZE); // move to the first block

  if(extend_heap(CHUNKSIZE / DSIZE)==NULL){
    return -1;
  }
  return 0;
}
```

mm_malloc

`mm_malloc`完成内存分配，步骤如下：

1. 根据 `size` 查找合适的空闲链表，并沿链表查找第一个足够大的 `block`，若当前链表无合适 `block`，则查找更大尺寸的链表。
2. 将找到的 `block` 从空闲链表中删除，方法是将其前驱和后继节点连接。
3. 在该 `block` 中分配 `size` 字节。确保剩余空间若大于 32 字节，则拆分出一个新 `block`，并调用

`coalesce` 合并相邻的空闲块。

4. `coalesce` 函数会检查当前 `block` 的前后是否有空闲块，若有则合并，移除并重新插入空闲链表。
5. 若找不到合适空间，调用 `extend_heap` 函数扩展堆空间，并通过 `coalesce` 合并新空间，再继续分配。

```
void *mm_malloc(size_t size) {
  if (size == 0){
    return NULL;
  }
  // int newsize = ALIGN(size + SIZE_T_SIZE);
  // void *p = mem_sbrk(newsize);
  // if (p == (void *)-1)
  //   return NULL;
  // else {
  //   *(size_t *)p = size;
  //   return (void *)((char *)p + SIZE_T_SIZE);
  // }
  // todo
  size = ALIGN(reg(size) + 2 * DSIZE);
  void * p;
  if ((p = find_fit(size)) != NULL) {
    list_remove(p);
    return alloc(p, size);
  }
  else{
    p = extend_heap(MAX(size, CHUNKSIZE));
    if( p == NULL)return NULL;
    list_remove(p);
    return alloc(p, size);
  }
}
```

mm_free

`mm_free(ptr)` 负责释放内存。将 `ptr` 指向的 `block` 的 `header` 和 `footer` 标记为未分配，并调用 `coalesce` 函数，合并相邻的空闲块并更新空闲链表。

```
void mm_free(void *ptr) {
  // todo
  if (ptr == NULL) return;
  BP_REMOVE_ALLOC(ptr); // remove the allocated bit
  coalesce(ptr);
  return;
}
```

mm_realloc

`mm_realloc(ptr, size)` 实现重新分配内存，主要流程如下：

1. 处理简单情况，如 `ptr == NULL` 或 `size == 0`，直接调用 `mm_malloc` 或 `mm_free`。
2. 如果 `size` 小于原 `block` 大小，则缩小 `ptr` 指向的 `block`，并调用 `coalesce` 合并剩余空间。
3. 若 `size` 大于原 `block` 大小且后续空间足够，直接扩展当前 `block`，并将剩余空间保持未分配状态。
4. 如果 `ptr` 已在堆的末尾，可以通过 `mem_sbrk` 扩展堆空间。
5. 若以上条件不满足，则分配新空间，复制原内容，再释放原 `block`。

```
void *mm_realloc(void *ptr, size_t size) {
  // todo
  if (size == 0){ // if size is 0, then realloc is equivalent to free
    mm_free(ptr);
    return NULL;
  }
  if (ptr == NULL){ // if ptr is NULL, then realloc is equivalent to malloc
    return mm_malloc(size);
  }
  size = ALIGN(regu(size) + 2 * DSIZE);
  size_t copySize = BP_GET_SIZE(ptr); // ge the original size

  if(size <= copySize){ // // If the new size is less than or equal to the current size, no need to reallocate
    return ptr; // no need to realloc
  }
  // Check if the next block is free and if it can be merged to provide enough space
  if(BP_GET_ALLOC(NEXT_BLKP(ptr)) == 0 && size > copySize){
    // if the next block is free and the size is enough
    size_t newsize = BP_GET_SIZE(ptr) + BP_GET_SIZE(NEXT_BLKP(ptr));
    // If the combined size of the current and next blocks is large enough
    if(newsize >= size){
      list_remove(NEXT_BLKP(ptr));
      PUT(HDRP(ptr), PACK(newsize, 0));
      PUT(FTRP(ptr), PACK(newsize, 0));
      alloc(ptr ,size);
      return ptr;
    }
  }
  // If the next block is the epilogue block (end of heap), we extend the heap
  if ((!BP_GET_SIZE(NEXT_BLKP(ptr))) && size > copySize){

    size_t req_size = size - copySize;
    void* bp;
    if((long)(mem_sbrk(req_size))==-1)return NULL;
```

```
35      // if the next block is the epilogue block
36      PUT(HDRP(ptr), PACK(copySize + req_size, 0));
37      PUT(FTRP(ptr), PACK(copySize + req_size, 0));
38      alloc(ptr, size);
39      PUT(HDRP(NEXT_BLKP(ptr)), PACK(0, 1));
40
41      return ptr;
42    }
43
44    size_t oldSize = BP_GET_SIZE(ptr);  // If none of the above conditions are met,
    allocate a new block
45    void * p = mm_malloc(size);
46    if(p == NULL){
47      return NULL;
48    }
49    // // Copy the data from the old block to the new block
50    memmove(p, ptr, MIN(size, oldSize));
51    mm_free(ptr);
52    return p;
53  }
54
```

Remark

1. 由于 `header` 和 `footer` 占用 8 字节，理论上最小 `block` 大小为 16 字节，但这样的 `block` 无法存储前驱和后继指针，因此空闲块最小需要 32 字节。如果分配时剩余 16 字节，会将其一并分配，以避免段错误。
2. 空闲链表的第一位是一个指针而不是 `block`，因此插入和删除操作时需要特殊处理。
3. 分配内存时，确保及时移除空闲块。我最终选择在执行具体分配函数前移除空闲块。
4. 操作时始终确保 16 字节对齐，使用 `ALIGN` 宏对 `size` 进行调整。

Running Result