Malloc Lab 实验报告

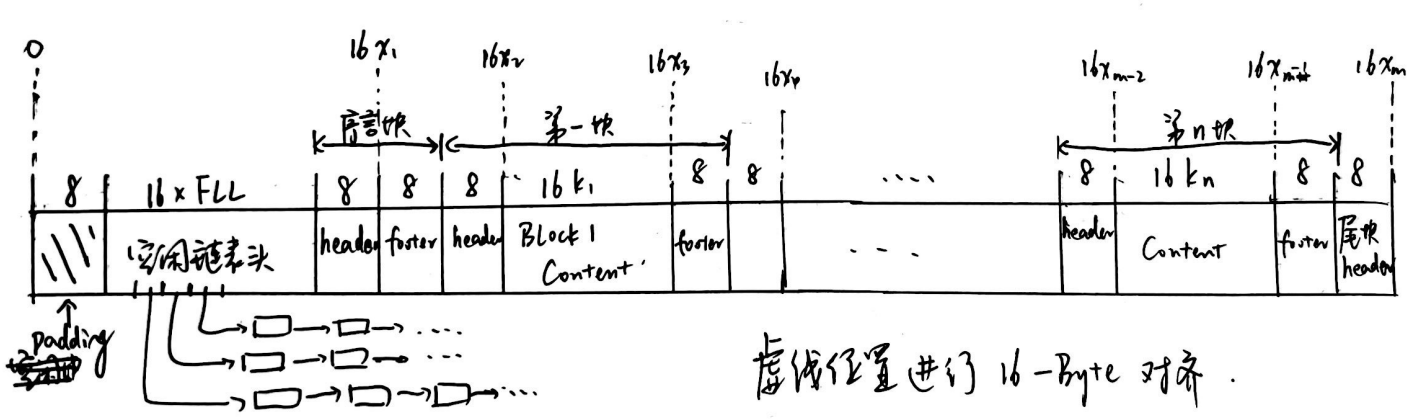
张驰 2022010754 zhang-ch22@mails.tsinghua.edu.cn

1. 堆组织形式

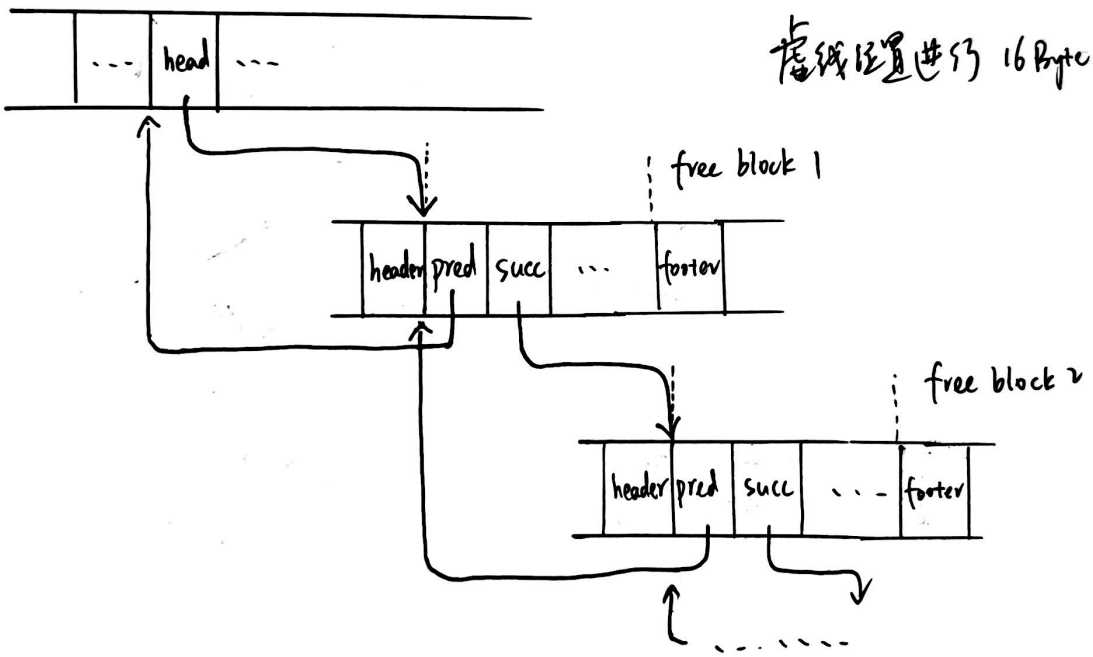
我们把内存划分为许多个block来进行管理，每一个block的地址都经过16字节对齐。为了能够以链表的方式进行定位和查询，每个block前面和后面分别保存了一个header和一个footer，各占据16字节。header和footer用于存储当前block的长度。由于长度必定是16的倍数，所以用二进制表示时最后一个字节全都为0，我们再用最后一个字节的最后一位来表示当前block是否被分配。这样我们就可以把所有block串联成一个双向链表的形式。

为了更加快速地找到未分配且大小足够的block用来分配，我们使用分离的空闲链表方法来存储未被分配的空间。我们将大小在一定范围内的未分配的block用双向链表串联起来，在分配空间时，根据要分配的空间的大小，决定从哪一条链表中寻找合适的未分配block，如果这条链表上没有合适的block，便在更大的链表里寻找。如果我们的链表足够多，那么可以极大地降低寻找合适block的时间。

我们将这些分离链表的头部放在堆的起始位置，使得我们整个堆的结构大概如下：



其中，未分配的block的结构以及其组成的空闲链表如下：



2. 分配算法

2.1. mm_init

`mm_init` 中我们需要完成堆的初始化工作。我们首先推入一个用于对齐的空闲字（8byte），然后推入空闲链表的各个分离表头。若分离链表共由 `FREE_LIST_LEN` 个链表组成，则共需要 `FREE_LIST_LEN * 8byte` 的空间。然后我们推入序言块，这是一个16byte的block，其中header和footer各8block，里面的信息表示此block占据空间大小为16byte且处于已分配状态。最后推入一个结尾块的header，其中信息为此block大小为0，且已分配。这样我们在分配时便不用单独考虑特殊情况，且可以用块的大小是否为0判断是否到达堆的结尾。

2.2. mm_malloc

在函数 `mm_malloc(size)` 中我们需要完成分配工作，大概包含以下几步：

1. 根据 `size` 找到合适的空闲链表，沿着链表找到第一个大小足够的block（如果当前范围的链表没有，则到更大block大小的链表中查找）。
2. 将找到的block从空闲链表中删除。直接把它的前驱和后继节点连到一起即可实现这一点。

3. 在这个block分配 `size` 个字节的空间。由我们的查找过程我们可以保证这个block的空间必然大于size。如果剩余空间很多（大于32byte）足够再划分一个block的话，我们就创造一个新的block，将剩余空间划分成一个新的block，并调用 `coalesce` 函数将这个新的block与相邻的空闲block合并。
4. 在 `coalesce` 函数中，根据当前的block与它的前一个和后一个block的分配情况分开讨论，把临近的空闲block移除空闲链表，合并成一个，再插入回对应大小范围的空闲链表中。
5. 如果遍历了空闲链表但没有足够的剩余空间，那么我们调用 `extend_heap` 函数，使用 `mem_sbrk` 扩大堆的大小，并把新开辟的未分配空间用 `coalesce` 与当前的未分配空间合并，并重新创造结尾块的header。然后再在新的空闲空间进行分配。

2.3. mm_free

在函数 `mm_free(ptr)` 中我们需要完成释放工作。这一步相对简单，我们只需要将 `ptr` 指向的block的header和footer的最后一位改为0，表示这个block未被分配。然后调用 `coalesce` 函数，将这个block与相邻的空闲block合并，并更新对应的空闲链表。

2.4. mm_realloc

在函数 `mm_realloc(ptr, size)` 中我们需要完成重新分配工作。在这一步中，我们主要考虑当前的 `ptr` 后面是否有足够大的空间供我们直接扩容到 `size` 大小：

1. 如果遇到平凡情况，如 `ptr==NULL` 或者 `size==0` ，我们直接调用 `mm_malloc` 或 `mm_free` 即可。
2. 如果需要分配的 `size` 比原来 `ptr` 指向的内存block小，那么我们可以直接将 `ptr` 指向的block缩小，然后调用 `coalesce` 函数，将剩余的空间合并到相邻的空闲block中。
3. 设 `ptr` 指向的block大小为 `old_size` ，如果 `old_size < size` 但是其紧跟有足够多的空间来扩容到 `size` ，那么我们可以直接将 `ptr` 指向的block的后面的一个block拆出一个大小为 `size - old_size` 大小的空间，与 `ptr` 指向的block合并。剩余的空间维持未分配状态。
4. 如果 `ptr` 指向的block已经在堆的结尾，那么我们可以直接使用 `mem_sbrk` 扩大堆的大小，并把新扩大的大小为 `size - old_size` 的空间与 `ptr` 指向的block合。
5. 若上面的情况都不满足，即堆一般的最坏情况，我们只能新分配一段大小为 `size` 的空间，把 `ptr` 指向的block的内容复制过去，然后释放 `ptr` 指向的block。

3. 实现问题

在debug的过程中遇到了不少困难，也是一些需要特别注意的实现上的问题：

1. 由于我们的header和footer都是8字节，所以理论上一个block最小可以只有16字节大小。但实际上这样的block无法被释放，因为被释放时，没有足够的空间来存储它在空闲链表中的前驱和后继。所以我们需要保证最小的空闲单元大于32字节。如果在分配时又16字节的剩余空间，我们就把他也分配给同一个block，以免出现segmentation fault。

2. 在空闲链表中的第一位不是一个空闲的block类型，而是一个单独的指针。所以在这一位置进行插入和删除操作的时候需要特殊判断。使用 `*head` 获取地址，而非用于一般空闲block查看前驱和后继的宏。
3. 在将空闲的block分配内存的时候，必须确保把空闲的block移除空闲链表。由于函数调用关系复杂，我一开始没有正确处理移除操作的时机。我最终选择在执行具体分配函数之前完成移除操作。
4. 在进行各种操作时，必须时刻保证16位对齐，即用 `ALIGN` 宏堆size进行调整。

4. 实验感想

感觉这个实验非常困难，花费了很多时间想明白malloc的实现方式，然后又费了很大力气才de完了所有的bug，最后提高性能也很有难度。充分认识到了编写底层代码需要非常严谨仔细，才能减少错误。

不过完成这个实验也很有收获，对于malloc的实现方式有了更深刻的理解，同时了解了宏在底层代码中的强劲作用。

5. 参考资料

参考了CSAPP书上和官网上的部分实现代码。