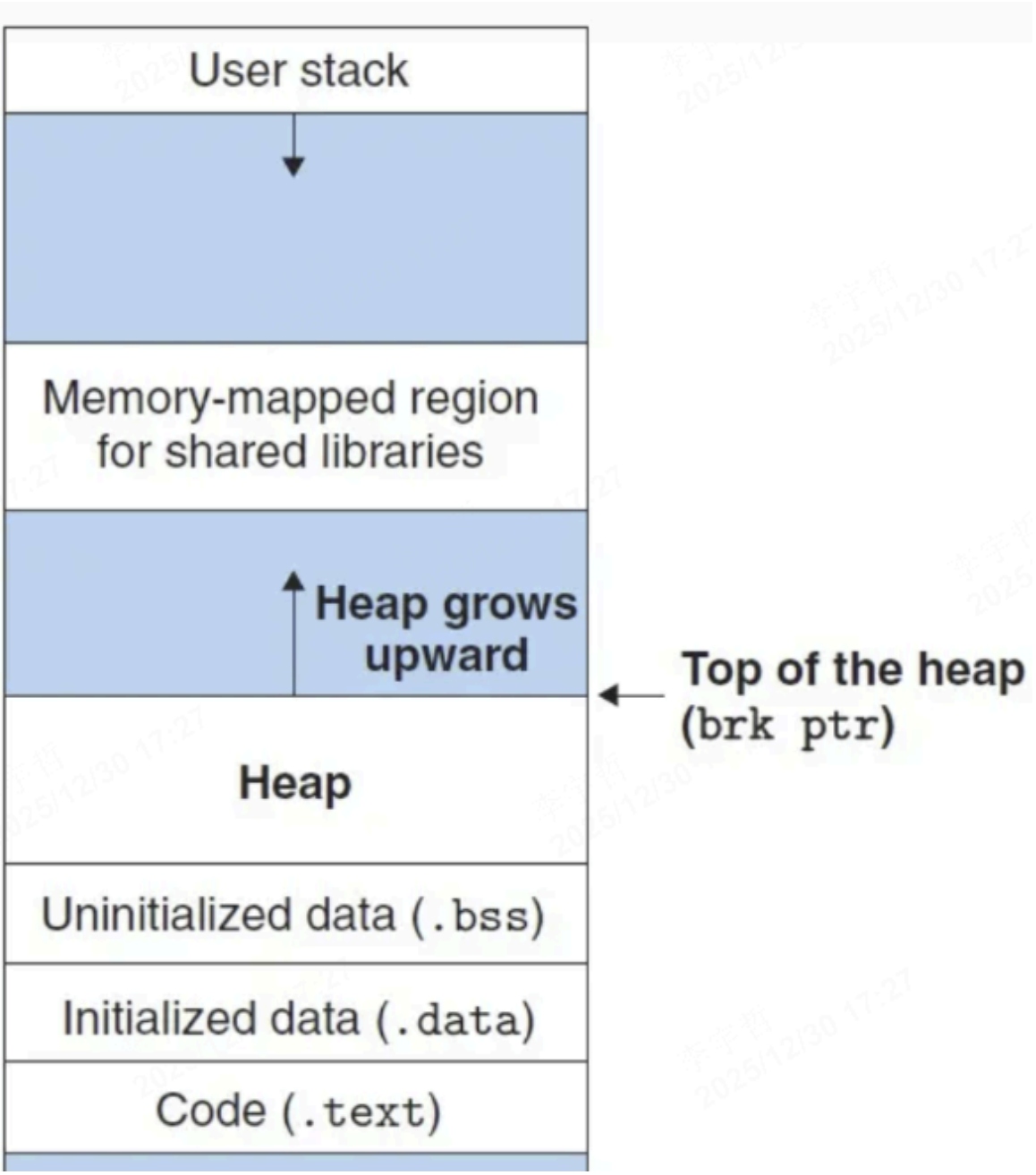
Lab5 Malloc Lab

> notes：这个 Lab 源自 CMU 的 CSAPP 中的 malloclab
>
> 姓名：李宇哲
>
> 学号：SA25011049

这个实验的要求是实现一个 类似 C 中的 malloc，free，realloc 函数，唯一要修改的文件是 `mm.c`

implementation ideas

动态内存申请器（malloc，realloc）为那些在程序运行过程中才能确定大小的数据结构申请虚拟内存空间，动态内存申请器将堆当作一系列大小不同的块来管理，块或者已经申请的，或者是空闲的。



分配器将堆视为一组大小不同的块的集合来维护，地址连续

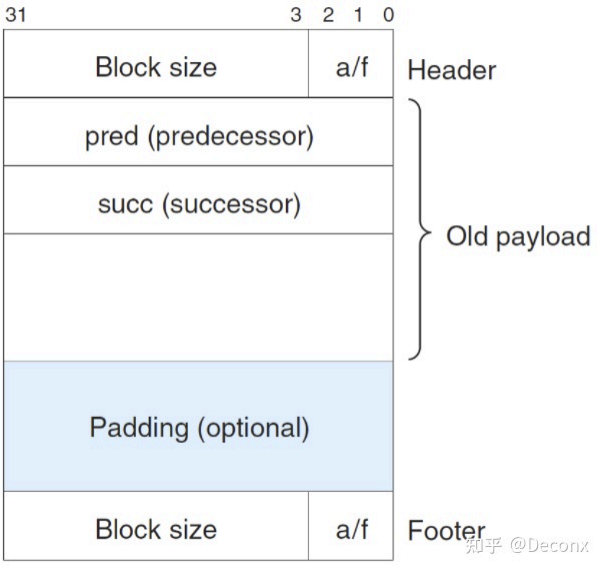
实际上要实现 4个 函数接口

- mm_init()：初始化内存分配器
- mm_malloc()：分配指定大小的内存块
- mm_free()：释放已分配的内存块
- mm_realloc()：重新分配内存块的大小

总体采用 **分离式空闲链表** 实现

核心数据结构

块结构

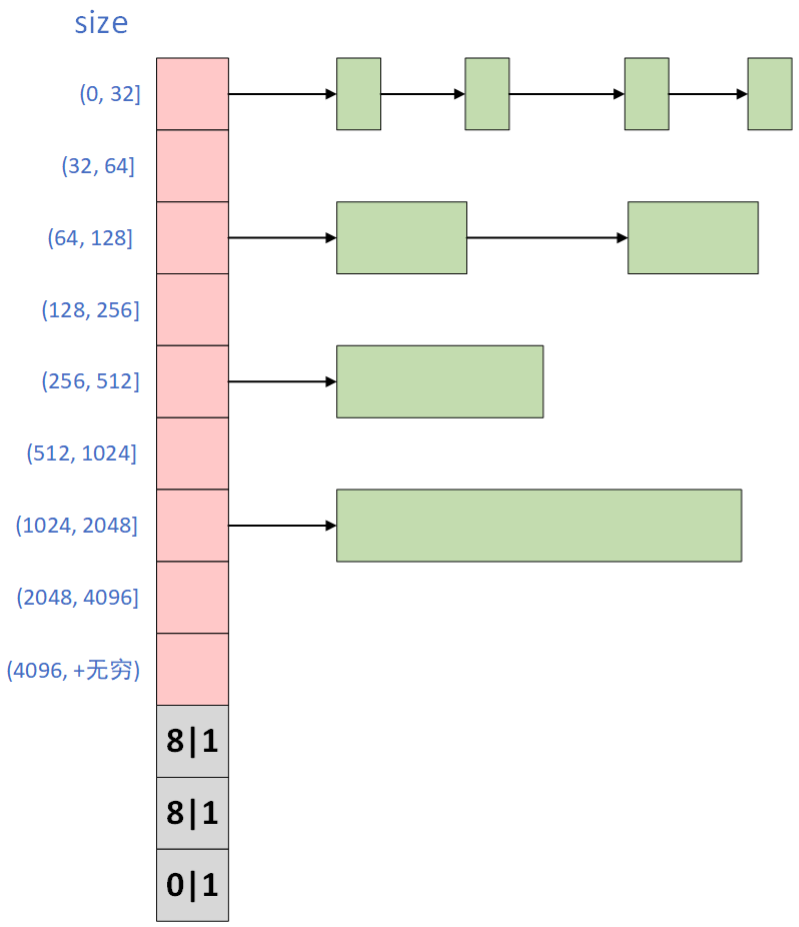


每个内存块由以下部分组成：

```
+------------------+
| Header (4 bytes) |  <- 存储大小和分配状态
+------------------+
| Payload Area     |  <- 用户可用空间
| (空闲块存储)     |     - Prev指针 (8 bytes)
|                  |     - Next指针 (8 bytes)
+------------------+
| Footer (4 bytes) |  <- 存储大小和分配状态（用于合并）
+------------------+
```

- **Header/Footer**：使用4字节（`unsigned int`）存储块大小和分配位
- **最小块大小**：24字节（Header 4 + Footer 4 + Prev 8 + Next 8 = 24）
- **对齐要求**：所有块大小必须是8字节的倍数

分离式空闲链表



维护15个大小类的空闲链表（`SEG_LEN = 15`），采用对数划分策略：

```
get_index(size_t size) {
    if (size <= 24) return 0;
    size = size >> 5;  // 除以32
    // 然后按位右移递增索引
}
```

大小类划分逻辑：

- **索引0**: size ≤ 24
- **索引1**: 24 < size ≤ 64
- **索引2**: 64 < size ≤ 128
- **索引3**: 128 < size ≤ 256
- **索引4**: 256 < size ≤ 512
- **索引5**: 512 < size ≤ 1024
- **索引6**: 1024 < size ≤ 2048
- **索引7**: 2048 < size ≤ 4096
- **索引8-14**: size > 4096（进一步细分）

核心算法

内存分配（mm_malloc）：

- 调整请求大小，对齐到8字节，至少24字节
- 在合适的分离链表中查找适配块（同时支持 first-fit 和 best-fit）
- 如果找到，调用 place() 分割并分配
- 如果为找到，扩展堆并分配

```
void *mm_malloc(size_t size)
{
    size_t asize;
    size_t extendsize;
    char *bp;

    if (size == 0) return NULL;
    if (size <= DSIZE + 8)
        asize = MIN_BLK_SIZE;
    else
        asize = DSIZE * ((size + (DSIZE) + (DSIZE - 1)) / DSIZE);

    if ((bp = find_fit(asize)) != NULL) {
        return place(bp, asize);
    }
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize)) == NULL)
        return NULL;

    return place(bp, asize);
}
```

分配策略：

- **首次适配（First Fit）**：从对应大小类开始搜索，找到第一个满足要求的块
- 较为智能的一种分配方案：
    - 小块（≤64字节）：分配到块的起始位置，剩余部分在后
    - 大块（>64字节）：分配到块的末尾位置，剩余部分在前（减少碎片）

内存释放（mm_free）：

- 标记块为空闲，清除分配位
- 调用 coalesce() 合并相邻空闲块

```c
void mm_free(void *ptr)
{
    if (ptr == NULL) return;

    size_t size = GET_SIZE(HDRP(ptr));

    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));

    coalesce(ptr);
}
```

块合并（coalesce）：

实现四种可能存在的合并情况：

- 前后都分配，将块加入空闲链表
- 前序已经分配，后空闲，与后块合并
- 前块空闲，后块已经分配，与前块合并
- 前后都空闲，合并三个块

```c
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 */
        insert_free_block(bp);
        return bp;
    }

    else if (prev_alloc && !next_alloc) {      /* Case 2 */
        void *next_bp = NEXT_BLKP(bp);
        delete_free_block(next_bp);

        size += GET_SIZE(HDRP(next_bp));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));

        insert_free_block(bp);
    }

    else if (!prev_alloc && next_alloc) {      /* Case 3 */
        void *prev_bp = PREV_BLKP(bp);
        delete_free_block(prev_bp);

        size += GET_SIZE(HDRP(prev_bp));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(prev_bp), PACK(size, 0));

```

```
        bp = prev_bp;
        insert_free_block(bp);
    }

    else {                                      /* Case 4 */
        void *prev_bp = PREV_BLKP(bp);
        void *next_bp = NEXT_BLKP(bp);

        delete_free_block(prev_bp);
        delete_free_block(next_bp);

        size += GET_SIZE(HDRP(prev_bp)) + GET_SIZE(HDRP(next_bp));
        PUT(HDRP(prev_bp), PACK(size, 0));
        PUT(FTRP(next_bp), PACK(size, 0));

        bp = prev_bp;
        insert_free_block(bp);
    }
    return bp;
}
```

块分割（place）：

- 从空闲链表移除块
- 计算剩余大小
- 如果剩余大小 >= 24 字节，则分割
  - 小块：分配前序，剩余后续
  - 大块：分配后续，剩余前序
- 否则整块分配

```
static void *place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));
    size_t remainder = csize - asize;
    delete_free_block(bp);
    if (remainder >= MIN_BLK_SIZE) {

        if (asize > 64) {
             PUT(HDRP(bp), PACK(remainder, 0));
             PUT(FTRP(bp), PACK(remainder, 0));
             insert_free_block(bp);

             void *next_bp = NEXT_BLKP(bp);
             PUT(HDRP(next_bp), PACK(asize, 1));
             PUT(FTRP(next_bp), PACK(asize, 1));
             return next_bp;
        } else {
            PUT(HDRP(bp), PACK(asize, 1));
            PUT(FTRP(bp), PACK(asize, 1));

            void *next_bp = NEXT_BLKP(bp);
```

```
            PUT(HDRP(next_bp), PACK(remainder, 0));
            PUT(FTRP(next_bp), PACK(remainder, 0));
            insert_free_block(next_bp);
            return bp;
        }
    } else {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
        return bp;
    }
}
```

具体实现

一些必要的宏定义

```
/* 基础操作 */
GET(p), PUT(p, val)               // 读写4字节字
GET_PTR(p), SET_PTR(p, ptr)       // 读写8字节指针（64位适配）

/* 块操作 */
HDRP(bp), FTRP(bp)                // 获取头部和脚部地址
NEXT_BLKP(bp), PREV_BLKP(bp)      // 获取相邻块地址

/* 链表操作 */
GET_PRED(bp), GET_SUCC(bp)        // 获取前驱和后继
SET_PRED(bp, ptr), SET_SUCC(bp, ptr) // 设置前驱和后继
```

初始化流程

```
int mm_init(void) {
    1. 分配分离链表数组（15 * 8 = 120字节）
    2. 初始化所有链表头为NULL
    3. 创建初始堆结构：
       - 对齐填充（4字节）
       - 序言块头部（已分配，8字节）
       - 序言块脚部（已分配，8字节）
       - 结束块头部（已分配，0字节）
    4. 扩展堆，分配初始空闲块
}
```

具体代码实现如下：

```
int mm_init(void)
{
    // 1. 分配分离链表头指针数组 (SEG_LEN * 8 bytes)
    if ((segregated_free_lists = mem_sbrk(SEG_LEN * sizeof(void *))) == (void *)-1)
        return -1;

    // 初始化链表头为 NULL
    for (int i = 0; i < SEG_LEN; i++) {
```

```
        segregated_free_lists[i] = NULL;
    }

    // 2. 创建初始堆 (4个字: Padding, Prol H, Prol F, Epil H)
    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1)
        return -1;

    PUT(heap_listp, 0);                            /* Alignment padding */
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1)); /* Prologue header */
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1));     /* Epilogue header */
    heap_listp += (2 * WSIZE);

    // 3. 扩展堆
    if (extend_heap(CHUNKSIZE) == NULL)
        return -1;

    return 0;
}
```

扩展堆

实现为一个 `extend_heap`，一个辅助函数

```
static void *extend_heap(size_t size) {
    1. 对齐请求大小
    2. 调用mem_sbrk()扩展堆
    3. 初始化新块的Header和Footer
    4. 更新结束块头部
    5. 尝试与前一空闲块合并
}
```

具体代码：

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

    asize = ALIGN(size);

    if ((long)(bp = mem_sbrk(asize)) == -1)
        return NULL;
    PUT(HDRP(bp), PACK(asize, 0));
    PUT(FTRP(bp), PACK(asize, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));

    return coalesce(bp);
}
```

realloc

```
void *mm_realloc(void *ptr, size_t size) {
    1. 处理边界情况（ptr==NULL, size==0）
    2. 计算新旧大小
    3. 如果大小相同，直接返回
    4. 否则：malloc新块 → 拷贝数据 → free旧块
}
```

具体代码：

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

     void *newptr;
     size_t copySize;
     size_t oldSize = GET_SIZE(HDRP(ptr));
     size_t asize;

     if (size <= DSIZE + 8)
         asize = MIN_BLK_SIZE;
     else
         asize = DSIZE * ((size + (DSIZE) + (DSIZE - 1)) / DSIZE);

     if (oldSize == asize) return ptr;
     newptr = mm_malloc(size);
     if (newptr == NULL) return NULL;
     copySize = oldSize - DSIZE;
     if (size < copySize) copySize = size;

     memcpy(newptr, ptr, copySize);
     mm_free(ptr);
     return newptr;
 }
```

实验结果

测试方法：编译之后

- ./mdriver -V -f ./short1-bal.rep
- ./mdriver -V -f ./short2-bal.rep

得到具体的分数如下：

```
(base)  base  innerpeace@innerpeace  ~/csapp/lab/lab5  ./mdriver -V -f ./short1-bal.rep
Team Name:SA25011049
Member 1 :Li Yuzhe:lyz1810@mail.ustc.edu.cn
Measuring performance with gettimeofday().

Testing mm malloc
Reading tracefile: ./short1-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   98%      12  0.000000 60000
Total          98%      12  0.000000 60000

Perf index = 59 (util) + 40 (thru) = 99/100
(base)  base  innerpeace@innerpeace  ~/csapp/lab/lab5  ./mdriver -V -f ./short2-bal.rep
Team Name:SA25011049
Member 1 :Li Yuzhe:lyz1810@mail.ustc.edu.cn
Measuring performance with gettimeofday().

Testing mm malloc
Reading tracefile: ./short2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%      12  0.000000 40000
Total          89%      12  0.000000 40000

Perf index = 53 (util) + 40 (thru) = 93/100
```

这已经是一个比较好的分数了，对比之前的显示空闲链表和隐式空闲链表的实现，已经我并不会再做后续优化。