ICS II Malloc Lab 报告

中国人民大学 sheriyuo

1 基本框架

1.1 存储结构

采用分离存储的显式空闲链表来进行处理，以 2 的幂次分割为 $[2^4, 2^5), \cdots, [2^{18}, 2^{19})$ 共 15 个大小类，采用 first-fit & best-fit 混合处理的查询方式。

对于堆内存，先分配每个大小类的空闲链表首指针，空闲块按照地址顺序排序，然后分配序言块、已分配块和结尾块，

1.2 宏定义

```
typedef unsigned int U_INT;
#define WSIZE 4                                     // 单字大小
#define DSIZE 8                                     // 双字大小
#define LIST_LEN 15                                 // 大小类个数
#define CHUNK_SIZE 136                              // 扩展内存大小
#define BLK(size, alloc) ((size) | (alloc))         // 首尾块结构
#define READ(p) (*(U_INT *)(p))                     // 读一个单字
#define WRITE(p, val) (*(U_INT *)(p) = (U_INT)(val)) // 写一个单字
#define SIZE(p) (READ(p) & ~0x7)                    // 获取块大小
#define ALLOC(p) (READ(p) & 0x1)                    // 获取块分配
#define HEAD(bp) ((bp) - WSIZE)                     // 首块
#define FOOT(bp) ((bp) + SIZE(HEAD(bp)) - DSIZE)    // 尾块
#define PREV(bp) ((bp) - SIZE(HEAD(bp) - WSIZE))    // 前一个块
#define NEXT(bp) (FOOT(bp) + DSIZE)                 // 后一个块
#define CSIZE(bp) SIZE(HEAD(bp))                    // 当前块大小
#define PSIZE(bp) SIZE(HEAD(PREV(bp)))              // 前一个块大小
#define NSIZE(bp) SIZE(HEAD(NEXT(bp)))              // 后一个块大小
#define PRED(bp) ((bp) + WSIZE)                     // 前一个空闲块
#define SUCC(bp) (bp)                               // 后一个空闲块
#define RPRED(bp) READ(PRED(bp)) // 跳转至前一个空闲块
#define RSUCC(bp) READ(SUCC(bp)) // 跳转至后一个空闲块
```

1.3 辅助函数

```
int get_index(size_t x);          // 获取大小类编号
void insert(char *bp);            // 插入空闲块
void erase(char *bp);             // 删除空闲块
char *extend(size_t x);           // 扩展堆
char *coalesce(char *bp);         // 合并空闲块
size_t align(size_t x);           // 获取对齐大小
char *find_fit(size_t x, int i); // 获取分配块
char *place(char *bp, size_t x); // 分配块
void unix_error(char *msg);       // 错误处理函数
```

2 具体实现

2.1 get_index

对于一个对齐后大小为 x 的块，其属于的大小类为 $[2^{\lfloor \log_2 x \rfloor}, 2^{\lfloor \log_2 x \rfloor + 1})$，于是只需要实现求 $\log_2 x$ 的函数。

笔者采用 GCC 内建函数 `__builtin_clz` 来实现 $O(1)$ 求 $\log_2 x$，同时对齐最小块为 2^4B 的标号。

```
int get_index(size_t x) {
    int bit = 31 - __builtin_clz(x) - 4;
    if (bit < 0)
        bit = 0;
    if (bit >= LIST_LEN)
        bit = LIST_LEN - 1;
    return bit;
}
```

2.2 insert

对于一个空闲块，首先获取其大小类标号，得到该大小类空闲链表首指针 `it`。随后依次向后遍历空闲链表，找到满足地址顺序的位置插入空闲块，并更新链表指针。

若该大小类内无空闲块或当前块地址最大，则将其分配在空闲链表最后。

```
void insert(char *bp) {
    int index = get_index(CSIZE(bp));
    char *it = list + index * WSIZE;
    while (RSUCC(it)) {
        it = (char *)RSUCC(it);
```

```
        if ((U_INT)it >= (U_INT)bp) {
            char *succ = it;
            it = (char *)RPRED(it);
            WRITE(SUCC(it), bp);
            WRITE(PRED(bp), it);
            WRITE(SUCC(bp), succ);
            WRITE(PRED(succ), bp);
            return;
        }
    }
    WRITE(SUCC(it), bp);
    WRITE(PRED(bp), it);
    WRITE(SUCC(bp), NULL);
}
```

2.3 erase

删除的块位于空闲链表的中间节点，判断其前后指针的存在情况后，将其从链表中删除，并置 PRED 和 SUCC 指针为 NULL。

```
void erase(char *bp) {
    if (RPRED(bp) && RSUCC(bp)) {
        WRITE(SUCC(RPRED(bp)), RSUCC(bp));
        WRITE(PRED(RSUCC(bp)), RPRED(bp));
    } else if (RPRED(bp)) {
        WRITE(SUCC(RPRED(bp)), NULL);
    }
    WRITE(PRED(bp), NULL);
    WRITE(SUCC(bp), NULL);
}
```

2.4 extend

调用 mem_sbrk 函数进行堆扩展，为其返回的头指针分配其首尾块，作为空闲块传入 coalesce 函数中进行空闲块的合并与插入。

```
char *extend(size_t x) {
    char *bp = mem_sbrk(x);
    if (bp == (void *)-1)
        return NULL;
    WRITE(HEAD(bp), BLK(x, 0));
    WRITE(FOOT(bp), BLK(x, 0));
    WRITE(HEAD(NEXT(bp)), BLK(0, 1));
    return coalesce(bp);
```

```
}
```

每次扩展的最小单位为 CHUNK_SIZE，经过多次调参后取 128+8=136，可以在最优适配 traces/binary2-bal.rep(a 16,a 112,a 128) 的情况下保持 util 的最高。

表 1 CHUNK_SIZE 对 util 的影响

CHUNK_SIZE	binary-bal	binary2-bal	realloc-bal	realloc2-bal	util
64	55%	65%	100%	100%	91%
72	55%	76%	100%	100%	92%
128	52%	80%	100%	99%	92%
136	97%	88%	100%	99%	97%
144	73%	50%	100%	99%	92%
256	81%	56%	100%	99%	93%
264	80%	50%	97%	99%	92%
512	87%	74%	100%	98%	94%
520	97%	58%	25%	98%	87%

其中 CHUNK_SIZE 为 520 时 binary-bal 的效果最好，但是 realloc-bal 的效果极差；取 136 的总效果最好。

2.5 coalesce

对于前后合并的 4 种情况分类讨论，在空闲链表中删去被合并的空闲块，并将合并得到的新块重新插入空闲链表中。

```
char *coalesce(char *bp) {
    size_t palloc = ALLOC(FOOT(PREV(bp)));
    size_t nalloc = ALLOC(HEAD(NEXT(bp)));
    size_t size = CSIZE(bp);
    if (!palloc && !nalloc) {
        size += PSIZE(bp) + NSIZE(bp);
        erase(PREV(bp));
        erase(NEXT(bp));
        WRITE(HEAD(PREV(bp)), BLK(size, 0));
        WRITE(FOOT(NEXT(bp)), BLK(size, 0));
        bp = PREV(bp);
        WRITE(PRED(bp), NULL);
        WRITE(SUCC(bp), NULL);
    } else if (palloc && !nalloc) {
        size += NSIZE(bp);
        erase(NEXT(bp));
        WRITE(HEAD(bp), BLK(size, 0));
```

```
        WRITE(FOOT(bp), BLK(size, 0));
        WRITE(PRED(bp), NULL);
        WRITE(SUCC(bp), NULL);
    } else if (!palloc && nalloc) {
        size += PSIZE(bp);
        erase(PREV(bp));
        WRITE(FOOT(bp), BLK(size, 0));
        WRITE(HEAD(PREV(bp)), BLK(size, 0));
        bp = PREV(bp);
        WRITE(PRED(bp), NULL);
        WRITE(SUCC(bp), NULL);
    }
    insert(bp);
    return bp;
}
```

2.6 align

将有效负载加上首尾块后对齐 8 字节，如果小于最小块大小就设为 16B。

```
size_t align(size_t x) {
    if (x <= DSIZE)
        return 2 * DSIZE;
    else
        return DSIZE * ((x + 2 * DSIZE - 1) / DSIZE);
}
```

2.7 find_fit

从第 i（由 get_index 求出）个大小类开始查找，若找不到则继续向后查找。

在 first-fit 的基础上合并 best-fit，在找到第一个适配块后继续向后查找 try_times 次，可在不影响 thru 的情况下有一定优化效果。

```
char *find_fit(size_t x, int i) {
    int try_times = 10;
    char *ans = NULL;
    while (i < LIST_LEN) {
        char *bp = list + i * WSIZE;
        while (RSUCC(bp)) {
            bp = (char *)RSUCC(bp);
            if (CSIZE(bp) >= x) {
                try_times--;
                if (ans == NULL || CSIZE(ans) > CSIZE(bp))
                    ans = bp;
```

```
                if (try_times <= 0) {
                    return ans;
                }
            }
        }
        i++;
    }
    return ans;
}
```

2.8 place

在分配块时，如果适配块大小比分配大小大很多，可以通过分割块的方式来减少内部碎片。

如果剩余大小不超过最小块大小，则直接分配；否则，将其分配在空闲块的后部，前部分割出来作为新的空闲块。

```
char *place(char *bp, size_t x) {
    size_t size = CSIZE(bp);
    size_t rsize = size - x;
    if (!ALLOC(HEAD(bp)))
        erase(bp);
    if (rsize >= 2 * DSIZE) {
        WRITE(HEAD(bp), BLK(rsize, 0));
        WRITE(FOOT(bp), BLK(rsize, 0));
        WRITE(HEAD(NEXT(bp)), BLK(x, 1));
        WRITE(FOOT(NEXT(bp)), BLK(x, 1));
        insert(bp);
        return NEXT(bp);
    } else {
        WRITE(HEAD(bp), BLK(size, 1));
        WRITE(FOOT(bp), BLK(size, 1));
    }
    return bp;
}
```

2.9 mm_init

使用 `mem_sbrk` 为初始堆分配空间，分配后 s 以一个 `CHUNK_SIZE` 开始栈空间的分配。

```
int mm_init(void) {
    if ((list = mem_sbrk((LIST_LEN + 3) * WSIZE)) == (void *)-1)
        unix_error("Init list error");
```

```
    for (int i = 0; i < LIST_LEN; i++)
        WRITE(list + i * WSIZE, NULL);
    WRITE(list + LIST_LEN * WSIZE, BLK(DSIZE, 1));
    WRITE(list + (LIST_LEN + 1) * WSIZE, BLK(DSIZE, 1));
    WRITE(list + (LIST_LEN + 2) * WSIZE, BLK(0, 1));
    if (extend(CHUNK_SIZE) == NULL)
        unix_error("Extend heap error");
    return 0;
}
```

2.10 mm_malloc

求出对齐后的空间，并在空闲链表中查找适配。若查找不到适配块，则扩展堆来分配空间。用 place 来分配块内存。

```
void *mm_malloc(size_t size) {
    size = align(size);
    if (size == 0)
        return NULL;
    char *bp = find_fit(size, get_index(size));
    if (bp == NULL) {
        if ((bp = extend(max(size, CHUNK_SIZE))) == NULL)
        return NULL;
    }
    return place(bp, size);
}
```

2.11 mm_free

释放块内存，并作为空闲块合并后插入空闲链表中。

```
void mm_free(void *ptr) {
    size_t size = CSIZE(ptr);
    WRITE(HEAD(ptr), BLK(size, 0));
    WRITE(FOOT(ptr), BLK(size, 0));
    coalesce(ptr);
}
```

2.12 mm_realloc

对于 realloc，考虑重分配的块的前后块，若有空闲块可合并则合并，无空闲块则讨论其后块的情况。如果其后是结尾块，则直接使用 extend 分配空间；否则，执行 free 后再次 malloc。

向前合并涉及到载荷数据的移动，实现较为复杂，只实现向后合并或者不实现合并的测试结果并无变化。

查看测试数据后发现，其只对一开始的分配块重分配。在 place 的实现中采用向前分割空闲块的方式，那么每次重分配块的后一个块都是结尾块，直接使用 extend 来分配空间可以极大提高利用率，从而实现接近 100% 的 util。

```
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 old_size = CSIZE(ptr);
    size = align(size);
    if (size == old_size)
        return ptr;
    // 此处合并可以不实现
    if (!ALLOC(HEAD(NEXT(ptr))) && (old_size + NSIZE(ptr) >= size)) {
        size_t new_size = old_size + NSIZE(ptr);
        erase(NEXT(ptr));
        WRITE(HEAD(ptr), BLK(new_size, 1));
        WRITE(FOOT(ptr), BLK(new_size, 1));
        place(ptr, new_size);
    } else if (!NSIZE(ptr) && size >= old_size) {
        if (mem_sbrk(size - old_size) == (void *)-1)
        return NULL;
        WRITE(HEAD(ptr), BLK(size, 1));
        WRITE(FOOT(ptr), BLK(size, 1));
        WRITE(HEAD(NEXT(ptr)), BLK(0, 1));
        place(ptr, size);
    } else {
        char *new_ptr = mm_malloc(size);
        if (new_ptr == NULL)
        return NULL;
        memcpy(new_ptr, ptr, min(old_size, size));
        mm_free(ptr);
        return new_ptr;
    }
    return ptr;
}
```

3 实验结果

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000161 35257
 1       yes  100%    5848  0.000179 32598
 2       yes   99%    6648  0.000205 32509
 3       yes  100%    5380  0.000155 34665
 4       yes   97%   14400  0.000119121212
 5       yes   95%    4800  0.000273 17563
 6       yes   95%    4800  0.000273 17576
 7       yes   97%   12000  0.010386  1155
 8       yes   88%   24000  0.036980   649
 9       yes  100%   14401  0.000104138738
10       yes   99%   14401  0.000127113215
Total          97%  112372  0.048962  2295

Perf index = 58 (util) + 40 (thru) = 98/100
```

图 1: 本地实验结果

对于 `binary2-bal` 的优化已接近显式空闲列表的上限，总体达到了 `97%` 的 `util`，优于很多使用了 BST 的实验结果，在不面向数据特判的情况下已经十分理想。