分配器搜索改进

通过前文的分析可知分配器优化的方向，那就是减少空闲块的搜索时间。前文的搜索算法速度与堆中的所有内存块总数量成正比，当堆中分配块数量不多时这种方法没有什么问题，但是当堆中分配块数量较多并且集中于前部时，那么搜索空闲块时不得不一个个的检索这些分配块，因此造成了搜索时间开销。既然我们想减少搜索空闲块时间，那为何不只搜索空闲块呢？基于这一想法有了下文的优化方法。

显式空闲链表块

既然我们想在查找空闲块时只搜索空闲块，那就必须将所有空闲块通过某种方式连接起来。考虑到空闲块还没有被使用，并且负载至少有两个字的空闲，因此可使用空闲块负载的前两个字作为指向前一空闲块（prev）和后一空闲块（next）的指针，这种通过直接地址来连接的方式称为显式空闲链表，如下图所示。

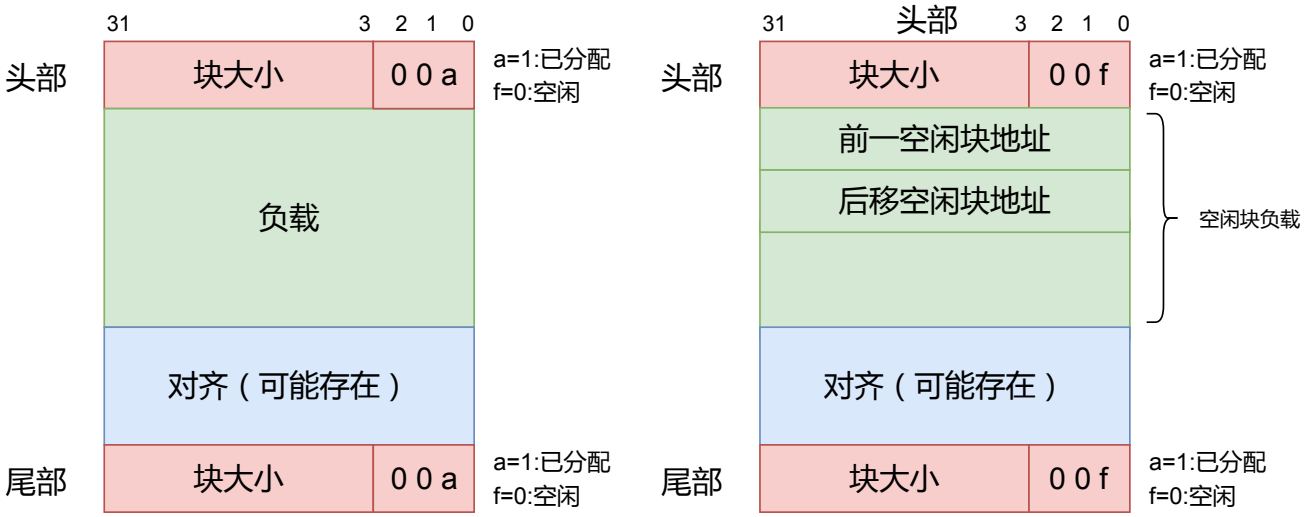

图1 带指针的空闲块

通过这种空闲块的连接方式，相当于在原来隐式空闲链表方式的基础上多出了一个空闲块的显式链表，在分配内存寻址合适的空闲块时就不再需要检索分配块，从而缩短搜索时间。

修改代码

宏和全局变量补丁

```
#define EXP_LIST
...
#ifdef EXP_LIST
#define NEXT_FREE_BLKP(bp)  ((char *)GET((char *)bp + WSIZE))           /* 下一空闲块 */
#define PREV_FREE_BLKP(bp)  ((char *)GET(bp))                           /* 前一空闲块 */
static char *exp_list_head = NULL;           /* 指向第一块空闲块，如果没有空闲块则为NULL */
static char *exp_list_tail = NULL;           /* 指向最后一块空闲块，如果没有空闲块则为NULL */
static int free_blocks_num = 0;             /* 空闲块数量 */
#endif
```

通过EXP_LIST宏来增加显式空闲链表部分的补丁代码。根据显式空闲链表的组织方式额外增加以上宏和全局变量，它们的含义见注释，需要注意的是exp_list_head和exp_list_tail，当堆中没有空闲块时它们都为NULL，当只有一个空闲块时它们两个相等，除了NULL之外，exp_list_head指向的空闲块的prev指针指向exp_list_tail指向的空闲块，exp_list_list指向的空闲块的next指针指向exp_list_head指向的空闲块。

mm_init补丁

```
/*
 * 创建并初始化堆
 */
int mm_init(void)
{
    void *ptr;
    /* 首先初始化堆内存地址空间 */
    mem_init();

    /* 创建一个空堆 */
    heap_listp = mem_heap;
    PUT(heap_listp, 0);                          // 堆头块初始化
    PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); // 预言块头部初始化
    PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); // 预言块尾部初始化
    PUT(heap_listp + (3*WSIZE), PACK(0, 1));     // 堆尾块初始化
    heap_listp += (2*WSIZE);                     // list_listp初始化指向预言块
    mem_brk = heap_listp + (2 * WSIZE);          // mem_brk初始化指向当前堆的
                                                 // 最后一个字节的后面一个字节

#ifdef NEXT_FIT
    rover = heap_listp;                          // rover初始化
#endif

    /* 给空堆增加一个CHUNKSIZE字节的空闲块 */
    if ((ptr = extend_heap(CHUNKSIZE/WSIZE)) == NULL)
        return -1;

    /* 显式空闲链表补丁 */
#ifdef EXP_LIST
    exp_list_head = heap_listp + (2 * WSIZE);       //初始化指向第一块空闲块的指针
    exp_list_tail = exp_list_head;                  //初始化指向最后一块空闲块的指针
    PUT(exp_list_head, exp_list_head);              //初始化唯一空闲块的前一块地址域
    PUT(exp_list_head + WSIZE, exp_list_head);      //初始化唯一空闲块的后一块地址域
    free_blocks_num = 1;                            //当前有一个空闲块
#endif

    return 0;
}
```

在原来堆初始化后会有一个空闲块，此时通过初始化补丁代码修改该块的prev和next指针分别指向自身，初始化exp_list_head、exp_list_list和free_blocks_num。

extend_heap补丁

```
/*
```

```
 * 延长堆并返回新空闲块地址,可能是经过合并后的
 */
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;

    /* 分配一个双字对齐的新堆空间 */
    size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
    if ((bp = mem_sbrk(size)) == NULL)
        return NULL;

    /* 初始化新的空闲块的头部和尾部以及新堆尾块 */
    PUT(HDRP(bp), PACK(size, 0));          // 初始化新空闲块头部
    PUT(FTRP(bp), PACK(size, 0));          // 初始化新空闲块尾部
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); // 新堆尾块

    /* 显式空闲链表补丁 */
#ifdef EXP_LIST
    if (free_blocks_num == 0) {          /* 如果extend之前没有空闲块 */
        PUT(bp, (char *)bp);
        PUT((char *)bp + WSIZE, (char *)bp);
        exp_list_head = bp;
        exp_list_tail = bp;
        free_blocks_num = 1;
    } else {                             /* 如果extend之前有空闲块 */
        PUT(bp, exp_list_tail);                        //初始化新空闲块的前一空闲块域
        PUT((char *)bp + WSIZE, exp_list_head);      //初始化新空闲块的后一空闲块域

        PUT(exp_list_head, (char *)bp);              // 初始化第一空闲块的前空闲块域
        PUT((char *)exp_list_tail + WSIZE, (char *)bp);      // 初始化最后空闲块的后空闲块域

        exp_list_tail = bp;
        free_blocks_num += 1;
    }
#endif

    /* 如果新空闲块的前一块是空闲块则合并 */
    return coalesce(bp);

}
```

如果extend之前没有空闲块，说明这是唯一空闲块，然后初始化exp_list_head和exp_list_head都指向当前空闲块，并修改当前空闲块的prev和next都指向自己，free_blocks_num初始化为1；如果extend之前有空闲块，那么当前新增的空闲块则添加在显式空闲链表末尾，修改原来的最后一块空闲块的next指向新增的空闲块，修改第一块空闲块的prev指向新增的空闲块，初始化新增的空闲块的prev和next分别指向原来的最后一块空闲块和第一块空闲块，修改exp_list_tail指向新空闲块，free_blocks_num加。

find_fit补丁

```
/*
 * 找到一个至少为asize字节大小的空闲块
 */
```

```c
static void *find_fit(size_t asize)
{
    /* 显式空闲链表补丁 */
#ifdef EXP_LIST     /* 显式空闲链表查找 */
    if (free_blocks_num == 0)
        return NULL;
    char *find = exp_list_head;
    for ( ; (GET_SIZE(HDRP(find)) < asize) && find != exp_list_tail; )
        find = NEXT_FREE_BLKP(find);
    /* 到这里说明找到了合适空闲块或者检索到了最后一个空闲块 */
    if (find != exp_list_tail) {
        return find;
    } else {
        if (GET_SIZE(HDRP(exp_list_tail)) >= asize)
            return find;
        else
            return NULL;
    }
#endif

#ifdef NEXT_FIT     /* 下一次适配搜索 */
    /*
     * 获取上次rover停留的地址，即可能是上一次搜索到的空闲块用来
     * 放置分配块的负载首地址，也可能是后文调用mm_free释放后得到
     * 空闲块再经过合并后的最终空闲块的负载首地址
     * 也就是说rover根据mm_malloc和mm_free而移动
     */
    char *oldrover = rover;

    /* 从oldrover搜索到链表结尾来找到第一个满足要求的空闲块 */
    for ( ; GET_SIZE(HDRP(rover)) > 0; rover = NEXT_BLKP(rover))
        if (!GET_ALLOC(HDRP(rover)) && (asize <= GET_SIZE(HDRP(rover))))
            return rover;  //返回找到的空闲块负载首地址

    /* 如果上一步没有找到，则再从链表开头搜索到oldrover来找到第一个满足要求的空闲块 */
    for (rover = heap_listp; rover < oldrover; rover = NEXT_BLKP(rover))
        if (!GET_ALLOC(HDRP(rover)) && (asize <= GET_SIZE(HDRP(rover))))
            return rover;  //返回找到的空闲块负载首地址

    return NULL;  // 没有找到合适空闲块
#else   /* 第一次适配搜索 */
    void *bp;

    for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
        if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
            return bp;
        }
    }
    return NULL;
#endif

}
```

查找空闲块的方法完全替换掉了之前的方法，搜索空闲块只在仅包含空闲块的显式空闲链表中，从显式空闲链表第一块开始查找直到找到合适的空闲块或者搜完所有空闲块，成功则返回对应空闲块指针，失败返回NULL。

place补丁

```
/*
 * 将整块大小为asize的分配块放置到bp为负载首地址的空闲块中
 * 然后分割余下的空闲块,此时传进来的asize一定小于等于bp空闲块负载
 */
static void place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));   //获取空闲块长度

#ifdef EXP_LIST
    char *prev, *next;
    if ((csize - asize) >= (2*DSIZE)) {     /* 还可以分割 */
        prev = GET(bp);                          //提取前一空闲块地址
        next = GET((char *)bp + WSIZE);          //提取后一空闲块地址
        PUT(HDRP(bp), PACK(asize, 1));           //初始化新分配块头部
        PUT(FTRP(bp), PACK(asize, 1));           //初始化新分配块尾部
        bp = NEXT_BLKP(bp);                      //余下的新空闲块负载首地址
        PUT(HDRP(bp), PACK(csize-asize, 0));     //初始化分割出的新空闲块头部
        PUT(FTRP(bp), PACK(csize-asize, 0));     //初始化分割出的新空闲块头部
        if (exp_list_head == PREV_BLKP(bp))      /* 如果exp_list_head指向未place前的空闲块则更新
*/
            exp_list_head = (char *)bp;
        if (exp_list_tail == PREV_BLKP(bp))      /* 如果exp_list_tail指向未place前的空闲块则更新
*/
            exp_list_tail = (char *)bp;

        if (free_blocks_num > 1) {               /* 如果链表有多个空闲块 */
            PUT(bp, prev);                           //初始化新空闲块的前一空闲块域
            PUT((char *)bp + WSIZE, next);           //初始化新空闲块的后一空闲块域
            PUT(prev + WSIZE, (char *)bp);           //初始化前一空闲块指向新空闲块
            PUT(next, (char *)bp);                   //初始化后一空闲块指向新空闲块
        } else {                                 /* 如果链表只有一个空闲块 */
            PUT(bp, (char *)bp);                     //初始化新空闲块的前一空闲块域指向自己
            PUT((char *)bp + WSIZE, (char *)bp);     //初始化新空闲块的后一空闲块域指向自己
        }
    }else { /* 不可以分割 */
        prev = GET(bp);                          //提取前一空闲块地址
        next = GET((char *)bp + WSIZE);          //提取后一空闲块地址
        PUT(HDRP(bp), PACK(csize, 1));           //初始化新分配块头部
        PUT(FTRP(bp), PACK(csize, 1));           //初始化新分配块尾部

        if (free_blocks_num == 1) {              /* 如果链表有多个空闲块 */
            PUT(prev + WSIZE, next);                 //初始化前一空闲块next域
            PUT(next, prev);                         //初始化后一空闲块prev域
            if (exp_list_head == bp)                 /* 如果exp_list_head指向bp则更新指向下一空
闲块 */
                exp_list_head = NEXT_FREE_BLKP(bp);
            if (exp_list_tail == bp)                 /* 如果exp_list_tail指向bp则更新指向前一空
闲块 */
```

```
                exp_list_tail = PREV_FREE_BLKP(bp);
            free_blocks_num -= 1;                   //空闲块数量减1

        } else {                                    /* 如果链表只有一个空闲块 */
            exp_list_head = NULL;
            exp_list_tail = NULL;
            free_blocks_num = 0;
        }
    }
#else
    /* 如果余下的空闲块至少有16字节则分割成一个独立的新空闲块，否则全部用来放置分配块 */
    if ((csize - asize) >= (2*DSIZE)) {     /* 还可以分割 */
        PUT(HDRP(bp), PACK(asize, 1));          //初始化新分配块头部
        PUT(FTRP(bp), PACK(asize, 1));          //初始化新分配块尾部
        bp = NEXT_BLKP(bp);                     //余下的新空闲块负载首地址
        PUT(HDRP(bp), PACK(csize-asize, 0));    //初始化分割出的新空闲块头部
        PUT(FTRP(bp), PACK(csize-asize, 0));    //初始化分割出的新空闲块头部
    } else {                                    /* 不可以分割 */
        PUT(HDRP(bp), PACK(csize, 1));          //初始化新分配块头部
        PUT(FTRP(bp), PACK(csize, 1));          //初始化新分配块尾部
    }
#endif

}
```

首先基于隐式空闲链表分割，然后根据不同的情况来修改显式空闲链表，详见代码注释。

coalesce补丁

```
/*
 * 将bp指定负载的空闲块与相邻空闲块合并，
 * 返回合并得到的空闲块的负载首地址
 */
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp))); //前一块的分配状态
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp))); //后一块的分配状态
    size_t size = GET_SIZE(HDRP(bp));                   //当前空闲块的大小

    if (prev_alloc && next_alloc) {              /* 情况 1 */
        return bp;
    } else if (prev_alloc && !next_alloc) {      /* 情况 2 */
#ifdef EXP_LIST
        if (exp_list_head == (char *)NEXT_BLKP(bp) &&
            exp_list_tail == exp_list_head) {
            PUT(bp, (char *)bp);
            PUT((char *)bp + WSIZE, (char *)bp);
            exp_list_head = bp;
            exp_list_tail = bp;
        } else if (exp_list_head == (char *)NEXT_BLKP(bp)) {
            PUT(bp, exp_list_tail);
            PUT((char *)bp + WSIZE, NEXT_FREE_BLKP(NEXT_BLKP(bp)));
            PUT(NEXT_FREE_BLKP(NEXT_BLKP(bp)), bp);
```

```
            PUT(exp_list_tail + WSIZE, bp);
            exp_list_head = bp;
        } else if (exp_list_tail == (char *)NEXT_BLKP(bp)) {
            PUT(bp, PREV_FREE_BLKP(NEXT_BLKP(bp)));
            PUT((char *)bp + WSIZE, exp_list_head);
            PUT((char *)NEXT_FREE_BLKP(NEXT_BLKP(bp)) + WSIZE, bp);
            PUT(exp_list_head, bp);
            exp_list_tail = bp;
        } else {
            PUT(bp, PREV_FREE_BLKP(NEXT_BLKP(bp)));
            PUT((char *)bp + WSIZE, NEXT_FREE_BLKP(NEXT_BLKP(bp)));
            PUT((char *)NEXT_FREE_BLKP(NEXT_BLKP(bp)) + WSIZE, bp);
            PUT(NEXT_FREE_BLKP(NEXT_BLKP(bp)), bp);
        }
        free_blocks_num -= 1;
#endif
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size,0));
    } else if (!prev_alloc && next_alloc) {      /* 情况 3 */
#ifdef EXP_LIST
        free_blocks_num -= 1;
#endif
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    } else {                                     /* 情况 4 */
#ifdef EXP_LIST
        if (exp_list_head == PREV_BLKP(bp) &&
            exp_list_tail == NEXT_BLKP(bp)) {
            PUT(PREV_BLKP(bp), PREV_BLKP(bp));
            PUT(PREV_BLKP(bp) + WSIZE, PREV_BLKP(bp));
            exp_list_tail = exp_list_head;
        } else if (exp_list_head == PREV_BLKP(bp)) {
            PUT(PREV_BLKP(bp) + WSIZE, PREV_FREE_BLKP(PREV_BLKP(bp)));
            PUT(exp_list_tail + WSIZE, PREV_BLKP(bp));
            PUT(NEXT_FREE_BLKP(NEXT_BLKP(bp)), PREV_BLKP(bp));
        } else if (exp_list_tail == NEXT_BLKP(bp)) {
            PUT(PREV_BLKP(bp) + WSIZE, exp_list_head);
            PUT(exp_list_head, PREV_BLKP(bp));
            exp_list_tail = PREV_BLKP(bp);
        } else {
            PUT(PREV_BLKP(bp) + WSIZE, NEXT_FREE_BLKP(NEXT_BLKP(bp)));
            PUT(NEXT_FREE_BLKP(NEXT_BLKP(bp)), PREV_BLKP(bp));
        }
        free_blocks_num -= 2;
#endif
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
```

```
#ifdef NEXT_FIT
    /* 如果rover指向合并后的空闲块内部，则需要移动到合并后的空闲块负载首地址 */
    if ((rover > (char *)bp) && (rover < NEXT_BLKP(bp)))
        rover = bp;
#endif

    return bp;
}
```

首先基于隐式空闲链表合并，然后根据不同的情况来修改显式空闲链表，详见代码。