操作系统实验 3

题目：动态内存分配器的实现

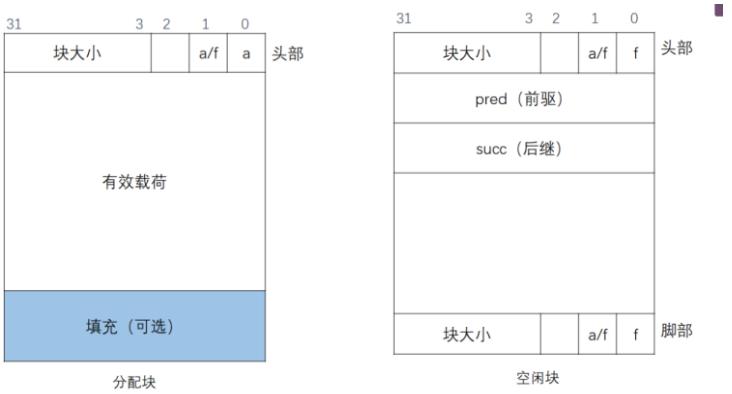
廖洲洲，PB17081504

一、 实验目标

- 使用显示空闲链表实现一个 32 位系统堆内存分配器。

二、 实验步骤

- 块的格式



1) 分配块：由头部、有效载荷部分、可选的填充部分组成。
   - ✧ 头部大小为一个字(32 bits)。
   - ✧ 其中第 3-31 位为存储该块大小的高位。（因为双字对齐，所以低三位都 0）
   - ✧ 第 0 位的值表示该块是否已分配，0 表示未分配（空闲块），1 表示已分配（分配块）。
   - ✧ 第 1 位的值表示该块前面的邻居块是否已分配，0 表示前邻居未分配），1 表示前邻居已分配。
2) 空闲块：由头部、前驱、后继、其余空闲部分、脚部组成。
   - ✧ 头部、脚部的信息与分配块的头部信息格式一样。
   - ✧ 前驱表示在空闲链表中前一个空闲块的地址。后继表示在空闲链表中后一个空闲块的地址。前驱和后继是组成空闲链表的关键。

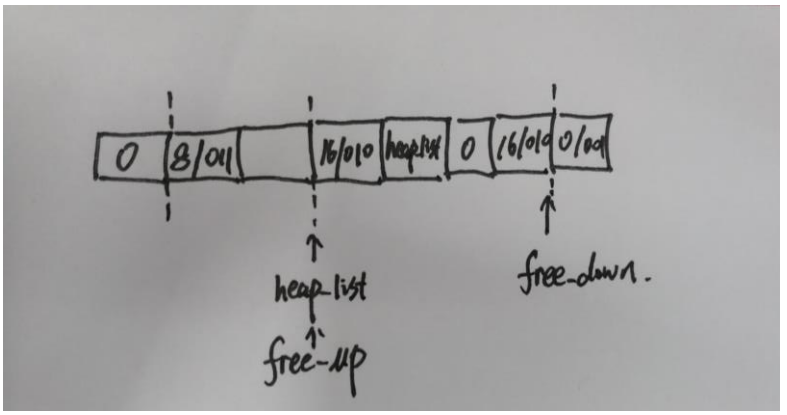
- 链表管理

heap_list 总是指向链表中的第一个空闲块，当它为空时，说明无空闲块。根据空闲块存储的前驱和后继可以遍历所有空闲块。当一个空闲块的后继为空时，说明它为最后一个空闲块。除此外，设置了一个 free_up 和 fre_down 指向已分配有效内存的上下界，这样可以用于判断释放的内存是否合法。

- 放置策略

采用首次适配：从头开始搜索链表，找到第一个大小合适的空闲内存块便返回。

- 初始化分配器



1) 初始化时分配 8 个字大小的内存，2 个字用于一个分配块（序言块），4 个字用于一个空闲块，一个字用于结尾块，一个字是为了双字对齐的填充块。

代码：

```
PUT(heap_list,0);
PUT(heap_list+WSIZE,PACK(8,3));
PUT(heap_list+WSIZE+DSIZE,PACK(16,2));
PUT(heap_list+DSIZE+DSIZE,heap_list+DSIZE+DSIZE);
PUT(heap_list+DSIZE+DSIZE+WSIZE,0);
PUT(heap_list+DSIZE+DSIZE+WSIZE,0);
PUT(heap_list+DSIZE+DSIZE+DSIZE,PACK(16,2));
PUT(heap_list+DSIZE+DSIZE+DSIZE+WSIZE,PACK(0,1));
```

2) 初始化后进行扩堆操作。首先对 size 进行调整，同时将结尾块放置在分配内存的最后位置，然后对空闲块进行合并。这里设置了 end_alloc，用于判断最后一个空闲块是否被分配，方便进行扩堆时空闲块的合并。

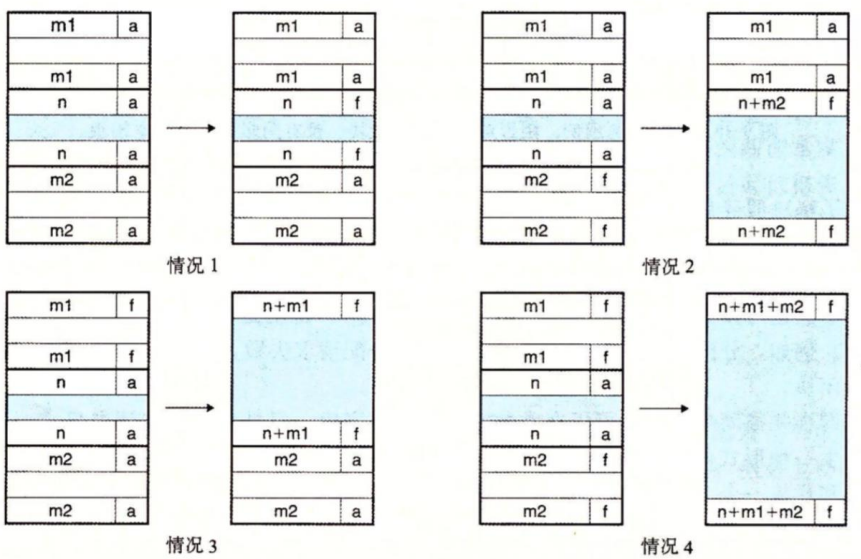
代码：

```
size = (words%2) ? (words + 1)*WSIZE : words *WSIZE;
if((long)(bp = mem_sbrk(size)) == -1)
    return NULL;
free_down +=size;//扩堆成功，修改下界
PUT(HDRP(bp),PACK(size,end_alloc));
PUT(FTRP(bp),PACK(size,end_alloc));
PUT(HDRP(NEXT_BLKP(bp)),PACK(0,1));
return coalesce(bp,end_alloc);//前块未被分配,合并
```

- 合并空闲块

1) 当扩堆时分配的内存最后一块是空块，或者当调用 free 释放某个块后，如果该块相邻有其他的空闲块，则需要将这些块合并成一个大的空闲块，避免出现“假碎片”现象。
2) 判断相邻块是否空闲：由空闲块的头部的 1 号位数据可直接得到上一块邻居块是否空闲的信息，然后通过计算下一块邻居块地址，可以直接取出下一块头部的信息，然后判断其是否是空闲块。
3) 合并操作：有 4 种情况，第一步都是要修改相应的头部脚部。
   - ✧ 前面的块和后面的块都已分配
     通过 get_pre_emptyp(bp)函数得到其上一块空闲块的地址，同时也就知道了下一块空闲块的地址。这里再分为其前面无空闲块，后面无空闲块，前后都有空闲块 3 种情况。一般情况下，修改 3 个空闲块的前驱后继即可，假如其前面无空闲块，还需要修改 heap_list。
   - ✧ 前面的块已分配，后面的块空闲
     这里同样可以分为自己作为第一块空闲块和自己不作为第一空闲块的情况。前者需要修改 heap_list，后者修改相应的前驱后继，注意下一块的空闲块的后继是否为空。
   - ✧ 前面的块空闲，后面的块已分配
     只要修改上一空闲块的头部和脚部即可。
   - ✧ 前后块都空闲
     修改头部和脚部后，修改上邻居块的后继为下邻居块的后继，如果下邻居块的后继不为空，修改下邻居块的后继的前驱。

   图片和隐式的相似：



代码：

```
if(pre_alloc && next_alloc) { //case1
```

```
//      printf("case1\n");
        pre_emptyp = get_pre_emptyp(bp);
        PUT(HDRP(bp),PACK(size,2));
                PUT(FTRP(bp),PACK(size,2));
        if( pre_emptyp == NULL){ //自己作为第一个空块
        //   PUT(HDRP(bp),PACK(size,2));
        //   PUT(FTRP(bp),PACK(size,2));
             PUT_PRE(HDRP(bp),HDRP(bp));
             PUT_SUC(HDRP(bp),heap_list);
             if(heap_list)//作为唯一一个空块
                PUT_PRE(heap_list,HDRP(bp));
             heap_list = HDRP(bp);
             return (void *) bp;
                }
        else if(GET_SUC(pre_emptyp) == 0){  //作为最后一个空块
            PUT_PRE(HDRP(bp),pre_emptyp);
            PUT_SUC(HDRP(bp),GET_SUC(pre_emptyp));
            PUT_SUC(pre_emptyp,HDRP(bp));
            return(void*) bp;
            }
        else { // 其他普通情况
             PUT_PRE(HDRP(bp),pre_emptyp);
             PUT_SUC(HDRP(bp),GET_SUC(pre_emptyp));
            PUT_PRE(GET_SUC(pre_emptyp),HDRP(bp));
            PUT_SUC(pre_emptyp,HDRP(bp));
            return (void*)bp;
            }
        }

    else if(pre_alloc && !next_alloc) {//case2,先修改大小一类
//      printf("case 2\n");
        next_emptyp = HDRP(NEXT_BLKP(bp));
        newsize=size+GET_SIZE(next_emptyp);
        PUT(HDRP(bp),PACK(newsize,2));
        PUT(FTRP(bp),PACK(newsize,2));
        if(heap_list == next_emptyp) {  //后空块是第一空块，即前
面无空块
             PUT_PRE(HDRP(bp),HDRP(bp));
             PUT_SUC(HDRP(bp),GET_SUC(heap_list));
             if(GET_SUC(heap_list))
                PUT_PRE(GET_SUC(heap_list),HDRP(bp));
```

```
                heap_list = HDRP(bp);
                  }
            else {                          //前面有空块
                pre_emptyp = GET_PRE(next_emptyp);
                PUT_PRE(HDRP(bp),pre_emptyp);
                PUT_SUC(HDRP(bp),GET_SUC(next_emptyp));
                if(GET_SUC(next_emptyp))
                    PUT_PRE(GET_SUC(next_emptyp),HDRP(bp));

                PUT_SUC(pre_emptyp,HDRP(bp));
    //          PUT_PRE(next_emptyp,HDRP(bp));
                }
            return (void *)bp;
            }
        //case ok
        else if( !pre_alloc && next_alloc){ //case 3,前面是空块，后面是分配块
    //      printf("case 3\n");
            pre_emptyp = HDRP(PREV_BLKP(bp));
    //      printf("pre_empty:%o\n",pre_emptyp);
            newsize=size+GET_SIZE(pre_emptyp);
            PUT(pre_emptyp,PACK(newsize,2));
    //      PUT(pre_emptyp,PACK(size,2));

    PUT(FTRP(pre_emptyp+WSIZE),PACK(newsize,2));//ERROR&&&&****这里不能用 size+GET_SIZE(pre_empyp),因为 pre_emptyp 的大小在前面已经被修改了
    //      next_emptyp = GET_SUC(pre_emptyp);//合并了，后继前驱都不用变，直接改大小即可
    //      printf("%o=>%o,%o=>%o\n",pre_emptyp,*(unsigned int *)pre_emptyp,FTRP(bp),*(unsigned int *)FTRP(bp));
            return (void *) (pre_emptyp + WSIZE);
    //      if((int)next_emptyp == 0){ //前面的空块是最后一个空块,
    //            PUT_PRE(HDRP(bp),pre_emptyp);
    //            PUT_SUC(HDRP(bp),next_emptyp);
    //            PUT_SUC(pre_emptyp,HDRP(bp));
    //          }
            }

        else { //前后都是空块,直接可得前后空块，修改前空块大小和后继即可
```

```
//      printf("case 4\n");
        next_emptyp = HDRP(NEXT_BLKP(bp));
        pre_emptyp = HDRP(PREV_BLKP(bp));
        void *next_emptyp_next=GET_SUC(next_emptyp);
        newsize=size+GET_SIZE(pre_emptyp)+GET_SIZE(next_emptyp);
        PUT(pre_emptyp,PACK(newsize,2));
        PUT(FTRP(pre_emptyp+WSIZE),PACK(newsize,2));
        PUT_SUC(pre_emptyp,next_emptyp_next);  //修改后继为下一个
空块的后继
        if(next_emptyp_next)  //下一空块的后继不为空，修改前驱
            PUT_PRE(next_emptyp_next,pre_emptyp);
        return (void*)(pre_emptyp +WSIZE);
    }
```

- 释放分配块

1) 修改其分配位为本身未分配
2) 如果下一邻居块为分配块，修改其分配位标志上一块为未分配
3) 合并空闲块

代码：

```
size_t size = GET_SIZE(HDRP(bp));
    size_t pre_alloc = GET_PREALLOC(HDRP(bp));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    if(next_alloc){//下一邻居块被分配的
        next_alloc = GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(NEXT_BLKP(bp)),PACK(next_alloc,1));
        }
    PUT(HDRP(bp),PACK(size,pre_alloc));
    PUT(FTRP(bp),PACK(size,pre_alloc));
coalesce(bp,pre_alloc);
```

- 分配内存

1) 分配内存时首先通过链表查找第一个大小大于要求内存的空闲块
2) 若查找成功，返回地址，然后进行分割操作
3) 查找不成功，扩堆，然后分割

代码：

```
if(size == 0)
    return NULL;

if(size <= WSIZE)
    asize = DSIZE;
else
    asize = DSIZE * ((size + WSIZE + (DSIZE -1)) / DSIZE);
//调整后的大小
```

```
    if( (bp = find_fit(asize)) != NULL) {
//      printf("AFTER FIND_FIT\n");
        place(bp,asize);
        return (void *) bp;
        }

    extendsize = MAX(asize,CHUNKSIZE);
    if((bp = extend_heap(extendsize/WSIZE)) == NULL)
        return NULL;
//  printf("AFTER EXTEND_HEAP");
    place(bp,asize);
```

- 分割空闲块

1) 当分配器找到一个合适的空闲块后，如果空闲块大小大于请求的内存大小（需要大于 4 字，因为一个空块最小的大小为 4 字），则需要分割该空闲块，避免内存浪费。具体步骤为：
   - ✧ 修改空闲块头部，将大小改为分配的大小，并标记该块为已分配。
   - ✧ 为多余的内存添加一个块头部和脚部，记录其大小并标记为未分配，前邻居块标志为已分配，使其成为一个新的空闲内存块。调用合并函数，将其合并入空闲链表。
   - ✧ 返回分配的块指针。

代码：if(GET_SIZE(HDRP(bp)) <= (asize+4*WSIZE)) { //由于最小空闲块的大小为 4 字（16 字节）若找的的空闲块分配后小于 4 字则全部分配完

//尾部块的分配情况判断，当分配的是最后一个空闲块，且该块的下一块是尾端时，说明最后一块是被分配的

```
            if(HDRP(NEXT_BLKP(bp)) == free_down)
                end_alloc=2;//最后一块被分配
            PUT(HDRP(bp),PACK(GET_SIZE(HDRP(bp)),3));
            if(HDRP(bp)==heap_list){  //第一个空块被分配
//              printf("!!!!!\n");
                heap_list = next_emptyp;
                if(next_emptyp)
                    PUT_PRE(next_emptyp,next_emptyp);
                }
            else {//在中间的块或最后一块被分配
//              printf("@@@@@\n");

                PUT_SUC(pre_emptyp,next_emptyp);
                if(next_emptyp)
                    PUT_PRE(next_emptyp,pre_emptyp);
            }
```

```
            }

                else{ //空闲块足够大，需要分割，这种情况下不会出现将最后端分配的情况
                    size = GET_SIZE(HDRP(bp)) - asize;
                    new_emptyp = HDRP(bp) +asize;  //新的空块
                    PUT(HDRP(bp),PACK(asize,3));
                    PUT(new_emptyp,PACK(size,2));
                    PUT(FTRP(new_emptyp+WSIZE),PACK(size,2));
                    if(HDRP(bp) == heap_list){//是第一个空块,修改首空块地址
        //              printf("#####\n");
                        heap_list=new_emptyp;
                        PUT_PRE(new_emptyp,new_emptyp);
                        PUT_SUC(new_emptyp,next_emptyp);
                        if(next_emptyp)   //下一空块存在，修改前驱
                            PUT_PRE(next_emptyp,new_emptyp);
                    }
                    else{ //是中间空块或最后空块
        //              printf("$$$$$\n");
                        PUT_PRE(new_emptyp,pre_emptyp);
                        PUT_SUC(new_emptyp,next_emptyp);
                        PUT_SUC(pre_emptyp,new_emptyp);
                        if(next_emptyp)
                            PUT_PRE(next_emptyp,new_emptyp);
                    }
                }
```

- 合适空闲块的寻找

1) 从第一个空闲块开始向后查找，直到找到第一个大小大于要求内存的空闲块

代码：

```
    if(heap_list==0)
         return NULL;
      p_list = heap_list;
      while( (GET_SIZE(p_list) < asize) &&  (next_list = GET_SUC(p_list)))
         p_list = next_list;
      if(GET_SIZE(p_list) >+asize)
    //    if(!GET_ALLOC(p_list) && GET_SIZE(p_list) >= asize)
         return (void*)(p_list + WSIZE);
  return NULL;
```

三、 实验结果

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.001172  4860
 1       yes   55%   12000  0.001322  9078
 2       yes   51%   24000  0.000889 26984
 3       yes   99%    5848  0.000247 23628
 4       yes   33%   14400  0.000462 31189
 5       yes   99%    6648  0.000284 23400
 6       yes   99%    5380  0.000221 24388
 7       yes   95%    4800  0.001064  4512
 8       yes   94%    4800  0.001068  4494
Total          81%   83570  0.006729 12420

Perf index = 4.83 (util) + 4.00 (thru) = 8.8/10
```

四、 实验问题

- 要注意地址和该地址中的值的区别。在对一个地址赋 int 值时要将它强制类型转换为 unsigned int*，否则如果该地址是 chr *的话只能将一个字节的数字赋值成功，从而出错。在取值时也需要将其转换为相应的指针类型。
- 注意返回的是块的首地址还是首地址+4。
- 注意修改后继的前驱时要判断后继是否为空。
- 由于最小空闲块的大小为 4 字（16 字节），若找的的空闲块分配后小于 4 字则要全部分配完。
- 不能在合并空闲块的时候修改了头部大小再修改脚部大小时仍给脚部赋大小值为 size+GET_SIZE(pre_empyp),，因为 GET_SIZE(pre_empyp)已经发生了改变，应该直接赋值为 GET_SIZE(pre_empyp)已经即可。

五、 实验总结

- 通过这次实验我使用显示空闲链表的方法实现了一个动态内存分配器。
- 这次实验让我更加明白了内存再计算机中的分配过程，了解了基础的内存分配算法。学习了内存管理模型及其数据结构。
- 这次实验让我对指针的操作更加熟练了。