

# 实验三

## 动态内存分配器的实现

### 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。

以下讲解均基于32位系统。

### 二、API简介

malloc/free是c标准库函数，用于从堆中分配内存。

```
void* malloc(size_t size);
```

malloc函数返回一个指针，指向大小至少为size字节的空闲内存块。

返回的内存块首地址是双字对齐的，即：在64位系统中地址为16的倍数，32位系统中为8的倍数。

```
void *sbrk(intptr_t incr);
```

进程的堆内存是有限的，内核维护一个指针brk指向当前堆顶。

当堆中剩余内存不足以响应malloc请求时，需要向操作系统申请更多的堆内存。sbrk函数通过将brk指针增加incr来扩展和收缩堆（即当incr为负数时收缩堆）。成功时返回旧brk的地址（因此扩展堆后返回值指向一块大小为incr的空闲内存），失败时返回-1。

```
void free(void *ptr);
```

free函数释放ptr指向的内存块，释放出的内存可以在之后的malloc调用中被使用。

注意：ptr必须指向由malloc、calloc、realloc分配的内存块的起始位置。

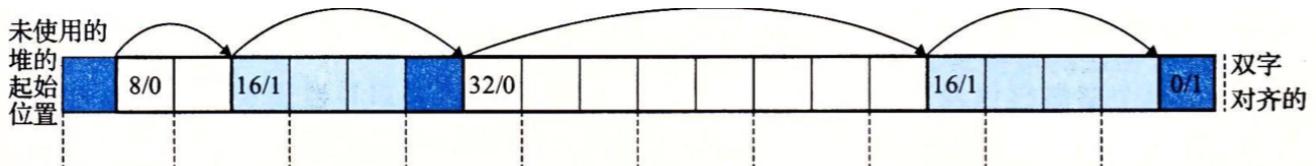
### 三、示例：隐式空闲链表管理

这里通过一种简单的堆内存管理方式，隐式空闲链表，为例来讲解内存分配器的实现。

隐式空闲链表将堆中的内存块按地址顺序串成一个链表，接受到内存分配请求时，分配器遍历该链表来找到合适的空闲内存块并返回。

当找不到合适的空闲内存块时（如：堆内存不足，或没有大小足够的空闲内存块），调用sbrk向堆顶扩展更多的内存。

隐式空闲链表如图所示



图中淡蓝色部分为已分配块，深蓝色为填充块（为了内存双字对齐），数字为块头部，见下节。

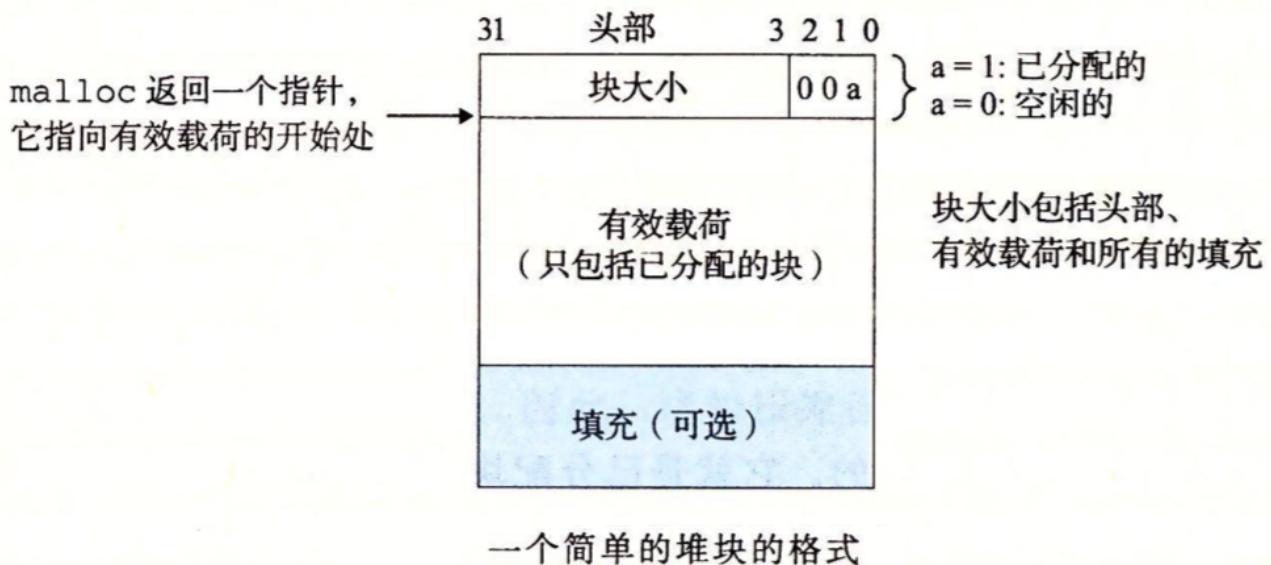
(图中链表指针指向块的起始处，实际实现时是**指向块头部后面一个字，即有效载荷地址**，见下节图)

## 块头部

堆中的各内存块需要某种标志来区分块的边界，记录块的大小，以及标记该内存块是否已被使用。因此为每个内存块保留一个字（4字节）的头部记录这些数据。

块头部记录了该内存块的大小。由于内存块以8字节对齐，块大小二进制的最低3位一定为0，因此可以用最后一位来标记该块是否已被分配。

综上，引入头部后一个内存块的格式如下图所示：



块大小包括头部在内，因此对于一个32字节的内存分配请求，考虑块头部以及内存对齐后，需要为其分配至少40字节的内存块。

## 链表管理

在隐式链表管理方案下，分配器维护一个指针heap\_listp指向堆中的第一个内存块，也即链表中的第一个块。

根据该内存块头部中记录的块大小信息便可计算出下一块的位置(heap\_listp+size)，依此类推。

## 放置策略

当请求一个k字节的内存块时，分配器需要搜索堆中的内存块找到一个足够大的空闲块并返回。具体选择哪一个内存块由放置策略决定。主要有两种：

- 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。
- 最佳适配。搜索整个链表，返回满足需求的最小的空闲块。

两者相比较，首次适配速度较快，最佳适配内存利用率更高。后面的实现采用首次适配方法。

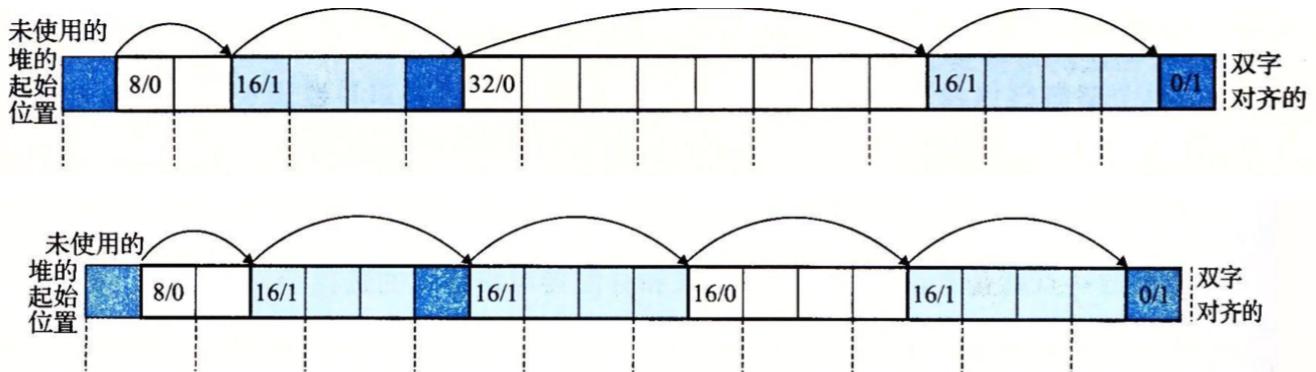
## 分割空闲块

当分配器找到一个合适空闲块后，如果空闲块大小大于请求的内存大小，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，记录其大小并标记为未分配，使其成为一个新的空闲内存块。
- 返回分配的块指针。

例如在如图堆中分配一个16字节的块会将中间32字节的块分割成两个



## 合并空闲块

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（多个小空闲块相邻，无法满足大块内存分配请求）。

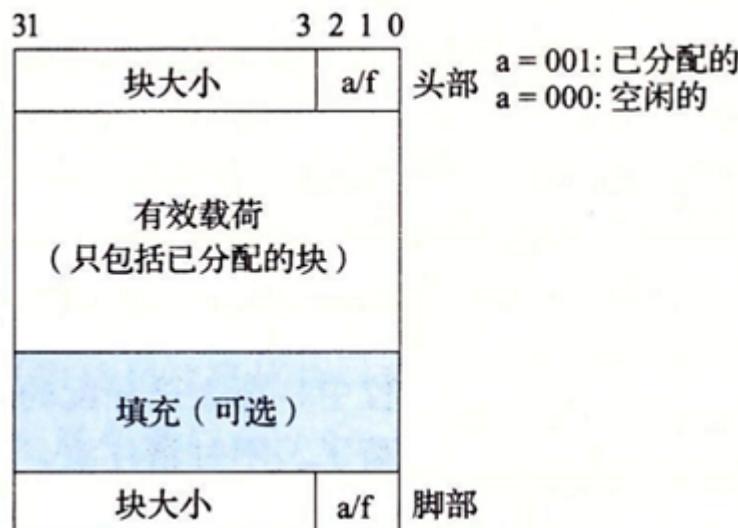
### 判断相邻块是否空闲

判断相邻的下一个块是否空闲很简单：根据当前块的大小即可计算出下一块的头部位置。

但是，对于相邻的前一个块，由于不知道其头部位置，只能从头开始遍历链表，这样性能很差。

解决这个问题的办法是，为每个块再维护一个脚部，内容为头部的复制。有了脚部以后，当前块头部地址向前4个字节便是前一个块的脚部，因此就可以快速地获取前一个块的元数据了。

添加脚部以后，块的格式如图所示：

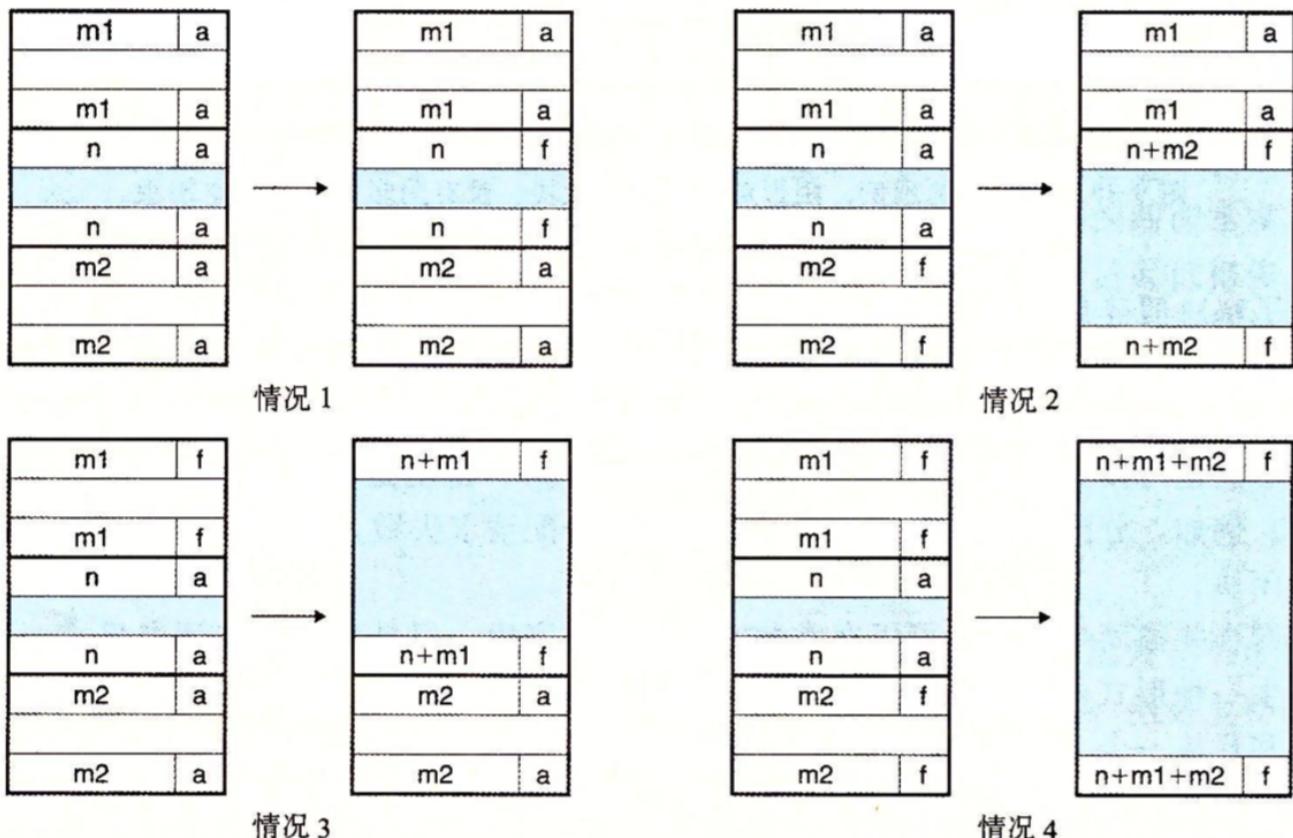


## 合并步骤

释放当前内存块时，根据相邻块的分配状态，有如下四种不同情况：

1. 前面的块和后面的块都已分配
2. 前面的块已分配，后面的块空闲
3. 前面的块空闲，后面的块已分配
4. 前后块都空闲

以下为这四种情况的合并前后示意图



图中m/n表示块大小，a表示已分配，f表示未分配。即根据合并结果修改当前块的头/脚部元数据。

## 扩充堆

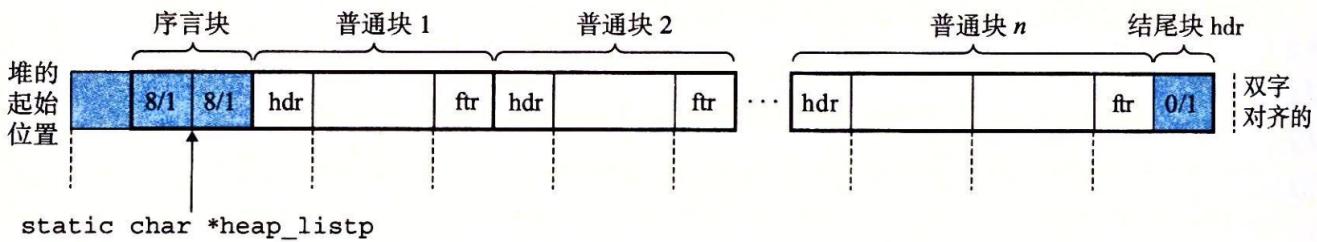
当搜索整个链表都找不到可用的空闲块时，调用sbrk向系统请求更多堆内存，新获得的内存放在当前堆的尾部形成一个新的大空闲块。

## 实现细节

下面介绍隐式空闲链表管理的具体实现代码。

### 堆内存组织格式

以下为该实现中堆内存的组织方式示意图。



隐式空闲链表的恒定形式

堆内存中第一个字是一个为了内存对齐的填充字。填充字后面紧跟一个特殊的序言块，它是一个8字节的已分配块，只由一个头部和一个脚部组成。序言块是在初始化时创建的，并且永不释放。序言块后面是普通块。堆的最后一个字是一个特殊的结尾块，它是一个有效大小为0的已分配块，只由一个头部组成。

序言块和结尾块的作用是消除空闲块合并时的边界检查，在后续代码中可以看到。

分配器使用一个私有全局变量heap\_listp表示链表头，它总是指向序言块。

注意：所有内存块指针指向该块的有效载荷开始处，即头部向后一个字

## 操作空闲链表的常用宏

分配器实现会涉及大量的指针操作，包括从地址取值/赋值，查找块头/块脚地址等。为了方便操作以及保障操作性能，定义如下宏操作。

```

1  /* Basic constants and macros */
2  #define WSIZE      4      /* Word and header/footer size (bytes) */
3  #define DSIZE      8      /* Double word size (bytes) */
4  #define CHUNKSIZE  (1<<12) /* Extend heap by this amount (bytes) */
5
6  #define MAX(x, y) ((x) > (y)? (x) : (y))
7
8  /* Pack a size and allocated bit into a word */
9  #define PACK(size, alloc) ((size) | (alloc))
10
11 /* Read and write a word at address p */
12 #define GET(p)        (*(unsigned int *) (p))
13 #define PUT(p, val)   (*(unsigned int *) (p)) = (val)
14
15 /* Read the size and allocated fields from address p */
16 #define GET_SIZE(p)   (GET(p) & ~0x7)
17 #define GET_ALLOC(p)  (GET(p) & 0x1)
18
19 /* Given block ptr bp, compute address of its header and footer */
20 #define HDRP(bp)      ((char *) (bp) - WSIZE)
21 #define FTRP(bp)      ((char *) (bp) + GET_SIZE(HDRP(bp)) - DSIZE)
22
23 /* Given block ptr bp, compute address of next and previous blocks */
24 #define NEXT_BLKP(bp) ((char *) (bp) + GET_SIZE(((char *) (bp) - WSIZE)))
25 #define PREV_BLKP(bp) ((char *) (bp) - GET_SIZE(((char *) (bp) - DSIZE)))

```

- 针对32位系统，定义字长为4byte (WSIZE)。
- CHUNKSIZE为内存分配器扩充堆内存的最小单元。

- PACK将块大小和分配位结合返回一个值（即将size的最低位赋值为分配位）
- GET/PUT分别对指针p指向的位置取值/赋值
- GET\_SIZE/GET\_ALLOC分别从p指向位置获取块大小和分配位。注意：p应该指向头/脚部
- HDRP/FTRP返回bp指向块的头/脚部
- NEXT\_BLKP/PREV\_BLKP返回与bp相邻的下一/上一块

## 初始化分配器

在开始调用malloc分配内存前，需要先调用mm\_init函数初始化分配器，其主要工作是分配初始堆内存，分配序言块和尾块，以及初始化空闲链表。如下代码所示。

```

1 int mm_init(void)
2 {
3     /* Create the initial empty heap */
4     if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
5         return -1;
6     PUT(heap_listp, 0);                                /* Alignment padding */
7     PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* Prologue header */
8     PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
9     PUT(heap_listp + (3*WSIZE), PACK(0, 1));      /* Epilogue header */
10    heap_listp += (2*WSIZE);
11
12    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
13    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
14        return -1;
15    return 0;
16 }
```

```

1 static void *extend_heap(size_t words)
2 {
3     char *bp;
4     size_t size;
5
6     /* Allocate an even number of words to maintain alignment */
7     size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
8     if ((long)(bp = mem_sbrk(size)) == -1)
9         return NULL;
10
11    /* Initialize free block header/footer and the epilogue header */
12    PUT(HDRP(bp), PACK(size, 0));      /* Free block header */
13    PUT(FTRP(bp), PACK(size, 0));      /* Free block footer */
14    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */
15
16    /* Coalesce if the previous block was free */
17    return coalesce(bp);
18 }
```

mm\_init函数首先通过sbrk请求4个字的内存(line4~5), 然后将这四个字分别作为填充块 (为了对齐) , 序言块头/脚部, 尾块。并将heap\_listp指针指向序言块使其作为链表的第一个节点(line6~10)。之后调用extend\_heap函数向系统申请一个CHUNKSIZE的内存作为堆的初始内存(line13~14)。

extend\_heap函数是对sbrk的一层封装, 接收的参数是要分配的字数, 在堆初始化以及malloc找不到合适内存块时使用。它首先对请求大小进行地址对齐, 然后调用sbrk获取空间 (line7 ~ 9) 。

成功后, 将获取的空间标志为新的空闲块 (line12 ~ 14) 。注意这里实际操作是将扩展前的尾块作为了新空闲块的头块, 然后在新的堆末尾分配一个新的尾块。

## 释放和合并块

mm\_free函数释放一个指针指向的已分配的内存块, 如下页代码所示。

mm\_free首先将其该内存标记为未分配 (line5~6), 然后调用coalesce函数尝试合并相邻空闲块。

coalesce函数首先从前一块的脚部后后一块的头部获取它们的分配状态(line12~13), 然后根据前文所述的4种不同情况作相应处理, 最后返回合并后的指针。

由于序言块和尾块的存在, 不需要考虑边界条件。

```
1 void mm_free(void *bp)
2 {
3     size_t size = GET_SIZE(HDRP(bp));
4
5     PUT(HDRP(bp), PACK(size, 0));
6     PUT(FTRP(bp), PACK(size, 0));
7     coalesce(bp);
8 }
9
10 static void *coalesce(void *bp)
11 {
12     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
13     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
14     size_t size = GET_SIZE(HDRP(bp));
15
16     if (prev_alloc && next_alloc) { /* Case 1 */
17         return bp;
18     }
19
20     else if (prev_alloc && !next_alloc) { /* Case 2 */
21         size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
22         PUT(HDRP(bp), PACK(size, 0));
23         PUT(FTRP(bp), PACK(size, 0));
24     }
25
26     else if (!prev_alloc && next_alloc) { /* Case 3 */
27         size += GET_SIZE(HDRP(PREV_BLKP(bp)));
28         PUT(FTRP(bp), PACK(size, 0));
29         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
30         bp = PREV_BLKP(bp);
31     }
32
33     else { /* Case 4 */
34         size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
35             GET_SIZE(FTRP(NEXT_BLKP(bp)));
36         PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
37         PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
38         bp = PREV_BLKP(bp);
39     }
40     return bp;
41 }
```

## 分配块

最后介绍mm\_malloc函数，向堆申请size大小的内存并返回指针。

```
1 void *mm_malloc(size_t size)
2 {
3     size_t asize;      /* Adjusted block size */
4     size_t extendsize; /* Amount to extend heap if no fit */
5     char *bp;
6
7     /* Ignore spurious requests */
8     if (size == 0)
9         return NULL;
10
11    /* Adjust block size to include overhead and alignment reqs. */
12    if (size <= DSIZE)
13        asize = 2*DSIZE;
14    else
15        asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
16
17    /* Search the free list for a fit */
18    if ((bp = find_fit(asize)) != NULL) {
19        place(bp, asize);
20        return bp;
21    }
22
23    /* No fit found. Get more memory and place the block */
24    extendsize = MAX(asize,CHUNKSIZE);
25    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
26        return NULL;
27    place(bp, asize);
28    return bp;
29 }
```

首先将申请内存大小加上块头/尾部大小并进行对齐(line12~15)，然后调用find\_fit函数(想想怎么实现)从内存块链表中找到合适的块，如果成功找到则调用place函数判断是否需要对该块作分割操作(line18~21)。

如果查找失败则向系统请求分配更多堆内存。为了避免频繁请求，一次最少申请CHUNKSIZE的内存(line24~28)

至此一个隐式链表管理方式的堆内存分配器实现完成。

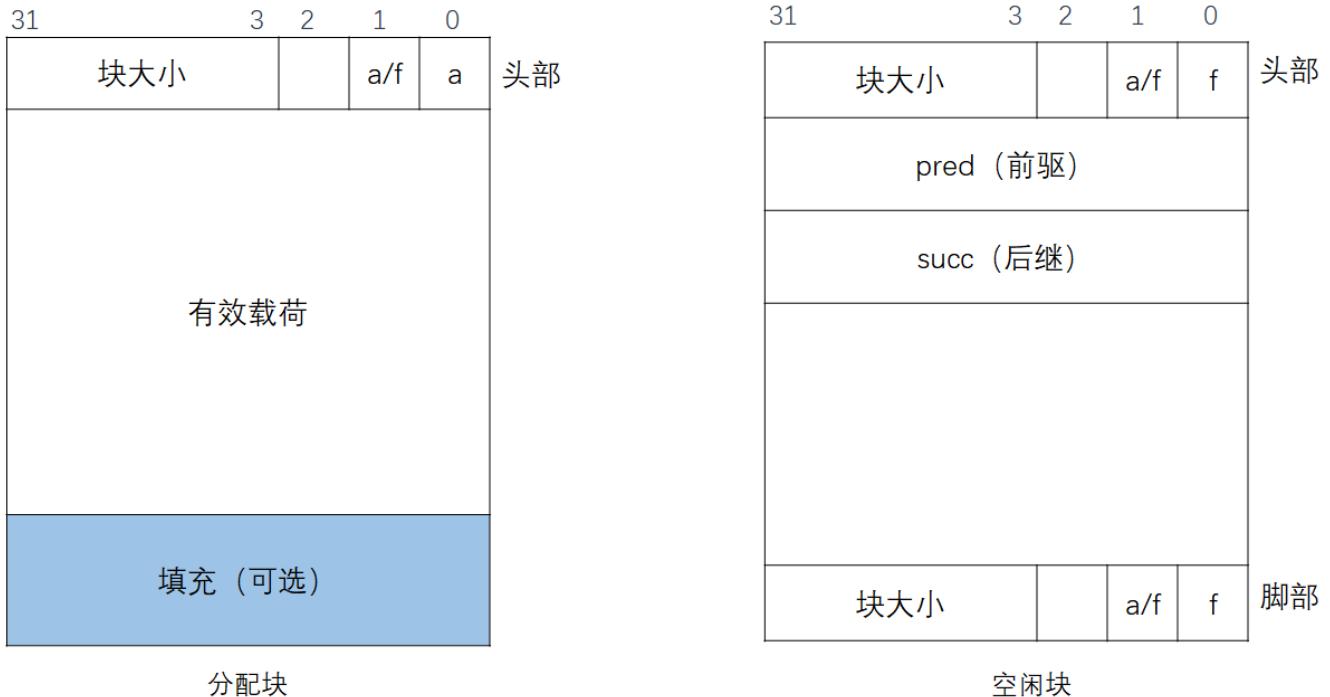
## 四、显式空闲链表

- 隐式空闲链表存在的问题
  - 隐式空闲链表为我们提供了一种简单的分配方式。但是，在隐式空闲链表方案中：块分配时间复杂度与堆中块的总数呈线性关系。这在实际中是不能接受的。

下面介绍一种由双向链表组织的显式空闲链表方案。

## 块的格式

实际实现中通常将空闲块组织成某种形式的显式数据结构（如，链表）。由于空闲块的空间是不用的，所以实现链表的指针可以存放在空闲块的主体里。例如，将堆组织成一个双向的空闲链表，在每个空闲块中，都包含一个 pred（前驱）和 succ（后继）指针，如下图所示：



对比隐式空闲链表，双向空闲链表的方式使首次适配的分配时间由块总数的线性时间减少到空闲块数量的线性时间，因为它不需要搜索整个堆，而只是需要搜索空闲链表即可。

由上图可以看到，与隐式空闲链表相比，分配块和空闲块的格式都有变化。

- 首先体现在分配块没有了脚部，这可以优化空间利用率。回想前面的介绍，当进行块合并时，只有当前块的前面邻居块是空闲的情况下，才会使用到前邻居块的脚部。如果我们把前面邻居块的已分配/空闲信息位保存在当前块头部中未使用的低位中（比如第1位中），那么已分配的块就不需要脚部了。但是，一定注意：空闲块仍然需要脚部，因为脚部需要在合并时用到。
- 其次，空闲块中多了pred（前驱）和succ（后继）指针。正是由于空闲块中多了这两个指针，再加上头部、脚部的大小，所以最小的块大小为4字节。

下面详细说明一下分配块和空闲块的格式

- 分配块：
  - 由头部、有效载荷部分、可选的填充部分组成。其中最重要的是头部的信息：
    - 头部大小为一个字(32 bits)，
    - 其中第3-31位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。
- 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。

## 空闲块的合并与分割

- 合并分割的思想与隐式空闲链表时的分析一致，只是在代码实现方式上不同。
- 注意：分割合并块时，一定要保证操作前和操作后所有块在空闲链表中不会互相覆盖。

## 五、实验要求

### 实验任务

#### 可选任务一：补全隐式空闲链表的实现

- 由于在上述讲解隐式空闲链表的过程中，已经提供并分析了隐式链表的代码实现，所以你只需要将其抄到mm.c文件中，并补全部分函数即可。待补全的函数如下：
  - `static void *find_fit(size_t asize);`
    - 针对某个内存分配请求，该函数在隐式空闲链表中执行首次适配搜索。
    - 参数asize表示请求块的大小。
    - 返回值为满足要求的空闲块的地址。若为NULL，表示当前堆块中没有满足要求的空闲块。
  - `static void place (void *bp, size_t asize);`
    - 该函数将请求块放置在空闲块的起始位置。只有当剩余部分大于等于最小块的大小时，才进行块分割。
    - 参数bp表示空闲块的地址。参数asize表示请求块的大小。
- 本次实验满分为10分，由于任务一很简单，所以选择该任务的最高得分为6分

#### 可选任务二：显式空闲链表的实现

- 选择该任务的最高分数为10分
- 实验要求：
  - 基于显式空闲链表实现块的分配和释放。
  - 分配块没有脚部。空闲块有脚部。
  - 必须实现块的合并与分裂。
  - 空闲链表采用后进先出的排序策略：将新释放的块放置在链表的开始处。
  - 适配方式采用首次适配。

## 代码运行

- 你只需要在mm.c中写入自己的代码即可，其他文件均为辅助文件，无需修改。
- trace目录下的文件为测试用例，后续可以用于测试。关于文件的功能介绍请阅读README文件。

使用如下命令编译运行程序：

```
#进入源码目录  
make #编译，执行后可以看到生成了一个名为mdriver的可执行文件
```

```

os@ubuntu:~/oslab/lab-3/malloclab-handout$ make ←
gcc -g -Wall -m32 -c -o mdriver.o mdriver.c
mdriver.c: In function ‘remove_range’:
mdriver.c:465:6: warning: variable ‘size’ set but not used [-Wunused-but-set-variable]
    int size;
    ^
gcc -g -Wall -m32 -c -o mm.o mm.c
gcc -g -Wall -m32 -c -o memlib.o memlib.c
gcc -g -Wall -m32 -c -o fsecs.o fsecs.c
gcc -g -Wall -m32 -c -o fcyc.o fcyc.c
gcc -g -Wall -m32 -c -o clock.o clock.c
gcc -g -Wall -m32 -c -o ftimer.o ftimer.c
gcc -g -Wall -m32 -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ftimer.o
os@ubuntu:~/oslab/lab-3/malloclab-handout$ ls ←
clock.c  fcyc.c  fsecs.o  mdriver  memlib.o          output.txt
clock.h   fcyc.h   ftimer.c  mdriver.c  mm.c           README
clock.o   fcyc.o   ftimer.h  mdriver.o  mm.h           short1-bal.rep
config.h  fsecs.c  ftimer.o  memlib.c  mm-implicit-free-list.c short2-bal.rep
core      fsecs.h   Makefile  memlib.h  mm.o           traces
os@ubuntu:~/oslab/lab-3/malloclab-handout$ █

```

#实验中，我们主要用到mdriver的如下功能：

```

./mdrive -h #显示帮助
./mdrive -f <file> #执行某个测试用例
./mdrive -v #执行所有测试用例，并显示结果
./mdrive -vv #执行所有测试用例，并显示详细信息

```

正确编译后，执行./mdriver -v的结果如下：

trace列表示测试用例的编号；valid列表示运行结果是否正确；util列表示空间利用率；ops列表示执行的总操作数量；secs列表示运行时间；Kops列表示每秒执行的千操作数（也就是throughput）。最后一行是得分，图中"4.66 (util) + 4.00 (thru) = 8.7/10"表示：满分10，得分8.7，其中空间利用率得分4.66，throughput得分4.00。

(注意：如果有用例测试没通过，即某行trace的valid列值为"no"，则为0分)

```

os@ubuntu:~/oslab/lab-3/malloclab-stu$ ./mdriver -v
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid   util     ops      secs    Kops
      0     yes   86%    5694  0.000978  5823
      1     yes   55%   12000  0.003752  3198
      2     yes   51%   24000  0.003339  7188
      3     yes   90%    5848  0.000514  11371
      4     yes   66%   14400  0.000477  30214
      5     yes   93%    6648  0.000317  20991
      6     yes   95%    5380  0.000330  16318
      7     yes   83%    4800  0.000783  6128
      8     yes   80%    4800  0.000811  5916
Total                  78%   83570  0.011301  7395

Perf index = 4.66 (util) + 4.00 (thru) = 8.7/10

```

## 评分标准

- 得分由两部分构成：代码运行得分，回答问题得分。
  - 代码运行得分即上一部分讲的使用`./mdriver -v`得到的分数。（存在bug或存在用例测试不通过，则代码得分为0）
  - 回答问题得分是指：针对代码实现，助教提问问题，根据回答情况在代码分的基础上加分/扣分。

## 提交方式

- 将实验报告、源代码打包为压缩包提交。
  - 本次实验只要求修改`mm.c`文件，不能修改其他文件，所以源代码只提交`mm.c`文件即可。
- 提交至邮箱：[ustc\\_os2019@163.com](mailto:ustc_os2019@163.com)
  - 邮件主题、文件名称、压缩包名称**均采用以下格式命名
    - x-学号-姓名（x：代表第x次实验，本次为第3次实验）
      - 例如张三的第3次实验命名为“3-PB17011010-张三”
      - 未按照规范命名的邮件会被忽略、删除
- 实验验收和报告提交截止日期请关注课程主页。

## 六、参考资料

---

- 《Computer Systems: A Programmer's Perspective 3rd》