堆内存管理器设计与实现

一、堆的地址空间划分

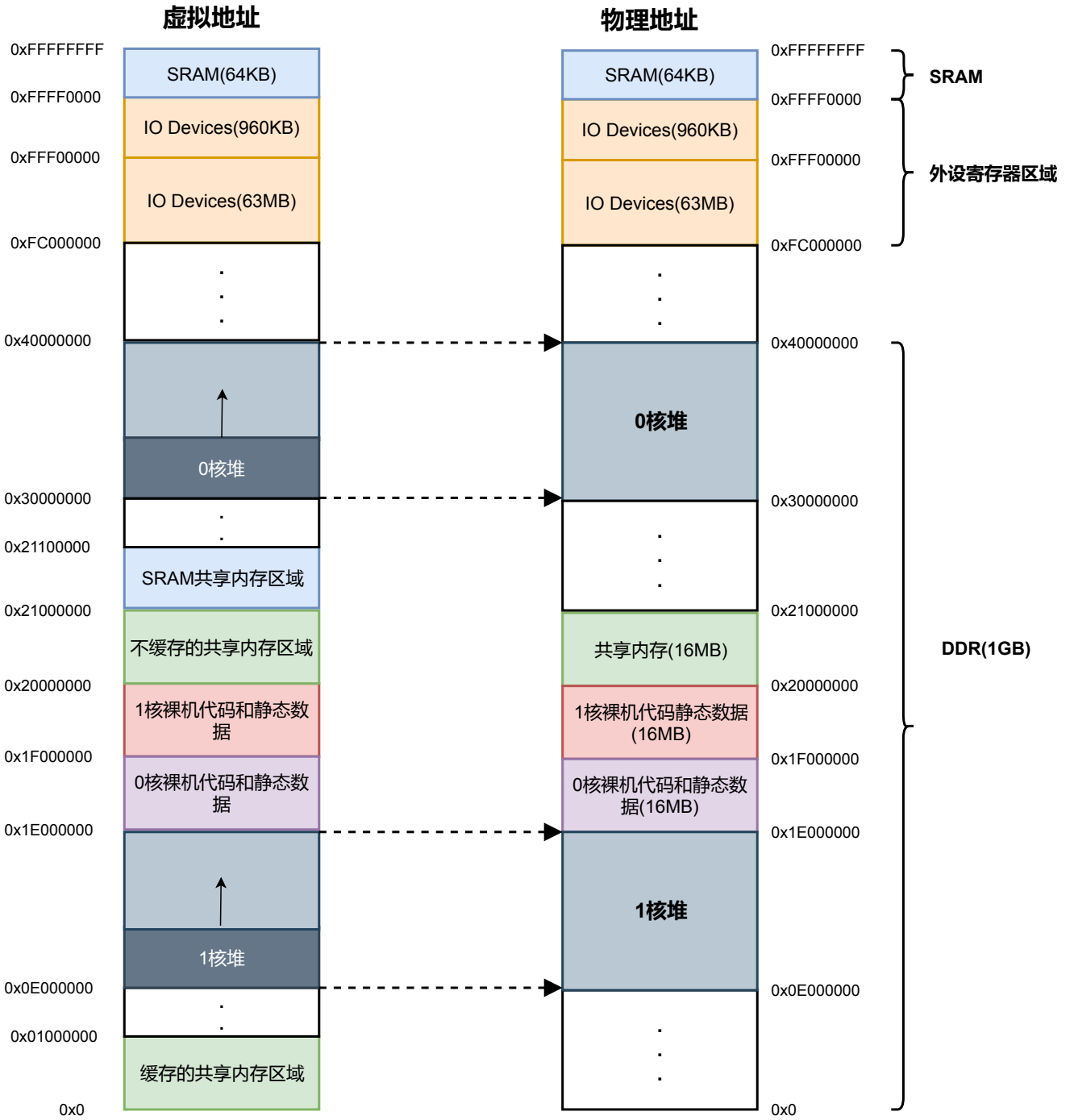


图1 堆的划分

如上图所示，上图为堆的映射划分。0核和1核分别分配256M的堆空间，虚拟堆空间和物理堆空间地址一一对应，0核堆空间为0x30000000$_{0x40000000}$，1核堆空间为0x0E000000$_{0x1E000000}$。其中深色部分是在使用的堆空间，包括已分配块和释放的空闲块，如果当前使用的堆空间没有足够大的空闲块容纳新的分配需求，则内存管理器将堆往上增长。

二、用户接口说明

对于在操作系统上运行的应用程序的堆的使用，最基本也是最常用的是malloc和free函数，因此在本方案的堆内存管理器实现中就是以实现安全可靠，高效的malloc和free接口为主要目标。

malloc、calloc和realloc函数的原型:

```
void *malloc(size_t size);
                参数：size是分配的内存块大小
                返回：如果成功返回新分配的内存块的首地址；
                      如果失败返回NULL

void *calloc (size_t count, size_t nbytes);
                参数：count是分配的元素个数
                      nbytes表示每个元素多少字节
                返回：如果成功返回新分配的内存块的首地址，该内存块内容全为0；
                      如果失败返回NULL

void *realloc(void *ptr, size_t size);
                参数：ptr是之前通过malloc或calloc分配的内存块的首地址
                      size是变化后的内存块大小
                返回：如果成功返回大小变化成size后的内存块首地址
                      如果失败返回NULL
```

malloc函数成功时返回一个void *指针指向至少size字节的内存块，该地址8字节对齐。malloc并没有初始化所分配到的内存块，因此如果想分配初始化的内存块，可以使用包装malloc的calloc函数，calloc在利用malloc分配内存块后再将该内存块清零。同时如果应用程序想改变之前分配的内存块的大小可以使用realloc。

free函数原型：

```
void free(void *ptr);
                参数：ptr是之前分配的内存块的首地址
                返回：无
```

free函数将之前通过malloc或者calloc、realloc分配的内存块释放，从而可以让该内存块在后面被重新使用。

sbrk函数原型：

```
void *sbrk(s32 incr);
                参数：incr是额外需要申请的堆空间大小
                返回：如果成功返回原先的可用堆的末尾地址brk
                      如果失败返回NULL
```

sbrk函数一般是在分配接口的实现中所使用，一般用户不会自己使用，也不建议用户使用。当分配堆的接口如malloc发现当前可用的堆空间无法满足用户的分配需求时，就会通过sbrk额外申请一片堆空间，通过将指向可用堆末尾地址的指针brk增加incr(会有对齐和大小限制要求)字节来表示申请额外incr的堆空间。如果新的brk没有超过最大堆地址则成功。

在本文的分配器实现中，分配器的函数都用mm_作为前缀，比如mm_malloc对应的是malloc函数。因此为了让用户像在操作系统上一样通过不用前缀的函数来使用堆，在分配器的头文件mm.h中定义了导出给用户使用的接口函数的别名宏。通过这些宏定义我们就可以在裸机程序中直接使用没有前缀的函数名。

```
#define malloc mm_malloc
#define calloc mm_calloc
#define realloc mm_realloc
#define free mm_free
```

三、底层原理与实现

1、一个简单的展示

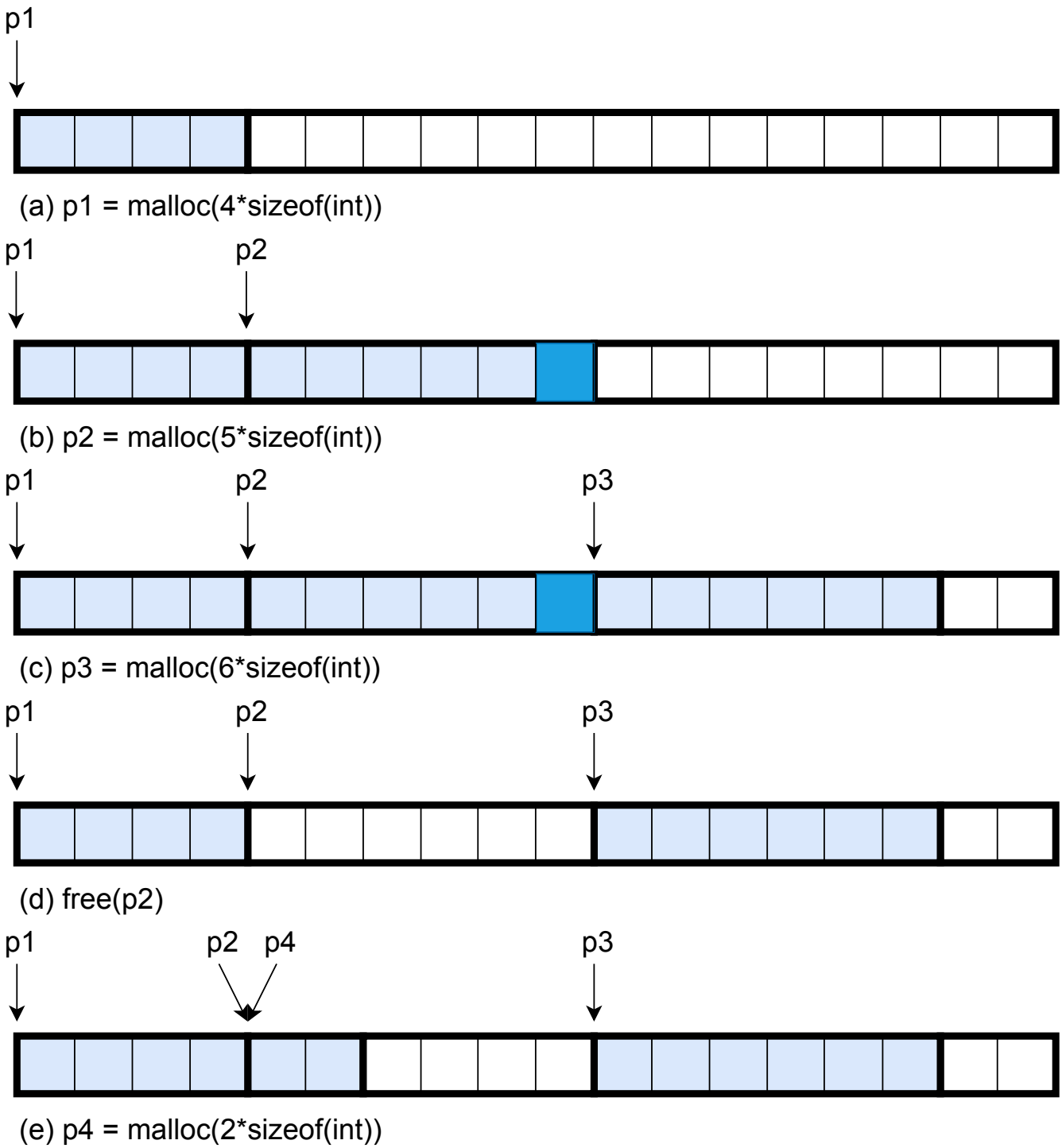


图2 malloc和free展示

上图展示了一个通过malloc和free来管理一个18字大小的堆的简单实现。每个小格子代表4字节的字，浅蓝色部分代表已分配的内存块，白色部分代表空闲的内存块。刚开始时堆只有一个双字对齐的16字大小的空闲内存块。以下分别展示了5个内存分配或者释放后的堆的变化：

(a)程序申请一个4字大小的内存块。通过malloc申请到了该空闲块的前4字空间，并返回第一个字的地址p1。
(b)程序申请一个5字大小的内存块。通过malloc申请到了剩下的空闲块的前6字空间，多的一个字是为了双字对齐 的要求，并返回第一个字的地址p2。
(c)程序申请一个6字大小的内存块。通过malloc申请到了剩下的空闲块的前6字空间，并返回第一个字的地址p3。
(d)程序释放(b)中申请的内存块。通过free将p2指向的已分配的6字内存块释放，该块内存重新变为空闲内存。

(e)程序申请一个2字大小的内存块。通过malloc申请到了(d)中释放得到的空闲块的前2字空间，并返回第一个字的 地址p4。

2、一些问题

通过上一小节展示了一个通过malloc和free使用堆时堆的已分配内存和空闲内存的简单变化过程。**但问题是如何记录堆里面的分配块和空闲块，如何让应用程序申请内存时更快速找到更合适的空闲块。**

另外还有一个称为碎片的问题。有内部碎片和外部碎片两种形式。**内部碎片**产生的主要原因有两个：一是分配到的实际内存块大于申请的大小。比如图2(b)中程序申请的是5字的空间，但是由于分配器的双字对齐的限制实际上分配的是6字的空间，多出一个深蓝色表示的字，这就产生了一个字的内部碎片；另外一个是有可能内存分配会强制一个分配的内存块的最小约束，如果申请的内存小于最小约束也会产生内部碎片。**外部碎片**意思是堆中的所有空闲块的总大小满足申请的需求，但是没有一个完整的空闲块满足申请需求。比如图2(d)中如果要申请8字的内存空间，虽然堆中的两块空闲块加起来有8字大小，但却不是连续的完整内存块，因此内存管理器不得不额外申请一片内存来应对新的申请需求。**控制碎片可以从两方面考虑**：一是应用程序尽量申请对齐的内存块，减少内部碎片。第二个方面就要靠堆内存分配器的设计了，下一小节就来讲解如何实现一个能有效减少碎片的堆内存分配器。

3、堆内存分配器的设计与实现

3.1、设计目标

通过上一小节的分析明确了**堆内存分配器设计的目标：**

(1) 记录所有堆中的已分配内存块和空闲块。

(2) 应用程序申请内存时更快速找到更合适的空闲块。

(3) 减少碎片化。

根据以上目标我们需要考虑以下几个方面的技术实现:

块组织：如何在堆中组织分配快和空闲块？如何跟踪空闲块？

放置分配块：如何选择一个合适的空闲块放置申请的内存块？

分割空闲块：当一个新的分配内存块被放置在一个空闲内存块里之后，该如何处理剩下的空闲块？

合并空闲块：如何处理一个刚刚释放的空闲块？

3.2、块组织：隐式空闲链表

3.2.1、块中只含头部不含尾部

一个分配器首先要考虑的是如何区分已分配块和空闲块以及它们的起始地址和末尾地址，大多数分配器都将这些信息嵌入到块本身。图3展示了一个简单的块格式。

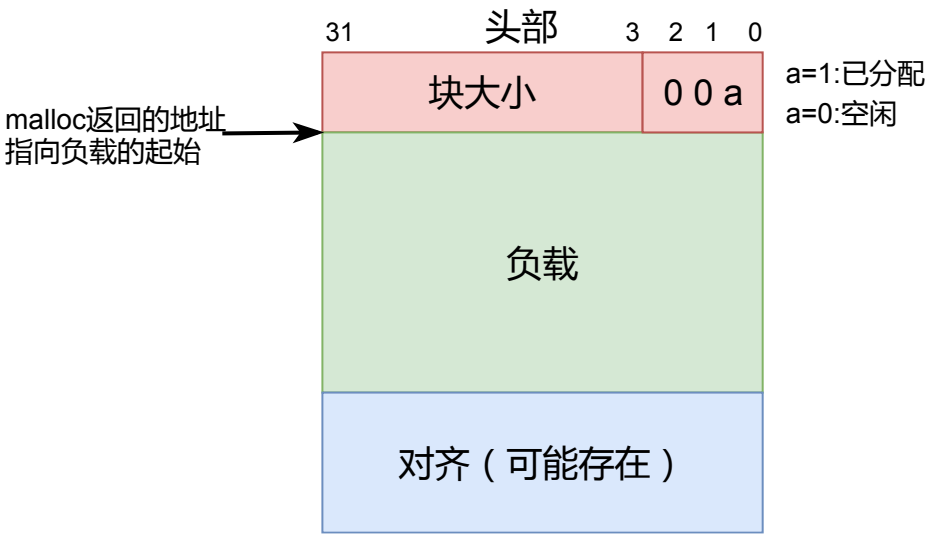


图3 只包含头部的块格式

在上图中，一个块(block)包含了一个一字大小的头部(header)、一个称为负载(payload)的实际分配使用的空间还有可能有一些对齐(padding)区域。头部中的块大小是包快这三部分的整个块的大小，它有双字即8字节的对齐要求因此不需要头部前三位，而这头部前三位刚好可以用来存放区分已分配快和空闲块的信息，确切的说是第0位，另外两位始终为0，当第0位为1时表示已分配块，当为0时表示空闲块。通过分配接口如malloc分配成功后返回的地址是负载的首地址，该地址需要8字节对齐。通过上图的块格式来看一下称为隐式空闲链表的堆组织方式：

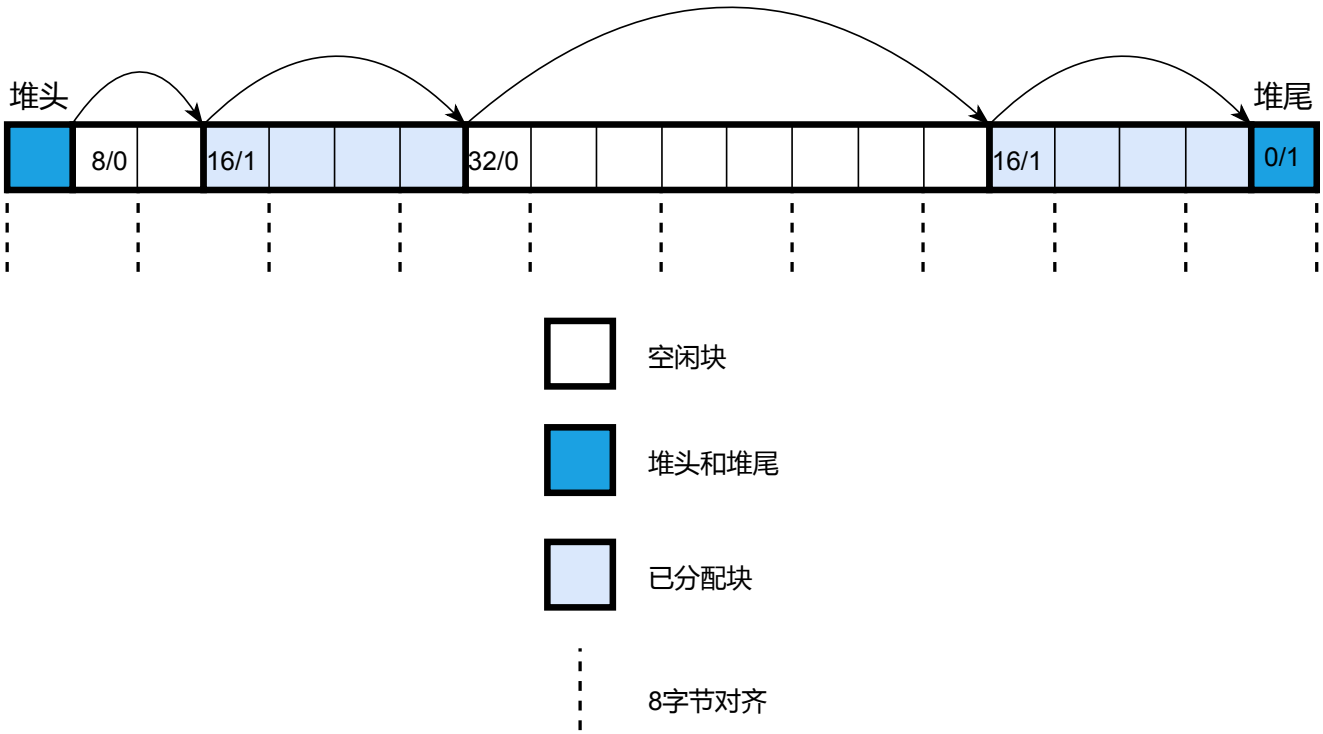


图4 块只含头部的隐式空闲链表

上图所展示的是1个8字节的空闲块、1个16字节的已分配块、1个32字节的空闲块、1个16字节的已分配块还有堆头堆尾块所组成的隐式空闲链表。正如图3格式所展示一样，每个块的头部包含块大小和分配信息，通过块大小就能知道下一个块的头部地址，不需要显式的地址来指向，因此叫做**隐式空闲链表**。必要时通过在块中添加对齐区域使得每个块的负载起始地址都8字节对齐。另外需要特别注意的是堆头和堆尾，这两个是特殊的块。其中堆头里面数据任意，因为分配器会有一个指针指向堆头，推头存在的意义就是对齐。而堆尾是一个只包含块头部的特殊块，并且块大小信息是0，分配标记1，不会与其他任何已分配块或者空闲块的头部信息冲突。因此可以利用这个信息来处理一些

边界情况，比如用户申请空间时当查找到堆尾说明没有合适的空闲块满足要求，这时分配器再额外分配堆空间，将原来的堆空间延长。

3.2.2、块中同时包含头部和尾部

在上一小节所展示的块格式的隐式空闲链表虽然能记录所有的块信息，并且能找到所有的块，但是有个明显的缺点是通过当前块只能立即获取下一块的信息，无法立即获取前一块的信息，这无论对于碎片化处理还是空闲块查找算法都是巨大的阻碍。大师高纳德(Donald Knuth)提出了一种替代方法，如下图：

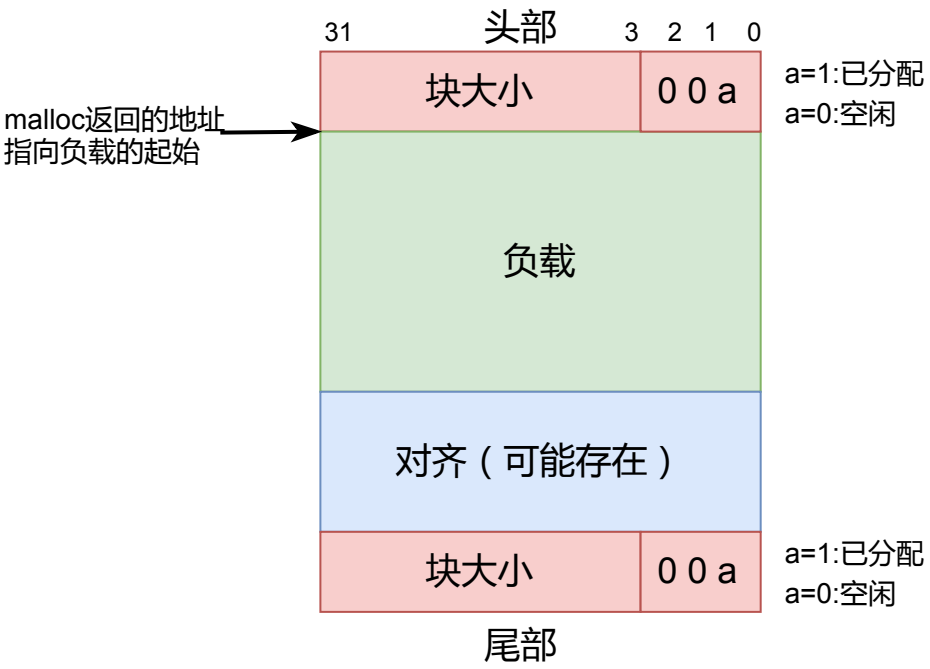


图5 包含头部和尾部的块格式

和上一小节的块格式相比这里只多出了一个和头部一摸一样的尾部信息，但是这个尾部信息可以为下一块提供该块的定位信息，因为当前块的头部刚好是上一块的尾部的接下来一个字。因此可以通过一个块的头部找到下一块的首地址也可以通过一个块的头部上方的上一块的尾部找到上一块的首地址。

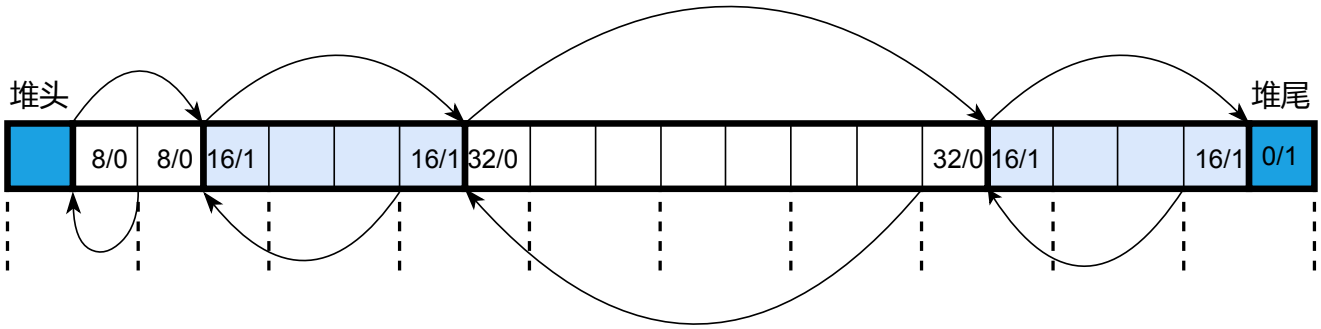


图6 块同时含头部和尾部的隐式空闲链表

图6是使用新的块格式后的空闲链表。从图中可以看出相对于之前没有尾部信息的空闲链表，这种方式的优点是能向前查找，而缺点是每一块多用了一个字的空间来给尾部保存块信息。这里优点的效益远大于缺点产生的空间损失，因为在实际应用程序中往往不会分配过于小的内存块，因此由于额外的尾部消耗的内存不会过大。而由于在块中提供尾部信息使得无论是对于块查找算法优化还是对于后文所讲的碎片合并都有极大的帮助。

3.3、放置分配块

当应用程序申请一块k字节的内存块时，分配器搜索堆链表来找到一个包含至少k字节的负载的空闲块。分配器的这种搜索空闲块的策略有**第一次适配、下一次适配和最佳适配。**

第一次适配是从堆链表起始开始搜索来找到第一个满足要求的空闲块。下一次适配和第一次适配相似，只不过不是从堆链表起始开始搜索，而是从上一次分配到的块为起始开始搜索。最佳适配则检查所有空闲块来选择一个满足要求的最小空闲块。

第一次适配的优点是具有将大空闲块保留在堆链表后部分的趋势，而缺点是在堆链表前部分会留下过多的小空闲块，这往往会增加大空闲块的搜索时间。下一次适配是由高纳德提出来作为一种替代第一次适配的方法，该算法从上次分配到的块开始搜索，解决第一次适配的不足，通常下一次适配搜索会比第一次适配快的多。而最佳适配虽然在碎片管理上有绝对的优势，但是在本文所用的空闲链表中搜索所有的空闲块太过耗时，需要另外一些比较复杂的数据结构和算法的优化，本文不予考虑。综上所述**本文采用下一次适配**算法来处理搜索空闲块来放置分配块。

3.4、分割空闲块

当分配器已经找到了可以放置分配块的空闲块后，此时要决定分配空闲块里的多少空间的策略。最简单的选择是使用空闲块的所有空间，尽管这种方法速度最快但是会产生很多内部碎片。更合适的方法是在满足对齐和最小块要求的情况下按需分配，剩下的空闲空间再分割出来成为一个独立的空闲块。

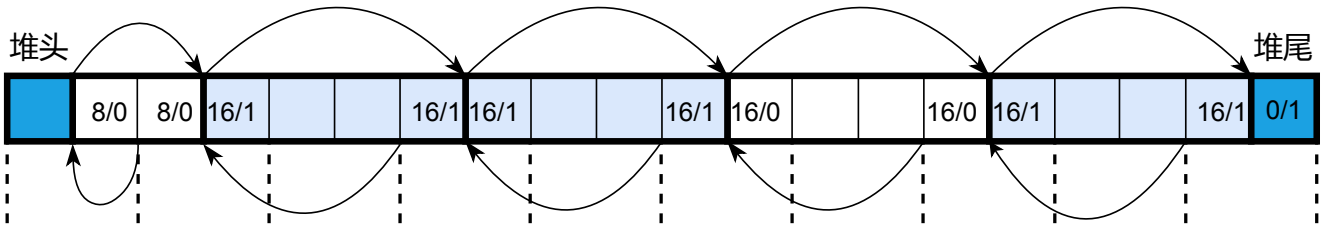


图7 分割空闲块

图7所示的是在分配器将图6中的32字节的空闲块前16字节用来放置分配块，余下的16字节重新被分割出去成为一个新的空闲块。

3.5、合并空闲块

当分配器释放了一个分配块后，有可能邻近块是空闲，因此会产生假碎片的现象。比如，图7中新分配的16字节块释放后的空闲链表如图8所示。此时如果申请四字负载的内存块则会失败，就算这两个连续的空闲块总量足够。

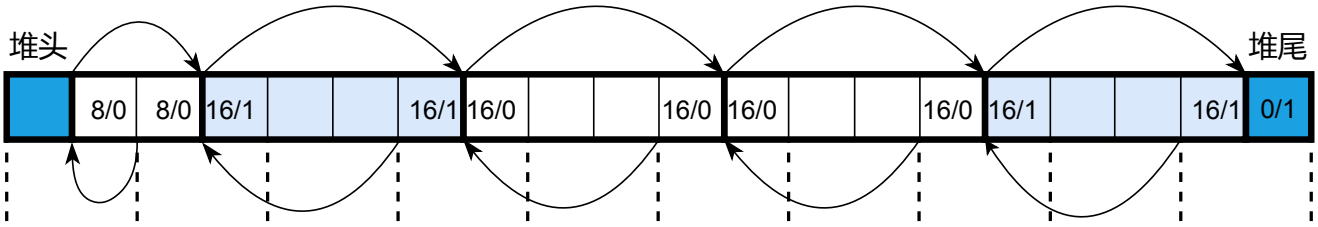


图8 假碎片

为了克服假碎片的问题，分配器需要在释放分配块后将它与邻近的空闲块合并。而利用同时具有头部和尾部的块格式就能简单和优雅地解决这个问题。当释放当前分配块时会有**如下4种情况：**

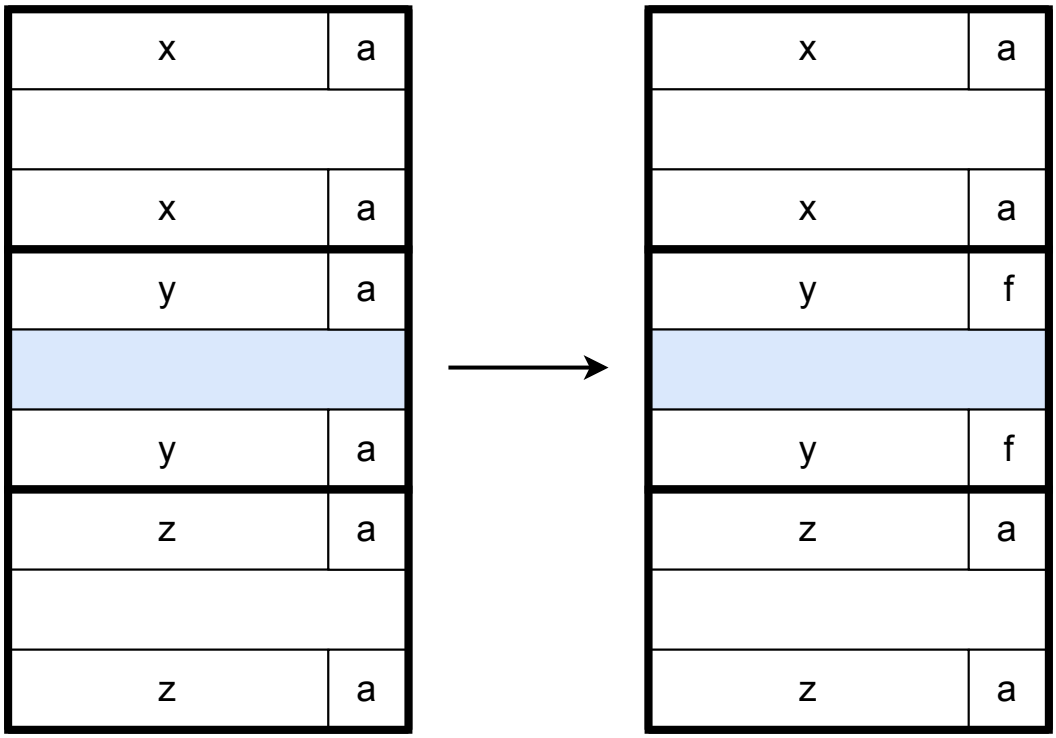
情况1：前一块和后一块都是已分配块。

情况2：前一块是已分配块，后一块是空闲块。

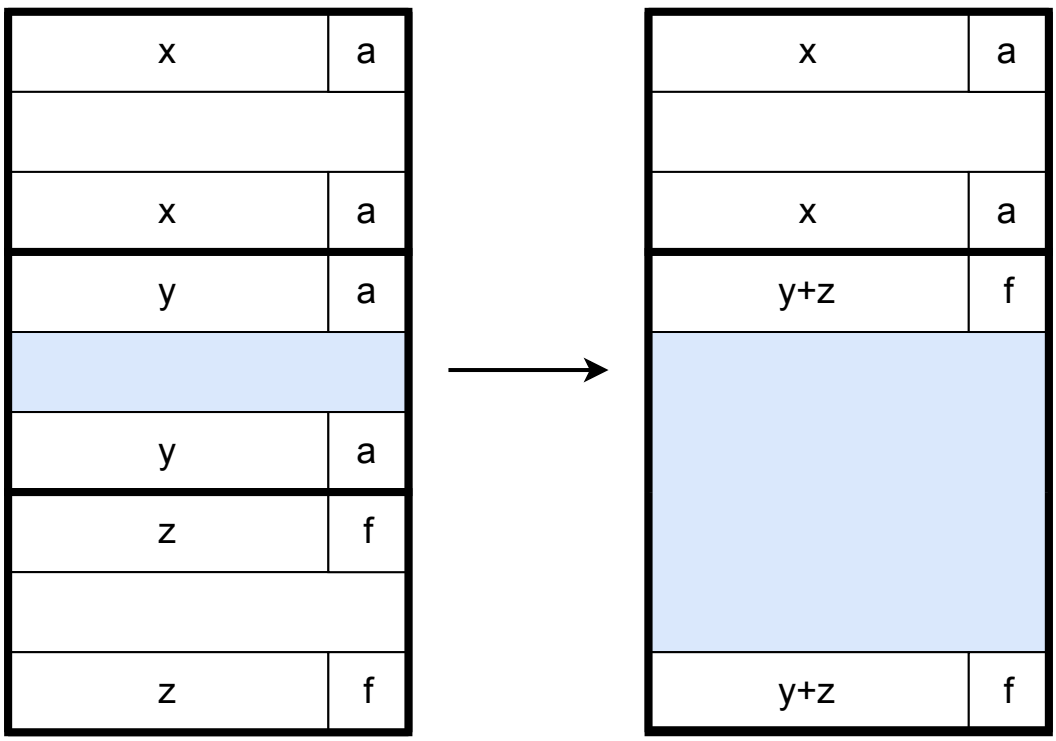
情况3：前一块是空闲块，后一块是已分配块。

情况4：前一块和后一块都是空闲块。

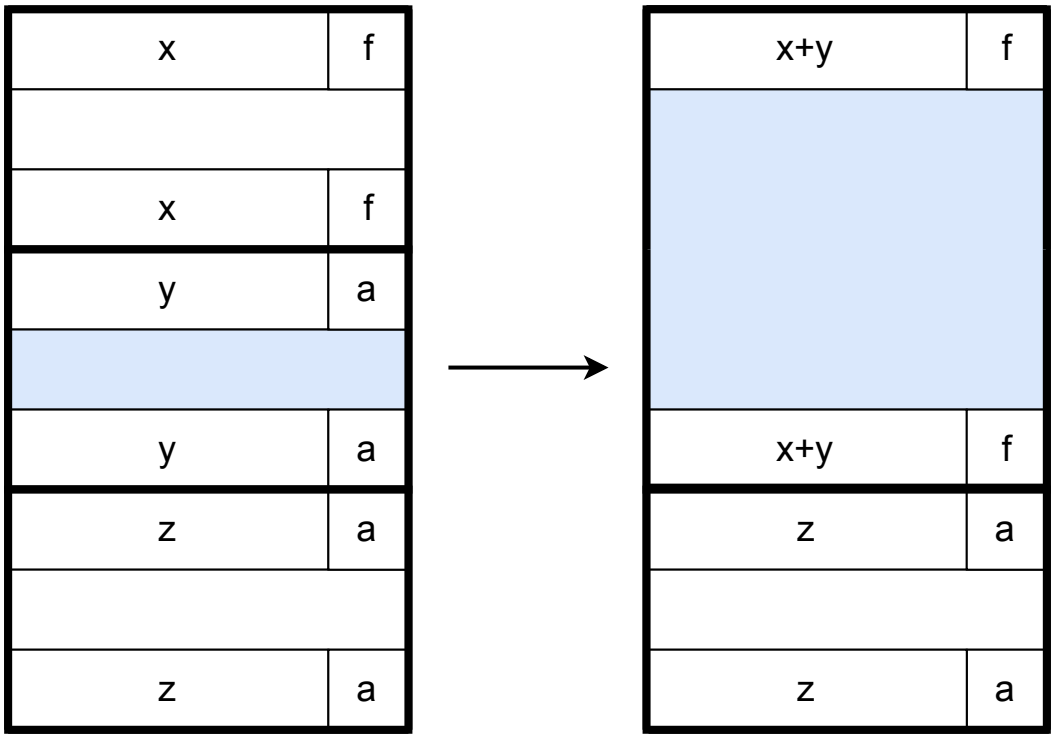
下面就来展示这4种情况的合并过程：



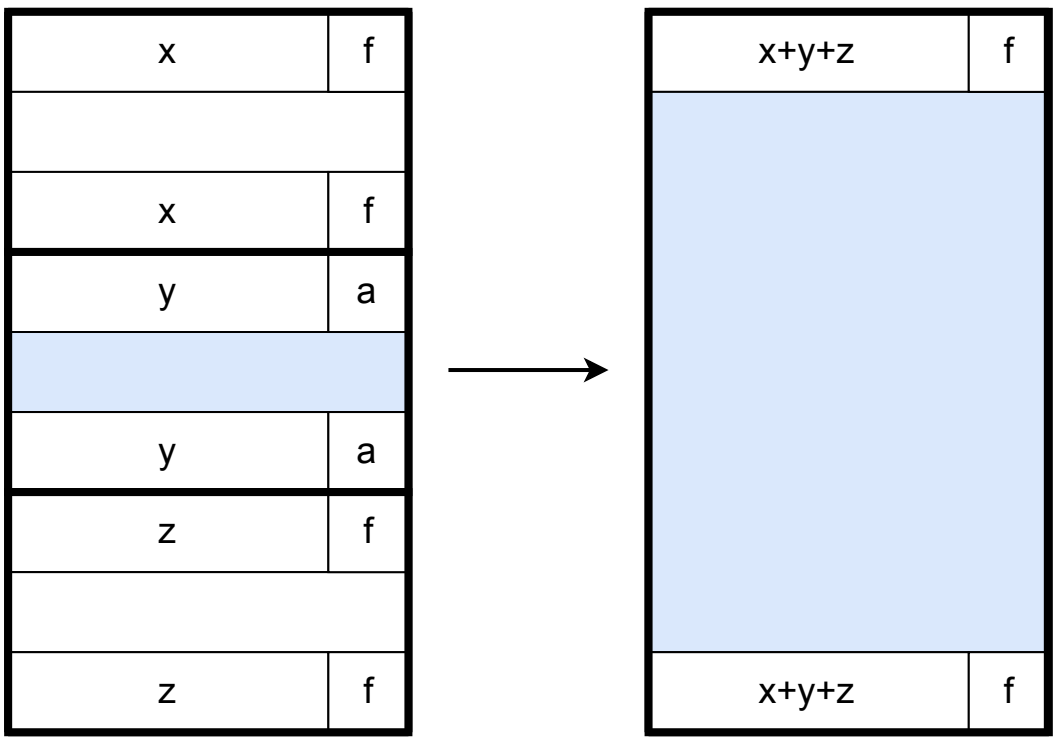
情况1



情况2



情况3



情况4

图9

上面详细地展现了所有可能的合并情况，左边负载区域标记为蓝色的块为当前需要释放的块，x,y,z表示块大小，a表示该块是已分配块，f表示该块是空闲块。

3.6、分配器代码实现

3.6.1、初始化堆内存地址空间

在初始化分配器之前需要先初始化堆内存地址空间，这个过程在memlib.c中实现。在memlib.c中定义了几个描述堆的全局变量和初始化函数。

```
---------------------------------------memlib.c-----------------------------------------
char *mem_heap;     /* 指向堆的第一个字节 */
char *mem_brk;      /* 指向当前堆的最后一个字节的后面一个字节 */
char *mem_max_addr; /* 指向最大堆的最后一个字节的后面一个字节 */

#define CPU0_START_HEAP 0x30000000      /* 0核堆的mem_heap */
#define CPU0_MAX_HEAP   0x40000000      /* 0核堆的mem_max_addr */

#define CPU1_START_HEAP 0x0E000000      /* 1核堆的mem_heap */
#define CPU1_MAX_HEAP   0x1E000000      /* 1核堆的mem_max_addr */

#define MAX_HEAP 0x10000000 /* 256M，0核和1核的最大堆空间 */

/*
 * 初始化堆内存地址空间
 */
void mem_init(void)
{
#ifdef CPU0
    mem_heap = (char *)CPU0_START_HEAP;
#else
    mem_heap = (char *)CPU1_START_HEAP;
#endif
    mem_brk = (char *)mem_heap;
    mem_max_addr = (char *)(mem_heap + MAX_HEAP);
}

/*
* sbrk函数的替代品，延伸当前堆incr字节
* 即将mem_brk增加incr字节
* 返回之前的mem_brk,即新分配的堆空间的第一字节地址
*/
void *mem_sbrk(int incr)
{
    char *old_brk = mem_brk;
    if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
        uartprintf("ERROR: mem_sbrk failed. Ran out of memory...\n");
        return NULL;
    }
    mem_brk += incr;
    return (void *)old_brk;
}
```

上述代码列出了初始化堆内存系统涉及的主要全局变量和函数，它们的含义和作用在代码注释中描述的很清楚。需要注意的是这些变量和函数设计的目的是为了给分配器使用，而不是由用户直接使用。

3.6.2、分配器空闲链表的精确定义

本文分配器的堆结构组织就是采用上文的同时具有头部和尾部的块格式所组成的隐式空闲链表，不过有些地方需要更严格精确地定义。

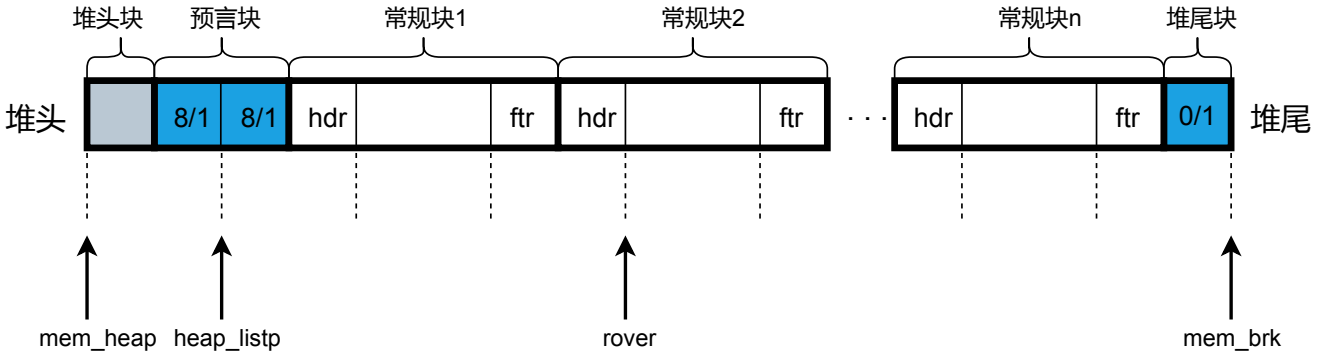


图10 最终空闲链表格式

上图所展示的是本文分配器设计所采用的最终空闲链表格式，它保持了前文所描述的空闲链表的绝大多数性质。下面详细列出它的各子模块属性：

堆头块：未使用，数据任意，它存在的目的是保持堆系统的对齐。

预言块：一个特殊的块，真正用来标记堆的开头的块，只有一个头部和一个尾部组成，标记为已分配，没有负载。

常规块：也就是已分配块和空闲块，hdr是头部，ftr是尾部，最小块大小是16字节，即负载最小8字节，同时负载大小也是8字节对齐。

堆尾块：另一个特殊的块，用来标记堆的结尾的块，只有一个头部没有尾部，块大小标记为0，标记为已分配。

根据以上描述的性质和限制可知预言块和堆尾块不会和任一常规块相同，因此能够很好地区分，为堆的搜索提供了边界条件，并且分配位标记为1避免了分配器在分配内存时当做空闲块分配出去，也避免了与相邻空闲块的合并。除此之外还有两个定义在分配器实现模块mm.c中的静态全局变量heap_listp和rover，heap_listp总是指向预言块，而rover会跟踪新分配的块。

3.6.3、操作空闲链表的基本常量和宏

在分配器的代码实现中会经常有很多对空闲链表的操作，为了代码的可读性与操作的简便，将这些常用的数据和操作定义宏。图11展示了分配器经常使用的基本的常量与宏。

1-3行定义三个基本的长度常量：一个字的字节数(WSIZE)、双字的字节数(DSIZE)，以及拓展堆时默认使用的长度CHUNKSIZE，它同时还是初始化堆空闲链表时唯一的一个空闲块的长度。第8行的PACK宏将8字节对齐的块大小size和分配位alloc打包成一个字。第10行的GET宏读取并返回参数p为地址的一个字的内容，第11行将参数val做为值的一个字写到参数地址p处。第15和16行的GET_SIZE和GET_ALLOC分别读取并返回块头部或者尾部地址p处的块长度和分配位。第19和20行的HDRP和FTRP根据给出的块的负载首地址分别计算出块的头部和尾部地址。第23和24行的NEXT_BLKP和PREV_BLKP则根据给出的块的负载首地址分别计算出下一块和前一块的负载首地址。

```
----------------------------------------mm.c-----------------------------------------------
1 #define WSIZE       4       /* 一个字包含4字节，每个块的头部和尾部都是一个字 */
2 #define DSIZE       8       /* 双字包含8字节 */
3 #define CHUNKSIZE  (1<<12)  /* 4KB,拓展堆时默认使用的长度，单位字节 */
4
5 #define MAX(x, y) ((x) > (y)? (x) : (y))
6
7 /* 将块长度和分配位打包进一个字 */
8 #define PACK(size, alloc)  ((size) | (alloc))
9
10 /* 读和写地址p的一个字 */
11 #define GET(p)       (*(unsigned int *)(p))
12 #define PUT(p, val)  (*(unsigned int *)(p) = (val))
13
```

```
/* 读取块头部或尾部地址p处额的块长度和分配位 */
#define GET_SIZE(p)  (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

/* 给出块的负载首地址bp，分别计算出块头部和尾部地址 */
#define HDRP(bp)       ((char *)(bp) - WSIZE)
#define FTRP(bp)       ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

/* 给出块的负载首地址bp，分别计算出下一块和前一块的负载首地址 */
#define NEXT_BLKP(bp)  ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp)  ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```

图11 操作空闲链表的基本常量和宏

3.6.4、创建和初始化堆

在调用mm_malloc或者mm_free之前，应用程序必须先调用mm_init来创建一个初始化的堆。mm_init首先初始化堆的地址空间，然后将mem_heap为起始的四个字的空间初始化为堆头块、预言块和堆尾块，然后初始化list_listp和rover，最后在这个空堆的基础上通过extend_heap增加一个CHUNKSIZE字节的空闲块来完成初始化工作。

```
----------------------------------------mm.c-------------------------------------------------
/*
 * 创建并初始化堆
 */
int mm_init(void)
{
    /* 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
    mem_brk = heap_listp + (2*WSIZE);            // mem_brk初始化

#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 (extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}
```

图12 mm_init 创建并初始化堆

extend_heap函数在两种情况下会使用，一种就是上面的初始化堆时使用，另一种是在mm_malloc分配内存块没有找到足够大的空闲块时使用。该函数只会分配8字节对齐的空闲块空间，它会将原来的堆尾块初始化为新的空闲块的头部，在新的空间末尾初始化新空闲块尾部和堆尾块。最后将这个新的空闲块合并前一块（如果它也是空闲块），将合并后得到的空闲块的负载首地址返回。

```
----------------------------------------mm.c-----------------------------------------------
/*
 * 延长堆并返回新空闲块地址
 */
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

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

图13 extend_heap通过一个新空闲块拓展堆

3.6.5、分配内存块与放置分配块

用户在通过mm_init来创建一个初始化的堆后接下来可以使用mm_malloc来申请堆内存。基于对用户使用的考虑，我们在mm_malloc中首先判断堆是否已初始化，如果没有初始化则自动初始化而不需要用户手动初始化。然后将所要申请的字节数size调整后得到一个最小16字节并8字节对齐的asize，在空闲链表中利用下一次适配算法搜索到第一个负载至少为asize的空闲块，如果找到就放置asize分配块，如果没找到就申请一块额外的堆并放置分配块，最后返回放置后的分配块的负载首地址bp给用户使用。

```
----------------------------------------mm.c-----------------------------------------------
/*
 * 分配一个负载至少为size字节的内存块，返回分配到的块的负载首地址
 */
void *mm_malloc(size_t size)
{
    size_t asize;       // 将size调整后的整个块的字节数，包括头部和尾部，而size只是负载
    size_t extendsize; // 如果没找到合适的空闲块则新分配extendsize字节的堆
    char *bp;           // 块指针，指向负载的首地址

    /* 如果用户没有初始化堆，则初始化堆 */
    if (heap_listp == NULL) {
        mm_init();
    }
```

```
    /* 忽略非法申请 */
    if (size == 0)
        return NULL;

     /* 将size加上块头和块尾后并对齐得到最小16字节，8字节对齐的块大小asize字节 */
    if (size <= DSIZE)
        asize = 2*DSIZE;
    else
        asize = DSIZE * ((size + DSIZE-1) / DSIZE) + DSIZE;

    /* 在空闲链表中利用下一次适配算法搜索到第一个负载至少为asize的空闲块 */
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);         //以bp为负载首地址，asize为块大小放置分配块
        return bp;                //返回bp给用户使用
    }

    /* 如果没找到合适的空闲块，那就申请一块额外的堆并放置asize的分配块 */
    extendsize = MAX(asize,CHUNKSIZE);      // 额外申请的堆大小
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
        return NULL;
    place(bp, asize);            //以bp为负载首地址，asize为块大小放置分配块
    return bp;                   //返回bp给用户使用
}
```

图14 分配内存块

接下来讲一下搜索空闲块的实现find_fit。在下面的代码中同时实现了第一次适配和下一次适配搜索，通过宏来决定使用哪种，文本分配器会定义NEXT_FIT宏，代表下一次适配。这里只讲下一次适配过程，第一次适配非常简单不再赘述。首先用一个rover变量来追踪堆的分配，会随着mm_malloc和mm_free的调用而移动，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，也可能是调用mm_free释放后得到空闲块再经过合并后得到的最终空闲块的负载首地址。然后开始从rover停留的地方搜索空闲块，每次搜索rover都会移动，直到最终找到合适的空闲块，此时rover会更新为该空闲块的负载首地址，下一次搜索就接着从这里开始。搜索流程主要为两步：第一步从oldrover搜索到链表结尾来找到第一个满足要求的空闲块，如果第一步不成功才会有第二步，从链表开头搜索到oldrover来找到第一个满足要求的空闲块。如果第二步也没有成功则返回NULL告知mm_malloc，mm_malloc会通过extend_heap来额外申请一片足够容纳asize的堆，最后在新的空闲块中放置该asize大小的分配块，返回负载地址给用户使用。

```
------------------------------------mm.c-----------------------------------------------
/*
 * 找到一个至少为asize字节大小的空闲块
 */
static void *find_fit(size_t asize)
{
#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
}
```

图15 搜索空闲块

放置分配块并分割余下的空闲块的实现如图16的place函数所示。place函数首先根据传进的被放置分配块的空闲块负载指针bp来获取该空闲块的块大小，然后和分配块大小asize比较来决定是否要继续分割。如果要继续分割，则初始化新分配块和新空闲块的头部和尾部。如果不分割，则只是将原来的空闲块的头部和尾部的分配位设置为1表示已分配。

```
---------------------------------------mm.c-------------------------------------------------
/*
 * 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
 */
static void place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));   //获取空闲块长度

    /* 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
    }
}
```

图16 放置分配块与分割空闲块

3.6.6、释放内存块与合并空闲块

用户通过mm_free释放mm_mallloc申请到的内存块。mm_free通过将bp指针指定的分配块释放变为空闲块然后与相邻空闲块合并，最后将合并后的得到的空闲块负载地址赋给rover。图17展示了释放与合并过程。

```
----------------------------------------mm.c-----------------------------------------------
/*
 * 释放一个bp指定负载的内存块，
 * 这个要释放的块必须是通过mm_malloc或者
 * mm_malloc的包装函数调用得到的，否则会破坏内存数据
 */
void mm_free(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));   //获取所要释放的块的大小
    PUT(HDRP(bp), PACK(size, 0));       //将块头部分配位置0
    PUT(FTRP(bp), PACK(size, 0));       //将块尾部分配位置0
    coalesce(bp);                       //将释放得到的空闲块与可能存在的相邻空闲块合并
}

/*
 * 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 */
        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 */
        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 */
        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;
}
```

图17 释放内存块与合并空闲块

释放过程由两部分组成。第一部分比较简单，只是将需要释放的块的头部和尾部的分配位置0。主要工作是第二部分的合并空闲块过程。合并的过程完全是3.5的合并空闲块技术的实现，处理了所有可能的情况，并且在必要的时候会将发生冲突的rover重新移动到合适的地方。

3.7、本文分配器优缺点总结分析

上文实现了一个完整的基于裸机使用的堆内存分配器。

这个分配器的优点有实现简单；释放块的操作速度极快，是常量级，因为不需要搜索链表的操作；在内存碎片化不严重的时候分配内存速度也非常快，这通常需要应用程序具有良好的动态内存使用习惯，分配的动态内存使用过后及时释放，避免由于不及时释放内存而之后接着申请内存使得堆中的内存块数量过多而造成新分配的时候搜索时间的延长。

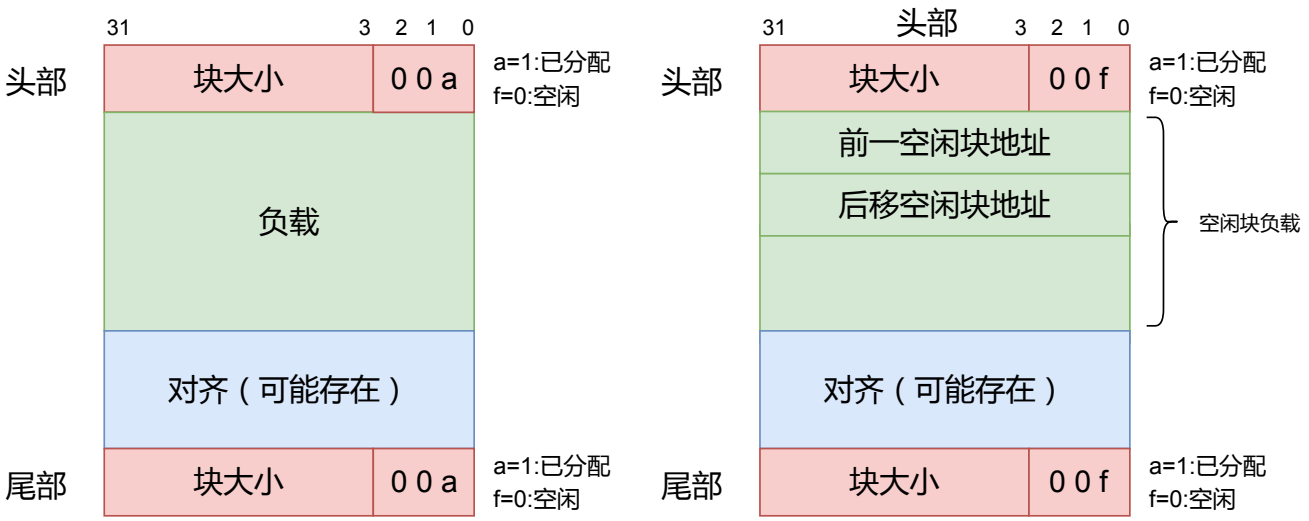
这个分配器的主要缺点也是上面提到的应用程序不及时释放内存造成的搜索时间的延长，除此之外的另外一个小缺陷是每个内存块都需要两个字来存储块的头部和尾部，不能用作用户使用，这是分配器的设计所决定的，必须使用这些额外的空间来实现分配器的分割、合并、对齐等操作，其实任何分配器都避免不了这样的问题，考虑到内存空间的充足和应用程序通常申请动态内存较大的原因，这个可以问题可以看作是分配器实现不可避免的一个小的代价，不必过分担心。因此该分配器的优化主要针对应用程序不及时释放内存而使内存块增多而造成的搜索时间的延长这个问题。

四、分配器优化

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。

4.1、显式空闲链表

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和后一空闲块的指针，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。



（a）分配块　　（b）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。

4.2、分离空闲链表

```
/* 思路 */
```

基于隐式空闲链表分配器以及显式空闲链表（到时候看看要不要），以下没有列出隐式空闲链表的行为过程，参考上文

1.初始化时，组织两个隐式空闲链表堆，一个小的元数据堆，一个大的用户数据堆。元数据堆用来来做存储散列表和里面的项，因为元数据大小恒定，因此可以用一个计数变量来统计元数据堆的使用情况（或者就正常方法）；用户数据堆存储用户真正动态申请内存来存储的数据。

2.分配时，如果在对应的类表（散列表中的一条子链表，由相同大小范围的空闲块组成）中找到合适的空闲项，则删除该项，将剩余的空闲块插入对应的类表头；如果在对应的类表没找到，则往下一层更大的类表中找；如果最后都找不到则mm_sbrk申请一块新空闲堆然后将这块新空闲块插入散列表，再重新申请一次（这一步有待商榷）。

3.释放时，如果没有合并其他空闲块，则直接在散列表对应类表头插入；如果合并了其他空闲块，要先找到相邻被合并的空闲块，然后在对应的类表中删除，最后将合并得到的空闲块插入对应类表。