



2023秋季

计算机系统概论

Introduction to Computer Systems

内存分配基础

⊗ Zhang, Youhui (张悠慧)

✉ zyh02@tsinghua.edu.cn





目录

CONTENTS

01 基本概念

»

02 Implicit free lists

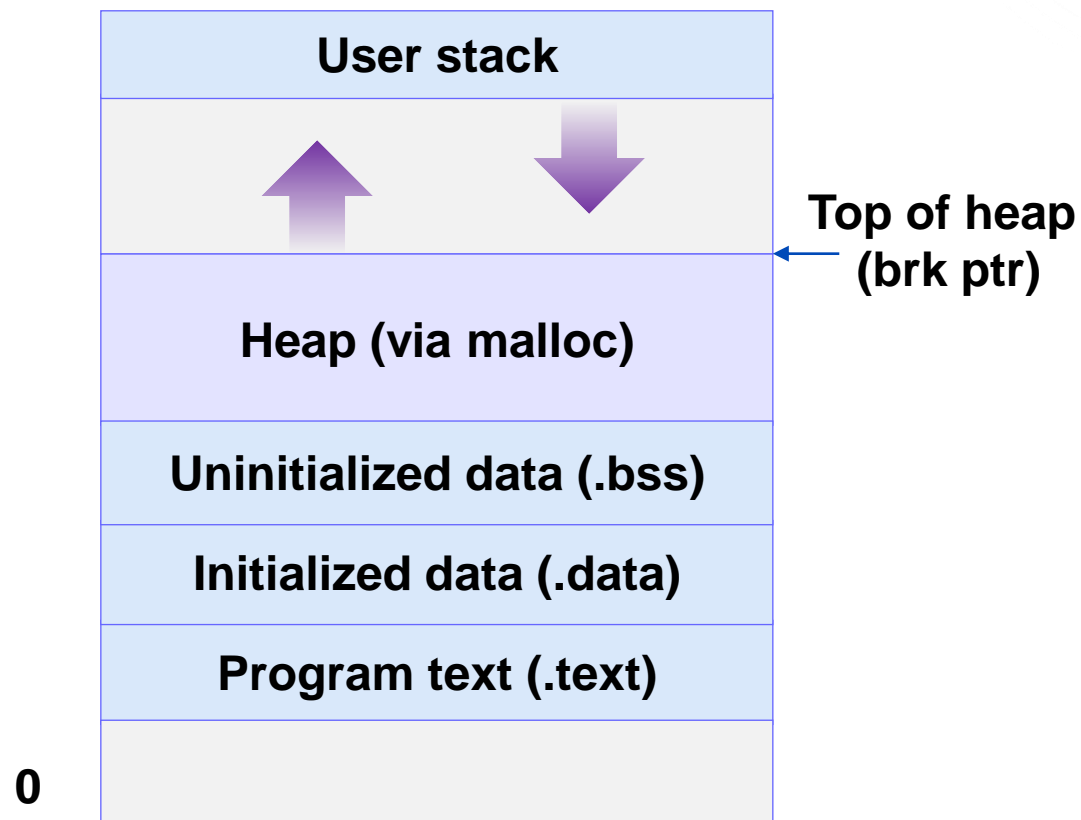
»

(用户层的) 动态内存分配

- 程序员使用动态内存分配器(如 malloc)在运行时获取虚存 (VM)

有些数据结构只有在运行时才知道大小

- 动态内存分配器管理一个称为**堆 (heap)** 的进程虚拟内存区。



Linux系统提供了名为brk的系统调用，用于设定**堆顶**，有兴趣同学可以搜索下Linux system call: *brk*

动态内存分配

- ◆ 动态内存分配器将**堆**视作不同大小的内存块的集合——这些块要么是已分配的，要么是空闲的

◆ 分配器类型

- ▶ **显式分配**: 由应用程序直接分配与释放内存
 - 比如, C语言里的malloc / free
- ▶ **隐式分配**: 应用程序分配, 但不释放
 - 例如, Java、ML和Lisp等语言中的垃圾收集

C函数malloc

```
#include <stdlib.h>
```

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

- Successful:

- Returns a pointer to a memory block of at least **size** bytes (typically) aligned to 8-byte boundary
- If **size == 0**, returns NULL

- Unsuccessful: returns NULL (0) and sets **errno**

```
void free(void *p)
```

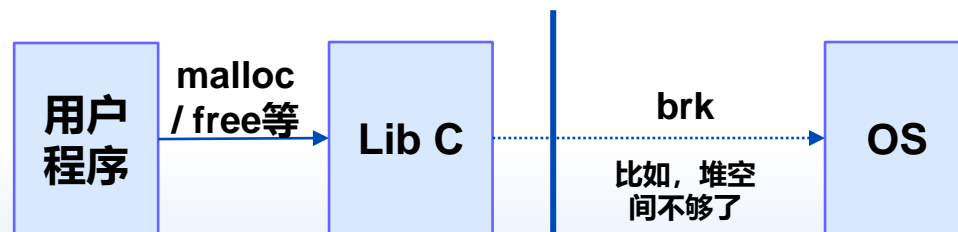
- Returns the block pointed to by **p** to pool of available memory
- **p** must come from a previous call to **malloc** or **realloc**

Other functions

- **calloc**: Version of **malloc** that initializes allocated block to zero.
- **realloc**: Changes the size of a previously allocated block.
- **sbrk**: Used internally by allocators to grow or shrink the heap

》注意malloc等是C函数，而非系统调用

- ◆ C运行时库通过**brk**等系统调用开辟**堆**空间，在其中初始化了相应的数据结构（如链表）来实现动态内存分配

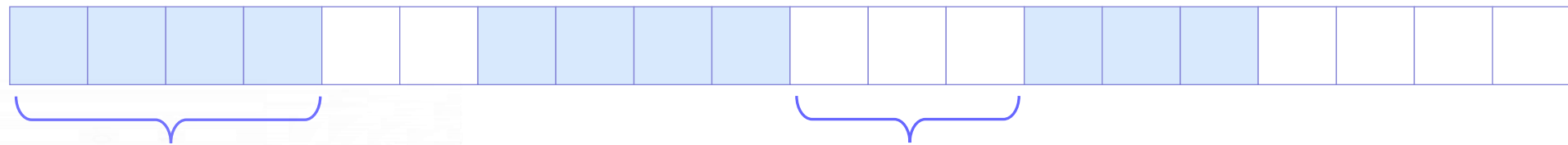


malloc 示例

```
void foo(int n, int m) {  
    int i, *p;  
  
    /* Allocate a block of n ints */  
    p = (int *) malloc(n * sizeof(int));  
    if (p == NULL) {  
        perror("malloc");  
        exit(0);  
    }  
  
    /* Initialize allocated block */  
    for (i=0; i<n; i++)  
        p[i] = i;  
  
    /* Return p to the heap */  
    free(p);  
}
```

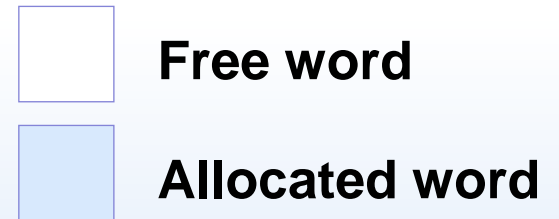

一些假设

内存按 '字' 编址
(指针长度等于字长)



Allocated block
(4 words)

Free block
(3 words)

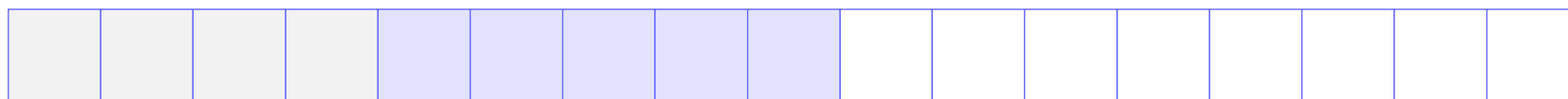


内存分配示例

p1 = malloc(4)



p2 = malloc(5)



p3 = malloc(6)



free(p2)



p4 = malloc(2)



一些约束条件

» 应用

- 可以发出任意序列的malloc和free请求
- free的必须是事先malloc的块

» 内存分配器

- 块的数量或大小可以是任意的
- 必须立即响应malloc请求
 - 也就是说，不能重新排序或缓存请求
- 必须从空闲内存中分配块
- 块地址必须满足对齐要求
 - Linux系统上malloc (C运行时库的malloc)是8字节对齐的
- 只能操作和修改空闲内存
- 不能移动已经分配出去的块
 - 即，无法压缩内存

目标: 吞吐率

» 针对一系列的malloc与free请求

- $R_0, R_1, \dots, R_k, \dots, R_{n-1}$

» 目标: 最大化吞吐率 与 峰值内存利用率

- 这两个目标有时候是冲突的

» 吞吐率 (Throughput)

- 定义: 单位时间内完成的请求数
- 示例:
 - 在10秒内完成5000个malloc和5000个free
 - 相应的吞吐量为1000次操作/秒

目标: 峰值内存利用率

» 针对一系列的malloc与free请求

- $R_0, R_1, \dots, R_k, \dots, R_{n-1}$

» 定义1: 总负载 P_k

- **malloc(p)** 返回一块有效载荷为p字节的内存
- 在请求 R_k 完成后, P_k 是当前已分配的有效载荷的总和

» 定义2: 运行时堆的大小 H_k

- 假设 H_k 是单调非递减的

» 定义3: 峰值内存利用率 (完成k个请求后)

- $U_k = (\max_{i \leq k} P_i) / H_k$

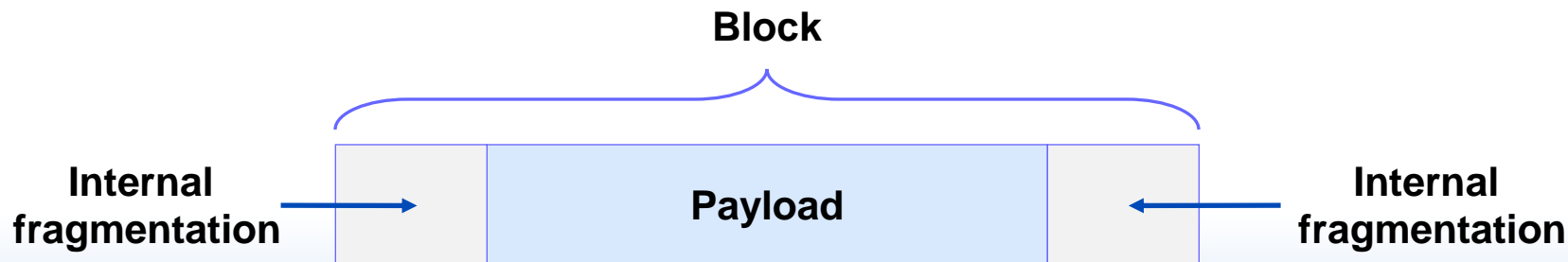
碎片问题导致内存利用率低下

internal fragmentation
(内部碎片化)

external fragmentation
(外部碎片化)

内部碎片化

» 对于给定的内存块，如果有效负载小于块大小，则会发生内部碎片



» 可能的原因

- 维护堆结构的开销
- 地址对齐用的数据填充 (padding)
- 与分配策略相关的额外开销
 - (例如, 返回一个大块来满足一个小请求)

» 仅取决于以往的请求序列

- 因此易于衡量

外部碎片化

» 当堆内有足够多的空闲内存总和，但没有一个单一的足够大的空闲块

`p1 = malloc(4)`



`p2 = malloc(5)`



`p3 = malloc(6)`



`free(p2)`



`p4 = malloc(6)`

Oops! (what would happen now?)

» 取决于未来的请求

- 因此难以衡量

一些实现方面的问题

01 释放给定的一个指针，具体需要释放多少内存？

02 如何跟踪空闲块？

03 当从一个空闲块中分配比该块尺寸小的数据结构时，如何处理多余空间？

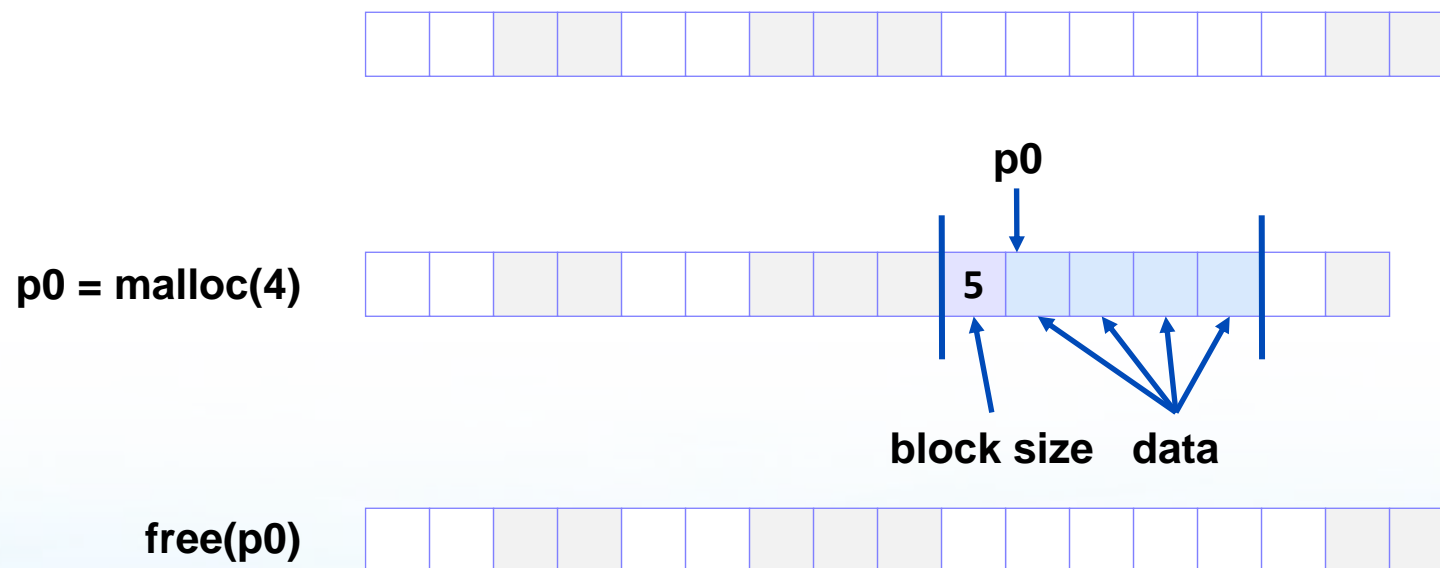
04 如何选择用于分配的空闲块（可能有许多块的大小都能满足）？

05 如何重新插入被释放的块？

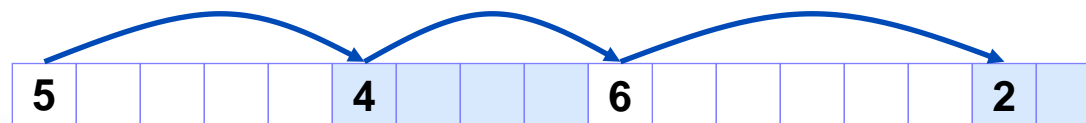
如何知道需要释放多少内存

》 常规方法

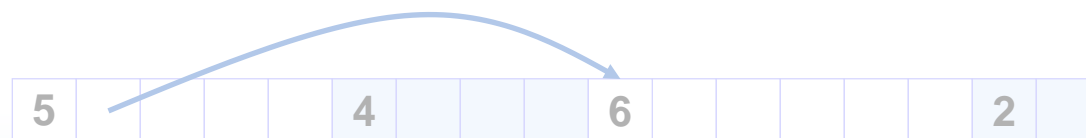
- 将块长度信息存于在该块之前
 - 记录该信息的字 (word) 被称为 **header field** 或 **header**
- 每个分配的块需要一个额外的字 (word)



» 方法1: **Implicit list** 通过块长度将所有块链接起来



» 方法2: 使用指针, 显式地链接空闲块



.....



目录

CONTENTS

01 基本概念

—————>>

02 Implicit free lists

—————>>

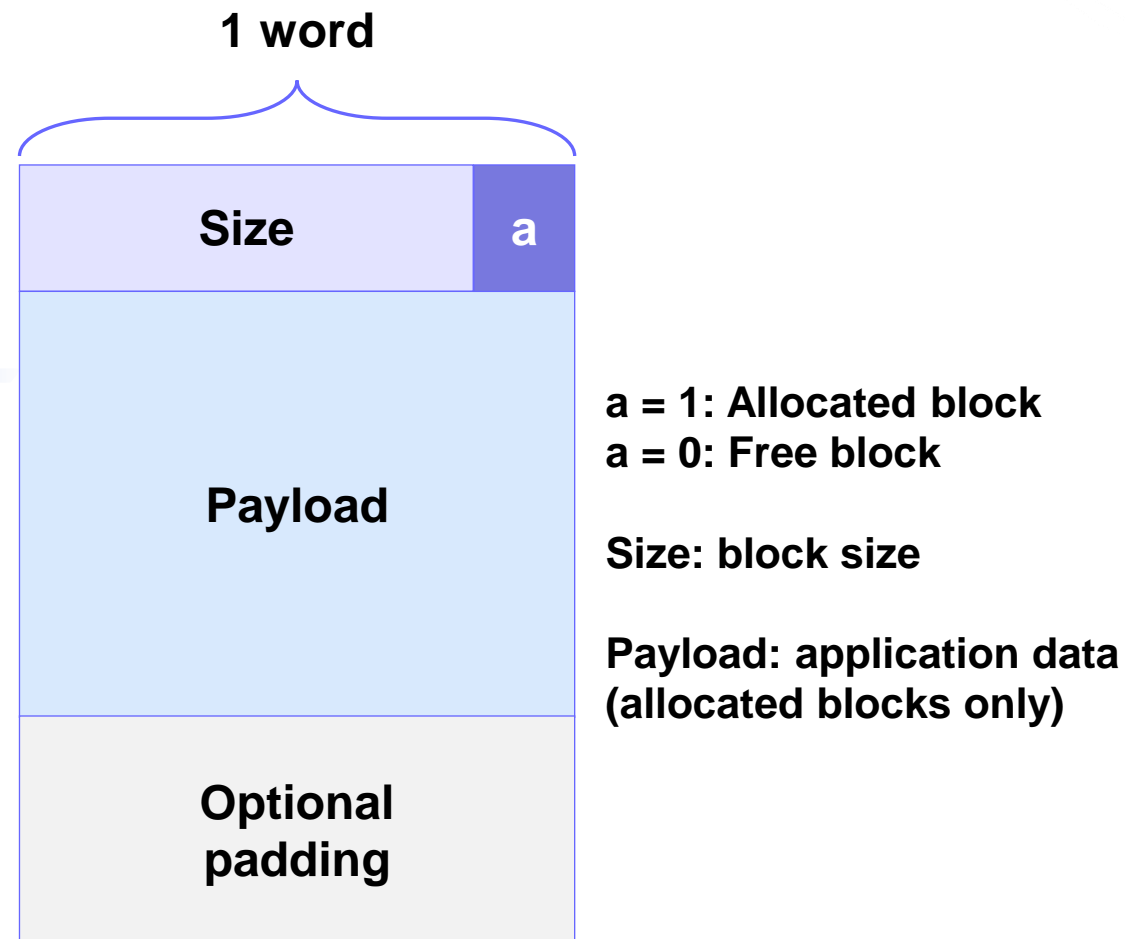
Implicit List

» 对于每个块，需要知道其大小和分配状态

- 显然，用两个字（word）来存储这些信息是很浪费的~

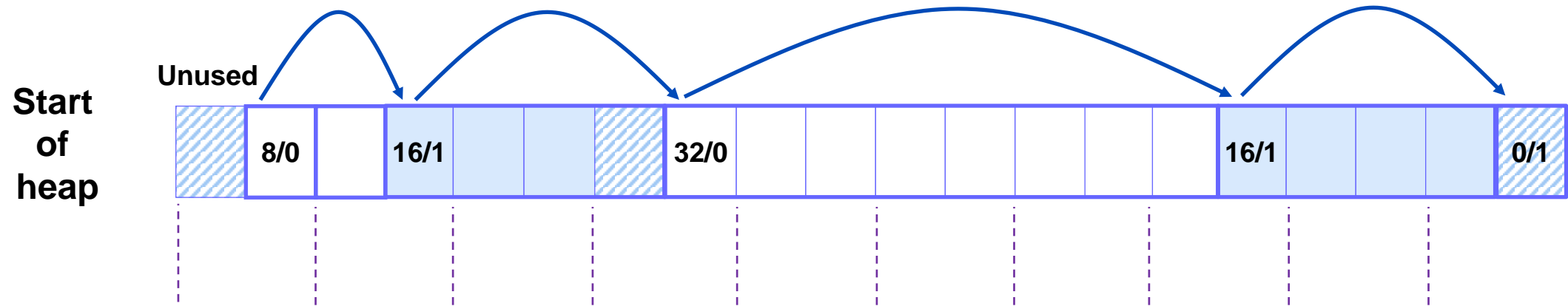
» 一种常规的技巧

- 如果块是对齐的，那么表示块大小的数值的低位总是0
- 这样可以将这些位作为分配/空闲标志
- 注意：在读取大小时，必须屏蔽掉这一位



Format of allocated and free blocks

一个具体示例



Double-word
aligned

Allocated blocks: shaded
Free blocks: unshaded
Headers: labeled with size in bytes/allocated bit

如何选择用于分配的空闲块

» First fit

- 从头搜索链表，选择第一个适合的空闲块：

```
p = start;  
while ((p < end) &&      \\ not passed end  
      ((*p & 1) ||      \\ already allocated  
      (*p <= len)))     \\ too small  
    p = p + (*p & -2);  \\ goto next block (word addressed)
```

- 时间复杂度：与总块数（包括已分配的与空闲的）呈线性关系
- 在实际操作中，会在链表的起始处引起“碎片化”

» Next fit

- 与first fit类似，不同之处在于本次搜索从上次搜索结束的地方开始
- 通常比first fit更快：因为避免重新扫描无用的块
- 但有研究表明，碎片化更糟糕

» Best fit

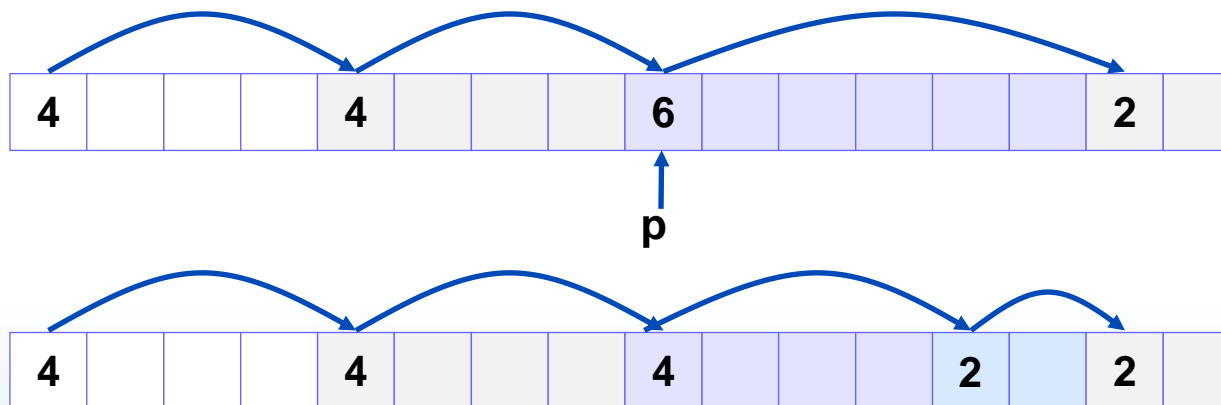
- 搜索列表，选择最合适的空闲块，即分配后该块的剩余字节最少
- 这种策略会产生大量难以利用的“小外部碎片”
- 通常比first fit要慢

分配时如何处理多余空间

» 切分 (splitting)

- 由于分配的空间小于空闲空间，所以需要分割该块

addblock(p, 4)

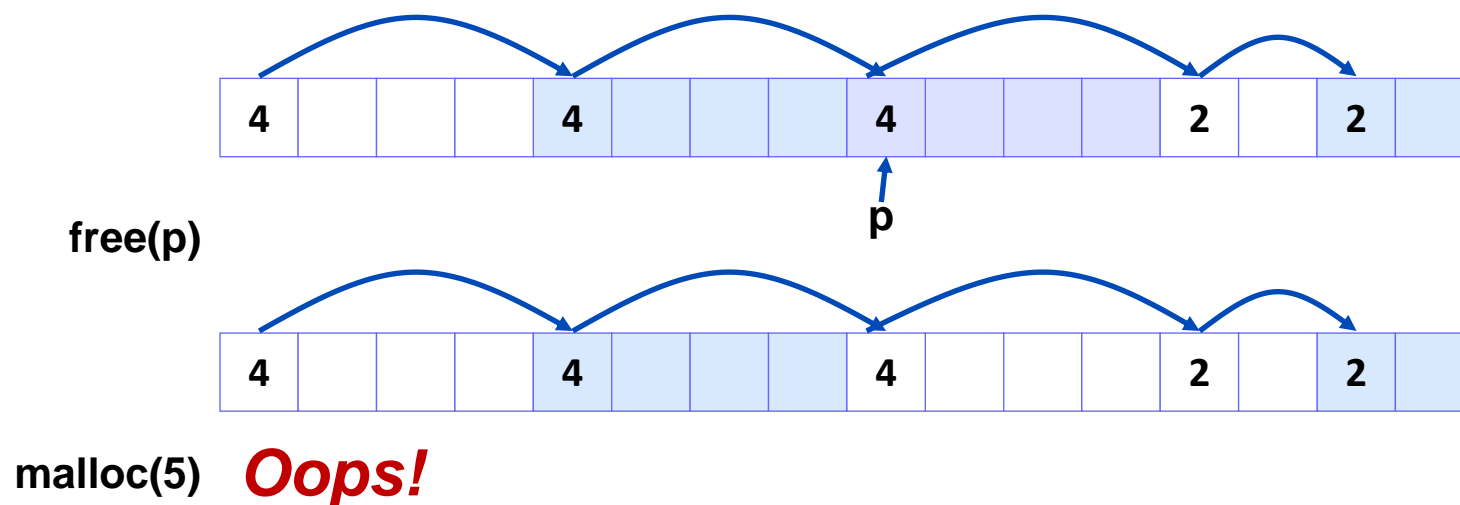


```
void addblock(ptr p, int len) {  
    int newsize = ((len + 1) >> 1) << 1; // round up to even  
    int oldsize = *p & -2; // mask out low bit  
    *p = newsize | 1; // set new length  
    if (newsize < oldsize)  
        *(p+newsize) = oldsize - newsize; // set length in remaining  
                                            // part of block  
}
```

如何释放块

» 一种简单的实现方式

- 仅仅清除 “已分配” 标志
 - `void free_block(ptr p) { *p = *p & -2; }`
- 但会导致 “伪碎片化”

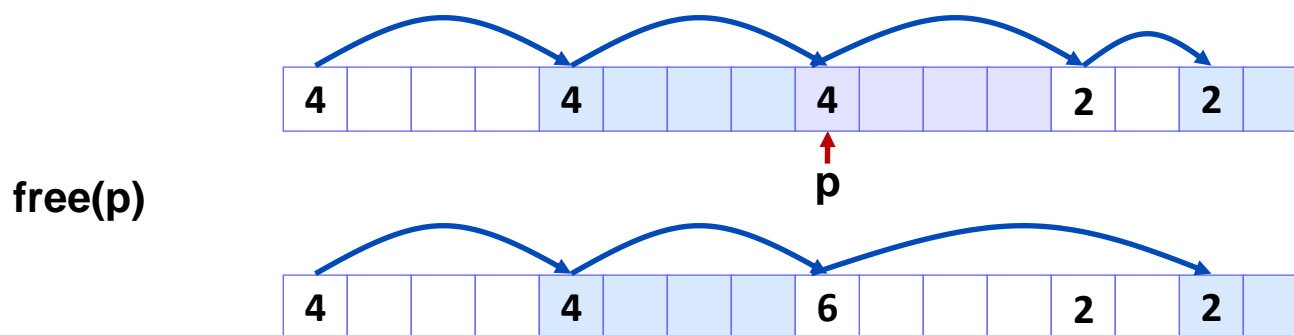


有足够的空闲空间，但分配器却无法使用它们！

解决方法：合并

» 合并空闲的前（后）块

- 示例：与后块合并



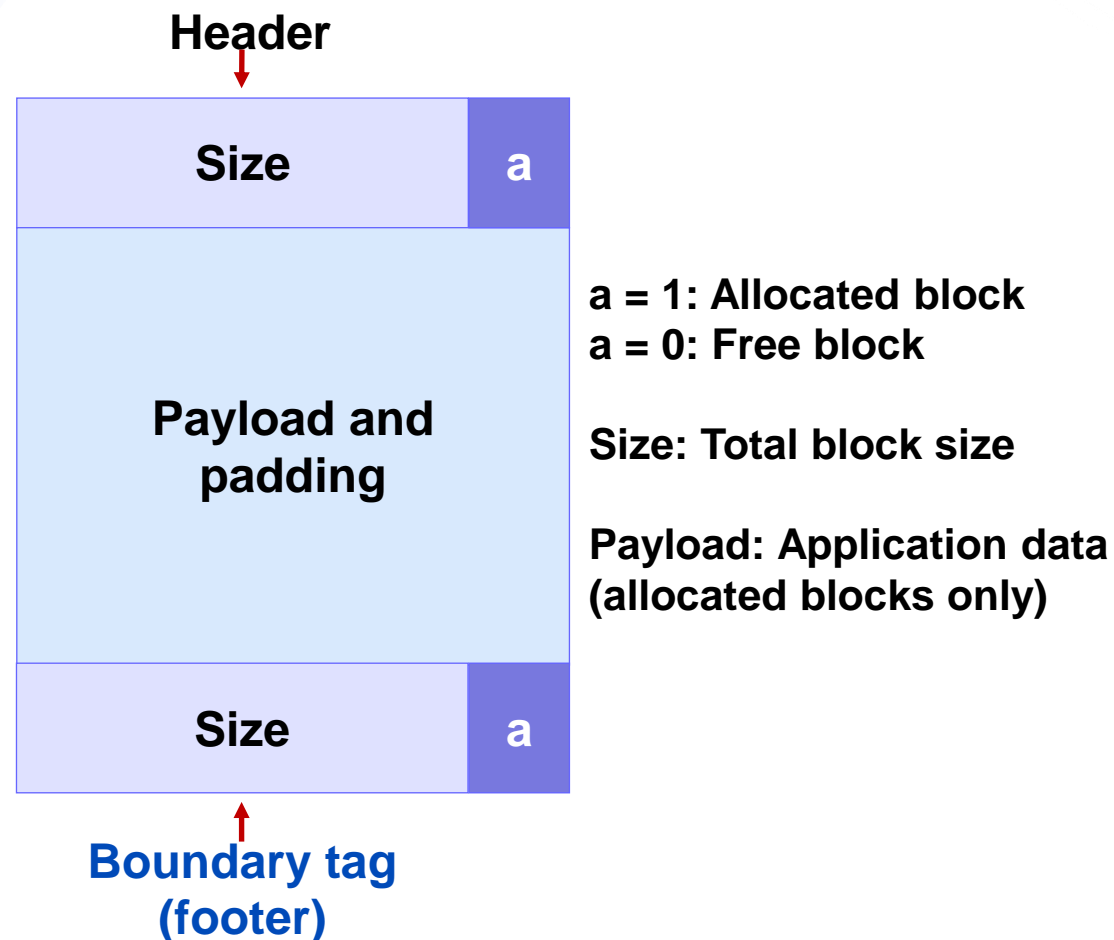
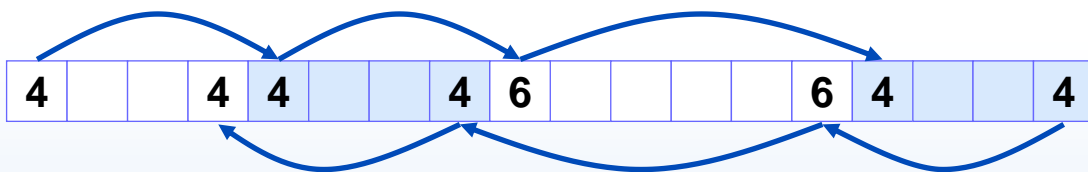
```
void free_block(ptr p) {  
    *p = *p & -2;           // clear allocated flag  
    next = p + *p;          // find next block  
    if ((*next & 1) == 0)  
        *p = *p + *next;    // add to this block if  
                             // not allocated  
}
```

- 但是如何与前块合并？

双向合并

» Boundary tags (边界标记) [Knuth73]

- 在空闲块的“底部” (即末尾)再存一份信息 (大小/分配状态)
- 这样就可以向前遍历列表



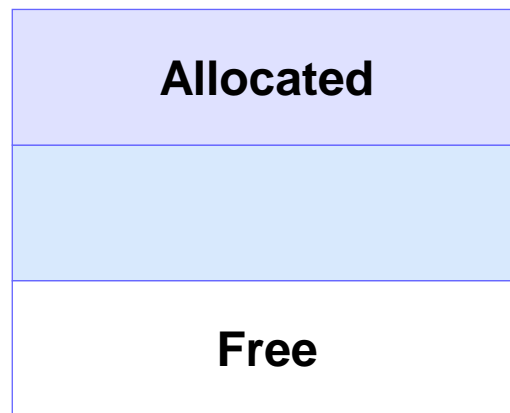
块格式

合并操作的时间复杂度：常数

Case 1



Case 2



Case 3

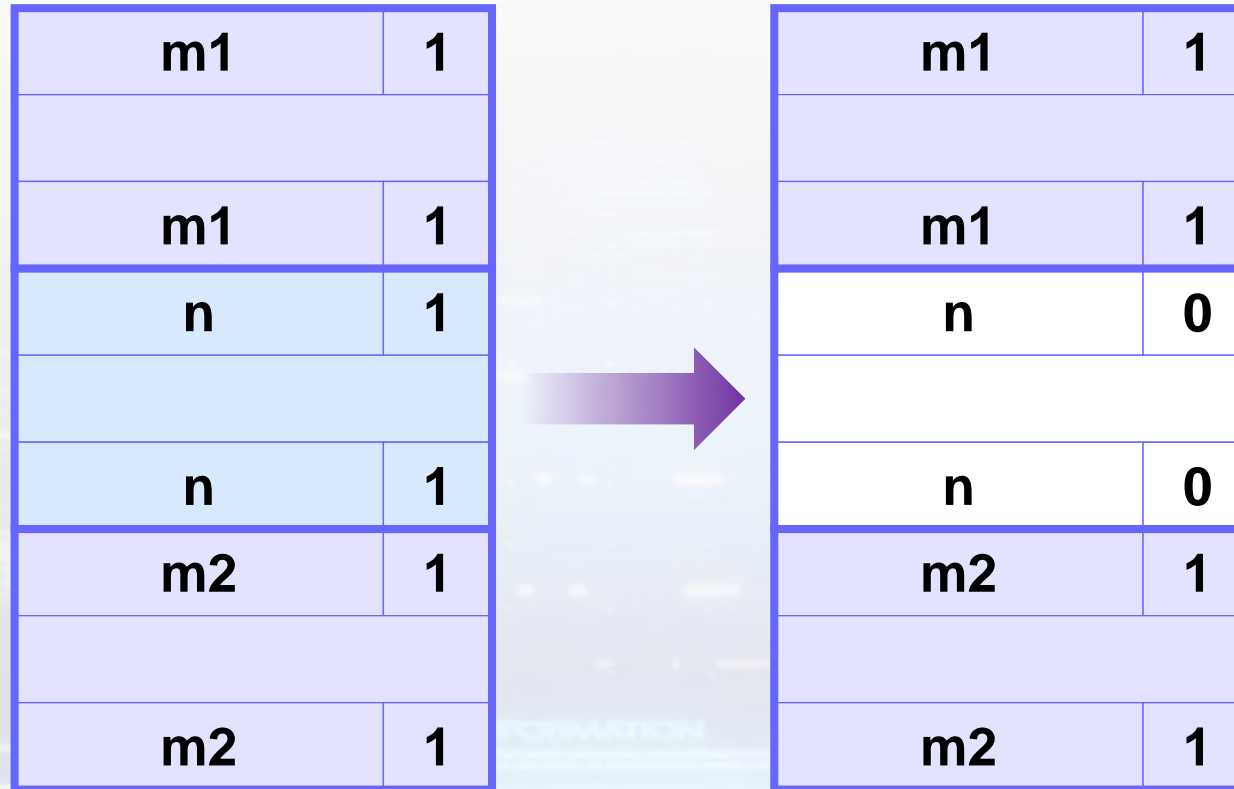


Case 4

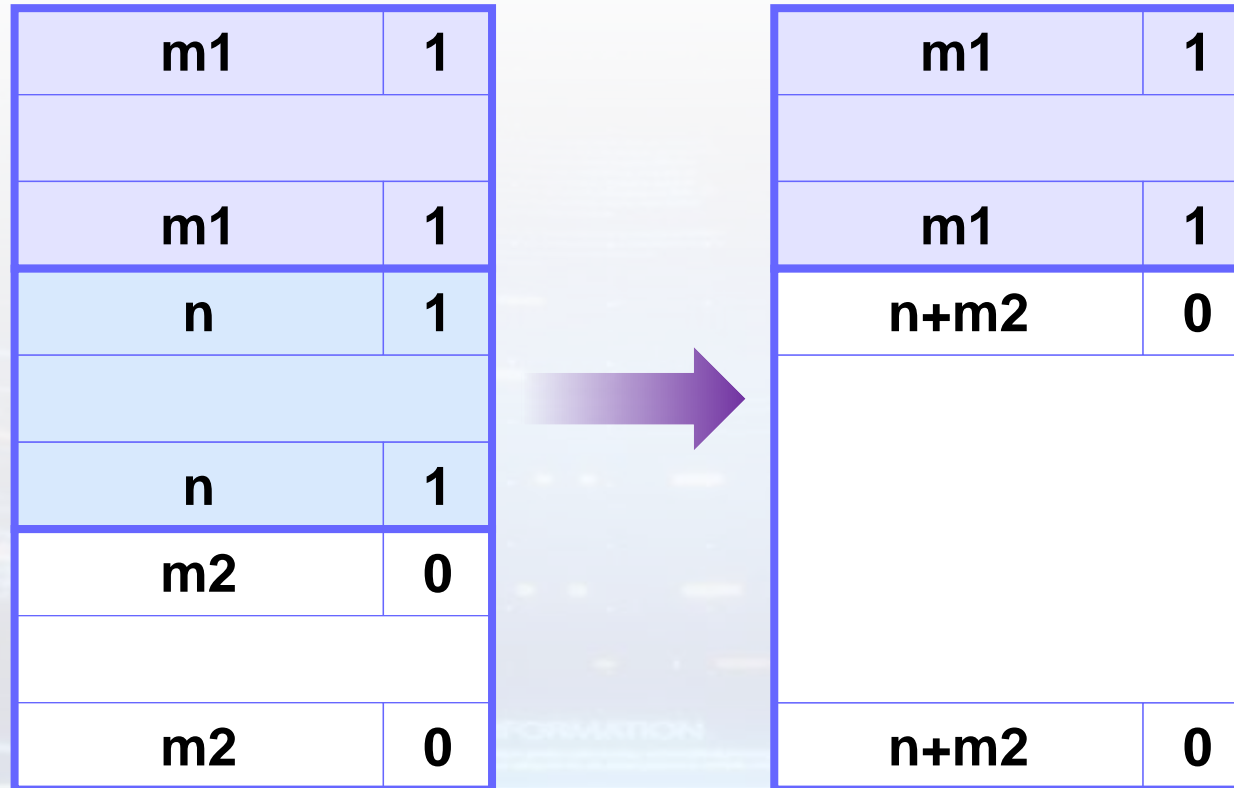


中间块被释放

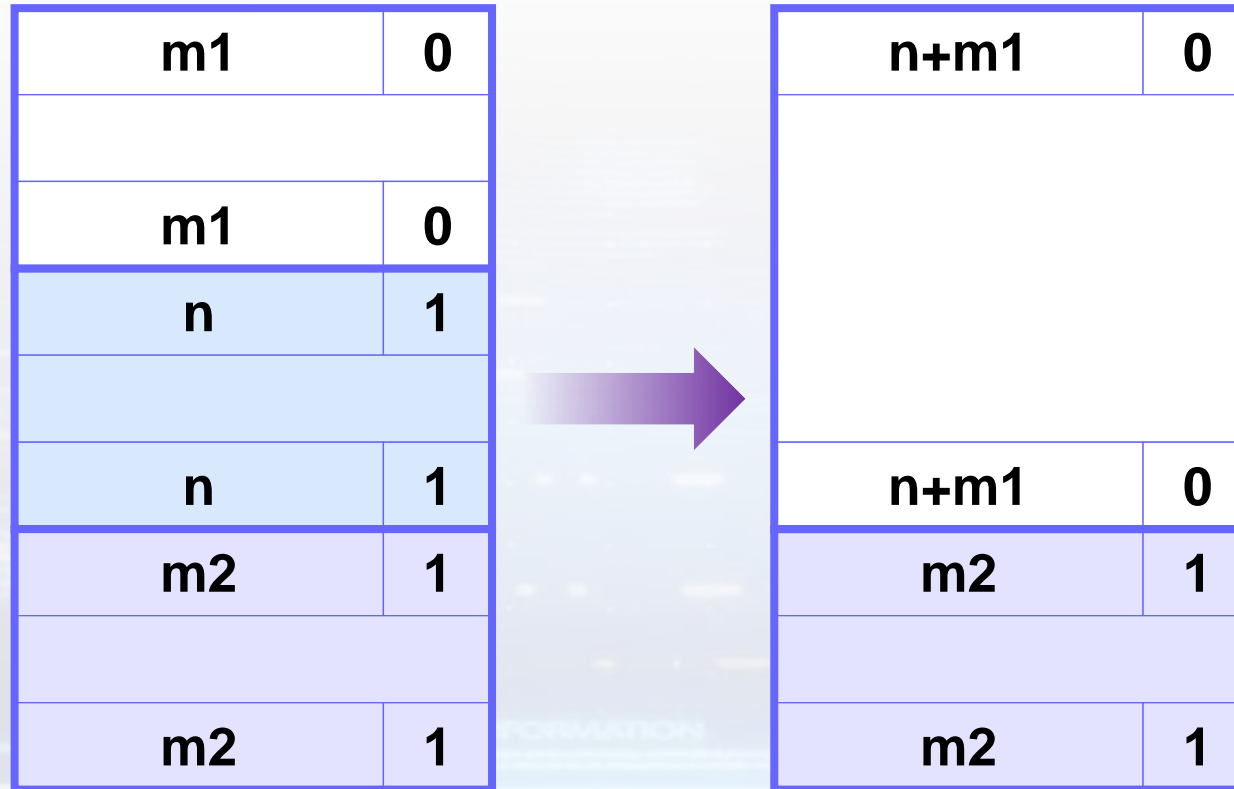
合并操作的时间复杂度：常数



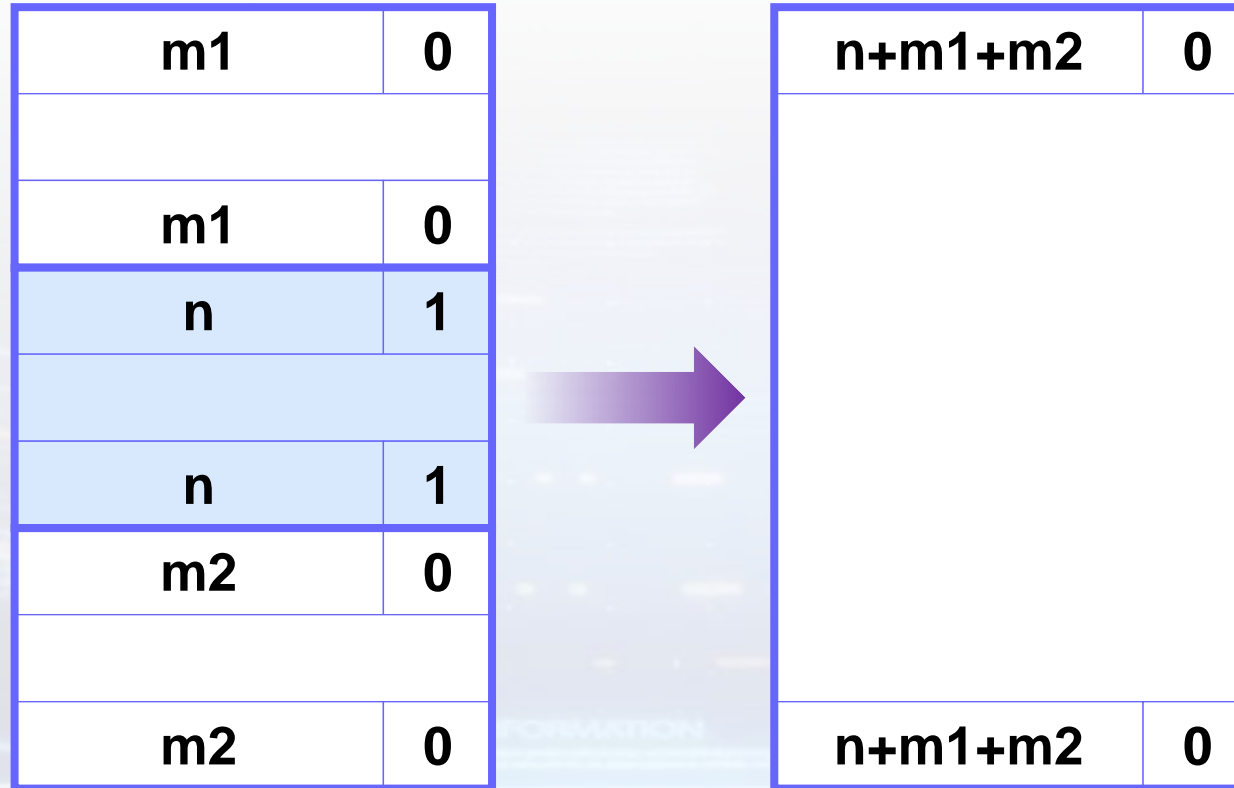
合并操作的时间复杂度：常数



合并操作的时间复杂度：常数



合并操作的时间复杂度：常数



“边界标记”方法的不足

» 内部碎片化

» 如何优化？

- 哪些块需要“底部”标记？
- 这意味着什么？



“边界标记”会产生内部碎片。

但实际上，已分配出去的块不需要设置“底部”标记，因为不会被合并。
为了确定前一个块是已分配的或是空闲的，可以在header里记录前一块的分配信息：header有3个比特是0（一般是8对齐或更高），目前只用了一个bit，剩下2个可用。

Implicit Lists: 小结

» 实现: 非常简单

» 分配的时间开销

- ◆ 线性复杂度

» 释放的时间开销

- ◆ 常数复杂度
- ◆ 包括发生前后合并的情况

» 内存利用率

- ◆ 取决于如何选择用于分配的空闲块
- ◆ First-fit / next-fit / best-fit

» 但是实践中不会应用Implicit Lists这类技术, 因为线性复杂度还是太高

- ◆

» 但是, 分配时的切分以及基于边界标记的合并等概念是通用的