

第9章 虚拟内存: 动态内存分配

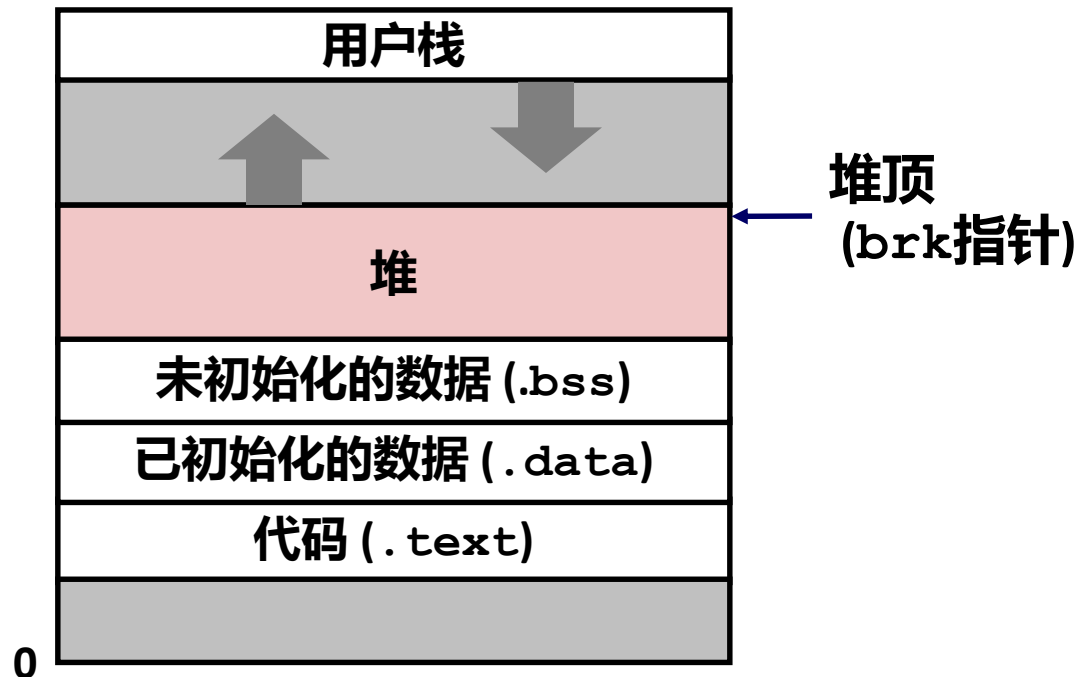
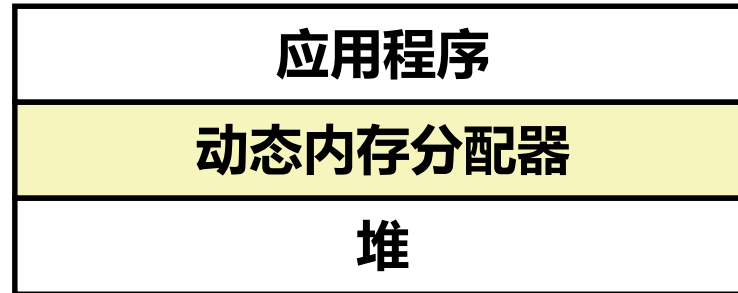
教 师: 郑贵滨
计算机科学与技术学院
哈尔滨工业大学

主要内容

- 动态内存分配(Dynamic Memory Allocation)
- 基本概念
- 隐式空闲列表

动态内存分配

- 在程序运行时程序员使用 **动态内存分配器** (比如 `malloc`) 获得虚拟内存
 - 数据结构的大小只有运行时才知道
- 动态内存分配器维护着进程的一个虚拟内存区域, 称为 **堆**



动态内存分配

- 分配器将堆视为一组不同大小的 **块(blocks)** 的集合来维护，每个块要么是已分配的，要么是空闲的。
- 分配器的类型
 - **显式分配器**: 要求应用显式地释放任何已分配的块
 - 例如，C语言中的 `malloc` 和 `free`
 - **隐式分配器**: 应用检测到已分配块不再被程序所使用，就释放这个块
 - 比如Java，ML和Lisp等高级语言中的垃圾收集 (garbage collection)
- 本节剩下部分将讨论显示分配器的设计和实现。

malloc程序包

- `#include <stdlib.h>`
- `void *malloc(size_t size)`
 - 成功: 返回已分配块的指针, 块大小至少 `size` 字节, 对齐方式依赖编译模式: 8字节 (32位模式), 16字节 (64位模式)。若 `size == 0`, 返回 `NULL` (0)
 - 出错: 返回 `NULL`, 同时设置 `errno`
- `void free(void *p)`
 - 将 `p` 指向的块归还给可用内存池
 - `p` 必须来自之前对 `malloc`、`realloc` 或 `calloc` 的调用, 指向已分配块的起始地址
- 其他函数
 - `calloc`: `malloc` 的另一版本, 将已分配块初始化为 0
 - `realloc`: 改变之前分配块的大小
 - `sbrk`: 分配器隐含地扩展或收缩堆

malloc示例

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

void foo(int n) {
    int i, *p;

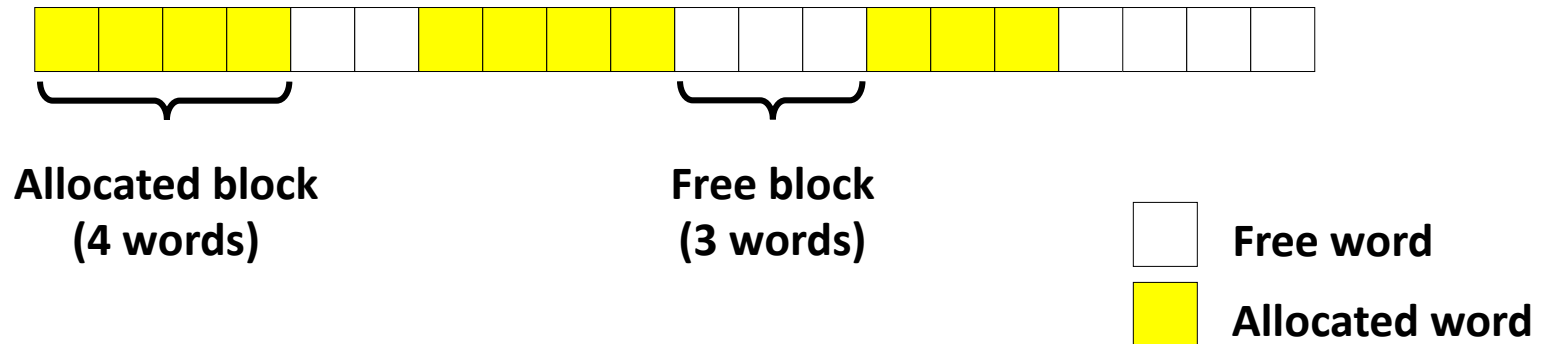
    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;

    /* Return allocated block to the heap */
    free(p);
}
```

本节中的假定

- 内存以字为单位
- 字是int类型的

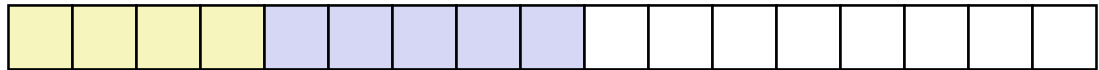


内存分配的例子

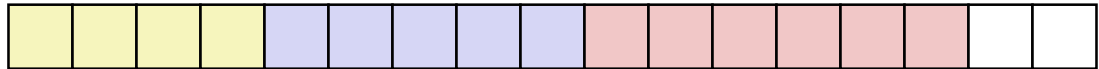
```
p1 = malloc(4)
```



```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(2)
```



限制条件

■ 应用

- 可以处理任意的分配(malloc)和释放(free)请求序列
- 只能释放已分配的块

■ 分配器(Allocators)

- 无法控制分配块的数量或大小
- 立即响应 malloc 请求
 - 不允许分配器重新排列或者缓冲请求
- 必须从空闲内存中分配块
- 必须对齐块，使它们可以保存任何类型的数据对象
 - 在 Linux 上：8字节 (x86) or 16字节 (x86-64) 对齐
- 只能操作或改变空闲块
- 一旦块被分配，就不允许修改或移动它了
 - 压缩已分配块的技术是不允许使用的

性能目标：吞吐量(Throughput)

- 假定n个分配和释放请求的某种序列：
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- 目标: 最大化吞吐量，最大化内存利用率
 - 这些目标经常是互相矛盾的
- 吞吐量(Throughput)
 - 单位时间内完成的请求数
 - 例如：
 - 10秒内完成5,000个分配请求和5,000个释放请求
 - 吞吐量是 1,000次操作/秒

性能目标：最大化内存利用率

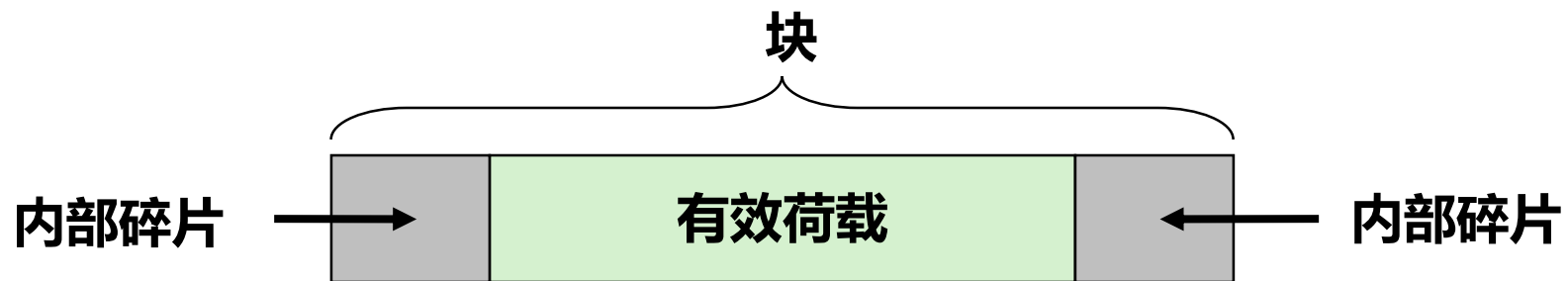
- 给定n个分配和释放请求的某种顺序：
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- 定义：聚集有效载荷 (*Aggregate payload*) P_k
 - `malloc(p)` 结果是一个有效载荷p字节的块
 - 请求 R_k 完成后, **聚集有效载荷** P_k 为当前全部已分配块的有效载荷之和
- 定义：堆的当前大小 H_k
 - 假设 H_k 是单调非递减的
 - 只有分配器使用sbrk时堆才会增大
- 定义：前 $k+1$ 个请求的峰值利用率(Peak Memory Utilization): $U_k = (\max_{i \leq k} P_i) / H_k$

碎片(Fragmentation)

- 碎片化导致内存利用率低
 - 内部 碎片
 - 外部 碎片

内部碎片

- 对于一个给定块, 当有效荷载小于块的大小时会产生内部碎片

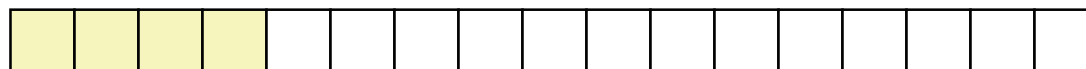


- 产生原因
 - 维护数据结构产生的开销
 - 增加块大小以满足对齐的约束条件
 - 显式的策略决定(Explicit policy decisions)
例如: 返回一个大块以满足一个小的请求
- 只取决于之前分配请求的模式
 - 易于量化

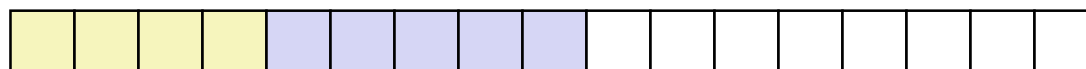
外部碎片

- 虽然空闲内存合起来足够满足一个分配请求，却没有足够大的独立空闲块能满足分配请求——有外部碎片。

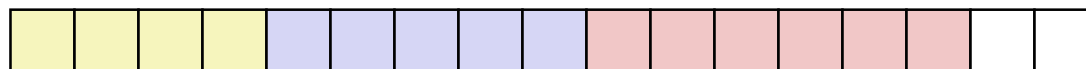
```
p1 = malloc(4)
```



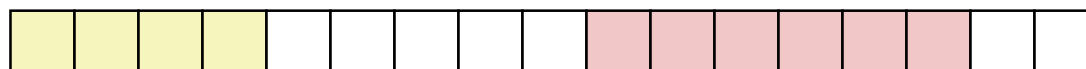
```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(6)
```

Oops! (what would happen now?)

- 取决于将来请求的模式
 - 难以量化

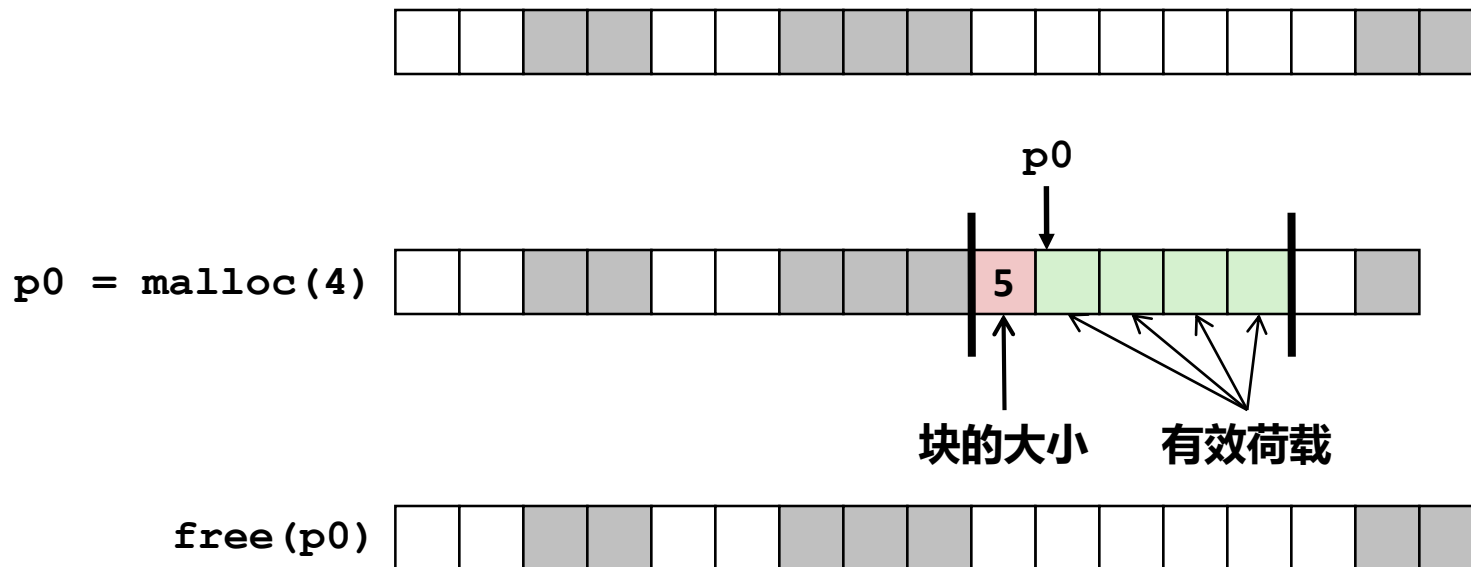
实现问题(Implementation Issues)

- 如何知道一个指针可以释放多少内存？
- 如何记录空闲块？
- 将一个新分配的块放置到某个比较大的空闲块后，如何处理这个空闲块中的剩余部分？
- 有很多合适的块，如何选择一个是空闲块去分配？
- 如何处理一个刚刚被释放的块？

释放一个块，怎么知道它多大？

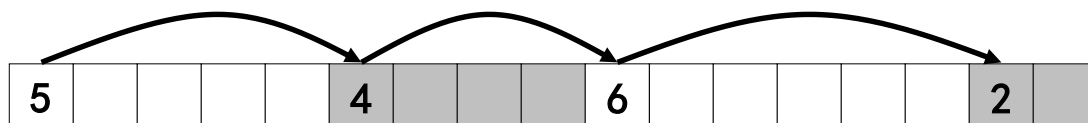
■ 标准方法

- 在块的前端放置一个字(word)，记录该块的长度
 - 这个字称为 **头部**
 - 每个已分配块都需要一个这样的**头部**

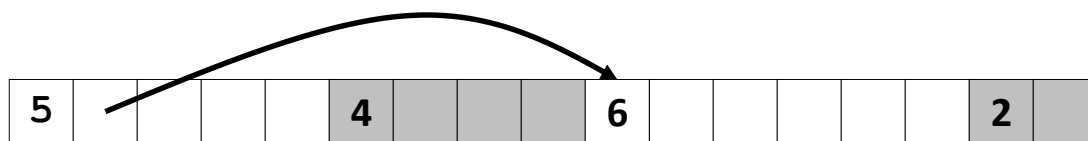


记录空闲块

- 方法 1: **隐式空闲链表 (Implicit list)** 通过头部中的长度字段—隐含地链接所有块



- 方法 2: **显式空闲链表 (Explicit list)** 在空闲块中使用指针



- 方法 3: **分离的空闲列表 (Segregated free list)**
 - 按照大小分类, 构成不同大小的空闲链表
- 方法 4: **块按大小排序**
 - 使用平衡树(如红黑树), 在每个空闲块中保存指针, 并用长度(块大小)作为key值

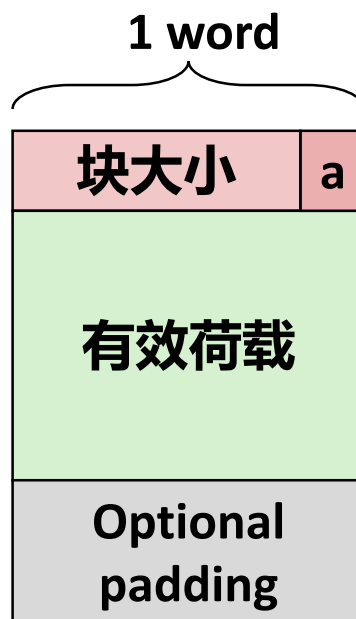
主要内容

- 基本概念
- 隐式空闲链表

方法1: 隐式空闲链表(Implicit List)

- 对于每个块我们都需要知道块的大小和分配状态
 - 虽然可以将这些信息存储在两个 words 中: 浪费!
- 标准技巧 (Standard trick)
 - 如果块是对齐的, 那么一些低阶地址位总是0
 - 不存储这些0位, 将它作为已分配/未分配的标志
 - 读大小字段时, 必须将其屏蔽掉

已分配块、空闲块的格式



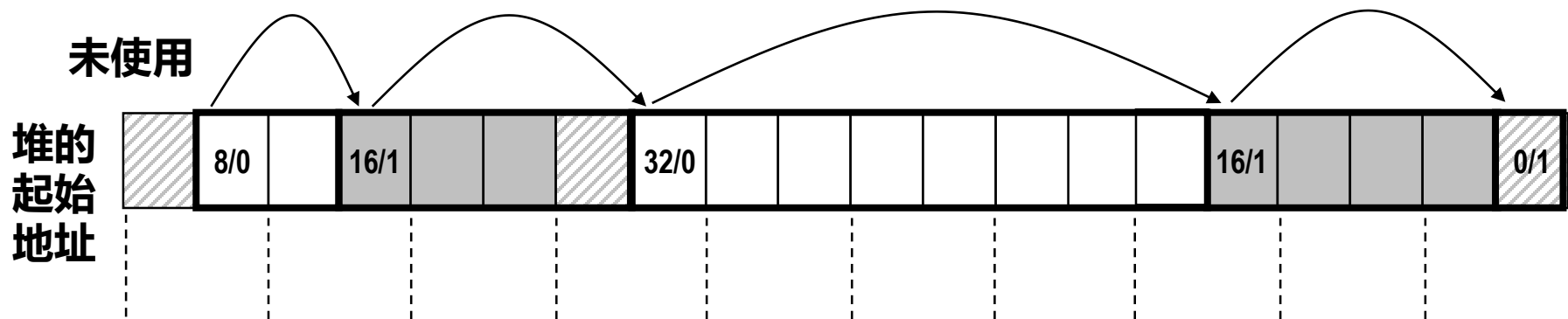
a = 1: 已分配块

a = 0: 空闲块

Size: 块的大小

**有效荷载: 应用数据
(只包括已分配的块)**

隐式空闲链表法细节的示例



双字对齐

已分配块: 阴影

空闲块: 无阴影的

头部: 标记为 (大小(字节)/已分配位)

隐式空闲链表： 找到一个空闲块

■ 首次适配 (First fit):

- 从头开始搜索空闲链表，选择第一个合适的空闲块:

```
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)
```

- 搜索时间：与总块数（包括已分配和空闲块）是线性关系
- 倾向在靠近链表起始处留下小空闲块的“碎片”
 - 增加对较大块的搜索时间

隐式空闲链表：找到一个空闲块

■ 下一次适配 (Next fit):

- 和首次适配相似，只是从链表中上一次查询结束的地方开始
- 比首次适应更快: 避免重复扫描那些无用块
- 一些研究表明，下一次适配的内存利用率要比首次适配低得多

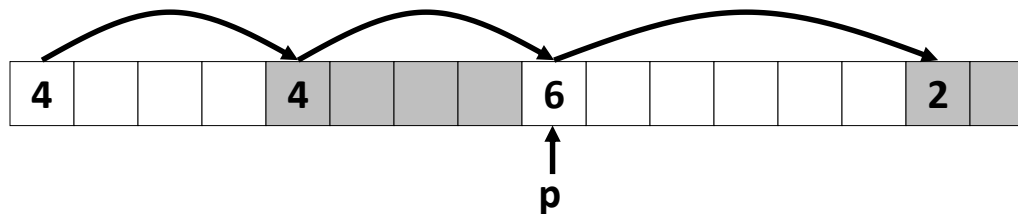
■ 最佳适配 (Best fit):

- 查询链表，选择一个[最好的](#)空闲块：适配，剩余空闲空间最少
- 保证碎片最小——提高内存利用率
- 运行速度通常会慢于首次适配

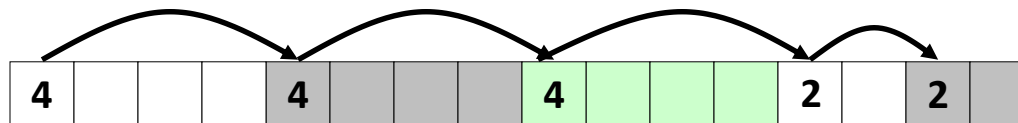
隐式空闲链表： 分配空闲块

■ 分配空闲块:分割 (splitting)

- 申请空间比空闲块小——可以把空闲块分割成两部分



`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
}
```

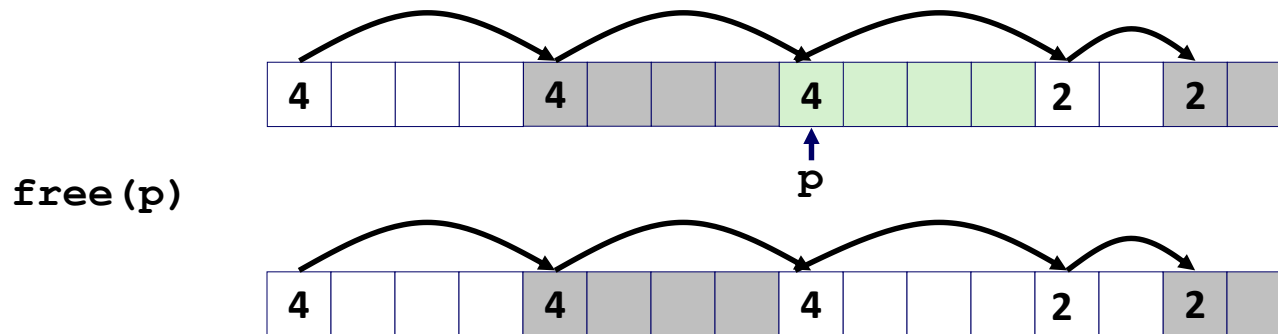
隐式空闲链表：释放一个块

■ 最简单的实现:

- 清除 “已分配 (allocated)” 标志

```
void free_block(ptr p) { *p = *p & -2 }
```

- 但有可能会产生 “假碎片 (false fragmentation)”

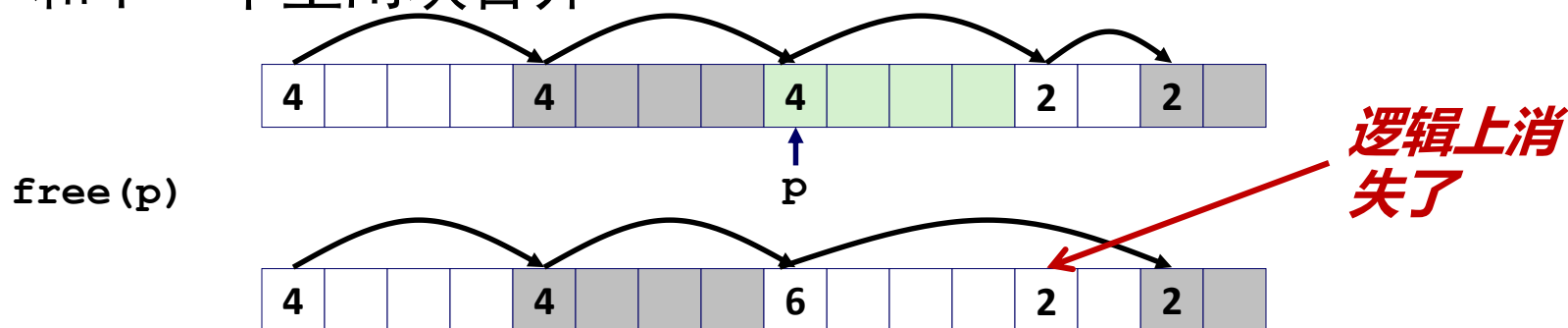


malloc(5) **Oops!**

有足够空间, 但分配器却无法找到它!

隐式空闲链表：合并(Coalescing)

- 合并 (coalesce)：合并相邻的空闲块
 - 和下一个空闲块合并



```
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
}
```

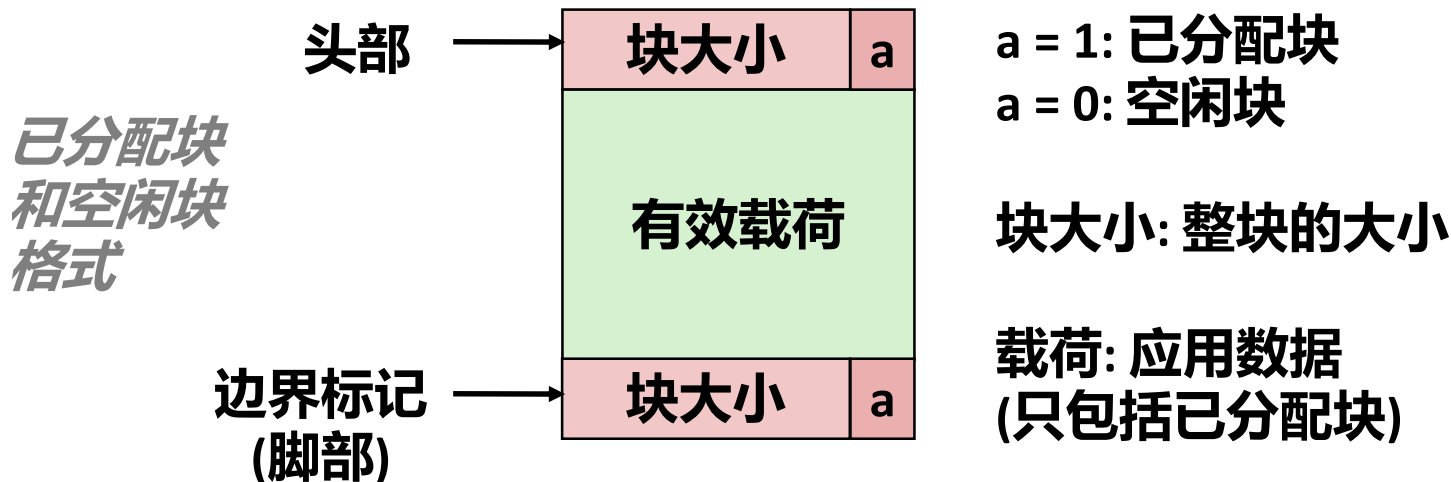
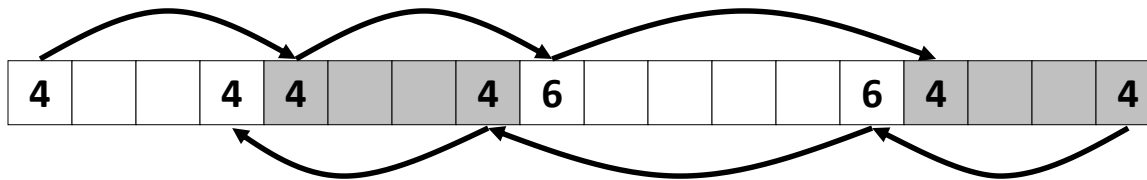
- 如何与前一块合并呢?

隐式空闲链表：双向合并

(Bidirectional Coalescing)

■ 边界标记 (*Boundary tags*) [Knuth73]

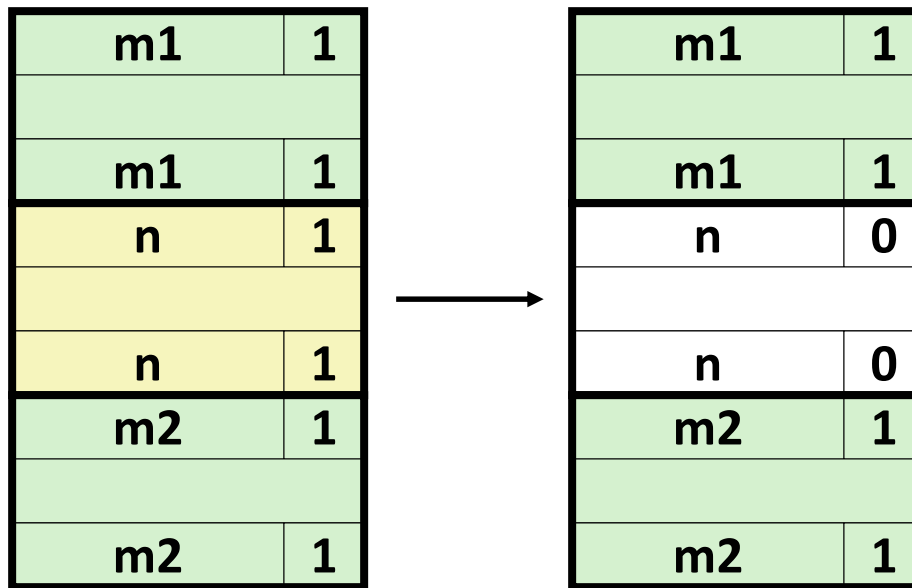
- 在空闲块的“底部”标记“大小/已分配”
- 可以反查“链表”，但需要额外的空间
- 重要且普遍的技术！



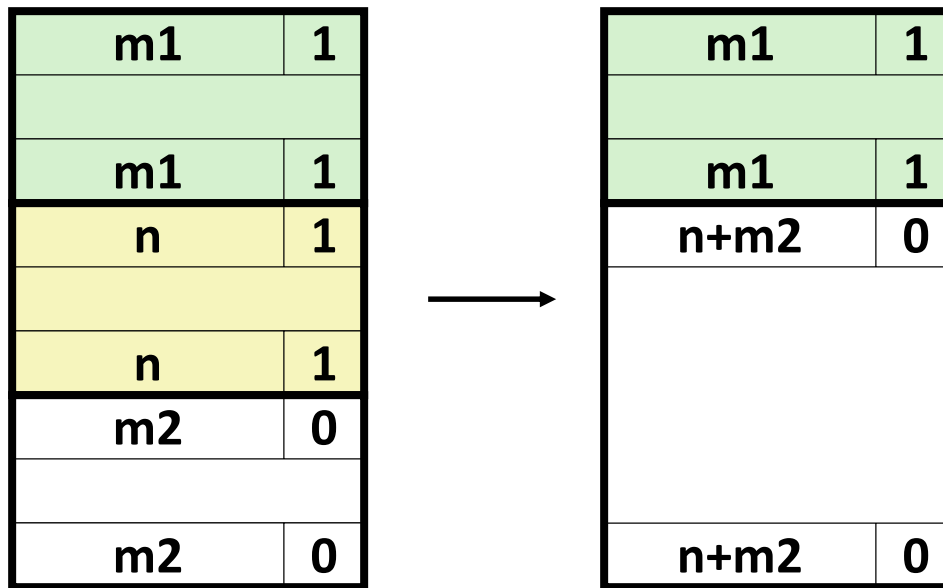
常数时间的合并 Constant Time Coalescing



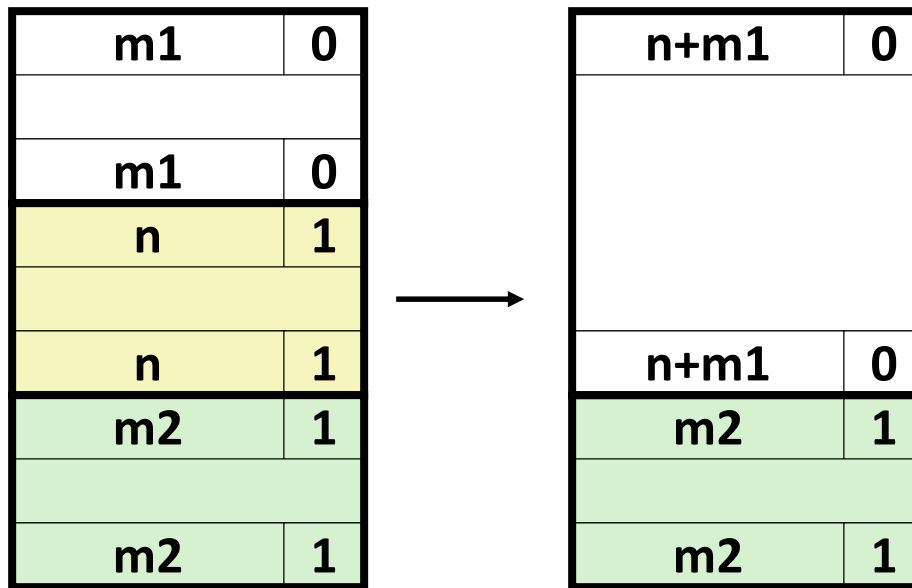
常数时间的合并(情况 1)



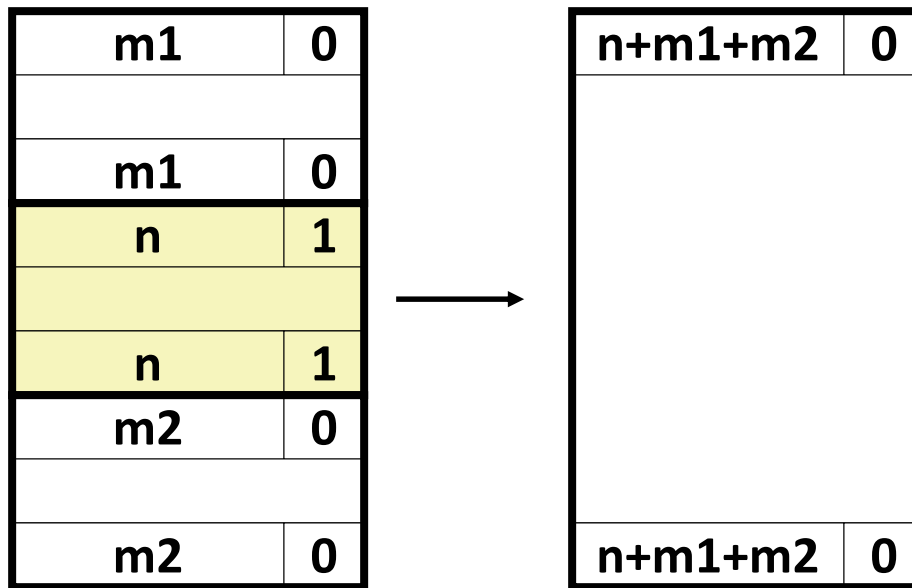
常数时间的合并(情况 2)



常数时间的合并(情况 3)



常数时间的合并(情况 4)



边界标记法的缺陷

■ 内部碎片

■ 能否优化？

■ 哪些块需要脚标？

- 空闲块

■ 有什么作用？

- 与之前块、之后的块合并时使用

■ 优化方法

- 将已分配块前面块是否空闲已分配/空闲位存放在当前块中多出来的低位中，不需要脚部了

关键分配策略总结

- 放置策略(Placement policy)
 - 首次适配, 下一次适配, 最佳适配, 等等.
 - 减少碎片以提高吞吐量
 - **有趣的观察**: 近似于最佳适配算法, 独立的空闲链表不需要搜索整个空闲链表
- 分割策略(Splitting policy)
 - 什么时候开始分割空闲块?
 - 能够容忍多少内部碎片?
- 合并策略(Coalescing policy)
 - **立即合并 (Immediate coalescing)**: 每次释放都合并
 - **延迟合并 (Deferred coalescing)**: 延迟合并, 直到需要才合并, 提高释放的性能。例如:
 - 为 `malloc` 扫描空闲链表时合并
 - 外部碎片达到阈值时合并

隐式链表总结

- 实现: 非常简单
- 分配开销: 最坏情况——线性时间
- 释放开销: 最坏情况——常数时间 (即使有合并)
- 内存使用:
 - 取决于放置策略
 - 首次适配(First-fit)、下一次适配(next-fit)或最佳适配(best-fit)
- 由于分配是线性时间的, 没有实际用于malloc/free
 - 用于许多特殊目标的应用
- 然而, 分割和边界标记合并的概念对于所有的分配器来说都是通用的