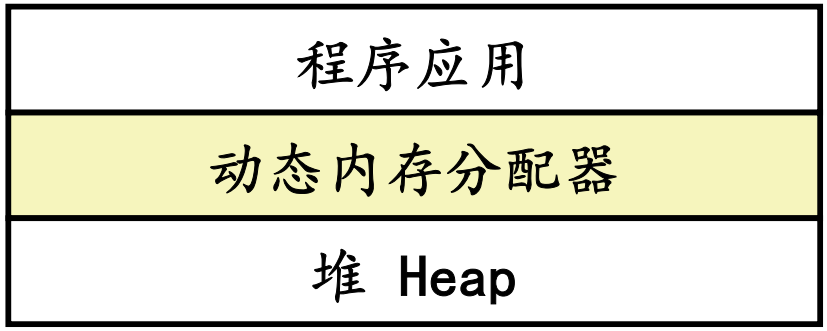
动态内存分配：基本概念
(Dynamic Memory Allocation)

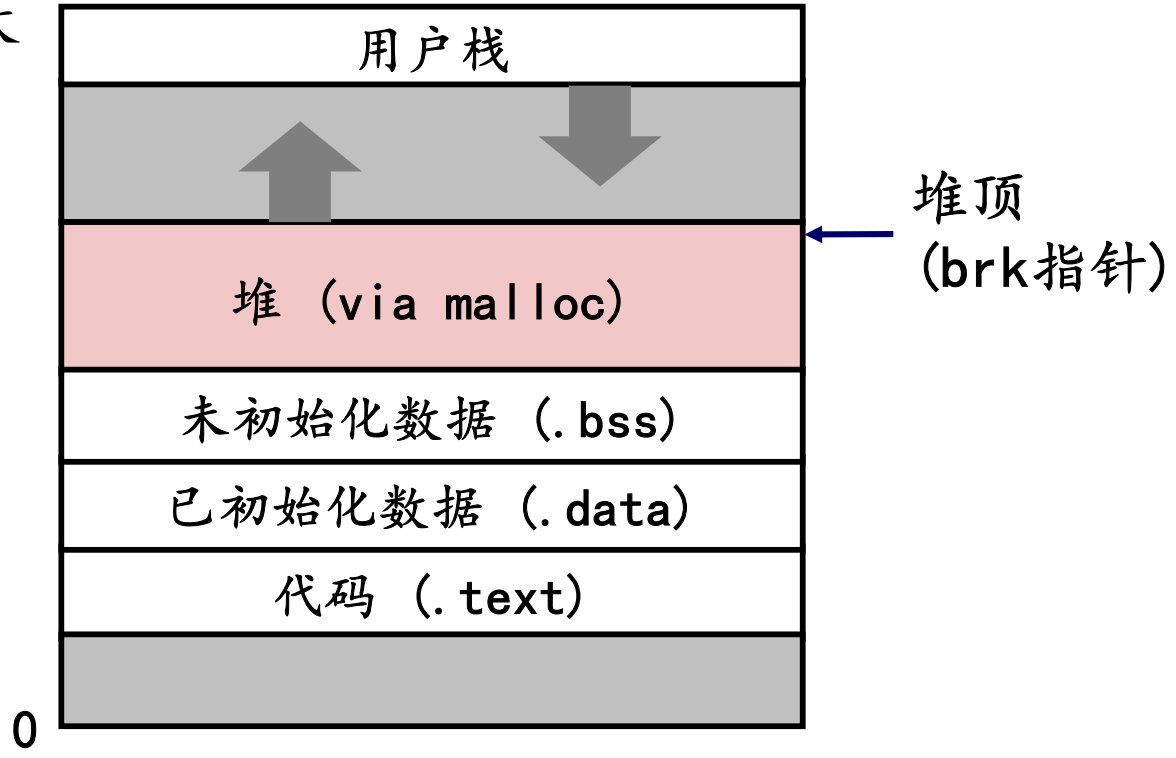
主要内容

- 基本概念
- 隐式空闲列表

动态内存分配

- 程序员使用动态内存分配器(Dynamic memory allocator)（例如malloc）在运行时获取VM。
  - 对于仅在运行时才知道大小的数据结构。
- 动态内存分配器管理称为堆(Heap)的进程虚拟内存区域。





动态内存分配

- 分配器将堆维护为可变大小的块的集合，这些块要么被分配，要么被释放
- 分配器的类型
  - 显式分配器：应用程序分配和释放空间
    - E.g.， C语言中 malloc 和 free
  - 隐式分配器：应用程序分配空间，但不释放空间
    - 例如 Java、ML 和 Lisp 中的垃圾回收
- 今天将讨论简单的显式内存分配

malloc程序包

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

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

- 成功时： ：
  - 返回一个指针，指向至少包含 size 字节的内存块，并且内存块起始地址按照 8 字节（x86）或 16 字节（x86-64）的边界对齐。
  - 如果 size == 0，返回 NULL。
- 失败时：返回 NULL (0) 并设置 errno。

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

- 将指针 p 所指向的内存块归还到可用内存池。
- 参数 p 必须来自于之前的 malloc 或 realloc 调用。

其他函数

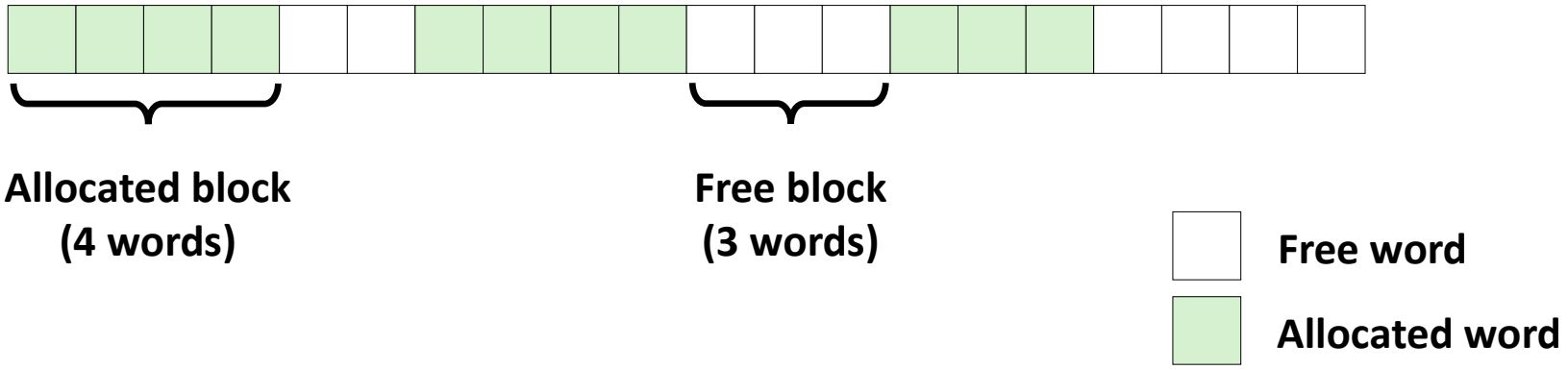
- calloc：malloc 的一个版本，分配的内存会初始化为 0。
- realloc：改变之前已经分配的内存块大小。
- sbrk：分配器在内部调用它来扩展或收缩堆。

Malloc例子

```
#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);
}
```

本课堂中的假设

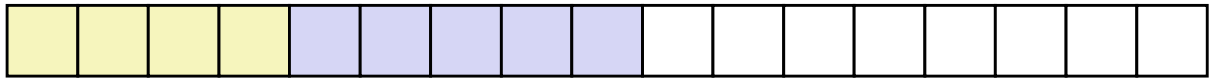
- 内存是按字寻址的。
- 字的大小是整型数**(int)**。



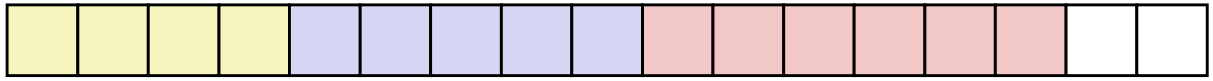
分配例子

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

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



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



```
free(p2)
```

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

约束条件

- 应用程序
  - 可以发出任意顺序的 malloc 和 free 请求
  - free 请求必须针对已 malloc 的块

- 分配器
  - 无法控制已分配块的数量或大小
  - 必须立即响应 malloc 请求
    - 即，无法重新排序或缓冲请求
  - 必须从空闲内存中分配块
    - 即，只能将已分配的块放置在空闲内存中
  - 必须对齐块以满足所有对齐要求
  - 在 Linux 系统上，8 字节（x86）或 16 字节（x86-64）对齐
  - 只能操作和修改空闲内存
  - 分配的块一旦 malloc 分配完毕，就无法移动
    - 即，不允许进行压缩

性能目标：吞吐量

- 给定一些 malloc 和 free 请求序列
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- 目标：最大化吞吐量和峰值内存利用率
  - 这些目标通常相互冲突
- 吞吐量：
  - 单位时间内完成的请求数量
  - 示例：
    - 10 秒内 5,000 次 malloc 调用和 5,000 次 free 调用
    - 吞吐量为每秒 1,000 次操作

性能目标：峰值内存利用率

- 给定一些 malloc 和 free 请求序列：
  - R_0，R_1，…，R_k，…，R_{n-1}
- 定义：总有效负载 P_k
  - malloc(p) 会分配一个大小为 p字节 的块（称为 payload）
  - 当请求 R_k 完成后，总有效负载 P_k 等于当前所有已分配块的 payload 之和
- 定义：当前堆大小 H_k
  - 假设 H_k 是单调不减的
  - 也就是说，堆大小只会在分配器调用 sbrk 时增长
- 定义：第 k+1 个请求后的峰值内存利用率 U_k

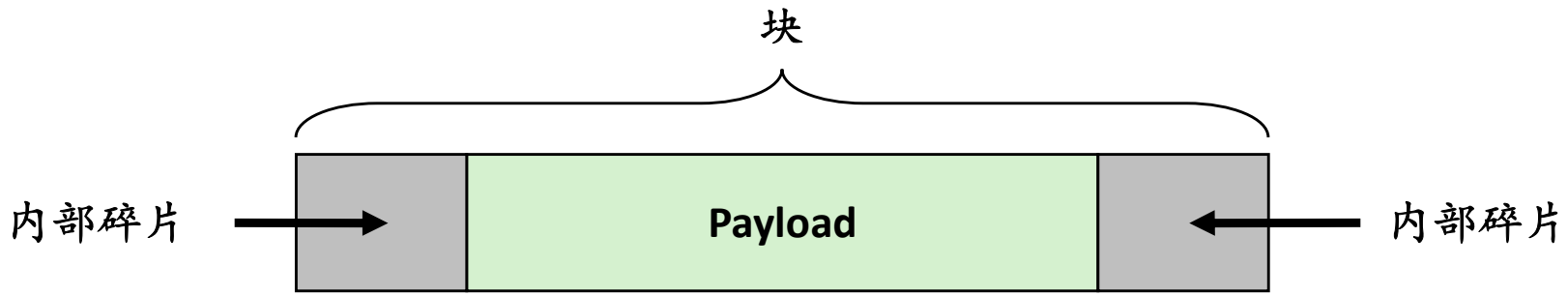
$$U_k = \frac{\max_{i \le k} P_i}{H_k}$$

碎片化

- 碎片导致内存利用率低
  - 内部碎片
  - 外部碎片

内部碎片

- 对于一个给定的块，如果 有效负载（payload）小于块的大小，就会产生内部碎片（internal fragmentation）。



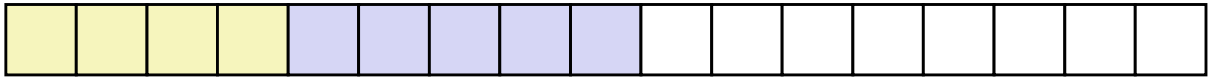
- 成因：
  - 维护堆数据结构的开销
  - 为对齐（alignment）而填充的字节
  - 明确的策略决策
    （例如：为了满足一个小请求，返回了一个较大的块）
- 特点：
  - 内部碎片 **只依赖于过去的请求模式**
  - 因此，它相对容易测量

外部碎片

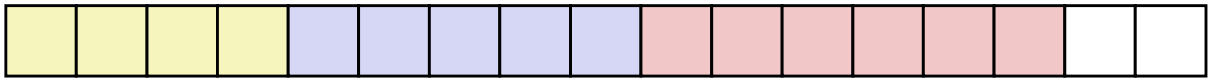
- 当堆中的总空闲内存足够大，但没有一个连续的空闲块足够大时，就会产生外部碎片。

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

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



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



```
free(p2)
```

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

这时会发生什么？

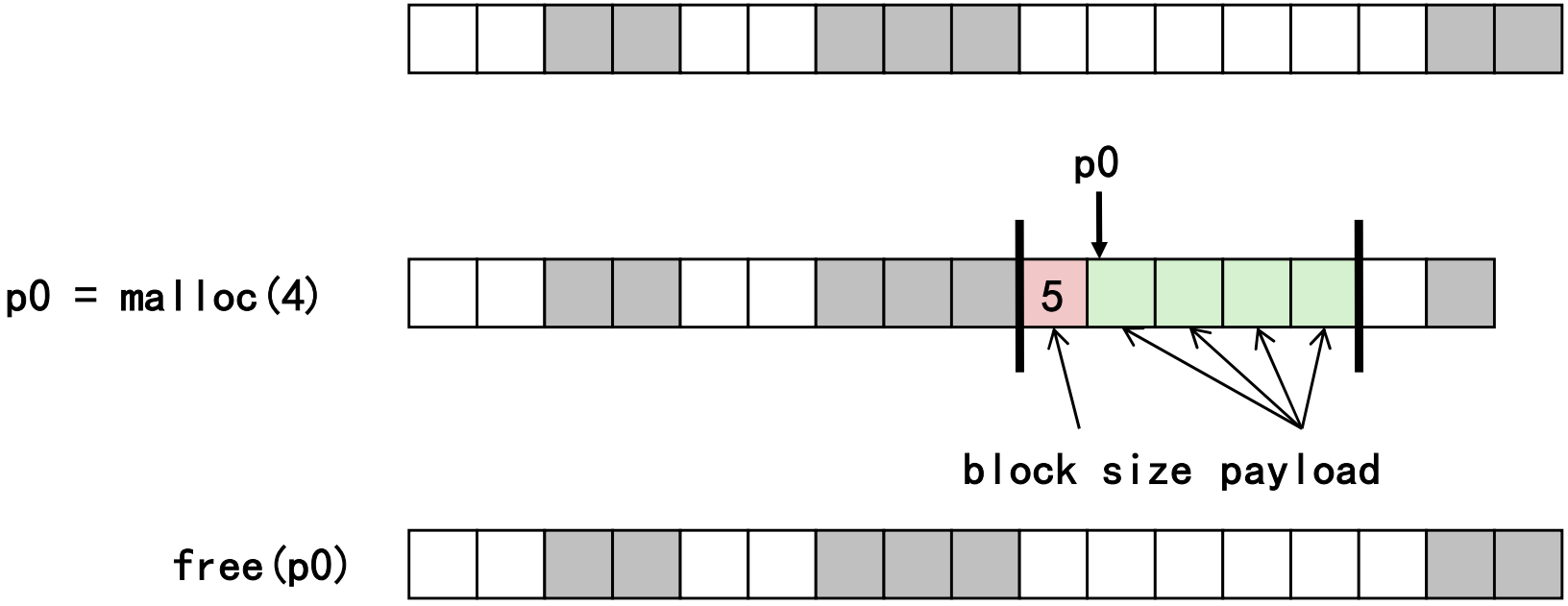
- 外部碎片 依赖于未来请求的模式
  - 因此，很难准确估计

实现中的问题

- 我们如何仅通过一个指针，知道需要释放多少内存？
- 我们如何追踪空闲的内存块？
- 当分配的结构比空闲块小的时候，多余的空间怎么办？
- 如果有多个合适的空闲块，我们该如何选择分配？
- 我们如何重新插入被释放的内存块？

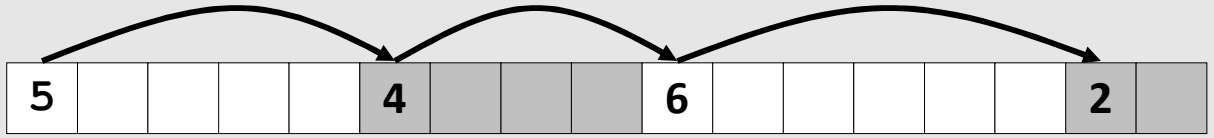
如何知道需要释放多少内存

- 标准方法
  - 将块的长度存储在该块前面的一个字中。
    - 这个字通常被称为头字段（header field）或头部（header）。
  - 每个已分配的块都需要额外的一个字来存储这些信息。

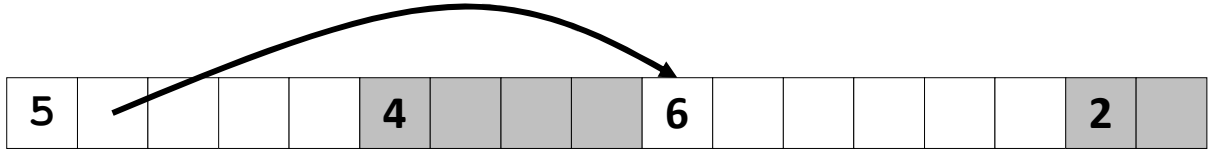


追踪空闲块的方法

- 方法 1：隐式链表（Implicit list）使用块的长度信息，把所有块串联起来



- 方法 2：显式链表（Explicit list）仅在空闲块之间使用指针建立链表



- 方法 3：分离空闲链表（Segregated free list）为不同大小类别维护不同的空闲链表

- 方法 4：按大小排序的块（Blocks sorted by size）
  - 使用平衡树（如红黑树），在每个空闲块里保存指针，并以块的长度作为关键字进行组织

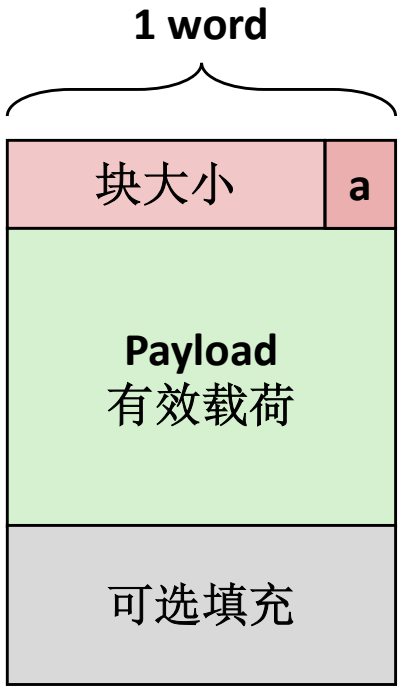
主要内容

- 基本概念
- 隐式空闲列表

方法 1：隐式链表

- 对每个块，我们需要同时记录大小和分配状态
  - 可以把这两个信息分开存到两个字中，但这样很浪费。
- 标准技巧
  - 如果块是按对齐存放的，那么低位地址中的某些比特永远是 0。
  - 与其浪费一个总是 0 的比特，不如把它用来作为「已分配 / 空闲」的标志位。
  - 当我们读取块大小时，就需要把这个标志位屏蔽掉（mask 掉）。

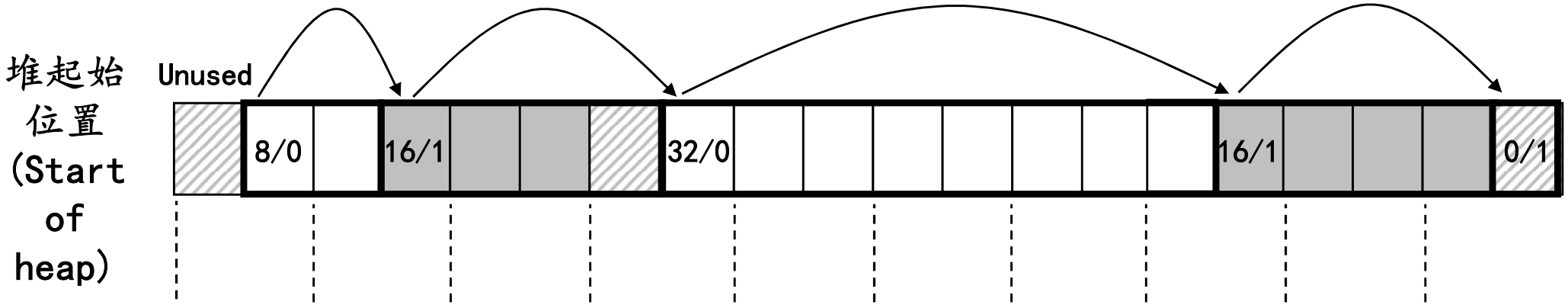
已分配块和空闲块的格式



a = 1: 已经分配的
a = 0: 空闲的

Payload:应用程序数据
(只包括已分配的块**)**

详细的隐式空闲链表示例



双字对齐 (Double-word aligned)

已分配块 (Allocated blocks)：用阴影表示
空闲块 (Free blocks)：用空白表示
块头 (Header)：标注为 “块大小/分配位”，单位是字节

隐式链表：寻找一个空闲块

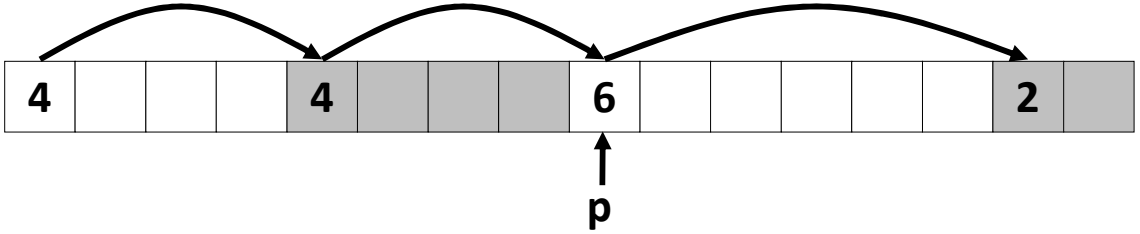
- 首次适配（First fit）：
  - 从链表开头开始查找，选择第一个能放下的空闲块。
  - 代码逻辑：

```
p = start;
while ((p < end) &&    // 没到末尾
       ((*p & 1) ||    // 已被分配
       (*p <= len)))   // 太小
    p = p + (*p & -2);  // 移到下一个块
```

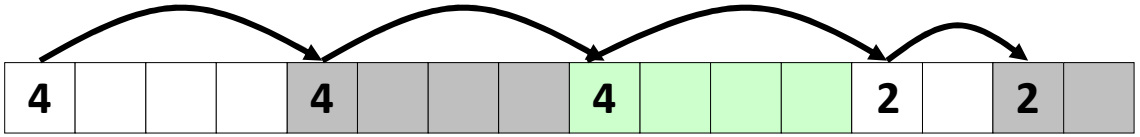
  - 时间复杂度可能是线性的，取决于块的数量（已分配+空闲）。
  - 实际中容易导致开头出现“碎片”。
- 下次适配（Next fit）：
  - 类似首次适配，但从上一次搜索结束的地方继续找。
  - 比首次适配快一些，因为避免了重复扫描没用的块。
  - 一些研究表明它的碎片化可能更严重。
- 最佳适配（Best fit）：
  - 在链表中查找，选择最合适的空闲块：能放下且剩余空间最少的。
  - 能保持碎片较小，通常提高内存利用率。
  - 但是运行速度通常比首次适配更慢。

隐式链表：在空闲块中分配

- 在空闲块中分配：分裂（splitting）
  - 由于申请的内存可能比空闲块要小，所以我们可能希望把空闲块拆分成两部分。



addblock(p, 4)



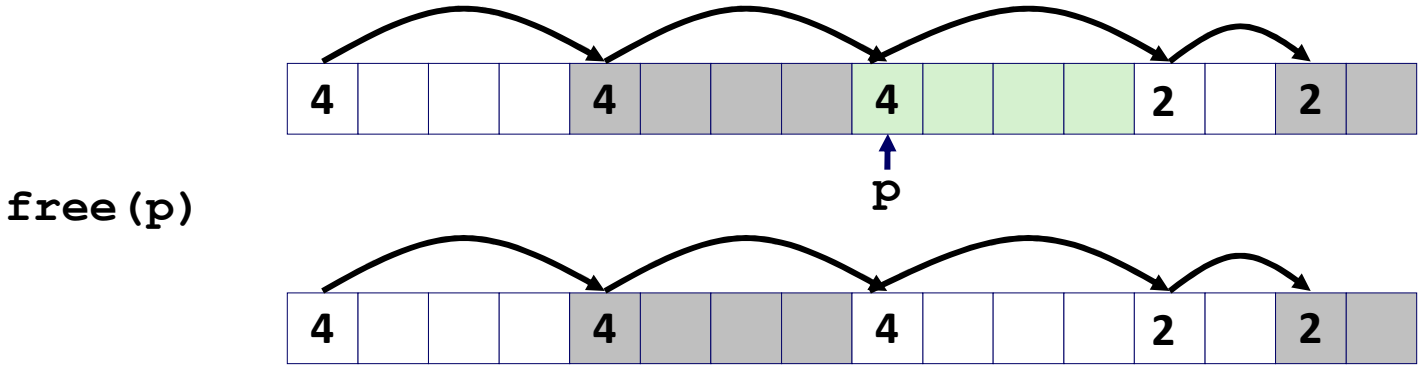

```
void addblock(ptr p, int len) {
    int newsize = ((len + 1) >> 1) << 1;    // 向上取偶数
    int oldsize = *p & -2;                  // 屏蔽最低位，得到原始大小
    *p = newsize | 1;                       // 设置新大小，并标记为已分配
    if (newsize < oldsize)
        *(p + newsize) = oldsize - newsize; // 剩余部分作为空闲块
}
```

隐式链表：释放一个块

- 最简单的实现方式：
  - 只需要清除“已分配”标志位

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

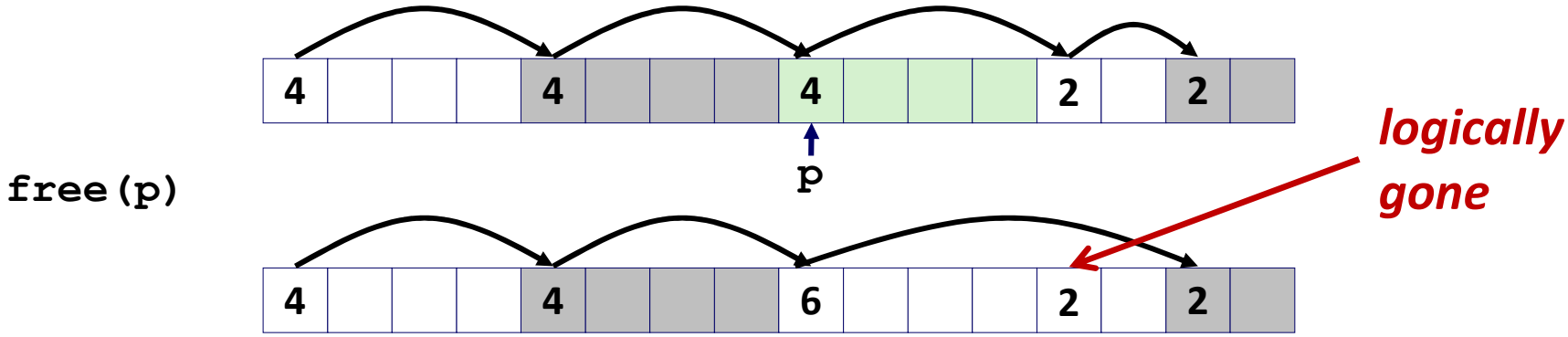
  - 但是，这可能会导致“虚假碎片化”（false fragmentation）。



有足够的空闲空间，但分配器没办法利用它。

隐式链表：合并块（Coalescing）

- 合并（coalesce）：如果相邻的块是空闲的，就把它们合并成一个更大的空闲块。
  - 例如：先和下一个空闲块合并。

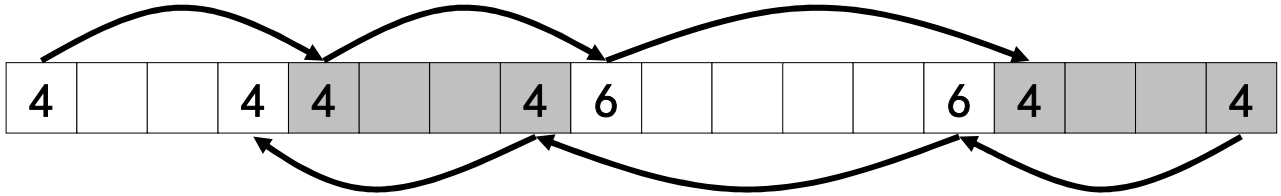



```
void free_block(ptr p) {
    *p = *p & -2;          // 清除分配标志
    next = p + *p;         // 找到下一个块
    if ((*next & 1) == 0) // 如果下一个块是空闲的
        *p = *p + *next; // 把下一个块的大小加到当前块
}
```

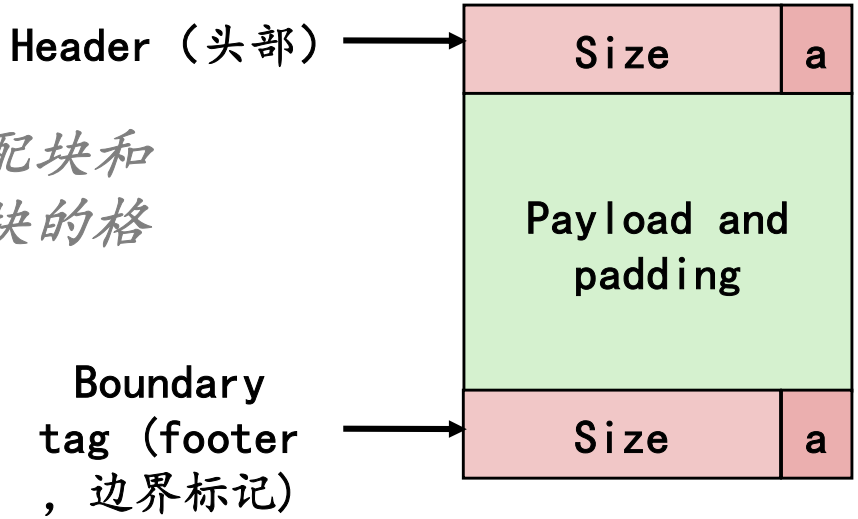
- 但是，如何与前一个块进行合并呢？

隐式链表：双向合并

- 边界标记（Boundary tags） [Knuth]
  - 在空闲块的“底部”（即块的末尾）复制一份大小/分配状态信息。
  - 这样就可以反向遍历链表，从后往前查找，但需要额外的空间。
  - 这是一个非常重要且通用的技术！



已分配块和空闲块的格式

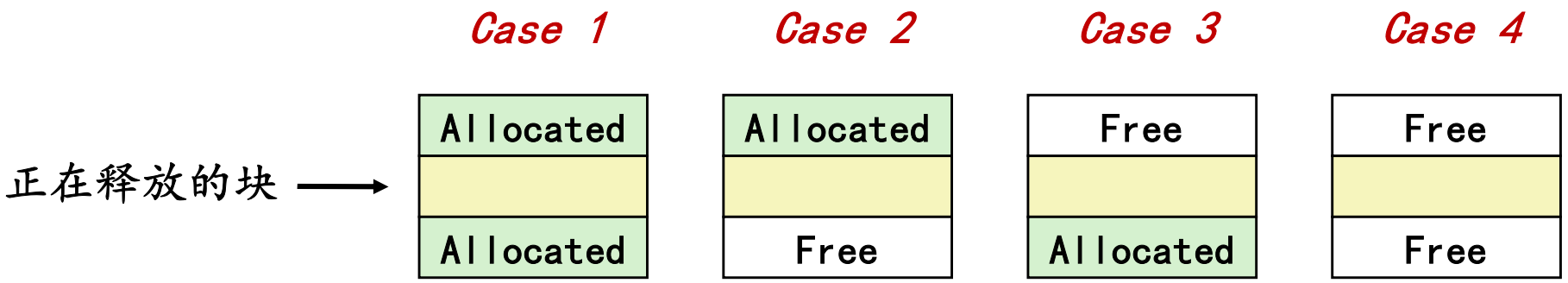


a = 1: 已经分配的
a = 0: 空闲的

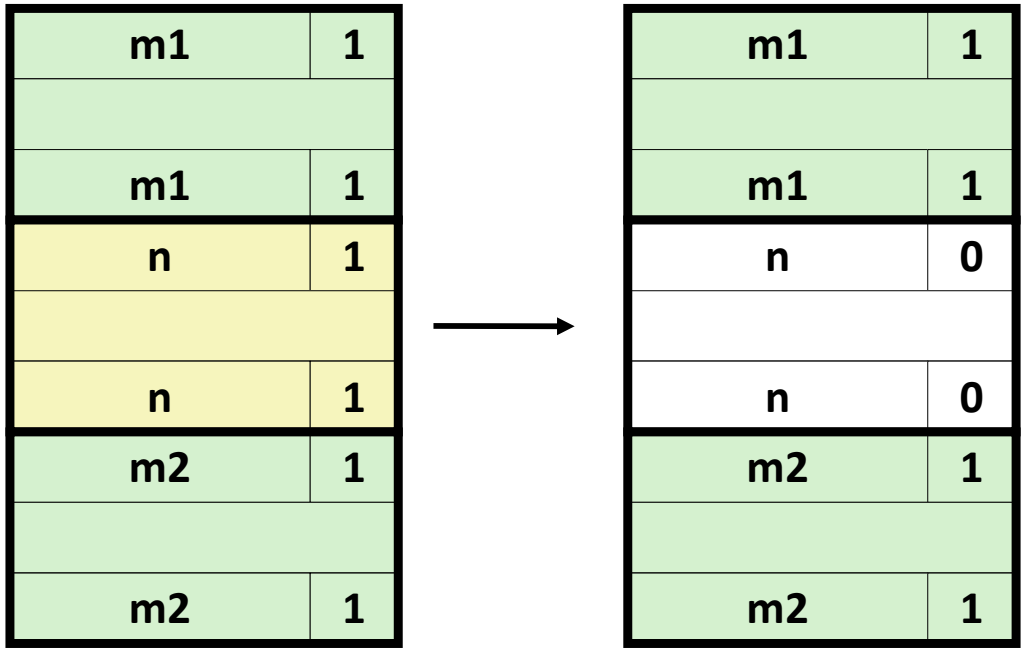
Size: 块大小

Payload:应用程序数据
(只包括已分配的块)

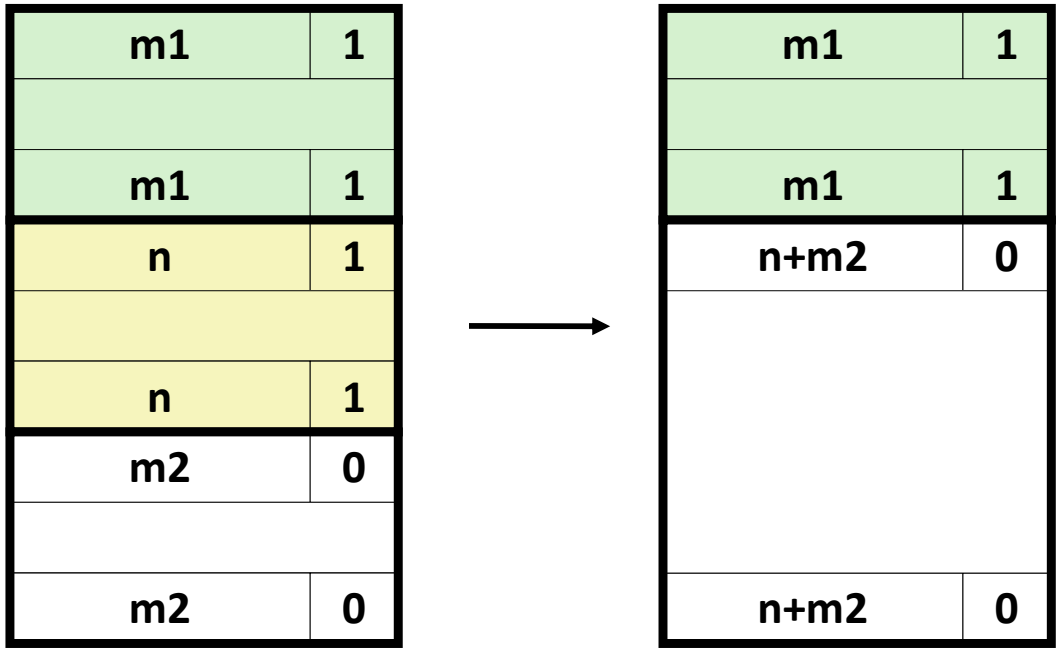
常数时间合并


Case 1
Case 2
Case 3
Case 4
正在释放的块
Allocated
Allocated
Allocated
Free
Free
Allocated
Free
Free

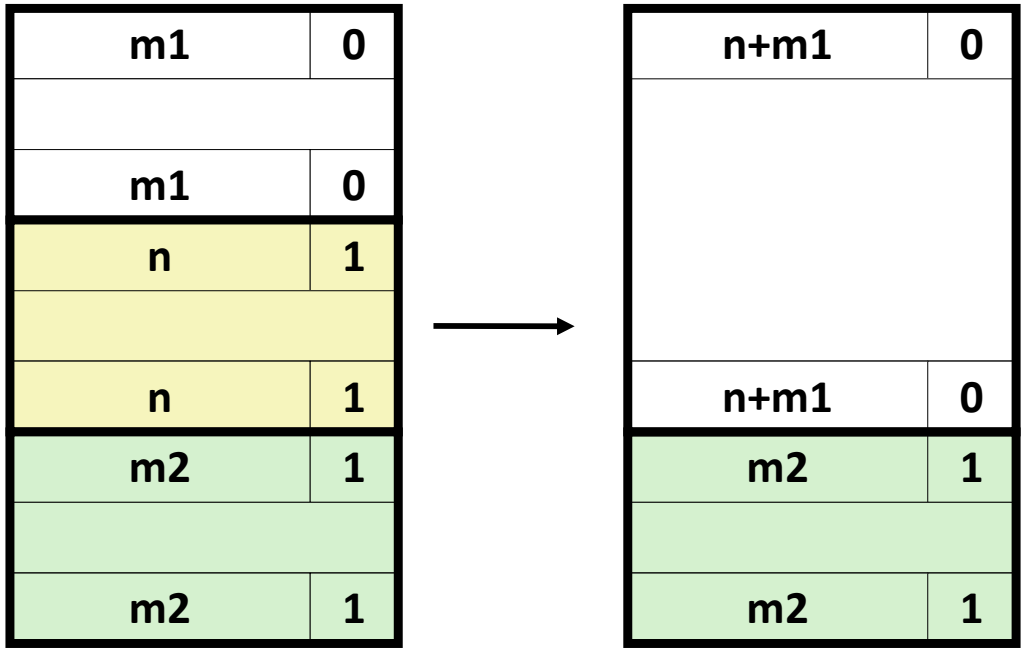
常数时间合并 (Case 1)



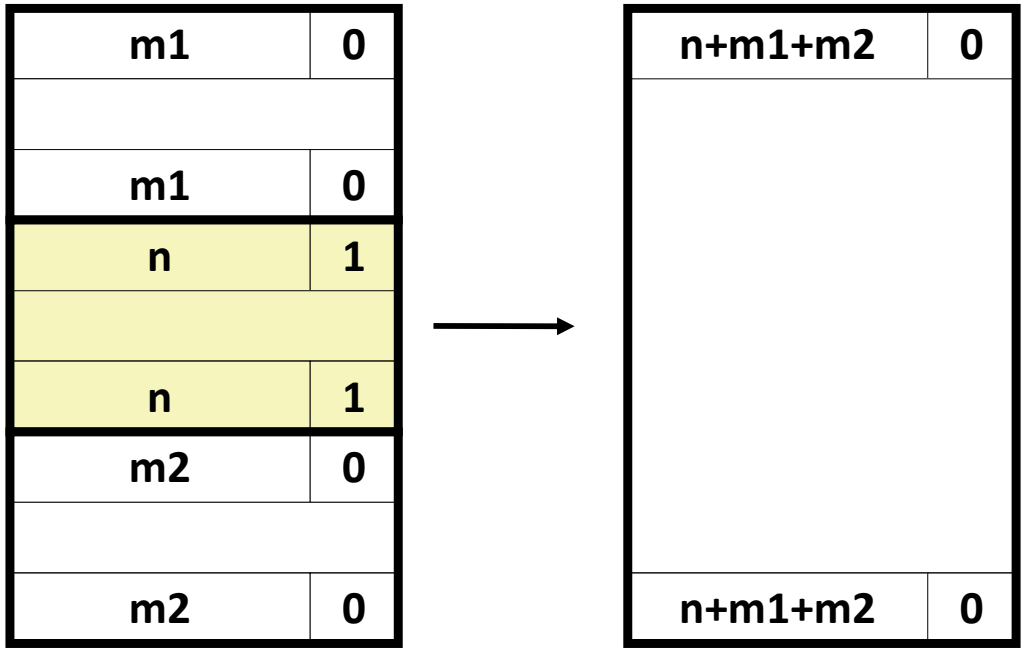
常数时间合并 (Case 2)



常数时间合并 (Case 3)



常数时间合并 (Case 4)



边界标记的缺点

- 内部碎片化
- 是否可以优化？
  - 哪些块需要尾部标记（footer tag）？
  - 这意味着什么？

关键分配策略总结

- 分配策略（Placement policy）：
  - 常见方法：首次适配（first-fit）、下次适配（next-fit）、最佳适配（best-fit）等。
  - 它们在“吞吐量较低”和“碎片较少”之间进行权衡。
  - 分离空闲链表（segregated free lists，会在下一讲展开）能够近似最佳适配策略，而不需要遍历整个空闲链表。
- 分割策略（Splitting policy）：
  - 什么时候需要把空闲块拆分？
  - 我们愿意容忍多少内部碎片？
- 合并策略（Coalescing policy）：
  - 立即合并（Immediate coalescing）： 每次调用 free 时立即合并。
  - 延迟合并（Deferred coalescing）： 为了提升性能，可以推迟合并，直到真正需要时再进行。例如：
    - 在 malloc 扫描空闲链表时顺便合并。
    - 当外部碎片量超过某个阈值时触发合并。

隐式链表：总结

- 实现：非常简单
- 分配成本（Allocate cost）：
  - 最坏情况下是线性时间
- 释放成本（Free cost）：
  - 最坏情况下是常数时间
  - 即使有合并（coalescing）操作
- 内存使用：
  - 依赖于放置策略（Placement policy）
  - 比如首次适配、下次适配、最佳适配
- 在实际的 malloc/free 实现中并不会使用隐式链表
  - 因为分配是线性时间的，效率太低
  - 但在很多特殊用途的场景下仍然会使用
- 然而，分割（splitting）和边界标记合并（boundary tag coalescing）的概念
  - 是所有分配器都通用的