第9章 虚拟内存: 动态内存分配
基 本 概 念

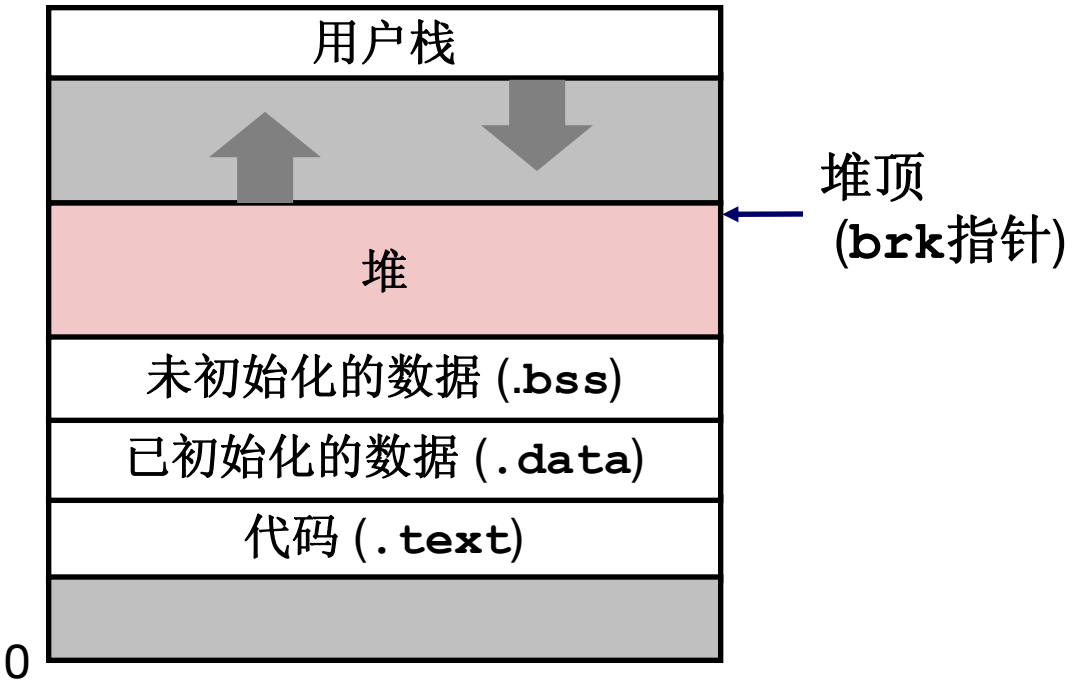
教　师：　夏文
计算机科学与技术学院
硬件与系统教研室
哈尔滨工业大学 深圳

主要内容

- 基本概念
- 隐式空闲列表

Dynamic Memory Allocation 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者一个进程的虚拟内存区域，称为*堆*.



Dynamic Memory Allocation
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快
    - 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。

The `malloc` Package

```
#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）
  - If `size == 0`, returns NULL
- 出错: 返回 NULL (0)， 同时设置 `errno`

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

- 将p指向的块返回到可用内存池
- `p` 必须 `malloc`、 `realloc`或`calloc`已分配块的起止地址

Other functions

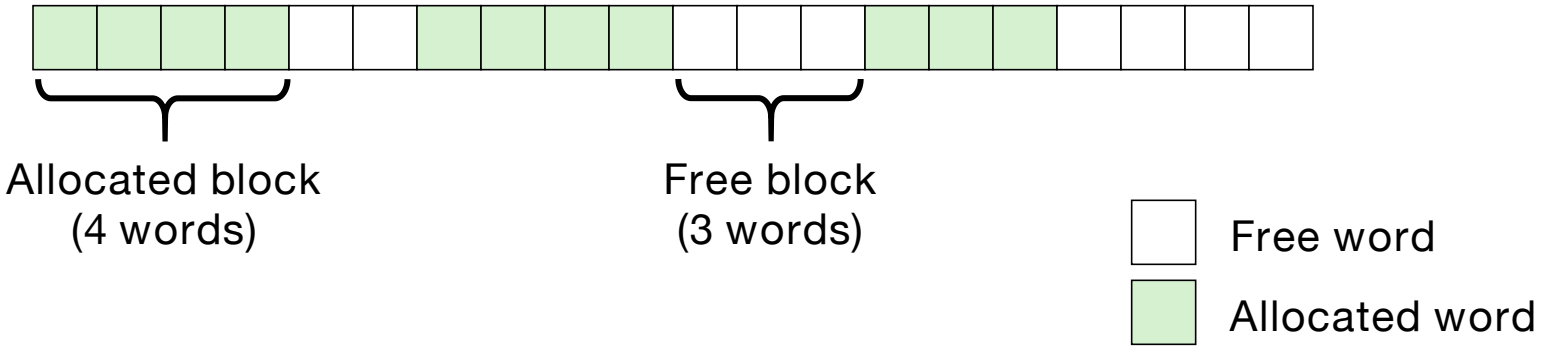
- `calloc`: `malloc`的另一版本，将已分配块初始化为`0`.
- `realloc`: 改变之前分配块的大小.
- `sbrk`: 分配器隐含地扩展或收缩堆

`malloc` Example　　malloc示例

```c
#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);
}
```

Assumptions Made in This Lecture
本节中的假设

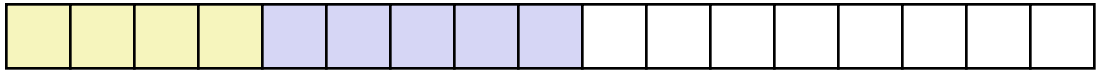
- 内存以字为单位.
- 字是int类型的.



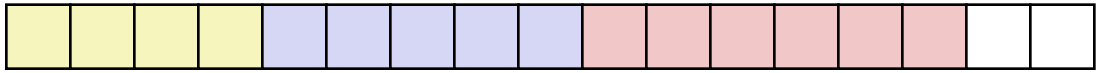
Allocation Example 分配示例

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

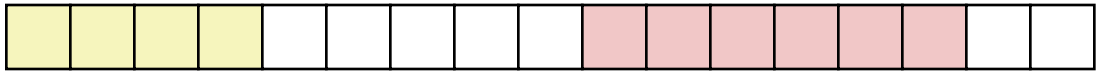
```
p2 = malloc(5)
```



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



```
free(p2)
```



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

Constraints 限制

- Applications 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

- Allocators 分配器
  - 无法控制分配块的数量或大小
  - 立即响应 `malloc` 请求
    - 比如, 不允许分配器重新排列或者缓冲请求
  - 必须从空闲内存分配块
  - 必须对齐块，使得它们可以保护任何类型的数据对象
    - 8字节 (x86) or 16字节 (x86-64) 对齐在 Linux 上框
  - 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
    - 比如, 压缩已分配块的技术是不允许使用的

Performance Goal: Throughput
性能目标：吞吐量

- 假定n个分配和释放请求的某种序列:
  - $R_0, R_1, ..., R_k, ... , R_{n-1}$
- 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:
  - 每个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

Performance Goal: Peak Memory Utilization
性能目标：最大化内存利用率

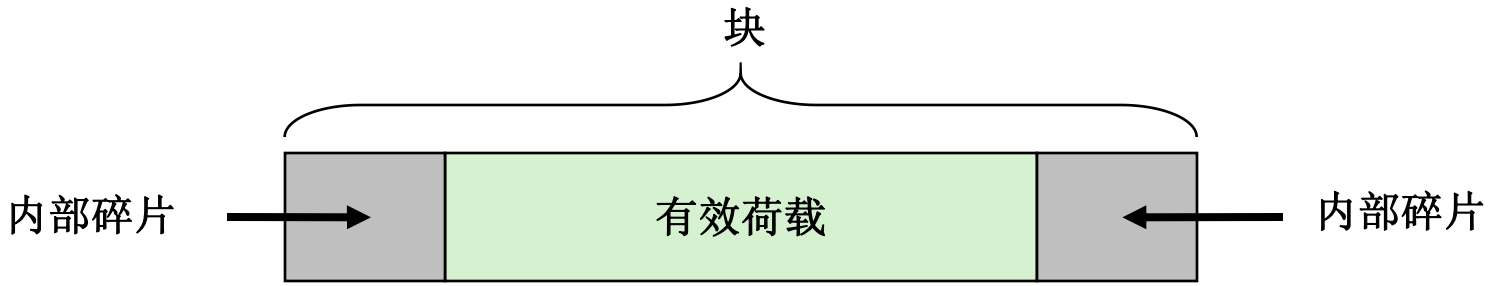
- 给定n个分配和释放请求的某种顺序:
  - $R_0, R_1, ..., R_k, ... , R_{n-1}$
- *定义: 聚集有效载荷* (Aggregate payload) P_k
  - `malloc(p)` 结果是一个有效载荷p字节的块
  - 请求 R_k 完成后, *聚集有效载荷* P_k 为当前已分配的块的有效载荷之和
- *定义: 堆的当前的大小* H_k
  - 假设 H_k 是单调非递减的
    - 比如, 只有分配器使用`sbrk`时堆才会增大
- *定义: 前* k+1 *个请求的峰值利用率*
  - $U_k = (\max_{i<=k} P_i) \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

Fragmentation 碎片

- *碎片化* 导致内存利用率低
  - *内部* 碎片
  - *外部* 碎片

Internal Fragmentation 内部碎片

- 对一个给定块, 当有效荷载小于块的大小时会产生*内部碎片*



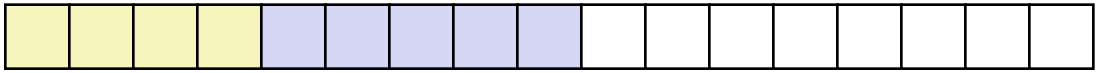
- 产生原因
  - 维护数据结构产生的开销
  - 增加块大小以满足对齐的约束条件
  - 显式的策略决定
    (比如, 返回一个大块以满足一个小的请求)
- 只取决于*以前*请求的模式
  - 易于量化

External Fragmentation 外部碎片

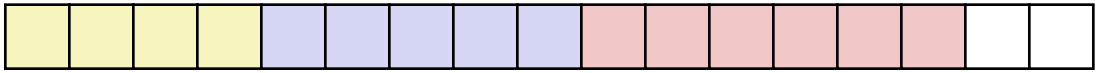
- 是当空闲内存合计起来足够满足一个分配请求，但是没有一个独立的空闲块足够大可以来处理这个请求时发生的。

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

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



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



```
free(p2)
```

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

Oops! (what would happen now?)

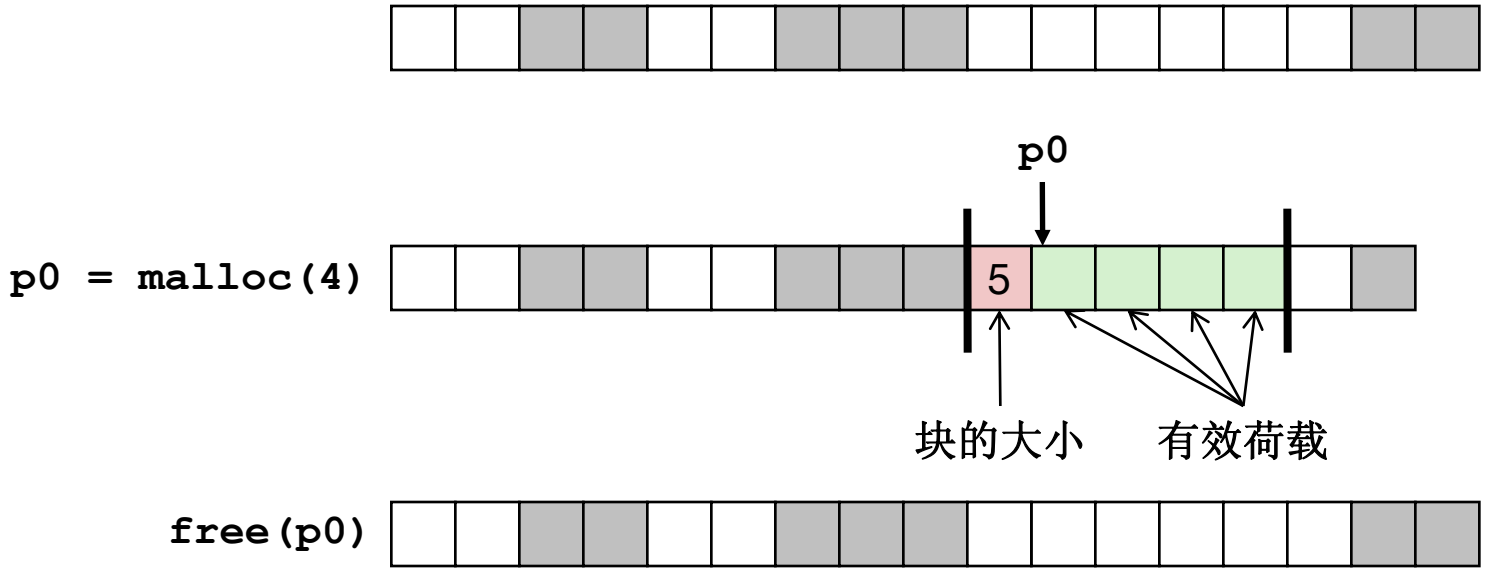
- 取决于将来请求的模式
  - 难以量化

Implementation Issues 实现问题

- 我们如何知道一个指针可以释放多少内存?
- 我们如何记录空闲块（区分空闲与分配）？
- 将一个新分配的块放置到某个比较大的空闲块后，我们如何处理这个空闲块中的剩余部分?
- 我们如何选择一个空闲块去分配 – 很多都合适呢?
- 我们如何处理一个刚刚被释放的块—前后也有空闲呢?

Knowing How Much to Free
知道释放多少

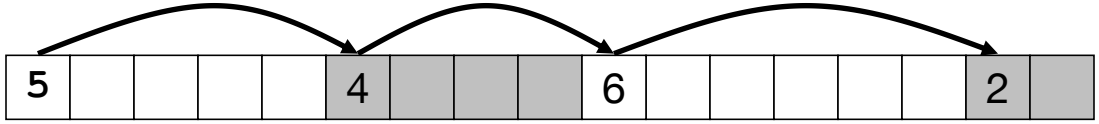
- Standard method 标准方法
  - 在块的前面的word中记录该块的长度.
    - 这个 word 被称为 *头部*
  - 每个被分配块都需要一个这样的 “word”



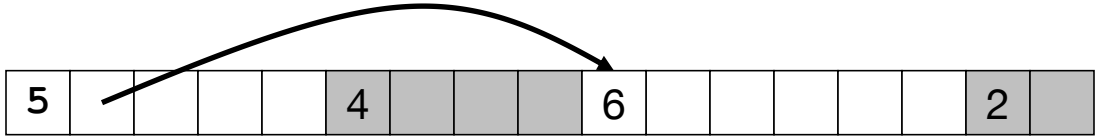
Keeping Track of Free Blocks

记录空闲块

- 方法 1: *隐式空闲链表* (Implicit list) 通过头部中的大小字段—隐含地连接所有块



- 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: 按*块按大小排序—平衡树*
  - 在每个空闲块中使用一个带指针的平衡树，并使用长度作为权值

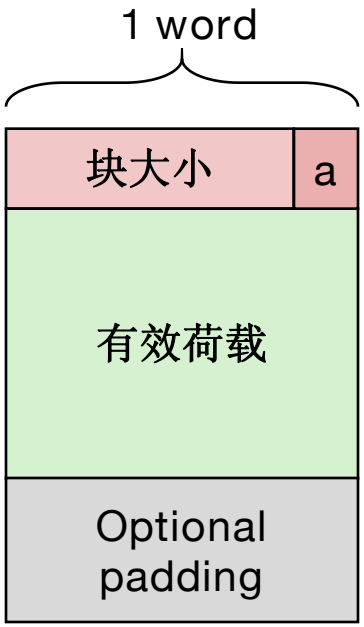
主要内容

- 基本概念
- 隐式空闲链表

Method 1: Implicit List 隐式空闲链表

- 对于每个块我们都需要知道块的大小和分配状态
  - 可以将这些信息存储在两个 words 中: 浪费!
- Standard trick 标准技巧
  - 如果块是对齐的，那么一些低阶地址位总是0
  - 不要存储这些0位，而是使用它作为一个已分配/未分配的标志
  - 读大小字段时，必须将其屏蔽掉

一个简单的堆块的格式

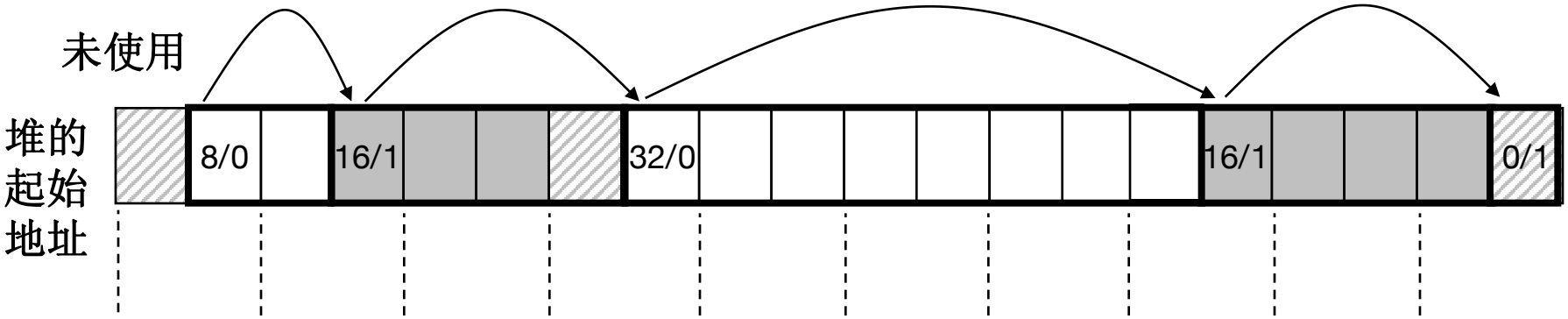


a = 1: 已分配块
a = 0: 空闲块

Size: 块的大小

有效荷载: 应用数据
(只包括已分配的块)

Detailed Implicit Free List Example
用隐式空闲链表法的细节示例



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）

Implicit List: Finding a Free Block
隐式链表法：找到一个空闲块

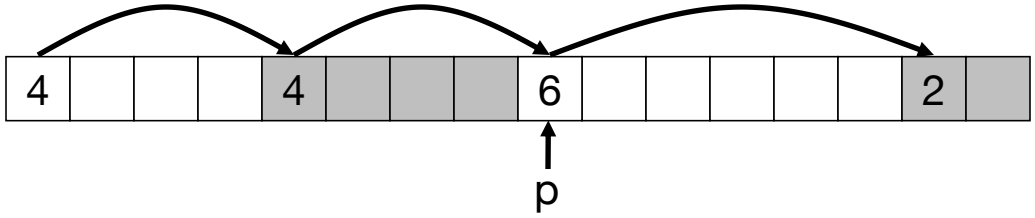
- *首次适配* (First fit):
  - 从头开始搜索空闲链表，选择*第一个*合适的空闲块:

```
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)
```

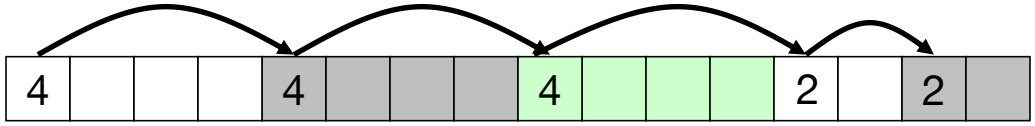
  - 可以取总块数 (包括已分配和空闲块) 的线性时间
  - 在靠近链表起始处留下小空闲块的 “碎片”
- *下一次适配* (Next fit):
  - 和首次适配相似，只是从链表中上一次查询结束的地方开始
  - 比首次适应更快: 避免重复扫描那些无用块
  - 一些研究表明，下一次适配的内存利用率要比首次适配低得多
- *最佳适配* (Best fit):
  - 查询链表，选择一个 *最好的* 空闲块： 适配，剩余最少空闲空间
  - 保证碎片最小——提高内存利用率
  - 通常运行速度会慢于首次适配

Implicit List: Allocating in Free Block
隐式链表----分配空闲块

- 分配空闲块:*分割* (splitting)
  - 既然分配块比空闲块小, 我们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
```

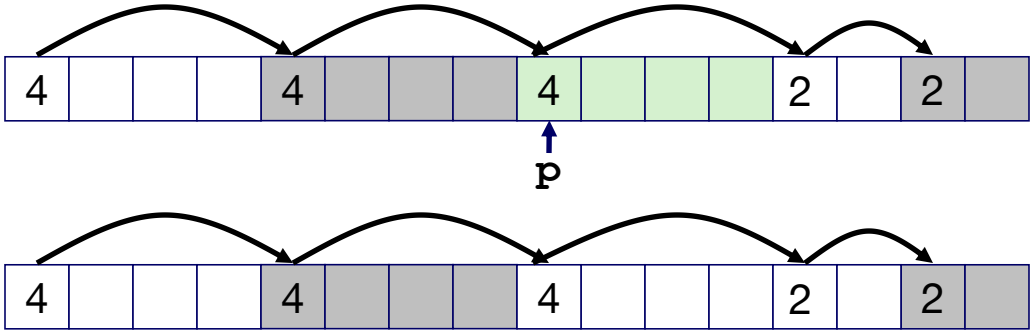
Implicit List: Freeing a Block

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)”



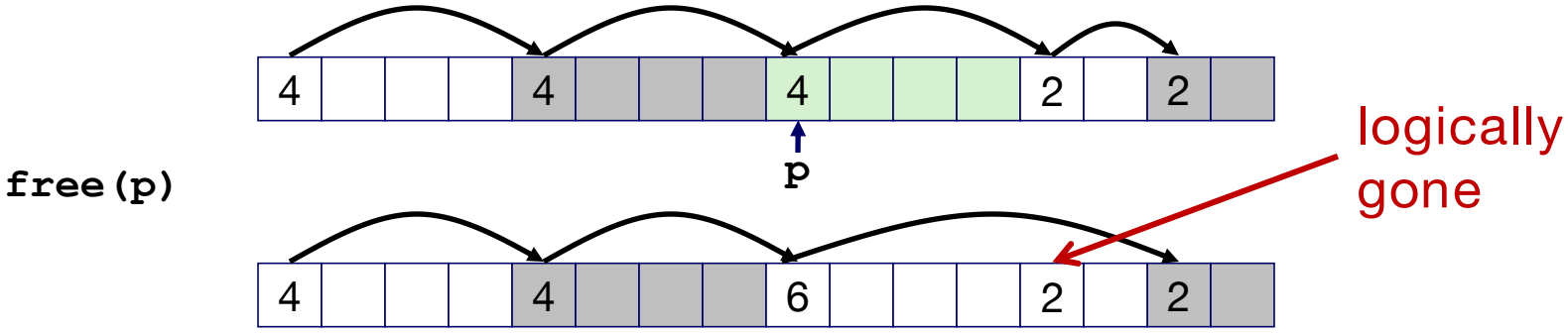
`free(p)`

`malloc(5)` Oops!

已经有足够空间, 但是分配器却无法找到它!

Implicit List: Coalescing 合并

- 合并 (coalesce)：合并相邻的空闲块
  - 和下一个空闲块合并

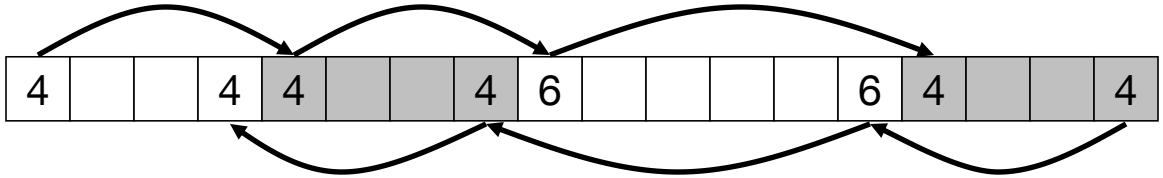



```
void free_block(ptr p) {
    *p = *p & -2;              // clear allocated flag
    next = p + *p;             // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;         // add to this block if
}                              //    not allocated
```

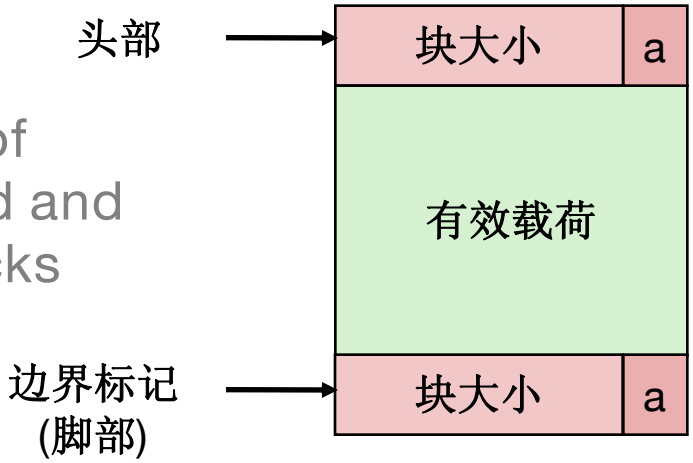
- 如何与前一块合并呢?

Implicit List: Bidirectional Coalescing
双向合并

- *边界标记* (Boundary tags) [Knuth73]
  - 在空闲块的“底部”标记 大小/已分配
  - 允许我们反查 “链表”，但这需要额外的空间
  - 重要且普遍的技术!



Format of allocated and free blocks



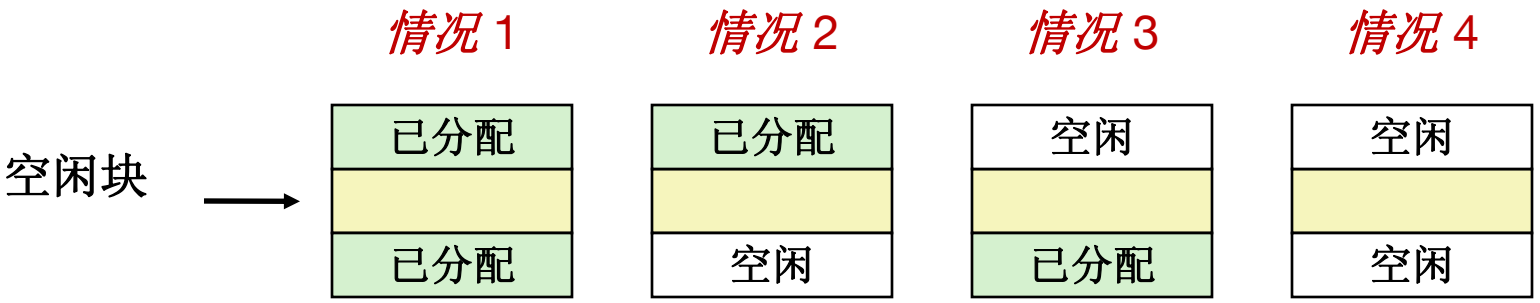
a = 1: 已分配块
a = 0: 空闲块

块大小: 整块的大小

载荷: 应用数据
(只包括已分配块)

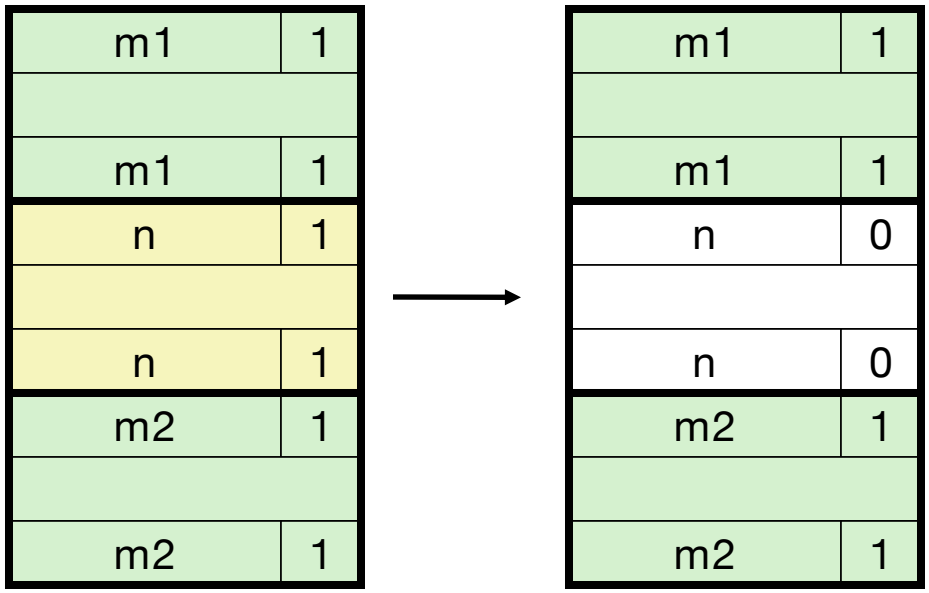
Constant Time Coalescing

合并情况



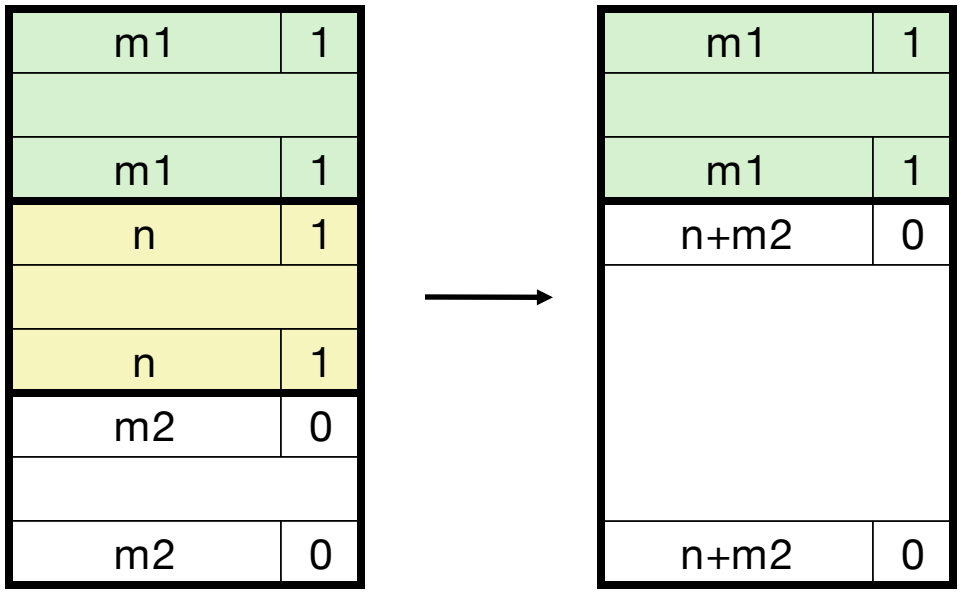
Constant Time Coalescing

合并 (情况 1)



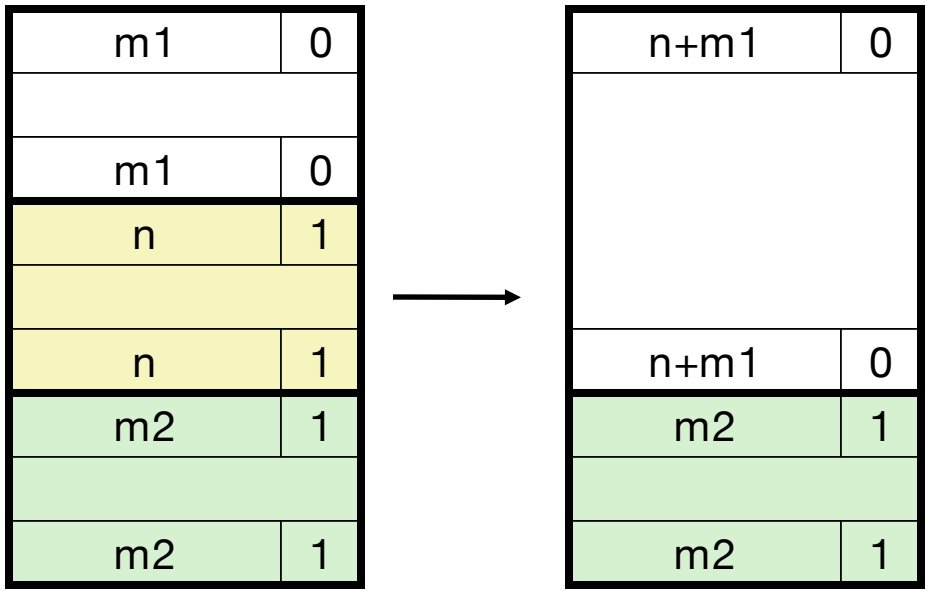
Constant Time Coalescing

合并 (情况 2)



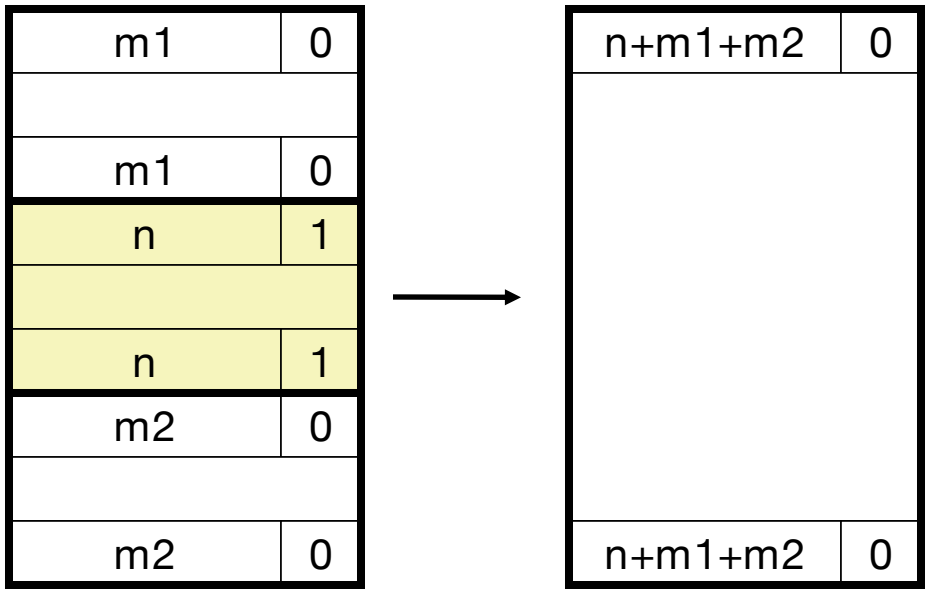
Constant Time Coalescing

合并 (情况 3)



Constant Time Coalescing

合并 (情况 4)



Disadvantages of Boundary Tags
边界标记法的缺陷

- Internal fragmentation 内部碎片
  - 头部脚部双标记，小块浪费空间
- Can it be optimized? 它是最优的么？
  - 把前面块的已分配/空闲位存放在当前块中多出来的地位中！！！
  - 哪些块需要脚标?
  - 有什么作用?

Summary of Key Allocator Policies
关键的分配策略总结

- Placement policy:放置策略
  - 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
- Splitting policy 分割策略:
  - 我们什么时候开始分割空闲块?
  - 我们能够容忍多少内部碎片?
- Coalescing policy合并策略:
  - *立即合并* (Immediate coalescing): 每次释放都合并
  - *延迟合并* (Deferred coalescing): 尝试通过延迟合并，即直到需要才合并来提高释放的性能.例如:
    - 为 `malloc`扫描空闲链表时可以合并
    - 外部碎片达到阈值时可以合并

Implicit Lists: Summary

隐式链表：总结

- 实现: 非常简单
- 分配开销:
  - linear time worst case 最坏情况下是线性时间
- Free cost 释放开销:
  - constant time worst case 最坏情况常数时间
  - even with coalescing 即使合并
- Memory usage 内存使用:
  - will depend on placement policy 取决于分配策略
  - First-fit, next-fit or best-fit 首次适配，下一次适配或最佳适配
- Not used in practice for **`malloc/free`** because of linear-time allocation 由于现行时间分配，没有用于 **`malloc/free`**
  - used in many special purpose applications 用于许多特殊目的的应用

Implicit Lists: Summary

隐式链表：总结

- However, the concepts of splitting and boundary tag coalescing are general to all allocators
  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

内部碎片和外部碎片

- **内部碎片**
  - 对一个给定块，当有效荷载小于块的大小时会产生*内部碎片*
  - 无法被垃圾回收，只能靠free回收

- **外部碎片**
  - 是当空闲内存合计起来足够满足一个分配请求，但是没有一个独立的空闲块足够大可以来处理这个请求时发生的。
  - 可以被合并回收

Hope you enjoyed the CSAPP course!