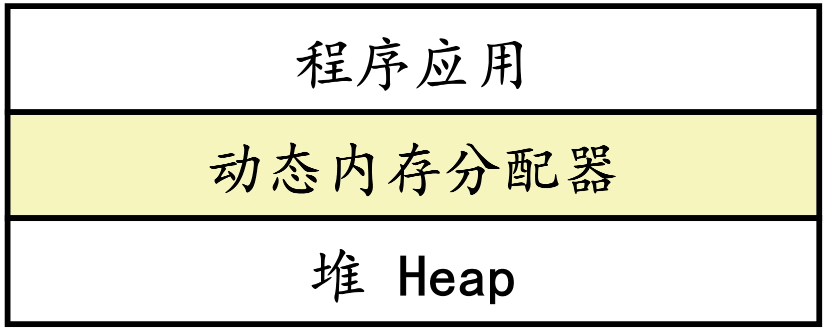
动态内存分配：基本概念
(Dynamic Memory Allocation)

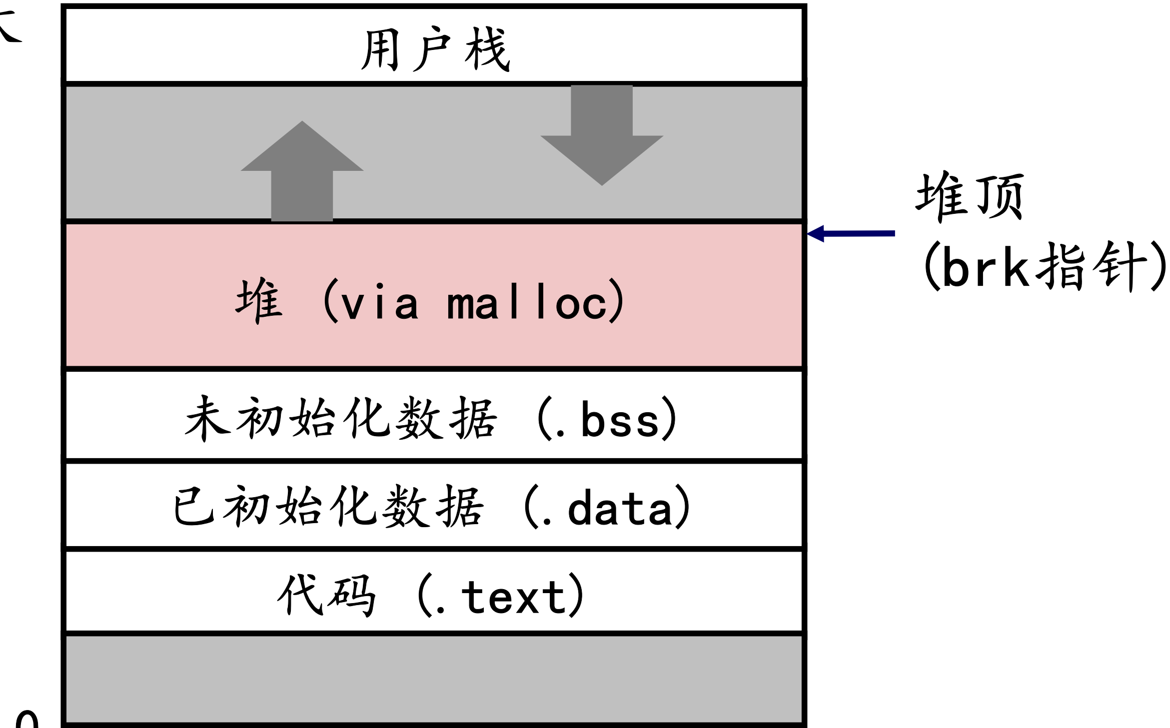
主要内容

- 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

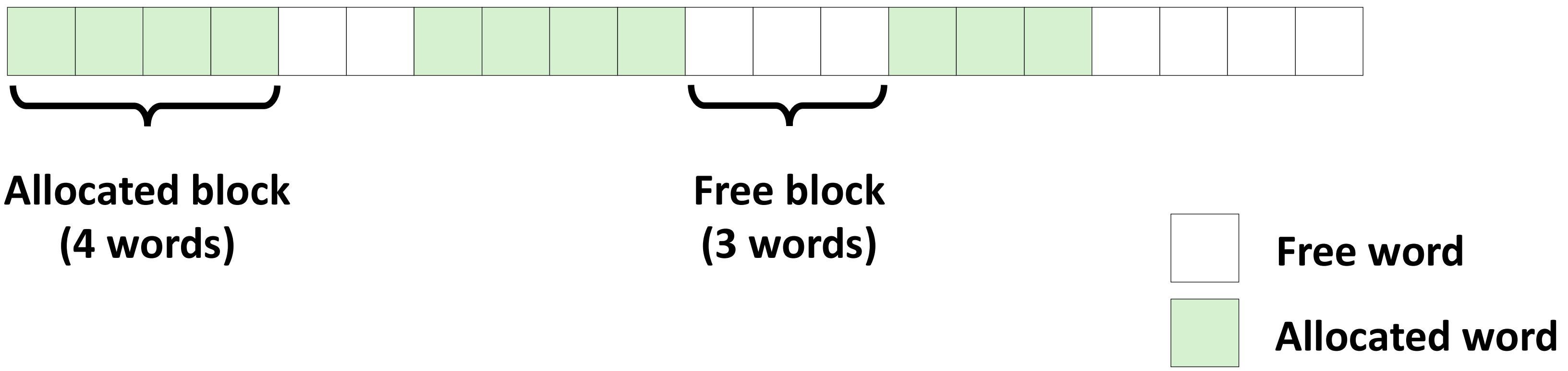
- calloc：malloc 的一个版本，分配的内存会初始化为 0。
- realloc：改变之前已经分配的内存块大小。
- sbrk：分配器在内部调用它来扩展或收缩堆。

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);
}
```

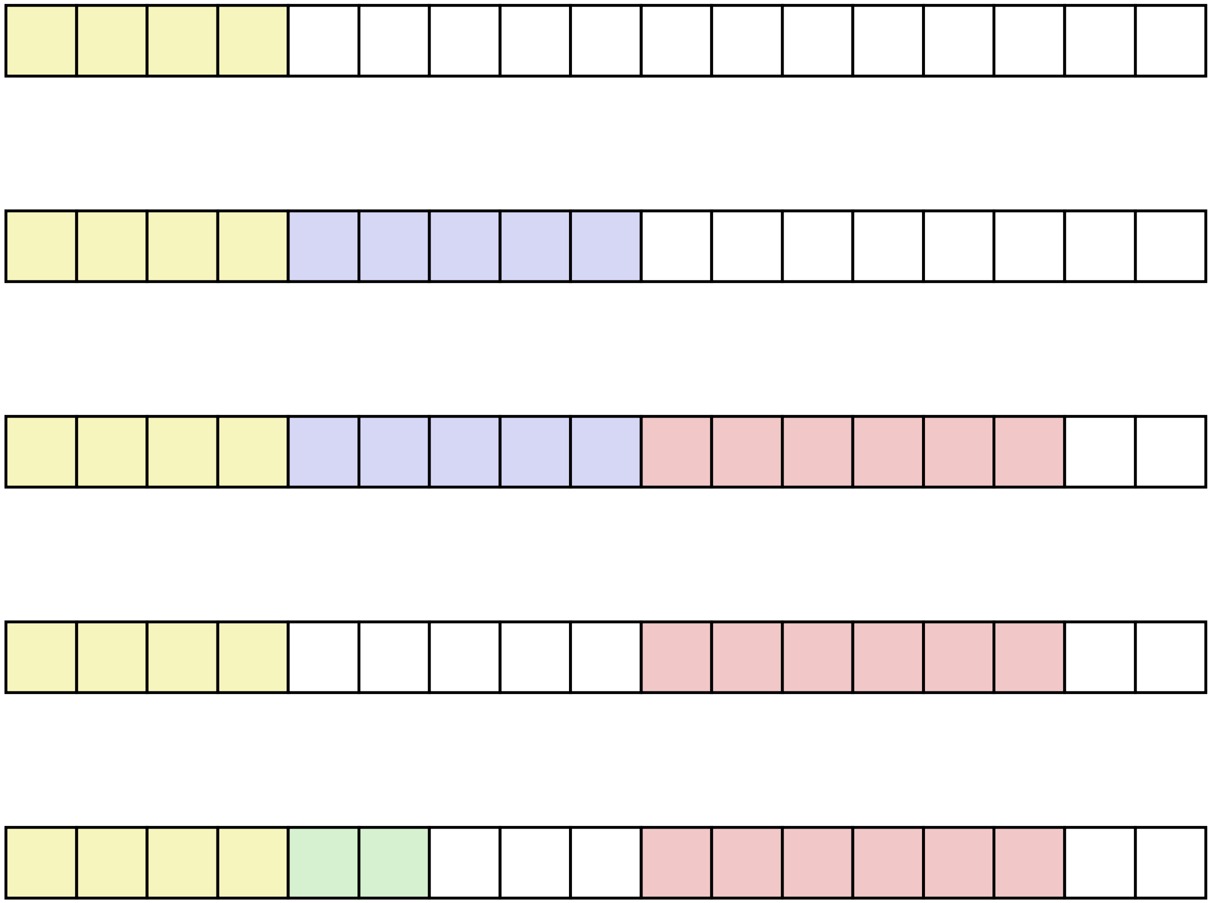
本课堂中的假设

- 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
  - $R_0, R_1, ..., R_k, ... , R_{n-1}$

- 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

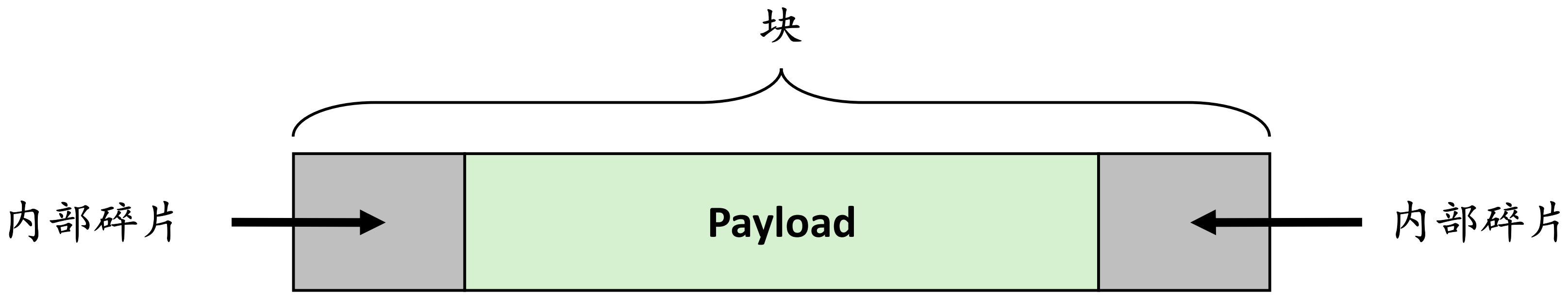
$$U_k = \frac{\max_{i \le k} P_i}{H_k}$$

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）。

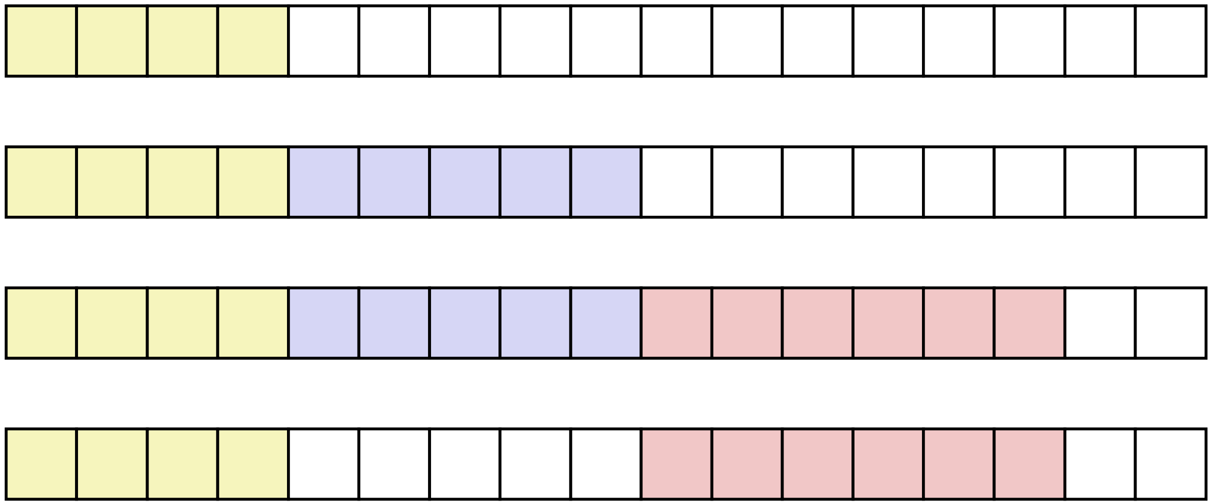


- **成因：**
  - 维护堆数据结构的开销
  - 为对齐（alignment）而填充的字节
  - 明确的策略决策
    （例如：为了满足一个小请求，返回了一个较大的块）
- **特点：**
  - 内部碎片 只依赖于过去的请求模式
  - 因此，它相对容易测量

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？

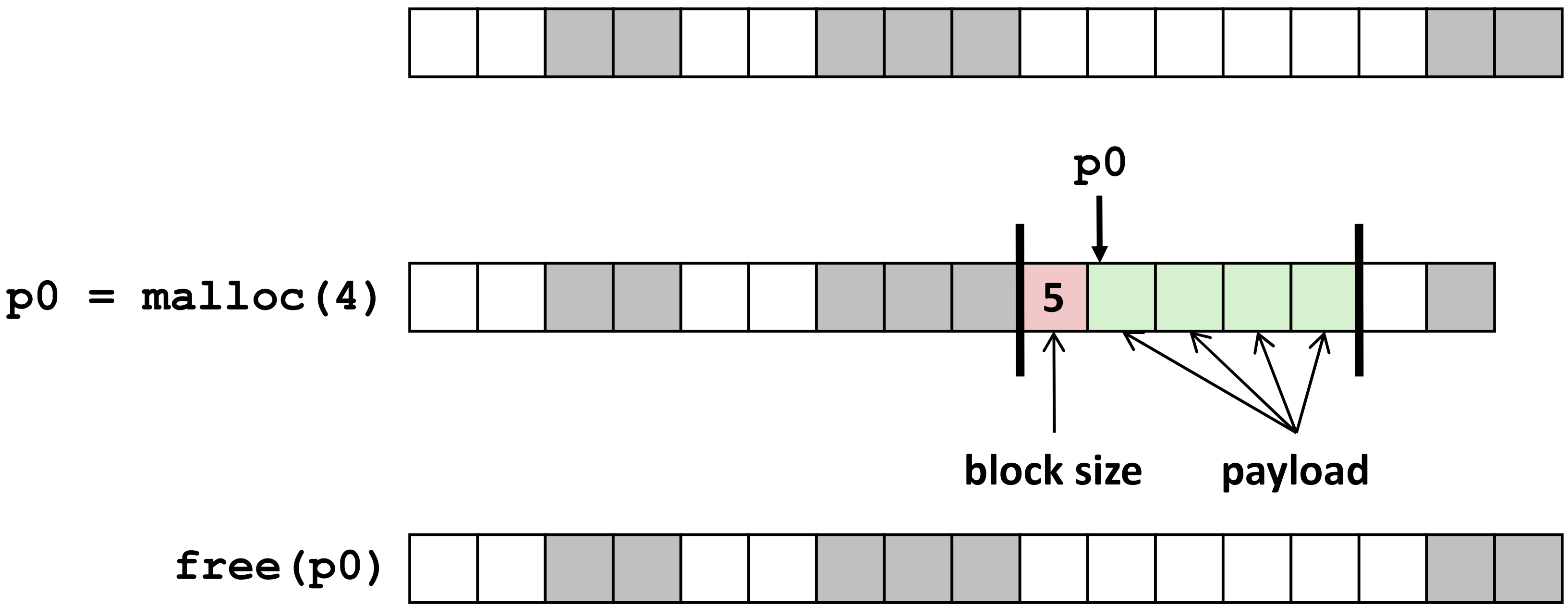
- 外部碎片 依赖于未来请求的模式
  - 因此，很难准确估计

Implementation Issues

- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation -- many might fit?
- How do we reinsert freed block?

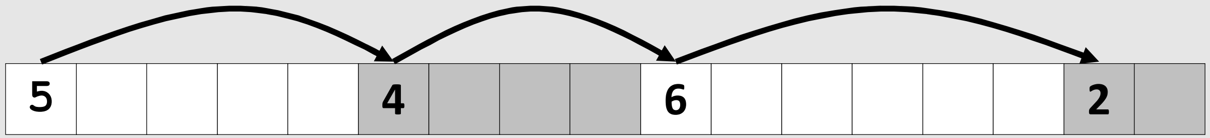
Knowing How Much to Free

- **Standard method**
  - Keep the length of a block in the word preceding the block.
    - This word is often called the ***header field*** or ***header***
  - Requires an extra word for every allocated block

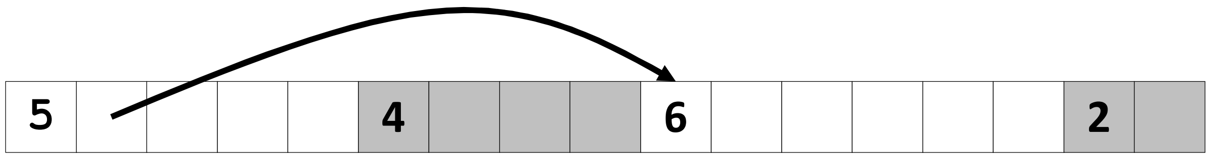


Keeping Track of Free Blocks

- **Method 1:** ***Implicit list*** **using length—links all blocks**



- **Method 2:** ***Explicit list*** **among the free blocks using pointers**



- **Method 3:** ***Segregated free list***
  - Different free lists for different size classes

- **Method 4:** ***Blocks sorted by size***
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

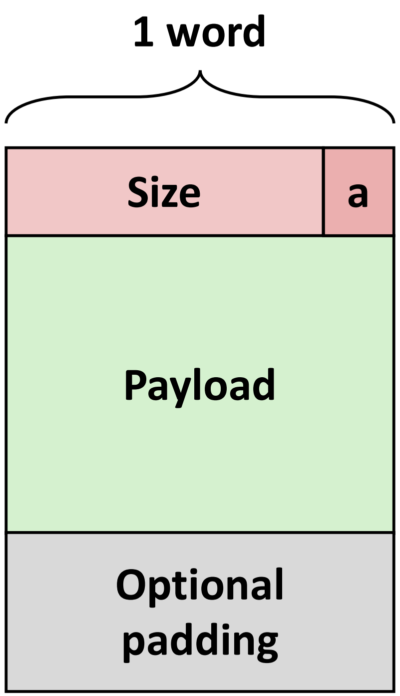
Today

- Basic concepts
- **Implicit free lists**

Method 1: Implicit List

- **For each block we need both size and allocation status**
  - Could store this information in two words: wasteful!
- **Standard trick**
  - If blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size word, must mask out this bit

Format of allocated and free blocks

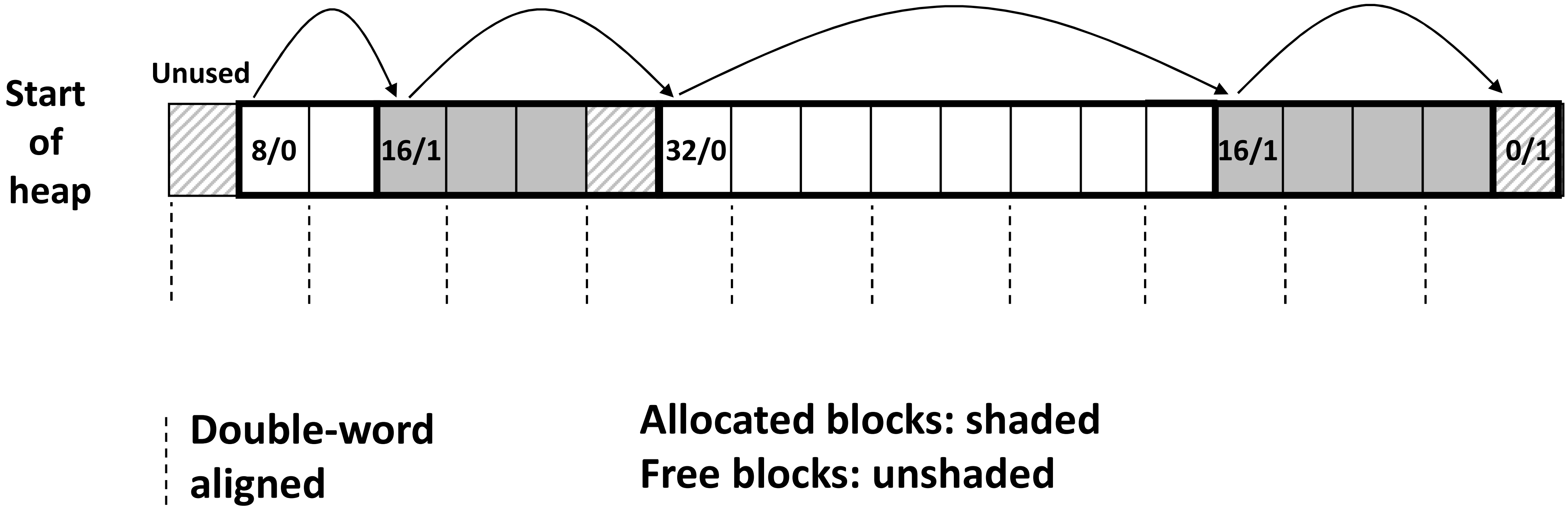


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Detailed Implicit Free List Example



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

Implicit List: Finding a Free Block

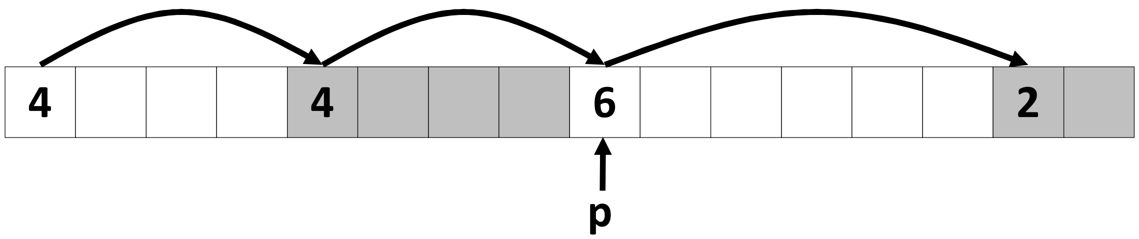
- ***First fit:***
  - Search list from beginning, choose ***first*** free block that fits:

```
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)
```

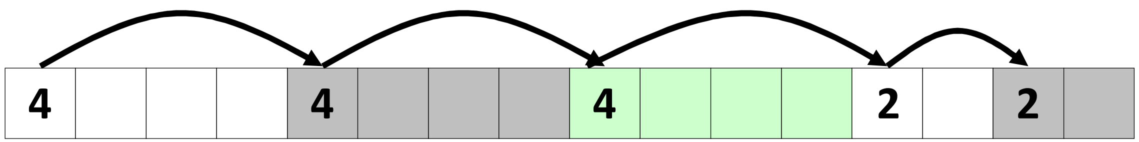
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- ***Next fit:***
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- ***Best fit:***
  - Search the list, choose the ***best*** free block: fits, with fewest bytes left over
  - Keeps fragments small—usually improves memory utilization
  - Will typically run slower than first fit

Implicit List: Allocating in Free Block

- **Allocating in a free block:** ***splitting***
  - Since allocated space might be smaller than free space, we might want to split the block



`addblock(p, 4)`



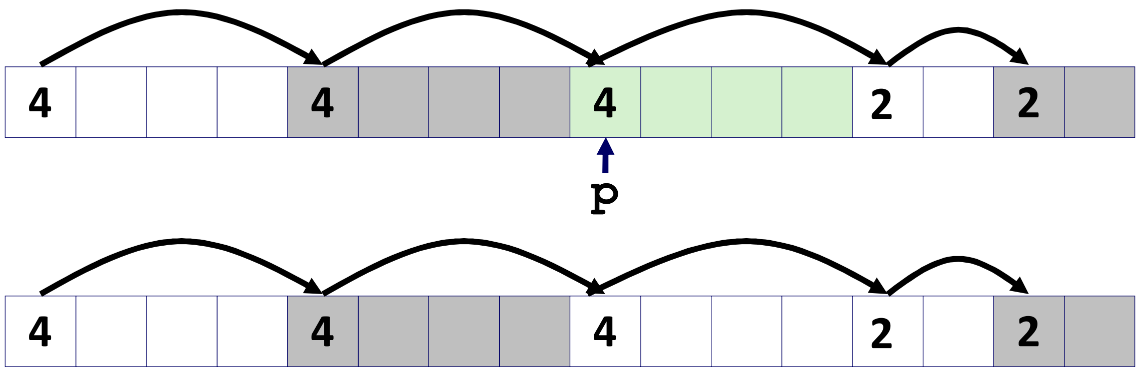

```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;   // round up to even
  int oldsize = *p & -2;                 // mask out low bit
  *p = newsize | 1;                      // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;    // set length in remaining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

Implicit List: Freeing a Block

- **Simplest implementation:**
  - Need only clear the "allocated" flag

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

  - But can lead to "false fragmentation"



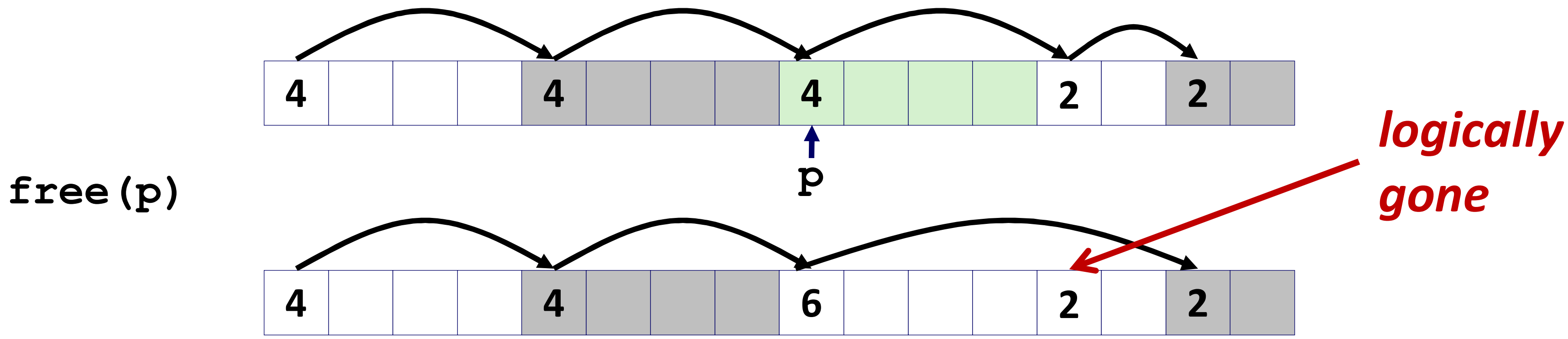
`free(p)`

`malloc(5)` ***Oops!***

There is enough free space, but the allocator won't be able to find it

Implicit List: Coalescing

- **Join *(coalesce)* with next/previous blocks, if they are free**
  - Coalescing with next block

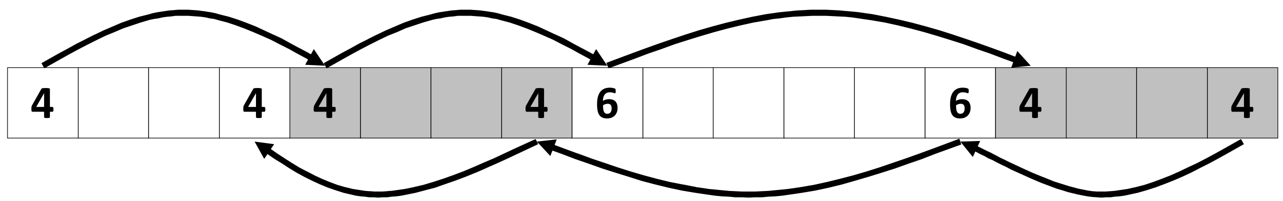



```
void free_block(ptr p) {
    *p = *p & -2;              // clear allocated flag
    next = p + *p;             // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;         // add to this block if
}                              //    not allocated
```

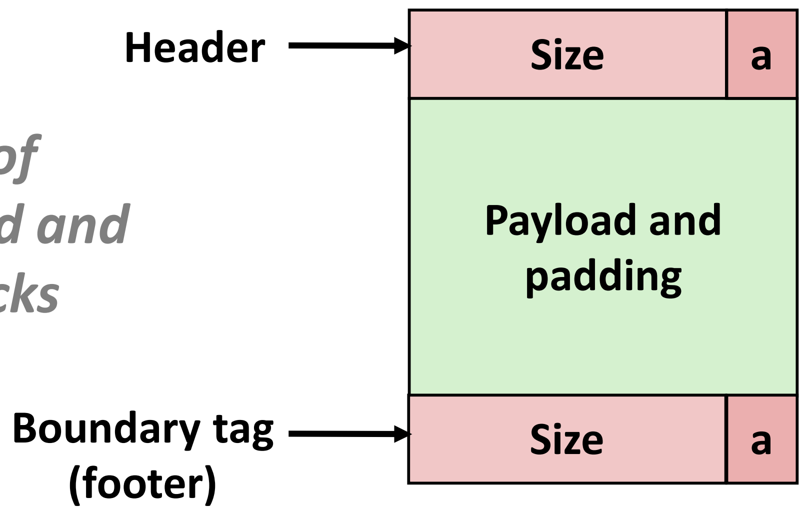
  - But how do we coalesce with *previous* block?

Implicit List: Bidirectional Coalescing

- ***Boundary tags*** [Knuth73]
  - Replicate size/allocated word at “bottom” (end) of free blocks
  - Allows us to traverse the “list” backwards, but requires extra space
  - Important and general technique!



Format of allocated and free blocks

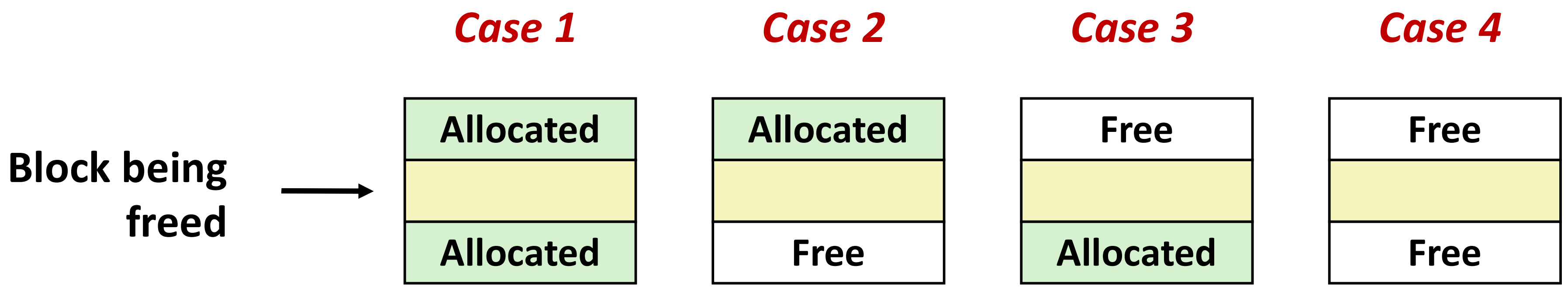


a = 1: Allocated block
a = 0: Free block

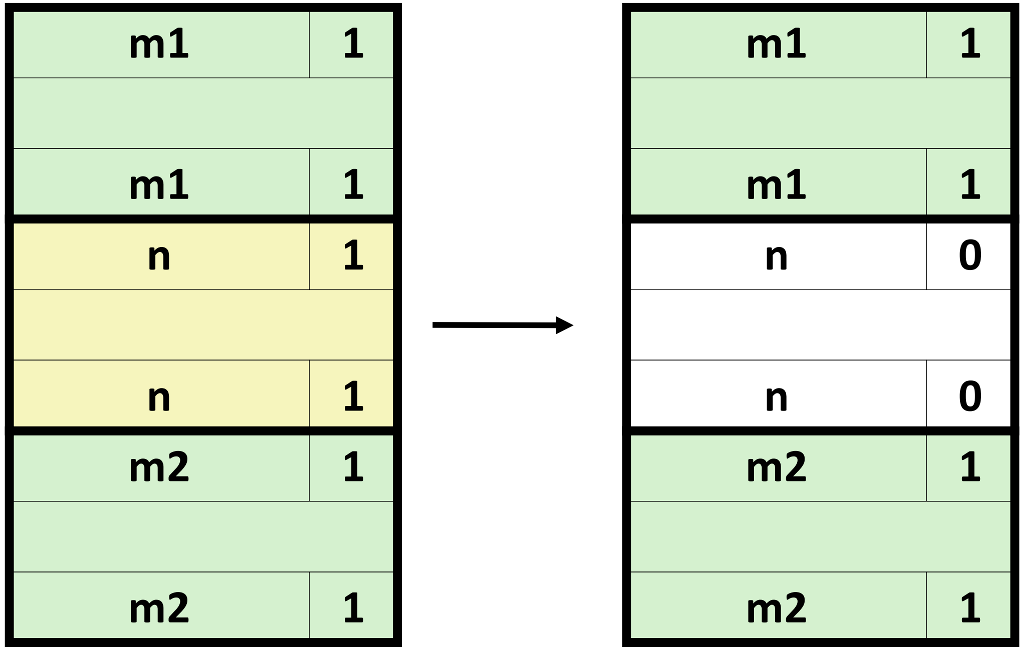
Size: Total block size

Payload: Application data (allocated blocks only)

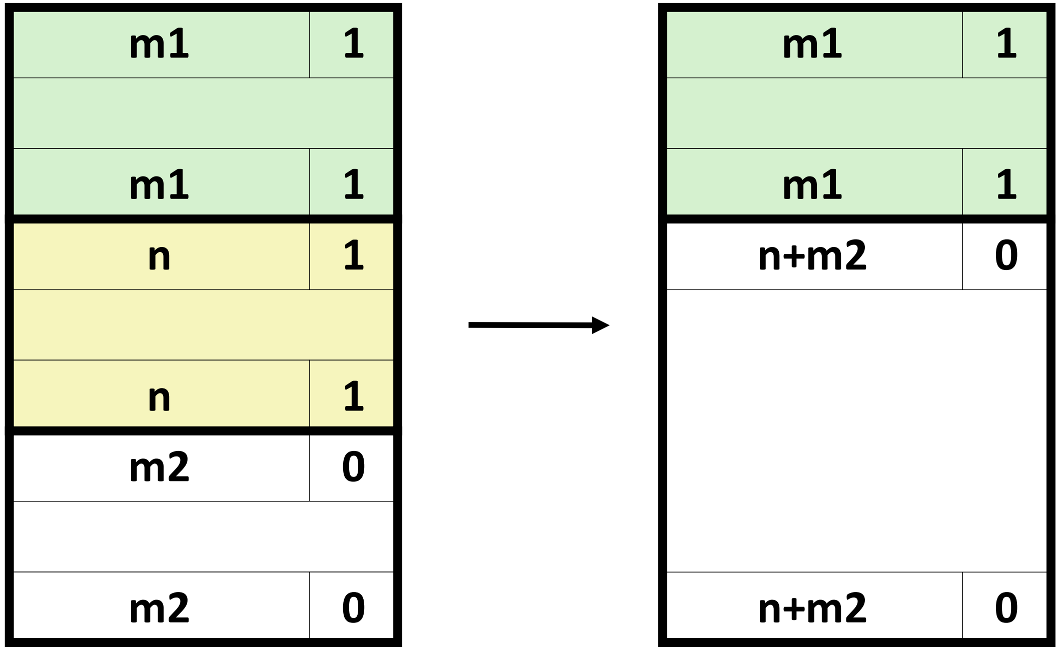
Constant Time Coalescing



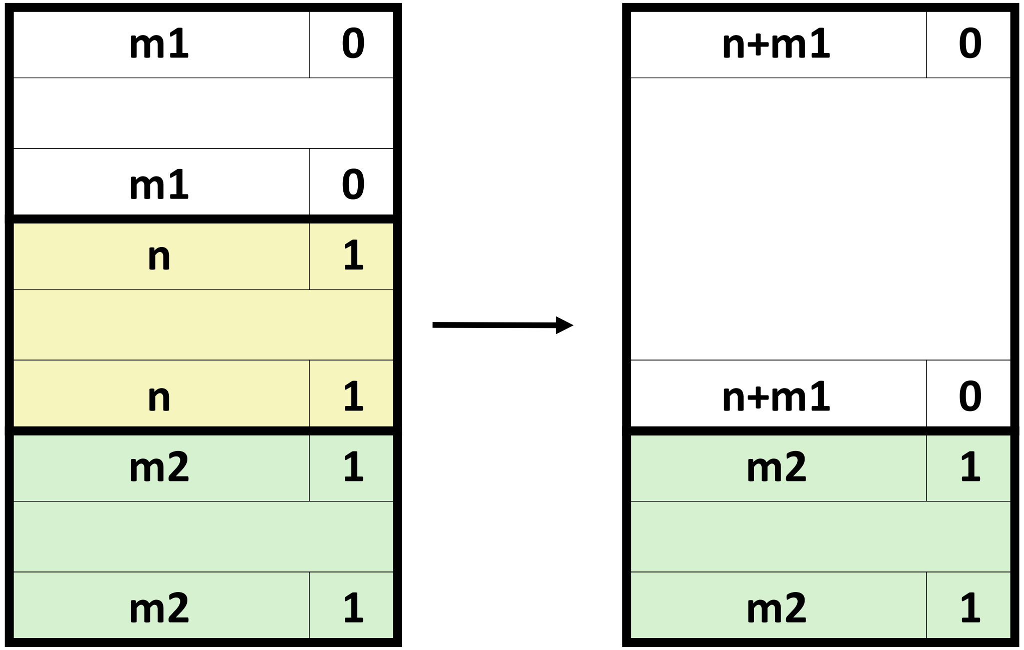
Constant Time Coalescing (Case 1)



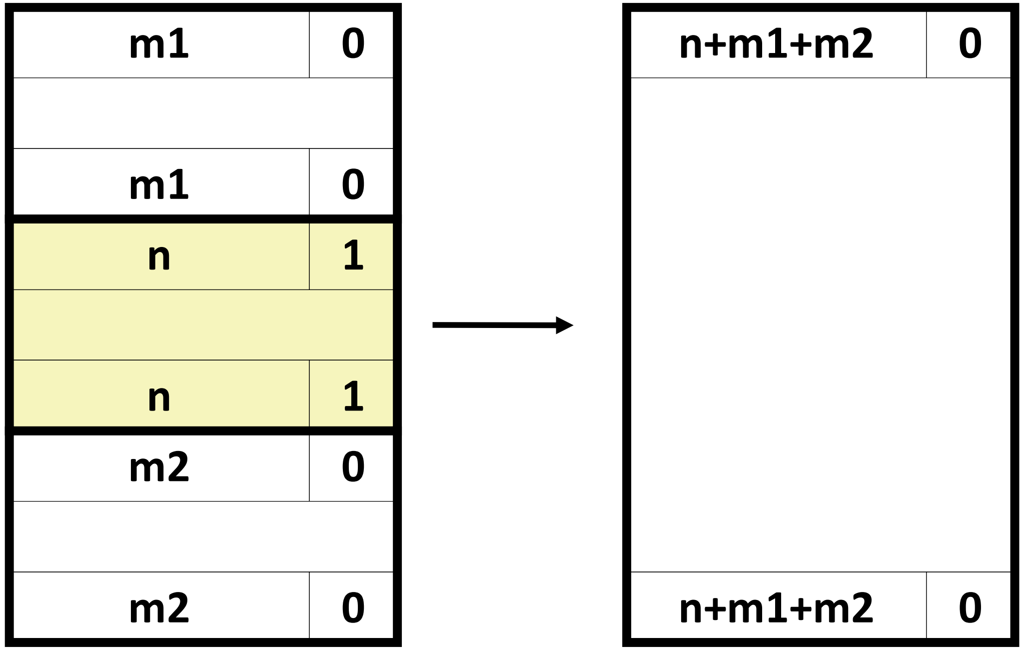
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



Disadvantages of Boundary Tags

- **Internal fragmentation**

- **Can it be optimized?**
  - Which blocks need the footer tag?
  - What does that mean?

Summary of Key Allocator Policies

- **Placement policy:**
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - ***Interesting observation***: segregated free lists (next lecture) approximate a best fit placement policy without having to search entire free list

- **Splitting policy:**
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?

- **Coalescing policy:**
  - ***Immediate coalescing:*** coalesce each time **`free`** is called
  - ***Deferred coalescing:*** try to improve performance of **`free`** by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for **`malloc`**
    - Coalesce when the amount of external fragmentation reaches some threshold

Implicit Lists: Summary

- **Implementation: very simple**
- **Allocate cost:**
  - linear time worst case
- **Free cost:**
  - constant time worst case
  - even with coalescing
- **Memory usage:**
  - will depend on placement policy
  - First-fit, next-fit or best-fit

- **Not used in practice for `malloc/free` because of linear-time allocation**
  - used in many special purpose applications

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