Dynamic Memory Allocation: Basic Concepts
动态存储分配:基本概念

100076202： 计算机系统导论

任课教师:

计卫星　　宿红毅　　张艳

原作者:

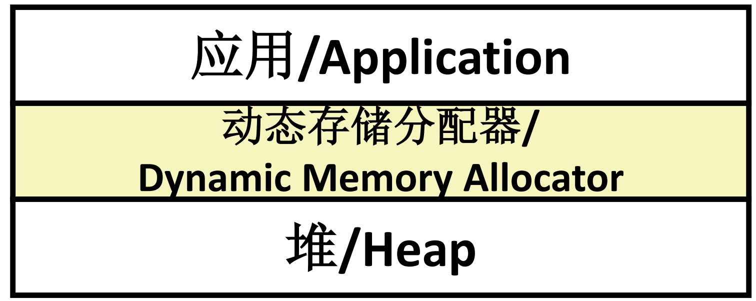
Randal E. Bryant and David R. O'Hallaron

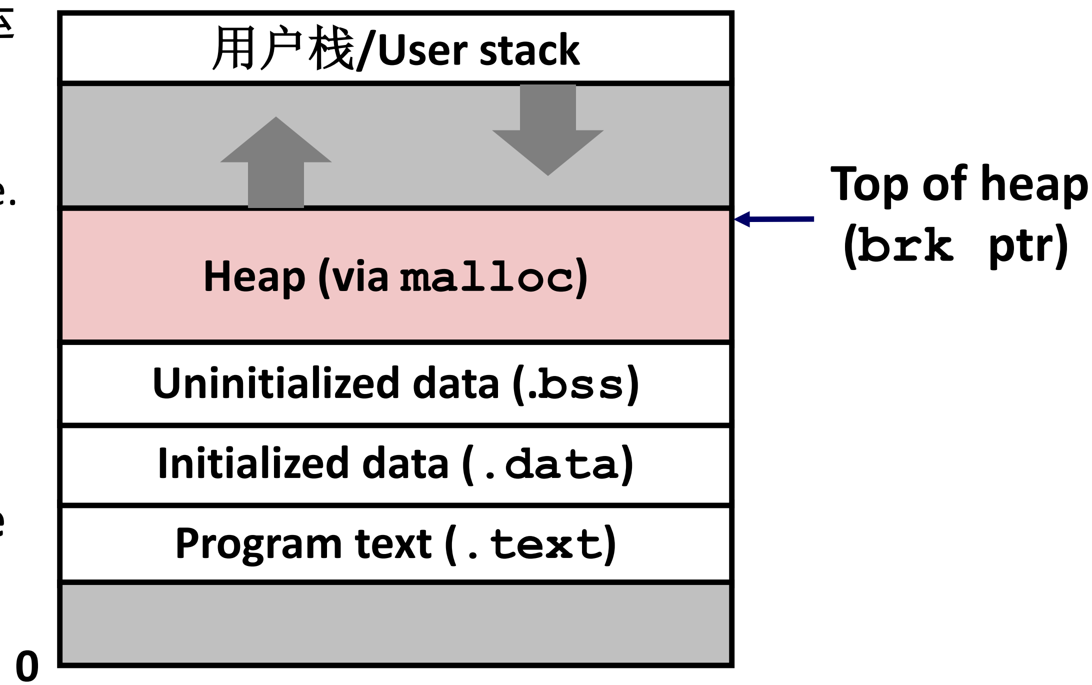
内容提纲/Today

- **基本概念/Basic concepts**
- **隐式空闲列表/Implicit free lists**

动态内存分配/Dynamic Memory Allocation

- 程序员使用动态内存分配器(malloc)在运行时申请内存/Programmers use ***dynamic memory allocators*** (such as `malloc`) to acquire VM at run time.
  - 对于那些数据结构大小在运行时才能知道的数据结构/For data structures whose size is only known at runtime.
- 动态内存分配器管理进程虚拟内存中一个称为堆的区域/Dynamic memory allocators manage an area of process virtual memory known as the ***heap***.





动态内存分配/Dynamic Memory Allocation

- 分配器将堆当做不同大小的块的集合进行管理，不是已分配就是空闲/**Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free***
- 分配器类型/**Types of allocators**
  - ***显式分配器/Explicit allocator***: 应用程序分配和释放空间/application allocates and frees space
    - 例如C中的malloc和free/E.g., `malloc` and `free` in C
  - ***隐式分配器/Implicit allocator:*** 应用只负责分配但是不释放/application allocates, but does not free space
    - 例如Java、ML和Lisp中的垃圾收集/E.g. garbage collection in Java, ML, and Lisp
- 今天主要讨论简单的显式内存分配/**Will discuss simple explicit memory allocation today**

malloc包/The `malloc` Package

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

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

- 成功/Successful:
  - 返回大小至少是size的内存块指针，x86上是按8字节对齐，x86-64是按16字节对齐/Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - 如果size为0，则返回NULL/If **`size == 0`**, returns NULL
- 不成功：返回NULL并设置errno/Unsuccessful: returns NULL (0) and sets **`errno`**

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

- 将p指向的内存块返回给可用内存池/Returns the block pointed at by **`p`** to pool of available memory
- **`p`**必须是之前调用`malloc`或者`realloc`获得的/**`p`** must come from a previous call to **`malloc`** or **`realloc`**

其他函数/Other functions

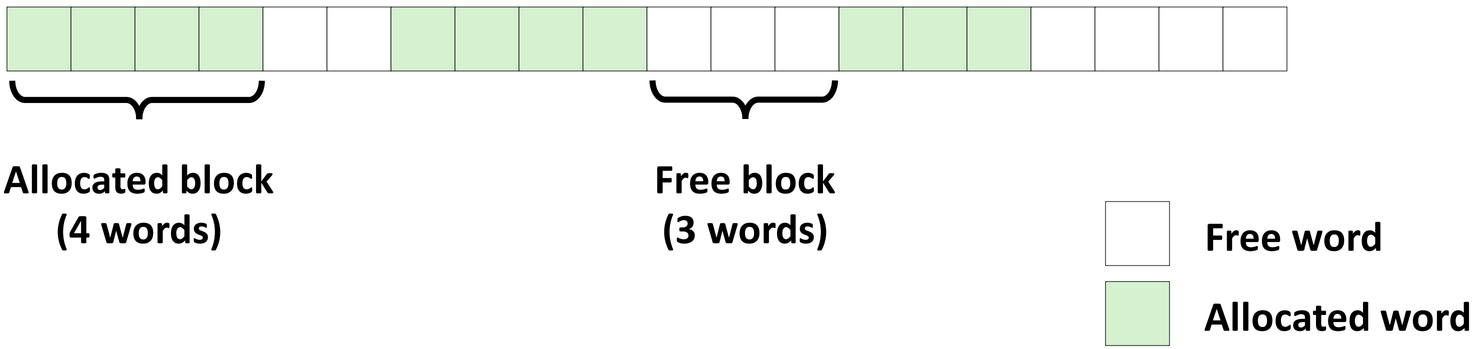
- **`calloc`**: malloc的另一个版本，会将分配的内存块初始化为0/Version of **`malloc`** that initializes allocated block to zero.
- **`realloc`**: 改变之前分配的块的大小/Changes the size of a previously allocated block.
- **`sbrk`**: 分配器内部用来增加或者减小堆的大小/Used internally by allocators to grow or shrink the heap

malloc示例/malloc Example

```
#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);
}
```

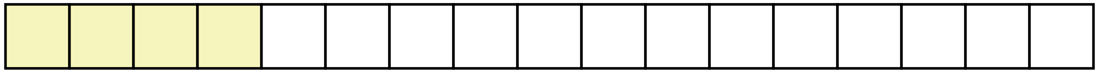
本节课假设/Assumptions Made in This Lecture

- 内存是按照字对齐的/Memory is word addressed.
- 字是整数倍大小的/Words are int-sized.

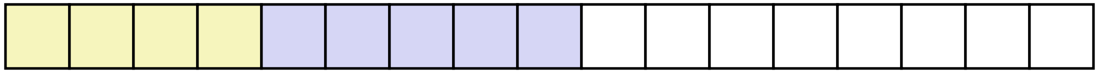


分配示例/Allocation Example

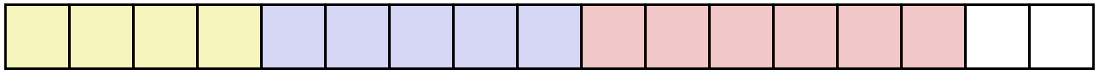
```
p1 = malloc(4)
```



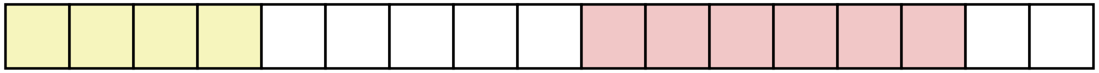
```
p2 = malloc(5)
```



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



```
free(p2)
```



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

限制/Constraints

- 应用/Applications
  - 可以发送任意malloc和free序列/Can issue arbitrary sequence of **`malloc`** and **`free`** requests
  - **`free`**请求必须针对一个`malloc`返回的块/**`free`** request must be to a **`malloc`**'d block

- 分配器/Allocators
  - 无法控制分配的块的数量和大小.Can't control number or size of allocated blocks
  - 必须及时响应malloc请求/Must respond immediately to **`malloc`** requests
    - 例如，不能对请求排序和缓冲/*i.e.*, can't reorder or buffer requests
  - 必须从空闲空间分配内存块/Must allocate blocks from free memory
    - 例如，分配的块必须在空闲内存中/*i.e.*, can only place allocated blocks in free memory
  - 必须按照需求实现块对齐/Must align blocks so they satisfy all alignment requirements
    - Linux中x86是8字节对齐，x86-64是16字节对齐/8-byte (x86) or 16-byte (x86-64) alignment on Linux boxes
  - 只能操作和修改空闲内存/Can manipulate and modify only free memory
  - 一旦分配后不能移动内存块/Can't move the allocated blocks once they are **`malloc`**'d
    - 例如，压缩是不允许的/*i.e.*, compaction is not allowed

性能目标：吞吐率/Performance Goal: Throughput

- **对于给定的malloc和free序列/Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

- **目标：最大化吞吐率和内存利用率/Goals: maximize throughput and peak memory utilization**
  - 这些目标通常是互相冲突的/These goals are often conflicting

- **吞吐率/Throughput:**
  - 单位时间内完成的请求数量/Number of completed requests per unit time
  - 例如：/Example:
    - 10秒内完成5000次malloc和5000次free/5,000 **`malloc`** calls and 5,000 **`free`** calls in 10 seconds
    - 吞吐率就是1000操作/秒/Throughput is 1,000 operations/second

性能目标：内存利用率最大/Performance Goal: Peak Memory Utilization

- 对于给定的malloc和free某个请求序列/Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, ..., R_k, ... , R_{n-1}$

- ***Def:*** *总有效载荷/Aggregate payload P_k*
  - `malloc(p)` 返回一个载荷为p字节的块/`malloc(p)` results in a block with a ***payload*** of `p` bytes
  - 请求R_k完成后，总有效载荷P_k是目前已分配的载荷的总大小/After request R_k has completed, the ***aggregate payload*** P_k is the sum of currently allocated payloads

- ***Def:*** 当前堆大小H_k/***Current heap size*** H_k
  - 假设H_k单调不递减/Assume H_k is monotonically nondecreasing
    - 例如，只有分配器调用sbrk时增加/i.e., heap only grows when allocator uses `sbrk`

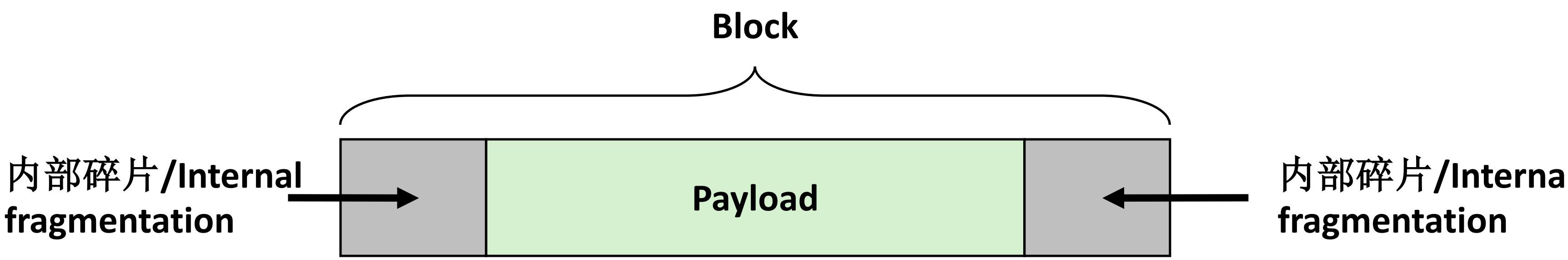
- ***Def: k+1次请求之后内存利用率峰值/Peak memory utilization after k+1 requests***
  - $U_k = (max_{i<=k} P_i) \ / \ H_k$

内存碎片/Fragmentation

- 由内存碎片导致的内存低利用率/**Poor memory utilization caused by *fragmentation***
  - 内部碎片/***internal*** fragmentation
  - 外部碎片/***external*** fragmentation

内部碎片/Internal Fragmentation

- **对于给定的块，如果载荷小于块大小就会导致内部碎片/For a given block, *internal fragmentation* occurs if payload is smaller than block size**



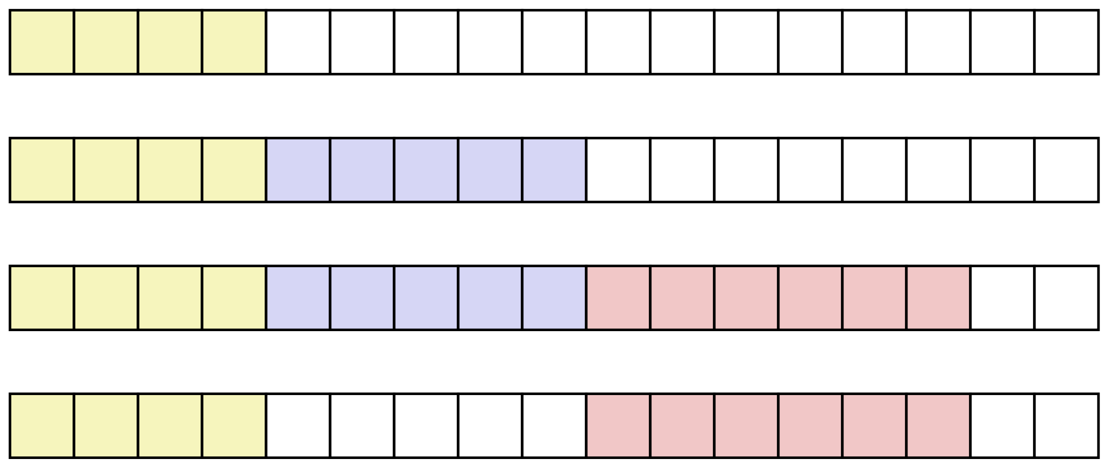
- **原因/Caused by**
  - 维护堆数据结构开销/Overhead of maintaining heap data structures
  - 为了对齐填充的部分/Padding for alignment purposes
  - 显式策略导致/Explicit policy decisions
    (例如：为了一个小的请求返回一个大的块/e.g., to return a big block to satisfy a small request)

- **只是与之前的请求的模式相关/Depends only on the pattern of *previous* requests**
  - 因此易于度量/Thus, easy to measure

外部碎片/External Fragmentation

- **当堆内存有足够的载荷，但是没有个单一的空闲块足够大/Occurs when there is enough aggregate heap memory, but no single free block is large enough**

```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(6)
```



Oops! (what would happen now?)

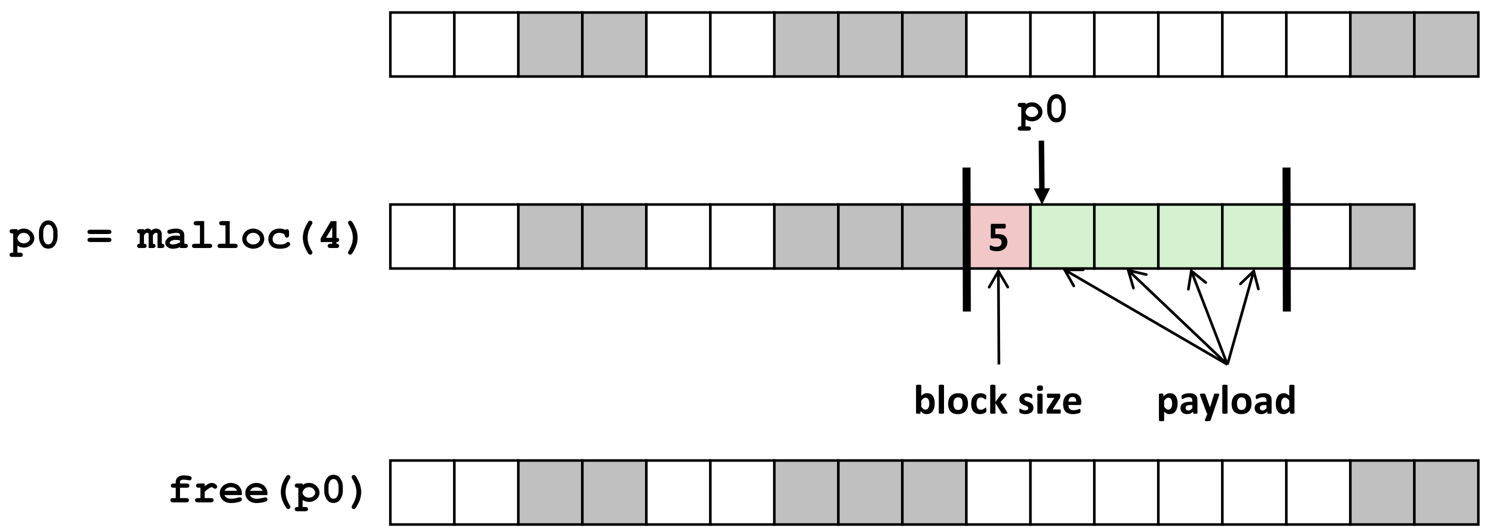
- **取决于后续的请求模式/Depends on the pattern of future requests**
  - 因此难以度量/Thus, difficult to measure

实现问题/Implementation Issues

- 给定一个指针，我们怎么知道要释放多大的空间/How do we know how much memory to free given just a pointer?
- 我们怎么跟踪空闲块/How do we keep track of the free blocks?
- 当分配的结构大小小于选择的空闲块时怎么办？/What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- 当有多个块可用时我们应该怎么选？/How do we pick a block to use for allocation -- many might fit?
- 如何再次插入空闲块？/How do we reinsert freed block?

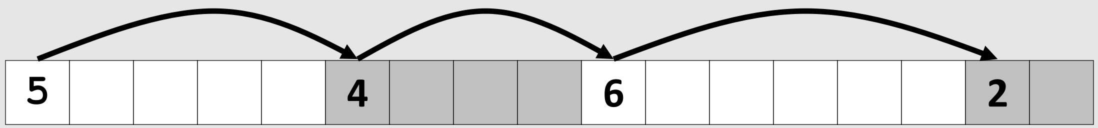
获取释放大小/Knowing How Much to Free

- **标准方法/Standard method**
  - 在块之前的字中保存/Keep the length of a block in the word preceding the block.
    - 这个字称为头域或者头/This word is often called the ***header field*** or ***header***
  - 每个分配的块需要一个额外的字/Requires an extra word for every allocated block

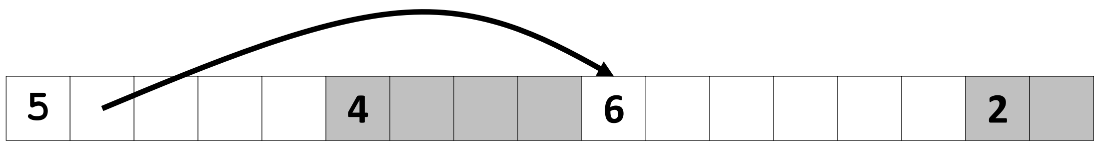


跟踪空闲块/Keeping Track of Free Blocks

- 方法1：隐式链表-使用长度链接所有块/Method 1: *Implicit list* using length—links all blocks



- 方法2：空间块之间使用指针的显式链表/Method 2: *Explicit list* among the free blocks using pointers



- 方法3：分离的空闲列表/Method 3: *Segregated free list*
  - 不同大小块使用不同的空闲列表/Different free lists for different size classes
- 方法4：根据大小对块排序/Method 4: *Blocks sorted by size*
  - 可以使用一个平衡树（红黑树）Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

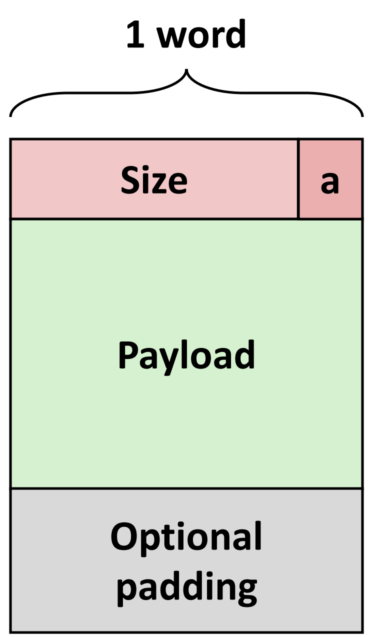
内容提纲/Today

- 基本概念/Basic concepts
- **隐式空闲列表/Implicit free lists**

方法1：隐式链表/Method 1: Implicit List

- 对每个块我们需要大小和分配的状态/**For each block we need both size and allocation status**
  - 可以放在两个字中:浪费/Could store this information in two words: wasteful!
- 标准技巧/**Standard trick**
  - 如果块是对齐的，则地址低位部分总是0/If blocks are aligned, some low-order address bits are always 0
  - 与其存储0，还不如将其作为分配/空闲的标志位/Instead of storing an always-0 bit, use it as a allocated/free flag
  - 读大小时需要将这些位屏蔽掉/When reading size word, must mask out this bit

分配和空闲块格式/ Format of allocated and free blocks

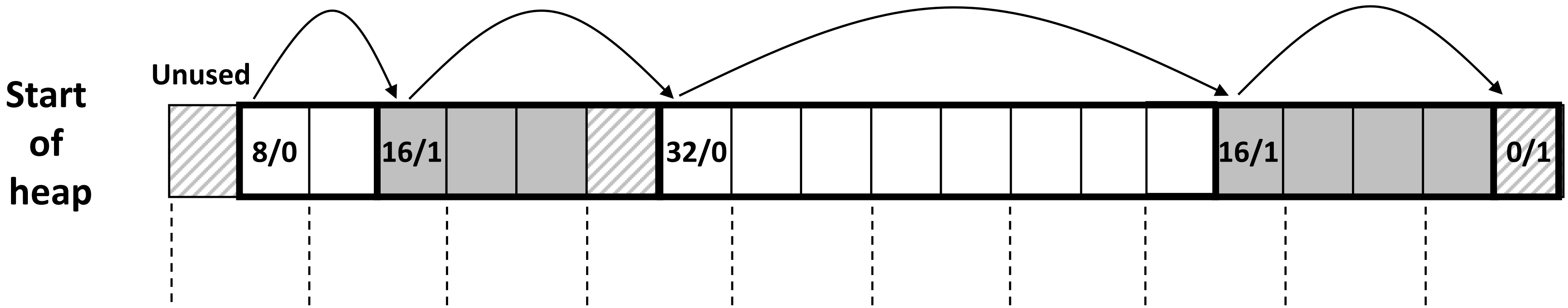


a = 1: Allocated block/分配的块
a = 0: Free block/空闲块

Size: block size/块大小

Payload: application data/载荷：应用数据 (allocated blocks only)

隐式空闲链表的详细例子/Detailed Implicit Free List Example



双字对齐 /Double-word aligned

分配的块:阴影/Allocated blocks: shaded
空闲块：无阴影/Free blocks: unshaded
头部：使用字节大小/分配位进行标记/
Headers: labeled with size in bytes/allocated bit

隐式链表：查找空闲块/Implicit List: Finding a Free Block

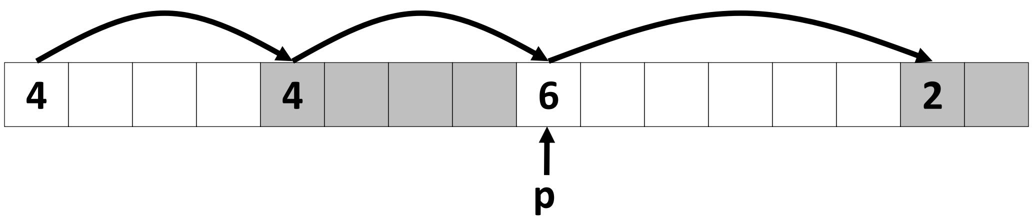
- ***First fit:***
  - 从链表开始搜索，选择第一个满足条件的空闲块/Search list from beginning, choose ***first*** free block

```
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)
```

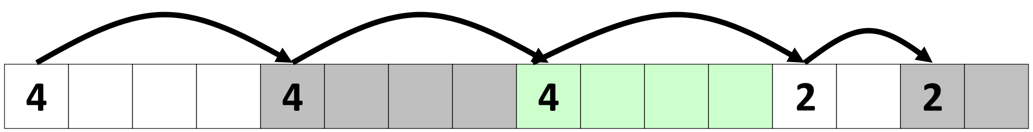
  - 与总块数成线性关系/Can take linear time in total number of blocks (allocated and free)
  - 实际上会在链表开始时造成碎片/In practice it can cause “splinters” at beginning of list
- ***Next fit:***
  - 与first fit类似，但是从上一次搜索结束的位置开始查找/Like first fit, but search list starting where previous search finished
  - 一般会比first fit块：避免了扫描无用的块/Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - 部分研究表明更容易造成内存碎片/Some research suggests that fragmentation is worse
- ***Best fit:***
  - 从链表中选择最佳的块：最小满足需求的块/Search the list, choose the ***best*** free block: fits, with fewest bytes left over
  - 保持内存碎片最小化-通常能改进内存利用率/Keeps fragments small—usually improves memory utilization
  - 一般会比first fit慢/Will typically run slower than first fit

隐式链表：从空闲块中分配/Implicit List: Allocating in Free Block

- 从一个空闲块分配：拆分/Allocating in a free block: *splitting*
  - 由于分配的空间可能会比空闲空间小，因此可能会拆分空闲块/Since allocated space might be smaller than free space, we might want to split the block



`addblock(p, 4)`



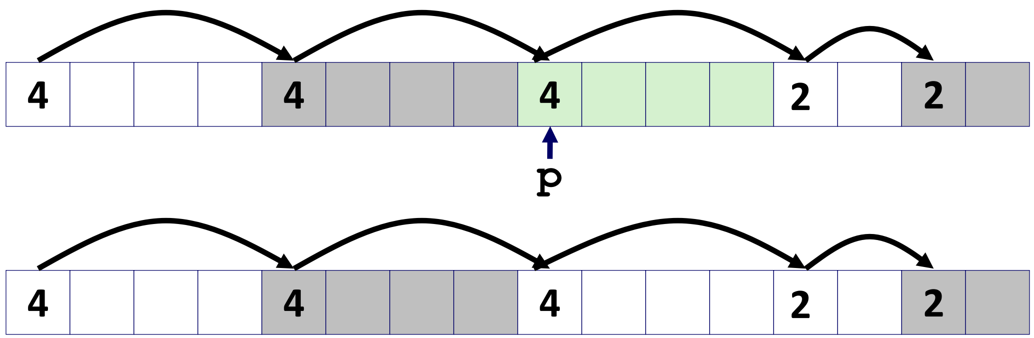

```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining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

隐式链表：释放一个块/Implicit List: Freeing a Block

- 最简单的实现/Simplest implementation:
  - 只需要清除“已分配”标记位/Need only clear the “allocated” flag

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

  - 但是可能会导致“伪碎片”/But can lead to “false fragmentation”



`free(p)`

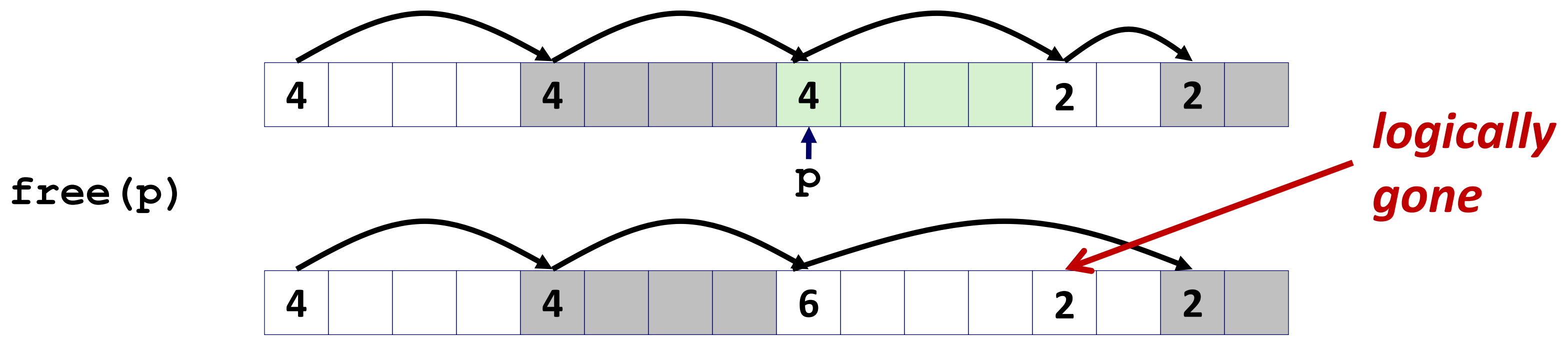
`malloc(5)` ***Oops!***

有足够的空闲空间，但是分配器找不到!

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

隐式链表：合并/Implicit List: Coalescing

- **与下一个/前一个空闲块合并/Join *(coalesce)* with next/previous blocks, if they are free**
  - 与下一个块合并/Coalescing with next block

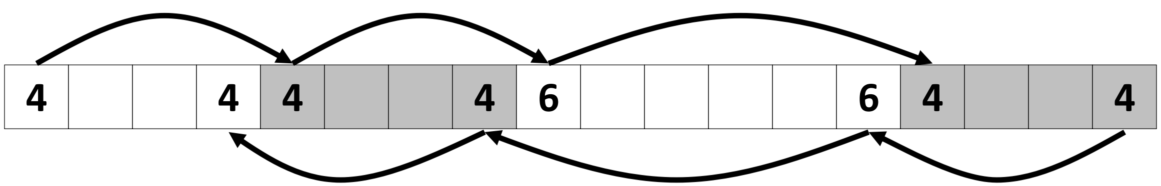


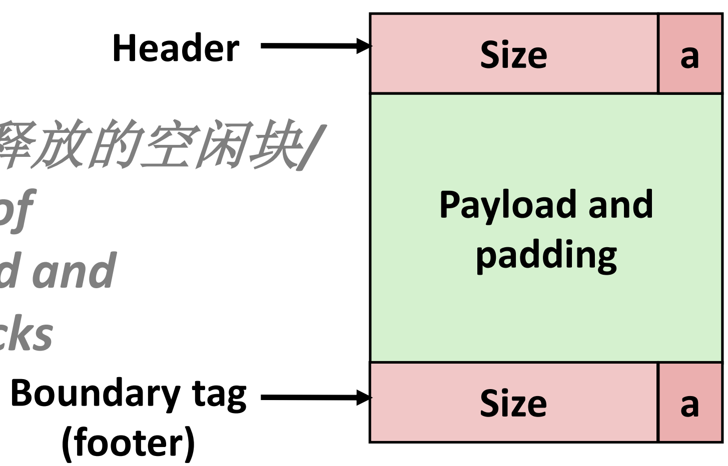

```
void free_block(ptr p) {
    *p = *p & -2;           // clear allocated flag
    next = p + *p;          // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;      // add to this block if
}                           //    not allocated
```

  - 但是怎么和前一个块合并？/But how do we coalesce with *previous* block?

隐式链表：双向合并/Implicit List: Bidirectional Coalescing

- ***边界标记/Boundary tags*** [Knuth73]
  - 在空闲块结束的位置重复标记大小/已分配字/Replicate size/allocated word at “bottom” (end) of free blocks
  - 以额外的空间换取反向遍历列表功能/Allows us to traverse the “list” backwards, but requires extra space
  - 重要和常用的技术/Important and general technique!





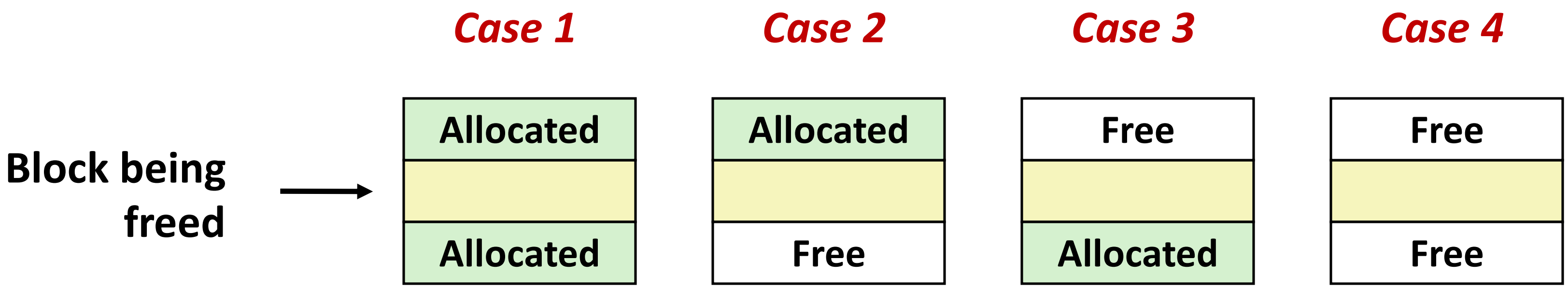
分配和释放的空闲块/ Format of allocated and free blocks

a = 1: Allocated block /分配的块
a = 0: Free block/空闲块

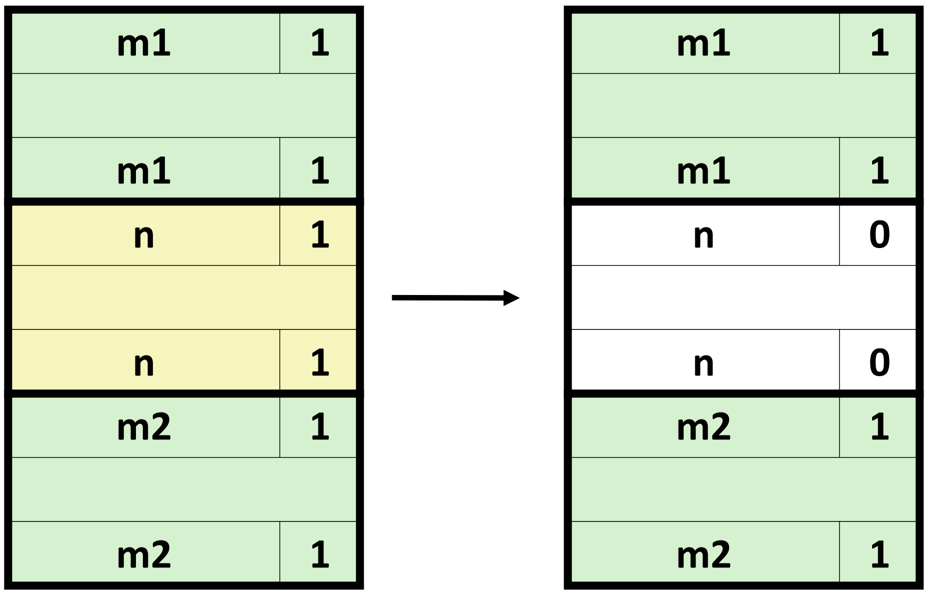
Size: Total block size/总体块大小

Payload: Application data (allocated blocks only)/应用程序数据

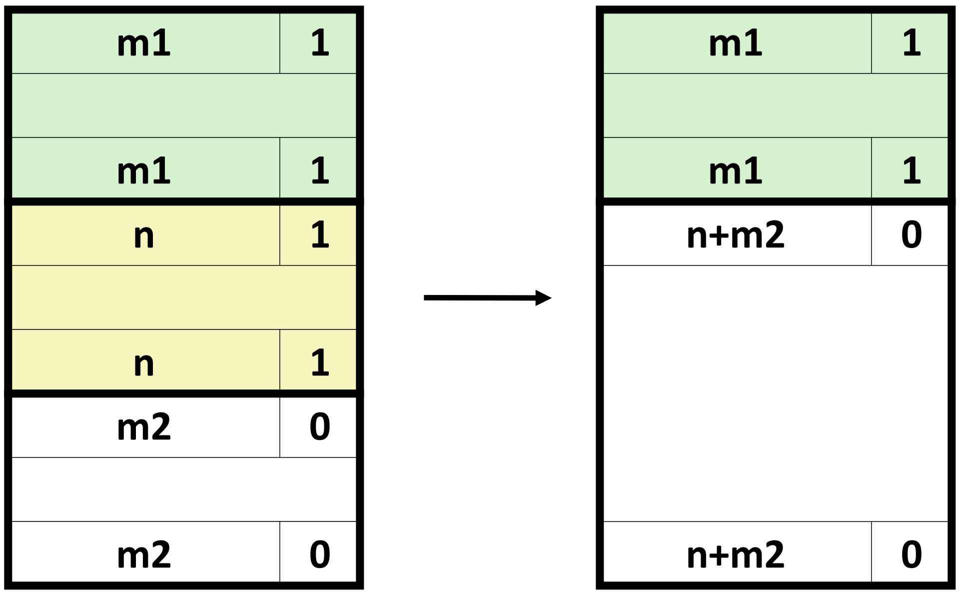
常量时间合并/ Constant Time Coalescing



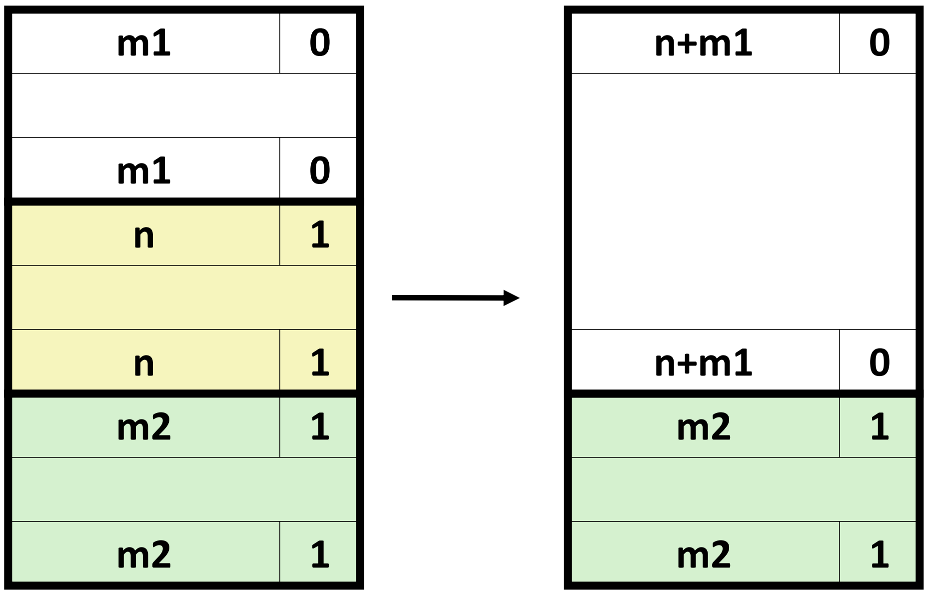
常量时间合并/ Constant Time Coalescing (Case 1)



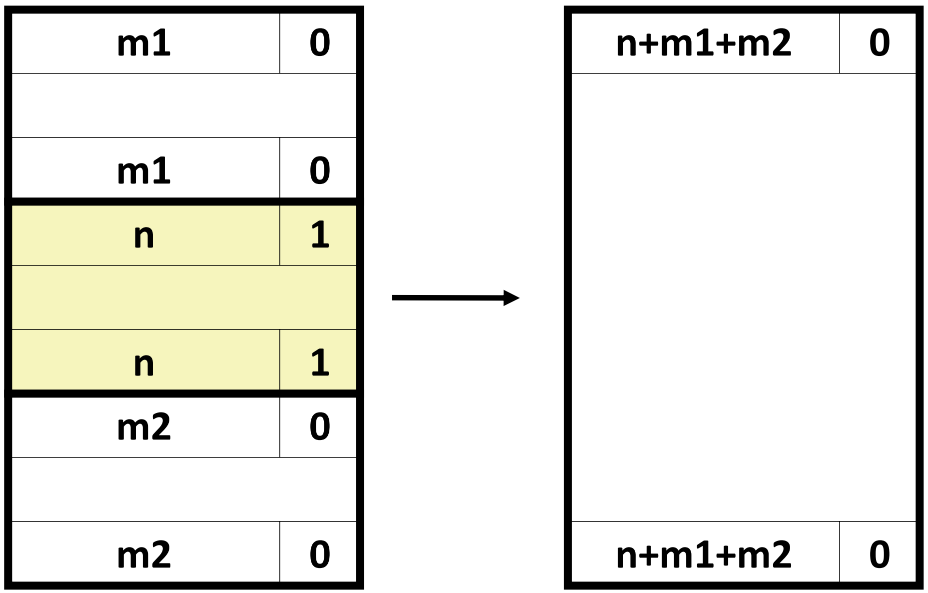
常量时间合并/ Constant Time Coalescing (Case 2)



常量时间合并/Constant Time Coalescing (Case 3)



常量时间合并Constant Time Coalescing (Case 4)



边界标记的缺点/Disadvantages of Boundary Tags

- **内部碎片/Internal fragmentation**

- **可以进一步优化吗/Can it be optimized?**
  - 哪些块需要结束标记？/Which blocks need the footer tag?
  - 这意味着什么？/What does that mean?

主要分配策略总结/Summary of Key Allocator Policies

- 选择策略/**Placement policy:**
  - First-fit, next-fit, best-fit, etc.
  - 在吞吐率和更少的碎片之间平衡/Trades off lower throughput for less fragmentation
  - ***有趣的观察***：多个空闲列表与最优选择策略接近，且不用搜索整个链表/***Interesting observation***: segregated free lists (next lecture) approximate a best fit placement policy without having to search entire free list

- 拆分策略：/**Splitting policy:**
  - 什么时候需要拆分空闲块？/When do we go ahead and split free blocks?
  - 我们可能容忍多少内部碎片？/How much internal fragmentation are we willing to tolerate?

- 合并策略：/**Coalescing policy:**
  - ***立即合并：***每次free时合并/***Immediate coalescing:*** coalesce each time **`free`** is called
  - ***延迟合并：***为了提升free的性能，当需要时再合并/***Deferred coalescing:*** try to improve performance of **`free`** by deferring coalescing until needed. Examples:
    - 由于malloc扫描空闲列表时进行合并/Coalesce as you scan the free list for **`malloc`**
    - 当外部碎片超过某个阈值时进行合并/Coalesce when the amount of external fragmentation reaches some threshold

隐式列表：总结/Implicit Lists: Summary

- **实现：非常简单/Implementation: very simple**
- **分配开销：/Allocate cost:**
  - 最差是线性时间/linear time worst case
- **释放开销：/Free cost:**
  - 最差常量时间/constant time worst case
  - 甚至包括合并/even with coalescing
- **内存使用/Memory usage:**
  - 依赖于选择策略/will depend on placement policy
  - First-fit, next-fit or best-fit

- **由于线性时间的分配开销，实际malloc和free并没有使用/Not used in practice for `malloc/free` because of linear-time allocation**
  - 在很多特殊目的的应用中使用/used in many special purpose applications

- **然而拆分和基于边界标记的合并对所有的分配器都是适用的/However, the concepts of splitting and boundary tag coalescing are general to *all* allocators**