第9章 虚拟内存

Dynamic Memory Allocation: Basic Concepts
动态内存分配:基本概念

100076202： 计算机系统导论

任课教师:

宿红毅　张艳　黎有琦　李秀星

原作者:

Randal E. Bryant and David R. O’Hallaron

议题 Today

- **基本概念 Basic concepts**
- 隐式空闲链表 Implicit free lists

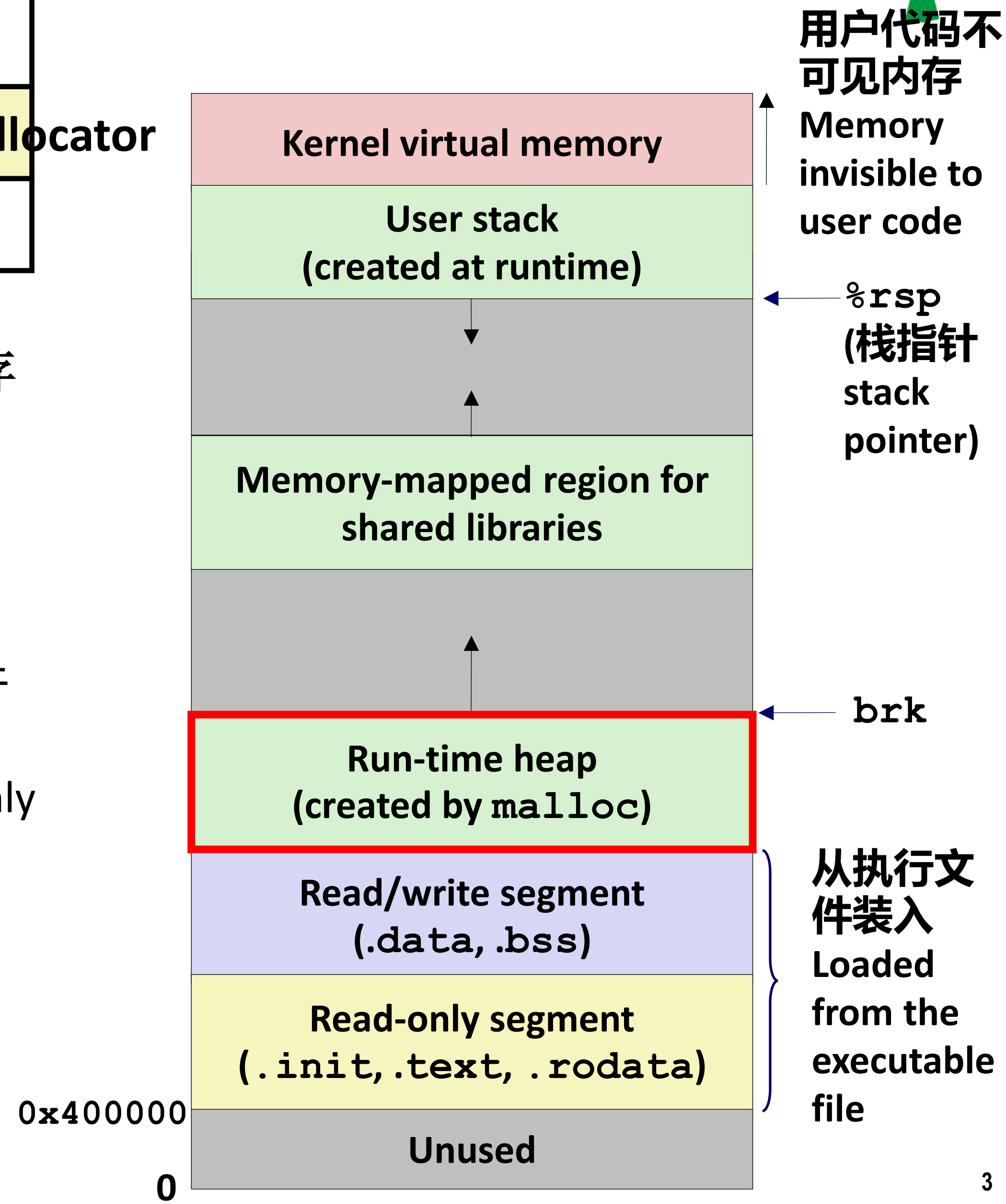
动态内存分配 Dynamic Memory Allocation

应用 Application

动态内存分配器 Dynamic Memory Allocator

堆 Heap

- 程序员使用**动态内存分配器**(malloc)在运行时申请虚拟内存 Programmers use ***dynamic memory allocators*** (such as `malloc`) to acquire virtual memory (VM) at runtime
  - 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 VM 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:*** 应用程序分配和释放空间 ***Explicit allocator*:** 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:*** 应用只负责分配但是不释放空间 ***Implicit allocator:*** 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**

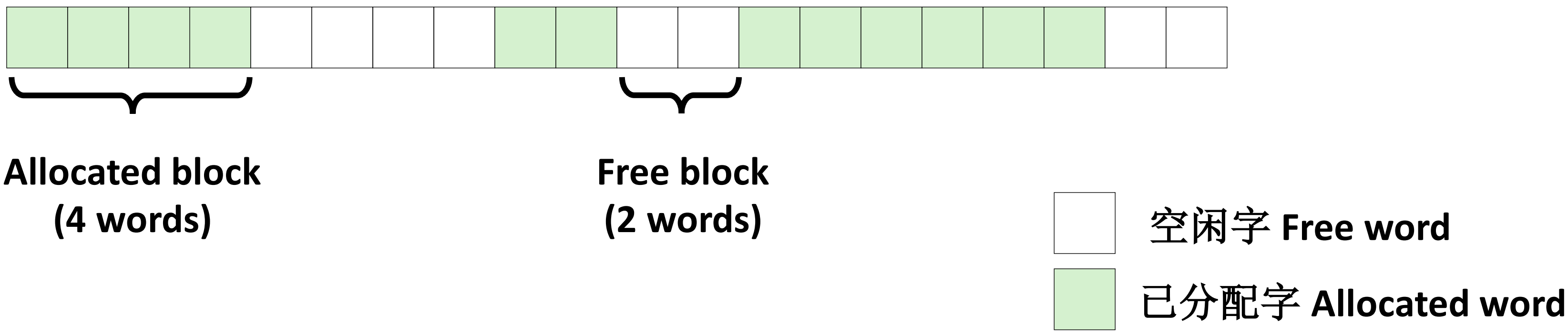
- **`calloc`:** malloc的另一个版本，会将分配的内存块初始化为0 Version of **`malloc`** that initializes allocated block to zero.
- **`realloc`:** 改变之前分配的块的大小 Changes the size of a previously allocated block.
- **`sbrk`:** 分配器内部用来增加或者减小堆的大小 Used internally by allocators to grow or shrink the heap

malloc示例 malloc Example

```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);
}
```

可视化展示规则 Visualization Conventions

- 显式8字节字为一个方块 Show 8-byte words as squares
- 分配采用双字对齐 Allocations are double-word aligned



分配示例 Allocation Example (概念上 Conceptual)

```
#define SIZ sizeof(size_t)
```

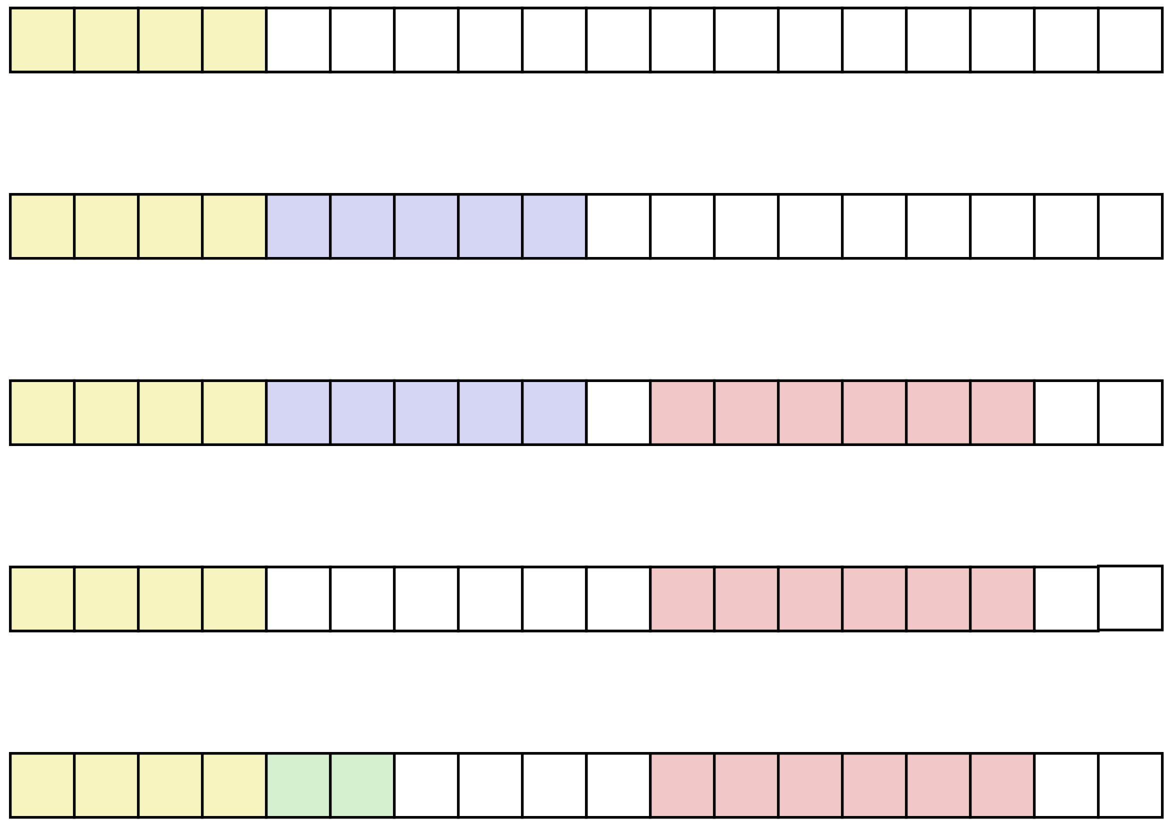
```
p1 = malloc(4*SIZ)
```

```
p2 = malloc(5*SIZ)
```

```
p3 = malloc(6*SIZ)
```

```
free(p2)
```

```
p4 = malloc(2*SIZ)
```



限制 Constraints

- 应用 **Applications**
  - 可以发出任意malloc和free请求序列 Can issue arbitrary sequence of **`malloc`** and **`free`** requests
  - **`free`**请求必须针对一个`malloc`请求的块 **`free`** request must be to a **`malloc`**’d block

- 显式分配器 **Explicit 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：最大化吞吐率和峰值内存利用率 **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次操作/秒 Throughput is 1,000 operations/second

性能目标：最小化开销
Performance Goal: Minimize Overhead

- 对于给定的malloc和free某个请求序列 **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- ***K次请求之后，我们得到： After k requests we have:***
- *定义：总有效载荷* ***Def: 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
- *定义：*当前堆大小H_k ***Def: Current heap size*** H_k
  - 假设H_k单调不递减 Assume H_k is monotonically nondecreasing
    - 即当分配器使用sbrk时堆增加 i.e., heap only grows when allocator uses `sbrk`
- *定义：* ***k+1次请求之后峰值内存利用率 Def: Peak memory utilization after k+1 requests***
  - $U_k = (\max_{i<=k} P_i) / H_k$

性能目标：最小化开销
Performance Goal: Minimize Overhead

- 对于给定的malloc和free一些请求序列 **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- ***K次请求之后，我们得到：After k requests we have:***
- ***定义：总有效载荷 Def: 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
  - 总有效载荷P_k是目前已分配的载荷的总和 The ***aggregate payload*** P_k is the sum of currently allocated payloads
  - 峰值总有效载荷是请求序列中任何点处最大总有效载荷 The ***peak aggregate payload*** $\max_{i \le k} P_i$ is the maximum aggregate payload at any point in the sequence up to request

性能目标：最小化开销
Performance Goal: Minimize Overhead

- 对于给定的malloc和free一些请求序列 **Given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

- ***K次请求之后，我们得到： After k requests we have:***

- *定义：*当前堆大小$\mathbf{H_k}$ ***Def:* Current heap size H_k**
  - 假设当分配器使用sbrk时堆仅增加，从不收缩 Assume heap only *grows* when allocator uses **`sbrk`**, never shrinks

- *定义：开销，O_k* ***Def:* Overhead, O_k**
  - 堆空间没有为程序数据使用的比例 Fraction of heap space *NOT* used for program data
  - $O_k = (H_k / \max_{i \le k} P_i) - 1.0$

基准测试示例 Benchmark Example

- 基准测试 Benchmark

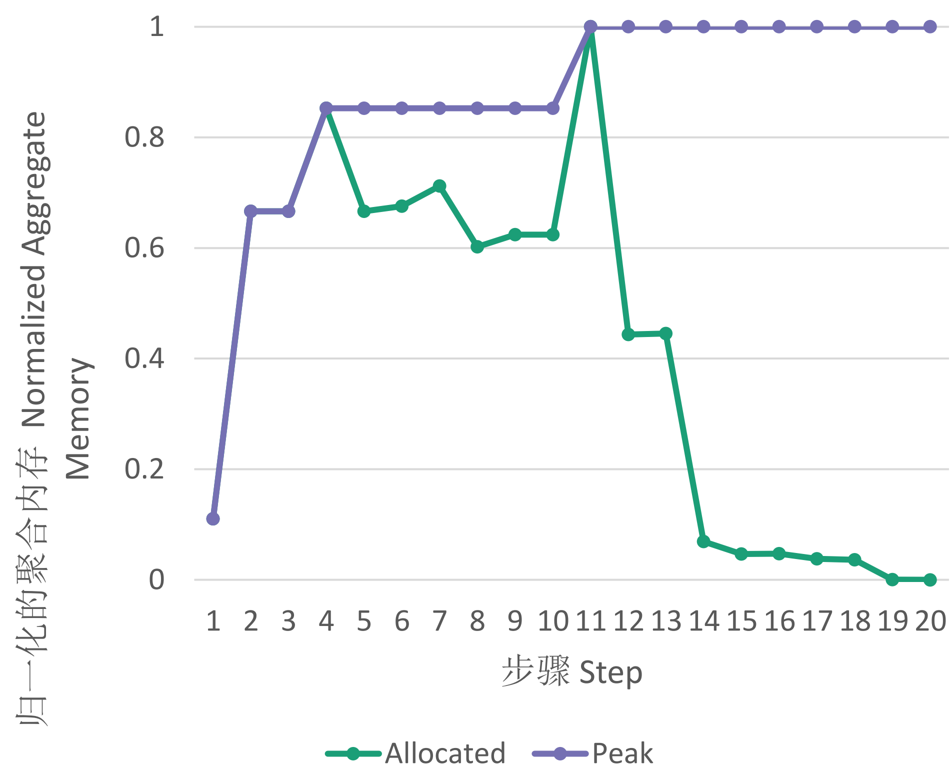
  syn-array-short

  - malloc实验提供的跟踪 Trace provided with malloc lab
  - 分配和释放各10个块 Allocate & free 10 blocks
  - a代表分配 a = allocate
  - f代表释放 f = free
  - 偏置在开始时分配，在结束时释放 Bias toward allocate at beginning & free at end
  - 块号1-10 Blocks number 1–10
  - 已分配：所有分配量的和 Allocated: Sum of all allocated amounts
  - 峰值：曾经分配的最大值 Peak: Max so far of Allocated

步骤 Step	命令 Command	偏置 Delta	已分配 Allocated	峰值 Peak
1	a 0 9904	9904	9904	9904
2	a 1 50084	50084	59988	59988
3	a 2 20	20	60008	60008
4	a 3 16784	16784	76792	76792
5	f 3	-16784	60008	76792
6	a 4 840	840	60848	76792
7	a 5 3244	3244	64092	76792
8	f 0	-9904	54188	76792
9	a 6 2012	2012	56200	76792
10	f 2	-20	56180	76792
11	a 7 33856	33856	90036	90036
12	f 1	-50084	39952	90036
13	a 8 136	136	40088	90036
14	f 7	-33856	6232	90036
15	f 6	-2012	4220	90036
16	a 9 20	20	4240	90036
17	f 4	-840	3400	90036
18	f 8	-136	3264	90036
19	f 5	-3244	20	90036
20	f 9	-20	0	90036

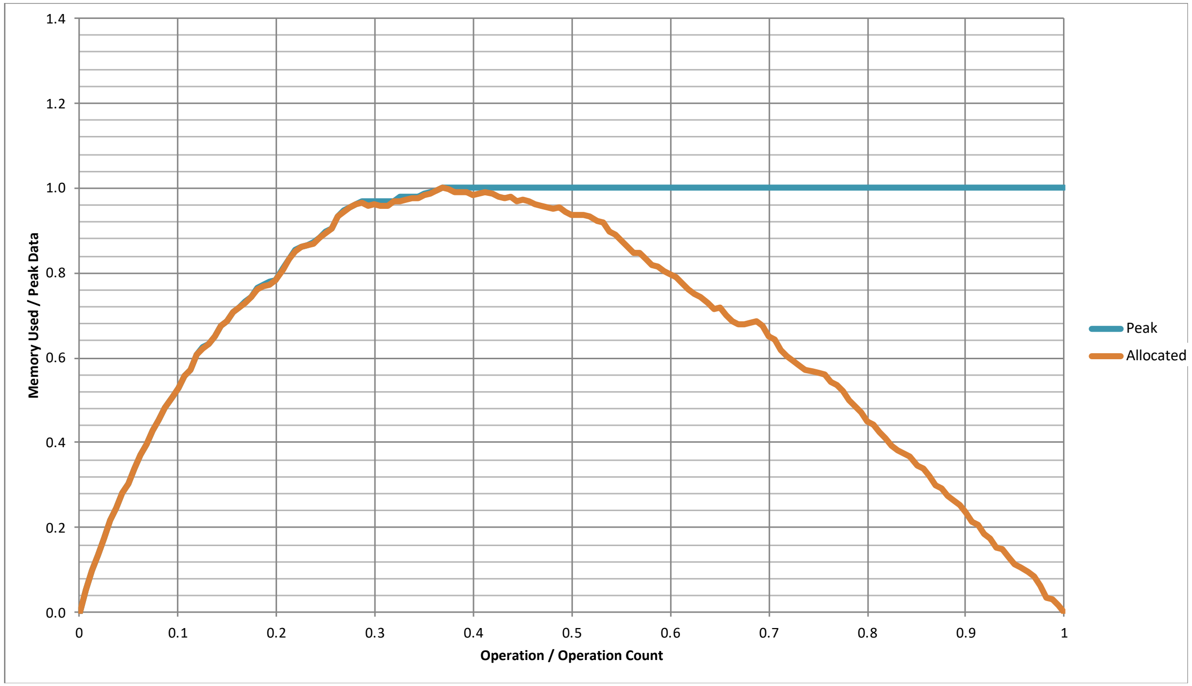
基准测试可视化 Benchmark Visualization

步骤 Step	命令 Command	偏置 Delta	已分配 Allocated	峰值 Peak
1	`a 0 9904`	9904	9904	9904
2	`a 1 50084`	50084	59988	59988
3	`a 2 20`	20	60008	60008
4	`a 3 16784`	16784	76792	76792
5	`f 3`	-16784	60008	76792
6	`a 4 840`	840	60848	76792
7	`a 5 3244`	3244	64092	76792
8	`f 0`	-9904	54188	76792
9	`a 6 2012`	2012	56200	76792
10	`f 2`	-20	56180	76792
11	`a 7 33856`	33856	90036	90036
12	`f 1`	-50084	39952	90036
13	`a 8 136`	136	40088	90036
14	`f 7`	-33856	6232	90036
15	`f 6`	-2012	4220	90036
16	`a 9 20`	20	4240	90036
17	`f 4`	-840	3400	90036
18	`f 8`	-136	3264	90036
19	`f 5`	-3244	20	90036
20	`f 9`	-20	0	90036



- 已分配内存和峰值内存是步骤k的函数绘图 Plot P_k (allocated) and $\max_{i \le k} P_k$ (peak) as a function of k (step)
- Y轴归一化处理—占最大值的比例 Y-axis normalized — fraction of maximum

典型的基准测试行为
Typical Benchmark Behavior



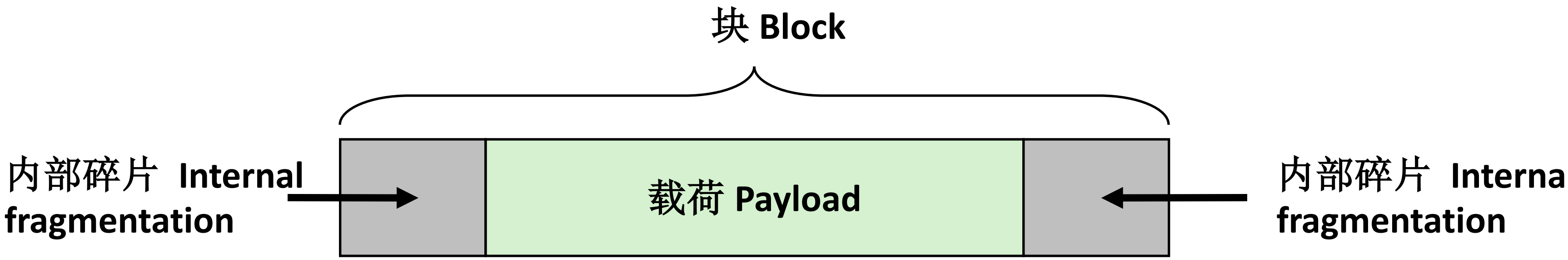
- 分配和释放内存的长序列（**40000**块） **Longer sequence of mallocs & frees (40,000 blocks)**
  - 开始都是分配内存，然后转向释放内存 Starts with all mallocs, and shifts toward all frees
- 分配器必须整个时间段内有效管理空间 **Allocator must manage space efficiently the whole time**
- 生产分配器可以收缩堆 **Production allocators can shrink the heap**

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

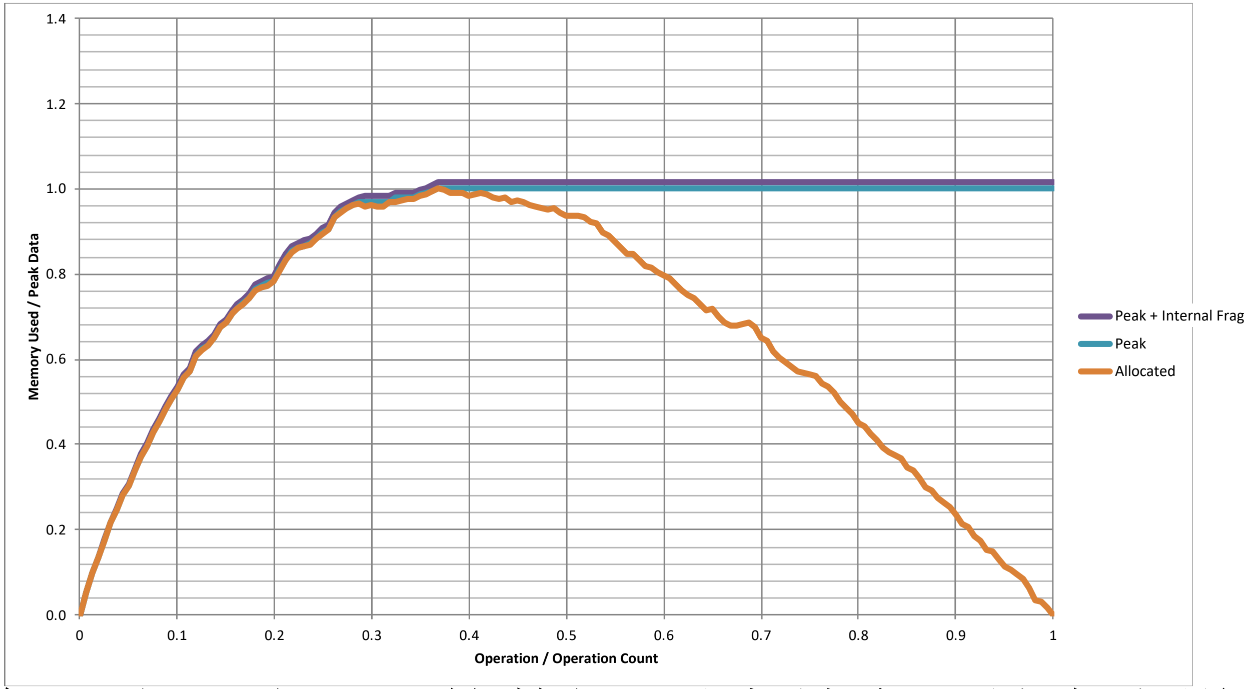
- 对于给定的块，如果载荷小于块大小就会导致内部碎片 **For a given block, *internal fragmentation* occurs if payload is smaller than block size**



- 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：为了满足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

内部碎片效应
Internal Fragmentation Effect



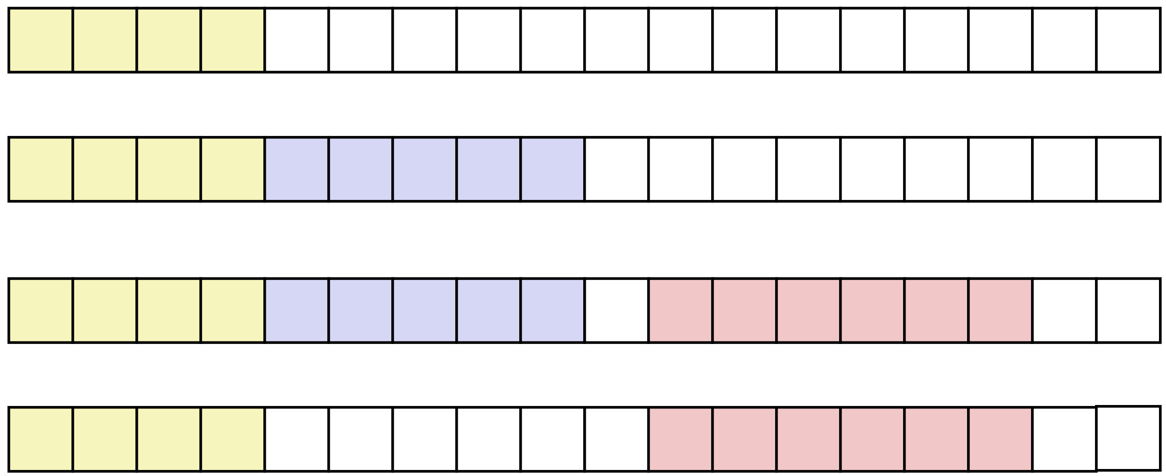
- 紫色线条：由于分配器的数据+对齐填充，堆大小增加 **Purple line: additional heap size due to allocator’s data + padding for alignment**
  - 对于该基准，1.5%的开销 For this benchmark, 1.5% overhead
  - 无法在实践中实现 Cannot achieve in practice
  - 特别是因为无法移动已分配的块 Especially since cannot move allocated blocks

外部碎片 External Fragmentation

```
#define SIZ sizeof(size_t)
```

- 当有足够的聚合堆内存，但是没有单一的空闲块足够大时产生外部碎片 **Occurs when there is enough aggregate heap memory, but no single free block is large enough**

```
p1 = malloc(4*SIZ)

p2 = malloc(5*SIZ)

p3 = malloc(6*SIZ)

free(p2)

p4 = malloc(7*SIZ)
```

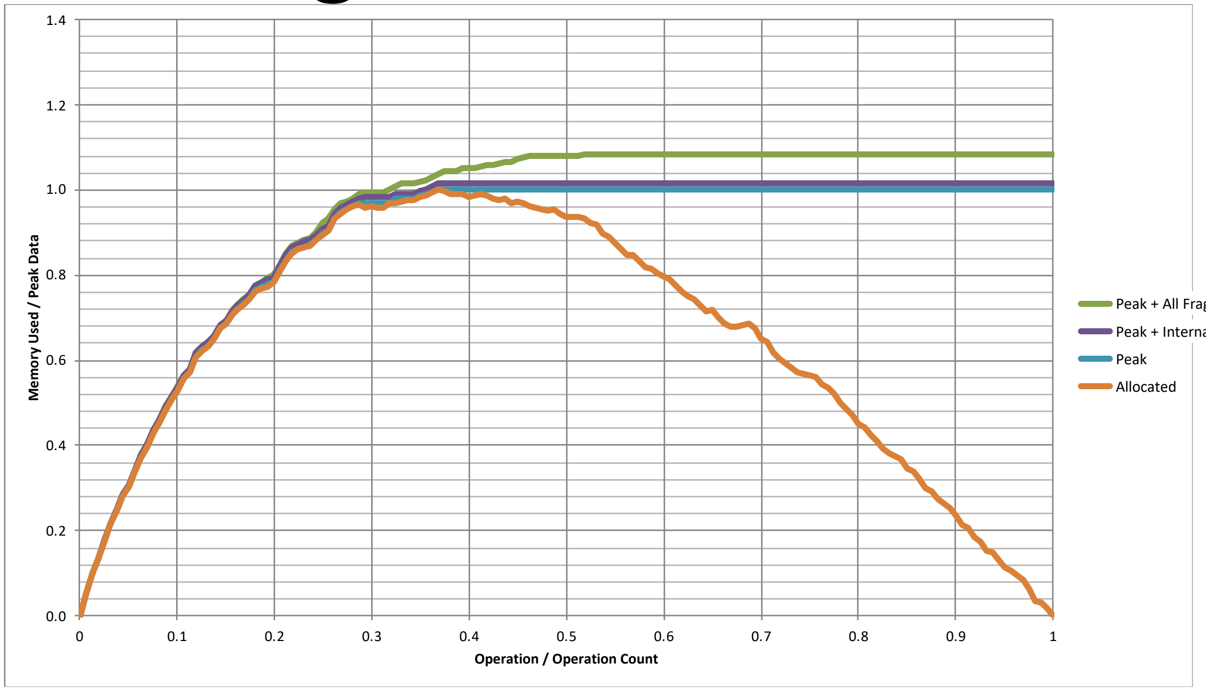


诶呀（现在会发生什么？）

Yikes! (what would happen now?)

- 取决于未来请求的模式 **Depends on the pattern of future requests**
  - 因此，难以测量 Thus, difficult to measure

外部碎片的效应
External Fragmentation Effect



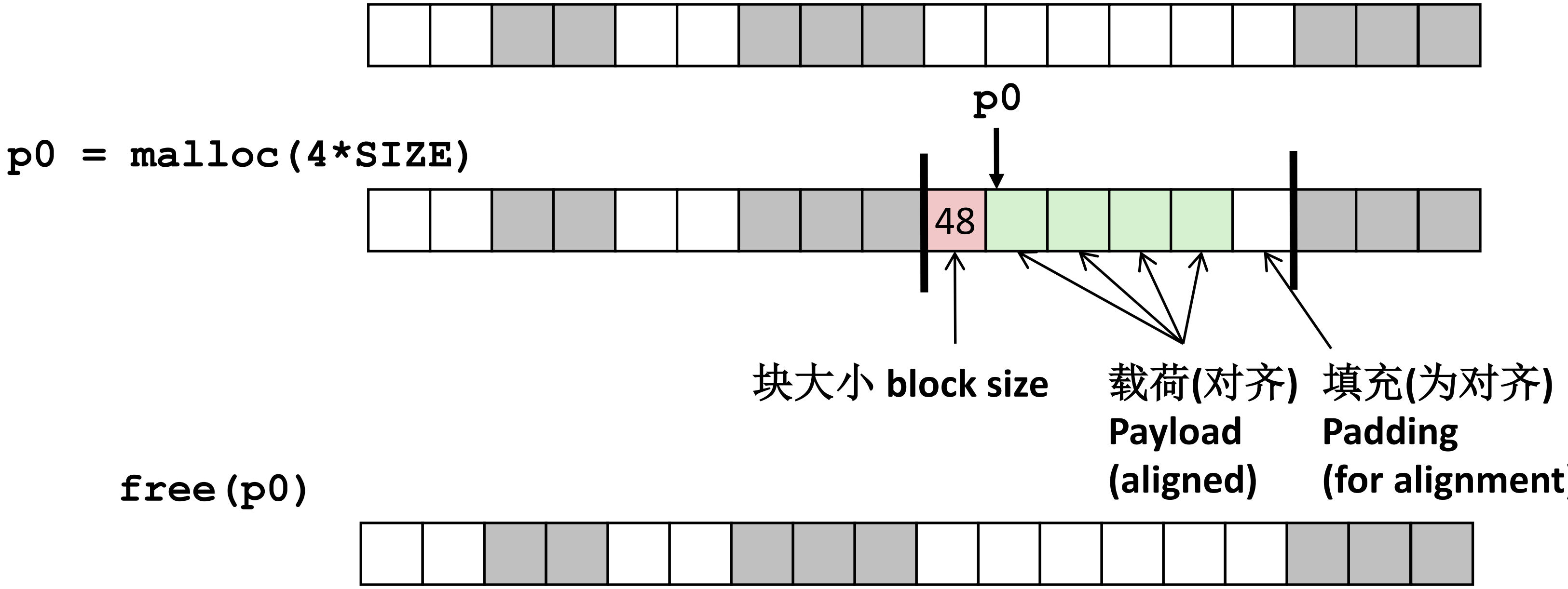
- **绿线：由于外部碎片导致的额外堆大小 Green line: additional heap size due to external fragmentation**
- **最佳匹配：一种分配策略 Best Fit: One allocation strategy**
  - （稍后讨论） (To be discussed later)
  - 总开销=本基准的8.3% Total overhead = 8.3% on this benchmark

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?

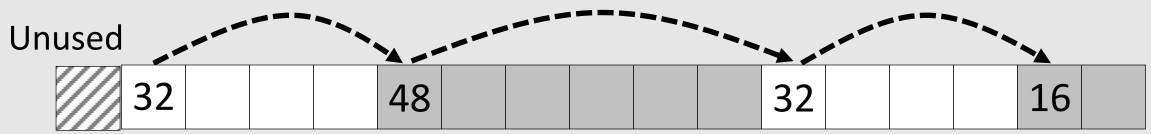
获取释放大小 Knowing How Much to Free

- 标准方法 **Standard method**
  - 在块之前的字中保存块长度 Keep the length (in bytes) of a block in the word *preceding* the block.
    - 包括头部 Including the header
    - 这个字称为头部域或者头部 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



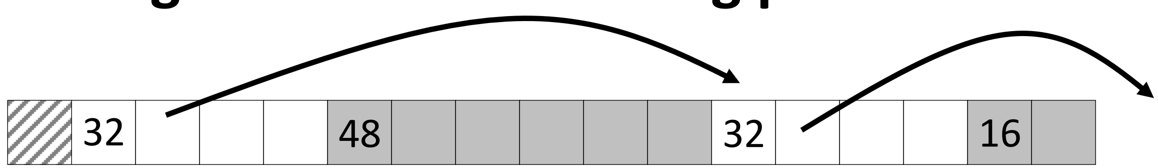
跟踪空闲块 Keeping Track of Free Blocks

- 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**



需要每个块标记为已分配/空闲
Need to tag each block as allocated/free

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



指针需要占空间
Need space for pointers

- 方法3：分离的空闲链表 **Method 3: *Segregated free list***
  - 不同大小块使用不同的空闲链表 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），每个空闲块内有指针和做为键值的长度 Can use a balanced tree (e.g., Red-Black tree) with pointers within each free block, and the length used as a key

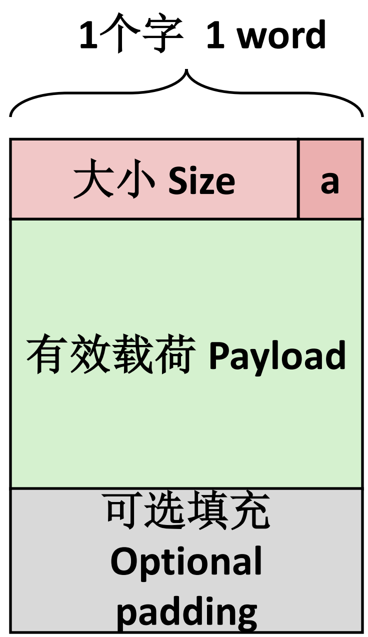
议题 Today

- 基本概念 Basic concepts
- **隐式空闲链表 Implicit free lists**

方法1：隐式空闲链表 Method 1: Implicit Free List

- 对每个块都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，还不如将其作为已分配/空闲的标志位 Instead of storing an always-0 bit, use it as a allocated/free flag
  - 读块大小那个字时需要将这些位屏蔽掉 When reading size word, must mask out this bit

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

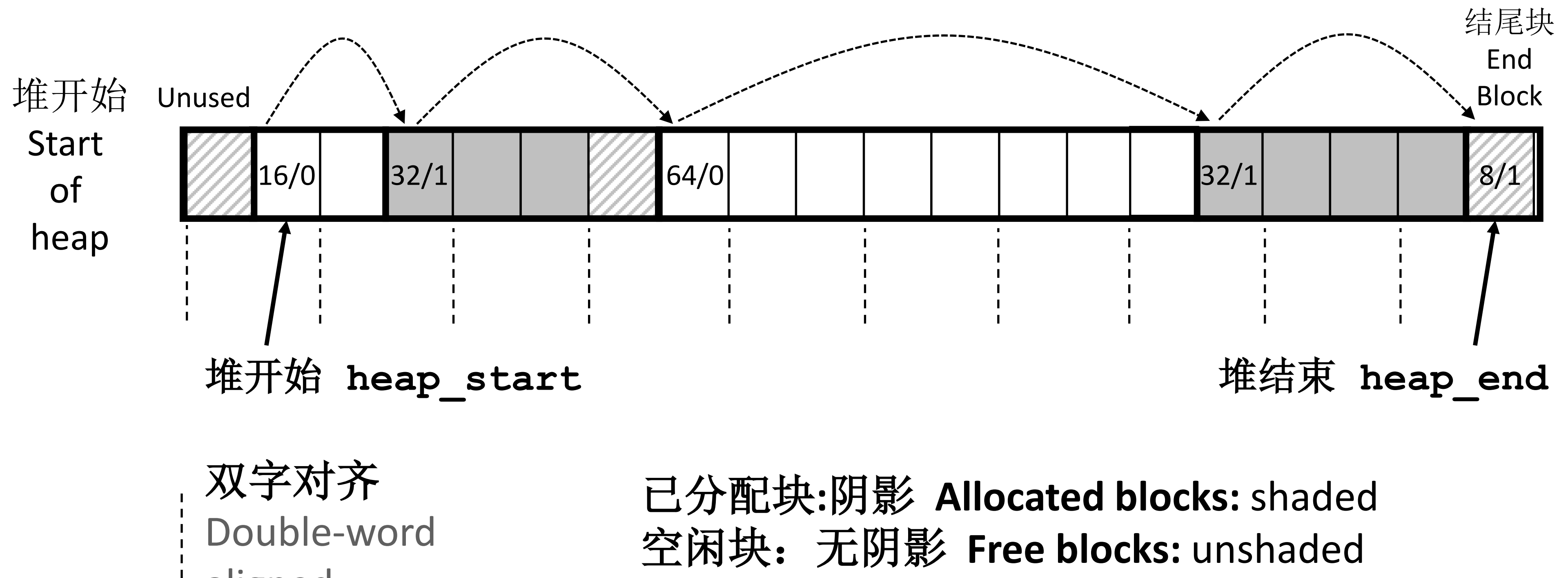


a = 1: Allocated block 分配的块
a = 0: Free block 空闲块

Size: block size 块大小

Payload: application data 载荷：应用数据（仅已分配的块）(allocated blocks only)

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



已分配块: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，头部位于非对齐位置 **Headers:** labeled with “size in words/allocated bit”
Headers are at non-aligned positions
➔ 有效载荷必须对齐 Payloads are aligned

隐式链表：数据结构
Implicit List: Data Structures

头部 header	有效载荷 payload

- 块声明 **Block declaration**

```
typedef uint64_t word_t;
```

```
typedef struct block
{
    word_t header;
    unsigned char payload[0];        // Zero length array
} block_t;
```

// `block_t *block`

- 从块指针获得有效载荷 **Getting payload from block pointer**

```
return (void *) (block->payload);
```

- 从有效载荷获得头部 **Getting header from payload** // bp points to a payload

```
return (block_t *) ((unsigned char *) bp
                     - offsetof(block_t, payload));
```

C语言函数offsetof(struct,member)返回member在struct中的偏移
C function `offsetof(struct, member)` returns offset of member within struct

隐式链表：访问头部
Implicit List: Header access

大小 Size	a

- 从头部获得分配位 **Getting allocated bit from header**

```
return header & 0x1;
```

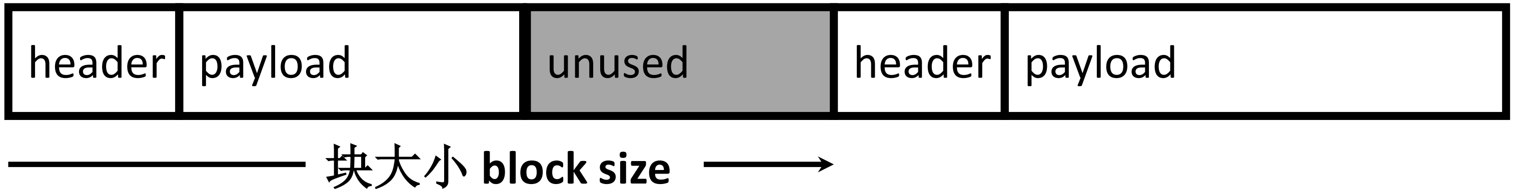
- 从头部获得块大小 **Getting size from header**

```
return header & ~0xfL;
```

- 初始化头部 **Initializing header** `//block_t *block`

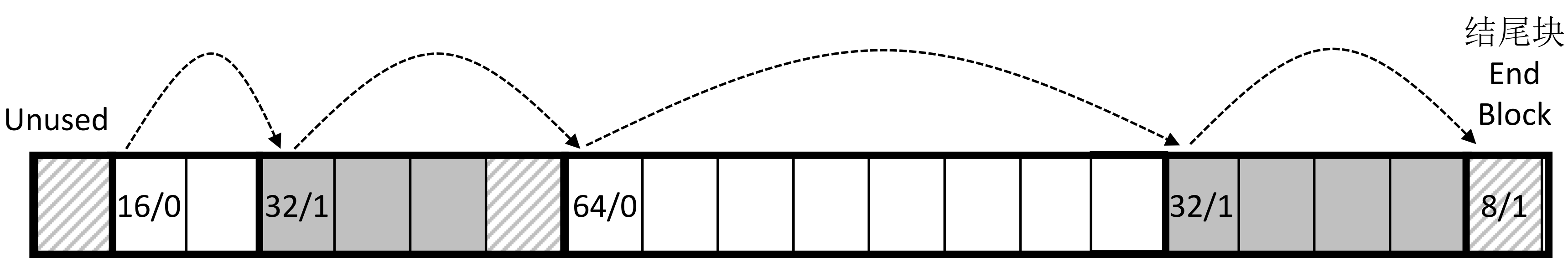
```
block->header = size | alloc;
```

隐式链表：遍历链表
Implicit List: Traversing list



- 查找下一个块 **Find next block**

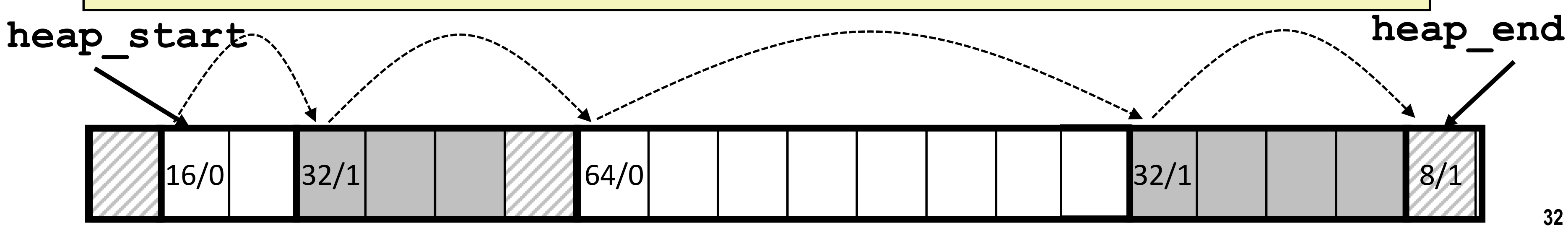
```
static block_t *find_next(block_t *block)
{
    return (block_t *) ((unsigned char *) block
                        + get_size(block));
}
```



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

- *首次匹配* ***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 that fits:
  - 查找asize字节的空间（包括头部） Finding space for **asize** bytes (including header):

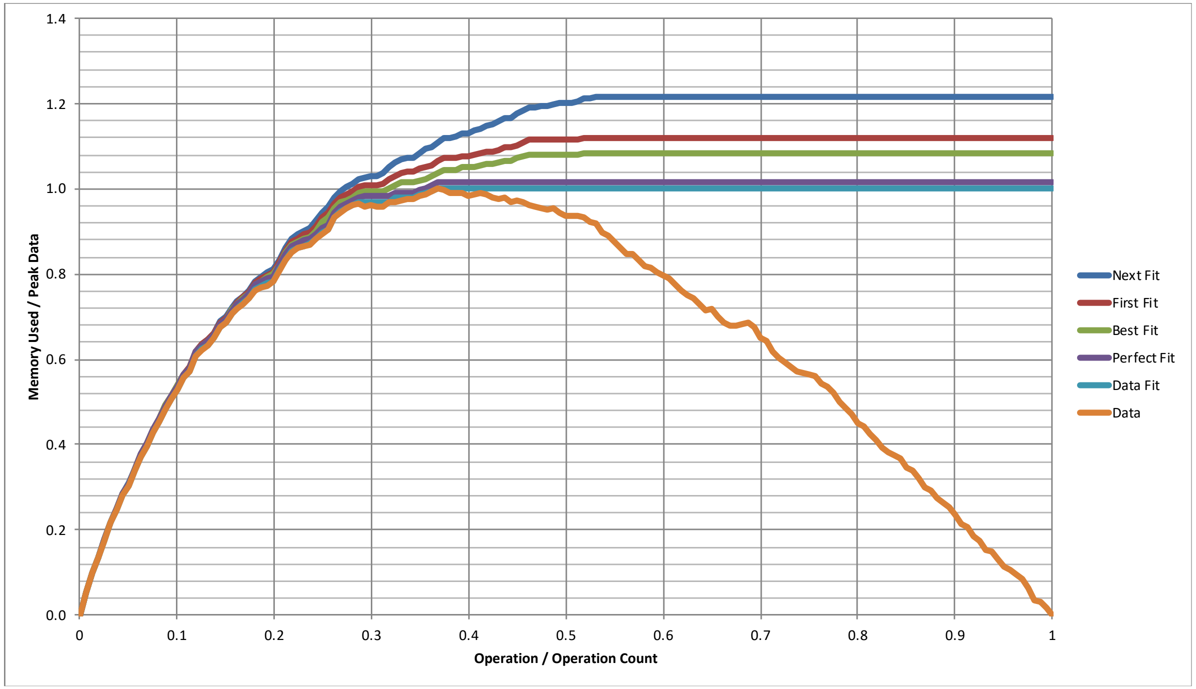
```
static block_t *find_fit(size_t asize)
{
    block_t *block;
    for (block = heap_start; block != heap_end;
         block = find_next(block)) {
    {
      if (!(get_alloc(block))
          && (asize <= get_size(block)))
         return block;
    }
    return NULL; // No fit found
}
```



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

- *首次匹配：* ***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 that fits:
  - 与总块数（分配和释放）成线性时间关系 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
- *下一次匹配：* ***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：避免了重扫描无用的块 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
- *最佳匹配：* ***Best fit:***
  - 从链表中选择最佳的空闲块：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

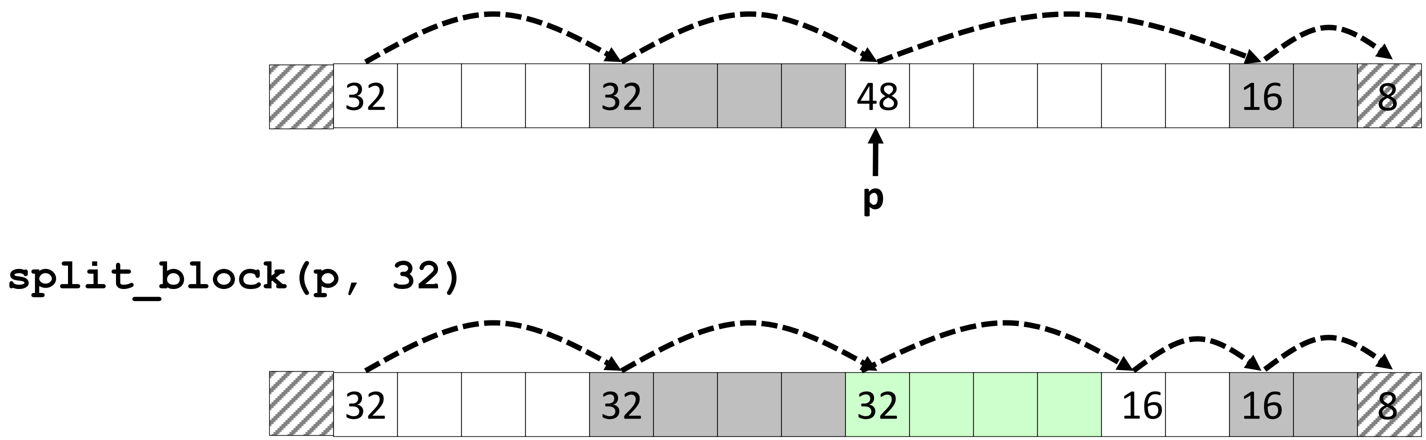
策略比较 Comparing Strategies



- 总开销（对本基准） **Total Overheads (for this benchmark)**
  - 完美匹配 Perfect Fit: 1.6%
  - 最佳匹配 Best Fit: 8.3%
  - 首次匹配 First Fit: 11.9%
  - 下次匹配 Next Fit: 21.6%

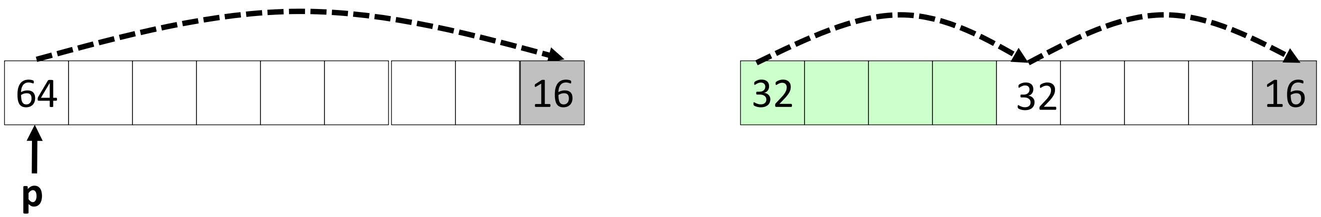
隐式链表：从空闲块中分配
Implicit List: Allocating in Free Block

- 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



隐式链表：拆分空闲块
Implicit List: Splitting Free Block

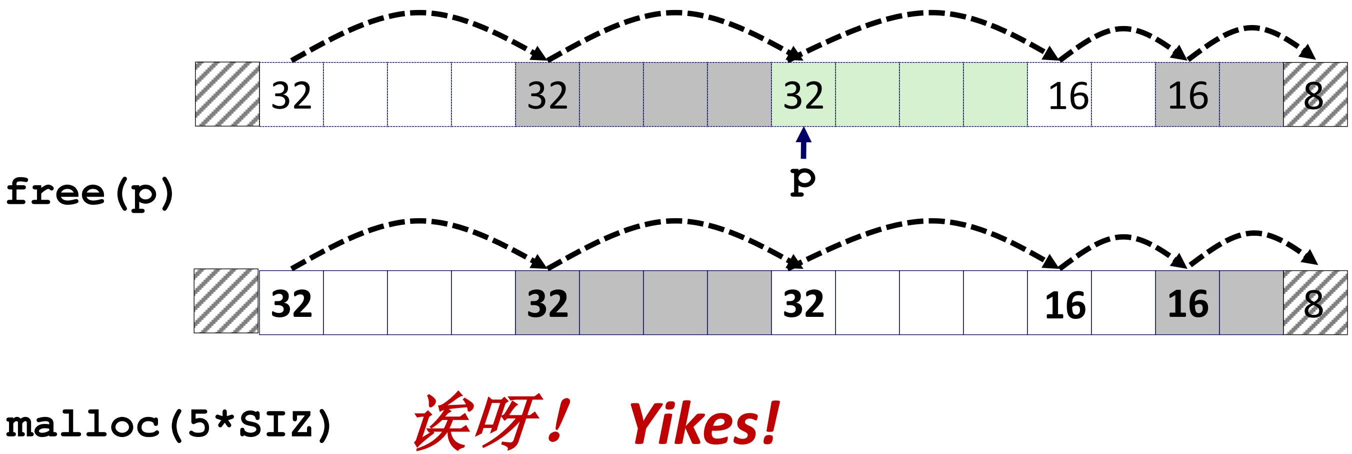
`split_block(p, 32)`




```
// Warning: This code is incomplete

static void split_block(block_t *block, size_t asize){
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next, block_size - asize, false);
    }
}
```

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



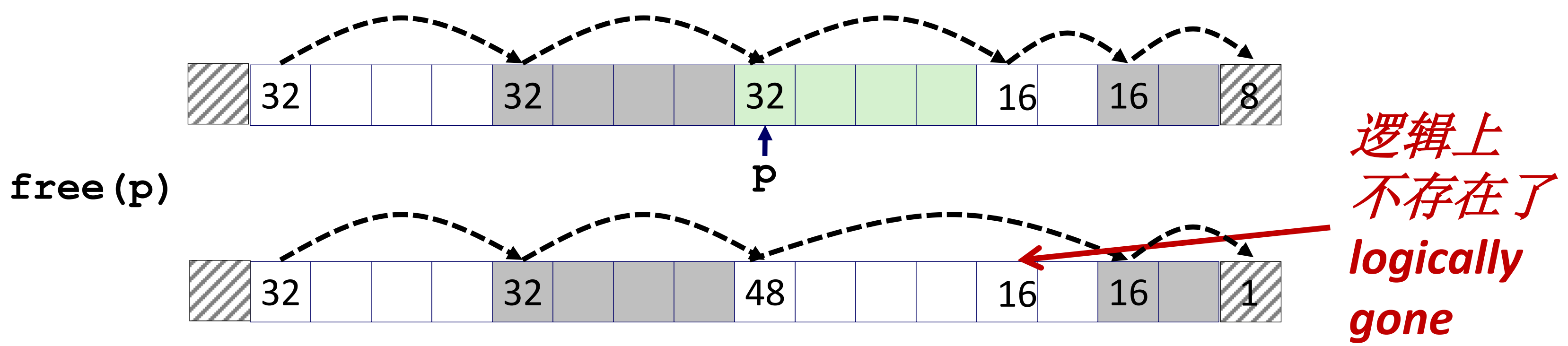
诶呀！ Yikes!

有足够的连续空闲空间，但是分配器找不到

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

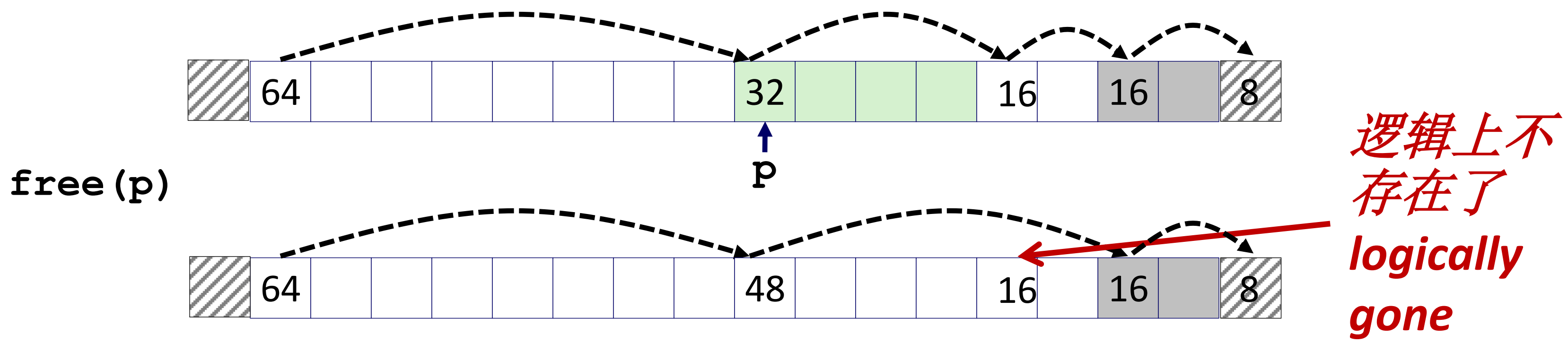
隐式链表：合并 Implicit List: Coalescing

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



隐式链表：合并 Implicit List: Coalescing

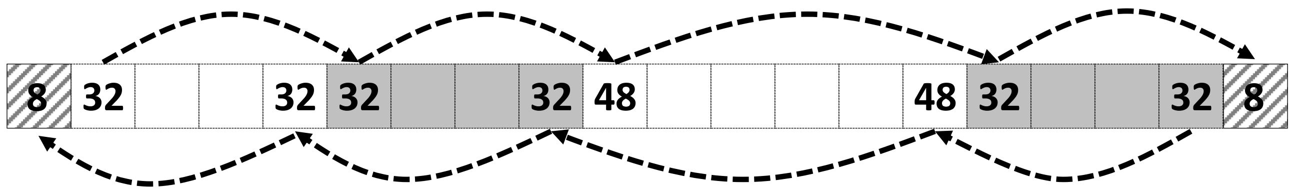
- 与下一个/前一个空闲块*合并*，如果有空闲块 **Join *(coalesce)* with next block, if it is free**
  - 与下一个块合并 Coalescing with next block



- 但是怎么和前一个块合并？ How do we coalesce with *previous* block?
  - 怎么知道从哪开始？ How do we know where it starts?
  - 怎么能确定是否已经分配出去了？ How can we determine whether its allocated?

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

- *边界标记* ***Boundary tags*** [Knuth73]
  - 在空闲块“底部”（结束）的位置复制块大小/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链表功能 Allows us to traverse the “list” backwards, but requires extra space
  - 重要和通用的技术 Important and general technique!



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

头部 **Header**

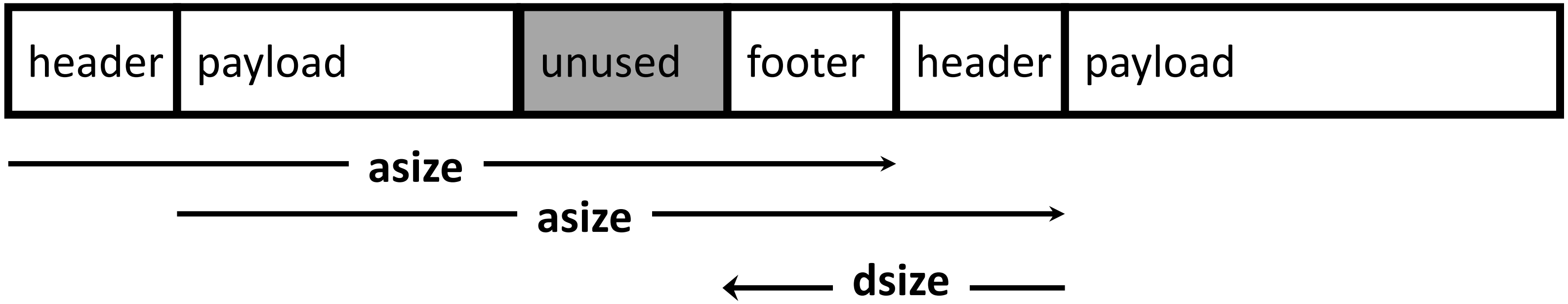
边界标记 **Boundary tag**
(脚部 **footer)**

a = 1: Allocated block 已分配块
a = 0: Free block 空闲块

Size: Total block size 大小：总的块大小

Payload: Application data 有效载荷：应用数据
(allocated blocks only) （仅已分配块）

脚部的实现 Implementation with Footers



- 定位当前块的脚部 **Locating footer of current block**

```
const size_t dsize = 2*sizeof(word_t);

static word_t *header_to_footer(block_t *block)
{
    size_t asize = get_size(block);
    return (word_t *) (block->payload + asize - dsize);
}
```

脚部的实现 Implementation with Footers

header	payload	unused	footer	header	payload

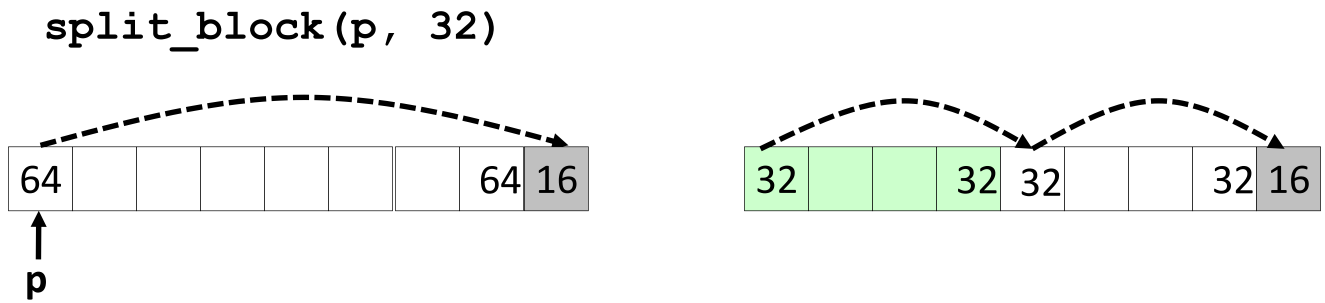
←

1个字 1 word

- 定位上一个块的脚部 **Locating footer of previous block**

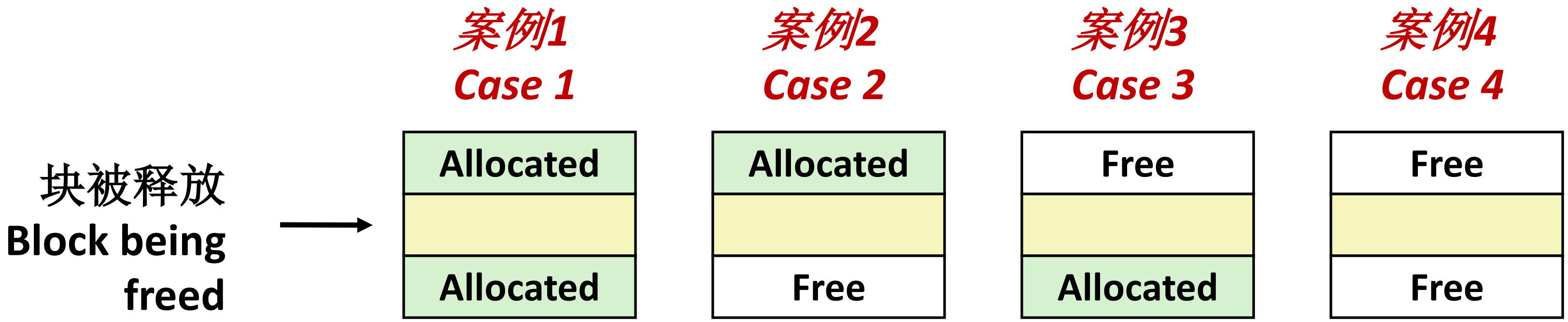
```
static word_t *find_prev_footer(block_t *block)
{
    return &(block->header) - 1;
}
```

拆分空闲块：完整版本
Splitting Free Block: Full Version

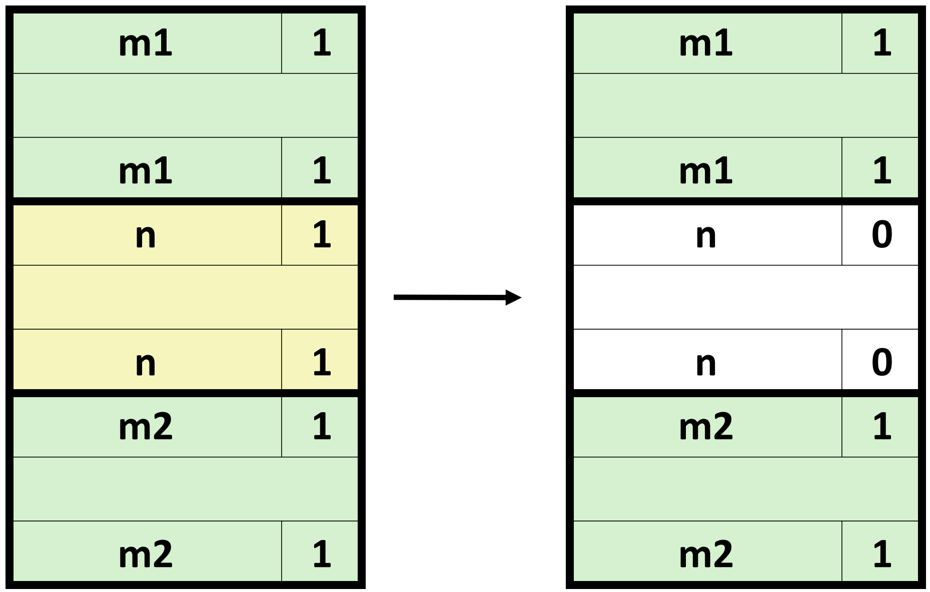



```
static void split_block(block_t *block, size_t asize){
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        write_footer(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next, block_size - asize, false);
        write_footer(block_next, block_size - asize, false);
    }
}
```

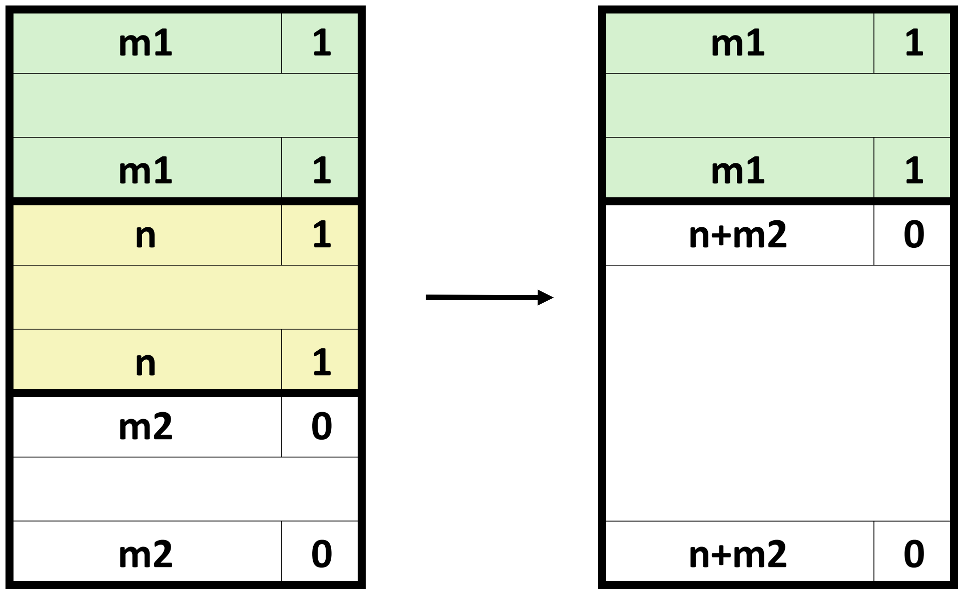
常量时间合并 Constant Time Coalescing



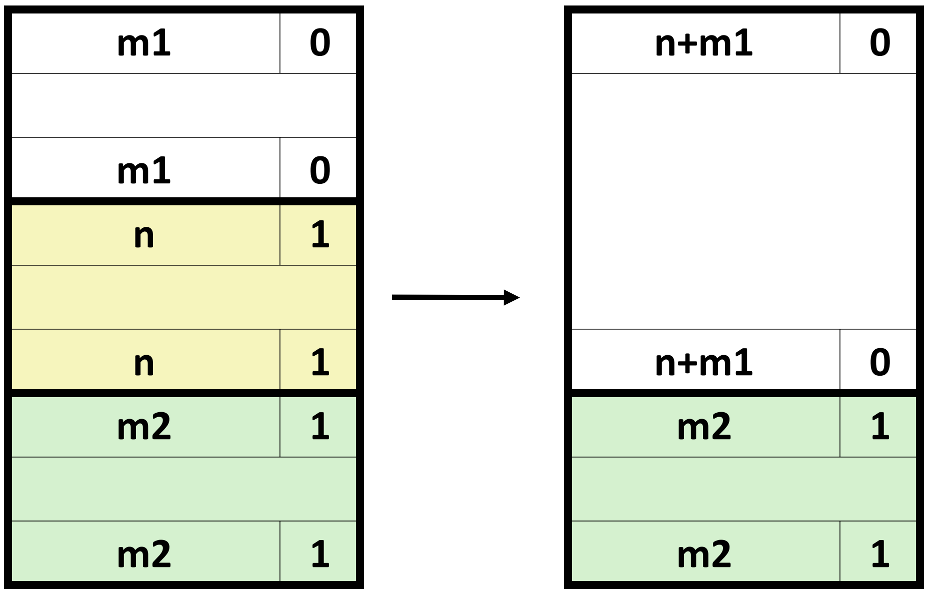
常量时间合并（案例1）
Constant Time Coalescing (Case 1)



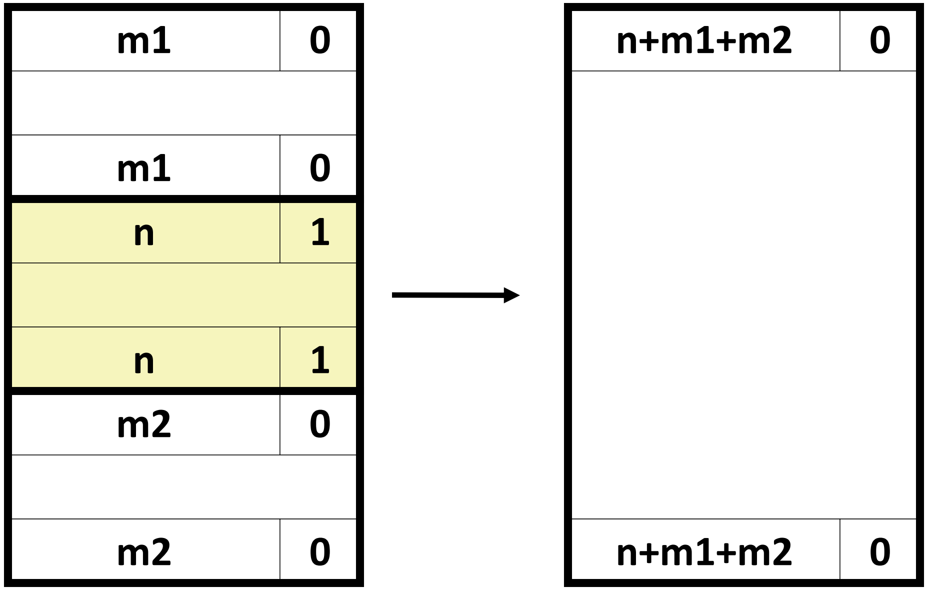
常量时间合并（案例2）
Constant Time Coalescing (Case 2)



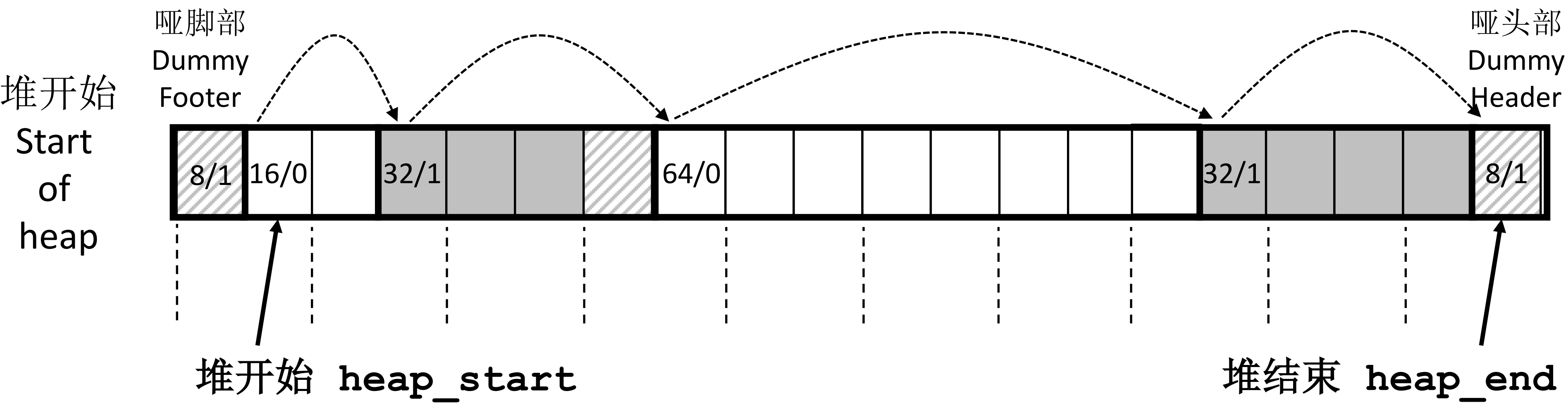
常量时间合并（案例3）
Constant Time Coalescing (Case 3)



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



堆结构 Heap Structure



- 第一个头部之前的哑脚部 **Dummy footer before first header**
  - 标记为已分配 Marked as allocated
  - 当释放第一个块时，防止意外合并 Prevents accidental coalescing when freeing first block
- 最后脚部之后的哑头部 **Dummy header after last footer**
  - 在释放最后一块时，防止意外合并 Prevents accidental coalescing when freeing final block

顶层Malloc代码 Top-Level Malloc Code

```
const size_t dsize = 2*sizeof(word_t);

void *mm_malloc(size_t size)
{
    size_t asize = round_up(size + dsize, dsize);

    block_t *block = find_fit(asize);

    if (block == NULL)
        return NULL;

    size_t block_size = get_size(block);
    write_header(block, block_size, true);
    write_footer(block, block_size, true);

    split_block(block, asize);

    return header_to_payload(block);
}
```

```
round_up(n, m)
=
m *((n+m-1)/m)
```

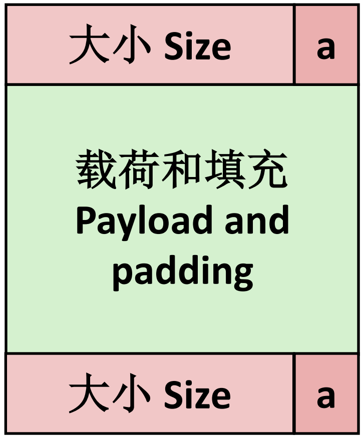
顶层Free代码 Top-Level Free Code

```
void mm_free(void *bp)
{
    block_t *block = payload_to_header(bp);
    size_t size = get_size(block);

    write_header(block, size, false);
    write_footer(block, size, false);

    coalesce_block(block);
}
```

边界标记的缺点 Disadvantages of Boundary Tags

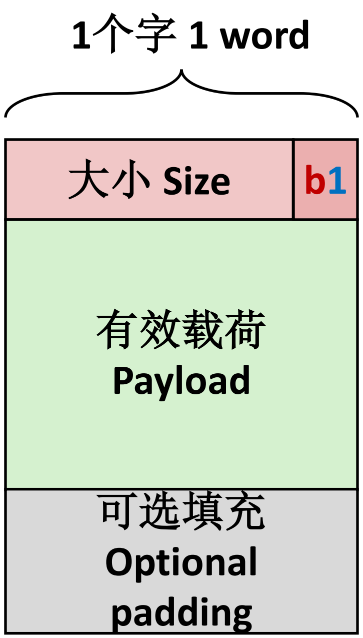
- 内部碎片 **Internal fragmentation**

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



已分配块没有边界标记
No Boundary Tag for Allocated Blocks

- 仅空闲块需要边界标记 **Boundary tag needed only for free blocks**
- 当块大小是16的整倍数，存在4个空闲位 **When sizes are multiples of 16, have 4 spare bits**

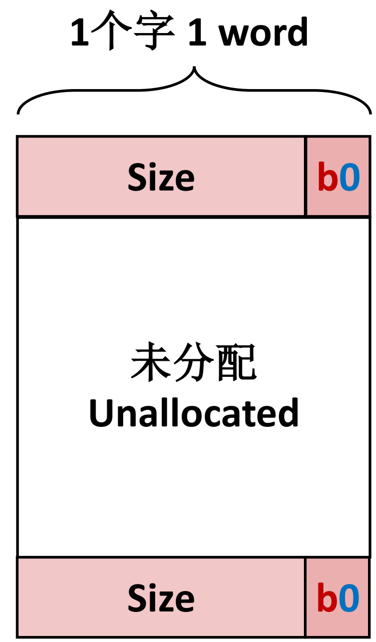


已分配块
Allocated Block

a = 1: Allocated block 已分配
a = 0: Free block 空闲块
上一个块已分配
b = 1: Previous block is allocated
上一个块是空闲的
b = 0: Previous block is free

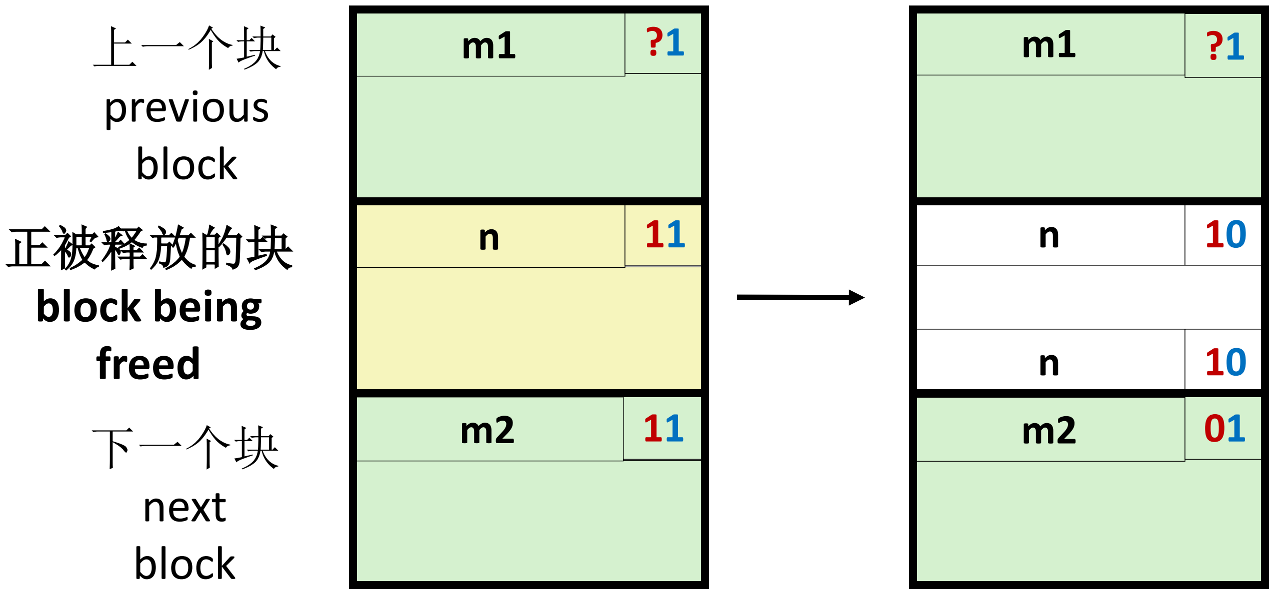
Size: block size 大小：块大小

Payload: application data 有效载荷：应用数据



空闲块
Free Block

已分配块没有边界标记（案例1）
No Boundary Tag for Allocated Blocks (Case 1)



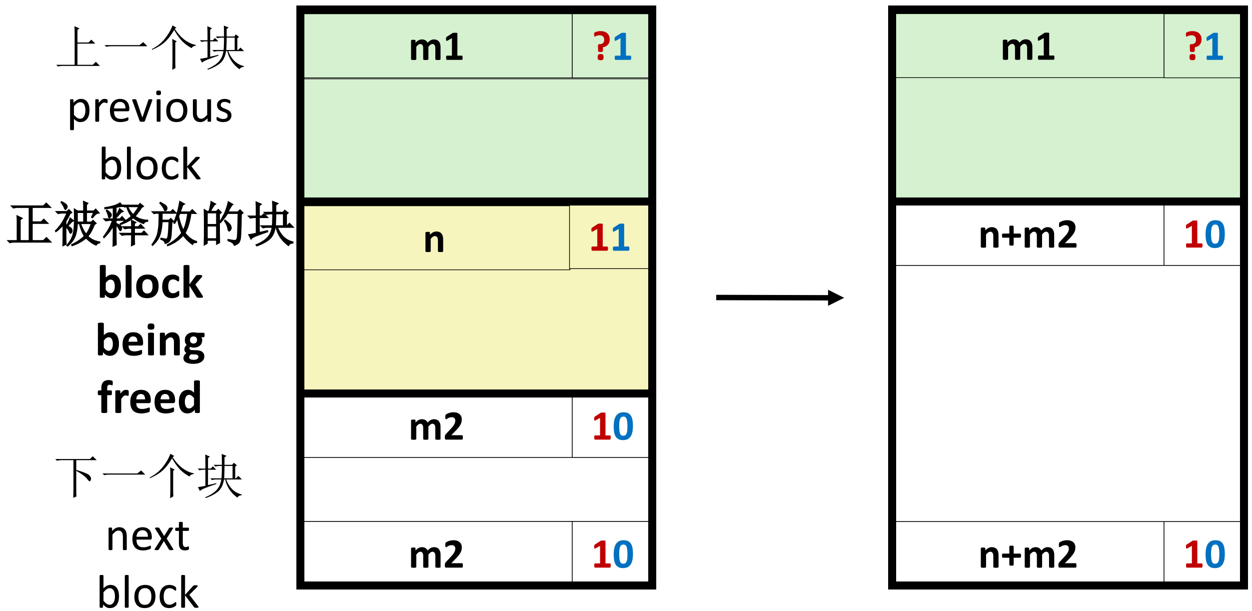
头部：使用2位（由于对齐的原因，这两个地址位始终为零）

Header: Use 2 bits (address bits always zero due to alignment):

上一个分配的块<<1 | 当前分配的块

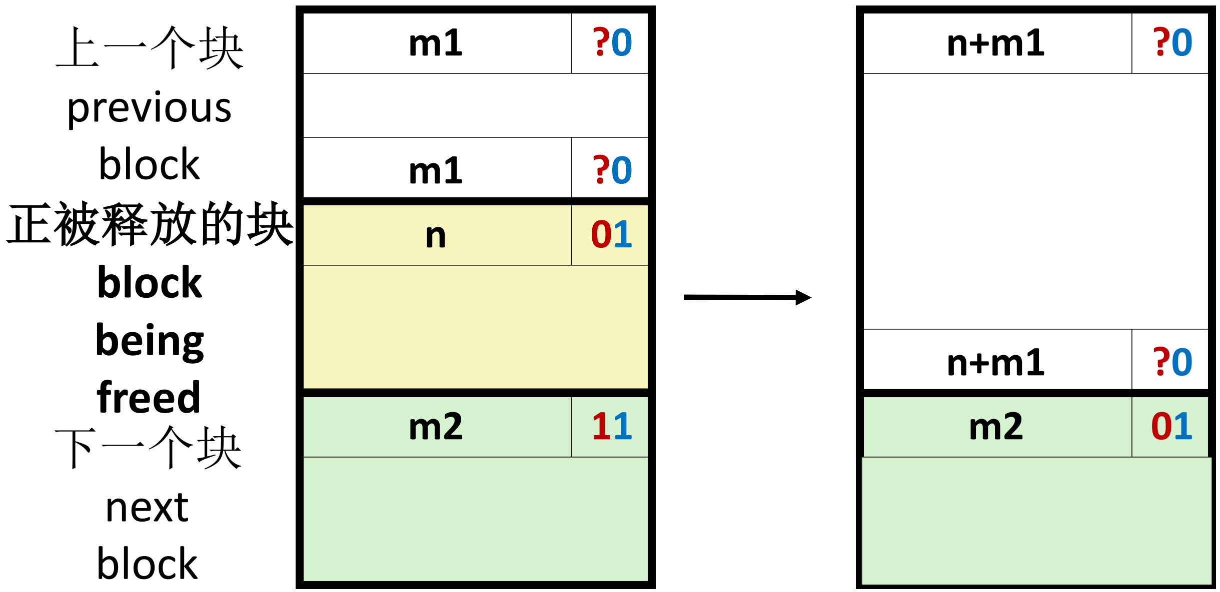
(previous block allocated)<<1 | (current block allocated)

已分配块没有边界标记（案例2）
No Boundary Tag for Allocated Blocks (Case 2)



头部：使用2位（由于对齐的原因，这两个地址位始终为零）
Header: Use 2 bits (address bits always zero due to alignment):
上一个分配的块<<1 | 当前分配的块
(previous block allocated)<<1 | (current block allocated)

已分配块没有边界标记（案例3）
No Boundary Tag for Allocated Blocks (Case 3)



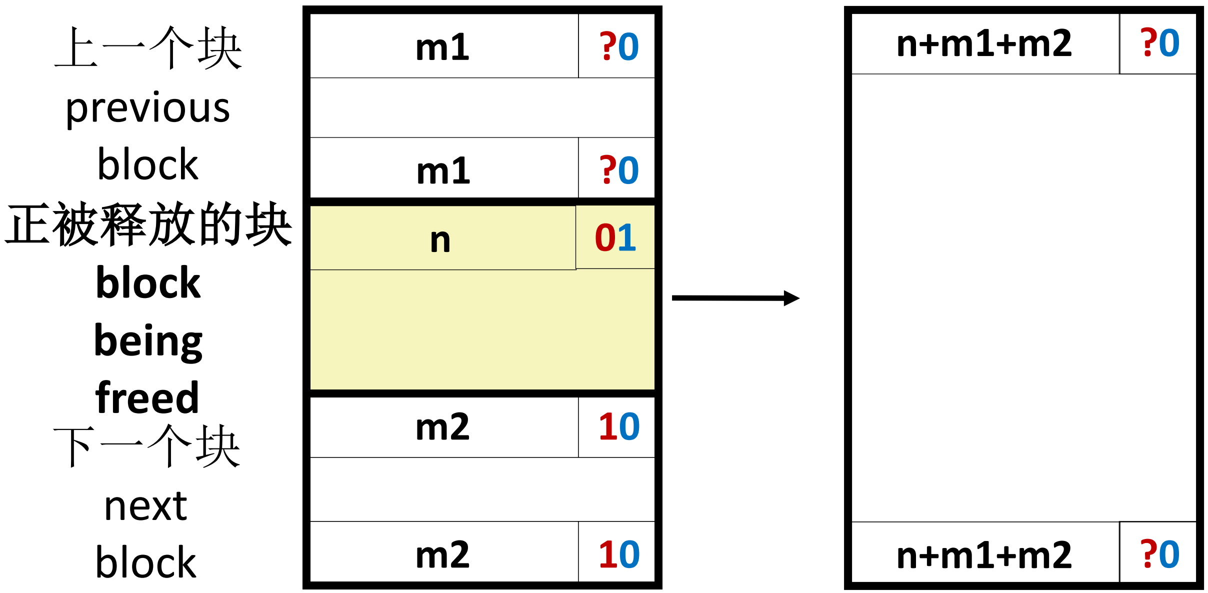
头部：使用2位（由于对齐的原因，这两个地址位始终为零）

Header: Use 2 bits (address bits always zero due to alignment):

上一个分配的块<<1 | 当前分配的块

(previous block allocated)<<1 | (current block allocated)

已分配块没有边界标记（案例4）
No Boundary Tag for Allocated Blocks (Case 4)



头部：使用2位（由于对齐的原因，这两个地址位始终为零）
Header: Use 2 bits (address bits always zero due to alignment):
上一个分配的块<<1 | 当前分配的块
(previous block allocated)<<1 | (current block allocated)

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

- 选择策略 **Placement policy:**
  - 首次匹配、下一次匹配、最佳匹配等 First-fit, next-fit, best-fit, etc.
  - 在更低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***：分离的空闲链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，例如： ***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

隐式链表：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
  - 首次匹配、下一次匹配或最佳匹配 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的概念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**