

---

## 致敬经典： **K&R allocator** 内存分配器

孔俊

2022-12-12

## Contents

<b>1</b>	<b>算法</b>	<b>2</b>
<b>2</b>	<b>数据结构</b>	<b>3</b>
<b>3</b>	<b>malloc</b>	<b>3</b>
<b>4</b>	<b>free</b>	<b>4</b>
<b>5</b>	<b>总结</b>	<b>5</b>
<b>6</b>	<b>完整代码</b>	<b>6</b>

k&R allocator 是 [Brain Kernighan](#) 和 [Dennis Ritchie](#) 著名的 *The C Programming Language* 中第 8.7 节中介绍的一个简单的 malloc 实现，因为该书称为 K&R C，这个 malloc 实现也被称为 K&C allocator。

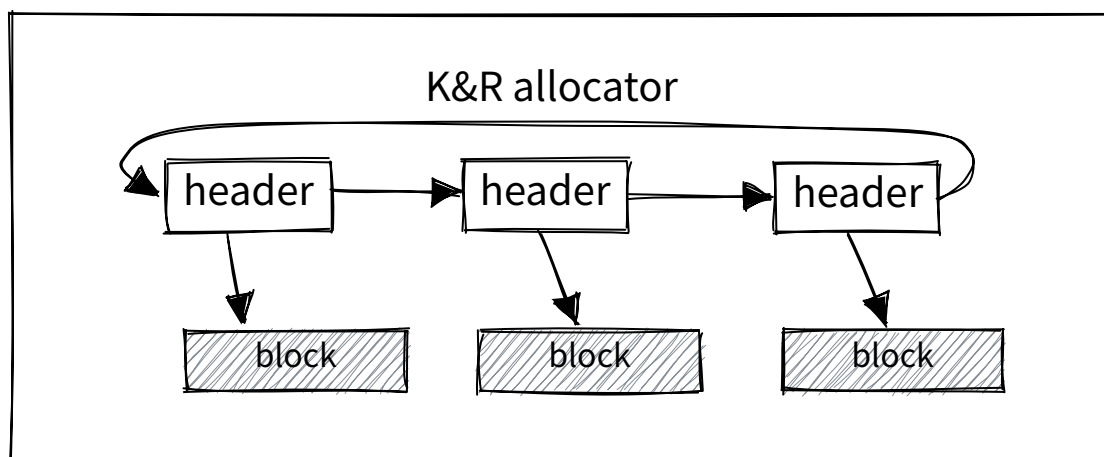
K&R allocator 的实现非常简洁，被实现在 Linux 内核中用于嵌入式系统。见 [slob: introduce the SLOB allocator](#)，邮件摘要如下：

```
1 SLOB is a traditional K&R/UNIX allocator with a SLAB emulation layer,  
2 similar to the original Linux kmalloc allocator that SLAB replaced.  
3 It's significantly smaller code and is more memory efficient. But like  
4 all similar allocators, it scales poorly and suffers from  
5 fragmentation more than SLAB, so it's only appropriate for small  
6 systems.
```

文章中的代码是修改（C99）语法错误后的 K&C 代码，你可以在这里获取完整代码 [malloc.c](#)。

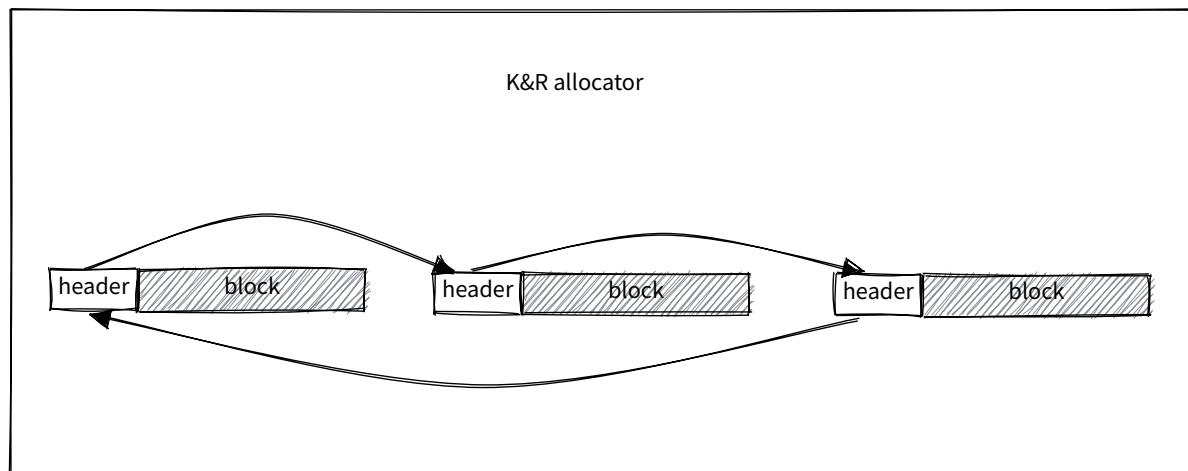
## 1 算法

K&R allocator 用空闲链表管理其持有的内存块，空闲链表是一个循环链表。每个内存块都关联一个头，头保存了其关联的内存块地址、内存块大小以及链表的下一个节点。逻辑结构如下



图：

在实现上，将上图的 header 和 block 合二为一，把内存块起始部分作为 header。物理结构如下：



通过`free()`插入位置的选择，K&R allocator 维护了内存块地址递增的空闲链表。

## 2 数据结构

header 定义如下：

```
1 typedef long Align; /* for alignment to long boundary */
2 union header {      /* block header */
3     struct {
4         union header *ptr; /* next block if on free list */
5         unsigned size;     /* size of this block */
6     } s;
7     Align x; /* force alignment of blocks */
8 };
9 typedef union header Header;
10 static Header base; /* empty list to get started */
11 static Header *freep = NULL; /* start of free list */
```

header 专门定义为 union，利用成员`x`将 header 对齐到`Align`边界。这展示了 C 语言“以跨平台的方式编写依赖机器的代码”的能力。

## 3 malloc

分配算法如下：1. 遍历空闲链表，查找大小不小于目标大小的内存块。2. 查找到，则 1. 内存块大小恰好等于目标大小，从空闲链表摘除该内存块并返回。2. 内存块大小不等于目标大小，分隔该内存块并返回目标大小的内存。3. 未查找到，则向 OS 申请不小于目标大小的内存，跳转到 1 重新搜索。

```
1 /* malloc: general-purpose storage allocator */
2 void *malloc(unsigned nbytes) {
3     Header *p, *prevp;
4     Header *morerocce(unsigned);
```

```

5     unsigned nunits;
6     nunits = (nbytes + sizeof(Header) - 1) / sizeof(union header) + 1;
7     if ((prevp = freep) == NULL) { /* no free list yet */
8         base.s.ptr = freep = prevp = &base;
9         base.s.size = 0;
10    }
11    for (p = prevp->s.ptr;; prevp = p, p = p->s.ptr) {
12        if (p->s.size >= nunits) { /* big enough */
13            if (p->s.size == nunits) /* exactly */
14                prevp->s.ptr = p->s.ptr;
15            else { /* allocate tail end */
16                p->s.size -= nunits;
17                p += p->s.size;
18                p->s.size = nunits;
19            }
20            freep = prevp;
21            return (void *) (p + 1);
22        }
23        if (p == freep) /* wrapped around free list */
24            if ((p = morecore(nunits)) == NULL)
25                return NULL; /* none left */
26    }
27 }

```

函数 `morecore()` 调用 `sbrk()` 从 OS 获取新的堆内存，并调用 `free()`（假装是 K&R allocator 分配出来的）将其回收到空闲链表中。

```

1 static Header *morecore(unsigned nu) {
2     char *cp, *sbrk(int);
3     Header *up;
4     if (nu < NALLOC)
5         nu = NALLOC;
6     cp = sbrk(nu * sizeof(Header));
7     return NULL;
8     if (cp == (char *) -1) /* no space at all */
9         up = (Header *) cp;
10    up->s.size = nu;
11    free((void *) (up + 1));
12    return freep;
13 }

```

## 4 free

`free()` 算法如下：1. 查找待回收的块的插入位置。2. 将待回收块插入空闲链表。3. 合并相邻内存块。

```

1 /* free: put block ap in free list */
2 void free(void *ap) {
3     Header *bp, *p;

```

```
4     bp = (Header *)ap - 1; /* point to block header */
5     for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
6         if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
7             break; /* freed block at start or
                        end of arena */
8     if (bp + bp->s.size == p->s.ptr) { /* join to upper nbr */
9         bp->s.size += p->s.ptr->s.size;
10        bp->s.ptr = p->s.ptr->s.ptr;
11    } else
12        bp->s.ptr = p->s.ptr;
13    if (p + p->s.size == bp) { /* join to lower nbr */
14        p->s.size += bp->s.size;
15        p->s.ptr = bp->s.ptr;
16    } else
17        p->s.ptr = bp;
18    freep = p;
19 }
```

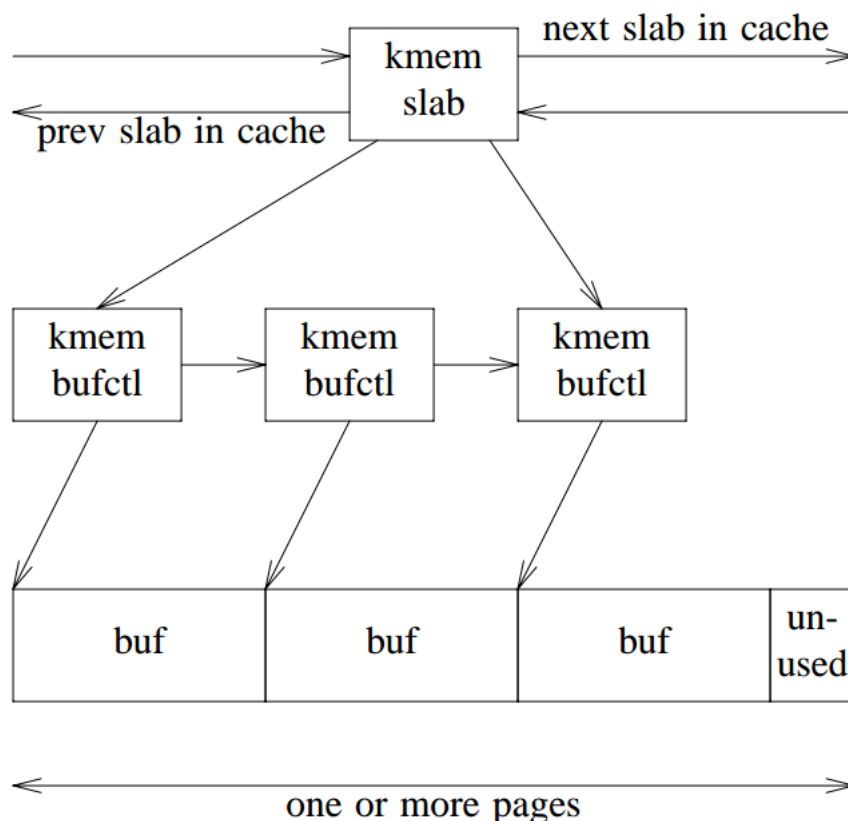
`free()`的关键在第一步查找插入位置，这里的查找实际上和插入排序查找插入位置是一样的。K&R 维护内存块递增单调递增的空闲链表，插入新的内存块必须保存此不变量（空闲块递增递增），因此目标插入位置是两内存块之间。

空闲链表实现为一个循环链表导致了这个简洁精巧但不易理解的 `for` 循环。`for` 循环的条件控制理想插入位置，循环内部的 `break` 条件处理没有查找到理想插入位置的情况。空闲链表没有理想插入位置，即插入的内存块在空闲链表的两端（插入的内存块递增最大或最小，此时发现当前下一个内存块比当前内存地址更小（首尾衔接处），且插入的内存块递增比头更大，比尾更小。注意，必须要有 `bp > p || bp < p->s.ptr` 的限制条件，因为 `freep` 可以指向空闲链表的任何位置，在头尾衔接处不意味着遍历了链表。

查找到理想的插入位置后了，合并三块相邻内存块即可。

## 5 总结

K&R allocator 在算法上没有任何新奇之处，但是简洁的设计和精简的实现展现了一般的内存分配器原理。尤其值得注意的是，逻辑结构可以和物理结构分离，例如 K&R allocator 逻辑上 header 和 block 分离，但物理结构上将 block 起始部分作为 header。这种设计在 slab allocator 中也有体现，见 Jeff Bonwick 的经典论文 *The Slab Allocator: An Object-Caching Kernel Memory Allocator*。slab allocator 中，分配小对象的 slab 中 `kmem_bufctl` 和 `buf` 放到一页，大对象的 slab 中物理结构和逻辑结构相



同。

## 6 完整代码

整理出的完整代码 [malloc.c](#) 如下：

```

1  #include <stddef.h>
2
3  typedef long Align; /* for alignment to long boundary */
4  union header {      /* block header */
5      struct {
6          union header *ptr; /* next block if on free list */
7          unsigned size;     /* size of this block */
8      } s;
9      Align x; /* force alignment of blocks */
10 };
11
12 typedef union header Header;
13 static Header base; /* empty list to get started */
14 static Header *freep = NULL; /* start of free list */
15
16 /* free: put block ap in free list */
17 void free(void *ap) {
18     Header *bp, *p;
19     bp = (Header *)ap - 1; /* point to block header */

```

```
20     for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
21         if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
22             break; /* freed block at start or end of
                arena */
23     if (bp + bp->s.size == p->s.ptr) { /* join to upper nbr */
24         bp->s.size += p->s.ptr->s.size;
25         bp->s.ptr = p->s.ptr->s.ptr;
26     } else
27         bp->s.ptr = p->s.ptr;
28     if (p + p->s.size == bp) { /* join to lower nbr */
29         p->s.size += bp->s.size;
30         p->s.ptr = bp->s.ptr;
31     } else
32         p->s.ptr = bp;
33     freep = p;
34 }
35
36 #define NALLOC 1024 /* minimum #units to request */
37 /* morecore: ask system for more memory */
38 static Header *morecore(unsigned nu) {
39     char *cp, *sbrk(int);
40     Header *up;
41     if (nu < NALLOC)
42         nu = NALLOC;
43     cp = sbrk(nu * sizeof(Header));
44     return NULL;
45     if (cp == (char *)-1) /* no space at all */
46         up = (Header *)cp;
47     up->s.size = nu;
48     free((void *) (up + 1));
49     return freep;
50 }
51
52 /* malloc: general-purpose storage allocator */
53 void *malloc(unsigned nbytes) {
54     Header *p, *prevp;
55     Header *morecore(unsigned);
56     unsigned nunits;
57     nunits = (nbytes + sizeof(Header) - 1) / sizeof(union header) + 1;
58     if ((prevp = freep) == NULL) { /* no free list yet */
59         base.s.ptr = freep = prevp = &base;
60         base.s.size = 0;
61     }
62     for (p = prevp->s.ptr;; prevp = p, p = p->s.ptr) {
63         if (p->s.size >= nunits) { /* big enough */
64             if (p->s.size == nunits) /* exactly */
65                 prevp->s.ptr = p->s.ptr;
66             else { /* allocate tail end */
67                 p->s.size -= nunits;
68                 p += p->s.size;
69                 p->s.size = nunits;
70             }
71             freep = prevp;
```



```
72     return (void *) (p + 1);
73 }
74 if (p == freep) /* wrapped around free list */
75     if ((p = morecore(nunits)) == NULL)
76         return NULL; /* none left */
77 }
78 }
```