

c010606f: 55
c0106070: 89 e5
c0106072: 56
c0106073: 53
c0106074: 83 ec 0c
c0106077: 68 44 a2 12 c0
c010607c: e8 ed 1c 00 00
c0106081: be 00 60 10 00
c0106086: c1 ee 0c
c0106089: 83 c4 08
c010608c: 56
c010608d: 68 6c a2 12 c0
c0106092: e8 d7 1c 00 00
c0106097: 83 c4 10
c010609a: bb 00 00 00 00
c010609f: eb 14
c01060a1: 89 d8
c01060a3: c1 e0 0c
c01060a6: 83 ec 0c
c01060a9: 50
c01060aa: e8 9f 0a 00 00
c01060af: 83 c3 01
c01060b2: 83 c4 10
c01060b3: f3
c01060b4: 83 c4 10
c01060b5: 83 c4 10
c01060b6: 83 c4 10
c01060b7: 83 c4 10
c01060b8: 83 c4 10
c01060b9: 83 c4 10
c01060ba: 83 c4 10
c01060bb: 83 c4 10
c01060bc: 83 c4 10
c01060bd: 83 c4 10
c01060be: 83 c4 10
c01060bf: 83 c4 10
c01060c0: 83 c4 10
c01060c1: 83 c4 10
c01060c2: 83 c4 10
c01060c3: 83 c4 10
c01060c4: 83 c4 10
c01060c5: 83 c4 10
c01060c6: 83 c4 10
c01060c7: 83 c4 10
c01060c8: 83 c4 10
c01060c9: 83 c4 10
c01060ca: 83 c4 10
c01060cb: 83 c4 10
c01060cc: 83 c4 10
c01060cd: 83 c4 10
c01060ce: 68 af a0 12 c0

```
push %ebp
mov %esp,%ebp
push %esi
push %ebx
sub $0x4,%esp
push $0xc012a26c
call c0107d6e <pthread_mutex_lock@plt>
add $0x10,%esp
mov $0xc012a26c,%ebx
jmp c01060b5 <_kernel_post_init+0x46>
mov %ebx,%eax
shl $0xc,%eax
sub $0xc,%esp
push %eax
call c0106b4e <vmm_unmap_page>
add $0x1,%ebx
add $0x10,%esp
cmp %esi,%ebx
c01060a1 <_kernel_post_init+0x32>
c0107312 <kalloc_init>
mov %eax,%eax
c01060c9 <_kernel_post_init+0x5a>
leal 3(%ebp),%esp
pop %ebx
pop %esi
pop %ebp
ret
sub $0x4,%esp
push $0x40
push $0xc012a26c
```



LunaixOS

从零开始

DEVELOPING YOUR OWN

自制操作系统

OPERATING SYSTEM FROM SCRATCH

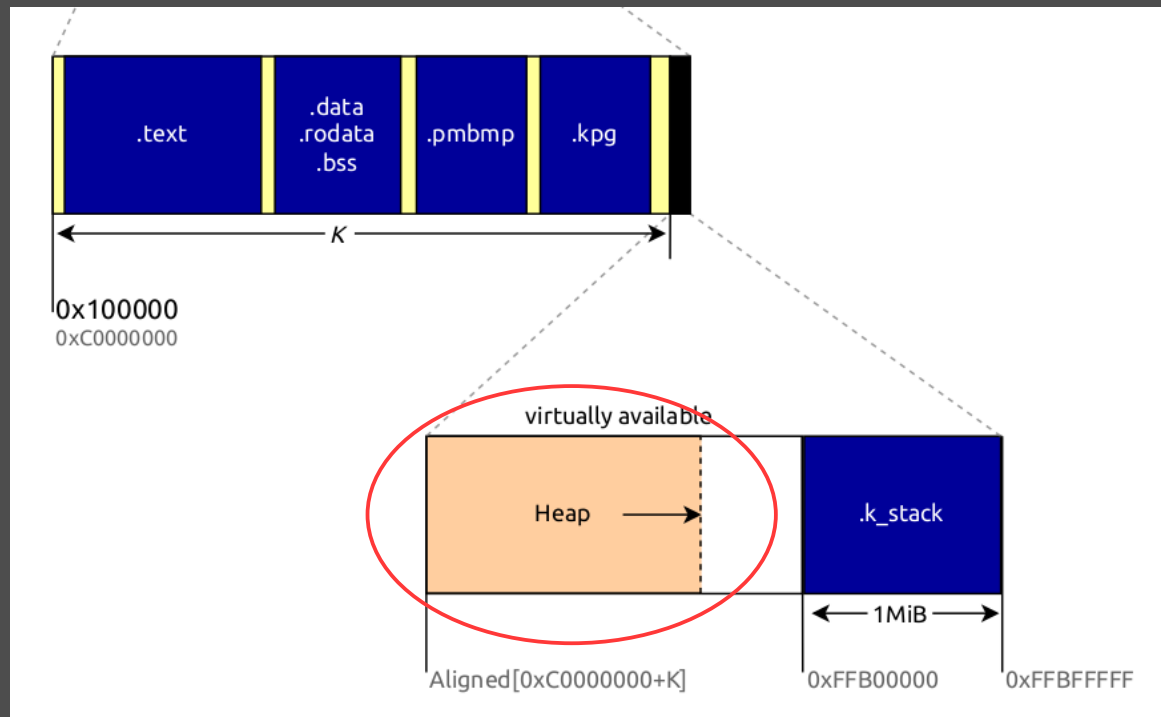
O Brave New Kernel

虚拟内存：实现 malloc

OPERATING SYSTEM DEVELOPMENT
TUTORIAL SERIES

EP 7-3





计划赶不上变化 —— 动态地，按需创建的空间

`malloc()`

`calloc()`

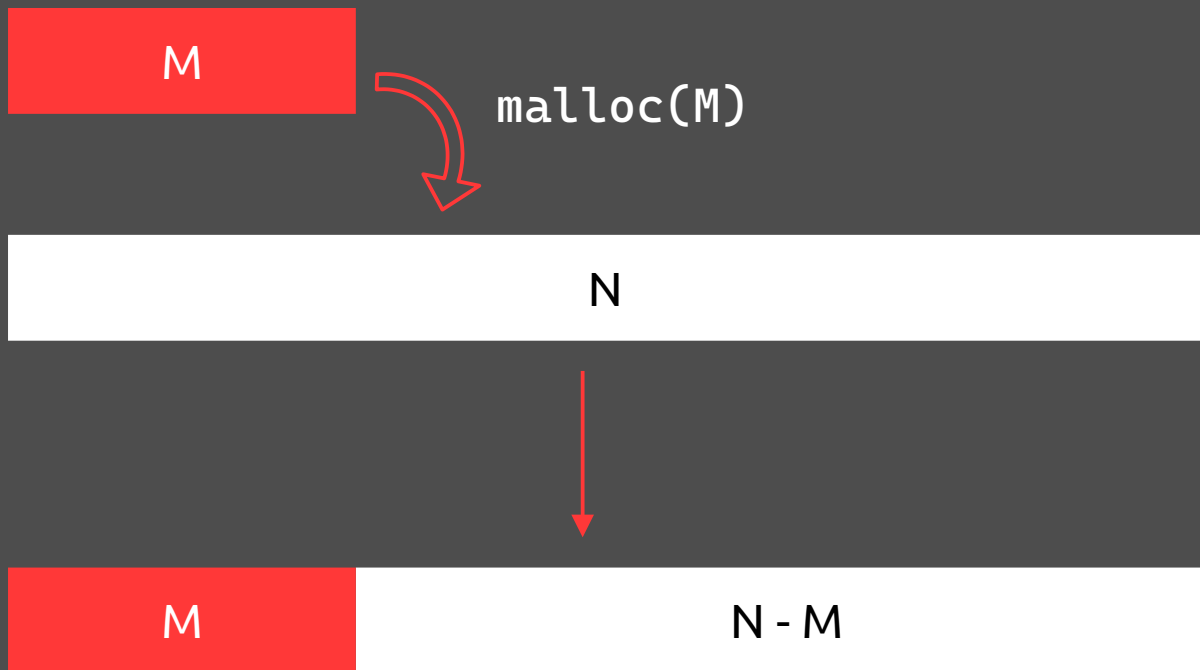
`realloc()`

`free()`

堆管理 [2]

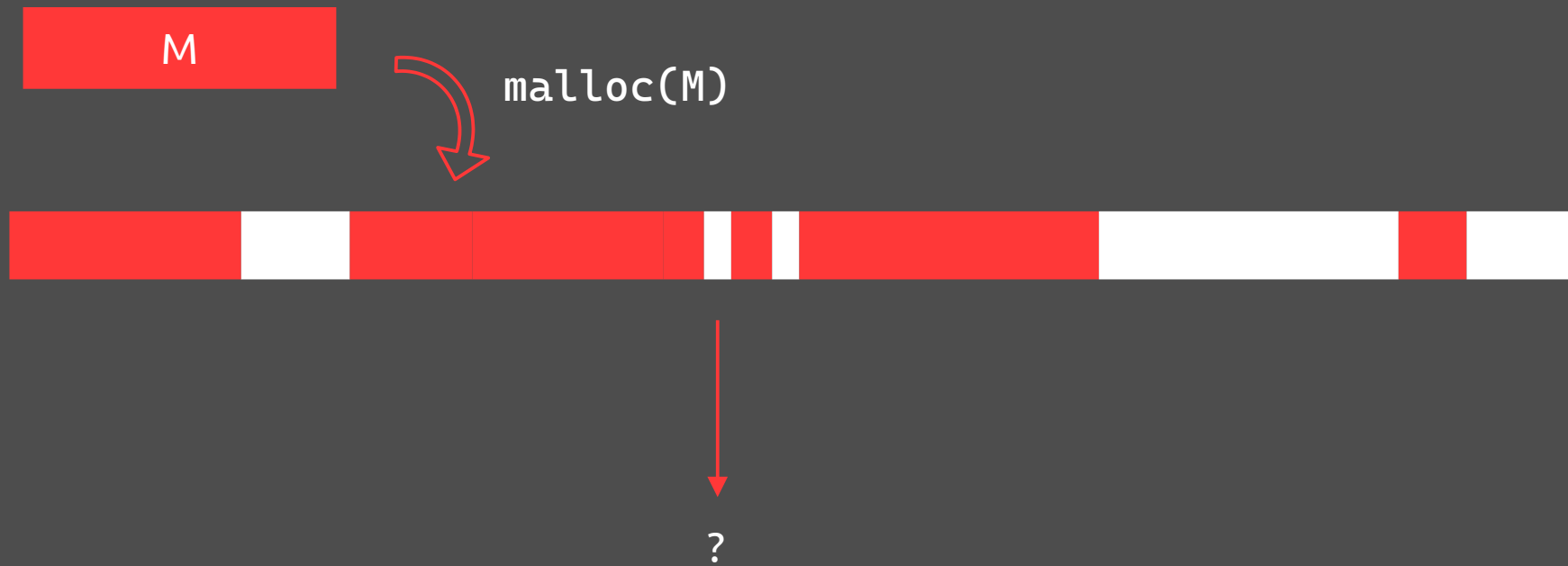


malloc 之原理





malloc 之原理



怎么办?



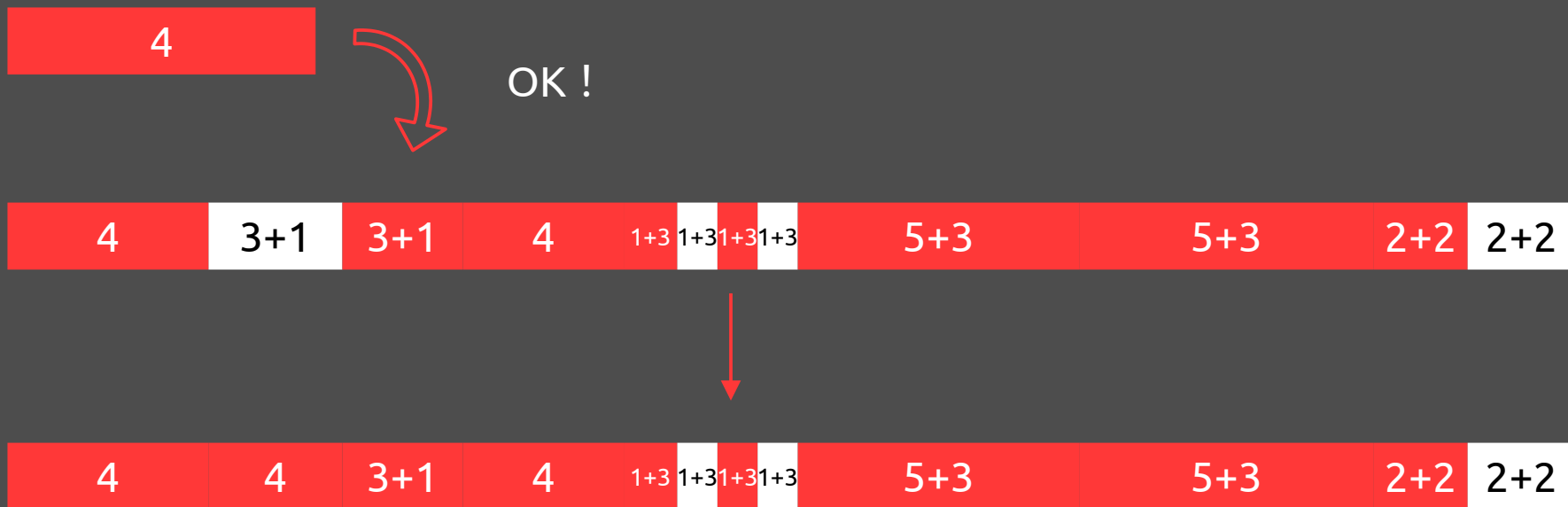
假设我们的 malloc 足够聪明，知道在哪里安放



外碎片 (External Fragmentation)



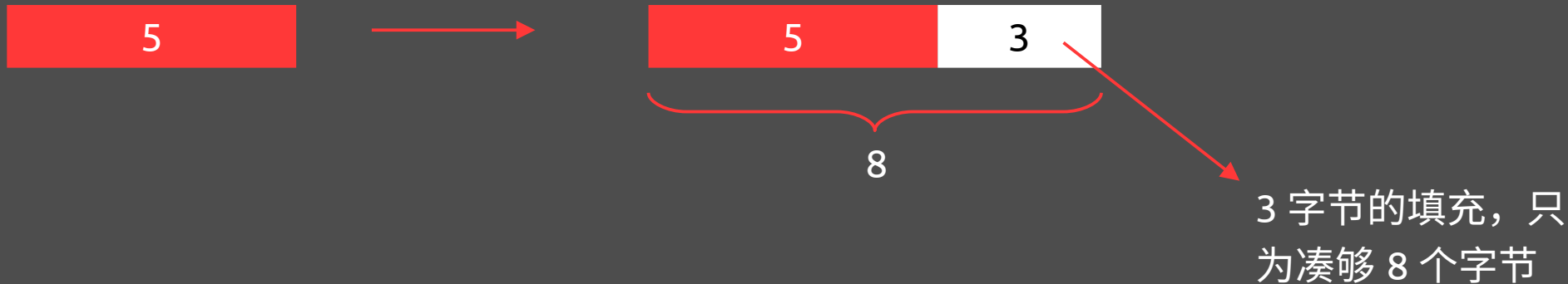
规定，所有分配的空间大小必须为 4 的倍数（地址 4 字节对齐）



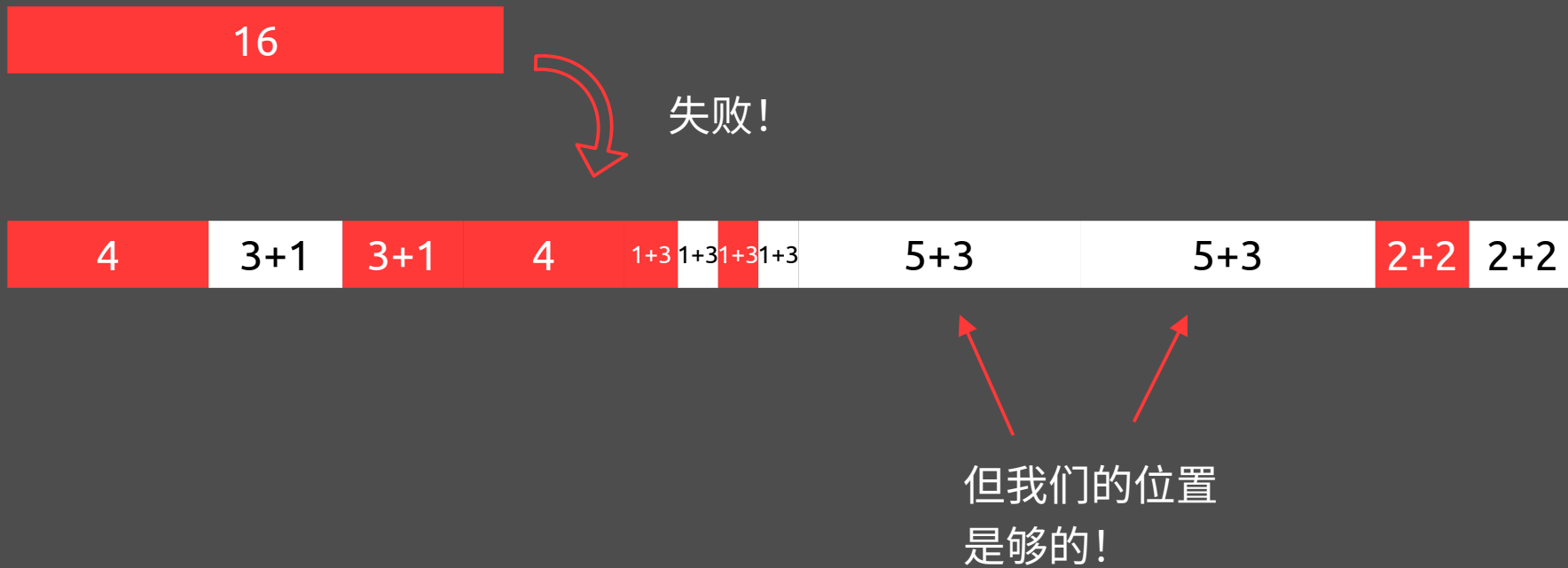
但这并不能解决所有外碎片问题



规定，所有分配的空间大小必须为 4 的倍数（地址 4 字节对齐）



内碎片（Internal Fragmentation）



虚碎片 (False Fragmentation)



malloc 之设计需求

如何高效的表示当前内存布局？

如何快速判断是否有空位？如果有，在哪里？

如何避免碎片的产生？

最大化时间效率

最大化空间效率

But...

我们无法完全避免碎片，和同时满足所有需求

最大化空间效率



移除填充，复杂的外碎片避免算法



降低时间效率

最大化时间效率



填充与一些额外的元数据段，不过分考虑外碎片问题



降低空间效率

Trade Off !

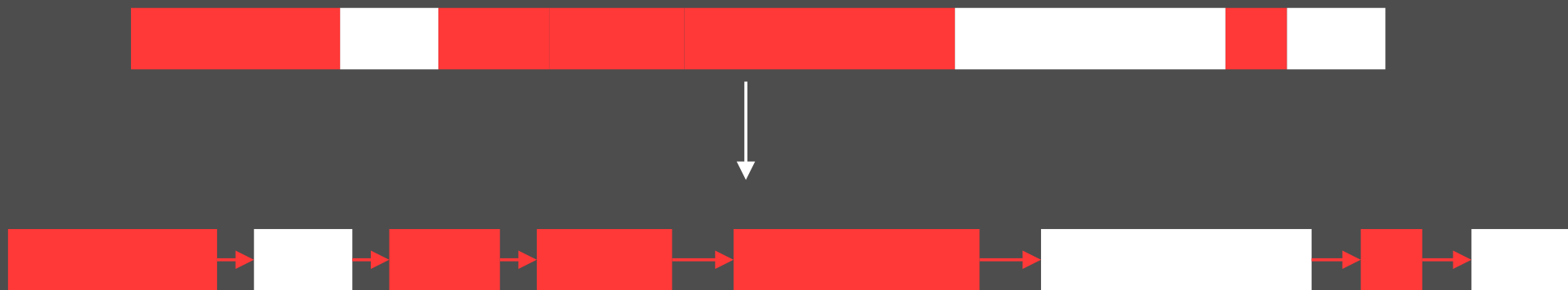


简单易懂——教科书式的算法

较为合理的空间 - 时间复杂度折中



Implicit Free List 无非就是链表

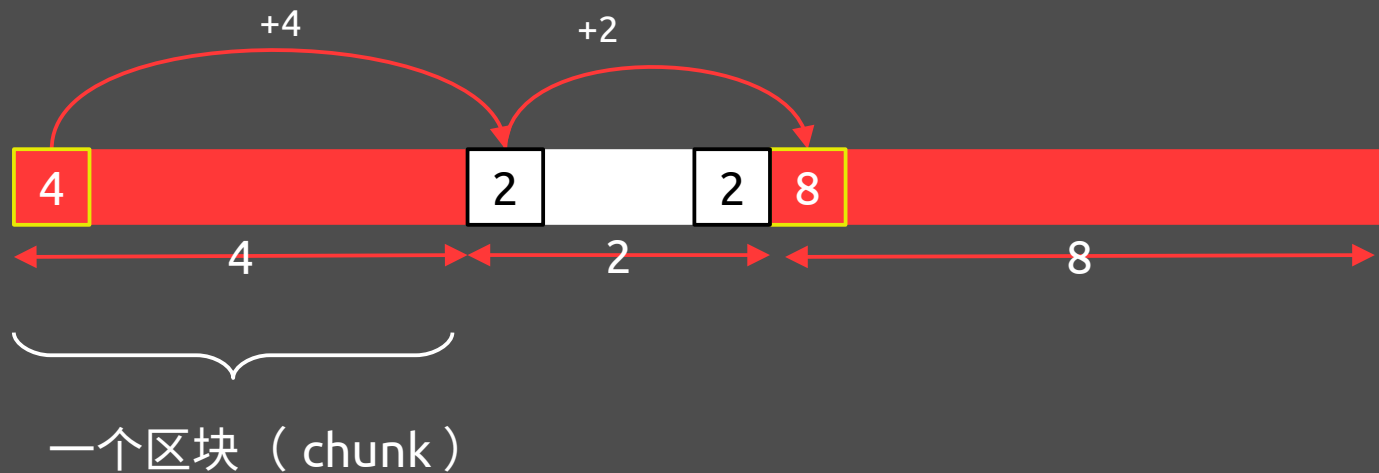


但是，我们如何存放这个链表？

直接就地存放就好了！

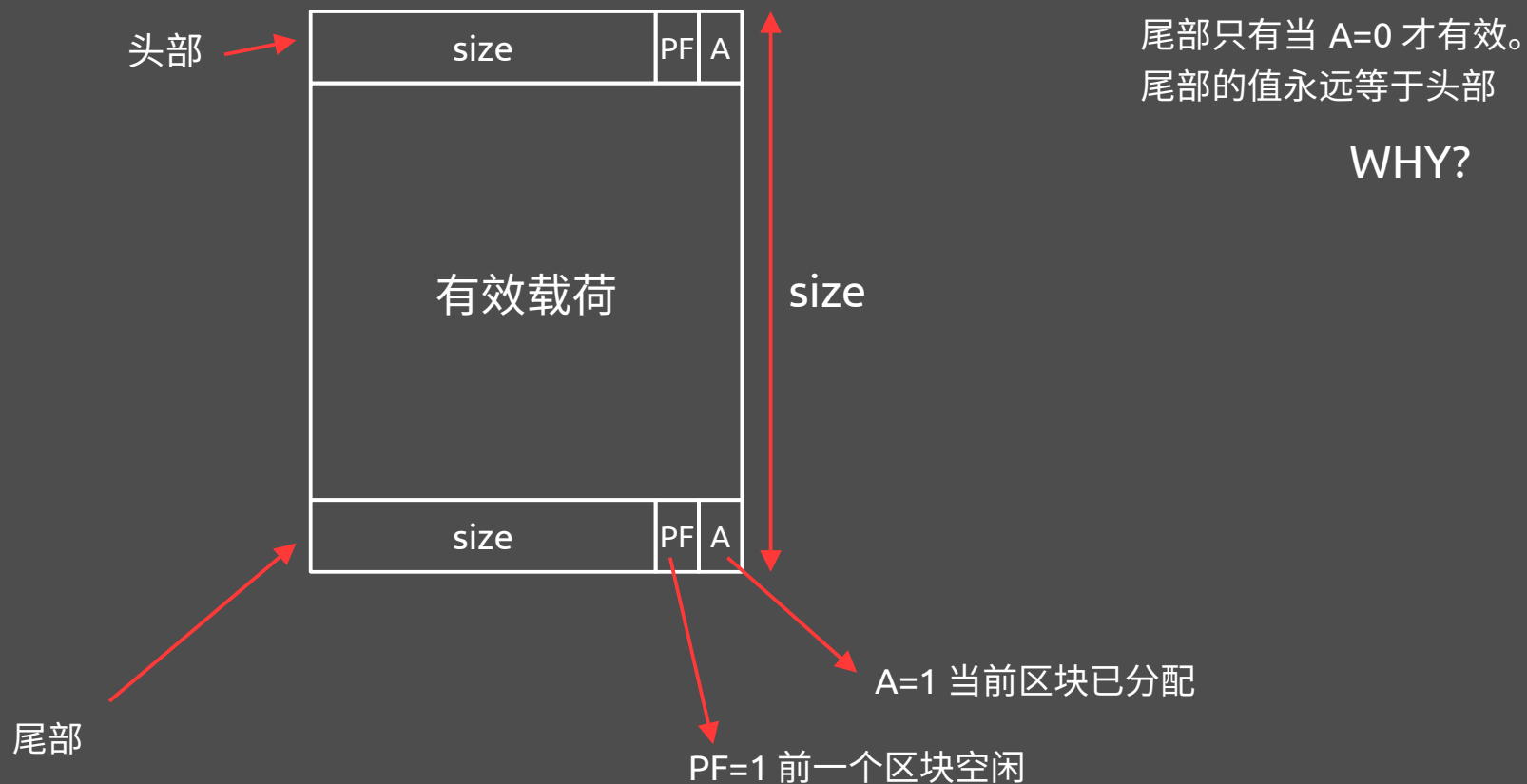


边界标签法（Boundary Tag）——在头尾加上标签，写上一些必要元数据



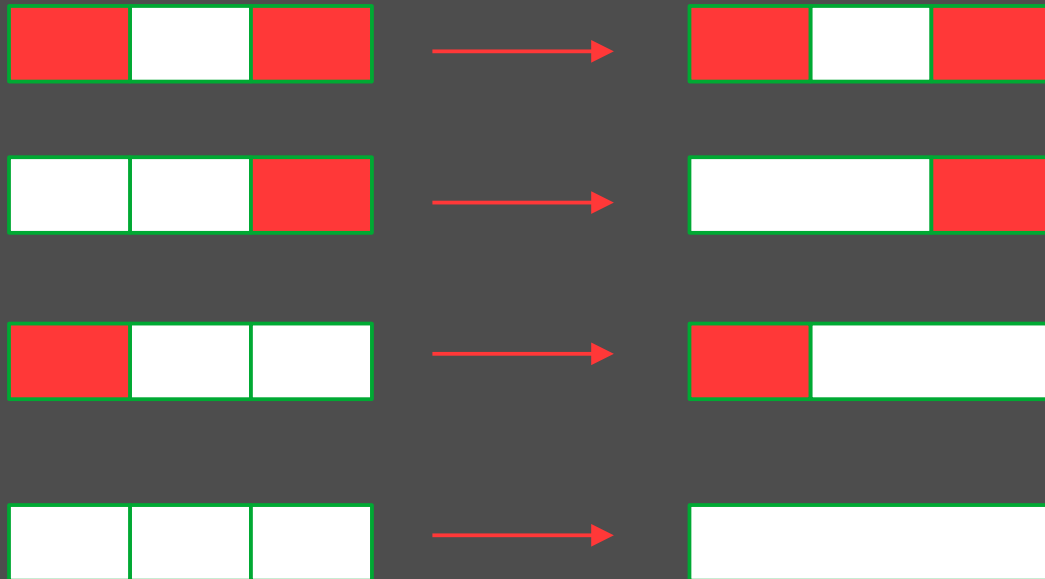


Implicit Free List - 边界标签法





合并左右相邻的任何空闲区块

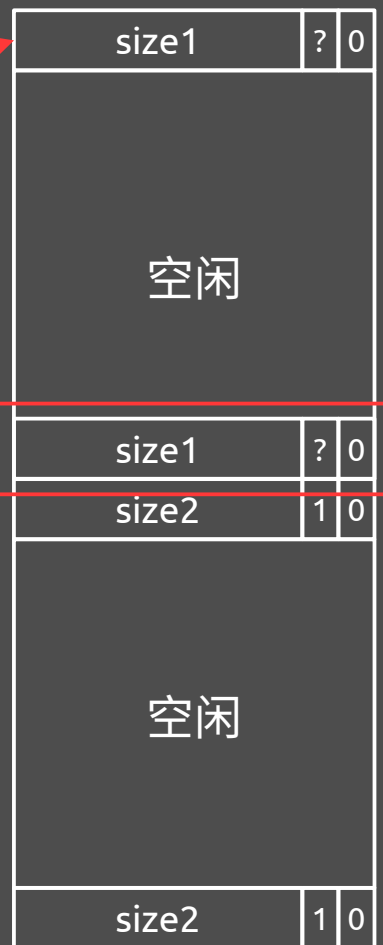


Implicit Free List - 消除虚碎片



不需要合并

尾部用来帮我们找到头





堆两头的区块进行合并操作需要我们做额外的判断（如何避免？）



前序后序标签（Prologue & Epilogue Tag）

前序： 区块大小为一个 tag ， 载荷大小为 0 ， 永远位于堆空间的最开头

后序： 区块大小为 0 ， 载荷大小为 0 ， 永远位于堆空间的最末端



空间不够怎么办?

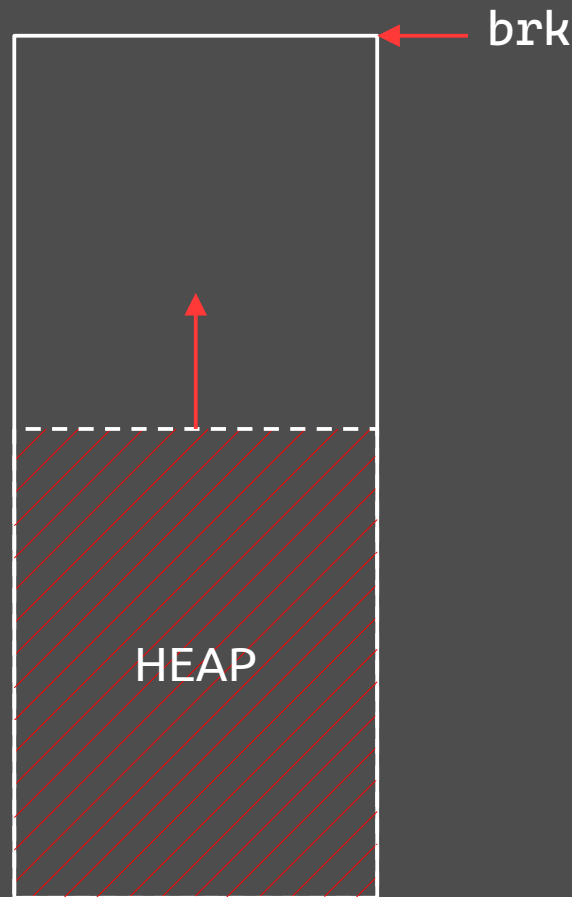
Unix 的解决方案

`int sbrk(void*)`

设置上限至新的地址 ^[2]

`void* brk(size_t)`

将当前上限往上拓展 x 个字节 ^[2]





开始手搓 malloc !

dmm_init 初始化堆空间

coalesce 合并空间，消除虚碎片

lxmalloc

place_chunk 切分空间，分配区块

lxfree

lxsbrk

} 设置堆的新上限，如有
必要，则分配新的虚拟
页

lxbrk

对齐： 4 字节

grow_heap 拓展堆（增加上限 + 初始化）

Code Time



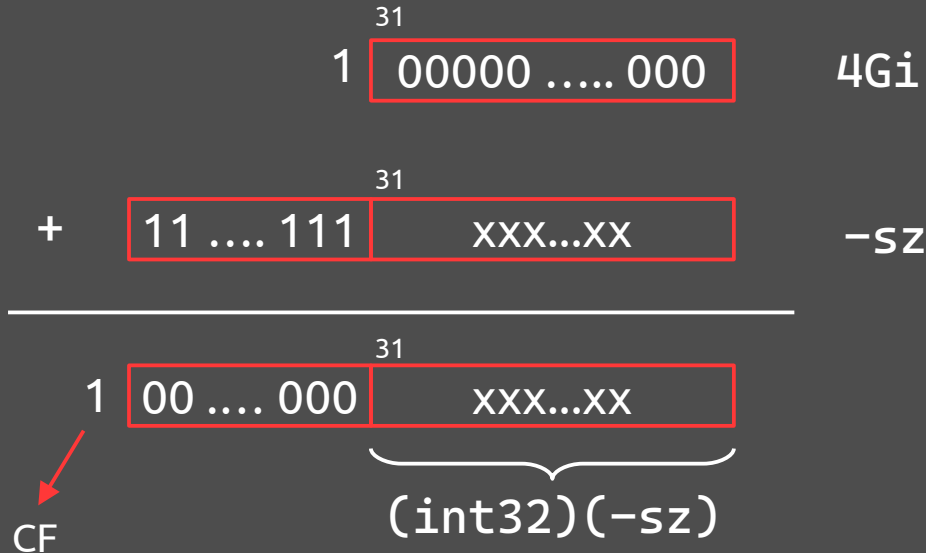
```
assert_msg(((uintptr_t)ptr < (uintptr_t)(-sz)) && !((uintptr_t)ptr & 0x3),
           "free(): invalid pointer");
```

P.s. 从 glibc malloc/malloc.c:4437 那儿偷来的 xD

初步防止用户恶意修改 header 的 size，或者修改指针

$$\text{ptr} + \text{sz} < 4\text{Gi}$$

$$\Rightarrow \text{ptr} < 4\text{Gi} + -\text{sz}$$

$$\Rightarrow \text{ptr} < (\text{int32})(-\text{sz})$$




一些安全性的忧虑

所有的区块都分配在一个连续的空间里 → 高可预测性的结构

元数据与用户操作区域混在一起 → 违反封装原则

数据损毁，奇怪的 bug，更广的攻击平面！

Not so good...

目前的（ e.g., OpenBSD, uClibc ）解决方案 ^{[4][5]}：使用
mmap 来分配区域 → 非连续的区块儿 → 减少越界
访问，数据损坏的情况



1. Computer System: A Programmer's Perspective (Thrid Edition) Section 9.6
2. Linux man page for malloc, realloc, calloc, free, sbrk, brk
3. GNU C Standard Library: `malloc/malloc.c`
4. uClibc-ng: `libc/stdlib/malloc-simple/alloc.c`
5. A new malloc(3) for OpenBSD