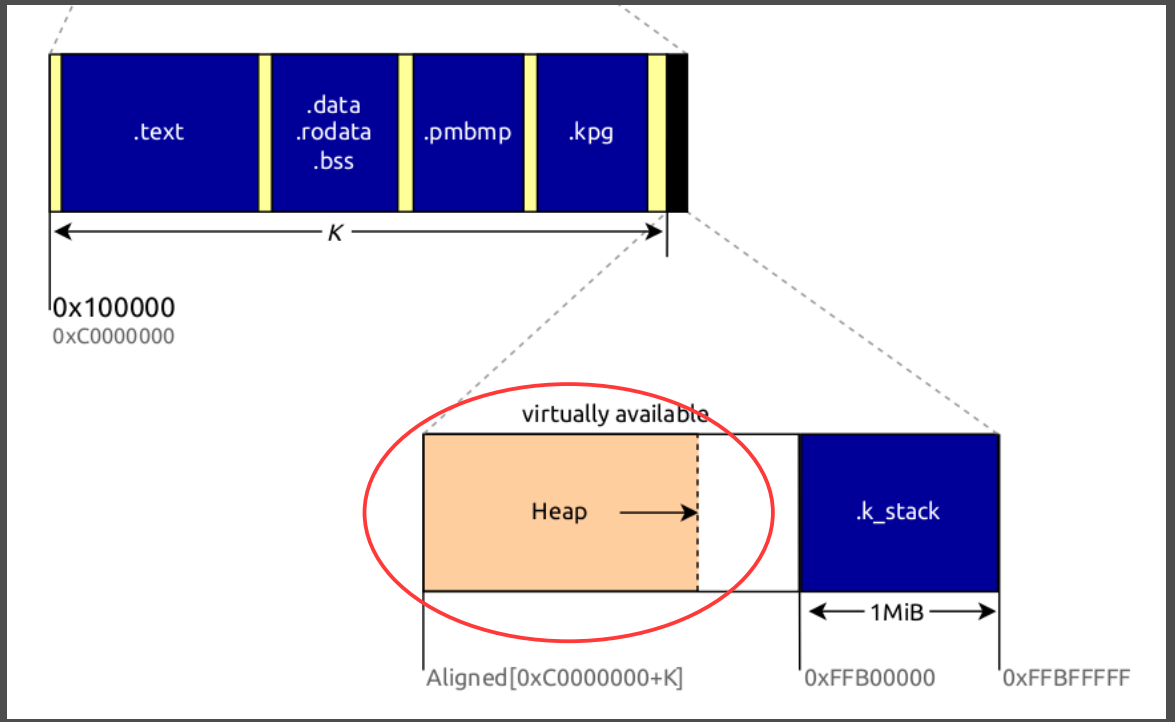

LunaixOS
从零开始
DEVELOPING YOUR OWN
自制操作系统
OPERATING SYSTEM FROM SCRATCH
O Brave New Kernel
虚拟内存：实现 malloc
OPERATING SYSTEM DEVELOPMENT
TUTORIAL SERIES
EP 7-3

堆空间——程序的仓库

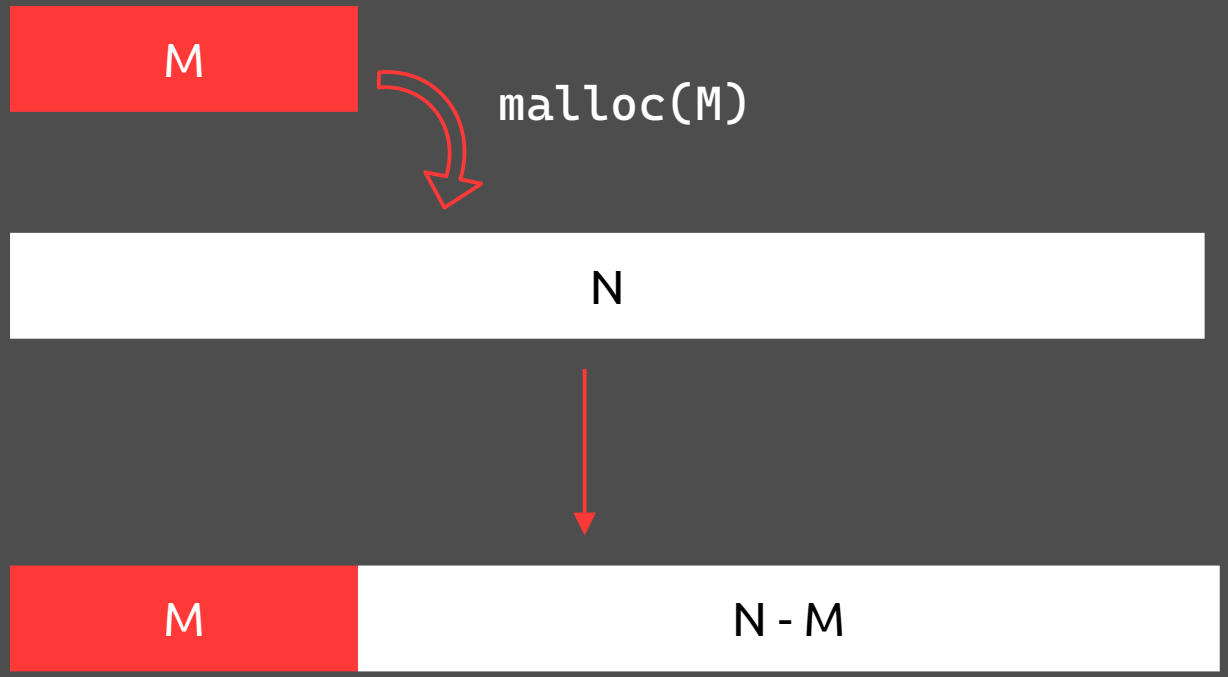


计划赶不上变化 —— 动态地，按需创建的空间

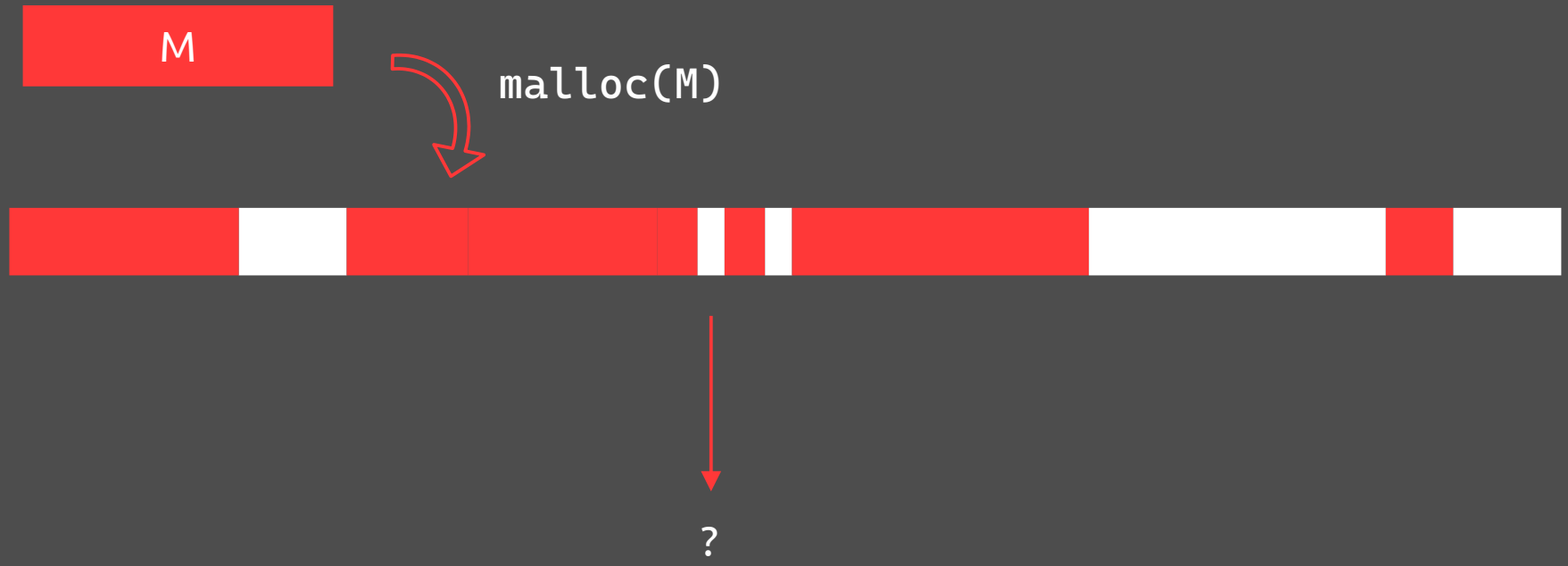
- `malloc()`
- `calloc()`
- `realloc()`
- `free()`

堆管理 [2]

malloc 之原理



malloc 之原理



怎么办?

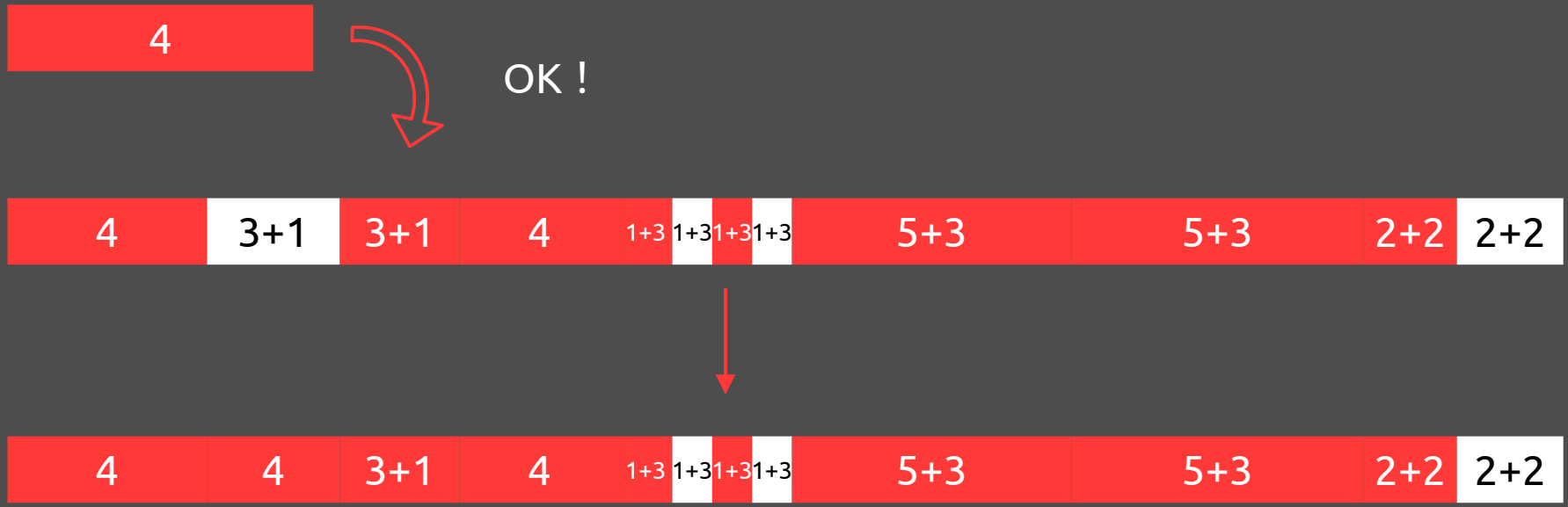
malloc 之原理

假设我们的 malloc 足够聪明，知道在哪里安放

外碎片（External Fragmentation）

malloc 之原理 - 消除碎片的一个尝试

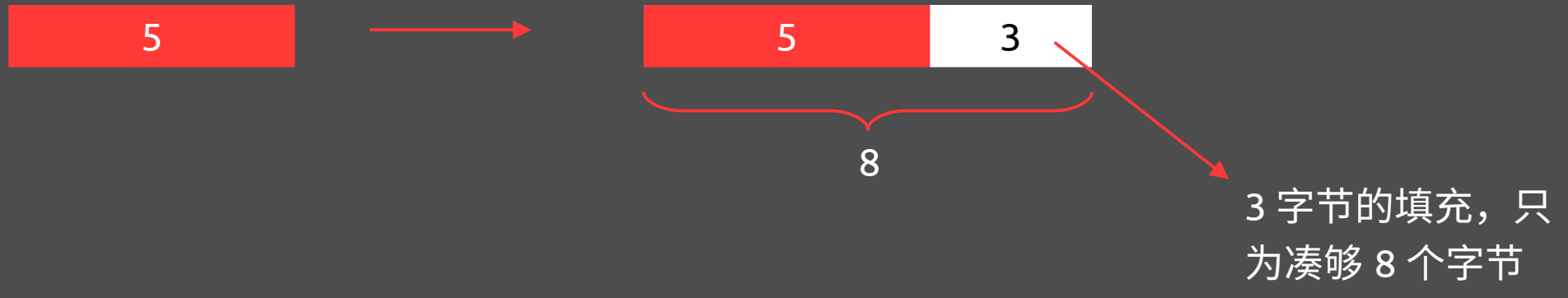
规定，所有分配的空间大小必须为 4 的倍数（地址 4 字节对齐）



但这并不能解决所有外碎片问题

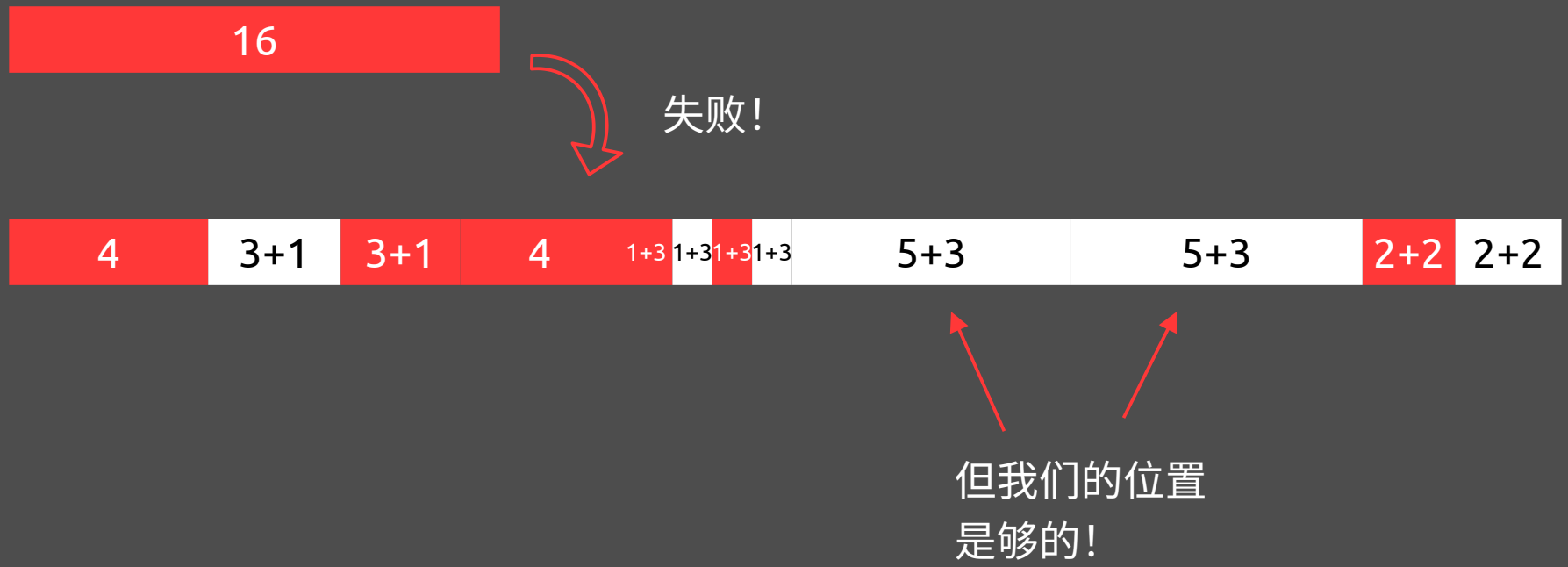
malloc 之原理 - 新的问题

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）

malloc 之原理 - 新的新的问题



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

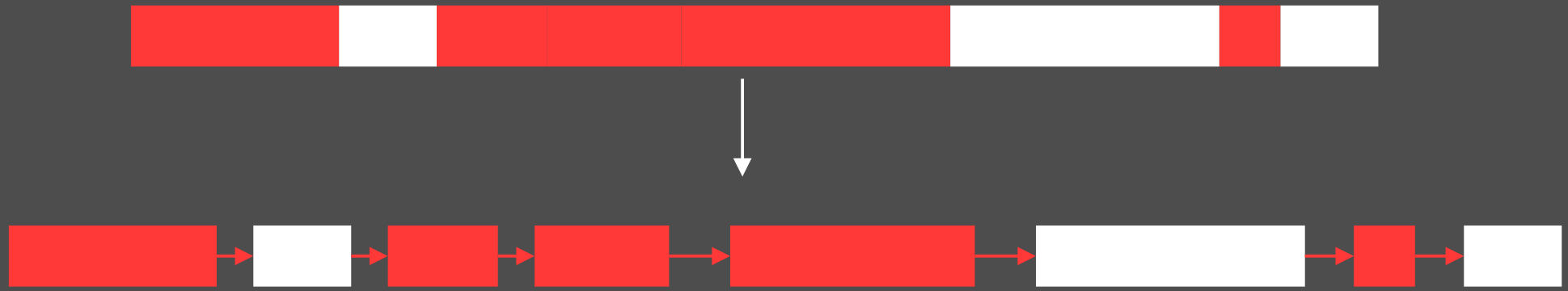
Trade Off !

LunaixOS 的实现—— Implicit Free List[1]

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

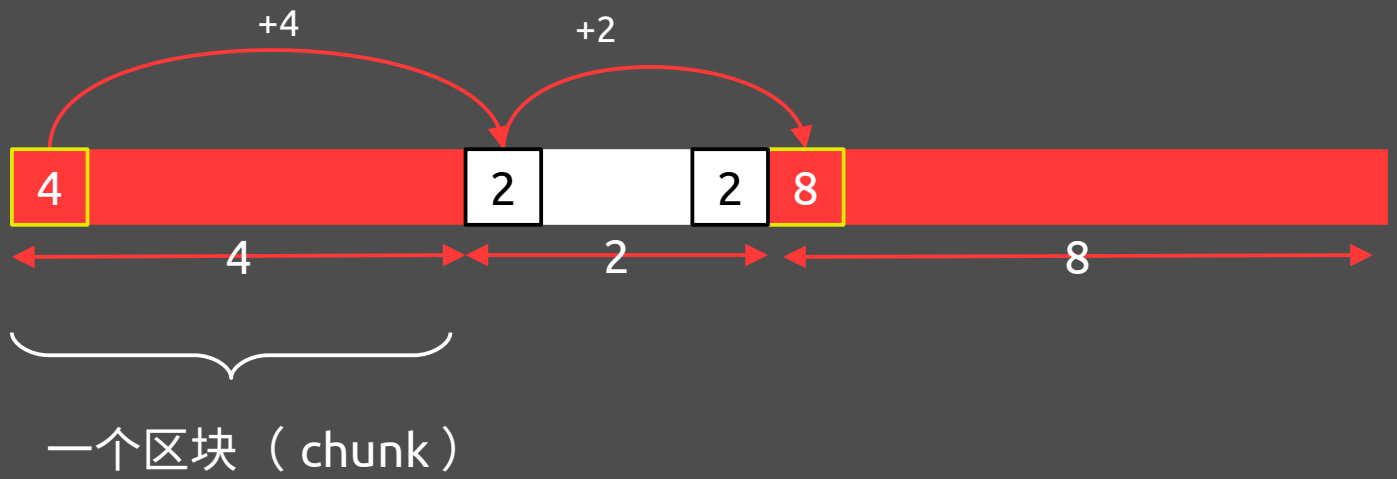


但是，我们如何存放这个链表？

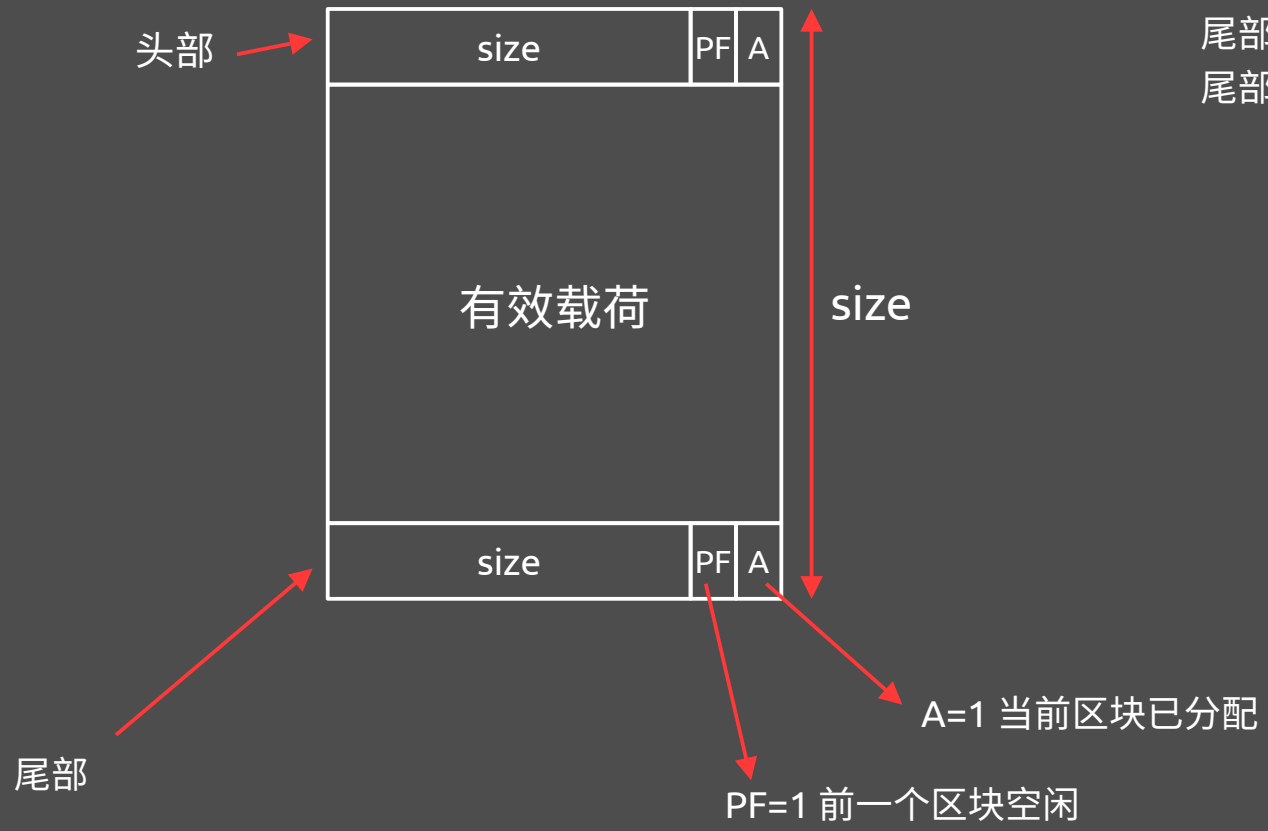
直接就地存放就好了！

Implicit Free List - 边界标签法

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

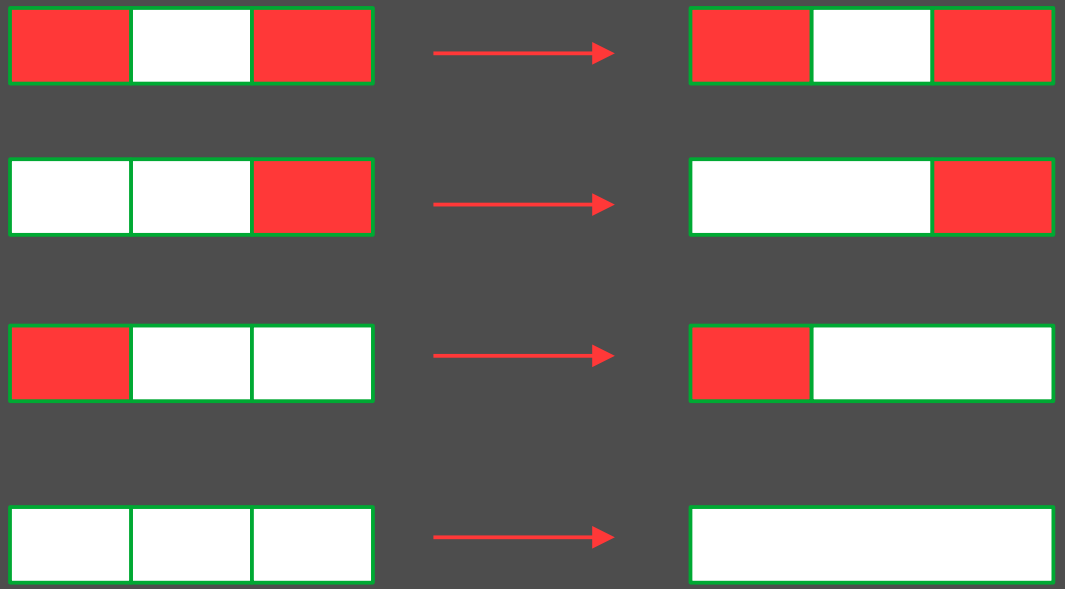


尾部只有当 A=0 才有效。
尾部的值永远等于头部

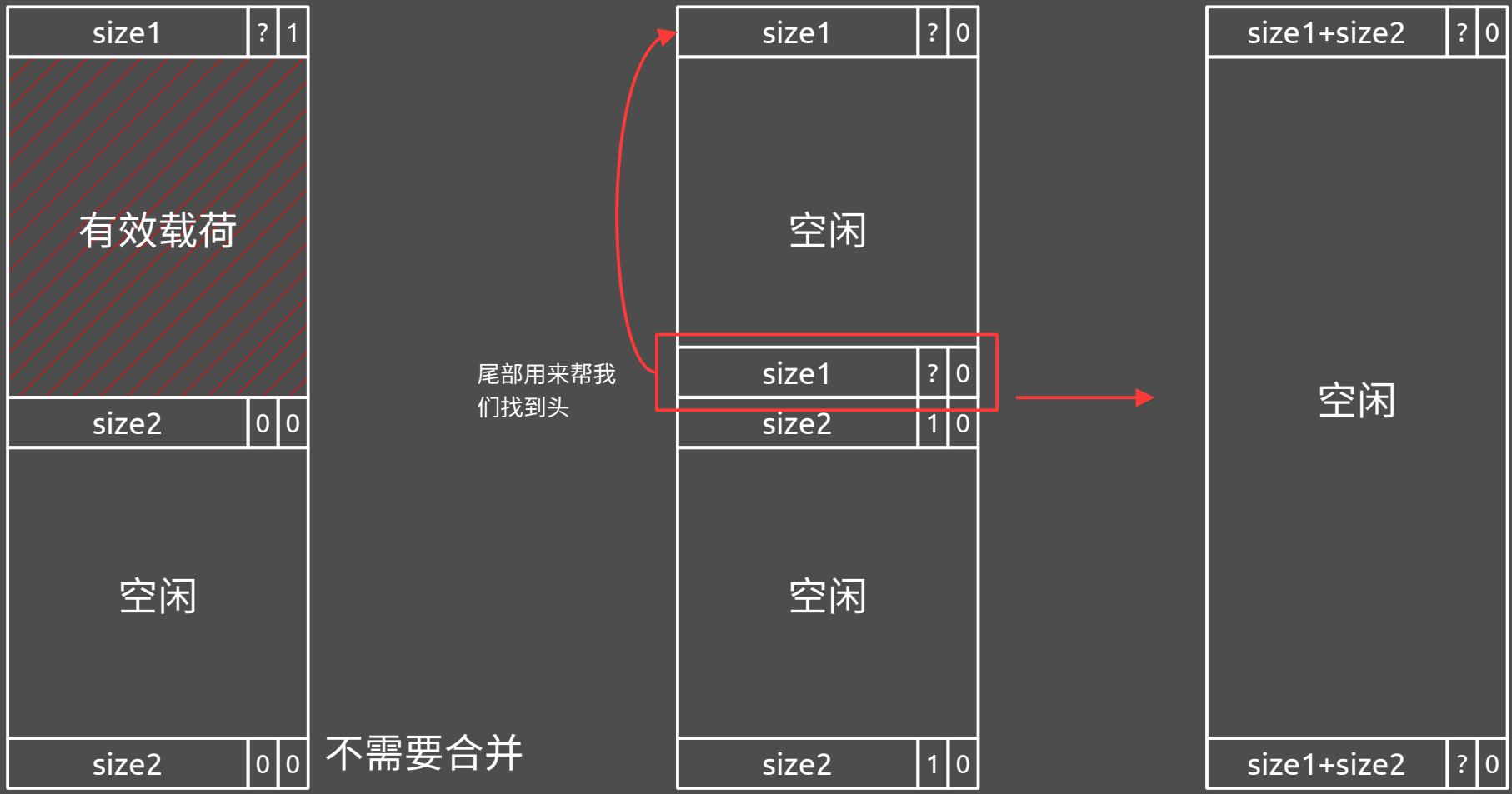
WHY?

Implicit Free List - 消除虚碎片

合并左右相邻的任何空闲区块



Implicit Free List - 消除虚碎片



malloc 具体之实现

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

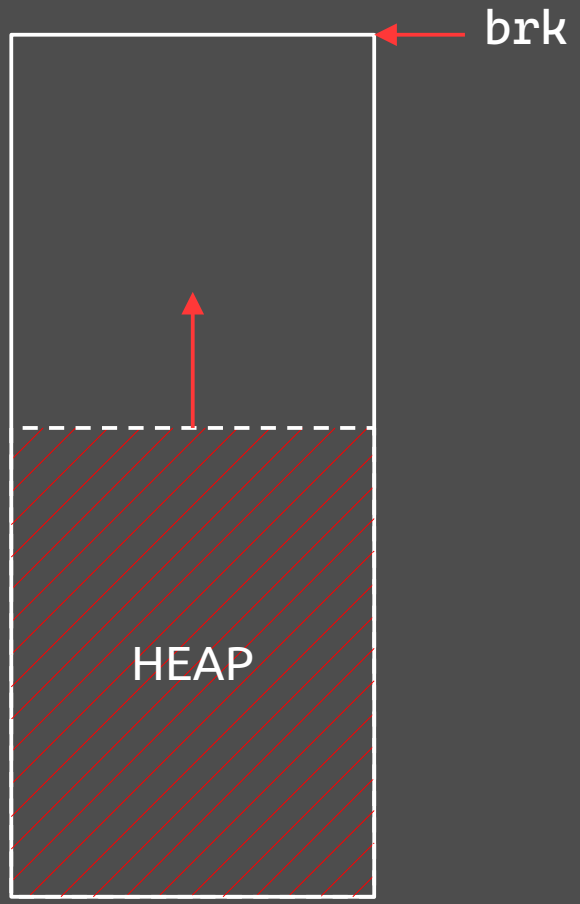
后序：区块大小为 0 ， 载荷大小为 0 ， 永远位于堆空间的最末端

malloc 具体之实现

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
lxmalloc	
lxfree	
lxsbrk	设置堆的新上限，如有必要，则分配新的虚拟页
lxbrk	
grow_heap	拓展堆（增加上限 + 初始化）

coalesce	合并空间，消除虚碎片
place_chunk	切分空间，分配区块

对齐： 4 字节

Code Time

一个有趣的指针有效性校验

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

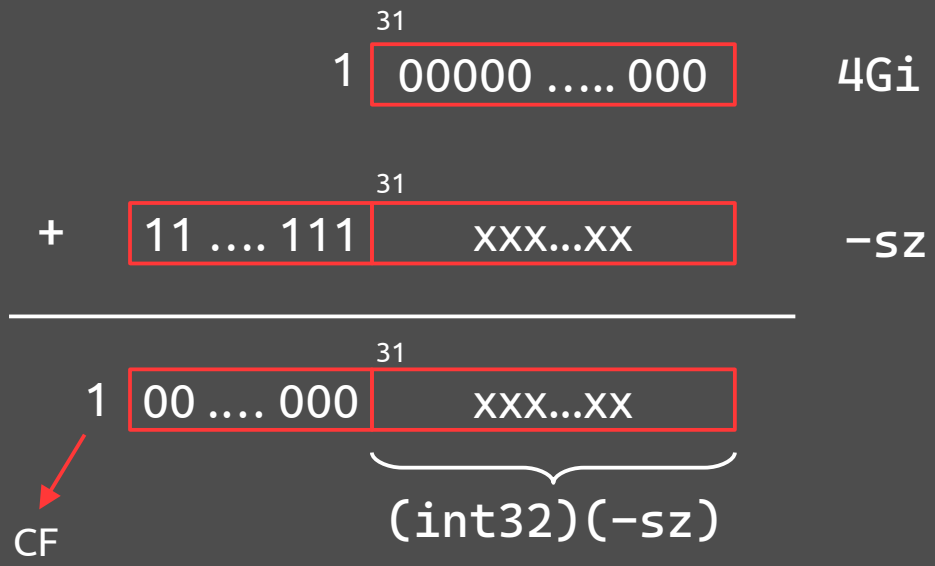
P.s. 从 glibc malloc/malloc.c:4437 那儿偷来的 xD

初步防止用户恶意修改 header 的 size，或者修改指针

```
ptr + sz < 4Gi

⇒ ptr < 4Gi + -sz

⇒ ptr < (int32)(-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

参考资料

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