

CVTE Dream·Future  
视源股份

2020  
TECH DAY

# 梦想引领未来

2020.11.27-11.28

# 内存问题探微

---

张亚

2020/11/27

# Contents

---

- 为什么要讲这个主题
- Linux 内存知识的底层原理
- 开发相关的内存问题说明

# 为什么要讲这个主题

内存问题出现的非常频繁

OOM

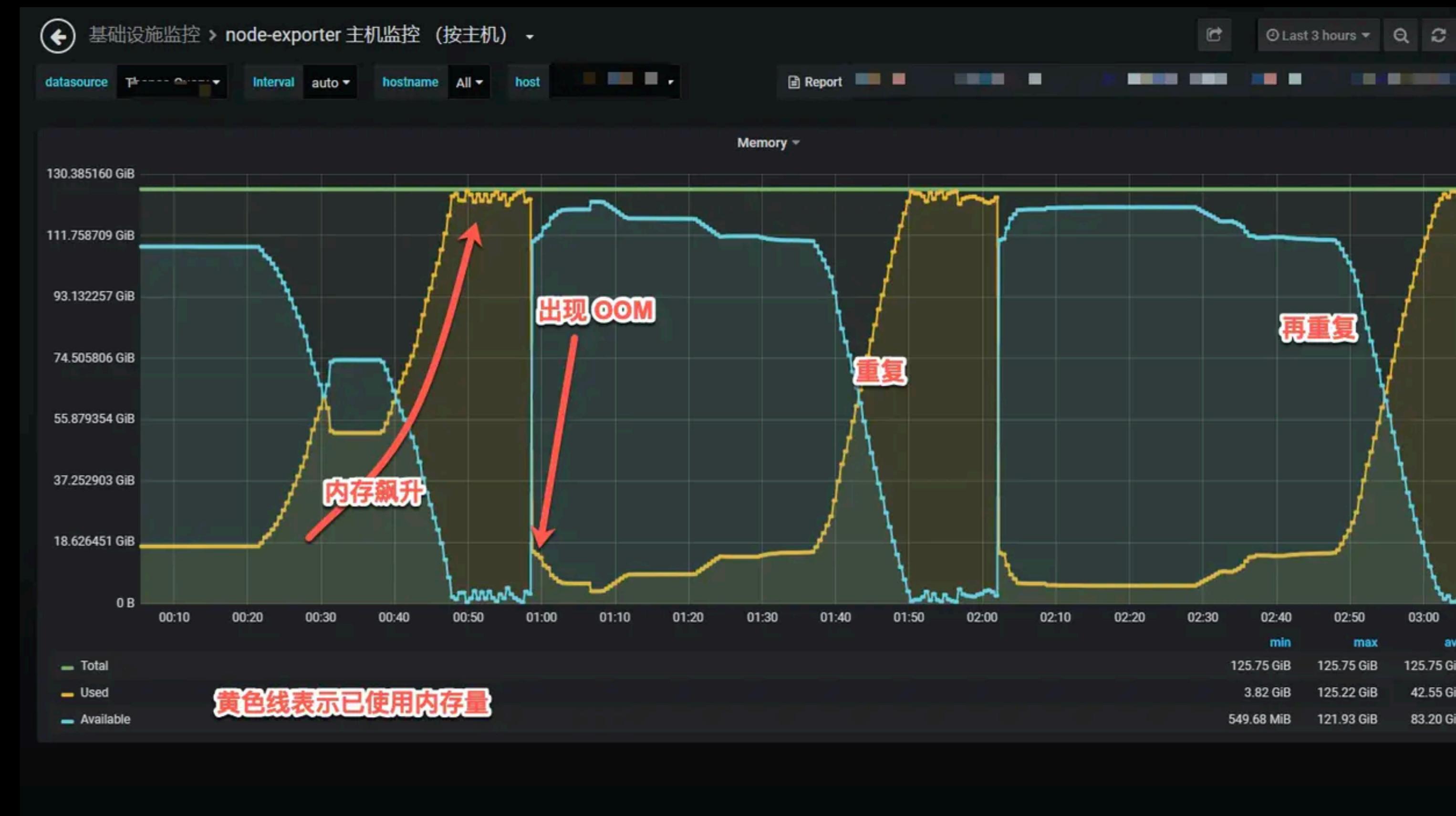
内存使用超过配额

内存泄露

堆外内存分析困难

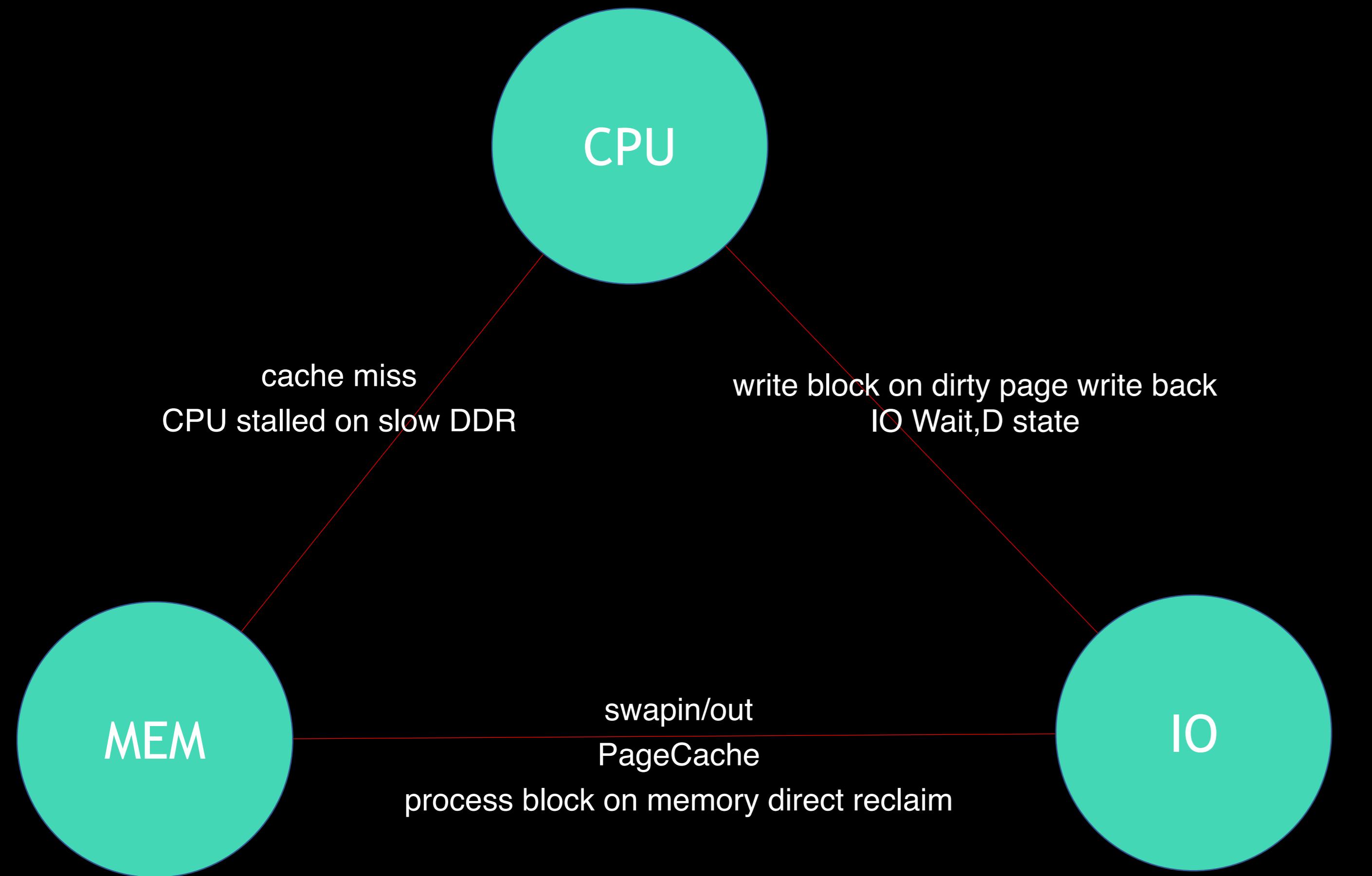
# 为什么要讲这个主题

内存问题出现的非常频繁



# 为什么要讲这个主题

# 内存关联的知识非常庞大



# 为什么要讲这个主题

## 我们的 8 点《早读会》

- 每天早上 8 点到 9 点分享一个小时
  - 花了将近三周的时间讲内存相关的知识
- (18 小时以上)



# 为什么要讲这个主题

可以让我们更深入理解一些问题

- 为什么 golang 原生支持函数多返回值
- golang 逃逸分析是怎么做的
- Java 堆外内存泄露如何分析
- Java 的 RandomAccessFile 原理是什么
- C++ 智能指针是如何实现的

人类的三个终极问题

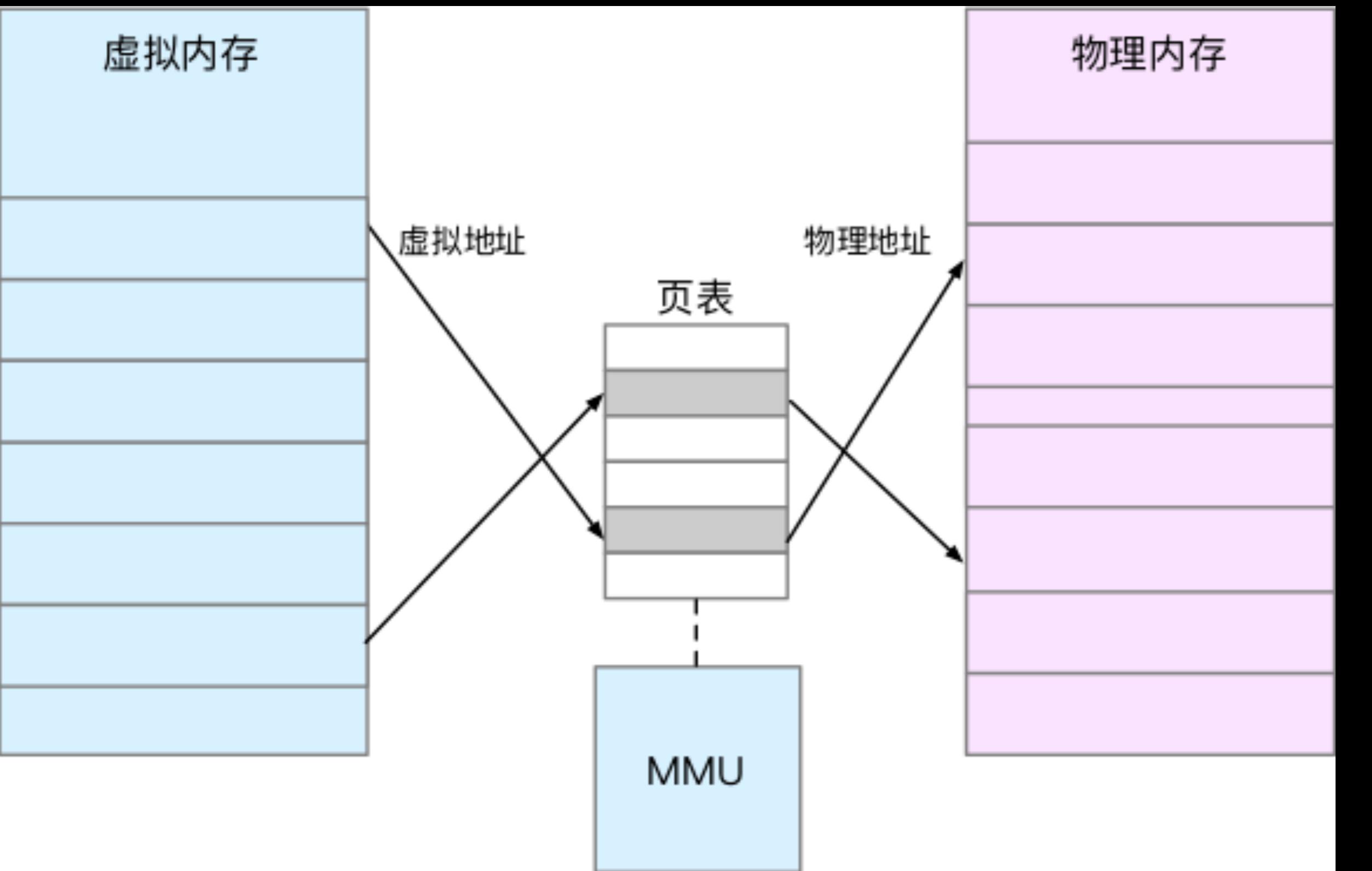
你是谁？

你从哪里来？

你要到哪里去？

第二部分：Linux 内存管理的原理

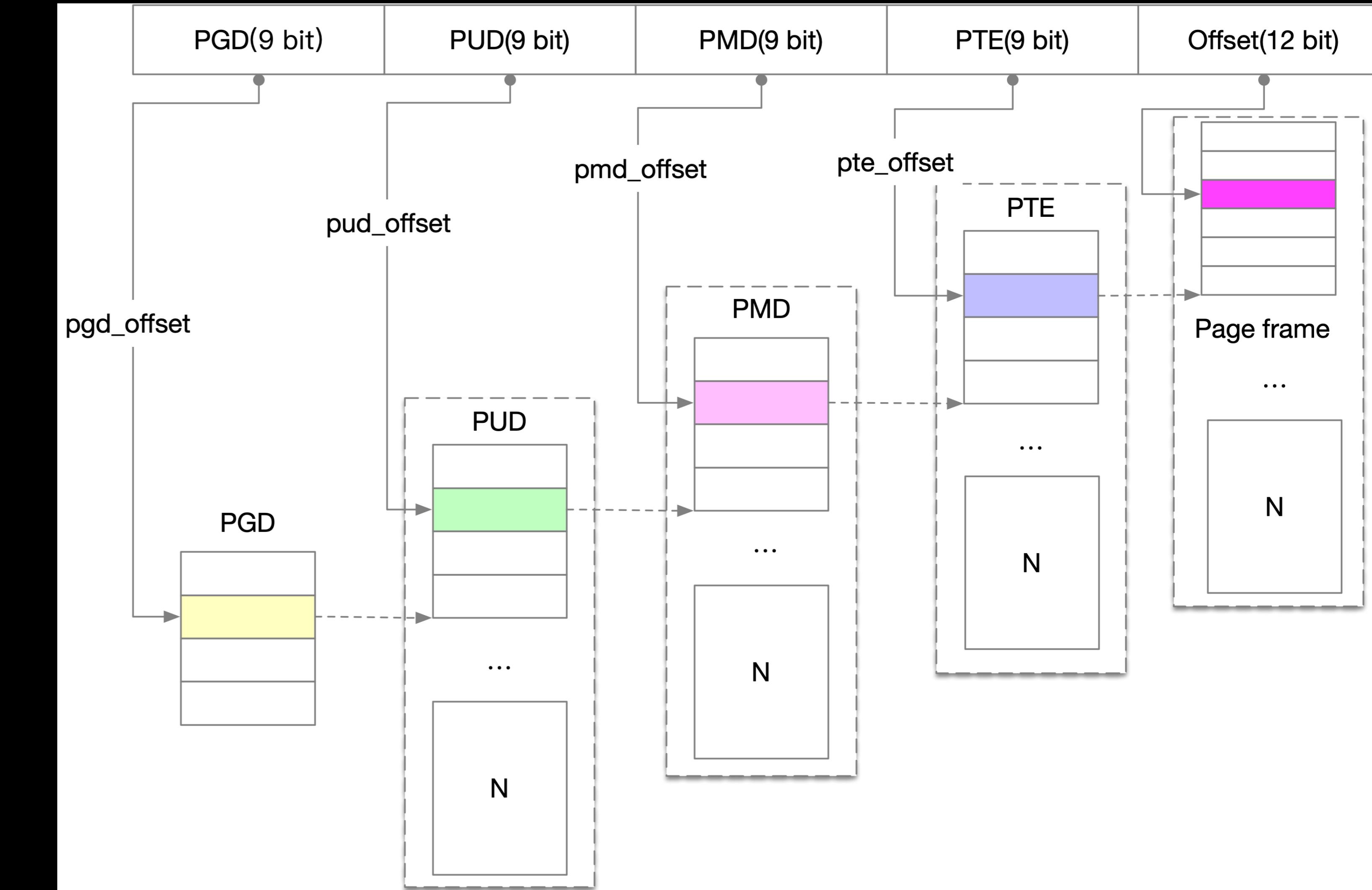
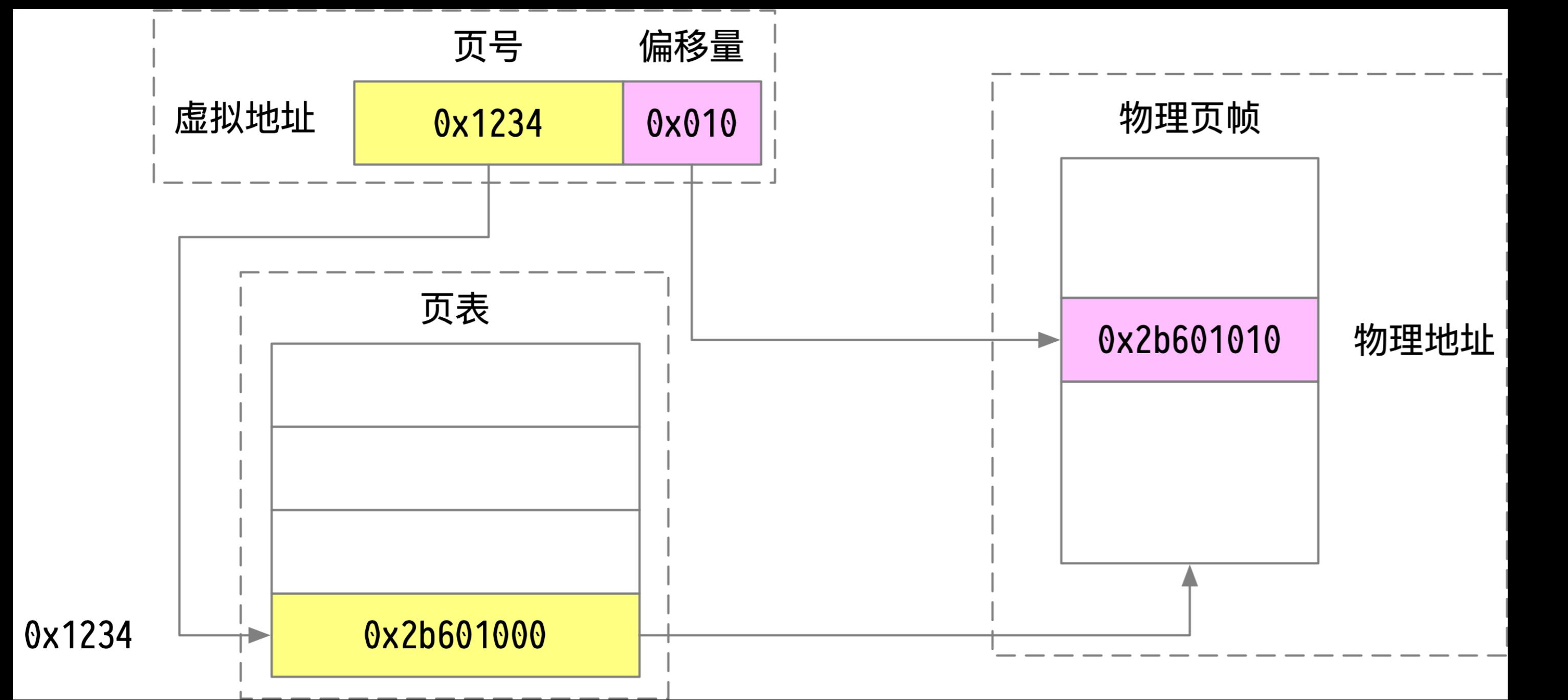
# 虚拟内存与物理内存



计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决

- 每个进程拥有独立的、连续的、统一的虚拟地址空间（好一个错觉）
- 应用程序看到的都是虚拟内存，通过 MMU 进行虚拟内存到物理内存的映射

# 虚拟内存与物理内存



# 虚拟内存与物理内存

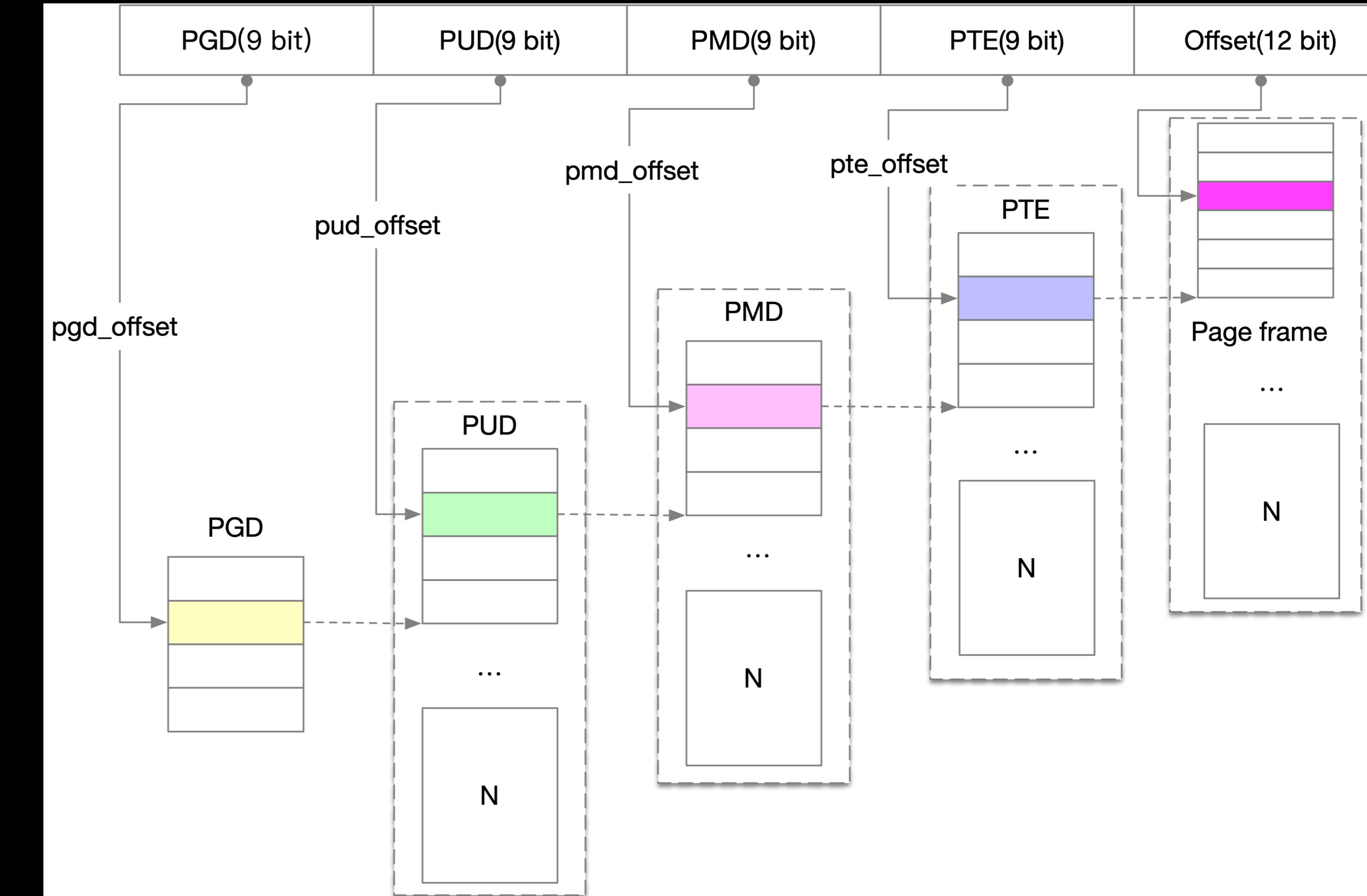
● ● ●

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    char *p = NULL;
    p = malloc(1024 * 1024);
    *p = 0;
    printf("ptr: %p, pid: %d\n", p, getpid());
    getchar();
    return 0;
}
```

0x1234

ptr: 0x7ffff7eec010, pid: 2621

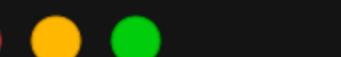
物理地址



# 虚拟内存与物理内存

```

● ● ●
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    char *p = NULL;
    p = malloc(1024 * 1024);
    *p = 0;
    printf("ptr: %p, pid: %d\n", p, getpid());
    getchar();
    return 0;
}
ptr: 0x7ffff7eec010, pid: 2621
0x1234
  
```



```

#include <linux/module.h>
...
int my_module_init(void) {
    unsigned long pa = 0;
    pgd_t *pgd = NULL; pud_t *pud = NULL;
    pmd_t *pmd = NULL; pte_t *pte = NULL;

    struct pid *p = find_vpid(pid);
    struct task_struct *task_struct = pid_task(p, PIDTYPE_PID);

    pgd = pgd_offset(task_struct->mm, va);
    pud = pud_offset(pgd, va);
    pmd = pmd_offset(pud, va);
    pte = pte_offset_kernel(pmd, va);

    unsigned long page_addr = pte_val(*pte) & PAGE_MASK;
    unsigned long page_addr &= 0xfffffffffffffull;

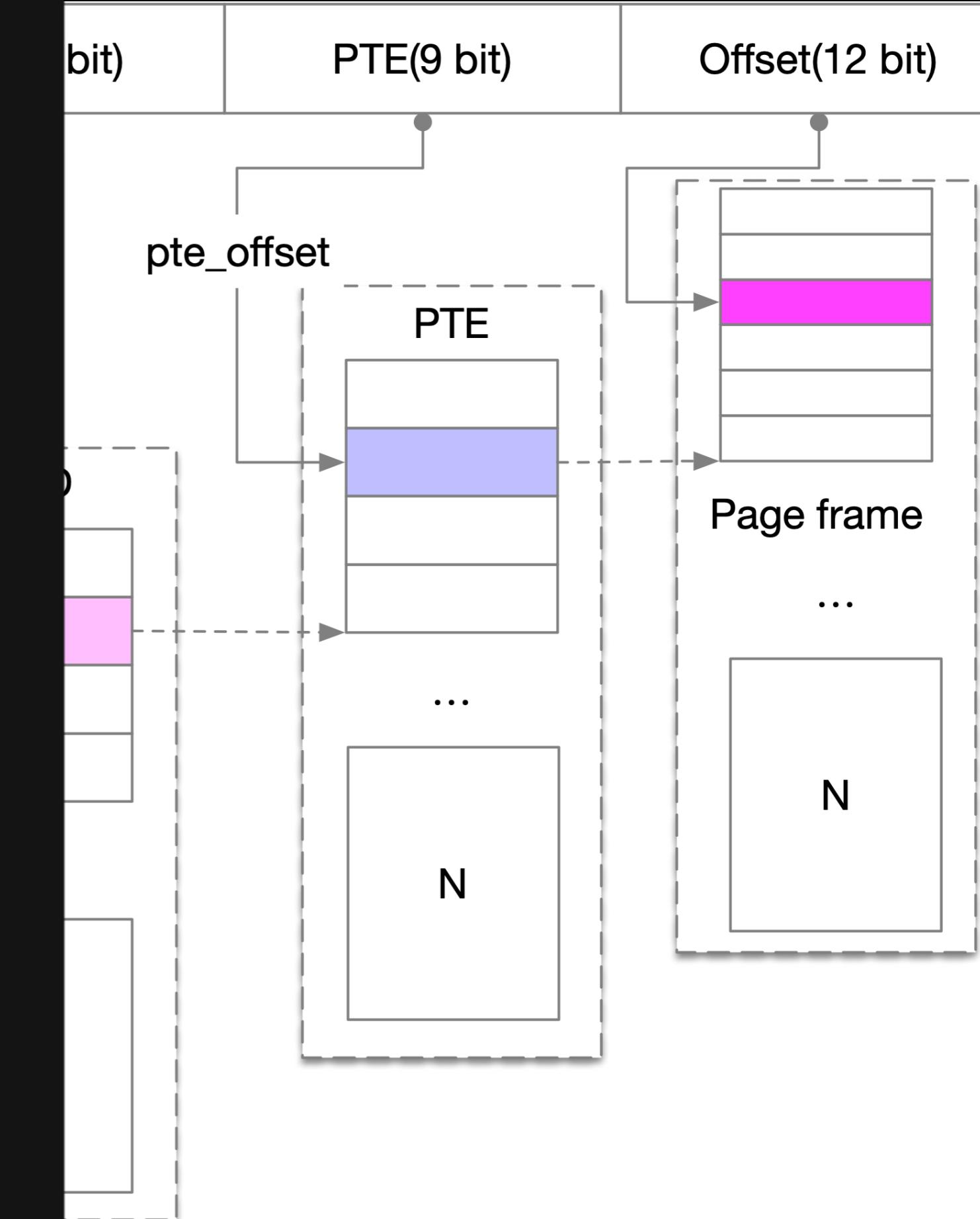
    page_offset = va & ~PAGE_MASK;
    pa = page_addr | page_offset;

    printk("virtual address 0x%lx in RAM Page is 0x%lx\n", va, pa);

    return 0;
}
void my_module_exit(void) {
    printk("module exit!\n");
}

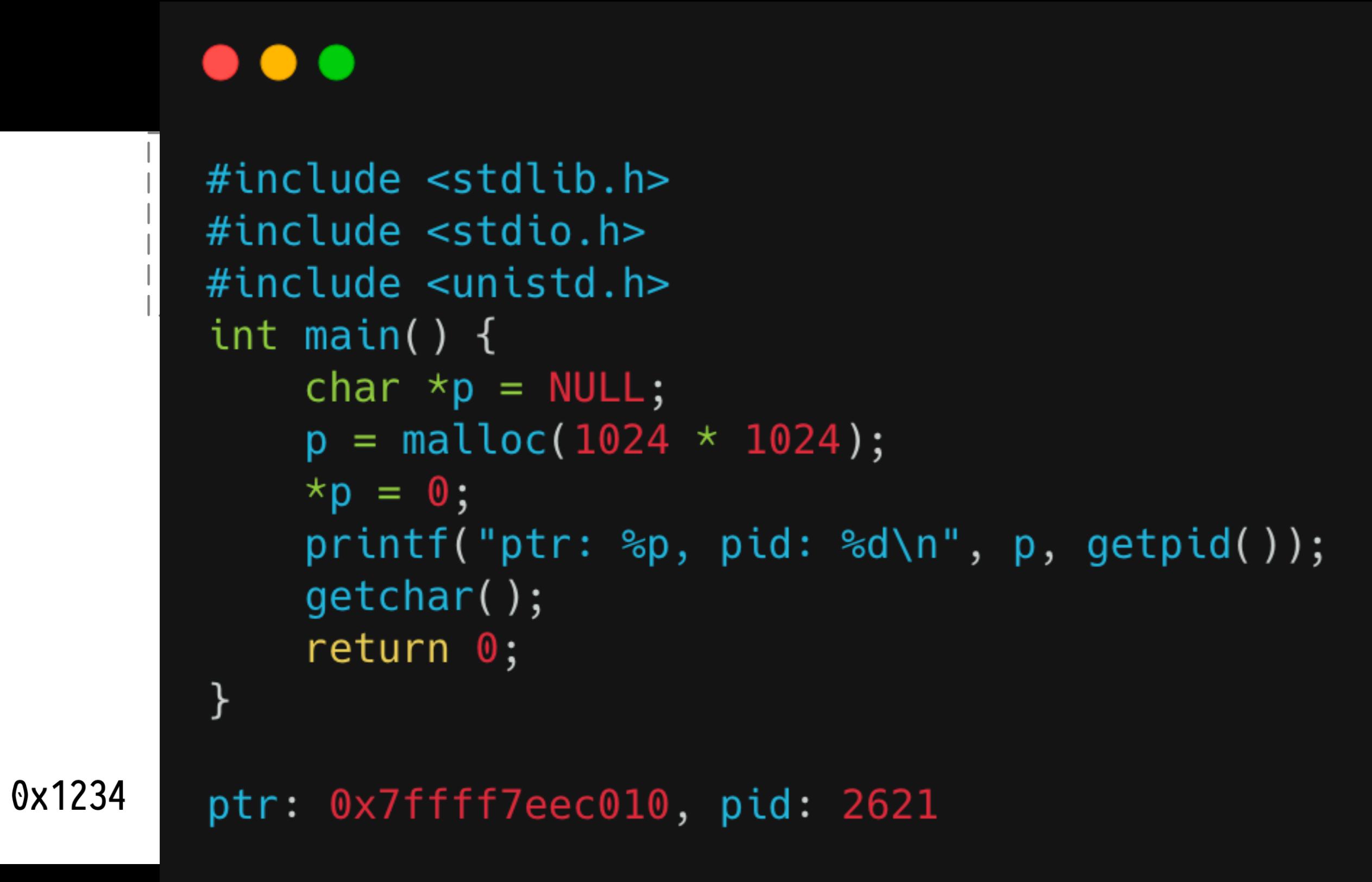
module_init(my_module_init);
module_exit(my_module_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Arthur.Zhang");
MODULE_DESCRIPTION("A simple virtual memory inspect");
  
```





# 虚拟内存与物理内存



0x1234

ptr: 0x7ffff7eec010, pid: 2621

```
#include <linux/module.h>

void my_module_init(void) {
    unsigned long pa = 0;
    pgd_t *pgd = NULL; pud_t *pud = NULL;
    pmd_t *pmd = NULL; pte_t *pte = NULL;

    struct pid *p = find_vpid(pid);
    struct task_struct *task_struct = pid_task(p, PIDTYPE_PID);

    pgd = pgd_offset(task_struct->mm, va);
    pud = pud_offset(pgd, va);
    pmd = pmd_offset(pud, va);
    pte = pte_offset_kernel(pmd, va);

    unsigned long page_addr = pte_val(*pte) & PAGE_MASK;
    unsigned long page_addr &= 0x7ffffffffffffFULL;

    page_offset = va & ~PAGE_MASK;
    pa = page_addr | page_offset;

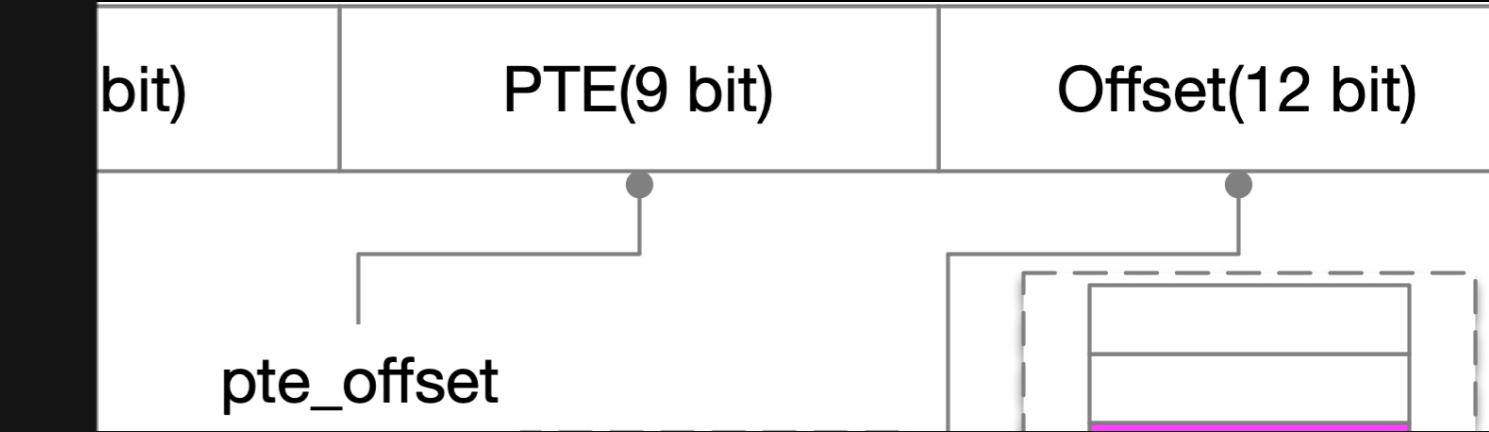
    printk("virtual address 0x%lx in RAM Page is 0x%lx\n", va, pa)

    return 0;
}

void my_module_exit(void) {
    printk("module exit!\n");
}

MODULE_init(my_module_init);
MODULE_exit(my_module_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Arthur.Zhang");
MODULE_DESCRIPTION("A simple virtual memory inspect");
```



?WD ) module

f7eec010 in RAM Page is 0x2358a4010



# 进程的内存布局

**pmap -x `pidof java`**

Stack (向下生长)	Address	Kbytes	RSS	Dirty	Mode	Mapping
	000000000400000	4	4	0	r-x--	java
	...					
	000000000763000	1632	1436	1436	rw---	[ anon ]
	00000000080000000	2122192	2121988	2121988	rw---	[ anon ]
	0000000101874000	1023536	0	0	-----	[ anon ]
	00007fd1040c7000	1016	28	28	rw---	[ anon ]
	...					
	00007fd1279ff000	1016	28	28	rw---	[ anon ]
	....					
Memory Mapping Segment	00007fd58c14a000	64216	0	0	-----	[ anon ]
	00007fd590000000	1336	1336	1336	rw---	[ anon ]
	00007fd59014e000	64200	0	0	-----	[ anon ]
	00007fd594000000	1384	1384	1384	rw---	[ anon ]
	...					
Heap (向上生长)	00007fd6f9b3b000	516	516	0	r--s-	easicare-server-1.0.3.jar
	...					
	00007fd6f9bdb000	48	48	0	r--s-	jedis-2.9.0.jar
	...					
Data Segment (R+W)	00007fd7e57dc000	196	196	196	rw---	libjvm.so
	00007fd7e6032000	92	72	0	r-x--	libpthread-2.17.so
	...					
Code Segment (R+X)	total kB	31802228	3161028	3135476		

# Linux 内存管理的三个层面

用户管理层

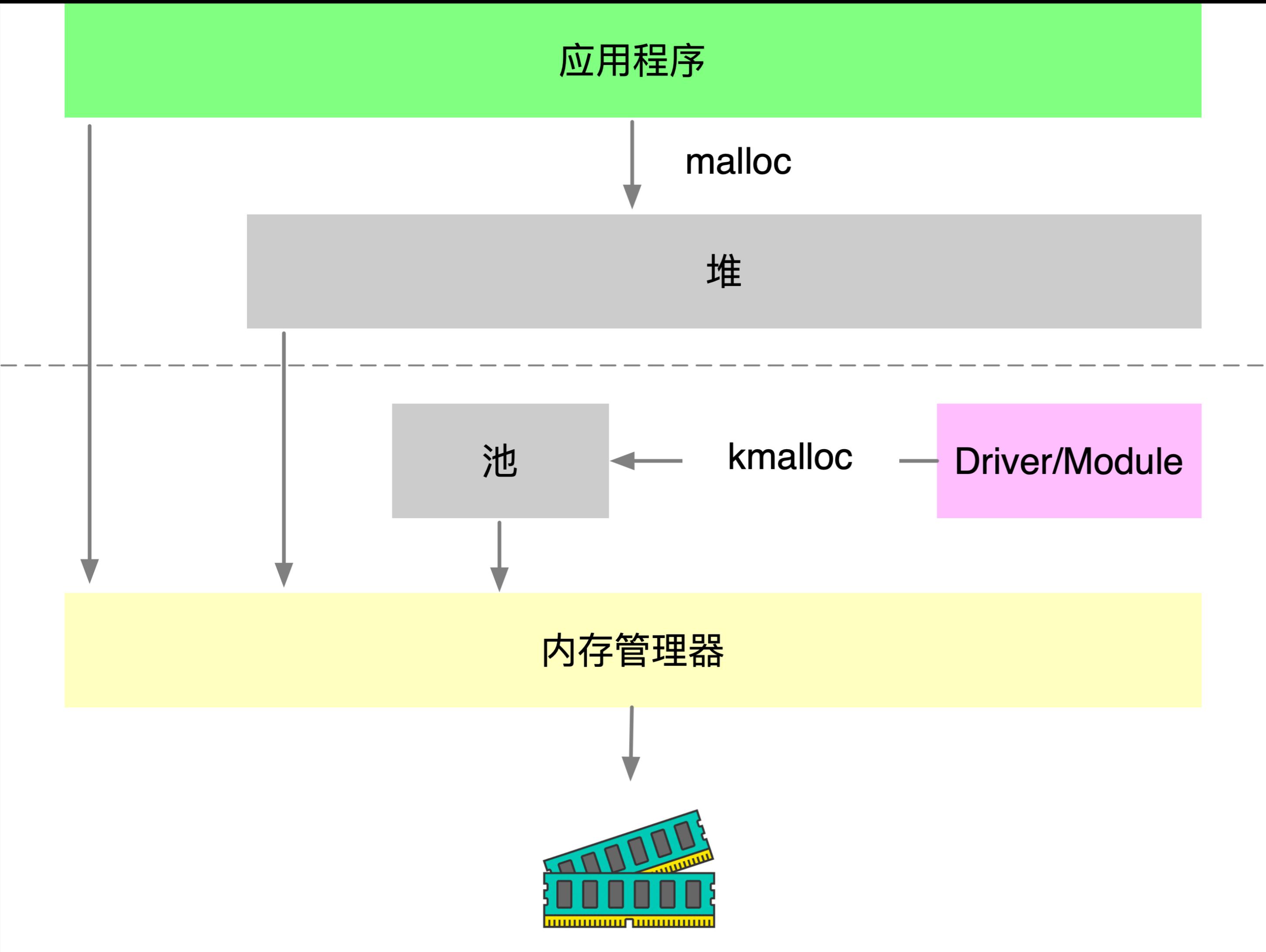
C 运行时库

内核层

`/usr/lib64/libc-2.17.so`

# libc 内存管理原理探究

# Linux 内存管理鸟瞰



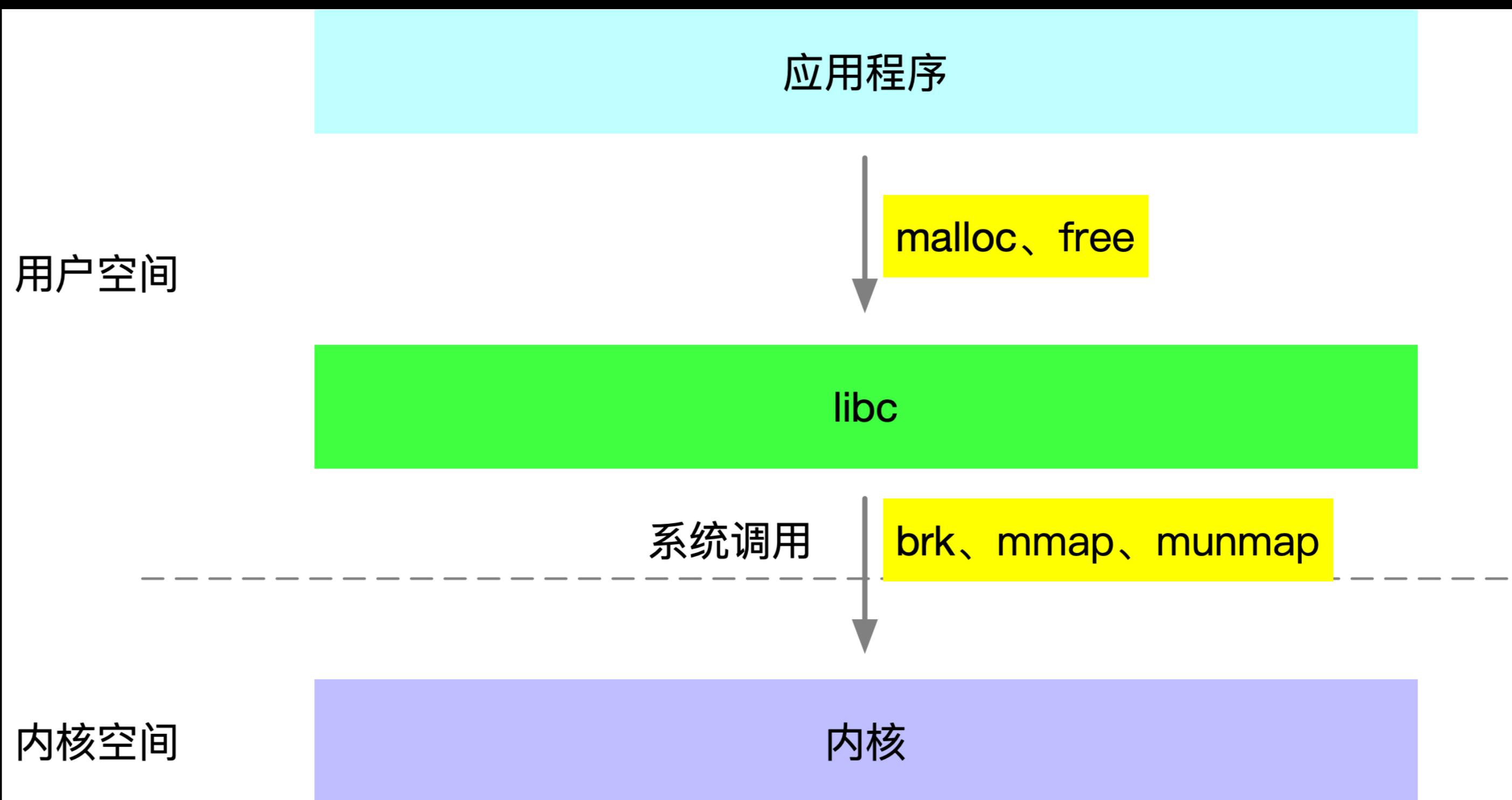
## 核心思想：

- 分层管理
- 批发与零售
- 隐藏内部细节

堆的管理是针对小内存分配来设计的，  
为了编程的统一，大内存也支持

# malloc 与 free

```
#include <stdlib.h>  
  
void *malloc(size_t size);  
  
void free(void *ptr);
```



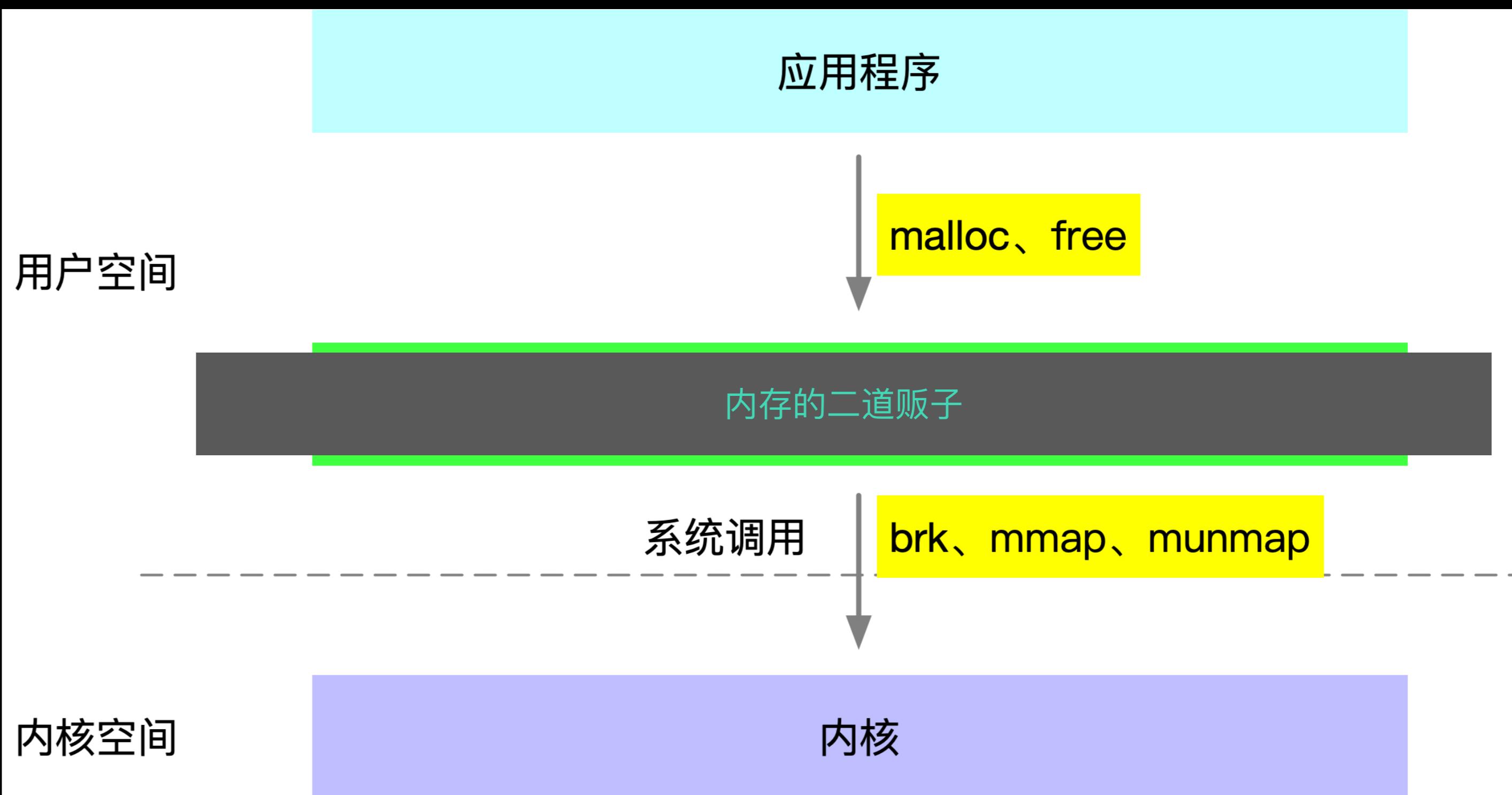
有了 brk、mmap 为什么还需要 libc 内存分配器？

# malloc 与 free

```
#include <stdlib.h>

void *malloc(size_t size);

void free(void *ptr);
```



有了 brk、mmap 为什么还需要 libc 内存分配器？

# Linux 内存分配器

dlmalloc

Doug Lea 开发

ptmalloc

基于 dlmalloc  
增加多线程支持

jemalloc

FreeBSD/Firefox

tcmalloc

Google/Golang

致力于解决两个问题：

- 1、多线程锁的粒度：全局锁、局部锁、无锁
- 2、小内存回收和内存碎片

# Linux 内存分配器

dmalloc

Doug Lea 开发

ptmalloc

基于 dmalloc  
增加多线程支持

jemalloc

FreeBSD/Firefox

tcmalloc

Google/Golang

致力于解决两个问题：

- 1、多线程锁的粒度：全局锁、局部锁、无锁
- 2、小内存回收和内存碎片

# ptmalloc2 的核心概念



内存分配主战场  
多线程

大块连续的内存  
区域

内存分配的单元

小块内存回收站

# 内存分配的主战场 Arena



- Arena 的出现用来优化多线程下全局锁的问题
- 尽可能让一个线程独占一个 Arena
- 一个线程申请的一个或多个堆，释放的内存会进入回收站，Arena 就是用来管理这些堆和回收站的

# Arena 的数据结构

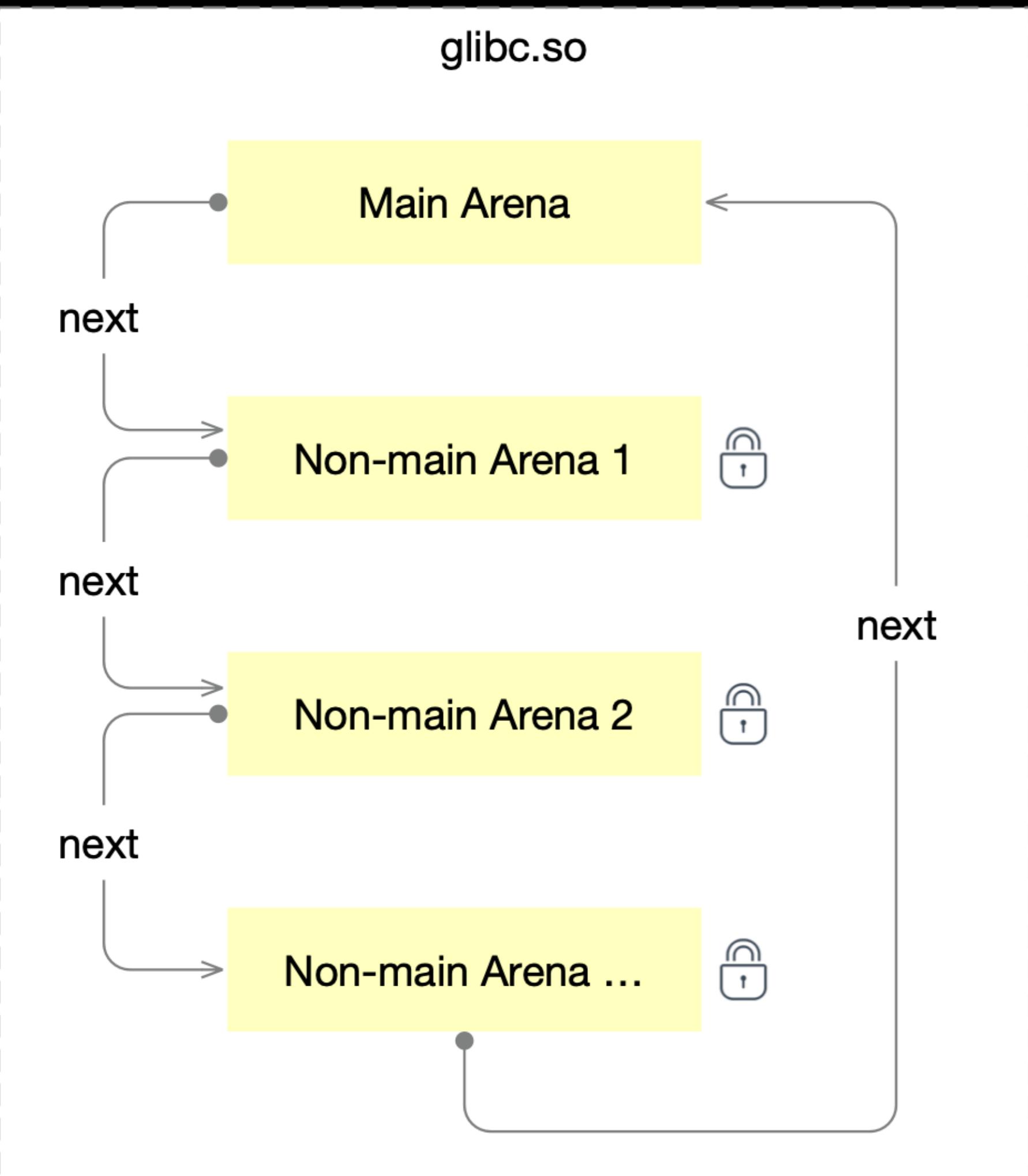


```
struct malloc_state {  
    /* Serialize access. */  
    mutex_t mutex;  
    /* Flags (formerly in max_fast). */  
    int flags;  
    /* Fastbins */  
    mfastbinptr fastbinsY[NFASTBINS];  
    /* Base of the topmost chunk -- not otherwise kept in a bin */  
    mchunkptr top;  
    /* The remainder from the most recent split of a small request */  
    mchunkptr last_remainder;  
    /* Normal bins packed as described above */  
    mchunkptr bins[NBINS * 2 - 2];  
    /* Bitmap of bins */  
    unsigned int binmap[BINMAPSIZE];  
    /* Linked list */  
    struct malloc_state *next;  
    /* Linked list for free arenas. */  
    struct malloc_state *next_free;  
    /* Memory allocated from the system in this arena. */  
    INTERNAL_SIZE_T system_mem;  
    INTERNAL_SIZE_T max_system_mem;  
};
```

## Arena 是什么？

- 一个单向循环链表
- 使用 **mutex** 锁来处理多线程竞争
- 释放的小块内存存储在 **bins** 中

# 主分配区 & 非主分配区



主分配区只有一个

总的分配区数量有上限: **cpu-cores \* 8**

多个 **Arena** 组成单向循环链表

# 写个代码打印 arena 列表

对于一个确定的程序，`main_arena` 的地址是位于 `libc` 库的一个确定的地址

```
(gdb) p &main_arena
$1 = (struct malloc_state *) 0x7ffff7dd4760 <main_arena>
(gdb)
```

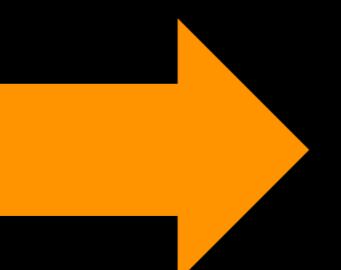
```
(gdb) ptype &main_arena
type = struct malloc_state {
    mutex_t mutex;
    int flags;
    mfastbinptr fastbinsY[10];
    mchunkptr top;
    mchunkptr last_remainder;
    mchunkptr bins[254];
    unsigned int binmap[4];
    struct malloc_state *next;
    struct malloc_state *next_free;
    size_t attached_threads;
    size_t system_mem;
    size_t max_system_mem;
} *
```

# 写个代码打印 arena 列表

do while 遍历这个循环链表

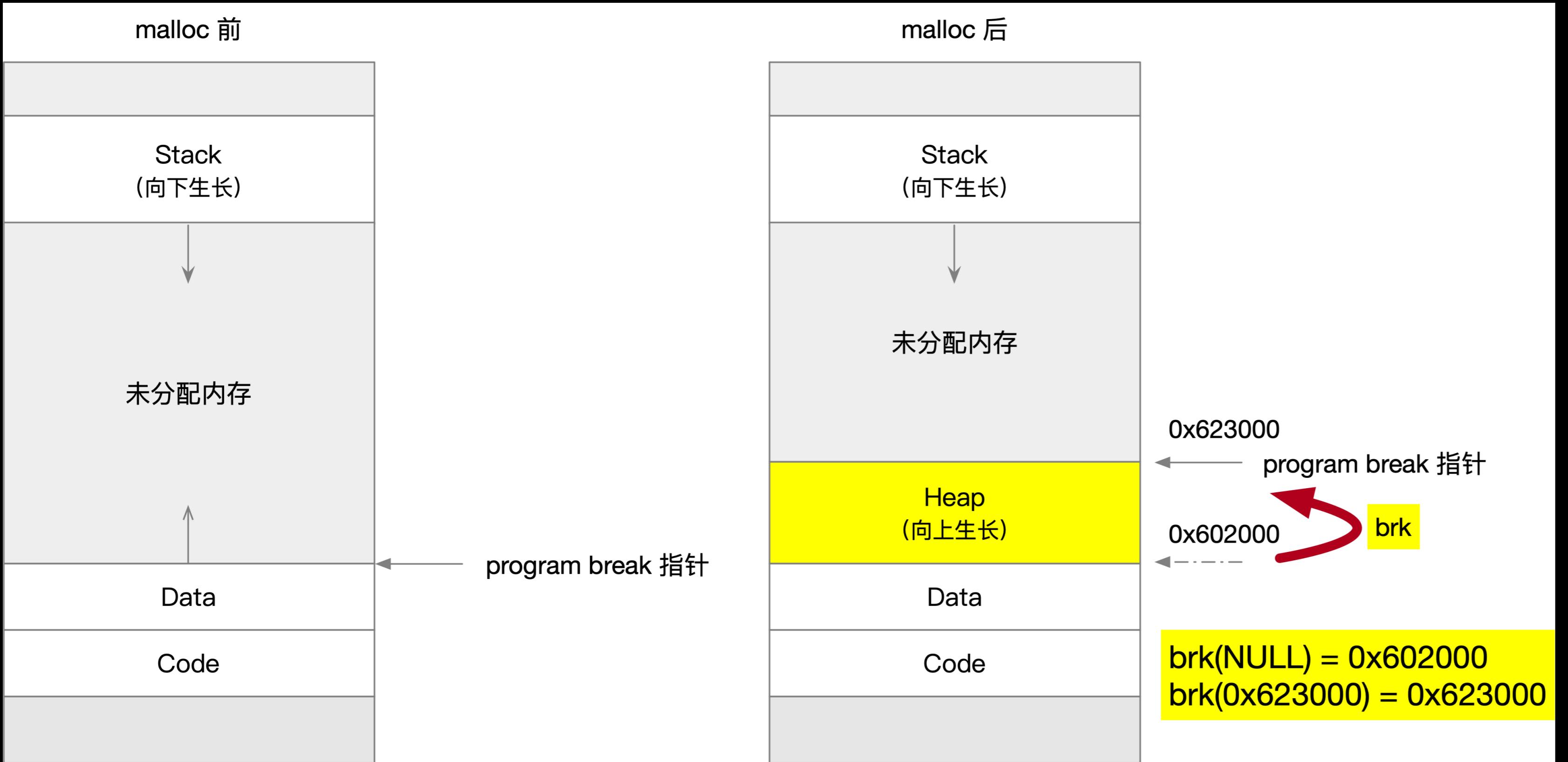
```
● ● ●  
  
#define MAIN_arena_ADDR 0x7ffff7bb8760  
  
print_arenas((void*)MAIN_arena_ADDR);
```

```
● ● ●  
  
void print_arenas(struct malloc_state *main_arena) {  
    struct malloc_state *ar_ptr = main_arena;  
    int i = 0;  
    do {  
        printf("arena[%02d] %p\n", i++, ar_ptr);  
        ar_ptr = ar_ptr->next;  
    } while (ar_ptr != main_arena);  
}
```



```
● ● ●  
  
arena[00] 0x7ffff7bb8760  
arena[01] 0x7fff80000020  
arena[02] 0x7fffcc000020  
arena[03] 0x7fff84000020  
arena[04] 0x7fff88000020  
arena[05] 0x7fffbc000020  
arena[06] 0x7fffc4000020  
arena[07] 0x7fffc0000020  
arena[08] 0x7fffc8000020  
arena[09] 0x7ffd00000020  
arena[10] 0x7ffd80000020  
arena[11] 0x7ffe40000020  
arena[12] 0x7ffe00000020  
arena[13] 0x7fffec000020  
arena[14] 0x7ffe80000020  
arena[15] 0x7fff00000020
```

# 主分配区

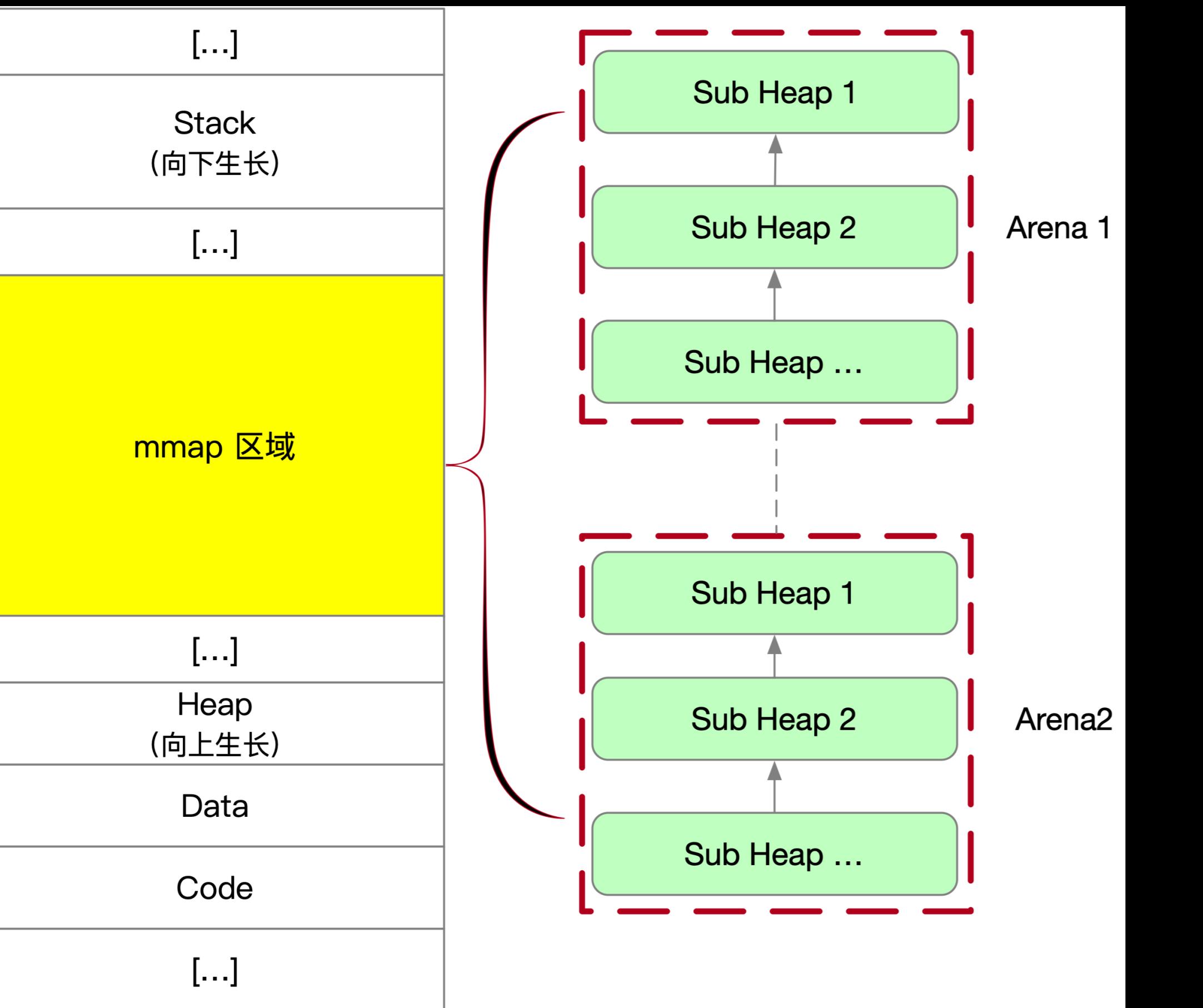


## 主分配区：

- 只有一个
- 特权：可以使用 Heap 区
- 通过 `brk` 指针来申请释放内存

从某种角度来讲，  
**HEAP 区不过是 DATA**  
 段的扩展而已

# 非主分配区：我可没有什么特权



像分封在外地自主创业的王爷

非主分配区：

- 使用 mmap 批发大块内存 (64M) 作为子堆 (Sub Heap)，再慢慢零售
- 一个 Sub Heap 用完，再开辟一个新的
- 一个 Arena 可以有多个子堆

# ptmalloc2 的核心概念



内存分配主战场

多线程



大块连续的内存

区域

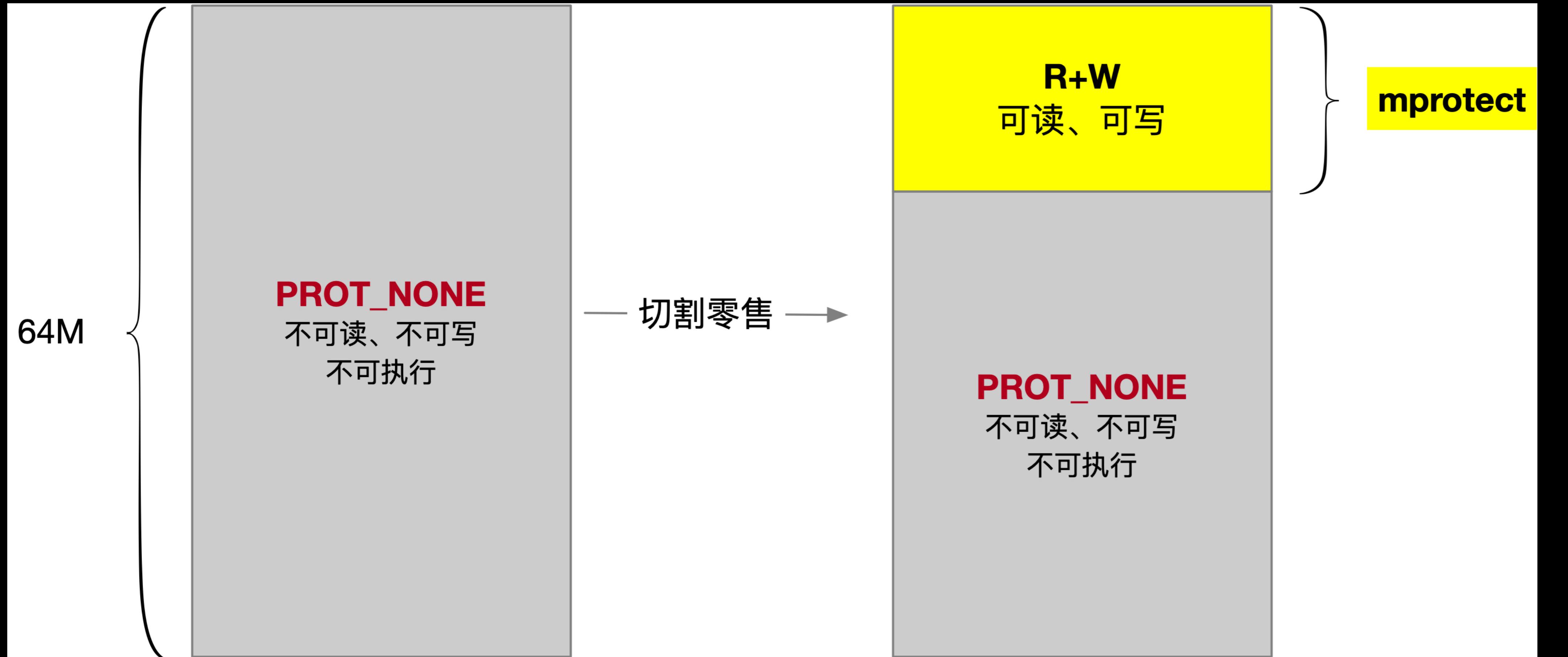


内存分配的单元

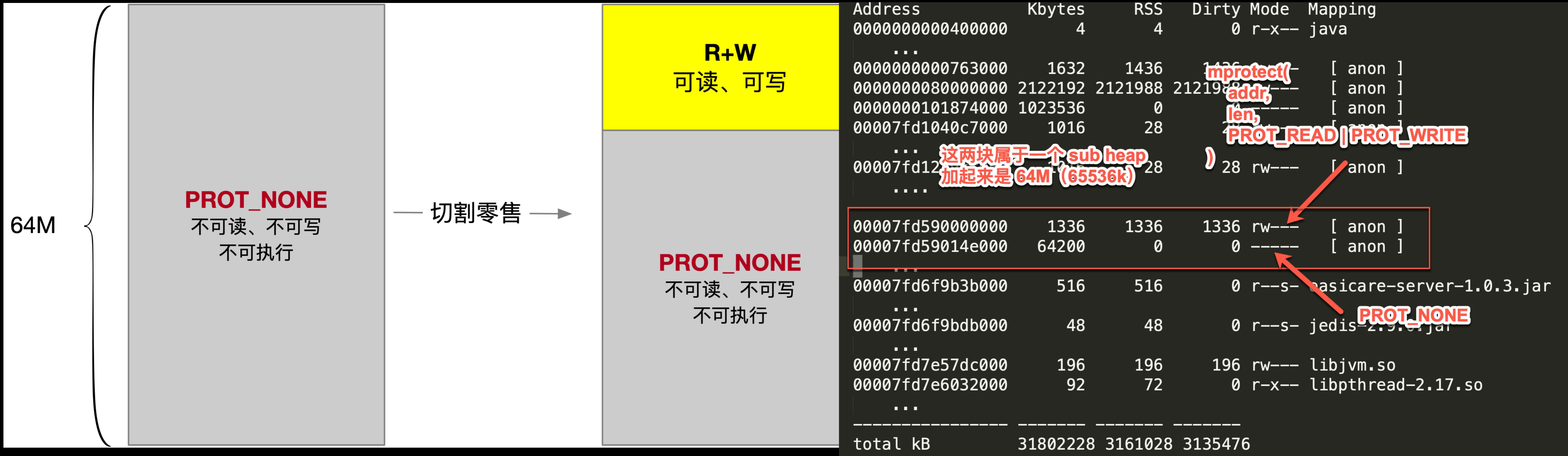


小块内存回收站

# 模拟堆（Sub Heap）的切割零售是什么意思



# 模拟堆 (Sub Heap) 的切割零售是什么意思



# 模拟堆 (Sub Heap) 的切割零售是什么意思

64M

**PRO**

不可逆 不可

cloud.tencent.com › developer › article ›

**当Java虚拟机遇上Linux Arena内存池- 云+社区- 腾讯云**

2018年3月7日 — JAVA 进程在64位LINUX下占用巨大内存的分析. 文章链接: <https://blog.chou.it/2014/03/java-consume-huge-memory-on-64bit-linux>. Linux glibc ...

您已浏览过该网页 3 次。上次访问日期: 20年11月3日

www.xuetimes.com › archives ›

**Java 进程在64位linux下占用巨大内存的分析- 学时网**

2018年9月25日 — Java 进程在64位linux下占用巨大内存的分析 ... 数量级, 接下来我在谷歌上搜索"java huge memory usage 64m"发现别人也遇到过这个问题, 在 ...

blog.11034.org › 64bits\_linux\_arena\_memory ›

**64位Linux下Java进程堆外内存迷之64M问题 - Fly あ梦**

此进程主要处理Socket IO读写, 使用的是Java NIO。 <http://stackoverflow.com/questions/561245/virtual-memory-usage-from-java-under-linux-too-large> ...

您已浏览过该网页 3 次。上次访问日期: 20年11月17日

club.perfma.com › article ›

**一次Java 进程OOM 的排查分析 (glibc 篇)**

Linux 中典型的大量64M 内存区域问题; glibc 的内存分配器ptmalloc2 的底层原理 ... int main() { char \*ptrs[MAXNUM]; int i; // malloc large block memory for (i = 0; ...

您已浏览过该网页 2 次。上次访问日期: 20年10月28日

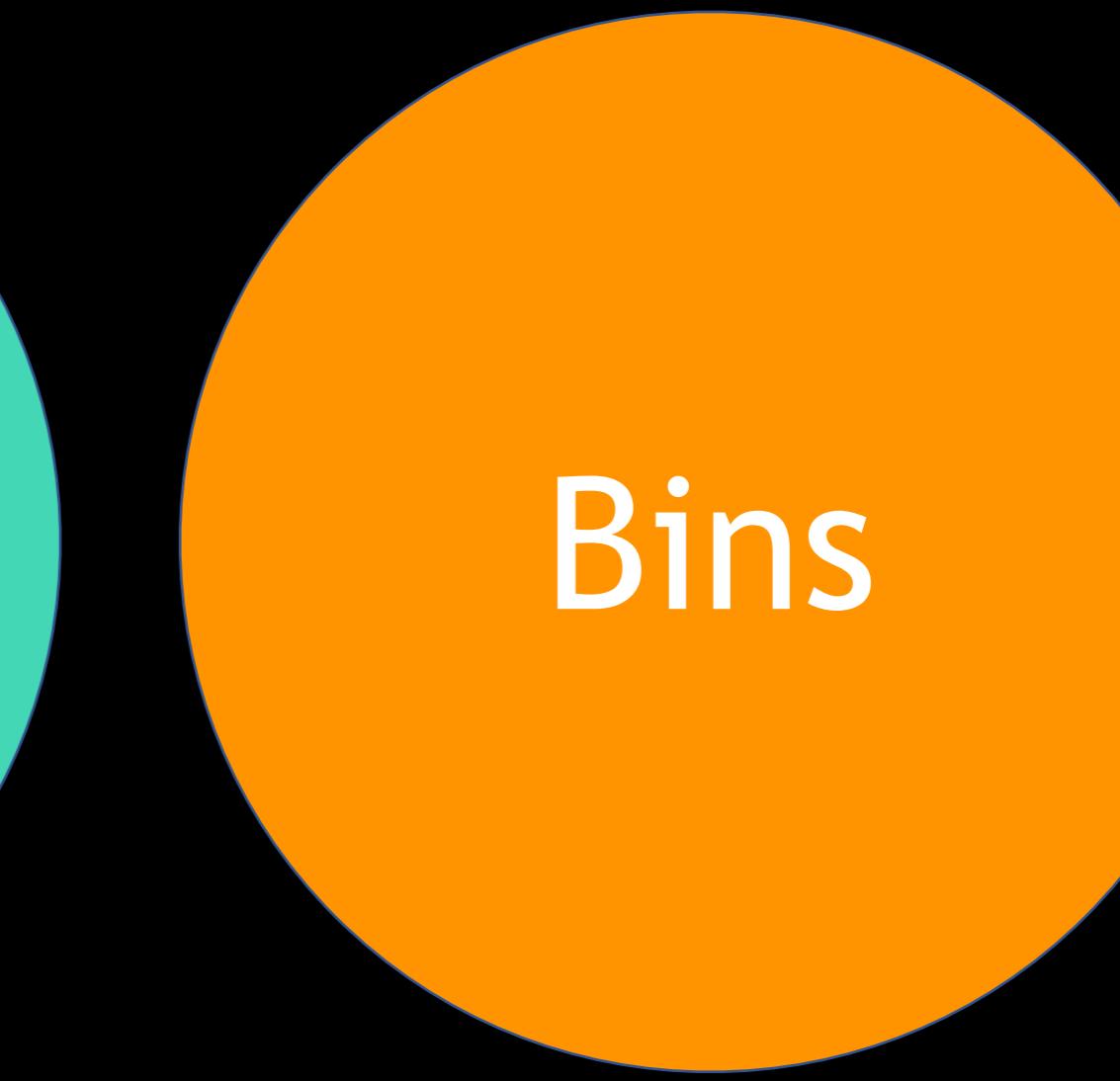
Kbytes	RSS	Dirty	Mode	Mapping
4	4	0	r-x--	java
1632	1436	1436	mprotect( addr, len, PROT_READ   PROT_WRITE )	[ anon ]
2122192	2121988	2121988	-----	[ anon ]
1023536	0	0	-----	[ anon ]
1016	28	28	-----	[ anon ]
<b>于一个sub heap 64M (65536k)</b>				) 28 rw--- [ anon ]
1336	1336	1336	rw---	[ anon ]
64200	0	0	-----	[ anon ]
516	516	0	r--s-	basicare-server-1.0.3.jar
48	48	0	r--s-	jedis-2.9.6.jar
196	196	196	rw---	libjvm.so
92	72	0	r-x--	libpthread-2.17.so
-----				
31802228	3161028	3135476		

# 模拟堆 (Sub Heap) 遍历

```
● ● ●  
#define MAIN_arena_ADDR 0x7ffff7bb8760  
dump_non_main_subheaps((void*)MAIN_arena_ADDR);
```

```
● ● ●  
void dump_non_main_subheaps(struct malloc_state *main_arena) {  
    struct malloc_state *ar_ptr = main_arena->next;  
    int i = 0;  
    while (ar_ptr != main_arena) {  
        printf("arena[%d]\n", ++i);  
        struct heap_info *heap = heap_for_ptr(ar_ptr->top);  
        do {  
            printf("arena:%p, heap: %p, size: %d\n", heap->ar_ptr, heap, heap->size);  
            heap = heap->prev;  
        } while (heap != NULL);  
        ar_ptr = ar_ptr->next;  
    }  
}
```

# ptmalloc2 的核心概念



内存分配主战场

大块连续内存区

内存分配的单元

小块内存回收站

多线程

域

# 从 malloc 看 Chunk 的结构

```
● ● ●

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
    void *p;

    p = malloc(1024);
    printf("%p\n", p);

    p = malloc(1024);
    printf("%p\n", p);

    p = malloc(1024);
    printf("%p\n", p);

    getchar();
    return (EXIT_SUCCESS);
}

./malloc_test
```

$1024=0x400$   
指针地址相差  $0x410$

多出来的  $0x10$  字节是什么？

# 再看 malloc 与 free

```
#include <stdlib.h>  
  
void *malloc(size_t size);  
  
void free(void *ptr);
```

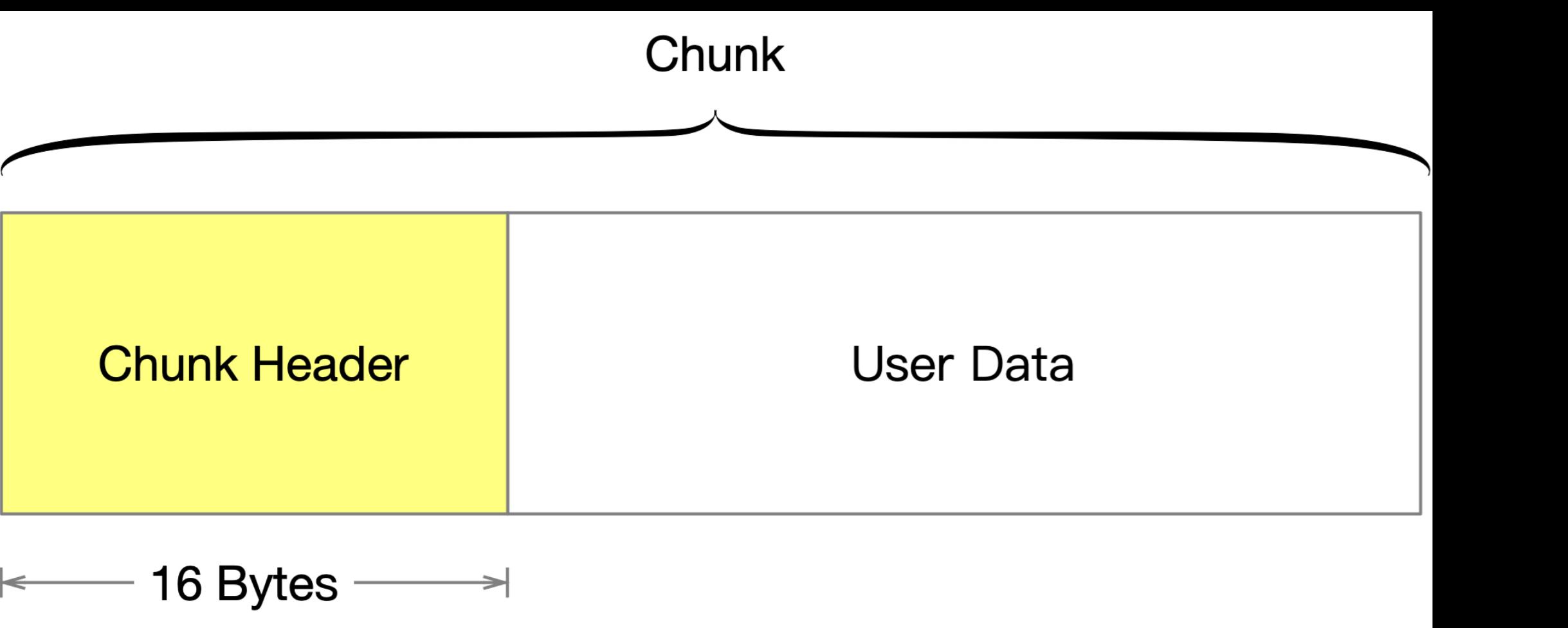
free 函数的参数只有一个指针，它是怎么知道要释放多少内存的？

---

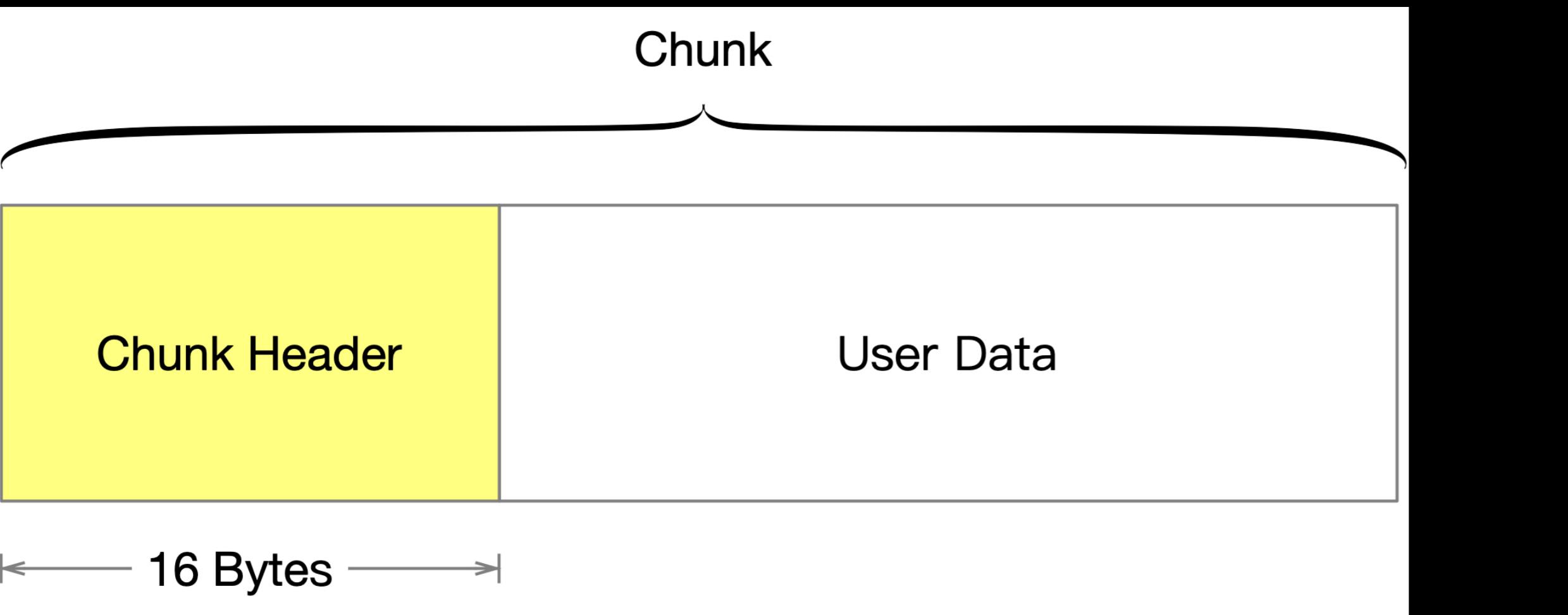
凡事皆有代价，快乐的代价便是痛苦

张小娴

# malloc 记录内存的额外开销

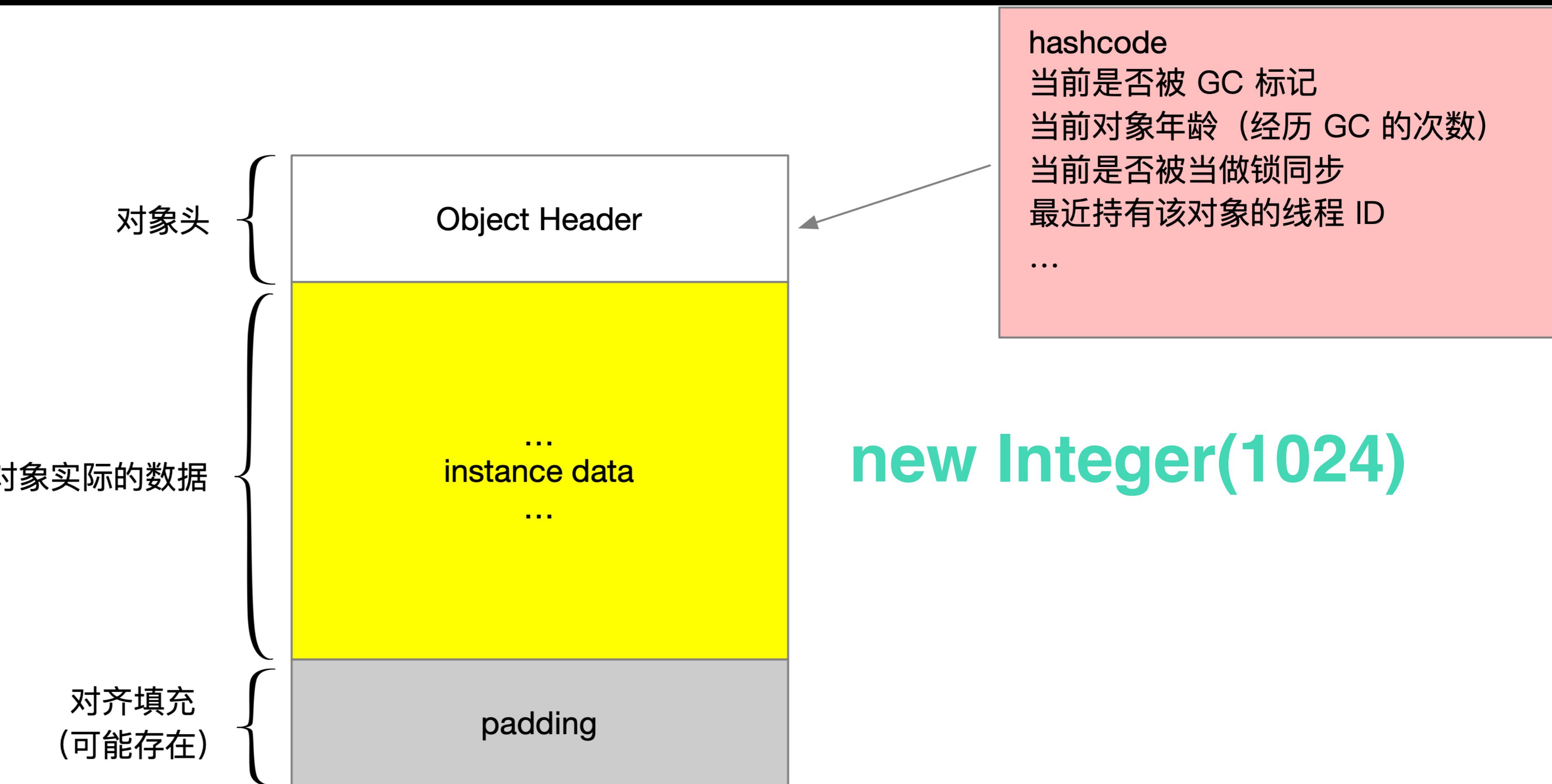
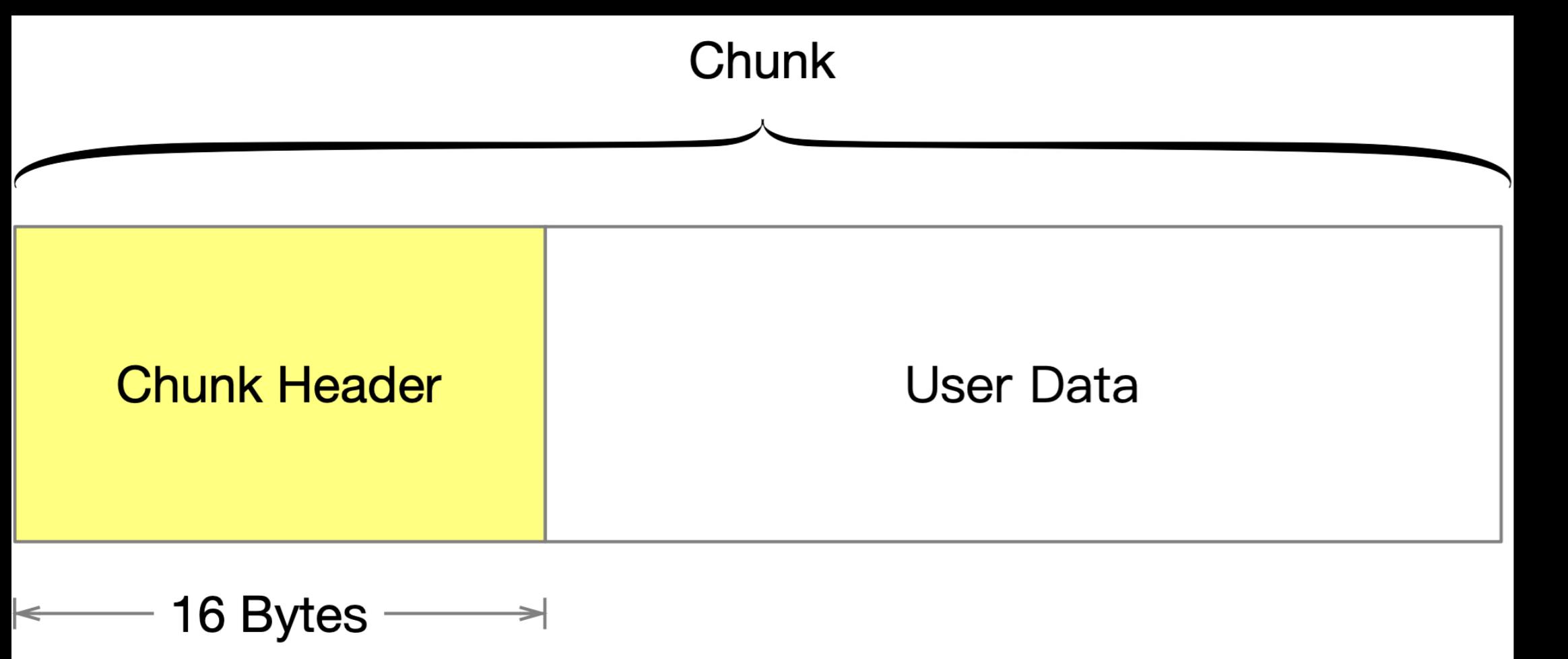


# malloc 记录内存的额外开销

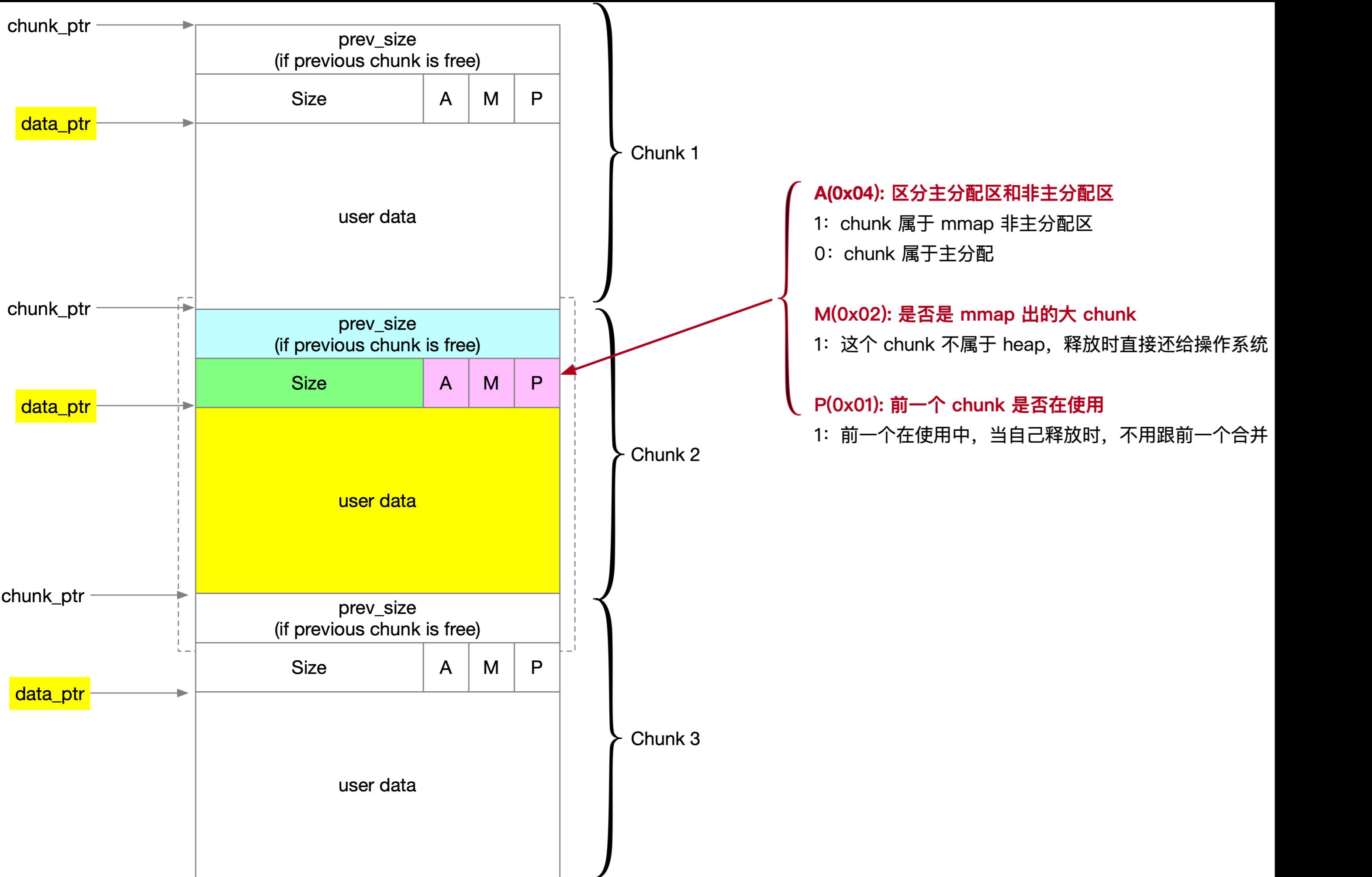


**new Integer(1024)**

# malloc 记录内存的额外开销



# malloc chunk 的基本结构



**A(0x04): 区分主分配区和非主分配区**

1: chunk 属于 mmap 非主分配区  
0: chunk 属于主分配

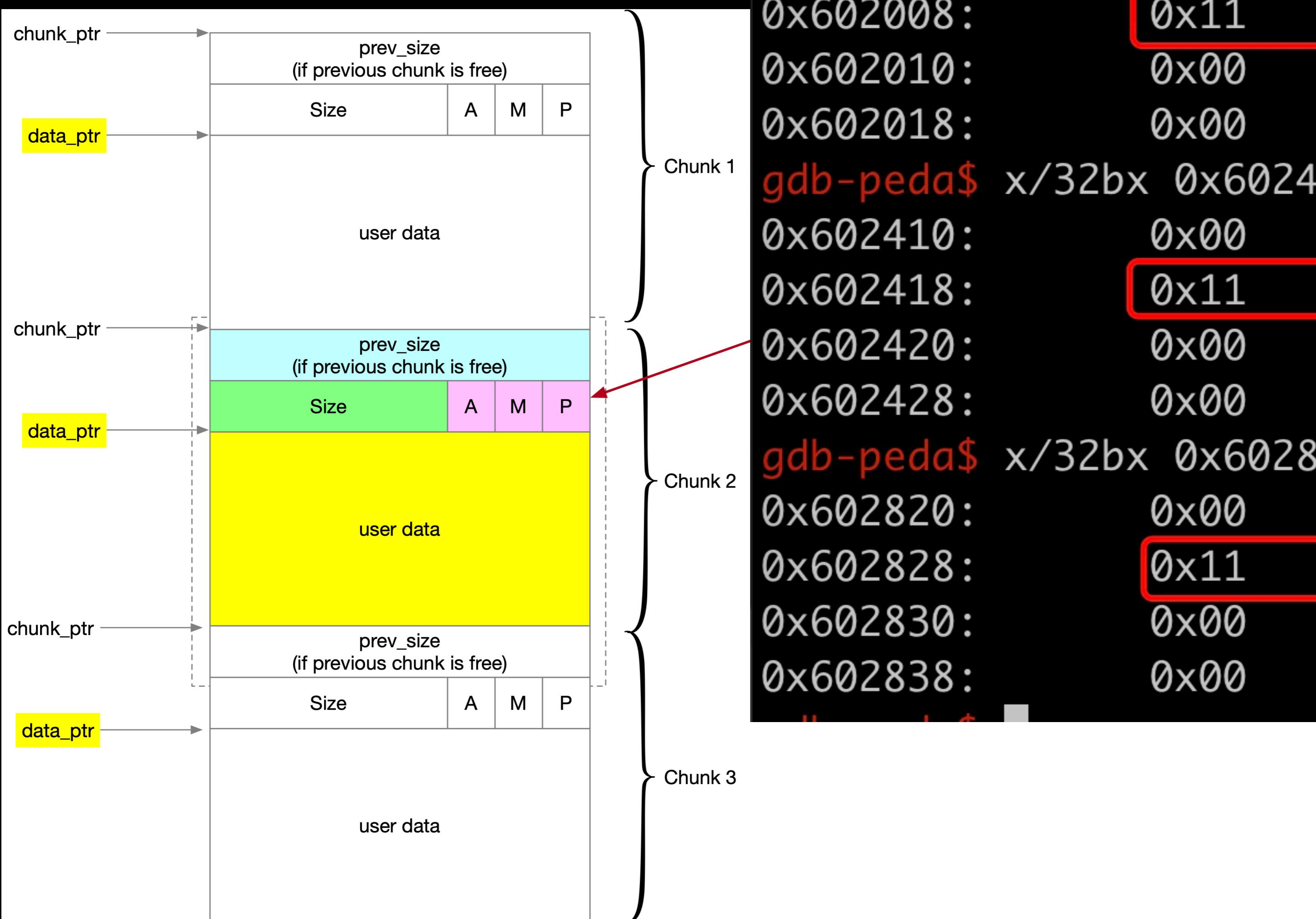
**M(0x02): 是否是 mmap 出的大 chunk**

1: 这个 chunk 不属于 heap, 释放时直接还给操作系统

**P(0x01): 前一个 chunk 是否在使用**

1: 前一个在使用中, 当自己释放时, 不用跟前一个合并

# malloc chunk 的基本结构



```
gdb-peda$ x/32bx 0x602010- 0x1
```

0x602000: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

0x602008: 0x11 0x04 0x00 0x00 0x00 0x00 0x00 0x00

0x602010: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

0x602018: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

0x002010: 0x00 0x00  
adb nodes\$ x/32bx 0x603420 0x1

gub-peaus\$ x/32Dx 0x002420- 0x10

0x602410: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

0x602418: 0x11 0x04 0x00 0x00 0x00 0x00 0x00 0x00

0x602420: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

0x602428: 0x00 0x00

*gdb-peda\$ x/32bx 0x602830- 0x10*

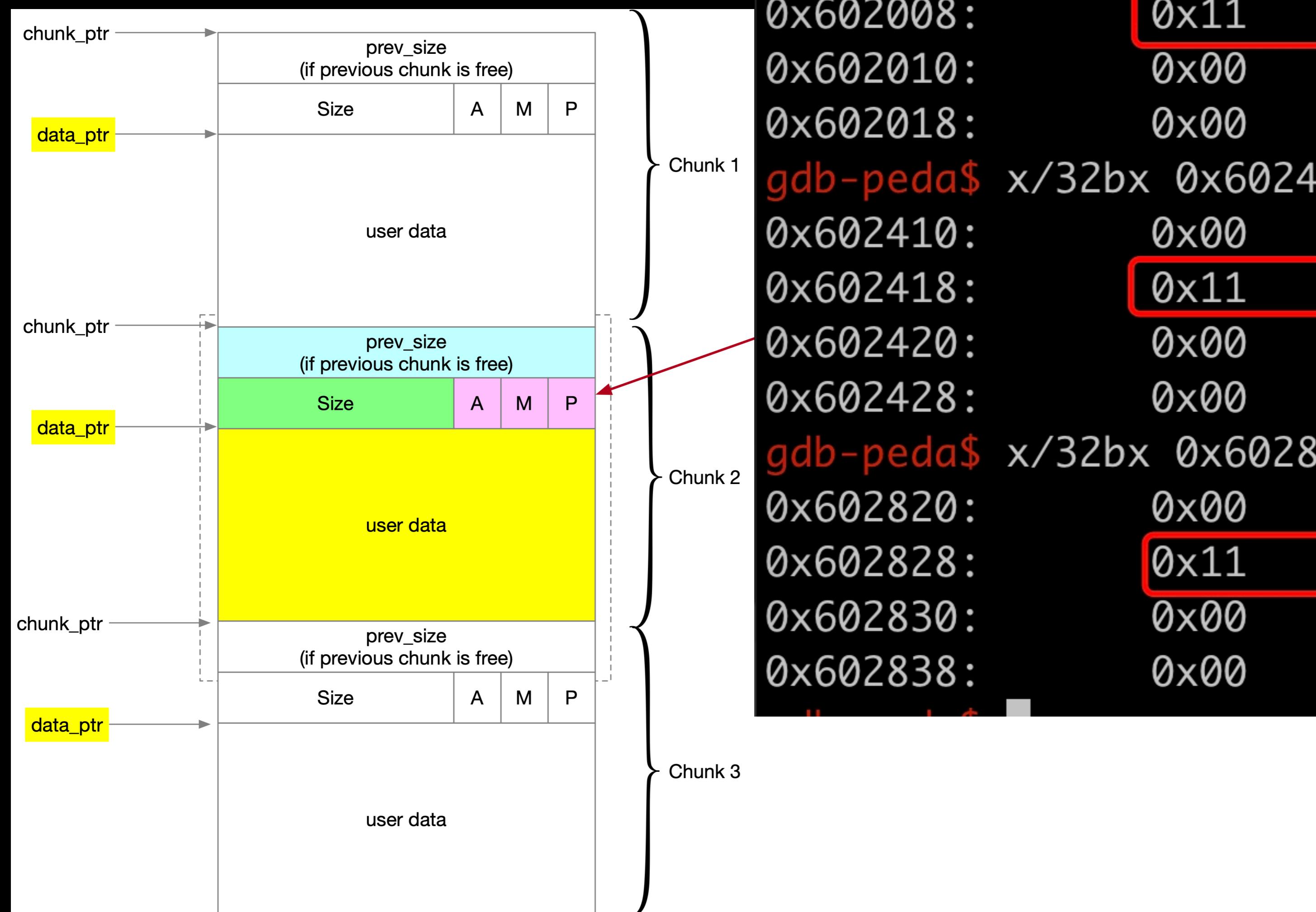
0x602820: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

0x602828: 0x11 0x04 0x00 0x00 0x00 0x00 0x00 0x00

0x602830: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

0x602838 : 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

# malloc chunk 的基本结构



gdb-peda\$ x/32bx 0x602010- 0x10

0x602000:	0x00						
0x602008:	0x11	0x04	0x00	0x00	0x00	0x00	0x00
0x602010:	0x00						
0x602018:	0x00						

gdb-peda\$ x/32bx 0x602420- 0x10

0x602410:	0x00						
0x602418:	0x11	0x04	0x00	0x00	0x00	0x00	0x00
0x602420:	0x00						
0x602428:	0x00						

gdb-peda\$ x/32bx 0x602830- 0x10

0x602820:	0x00						
0x602828:	0x11	0x04	0x00	0x00	0x00	0x00	0x00
0x602830:	0x00						
0x602838:	0x00						

(size + 8) 对齐到 16B + b001

$0x0400 + 0x10 + 0x01 = 0x0411$

# ptmalloc2 的核心概念



内存分配主战场

多线程



大块连续的内存

区域



内存分配的单元

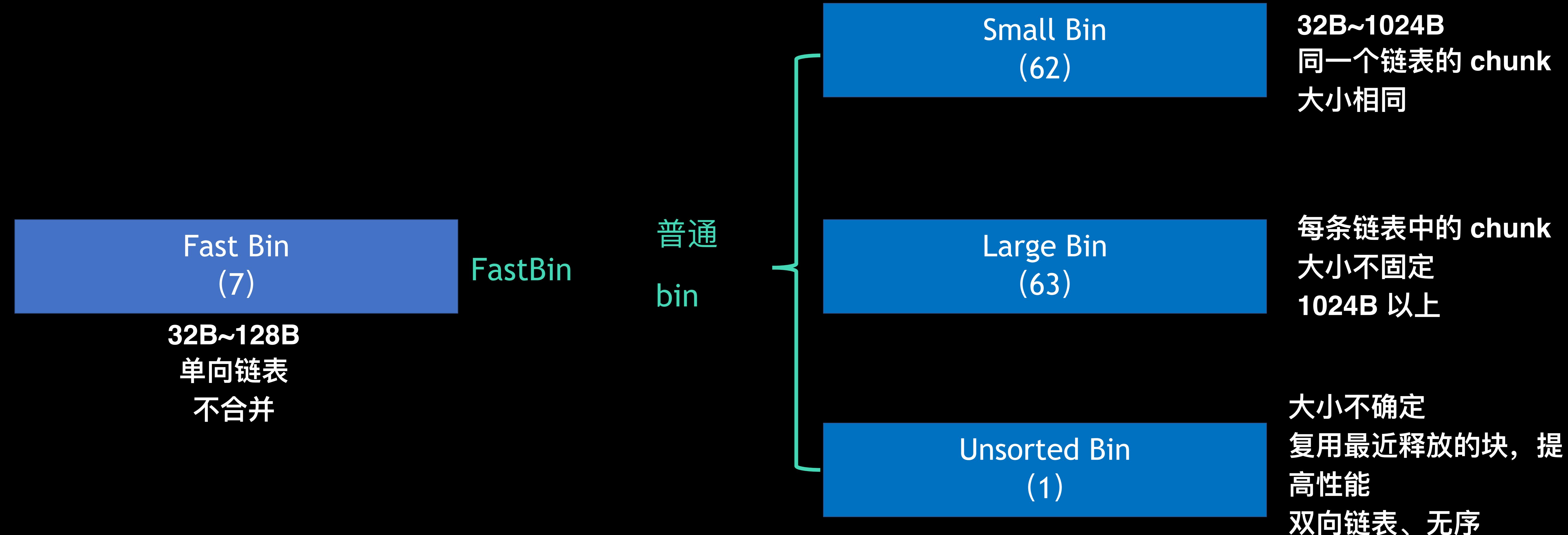
区域



Bins

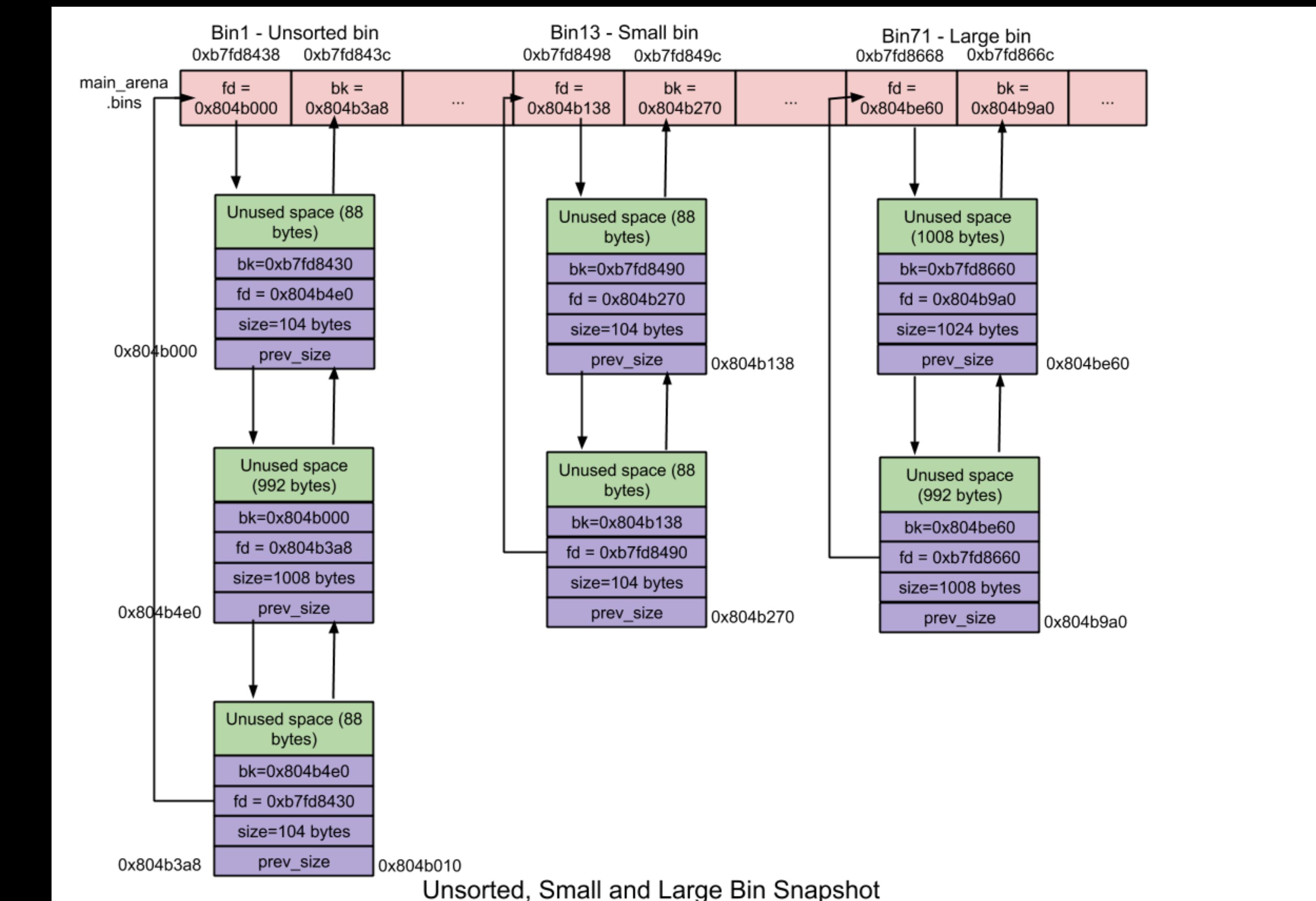
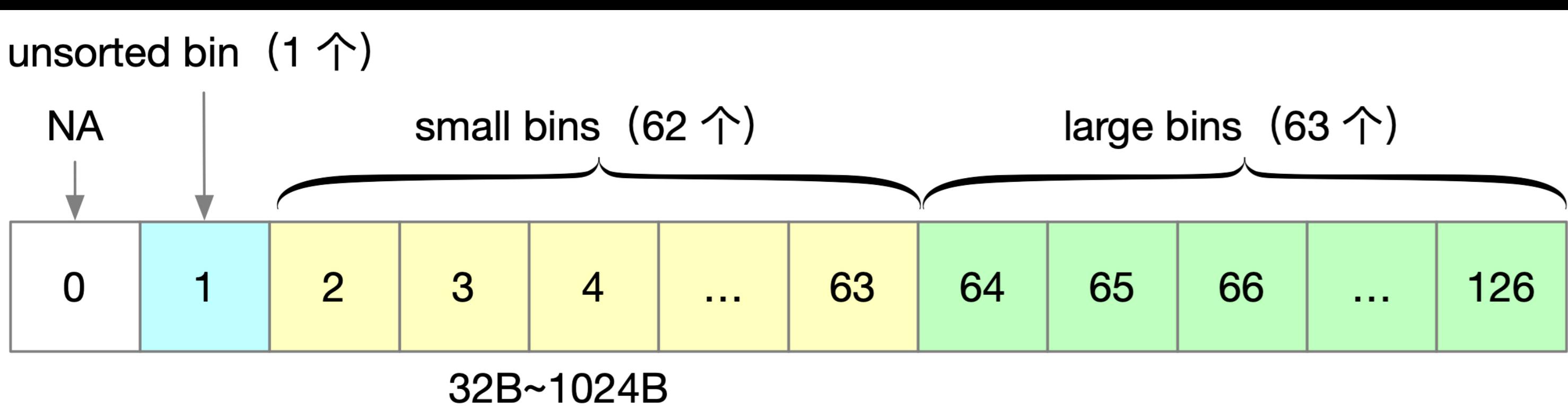
小块内存回收站

# chunk 的回收站：看人下饭

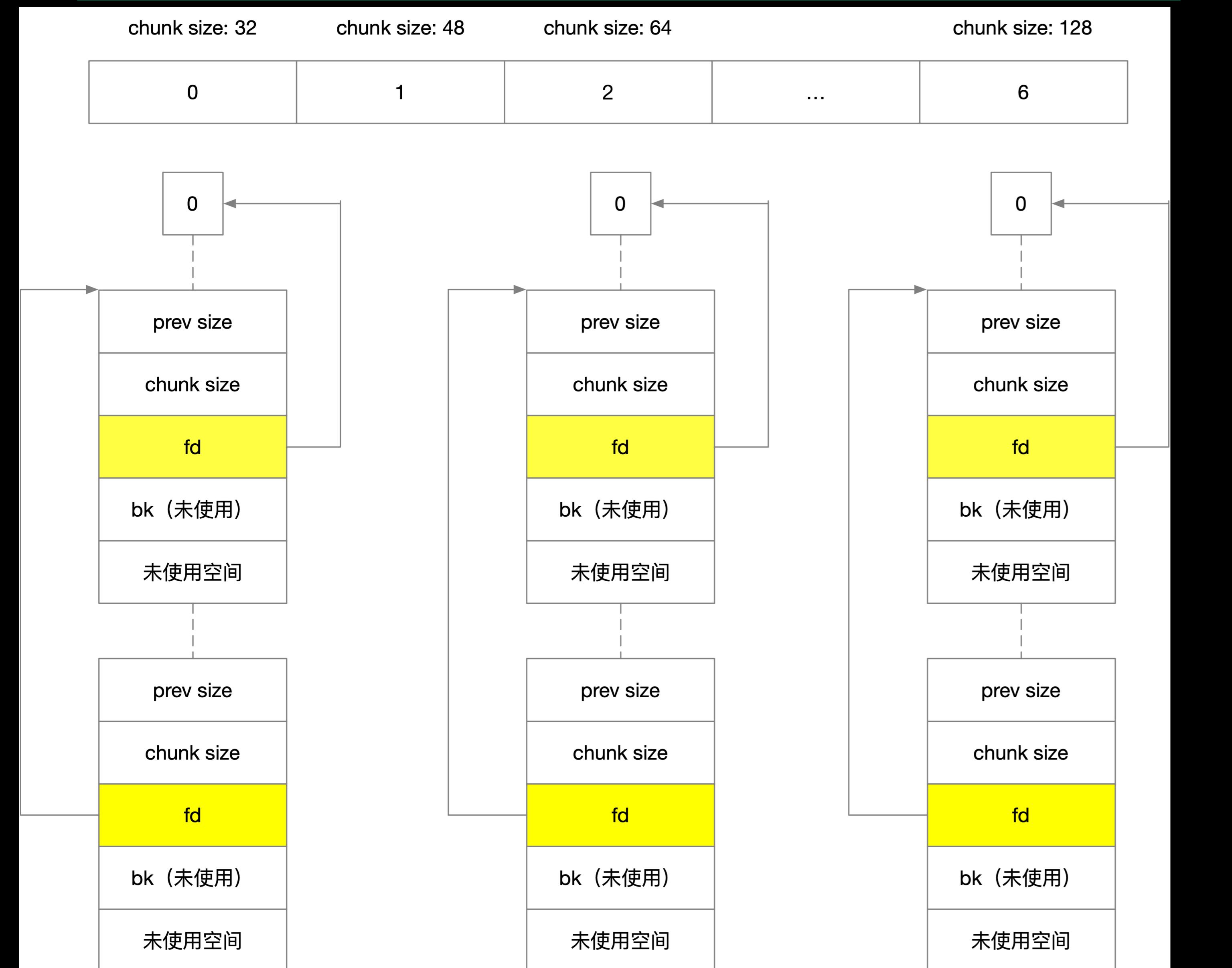


# 普通 bin 的结构一览

```
(gdb) ptype &main_arena
type = struct malloc_state {
    mutex_t mutex;
    int flags;
    mfastbinptr fastbinsY[10];
    mchunkptr top;
    mchunkptr last_remainder;
    mchunkptr bins[254];
    unsigned int binmap[4];
    struct malloc_state *next;
    struct malloc_state *next_free;
    size_t attached_threads;
    size_t system_mem;
    size_t max_system_mem;
} *
```



# FastBin



- 专门用来提高小内存的分配效率
- 小于 128B 的内存分配会先在 Fast Bin 中查找
- 单向链表，每条链表中的 chunk 大小相同
- P 标记始终为 1，一般情况下不合并
- FIFO，添加和删除都从队尾进行

# 小块内存申请简化流程



# free 如何处理

大小符合 Fast Bin

直接放入 fastbin 单链表

快速释放

这么点空间，值得我处理半天吗？

超大块

直接还给内核

不进入 bin 的管理

大客户要特殊处理，大客户是少数情况

介于中间的

放入 Unsorted Bin

根据情况合并、迁移空闲块，靠近 top 则更

新 top

这才是人生常态

# 小结

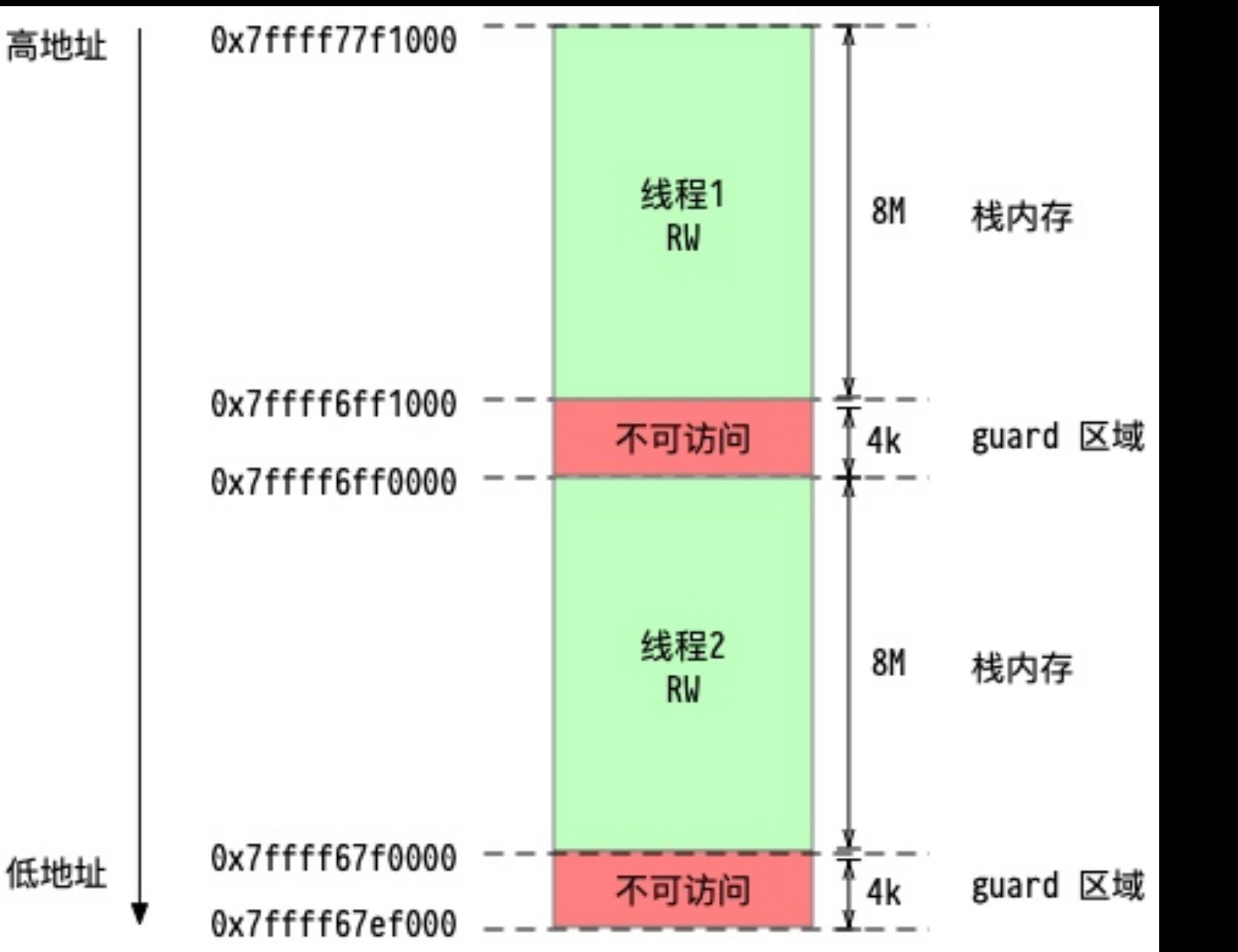
- 虚拟内存物理内存、内存布局
- 分层管理：应用层、libc C 库层、内核层
- 批发与零售：子堆、bins
- 隐藏内部细节：malloc 和 free 对底层实现细节的屏蔽

---

# 不可忽视的栈内存

# 关于栈内存

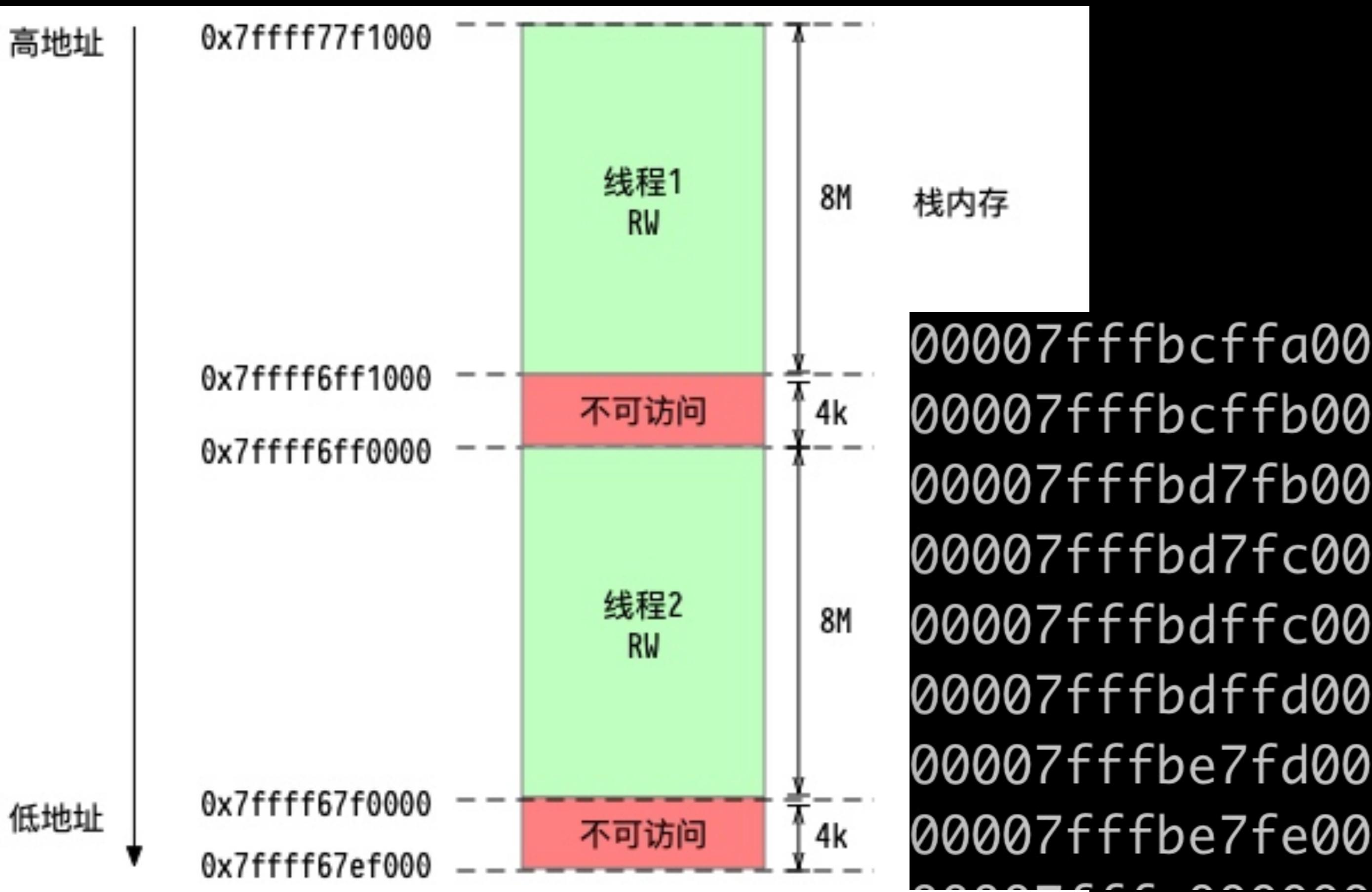
## Linux pthread 栈内存



默认为 8M + 4k guard page

# 关于栈内存

Linux pthread 栈内存



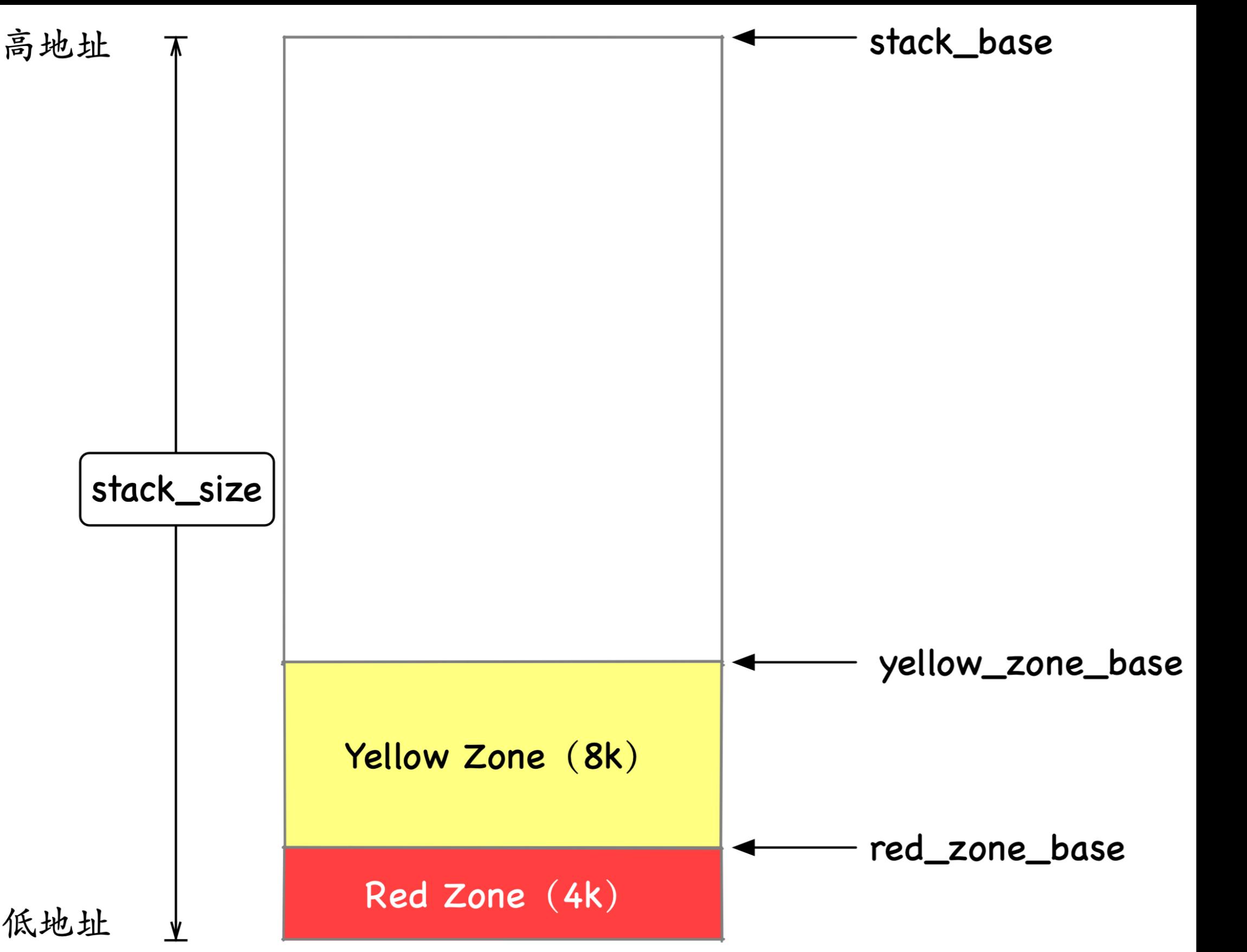
默认为 8M + 4k guard page

00007fffbcffa000	4	0	0	-----	[ anon ]
00007fffbcffb000	8192	8	8	rw---	[ anon ]
00007fffbd7fb000	4	0	0	-----	[ anon ]
00007fffbd7fc000	8192	8	8	rw---	[ anon ]
00007fffbdffc000	4	0	0	-----	[ anon ]
00007fffbdffd000	8192	8	8	rw---	[ anon ]
00007fffbe7fd000	4	0	0	-----	[ anon ]
00007fffbe7fe000	8192	8	8	rw---	[ anon ]

4k guard page

8M Stack

# 关于栈内存



Java 栈内存

默认为 1M + 4k Red Zone + 8K Yellow Zone

## 第三部分：开发相关的内存问题说明

# 一些误区的澄清

---

为什么Java 进程占用的内存远大于Xmx

# 一些误区的澄清



- Q: [Java using much more memory than heap size](#)
- Q: [Why does java not seem to respect memory limits?](#)
- Q: [Java memory usage much higher than heap+nonheap](#)
- Q: [Java using up far more memory than allocated with -Xmx](#)
- Q: [Limiting Java 8 Memory Consumption](#)
- Q: [How do I keep memory used by JVM under control?](#)
- Q: [How to find what is using physical memory in a Java process](#)
- Q: [Is Linux RSS not equivalent to java Xmx + MaxMetaspaceSize?](#)
- Q: [oom-killer kills java application in Docker](#)

# Java 内存都被谁消耗了

Heap  
Code Cache  
GC 开销  
Metaspace  
Thread Stack  
Direct Buffers  
Mapped files  
C/C++ Native 内存消耗  
malloc 本身的开销  
. . .

## Java 内存消耗

内存大户不是开玩笑的

根据多年实践 `Xmx` 设置为容器内存的 65% 左右比较合理

# 一些误区的澄清

问题 2:

TOP 命令中 RES 占用很高，是不是代表程序真正有大量消耗呢？

# RES 占用

java -Xms1G -Xmx1G MyTest

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
23779	ya	20	0	3205.8m	24.6m	9.9m	S	0.0	1.3	0:00.12	java

# RES 占用

**java -XX:+AlwaysPreTouch -Xms1G -Xmx1G MyTest**

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
23977	ya	20	0	3205.8m	1.024g	10.0m	S	0.0	57.1	0:00.20	java
23978	.	20	0	540.6m	1.024g	5.0m	S	0.0	0.0	0:00.17	.

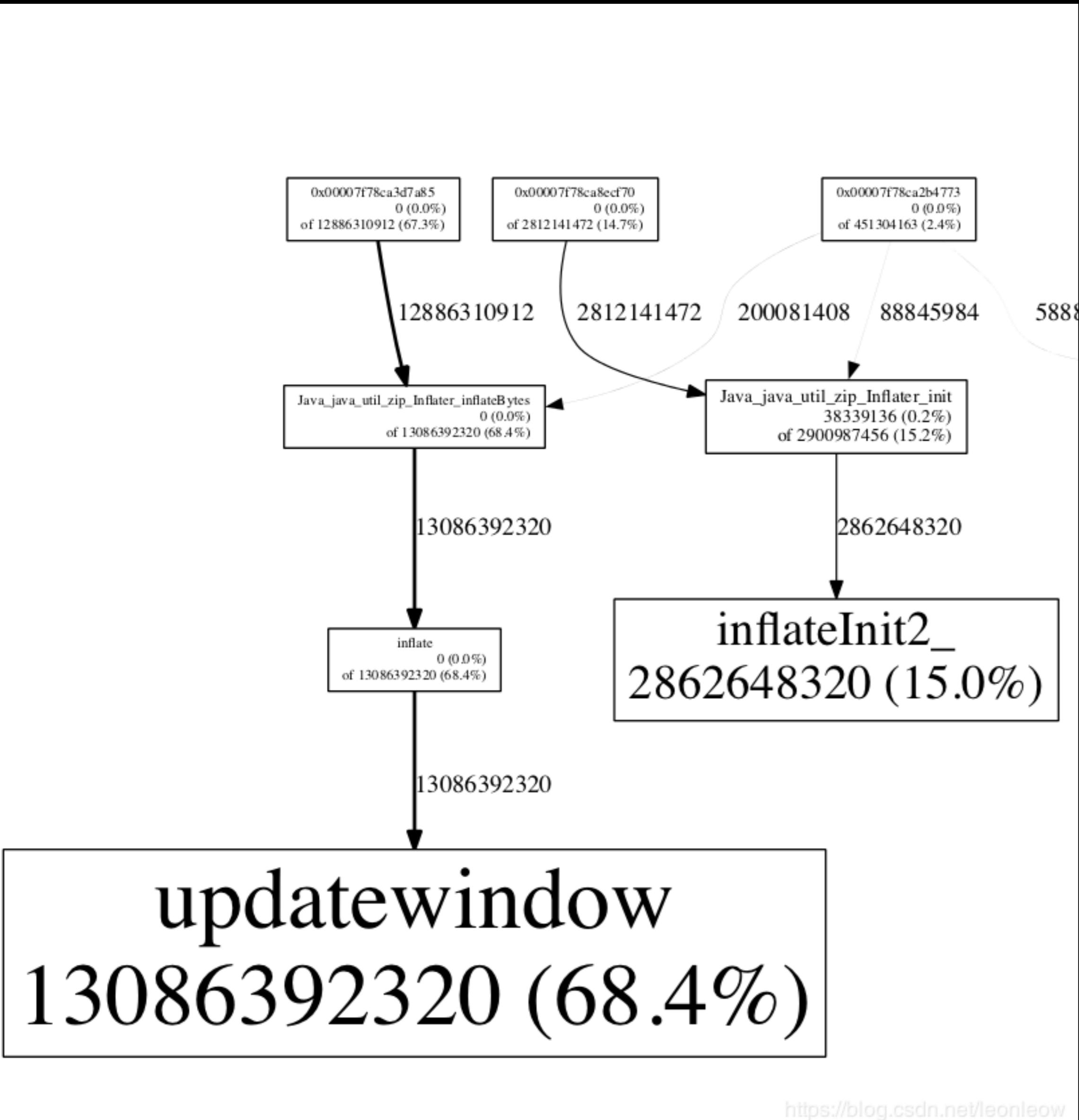
RES 显示占用的内存并不一定是你程序实际消耗的  
另外：内存占用并不是越小越好，要通盘考虑

# 是时候考慮替換默认的内存分配器了

**LD\_PRELOAD=/usr/local/lib/libjemalloc.so**

RSS: 7G-> 3G

# jemalloc 强大的 profiler 功能



export

```
MALLOC_CONF=prof:true,lg_prof_sample:1,lg_prof_interval:30,prof_prefix:jeprof.out
```

```
jeprof -svg /path/to/svg jeprof.out.* > out.svg
```

## Native 内存排查神器

# THANKS

