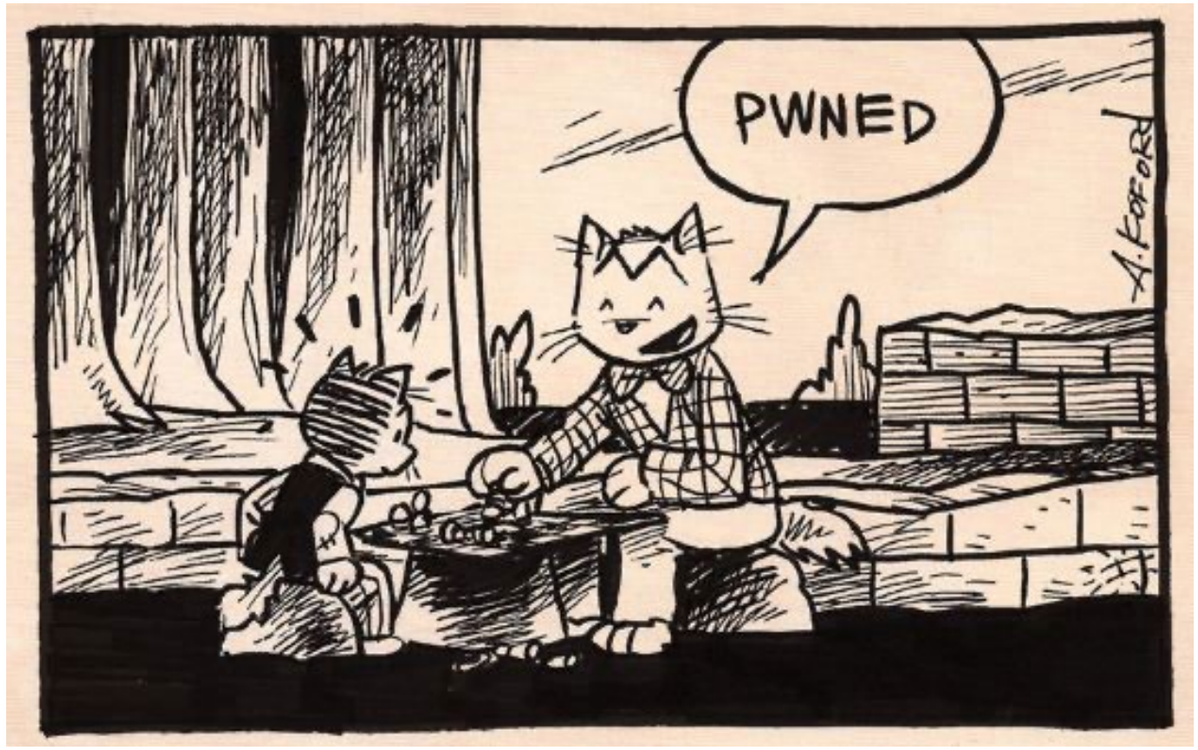
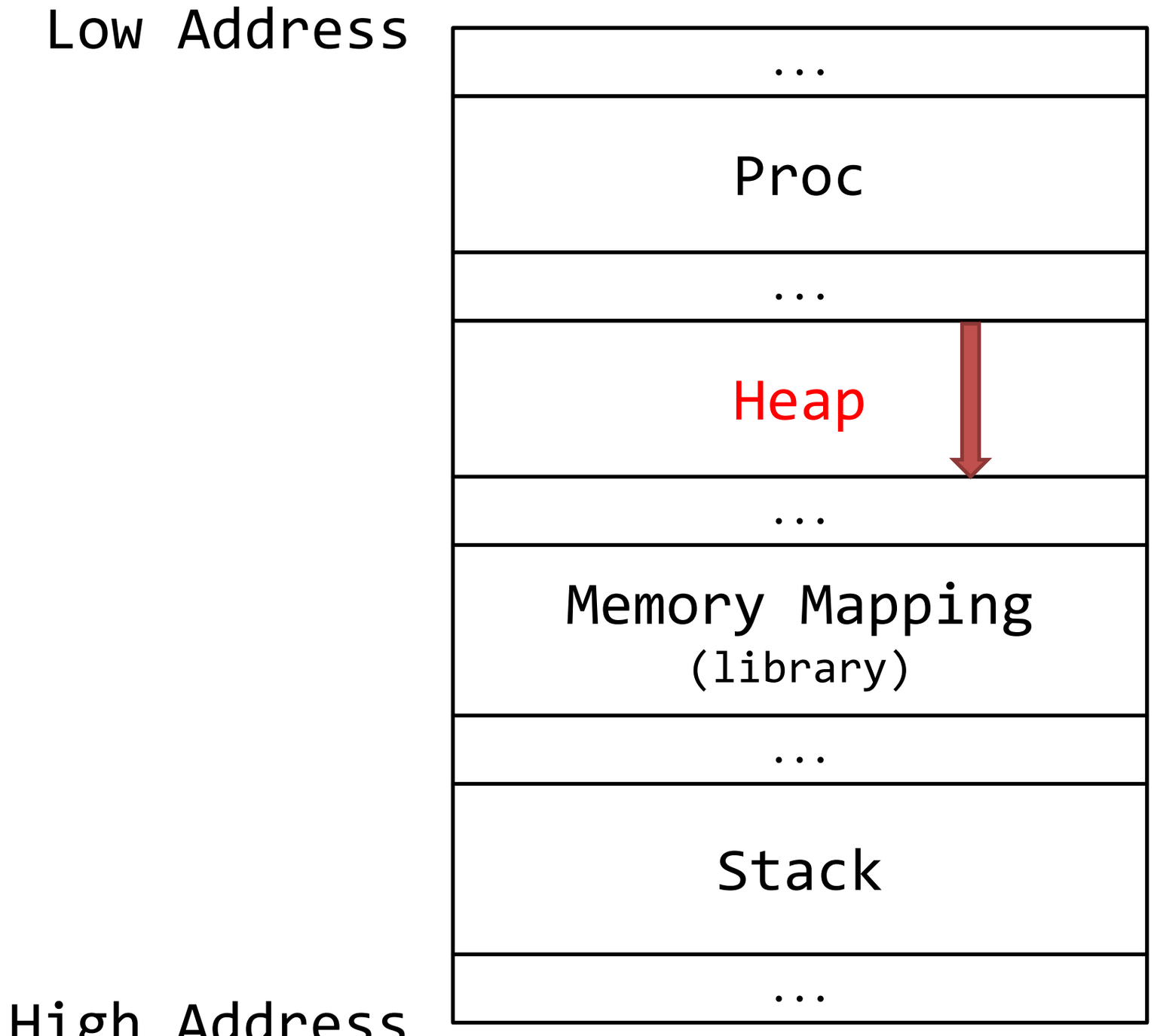
pwn 专题 03 – Heap



What is Heap?

堆主要用于存放程序运行过程中动态申请的内存段。在libc中，程序通过malloc函数申请内存，通过free函数释放内存

Where is Heap?



Recall: Memory Layout

Linux-x64下一般程序的内存布局（后续内容的背景均默认为64位系统）：

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start                End                  Offset               Perm Path
0x00556a63776000 0x00556a63777000 0x00000000000000 r-- /home/esifiel/demo
0x00556a63777000 0x00556a63778000 0x00000000001000 r-x /home/esifiel/demo
0x00556a63778000 0x00556a63779000 0x00000000002000 r-- /home/esifiel/demo
0x00556a63779000 0x00556a6377a000 0x00000000002000 r-- /home/esifiel/demo
0x00556a6377a000 0x00556a6377b000 0x00000000003000 rw- /home/esifiel/demo
0x00556a63a2e000 0x00556a63a4f000 0x00000000000000 rw- [heap]
0x007f648c72a000 0x007f648c74c000 0x00000000000000 r-- /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x007f648c74c000 0x007f648c8c4000 0x00000000022000 r-x /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x007f648c8c4000 0x007f648c912000 0x0000000019a000 r-- /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x007f648c912000 0x007f648c916000 0x000000001e7000 r-- /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x007f648c916000 0x007f648c918000 0x000000001eb000 rw- /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x007f648c918000 0x007f648c91e000 0x00000000000000 rw-
0x007f648c93d000 0x007f648c93e000 0x00000000000000 r-- /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x007f648c93e000 0x007f648c961000 0x00000000001000 r-x /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x007f648c961000 0x007f648c969000 0x00000000024000 r-- /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x007f648c96a000 0x007f648c96b000 0x0000000002c000 r-- /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x007f648c96b000 0x007f648c96c000 0x0000000002d000 rw- /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x007f648c96c000 0x007f648c96d000 0x00000000000000 rw-
0x007ffd565d3000 0x007ffd565f5000 0x00000000000000 rw- [stack]
0x007ffd565f8000 0x007ffd565fc000 0x00000000000000 r-- [vvar]
0x007ffd565fc000 0x007ffd565fe000 0x00000000000000 r-x [vdso]
0xffffffffff600000 0xffffffffff601000 0x00000000000000 --x [vsyscall]
```

在gef中查看内存布局的命令：**vmmap**

How to manage Heap?

不同的库可能有不同的堆管理实现，例如：

- ptmalloc2
- jemalloc
- libmalloc
- tcmalloc
- dlmalloc
- ...

本课程只涉及Linux中Glibc的堆管理器：ptmalloc2

Where is Heap from? (in ptmalloc2)

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    malloc(0x1000);

    for(int i = 0; i < 0x22; i++)
        malloc(0x1000);

    malloc(0x1000000);

    return 0;
}
```

before the first malloc
(no heap space)

```
Start              End                Offset            Perm Path
0x0055b23f675000 0x0055b23f676000 0x00000000000000 r-- /home/esifiel/demo
0x0055b23f676000 0x0055b23f677000 0x00000000001000 r-x /home/esifiel/demo
0x0055b23f677000 0x0055b23f678000 0x00000000002000 r-- /home/esifiel/demo
0x0055b23f678000 0x0055b23f679000 0x00000000002000 r-- /home/esifiel/demo
0x0055b23f679000 0x0055b23f67a000 0x00000000003000 rw- /home/esifiel/demo
0x007f290bdbe000 0x007f290bde0000 0x00000000000000 r-- /usr/lib/x86_64-lin
```

在glibc中，初始的堆空间由brk系统调用产生。如果malloc申请的空间超过了当前可用的空闲内存，glibc会继续使用brk系统调用扩展堆空间或使用mmap系统调用申请内存。

Where is Heap from? (in ptmalloc2)

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    malloc(0x1000);

    for(int i = 0; i < 0x22; i++)
        malloc(0x1000);

    malloc(0x1000000);

    return 0;
}
```

after the first malloc
(brk syscall called)

```
Start              End                Offset             Perm Path
0x0055b23f675000 0x0055b23f676000 0x00000000000000 r-- /home/esifiel/demo
0x0055b23f676000 0x0055b23f677000 0x00000000001000 r-x /home/esifiel/demo
0x0055b23f677000 0x0055b23f678000 0x00000000002000 r-- /home/esifiel/demo
0x0055b23f678000 0x0055b23f679000 0x00000000002000 r-- /home/esifiel/demo
0x0055b23f679000 0x0055b23f67a000 0x00000000003000 rw- /home/esifiel/demo
0x0055b24141f000 0x0055b241440000 0x00000000000000 rw- [heap]
0x007f290bdbe000 0x007f290bde0000 0x00000000000000 r-- /usr/lib/x86_64-lin
```

Where is Heap from? (in ptmalloc2)

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    malloc(0x1000);

    for(int i = 0; i < 0x22; i++)
        malloc(0x1000);

    malloc(0x1000000);

    return 0;
}
```

when used out
(brk called again)

```
Start              End                Offset            Perm Path
0x0055b23f675000 0x0055b23f676000 0x00000000000000 r-- /home/esifiel/demo
0x0055b23f676000 0x0055b23f677000 0x00000000001000 r-x /home/esifiel/demo
0x0055b23f677000 0x0055b23f678000 0x00000000002000 r-- /home/esifiel/demo
0x0055b23f678000 0x0055b23f679000 0x00000000002000 r-- /home/esifiel/demo
0x0055b23f679000 0x0055b23f67a000 0x00000000003000 rw- /home/esifiel/demo
0x0055b24141f000 0x0055b241461000 0x00000000000000 rw- [heap]
0x007f290bdbe000 0x007f290bde0000 0x00000000000000 r-- /usr/lib/x86_64-lin
```

Where is Heap from? (in ptmalloc2)

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    malloc(0x1000);

    for(int i = 0; i < 0x22; i++)
        malloc(0x1000);

    malloc(0x1000000);

    return 0;
}
```

malloc(0x100000)
(mmap syscall called)

```
Start              End                Offset            Perm Path
0x0055b23f675000 0x0055b23f676000 0x00000000000000 r-- /home/esifiel/demo
0x0055b23f676000 0x0055b23f677000 0x00000000001000 r-x /home/esifiel/demo
0x0055b23f677000 0x0055b23f678000 0x00000000002000 r-- /home/esifiel/demo
0x0055b23f678000 0x0055b23f679000 0x00000000002000 r-- /home/esifiel/demo
0x0055b23f679000 0x0055b23f67a000 0x00000000003000 rw- /home/esifiel/demo
0x0055b24141f000 0x0055b241461000 0x00000000000000 rw- [heap]
0x007f290adbd000 0x007f290bdbe000 0x00000000000000 rw-
0x007f290bdbe000 0x007f290bde0000 0x00000000000000 r-- /usr/lib/x86_64-lin
```

如果用户申请的内存过大，glibc会选择通过mmap系统调用创建匿名映射段供用户使用（该段位于libc上方）

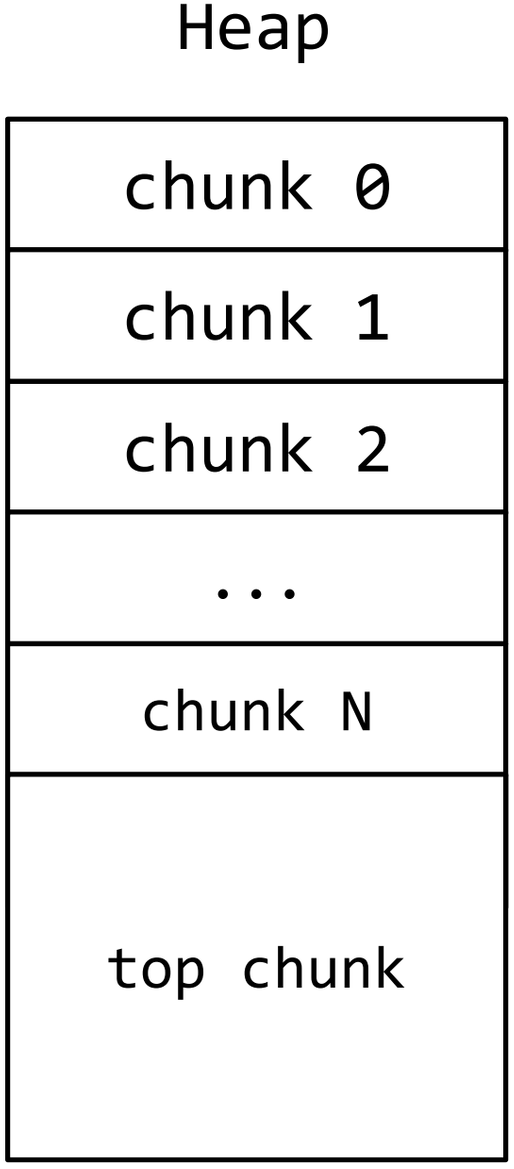
Basic Concept: chunk

chunk是内存分配的基本单位。

对于一个Linux的应用程序而言，chunk在堆中的的分布可以简单地如右图所示。

chunk的具体分类：

- allocated chunk：已分配的chunk
- free chunk：已释放的chunk
- top chunk：位于整个heap中最高地址处的未分配的内存
- last remainder chunk：一个chunk被切割后剩下的部分



What's inside chunk?

在glibc的[malloc.c]源码中描述了chunk内部的数据结构：

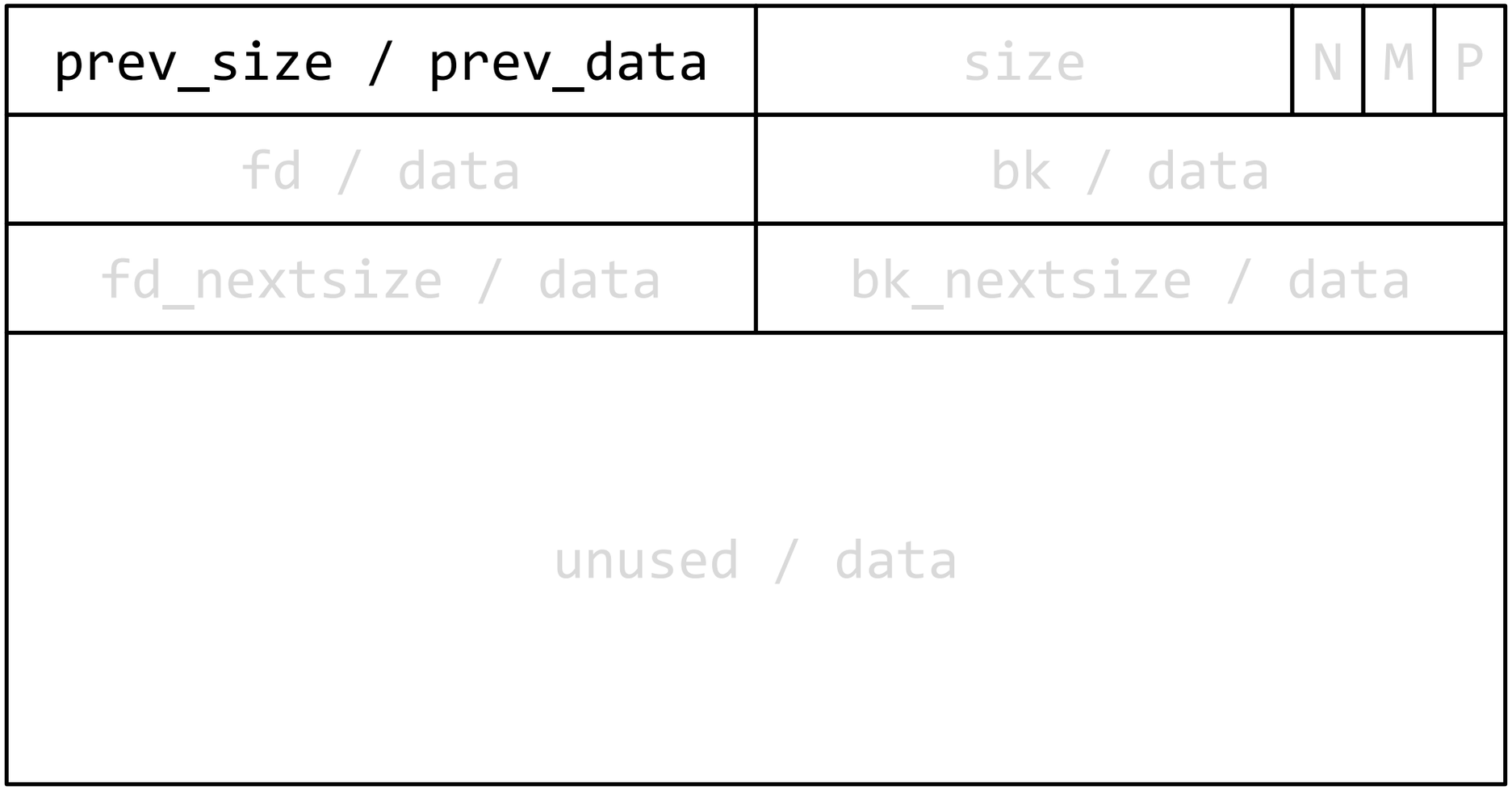
```
struct malloc_chunk {
    INTERNAL_SIZE_T      mchunk_prev_size; /* Size of previous chunk (if free). */
    INTERNAL_SIZE_T      mchunk_size;      /* Size in bytes, including overhead. */

    struct malloc_chunk* fd;               /* double links -- used only if free. */
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size. */
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
    struct malloc_chunk* bk_nextsize;
};
```

但是根据chunk的状态，并非每个成员变量都有意义。为了节省内存，glibc对某些字段做了复用。

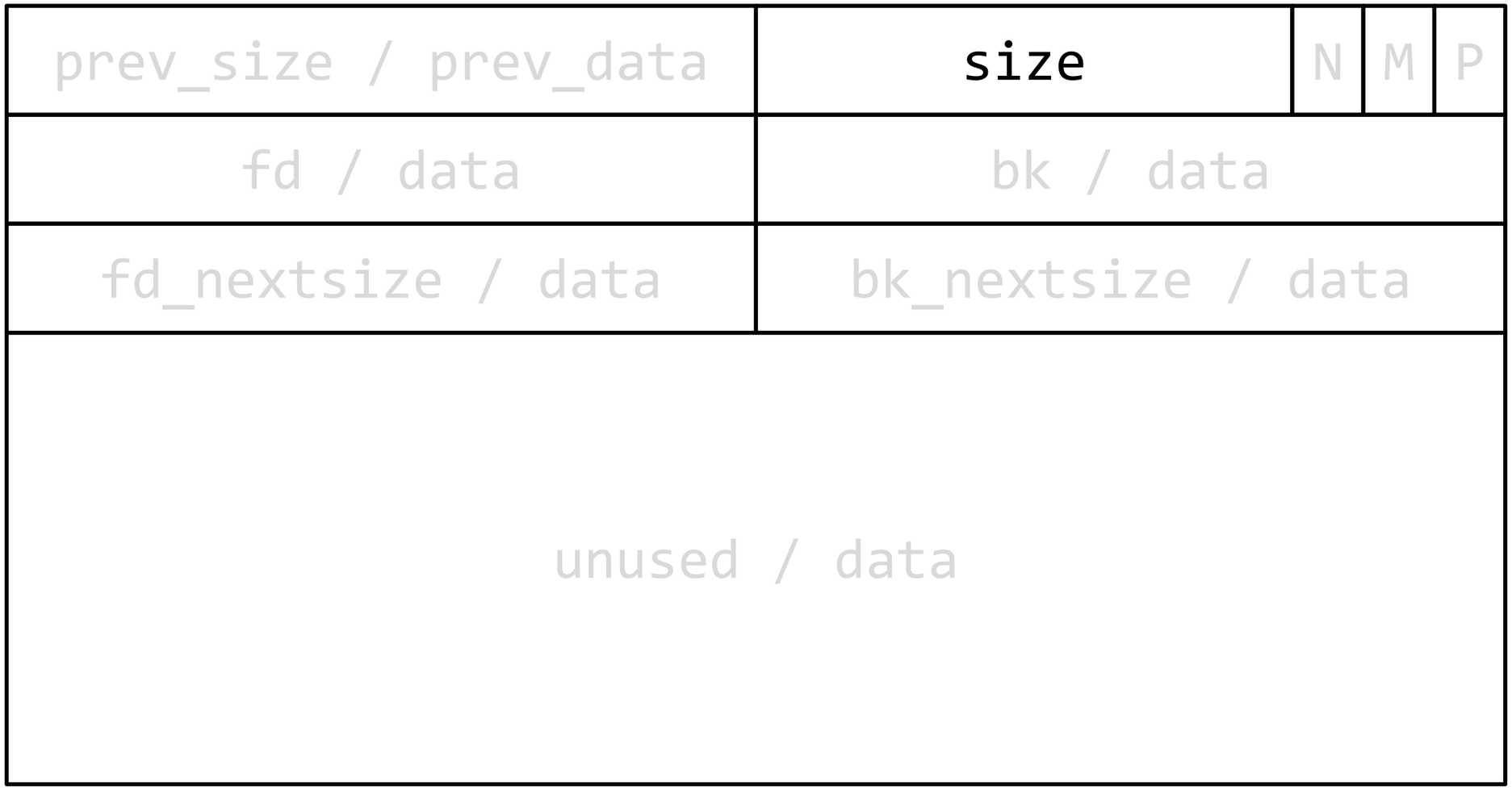
Data structure: chunk



prev_size / prev_data:

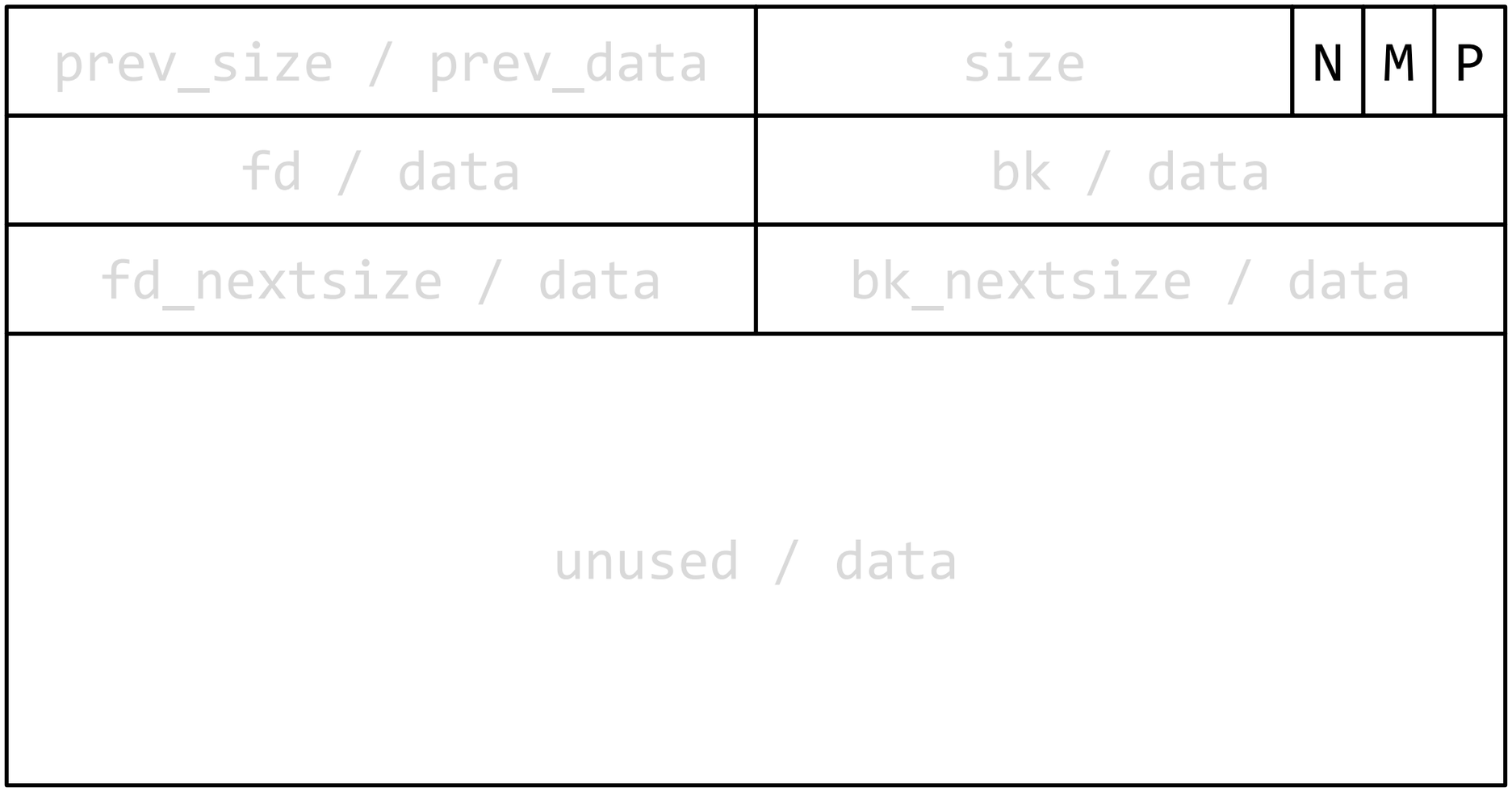
- 如果前一个chunk是allocated chunk(P=1)，则此字段属于前一个chunk可用的data部分
- 如果前一个chunk是free chunk(P=0)，则此字段表示前一个chunk的size(prev_size)

Data structure: chunk



size：当前chunk大小(size)

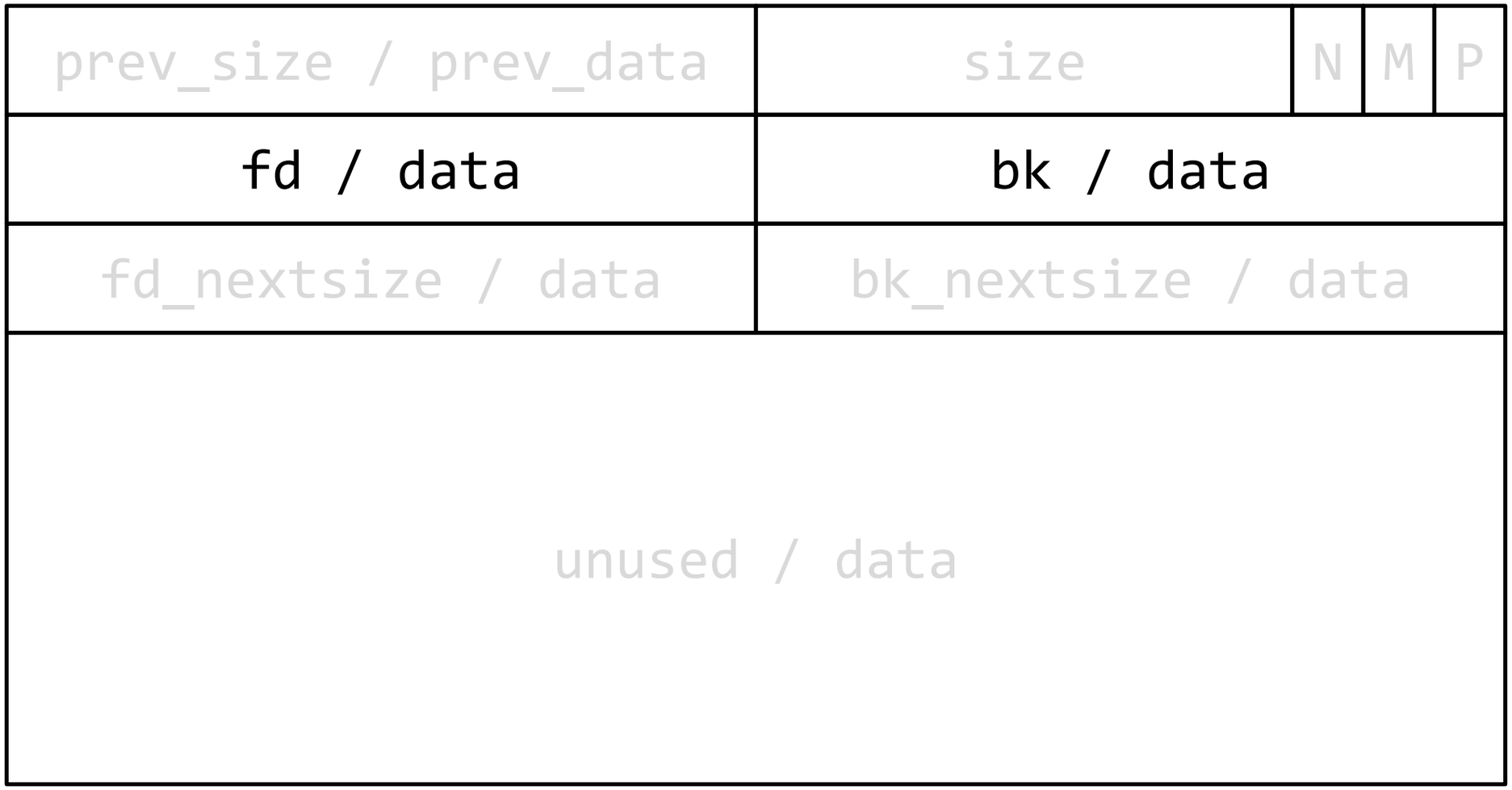
Data structure: chunk



标志位（size字段的低3bit）

- N: NON_MAIN_ARENA flag，表示chunk是否属于主线程
- M: IS_MMAPPED flag，表示是否由mmap分配
- P: PREV_INUSE flag，前一个chunk是否处于使用状态

Data structure: chunk

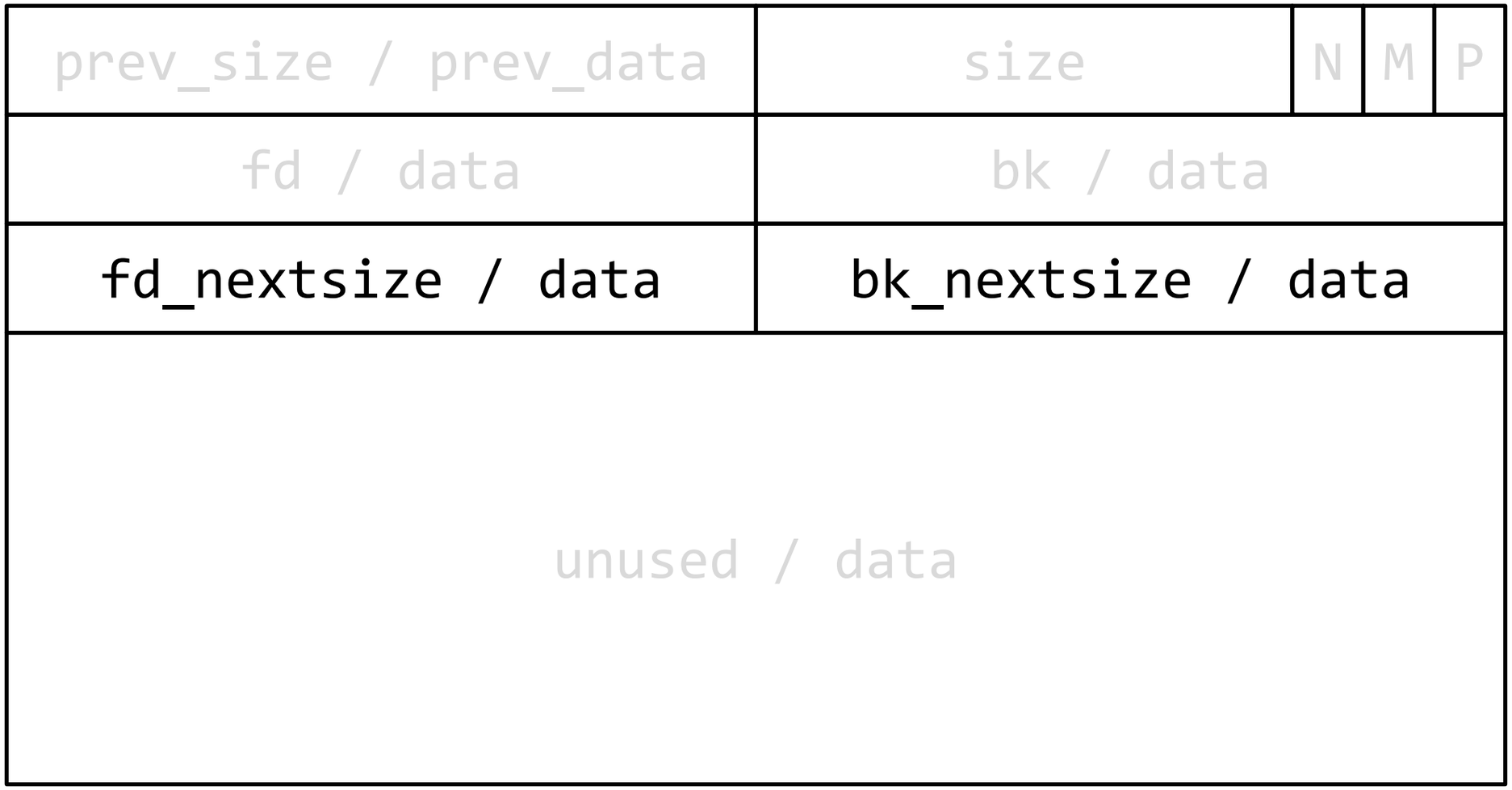


如果当前chunk已经被free到bin中,

- fd：指向bin中后一个空闲块的指针
- bk：指向bin中前一个空闲块的指针

（后一个和前一个均不一定是物理相邻的）

Data structure: chunk



如果当前chunk已经被free到large bin（后面马上会提到）中,

- fd_nextsize: 指向large bin中后一个与自己大小不同的chunk的指针
- bk_nextsize: 指向large bin中前一个与自己大小不同的chunk的指针

See in gdb

执行 `char *p = malloc(0x20);` `strcpy(p, "AAAAAAAA");` 之后发生了什么呢?

```
gef➤  heap chunks
Chunk(addr=0x55dfe621a010, size=0x290, flags=PREV_INUSE)
    [0x000055dfe621a010     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x55dfe621a2a0, size=0x30, flags=PREV_INUSE)
    [0x000055dfe621a2a0     41 41 41 41 41 41 41 41 00 00 00 00 00 00 00 00    AAAAAAAA........]
Chunk(addr=0x55dfe621a2d0, size=0x20d40, flags=PREV_INUSE)
    [0x000055dfe621a2d0     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x55dfe621a2d0, size=0x20d40, flags=PREV_INUSE)  ← top chunk
gef➤  heap chunk 0x55dfe621a2a0
Chunk(addr=0x55dfe621a2a0, size=0x30, flags=PREV_INUSE)
Chunk size: 48 (0x30)
Usable size: 40 (0x28)
Previous chunk size: 0 (0x0)
PREV_INUSE flag: On
IS_MMAPPED flag: Off
NON_MAIN_ARENA flag: Off

gef➤  x/8gx 0x55dfe621a290
0x55dfe621a290: 0x0000000000000000      0x0000000000000031
0x55dfe621a2a0: 0x4141414141414141      0x0000000000000000
0x55dfe621a2b0: 0x0000000000000000      0x0000000000000000
0x55dfe621a2c0: 0x0000000000000000      0x0000000000020d41
```

在gef中查看当前堆中所有chunk的命令：**heap chunks**
在gef中查看指定地址上的chunk的命令：**heap chunk <addr>**

休息一下

思考题：为什么malloc(0x20)分配了一个size为0x30的chunk呢?

```
gef➤  heap chunk 0x55dfe621a2a0
Chunk(addr=0x55dfe621a2a0, size=0x30, flags=PREV_INUSE)
Chunk size: 48 (0x30)
Usable size: 40 (0x28)
Previous chunk size: 0 (0x0)
PREV_INUSE flag: On
IS_MMAPPED flag: Off
NON_MAIN_ARENA flag: Off
```

Alignment

Q: 为什么malloc(0x20)分配了一个size为0x30的chunk呢?

A: 在内存中，堆块大小会按0x10字节对齐，chunk最小为0x20字节

在glibc中，对齐由[malloc.c]中的request2size宏实现。可以简单将该操作理解为下表中的映射，即：实际size = 请求的size+8后对应的下一个0x10对齐的值

请求大小	实际分配大小
0x00 ~ 0x18	0x20
0x19 ~ 0x28	0x30
0x29 ~ 0x38	0x40
...	...
0xE9 ~ 0xF8	0x100
...	...

Basic Concept: bin

bin是堆空闲块的管理结构，是由free chunk组成的链表。当allocated chunk被释放后，会放入bin中或者合并到top chunk中。bin的主要作用是加快分配速度

bin的具体分类：

- fast bin
- unsorted bin
- small bin
- large bin
- tcache bin

其中，

- fast bin、tcache bin按照LIFO(last in first out)单链表组织，采用头插法
- unsorted bin、small bin按照FIFO(first in first out)双链表组织，采用头插法
- large bin按照双链表组织，插入节点时会保证size从大到小排序

Data structure: arena

在开始介绍各个bin之前，还需要先了解一下arena结构体。

arena是用于管理堆信息的结构体，主线程的arena称为main_arena，main_arena保存在libc的数据段中。

在malloc.c中，arena结构体实现如右所示，其中保存了bins、top chunk等信息：

```
typedef struct malloc_chunk *mfastbinptr;
typedef struct malloc_chunk *mchunkptr;
struct malloc_state {
    /* Serialize access. */
    __libc_lock_define (, mutex);

    /* Flags (formerly in max_fast). */
    int flags;

    /* Set if the fastbin chunks contain recently inserted free blocks. */
    /* Note this is a bool but not all targets support atomics on booleans. */
    int have_fastchunks;

    /* 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. Access to this field is
       serialized by free_list_lock in arena.c. */
    struct malloc_state *next_free;

    /* Number of threads attached to this arena. 0 if the arena is on
       the free list. Access to this field is serialized by
       free_list_lock in arena.c. */
    INTERNAL_SIZE_T attached_threads;

    /* Memory allocated from the system in this arena. */
    INTERNAL_SIZE_T system_mem;
    INTERNAL_SIZE_T max_system_mem;
};
```

太复杂了？

我们只需要关注它俩就可以了：


```
/* Fastbins */
mfastbinptr fastbinsY[NFASTBINS];

/* Normal bins packed as described above */
mchunkptr bins[NBINS * 2 - 2];
```

- fastbinsY：fast bin的管理结构，用于存储不同size的fast bin链表

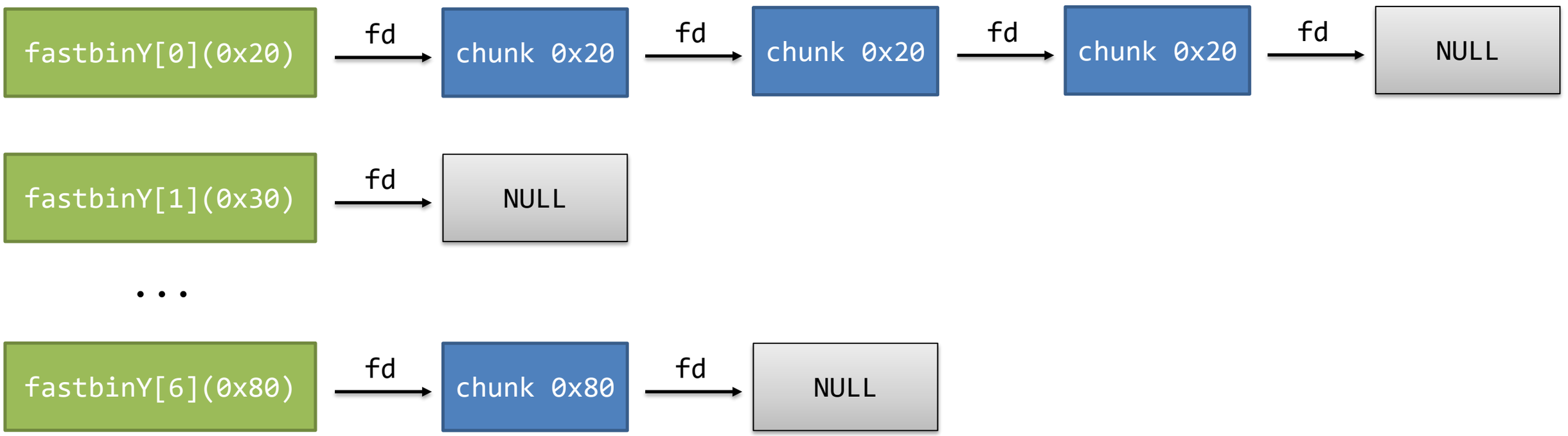
- bins：能够存放所有size范围的free chunk，共127个链表节点项，每个链表长度不限。
  - bin[0]为unsorted bin
  - bin[1] ~ bin[62]为small bin
  - bin[63] ~ bin[126]为large bin

  初始状态：bins数组的每个链表节点都形成自我闭环，表示双向链表为空。
  存取方式：FIFO(first in first out)，从头节点插入，从尾节点取出。
  注意：当一个chunk被free后放入bins中时，其next chunk的prev_inuse位会置0，且相应的size会被写入该chunk的prev_size

Data structure: fast bin

fast bin的组织形式：将小chunk（大小位于[0x20，0x80]范围中）单独管理，以size为单位，以单向链表的形式组织起来，链表长度不限。链表通过fd指针链接，在fast bin中，chunk的bk字段是未使用的。

注意：为了快速分配小内存，当一个chunk被free到fast bin中时，其物理相邻的下一个chunk的prev_inuse位并不会置0，以防止释放时对fast bin进行合并

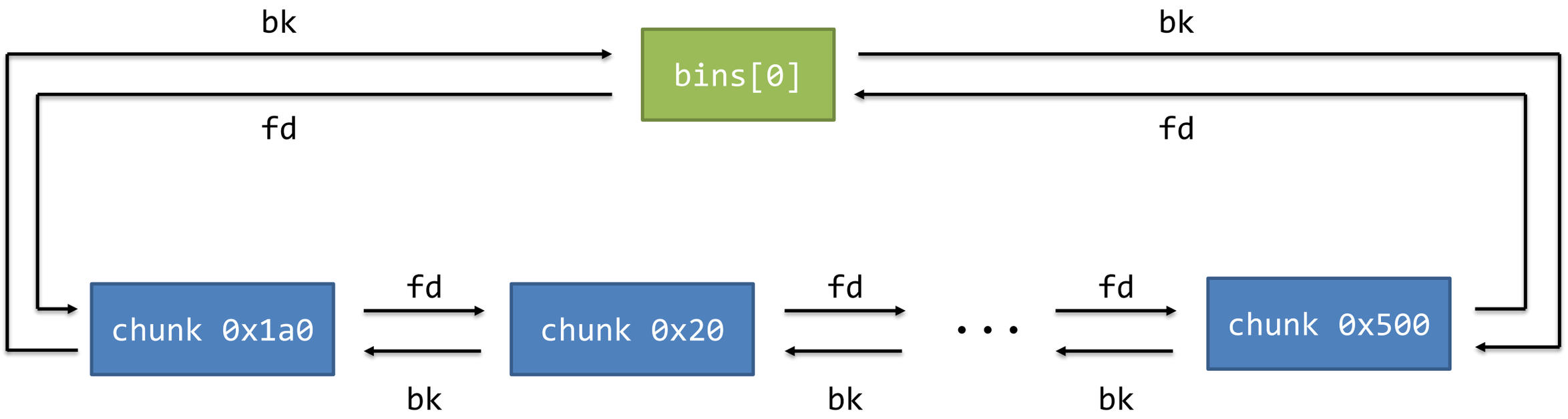


Data structure: unsorted bin

顾名思义，unsorted bin中free chunk的大小没有顺序，任何size的chunk都可能被放入到这个bin中。

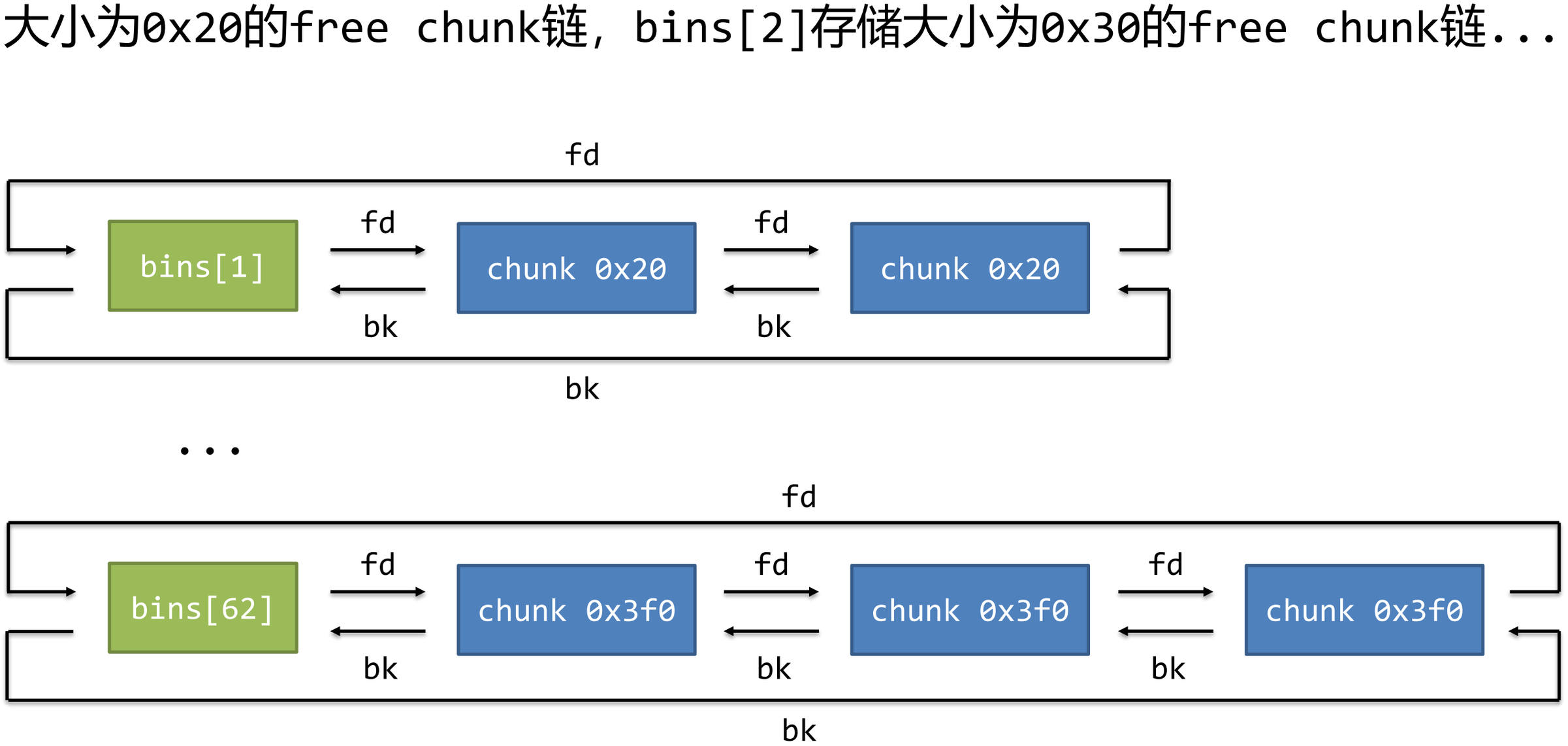
unsorted bin主要用于存放刚被释放的堆块以及大堆块分配后剩余的堆块（具体会在后面malloc内部实现中提到）。在实践中，一个被释放的chunk常常很快就会被重新使用，所以将其先加入unsorted bin可以加快分配的速度。

unsorted bin使用main_arena的bins[0]:



Data structure: small bin

small bin使用main_arena的bins[1] ~ bins[62]，管理大小在[0x20, 0x400)的free chunk，bins中每个entry对应一种大小，例如bins[1]存储大小为0x20的free chunk链，bins[2]存储大小为0x30的free chunk链...



Data structure: large bin

large bin使用main_arena的bins[63] ~ bins[127]，管理大小在small bin范围之外的所有free chunk

不同于small bin的bin和size一一对应，在large bin中，一个bin对应多个size，且：

- 同一个bin中的chunk，按照size从大到小组织。如果size相同，则后free会插在该size子链的末尾，而不是像unsorted bin和small bin一样新的free chunk总是插在开头
- 同一个bin中不同size子链的头节点会通过fd_nextsize和bk_nextsize指针组织起来，同样是循环双向链表的结构

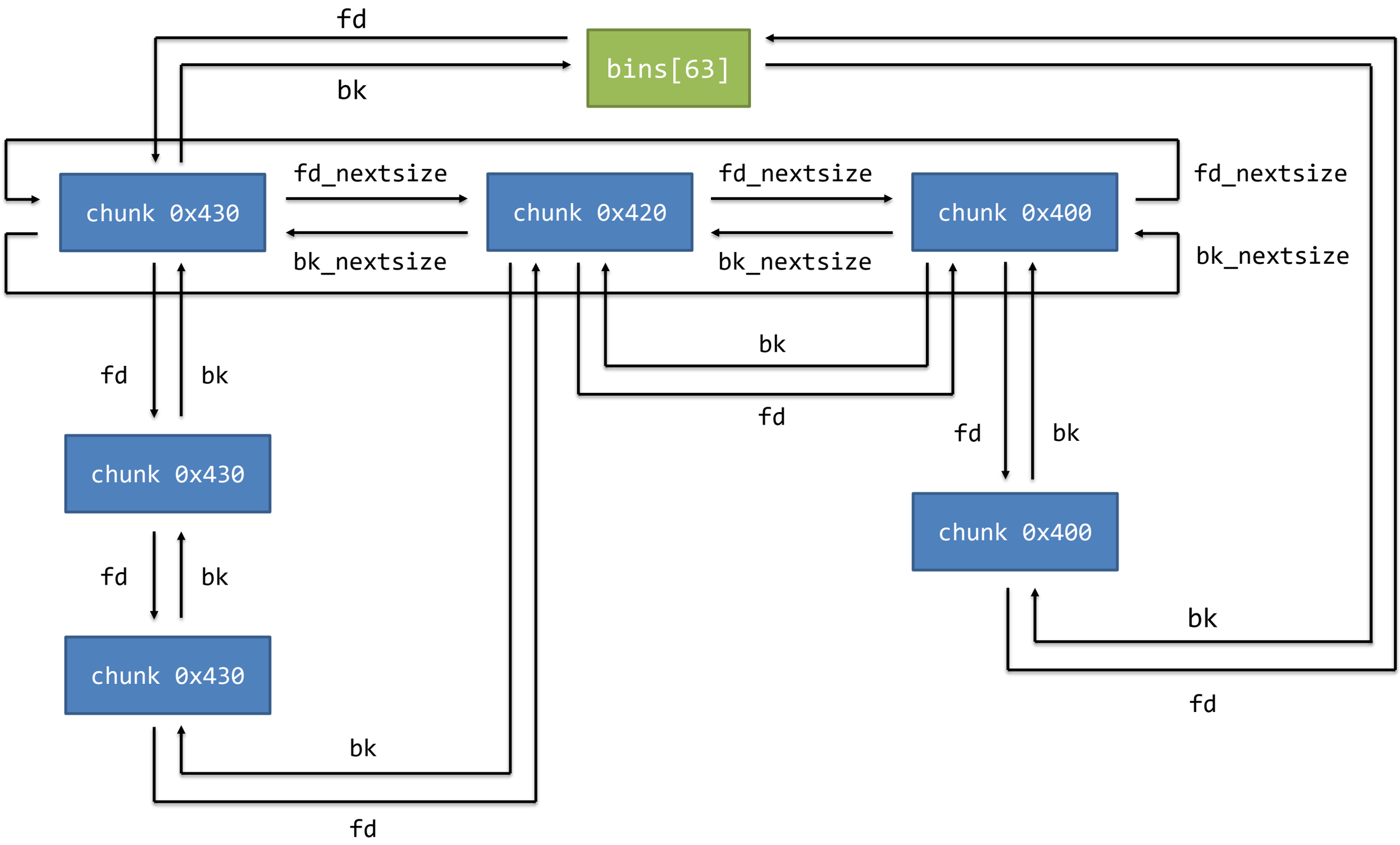
large bin的每个bin所能容纳的chunk大小按顺序排成等差数列，公差d如下：

```
32 bins with d = 0x40
16 bins with d = 0x200
8  bins with d = 0x1000
4  bins with d = 0x8000
2  bins with d = 0x40000
```

最后一个bin包含其他所有大小的free chunk

从large bin中取chunk时，会按照best-fit的方式取，且会优先选择不在nextsize链表中的chunk，以提高效率。

Data structure: large bin



Data structure: tcache bin

从glibc-2.26开始，ptmalloc2引入了tcache结构，目的是为了提升堆管理的性能。tcache全称Thread Local Caching，顾名思义，它为每个线程创建一个缓存，用于管理一些小的free chunk。每个线程默认使用64个单链表结构的bins，每个bins最多存放7个chunk，管理行为可类比fast bin，同样是LIFO

tcache的管理结构不在main_arena中，而是初始化在heap最开始的位置，即glibc会动态分配一个特殊的chunk，以作为tcache struct使用，这就是为什么在使用**heap chunks**查看堆块时可能会看见第一个chunk的大小是0x250（或0x290）。在libc-2.30之前，tcache struct数据结构如下（libc-2.30及之后的版本中，count的类型变为uint16_t）：

```
typedef struct tcache_perthread_struct
{
    char counts[TCACHE_MAX_BINS];
    tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

- entries数组可以类比于fastbinsY，每个bin都是一个单向链表。
- counts数组对相应的tcache bin中的chunk数量进行记录

See in gdb

看看各种bin都长啥样

在gef中查看当前bins状态的命令：**heap bins**

```
gef➤  heap bins
──────────────────────────────── Tcachebins for thread 1 ────────────────────────────────
All tcachebins are empty
──────────────────────────────── Fastbins for arena at 0x7f1797952b80 ────────────────────────────────
Fastbins[idx=0, size=0x20] 0x00
Fastbins[idx=1, size=0x30] 0x00
Fastbins[idx=2, size=0x40] 0x00
Fastbins[idx=3, size=0x50] 0x00
Fastbins[idx=4, size=0x60] 0x00
Fastbins[idx=5, size=0x70] 0x00
Fastbins[idx=6, size=0x80] 0x00
──────────────────────────────── Unsorted Bin for arena at 0x7f1797952b80 ────────────────────────────────
[+] Found 0 chunks in unsorted bin.
──────────────────────────────── Small Bins for arena at 0x7f1797952b80 ────────────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────────────────── Large Bins for arena at 0x7f1797952b80 ────────────────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

See in gdb

```
gef➤  heap bins
──────────────────────────────────── Tcachebins for thread 1 ────────────────────────────────────
Tcachebins[idx=1, size=0x30, count=1] ← Chunk(addr=0x5634cc12a2a0, size=0x30, flags=PREV_INUSE)
Tcachebins[idx=2, size=0x40, count=7] ← Chunk(addr=0x5634cc12a450, size=0x40, flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12a410,
size=0x40, flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12a3d0, size=0x40, flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12a390, size=0x4
0, flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12a350, size=0x40, flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12a310, size=0x40, flags
=PREV_INUSE)  ← Chunk(addr=0x5634cc12a2d0, size=0x40, flags=PREV_INUSE)
Tcachebins[idx=4, size=0x60, count=2] ← Chunk(addr=0x5634cc12db20, size=0x60, flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12dac0,
size=0x60, flags=PREV_INUSE)
Tcachebins[idx=26, size=0x1c0, count=7] ← Chunk(addr=0x5634cc12afd0, size=0x1c0, flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12ae1
0, size=0x1c0, flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12ac50, size=0x1c0, flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12aa90, siz
e=0x1c0, flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12a8d0, size=0x1c0, flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12a710, size=0x1c
0, flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12a550, size=0x1c0, flags=PREV_INUSE)
──────────────────────────────────── Fastbins for arena at 0x7fdcf6611b80 ────────────────────────────────────
Fastbins[idx=0, size=0x20] 0x00
Fastbins[idx=1, size=0x30] 0x00
Fastbins[idx=2, size=0x40]  ← Chunk(addr=0x5634cc12a510, size=0x40, flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12a4d0, size=0x40,
 flags=PREV_INUSE)  ← Chunk(addr=0x5634cc12a490, size=0x40, flags=PREV_INUSE)
Fastbins[idx=3, size=0x50] 0x00
Fastbins[idx=4, size=0x60] 0x00
Fastbins[idx=5, size=0x70] 0x00
Fastbins[idx=6, size=0x80] 0x00
──────────────────────────────────── Unsorted Bin for arena at 0x7fdcf6611b80 ────────────────────────────────────
[+] unsorted_bins[0]: fw=0x5634cc12c7c0, bk=0x5634cc12b180
 →  Chunk(addr=0x5634cc12c7d0, size=0x1c0, flags=PREV_INUSE)  →  Chunk(addr=0x5634cc12b190, size=0x510, flags=PREV_INUSE)
[+] Found 2 chunks in unsorted bin.
```

```
──────────────────────────────────── Small Bins for arena at 0x7efeaaffcb80 ────────────────────────────────────
[+] small_bins[26]: fw=0x55ac7fe6c190, bk=0x55ac7fe6bfb0
 →  Chunk(addr=0x55ac7fe6c1a0, size=0x1b0, flags=PREV_INUSE)  →  Chunk(addr=0x55ac7fe6bfc0, size=0x1b0, flags=PREV_INUSE)
[+] Found 2 chunks in 1 small non-empty bins.
──────────────────────────────────── Large Bins for arena at 0x7efeaaffcb80 ────────────────────────────────────
[+] large_bins[67]: fw=0x55ac7fe6c550, bk=0x55ac7fe6c550
 →  Chunk(addr=0x55ac7fe6c560, size=0x510, flags=PREV_INUSE)
[+] large_bins[100]: fw=0x55ac7fe6ded0, bk=0x55ac7fe6ca90
 →  Chunk(addr=0x55ac7fe6dee0, size=0x1420, flags=PREV_INUSE)  →  Chunk(addr=0x55ac7fe6caa0, size=0x1410, flags=PREV_INUSE)
[+] large_bins[106]: fw=0x55ac7fe6f320, bk=0x55ac7fe6f320
 →  Chunk(addr=0x55ac7fe6f330, size=0x2010, flags=PREV_INUSE)
[+] Found 4 chunks in 3 large non-empty bins.
```

malloc

malloc的具体实现可以查看[malloc.c]中_int_malloc函数，大致流程如下：

1. 将大小按规则对齐，得到实际要分配的大小size
2. 检查size是否符合**tcache bin**的大小。如果是，检查对应size的entry是否有free chunk。如果有，则分配返回
3. 检查size是否符合**fast bin**的大小。如果是，检查对应size的entry是否有free chunk。如果有，则分配返回
4. 循环遍历**unsorted bin**，寻找可用的free chunk
   - 如果遍历到的free chunk size正好和所需size相等，则分配返回
   - 如果遍历到的free chunk size和所需size不等，则将其从双链表中解链(unlink)，插入到对应大小的bins中
5. 根据size，以best-fit的方式，找到相应的**small bin**或者**large bin**
   - 对于small bin，如果size正好合适，那么unlink之后，直接将该chunk返回给用户；否则进行切割，剩下的部分重新插入到unsorted bin中。
   - 对于large bin，由于一个bin通常对应几个size，那么根据fd_nextsize的顺序，以size从大到小的顺序遍历chunk，同样采取best-fit的方式寻找合适的chunk，后续行为与small bin类似。
6. 使用**top chunk**，将top chunk进行切割：
   - 如果top chunk size足够，则将切割下来的部分返回，剩下的部分继续作为top chunk
   - 如果top chunk size不够，则需要通过sysmalloc申请更多的堆空间

free

free的具体实现可以查看malloc.c中_int_free函数，大致流程如下：

1. 如果free chunk的size属于**tcache**范围内，且对应大小的tcache bin没有满，则插入到相应的tcache bin中去
2. 如果free chunk的size属于**fast bin**范围内，且对应大小的tcache bin满了，则插入到fastbin中去
3. 如果上述条件均不满足，则通过该chunk的prev_inuse标志位检查是否可以前后向合并：
   - 如果可以合并，则将需要被合并的chunk先unlink下来，合并成一个更大的chunk后再插入到unsorted bin中（或合并到top chunk里面）
   - 如果不可以合并，则将该chunk直接插入到unsorted bin中
4. free chunk是mmap的chunk，那么调用munmap直接返回给系统

休息一下

思考题：根据目前所学的内容，尝试使用这两个函数构造出符合要求的bin
（-表示chunk之间的链接，十六进制值表示chunk的大小）：

(1) tcache: 0x50 - 0x50 - 0x50
(2) fast bin: 0x50 - 0x50 - 0x50 (libc-version >= 2.26)
(3) unsorted bin: 0x140 – 0x500 – 0x6a0
(4) small bin: 0x210 – 0x210 – 0x210
(5) large bin: 0x1410 – 0x1420

Then…

讲完了以上这些概念
终于要开始讲利用了！

Heap Exploit

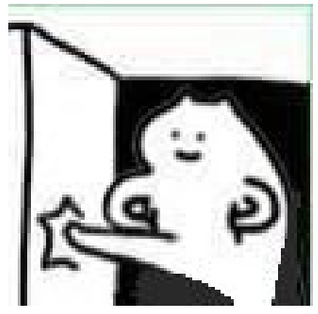
一般情况下，堆题都是保护全开的

```
[*] '/home/esifiel/demo'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

导致很难对栈空间或got表进行攻击，这时应该如何劫持控制流呢？

Still…

和覆写got表的原理类似，我们的主要目标仍然是劫持函数指针



malloc、free、realloc三个函数有自己的hook函数，以函数指针的形式存放在libc的数据段中，分别叫做__malloc_hook、__free_hook、__realloc_hook。在libc-2.34之前，如果对应hook函数非空，则这三个函数会转为执行hook函数，而非原来的逻辑

以malloc函数为例：

```
void *
__libc_malloc (size_t bytes)
{
    mstate ar_ptr;
    void *victim;

    _Static_assert (PTRDIFF_MAX <= SIZE_MAX / 2,
                    "PTRDIFF_MAX is not more than half of SIZE_MAX");

    void *(*hook) (size_t, const void *)
      = atomic_forced_read (__malloc_hook);

    if (__builtin_expect (hook != NULL, 0))
        return (*hook)(bytes, RETURN_ADDRESS (0));
...
```

Still…

值得一提的是，这三个hook函数中，攻击__free_hook是最稳定的

free函数接受一个指针参数，而system函数也接受一个指针参数，如果我们已经将__free_hook劫持为system，则触发时只需要在一个chunk中写上/bin/sh并free该chunk，就可以稳定执行system(“/bin/sh”)

对于__malloc_hook来说，由于malloc的参数是一个size_t类型的值，我们一般不能将其劫持为system，除非我们能将一个地址这么大的值作为size传给malloc，但是这在一个程序中通常是无法做到的。（一般是劫持为one gadget）

How?

传统堆题以菜单题为典型，此类程序通常会提供add、show、edit、delete四个功能，并会打印一个菜单供用户选择。我们需要利用这几个功能中的漏洞完成任意地址读写并get shell

任意地址读：leak libc，获取system等函数的地址
任意地址写：将system等函数的地址写入hook函数指针
get shell：调用malloc或free函数，从而调用hook函数劫持控制流

常见的程序错误：

- 堆溢出 heap overflow：

```
char *p = malloc(0x20);
read(0, p, 0x100);
```

- UAF(Use After Free)：

```
char *p = malloc(0x20);
free(p);
puts(p);
```

- double free：

```
char *p = malloc(0x20);
free(p);
free(p);
```

Exploit: UAF

UAF即Use after free，如果堆指针在释放后未被置空，则会形成dangling pointer，当下次访问该指针时，仍然能够访问拿到原指针所指向的堆数据，造成信息泄露或信息修改

UAF – 信息泄露

想想前面提过什么?

- main_arena保存在libc的数据段中
- bins中的chunk有指向对应entry的指针

So...

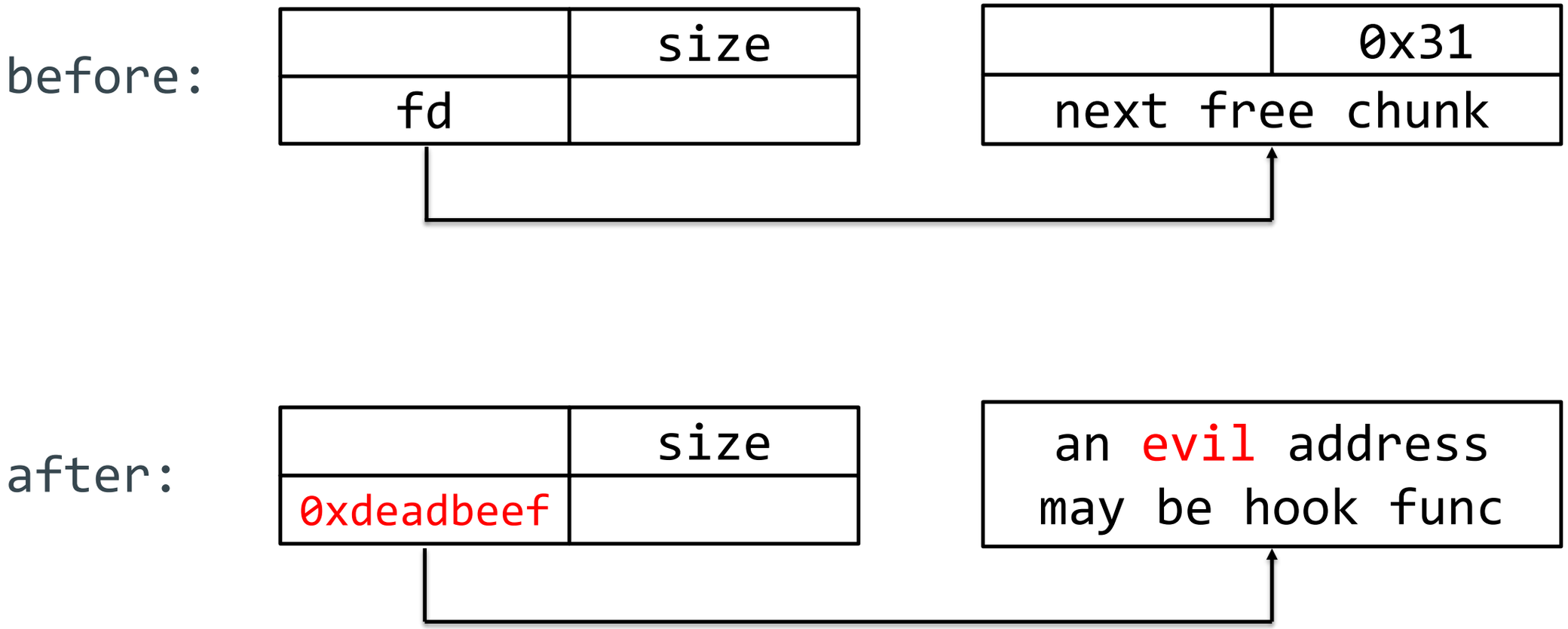
通常情况下，如果我们能够进行UAF，leak libc地址最快的方法就是打印一个unsorted bin chunk的fd指针，例如:

```
add(0, 0x500) # unsorted bin size chunk
add(1, 0x20)  # separate with top chunk
delete(0)     # get an unsorted bin chunk
show(0)       # uaf leak main_arena
```

这样一个操作序列将会打印出main_arena的地址，这个地址和libc基地址的偏移是固定的，于是我们能够计算出libc基地址

UAF – 信息修改

如果我们可以UAF一个free chunk，那么其中的fd、bk等指针就有可能被我们所控制。
如果我们劫持了fast bin或tcache的fd指针，我们就可以实现任意地址分配（不考虑malloc检查的情况下）



UAF – 信息修改

在从fast bin中malloc chunk时，glibc会有这样一个检查：

```
size_t victim_idx = fastbin_index (chunksize (victim));
if (__builtin_expect (victim_idx != idx, 0))
    malloc_printerr ("malloc(): memory corruption (fast)");
```

这个检查会要求该chunk的size必须在fast bin管理范围内，但并不要求对齐，因此通常我们可以利用某些地址上原有的数据当做size来绕过这个检查

相比之下，tcache并不检查size，即使伪造的fd指向的地方对应的chunk size为0，也可以正常申请出来

Exploit: double free

double free是一种特殊的UAF，针对的是能够对同一指针进行多次释放的函数。多次释放能够使堆块发生重叠，在此之后申请的堆块可能会同时存在于bin中。

fastbin double free

首先需要注意一个事情，glibc对fastbin double free是有检查的

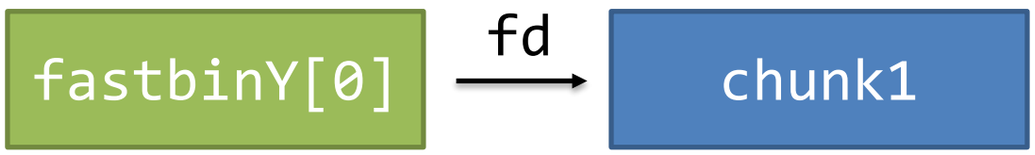
```
if (__builtin_expect (old == p, 0))
    malloc_printerr ("double free or corruption (fasttop)");
```

这个检查的意思是不允许fastbin的头chunk和当前要free的chunk相同。因此我们只需要在两次free同一chunk的操作之间插入一个free其他chunk的操作就可以绕过这个检查。

fastbin double free

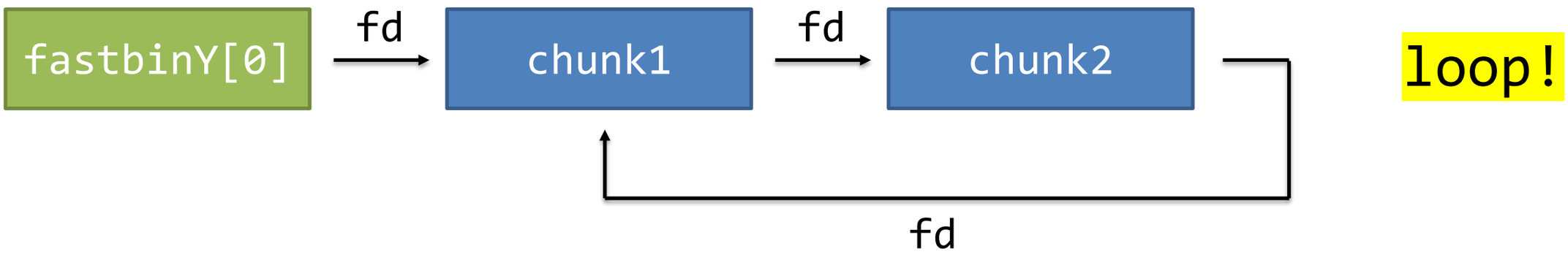
接下来看看fastbin double free的示意图

1. free(chunk1)



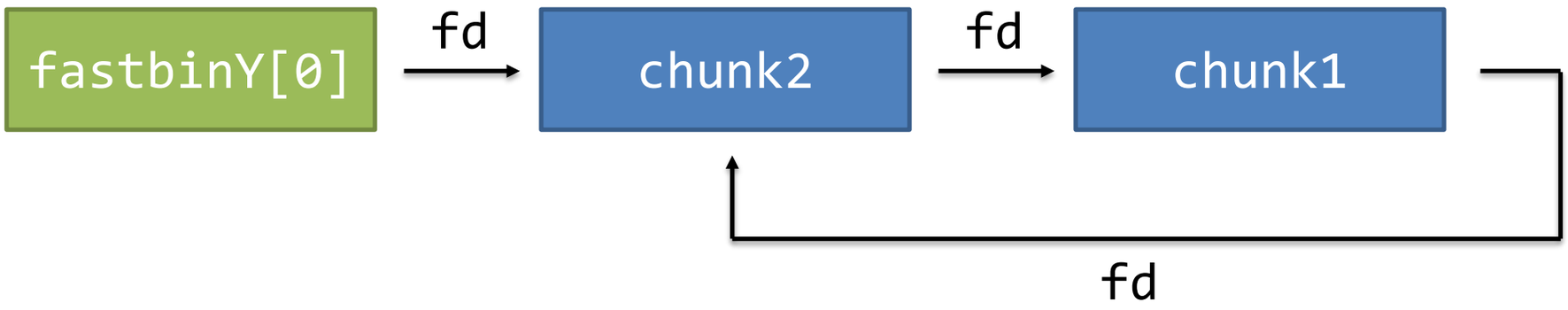
2. free(chunk2)

3. free(chunk1)

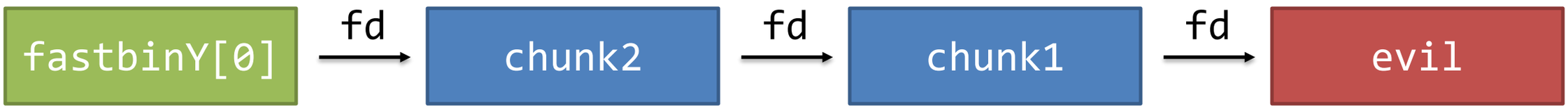


fastbin double free

4. chunk1 = malloc(0x10), but chunk1 still in fastbin!



5. memcpy(chunk1, “\xef\xbe\xad\xde”)



6. malloc(0x10), malloc(0x10), evil = malloc(0x10)

经过上述步骤，我们通过malloc分配到了一个我们想要的地址，从而可以进一步完成利用

tcache double free

tcache为了提升速度，牺牲了安全性，相关检查很少，对于利用而言比fastbin来得更为方便。

tcache在Ubuntu Glibc 2.27-3ubuntu1.2及以前没有double free检查，可以连续free同一个chunk，不需要像fastbin利用那样中间插入一个其他chunk

但需要注意的是，从tcache中malloc chunk时，该entry的count需要大于0，所以也不能单纯省略掉fastbin double free中free其他chunk的操作，而是应该先free一个相同大小的chunk使得count足够把我们劫持的fd给分配出来

tcache double free

但是在此glibc版本之后tcache添加了相应的check逻辑，tcache chunk结构中多了一个key成员：

```
typedef struct tcache_entry
{
    struct tcache_entry *next;
    /* This field exists to detect double frees. */
    struct tcache_perthread_struct *key;
} tcache_entry;
```

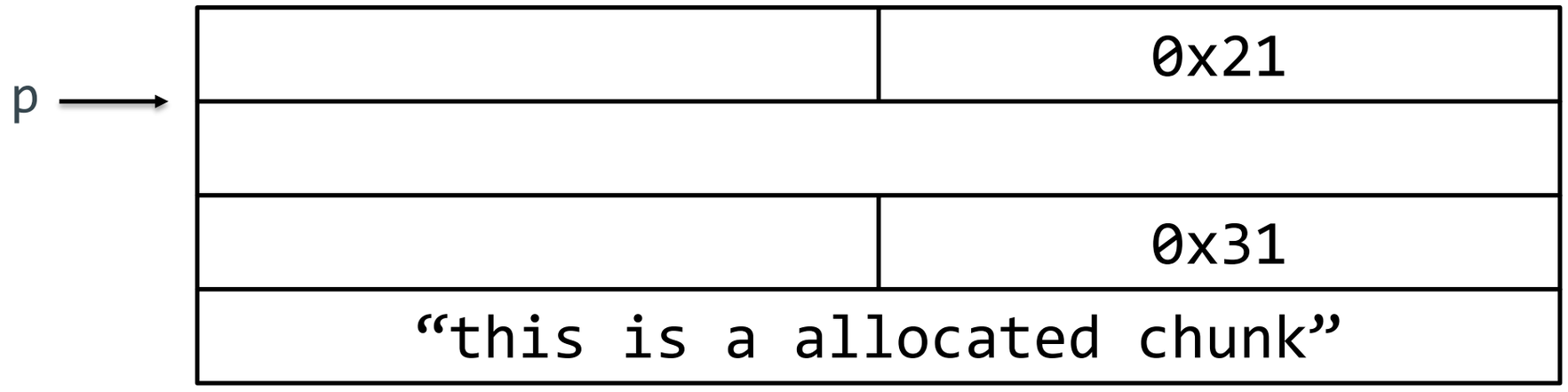
在某一个chunk被free放入tcache bin中时，其key成员会被置为tcache_perthread_struct的地址。如果一个chunk被free的时候，检查到其key正好是tcache_perthread_struct的地址且该chunk在tcache bin的链表中，那么就会被check到double free，从而abort。

在这种机制下，如果没有其他方法能够修改key，基本上是不可能单靠tcache来完成bypass了。

Exploit: heap overflow

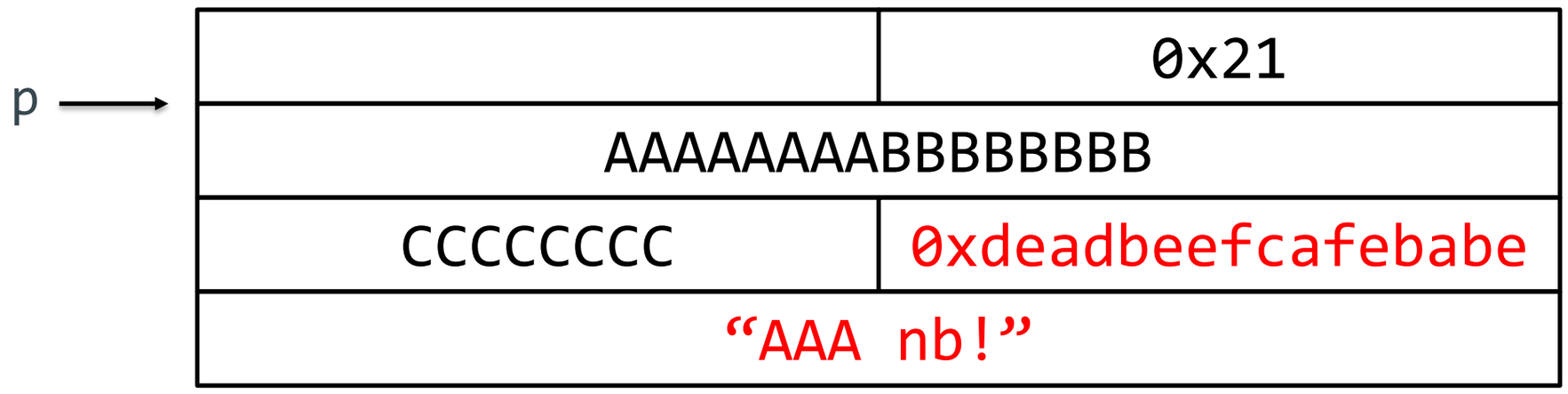
在溢出长度比较长的情况下，我们同样可以劫持free chunk的fd等指针，或是程序中位于堆上的某些结构体中的数据

before:

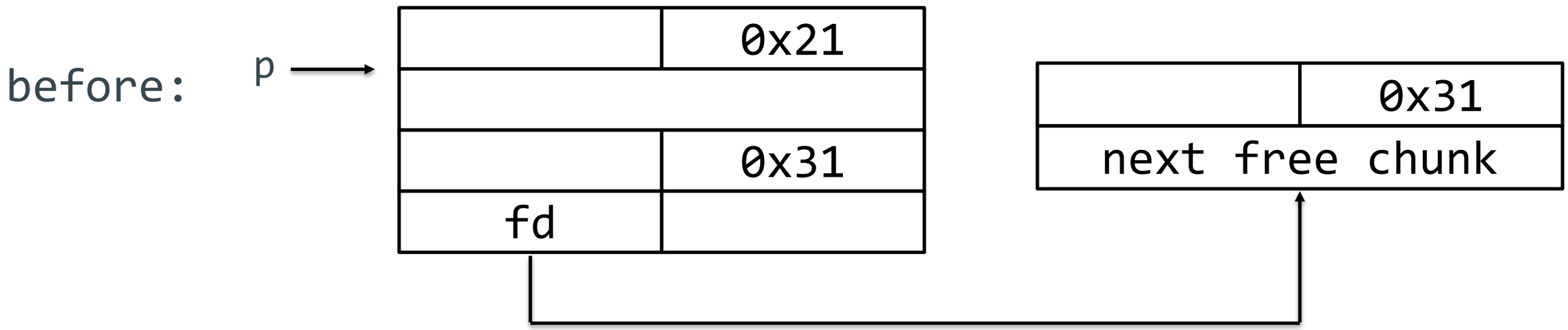



```
memcpy(p,
  "AAAAAAAABBBBBBBBCCCCCCCC\xbe\xba\xfe\xca\xef\xbe\xed\xdeAAA nb!");
```

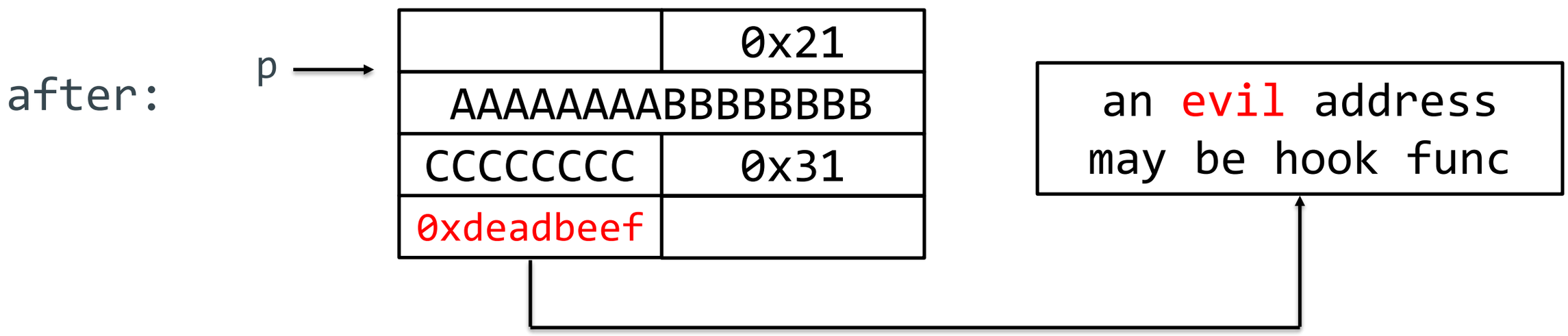
after:



Exploit: heap overflow




```
memcpy(p,
   "AAAAAAAABBBBBBBBCCCCCCCC“
   "\x31\x00\x00\x00\x00\x00\x00\x00")
   "\xef\xbe\xed\xde\x00\x00\x00\x00");
```

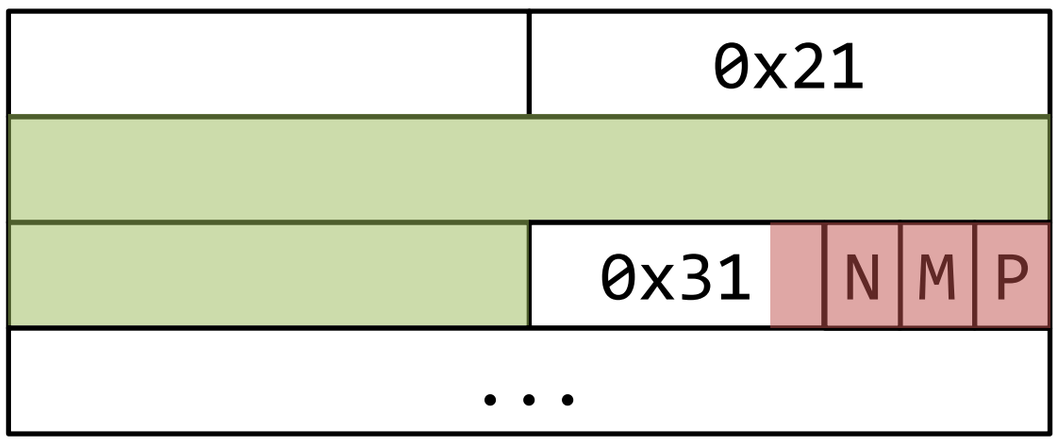


What if overflow only 1 byte?

只溢出了1字节的漏洞称为off-by-one

off-by-one通常是由循环边界错误造成的，或者是由于字符串处理函数如strcpy等在buffer末尾填了额外的0，此时也称为off-by-null

回想一下chunk的结构，off-by-one刚好可以覆盖下一堆块size的低字节，其中包括了三个标志位



So what?

只能溢出一字节有啥用?


```
char *p = malloc(0x10);
char *q = malloc(0xf8);
memcpy(p, "AAAAAAAABBBBBBBBCCCCCCCC\xf1", 0x19);
```

	0x21
	0x101
...	...

	0x21
AAAAAAAA	BBBBBBBB
CCCCCCCC	0x1f1
...	...

下一个chunk的size可以在一定范围内被改大或者改小

只能溢出一字节有啥用?

```
char *p = malloc(0x10);
char *q = malloc(0xf8);
strcpy(p, "AAAAAAAABBBBBBBBCCCCCCCC");
```

	0x21
	0x101
...	...

	0x21
AAAAAAAA	BBBBBBBB
CCCCCCCC	0x100
...	...

控制了标志位，prev_in_use位被置0后，前一个chunk会被认为是free chunk，从而在free其前后chunk时可能会执行合并操作

Exploit: Unlink Attack

在合并空闲块时，会对需要合并的块先执行unlink操作。unlink就是把一个双向链表中的空闲块拿出来。

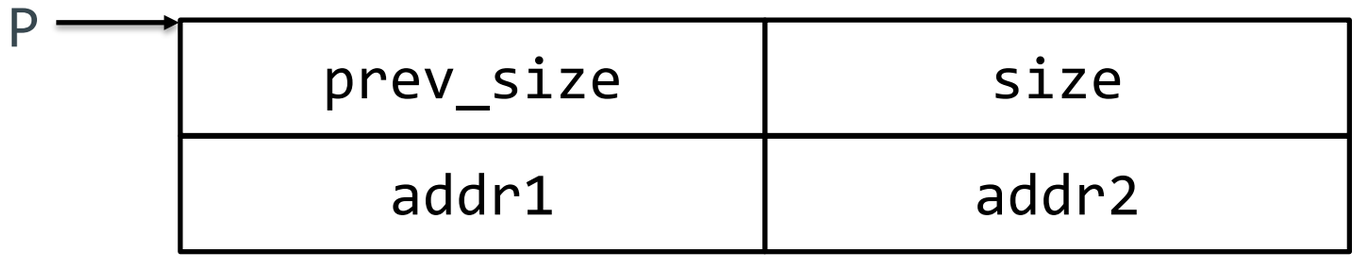
经典的双向链表解链操作：

```
FD = P->fd;
BK = P->bk;
```


```
FD->bk = BK;
BK->fd = FD;
```

What if…

如果chunk P的fd和bk被劫持了呢?




```
FD = P->fd     => FD = addr1

BK = P->bk     => BK = addr2

FD->bk = BK    => addr1->bk = addr2
               => *(addr1+0x18) = addr2

BK->fd = FD    => addr2->fd = addr1
               => *(addr2+0x10) = addr1
```

Exploit: Unlink Attack

unlink attack是指通过伪造chunk的fd、bk使得glibc在执行unlink操作时向伪造的fd和bk中写入特定的值

通常情况下，unlink attack是在off-by-null的场景中使用的（如果能直接通过UAF或堆溢出修改到free chunk的fd、bk，那就可以用更简单的方法进行利用，不需要使用unlink）。off-by-null使得glibc在free时相信前面的chunk已经被free了，并对其进行解链、合并的操作，而这个chunk实际上是一个allocated chunk，我们可以在其中伪造fd和bk

unlink attack -> 任意地址读写?

从前面的分析结果来看，*(addr1+0x18) = addr2 和 *(addr2+0x10) = addr1似乎做到了任意地址写，但实际上真的可以吗?

no way

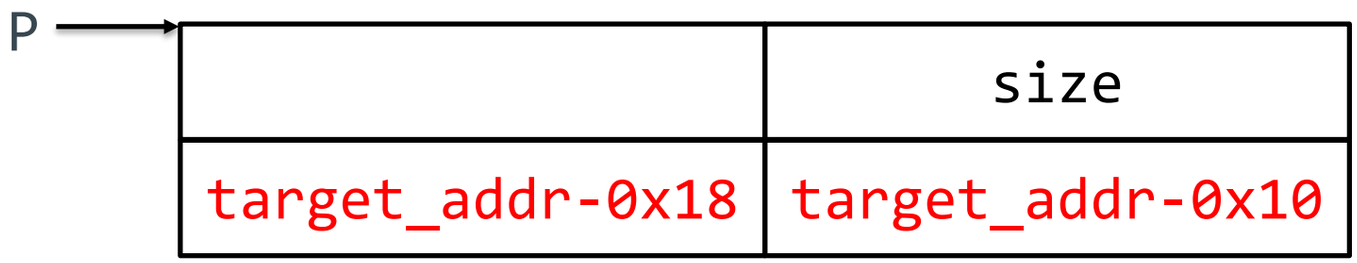
在glibc的实现中，unlink时会检查双向链表完整性:

```
// fd bk
if (__builtin_expect (FD->bk != P || BK->fd != P, 0)) \
    malloc_printerr (check_action, "corrupted double-linked list", P, AV); \
```

如果我们劫持P的fd和bk为任意值，那么P->fd->bk和P->bk->fd基本不可能满足等于P的要求

咋办

基于FD->bk == P和BK->fd == P这两个要求，我们可以进行如下构造：

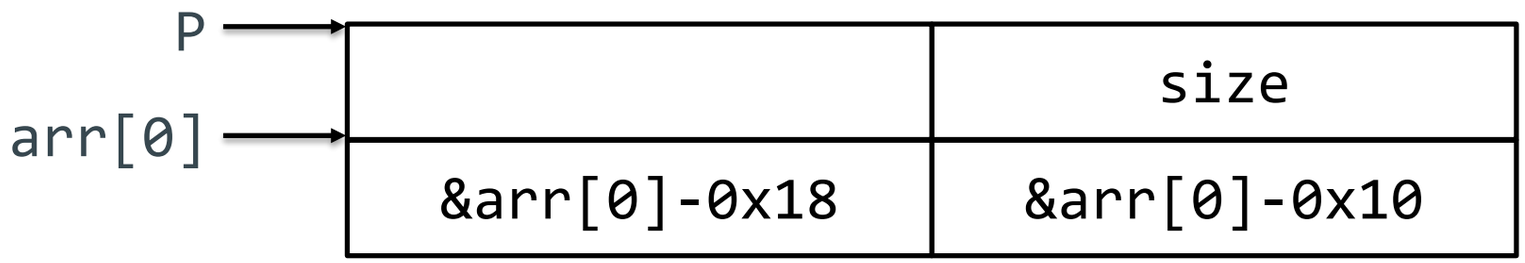


其中，target_addr地址上存放的是P指针
于是就通过了FD->bk == BK->fd == P的检查，并且unlink后的效果是*target_addr = target_addr-0x18，即向一个地址中写入这个地址本身-0x18的值

这个效果通常会用来控制指针数组，假设有一个已知地址的全局指针数组arr，我们通过unlink使得arr[0] = &arr[0]-0x18，于是在修改arr[0]的内容时，实际上是修改指针数组中的指针，从而可以进一步进行任意地址读写的操作

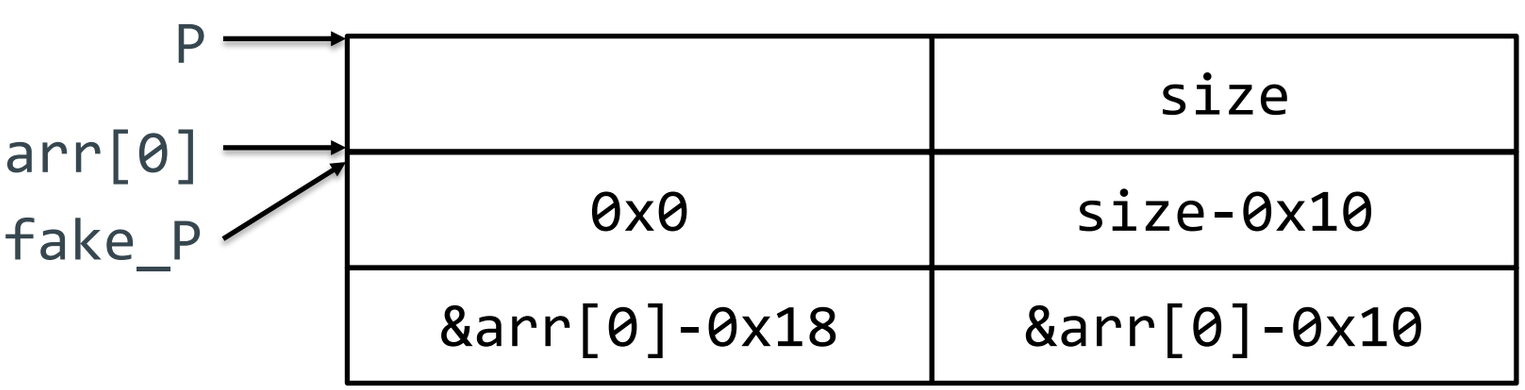
Any problem else?

malloc返回的指向chunk的指针实际上是指向chunk的data域，而bin中记录的指向free chunk的指针都是指向chunk头部的。因此如下布置并不能触发unlink，因为P->FD->bk不等于P，而是等于P+0x10：



我们需要在arr[0]指向的位置上伪造一个fake_P chunk，实际上unlink的对象是它，此时能够满足：

fake_P->FD->bk == arr[0] == fake_chunk_P

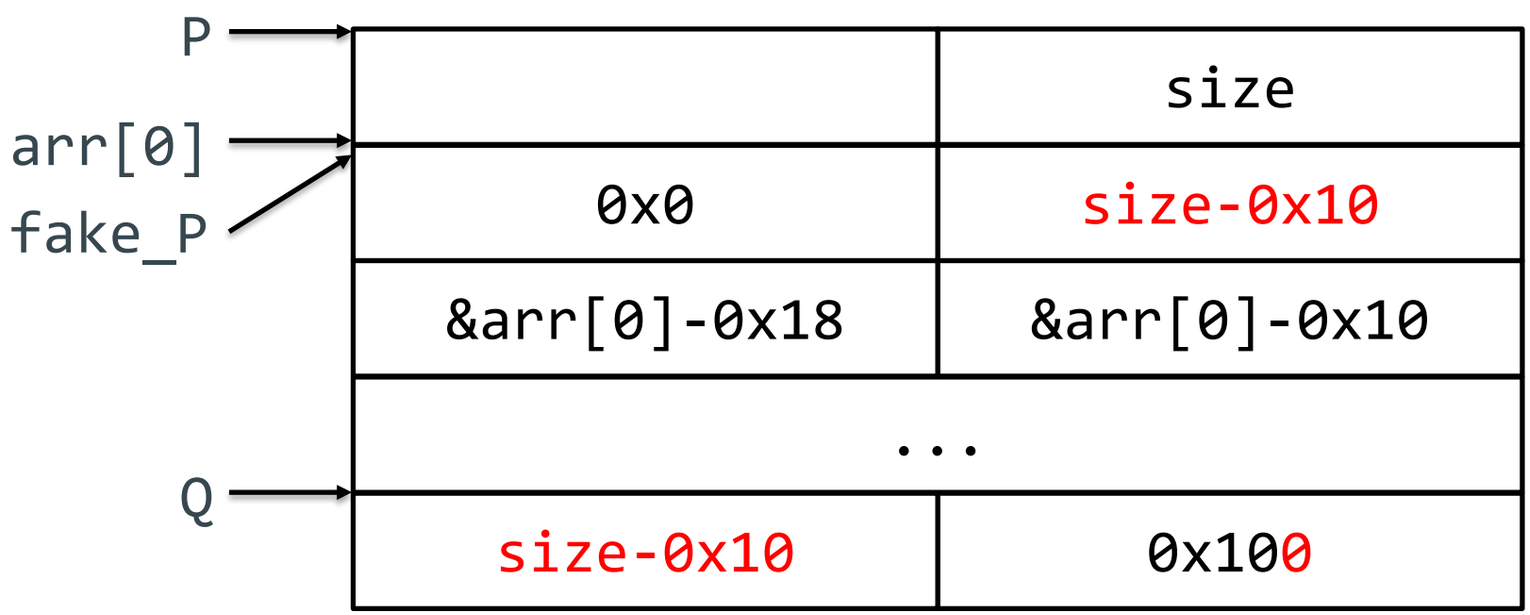


此外

在执行unlink操作时，还有另外一个检查：

```
if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0)) \
    malloc_printerr ("corrupted size vs. prev_size");                 \
```

因此被off-by-null的chunk的prev_size字段需要写上对应的fake_P的size：



Summary

1. 堆的宏观概念
2. 堆相关数据结构
   - chunk
   - arena
   - 各种bin
3. 基于这些数据结构实现分配与回收的具体操作
   - malloc
   - free
4. 基本堆利用手法
   - UAF
   - double free
   - unlink attack

真正的CTF pwn?

demo一下