长亭科技
CHAITIN.CN

二进制漏洞挖掘与利用

课时10: GLibc堆的实现

课时大纲

- 堆的概念与常见堆实现
- Arena、malloc_state、bin和chunk的结构
- malloc和free的工作流程
- 案例分析

堆的概念

- 栈通常用于为函数分配固定大小的局部内存
- 堆是可以根据运行时的需要进行动态分配和释放的内存，大小可变
  - Malloc/New
  - Free/Delete
- 堆的实现重点关注内存块的组织和管理方式，尤其是空闲内存块
  - 如何提高分配和 释放效率
  - 如何降低碎片化，提高空 间利用率
- 举例：浏览器的DOM树通常分配在堆上
  - 堆的实现算法影响堆分配网 页加载和动态效果速度
  - 堆的实现算法影响浏览器对内存的使用效率

常见堆实现

- dlmalloc - 通用分配器
- **ptmalloc2 - glibc**
  - **基于dlmalloc fork出来, 在2006年增加了多线程支持**
- jemalloc - FreeBSD、Firefox、Android
- tcmalloc - Google Chrome
- libumem - Solaris
- Windows 10 - segment heap

ptmalloc2的多线程支持

- 不同的线程维护不同的堆，称为 **per thread arena**
- 主线程创建的堆，称为 **main arena**
- Arena 数量受到CPU核数的限制
  - 对于32位系统: arena数量上限 = 2 * 核数.
  - 对于64位系统: arena数量上限 = 8 * n核数.

管理堆的各种结构

- arena
  - 指的是堆内存区域本身，并非 结构
  - 主线程的main arena通过sbrk创建
  - 其他线程arena通过mmap创建
- malloc_state
  - 管理arena的核心结构，包含堆的状 态信息、bins链表等
  - main arena对应的malloc_state结构存储在glibc的全局变量中
  - 其他线程arena对应的malloc_state存储在arena本身当中
- bins
  - bins用来管理空闲内存块，通常使用 链表结构来进行组织
- chunks
  - 内存块的结构

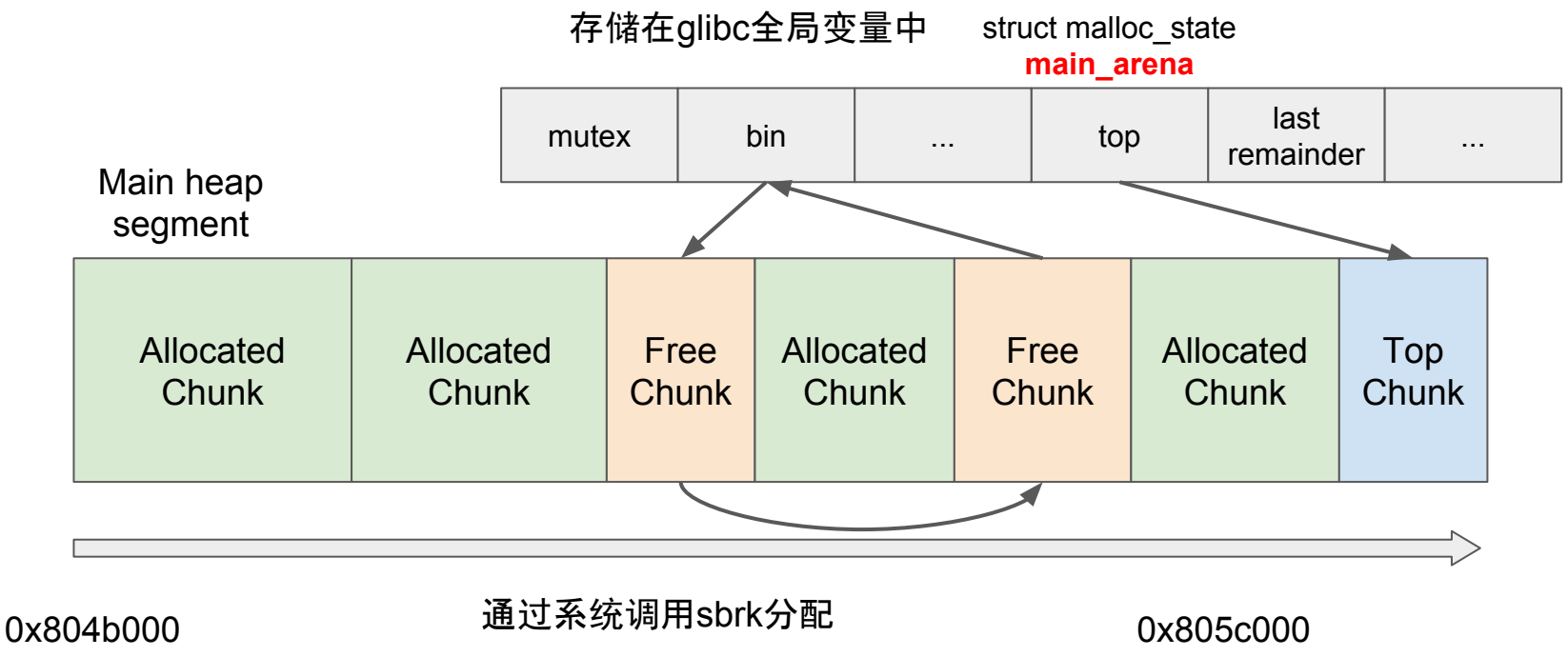
Arena 头部结构: malloc_state

malloc_state存储了Arena的状态，其中包括了很多用于管理空闲块的bins链表

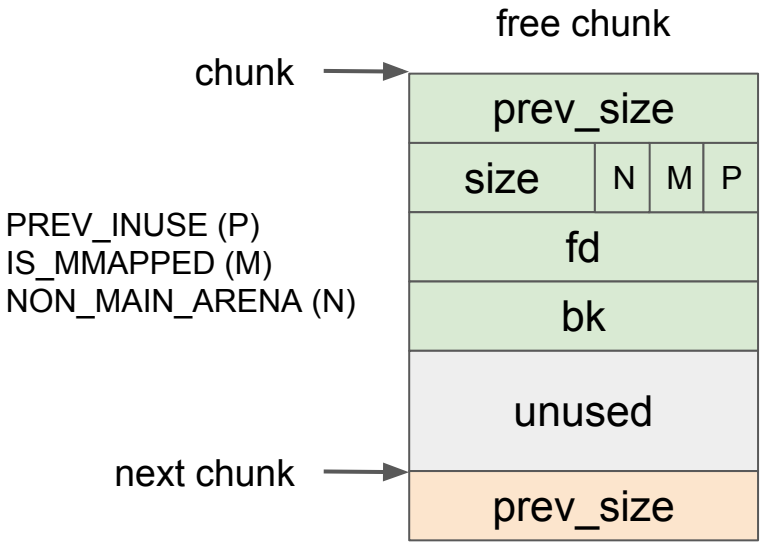
```
struct malloc_state {
   mutex_t mutex; /* Serialize access.  */
   int flags; /* Flags (formerly in max_fast).  */
   mfastbinptr fastbinsY[NFASTBINS]; /* Fastbins */
   mchunkptr top; /* Base of the topmost chunk, not otherwise kept in a bin */
   mchunkptr last_remainder; /* The remainder from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

static struct malloc_state main_arena; /* global variable in libc.so */
```

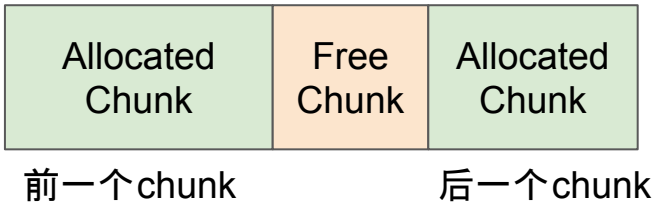
主线程的malloc_state结构存储在glibc的全局变量中，变量名为main_arena。

Main Arena概览



空闲内存块(free chunk)结构



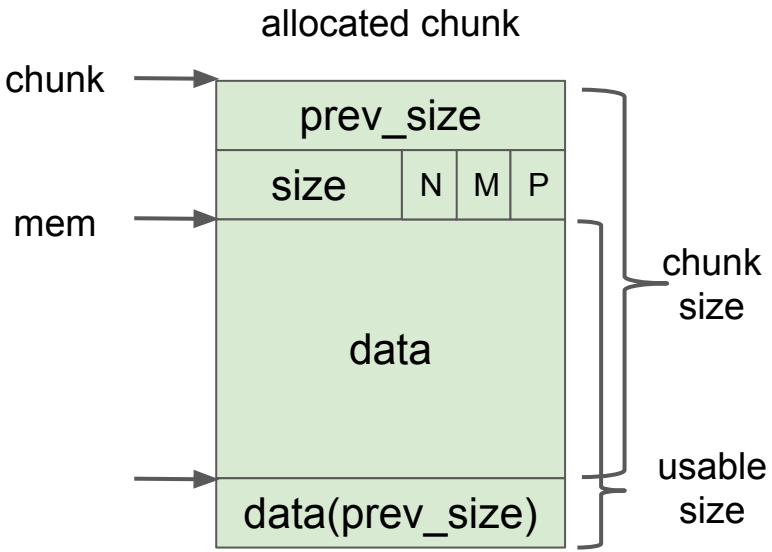


在x86 32位平台下，free chunk的第一个字段prev_size(4字节)存储了前一个chunk的大小。这里的“前一个”指的是内存地址连续概念下的“前一个”。这个字段只有在“前一个”chunk是free的情况下适用。

free chunk的第二个字段size记录了当前chunk的大小，该字段最低三个bit被用作其他含义。P代表PREV_INUSE，即代表前一个chunk是否被使用；M代表IS_MMAPPED，代表当前chunk是否是mmap出来的；N代表NON_MAIN_ARENA，代表改chunk是否属于非Main Arena。

第三字段fd和第四字段bk(4字节)前向指针和后向指针，这两个字段用于bin链表当中，用来链接大小相同或者相近的free chunk，便于后续分配时查找。

已分配内存块(allocated chunk)结构



在x86 32位平台下，allocated chunk的前两个字段和free chunk相通。

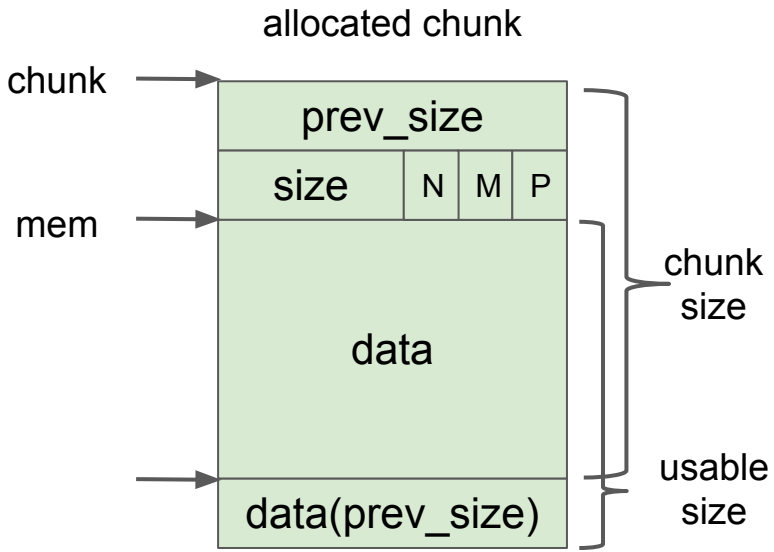
第三个字段开始到最后，chunk中存储的都是用户数据。甚至下一个chunk的第一个字段prev_size，也可被用来存放数据，原因是这个prev_size字段只有当“前一个”chunk是free的时候才有意义，如果“前一个”chunk是已经分配的，堆管理器并不关心。

所以对一个chunk来说，用户可用大小从偏移+8开始，一直到下一个chunk的prev_size字段。

在x86 32位平台下，chunk的大小一定是8字节的整数倍。malloc返回的指针为图中mem指向的位置，即数据开头。

malloc参数与chunk大小的关系

- malloc参数为用户申请的内存大小
- chunk包含数据和metadata
- 返回的chunk只要保证其中可用数据大小大于等于用户申请即可
- 在x86 32位平台下，chunk的大小一定是8字节的整数倍；x64平台下，chunk的大小一定是16字节的整数倍。



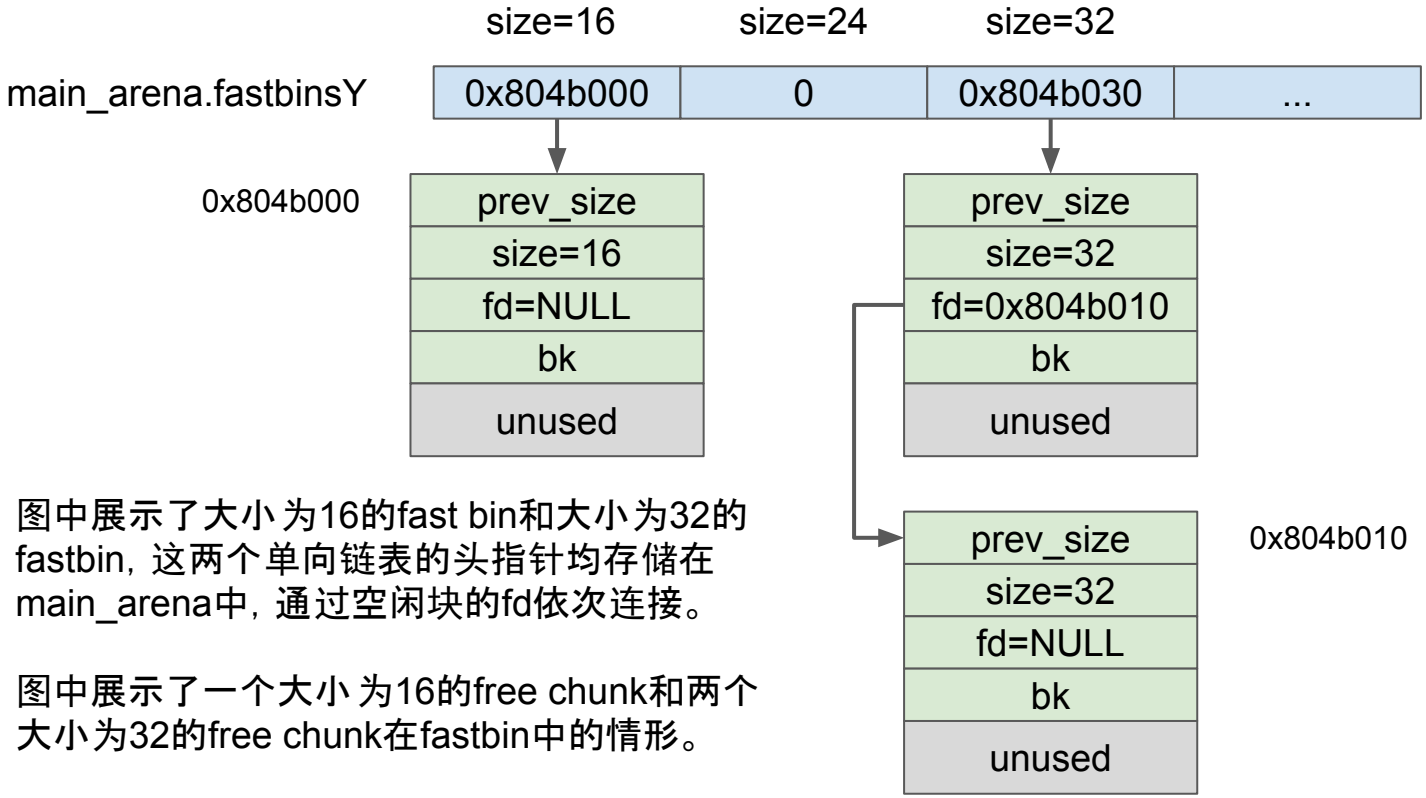
Bins结构

- Bins是用来管理和组织空闲内存块的链表结构，根据chunk的大小和状态，有许多中不同的Bins结构
- Fast bins
  - 用于管理小的 chunk
- Bins
  - small bins - 用于管理中等大小的chunk
  - large bins - 用于管理较大的chunk
  - unsorted bins - 用于存放未整理的chunk

Fast bins

- 大小
  - x86_32平台：16~64 字节
  - x64平台：32~128 字节
- 一共有10个fast bins
- 相同大小的chunk放在一个bin中
- 单向链表
- 后进先出
- 相邻的空闲fast bin chunk不会被合并
- 当chunk被free时，不会清理 PREV_INUSE 标志

Fast bin在内存中的结构示例



图中展示了大小为16的fast bin和大小为32的fastbin，这两个单向链表的头指针均存储在main_arena中，通过空闲块的fd依次连接。

图中展示了一个大小为16的free chunk和两个大小为32的free chunk在fastbin中的情形。

Small bins

- chunk 大小 < 512 bytes
- 共有62个 small bin
  - x86_32平台：16, 24, ..., 508 字节
- 相同大小的chunk放在一个bin中
- 双向循环链表
- 先进先出
- 当有空闲块相邻时，chunk会被合并成一个更大的chunk

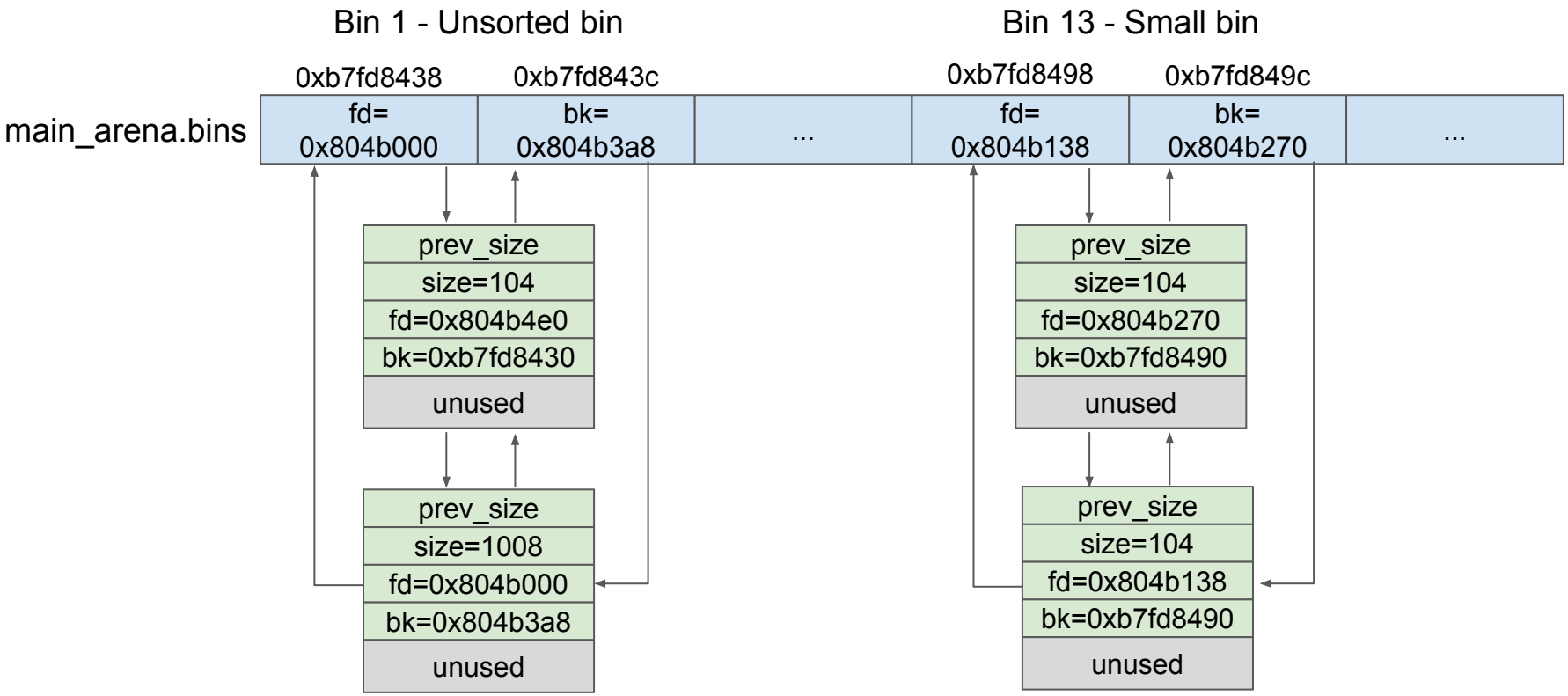
Large bins

- chunk 大小 >= 512 bytes
- 共有63个 large bin
- 大小相近的chunk放在同一个bin中
- 双向循环链表
- 先进先出
- chunk按照大小从大到小排序
- 当有空闲块相邻，chunk会被合并

Unsorted bin

- x86 32位平台中：chunk大小 > 64 字节
- 只存在唯一一个unsorted bin
- 双向循环链表
- 当一个chunk（非fast bin中）被free，它首先被放入 unsorted bin，等后续整理时才会放入对应的small bin/fast bin

Unsorted bins 与 small bins



其他 chunk

- Top chunk
  - 不属于任何 bin
  - 在 arena 中处于最高地址
  - 当没有其他空闲块时，top chunk就会被用于分配
  - 分裂时
    - 一块是请求大小的chunk
    - 另一块余下chunk将成为新的top chunk
- Last_remainder
  - 当请求small chunk大小的内存时，如发生分裂，则剩余的chunk保存为last_remainder

malloc()的工作流程

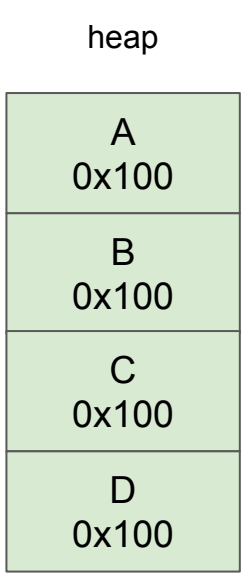
1. 在fast bins中寻找fast chunk，如找到则结束
2. 在small bins中寻找small chunk，如找到则结束
3. 循环
   a. 检查unsorted bin中的last_remainder
      i. 如果last_remainder足够大，则分裂之，将剩余的chunk标记为新的last_remainder
   b. 在 unsorted bin 中搜索，同时进行整理
      i. 如遇到精确大小，则返回，否则就把当前chunk整理到 small/large bin中去
   c. 在small bin和large bin中搜索最合适的chunk（不一定是精确大小）
4. 使用 top chunk

free()的工作流程

1. 如果是fast chunk, 放入 fast bin，结束
2. 如果前一个chunk是free的
   a. unlink前面的chunk
   b. 合并两个chunk，并放入unsorted bin
3. 如果后一个chunk是top chunk, 则将当前chunk并入top chunk
4. 如果后一个chunk时free的
   a. unlink后面的chunk
   b. 合并两个chunk，并放入unsorted bin
5. 前后chunk都不是free的，放入unsorted bin

案例分析


```
#include <stdio.h>
int main() {
    char *A, *B, *C, *D;
    A = malloc(0x100 - 8);
    B = malloc(0x100 - 8);
    C = malloc(0x100 - 8);
    D = malloc(0x100 - 8);

    free(A);
    free(C);

    A = malloc(0x100 - 8);
    free(A);
    A = malloc(0x80 - 8);
    free(B);

    return 0;
}
```



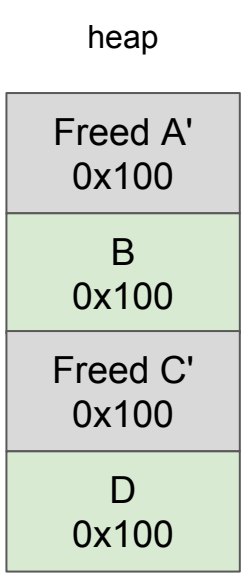
unsorted bins

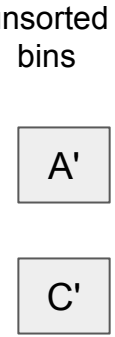
small bins

分配A、B、C、D四个大小为0x100的chunk

案例分析


```
#include <stdio.h>
int main() {
    char *A, *B, *C, *D;
    A = malloc(0x100 - 8);
    B = malloc(0x100 - 8);
    C = malloc(0x100 - 8);
    D = malloc(0x100 - 8);

    free(A);
    free(C);

    A = malloc(0x100 - 8);
    free(A);
    A = malloc(0x80 - 8);
    free(B);

    return 0;
}
```



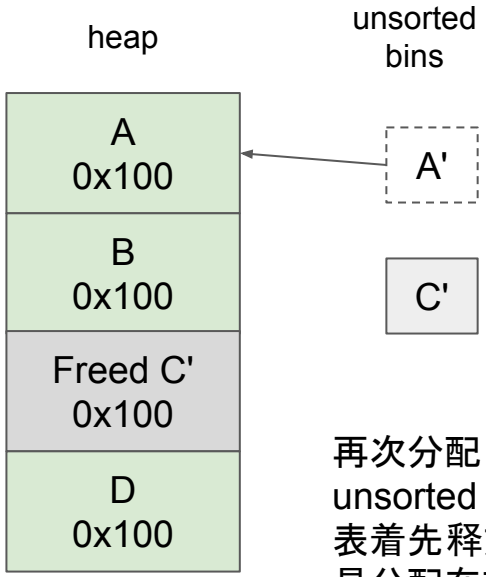


small bins

释放 chunk A 和 chunk C，这两个chunk不相邻，因此不会出现合并，直接进入unsorted bin，我们记录为chunk A'和chunk C'

案例分析

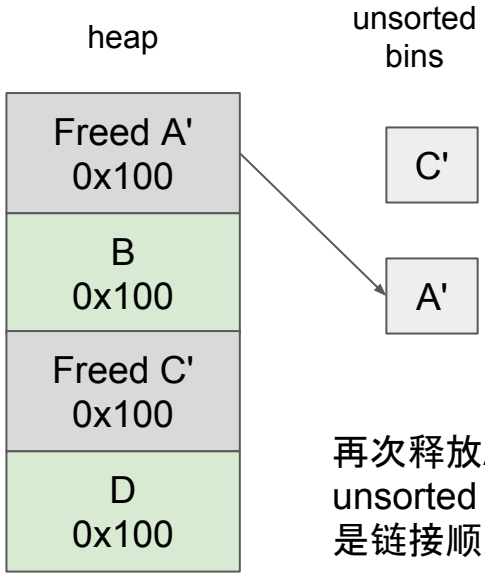

```
#include <stdio.h>
int main() {
    char *A, *B, *C, *D;
    A = malloc(0x100 - 8);
    B = malloc(0x100 - 8);
    C = malloc(0x100 - 8);
    D = malloc(0x100 - 8);

    free(A);
    free(C);

    A = malloc(0x100 - 8);
    free(A);
    A = malloc(0x80 - 8);
    free(B);

    return 0;
}
```



再次分配 0x100 大小的chunk，由于unsorted bin是先进先出的链表，代表着先释放的会优先被使用，因此还是分配在首次释放的位置，Chunk A的位置

案例分析


```
#include <stdio.h>
int main() {
    char *A, *B, *C, *D;
    A = malloc(0x100 - 8);
    B = malloc(0x100 - 8);
    C = malloc(0x100 - 8);
    D = malloc(0x100 - 8);

    free(A);
    free(C);

    A = malloc(0x100 - 8);
    free(A);
    A = malloc(0x80 - 8);
    free(B);

    return 0;
}
```

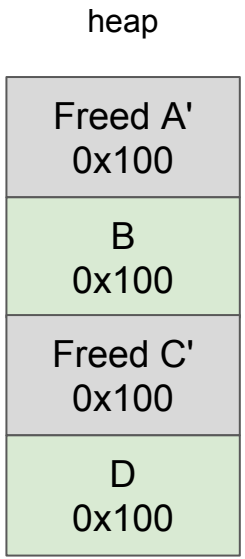


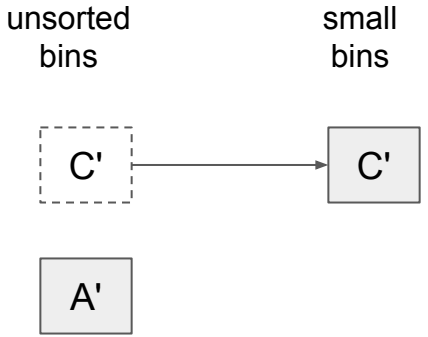
再次释放A，0x100的chunk再次回到unsorted bin，如图中的chunk A'，只是链接顺序变了

案例分析


```
#include <stdio.h>
int main() {
    char *A, *B, *C, *D;
    A = malloc(0x100 - 8);
    B = malloc(0x100 - 8);
    C = malloc(0x100 - 8);
    D = malloc(0x100 - 8);

    free(A);
    free(C);

    A = malloc(0x100 - 8);
    free(A);
    A = malloc(0x80 - 8);
    free(B);

    return 0;
}
```

(1/3)



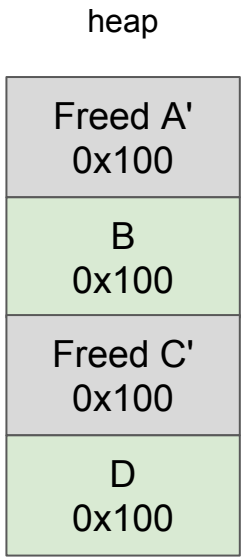


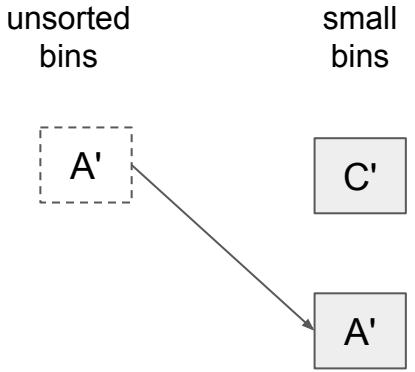
这次新分配一个大小为0x80的chunk，fast bin和small bin中都找不到合适的chunk，因此开始进入整理循环，遍历unsorted bin，chunk C'大小不符合，整理进入对应的small bin

案例分析


```
#include <stdio.h>
int main() {
    char *A, *B, *C, *D;
    A = malloc(0x100 - 8);
    B = malloc(0x100 - 8);
    C = malloc(0x100 - 8);
    D = malloc(0x100 - 8);

    free(A);
    free(C);

    A = malloc(0x100 - 8);
    free(A);
    A = malloc(0x80 - 8);
    free(B);

    return 0;
}
```

(2/3)



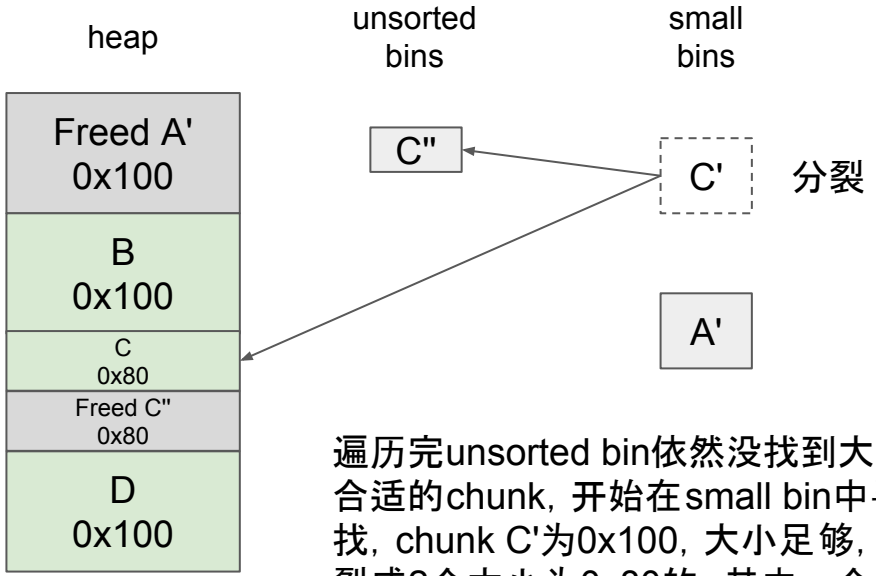


继续遍历unsorted bin，chunk A'大小不符合，整理进入对应的small bin

案例分析


```
#include <stdio.h>
int main() {
    char *A, *B, *C, *D;
    A = malloc(0x100 - 8);
    B = malloc(0x100 - 8);
    C = malloc(0x100 - 8);
    D = malloc(0x100 - 8);

    free(A);
    free(C);

    A = malloc(0x100 - 8);
    free(A);
    A = malloc(0x80 - 8);
    free(B);

    return 0;
}
```

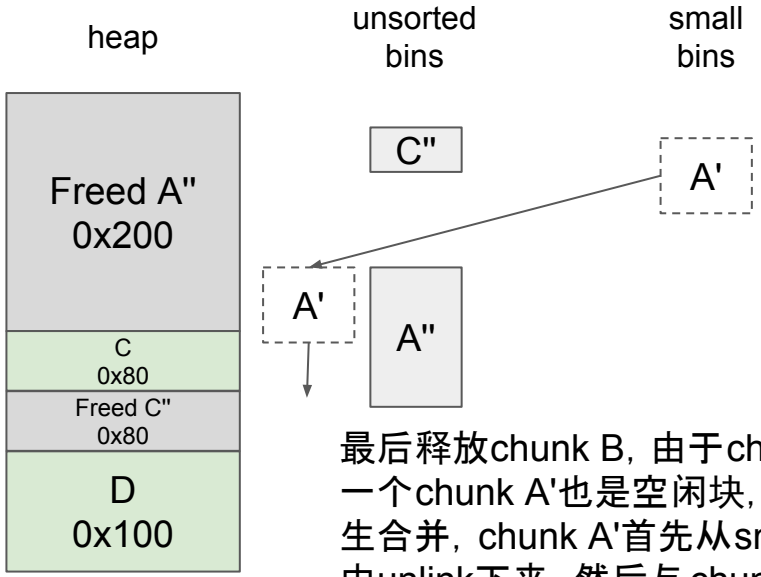
(3/3)



遍历完unsorted bin依然没找到大小合适的chunk，开始在small bin中寻找，chunk C'为0x100，大小足够，分裂成2个大小为0x80的，其中一个返回，剩下的一个 进入unsorted bin

案例分析


```
#include <stdio.h>
int main() {
    char *A, *B, *C, *D;
    A = malloc(0x100 - 8);
    B = malloc(0x100 - 8);
    C = malloc(0x100 - 8);
    D = malloc(0x100 - 8);

    free(A);
    free(C);

    A = malloc(0x100 - 8);
    free(A);
    A = malloc(0x80 - 8);
    free(B);

    return 0;
}
```



最后释放chunk B，由于chunk B前一个chunk A'也是空闲块，因此发生合并，chunk A'首先从small bins中unlink下来，然后与chunk