PWN 0x3

Glibc 2.31 Heap 101

WHOAMI

- YingMuo @ SQLab
- Balsn, TWN48
- HITCON 2022 Speaker
- PWN2OWN 2023 3rd

OUTLINE

- Heap Intro
- Vulnerability

Heap Info

Heap Intro

- 可以在 Runtime 動態分配及釋放記憶體，讓記憶體使用更有效率
- 目前常見的管理機制有
  - **Glibc – ptmalloc**
  - Google – tcmalloc
  - Facebook - jemalloc

Heap Intro

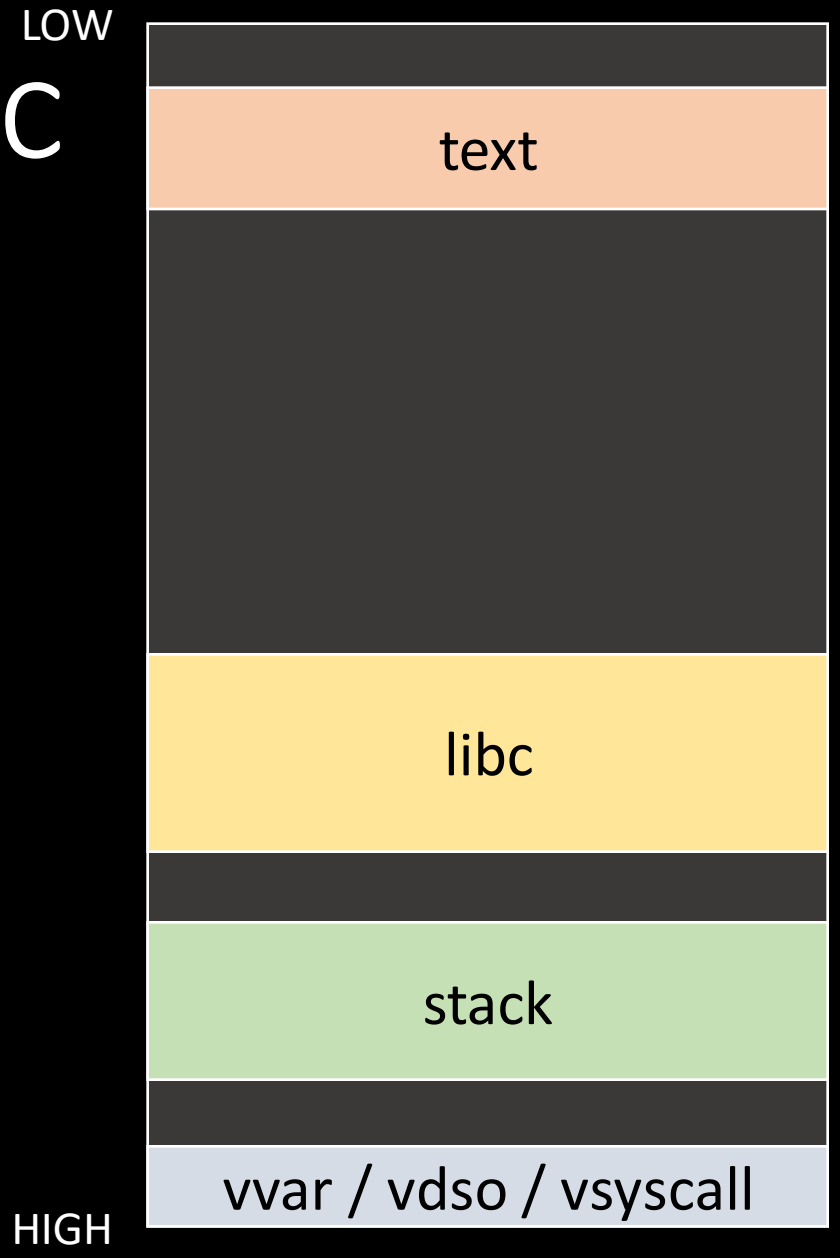
- Glibc 提供以下 function
  - malloc 分配記憶體
  - free 釋放記憶體
  - calloc 分配記憶體並清空
  - realloc 重新分配已分配的空間大小

Heap Intro

- allocate size >= 0x20000 bytes 會 syscall mmap
- 此外會使用 heap 的空間
  - 如果 freed 空間足夠就切一塊記憶體給使用者
  - 若 freed 空間不夠會 syscall brk 新增 0x21000 的記憶體空間

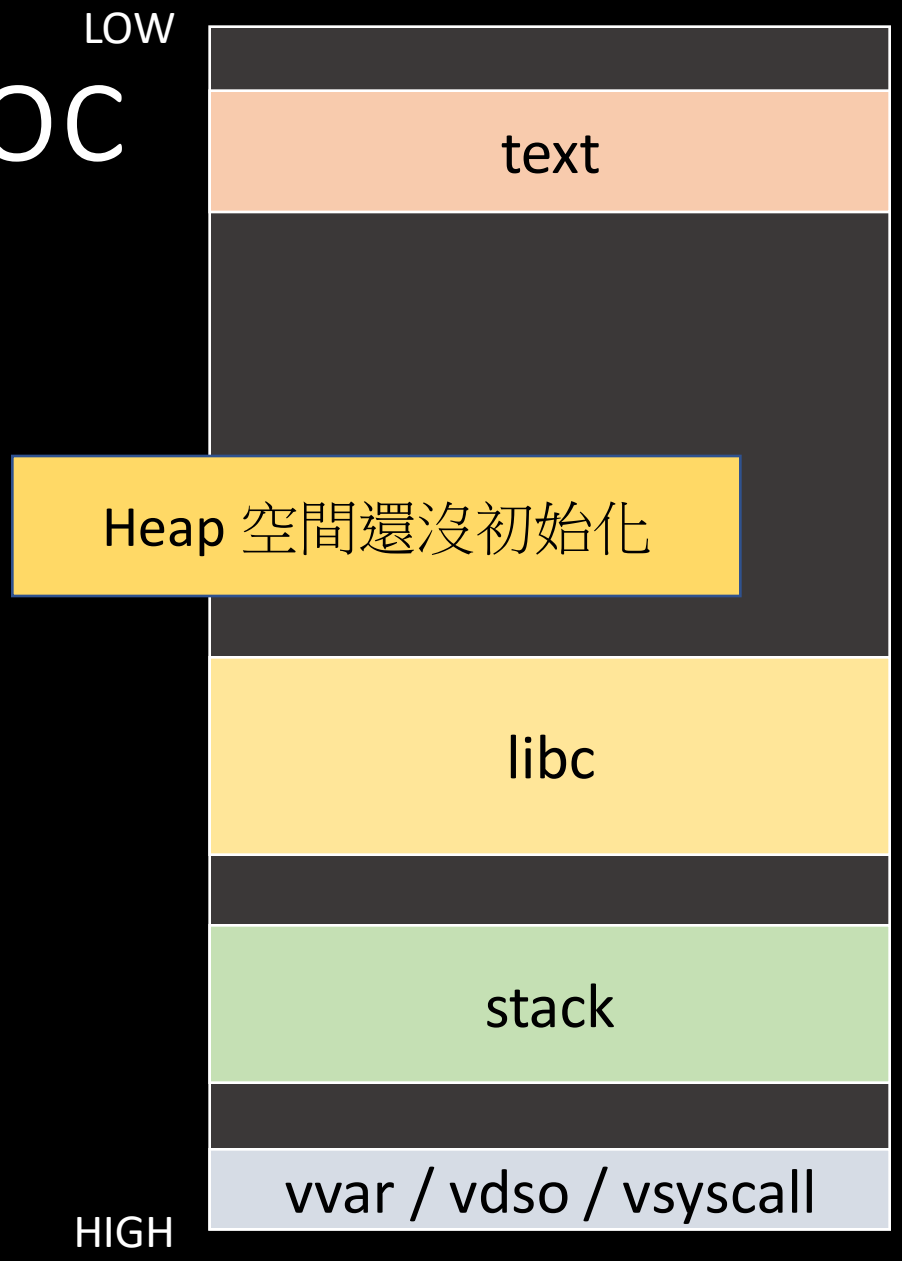
Mem Alloc

- ***Init state***
- Malloc 0x10000
- Malloc 0x10000
- Malloc 0x10000
- Malloc 0x21000



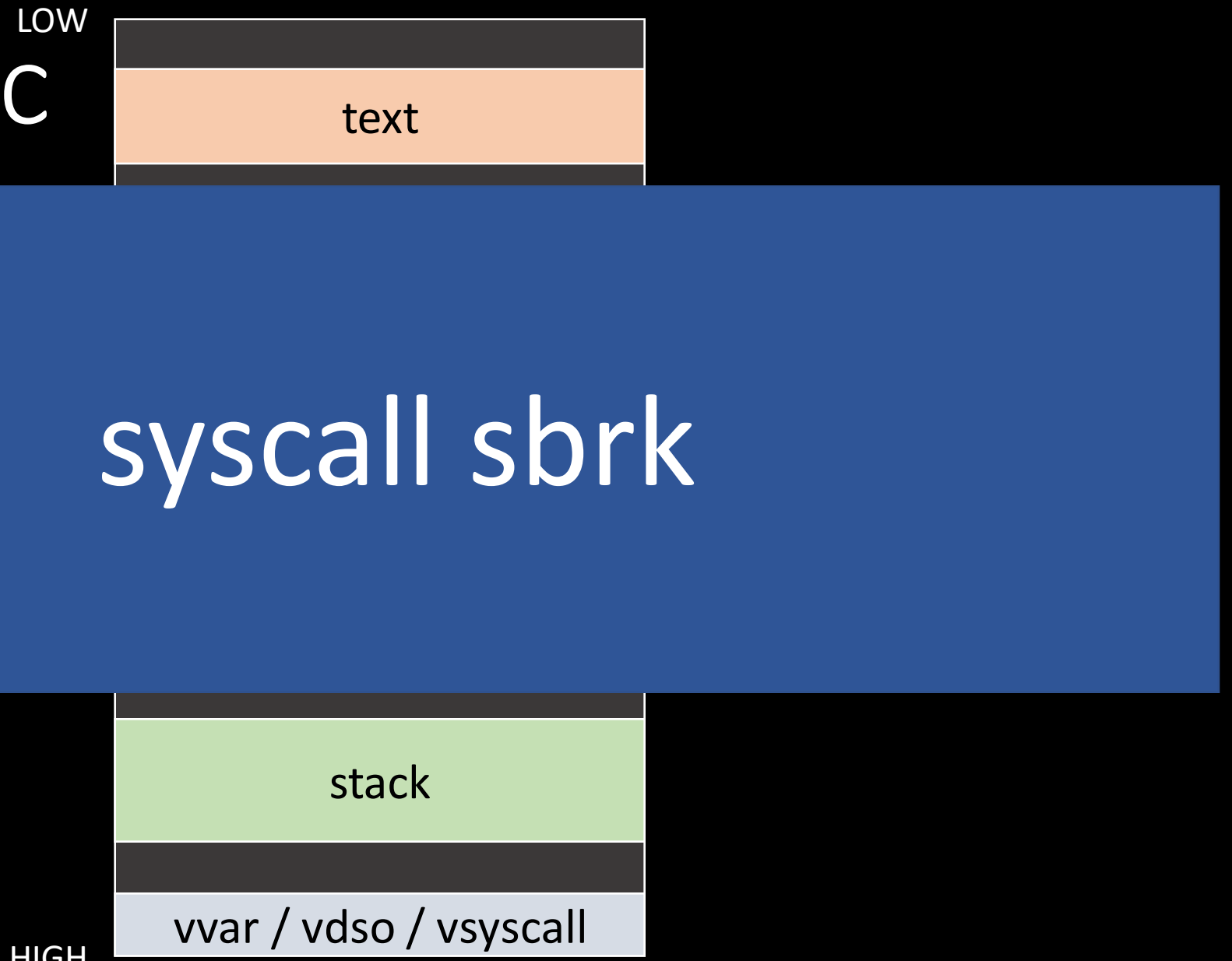
Mem Alloc

- Init state
- ***Malloc 0x10000***
- Malloc 0x10000
- Malloc 0x10000
- Malloc 0x21000



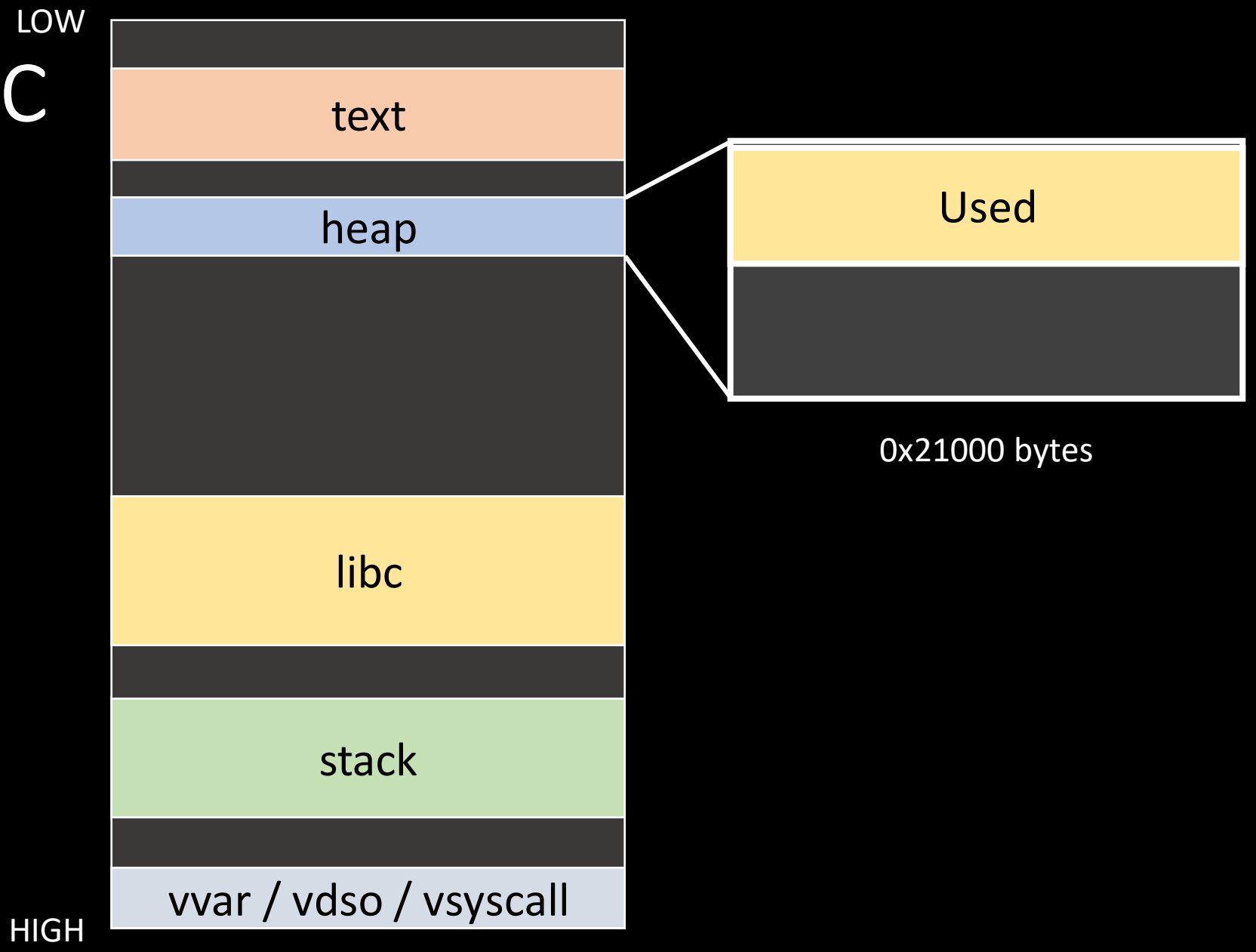
Mem Alloc

- I
- M
- N
- N
- Malloc 0x21000



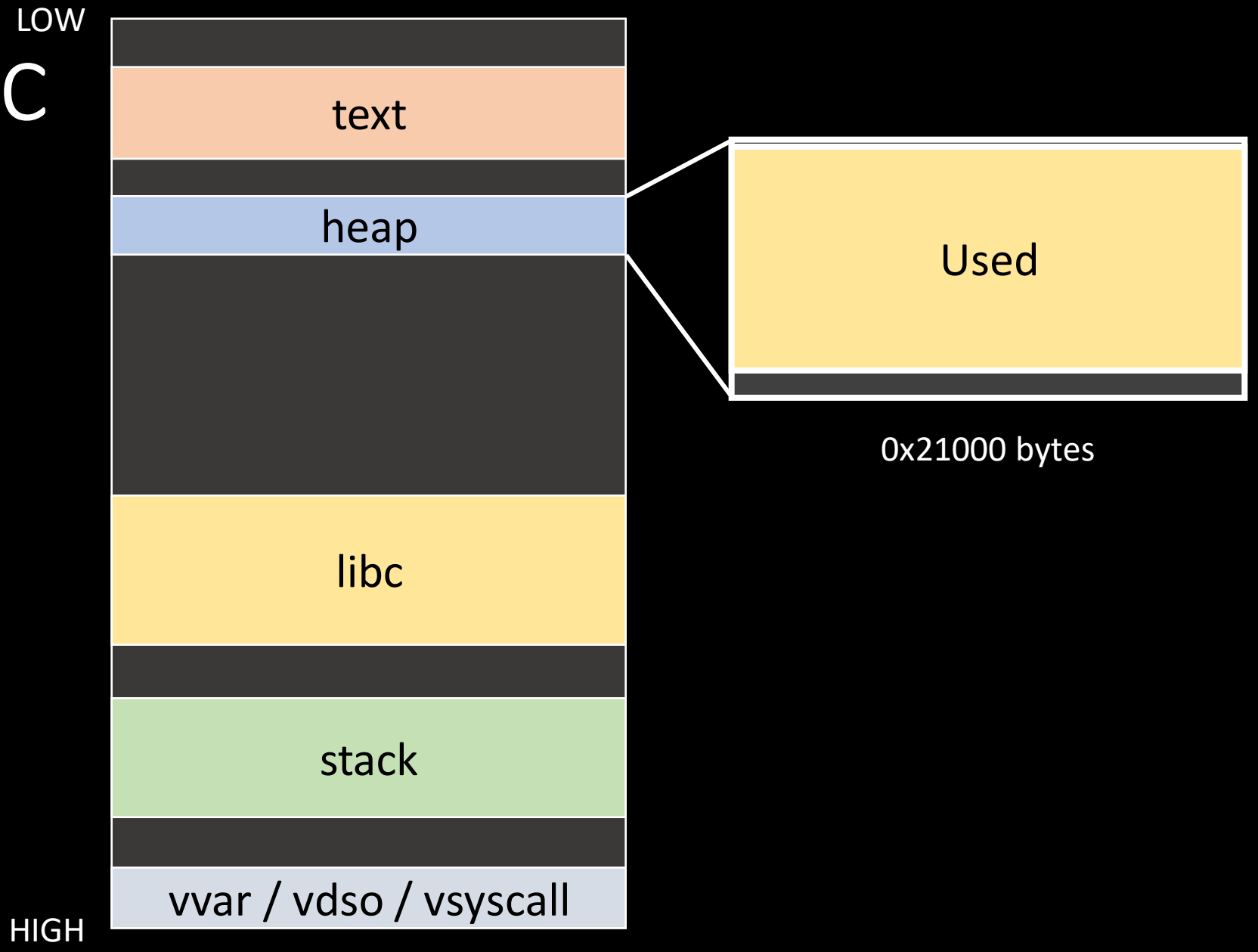
Mem Alloc

- Init state
- ***Malloc 0x10000***
- Malloc 0x10000
- Malloc 0x10000
- Malloc 0x21000



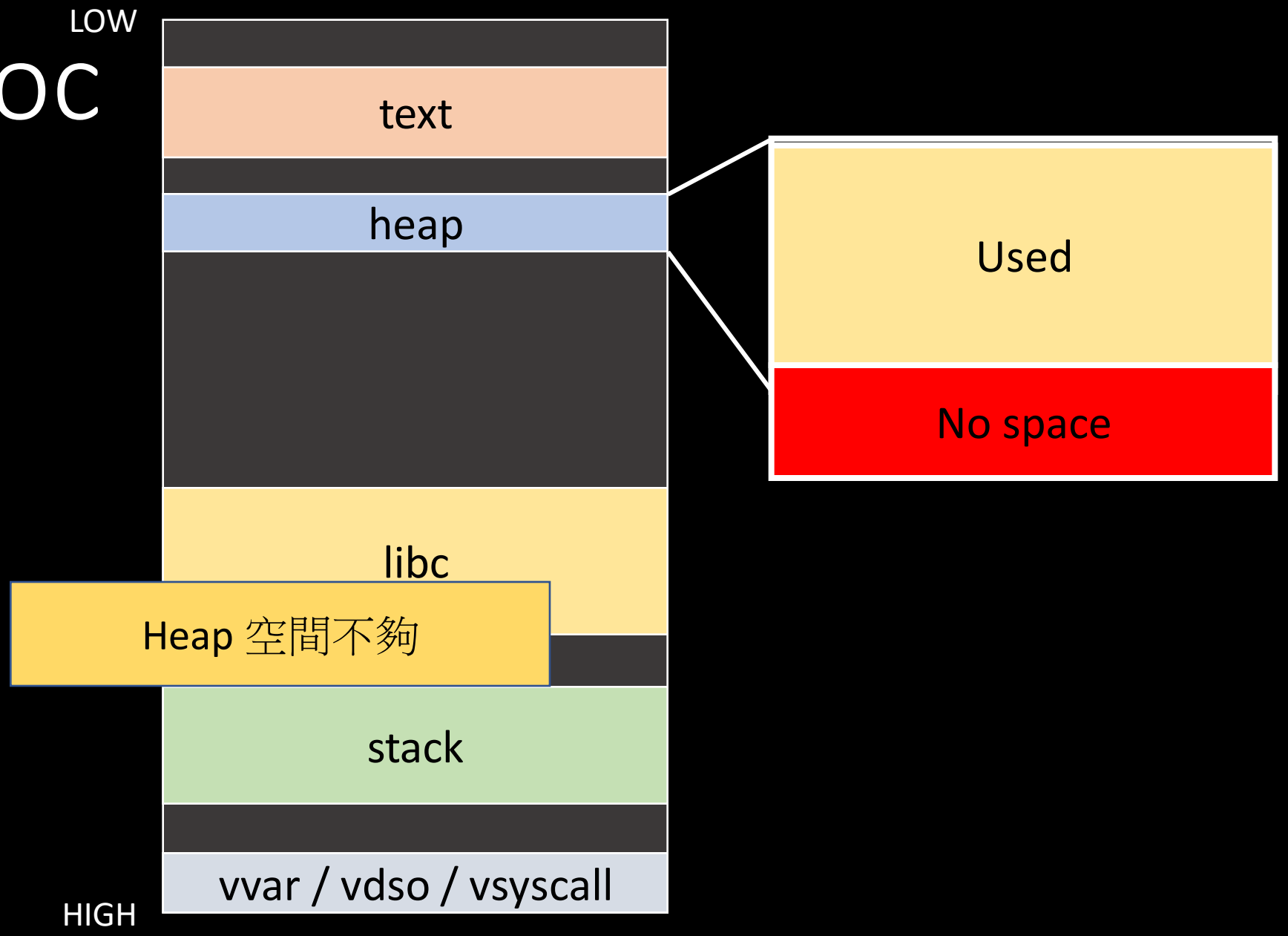
Mem Alloc

- Init state
- Malloc 0x10000
- ***Malloc 0x10000***
- Malloc 0x10000
- Malloc 0x21000



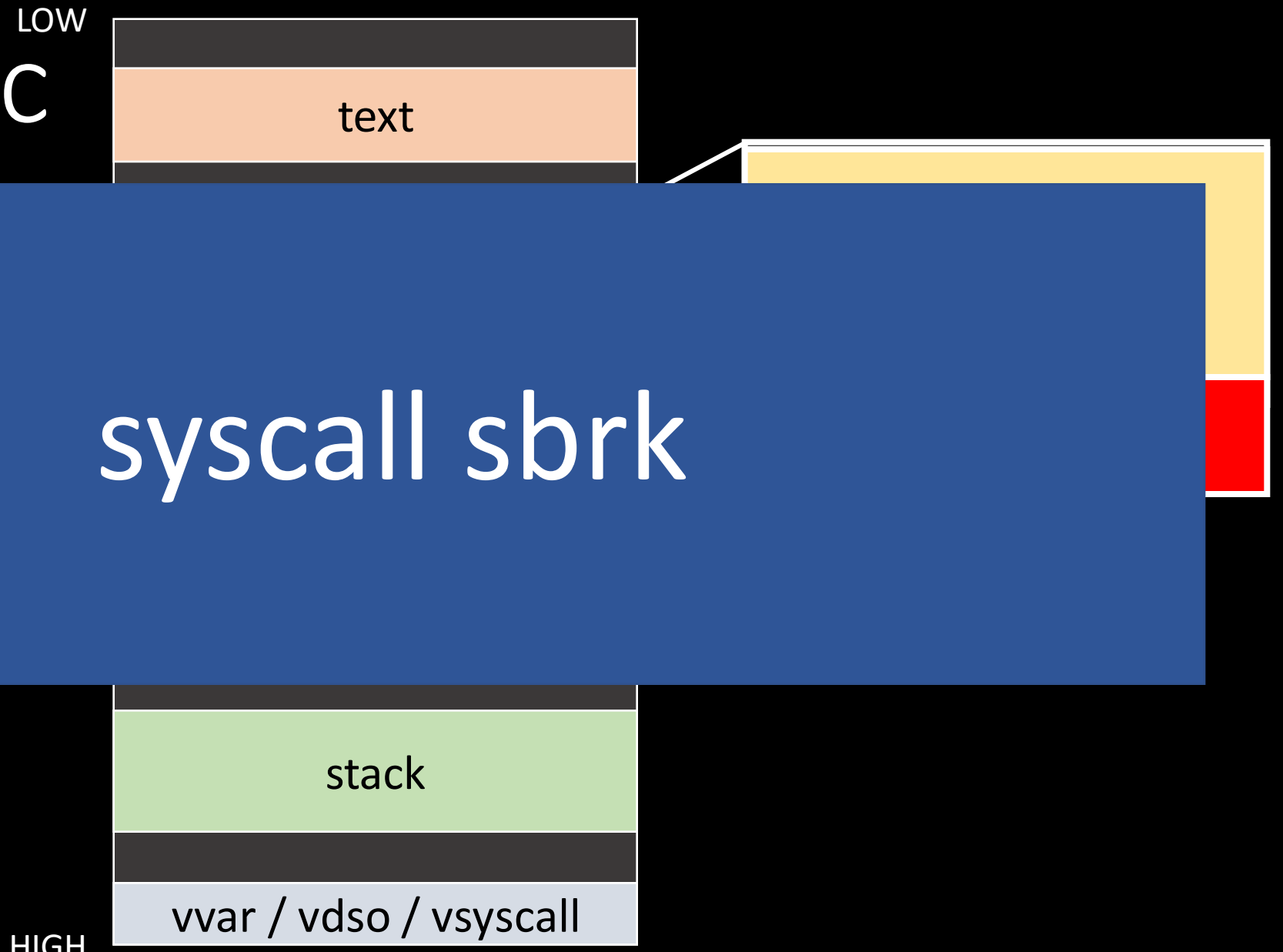
Mem Alloc

- Init state
- Malloc 0x10000
- Malloc 0x10000
- ***Malloc 0x10000***
- Malloc 0x21000



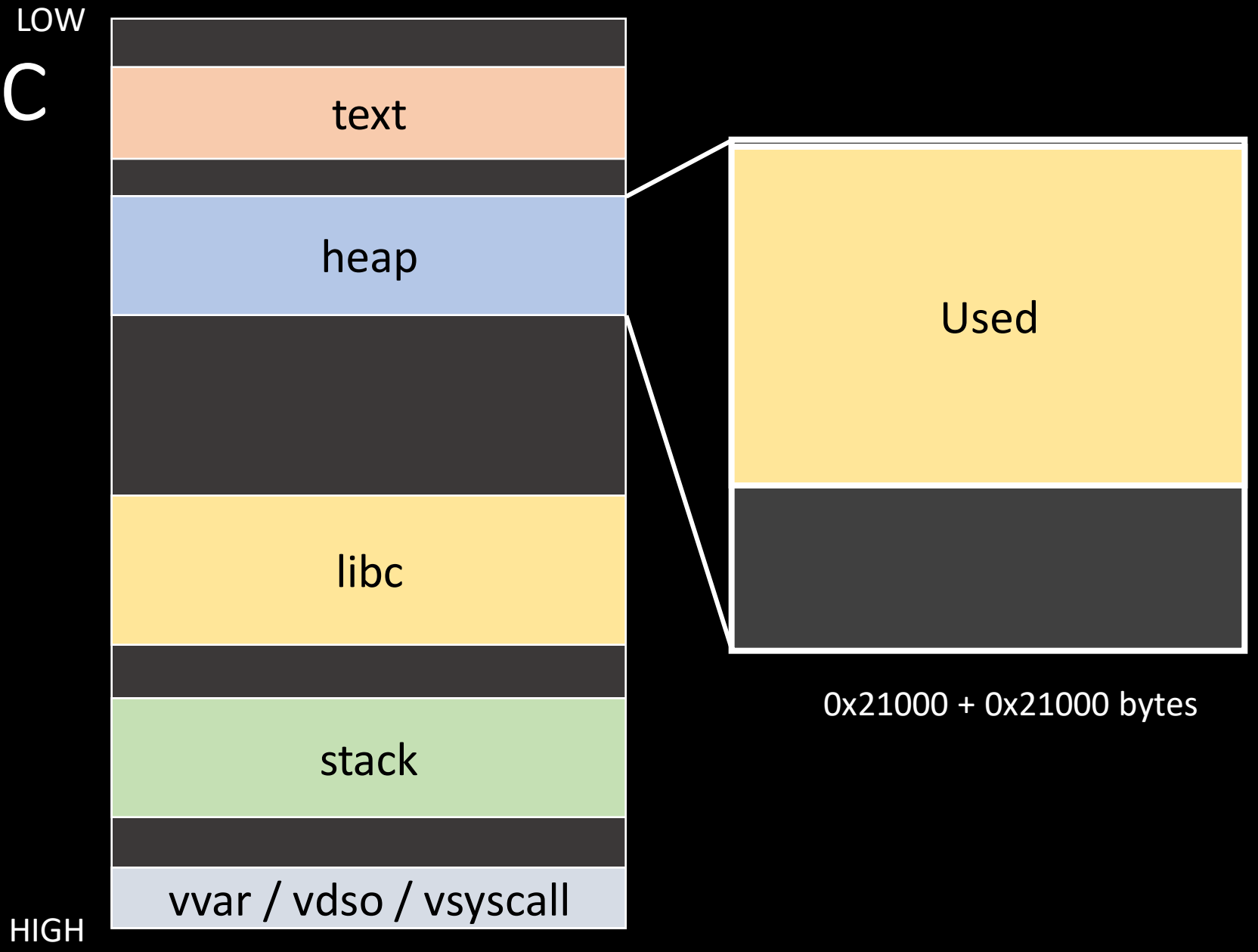
Mem Alloc

- I
- M
- M
- M
- Malloc 0x21000



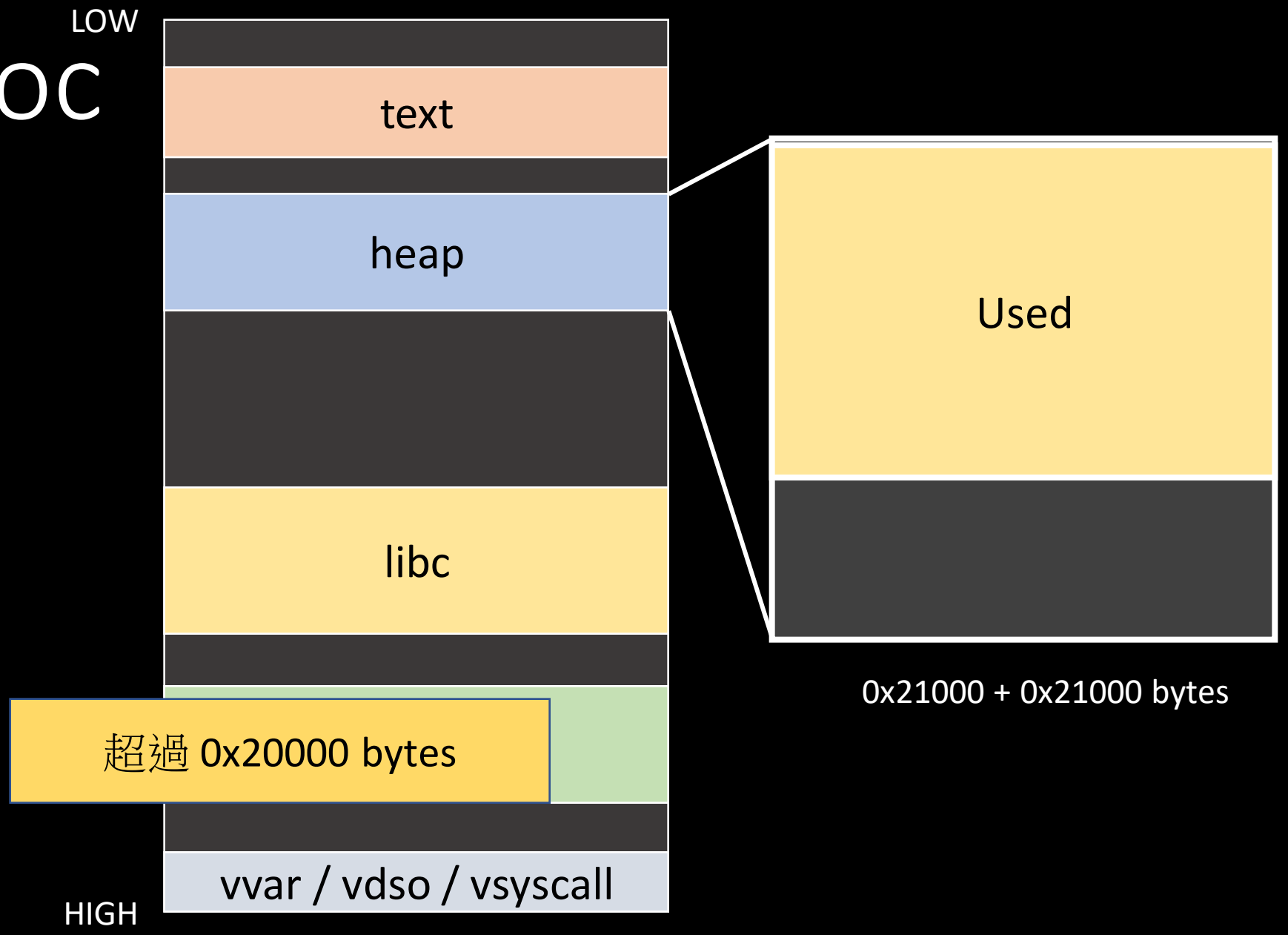
Mem Alloc

- Init state
- Malloc 0x10000
- Malloc 0x10000
- ***Malloc 0x10000***
- Malloc 0x21000



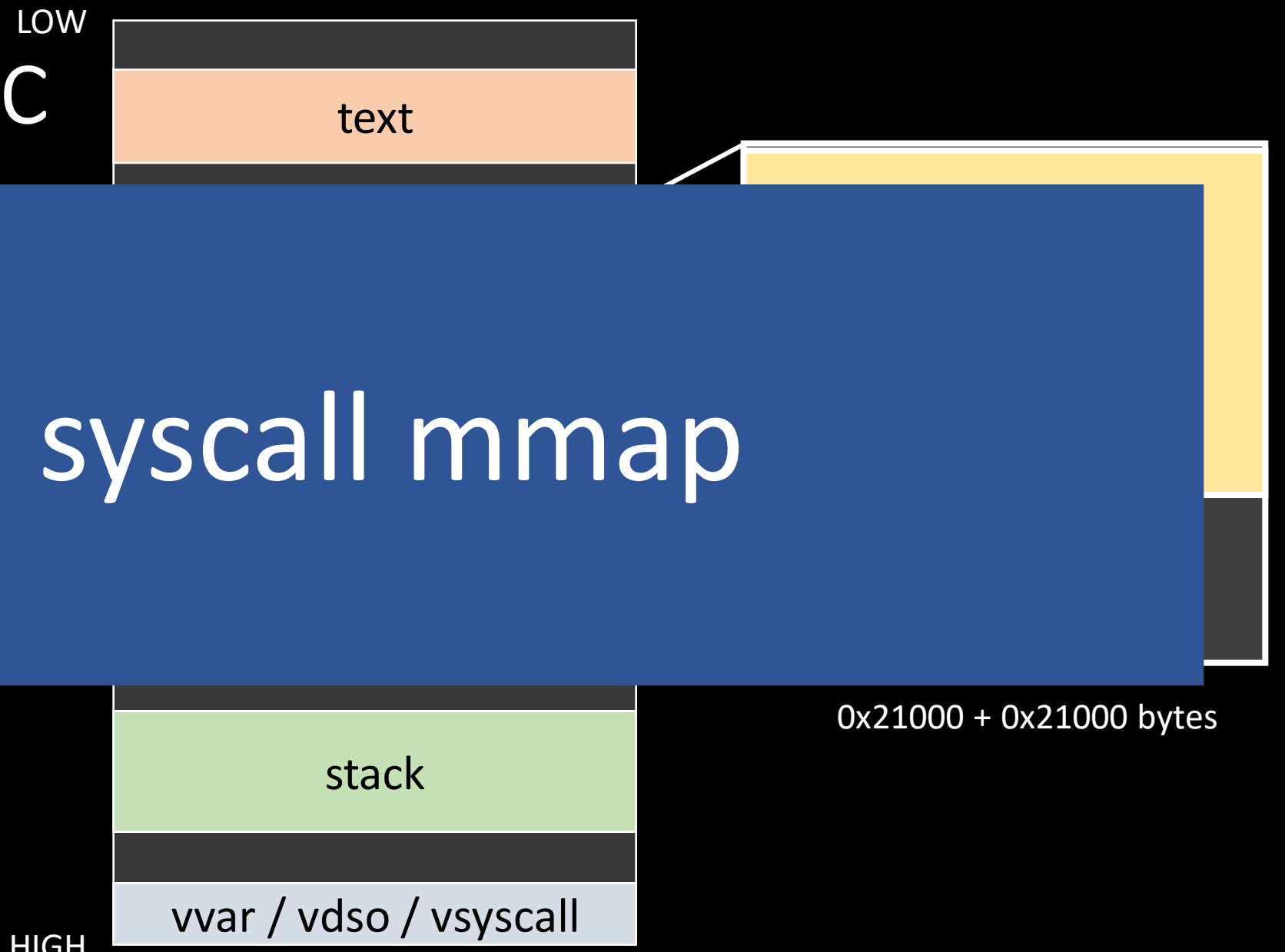
Mem Alloc

- Init state
- Malloc 0x10000
- Malloc 0x10000
- Malloc 0x10000
- ***Malloc 0x21000***



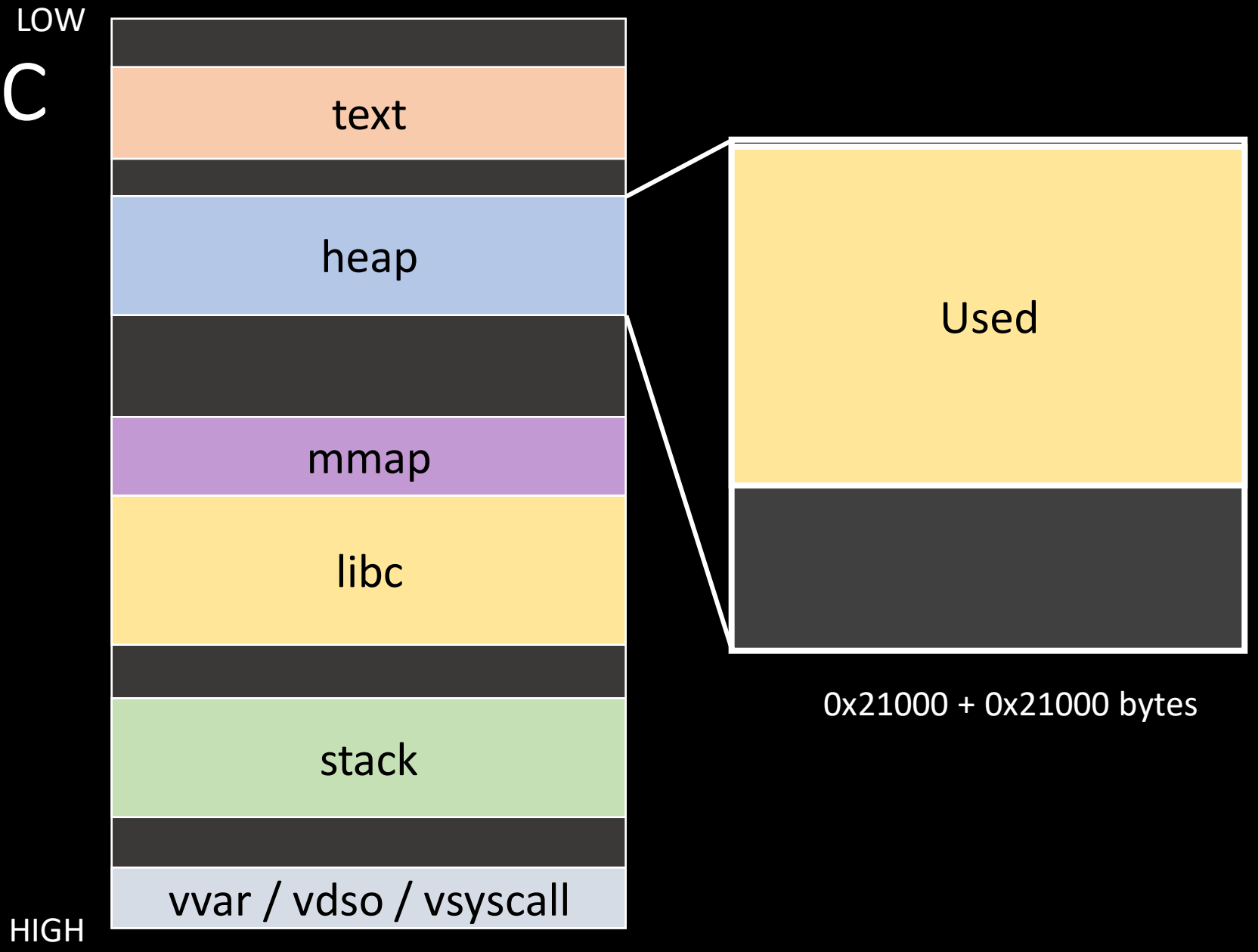
Mem Alloc

- I
- M
- M
- M
- ***Malloc 0x21000***



Mem Alloc

- Init state
- Malloc 0x10000
- Malloc 0x10000
- Malloc 0x10000
- ***Malloc 0x21000***

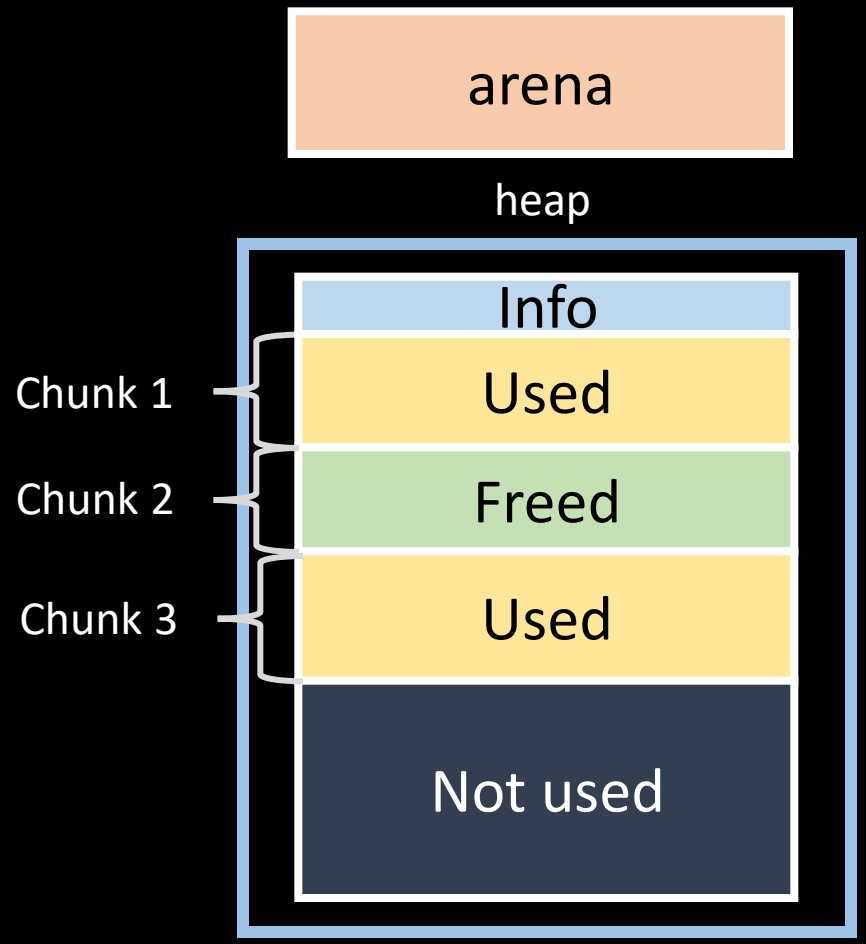


Mem Alloc

- mmap 會對齊 page size (0x1000)
  - malloc 0x21100 實際會 mmap 0x22000
- malloc < 0x20000 bytes 也會對齊
  - 64 bits 對齊 0x10 bytes
  - 32 bits 對齊 0x8 bytes

Data Structure

- 分配機制由三種結構來構成
  - chunk – 使用者 malloc 拿到的小塊記憶體
  - heap – 用來分發 chunk 的大塊記憶體
  - arena – 紀錄 heap 資訊、管理 chunk 的結構



Data Structure

- Chunk - struct malloc_chunk
- Heap – struct _heap_info
  - 每個 thread 會有 1 個 heap (main thread 沒有 heap 結構)
- Arena - struct malloc_state
  - 正常每個 heap 會有 1 個 arena (thread 太多時會共用)
  - main thread 的 arena 叫 main_arena 不在 heap 而是 libc 的 global variable
  - 這次課程只討論 single thread 所以不會有 heap 結構

Chunk

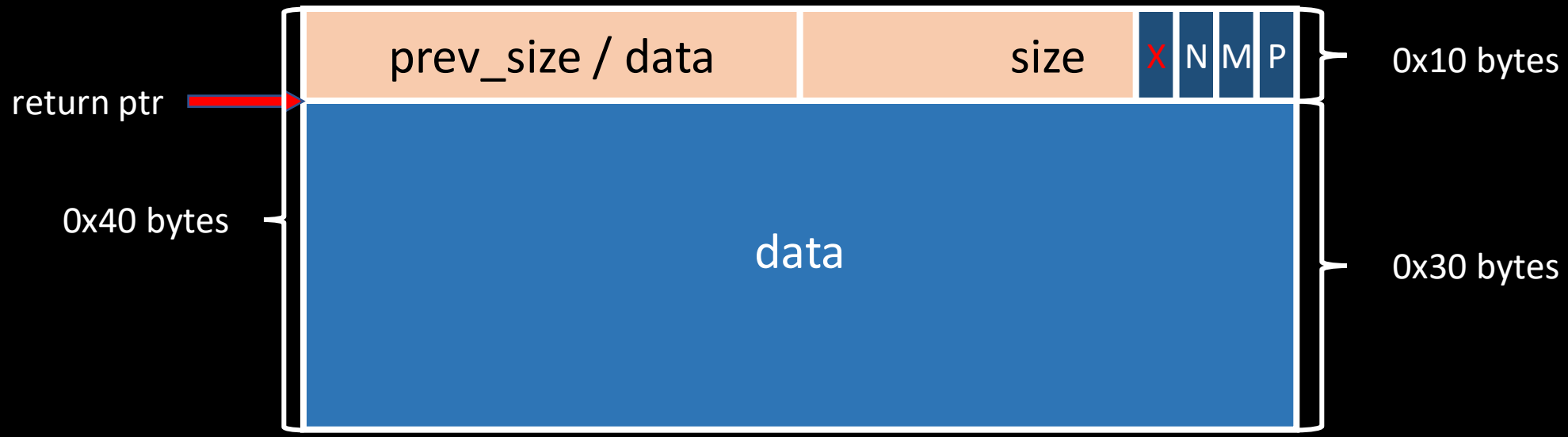
- 動態記憶體分配的基本單位，最小 0x20 bytes
- 對齊 0x10 bytes，也就是 0x20, 0x30, 0x40, …
- 正在使用的 chunk 叫 allocated chunk
- 已經被釋放的 chunk 叫 freed chunk
  - 釋放時會依照當前狀況放進不同的 bin 來回收管理 freed chunk
  - 之後在要求記憶體時就會從 bin 裡拿 chunk 給使用者

Chunk

```
struct malloc_chunk {
  /* header */
  INTERNAL_SIZE_T      mchunk_prev_size;  /* 若前一個是 freed chunk 就存它的大小  */
  INTERNAL_SIZE_T      mchunk_size;       /* 當前 chunk 的大小 */

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

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```

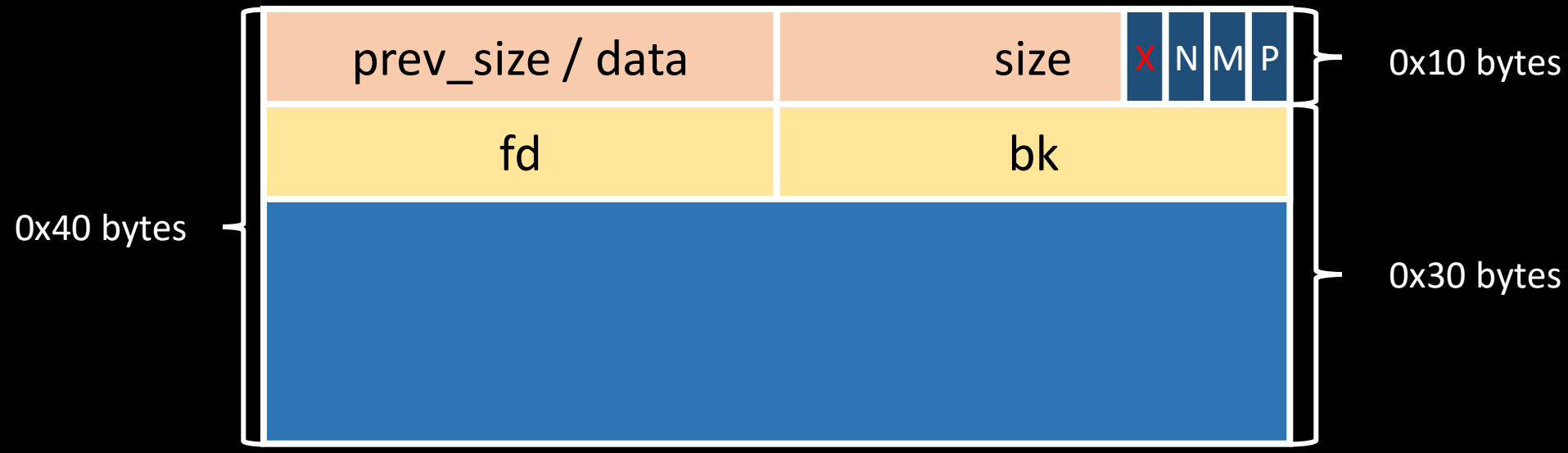
Allocated Chunk

- malloc(0x30)
- chunk size 0x40 bytes (header 0x10 bytes + data 0x30 bytes)

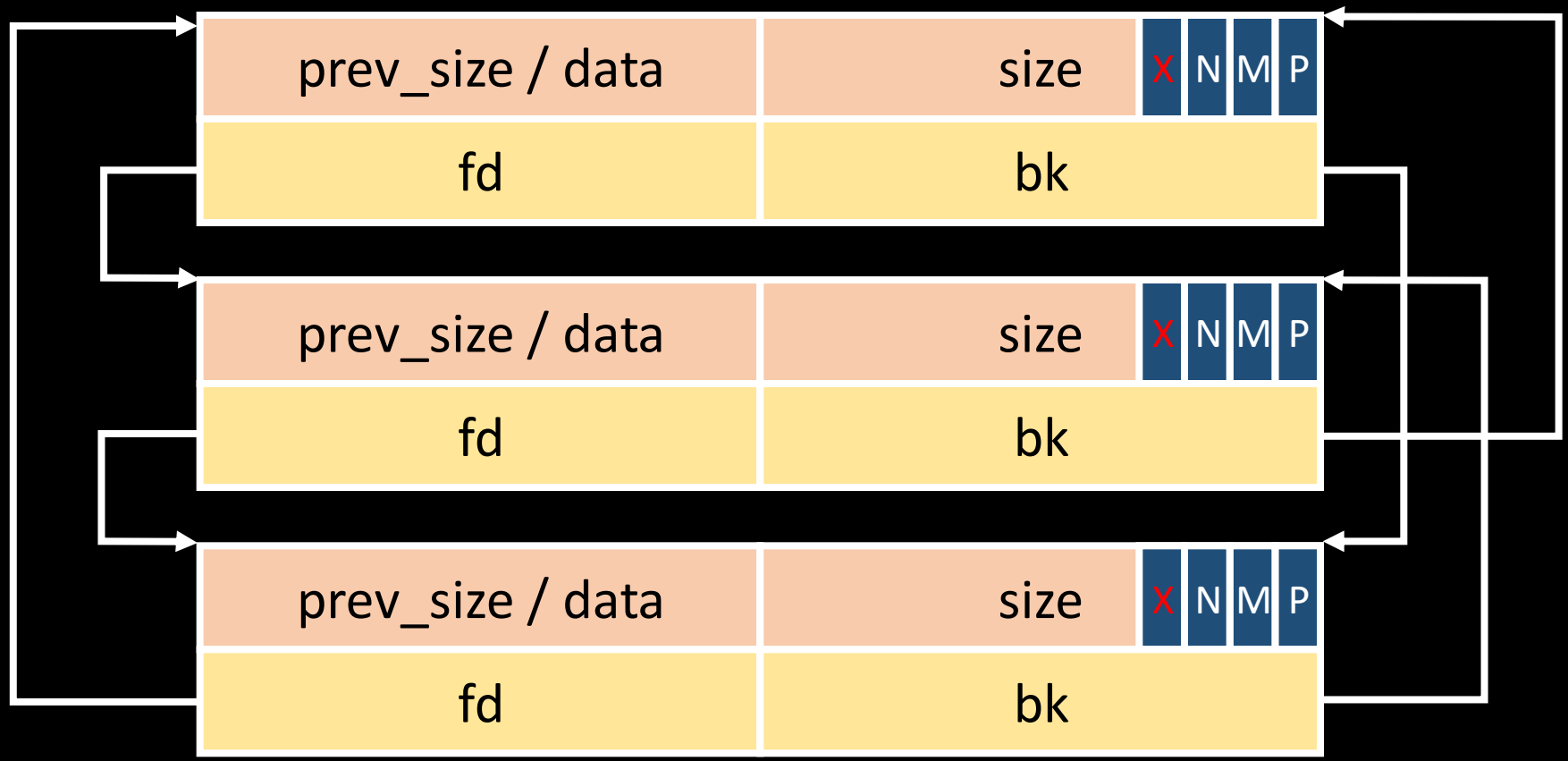


Freed chunk

- data 前 0x10 bytes 變成 doubly linked list 的 fd 和 bk
- bin 就是一個 freed chunk 組成的 linked list

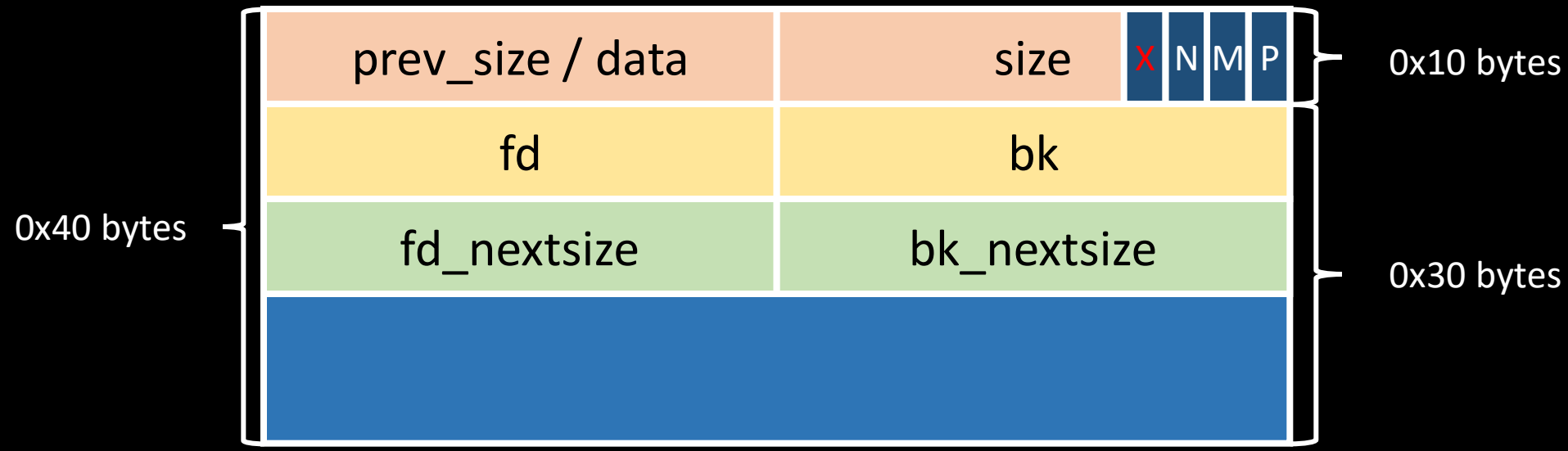


Freed chunk


prev_size / data
size
X
N
M
P
fd
bk
prev_size / data
size
X
N
M
P
fd
bk
prev_size / data
size
X
N
M
P
fd
bk

Freed chunk

- large bin 由於機制需求會多 fd_nextsize 和 bk_nextsize



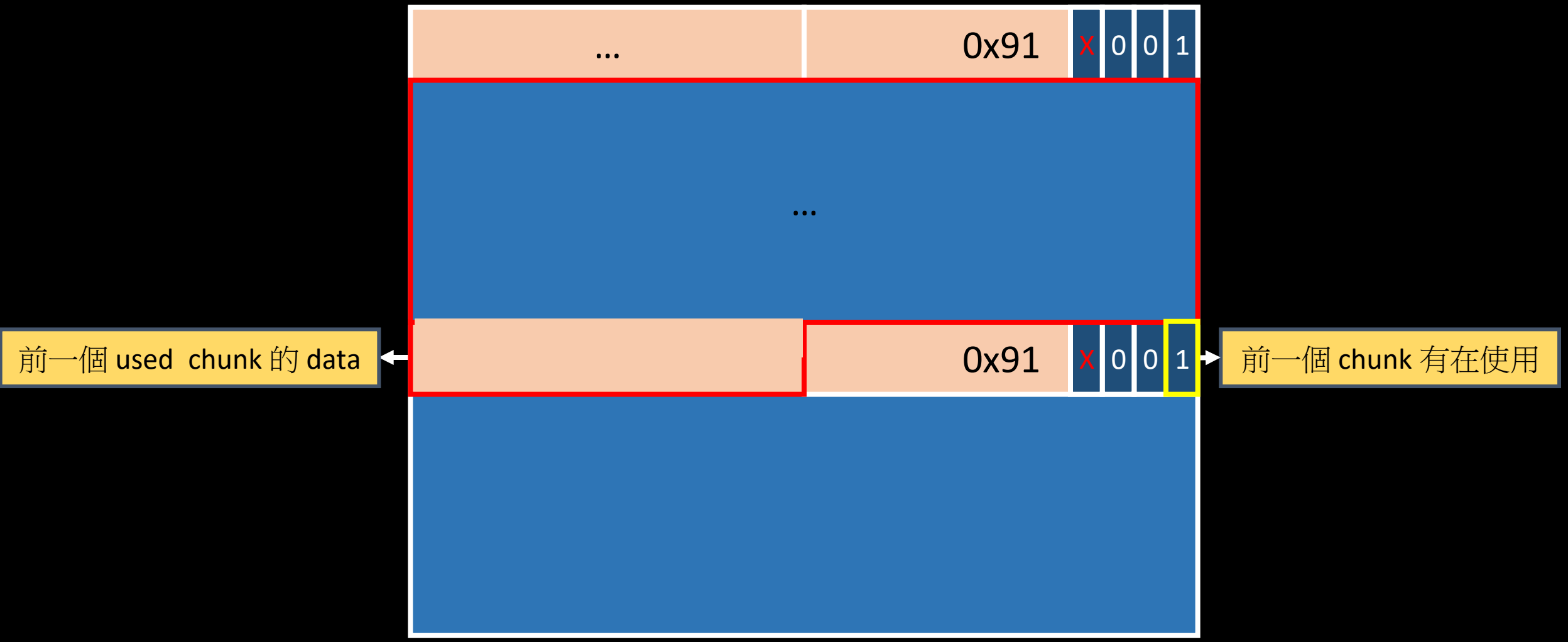
Chunk

- prev_size / data – 依照前一塊 chunk 有所不同
  - freed chunk 會記錄其大小 (prev_size)
  - allocated chunk 則是作為其資料使用 (data)
- 所以 chunk 的 data 實際上會多下個 chunk 的前 8 bytes 可以使用
  - malloc(0x28) 時會chunk size 是 0x28 – 0x8 (data) + 0x10 (header) 再向上對齊 0x10 -> 0x30 bytes
  - malloc(0x29) -> chunk size 是 0x29 – 0x8 + 0x10 = 0x31 向上對齊 0x10 -> 0x40 bytes

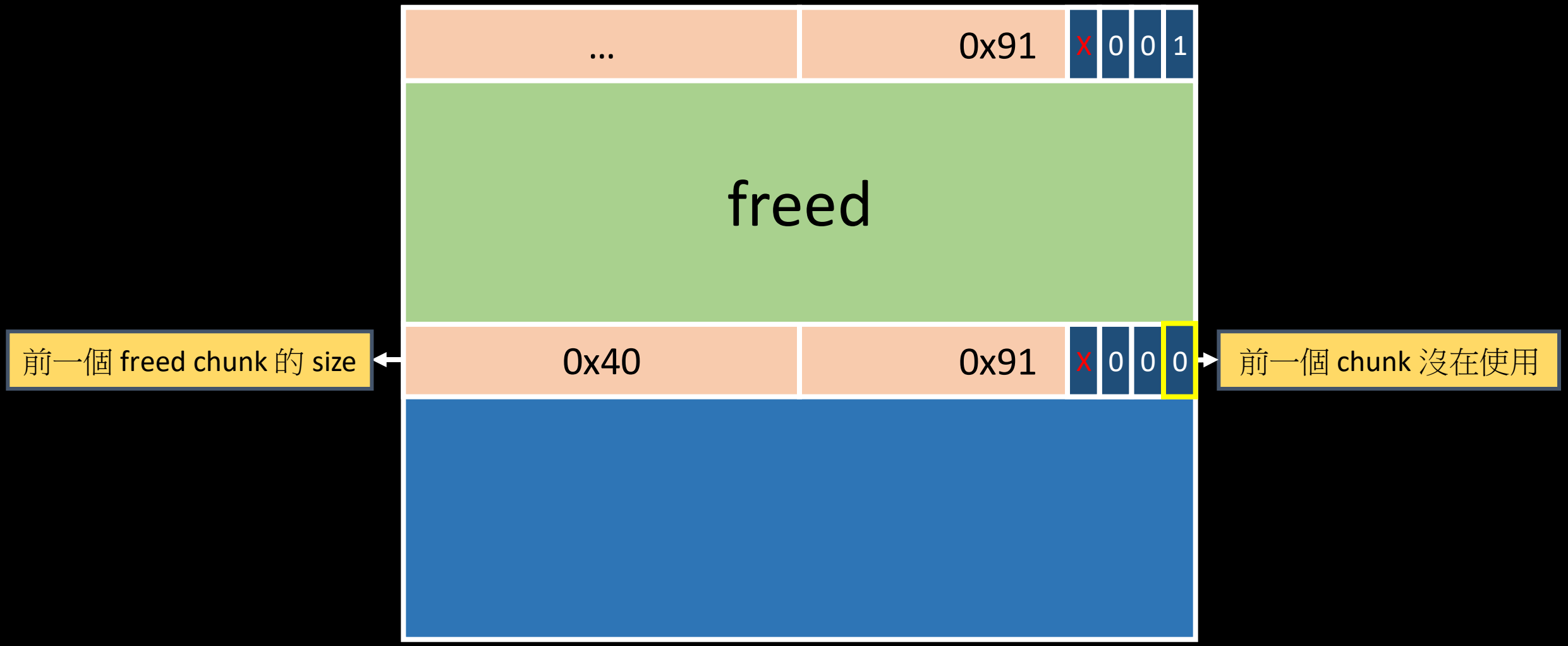
Chunk

- size – 當前 chunk size (包含 header)，對齊 0x10 (0b10000)
- 後面沒用的 4 bits 作為 metadata 使用 (X, N, M, P)
  - X – 沒用
  - N (NON_MAIN_ARENA) – 是否不在 main_arena
  - M (IS_MMAPED) – chunk 是否為 mmap 建立
  - P (PREV_INUSE) – 上一塊 chunk 是否正在使用

Chunk



Chunk


...
0x91
X
0
0
1
freed
前一個 freed chunk 的 size
0x40
0x91
X
0
0
0
前一個 chunk 沒在使用

Chunk

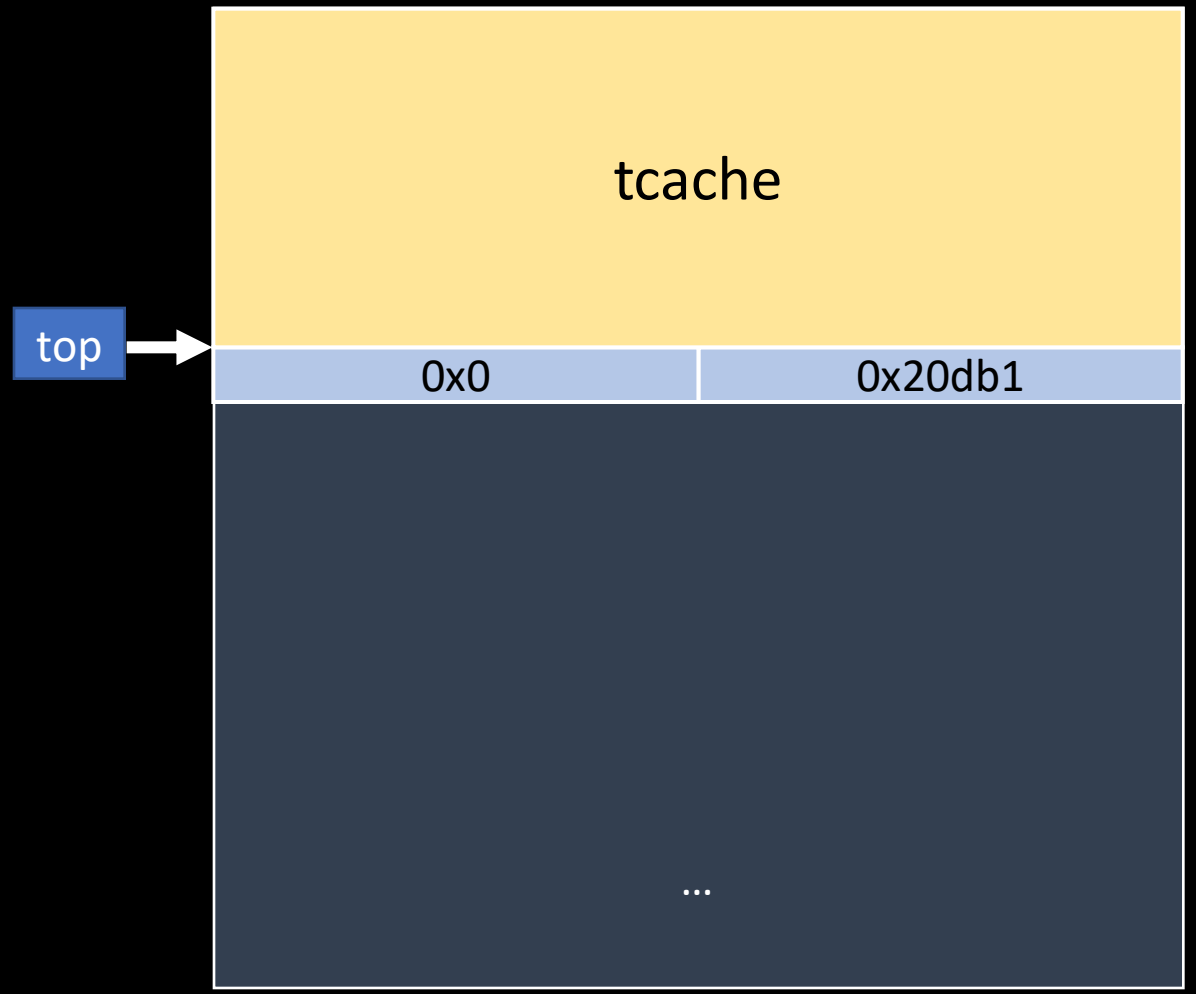
- fd – 指向 **同一個 bin** 的後一塊 freed chunk (forward)
- bk - 指向 **同一個 bin** 的前一塊 freed chunk (back)
- fd_nextsize - 在 **large bin** 的 freed chunk 指向後一個 大小 的 freed chunk
- bk_nextsize - 在 **large bin** 的 freed chunk 指向前一個 大小 的 freed chunk

Top chunk

- 結構為 struct malloc_chunk *
- 第一次 malloc 時會把 sbrk 回傳的空間設上 chunk head
- top 就會指到這個 chunk
- malloc 就會從 top 切一塊 chunk 給使用者，接著 top 會改指到剩下的空間
- top 儲存在 arena 內

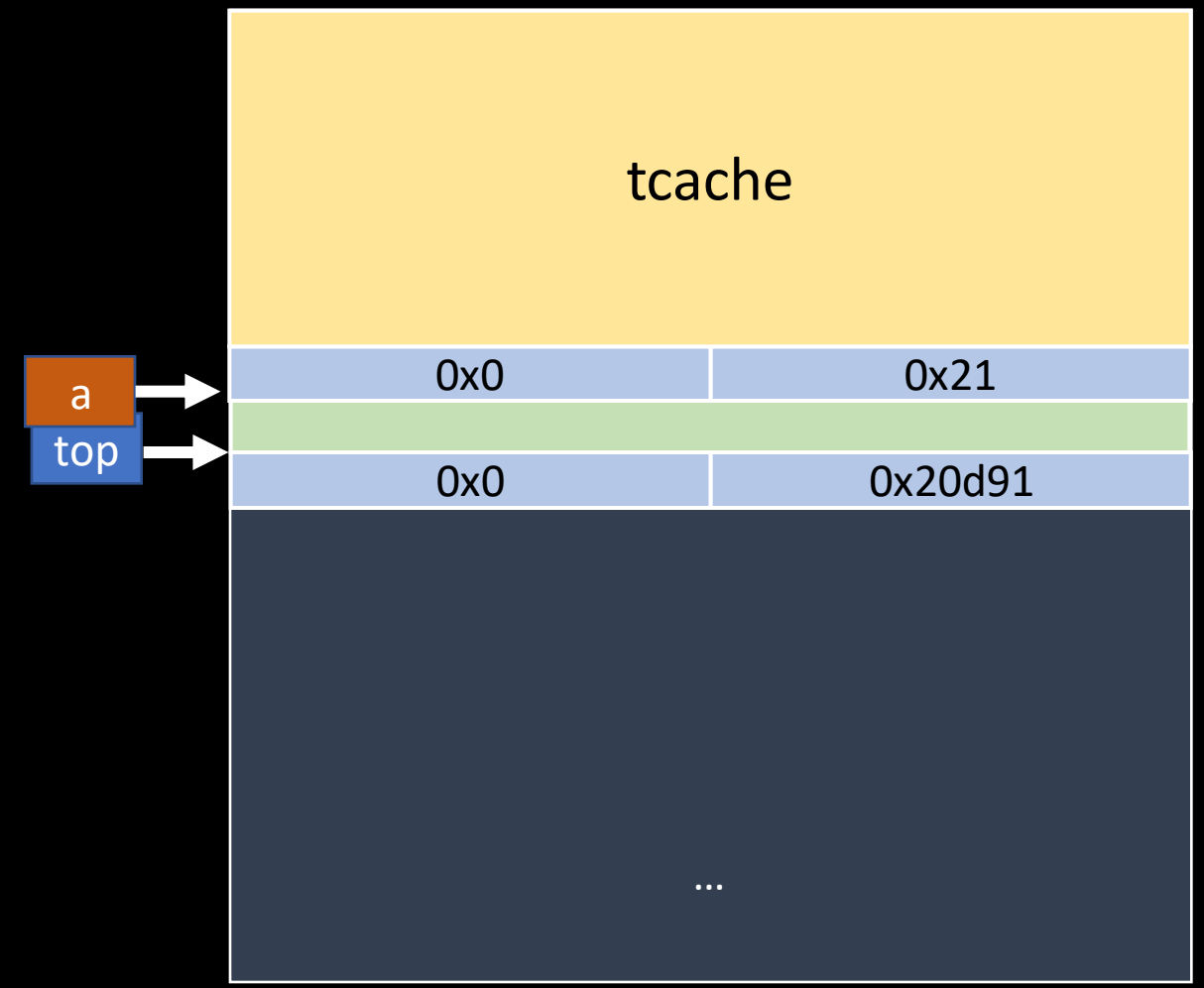
Top chunk

```
char *a = malloc(0x18);
char *b = malloc(0x28);
char *c = malloc(0x48);
```



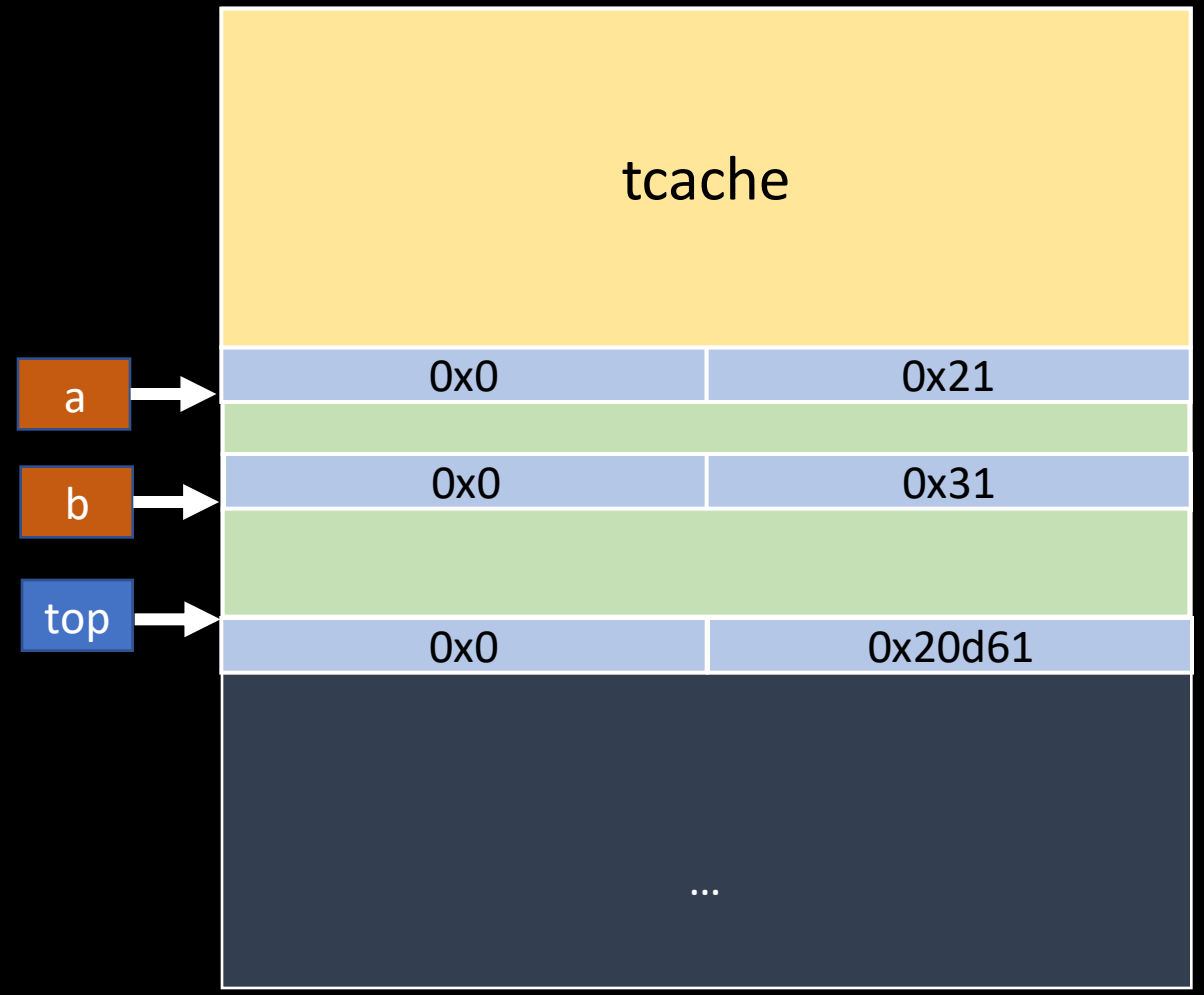
Top chunk

```
char *a = malloc(0x18);
char *b = malloc(0x28);
char *c = malloc(0x48);
```



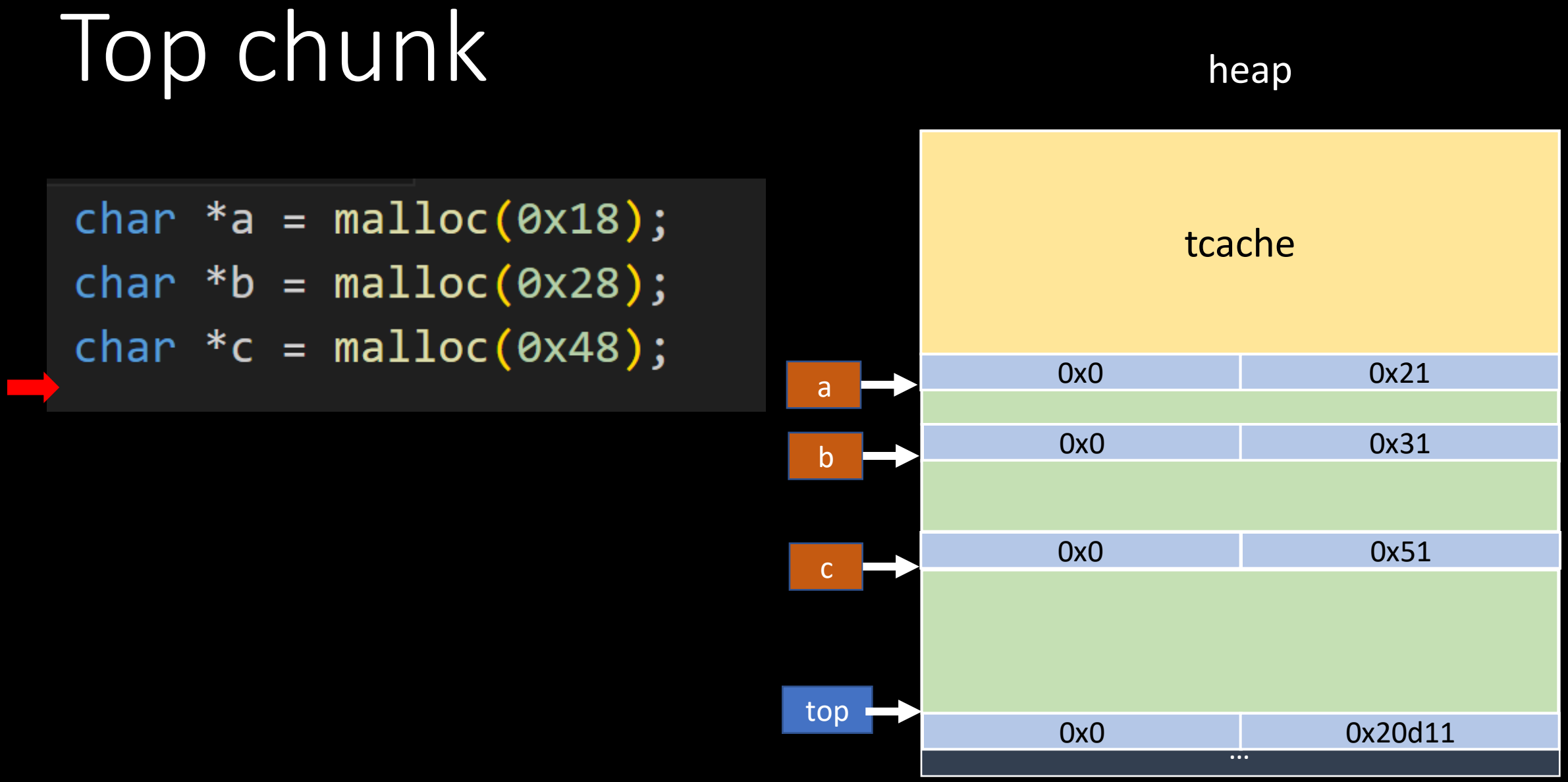
Top chunk

```
char *a = malloc(0x18);
char *b = malloc(0x28);
char *c = malloc(0x48);
```



Top chunk

```
char *a = malloc(0x18);
char *b = malloc(0x28);
char *c = malloc(0x48);
```



Bin

- Linked list 紀錄 freed chunk，依照機制和 chunk 大小分為五種 bin
  - Tcachce / 0x20 – 0x410 / 1st / Singly Linked list / FILO
  - Fastbin / 0x20 – 0x80 / 2nd / Singly Linked list / FILO
  - Small bin / 0x20 – 0x3f0 / 3rd / Doubly Linked list / FIFO
  - Large bin / >= 0x400 / 5th / Doubly Linked list /
  - Unsorted bin / >= 0x90 / 4th / Doubly Linked list / FIFO
- Tcache / Fastbin / Small bin 每 0x10 會有一個 subbin，也就是 0x20, 0x30, 0x40, ... 各自都會有一個 linked list head

Tcache

- Tcache 為 chunk 的 cache
- Size 為 0x20 – 0x410，Singly linked list 所以是 FILO
- 每個 entry (subbin) 最多紀錄 7 個 freed chunk
- 優先度最高，tcache 用完才會用其他 bin
- Free 時不在將下一個 chunk 清空 PREV_INUSE bit

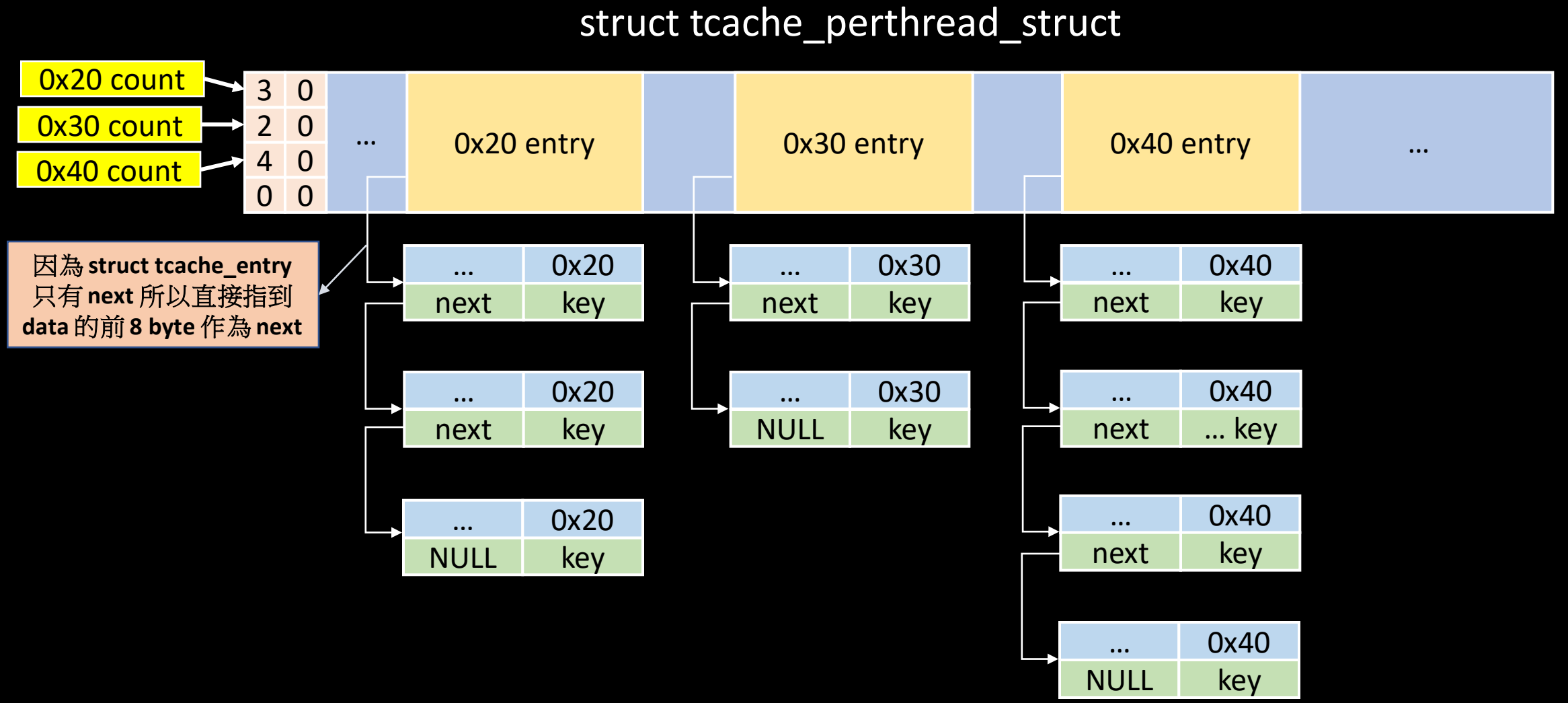
Tcache

- 而外以 struct tcache_entry 結構來建構 linked list
- struct tcache_perthread_struct 結構來記錄 entry 和其 count
- 在第一次 malloc 時會 tcache_init 在 heap 最前面 malloc 一個 struct tcache_perthread_struct 來儲存 tcache

Tcache

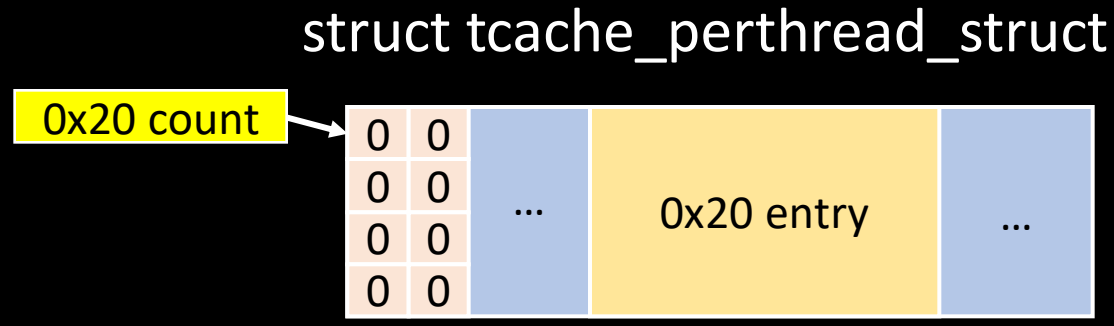
```
typedef struct tcache_entry
{
  struct tcache_entry *next;
  /* This field exists to detect double frees.  */
  struct tcache_perthread_struct *key;
} tcache_entry;

typedef struct tcache_perthread_struct
{
  uint16_t counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

Tcache


struct tcache_perthread_struct
0x20 count
0x30 count
0x40 count
3 0
2 0
4 0
0 0
...
0x20 entry
0x30 entry
0x40 entry
...
因為 struct tcache_entry 只有 next 所以直接指到 data 的前 8 byte 作為 next
... 0x20
next key
... 0x20
next key
... 0x20
NULL key
... 0x30
next key
... 0x30
NULL key
... 0x40
next key
... 0x40
next ... key
... 0x40
next key
... 0x40
NULL key

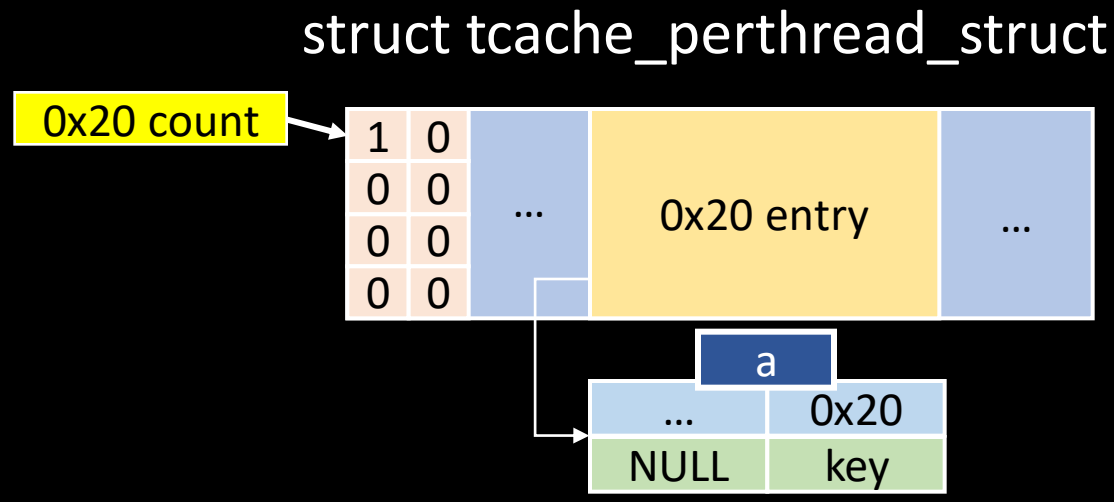
Tcache

```
a = malloc(0x18);
b = malloc(0x18);
c = malloc(0x18);
d = malloc(0x18);

free(a);
malloc(0x18);
free(b);
free(c);
free(d);
```



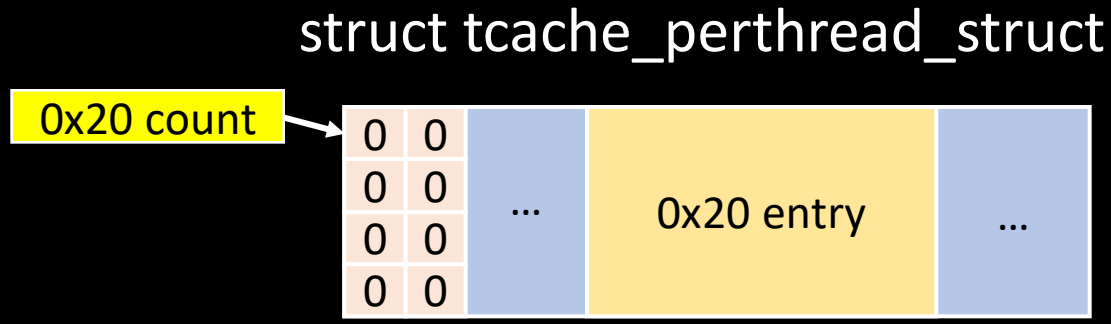
Tcache

```
a = malloc(0x18);
b = malloc(0x18);
c = malloc(0x18);
d = malloc(0x18);

free(a);
malloc(0x18);
free(b);
free(c);
free(d);
```



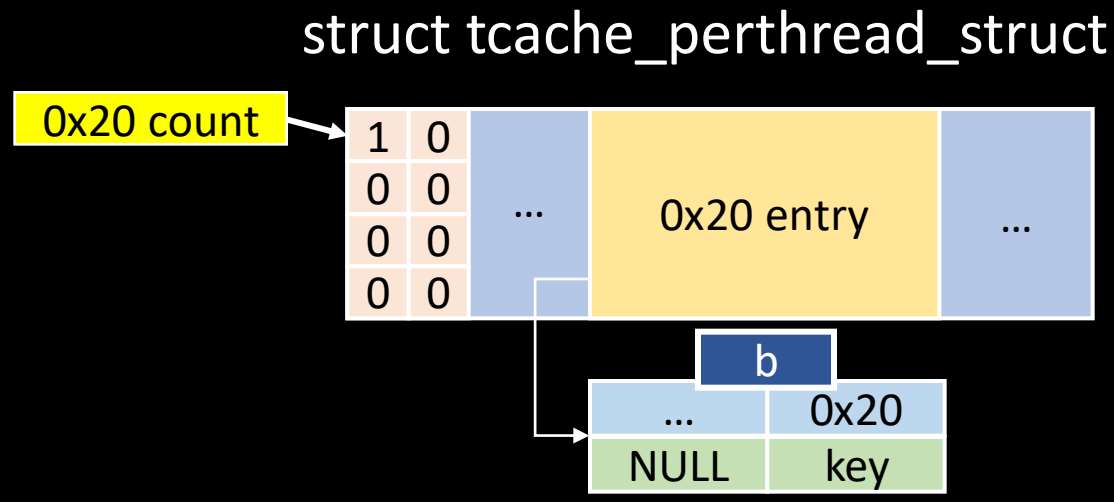
Tcache

```
a = malloc(0x18);
b = malloc(0x18);
c = malloc(0x18);
d = malloc(0x18);

free(a);
malloc(0x18);
free(b);
free(c);
free(d);
```



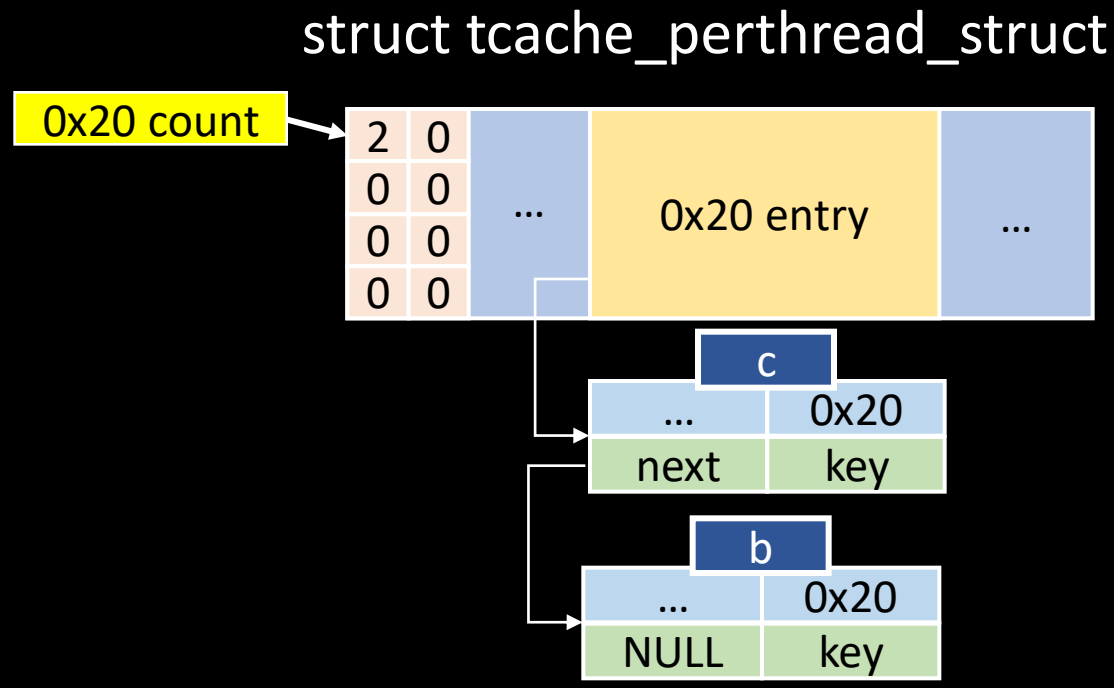
Tcache

```
a = malloc(0x18);
b = malloc(0x18);
c = malloc(0x18);
d = malloc(0x18);

free(a);
malloc(0x18);
free(b);
free(c);
free(d);
```



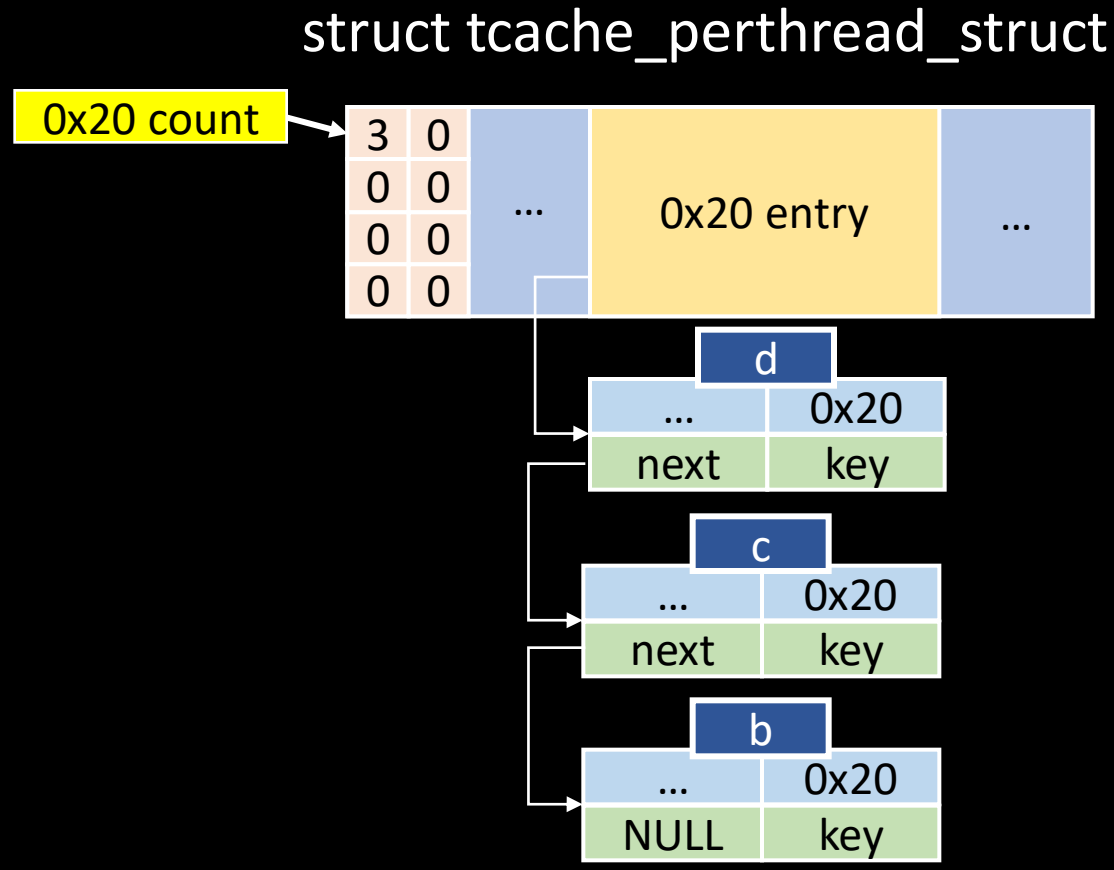
Tcache

```
a = malloc(0x18);
b = malloc(0x18);
c = malloc(0x18);
d = malloc(0x18);

free(a);
malloc(0x18);
free(b);
free(c);
free(d);
```



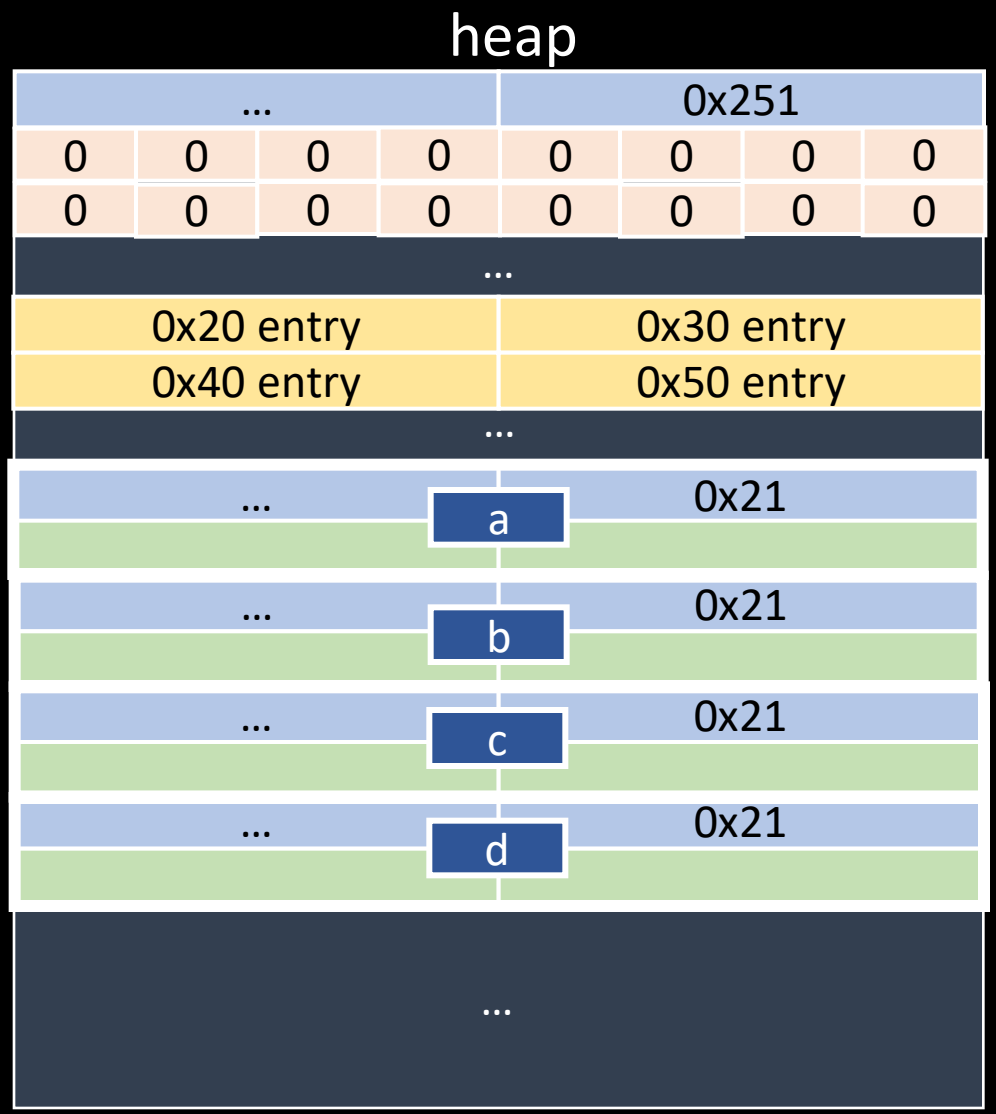
Tcache

```
a = malloc(0x18);
b = malloc(0x18);
c = malloc(0x18);
d = malloc(0x18);

free(a);
malloc(0x18);
free(b);
free(c);
free(d);
```



Tcache

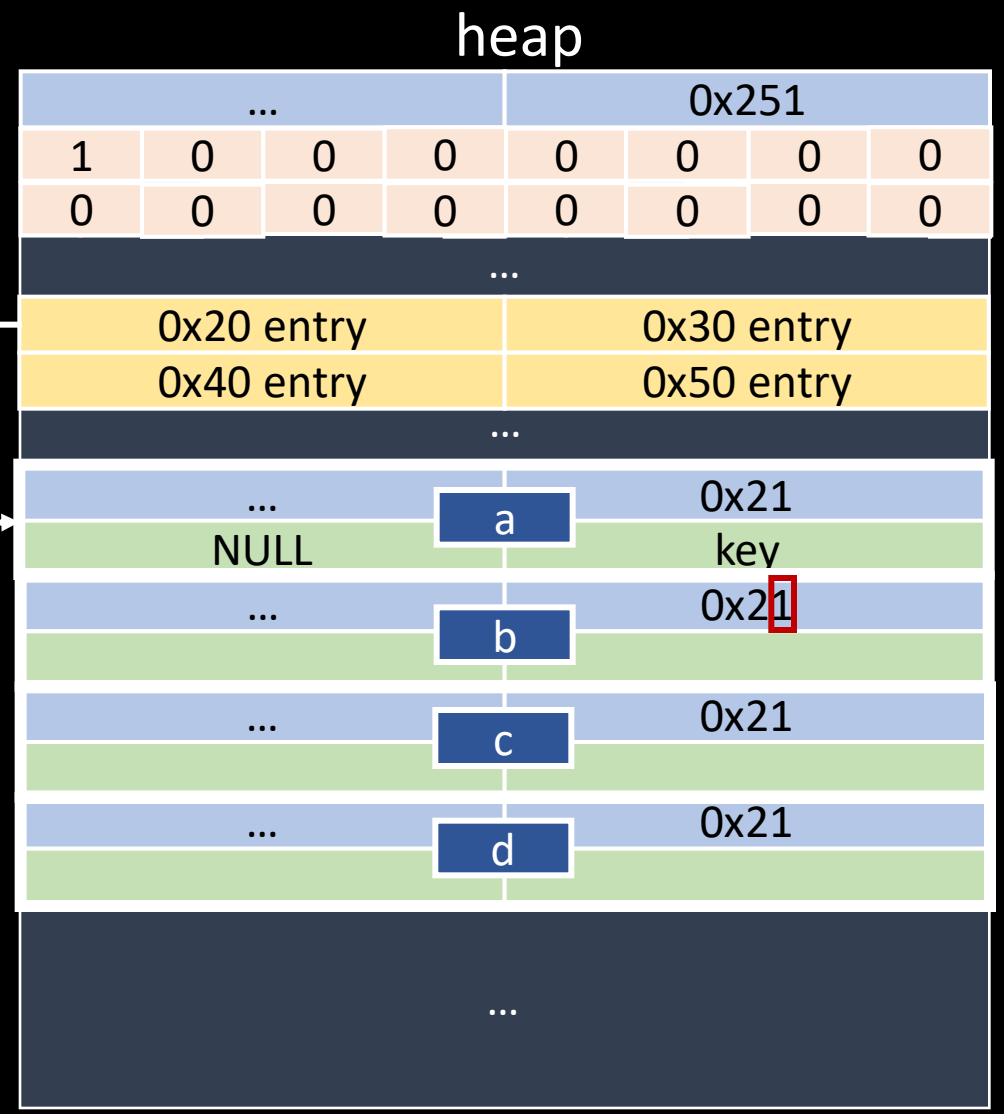
```
a = malloc(0x18);
b = malloc(0x18);
c = malloc(0x18);
d = malloc(0x18);

free(a);
malloc(0x18);
free(b);
free(c);
free(d);
```



Tcache

```
a = malloc(0x18);
b = malloc(0x18);
c = malloc(0x18);
d = malloc(0x18);

free(a);
malloc(0x18);
free(b);
free(c);
free(d);
```

不會清空 b 的 PREV_INUSE bit

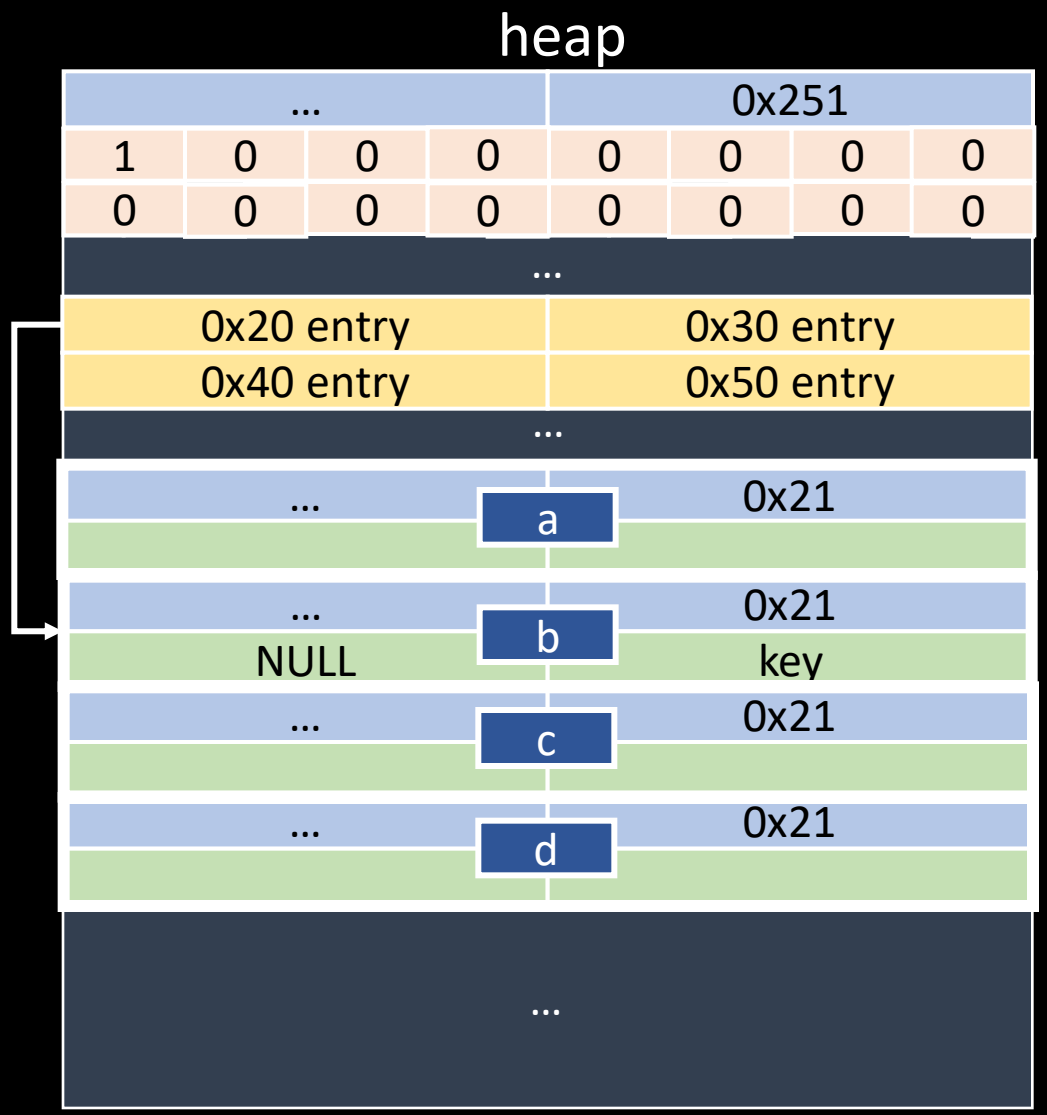


Tcache

```
a = malloc(0x18);
b = malloc(0x18);
c = malloc(0x18);
d = malloc(0x18);

free(a);
malloc(0x18);
free(b);
free(c);
free(d);
```

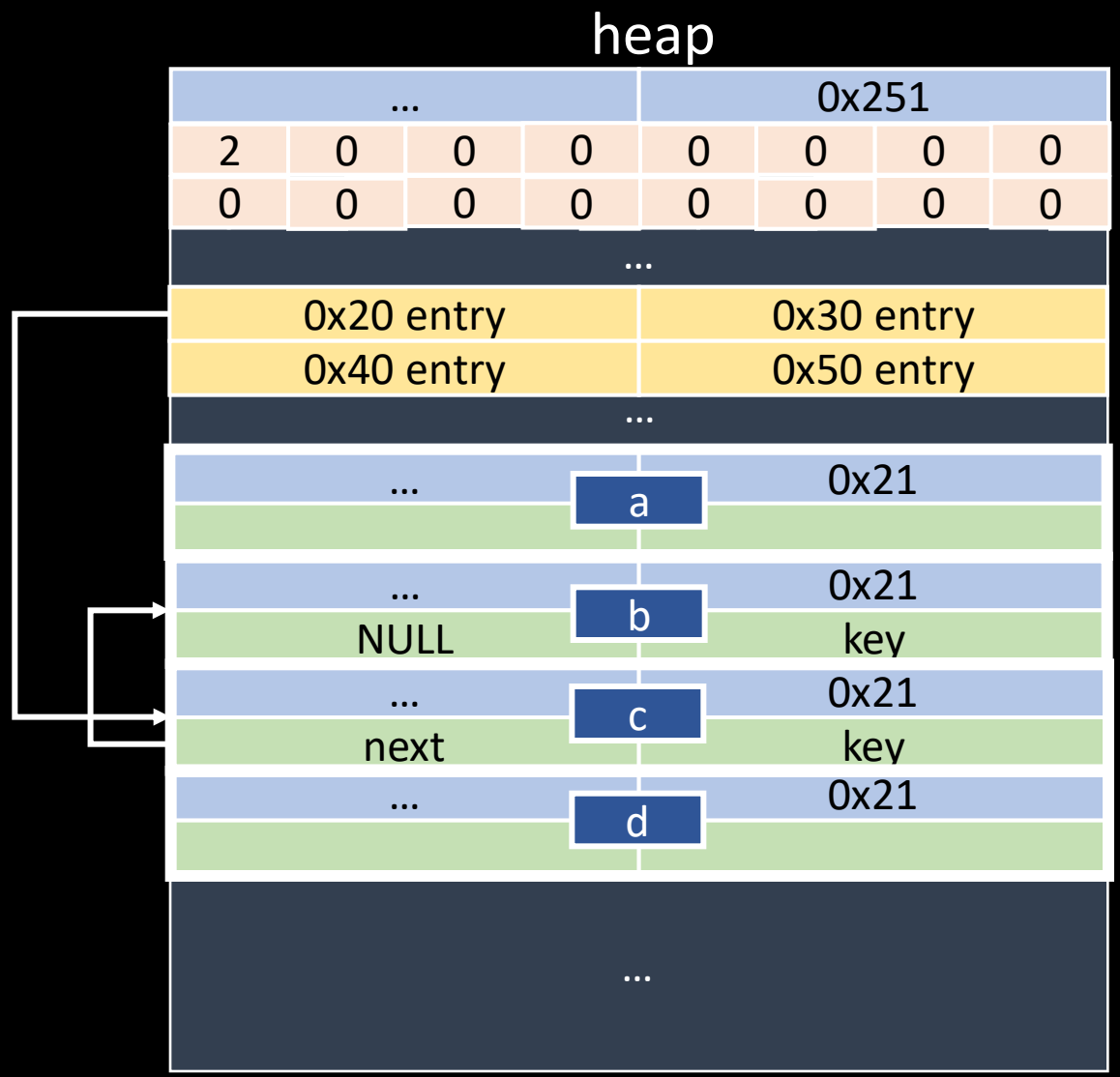
Tcache

```
a = malloc(0x18);
b = malloc(0x18);
c = malloc(0x18);
d = malloc(0x18);

free(a);
malloc(0x18);
free(b);
free(c);
free(d);
```



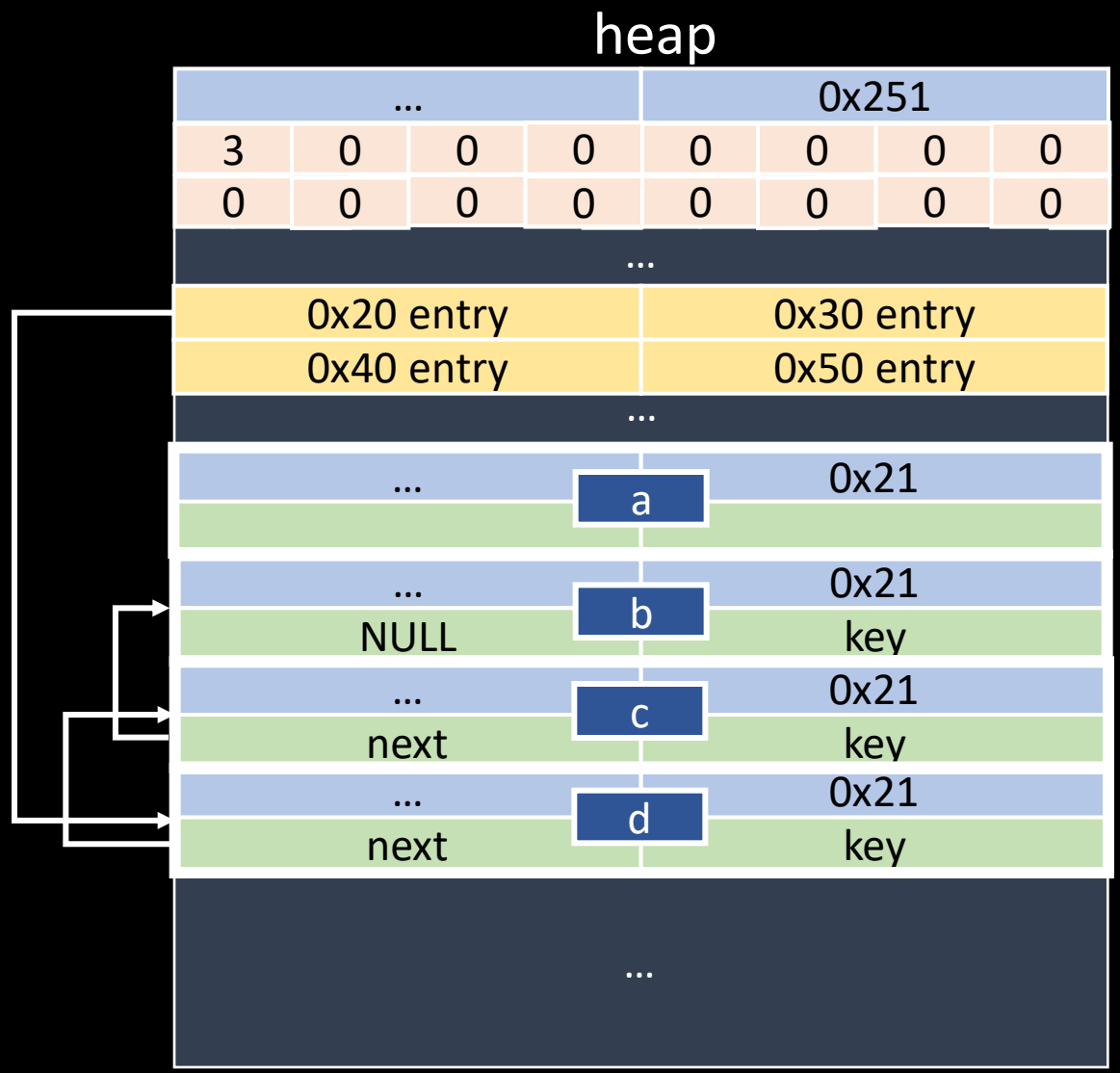
Tcache

```
a = malloc(0x18);
b = malloc(0x18);
c = malloc(0x18);
d = malloc(0x18);

free(a);
malloc(0x18);
free(b);
free(c);
free(d);
```



Tcache

```
a = malloc(0x18);
b = malloc(0x18);
c = malloc(0x18);
d = malloc(0x18);

free(a);
malloc(0x18);
free(b);
free(c);
free(d);
```



TCACHE DEMO

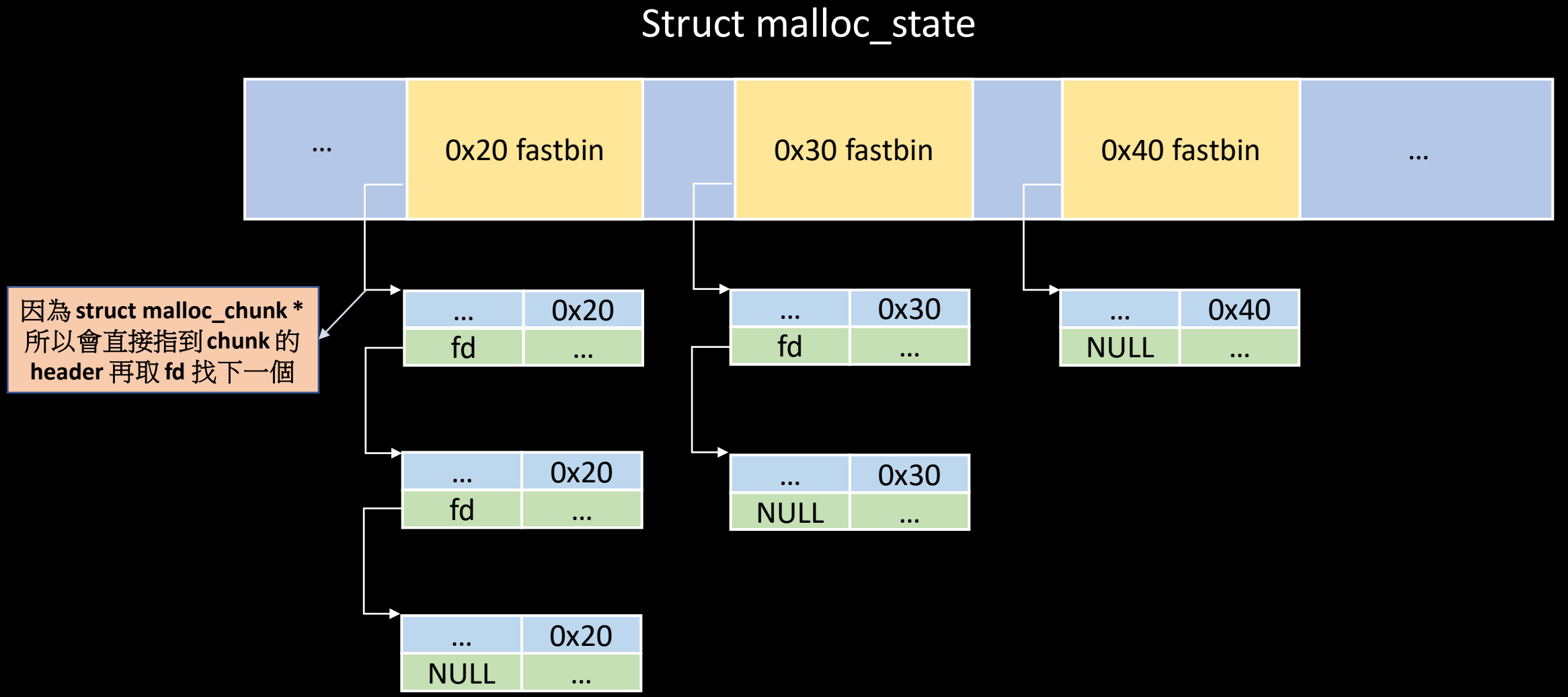
Fastbin

- 紀錄 size 較小的 freed chunk
- Size 為 0x20 – 0x80 ，Singly linked list 所以是 FILO
- Free 時不會清空下一個 chunk 的 PREV_INUSE
- Subbins 儲存在 arena 內
- 結構為 struct malloc_chunk * []

Fastbin

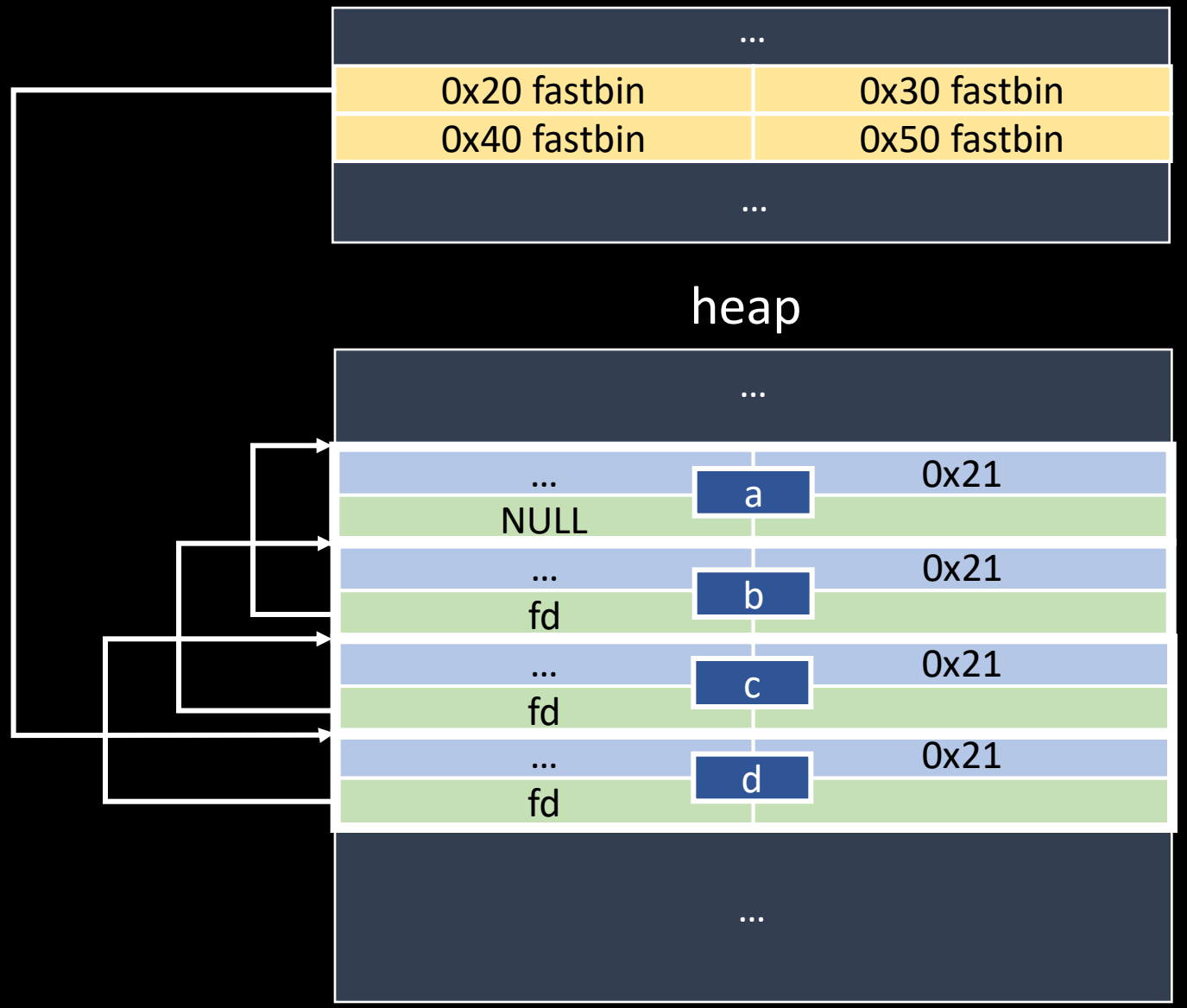
```
struct malloc_state
{
  ...
  int have_fastchunks;

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];

  ...
};
```

Fastbin


Struct malloc_state
...
0x20 fastbin
0x30 fastbin
0x40 fastbin
...
因為 struct malloc_chunk * 所以會直接指到 chunk 的 header 再取 fd 找下一個
... 0x20
fd ...
... 0x20
fd ...
... 0x20
NULL ...
... 0x30
fd ...
... 0x30
NULL ...
... 0x40
NULL ...

Fastbin

- 塞滿 tcache
- free(a)
- free(b)
- free(c)
- free(d)



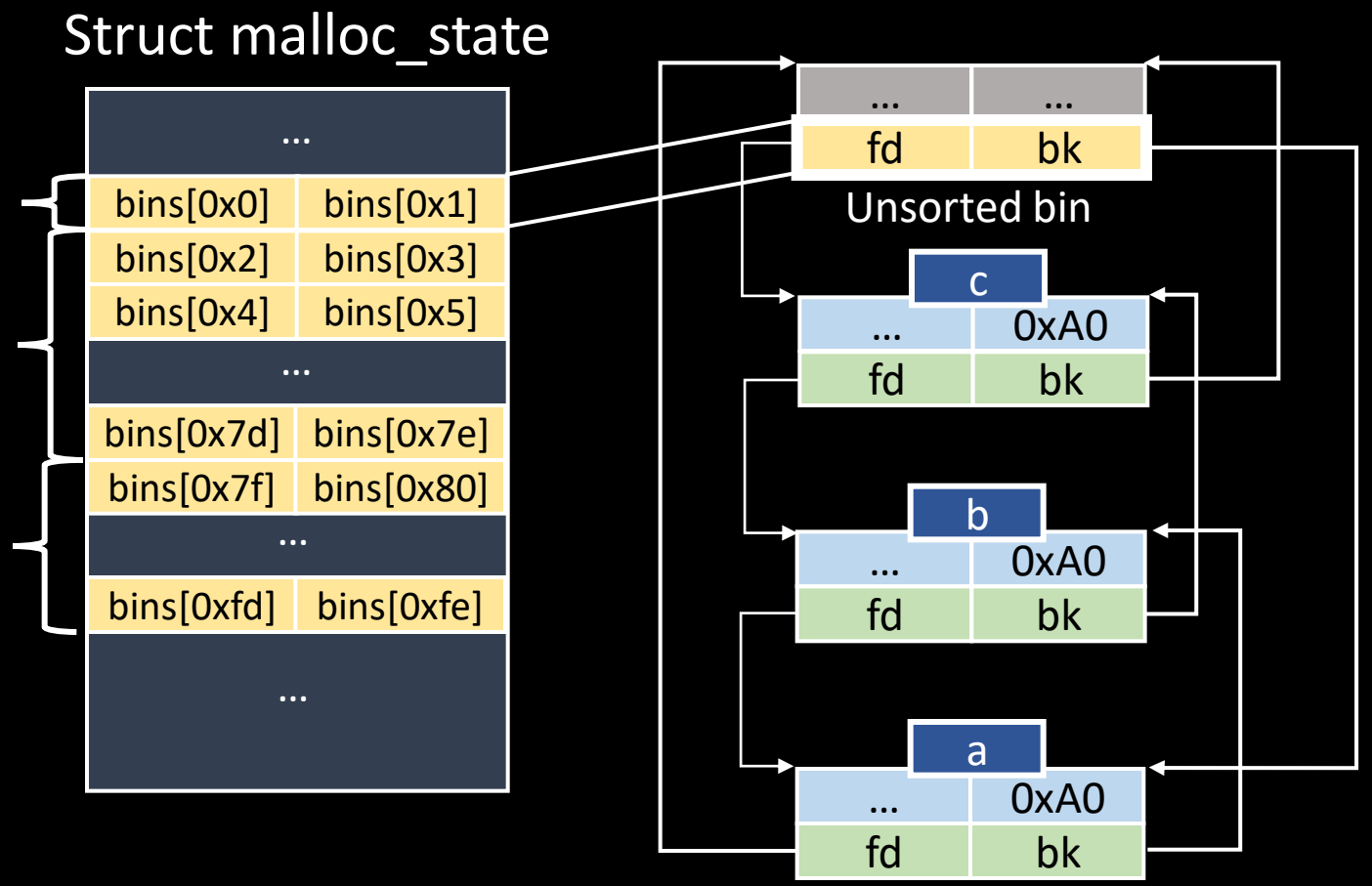
FASTBIN DEMO

Unsorted bin

- Doubly linked list
- Free 不是 tcache 和 fastbin 的 chunk 就會先放到 unsorted bin
- 只有一個 bin
- 儲存在 arena 內的 bins 裡的第一個 struct malloc_chunk * pair，pair 初始值會指到自己
- Free 時會把 freed chunk 放進 bin 的 fd

Unsorted bin

- a = malloc(0x90); malloc(0x10);
- b = malloc(0x90) ; malloc(0x10);
- c = malloc(0x90) ; malloc(0x10);
- 填滿 tcache
- free(a)
- free(b)
- free(c)

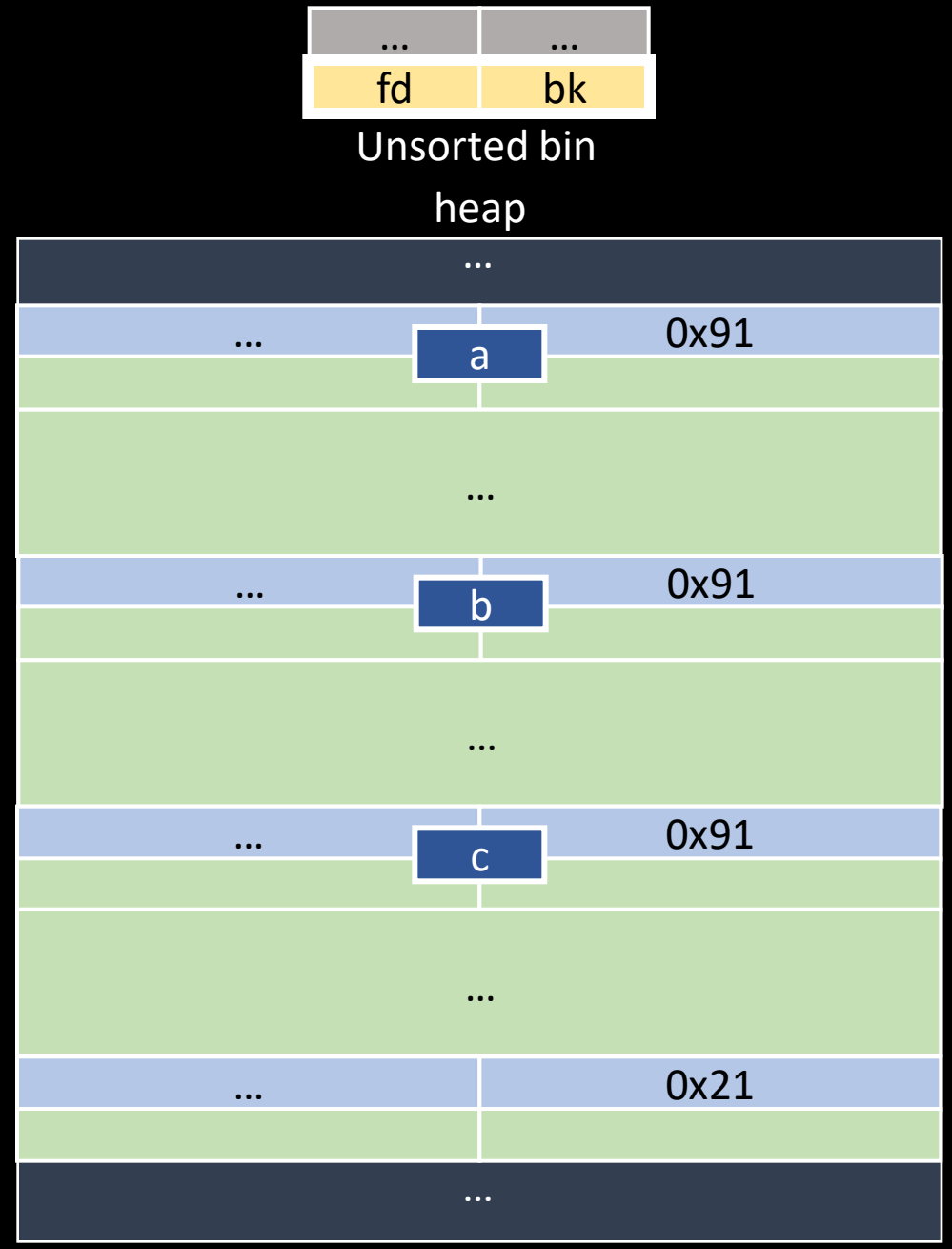


Consolidate

- Free 時會檢查上一塊和下一塊 chunk
  - 如果是 freed chunk 就會將其 unlink 在 Consolidate 成一個大的 chunk，然後再放進 unsorted bin
  - 如果下一塊 chunk 是 top chunk 就會 Consolidate 進 top chunk 裡
  - 檢查方式是看 PREV_INUSE 所以 fastbin 和 tcache 不會 Consolidate

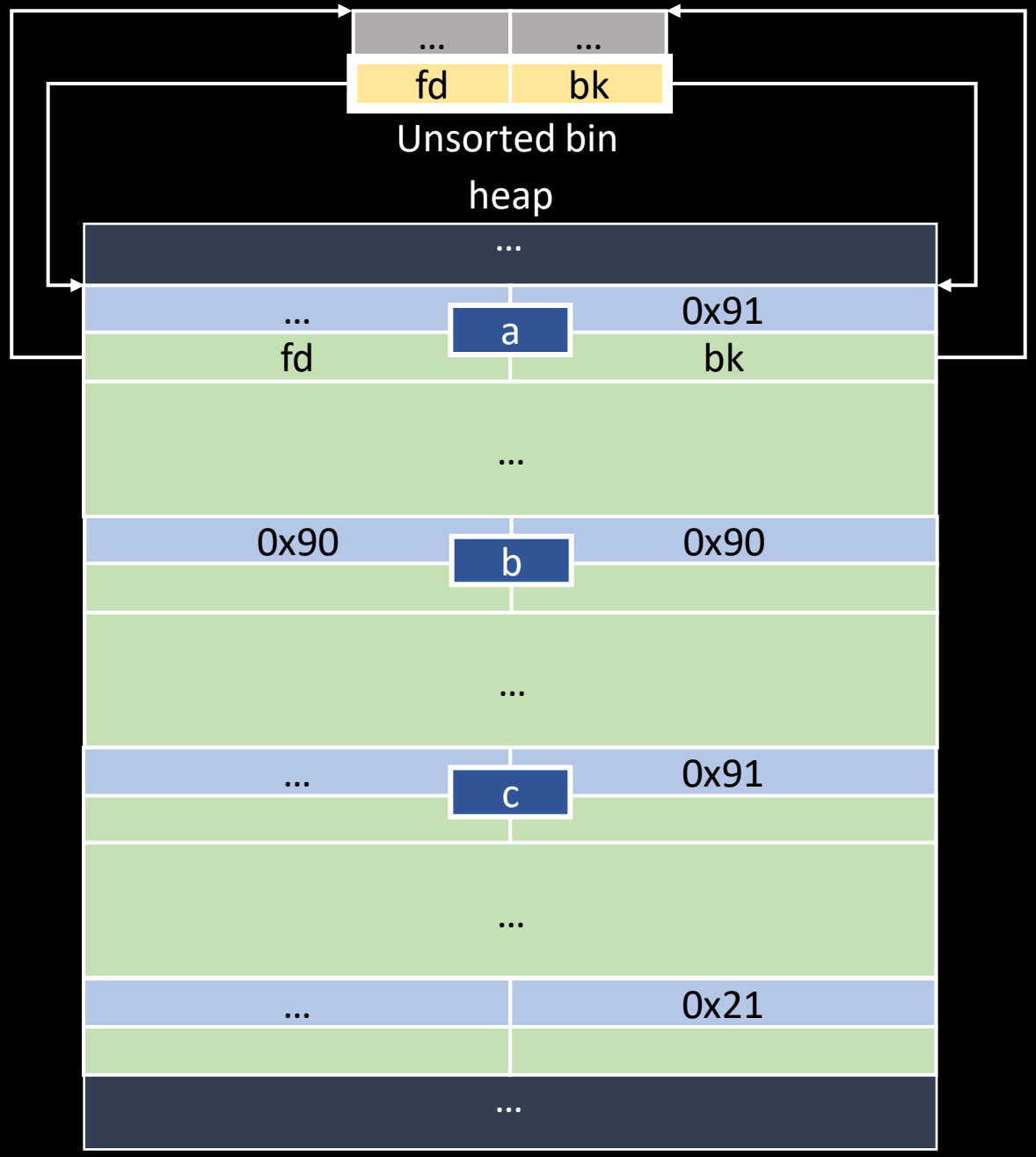
Consolidate

- a = alloc(0x90)
- b = alloc(0x90)
- c = alloc(0x90); malloc(0x10)
- 填滿 tcache
- free(a)
- free(c)
- free(b)



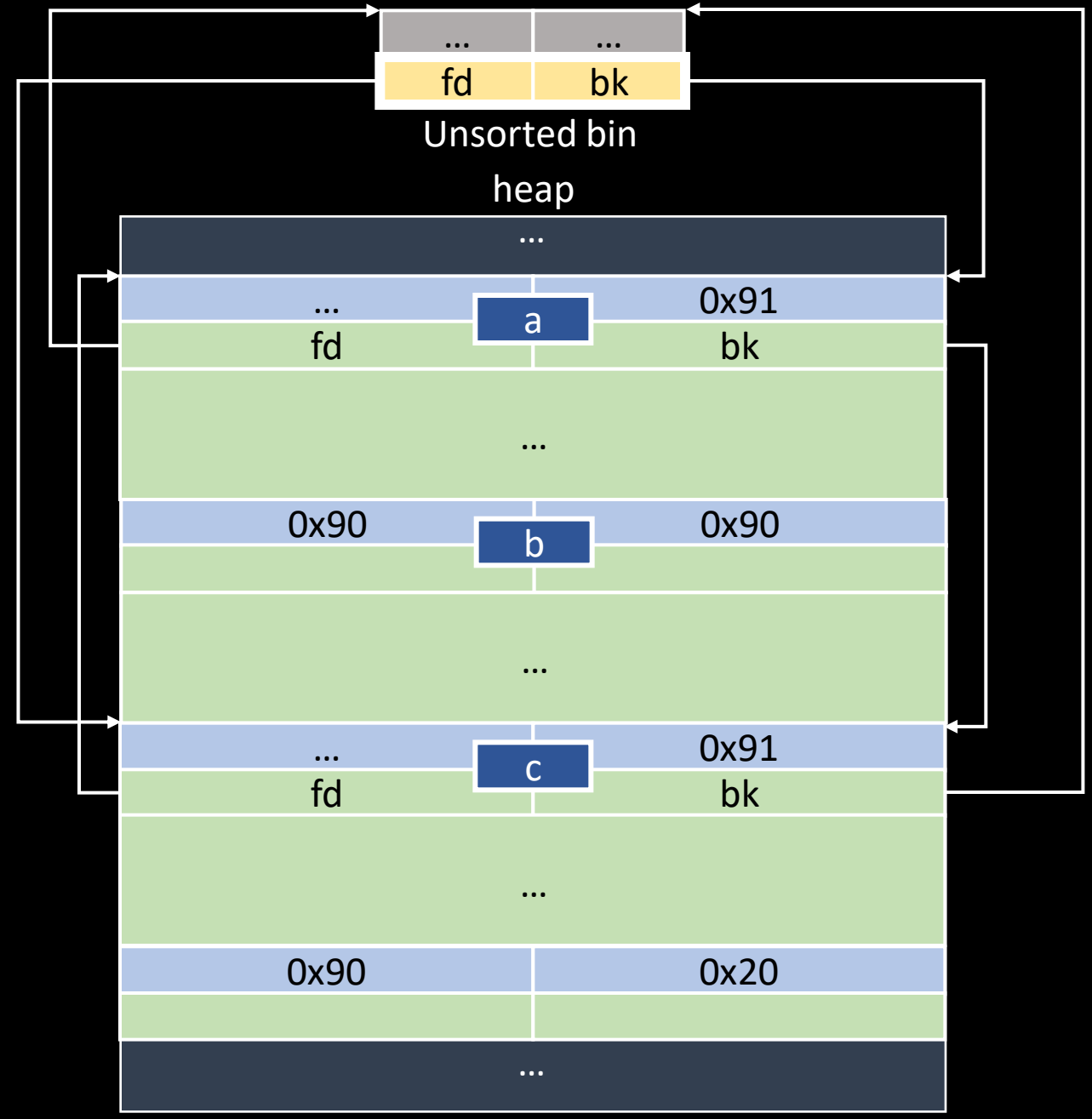
Consolidate

- a = alloc(0x90)
- b = alloc(0x90)
- c = alloc(0x90); malloc(0x10)
- 填滿 tcache
- free(a)
- free(c)
- free(b)



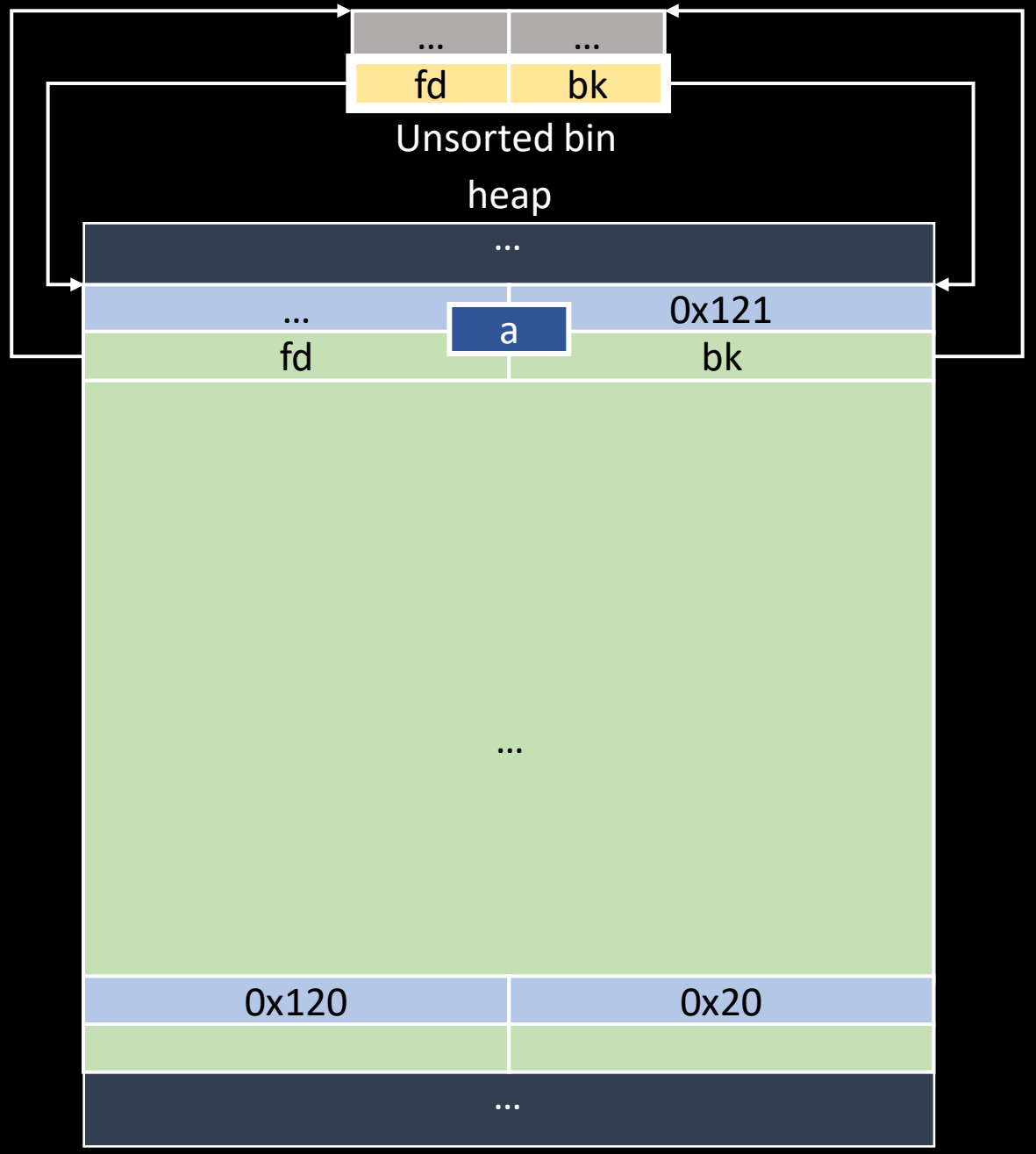
Consolidate

- a = alloc(0x90)
- b = alloc(0x90)
- c = alloc(0x90); malloc(0x10)
- 填滿 tcache
- free(a)
- free(c)
- free(b)



Consolidate

- a = alloc(0x90)
- b = alloc(0x90)
- c = alloc(0x90); malloc(0x10)
- 填滿 tcache
- free(a)
- free(c)
- free(b)



Consolidate DEMO

Unsorted bin

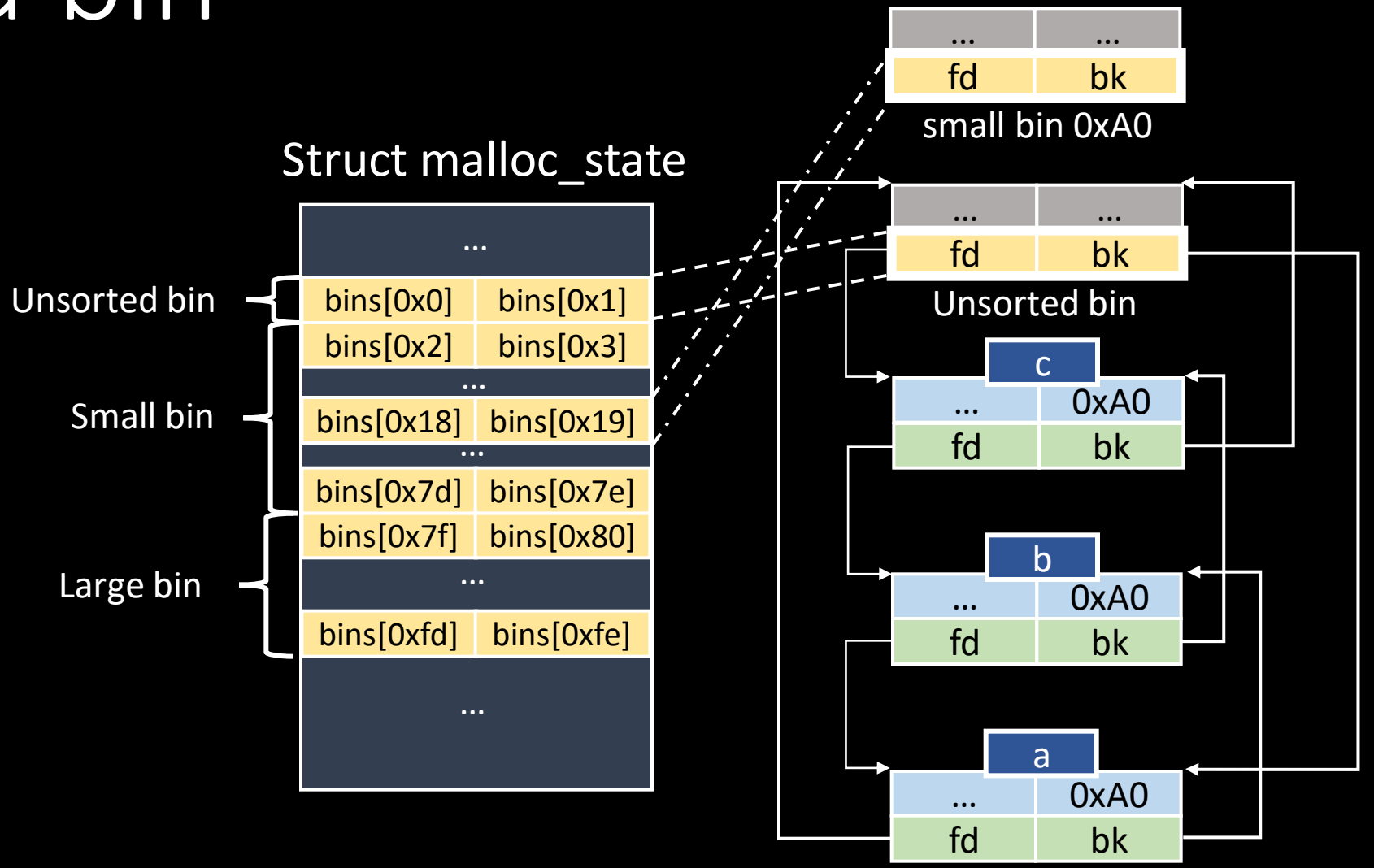
- 在 malloc 時會依序去找 tcache, fastbin, smallbin 有沒有 chunk
- 沒有就會遍歷 Unsorted bin
  - 如果 freed chunk size 符合就會 unlink chunk 並直接回傳
  - 否則嘗試放進對應大小的 tcache，無法則放進 small bin 或 large bin
- 取 chunk 時以 bk 來遍歷
- 遍歷完沒找到符合的 chunk 會從 small / large bin 找比目標大小大的 bin 中最小的 bin 取 chunk 來切，切剩的在放進 unsorted bin，last_reminder 也會指向切剩的 chunk
- 沒有就從 top 切一塊 chunk

Small bin

- Size 為 0x20 – 0x3f0
- 0x20 – 0x80 跟 fastbin 是重疊的
  - 從 unsorted bin 放到 small bin 的 chunk

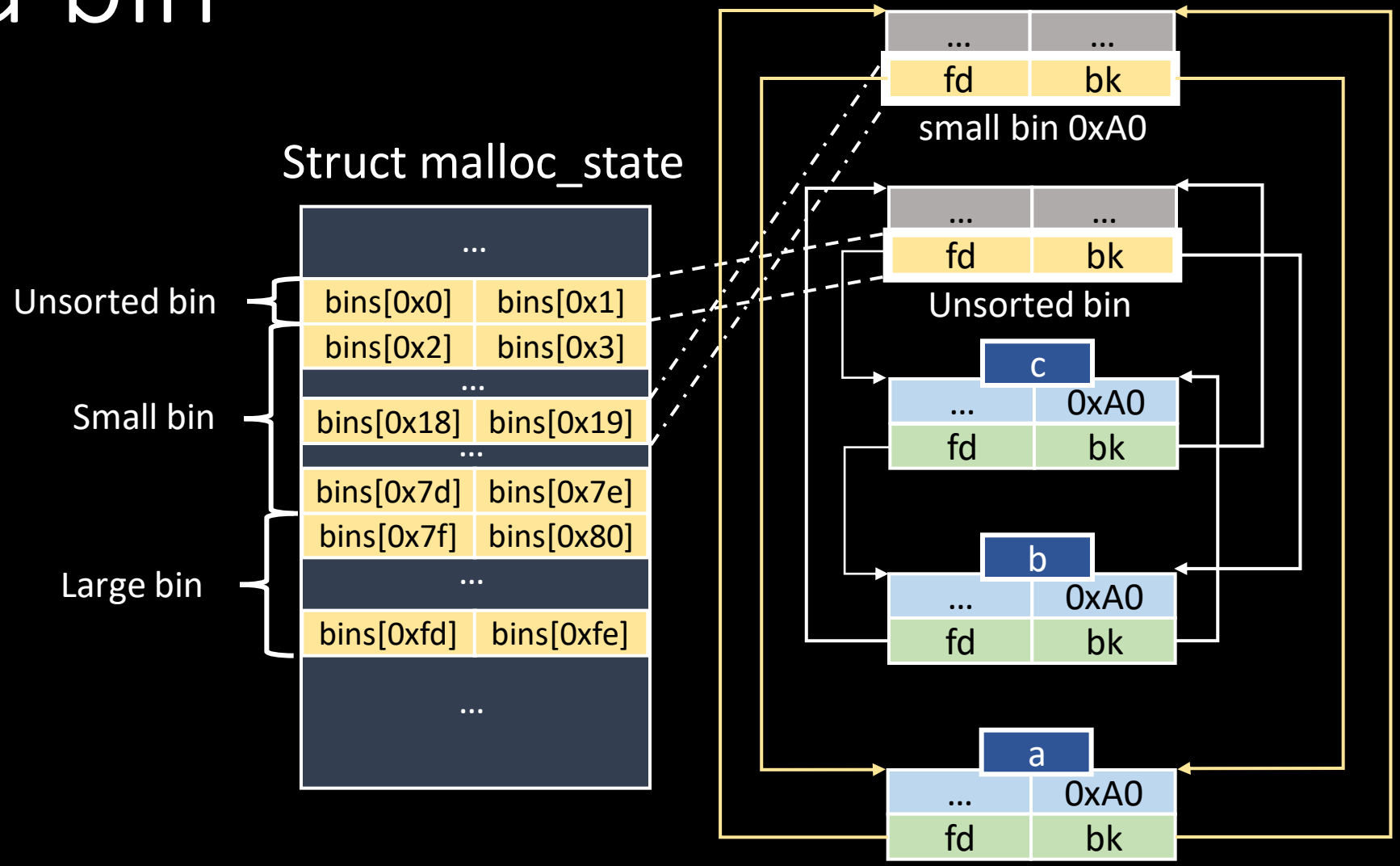
Unsorted bin

- d = malloc(0xA0)
  - Move a
  - Move b
  - Move c



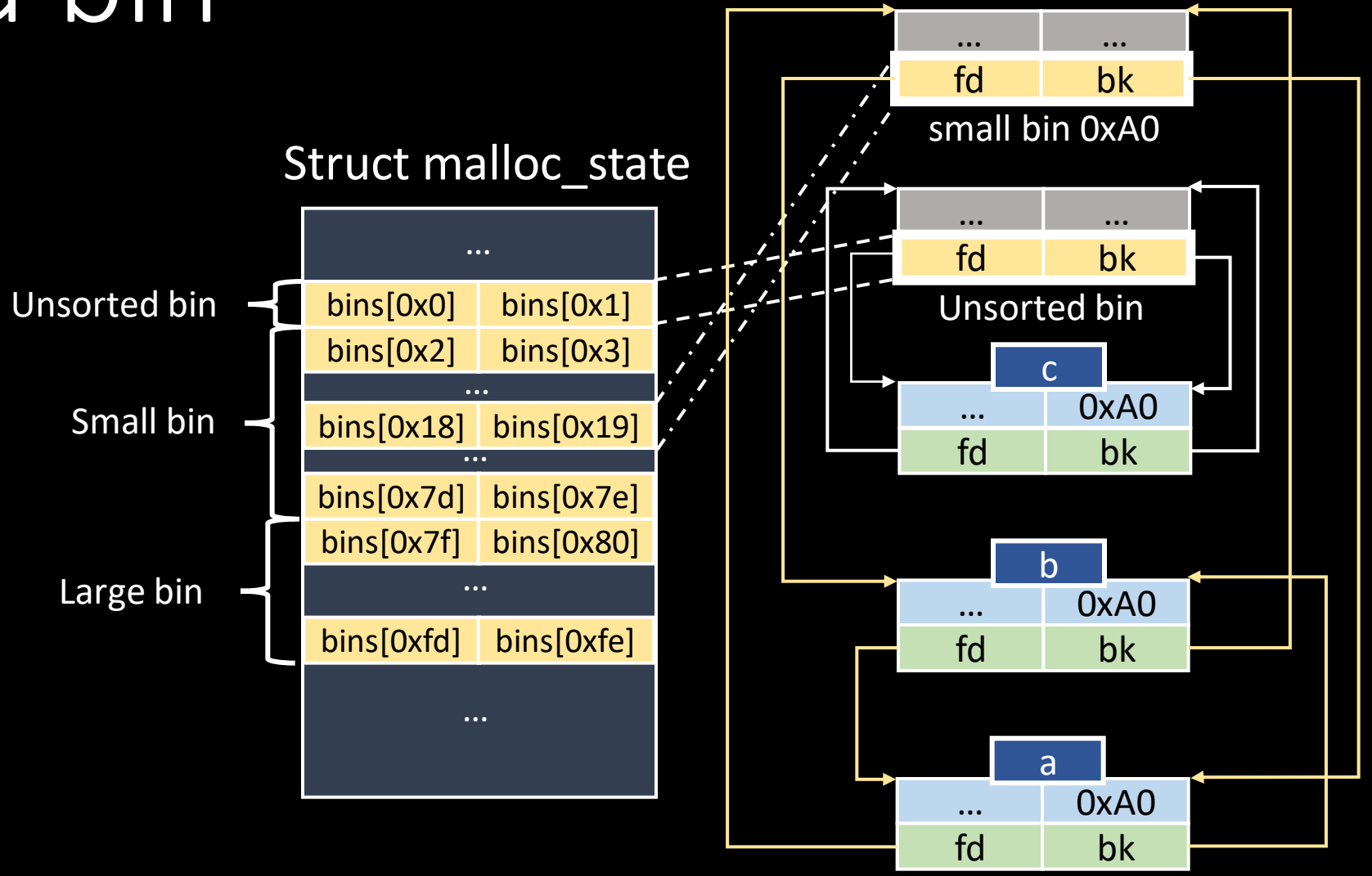
Unsorted bin

- d = malloc(0xA0)
  - Move a
  - Move b
  - Move c



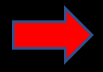
Unsorted bin

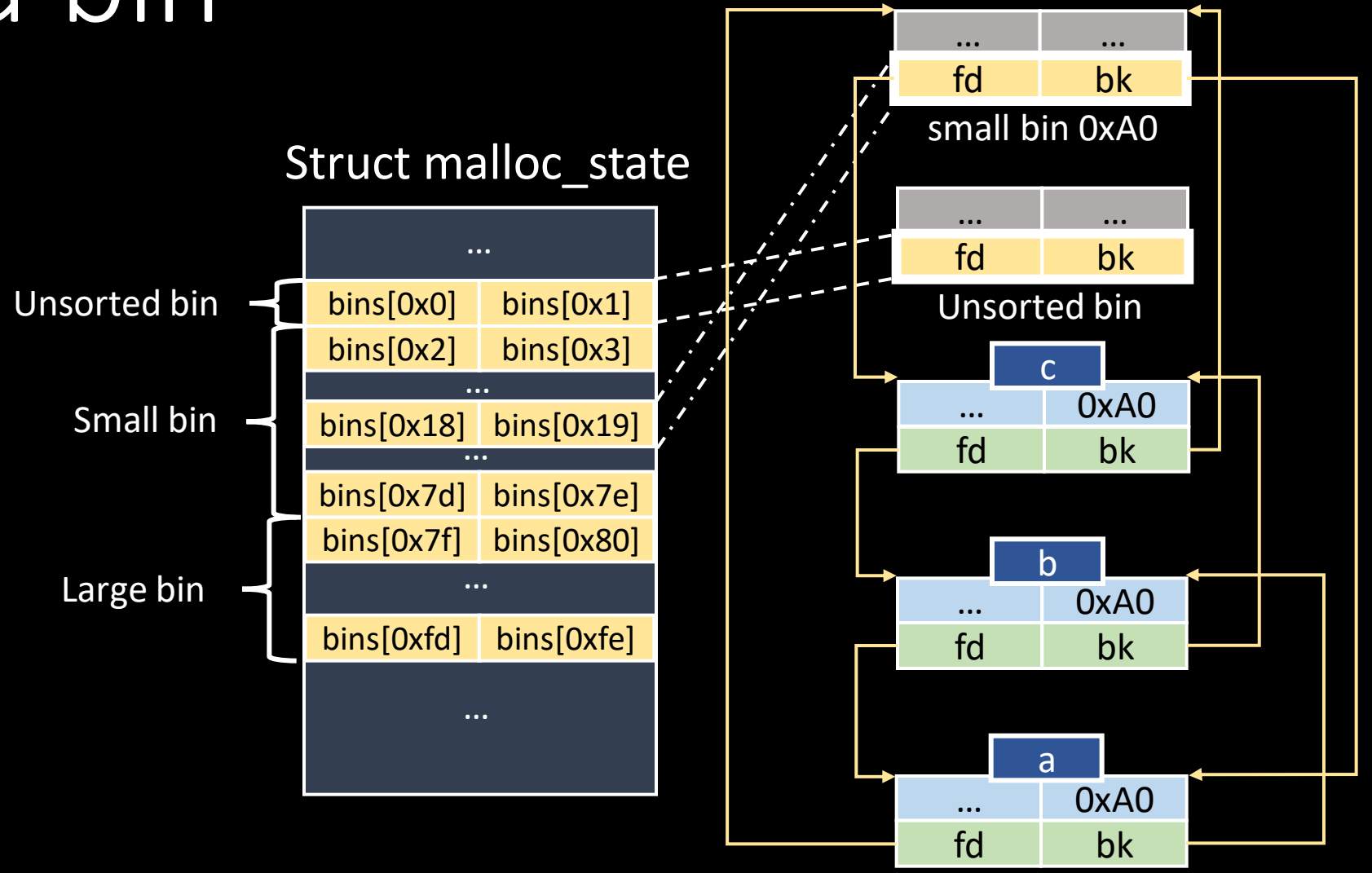
- d = malloc(0xA0)
  - Move a
  - Move b
  - Move c



Unsorted bin

- d = malloc(0xA0)
  - Move a
  - Move b
  - Move c





Small bin DEMO

Large bin 跳過 QQ

Trace code

https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c

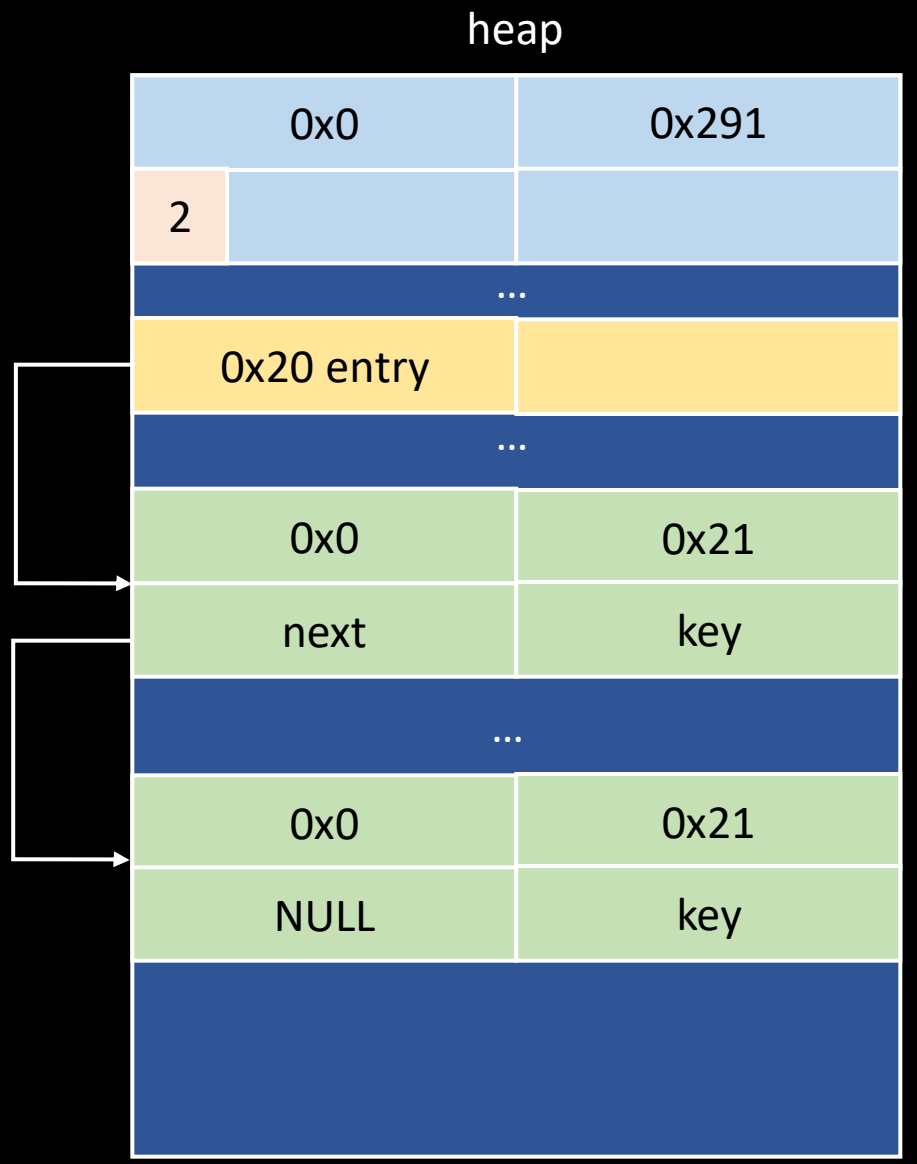
Vulnerability

UAF

- Use-After-Free
- Chunk 被 free 後沒有把 pointer 設成 NULL，導致 user 可以使用到被釋放的記憶體
- 這樣的 pointer 被稱為 dangling pointer
- 如果可以寫值可以改掉 fd, bk 來竄改 linked list 結構
- 如果有 function pointer 可以 malloc 同樣大小的 chunk 改其值來做到任意執行

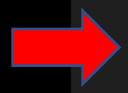
UAF Case 1

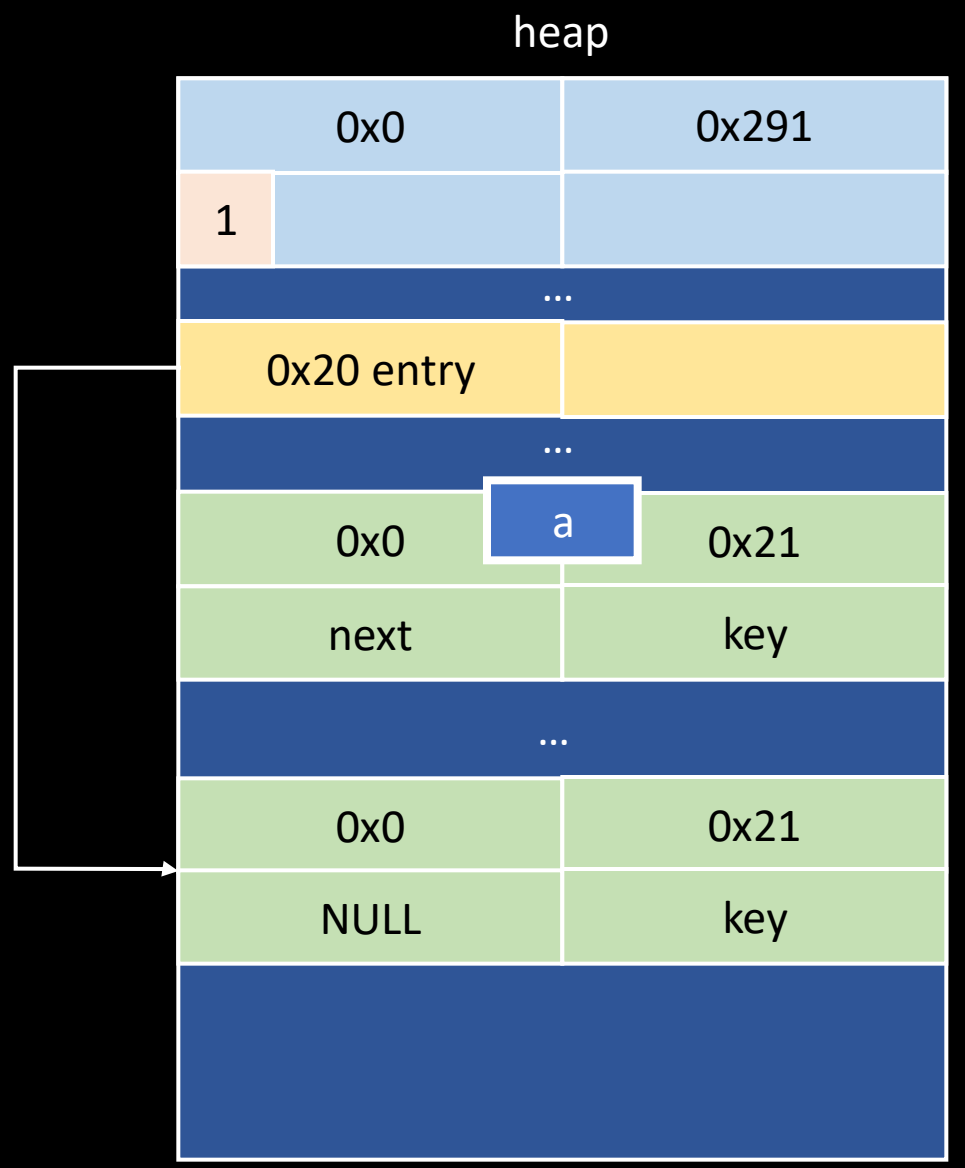
```
long *a = malloc(0x18);
*a = 0xC8763;
free(a);
*a = 0xC8763;
malloc(0x18);
malloc(0x18);
```



UAF Case 1

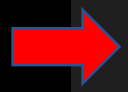
```
long *a = malloc(0x18);
*a = 0xC8763;
free(a);
*a = 0xC8763;
malloc(0x18);
malloc(0x18);
```

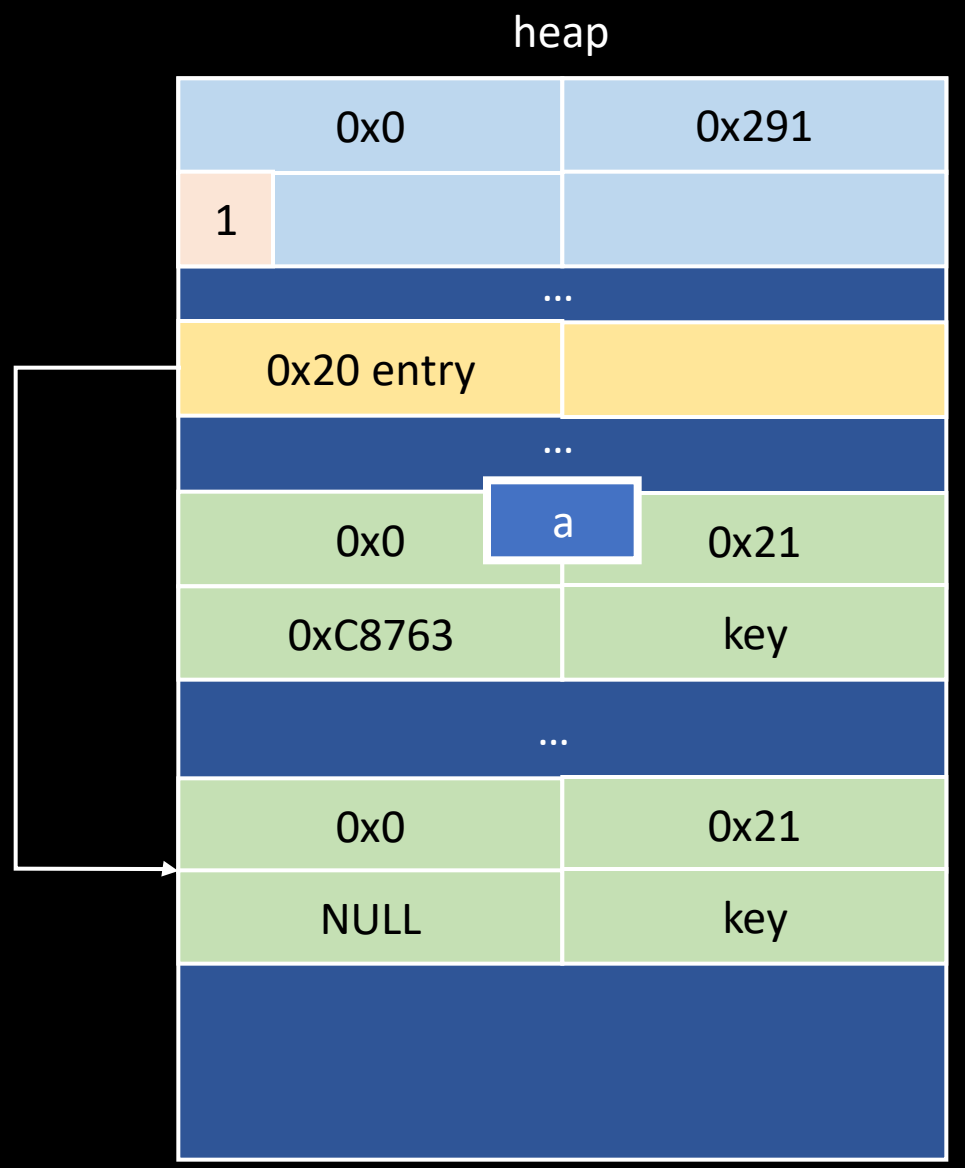




UAF Case 1

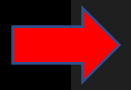
```
long *a = malloc(0x18);
*a = 0xC8763;
free(a);
*a = 0xC8763;
malloc(0x18);
malloc(0x18);
```

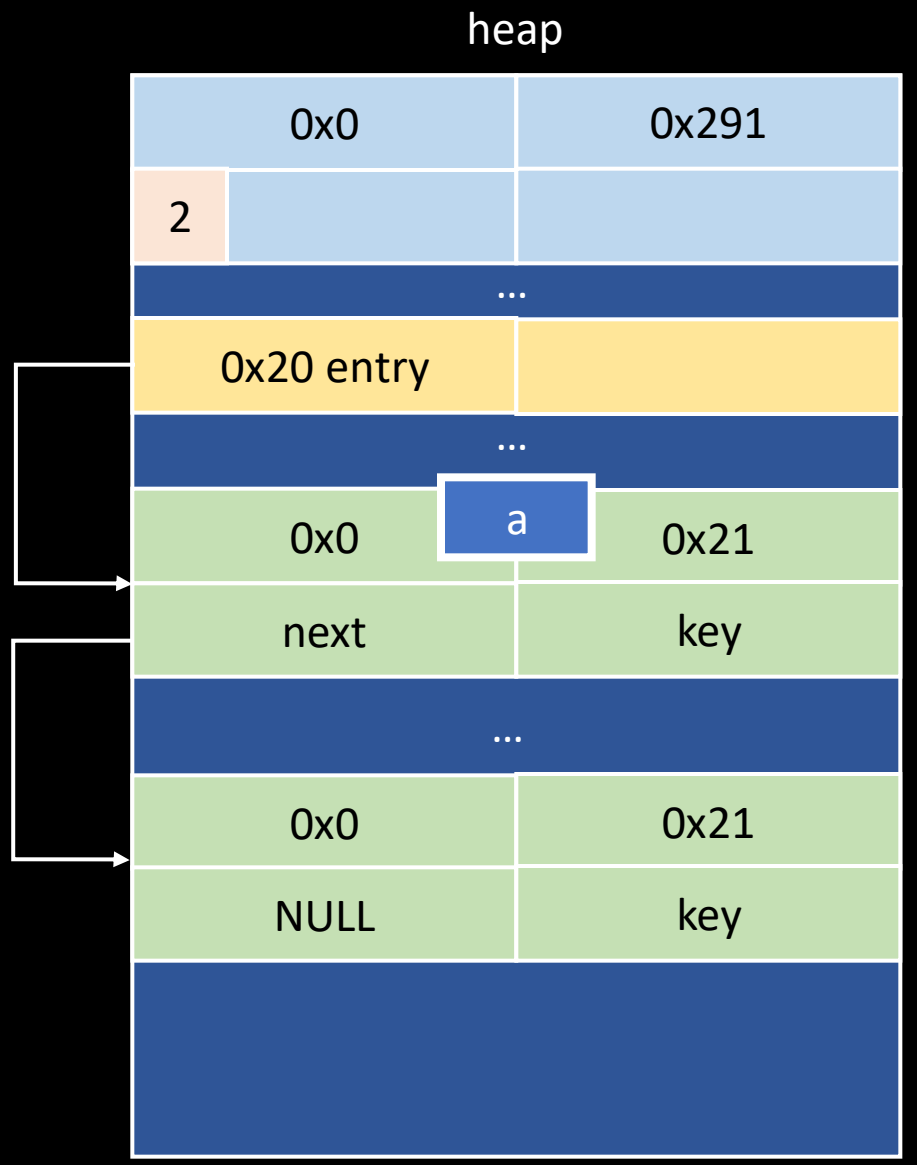




UAF Case 1

```
long *a = malloc(0x18);
*a = 0xC8763;
free(a);
*a = 0xC8763;
malloc(0x18);
malloc(0x18);
```

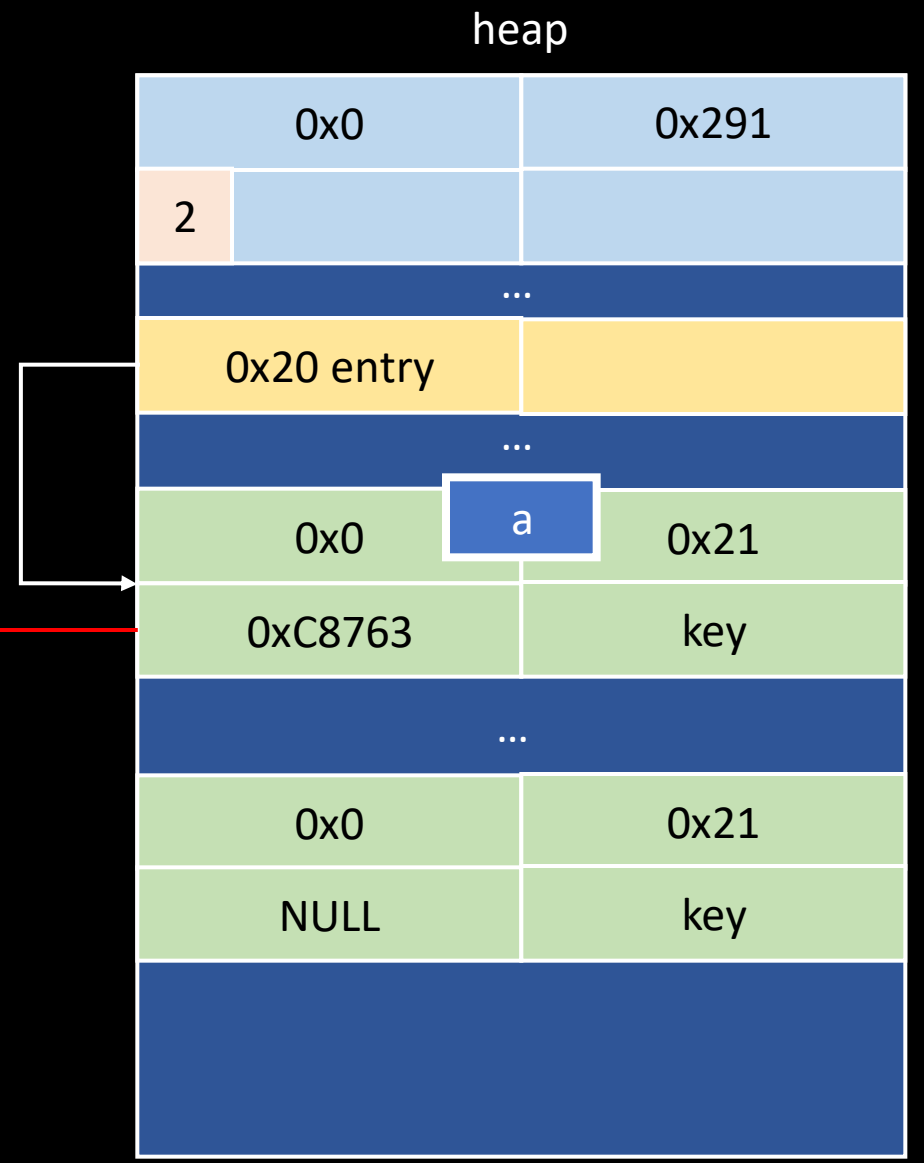




UAF Case 1

```
long *a = malloc(0x18);
*a = 0xC8763;
free(a);
*a = 0xC8763;
malloc(0x18);
malloc(0x18);
```

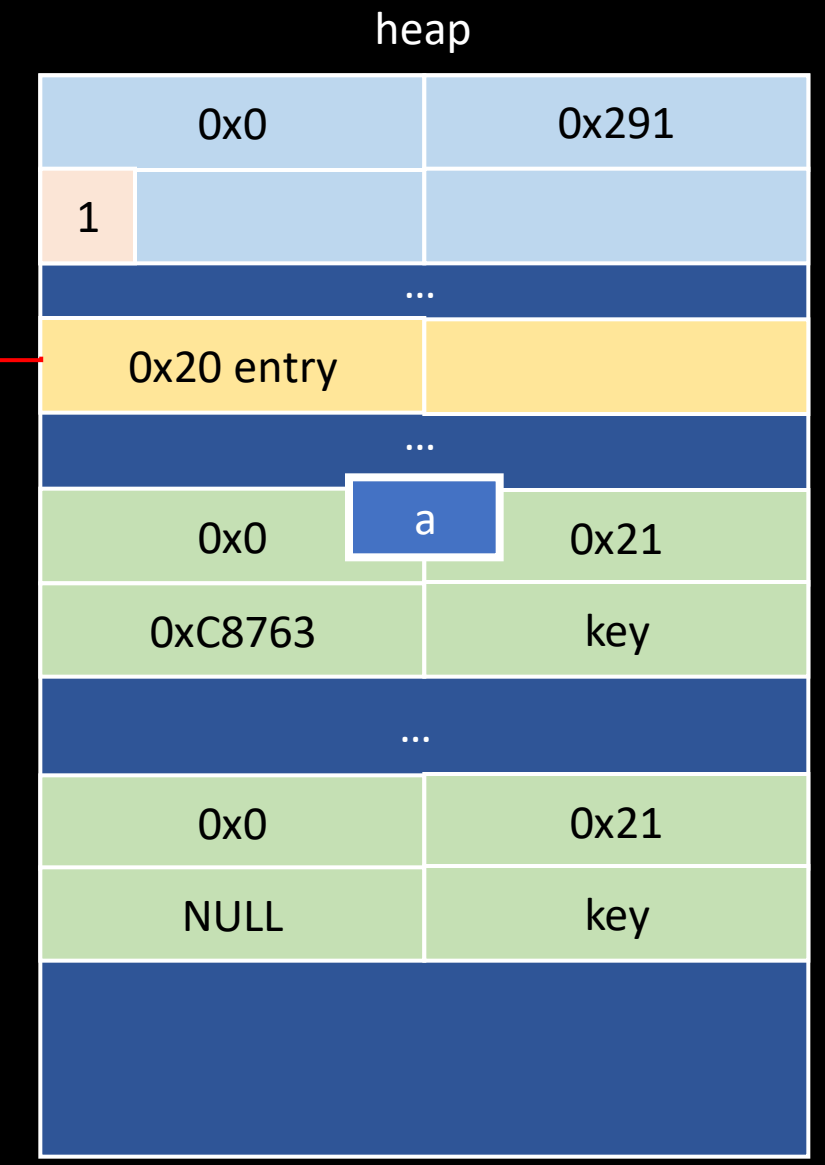
a->next 指向 0xC8763



UAF Case 1

```
long *a = malloc(0x18);
*a = 0xC8763;
free(a);
*a = 0xC8763;
malloc(0x18);
malloc(0x18);
```

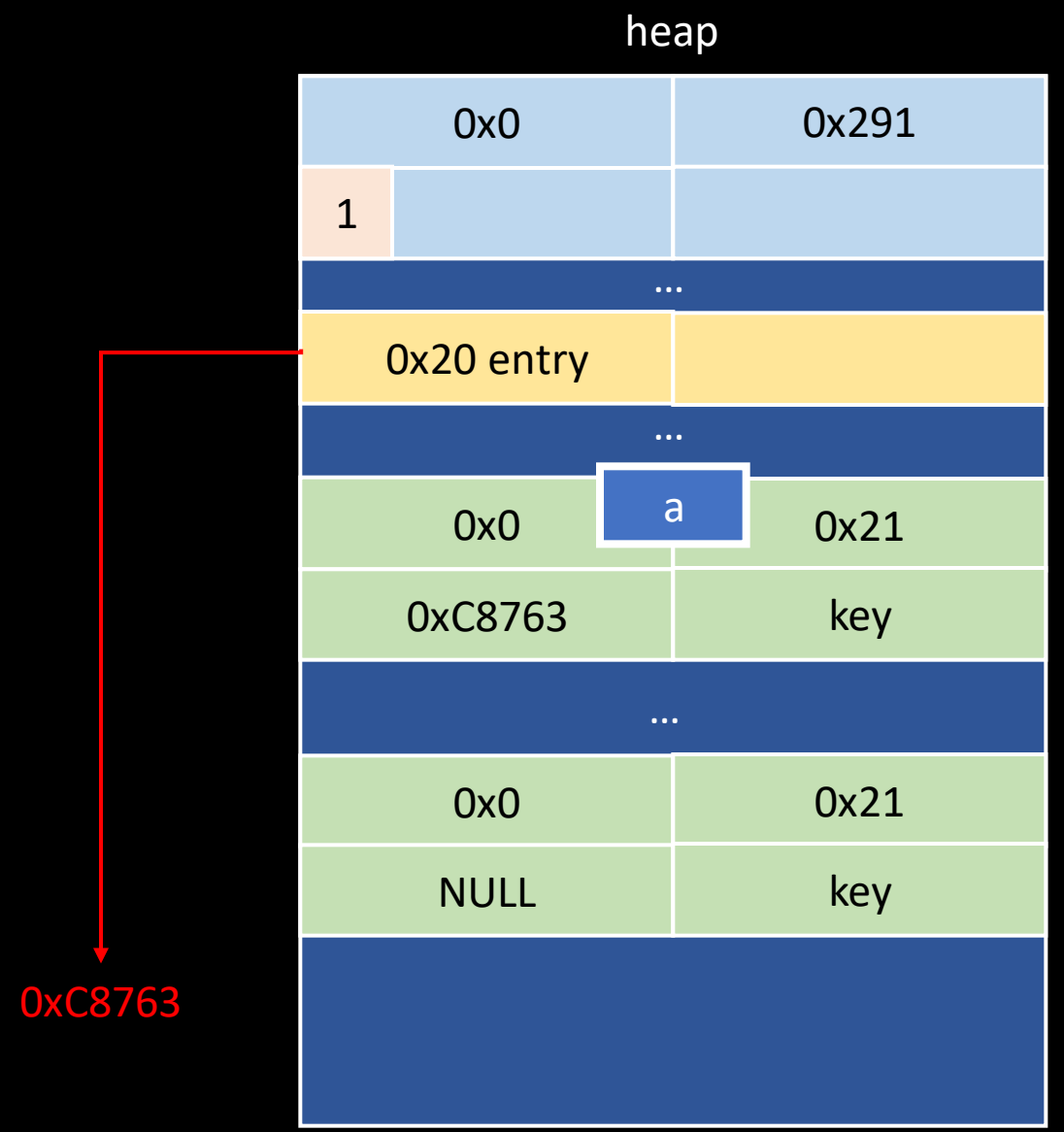
0x20 entry 指向 0xC8763



UAF Case 1

```
long *a = malloc(0x18);
*a = 0xC8763;
free(a);
*a = 0xC8763;
malloc(0x18);
malloc(0x18);
```

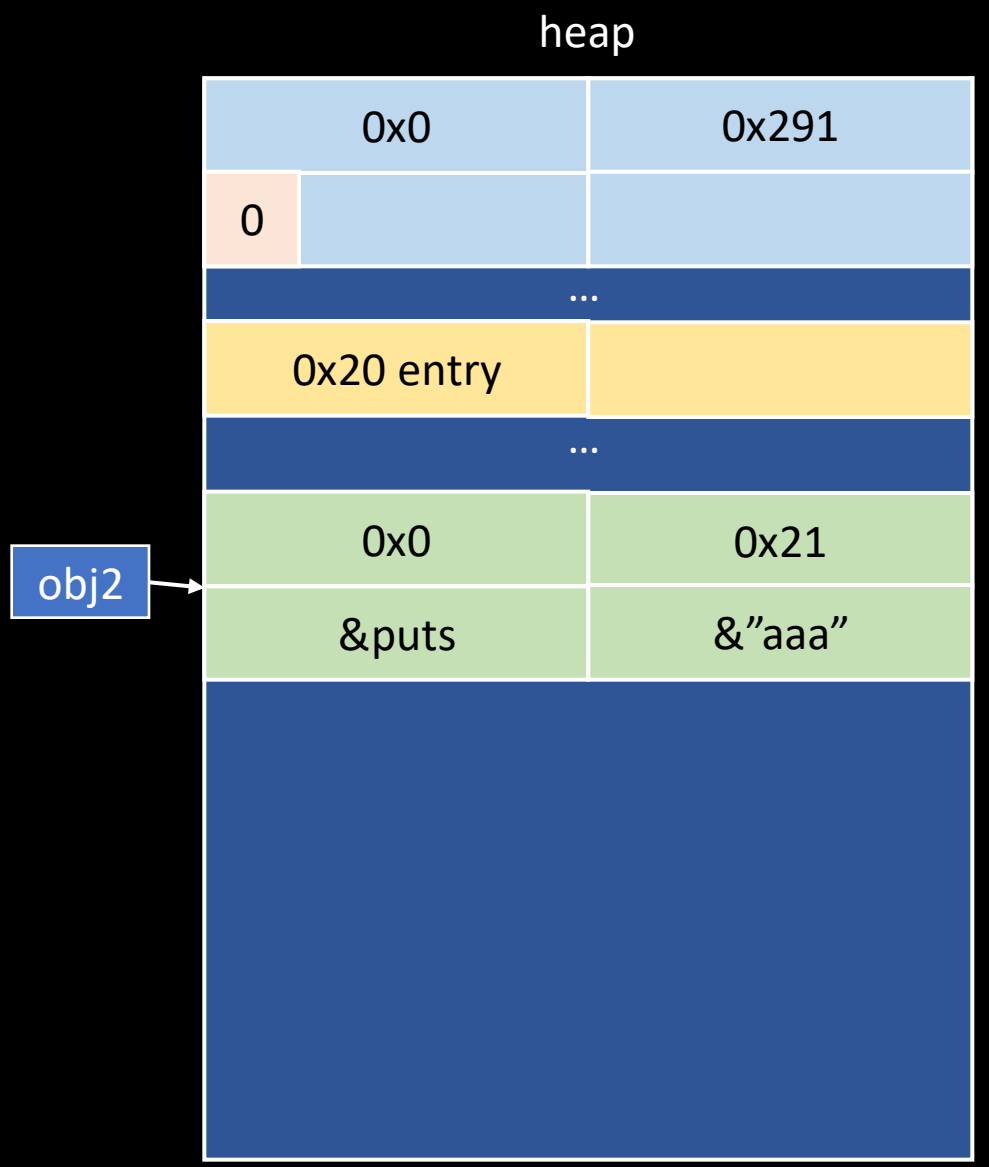
因為 0xC8763 是 Invalid address 所以會 crash



Demo UAF

UAF Case 2

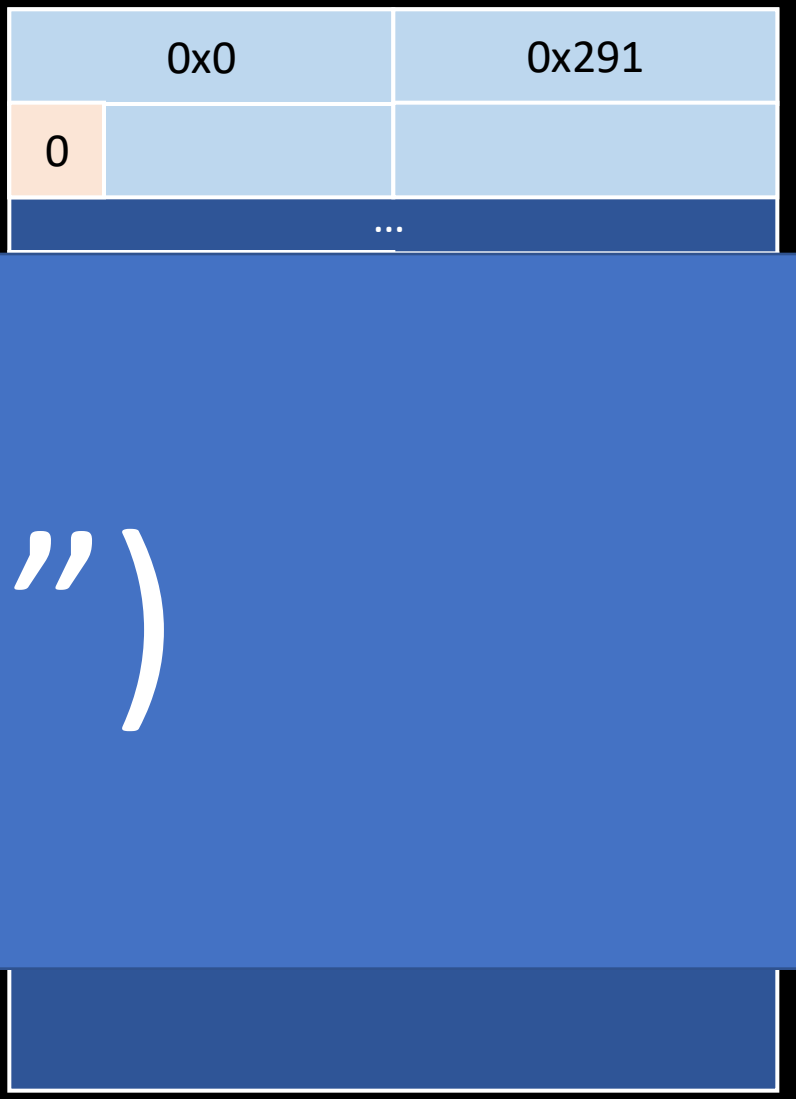
```
struct obj1 {long a; long b;};
struct obj2 {void (*fptr)(); char *c};

struct obj2 *obj2 = malloc(sizeof(obj2));
obj2->fptr = puts;
obj2->c = "aaa";
obj2->fptr(obj2->c);
free(obj2);
struct obj1 *obj1 = malloc(sizeof(obj1));
obj1->a = &system;
obj1->b = "sh";
obj2->fptr(obj2->c);
```



UAF Case 2

```
struct obj1 {long a; long b;};
```

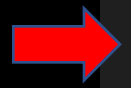
heap

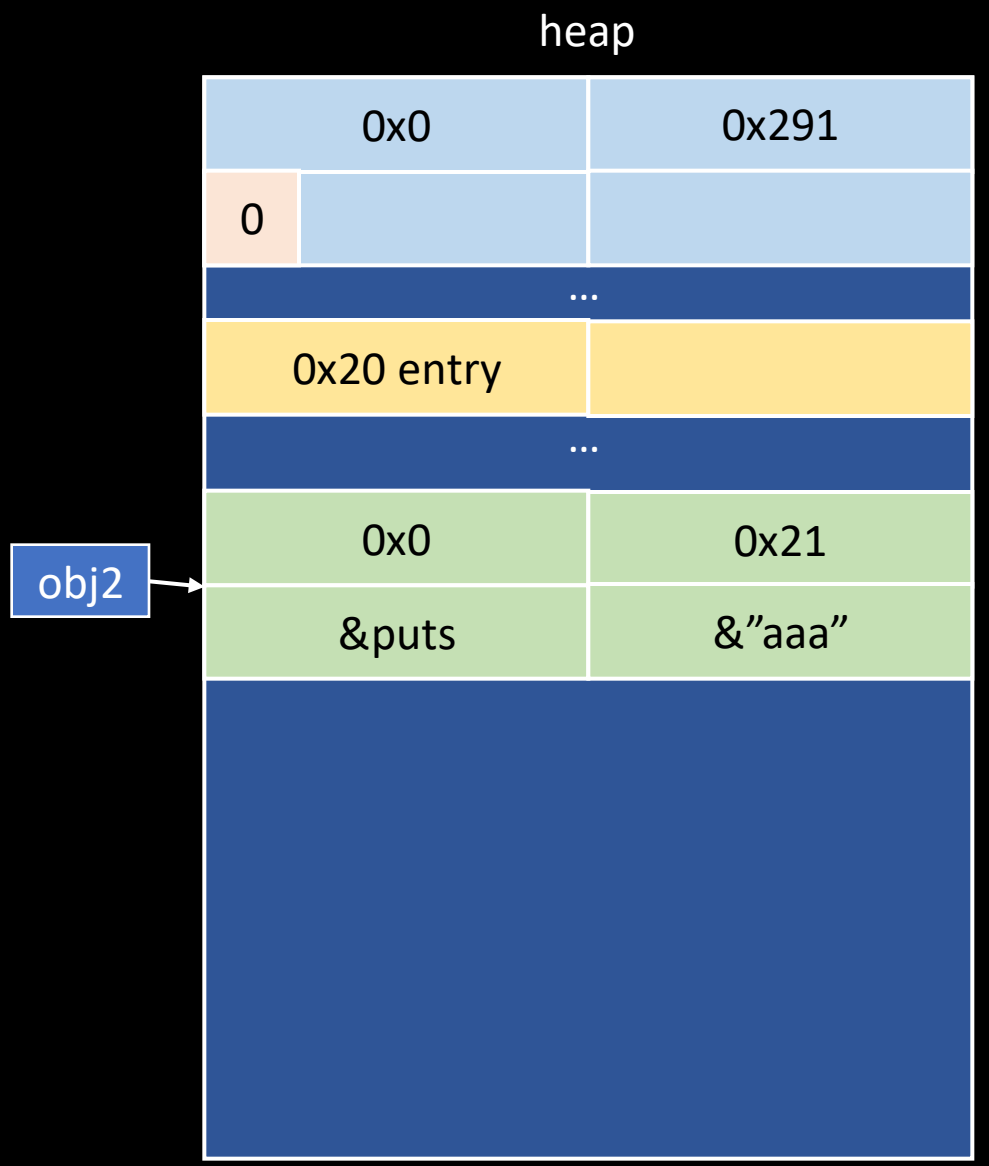


puts(“aaa”)

UAF Case 2

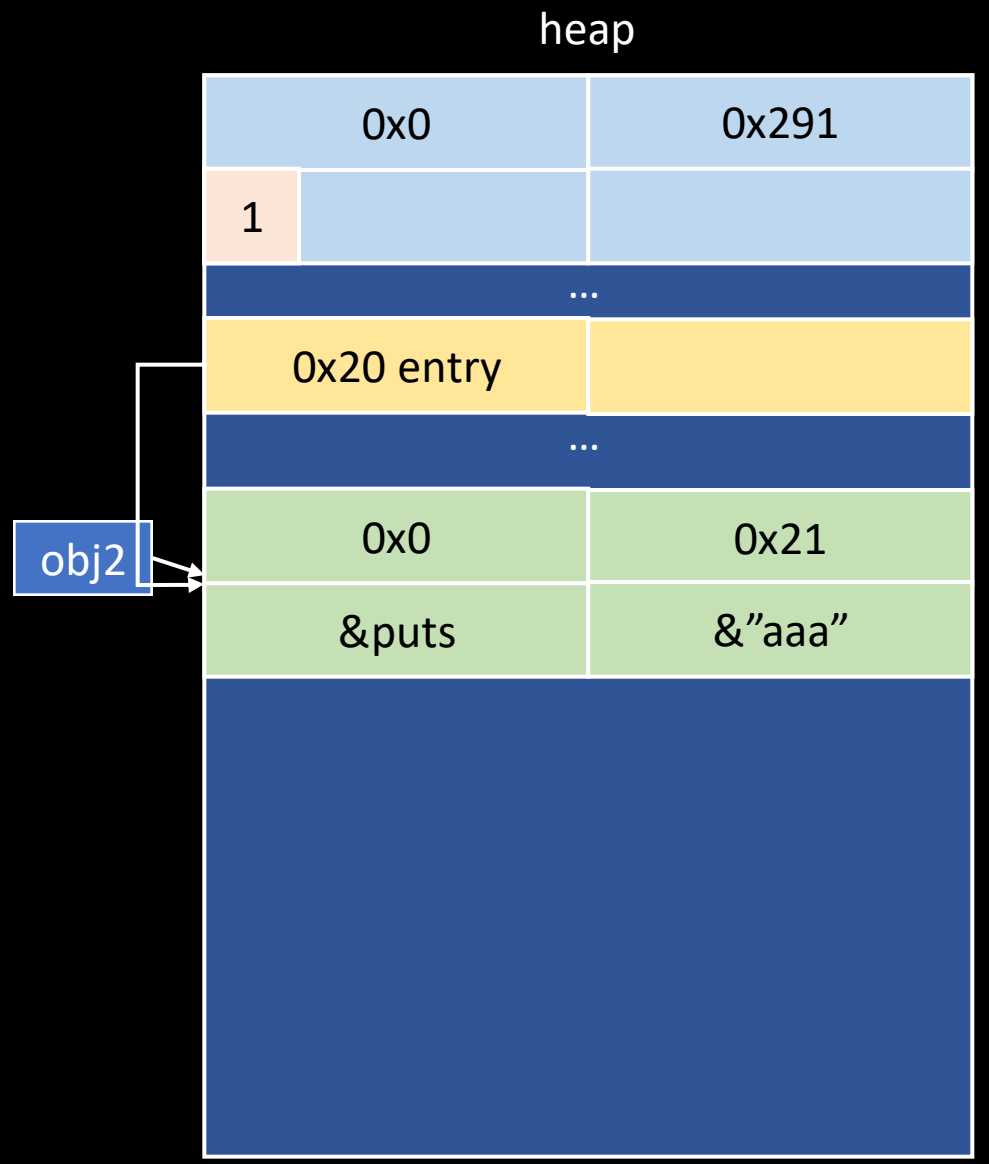
```
struct obj1 {long a; long b;};
struct obj2 {void (*fptr)(); char *c};

struct obj2 *obj2 = malloc(sizeof(obj2));
obj2->fptr = puts;
obj2->c = "aaa";
obj2->fptr(obj2->c);
free(obj2);
struct obj1 *obj1 = malloc(sizeof(obj1));
obj1->a = &system;
obj1->b = "sh";
obj2->fptr(obj2->c);
```





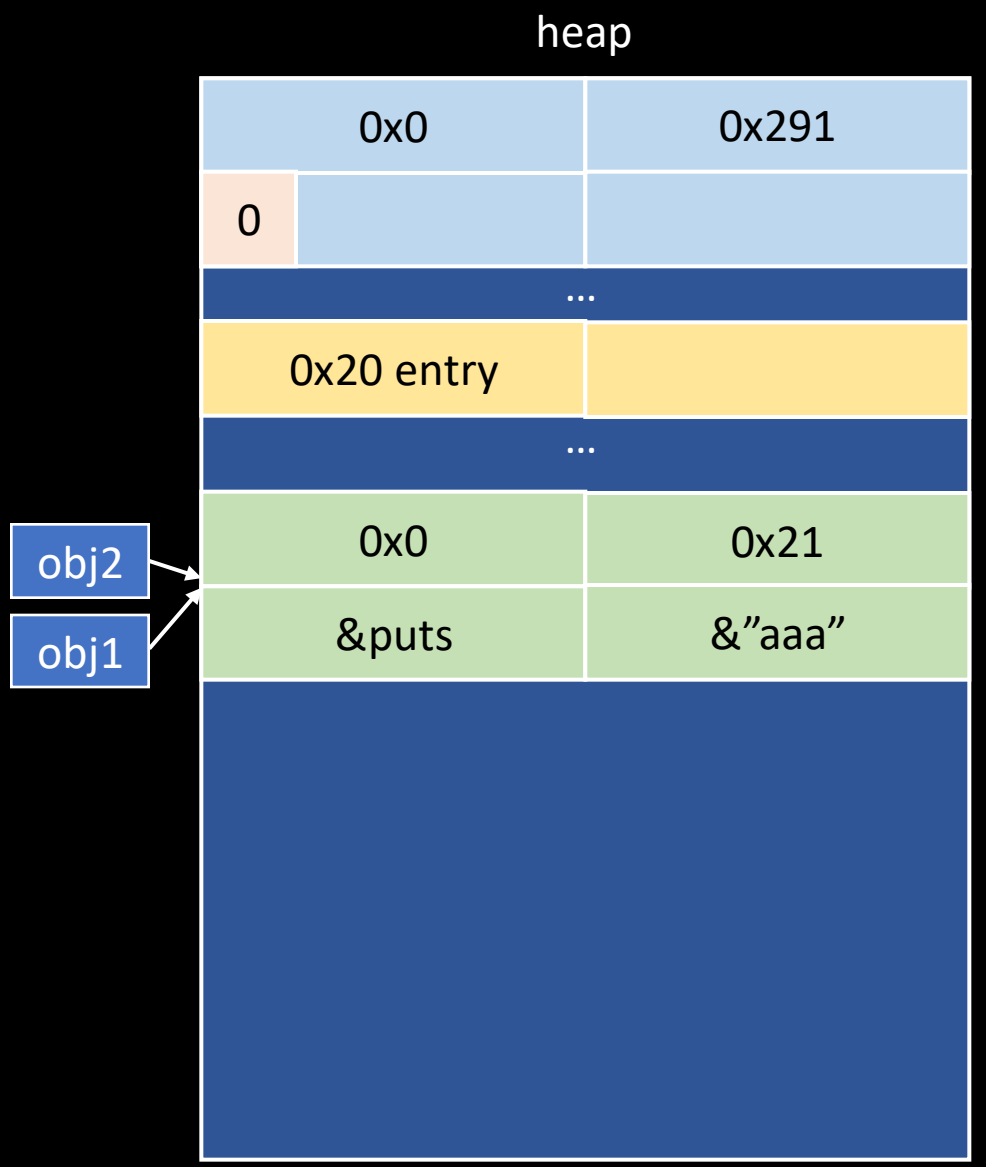
UAF Case 2

```
struct obj1 {long a; long b;};
struct obj2 {void (*fptr)(); char *c};

struct obj2 *obj2 = malloc(sizeof(obj2));
obj2->fptr = puts;
obj2->c = "aaa";
obj2->fptr(obj2->c);
free(obj2);
struct obj1 *obj1 = malloc(sizeof(obj1));
obj1->a = &system;
obj1->b = "sh";
obj2->fptr(obj2->c);
```



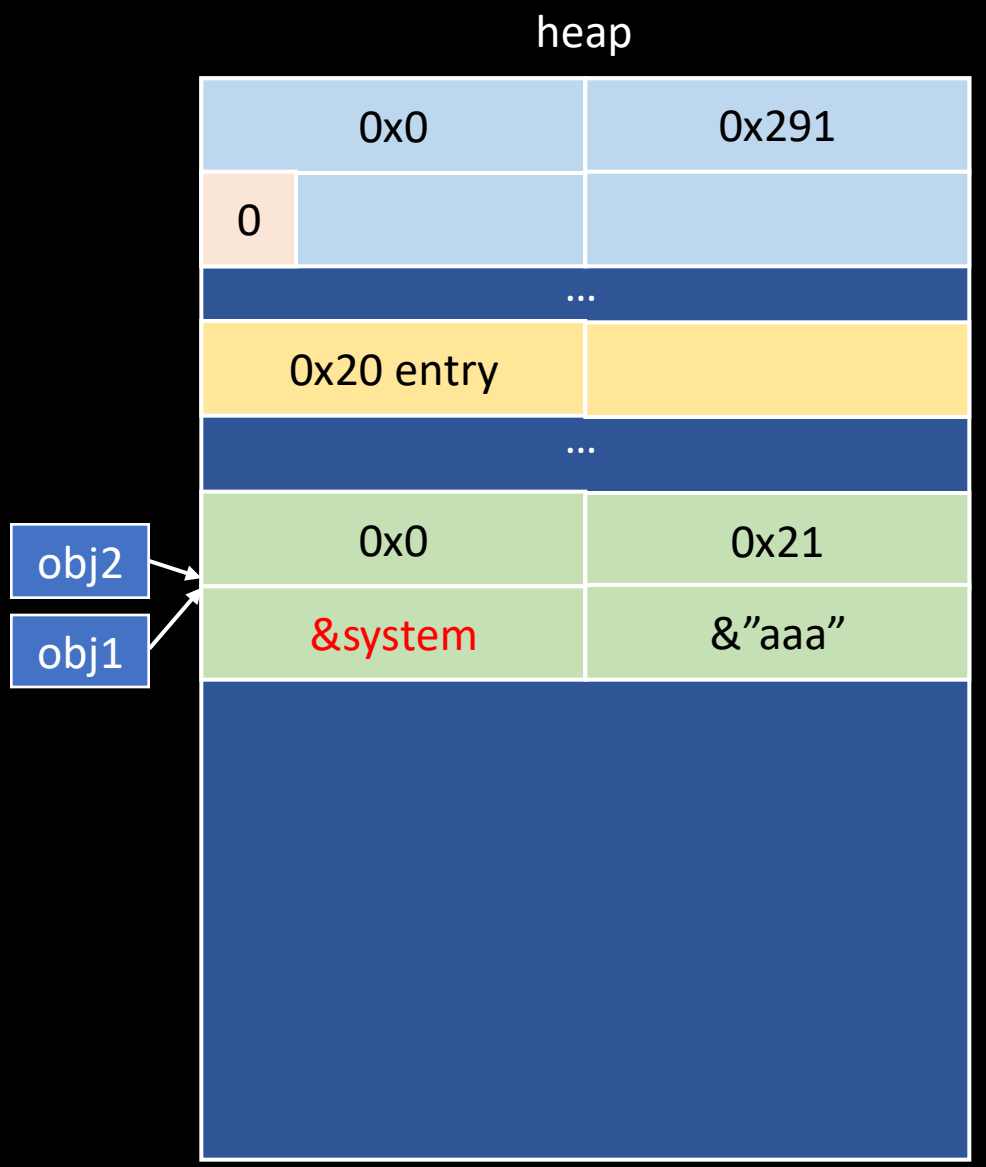
UAF Case 2

```
struct obj1 {long a; long b;};
struct obj2 {void (*fptr)(); char *c};

struct obj2 *obj2 = malloc(sizeof(obj2));
obj2->fptr = puts;
obj2->c = "aaa";
obj2->fptr(obj2->c);
free(obj2);
struct obj1 *obj1 = malloc(sizeof(obj1));
obj1->a = &system;
obj1->b = "sh";
obj2->fptr(obj2->c);
```



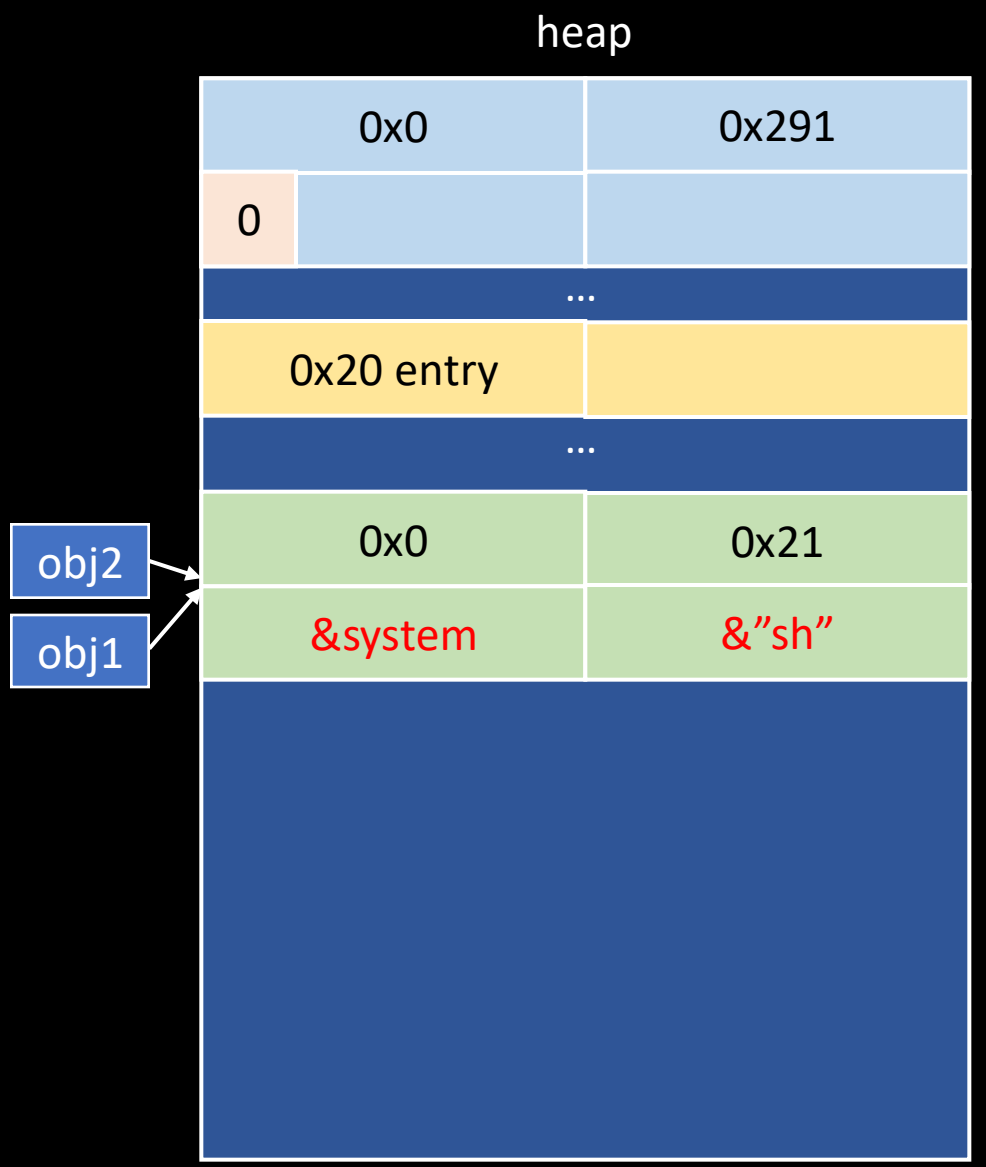
UAF Case 2

```
struct obj1 {long a; long b;};
struct obj2 {void (*fptr)(); char *c};

struct obj2 *obj2 = malloc(sizeof(obj2));
obj2->fptr = puts;
obj2->c = "aaa";
obj2->fptr(obj2->c);
free(obj2);
struct obj1 *obj1 = malloc(sizeof(obj1));
obj1->a = &system;
obj1->b = "sh";
obj2->fptr(obj2->c);
```



UAF Case 2

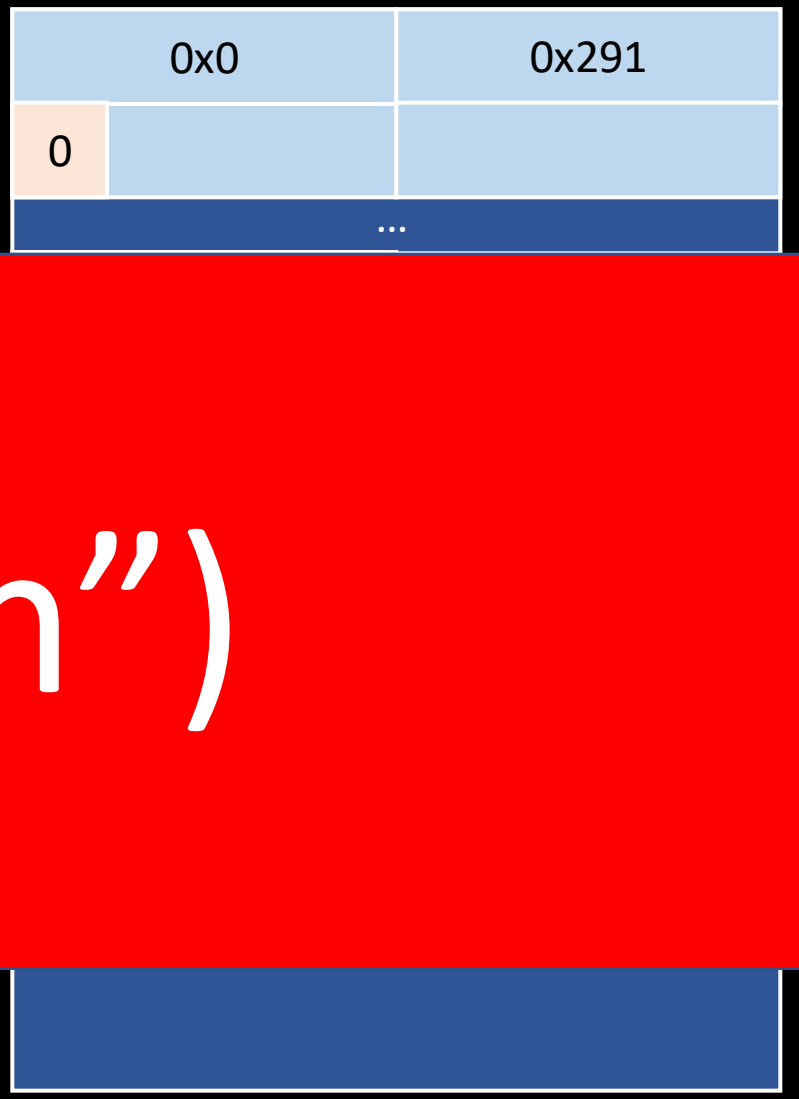
```
struct obj1 {long a; long b;};
struct obj2 {void (*fptr)(); char *c};

struct obj2 *obj2 = malloc(sizeof(obj2));
obj2->fptr = puts;
obj2->c = "aaa";
obj2->fptr(obj2->c);
free(obj2);
struct obj1 *obj1 = malloc(sizeof(obj1));
obj1->a = &system;
obj1->b = "sh";
obj2->fptr(obj2->c);
```



UAF Case 2

```
struct obj1 {long a; long b;};
```

heap



system(“sh”)

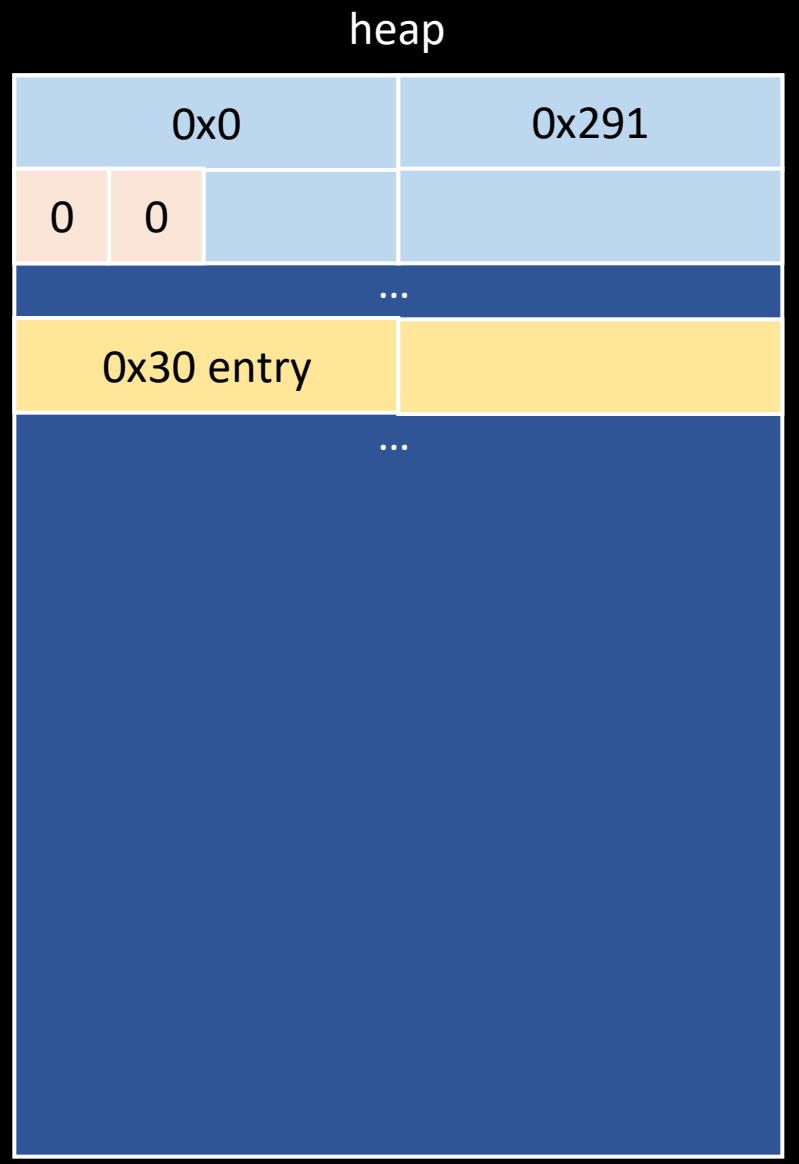
Lab UAF

Heap overflow

- 寫資料到 heap buffer 時沒檢查好長度導致 overflow，可以蓋到下面 chunk 的 header 和 data
- Header 的部分可以蓋 size 更改 chunk 大小或 PREV_INUSE 讓 glibc 誤以為上個 chunk 是 freed
- Data 的部分
  - 蓋到 allocated chunk 可以寫 chunk 內的敏感資料，如 function pointer
  - 蓋到 freed chunk 可以寫 fd, bk 來改變 linked list

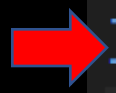
Heap overflow

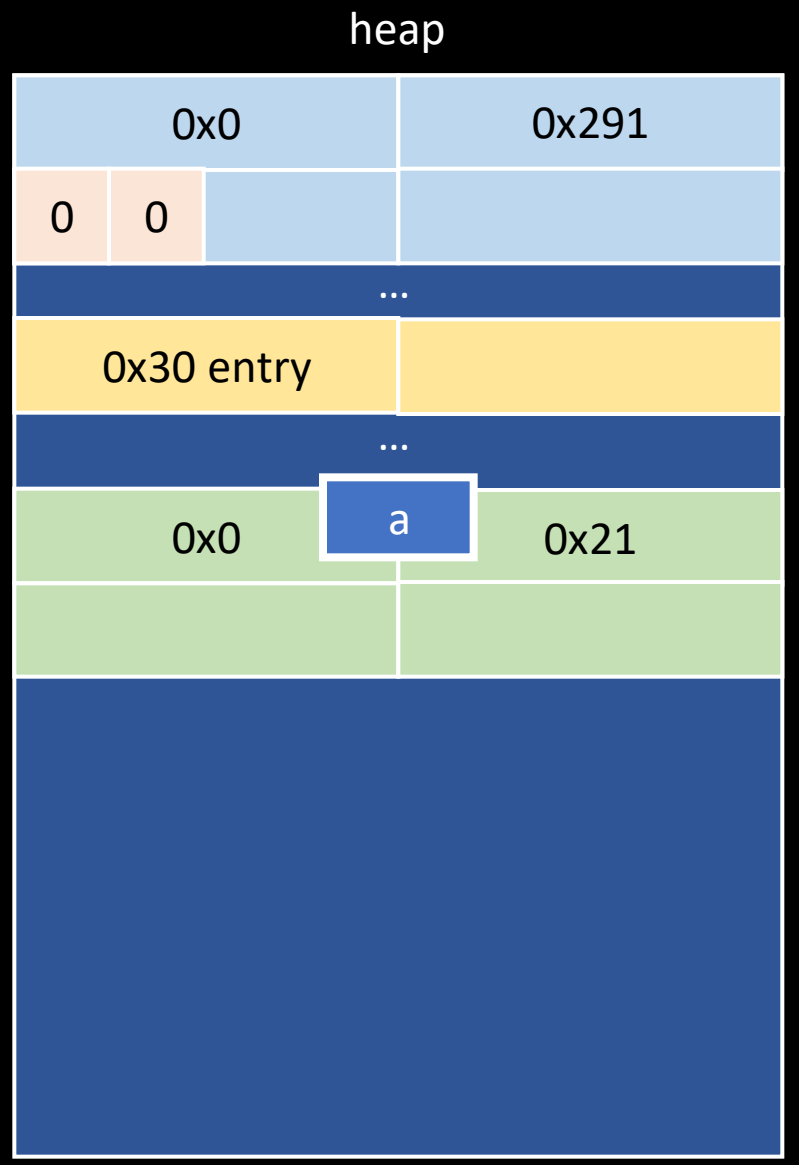
```
long *a = malloc(0x10);
long *b = malloc(0x10);
a[3] = 0x31;
free(b);
```



Heap overflow

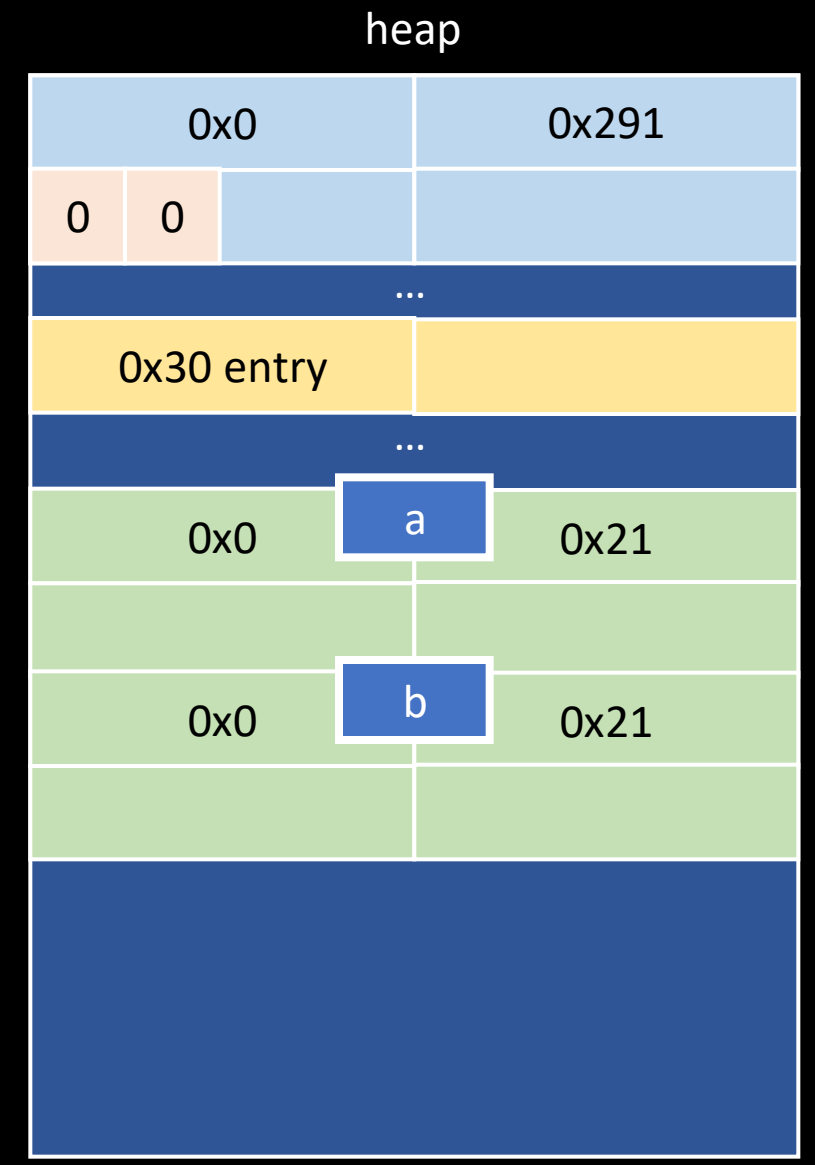
```
long *a = malloc(0x10);
long *b = malloc(0x10);
a[3] = 0x31;
free(b);
```





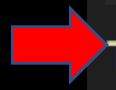
Heap overflow

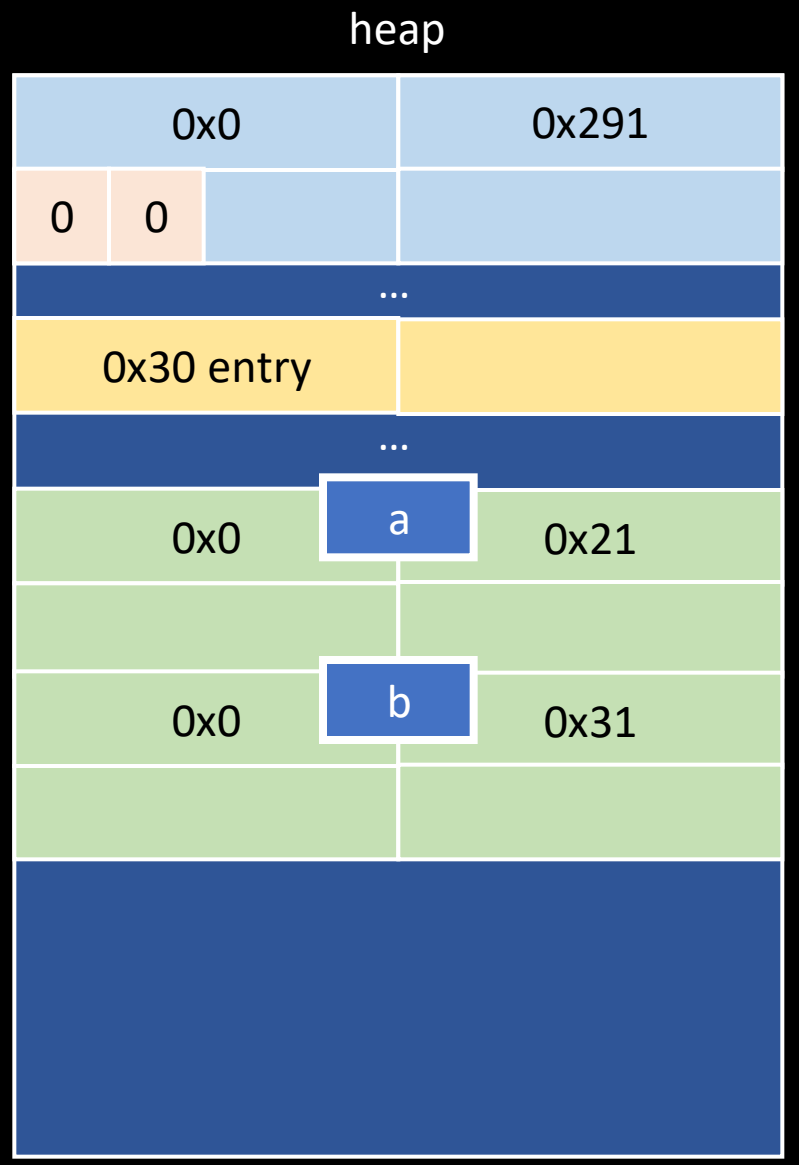
```
long *a = malloc(0x10);
long *b = malloc(0x10);
a[3] = 0x31;
free(b);
```



Heap overflow

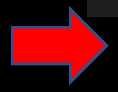
```
long *a = malloc(0x10);
long *b = malloc(0x10);
a[3] = 0x31;
free(b);
```

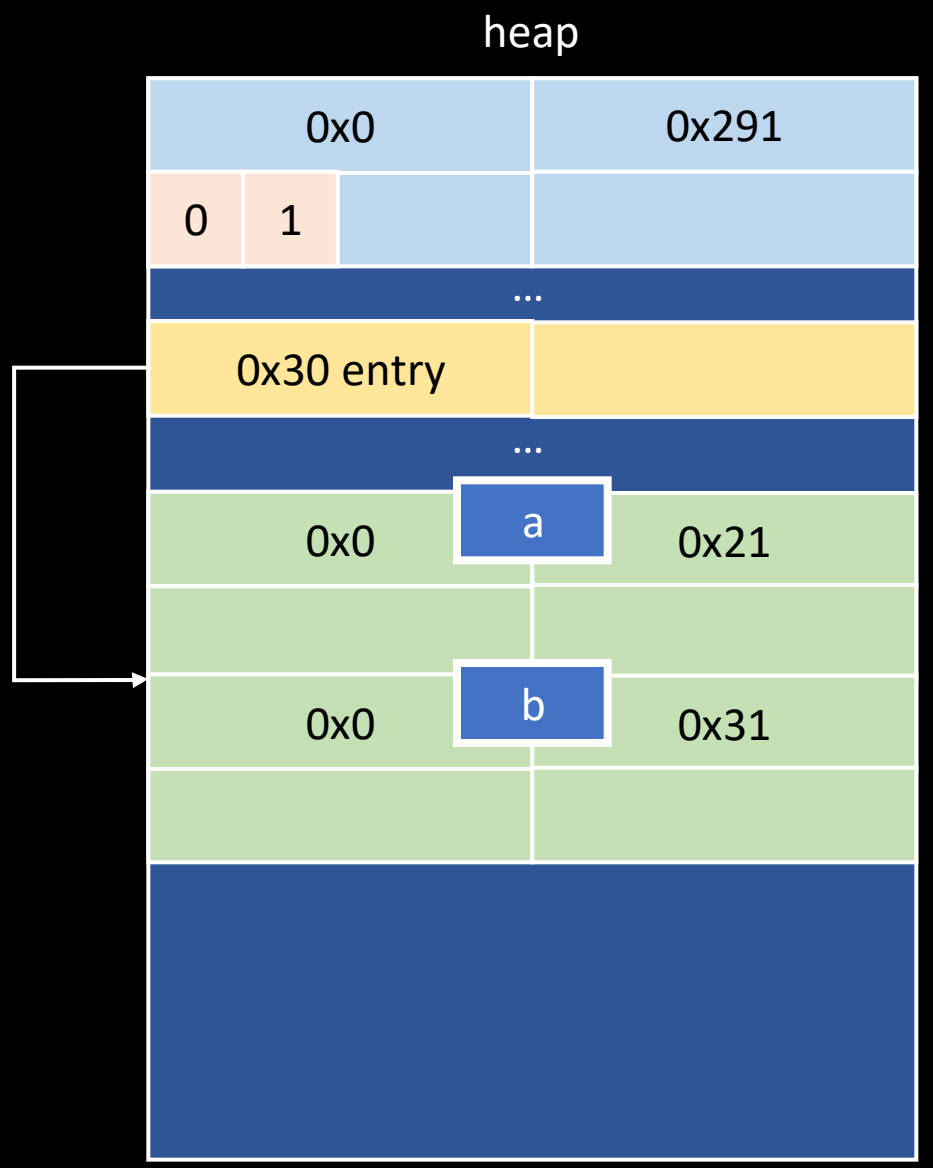




Heap overflow

```
long *a = malloc(0x10);
long *b = malloc(0x10);
a[3] = 0x31;
free(b);
```

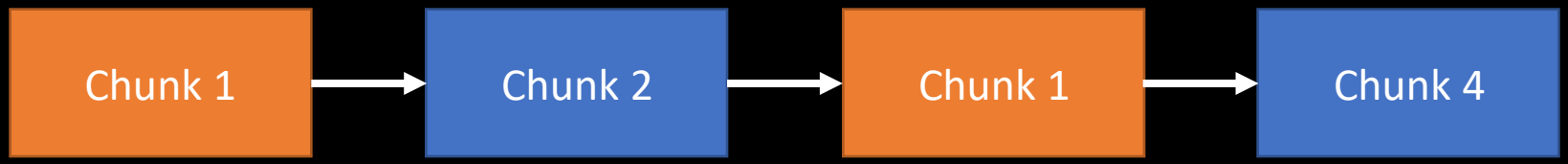




Demo Heap Overflow

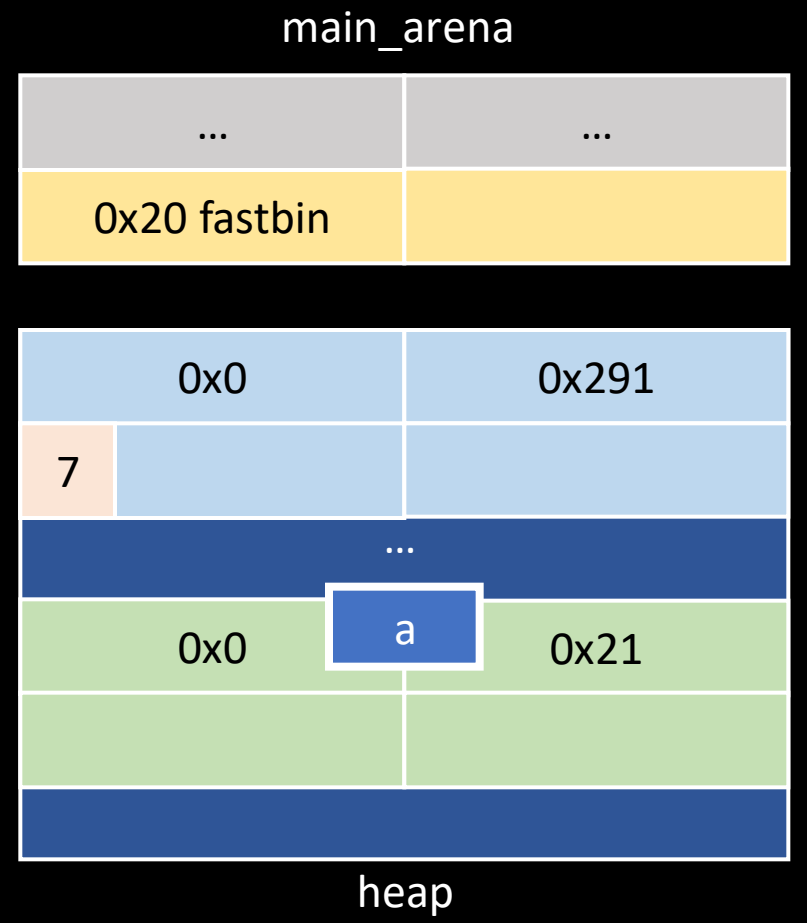
Double free

- Chunk 被連續 free 兩遍，導致 bin 紀錄同一個 chunk 兩次，就稱作 double free
- 在 malloc 取到 chunk 後 bin 裡還有同一個 chunk，這樣如果修改 chunk data 就等同改到 freed chunk
- 但 glibc 有檢查機制偵測 double free，所以要利用就要繞檢查



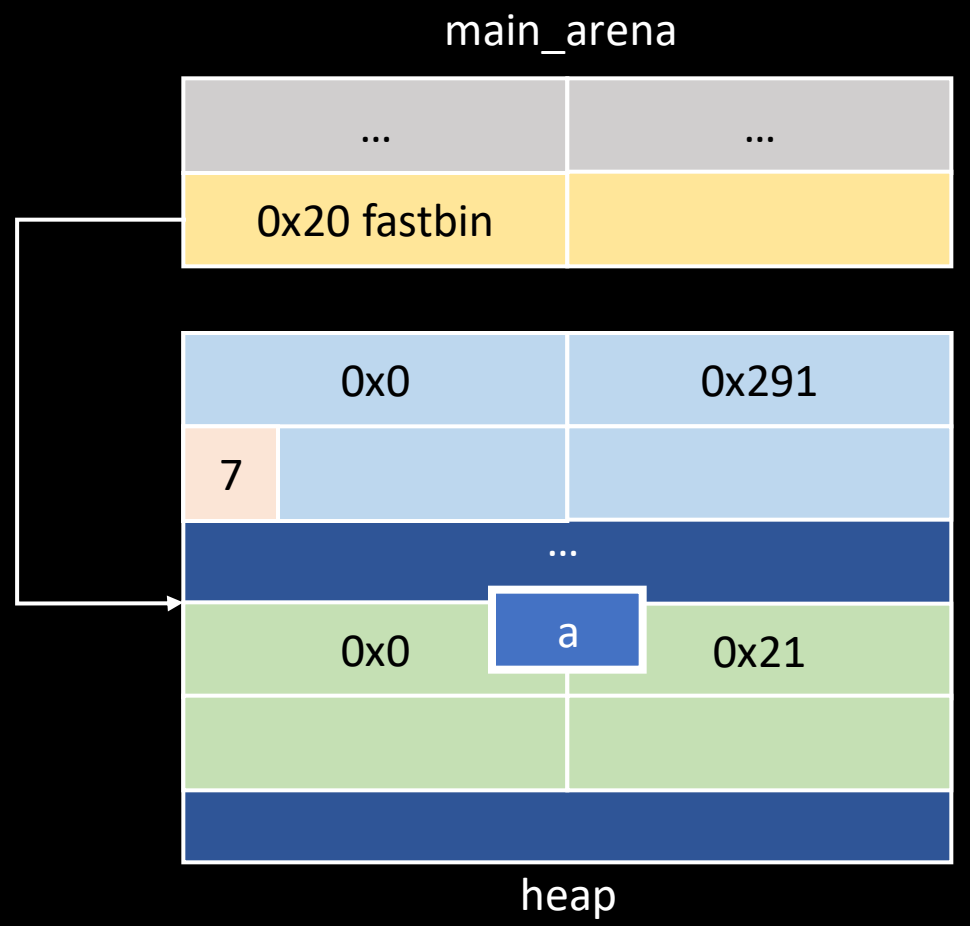
Double free

```
// tcache 已滿
free(a);
free(a);
```



Double free

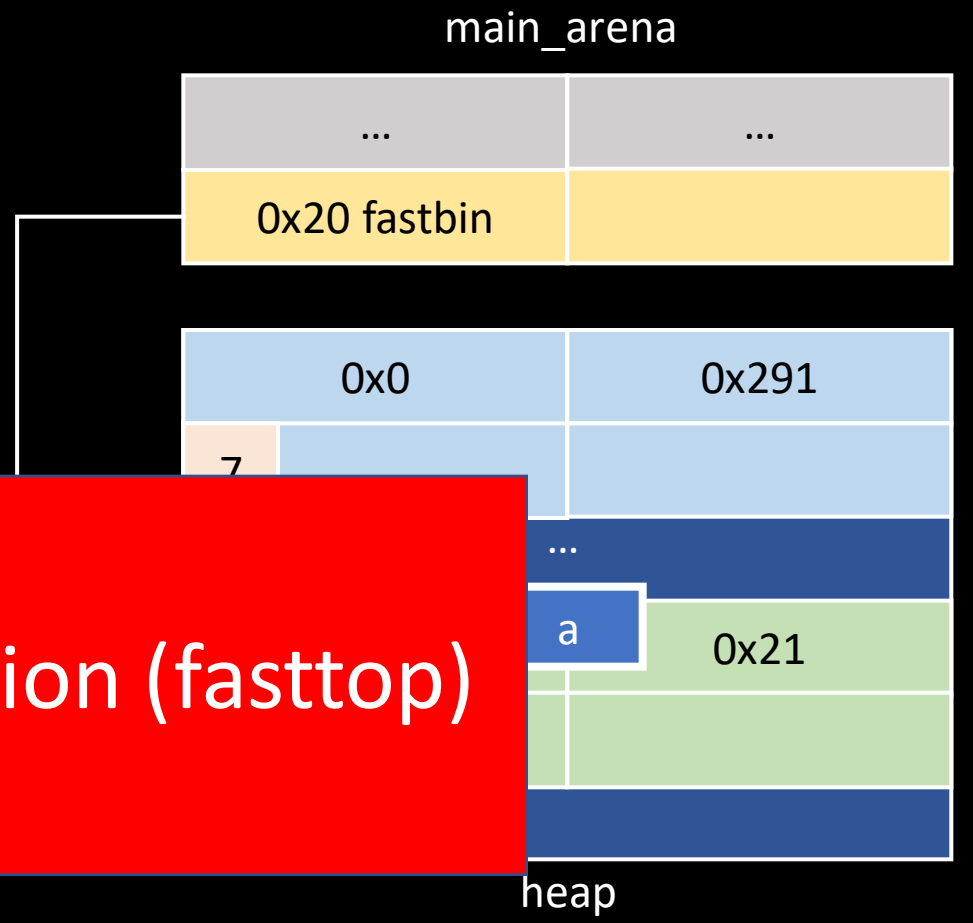
```
// tcache 已滿
free(a);
free(a);
```



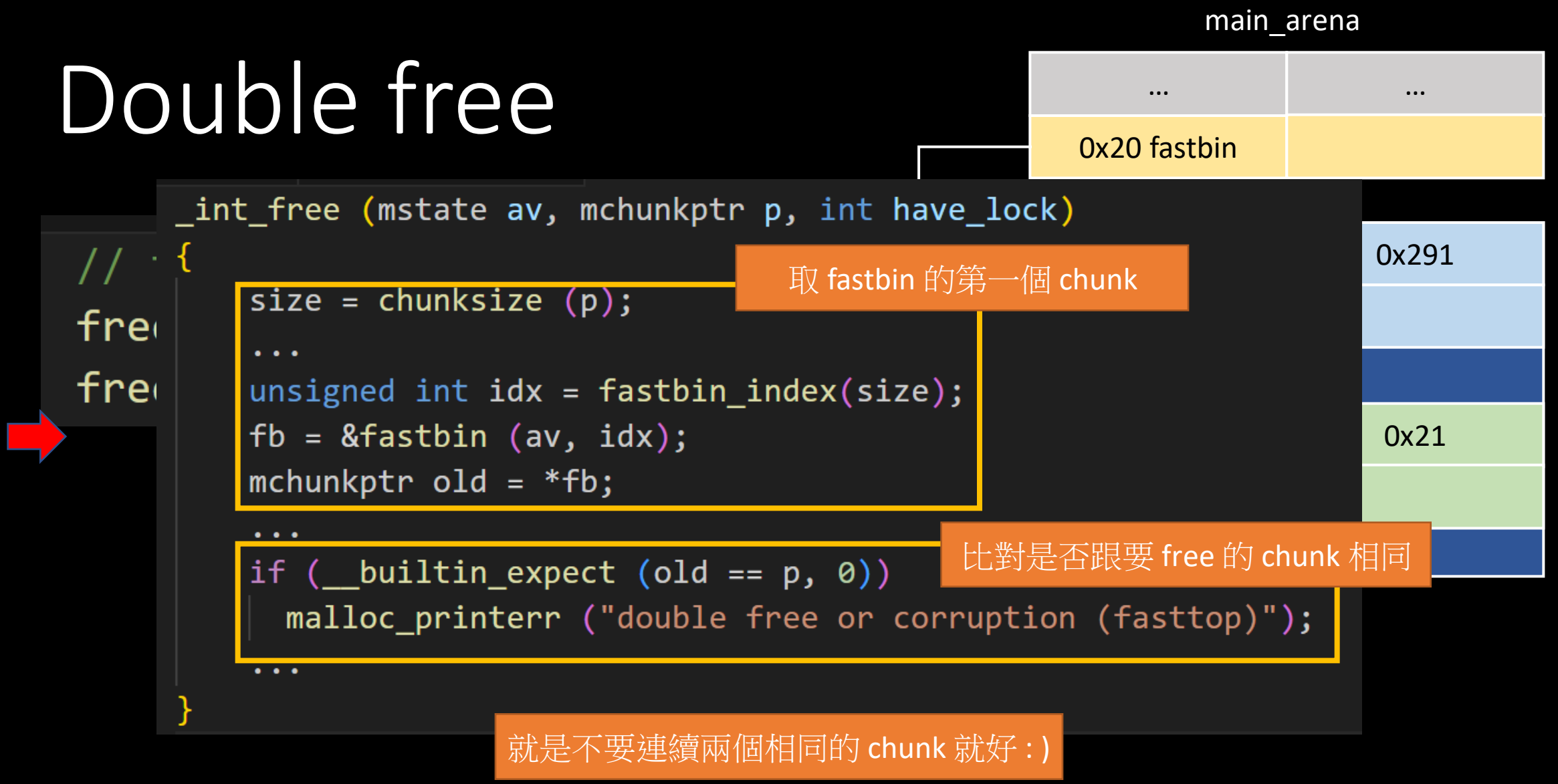
Double free

```
// tcache 已滿
free(a);
free(a);
```

double free or corruption (fasttop)



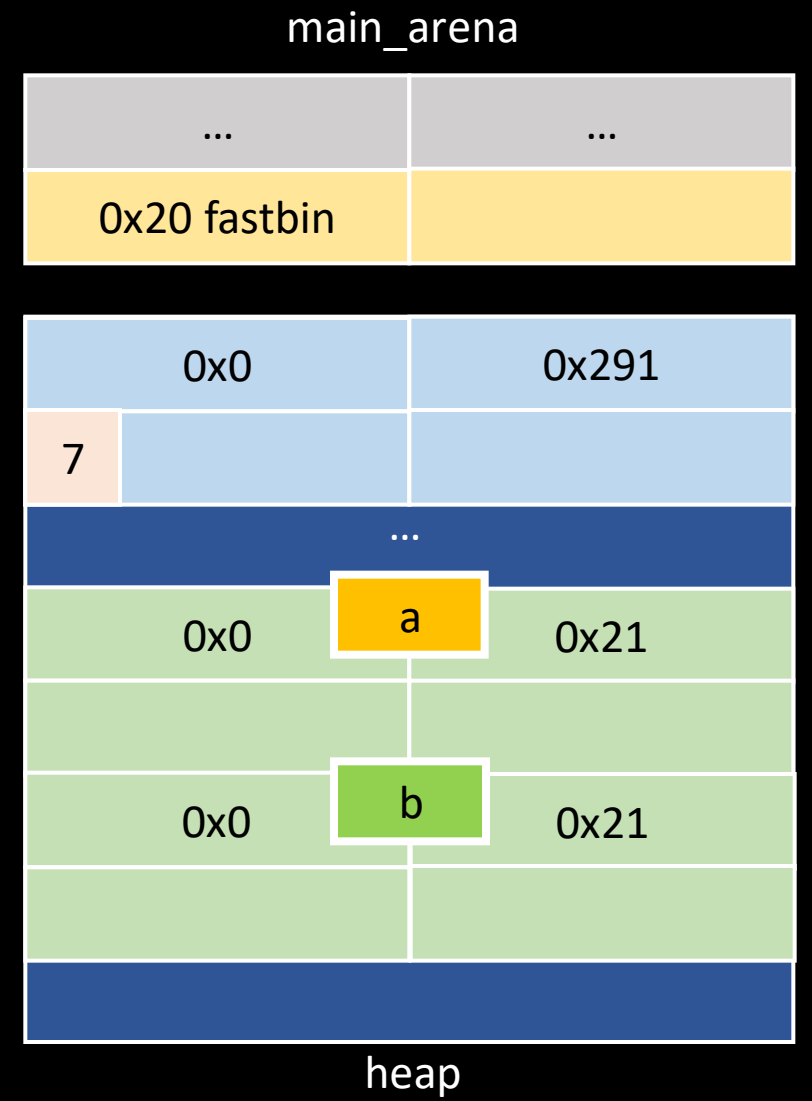
Double free


main_arena
...
...
0x20 fastbin
0x291
0x21
_int_free (mstate av, mchunkptr p, int have_lock)
{
size = chunksize (p);
...
unsigned int idx = fastbin_index(size);
fb = &fastbin (av, idx);
mchunkptr old = *fb;
...
if (__builtin_expect (old == p, 0))
malloc_printerr ("double free or corruption (fasttop)");
...
}
取 fastbin 的第一個 chunk
比對是否跟要 free 的 chunk 相同
就是不要連續兩個相同的 chunk 就好 :)

Double free

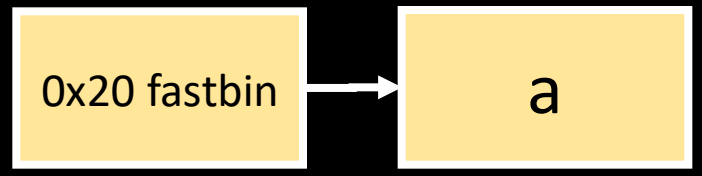
```
// fill tcache
free(a);
free(b);
free(a);

// clean tcahce
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xC8763
```

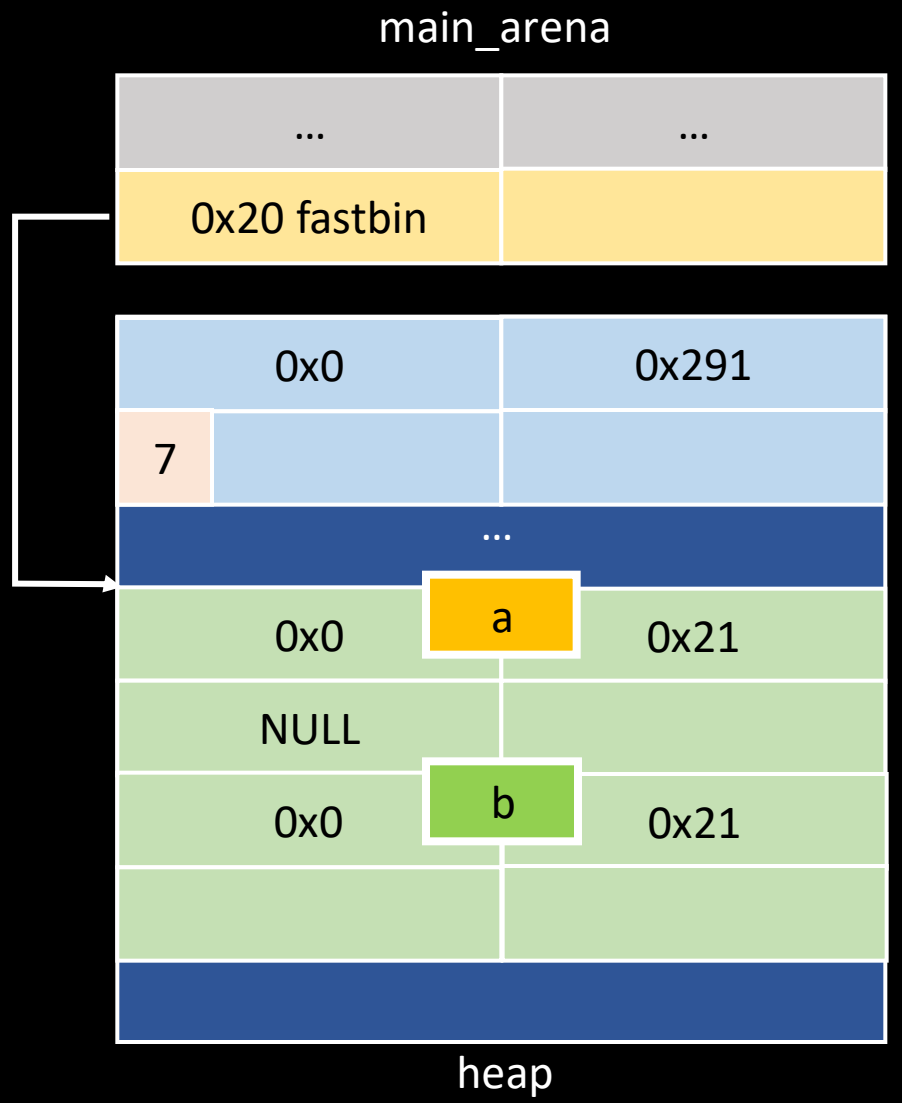
0x20 fastbin



Double free

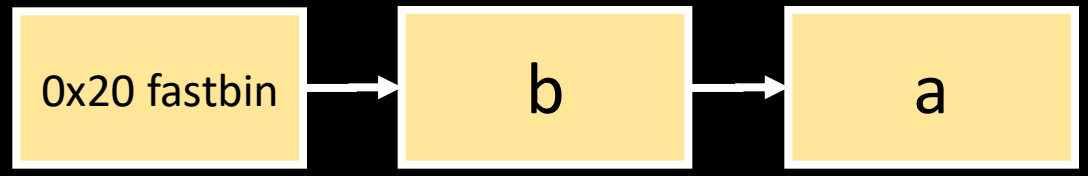
```
// fill tcache
free(a);
free(b);
free(a);

// clean tcahce
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xC8763
```

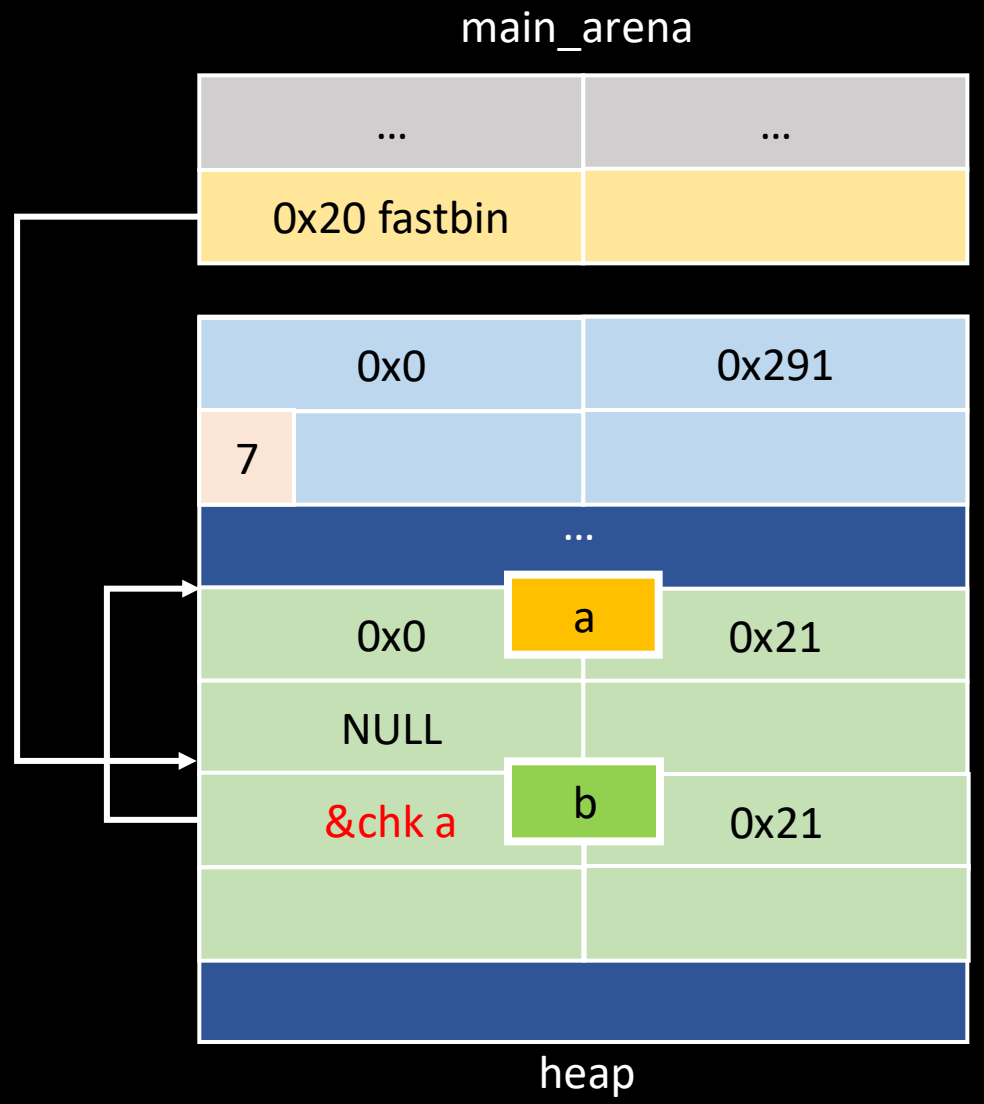




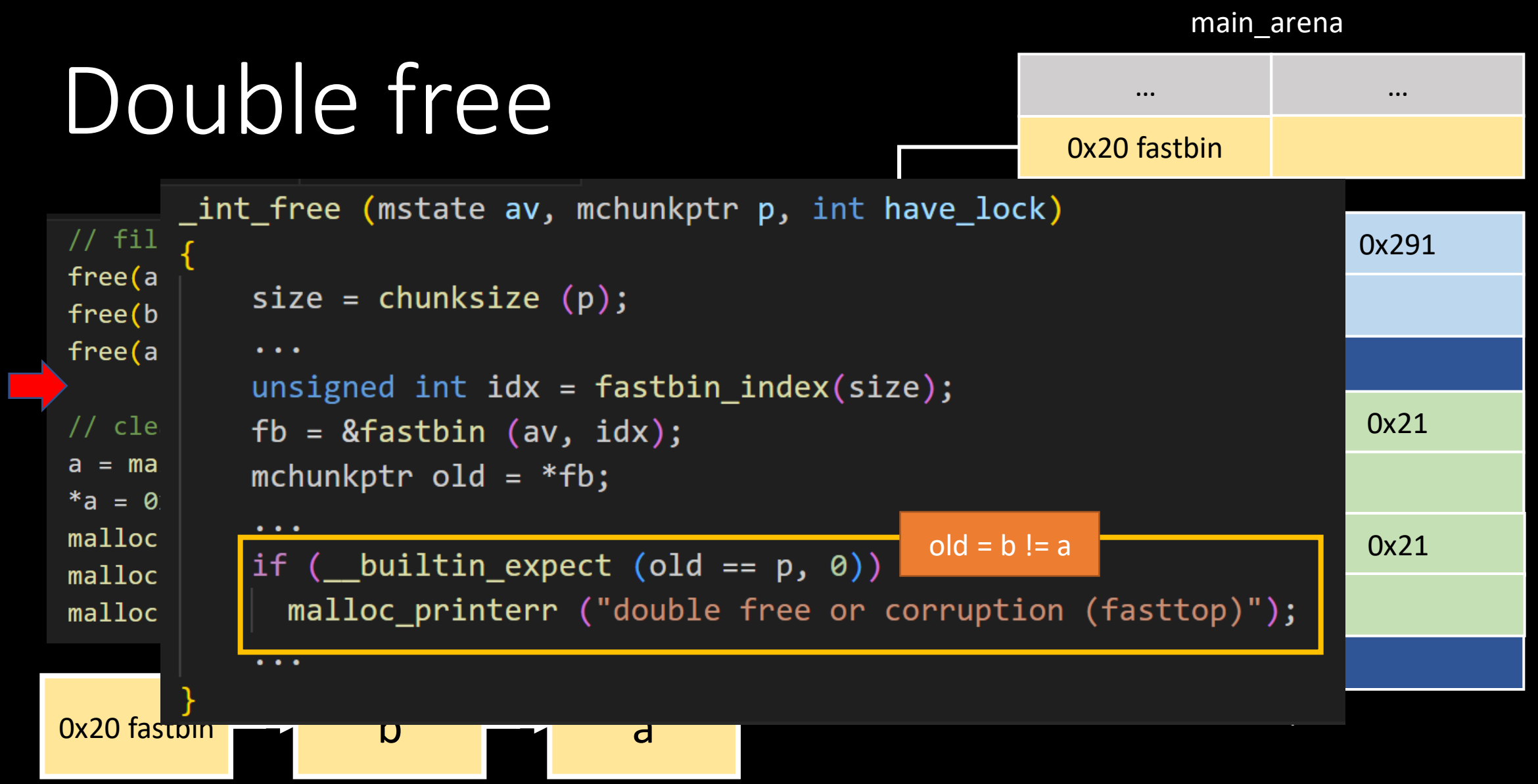
Double free

```
// fill tcache
free(a);
free(b);
free(a);

// clean tcahce
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xC8763
```



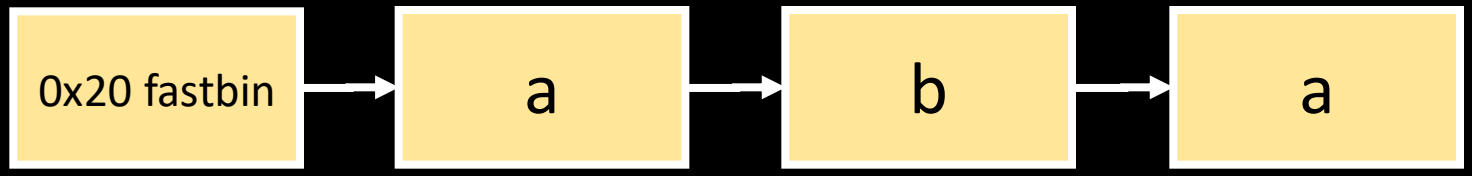


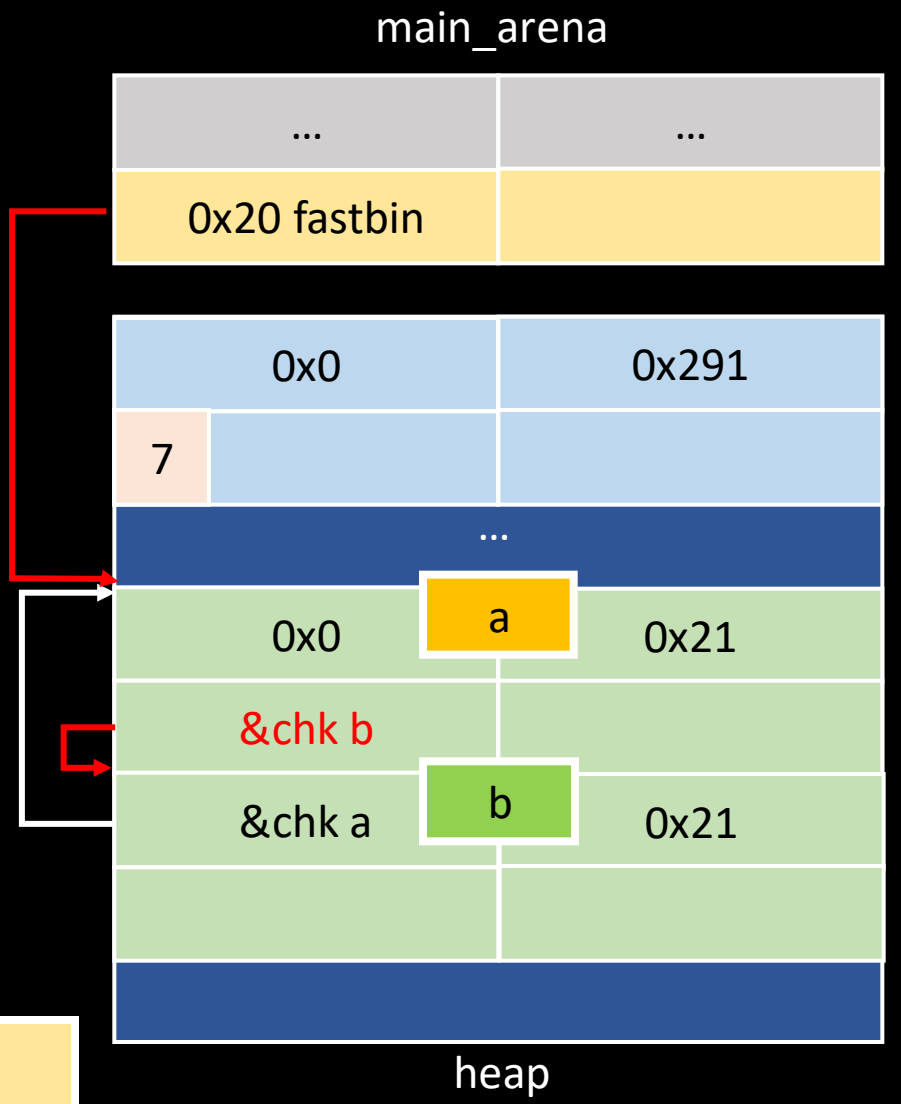
Double free


main_arena
...
...
0x20 fastbin
0x291
0x21
0x21
// fil
free(a
free(b
free(a
// cle
a = ma
*a = 0
malloc
malloc
malloc
_int_free (mstate av, mchunkptr p, int have_lock)
{
size = chunksize (p);
...
unsigned int idx = fastbin_index(size);
fb = &fastbin (av, idx);
mchunkptr old = *fb;
...
if (__builtin_expect (old == p, 0))
malloc_printerr ("double free or corruption (fasttop)");
...
}
old = b != a
0x20 fastbin
b
a

Double free

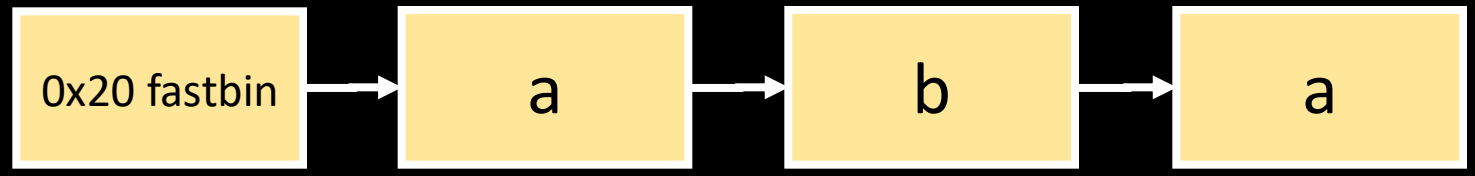
```
// fill tcache
free(a);
free(b);
free(a);

// clean tcahce
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xC8763
```

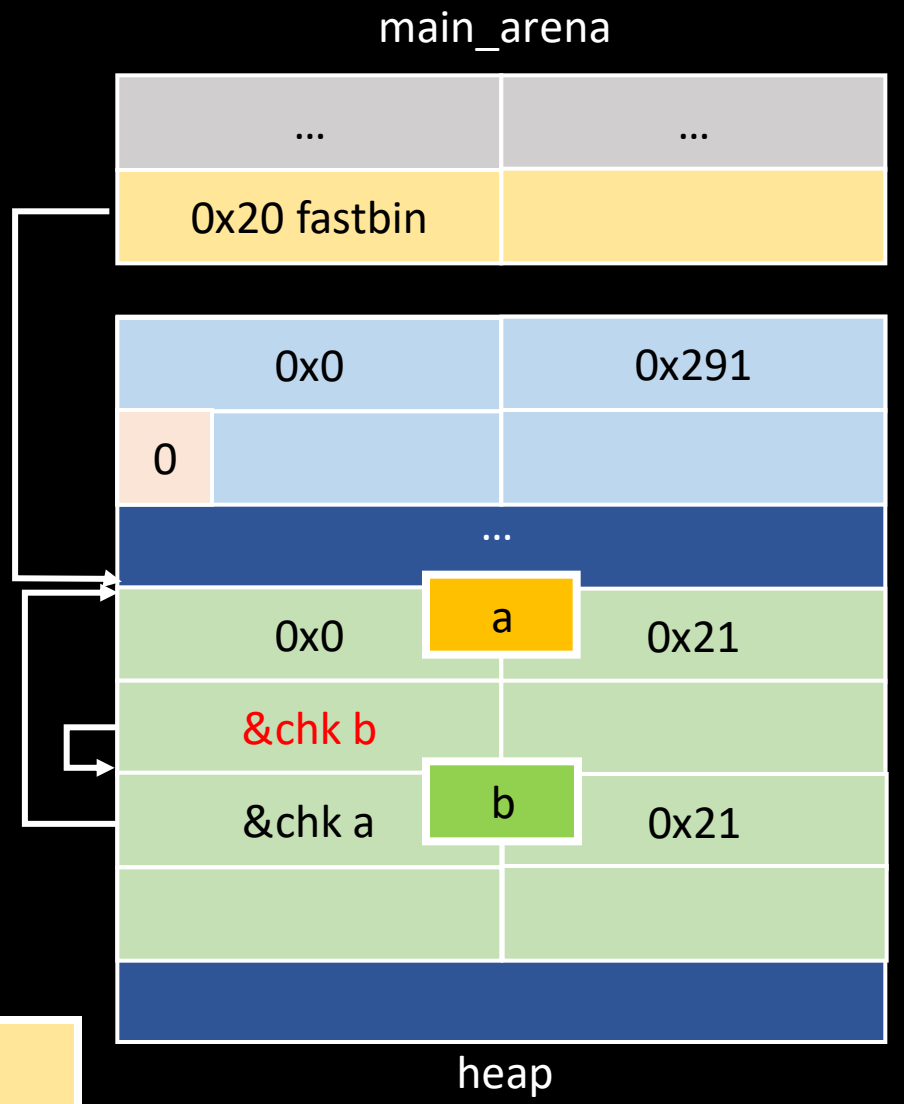




Double free

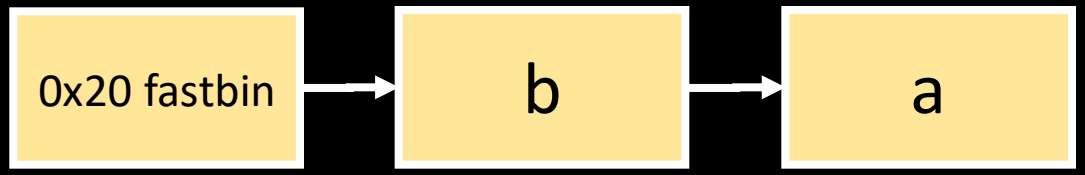
```
// fill tcache
free(a);
free(b);
free(a);

// clean tcahce
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xC8763
```

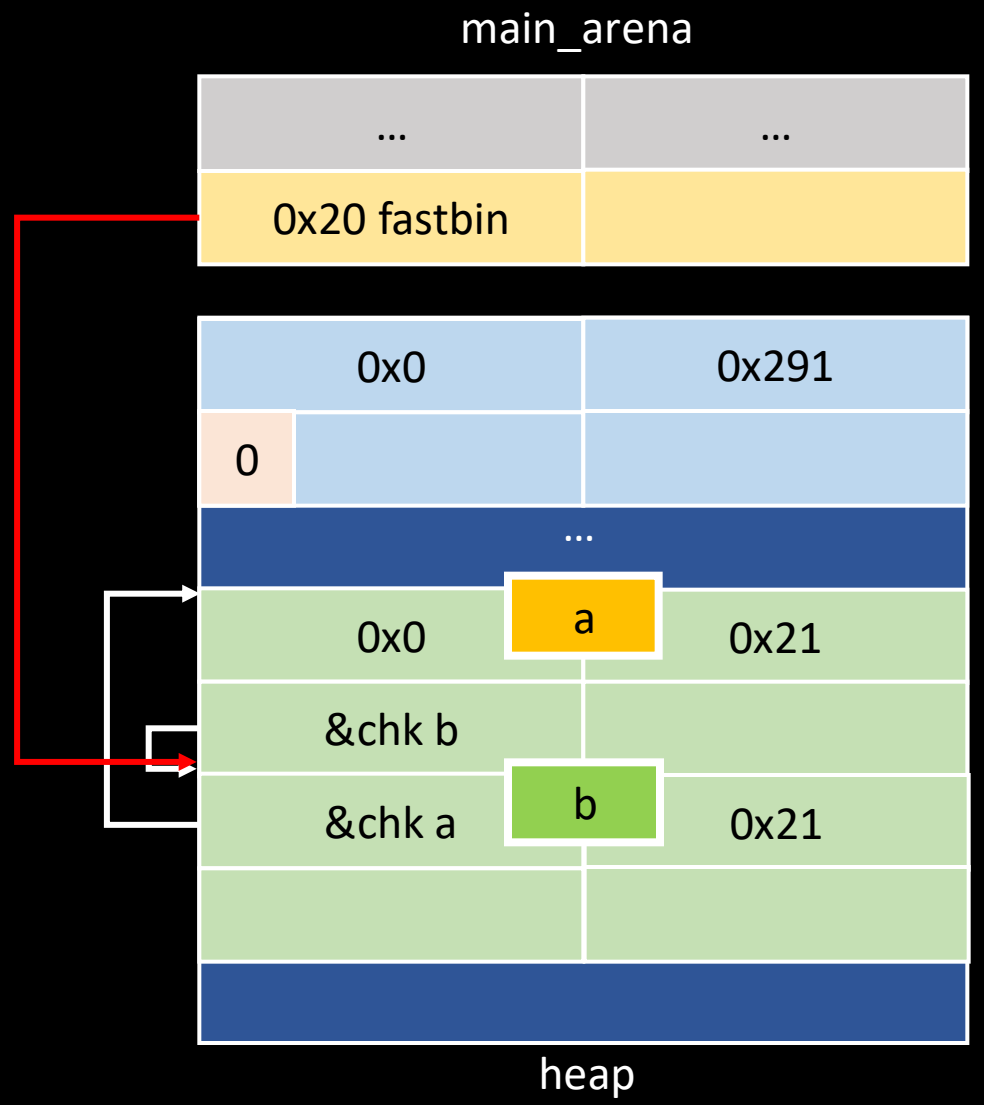




Double free

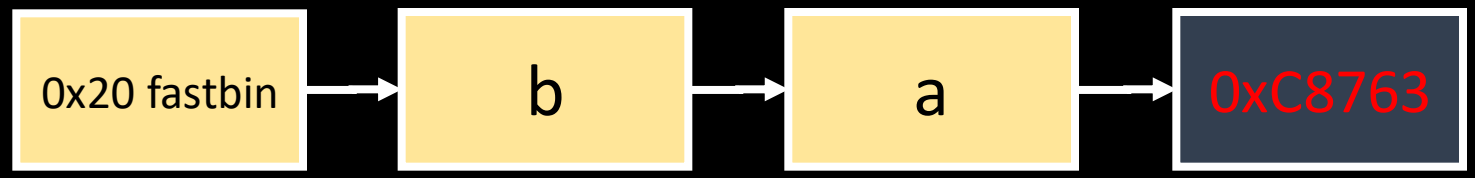
```
// fill tcache
free(a);
free(b);
free(a);

// clean tcahce
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xC8763
```

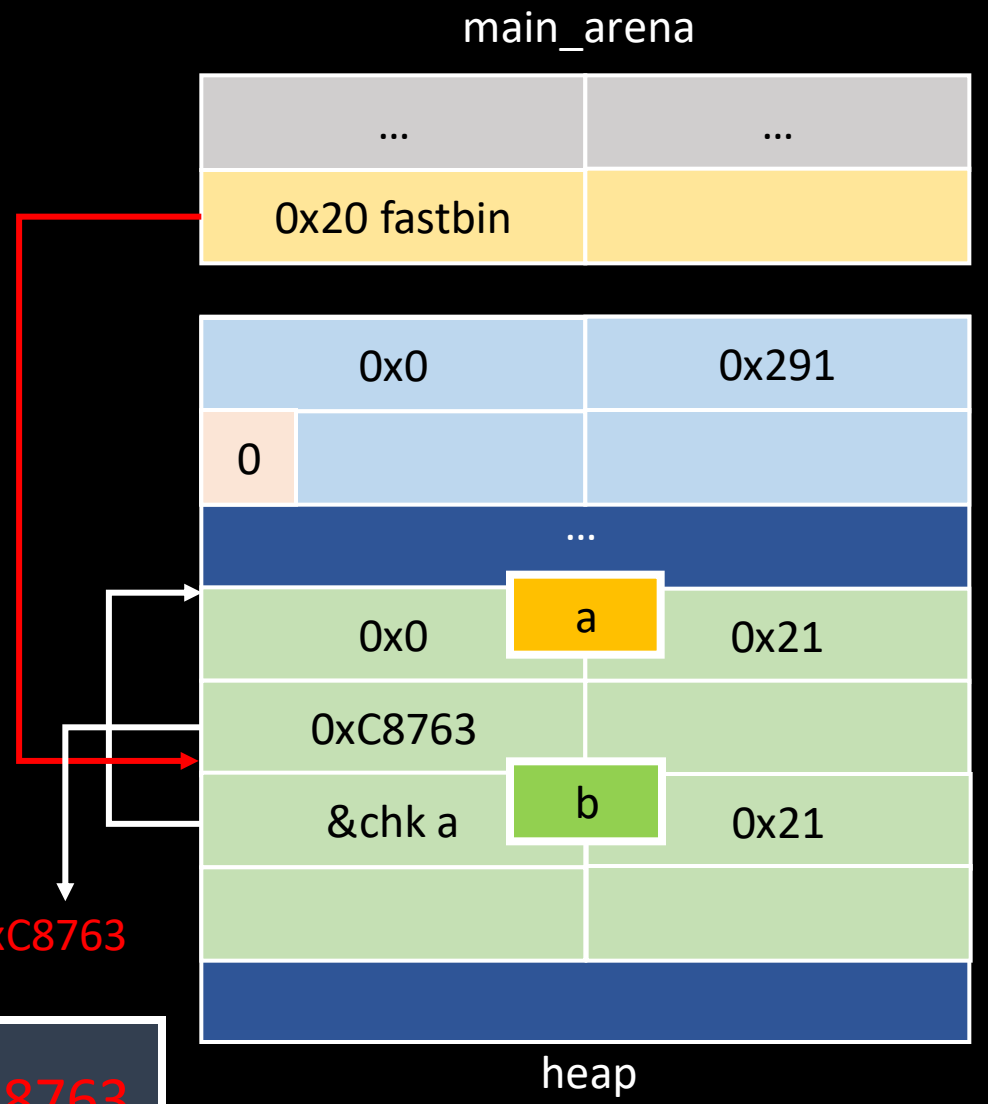




Double free

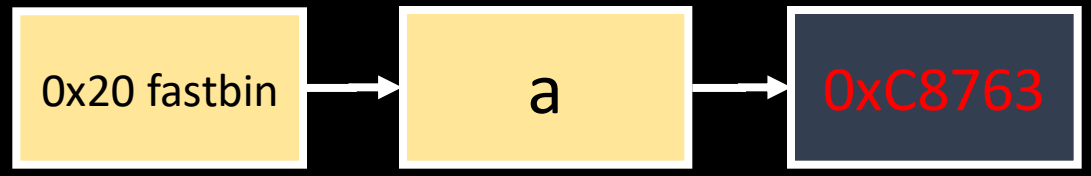
```
// fill tcache
free(a);
free(b);
free(a);

// clean tcahce
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xC8763
```

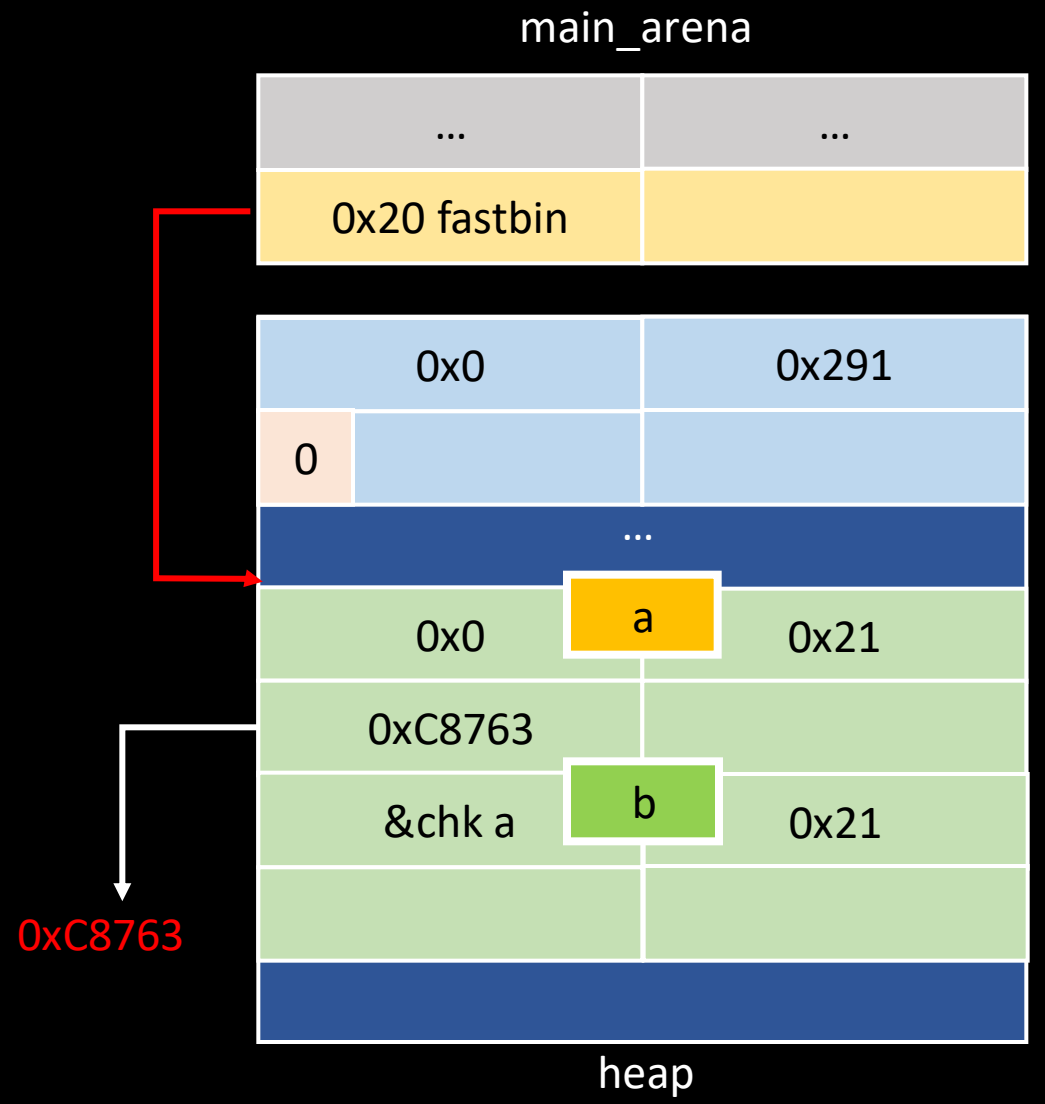




Double free

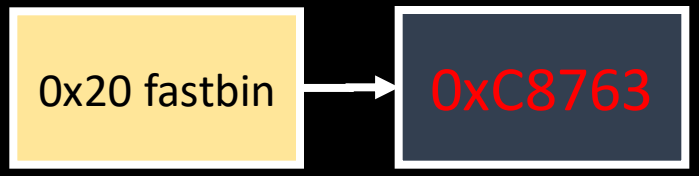
```
// fill tcache
free(a);
free(b);
free(a);

// clean tcahce
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xC8763
```

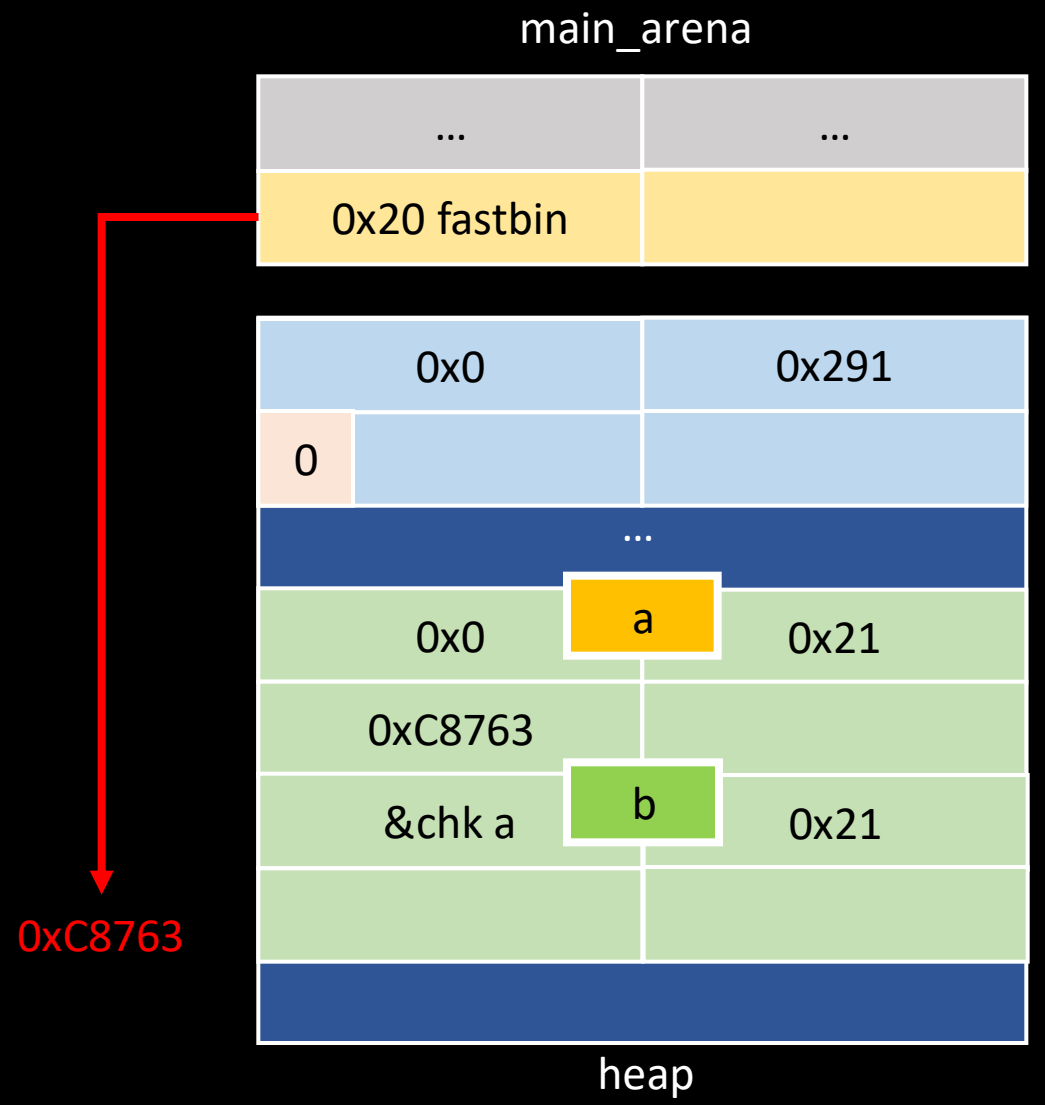




Double free

```
// fill tcache
free(a);
free(b);
free(a);

// clean tcahce
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xC8763
```

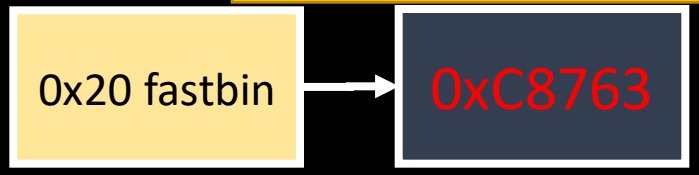


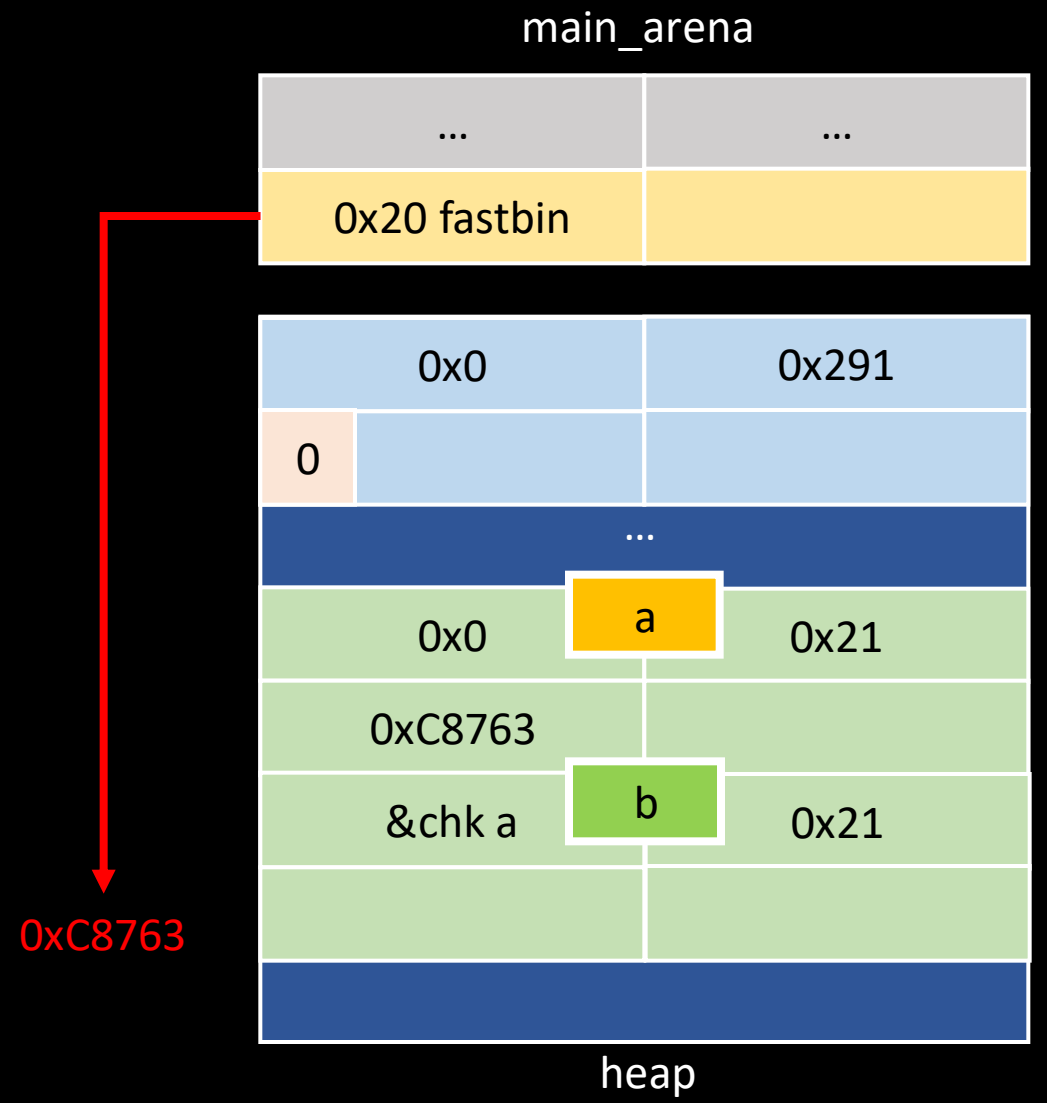


Double free

```
// fill tcache
free(a);
free(b);
free(a);

// clean tcahce
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xC8763
```

crash 因為 0xC8763 是 Invalid address





Exploitation goal

Exploitation goal

- 寫入 glibc 記憶體分配的 function 都有 hook 可以自訂行為
- 常見的有 __free_ hook, __malloc_hook, __realloc_hook
- 透過任意寫將 hook 改成想跳的位置就能任意執行
- Trace code

Exploitation goal

- __free_hook 可以寫成 system 然後 free chunk 內容為 “sh” 的 chunk 就能開 shell
- __malloc_hook 和 __realloc_hook 就只能寫成 one_gadget
  - one_gadget 為存在在 libc 的一個 code 片段只要條件符合跳上去就會執行 /bin/sh
  - 可以透過 one_gadget 來找

Exploitation tech

Exploitation tech

- 根據可以使用的功能會有不同的攻擊手法，這邊就只介紹下面兩種較常見的攻擊，其他的可以自行去 how2heap 看
  - Tcache poisoning
  - Overlapping chunks

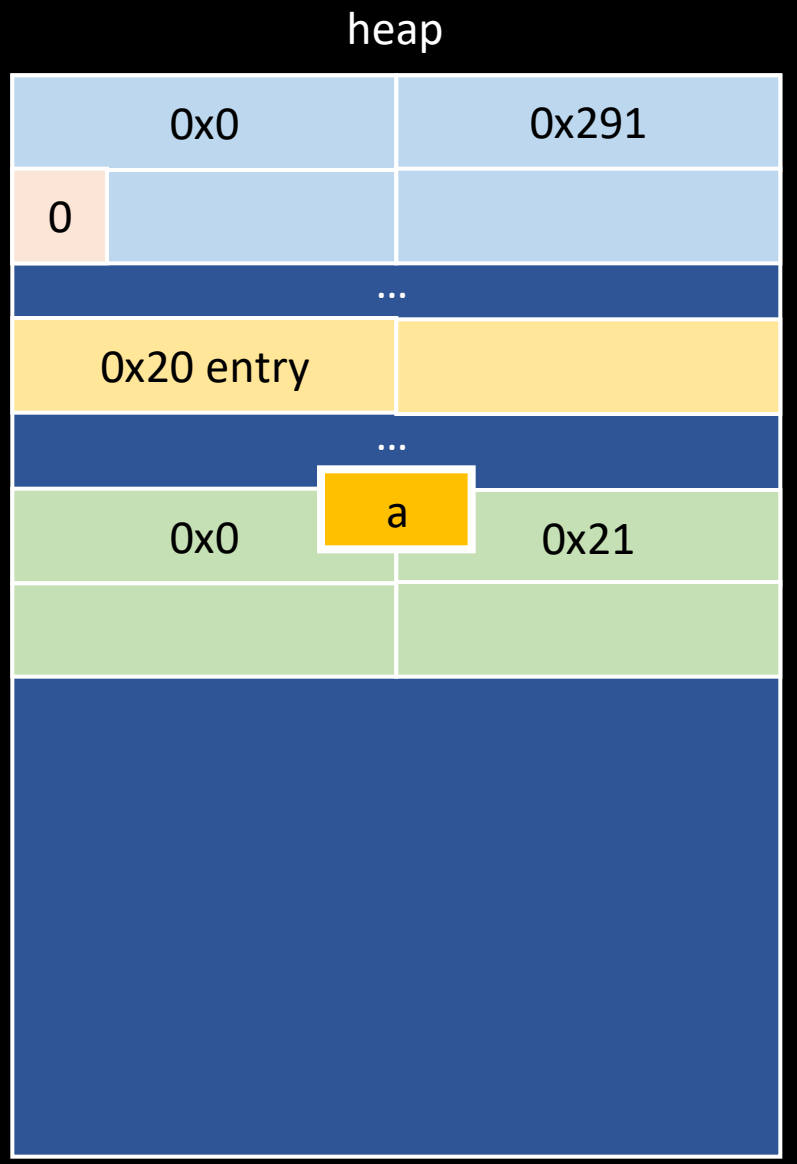
Tcache poisoning

- 透過 double free 讓 tcache 存在相同的兩個 chunk 來更改 next，再修改 next 來做到任意寫
- free 時會檢查 entry->key == tcache (tcache_struct address) 就去遍歷整個 tcache 找有沒有相同的 chunk，有的話就會 abort
  - 只要透過 UAF 或 heap overflow 把 key 寫成不是 tcahce 就好
- Tcache 要 count > 0 才能 malloc
  - 取得 tcache_struct chunk 再去改 counts 或是多 free 幾次就可以了
- 可以直接把 __free_hook 寫成 system

Tcache poisoning

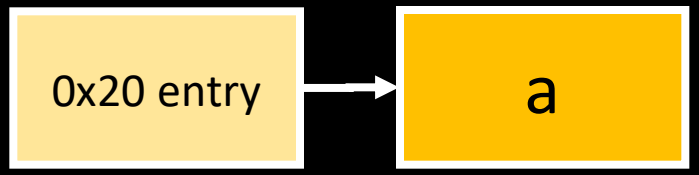
```
a = malloc(0x10);
free(a);
a[1] = 0;
free(a);
a[1] = 0;
free(a);
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10); // 0xC8763
```

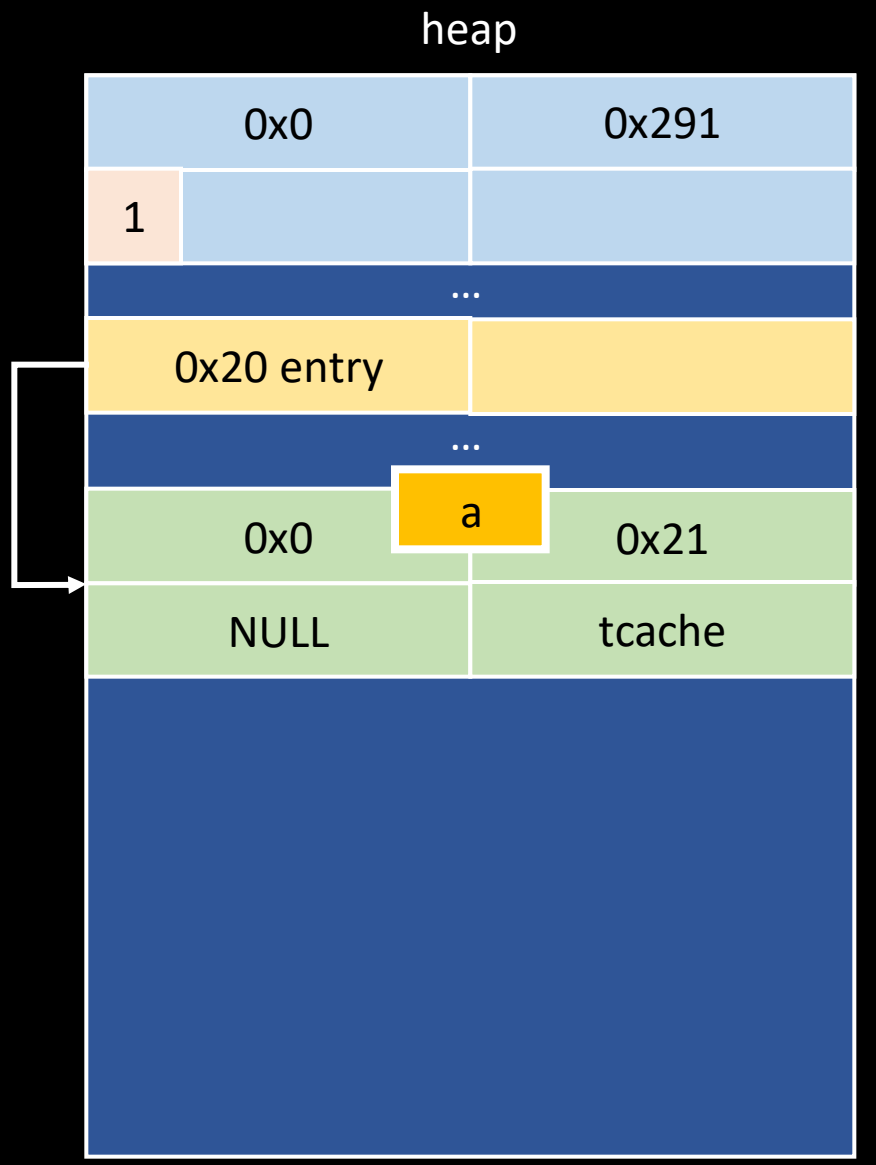
0x20 entry



Tcache poisoning

```
a = malloc(0x10);
free(a);
a[1] = 0;
free(a);
a[1] = 0;
free(a);
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10); // 0xC8763
```

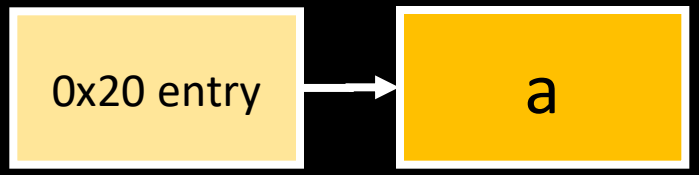


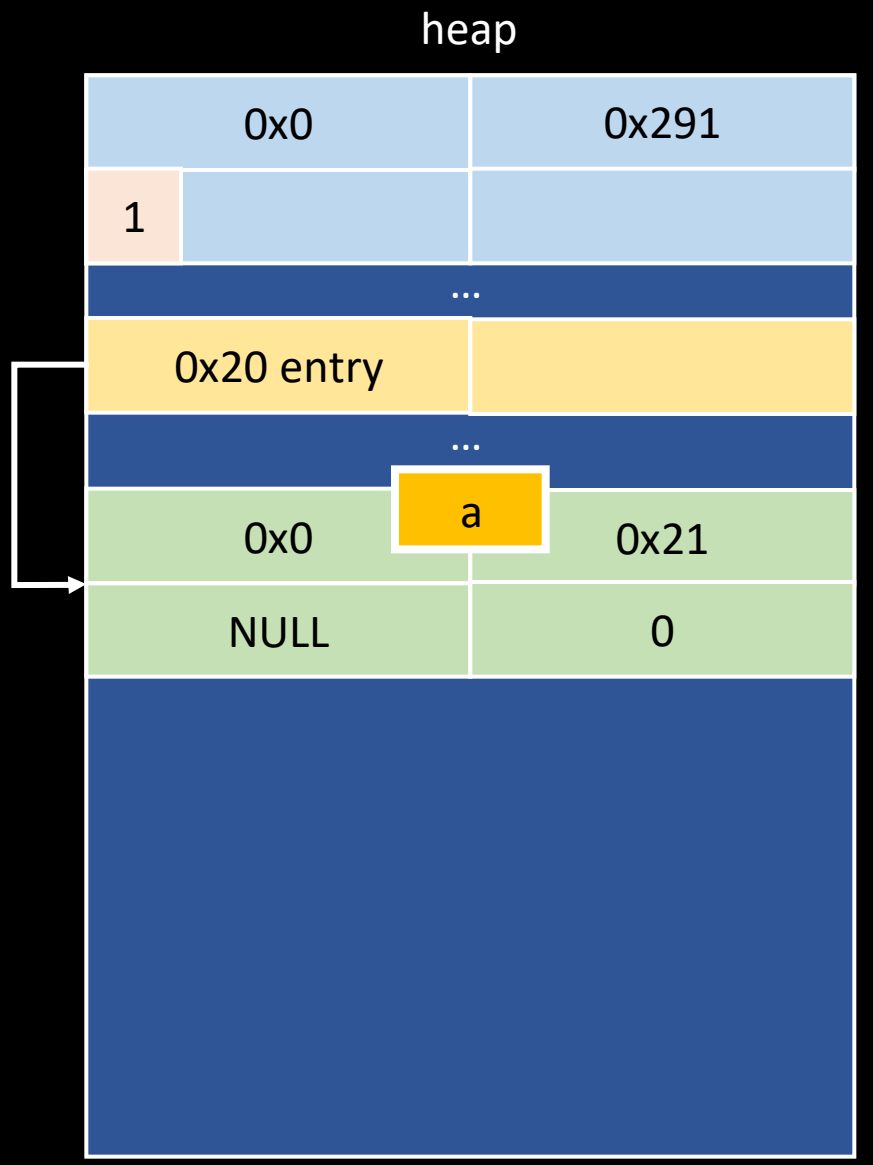


Tcache poisoning

```
a = malloc(0x10);
free(a);
a[1] = 0;
free(a);
a[1] = 0;
free(a);
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10); // 0xC8763
```

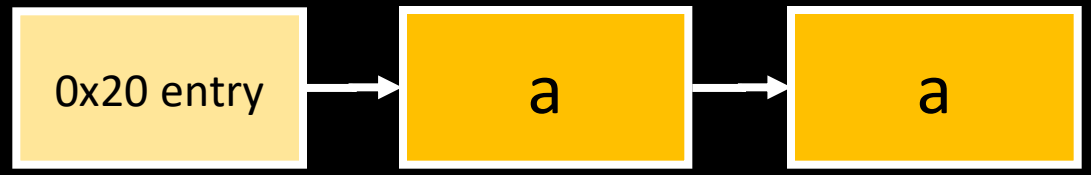
e->key = 0 != tcache

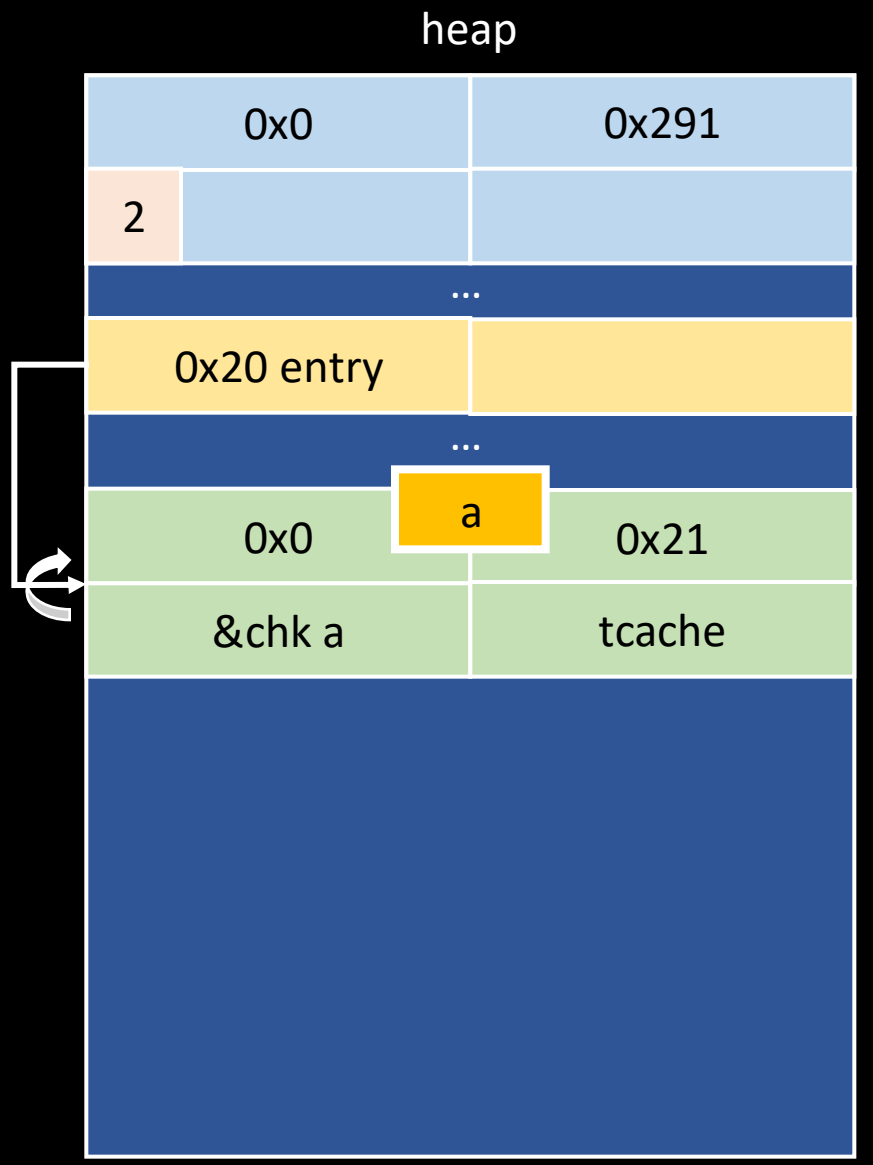




Tcache poisoning

```
a = malloc(0x10);
free(a);
a[1] = 0;
free(a);
a[1] = 0;
free(a);
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10); // 0xC8763
```

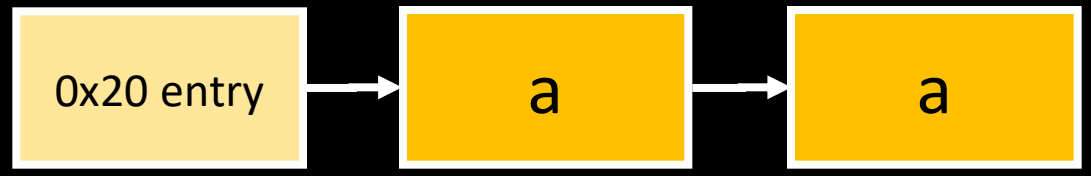


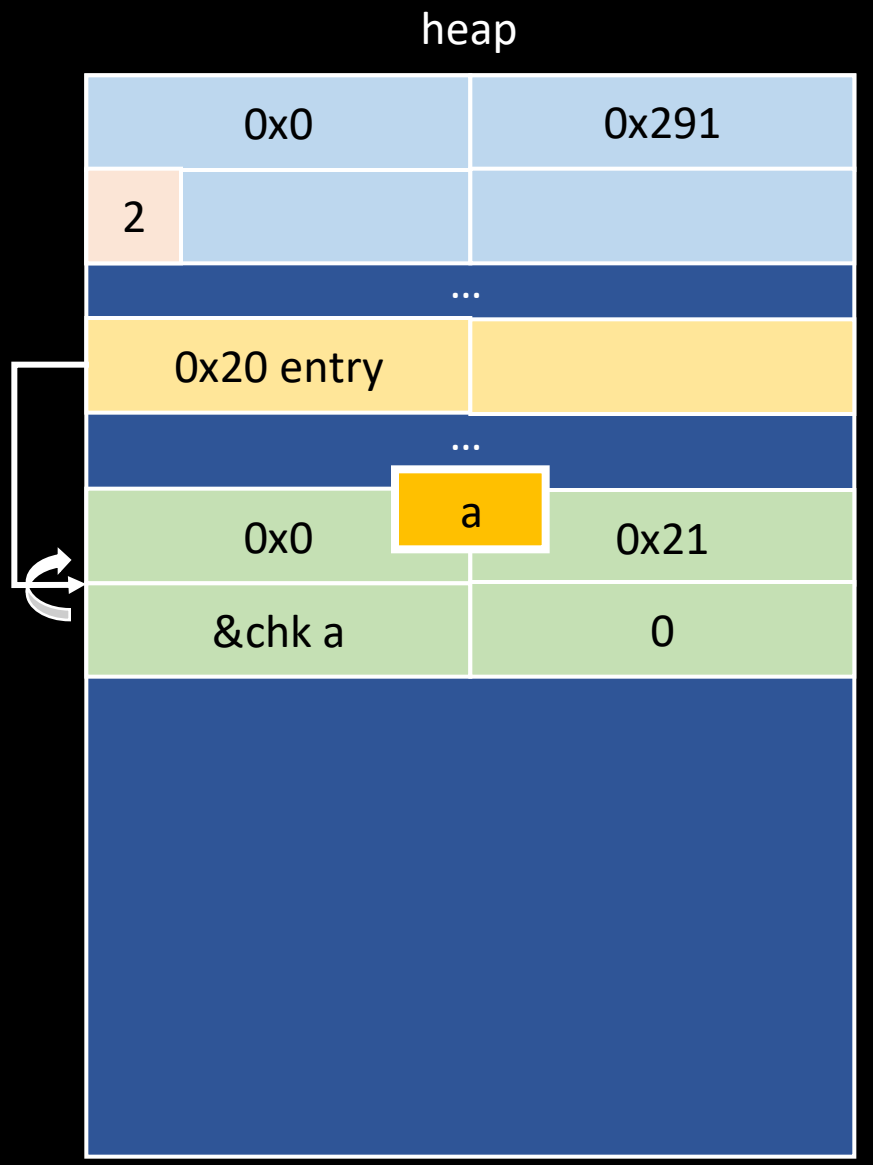


Tcache poisoning

```
a = malloc(0x10);
free(a);
a[1] = 0;
free(a);
a[1] = 0;
free(a);
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10); // 0xC8763
```

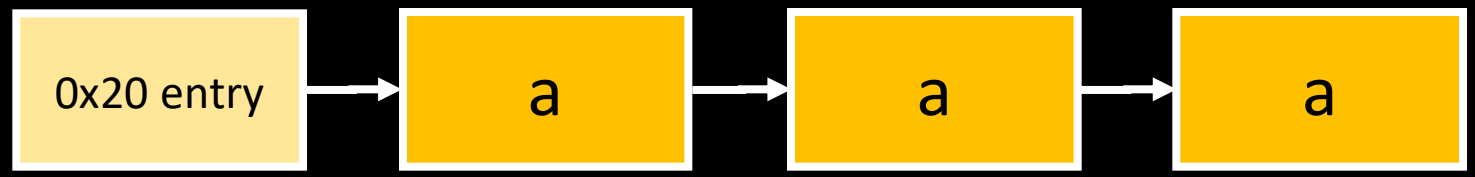
e->key = 0 != tcache

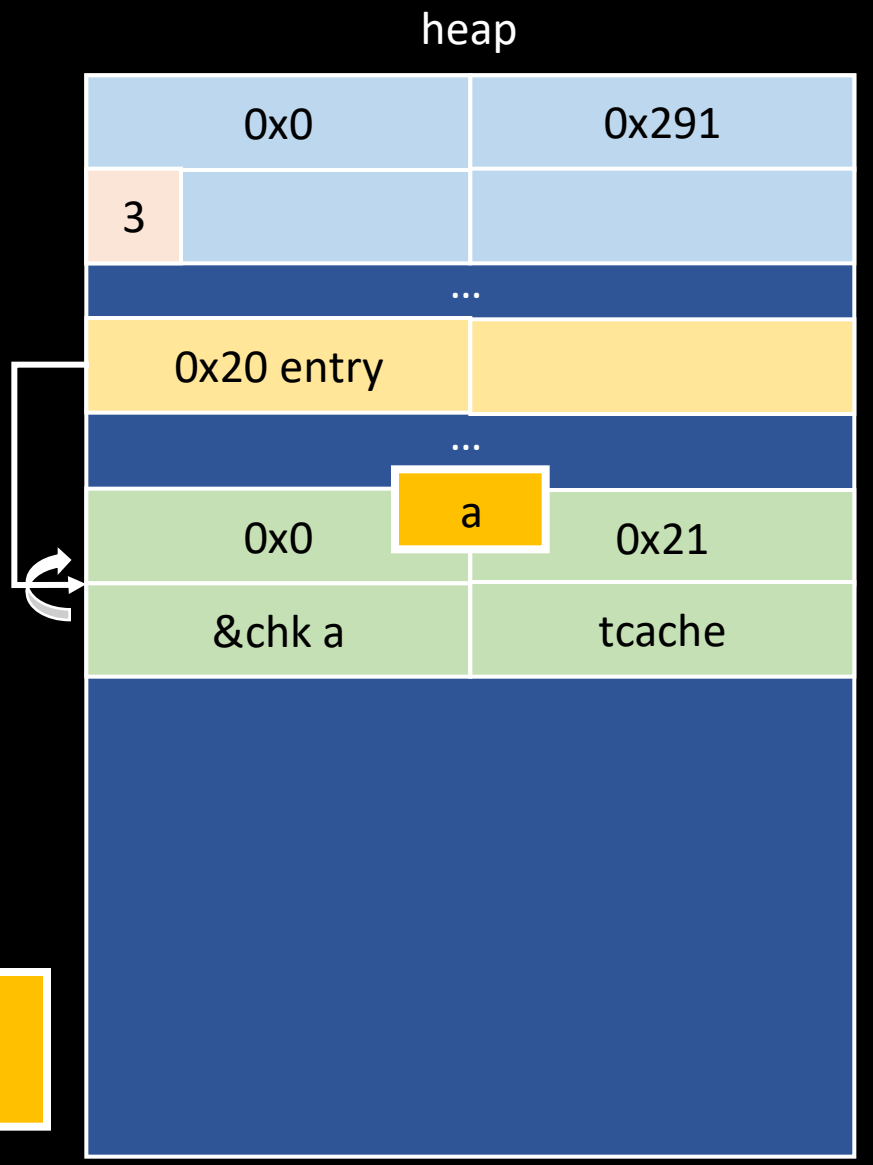




Tcache poisoning

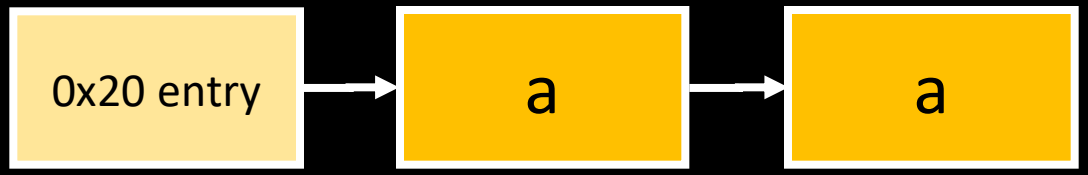
```
a = malloc(0x10);
free(a);
a[1] = 0;
free(a);
a[1] = 0;
free(a);
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10); // 0xC8763
```

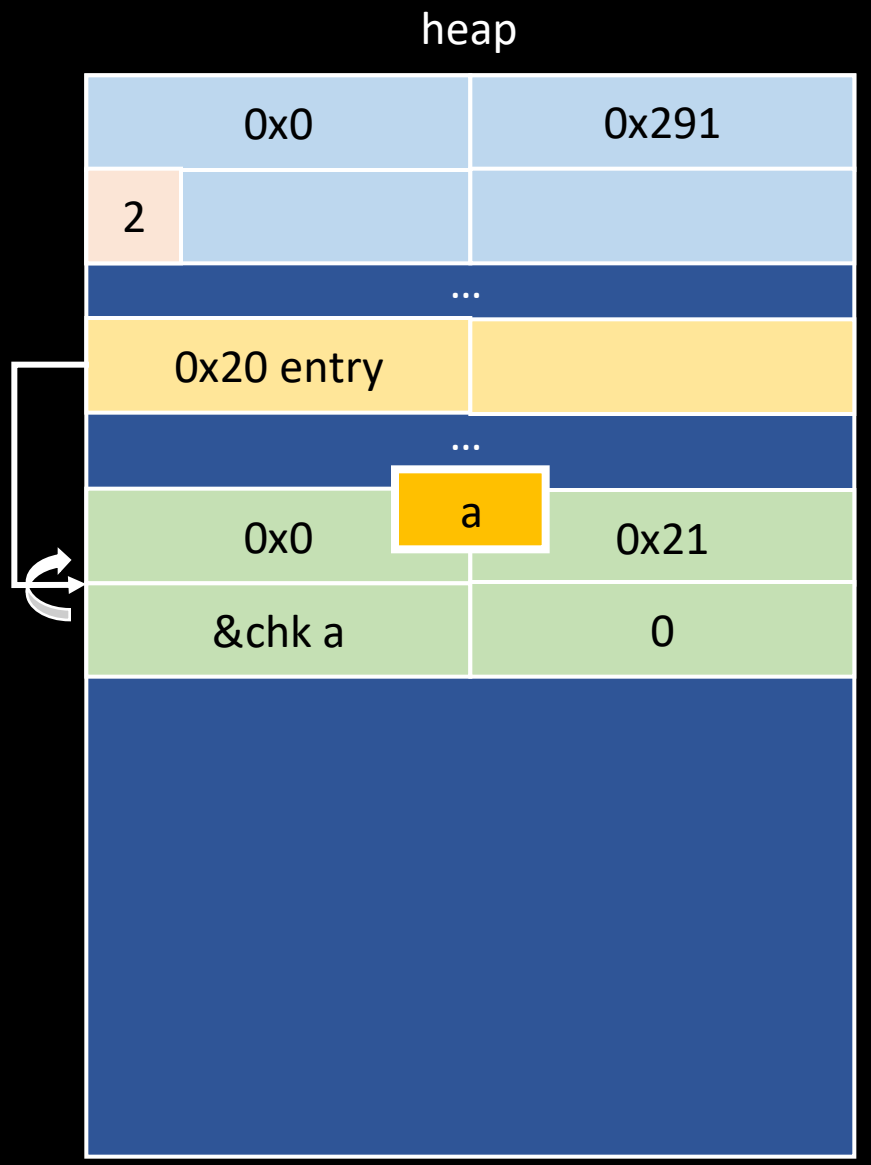




Tcache poisoning

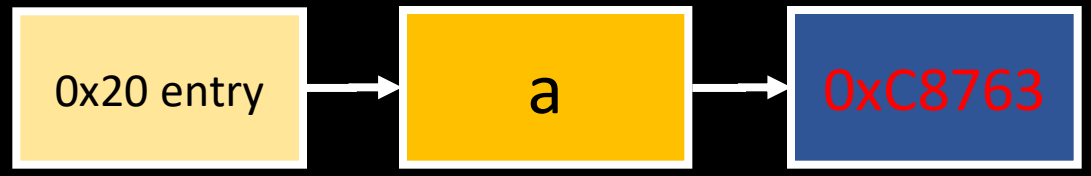
```
a = malloc(0x10);
free(a);
a[1] = 0;
free(a);
a[1] = 0;
free(a);
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10); // 0xC8763
```

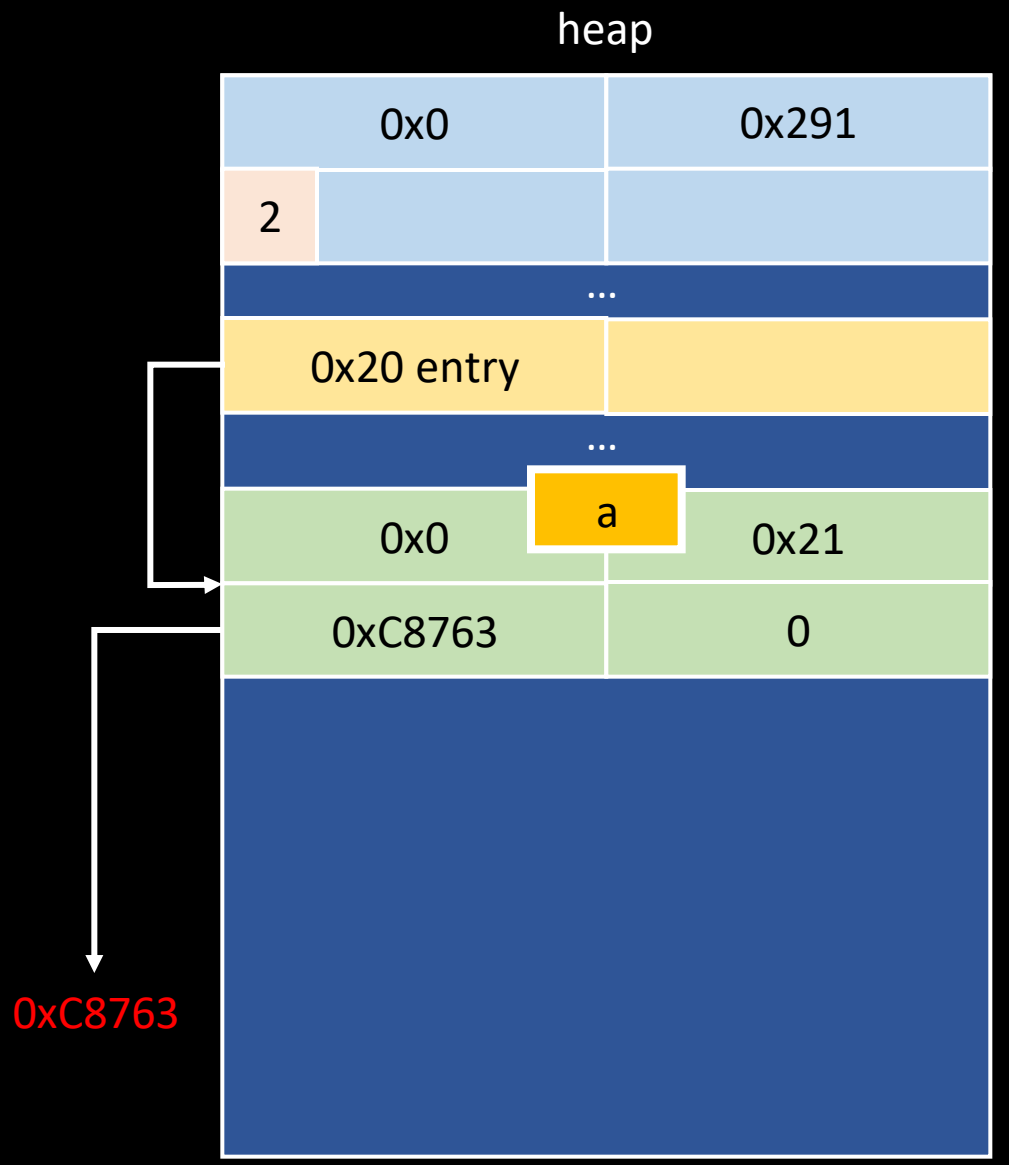




Tcache poisoning

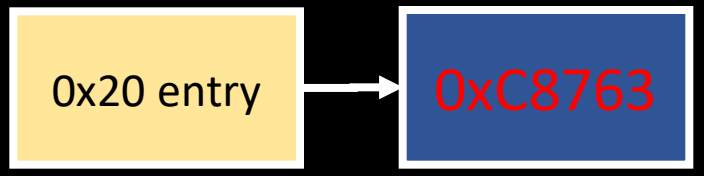
```
a = malloc(0x10);
free(a);
a[1] = 0;
free(a);
a[1] = 0;
free(a);
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10); // 0xC8763
```

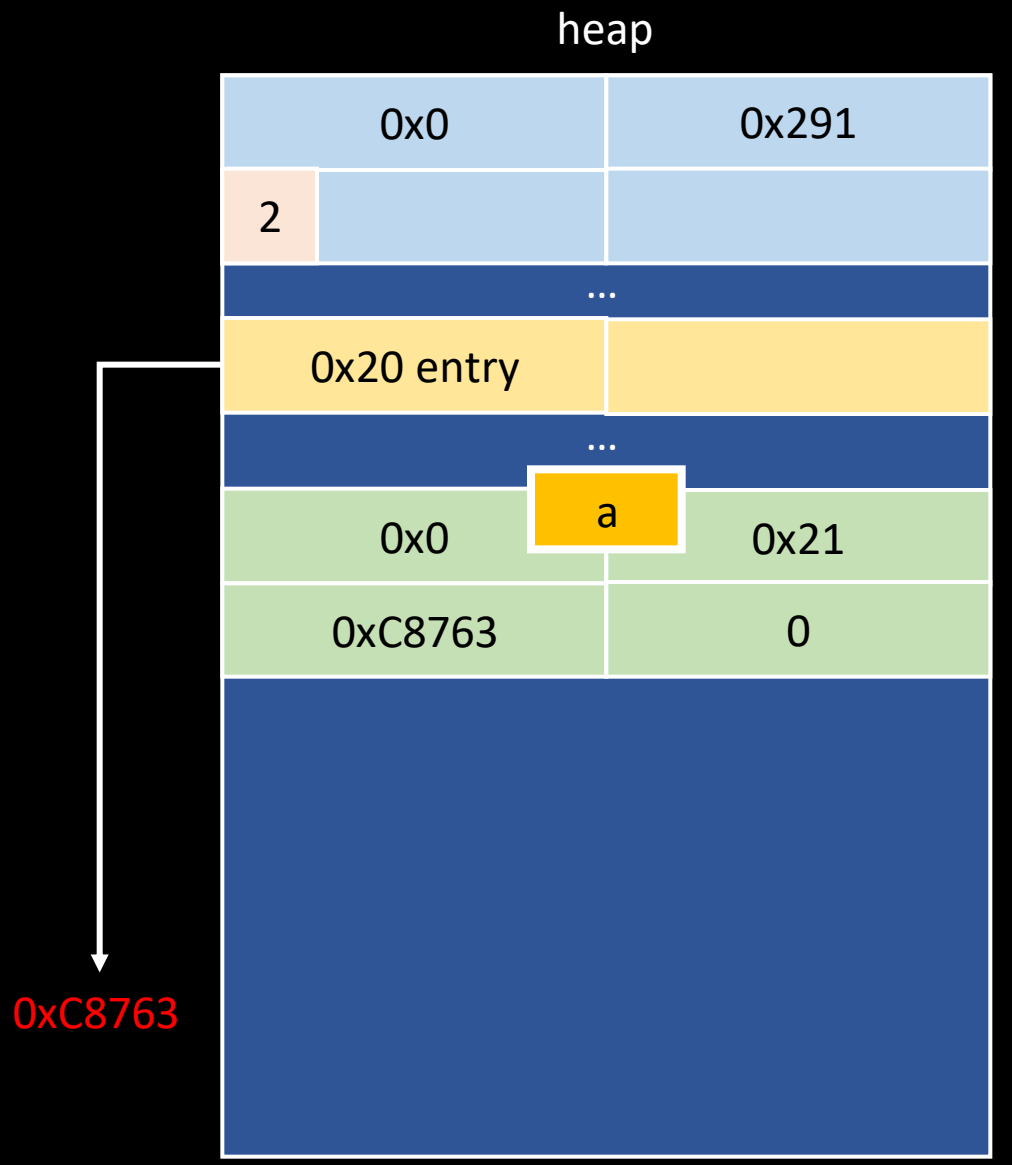




Tcache poisoning

```
a = malloc(0x10);
free(a);
a[1] = 0;
free(a);
a[1] = 0;
free(a);
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10); // 0xC8763
```

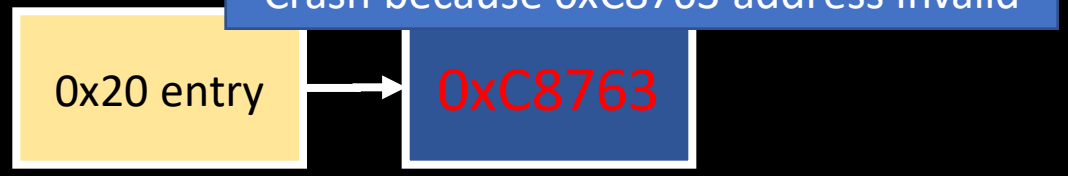


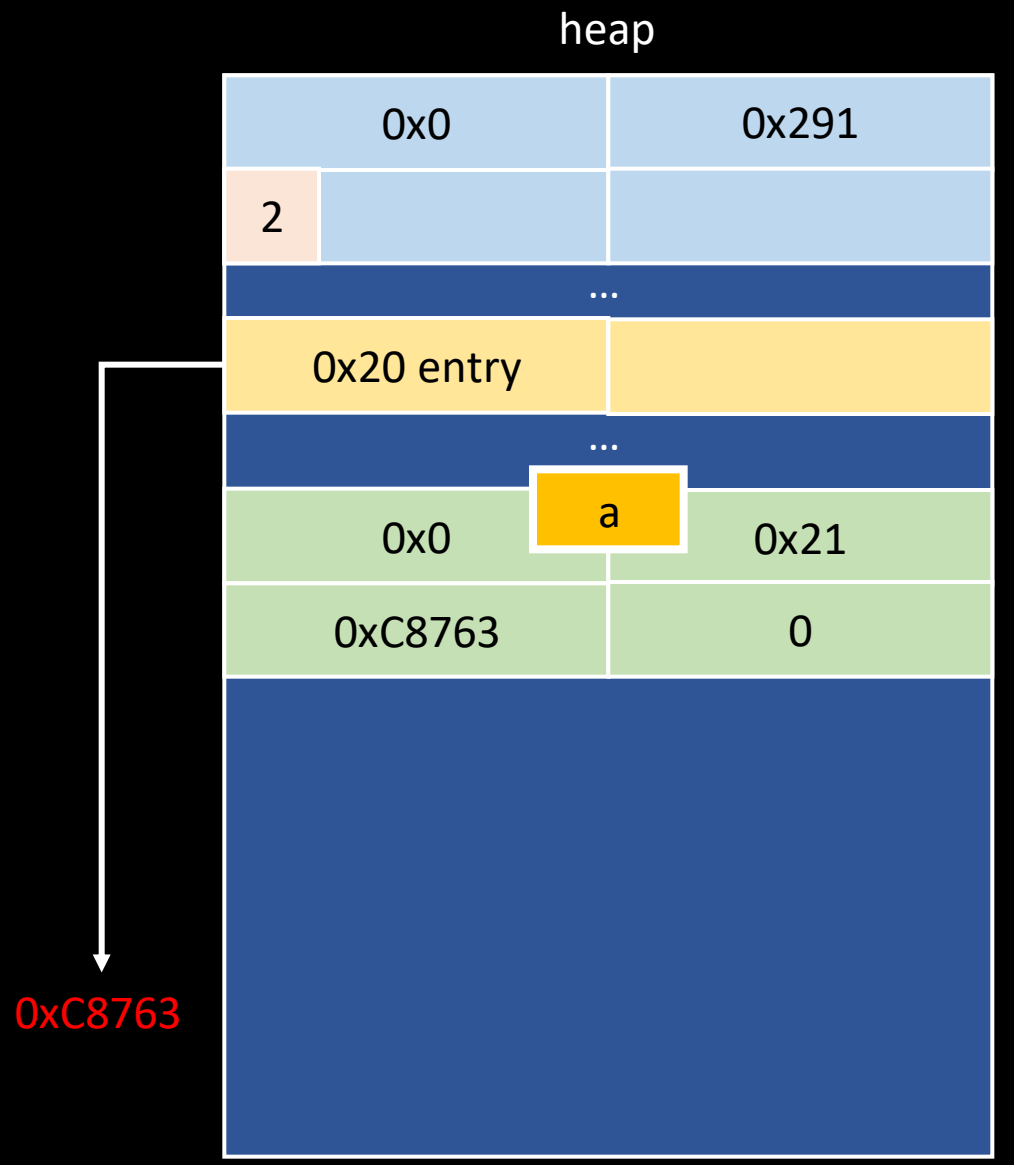


Tcache poisoning

```
a = malloc(0x10);
free(a);
a[1] = 0;
free(a);
a[1] = 0;
free(a);
a = malloc(0x10);
*a = 0xC8763;
malloc(0x10);
malloc(0x10); // 0xC8763
```

Crash because 0xC8763 address Invalid





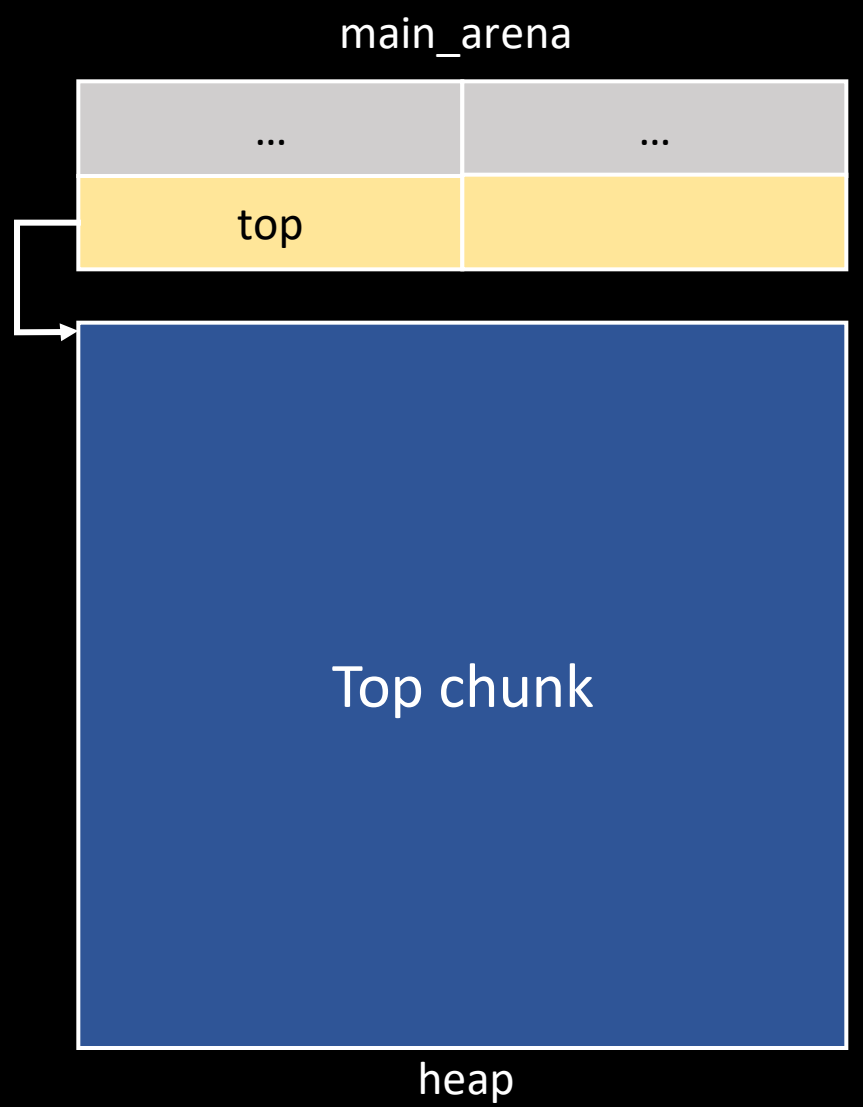
Lab Double Free

Overlapping chunks

- 透過更改 chunk size 讓 chunk 在被 consolidate 讓 allocated chunk 和 freed chunk 重疊
- 可以透過 allocate chunk 的功能去修改 freed chunk 的 fd, bk
- 或是在 malloc 取得 freed chunk 來修改 allocated chunk 的敏感資料

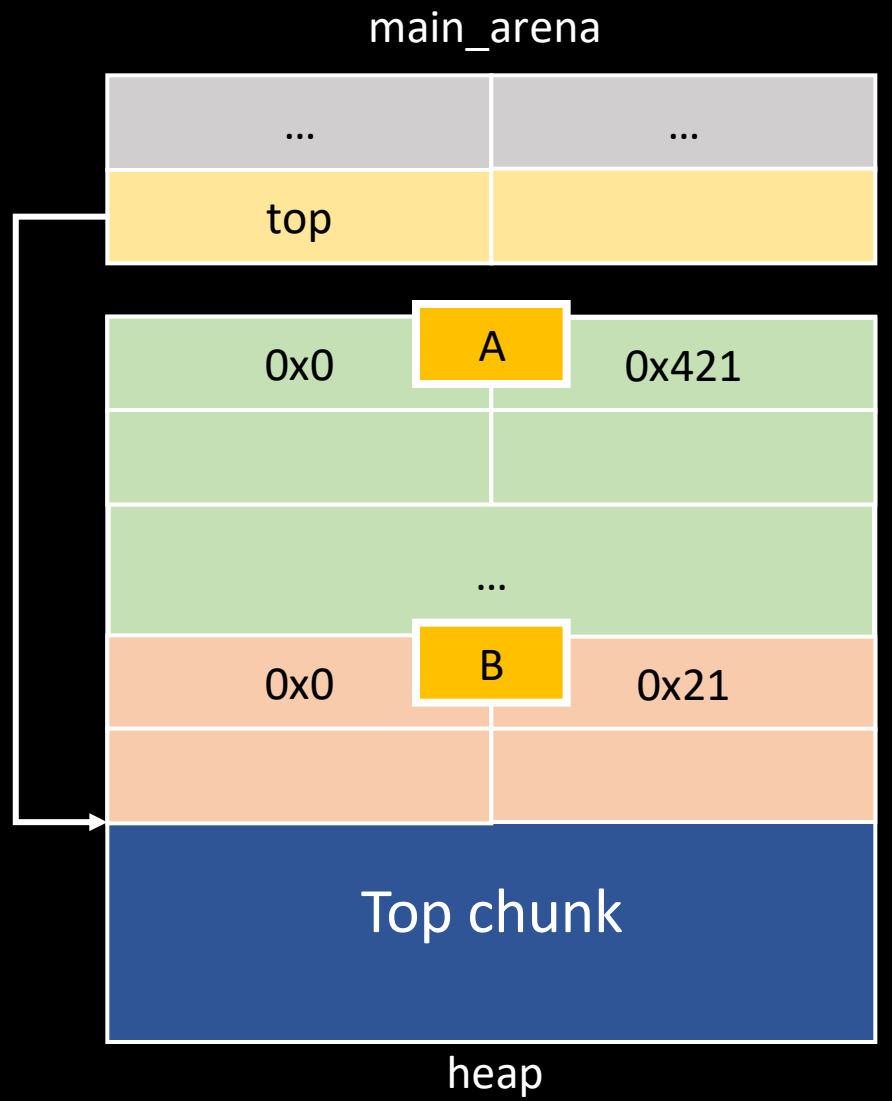
Overlapping chunks

```
A = malloc(0x410);
B = malloc(0x10);
*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk
A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```



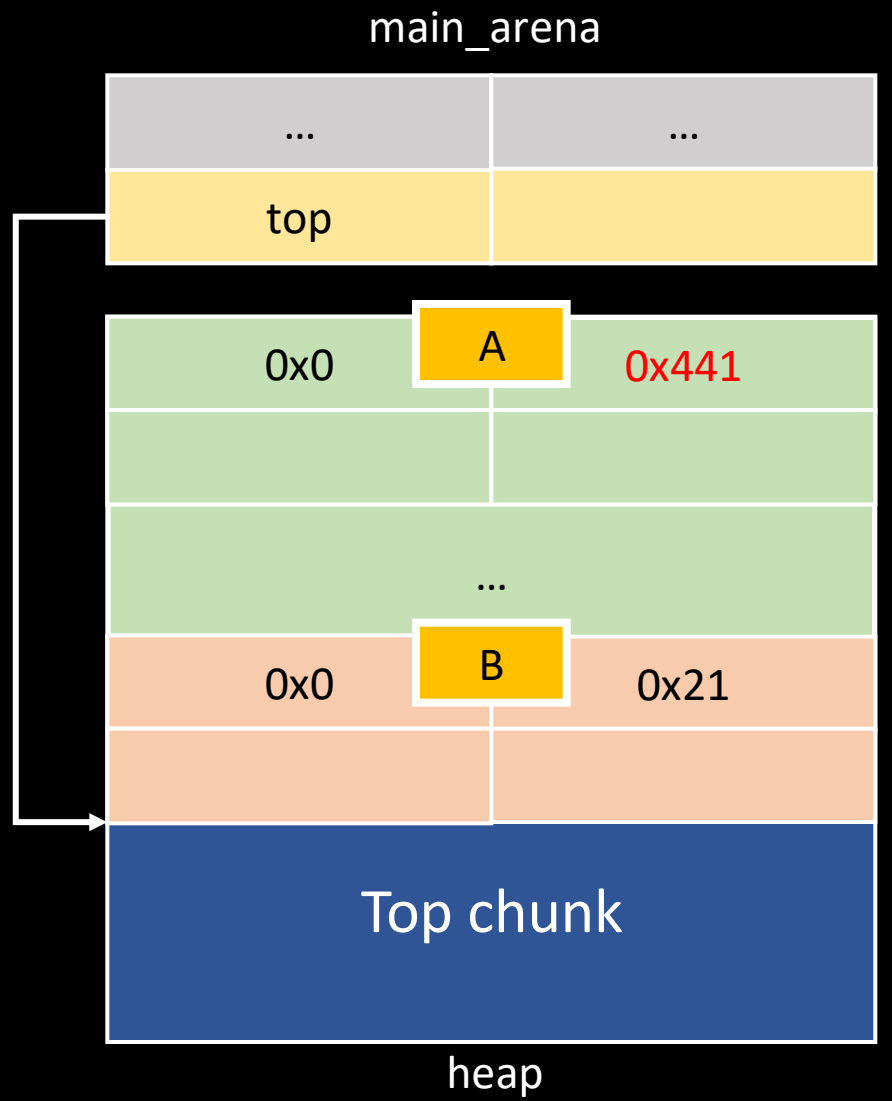
Overlapping chunks

```
A = malloc(0x410);
B = malloc(0x10);
*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk
A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```



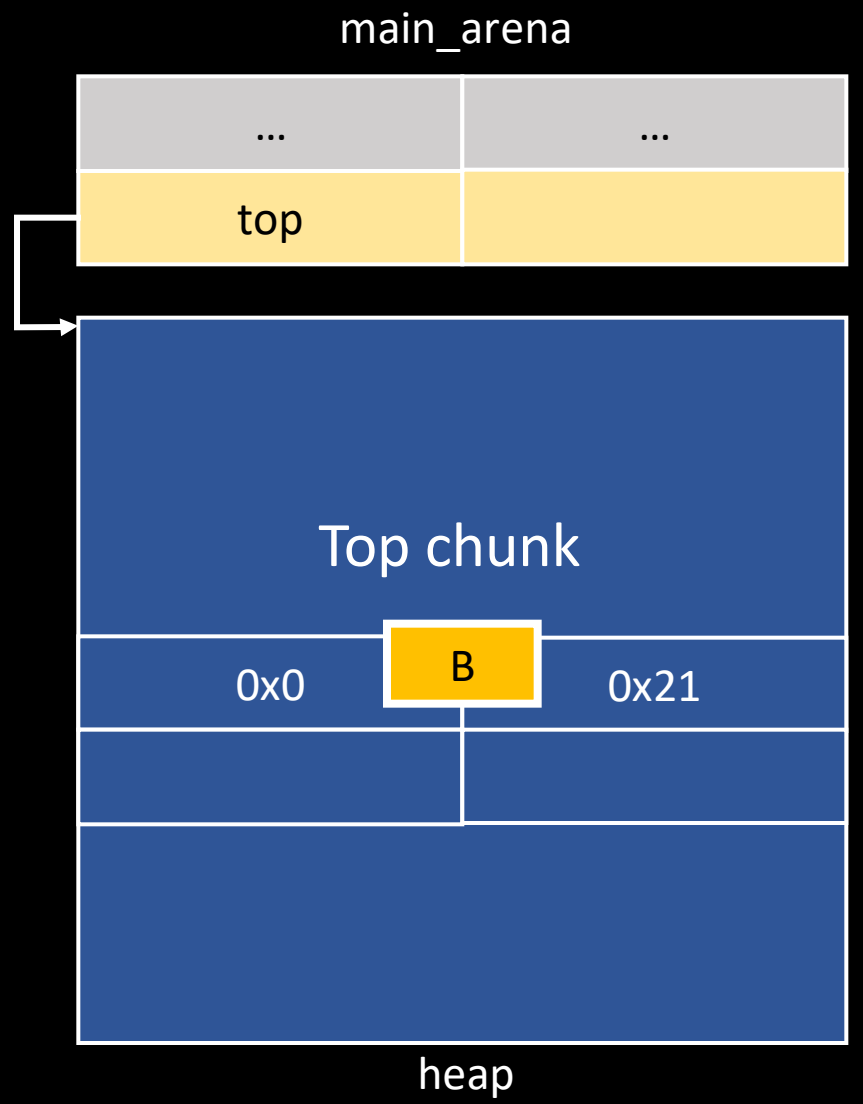
Overlapping chunks

```
A = malloc(0x410);
B = malloc(0x10);
*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk
A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```



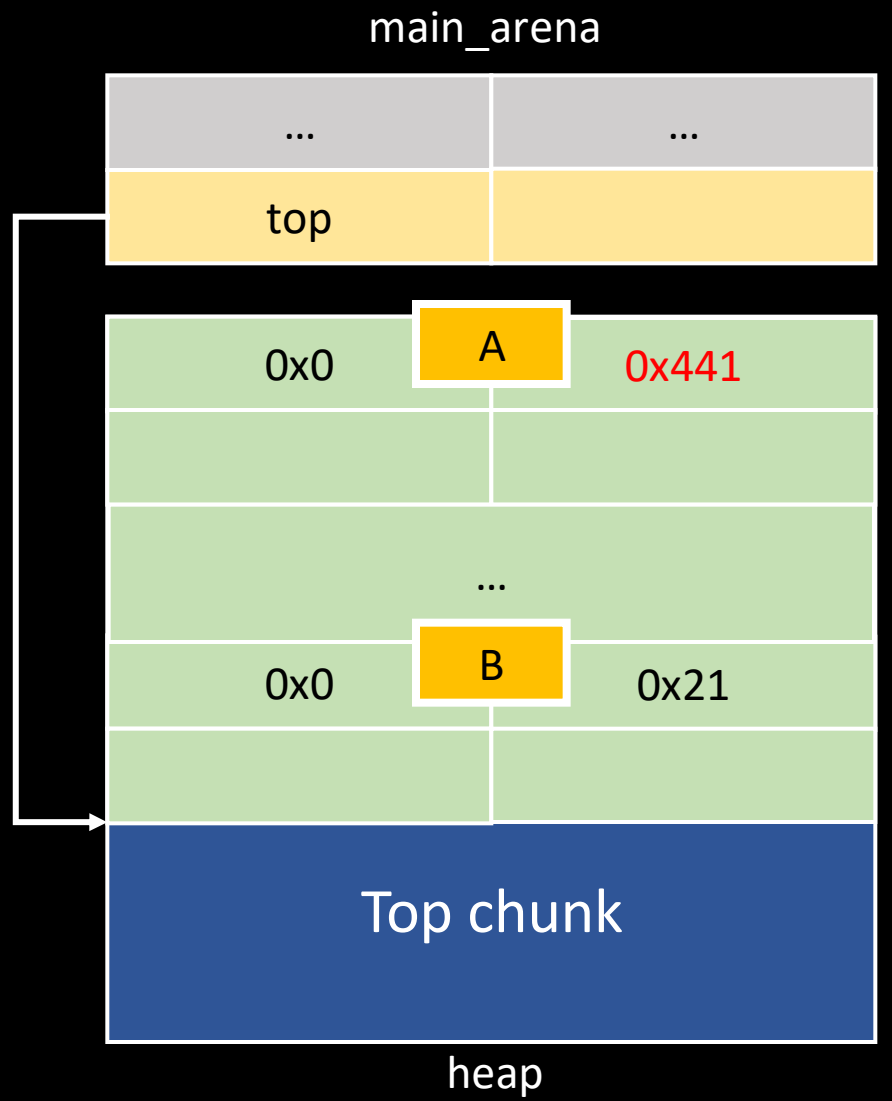
Overlapping chunks

```
A = malloc(0x410);
B = malloc(0x10);
*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk
A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```



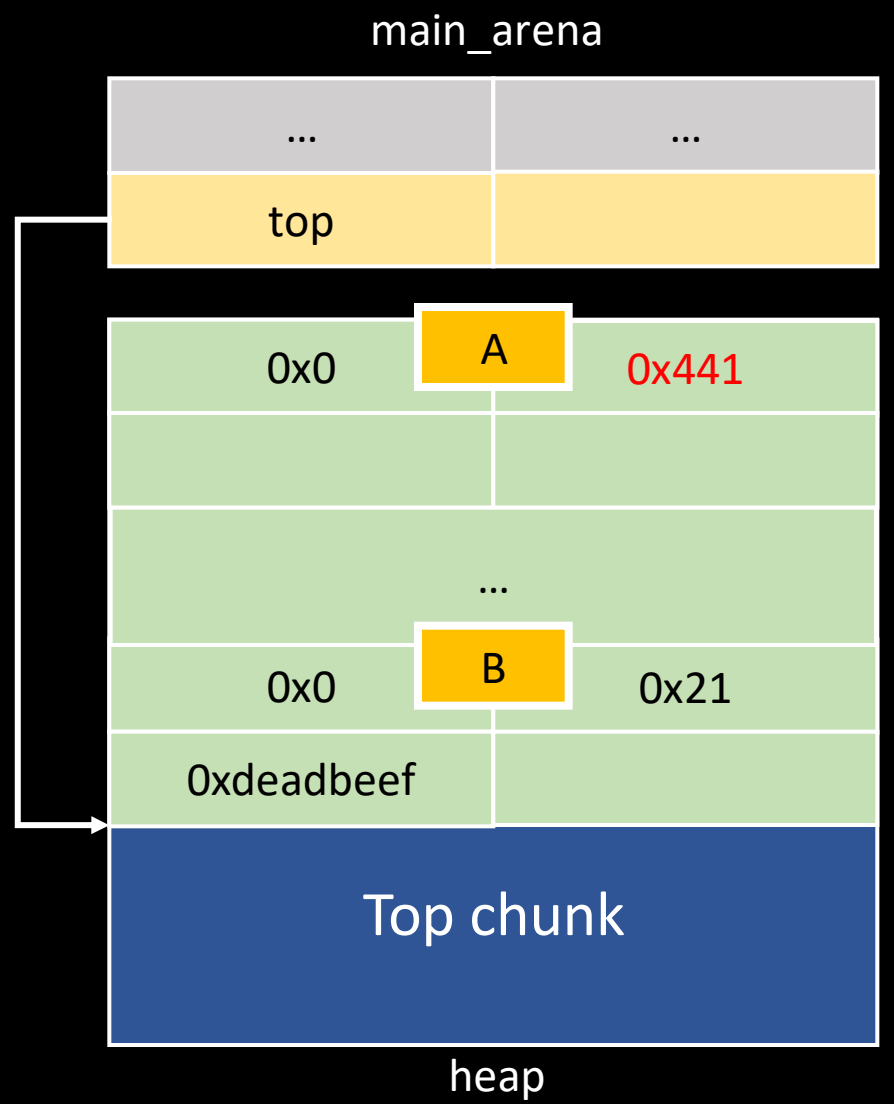
Overlapping chunks

```
A = malloc(0x410);
B = malloc(0x10);
*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk
A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```



Overlapping chunks

```
A = malloc(0x410);
B = malloc(0x10);
*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk
A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```



Demo Overlapping

Overlapping chunks

- 這周一樣會有一題作業
- 會盡量出簡單一點
- 可能是 UAF 或 Overlapping

End