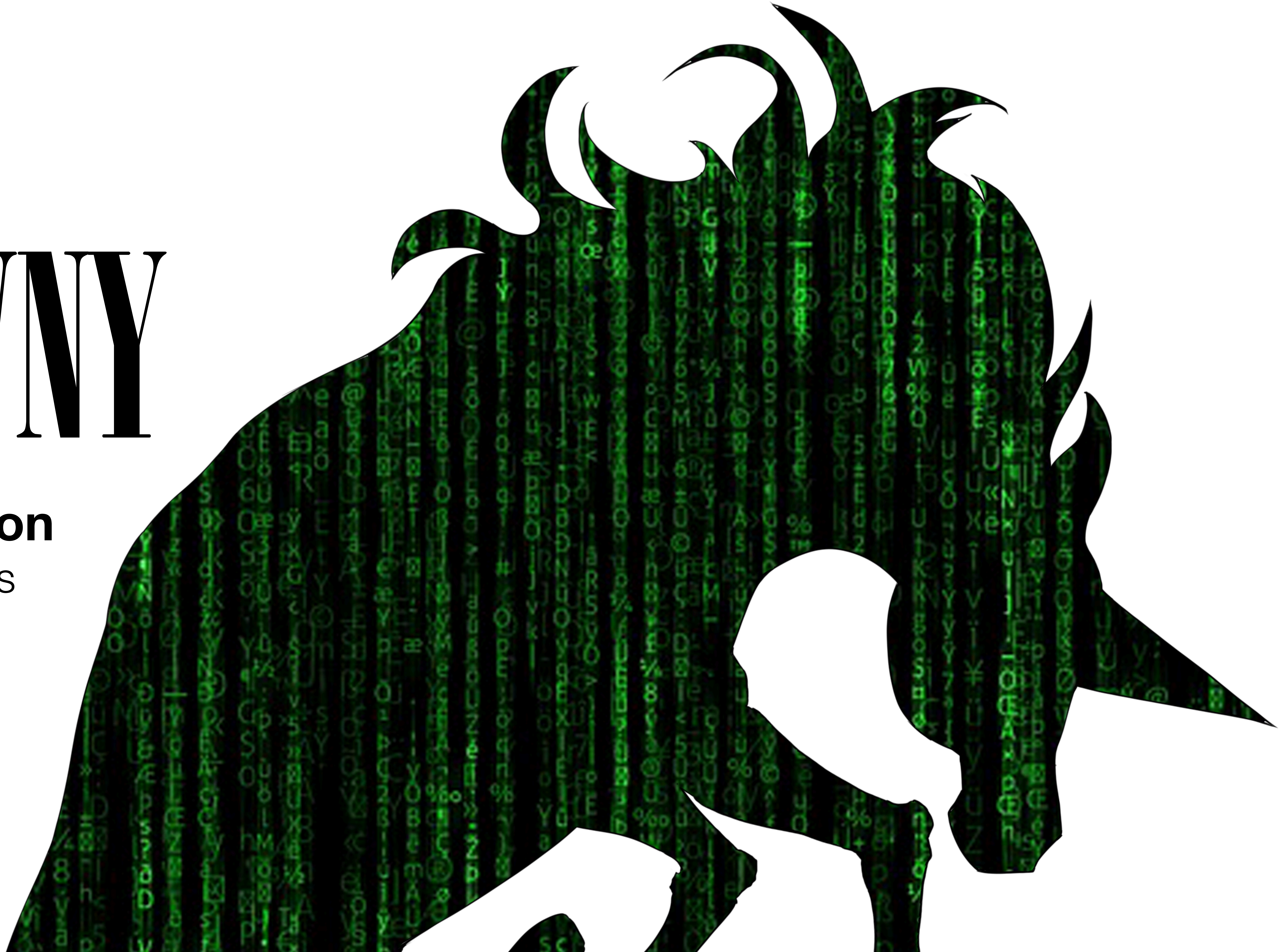
SIGPWNY

Heap Exploitation

Part 2- glibc Internals



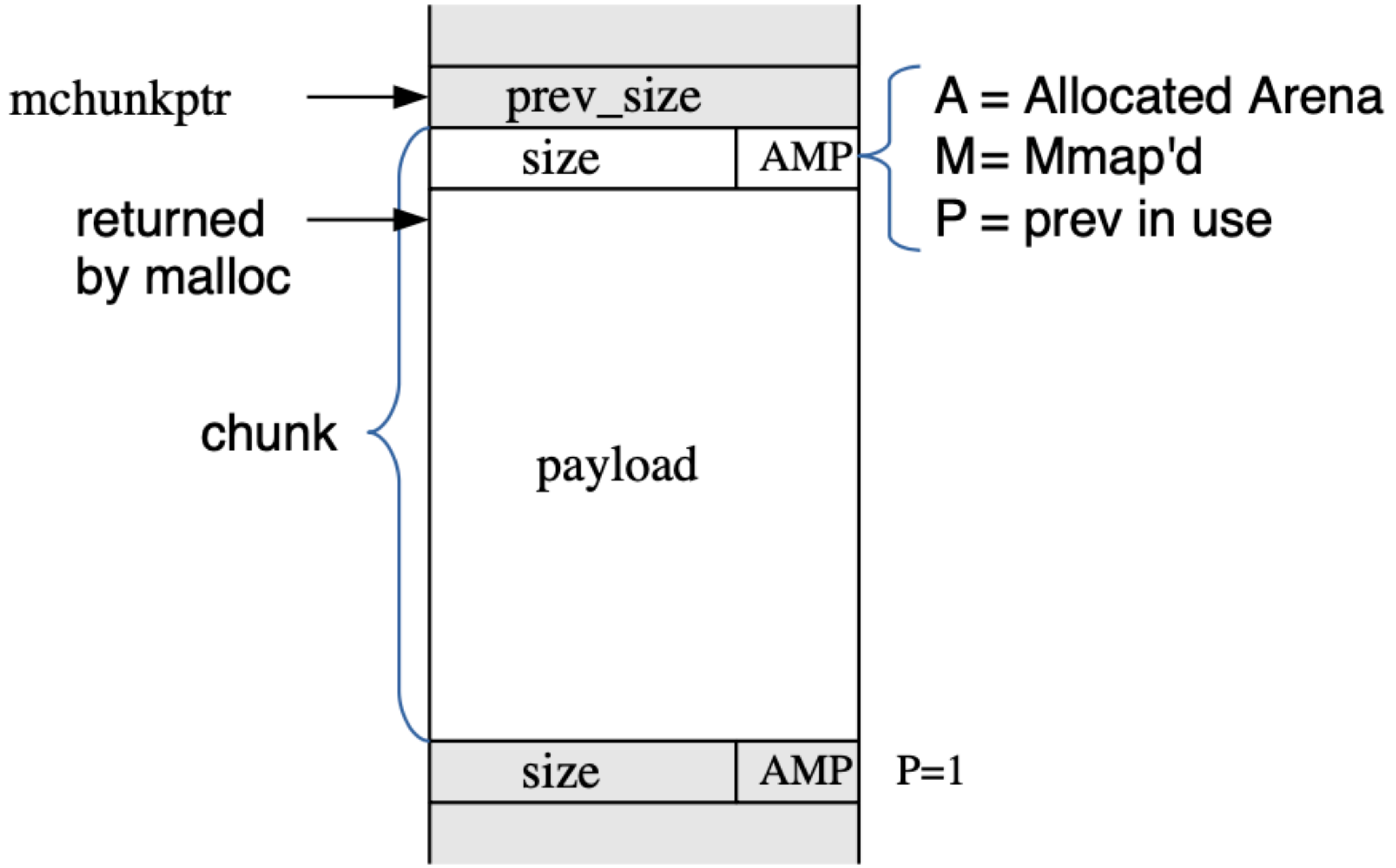
Let’s implement our own memory allocator.

What’s a chunk?

How do we know if a chunk is "in-use"?

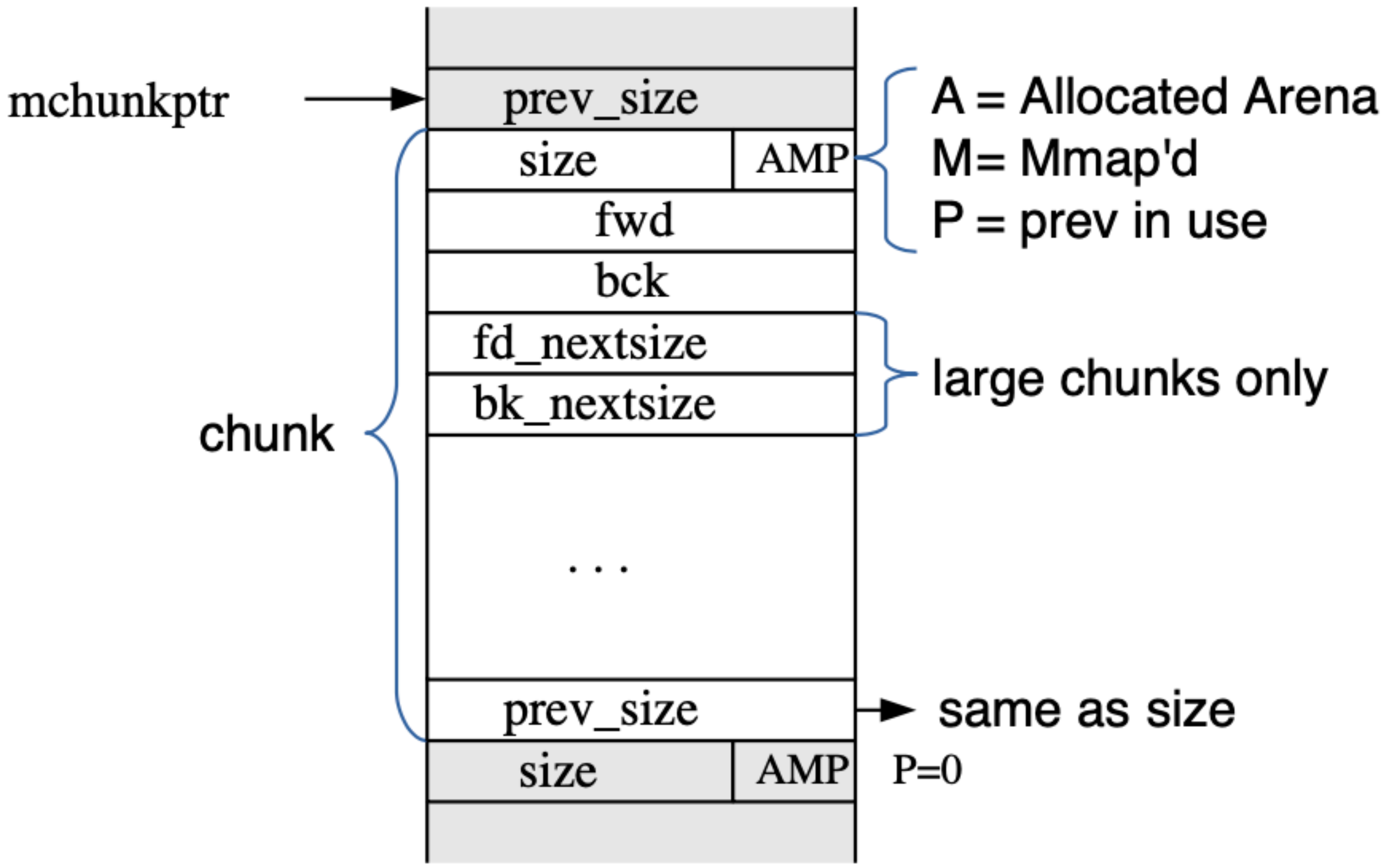
What happens when a chunk is `free()`'d?

Glibc In-Use Chunk



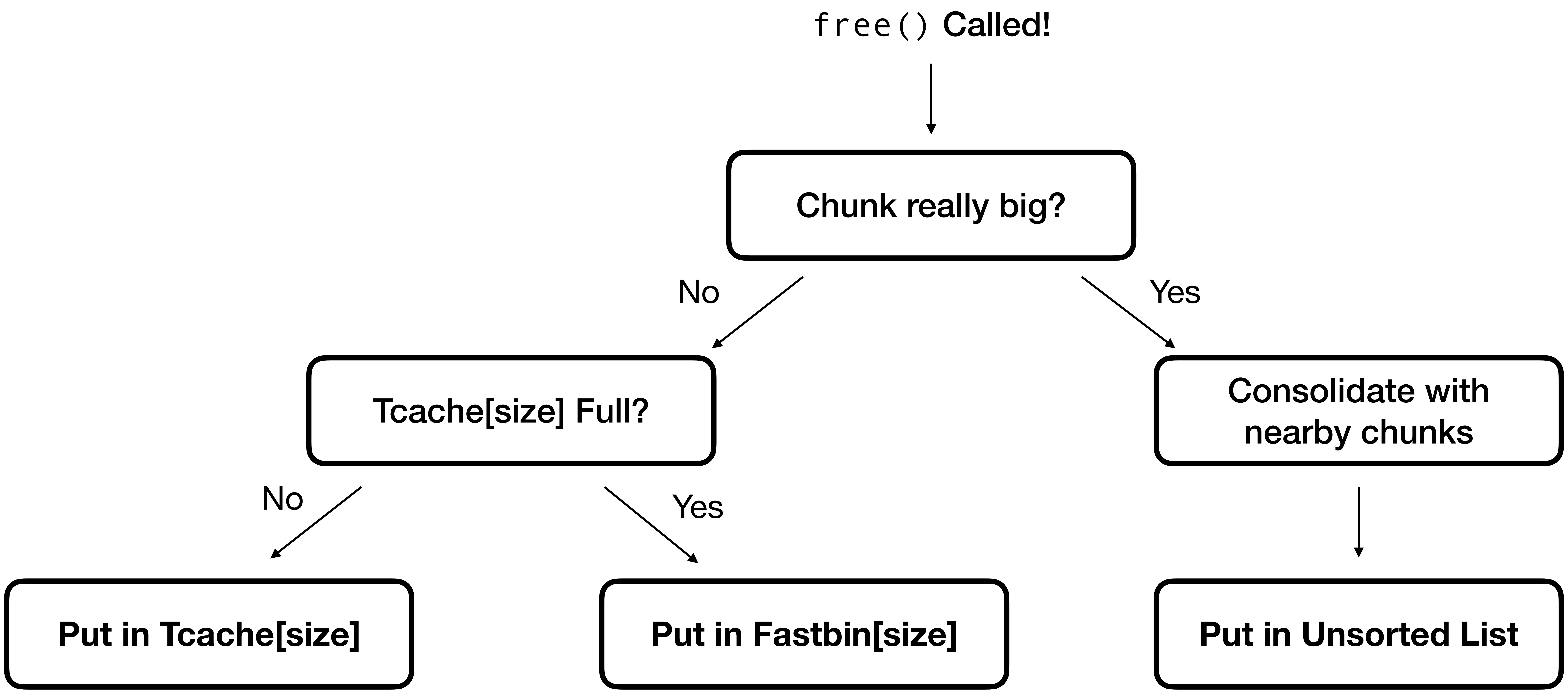
https://sourceware.org/glibc/wiki/MallocInternals

Glibc Free Chunk



https://sourceware.org/glibc/wiki/MallocInternals

Once Upon a `free()`...


free() Called!
Chunk really big?
No
Yes
Tcache[size] Full?
Consolidate with nearby chunks
No
Yes
Put in Tcache[size]
Put in Fastbin[size]
Put in Unsorted List

Glibc Bins

Fast Bins

For the smallest of chunks; never consolidated with nearby chunks; singly linked list.

Chunks are placed here immediately on `free()`.

Small Bins

Larger allocations than fastbins, consolidated with nearby chunks on free, doubly linked list.

Chunks of this size are placed in the unsorted list on `free()` & are placed in this list during further heap traversal.

Large Bins

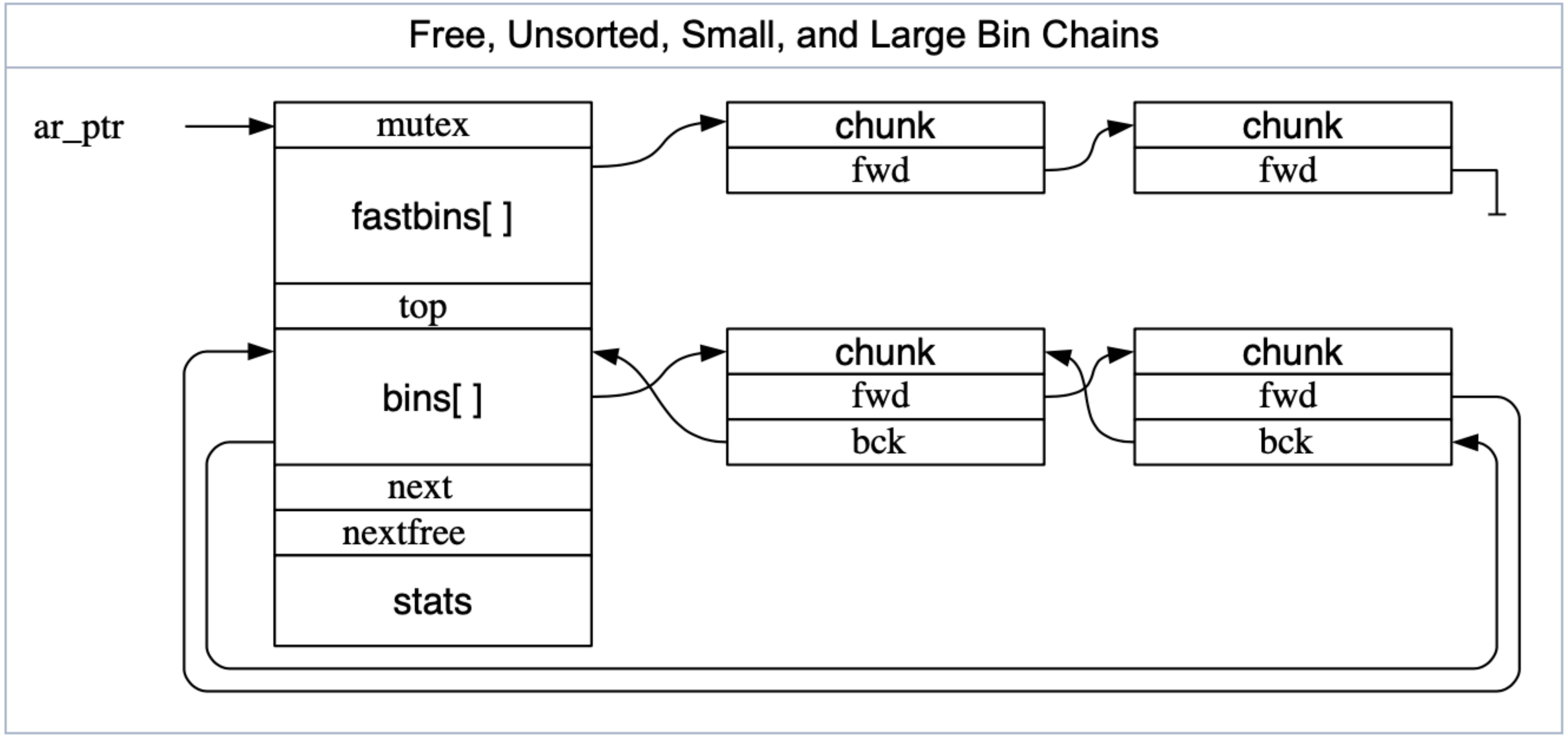
Largest allocations go here, consolidated with nearby chunks on free, size stored in chain as well.

Chunks of this size are placed in the unsorted list on `free()` & are placed in this list during further heap traversal.

Smaller → Bigger

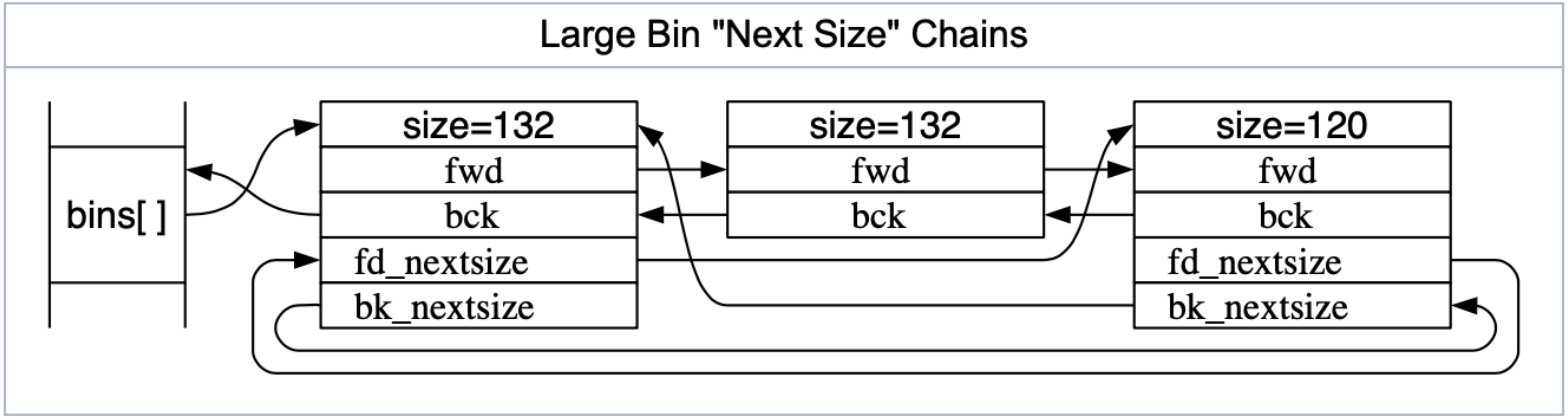
https://sourceware.org/glibc/wiki/MallocInternals

Glibc Bins



https://sourceware.org/glibc/wiki/MallocInternals

Glibc Bins



https://sourceware.org/glibc/wiki/MallocInternals

Tcache

Heap is shared by threads

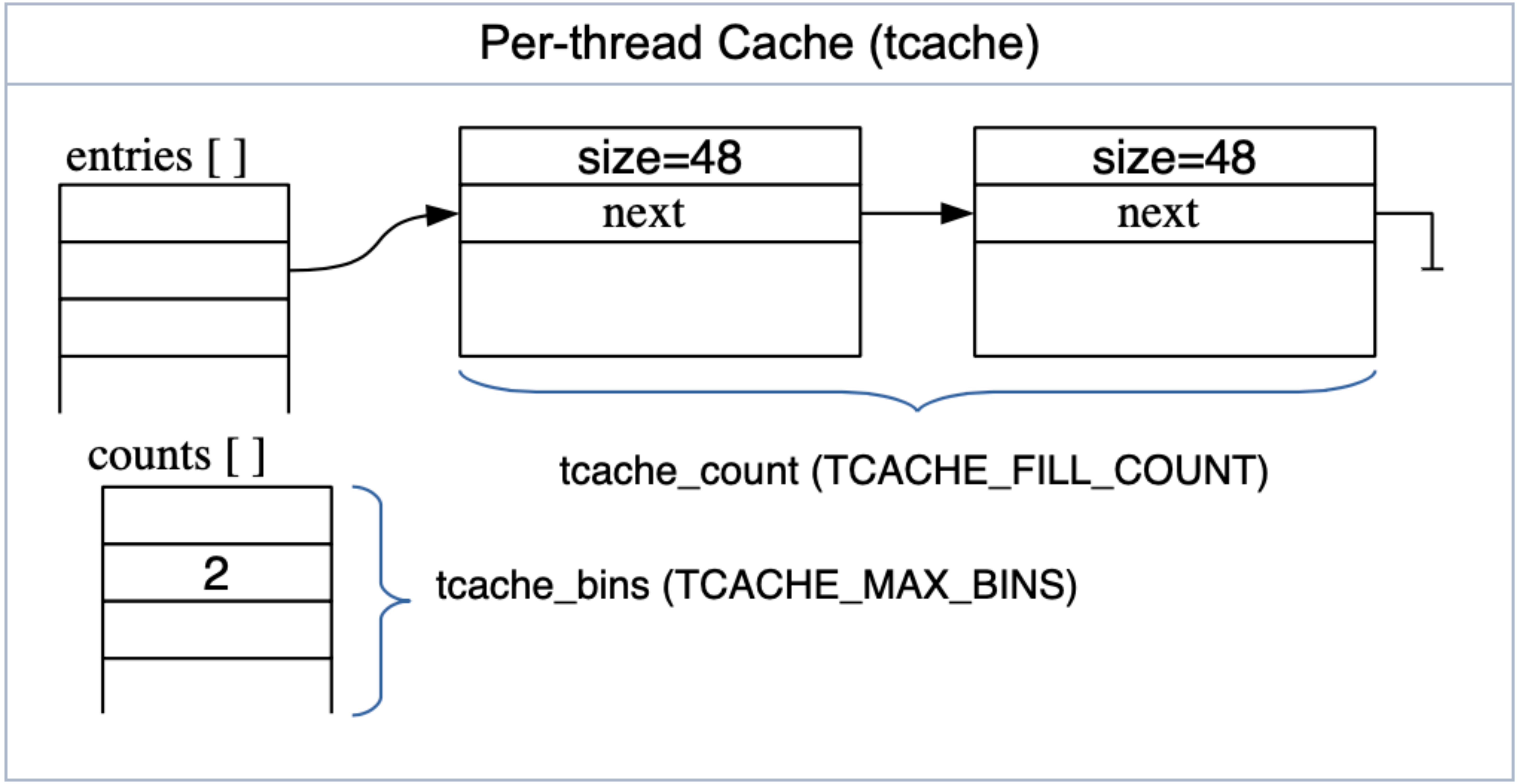
Want to use thread-local storage (TLS) to speed it up

Tcache = “per-thread fastbin”

Tcache chunks by definition <u>cannot</u> look at nearby chunks!

https://sourceware.org/glibc/wiki/MallocInternals

Tcache



https://sourceware.org/glibc/wiki/MallocInternals

Heap Layout in Memory

```
Logged in as sigpwny
sigpwny > create_user USER A
Created user 2
sigpwny > create_user USER B
Created user 3
sigpwny > create_doc
Created document 0
sigpwny > write_doc 0 DOCUMENT 0
[document 0] writing DOCUMENT 0
sigpwny > create_user USER C
Created user 4
sigpwny > 
[0] 0:heap4*
```

```
gef➤  heap chunks
Chunk(addr=0x55555555b010, size=0x250, flags=PREV_INUSE)
    [0x000055555555b010     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x55555555b260, size=0x50, flags=PREV_INUSE)
    [0x000055555555b260     61 64 6d 69 6e 00 00 00 00 00 00 00 00 00 00 00    admin...........]
Chunk(addr=0x55555555b2b0, size=0x50, flags=PREV_INUSE)
    [0x000055555555b2b0     73 69 67 70 77 6e 79 00 00 00 00 00 00 00 00 00    sigpwny.........]
Chunk(addr=0x55555555b300, size=0x50, flags=PREV_INUSE)
    [0x000055555555b300     55 53 45 52 20 41 00 00 00 00 00 00 00 00 00 00    USER A..........]
Chunk(addr=0x55555555b350, size=0x50, flags=PREV_INUSE)
    [0x000055555555b350     55 53 45 52 20 42 00 00 00 00 00 00 00 00 00 00    USER B..........]
Chunk(addr=0x55555555b3a0, size=0x60, flags=PREV_INUSE)
    [0x000055555555b3a0     44 4f 43 55 4d 45 4e 54 20 30 00 00 00 00 00 00    DOCUMENT 0......]
Chunk(addr=0x55555555b400, size=0x50, flags=PREV_INUSE)
    [0x000055555555b400     55 53 45 52 20 43 00 00 00 00 00 00 00 00 00 00    USER C..........]
Chunk(addr=0x55555555b450, size=0x20bc0, flags=PREV_INUSE)  ←  top chunk
gef➤
[0] 0:gdb*                                                                                "d
```

```
sigpwny > del_user 2
Deleted user 2 (named USER A)

sigpwny > del_user 3
Deleted user 3 (named USER B)

sigpwny > 
[0] 0:heap4*
```

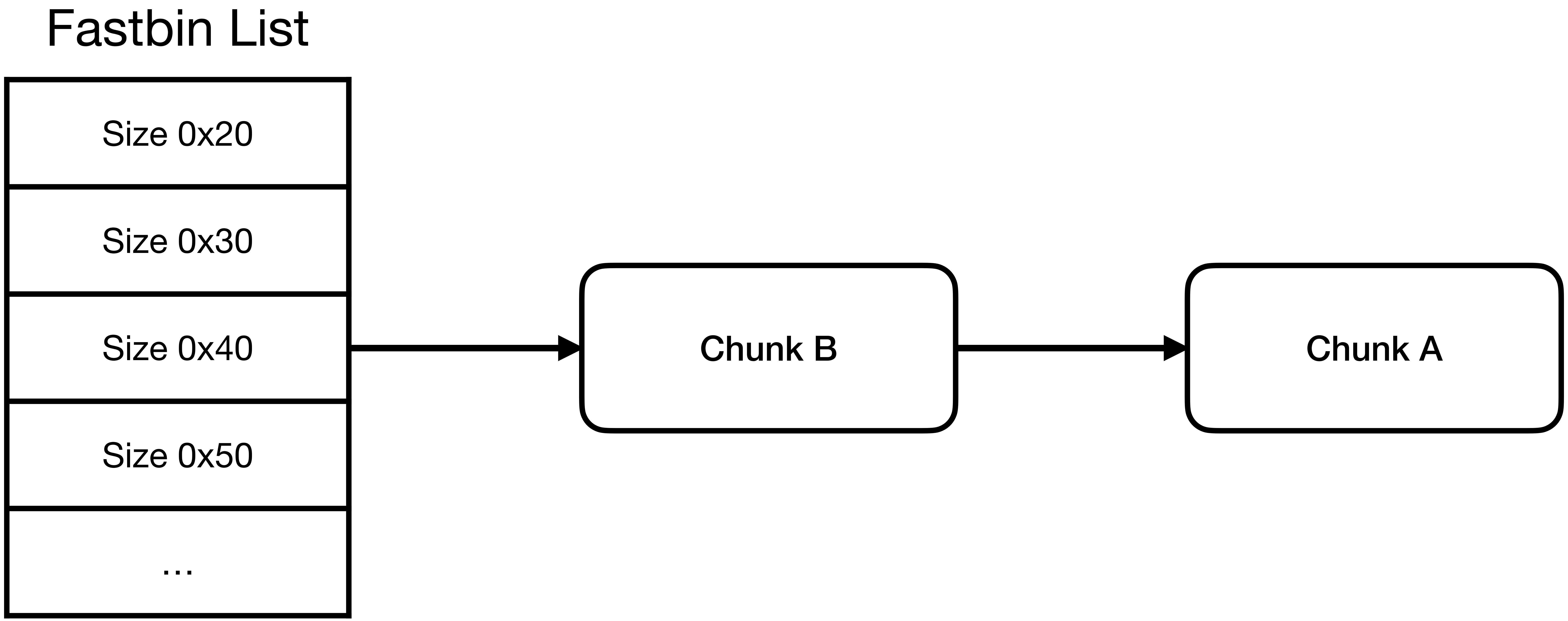
```
gef➤  heap bins
──────────────────────────── Tcachebins for arena 0x7ffff7dcdc40 ────────────────────────────
Tcachebins[idx=3, size=0x50] count=2  ←  Chunk(addr=0x55555555b350, size=0x50, flags=PREV_INUSE)  ←  Chunk(addr=0x55555555b300
, size=0x50, flags=PREV_INUSE)
───────────────────────────── Fastbins for arena 0x7ffff7dcdc40 ─────────────────────────────
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 'main_arena' ────────────────────────────
[+] Found 0 chunks in unsorted bin.
───────────────────────────── Small Bins for arena 'main_arena' ─────────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
───────────────────────────── Large Bins for arena 'main_arena' ─────────────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤  
[0] 0:gdb*                                                          "docker-desktop" 22:43 18-Feb-21
```

Let’s Take Another Look at Double Free

Heap 3

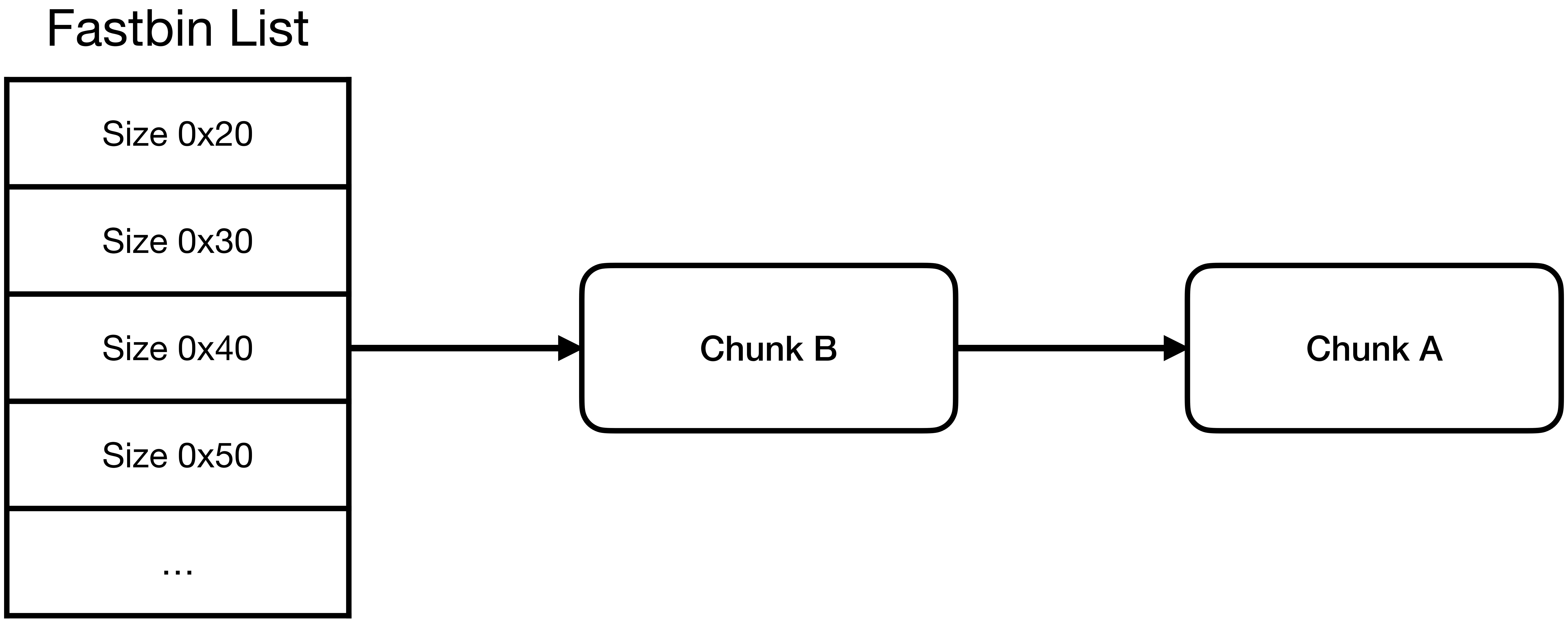
The following applies to fastbins only, not tcaches.

Fastbins


Fastbin List
Size 0x20
Size 0x30
Size 0x40
Size 0x50
...
Chunk B
Chunk A

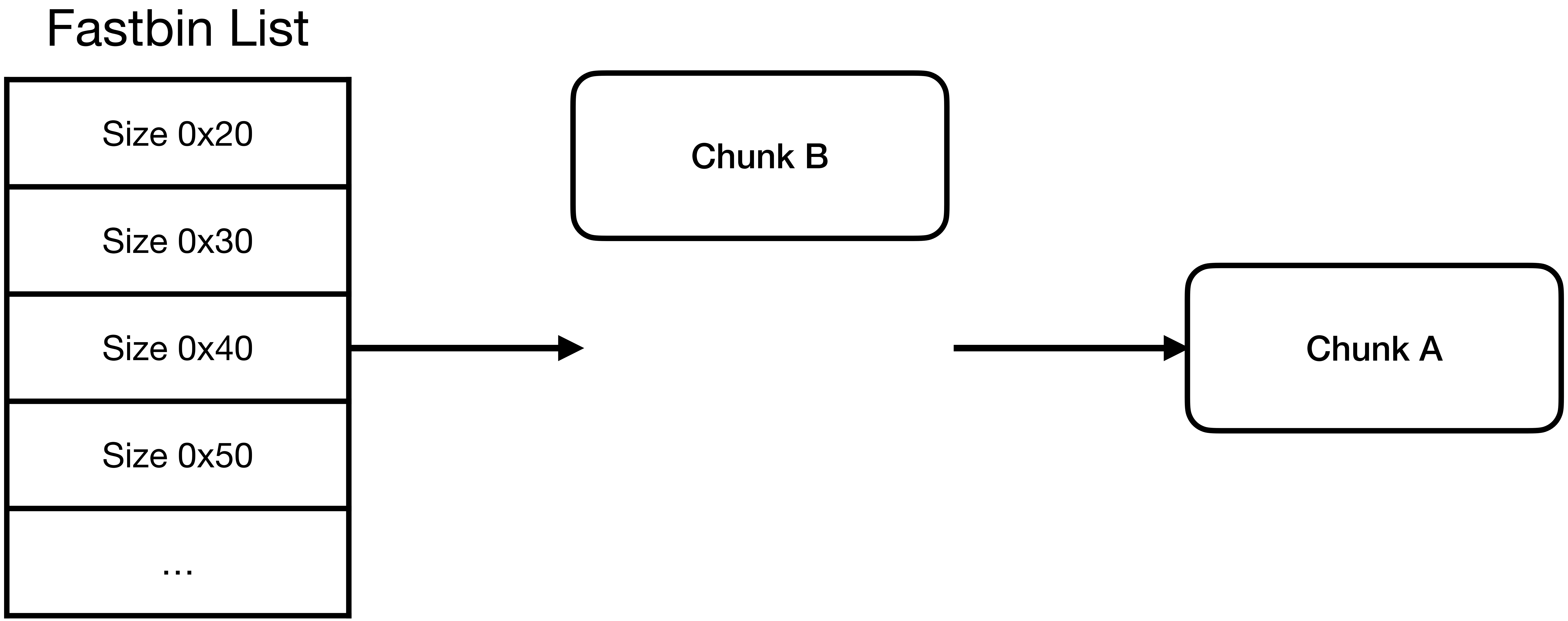
Fastbins

malloc wants
a 0x40 chunk

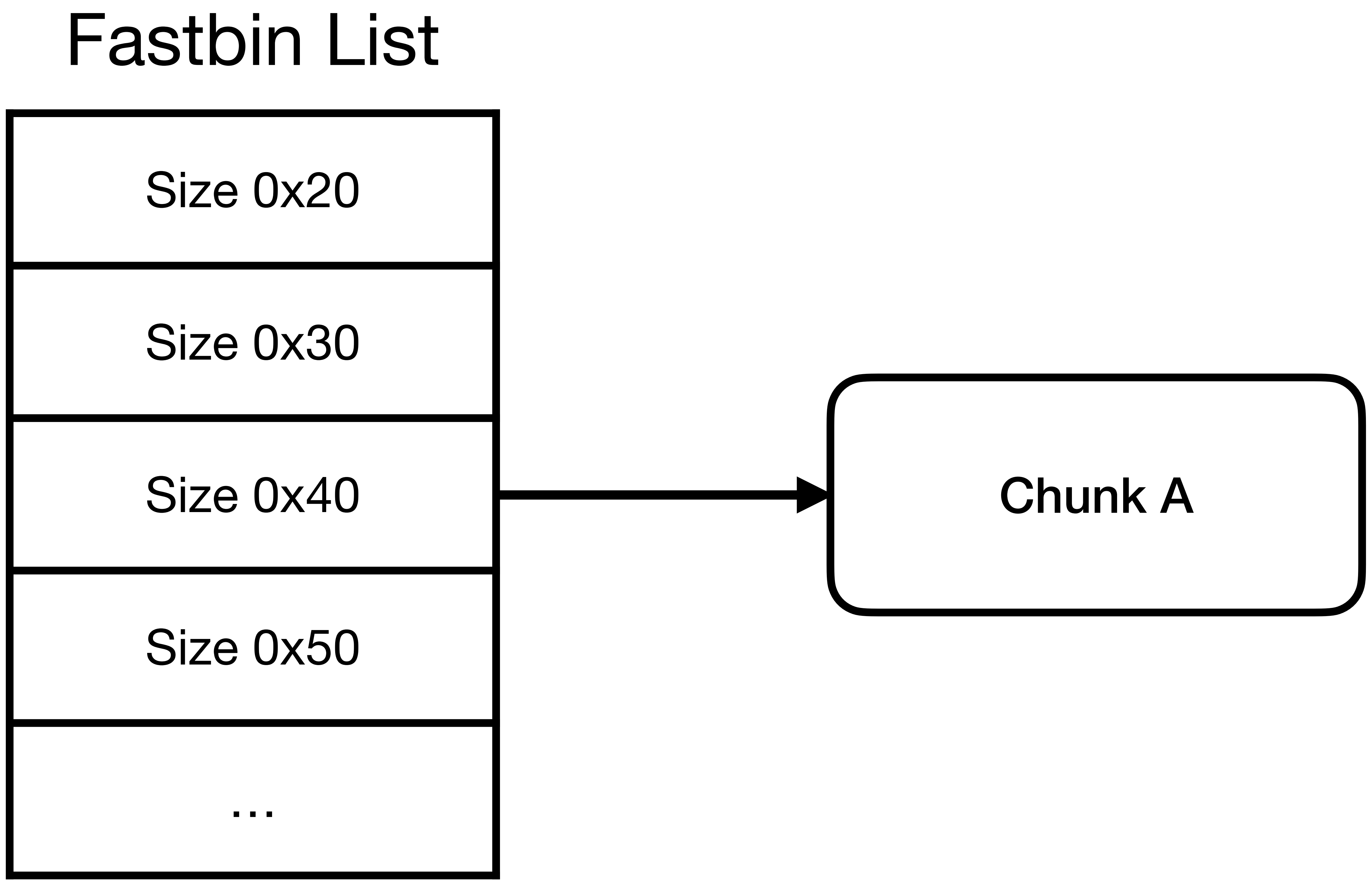


Fastbins

malloc wants
a 0x40 chunk

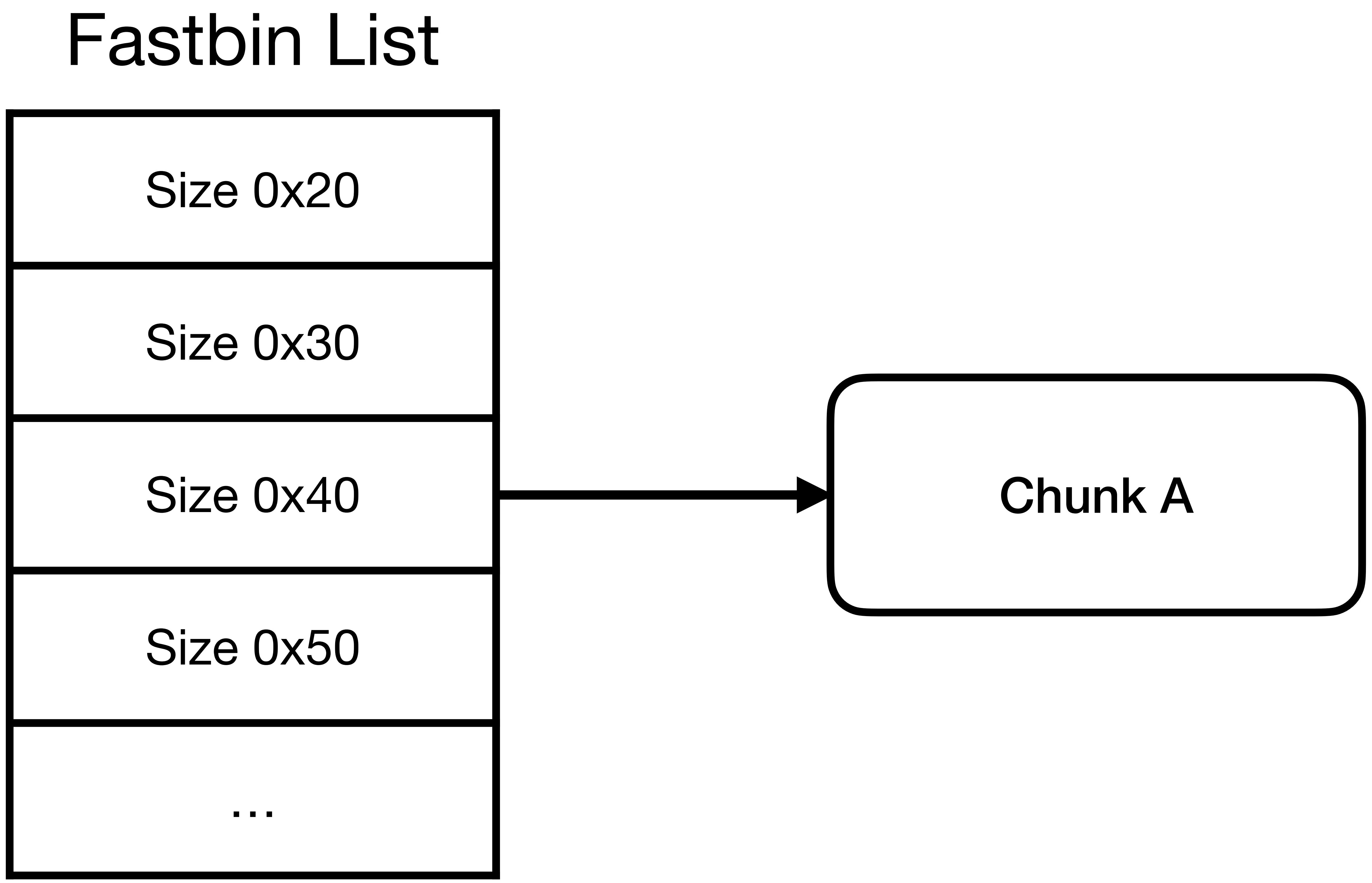


Fastbins



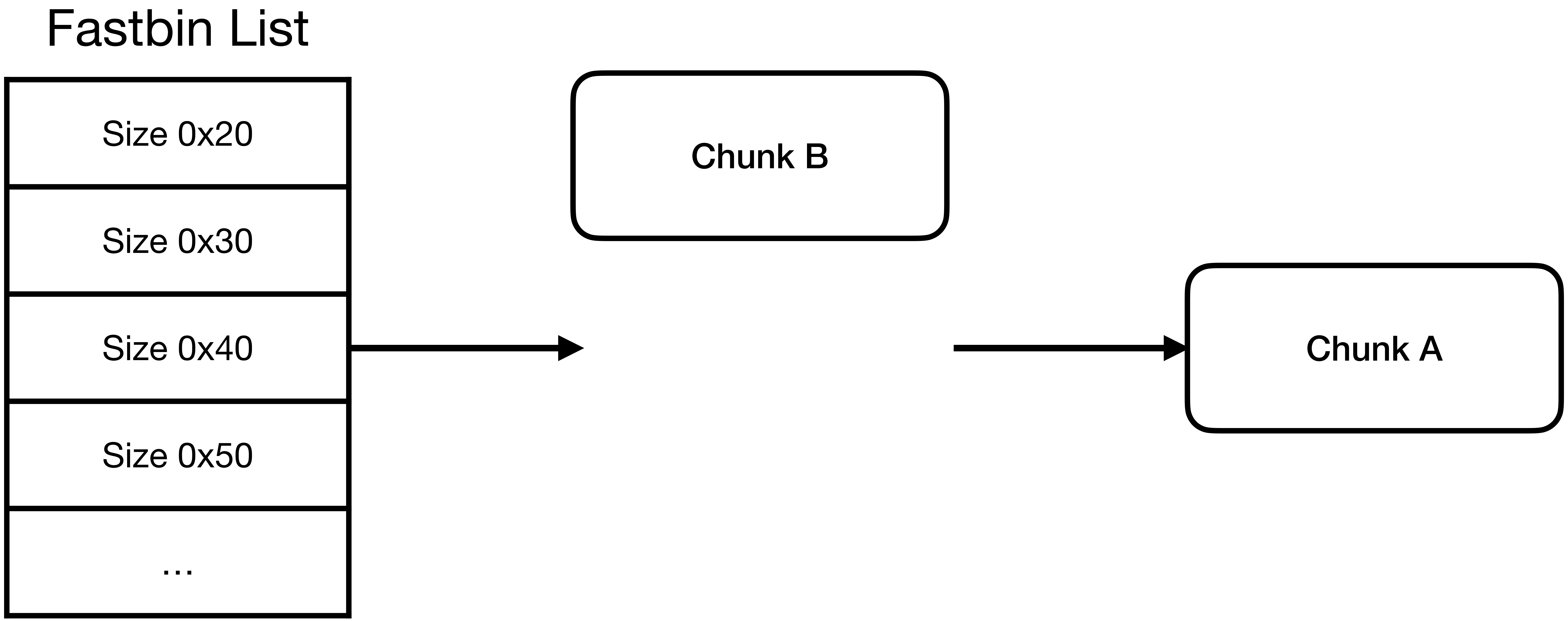
Fastbins

freed
Chunk B



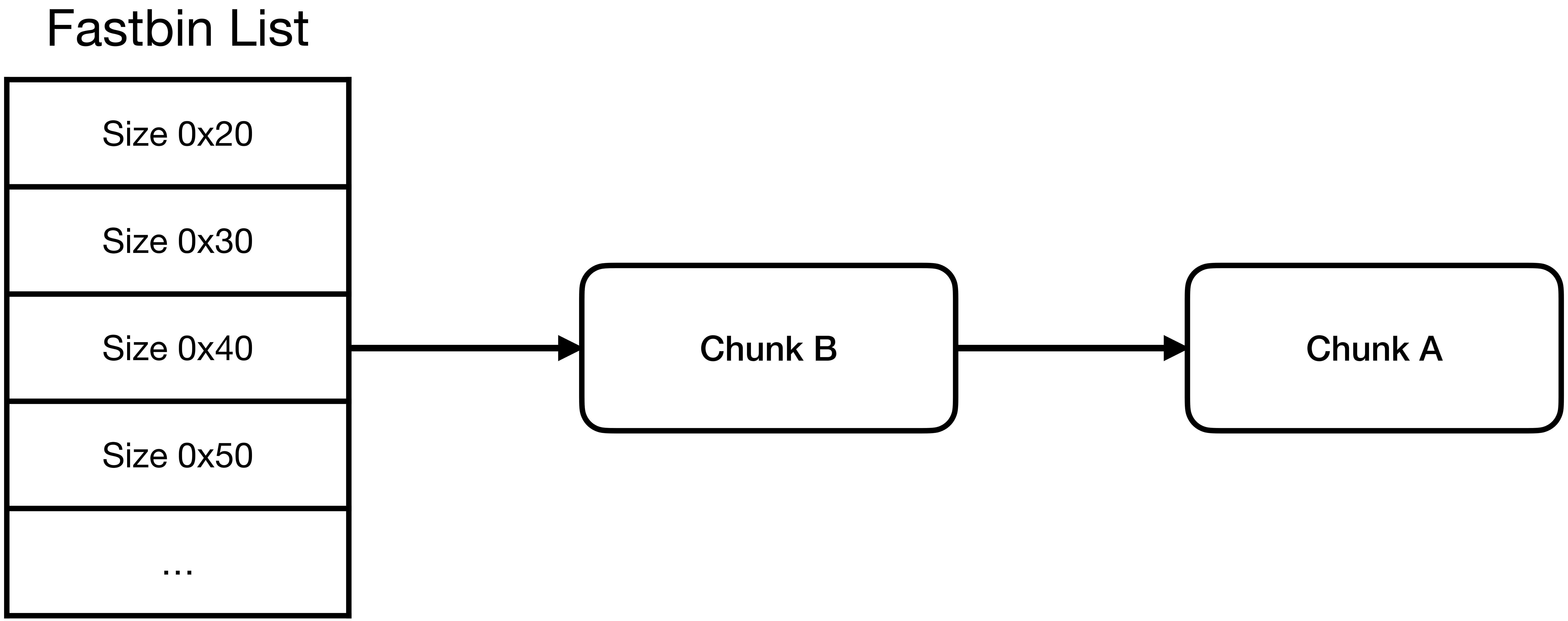
Fastbins

freed
Chunk B

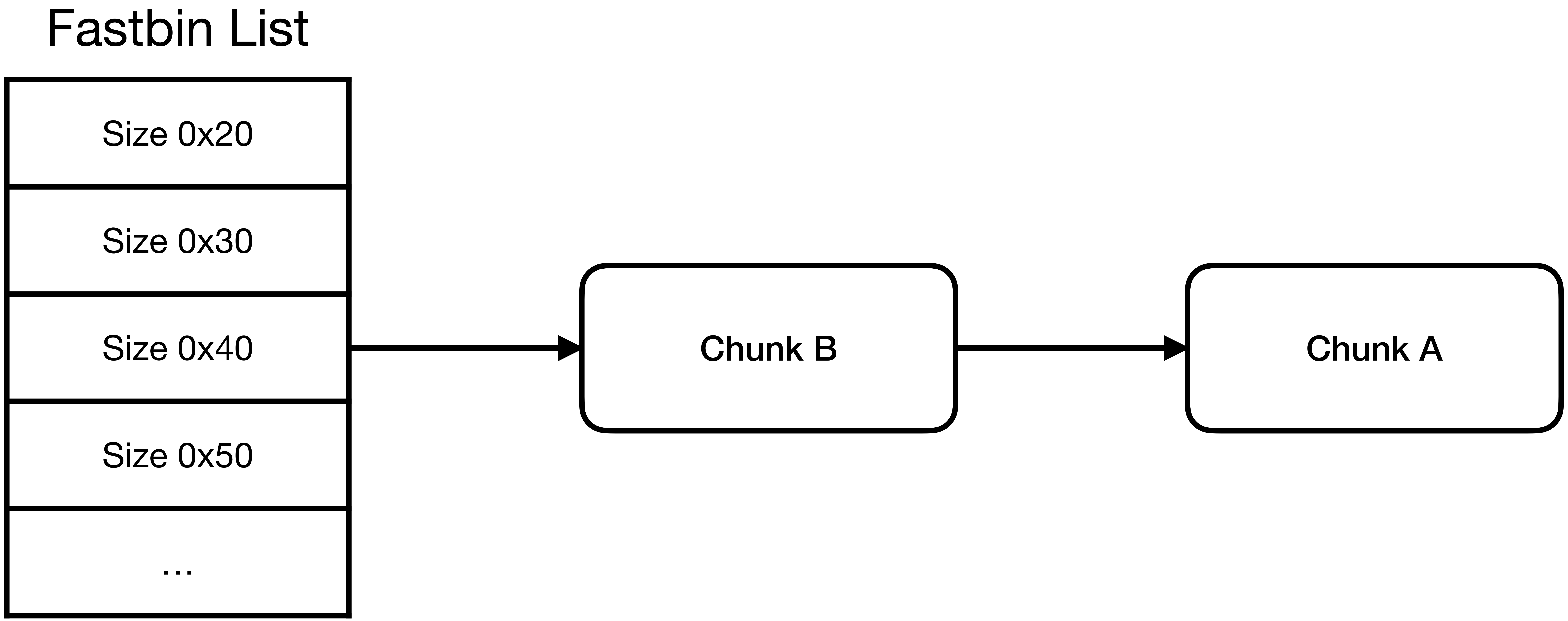


Fastbins

freed
Chunk B



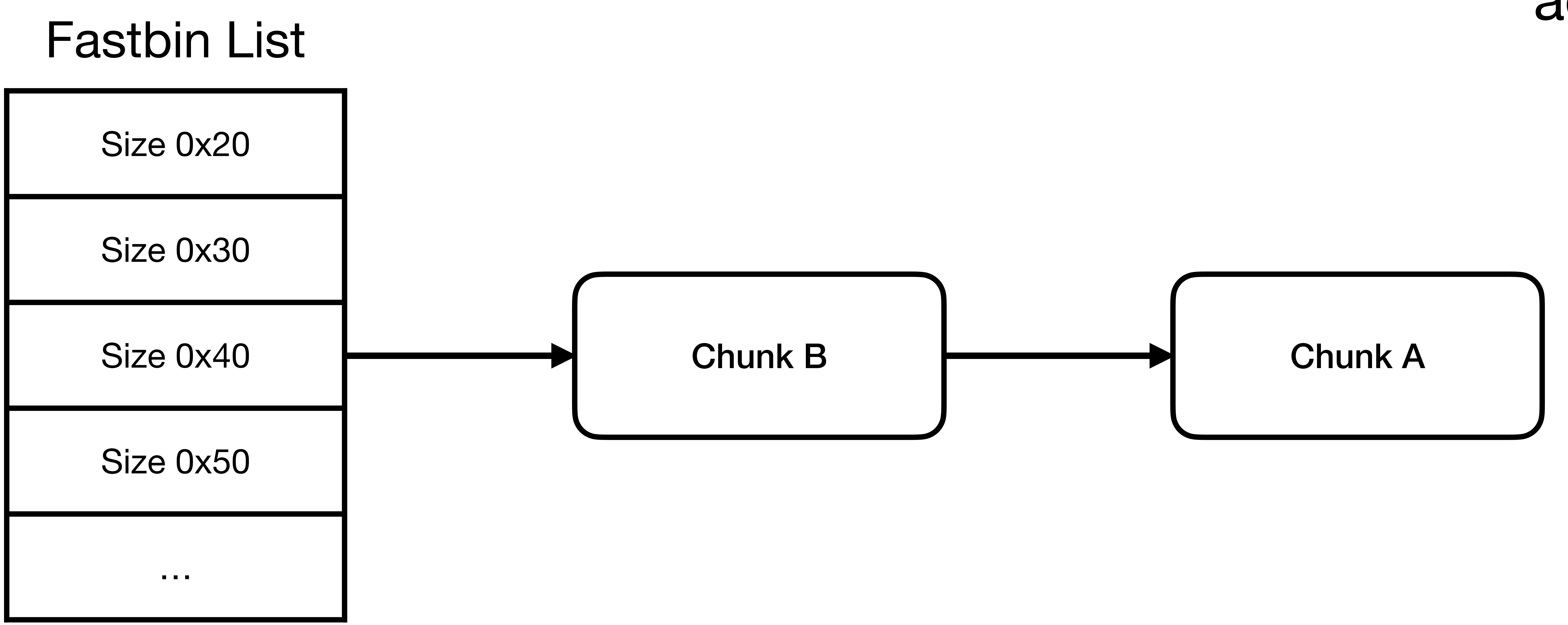
Fastbins



Fastbin List
Size 0x20
Size 0x30
Size 0x40
Size 0x50
...
Chunk B
Chunk A

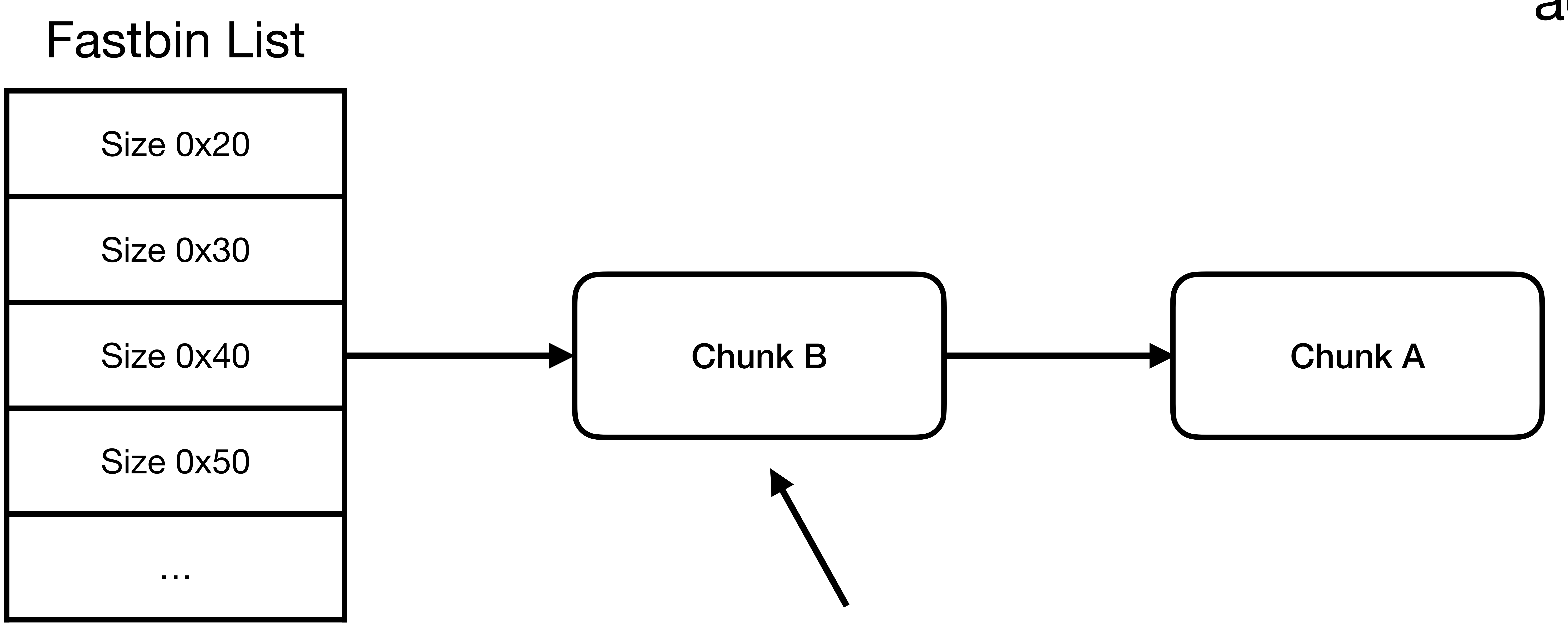
Fastbins

Can we free Chunk B again?



Fastbins

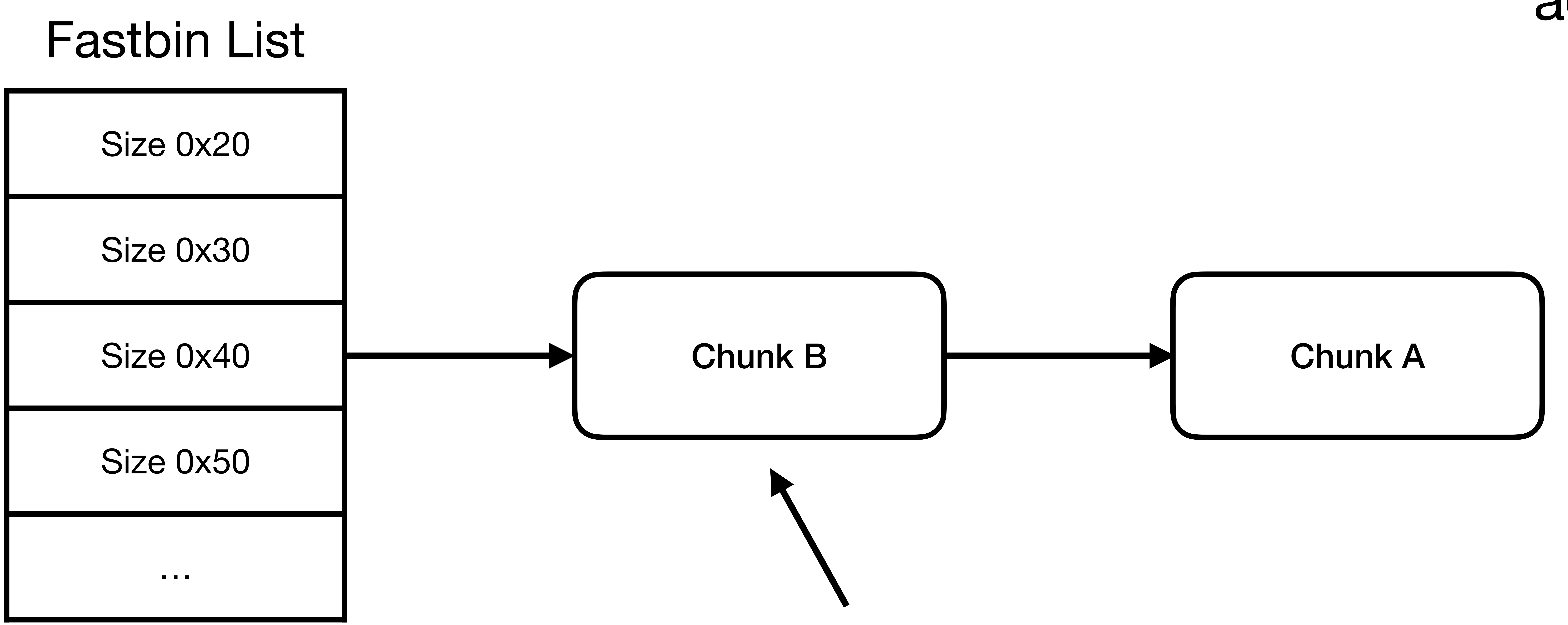
Can we free Chunk B again?



free checks if the first element is equal to what is about to be freed!

Fastbins

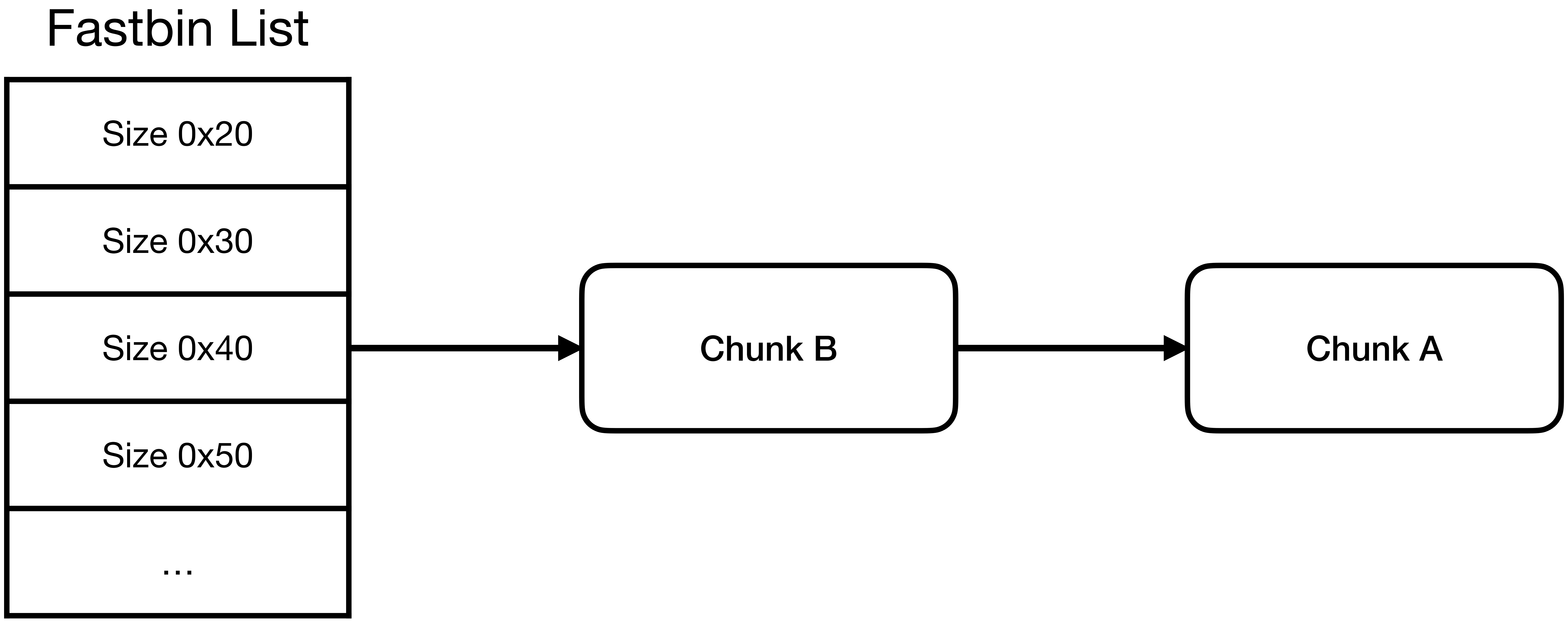
Can we free Chunk B again?



Attempting to free Chunk B here will cause libc to detect a double free, and crash the program.

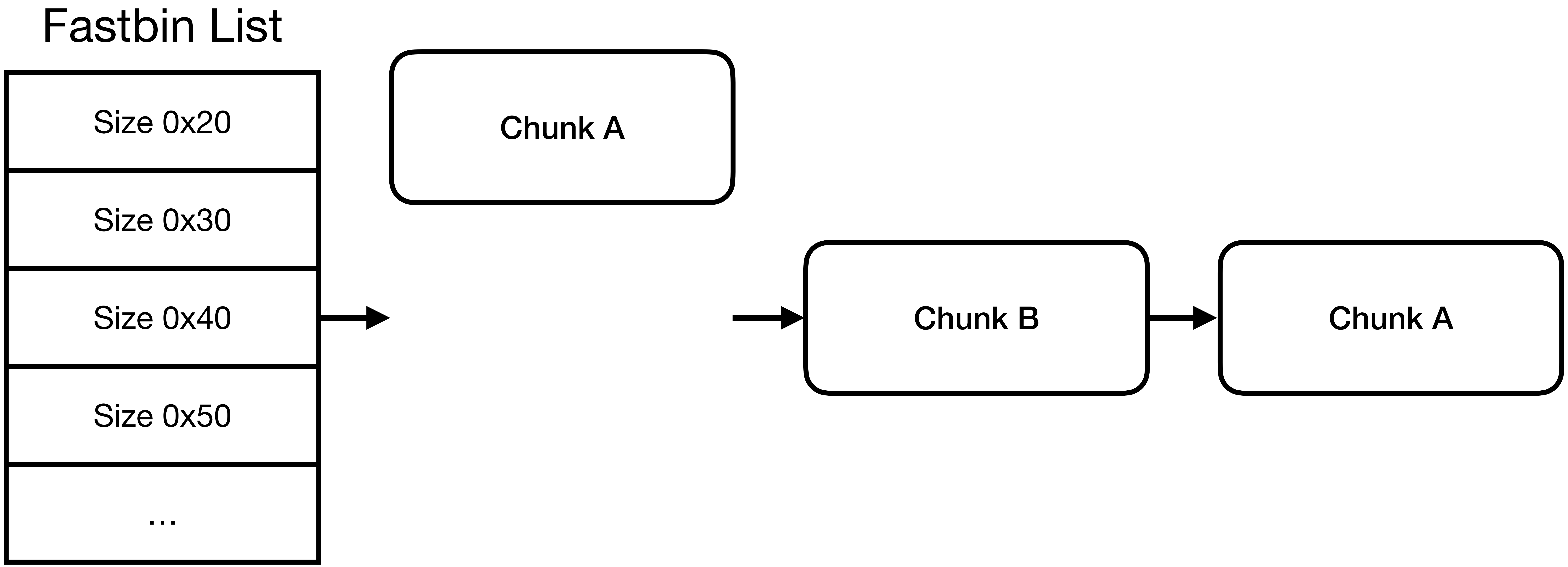
Fastbins

Free Chunk A again!



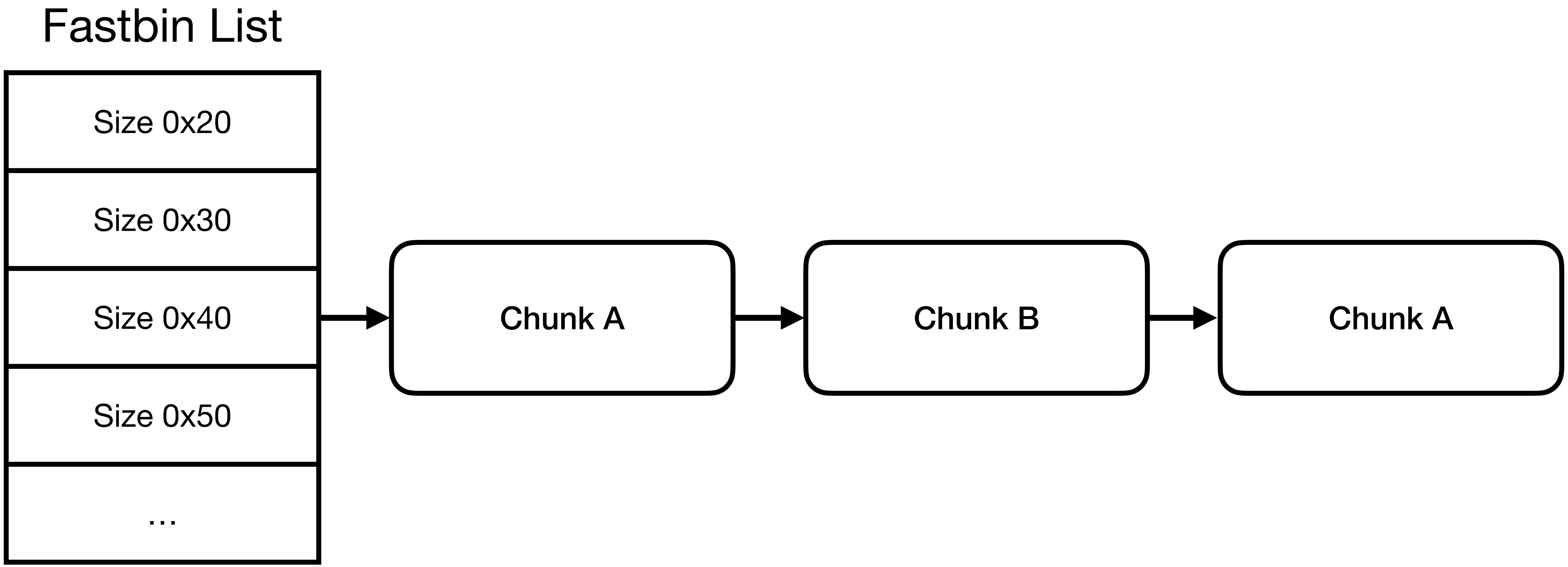
Fastbins

Free Chunk A again!

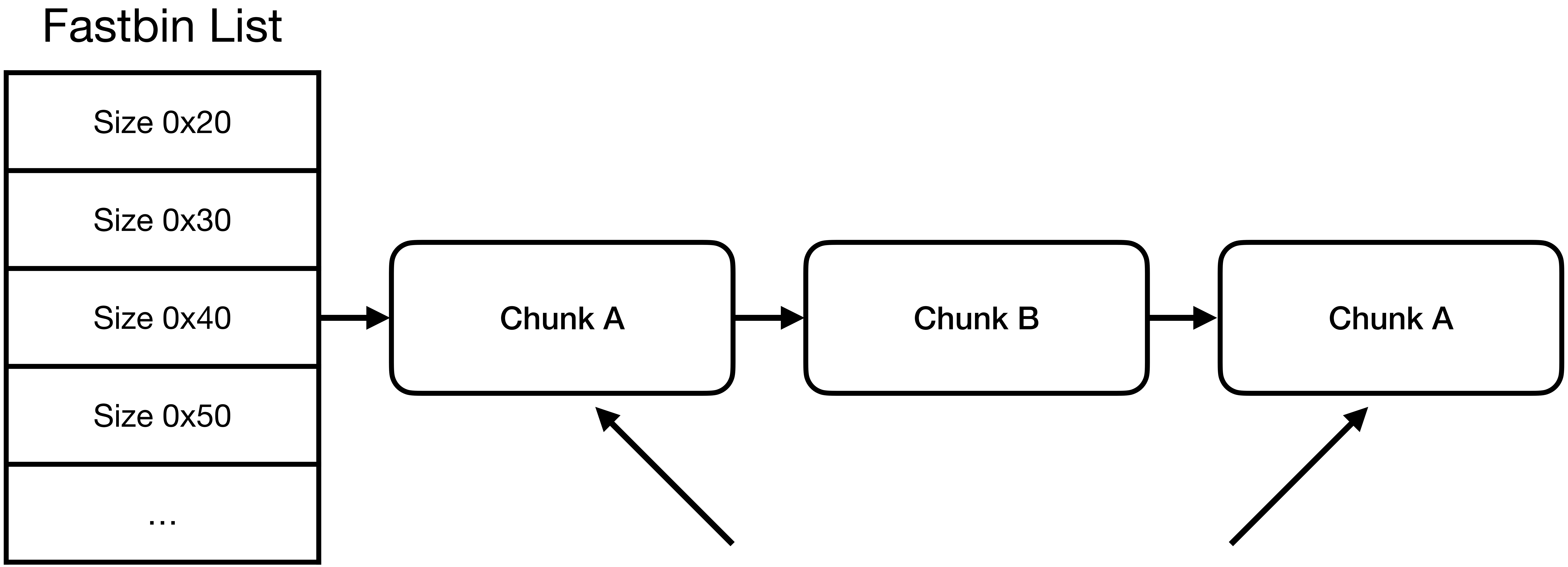


Fastbins

Free Chunk A again!

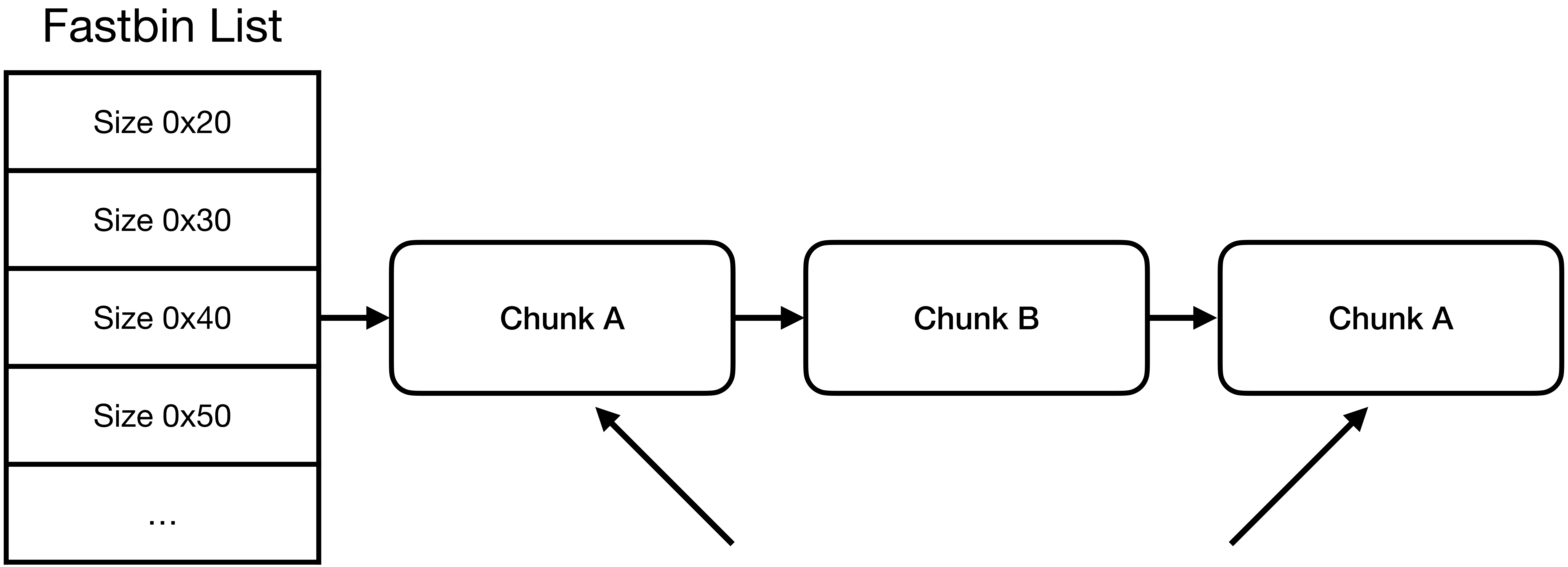


Fastbins



Chunk A will now be returned at the 1st and 3rd malloc calls

Fastbins



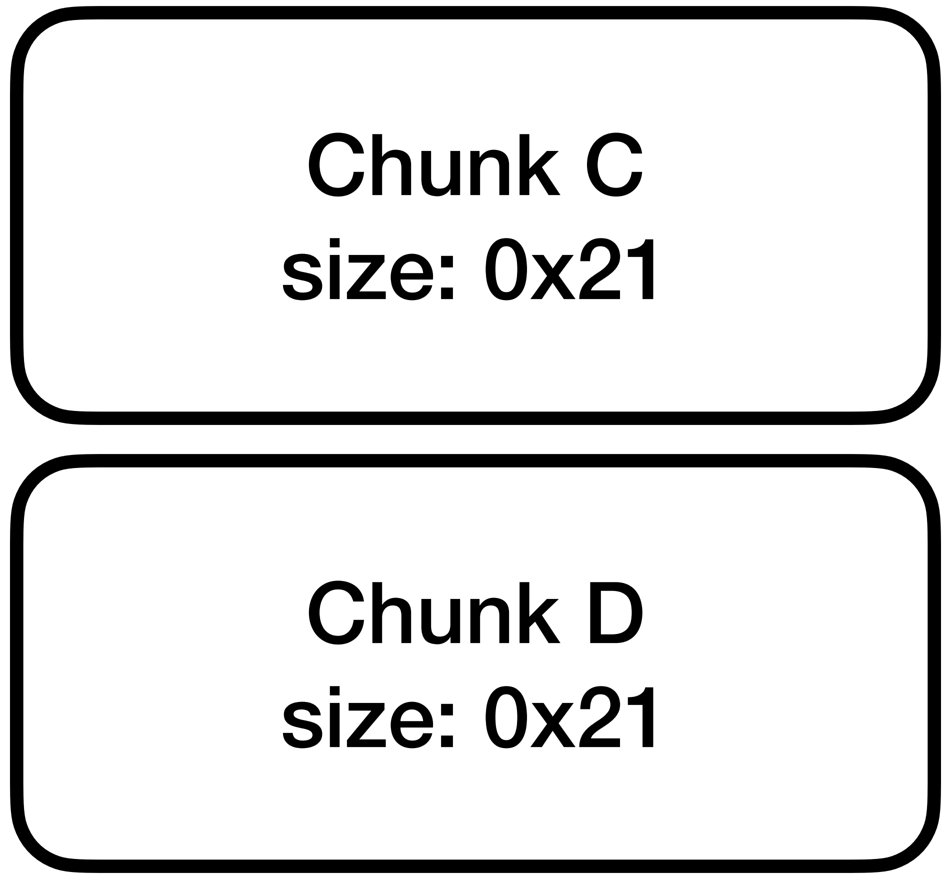
libc only checks the first thing in the list- as long as you don't free the same chunk twice, you're good.

Size Corruption to Create Overlapping Chunks

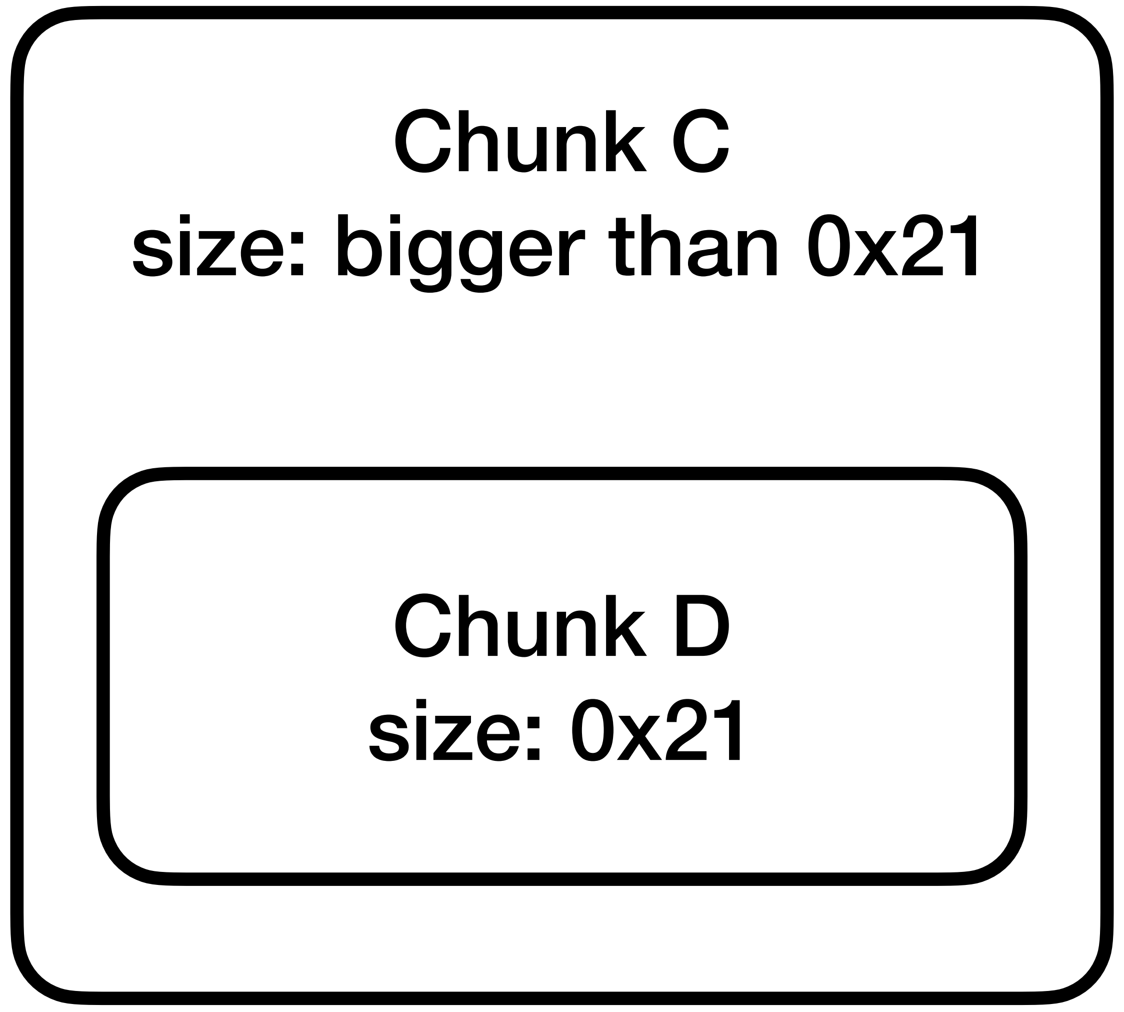
Heap 4

The following applies to tcaches only, not fastbins.

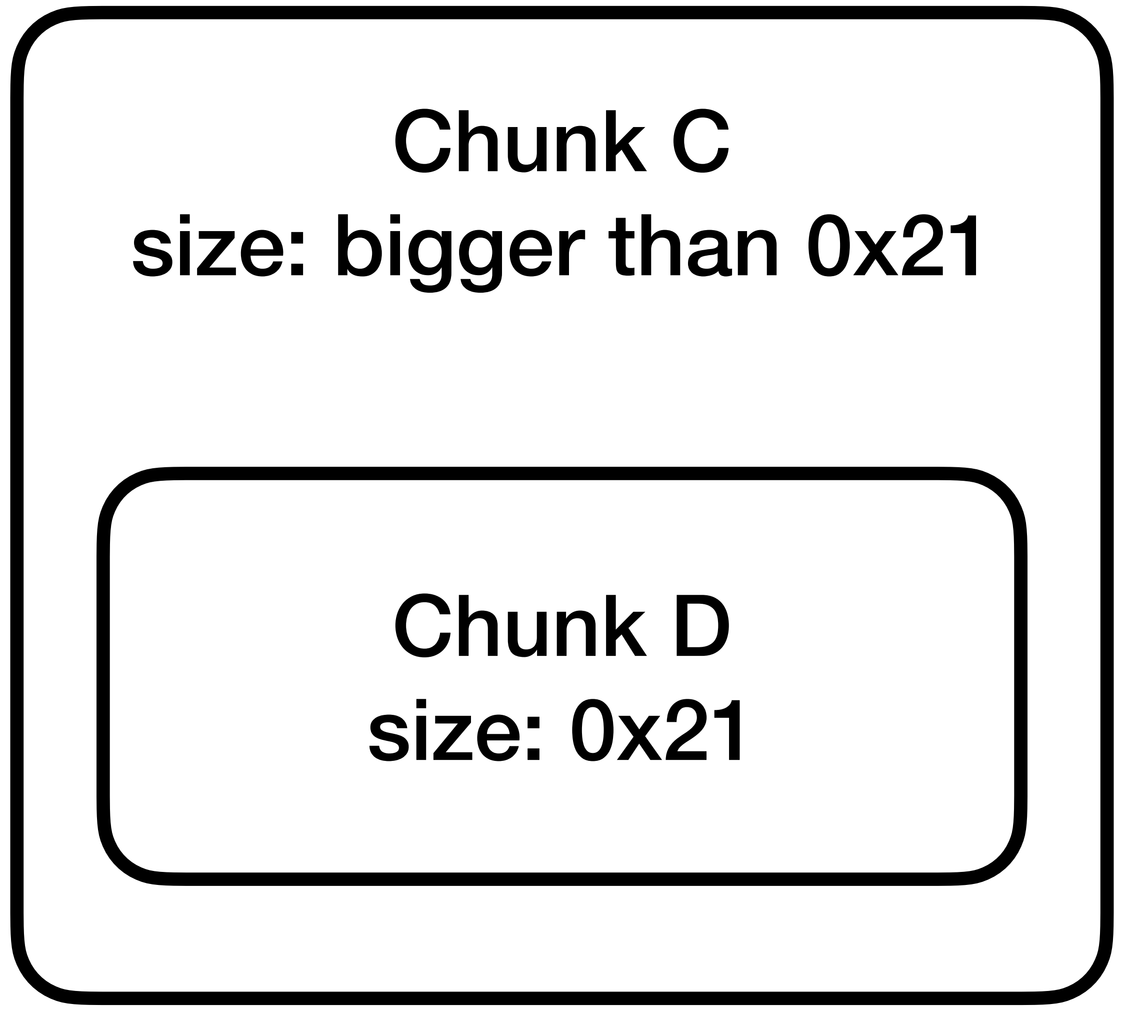
In Memory



We Want to Resize Chunk C



We Want to Resize Chunk C

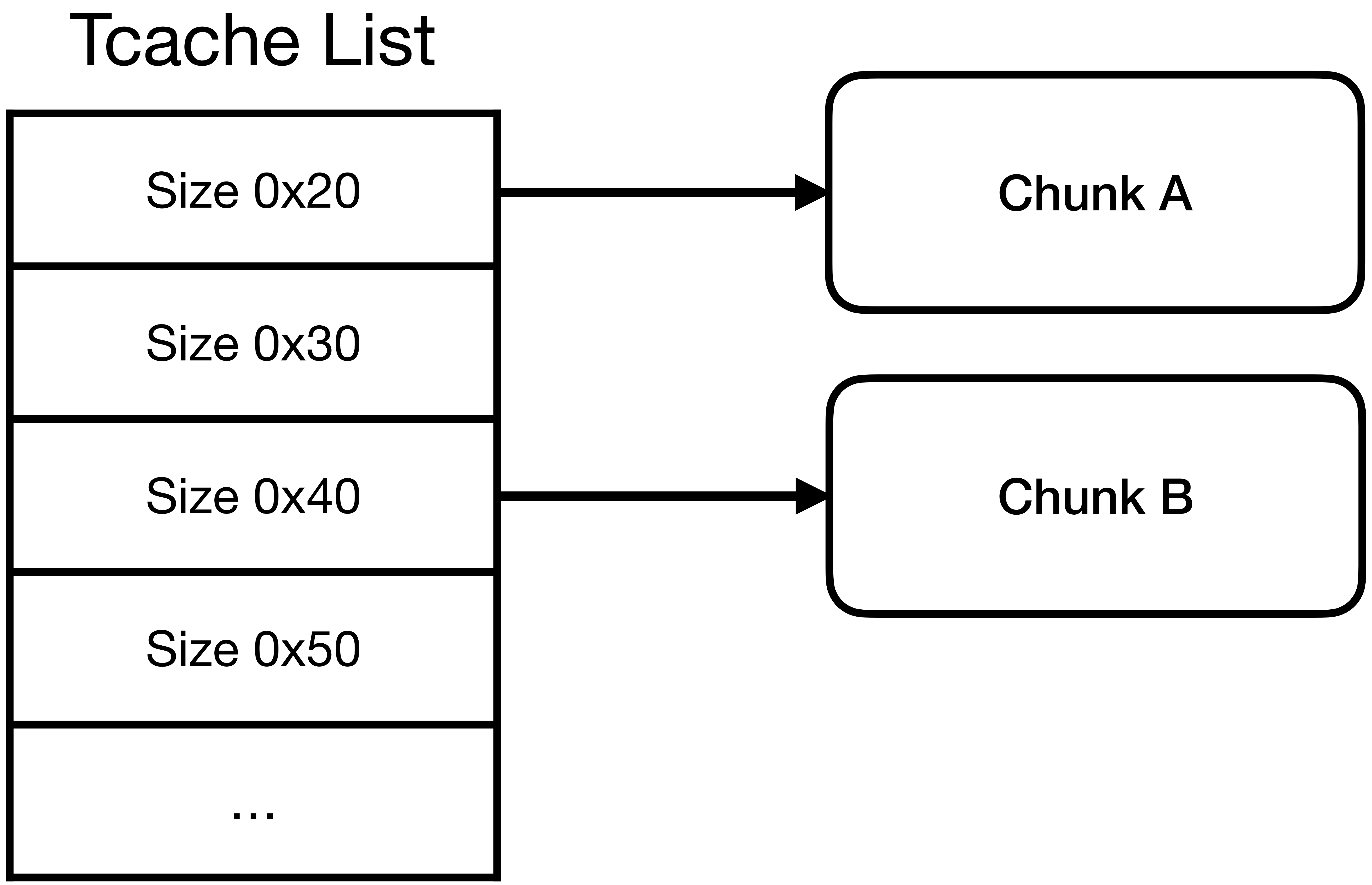


If we can write anywhere
in Chunk C, we can overwrite
Chunk D’s data!

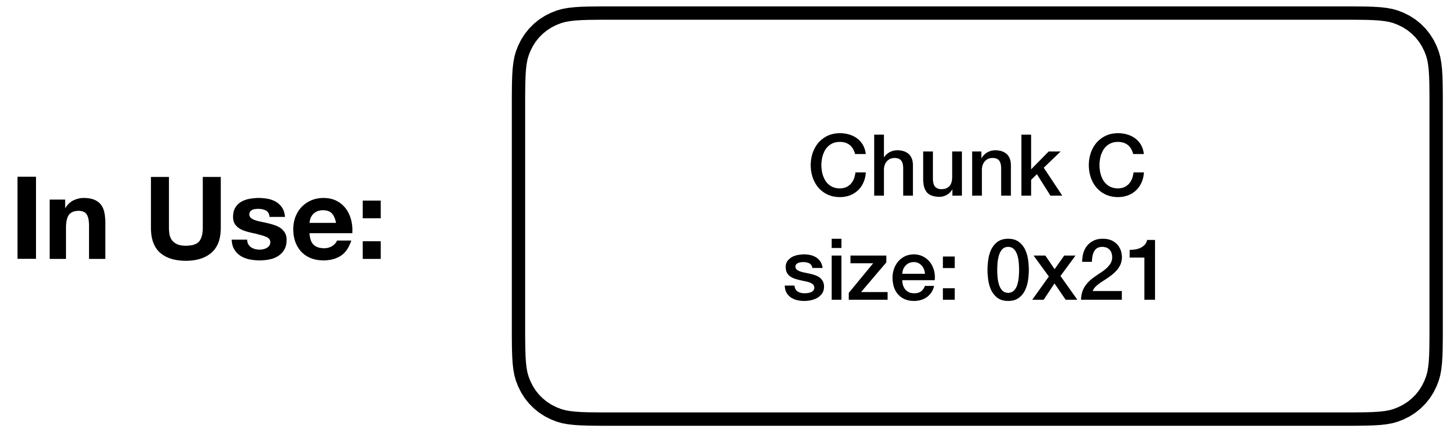
Since tcaches can’t look at other chunks, they can’t check the size value is correct.

In other words- overflow the size field of a tcache and you’ve changed its size!

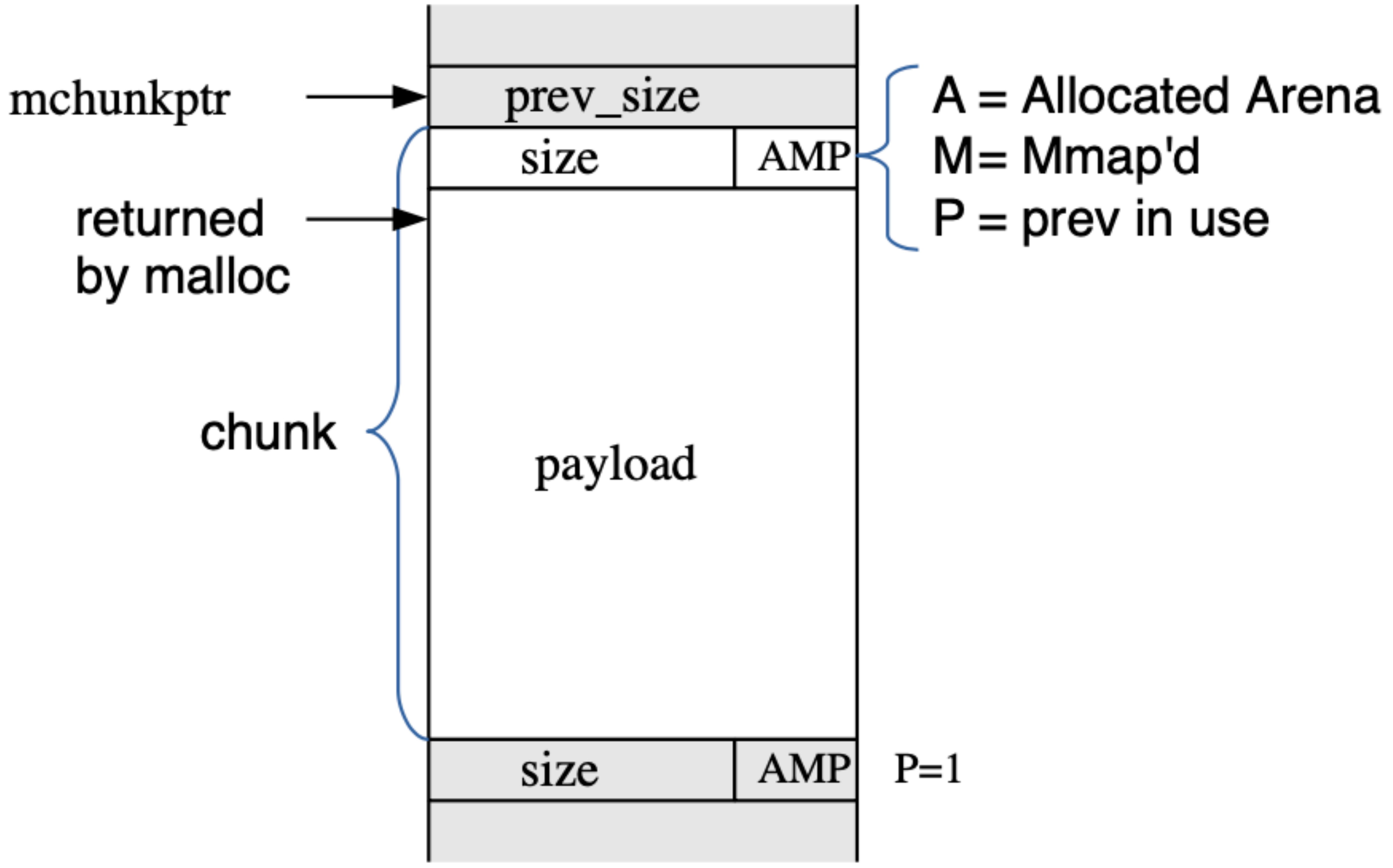
Tcaches



Goal: Get Chunk C into the tcache list of Chunk B



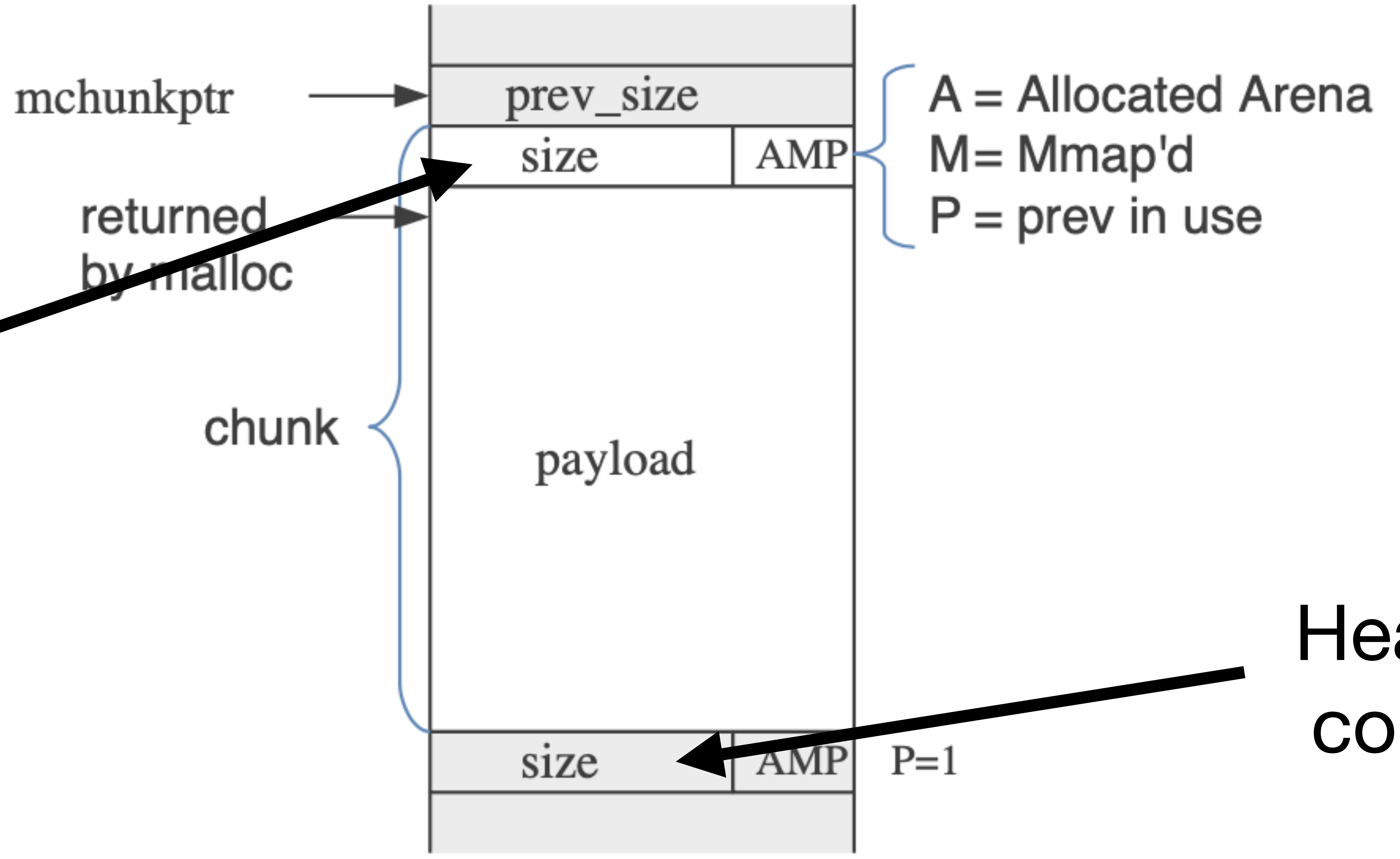
Recall the Glibc In-Use Chunk



https://sourceware.org/glibc/wiki/MallocInternals

Recall the Glibc In-Use Chunk

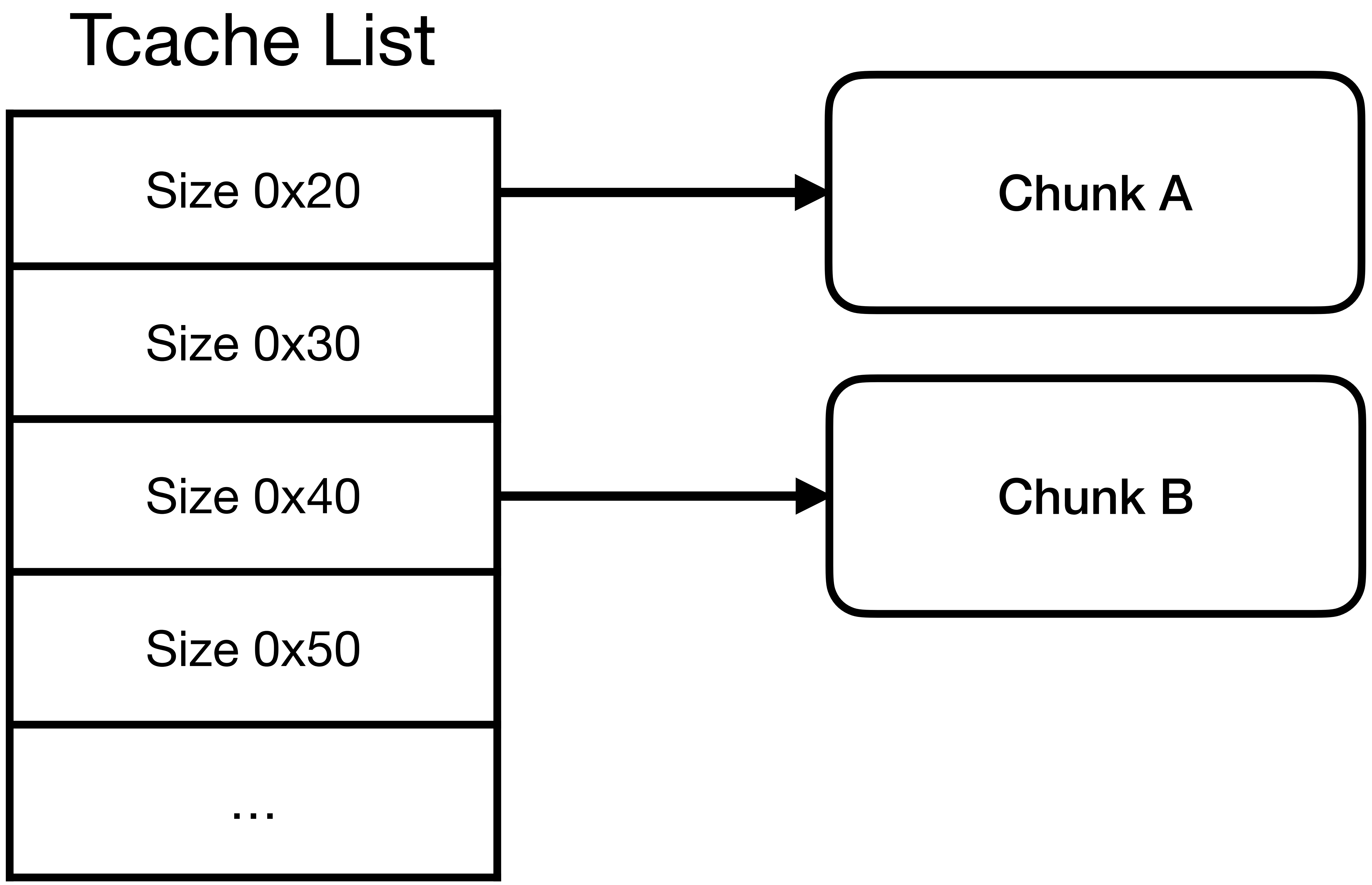
Change this



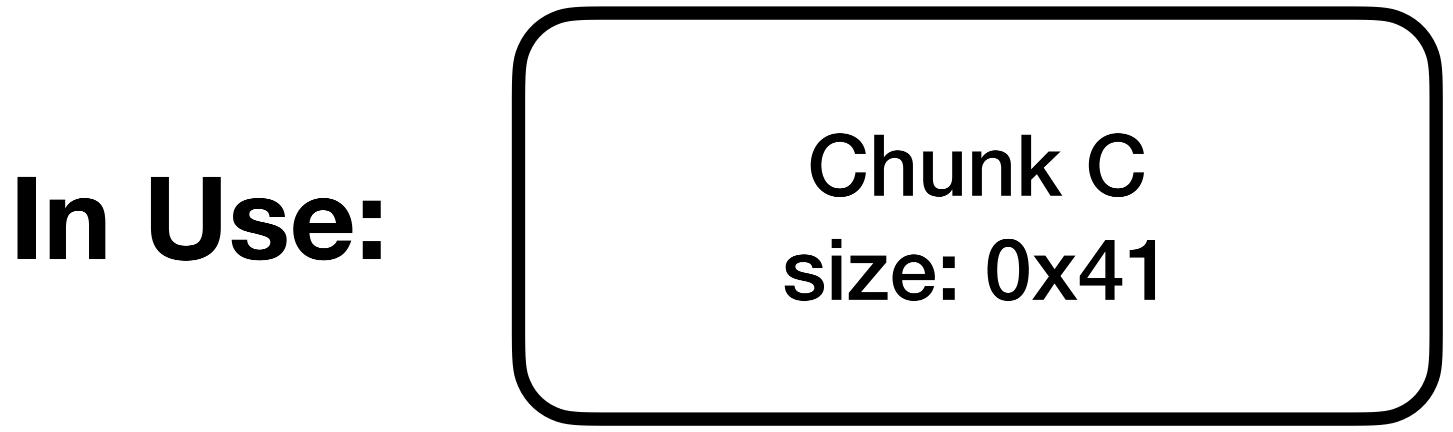
Heap can't check by comparing with this

https://sourceware.org/glibc/wiki/MallocInternals

Tcaches

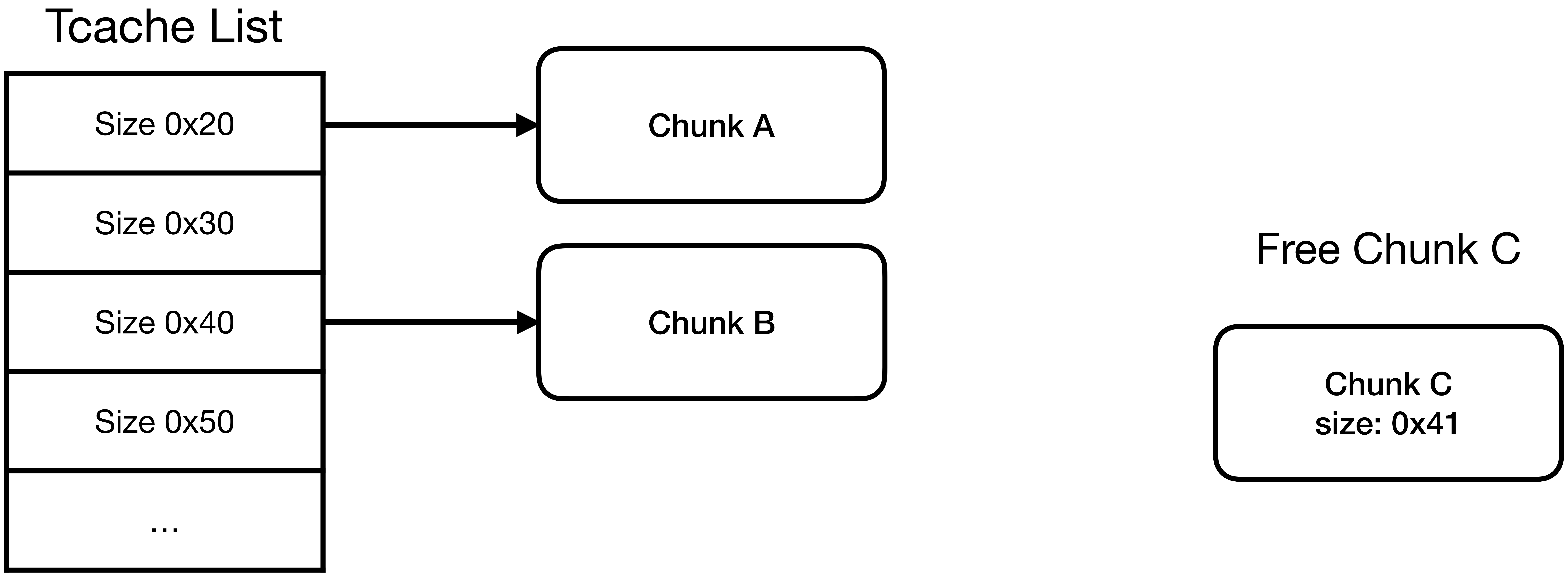


Find an overflow to change Chunk C's size field to 0x41

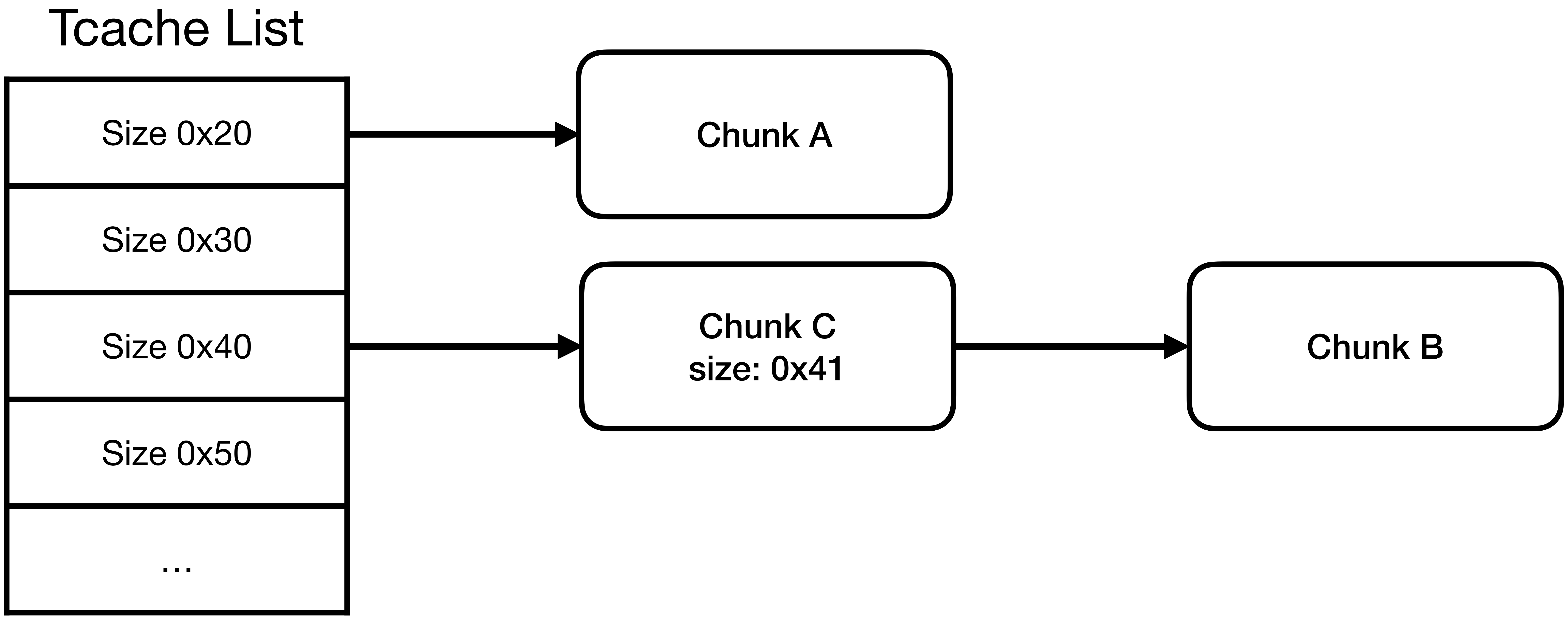


(Why 0x41 and not 0x40?)

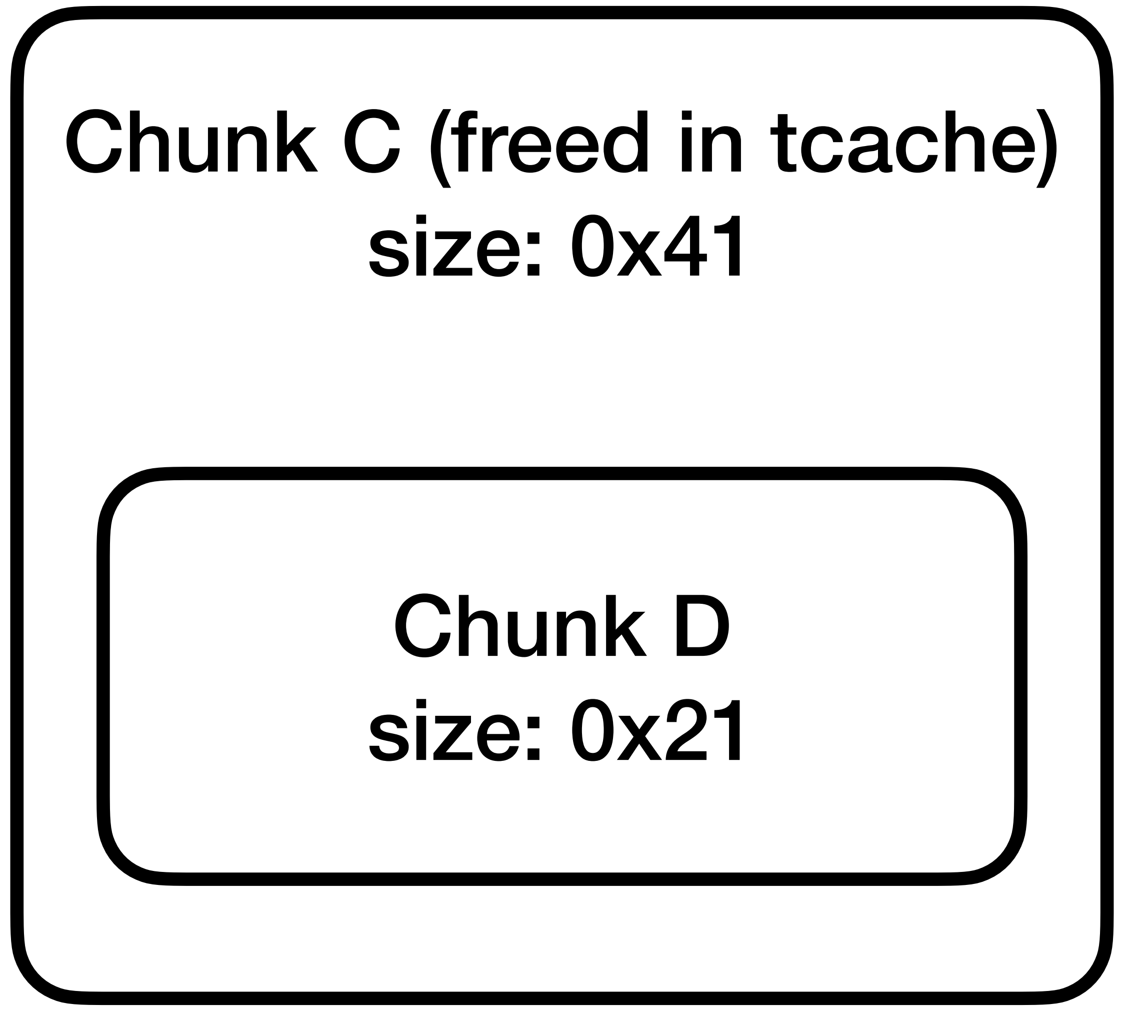
Tcaches


Tcache List
Size 0x20
Size 0x30
Size 0x40
Size 0x50
...
Chunk A
Chunk B
Free Chunk C
Chunk C
size: 0x41

Tcaches

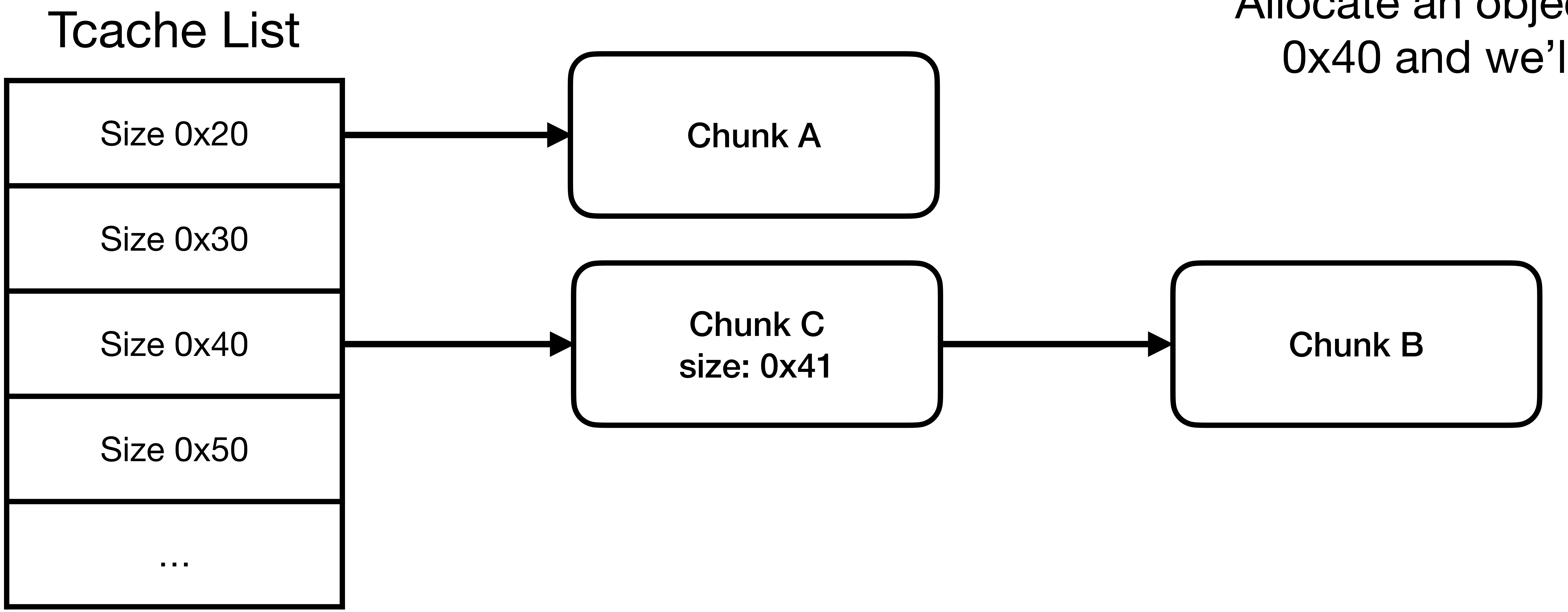

Tcache List
Size 0x20
Size 0x30
Size 0x40
Size 0x50
...
Chunk A
Chunk C
size: 0x41
Chunk B

In Memory

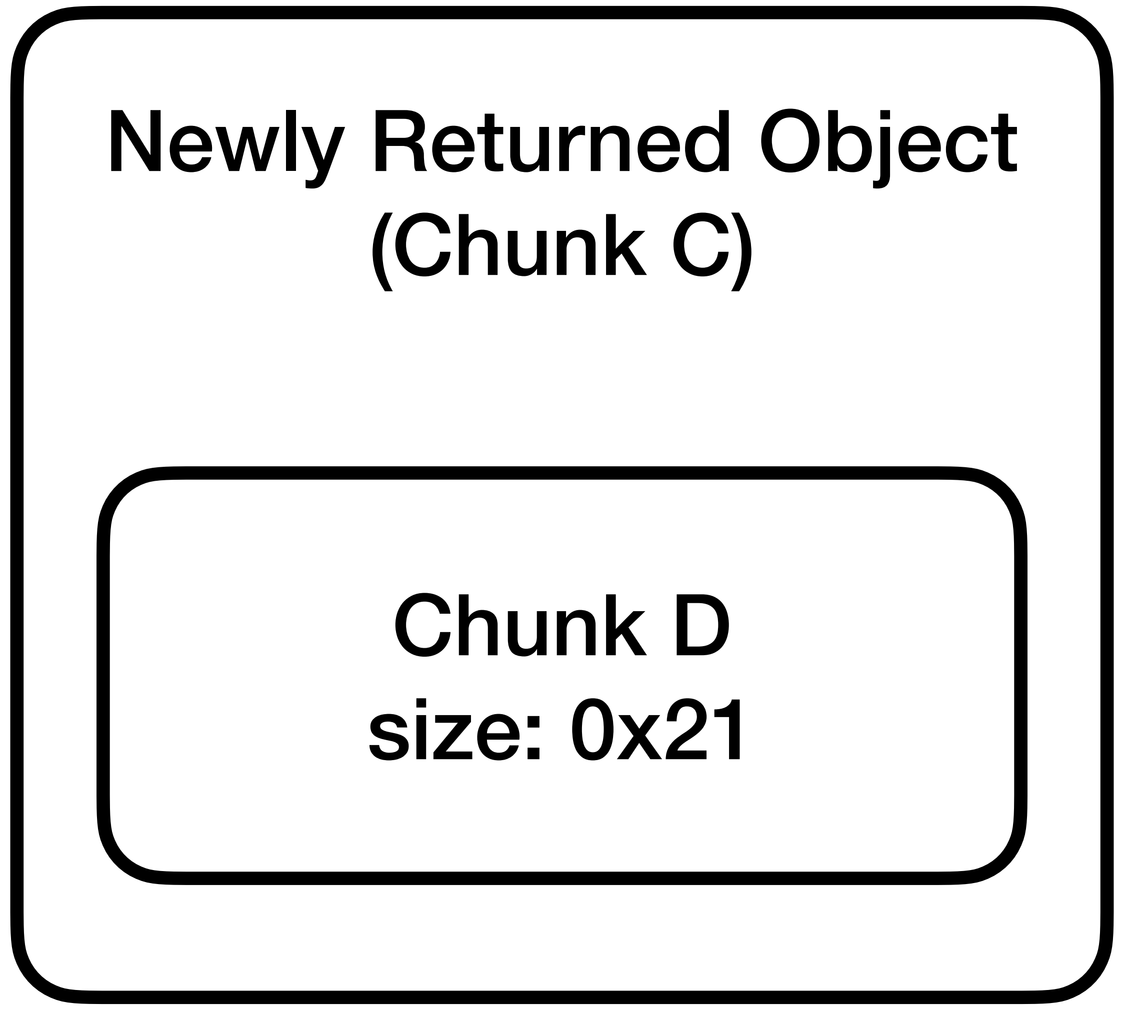


Tcaches

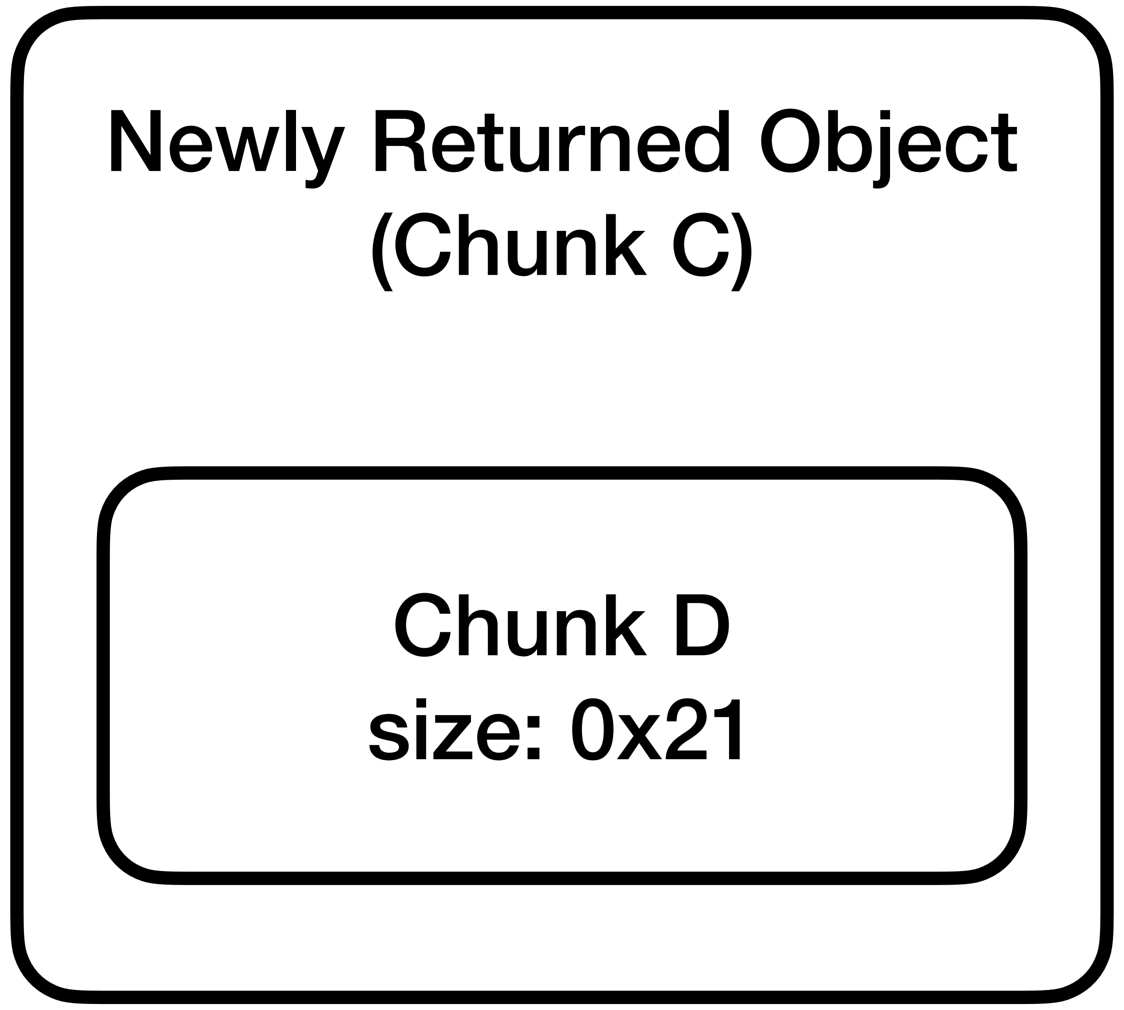
Allocate an object of size 0x40 and we'll get C



In Memory



In Memory



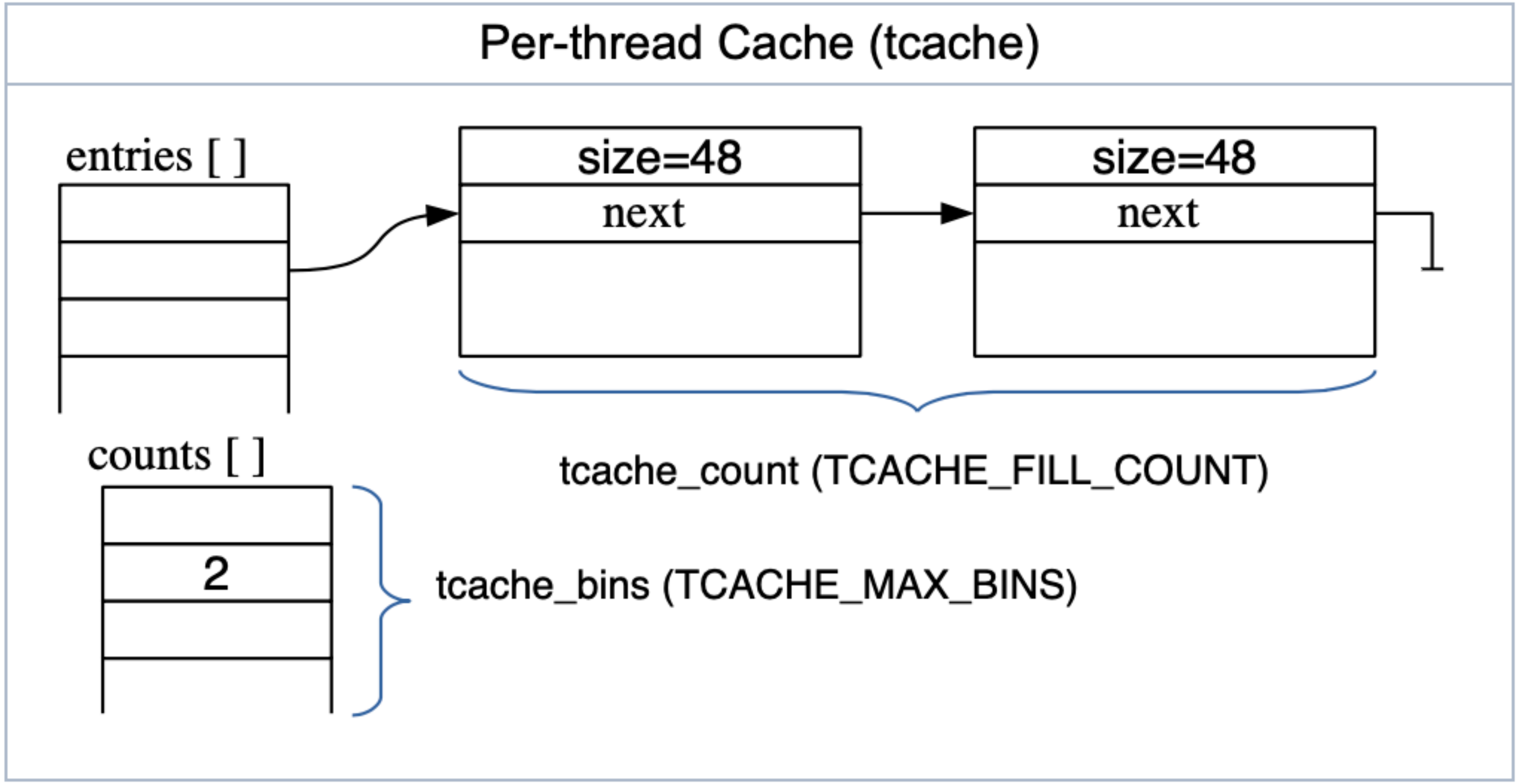
We can now overwrite Chunk D’s contents

Tcache Poisoning

Heap 5

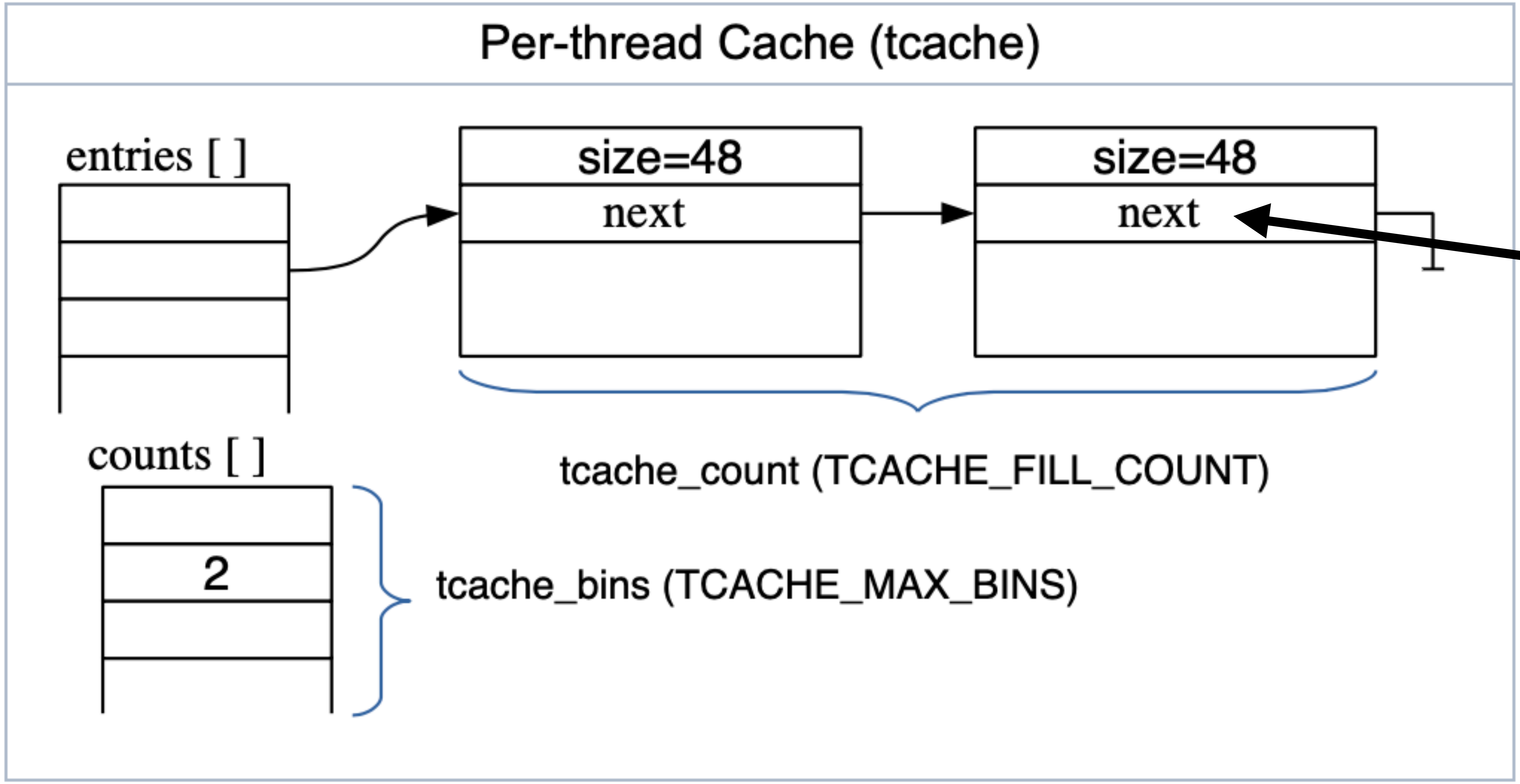
Goal: Return arbitrary memory location from `malloc()`

Recall the Glibc Tcache



https://sourceware.org/glibc/wiki/MallocInternals

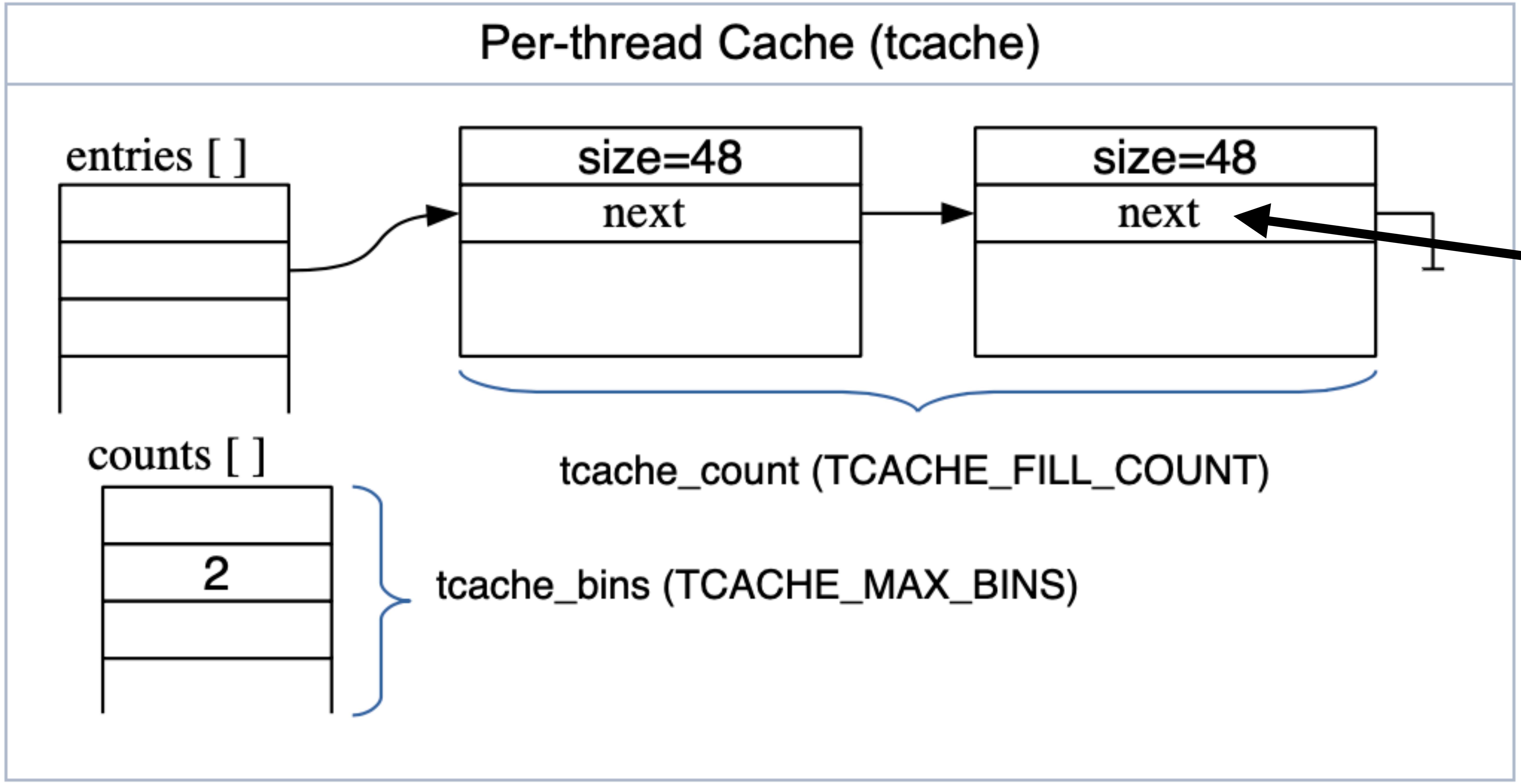
Recall the Glibc Tcache



With a UaF we
can overwrite the
next pointer to corrupt the
tcache linked list

https://sourceware.org/glibc/wiki/MallocInternals

Recall the Glibc Tcache



Recall tcache cannot look at any memory that isn't part of tcache- no way to confirm this corrupt list points to an invalid heap object!

https://sourceware.org/glibc/wiki/MallocInternals