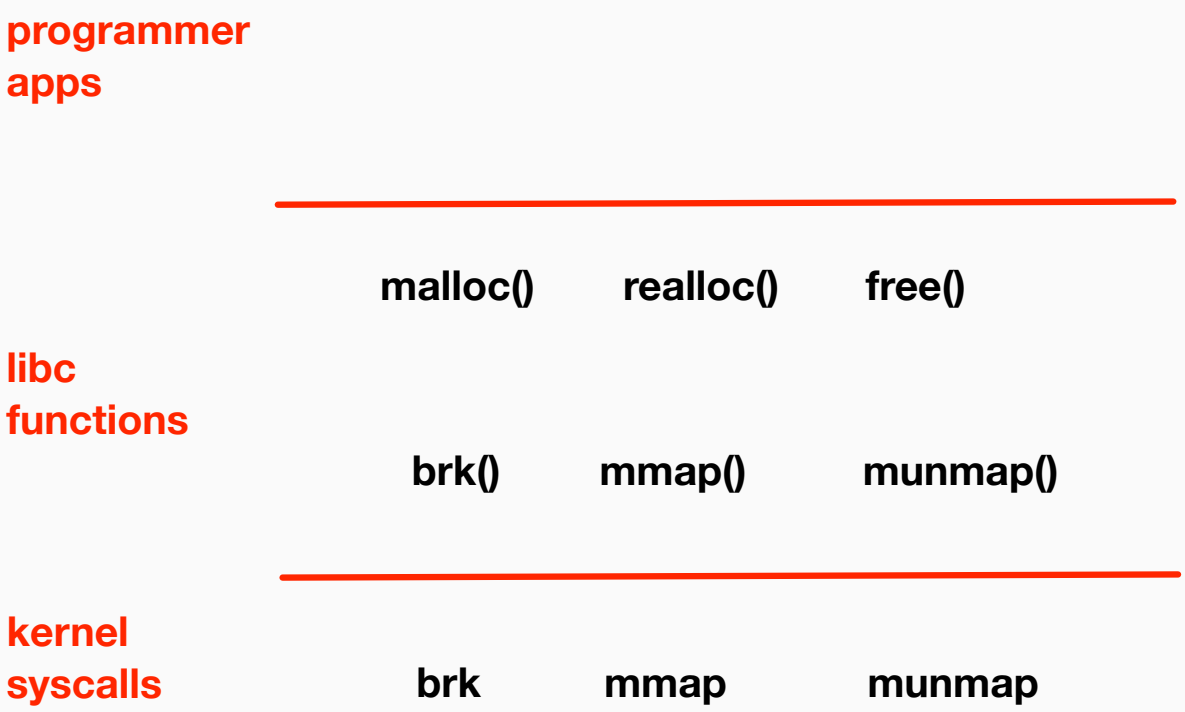
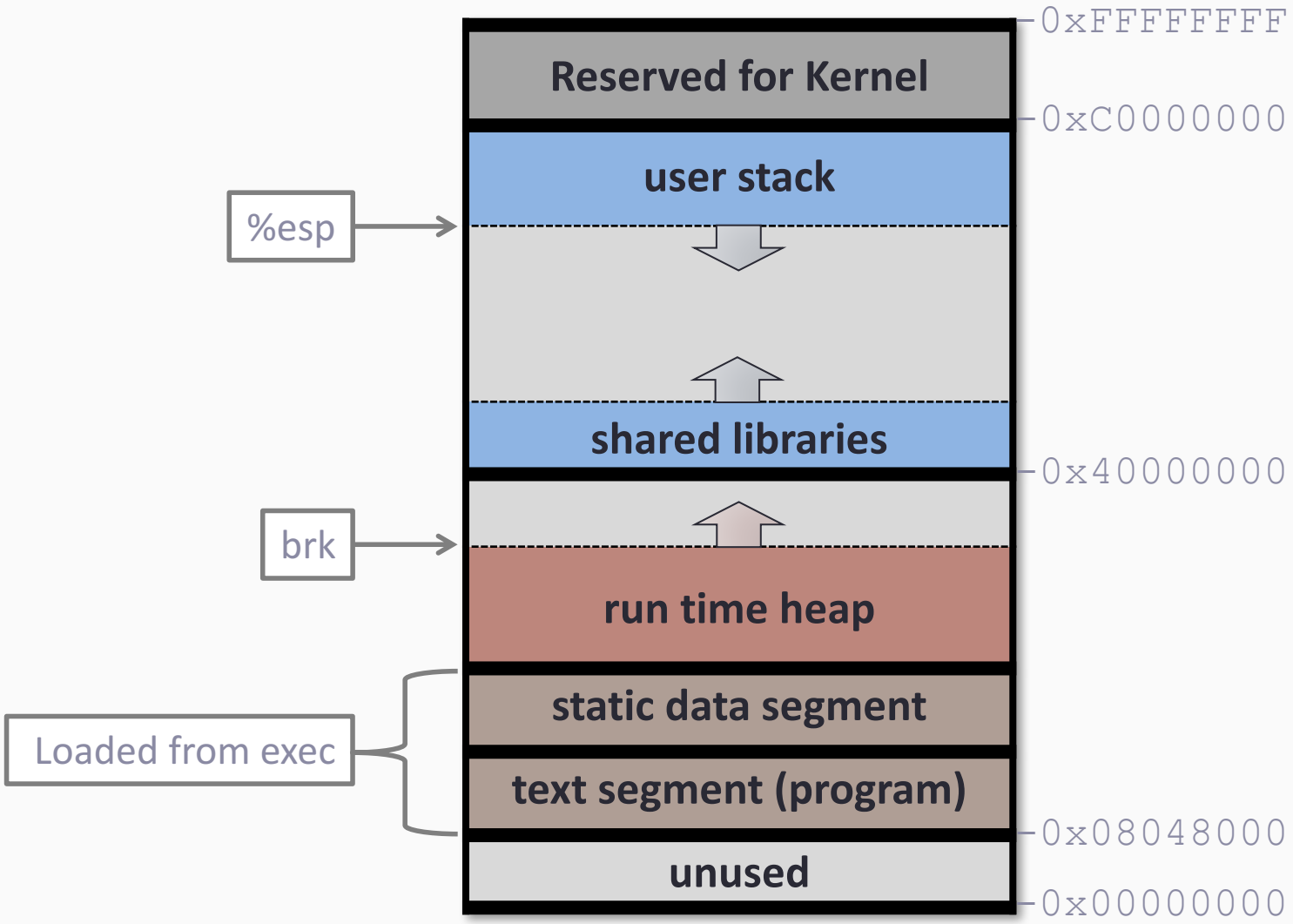
Heap

Chao Zhang
Tsinghua University

Heap Usage


programmer
apps
malloc()
realloc()
free()
libc
functions
brk()
mmap()
munmap()
kernel
syscalls
brk
mmap
munmap

Linux Process Memory Layout (32-bit)



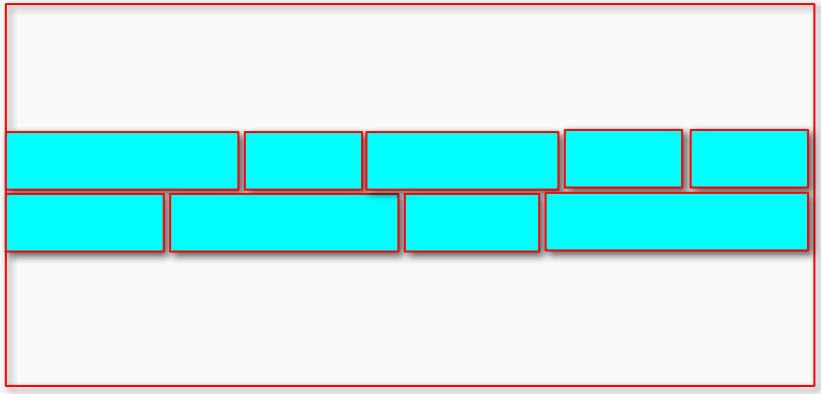
Heap vs. Stack

- Heap
  - dynamic memory allocation at runtime
  - objects, big buffers, structs, persistence, larger things
- Stack
  - Fixed memory allocations known at compile time
  - Local variables, return addresses, function args
- Slower, Manual
  - done by the programmer
  - malloc/calloc/recalloc/free
  - new/delete
- Fast, Automatic
  - done by the compiler
  - abstracts away any concept of allocating/de-allocating.

Memory Allocator's Goals

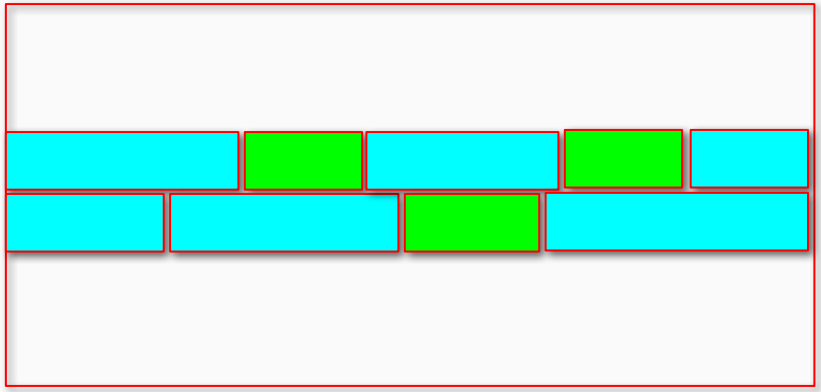
- Efficiency
  - fast to get memory
  - fast to free memory
- Memory fragments
  - very few wasted memory
  - very few fragments

From Memory Allocators' View



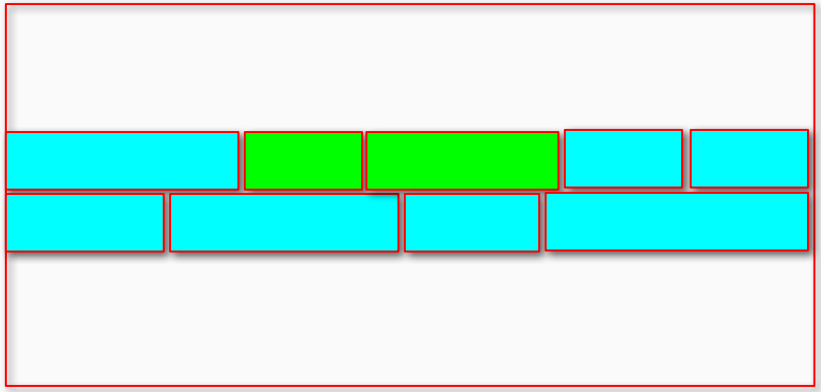
- How to track size of chunks (so we can free them)?
  - p = malloc()
  - free(p)

From Memory Allocators' View



- How to track location of free chunks (so we can reuse them)?

From Memory Allocators' View



- How to collapse continuous free chunks (so we can reduce memory fragments)?

Heap Implementations

- Tons of different heap implementations
  - dlmalloc
  - ptmalloc (glibc)
  - tcmalloc (Chrome, replaced)
  - jemalloc (Firefox/Facebook)
  - nedmalloc
  - Hoard

Glibc's Heap Implementation

Chunk

```
1104 struct malloc_chunk {
1105
1106   INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
1107   INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */
1108
1109   struct malloc_chunk* fd;         /* double links -- used only if free. */
1110   struct malloc_chunk* bk;
1111
1112   /* Only used for large blocks: pointer to next larger size.  */
1113   struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
1114   struct malloc_chunk* bk_nextsize;
1115 };
```

```
typedef struct malloc_chunk* mchunkptr;
```

- So, for every allocated chunk, 6 extra fields are added to the chunk?
  - could be, but too much overhead.

Chunk size

```
1104 struct malloc_chunk {
1105
1106   INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
1107   INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */
1108
1109   struct malloc_chunk* fd;         /* double links -- used only if free. */
1110   struct malloc_chunk* bk;
1111
1112   /* Only used for large blocks: pointer to next larger size.  */
1113   struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
1114   struct malloc_chunk* bk_nextsize;
1115 };
```

- Solution:
  - only the size field is attached to objects.
  - Other fields are shared with neighbors

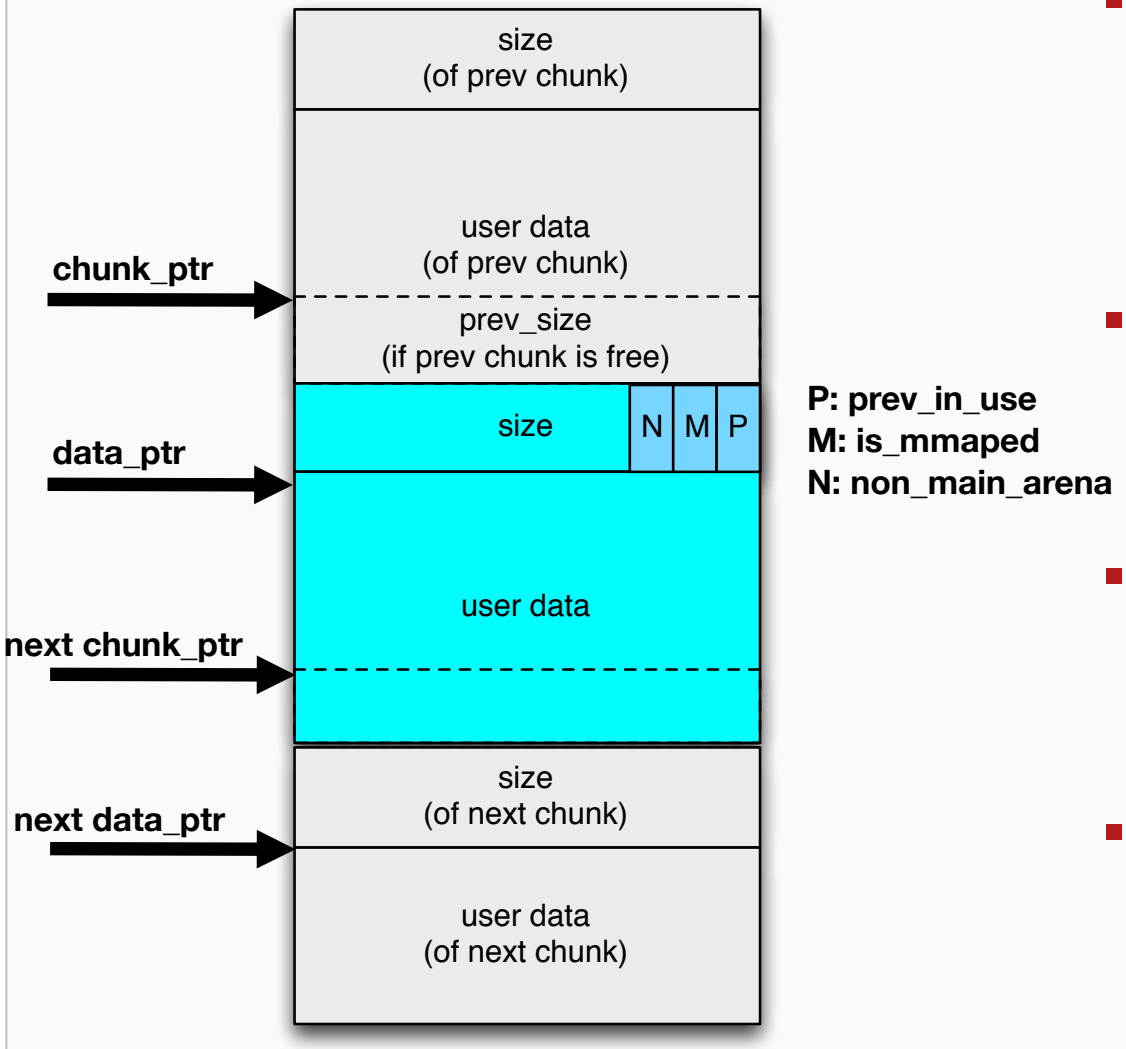
- 32bit: (x+4) align to 8

data size	0~4	5~12	13~20	...	53~60	61~68	69~76	77~84
chunk size	16	16	24	...	64	72	80	88

- 64bit: (x+8) align to 16

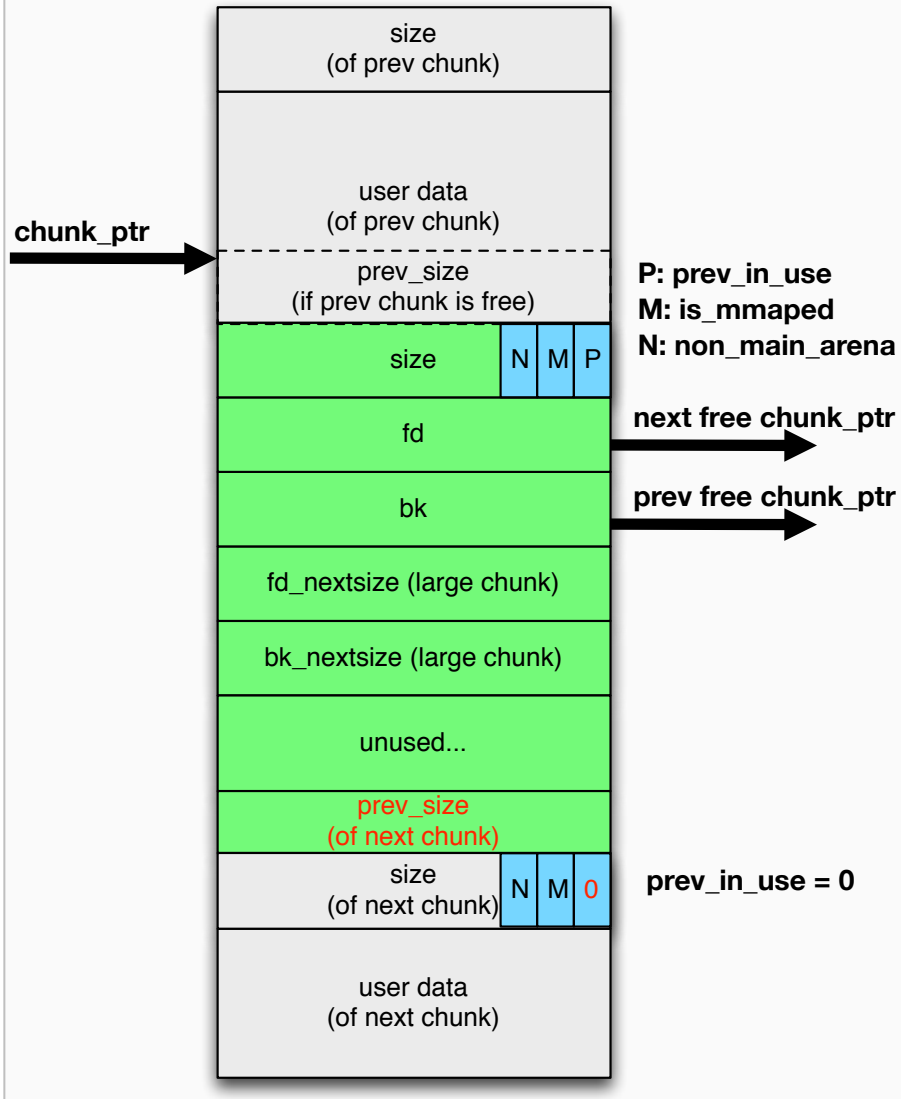
data size	0~8	9~24	25~40	...	105~120	121~136	137~152	153~168
chunk size	32	32	48	...	128	144	160	176

Chunk In Use



- prev_size field
  - belongs to previous chunk
  - is set when previous chunk is free (and not in fastbins)
- prev_in_use
  - is set when previous chunk is free (and not in fastbins)
- is_mmaped
  - this memory is from mmap()
- non_main_arena
  - this chunk is allocated by non-main thread

Chunk Freed



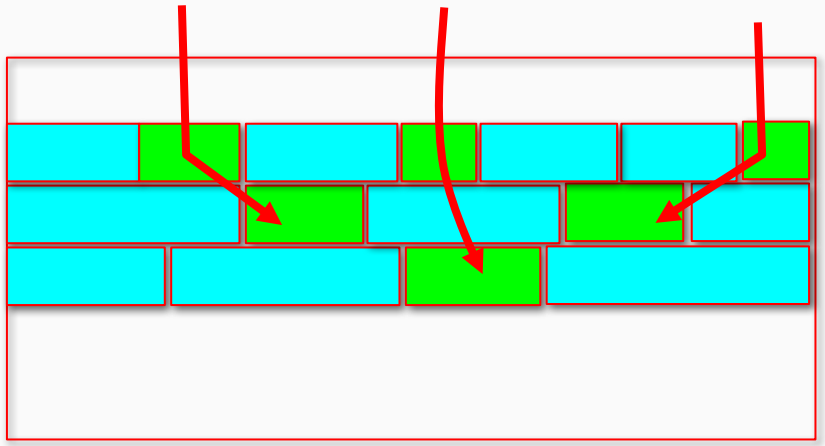
- prev_size of next chunk
  - is set to this free chunk's size
- prev_in_use of next chunk
  - is cleared
- But,
  - they are not set if this free chunk is kept in fastbins

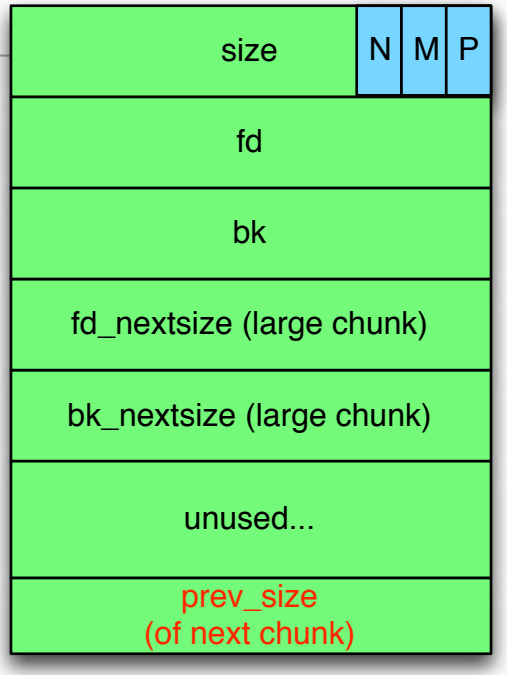
Bookkeeping

- Chunks in use
  - are referenced by programmers' data pointers
  - size are attached to the chunks themselves
    - free() could work thanks to it.
  - NO NEED to bookkeep them separately

- Chunks freed
  - are no longer referenced by programs
  - will be reused later for further memory allocation
    - due to the limitation of memory resources
  - WE NEED to bookkeep these free chunks
    - so malloc() could reuse them
  - How?

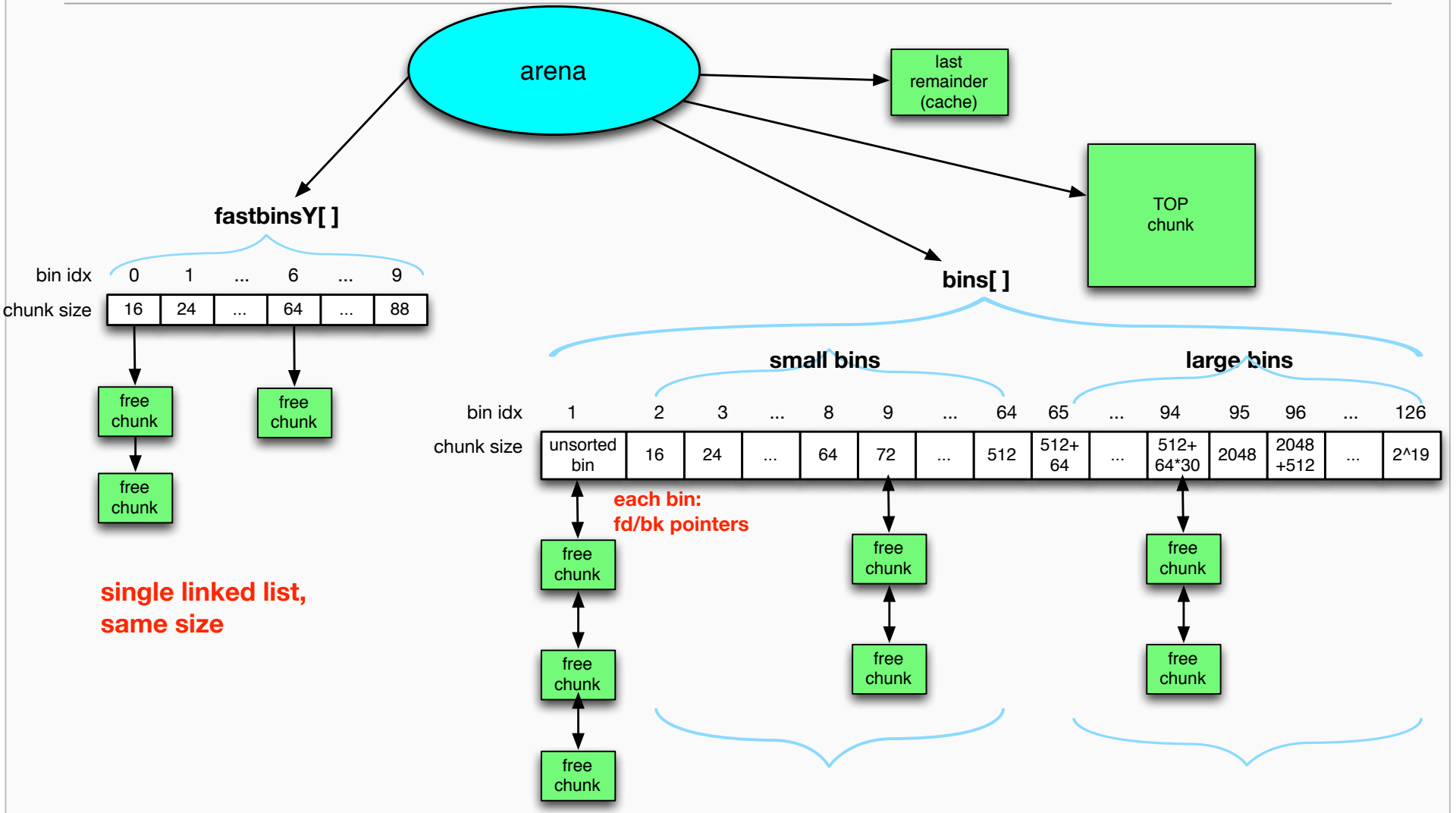
Bookkeep Free Chunks





- A link list to keep a set of free chunks
  - usually, one list has one same chunk size
- Several link lists are needed
  - different size
  - single linked, or double linked
- Link lists are grouped into several BINs
  - fastbins, unsorted_bin, smallbins, largebins

Arena (32-bit allocators)

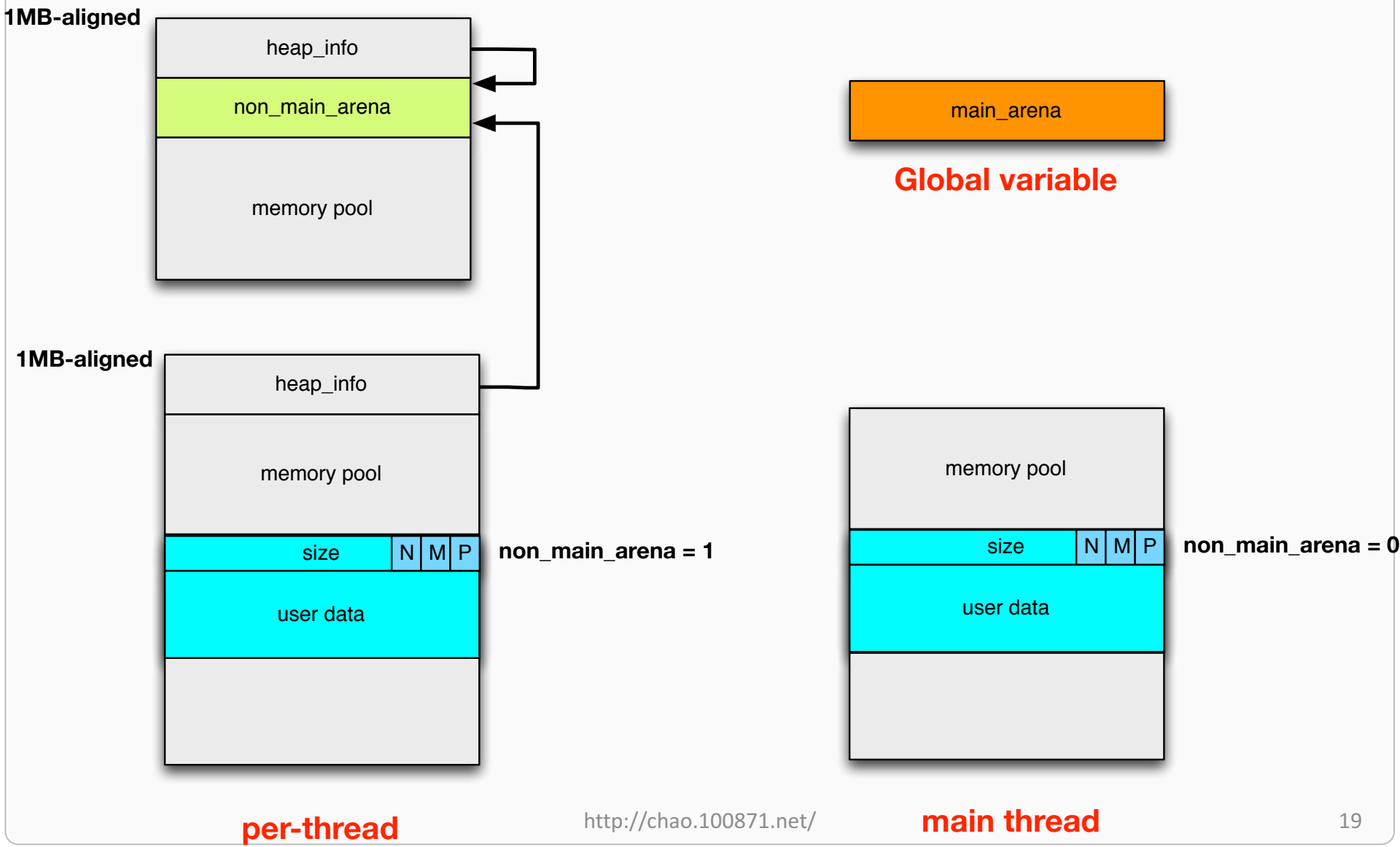


malloc_state: bookkeep free chunks

```
1667 struct malloc_state
1668 {
1669   /* Serialize access.  */
1670   mutex_t mutex;
1671
1672   /* Flags (formerly in max_fast).  */
1673   int flags;
1674
1675 #if THREAD_STATS
1676   /* Statistics for locking.  Only used if THREAD_STATS is defined.  */
1677   long stat_lock_direct, stat_lock_loop, stat_lock_wait;
1678 #endif
1679
1680   /* Fastbins */
1681   mfastbinptr fastbinsY[NFASTBINS];
1682
1683   /* Base of the topmost chunk -- not otherwise kept in a bin */
1684   mchunkptr top;
1685
1686   /* The remainder from the most recent split of a small request */
1687   mchunkptr last_remainder;
1688
1689   /* Normal bins packed as described above */
1690   mchunkptr bins[NBINS * 2 - 2];
1691
1692   /* Bitmap of bins */
1693   unsigned int binmap[BINMAPSIZE];
1694
1695   /* Linked list */
1696   struct malloc_state *next;
1697
1698   /* Linked list for free arenas.  */
1699   struct malloc_state *next_free;
1700
1701   /* Memory allocated from the system in this arena.  */
1702   INTERNAL_SIZE_T system_mem;
1703   INTERNAL_SIZE_T max_system_mem;
1704 };
```

```
typedef struct malloc_state *mstate;
```

Get Arena


1MB-aligned
heap_info
non_main_arena
memory pool
main_arena
Global variable
1MB-aligned
heap_info
memory pool
size
N
M
P
non_main_arena = 1
user data
memory pool
size
N
M
P
non_main_arena = 0
user data
per-thread
main thread

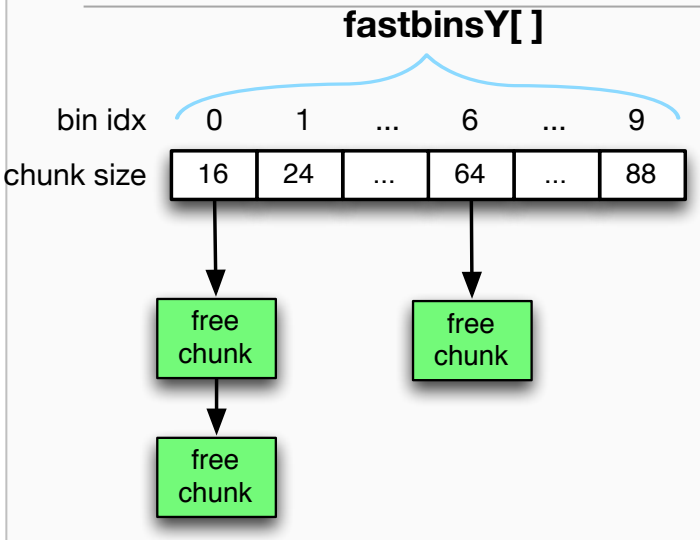
Get Arena

```
0125 #define heap_for_ptr(ptr) \
0126   ((heap_info *) ((unsigned long) (ptr) & ~(HEAP_MAX_SIZE - 1)))
0127 #define arena_for_chunk(ptr) \
0128   (chunk_non_main_arena (ptr) ? heap_for_ptr (ptr)->ar_ptr : &main_arena)
```

```
1741 static struct malloc_state main_arena =
1742 {
1743   .mutex = MUTEX_INITIALIZER,
1744   .next = &main_arena
1745 };
```

```
0056 typedef struct _heap_info
0057 {
0058   mstate ar_ptr; /* Arena for this heap. */
0059   struct _heap_info *prev; /* Previous heap. */
0060   size_t size;   /* Current size in bytes. */
0061   size_t mprotect_size; /* Size in bytes that has been mprotected
0062                            PROT_READ|PROT_WRITE.  */
0063   /* Make sure the following data is properly aligned, particularly
0064      that sizeof (heap_info) + 2 * SIZE_SZ is a multiple of
0065      MALLOC_ALIGNMENT. */
0066   char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK];
0067 } heap_info;
```

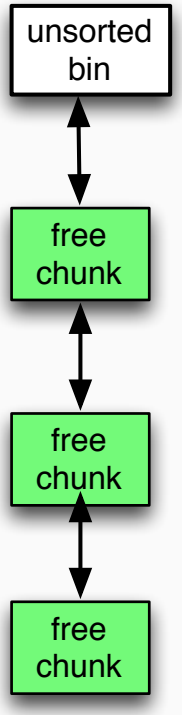
Fast-bins



single linked list, same size

- Keyword: fast
- malloc(fast_size)
  - get one chunk from the specific fast-bin if available
- free(fast_size)
  - place into the specific fast-bin
  - only set fd
  - not set
    - prev_size of next chunk
    - P bit of next chunk

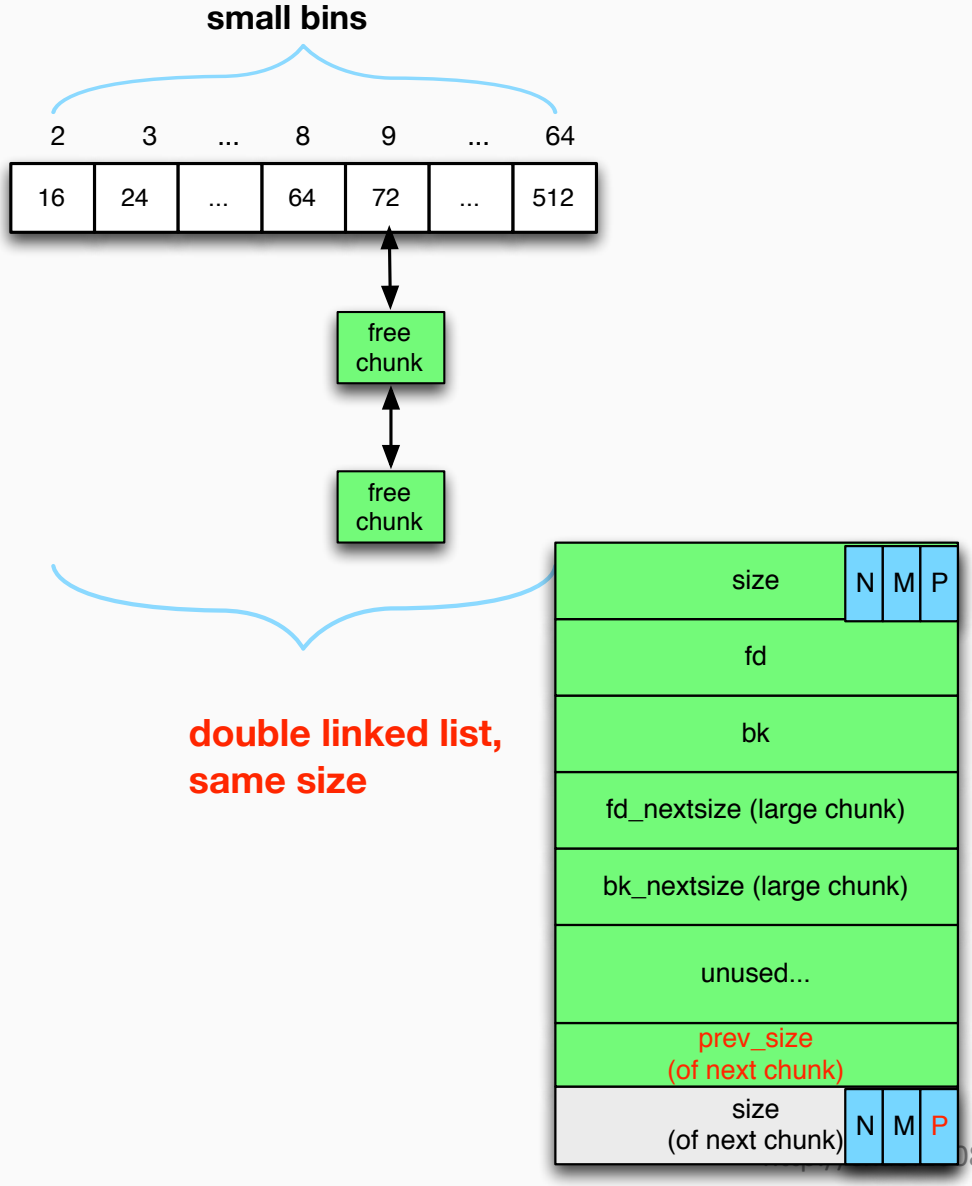
unsorted_bin



double linked list, unsorted size

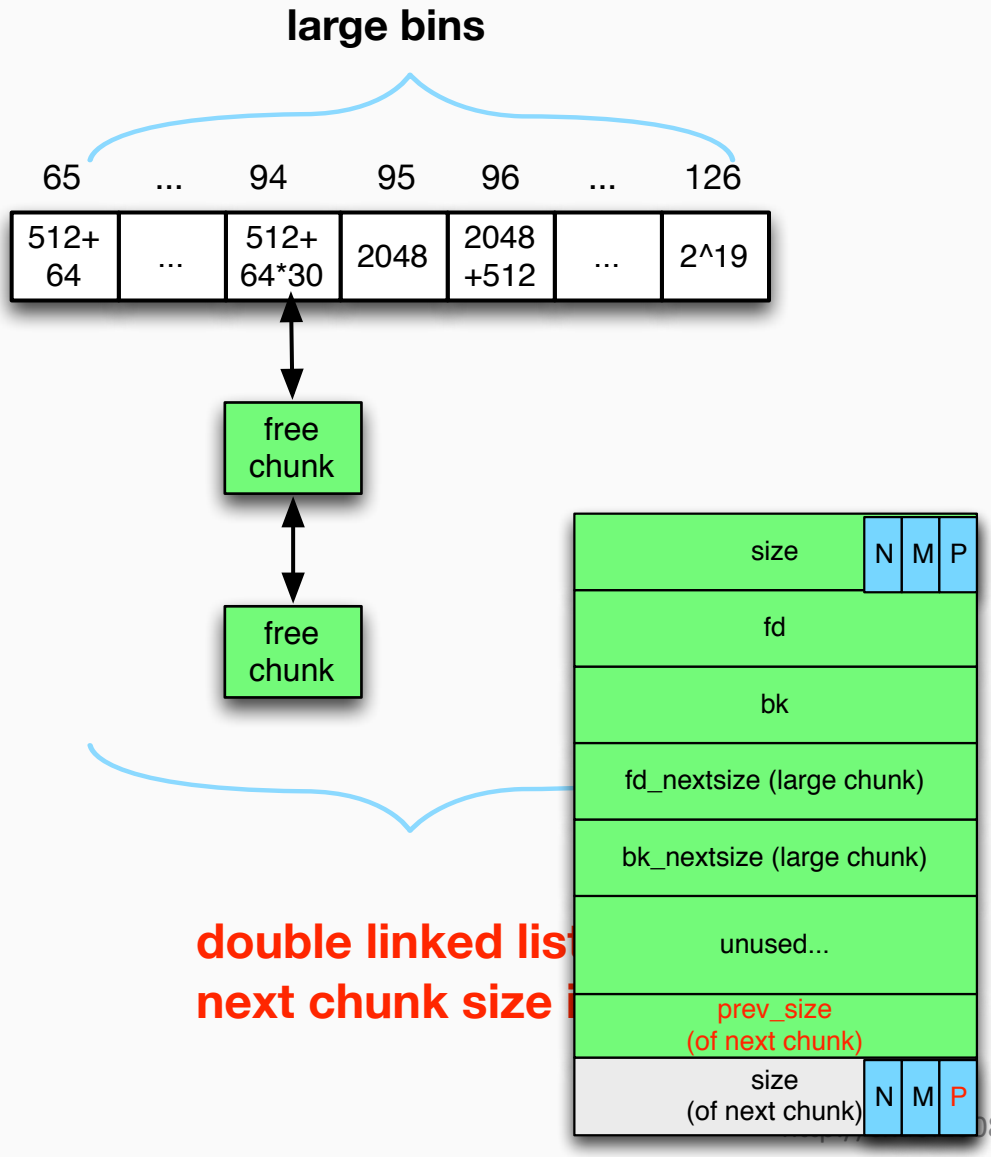
- malloc()
  - if unsorted_bin is searched, iterate chunks in it
    - puts the chunk into small/large bin
      - the only place to insert small/large bin
    - returns the chunk if its size fits and stops iterating
- free()
  - if the freed size is not fast_size (fast bin), always put it into unsorted_bin
    - no need to select small/large bin
  - collapse previous free chunk
    - check P bit of current chunk
  - collapse next free chunk
    - check P bit of next chunk
  - set
    - fd/bk of current chunk
    - prev_size, P bit of next chunk

Small-bins



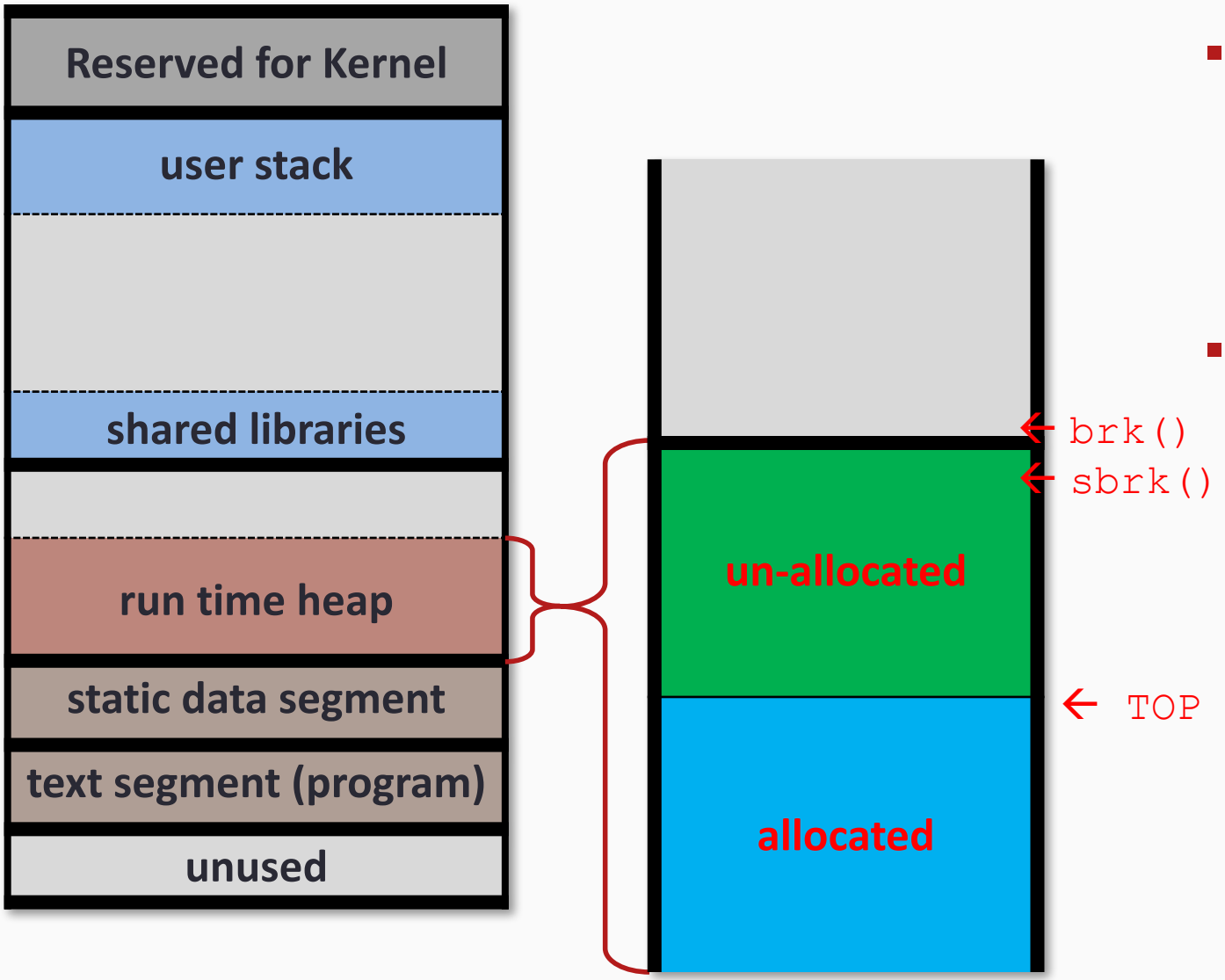
- malloc()
  - if small-bins are searched
    - return the chunk if its size fits
- free()
  - will not put into small-bin directly
- fd/bk pointers are set
  - when removed from unsorted_bin (i.e., unsorted_bin is searched for allocation)

Large-bins



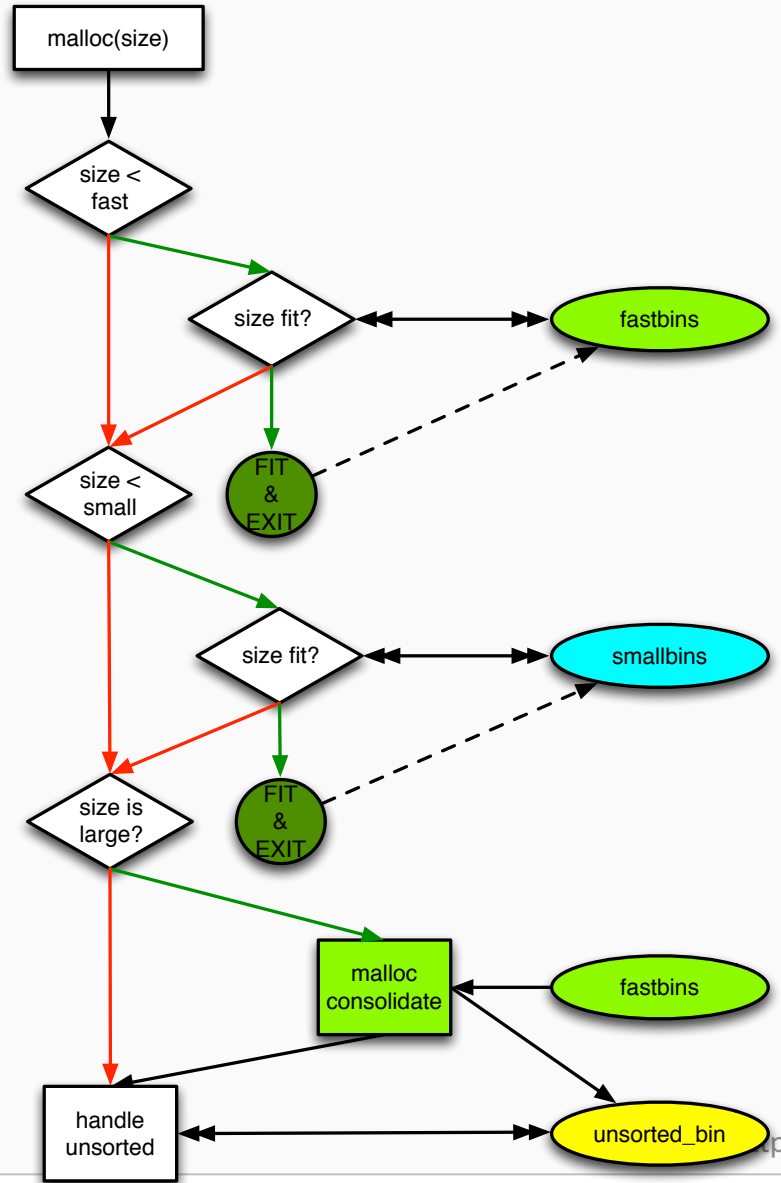
- malloc()
  - if large-bins are searched
    - find the smallest chunk whose size is big enough
    - split this large chunk
      - return one to user
      - put another in unsorted_bin
- free()
  - will not put into large-bin directly
- fd/bk, fd_nextsize/bk_nextsize are set
  - when removed from unsorted_bin

Top



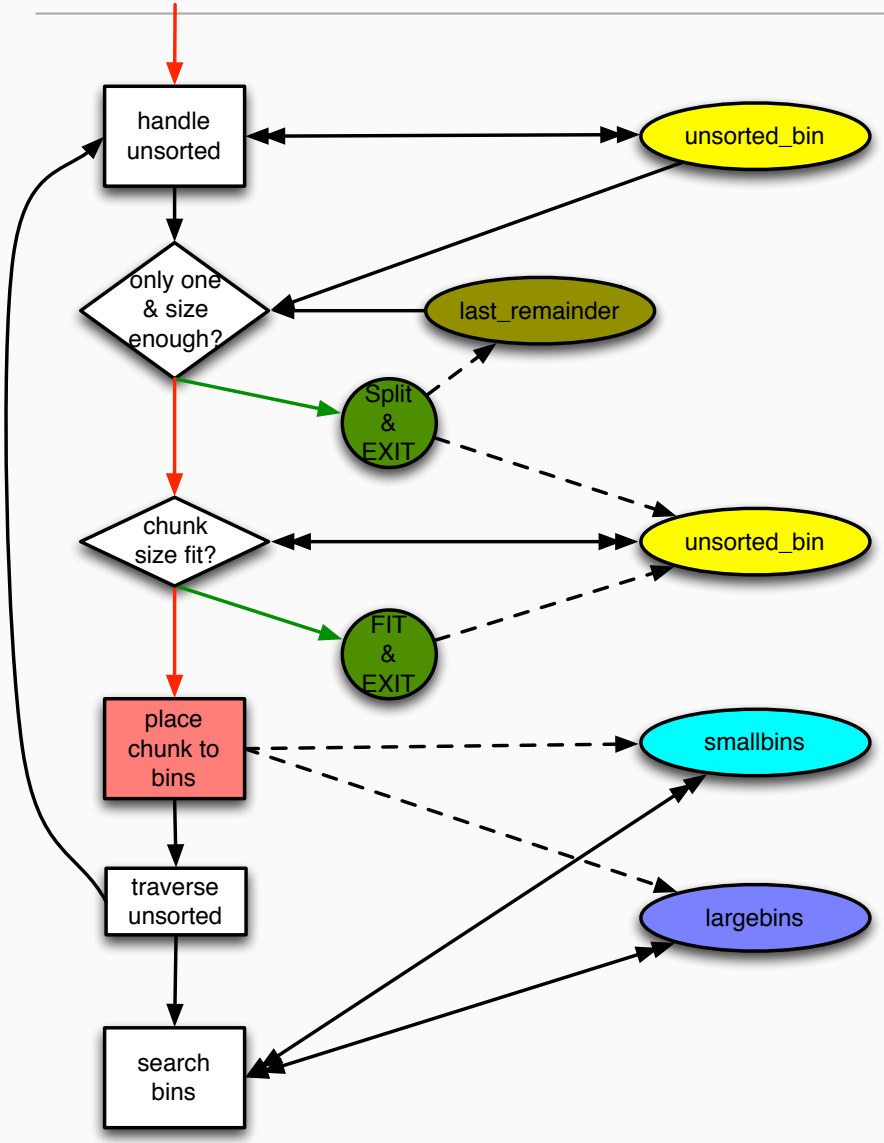
- memory pool allocated by kernel
- glibc allocate objects inside the pool

malloc process (1)



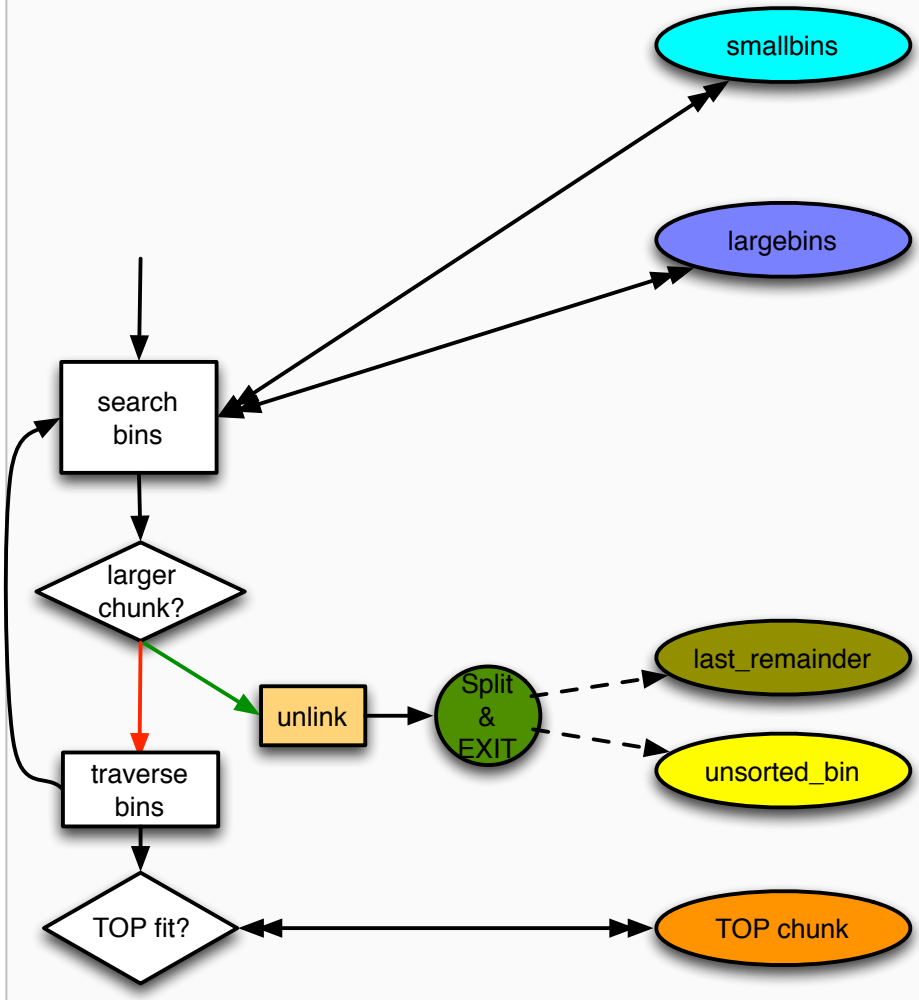
- search fastbins with size, and once found chunk
  - remove it from the single linked list
  - update fd pointer
- search smallbins with size, and once found chunk
  - remove it from the double linked list
  - update fd and bk pointers
- if try to allocate large size, don't search large-bin directly, but consolidate fastbins
  - put fastbins into unsorted_bin
  - fix prev_size and P bit of next chunks
  - merge free chunks if necessary and unlink chunks

malloc process (2)



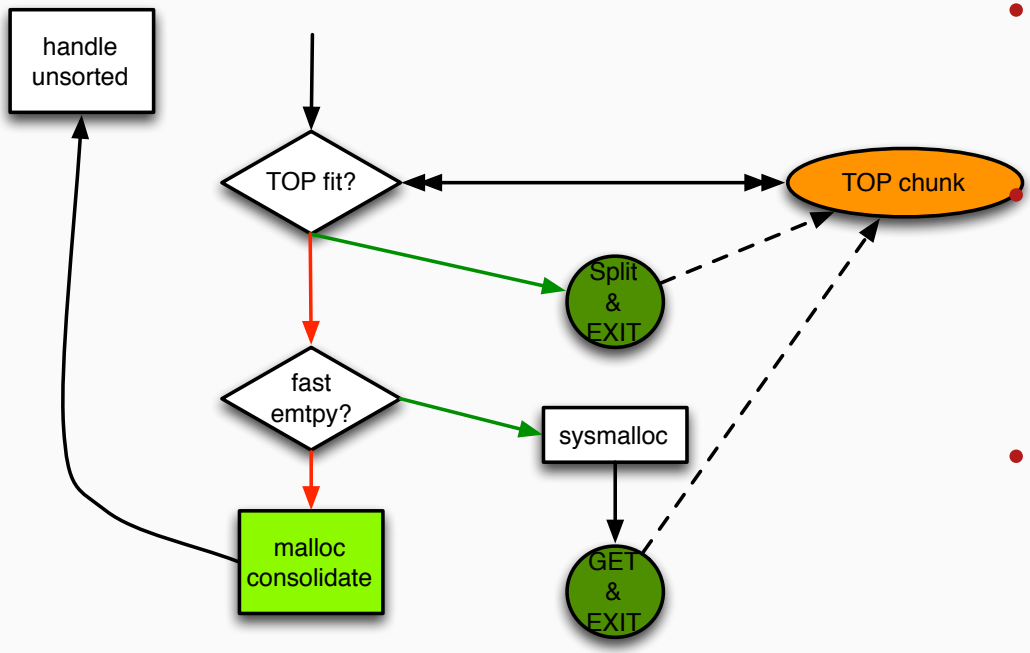
- search in unsorted_bin
  - split if last_remainder cache is big enough
  - returns if the iterated chunk's size fits, otherwise
  - place the iterated chunk into small/large bins
- update fd/bk pointers

malloc process (3)



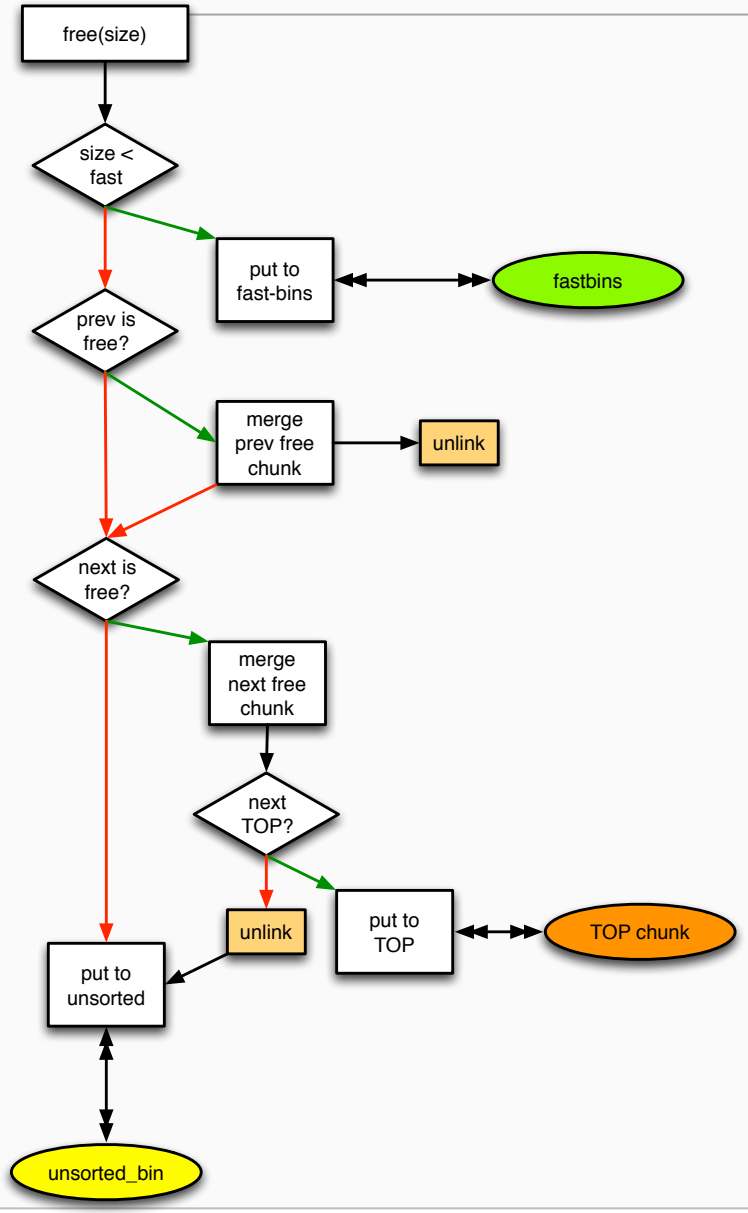
- search in small/large bins
  - find the smallest non-empty small/large bin that has a big enough chunk
    - unlink, split

malloc process (4)



- get memory from TOP
  - split TOP if big enough
- get memory from system if no fast-bins
  - consolidate fast-bins and handle unsorted_bin again.

free process

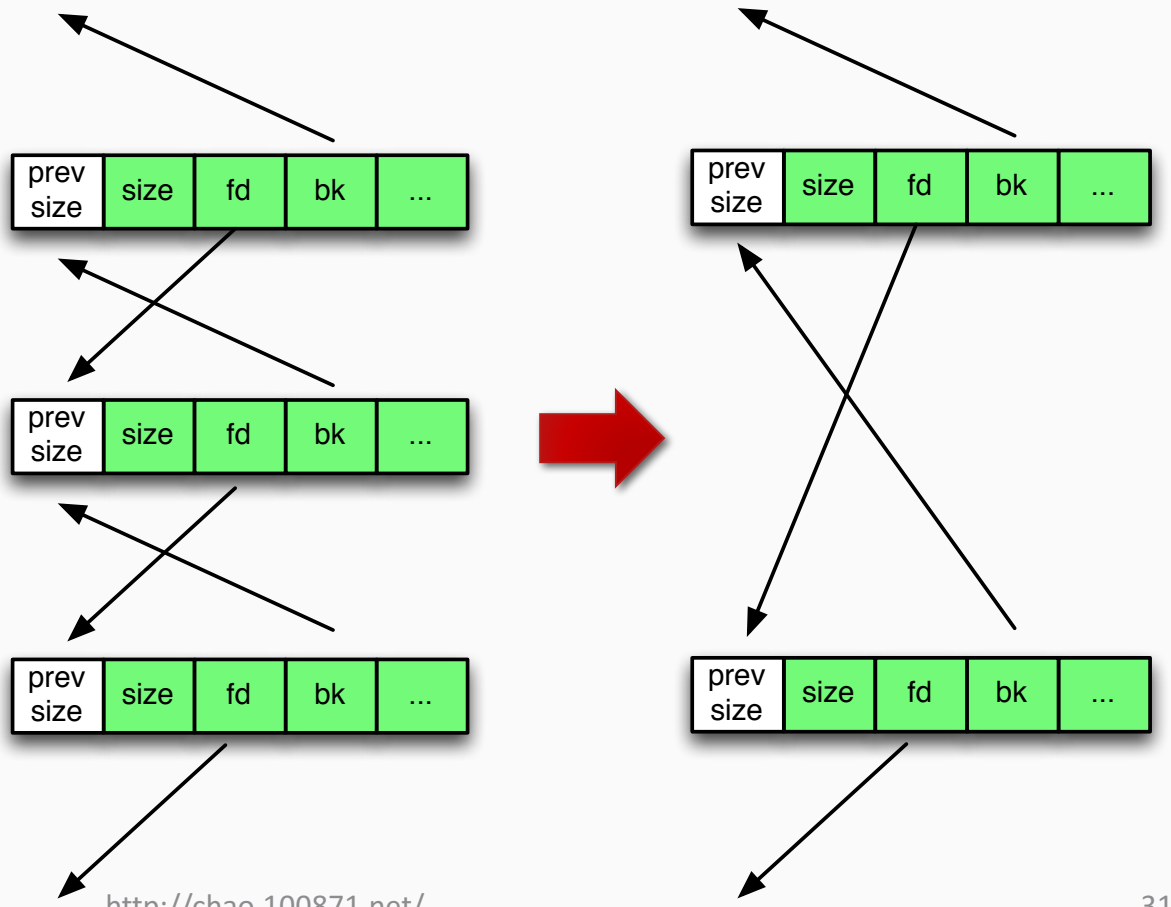


- metadata security check
  - e.g., ptr->fd->bk == ptr
- put into fast-bins if size is within fast-bin ranges
  - update fastbin, fd
- merge previous free chunk
  - unlink
- merge next free chunk
  - unlink, top

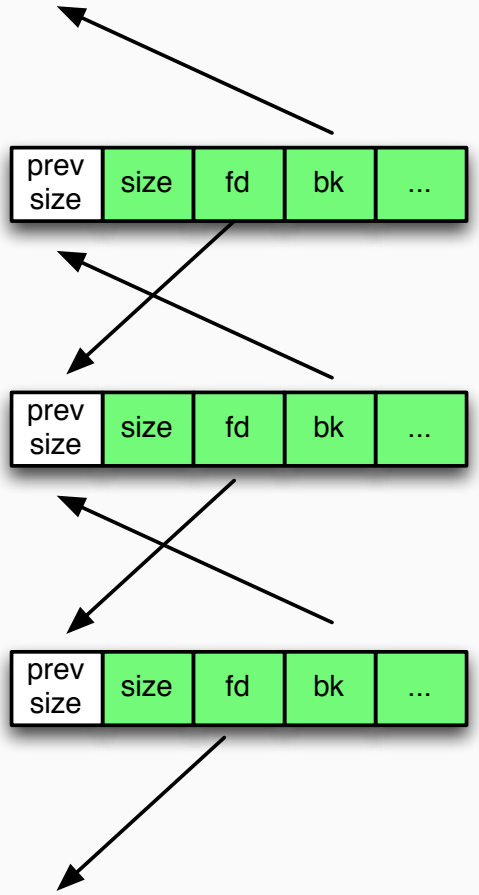
unlink

- remove node, and update BK/FD,

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



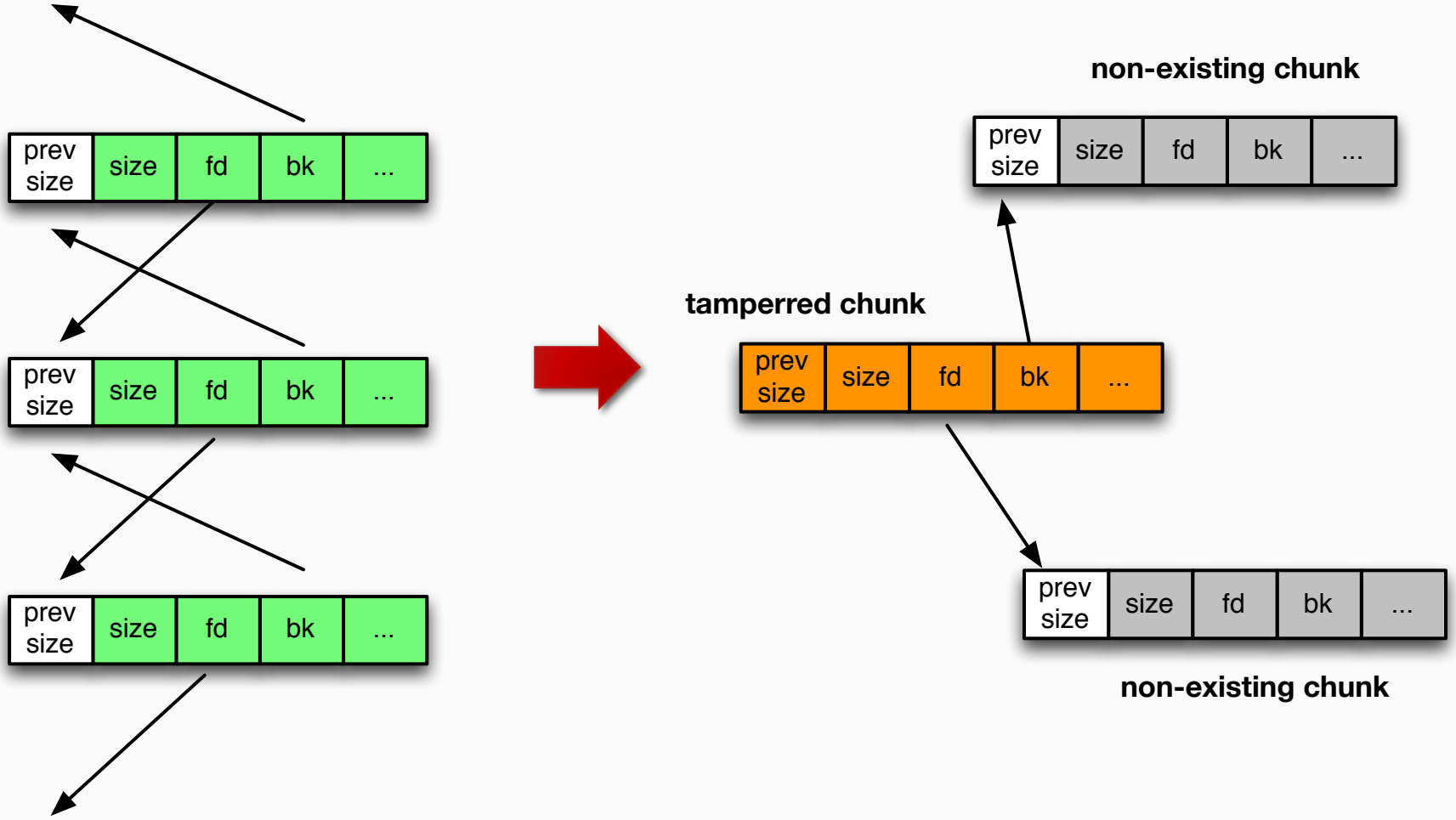
when to unlink?



- Case 1:
  - The free chunk is merged with previous/following neighbor free chunk
- Case 2:
  - The free chunk is picked for allocation.

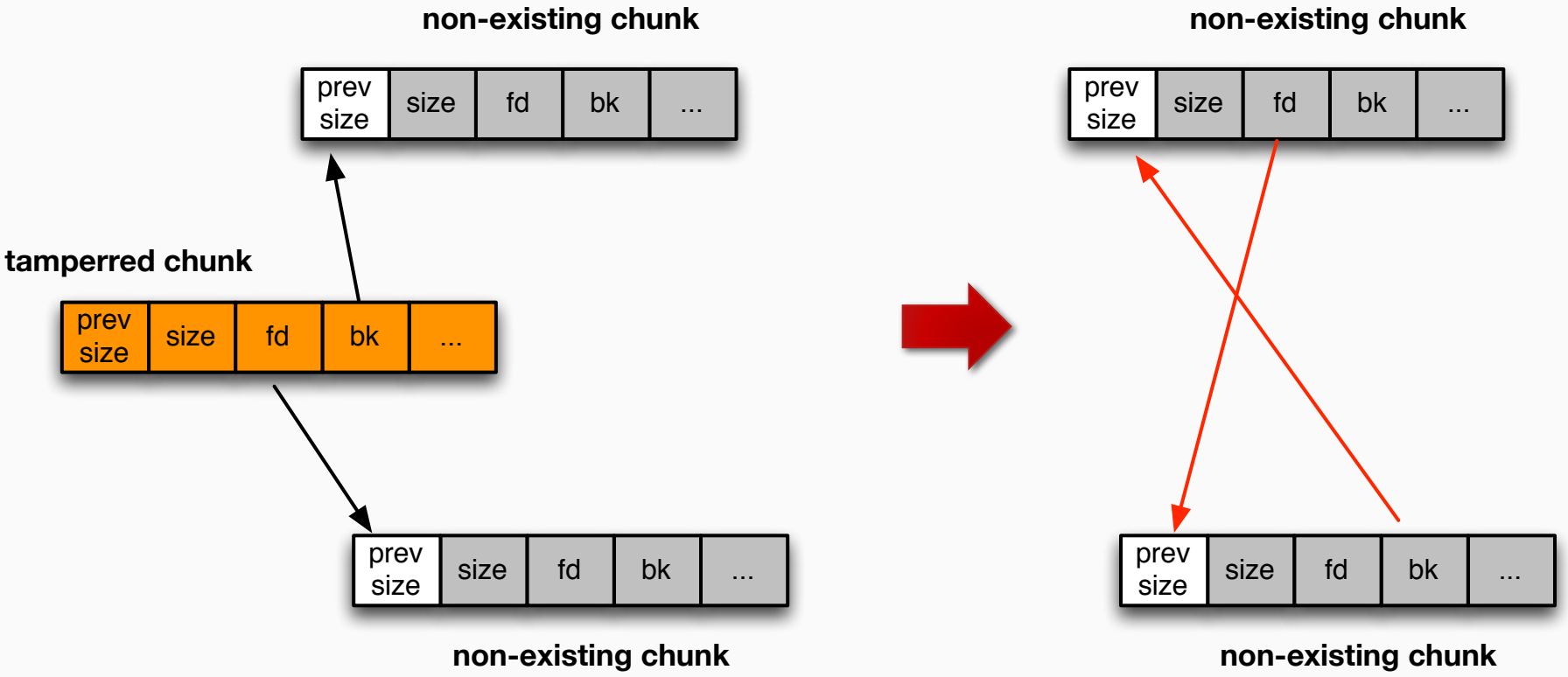
What will happen?

- if the free chunk is compromised?



Then...

- When this chunk is unlinked (e.g., malloc, malloc_considate, free)



Write arbitrary content to arbitrary address!

Sample Code: fastbin and smallbin

```
// fastbins
for(i=0; i < 8; i++)
{
  alloc_size = 8 * i + 4;
  chunk_size = (alloc_size + 7) / 8 *
  chunk_size = chunk_size<16 ? 16 : ch
  p1 = (char*) malloc(alloc_size);
  p2 = (char*) malloc(alloc_size);
  p3 = (char*) malloc(alloc_size);
  p4 = (char*) malloc(alloc_size);
  printf("alloc_size: %d, chunk_size:
          p1: %p, p2: %p, p3: %p, p4:
  memset(p1, 'a', alloc_size);
  memset(p2, 'b', alloc_size);
  memset(p3, 'c', alloc_size);
  memset(p4, 'd', alloc_size);
  free(p2);
  free(p3);   // will not collapse, al
}
```

```
// smallbins
for(i=8; i < 12; i++)
{
  alloc_size = 8 * i + 4;
  chunk_size = (alloc_size + 7) / 8 * 8;        //
  chunk_size = chunk_size<16 ? 16 : chunk_size;
  p1 = (char*) malloc(alloc_size);  // try order: s
  p2 = (char*) malloc(alloc_size);  // when try uns
  p3 = (char*) malloc(alloc_size);
  p4 = (char*) malloc(alloc_size);
  printf("alloc_size: %d, chunk_size: %d, \t\
          p1: %p, p2: %p, p3: %p, p4: %p\n", alloc_
  memset(p1, 'a', alloc_size);
  memset(p2, 'b', alloc_size);
  memset(p3, 'c', alloc_size);
  memset(p4, 'd', alloc_size);
  free(p2);
  free(p3);   // will collapse, put in unsorted_bin
}
```

gdb peda script (try yourself in gef)

```
def print_list(self, *arg):
    """
    Show single linked list
    Usage:
        MYNAME list_head_addr [fd_offset] [comma_separated_offsets_to_print]
    """
    (list_head_addr,fd_offset,extra_fields) = normalize_argv(arg, 3)
```

```
    node = list_head_addr
    while True:
        ptr = peda.read_int(node + fd_offset)
        _map = {}
        _map[fd_offset] = green( hex(ptr) )
        for field in extra_fields_int:
          _map[field] = blue(hex(peda.read_int(node + field)))
        _str = ['%d: %s' % (field, _map[field]) for field in sorted(_map)]
        _str = ", ".join(_str)
        msg("\t %s --> ( %s )" % (hex(node), _str))
        if ptr == node:
            break
        elif ptr == list_head_addr:
            break
        elif ptr == 0:
            break
        else:
            node = ptr
    return
```

- list header
- fd pointer offset
- other fields to print

Address of main_arena

- runtime libc base address

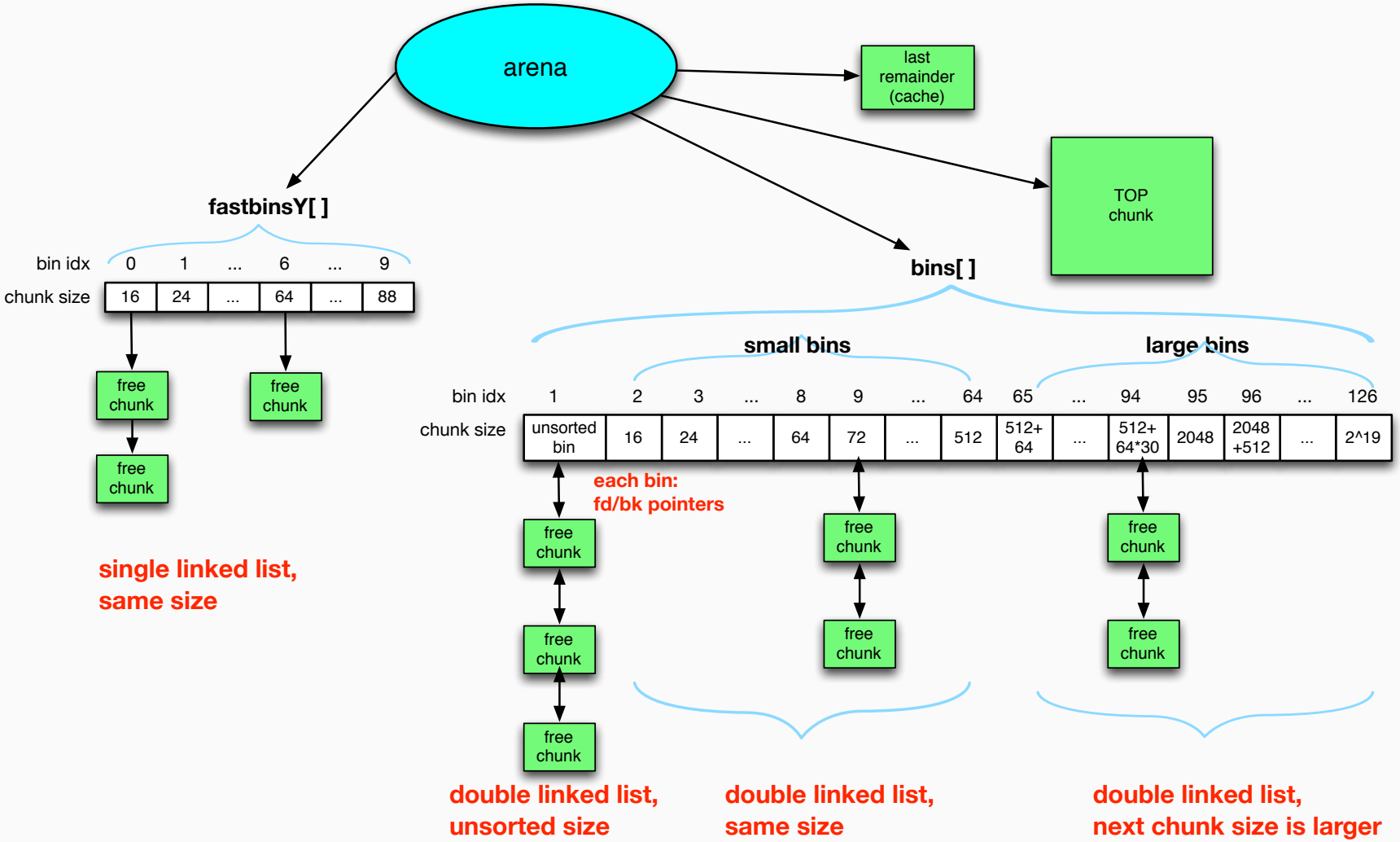
```
gdb-peda$ info proc map
process 2043
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/zhangchao/test/heap/fastbin
         0x8049000  0x804a000     0x1000        0x0 /home/zhangchao/test/heap/fastbin
         0x804a000  0x804b000     0x1000     0x1000 /home/zhangchao/test/heap/fastbin
        0xf7e15000 0xf7e16000     0x1000        0x0
        0xf7e16000 0xf7fcd000   0x1b7000        0x0 /usr/lib/libc-2.17.so
        0xf7fcd000 0xf7fce000     0x1000   0x1b7000 /usr/lib/libc-2.17.so
        0xf7fce000 0xf7fd0000     0x2000   0x1b7000 /usr/lib/libc-2.17.so
        0xf7fd0000 0xf7fd1000     0x1000   0x1b9000 /usr/lib/libc-2.17.so
```

- compile-time offset of main_arena

```
.data:001BA420 main_arena      dd ?                    ; DATA XREF: ptmalloc_lock_all:loc_732C7↑o
.data:001BA420                                         ; ptmalloc_unlock_all2:loc_733AE↑o ...
```

- runtime address of main_arena:
  - 0xf7e16000 + 0x001BA420= 0xf7fd0420

Address of main_arena (2)



- get it from the head/tail small_bin/large_bin chunk

Contents of main_arena

fastbinsY

```
gdb-peda$ x/40x 0xf7fd0420
0xf7fd0420 <main_arena>:     0x00000000     0x00000000     0x0804b040     0x0804b090
0xf7fd0430 <main_arena+16>:  0x0804b100     0x0804b190     0x0804b240     0x0804b310
0xf7fd0440 <main_arena+32>:  0x0804b400     0x00000000     0x00000000     0x00000000
0xf7fd0450 <main_arena+48>:  0x0804b8b8     0x0804b6f0     0x0804b798     0x0804b798
0xf7fd0460 <main_arena+64>:  0xf7fd0458     0xf7fd0458     0xf7fd0460     0xf7fd0460
0xf7fd0470 <main_arena+80>:  0xf7fd0468     0xf7fd0468     0xf7fd0470     0xf7fd0470
0xf7fd0480 <main_arena+96>:  0xf7fd0478     0xf7fd0478     0xf7fd0480     0xf7fd0480
0xf7fd0490 <main_arena+112>: 0x0804b518     0x0804b518     0x0804b5f8     0x0804b5f8
0xf7fd04a0 <main_arena+128>: 0x0804b6f0     0x0804b6f0     0xf7fd04a0     0xf7fd04a0
0xf7fd04b0 <main_arena+144>: 0xf7fd04a8     0xf7fd04a8     0xf7fd04b0     0xf7fd04b0
```

size fd

```
gdb-peda$ x/20x 0x0804b010
0x804b010:   0x00000000     0x00000011     0x00000000     0x62626262
0x804b020:   0x62626262     0x00000011     0x61616161     0x61616161
0x804b030:   0x61616161     0x00000011     0x64646464     0x00000000
0x804b040:   0x00000000     0x00000011     0x0804b010     0x63636363
0x804b050:   0x63636363     0x00000011     0x64646464     0x64646464
```

fastbinsY[0]

Contents of main_arena

```
gdb-peda$ x/40x 0xf7fd0420
0xf7fd0420 <main_arena>:     0x00000000     0x00000000     0x0804b040     0x0804b090
0xf7fd0430 <main_arena+16>:  0x0804b100     0x0804b190     0x0804b240     0x0804b310
0xf7fd0440 <main_arena+32>:  0x0804b400     0x00000000     0x00000000     0x00000000
0xf7fd0450 <main_arena+48>:  0x0804b8b8     0x0804b6f0     0x0804b798     0x0804b798
0xf7fd0460 <main_arena+64>:  0xf7fd0458     0xf7fd0458     0xf7fd0460     0xf7fd0460
0xf7fd0470 <main_arena+80>:  0xf7fd0468     0xf7fd0468     0xf7fd0470     0xf7fd0470
0xf7fd0480 <main_arena+96>:  0xf7fd0478     0xf7fd0478     0xf7fd0480     0xf7fd0480
0xf7fd0490 <main_arena+112>: 0x0804b518     0x0804b518     0x0804b5f8     0x0804b5f8
0xf7fd04a0 <main_arena+128>: 0x0804b6f0     0x0804b6f0     0xf7fd04a0     0xf7fd04a0
0xf7fd04b0 <main_arena+144>: 0xf7fd04a8     0xf7fd04a8     0xf7fd04b0     0xf7fd04b0
```

fastbins

TOP

unsorted bin

The last freed chunk is stored in unsorted_bin.

```
gdb-peda$ print_list 0x0804b798 8 0,4,12
        0x804b798 --> ( 0: 0x64646464, 4: 0xc1, 8: 0xf7fd0450, 12: 0xf7fd0450 )
        0xf7fd0450 --> ( 0: 0x804b8b8, 4: 0x804b6f0, 8: 0x804b798, 12: 0x804b798 )
```

Contents of main_arena

```
gdb-peda$ x/40x 0xf7fd0420
0xf7fd0420 <main_arena>:     0x00000000     0x00000000     0x0804b040     0x0804b090
0xf7fd0430 <main_arena+16>:  0x0804b100     0x0804b190     0x0804b240     0x0804b310
0xf7fd0440 <main_arena+32>:  0x0804b400     0x00000000     0x00000000     0x00000000
0xf7fd0450 <main_arena+48>:  0x0804b8b8     0x0804b6f0     0x0804b798     0x0804b798
0xf7fd0460 <main_arena+64>:  0xf7fd0458     0xf7fd0458     0xf7fd0460     0xf7fd0460
0xf7fd0470 <main_arena+80>:  0xf7fd0468     0xf7fd0468     0xf7fd0470     0xf7fd0470
0xf7fd0480 <main_arena+96>:  0xf7fd0478     0xf7fd0478     0xf7fd0480     0xf7fd0480
0xf7fd0490 <main_arena+112>: 0x0804b518     0x0804b518     0x0804b5f8     0x0804b5f8
0xf7fd04a0 <main_arena+128>: 0x0804b6f0     0x0804b6f0     0xf7fd04a0     0xf7fd04a0
0xf7fd04b0 <main_arena+144>: 0xf7fd04a8     0xf7fd04a8     0xf7fd04b0     0xf7fd04b0
```

fastbins

TOP

unsorted bin

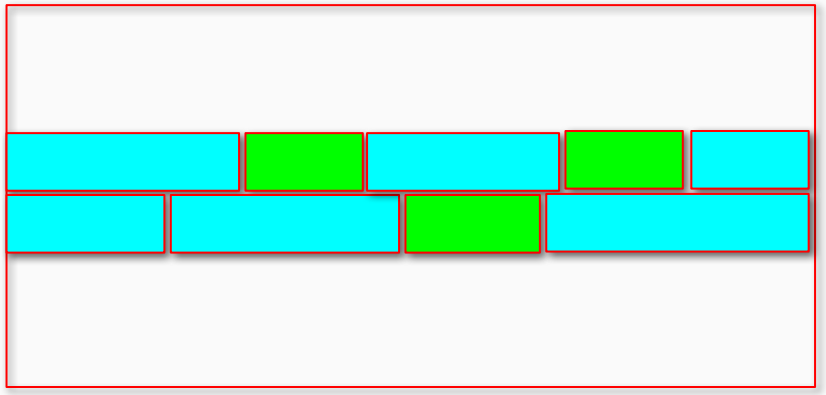
small bin size=72

```
gdb-peda$ print_list 0x0804b5f8 8 0,4,12
        0x804b5f8 --> ( 0: 0x61616161, 4: 0x49, 8: 0xf7fd0490, 12: 0xf7fd0490 )
        0xf7fd0490 --> ( 0: 0x804b518, 4: 0x804b518, 8: 0x804b5f8, 12: 0x804b5f8 )
```

In a Word

- free() and malloc() rely on metadata to
  - traverse bins, lists, chunks
  - update fd/bk/size/prev_size etc. of related chunks
  - return memory to users

- But these metadata are not well-protected
  - e.g., heap overflow, use-after-free



Project 2: Vulnerability Study

- Types of vulnerabilities not covered
  - integer overflow
  - uninitialized variables
  - race condition
    - time-of-check to time-of-use, TOC2TOU
- Common Weakness Enumeration
  - http://cwe.mitre.org/index.html
- Requirement:
  - finish it alone, write a report covering
  - study
    - understand root cause of the vulnerability, and how to exploit it
  - experiment
    - craft custom examples, exploit it

Project 1 will be assigned in this week.

? &#