Lecture 10 – Heap control data

Stephen Checkoway
Oberlin College

Today

We're going to look at a classic vulnerability in memory allocators: unlink

This vulnerability arises due to mixing program data and heap allocator metadata (and also memory unsafety)

The slides are once again looking at a 32-bit x86 implementation but the ideas extend beyond just a single memory allocator

Key idea for today: unlinking a node in a doubly linked list can lead to writing attacker-controlled data at attacker-controlled locations

Layout of program memory

- Heap is managed by malloc
  - Many different malloc implementations
  - glibc uses a modified version of Doug Lea's Malloc (dlmalloc)

- Responsibilities
  - Requesting pages of memory from the OS
  - Managing free *chunks* of memory
  - Allocating memory for the program

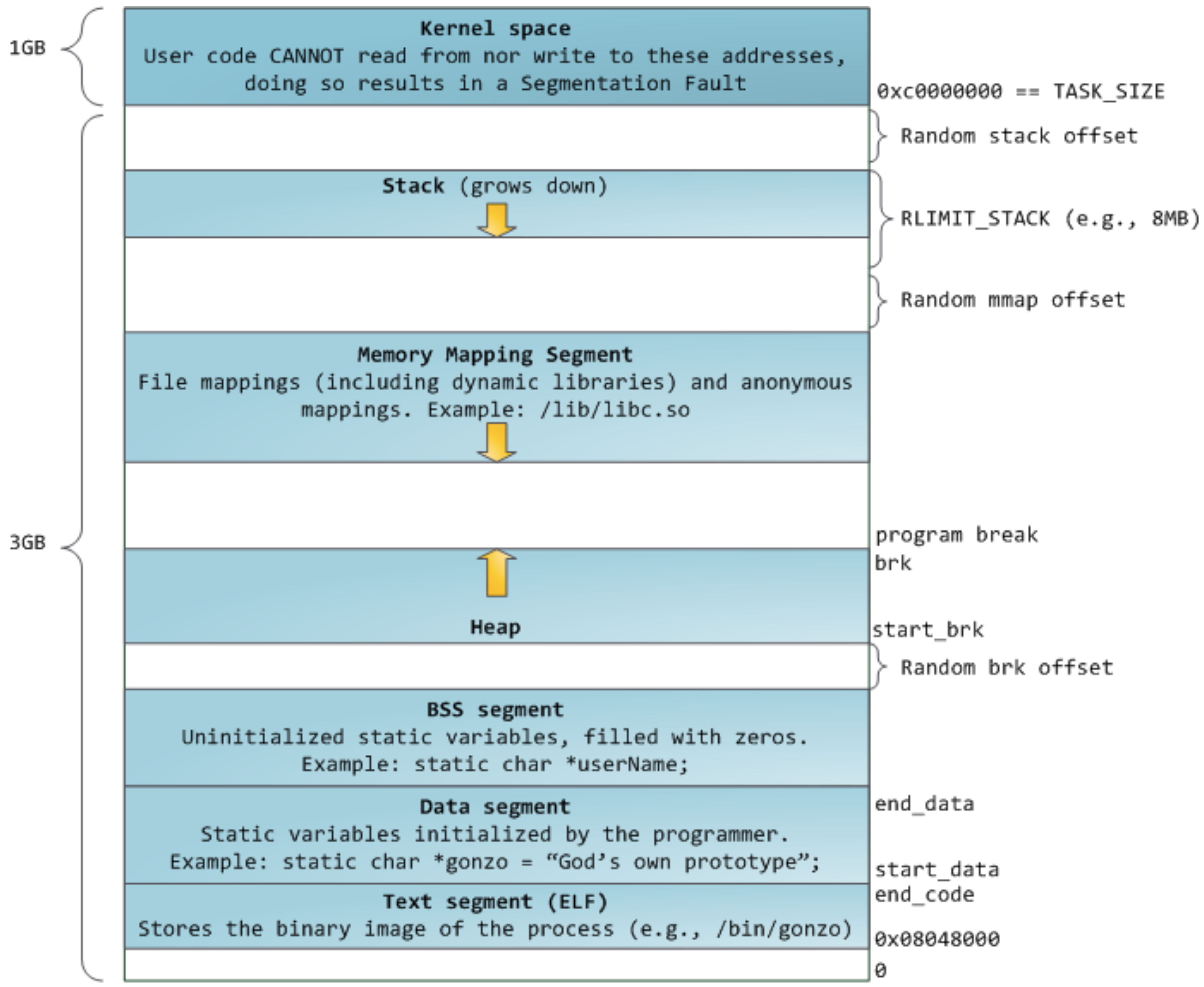


Image source: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

Memory allocation/deallocation

C programs allocate memory by calling void *malloc(size_t size)

- Memory is untyped; void * can be assigned to any pointer type
- E.g.,
  ```c
  struct foo *p = malloc(10 * sizeof *p);
  ```
  allocates memory sufficient to hold a 10 element array of struct foos
  ```c
  unsigned char *q = malloc(305);
  ```
  allocates memory to hold 305 bytes
- Memory is uninitialized (i.e., not set to 0) and holds whatever data was previously there
- Returned pointer is sufficiently aligned to hold any valid type (typically 16 byte or more aligned to hold SIMD vectors)

Memory allocated by `malloc()` is freed by calling

```c
void free(void *ptr)
```

Other allocation functions

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

- allocates `count * size` bytes and sets them all to 0

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

- Tries to resize `ptr` to hold `size` bytes
- If it cannot resize, it allocates `size` bytes and copies the data from `ptr` to the new allocation, and frees `ptr`
- If `ptr` is `NULL`, it acts like `malloc(size)`
- If `size` is 0, some systems treat it like `free(ptr)`, others like `malloc(0)` and return a non-`NULL` pointer to a 0-byte allocation
- C23 makes `realloc(ptr, 0)` undefined behavior for some reason!

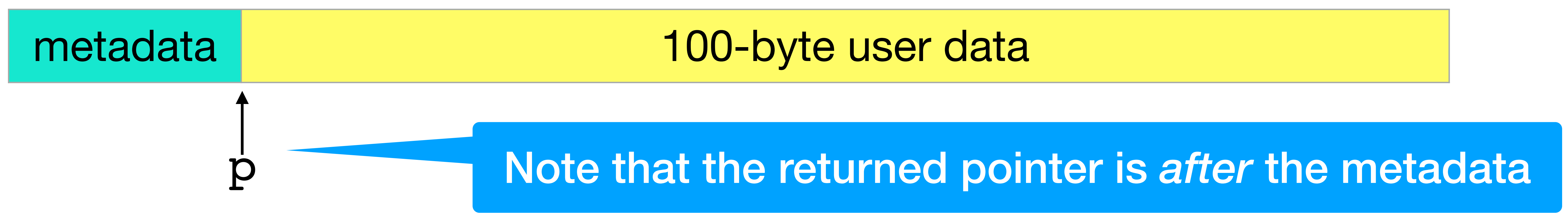
malloc implementation

We're going to look at how malloc (the allocator as a whole, not just the `malloc()` function) is implemented (one implementation anyway)

Main idea: OS hands largish regions of memory to malloc via `sbrk()`
[Who remembers `sbrk()` from 210?]

Malloc carves these regions up into chunks containing metadata for the allocator (such as the size of the allocation) as well as data for the application

Conceptually: void *p = malloc(100) allocates chunks that looks like this



free implementation

When a chunk is freed, it

1. gets merged with the surrounding chunks, if they are also free
2. gets put on a doubly linked list of free chunks for malloc to use for subsequent allocations

Before we look at the details

Why do you think the metadata is stored before the application data in the chunk?

Could the metadata be put somewhere else?

Chunks

- Basic unit of memory managed by malloc
- prev_size: size of the previous chunk in memory
- size: size of this chunk
  - lsb is 1 if the previous chunk is in use (PREV_IN_USE bit)
- fd: forward pointer in free list
- bk: backward pointer in free list

```
struct malloc_chunk {
    size_t prev_size;
    size_t size;
    struct malloc_chunk *fd;
    struct malloc_chunk *bk;
}
```

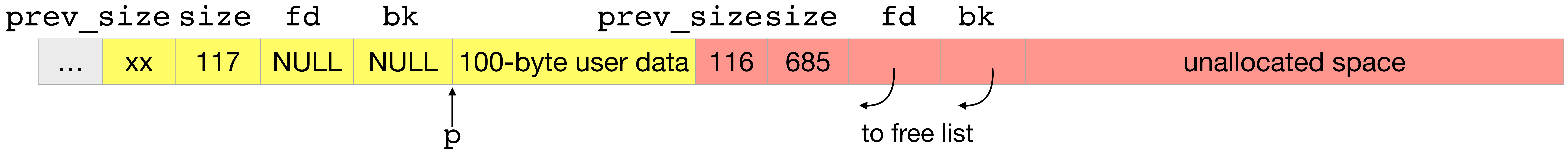
Free chunks/free lists

- A chunk can be allocated or free
- Free chunks are stored in doubly-linked lists using the fd and bk pointers
- prev_size refers to the size of the previous chunk adjacent to the current chunk, *not* the chunk pointed to by the bk pointer
- malloc maintains several different free lists for chunks of various sizes

Example (lie, truth shortly)

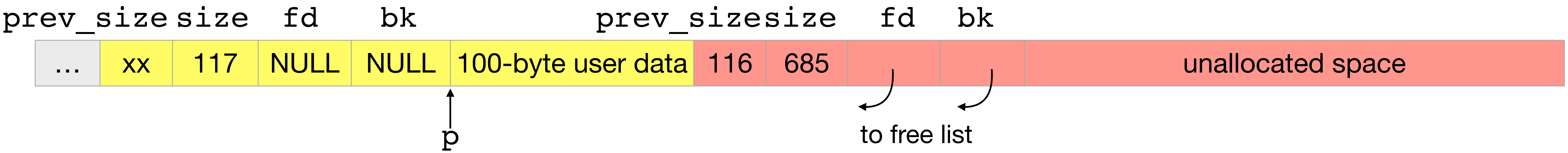
Example (lie, truth shortly)


```c
void *p = malloc(100);
```

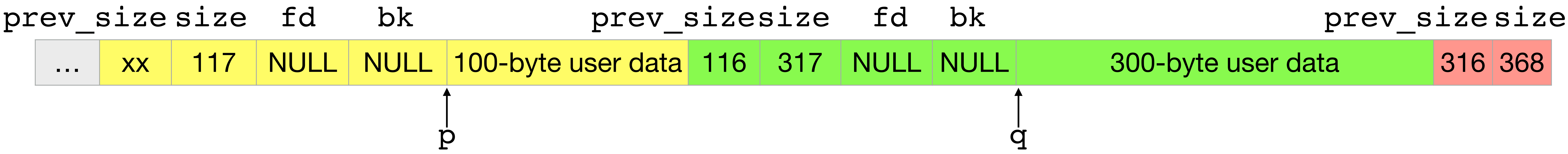


Example (lie, truth shortly)


```c
void *p = malloc(100);
```




```c
void *q = malloc(300);
```



Freeing chunks

- When freeing a chunk c, malloc looks at the chunk just before and the chunk just after c to see if they are free
- The adjacent free chunks are
  - removed from their free lists
  - combined with c to form a new, larger chunk c'
- c' (or c if neither neighbor were free) is added to a free list
- Malloc uses the prev_size and size fields plus some pointer arithmetic to find the preceding and following chunks
- Malloc uses the lsb of the size fields to determine if the previous chunks are in use or free

Optimization

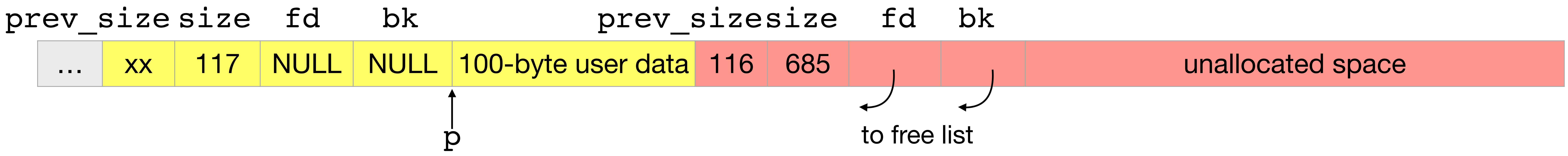
- fd and bk are only used when the chunk is free
- prev_size is only used when the previous chunk is free (to combine with the current chunk)
- Malloc saves space by overlapping these fields with user data

```
struct malloc_chunk {
    size_t prev_size;
    size_t size;
    struct malloc_chunk *fd;
    struct malloc_chunk *bk;
}
```

Optimization

- fd and bk are only used when the chunk is free
- prev_size is only used when the previous chunk is free (to combine with the current chunk)
- Malloc saves space by overlapping these fields with user data

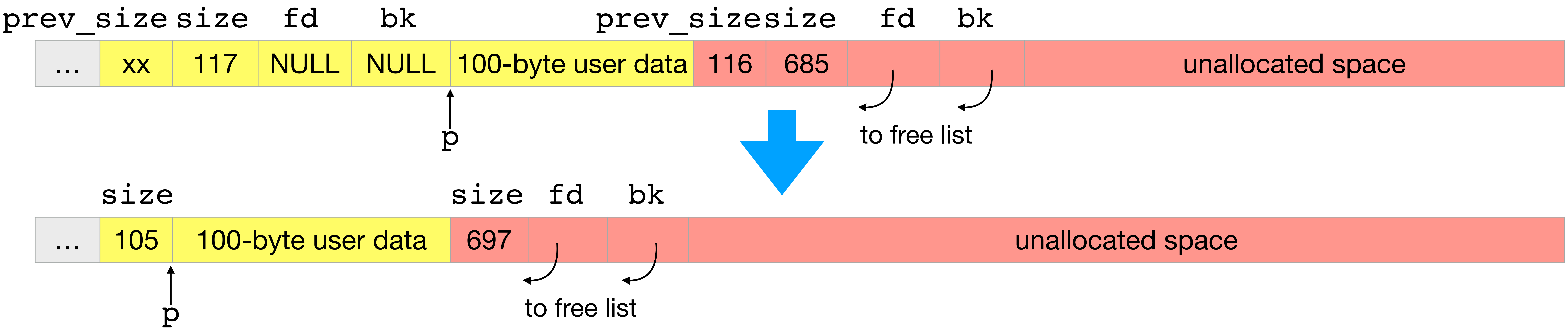
```
struct malloc_chunk {
    size_t prev_size;
    size_t size;
    struct malloc_chunk *fd;
    struct malloc_chunk *bk;
}
```



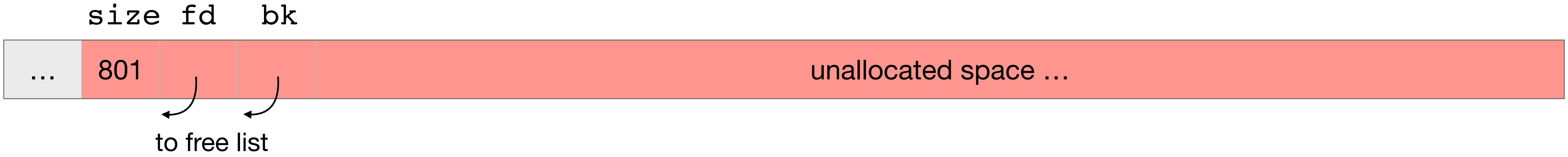
Optimization

- fd and bk are only used when the chunk is free
- prev_size is only used when the previous chunk is free (to combine with the current chunk)
- Malloc saves space by overlapping these fields with user data

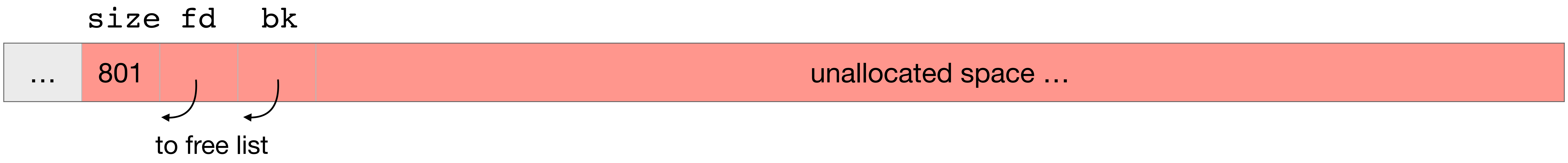
```
struct malloc_chunk {
    size_t prev_size;
    size_t size;
    struct malloc_chunk *fd;
    struct malloc_chunk *bk;
}
```



Example (truth but not to scale)

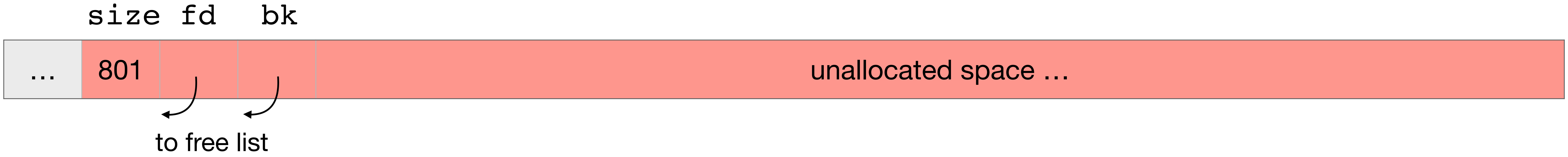


Example (truth but not to scale)

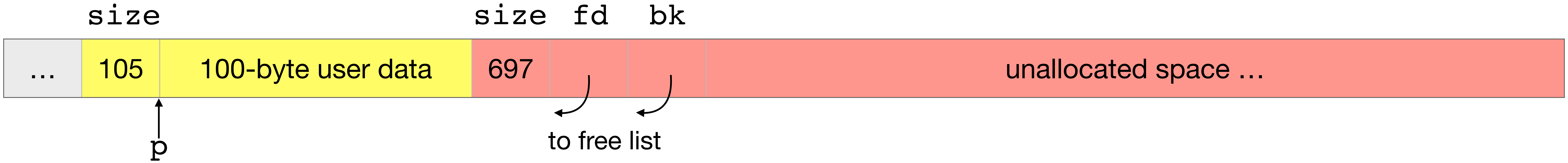



```
void *p = malloc(100);
```

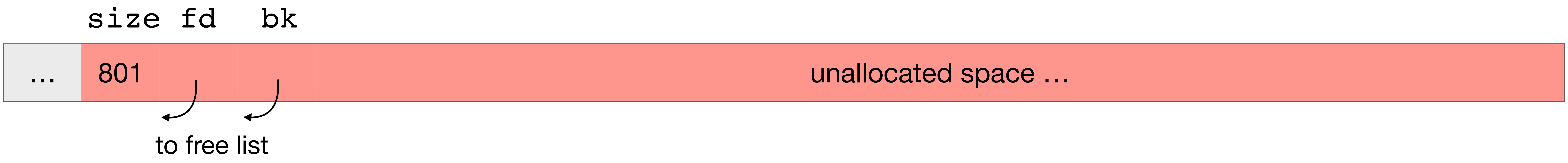
Example (truth but not to scale)



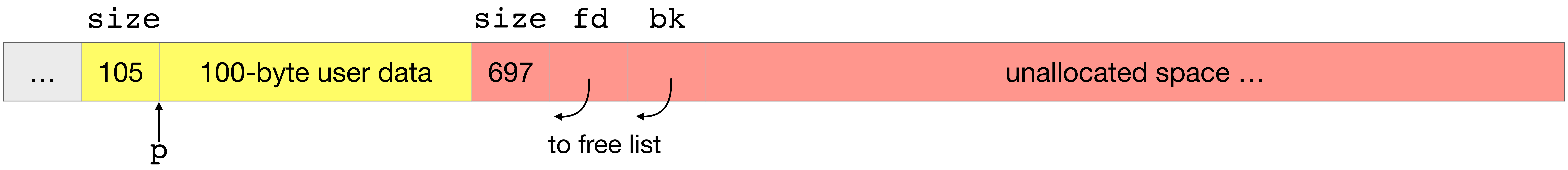

```
void *p = malloc(100);
```



Example (truth but not to scale)

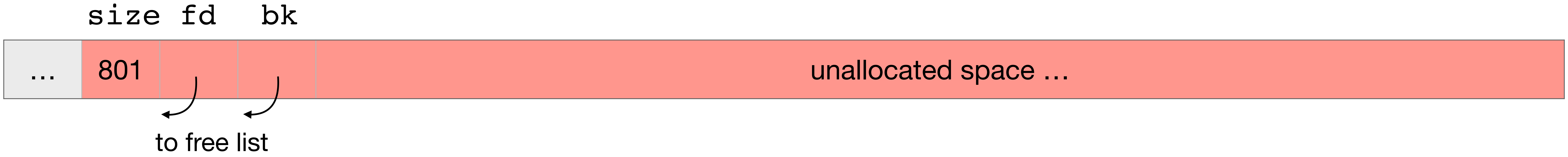



```
void *p = malloc(100);
```

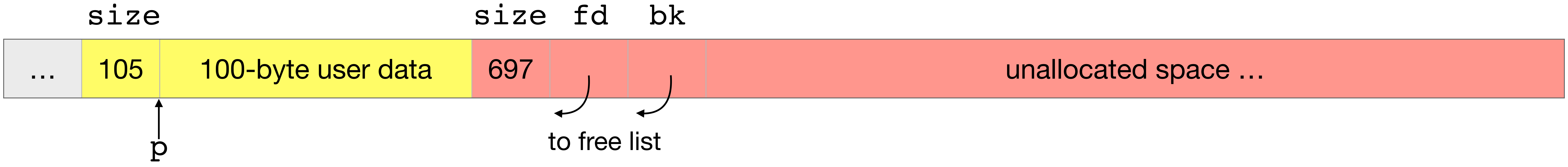



```
void *q = malloc(300);
```

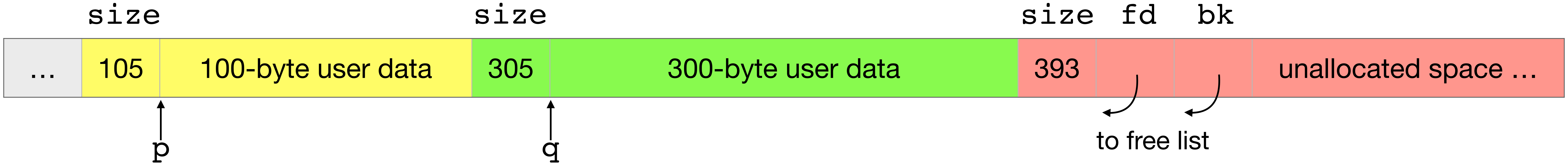
Example (truth but not to scale)



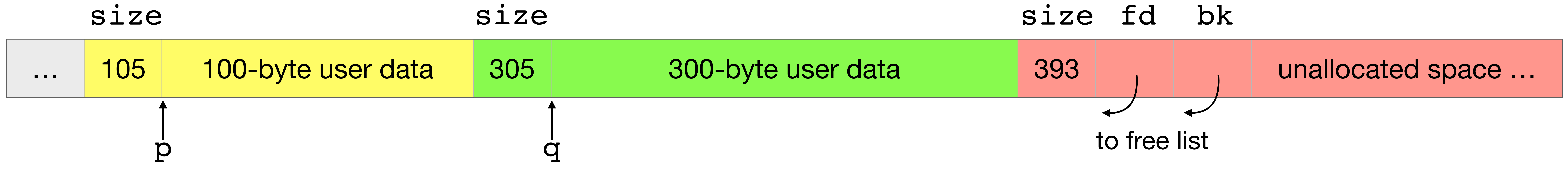

```c
void *p = malloc(100);
```



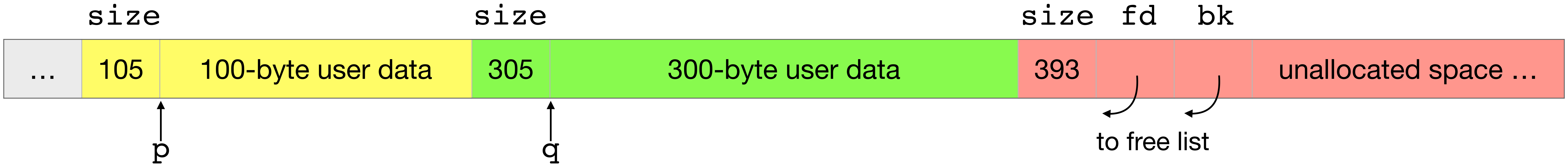

```c
void *q = malloc(300);
```



Example continued

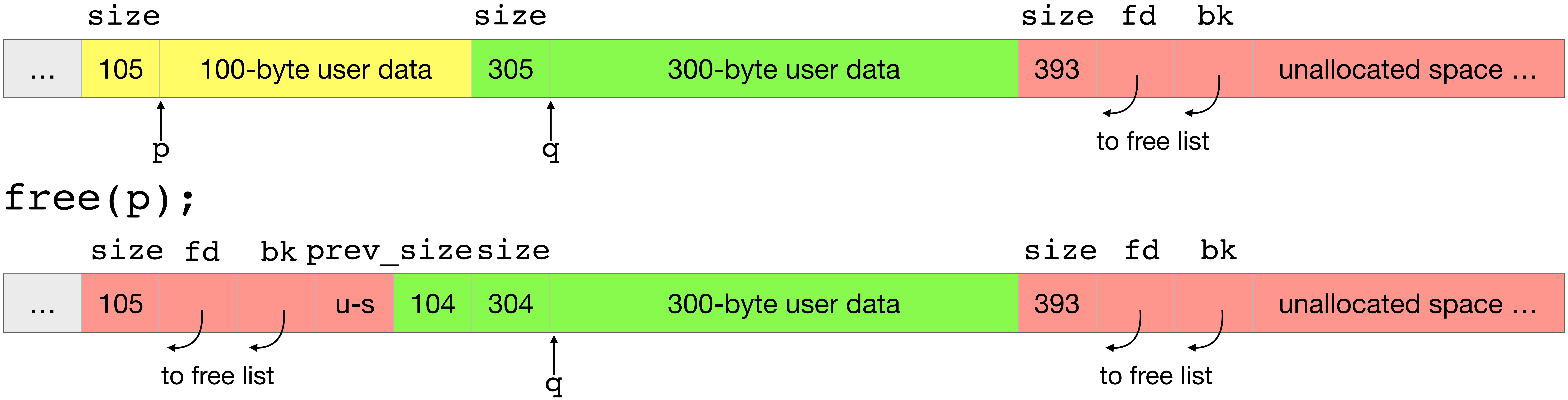


Example continued

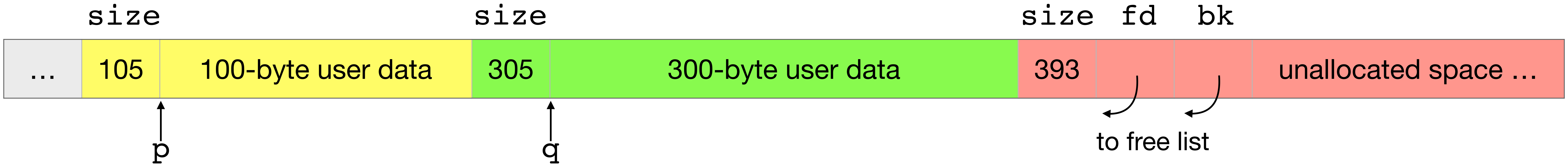



```
free(p);
```

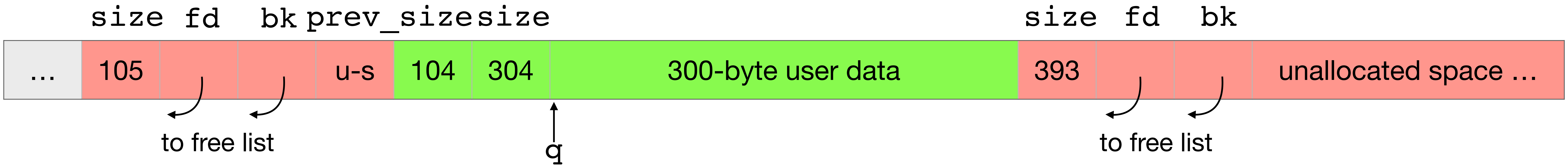
Example continued



Example continued

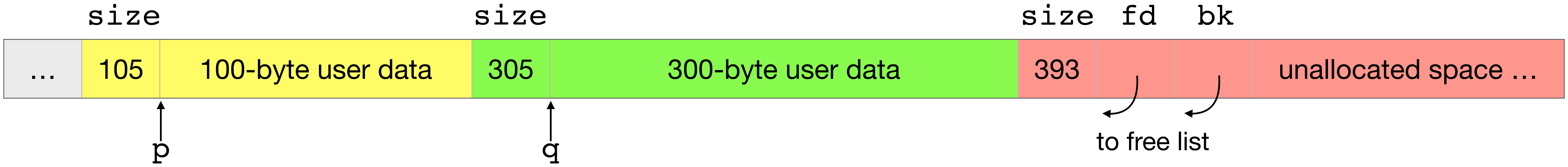



```
free(p);
```

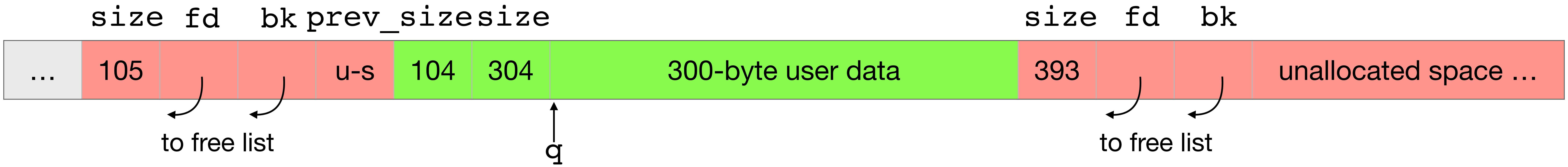



```
void *r = malloc(252);
```

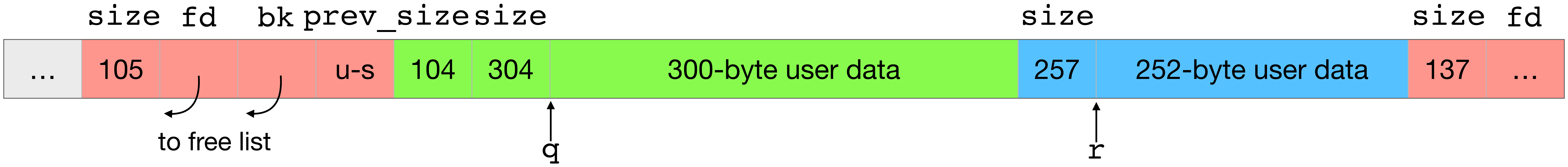
Example continued



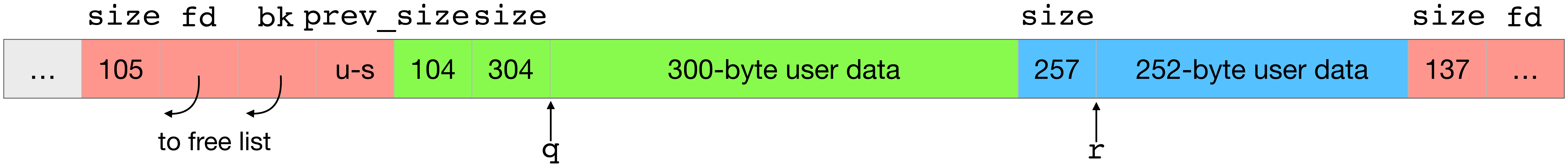

```
free(p);
```



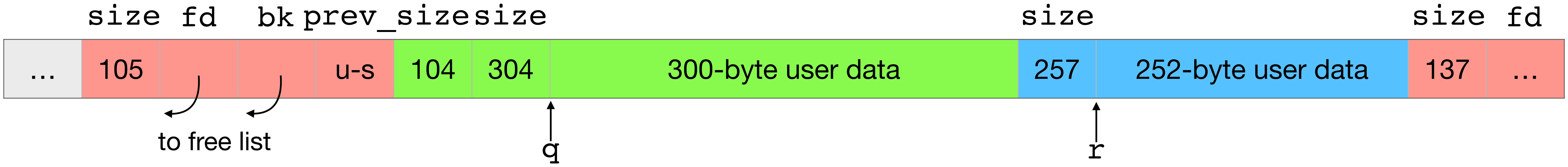

```
void *r = malloc(252);
```



Example continued

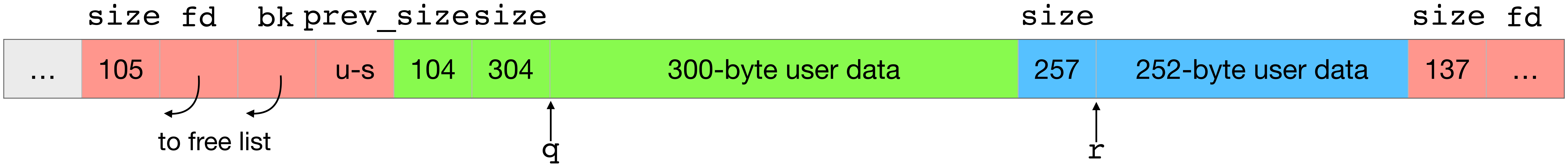


Example continued

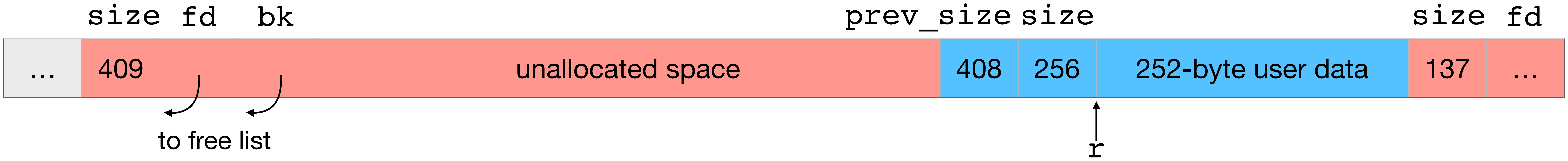



```
free(q);
```

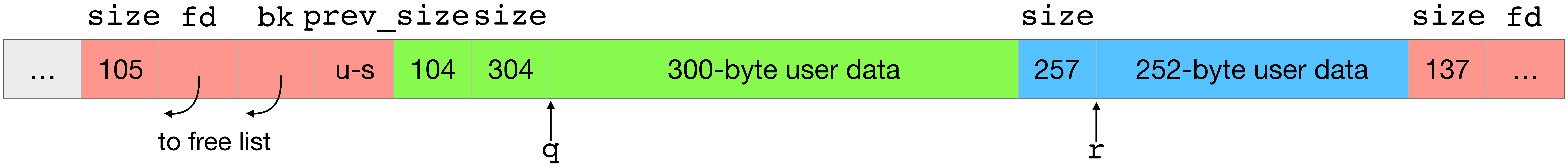
Example continued



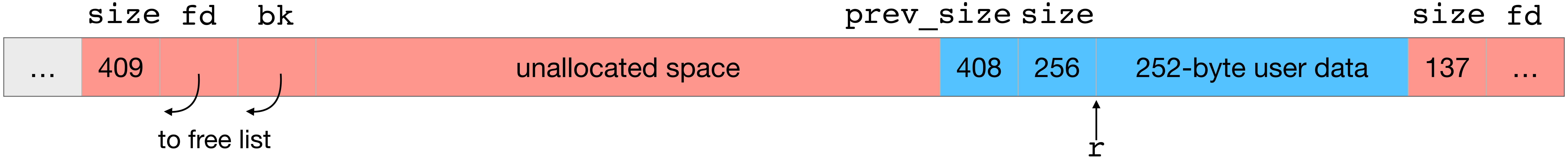
free(q);



Example continued

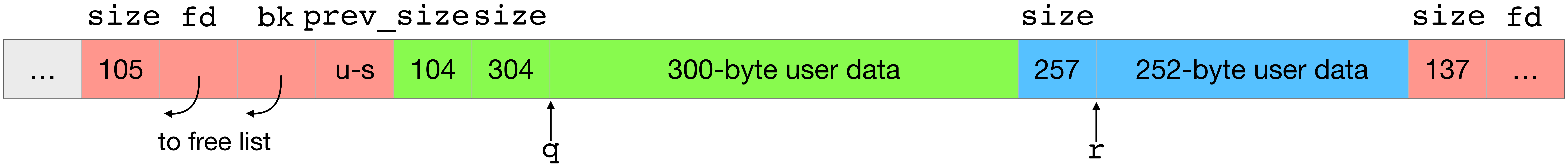



```
free(q);
```

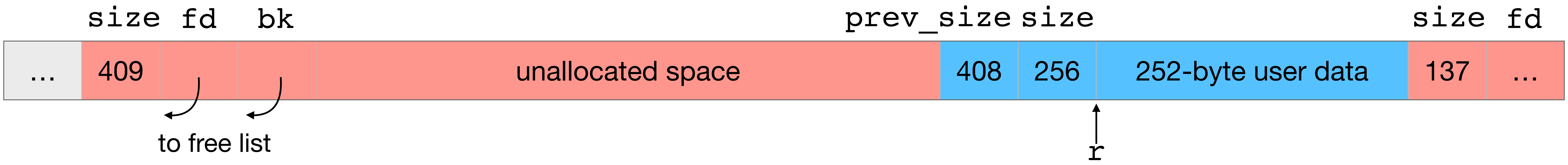



```
free(r);
```

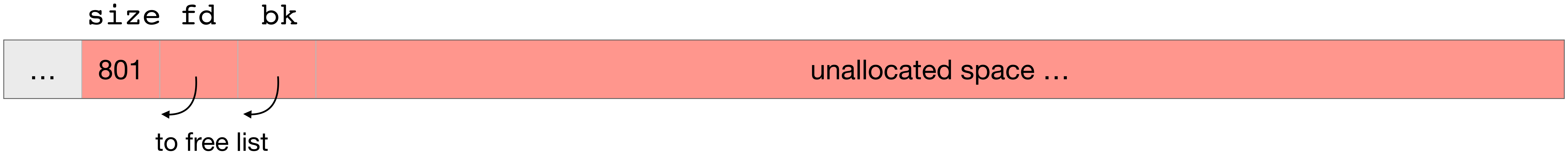
Example continued




```
free(q);
```




```
free(r);
```

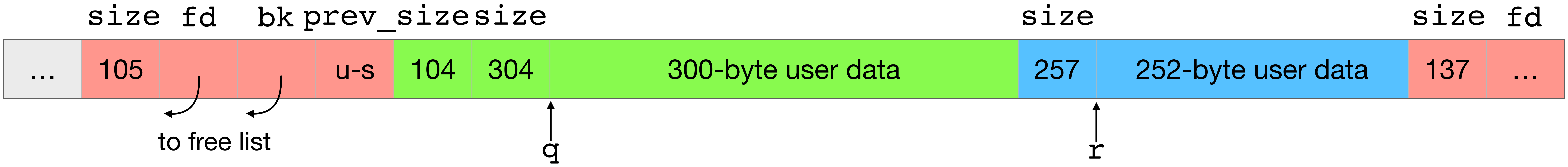


Recap

A chunk's metadata (prev_size, fd, and bk) overlaps the space for user data

prev_size is only valid if size & 1 == 0, i.e., if the previous chunk is free

fd, and bk are only valid if the chunk itself is free (i.e., if the next chunk's size field has a least significant bit of 1)



Removing chunks from free lists

- Chunks are removed using the unlink macro
- P is the chunk to unlink
- BK and FD are temporaries

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

Removing chunks from free lists

- Chunks are removed using the unlink macro
- P is the chunk to unlink
- BK and FD are temporaries

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

Removing chunks from free lists

- Chunks are removed using the unlink macro
- P is the chunk to unlink
- BK and FD are temporaries

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

Removing chunks from free lists

- Chunks are removed using the unlink macro
- P is the chunk to unlink
- BK and FD are temporaries

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

Removing chunks from free lists

- Chunks are removed using the unlink macro
- P is the chunk to unlink
- BK and FD are temporaries

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

Overwriting heap metadata

- The chunk metadata is inline (meaning the user data and the metadata are side-by-side)
- We can modify the metadata with a buffer overflow on the heap
- Consider

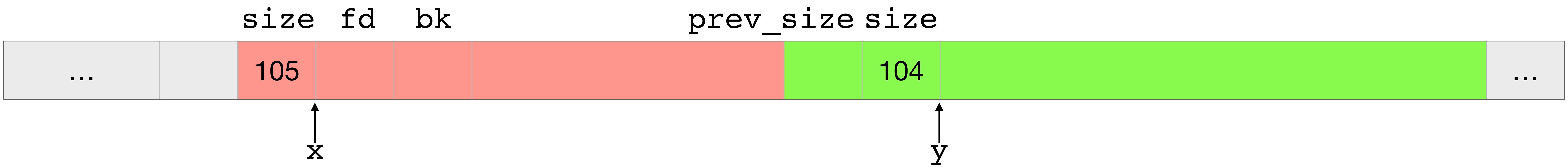
```
char *x = malloc(100);
void *y = malloc(100);
strcpy(x, attacker_controlled);
free(y);
```

- We can overflow x and overwrite y's metadata

Attacking malloc

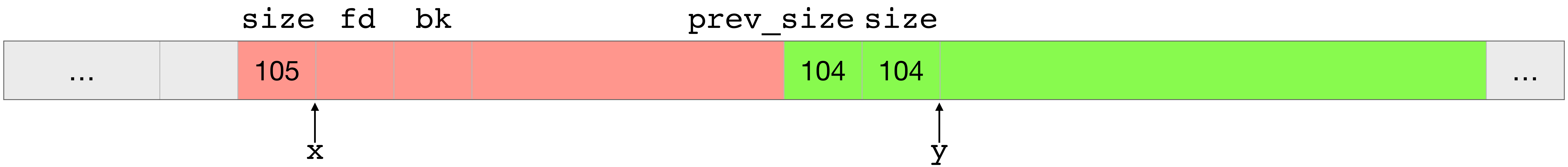
- When free(y) is called, it will examine x's chunk to see if it is free.
- If x's chunk is free, then unlink will be called on it to remove it from its free list
- We can carefully structure the attacker-controlled data to
  - convince free that x's chunk is free (how do we do this?)
  - convince the unlink macro to overwrite a saved instruction pointer on the stack by setting x's chunk's fd and bk pointers
  - inject shellcode
- When the function returns, our shellcode runs!

Attacking malloc



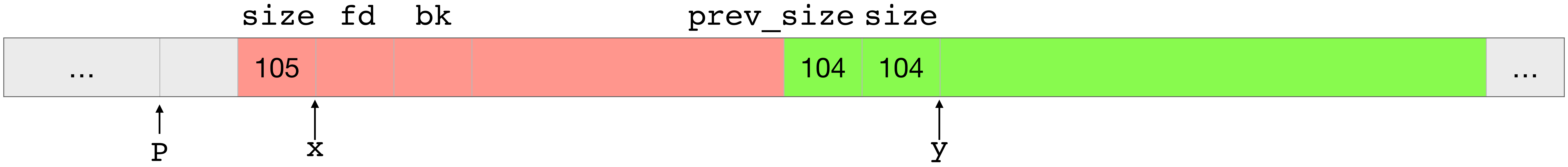
1. Change y's chunk's size from 105 to 104 (clears the PREV_IN_USE bit); y's chunk's prev_size and x's chunk's fd and bk are now used

Attacking malloc



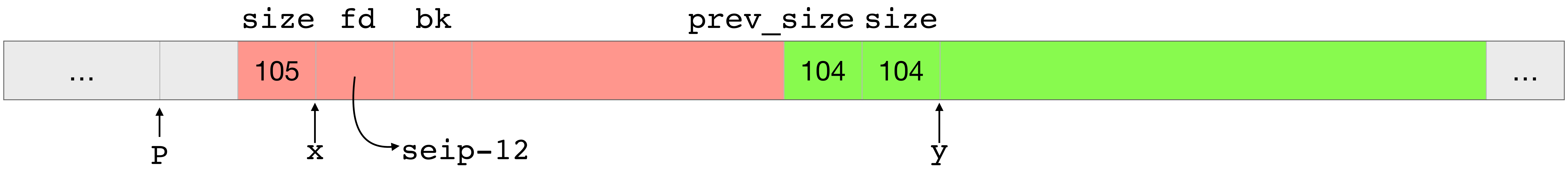
1. Change y’s chunk’s size from 105 to 104 (clears the PREV_IN_USE bit); y’s chunk’s prev_size and x’s chunk’s fd and bk are now used
2. Set y’s chunk’s prev_size to 104 so free looks back 104 bytes to find the start of the chunk to unlink

Attacking malloc



1. Change y's chunk's size from 105 to 104 (clears the PREV_IN_USE bit); y's chunk's prev_size and x's chunk's fd and bk are now used
2. Set y's chunk's prev_size to 104 so free looks back 104 bytes to find the start of the chunk to unlink
3. P in the unlink macro is x's chunk, so its fd and bk pointers need to be valid

Attacking malloc

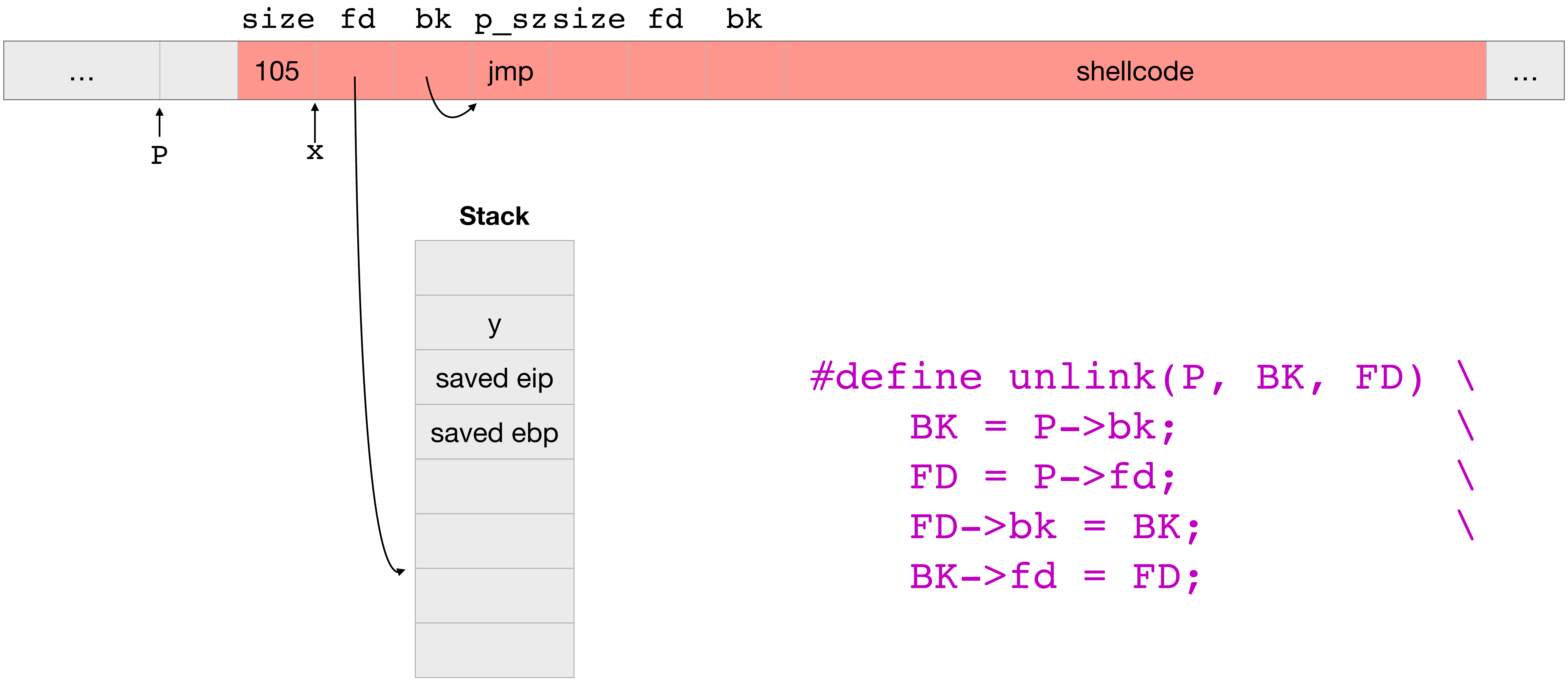


1. Change y’s chunk’s size from 105 to 104 (clears the PREV_IN_USE bit); y’s chunk’s prev_size and x’s chunk’s fd and bk are now used
2. Set y’s chunk’s prev_size to 104 so free looks back 104 bytes to find the start of the chunk to unlink
3. P in the unlink macro is x’s chunk, so its fd and bk pointers need to be valid
4. Point P->fd to saved eip (seip) - 12 [because the fields are 32-bits!]

Attacking malloc

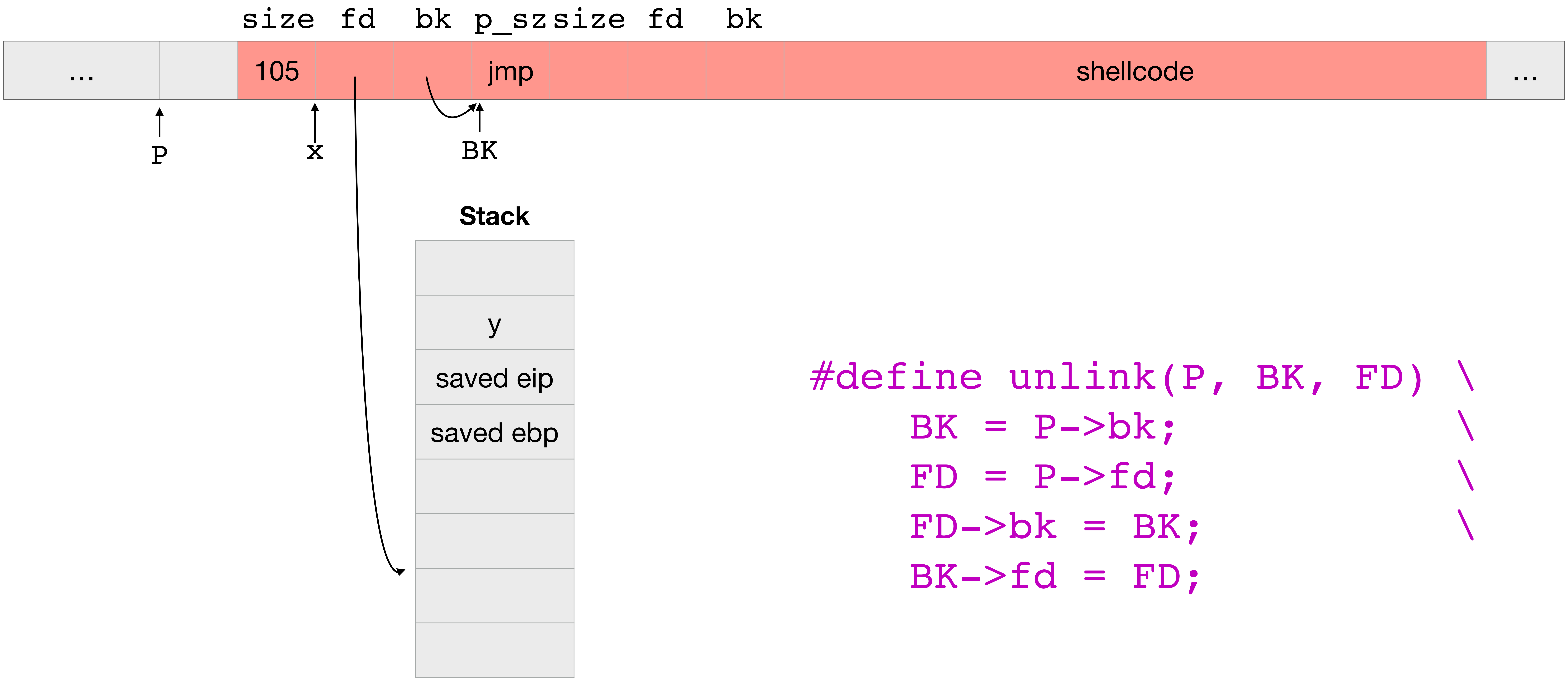
1. Change y's chunk's size from 105 to 104 (clears the PREV_IN_USE bit); y's chunk's prev_size and x's chunk's fd and bk are now used
2. Set y's chunk's prev_size to 104 so free looks back 104 bytes to find the start of the chunk to unlink
3. P in the unlink macro is x's chunk, so its fd and bk pointers need to be valid
4. Point P->fd to saved eip (seip) - 12 [because the fields are 32-bits!]
5. Point P->bk to a short jump to shellcode

Unlinking P (zooming in on P)



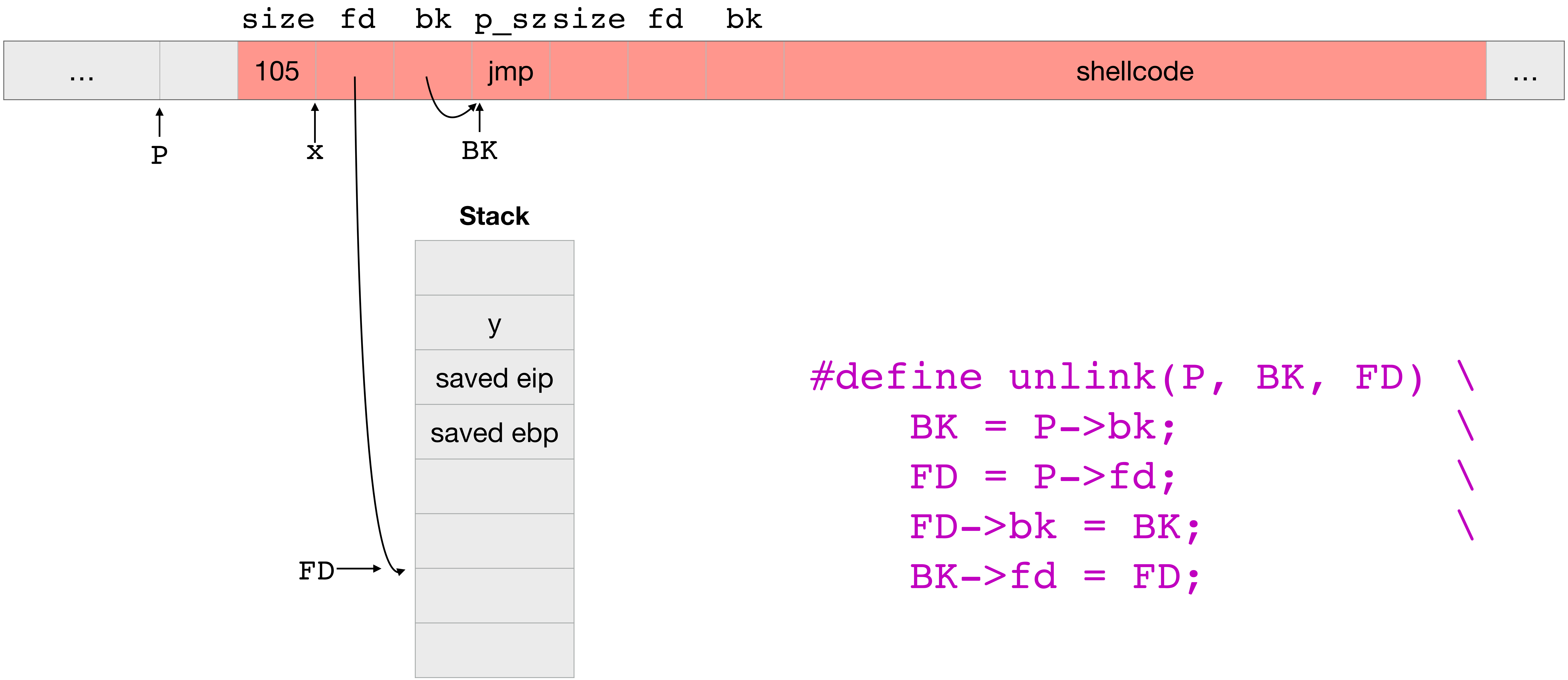

```
#define unlink(P, BK, FD) \
    BK = P->bk;           \
    FD = P->fd;           \
    FD->bk = BK;          \
    BK->fd = FD;
```

Unlinking P (zooming in on P)



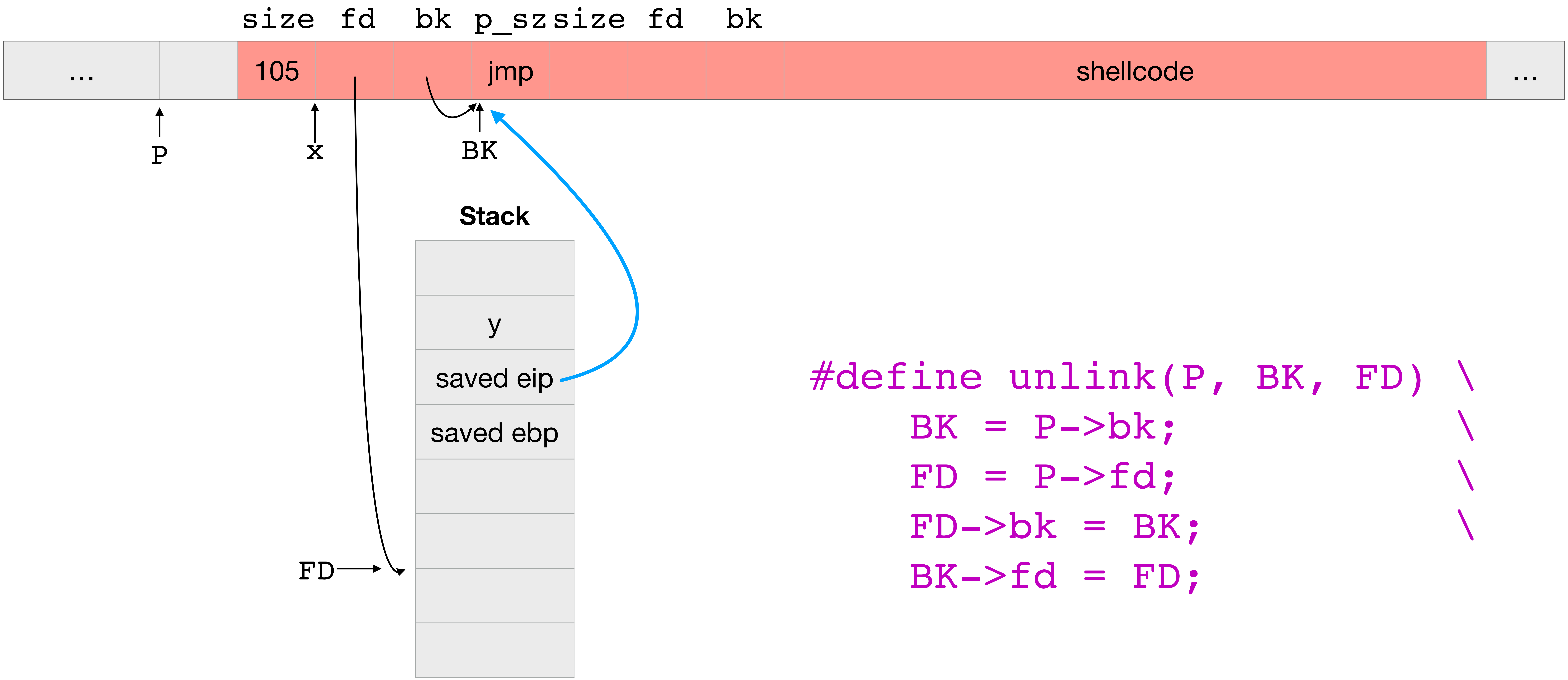

```
#define unlink(P, BK, FD) \
    BK = P->bk;           \
    FD = P->fd;           \
    FD->bk = BK;          \
    BK->fd = FD;
```

Unlinking P (zooming in on P)



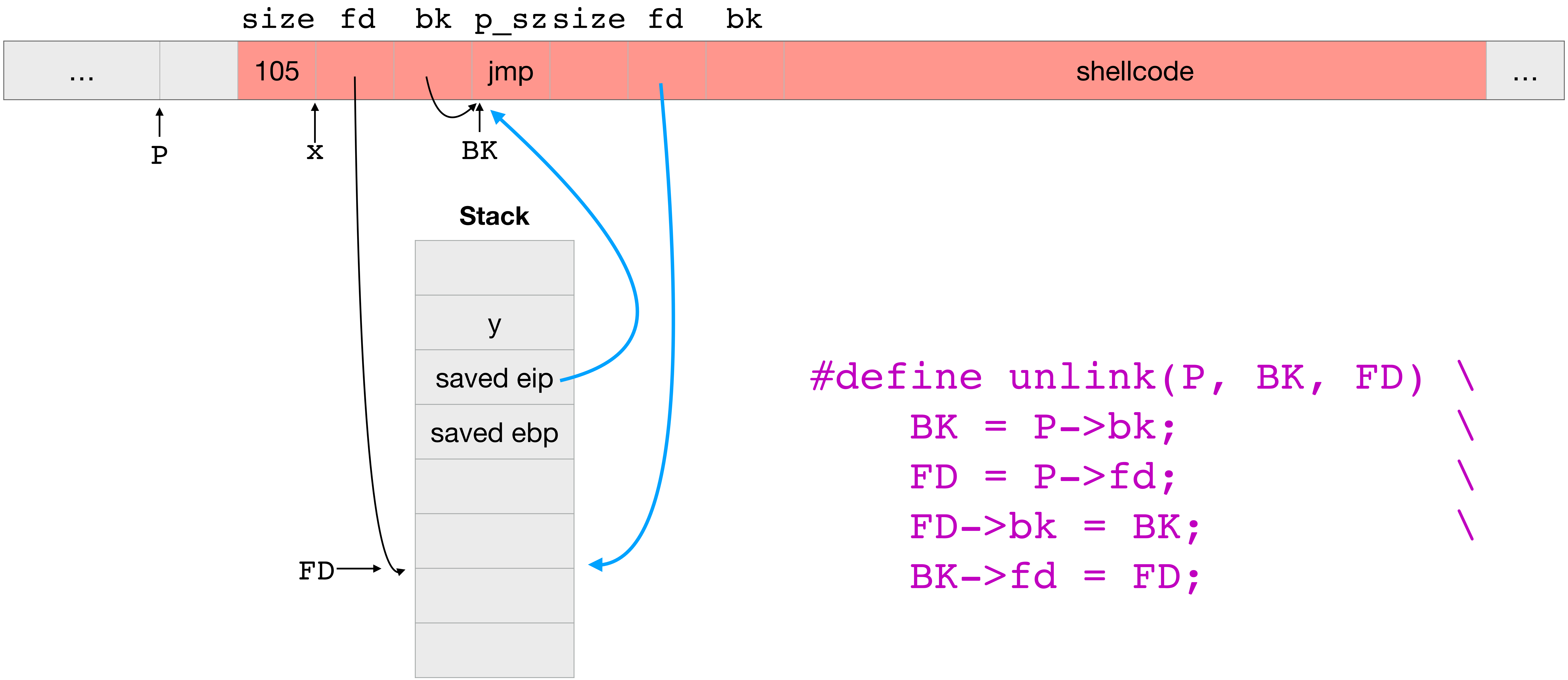

```
#define unlink(P, BK, FD) \
    BK = P->bk;           \
    FD = P->fd;           \
    FD->bk = BK;          \
    BK->fd = FD;
```

Unlinking P (zooming in on P)



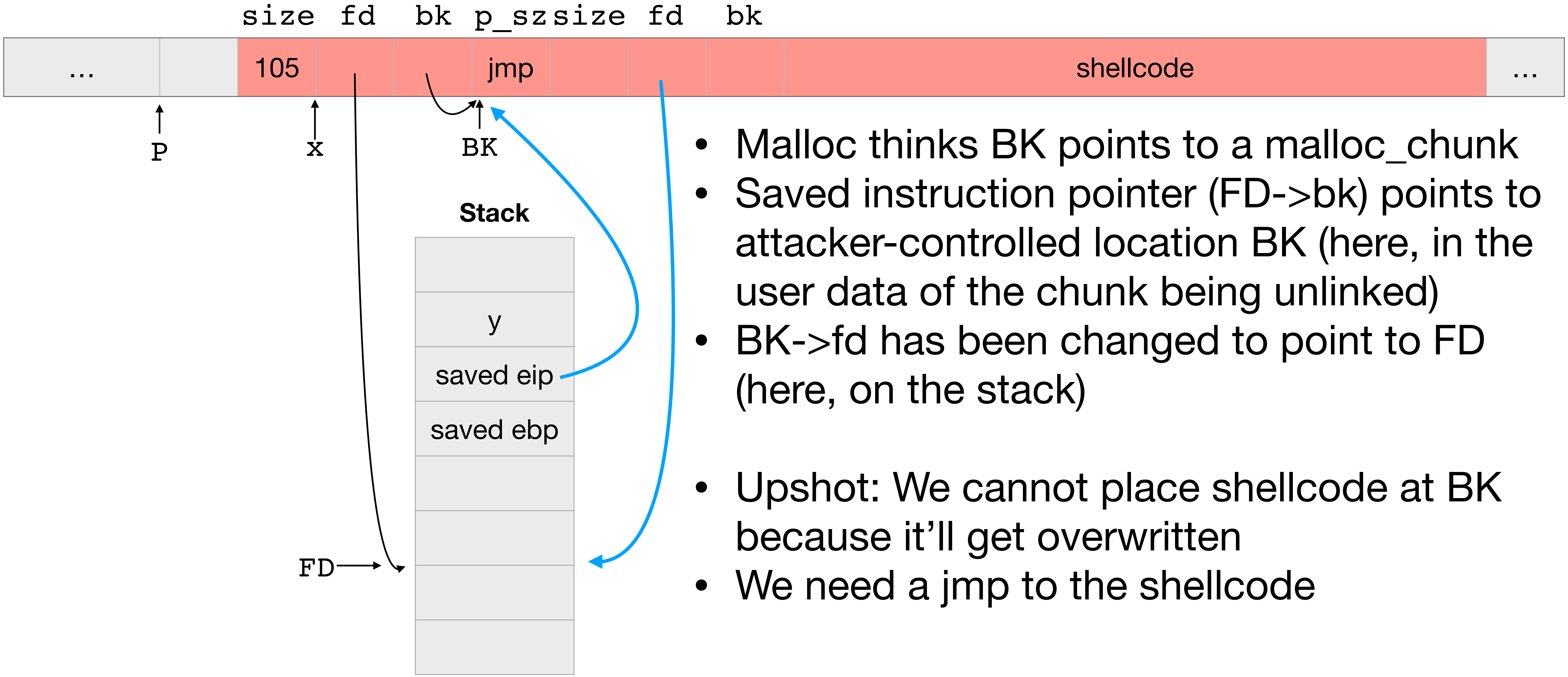

```
#define unlink(P, BK, FD) \
    BK = P->bk;           \
    FD = P->fd;           \
    FD->bk = BK;          \
    BK->fd = FD;
```

Unlinking P (zooming in on P)




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

After unlinking



- Malloc thinks BK points to a malloc_chunk
- Saved instruction pointer (FD->bk) points to attacker-controlled location BK (here, in the user data of the chunk being unlinked)
- BK->fd has been changed to point to FD (here, on the stack)

- Upshot: We cannot place shellcode at BK because it'll get overwritten
- We need a jmp to the shellcode

jmp to shellcode

JMP — Jump

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
EB *cb*	JMP *rel8*	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits

There are many jmp instructions, this is the one we want

Its encoding is 0xEB ⟨offset⟩ where offset is a 1-byte offset from the instruction *following* the jmp

Need to jump over the value written by UNLINK() at BK->fd

One small hitch for project 1

The slides show putting the shellcode in the heap

The heap is nonexecutable!

You’ll need to find some other place to put the shellcode that is executable

- There are several reasonable choices to do this
- Remember, you control how the target is run