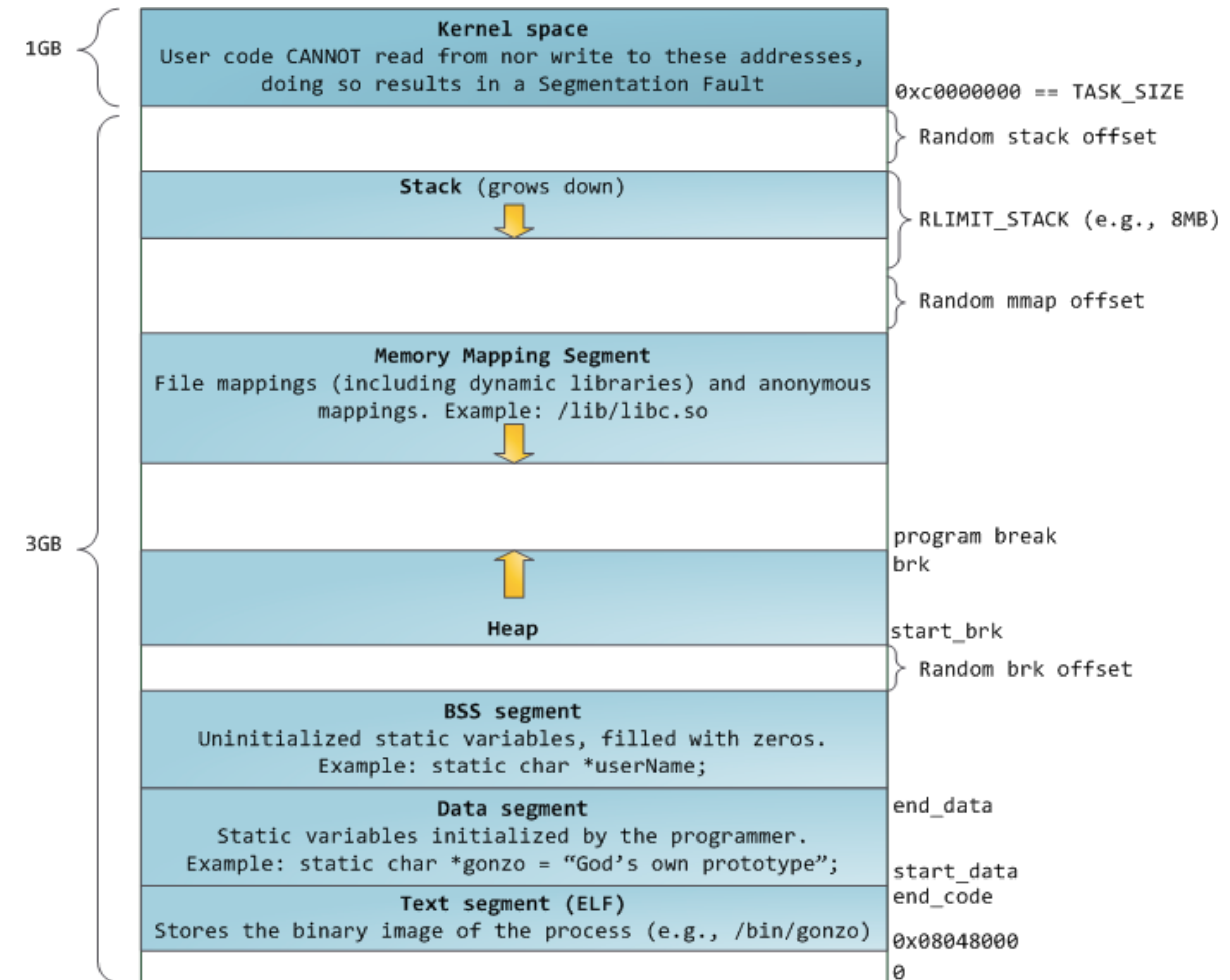


# Lecture 10 – Heap control data

Stephen Checkoway  
Oberlin College

# 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



# 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

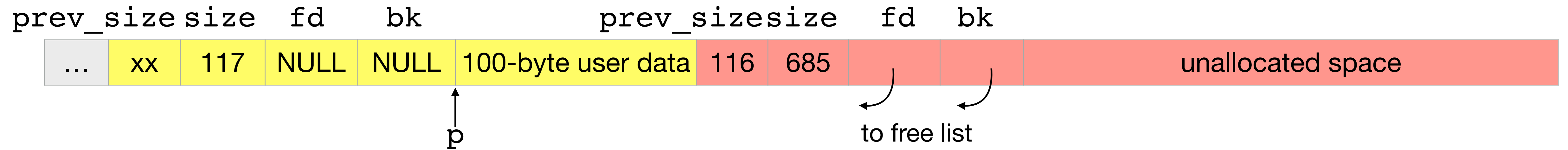
# Example (lie, truth shortly)



# Example (lie, truth shortly)



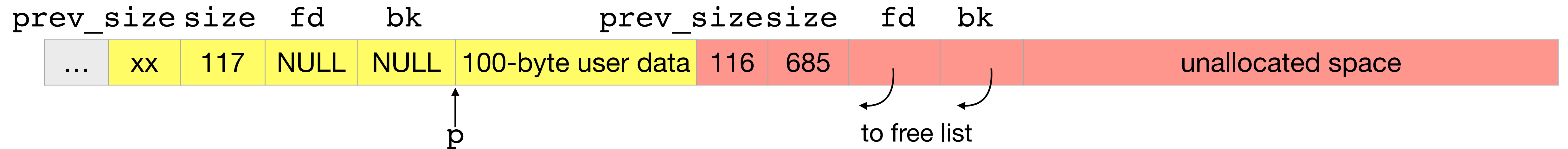
```
void *p = malloc(100);
```



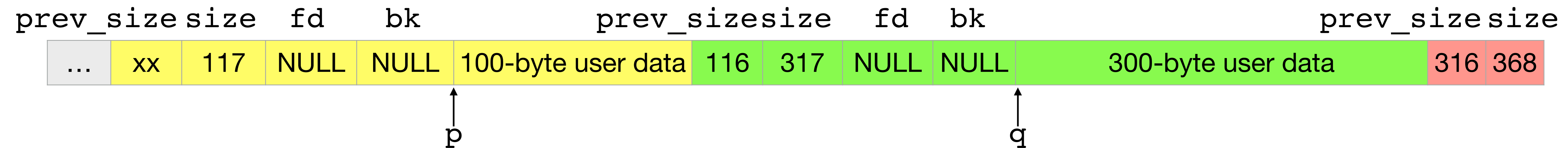
# Example (lie, truth shortly)



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



```
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

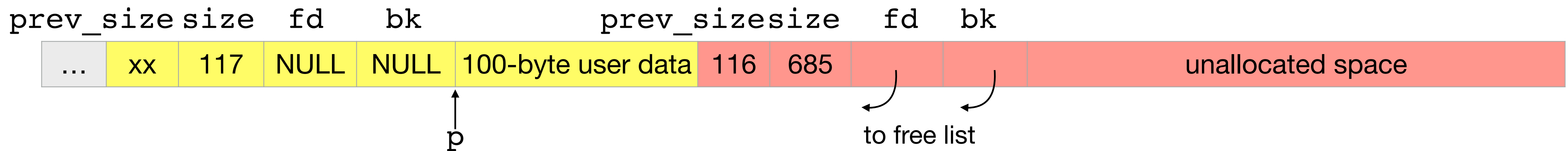
- fd and bk are only used when the chunk is free
- prev\_size is only used when the previous chunk is free (to combine with the current chunk)
- Malloc saves space by overlapping these fields with user data

```
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

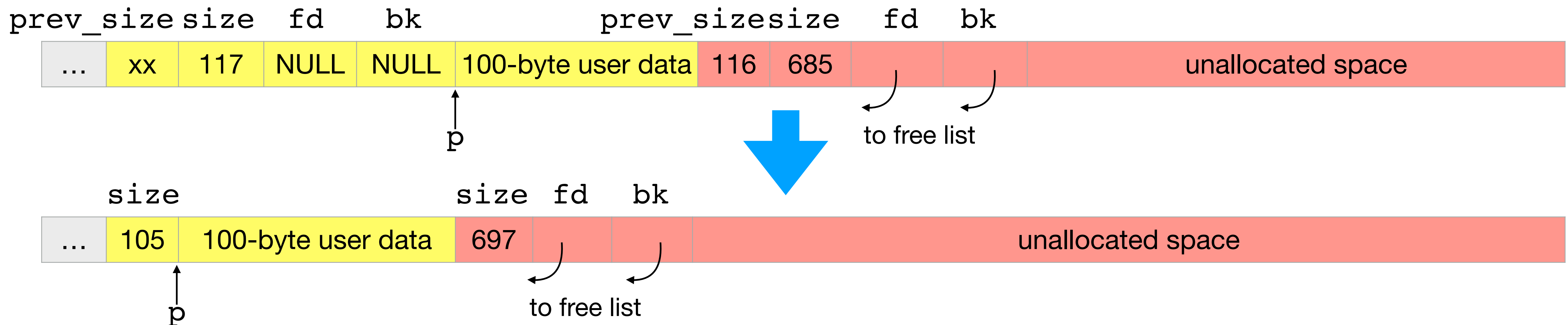
```
struct malloc_chunk {  
    size_t prev_size;  
    size_t size;  
    struct malloc_chunk *fd;  
    struct malloc_chunk *bk;  
}
```



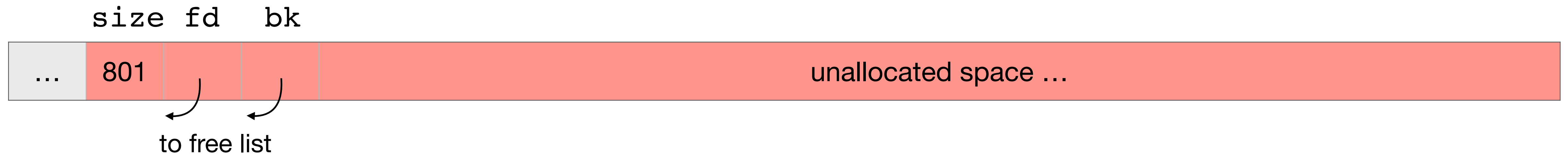
# Optimization

- 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

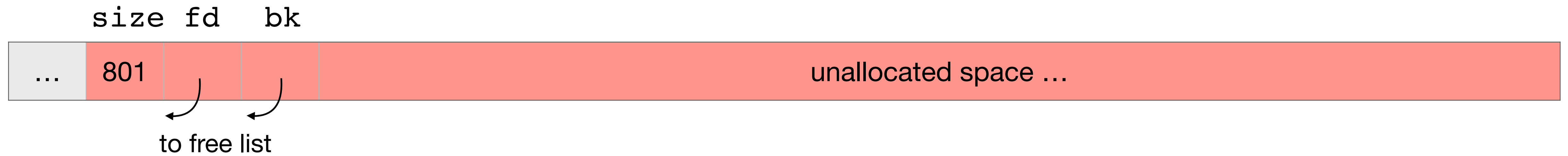
```
struct malloc_chunk {  
    size_t prev_size;  
    size_t size;  
    struct malloc_chunk *fd;  
    struct malloc_chunk *bk;  
}
```



# Example (truth but not to scale)

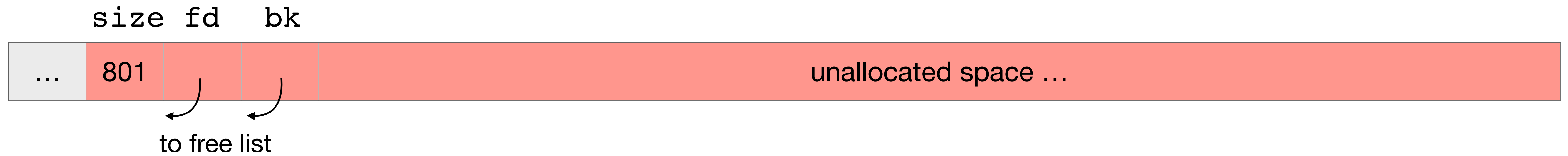


# Example (truth but not to scale)

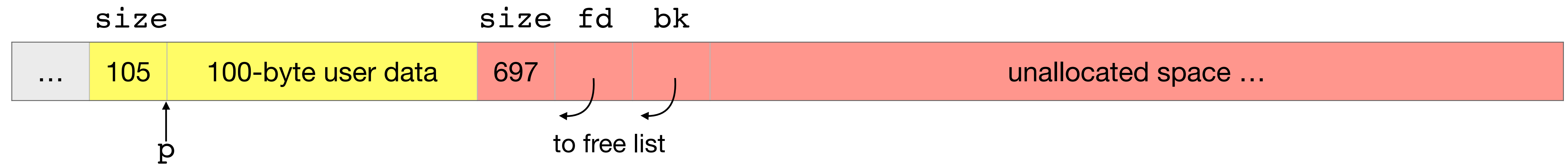


```
void *p = malloc(100);
```

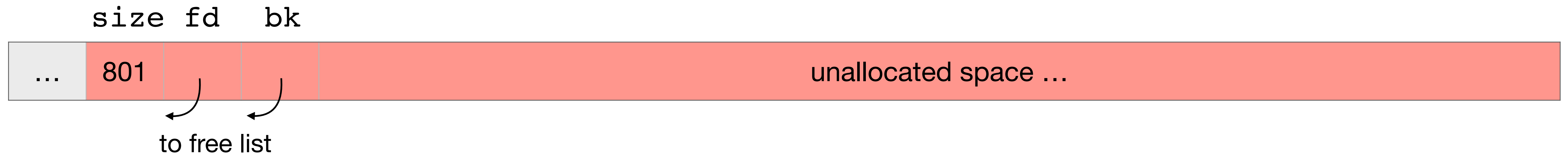
# Example (truth but not to scale)



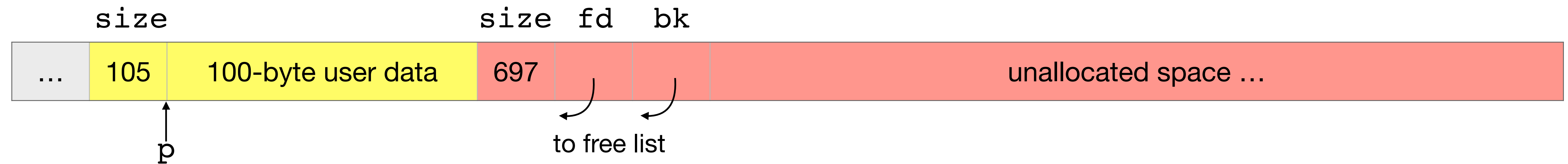
```
void *p = malloc(100);
```



# Example (truth but not to scale)

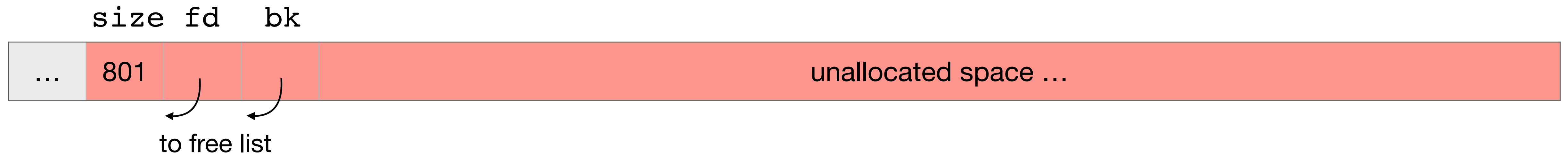


```
void *p = malloc(100);
```

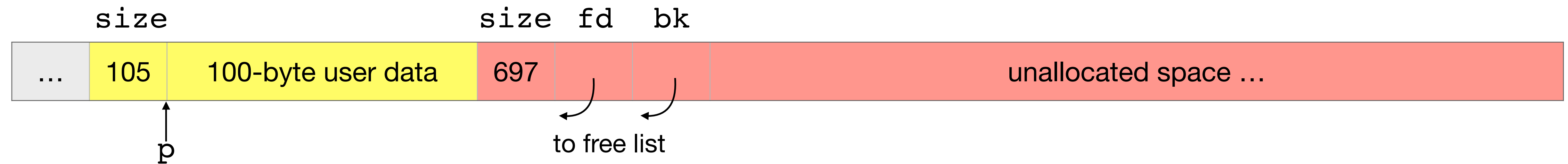


```
void *q = malloc(300);
```

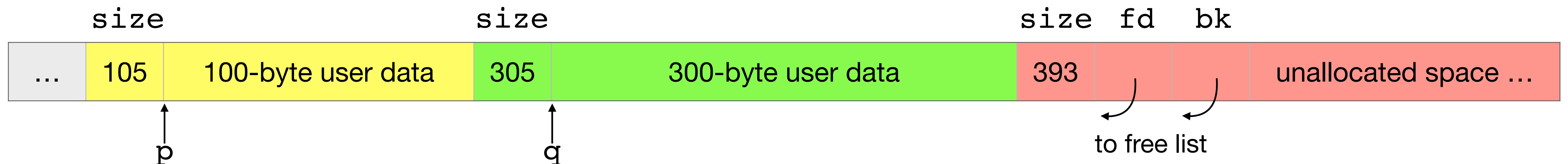
# Example (truth but not to scale)



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



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

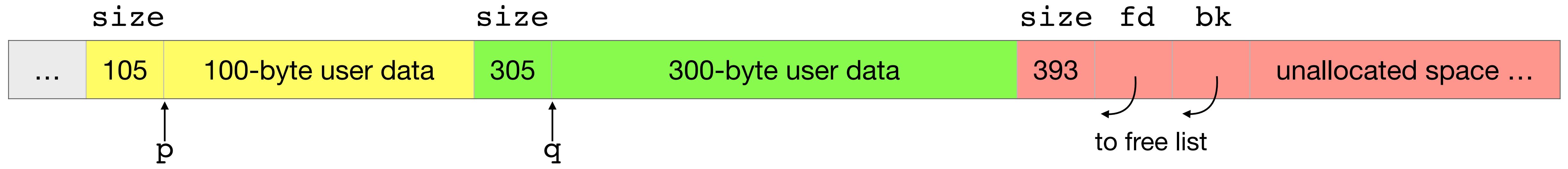




# Example continued



# Example continued

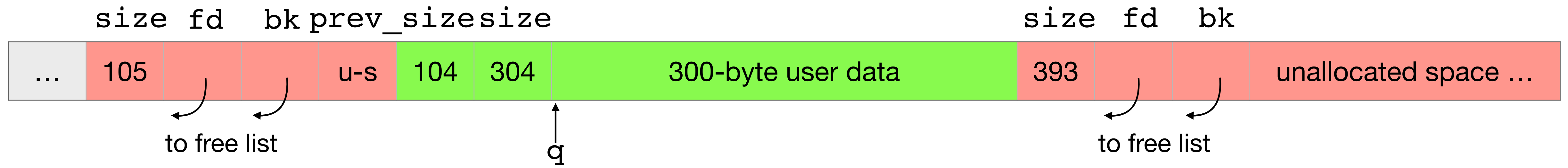


```
free(p);
```

# Example continued



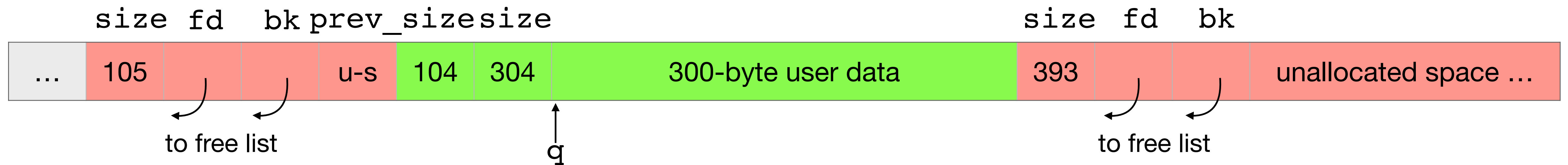
`free(p);`



# Example continued



```
free(p);
```

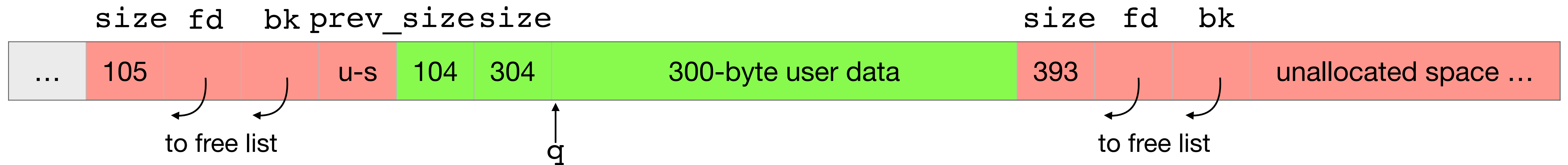


```
void *r = malloc(252);
```

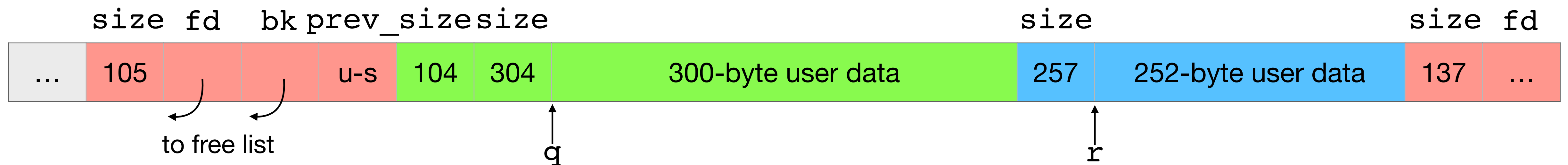
# Example continued



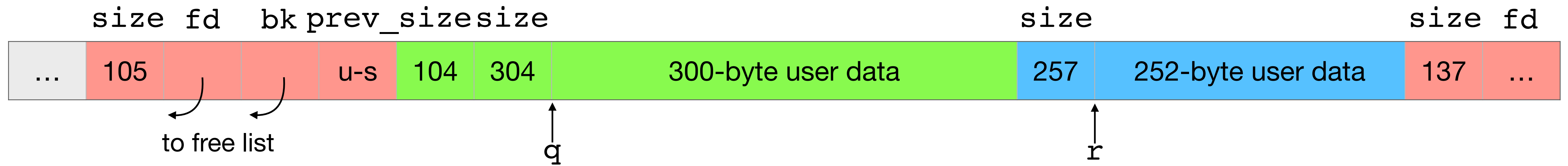
```
free(p);
```



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



# Example continued



# Example continued

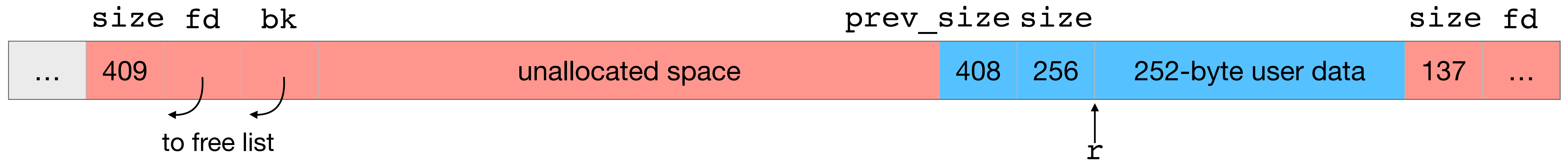


```
free(q);
```

# Example continued

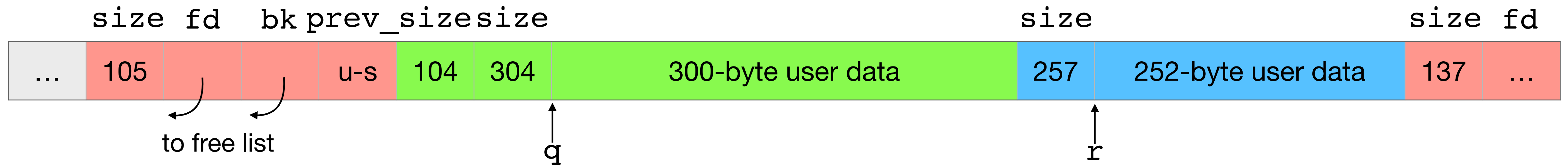


`free(q);`

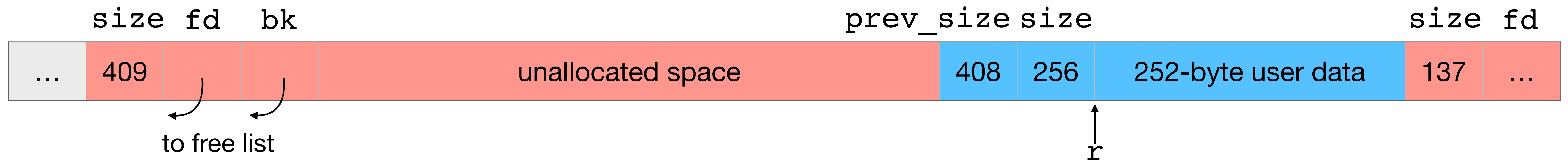




# Example continued

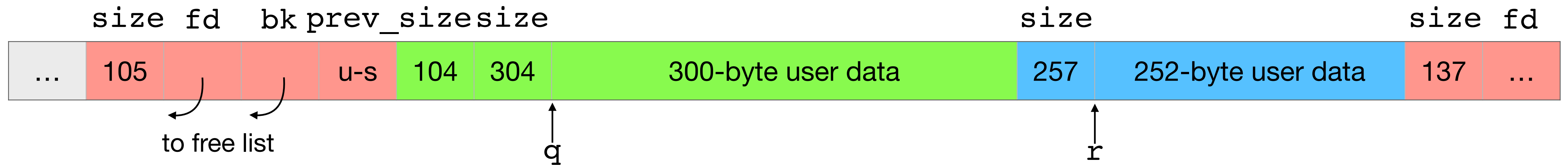


```
free(q);
```

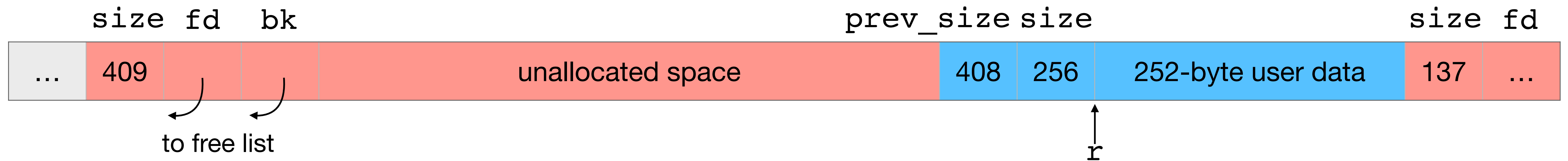


```
free(r);
```

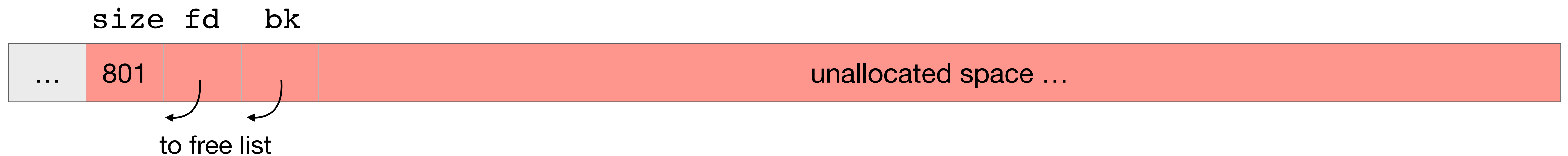
# Example continued



`free(q);`



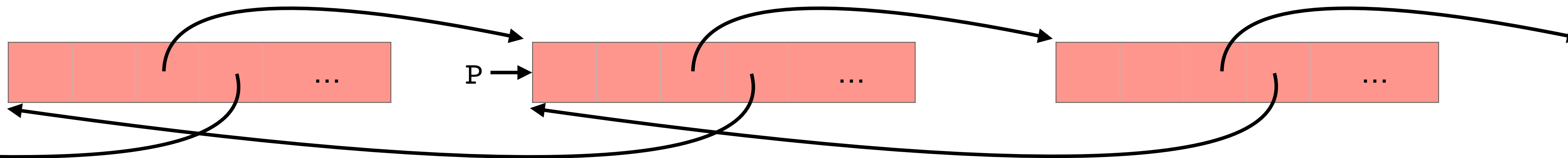
`free(r);`



# Removing chunks from free lists

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

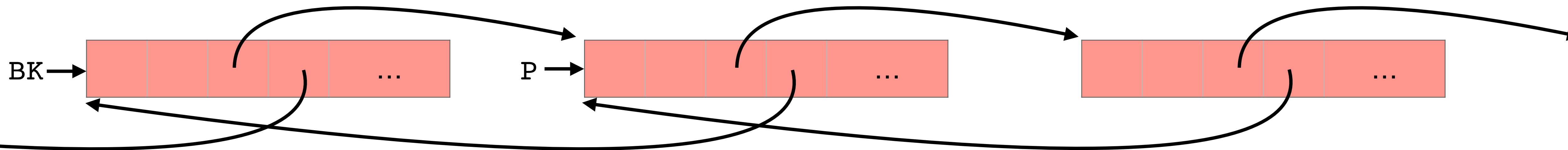
```
#define unlink(P, BK, FD) \  
    BK = P->bk; \  
    FD = P->fd; \  
    FD->bk = BK; \  
    BK->fd = FD;
```



# Removing chunks from free lists

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

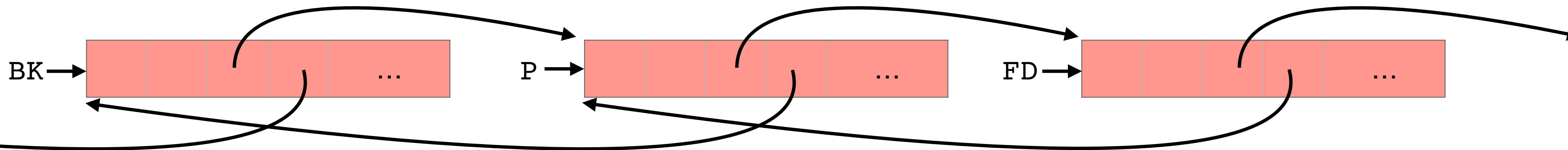
```
#define unlink(P, BK, FD) \  
    BK = P->bk; \  
    FD = P->fd; \  
    FD->bk = BK; \  
    BK->fd = FD;
```



# Removing chunks from free lists

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

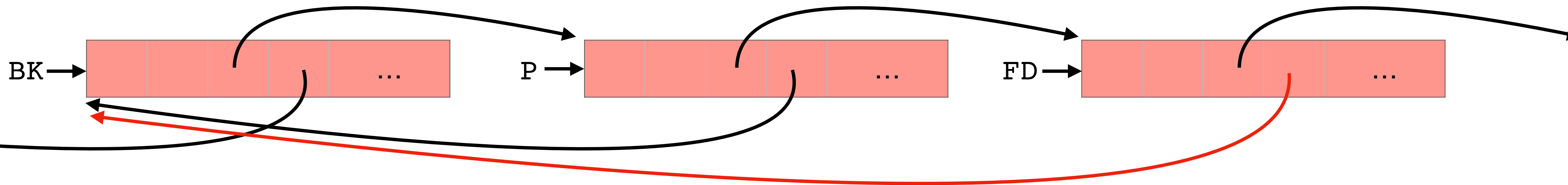
```
#define unlink(P, BK, FD) \  
    BK = P->bk; \  
    FD = P->fd; \  
    FD->bk = BK; \  
    BK->fd = P;
```



# Removing chunks from free lists

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

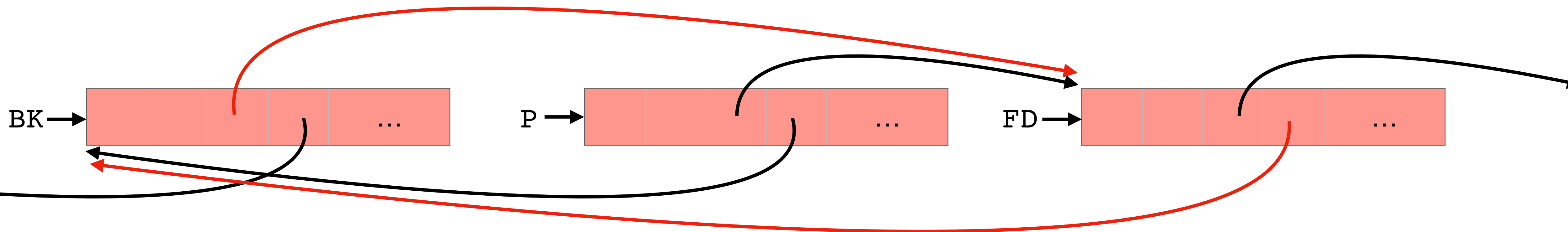
```
#define unlink(P, BK, FD) \  
    BK = P->bk; \  
    FD = P->fd; \  
    FD->bk = BK; \  
    BK->fd = P;
```



# 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 = P;
```



# 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 

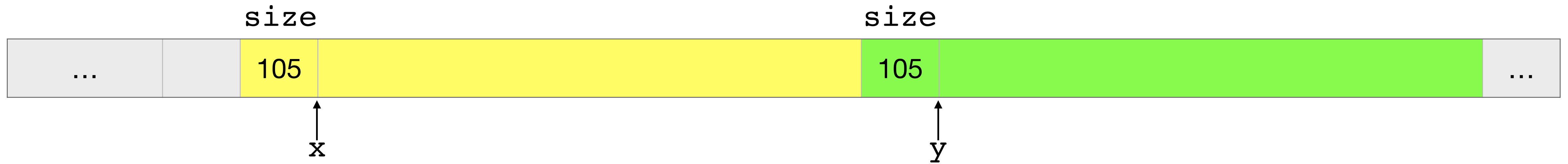
```
char *x = malloc(100);  
void *y = malloc(100);  
strcpy(x, attacker_controlled);  
free(y);
```



- We can overflow x and overwrite y's metadata

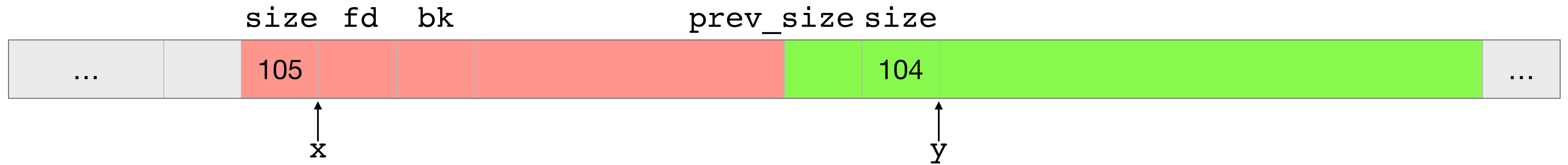


# Attacking malloc



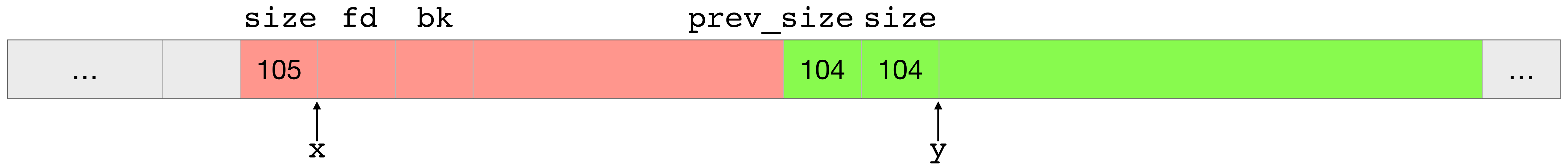
- When `free(y)` is called, it will examine `x`'s chunk to see if it is free.
- If `x`'s chunk is free, then `unlink` will be called on it to remove it from its free list
- We can carefully structure the attacker-controlled data to
  - convince `free` that `x`'s chunk is free (how do we do this?)
  - 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



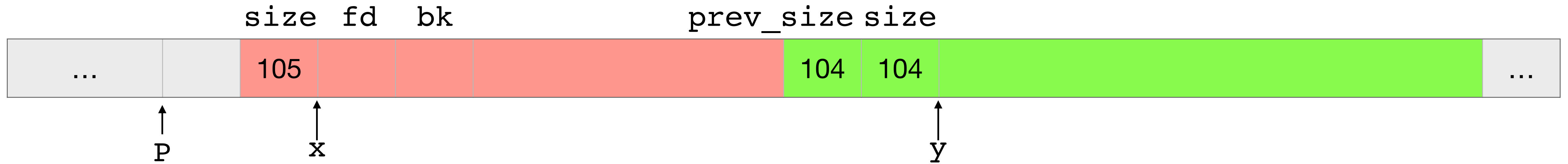
1. Change y's chunk's size from 105 to 104 (clears the PREV\_IN\_USE bit); y's chunk's prev\_size and x's chunk's fd and bk are now used

# Attacking malloc



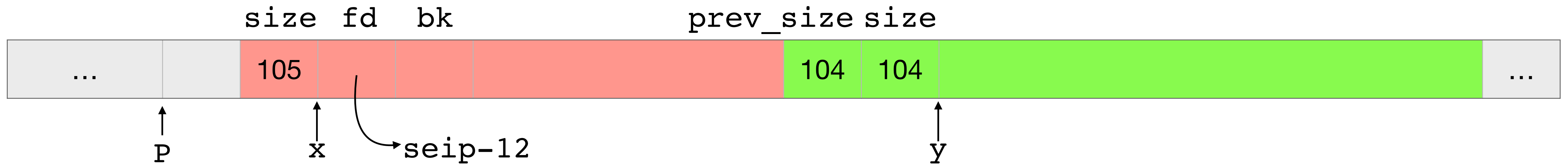
1. Change y's chunk's size from 105 to 104 (clears the PREV\_IN\_USE bit); y's chunk's prev\_size and x's chunk's fd and bk are now used
2. Set y's chunk's prev\_size to 104 so free looks back 104 bytes to find the start of the chunk to unlink

# Attacking malloc



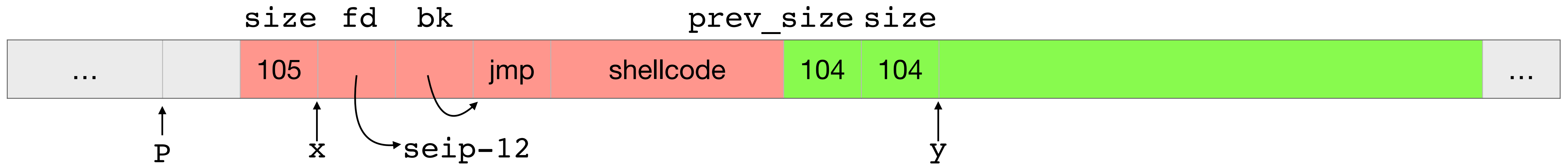
1. Change `y`'s chunk's size from 105 to 104 (clears the `PREV_IN_USE` bit); `y`'s chunk's `prev_size` and `x`'s chunk's `fd` and `bk` are now used
2. Set `y`'s chunk's `prev_size` to 104 so `free` looks back 104 bytes to find the start of the chunk to unlink
3. `P` in the `unlink` macro is `x`'s chunk,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

# Attacking malloc



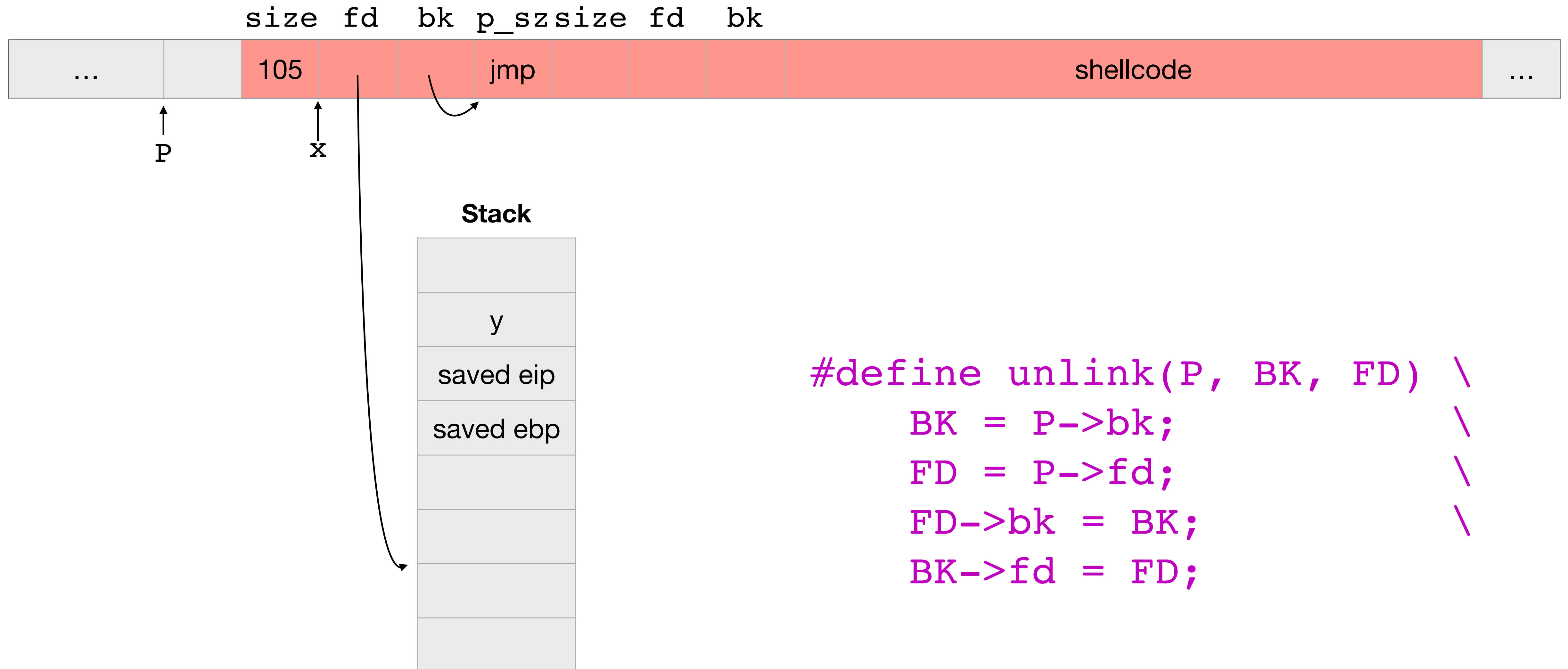
1. Change  $y$ 's chunk's size from 105 to 104 (clears the PREV\_IN\_USE bit);  $y$ 's chunk's prev\_size and  $x$ 's chunk's fd and bk are now used
2. Set  $y$ 's chunk's prev\_size to 104 so free looks back 104 bytes to find the start of the chunk to unlink
3.  $P$  in the unlink macro is  $x$ 's chunk, so its fd and bk pointers need to be valid
4. Point  $P \rightarrow fd$  to saved eip (seip) - 12

# Attacking malloc

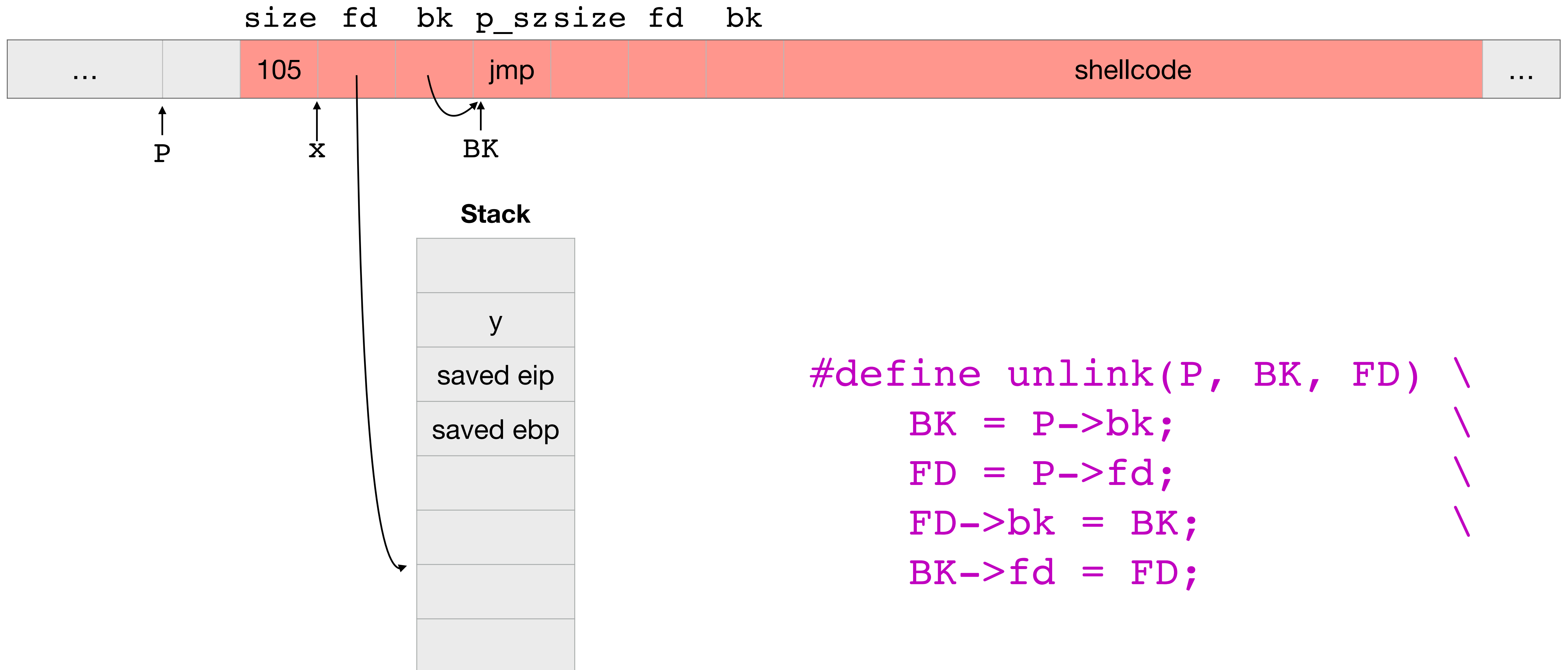


1. Change  $y$ 's chunk's size from 105 to 104 (clears the PREV\_IN\_USE bit);  $y$ 's chunk's prev\_size and  $x$ 's chunk's fd and bk are now used
2. Set  $y$ 's chunk's prev\_size to 104 so free looks back 104 bytes to find the start of the chunk to unlink
3.  $P$  in the unlink macro is  $x$ 's chunk, so its fd and bk pointers need to be valid
4. Point  $P \rightarrow fd$  to saved eip (seip) - 12
5. Point  $P \rightarrow bk$  to a short jump to shellcode

# Unlinking P (zooming in on P)

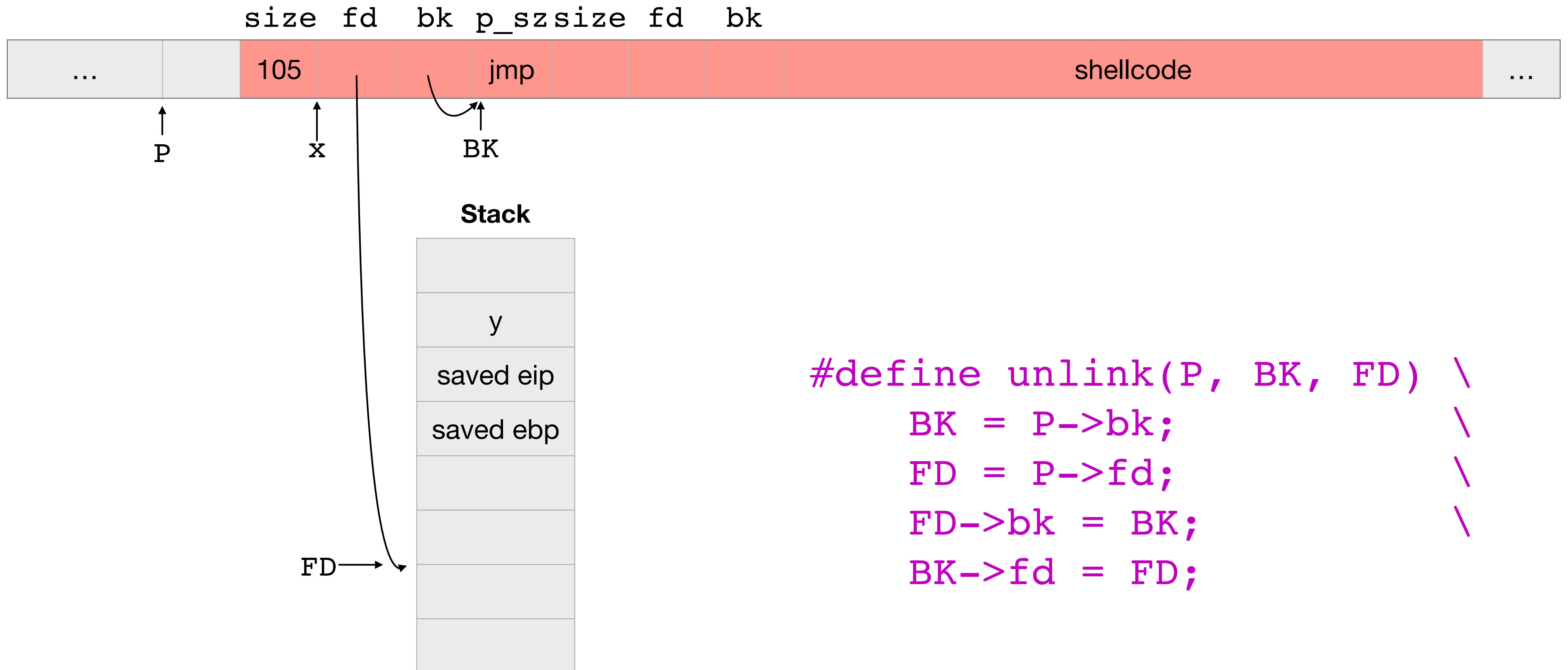


# Unlinking P (zooming in on P)

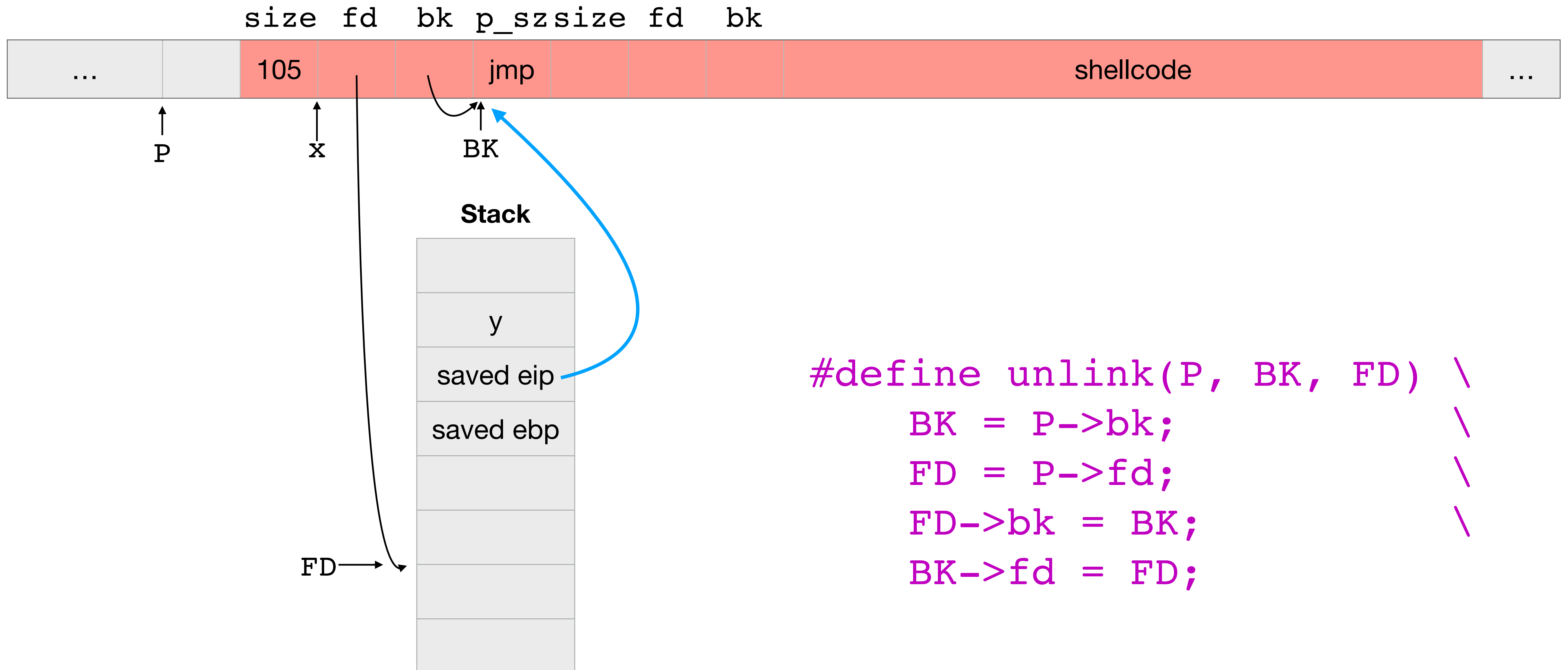




# Unlinking P (zooming in on P)



# 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)

