Allocator with Implicit Free List and Coalescing

Current allocator works... performance?

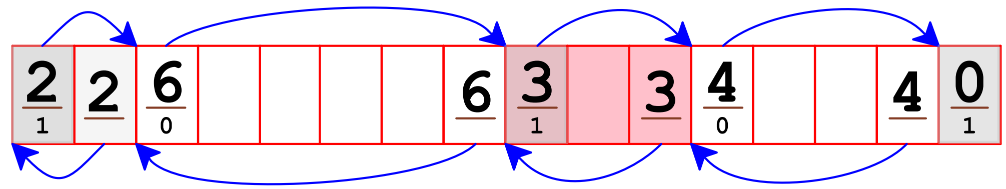
Utilization

Can create too much fragmentation

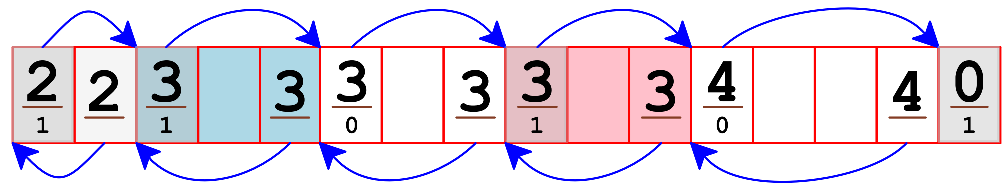
Throughput

Can take a long time to find a block

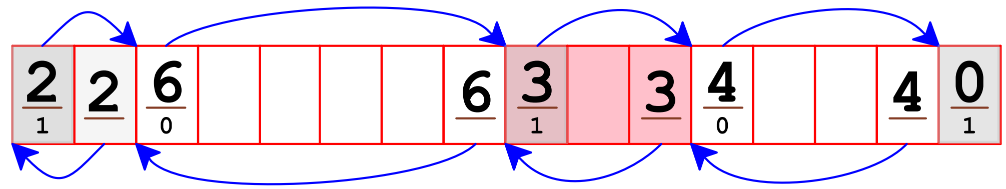
Choosing a Free Block



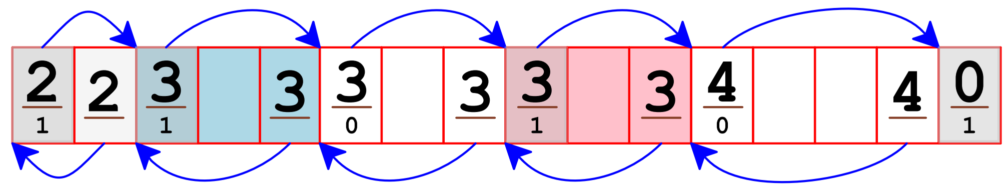
`p2 = malloc(1)`



Choosing a Free Block



`p2 = malloc(1)`

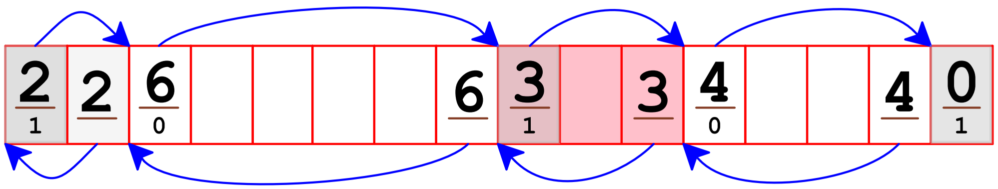


`p4 = malloc(4)`

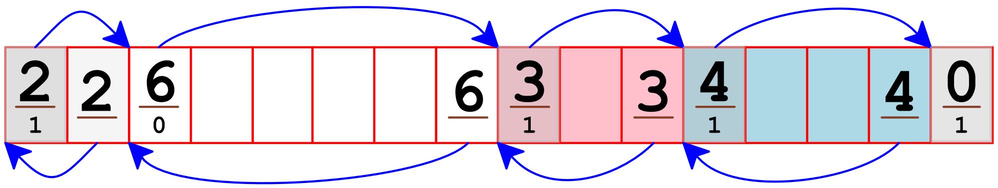
First fit: Use (and possibly split) the first block that works

danger of fragmentation

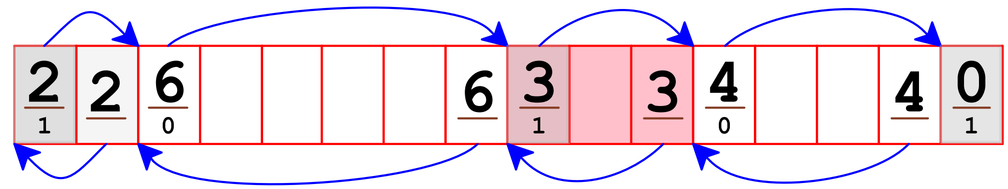
Choosing a Free Block



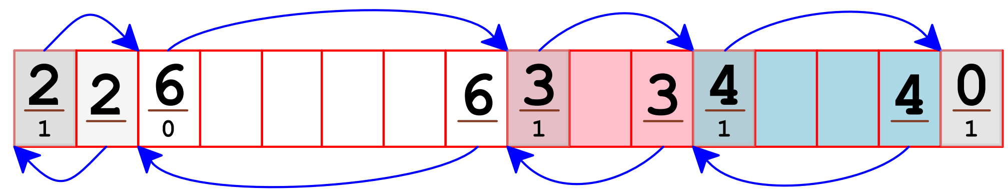
`p2 = malloc(1)`



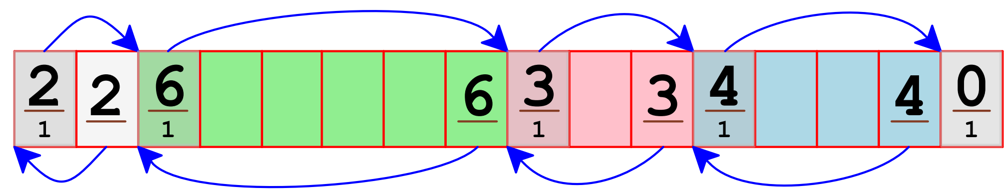
Choosing a Free Block



`p2 = malloc(1)`



`p4 = malloc(4)`



Best fit: Use (and possibly split) the smallest block that works

usually reduces fragmentation

First-Fit Implementation

Our current implementation is first-fit:

```
while (GET_SIZE(HDRP(bp)) != 0) {
  if (!GET_ALLOC(HDRP(bp))
      && (GET_SIZE(HDRP(bp)) >= new_size)) {
    set_allocated(bp, new_size);
    return bp;
  }
  bp = NEXT_BLKP(bp);
}
```

Best-Fit Implementation

A best-fit search:

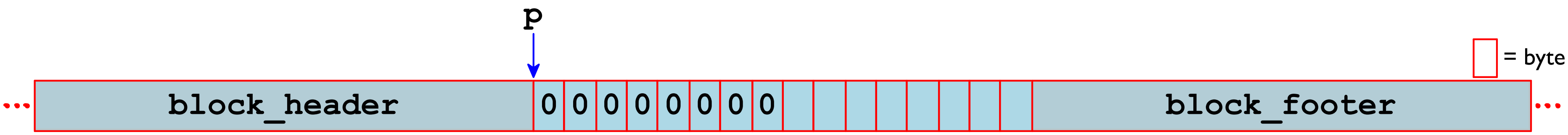
```
void *best_bp = NULL;

while (GET_SIZE(HDRP(bp)) != 0) {
  if (!GET_ALLOC(HDRP(bp))
      && (GET_SIZE(HDRP(bp)) >= new_size)) {
    if (!best_bp
        || (GET_SIZE(HDRP(bp)) < GET_SIZE(HDRP(best_bp))))
      best_bp = bp;
  }
  bp = NEXT_BLKP(bp);
}
if (best_bp) {
  set_allocated(best_bp, new_size);
  return best_bp;
}
```

Copy

Trades throughput for utilization

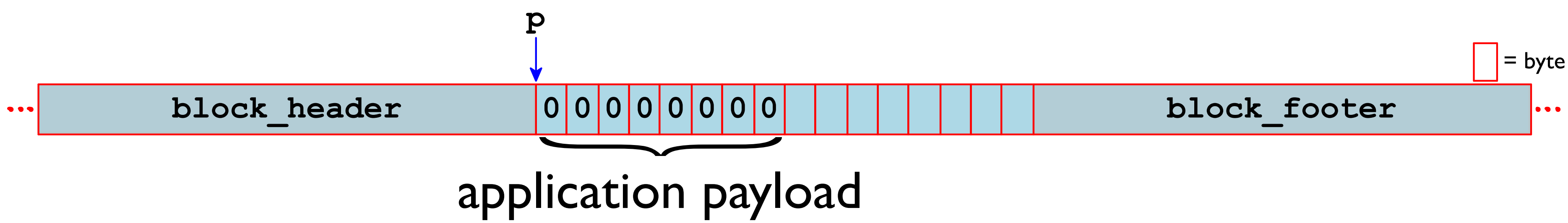
Internal Fragmentation

```
p = mm_malloc(8);
memset(p, 0, 8);
```



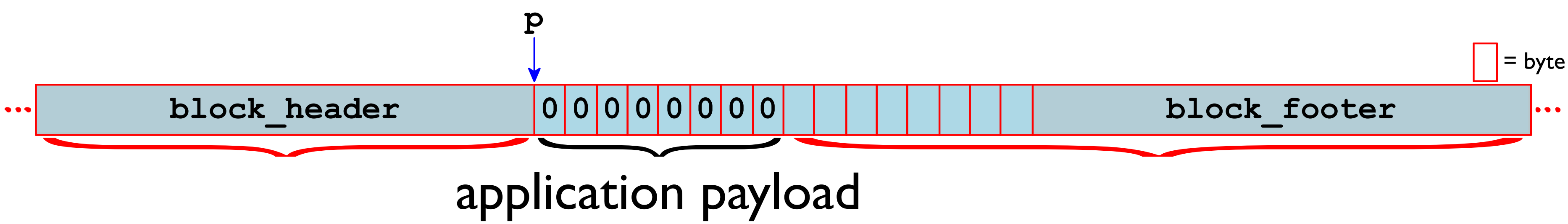
Internal Fragmentation

```
p = mm_malloc(8);
memset(p, 0, 8);
```



Internal Fragmentation

```
p = mm_malloc(8);
memset(p, 0, 8);
```

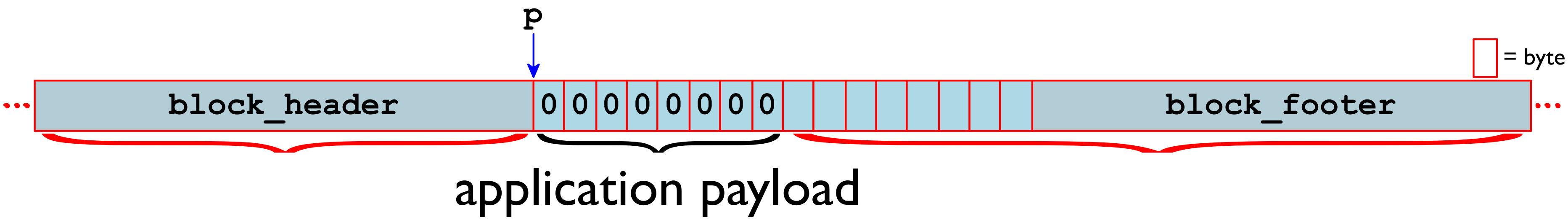


Everything except the application payload reduces utilization

Internal fragmentation refers to space within an allocated block that is unusable to the application

Internal Fragmentation

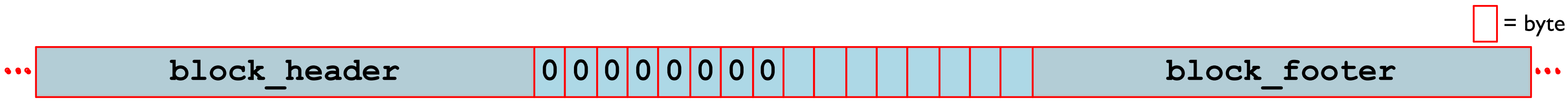
```
p = mm_malloc(8);
memset(p, 0, 8);
```



Sources of internal fragmentation:

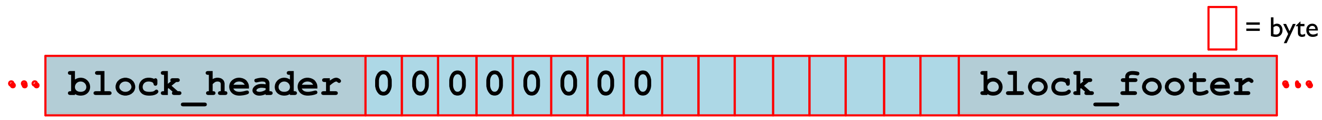
- headers and footers
- empty space to maintain alignment
- empty space due to choice of fit

Encoding Header and Footer Information




```
typedef struct {
  size_t size;
  char   allocated;
} block_header;

typedef struct {
  size_t size;
  int filler;
} block_footer;
```

Encoding Header and Footer Information

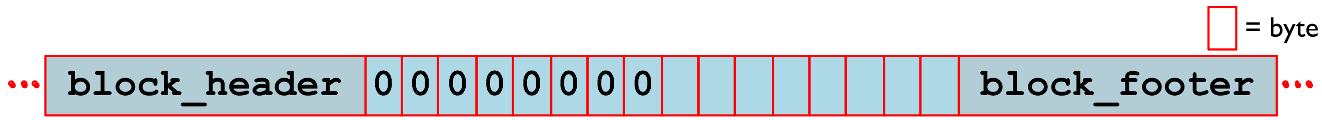



```
typedef size_t block_header;

typedef size_t block_footer;
```

Since a block size is always a multiple of 16, low 4 bits are always 0

Idea: use the low bit to indicate allocation status

Encoding Header and Footer Information




```
typedef size_t block_header;

typedef size_t block_footer;
```

```
#define GET(p)  (*(size_t *)(p))

#define GET_ALLOC(p)  (GET(p) & 0x1)
#define GET_SIZE(p)   (GET(p) & ~0xF)

#define PUT(p, val)  (*(size_t *)(p) = (val))

#define PACK(size, alloc)  ((size) | (alloc))
```

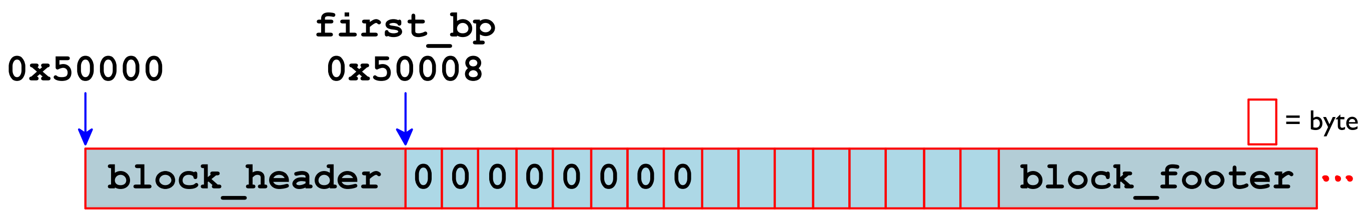
Packing Demo

```
#include <stdio.h>
#include <stdlib.h>

#define GET(p)       (*(size_t *)(p))
#define PUT(p, val)  (*(size_t *)(p) = (val))

#define GET_ALLOC(p)  (GET(p) & 0x1)
#define GET_SIZE(p)   (GET(p) & ~0xF)

#define PACK(size, alloc) ((size) | (alloc))

int main() {
  void *p = malloc(sizeof(size_t));

  PUT(p, PACK(48, 1));
  printf("%ld %s\n",
         GET_SIZE(p),
         (GET_ALLOC(p) ? "alloc" : "unalloc"));
}
```

Copy

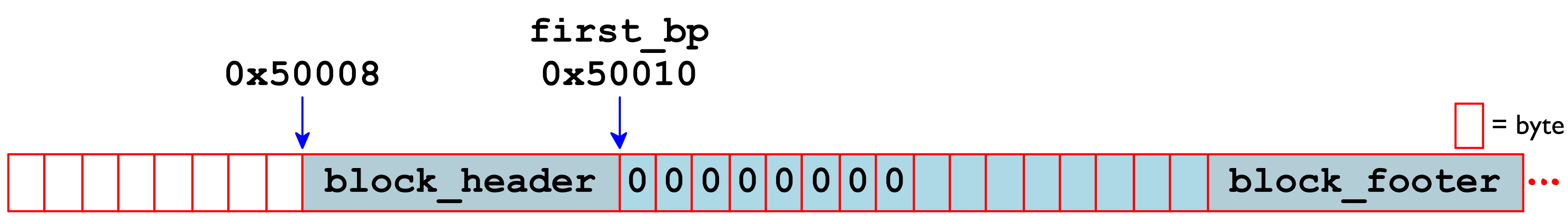
Alignment

A smaller header can break our alignment strategy:



Solution:

- Make sure **`first_bp`** has correct alignment

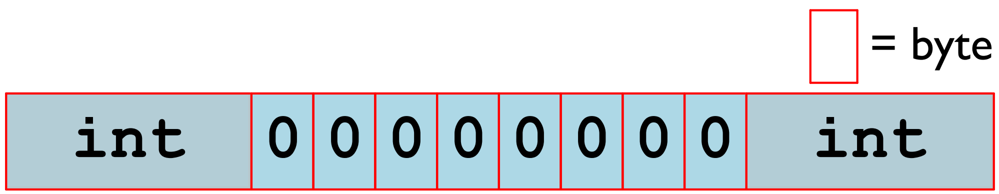


- Align total block size, not payload size

Even Smaller Headers and Footers

If the block size is constrained to be < 2^{32}:

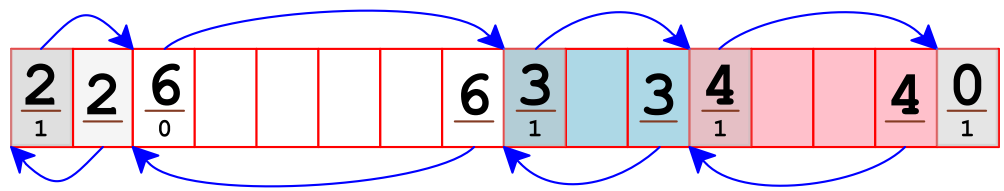
```
typedef int block_header;
typedef int block_footer;
#define GET(p)          (*(int *)(p))
#define PUT(p, val)  (*(int *)(p) = (val))
```



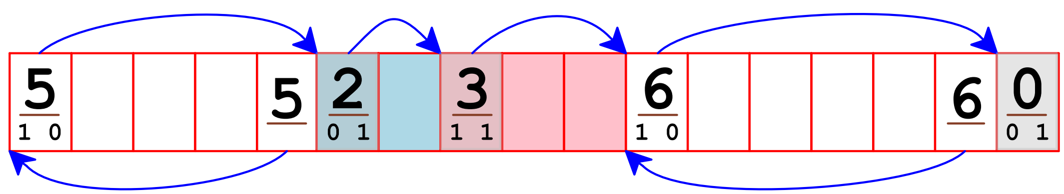
Allocator might treat very large blocks differently

Advanced: Footers Only for Unallocated Blocks

Our allocator needs to go backwards only for coalescing:



Idea: record in block header whether *previous* block is allocated

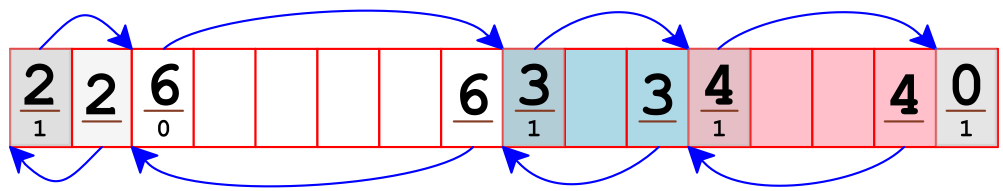


Make sure block size is big enough for footer to be added

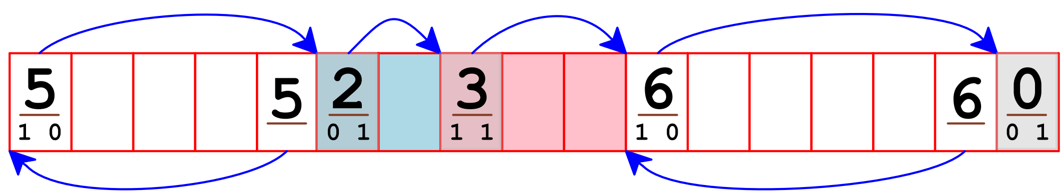
`free(p2)` ⇒ previous block is unallocated, so use `PREV_BLKP`

Advanced: Footers Only for Unallocated Blocks

Our allocator needs to go backwards only for coalescing:



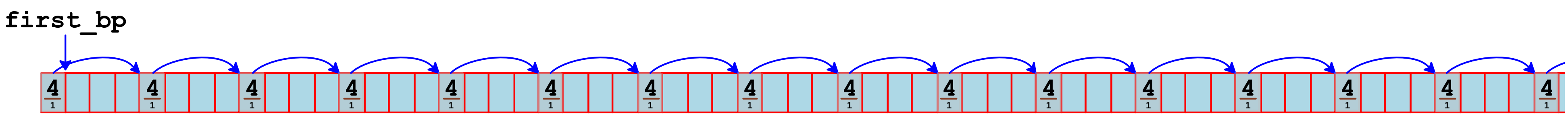
Idea: record in block header whether *previous* block is allocated



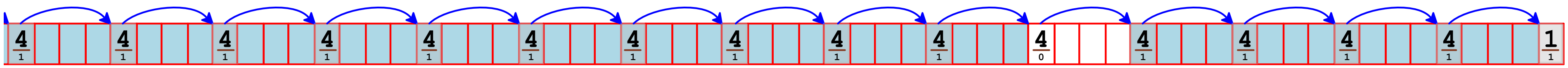
Make sure block size is big enough for footer to be added

`free(p3)` ⇒ previous block is allocated, so don't try **`PREV_BLKP`**

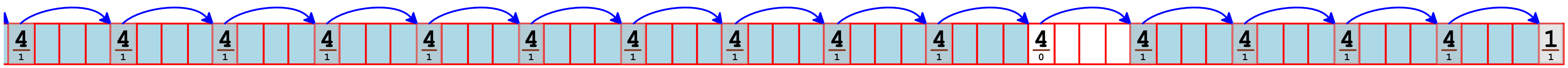
Looking for an Unallocated Block



Looking for an Unallocated Block

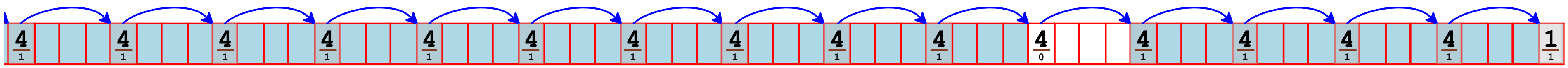


Looking for an Unallocated Block



Finding an unallocated is a significant limitation on throughput

Looking for an Unallocated Block



Finding an unallocated is a significant limitation on throughput

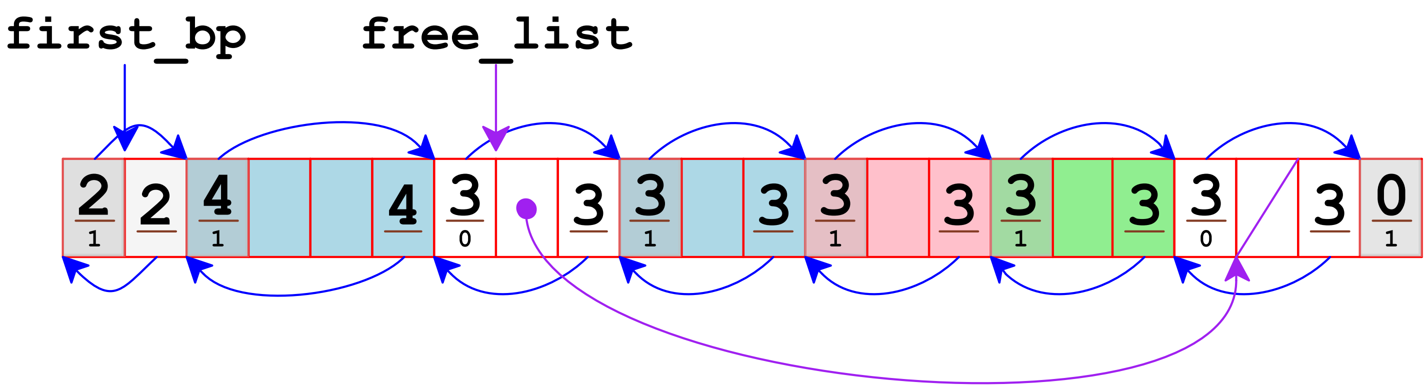
Instead of searching through *all* blocks, keep a list of just the free ones

The allocator will have an ***explicit free list*** instead of an ***implicit free list***

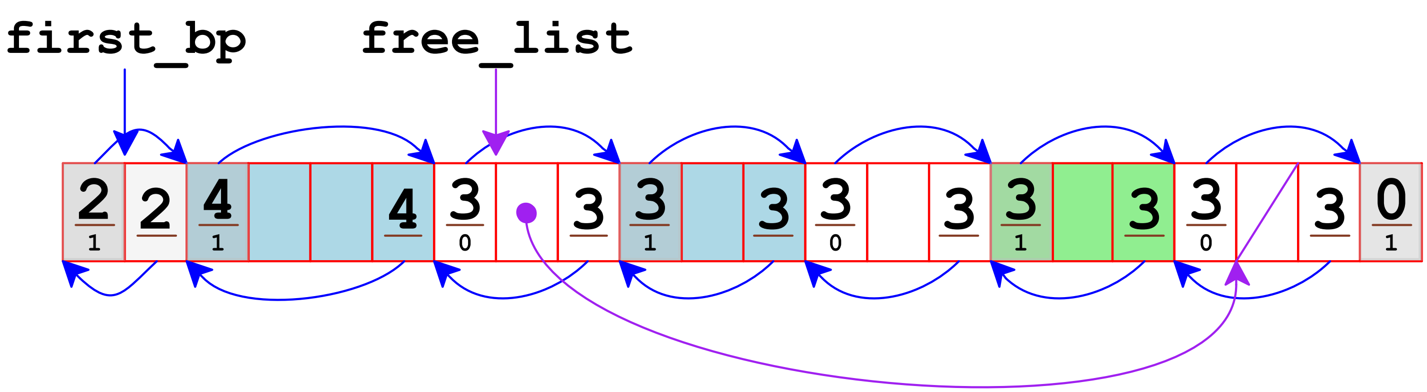
Explicit Free Lists

Make sure that every block has room for a pointer

replaces the application's payload



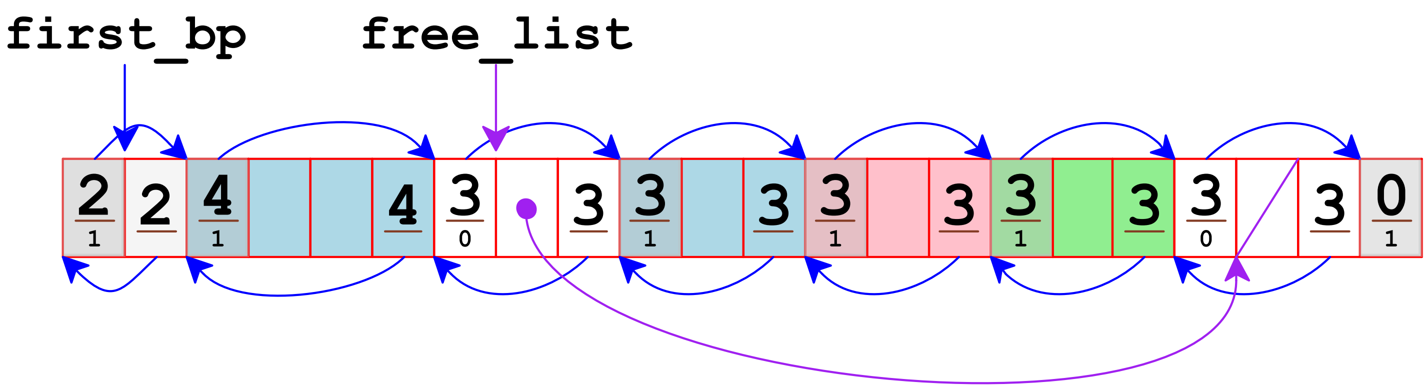
free(p3)



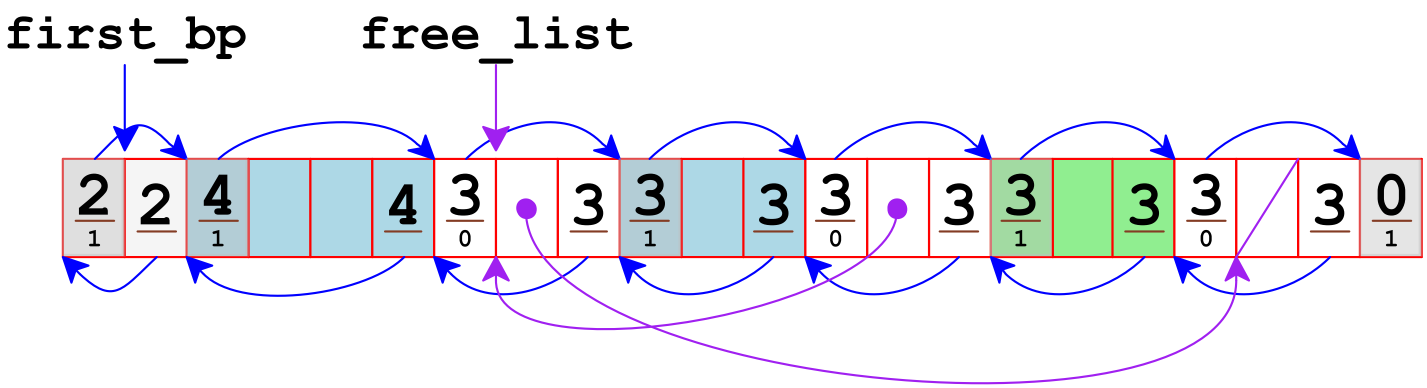
Explicit Free Lists

Make sure that every block has room for a pointer

replaces the application's payload



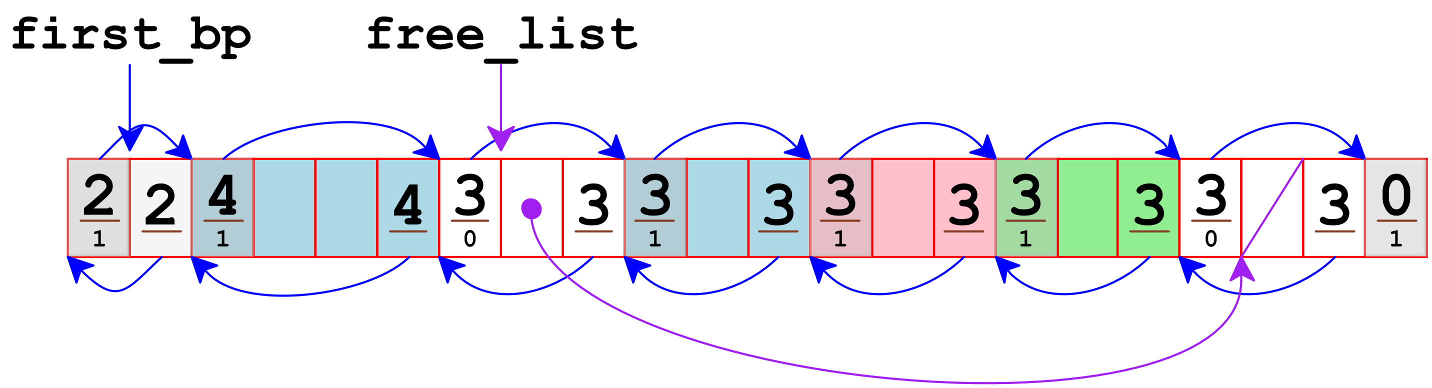
free(p3)



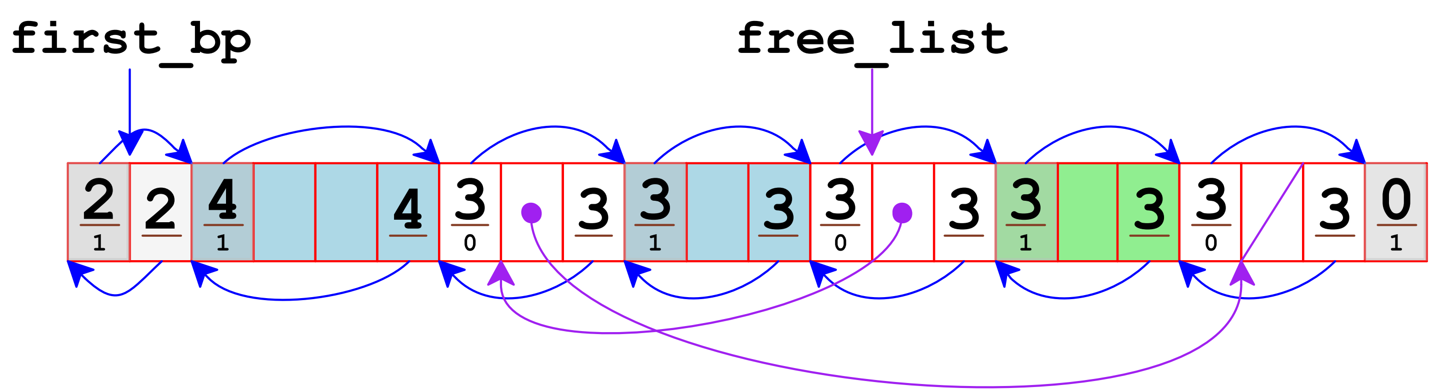
Explicit Free Lists

Make sure that every block has room for a pointer

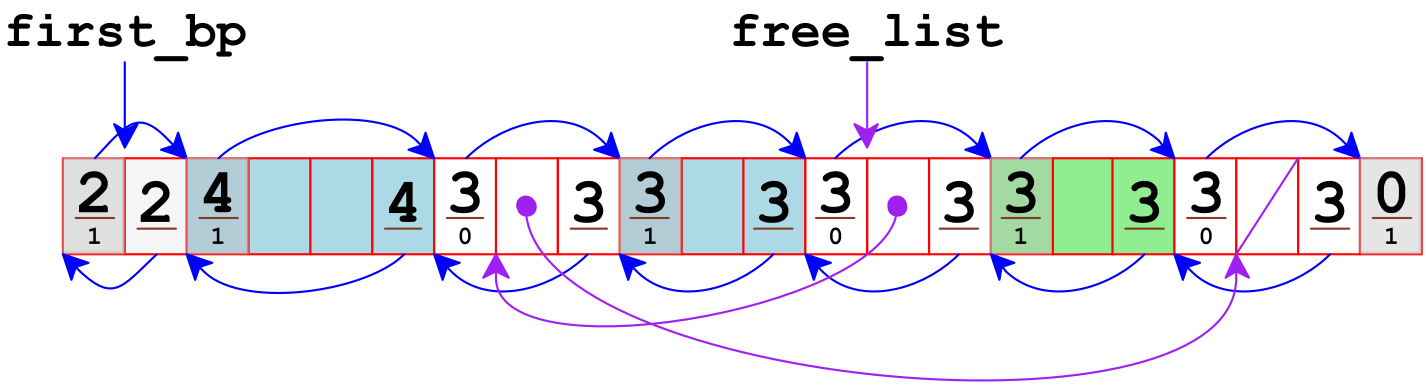
replaces the application's payload



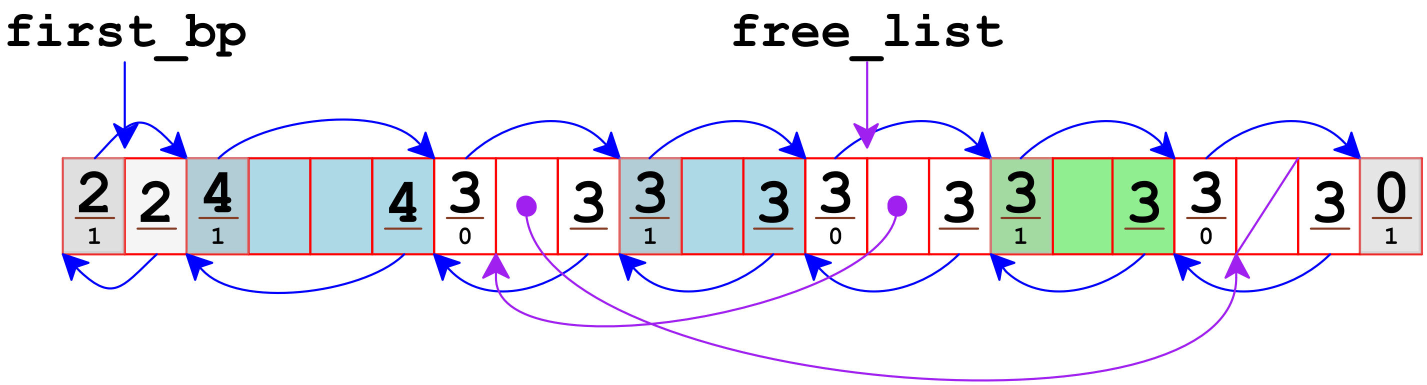
`free(p3)`



Free Lists and Coelescing

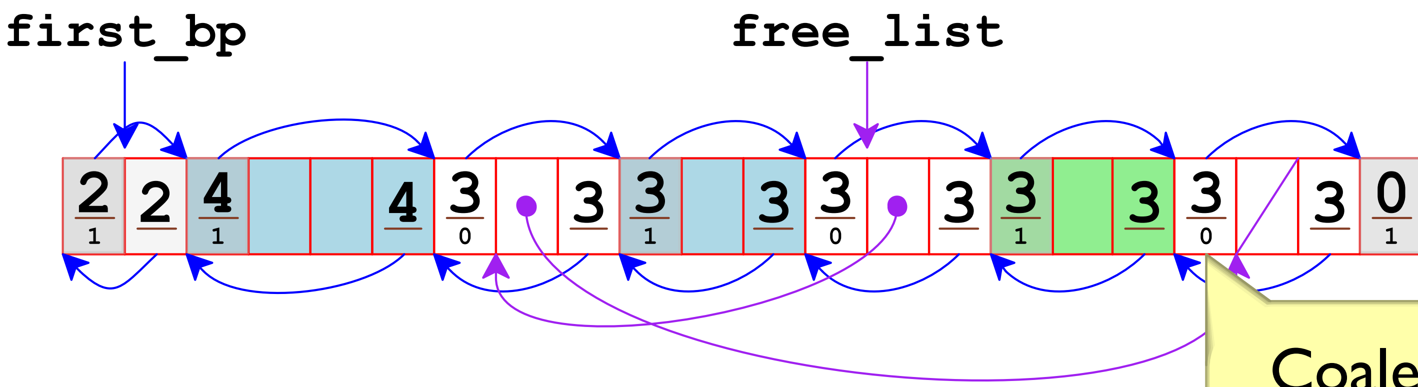


Free Lists and Coelescing



free(p4)

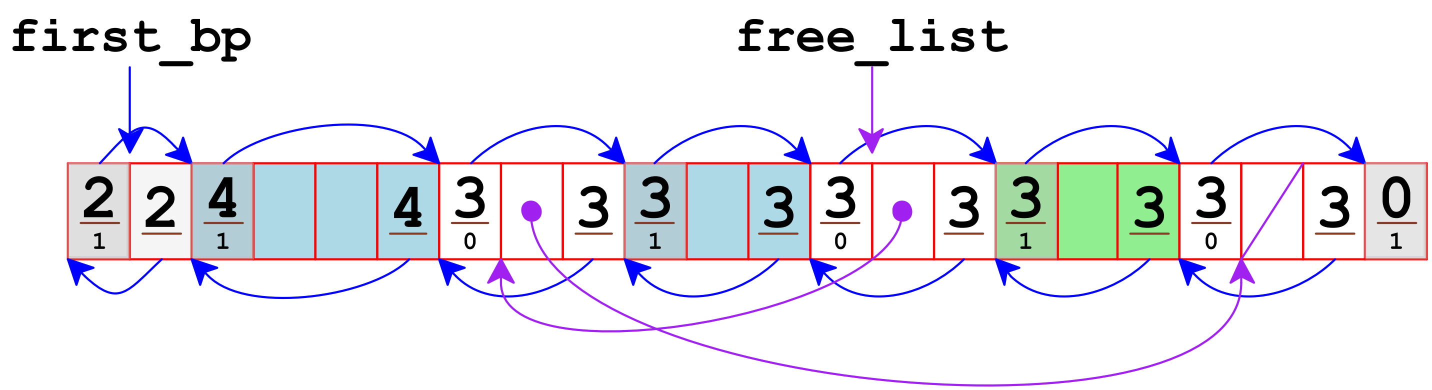
Free Lists and Coelescing



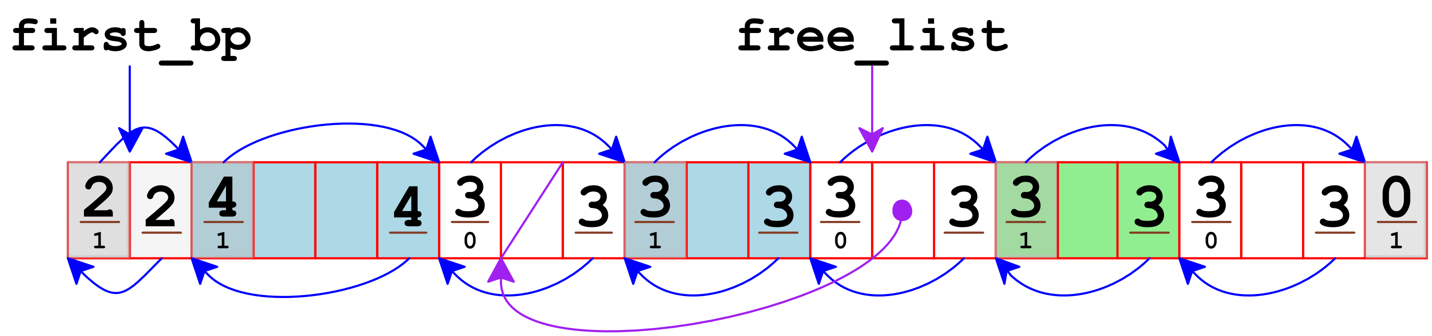
`free(p4)`

Coalesce next block: first, remove it from free list

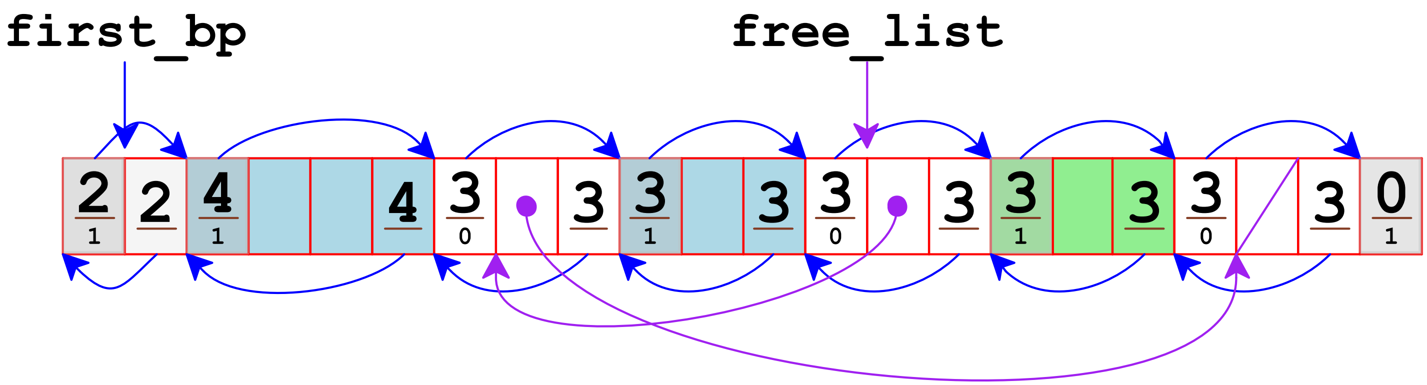
Free Lists and Coelescing



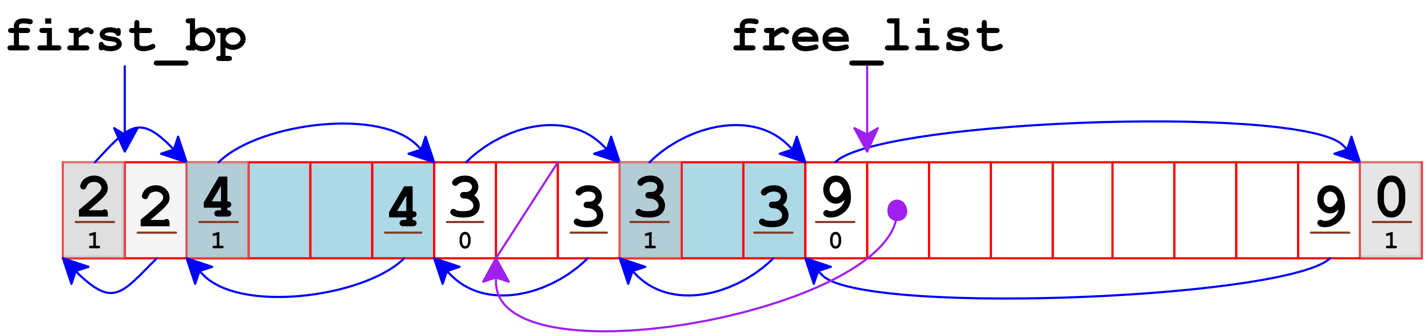
free(p4)



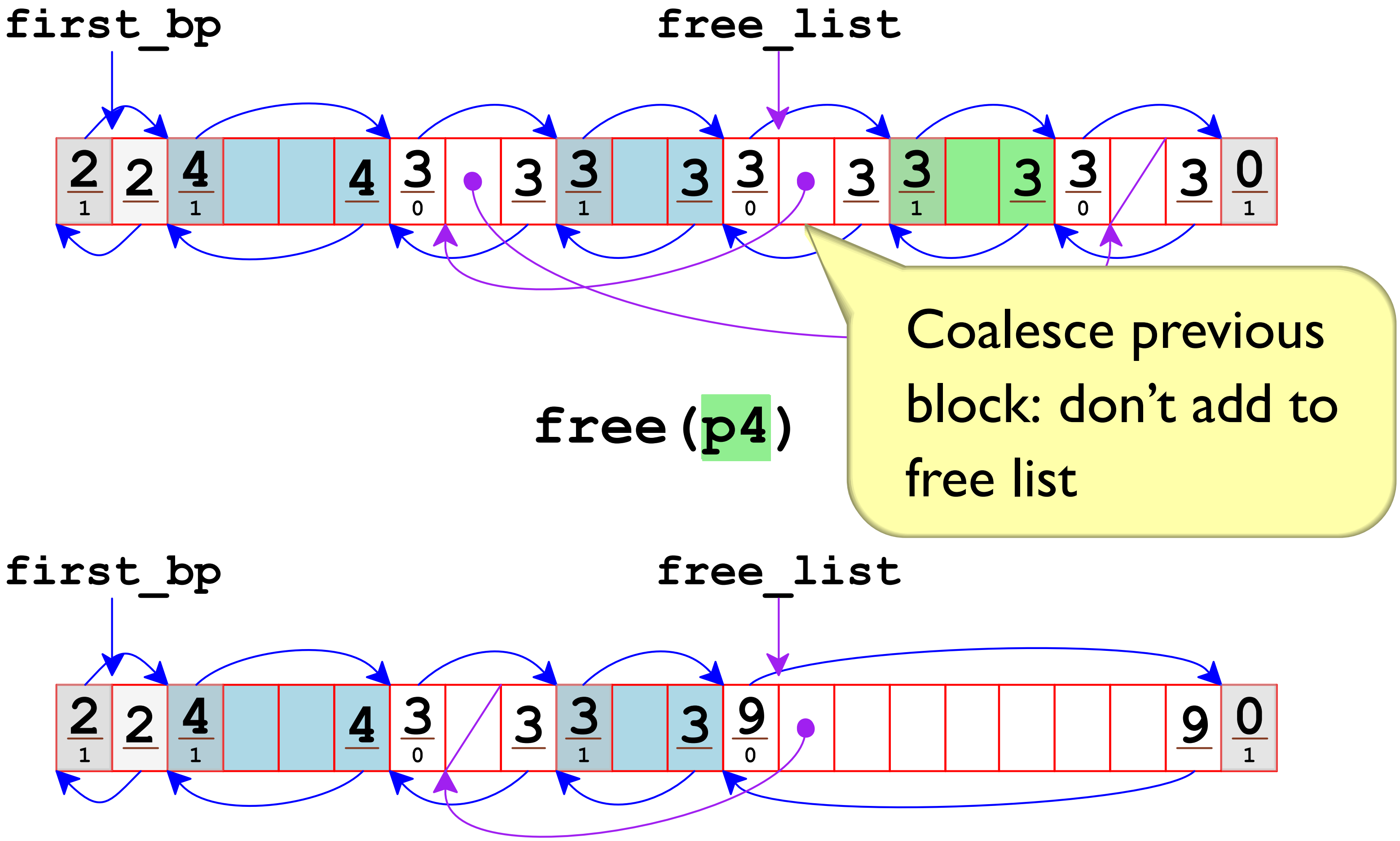
Free Lists and Coelescing



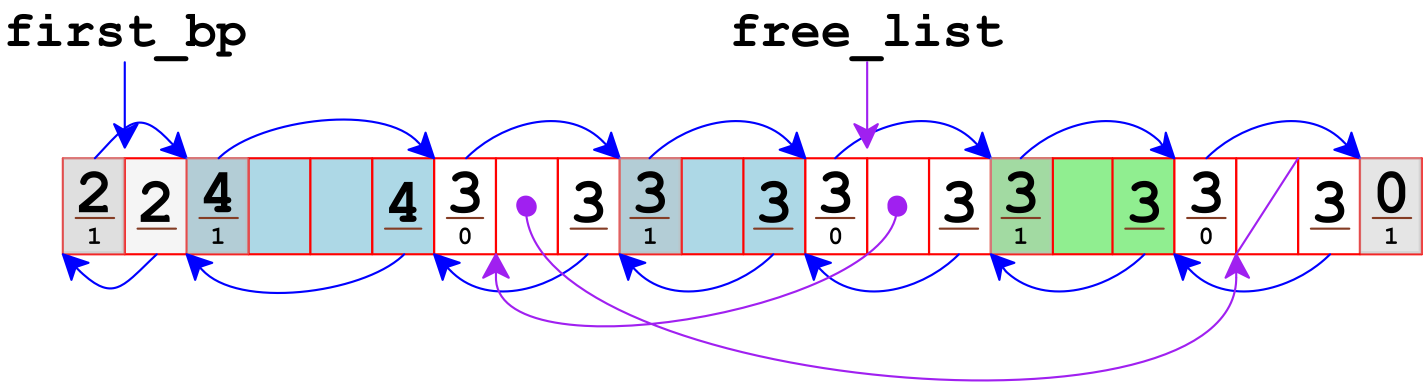
`free(p4)`



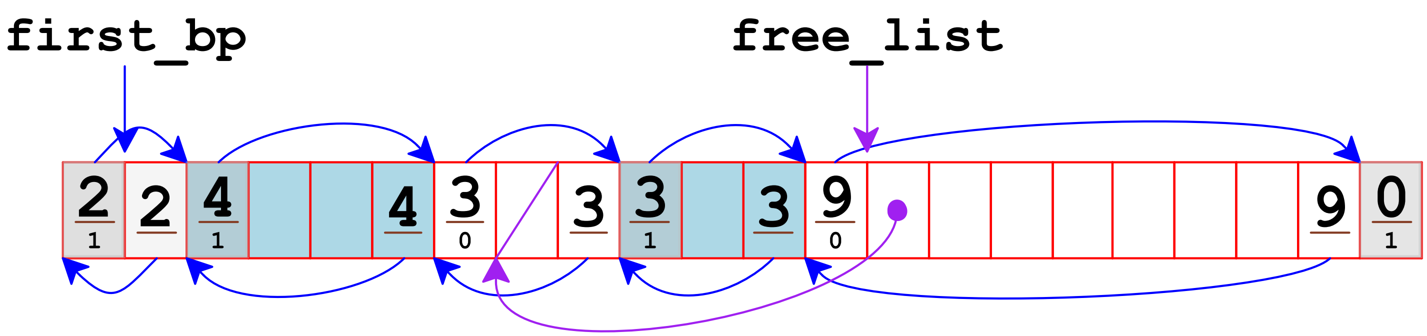
Free Lists and Coelescing



Free Lists and Coelescing



`free(p4)`



Invariant: every unallocated block is on the free list

Free List Data Structure

Linked list not convenient?

Use a doubly-linked lists

```
typedef struct list_node {
   struct list_node *prev;
   struct list_node *next;
} list_node;

....

void *mm_malloc(size_t size) {
  int need_size = max(size, sizeof(list_node));
  int new_size = ALIGN(need_size + OVERHEAD);
  ....
}
```

Free List Data Structure

Linked list not convenient?

Use a doubly-linked lists

```
void *coalesce(void *bp) {
  ....

  if (prev_alloc && next_alloc) {      /* Case 1 */
    add_to_free_list((list_node *)bp);
  }

  ....
}
```

Selecting from a Free List

First fit and ***best fit*** are still options with an explicit free list

More options:

LIFO — add to front of free list; take from front of list

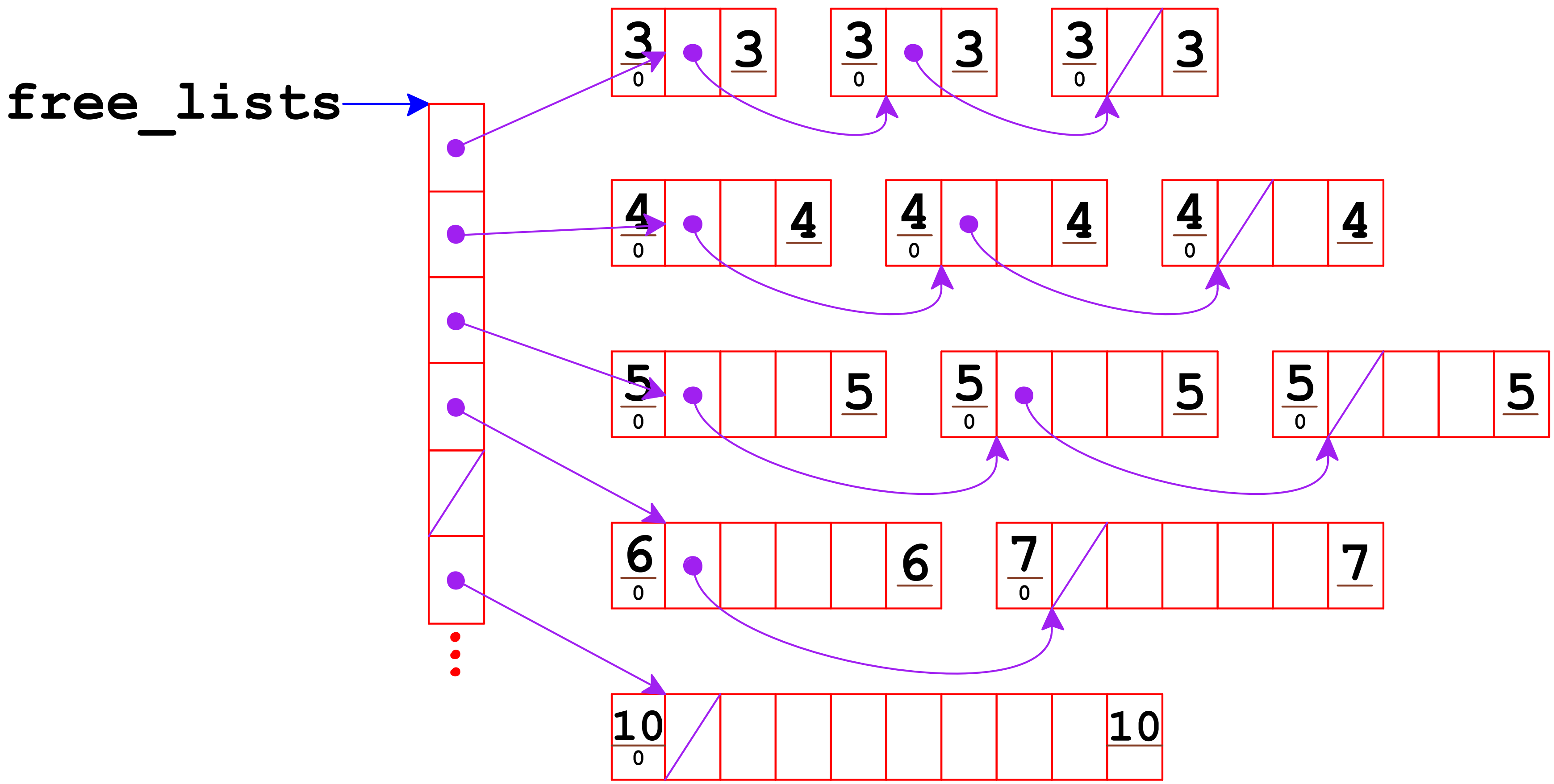
a kind of first-fit that tends to promote locality

address ordered — pick block with lowest address

may reduce fragmentation

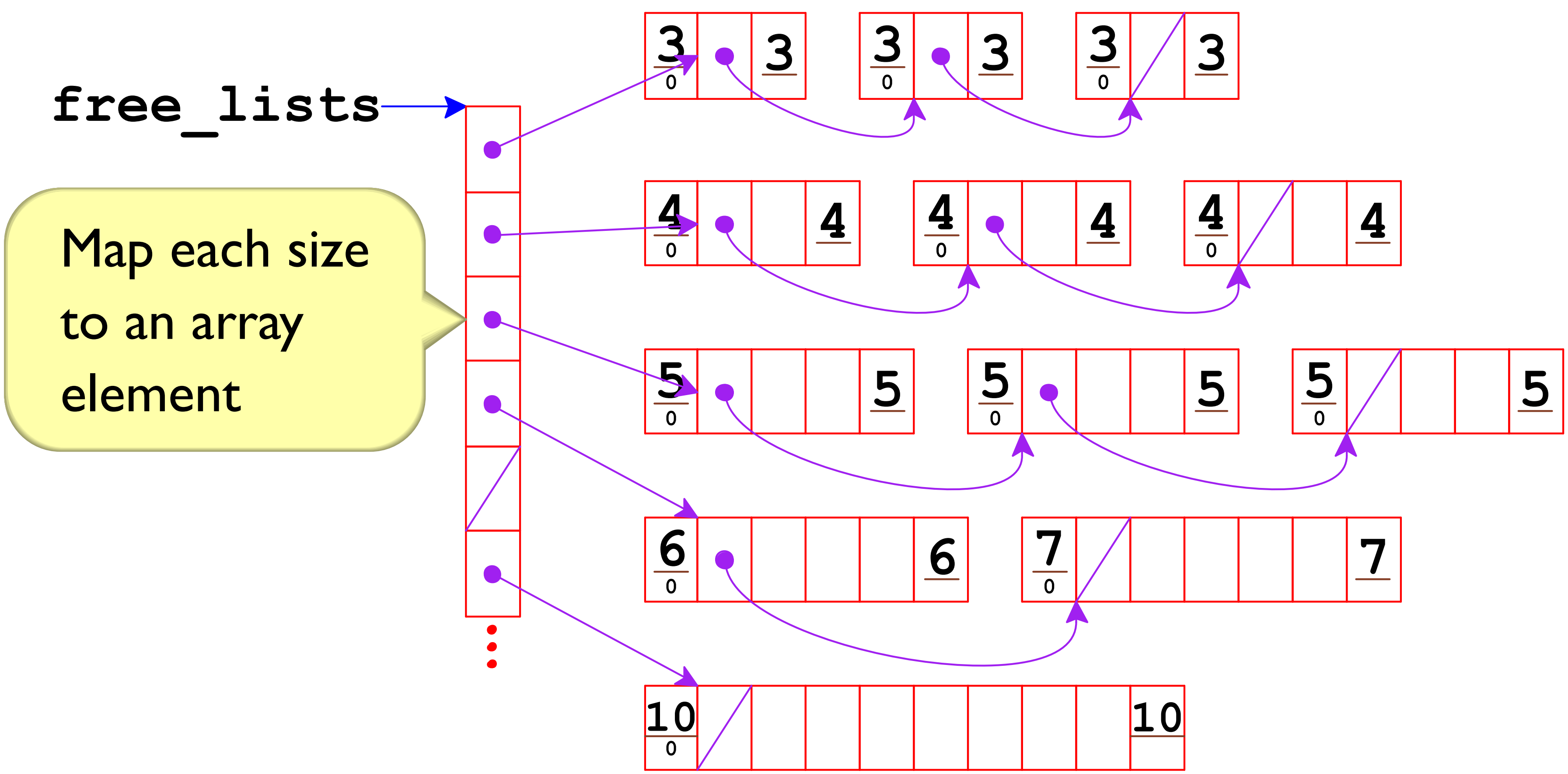
Segregated Free List

A ***segregated free list*** is an array of free lists, where each list has objects of a particular size class



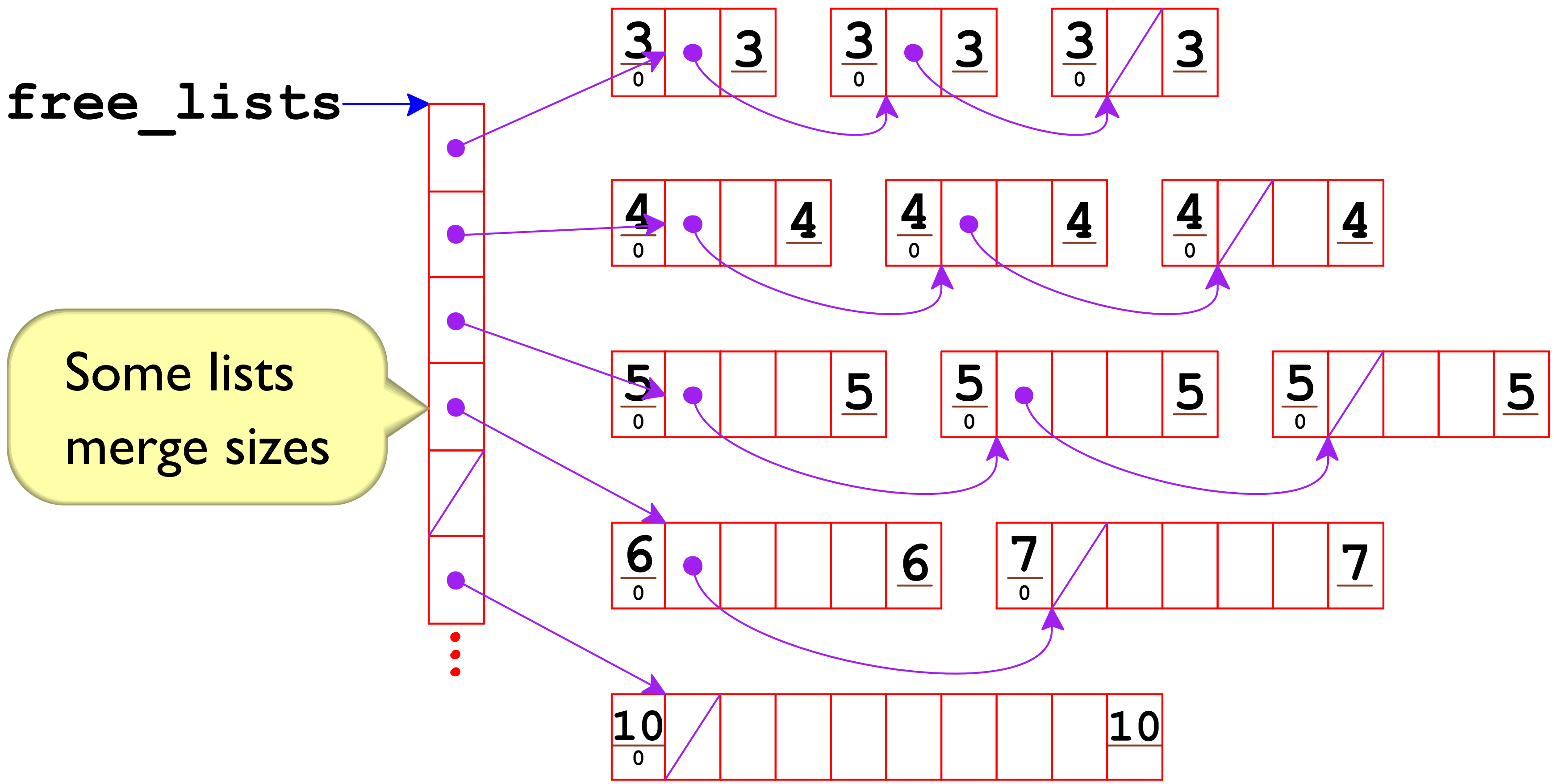
Segregated Free List

A ***segregated free list*** is an array of free lists, where each list has objects of a particular size class



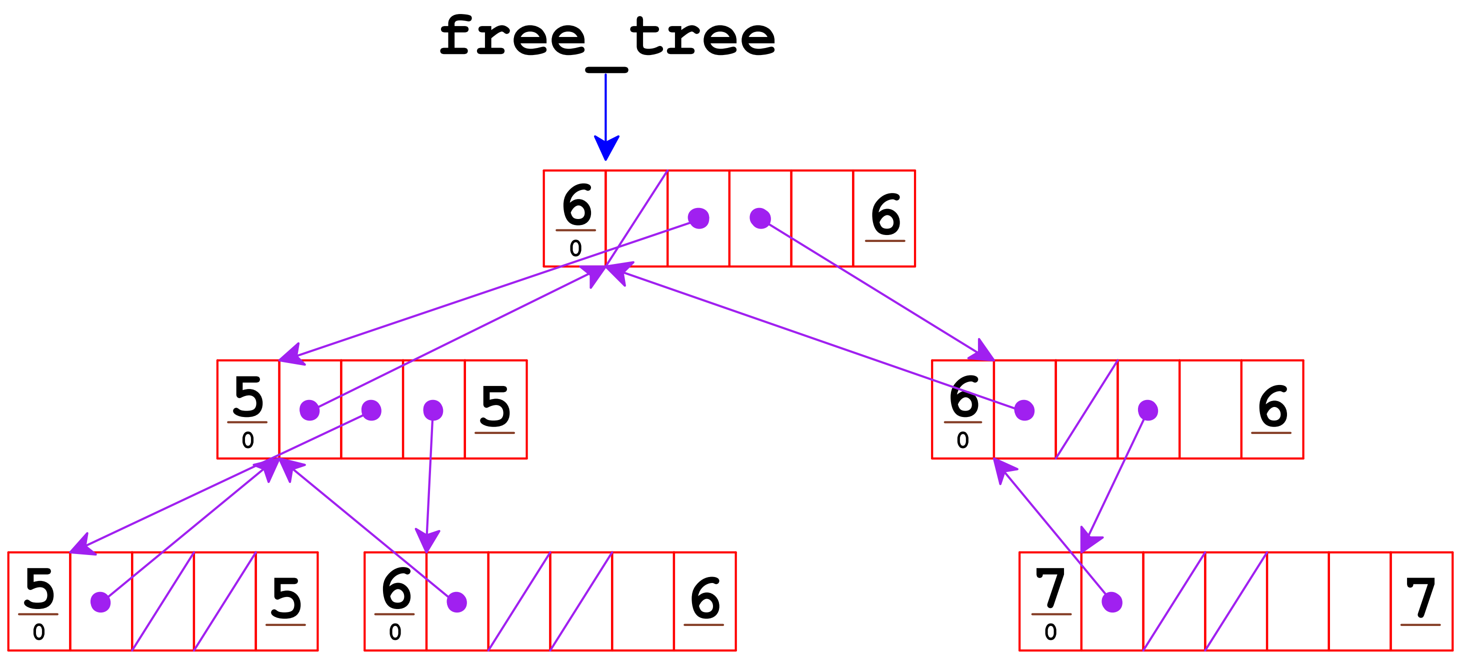
Segregated Free List

A ***segregated free list*** is an array of free lists, where each list has objects of a particular size class



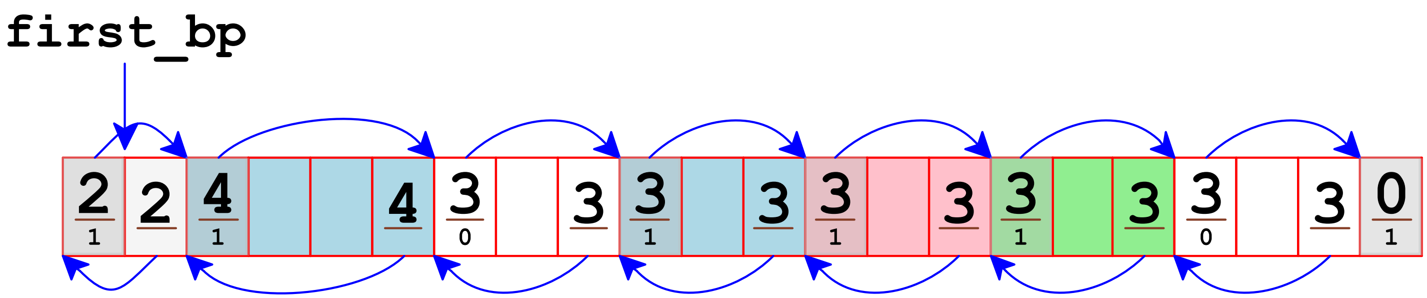
Balanced Binary Tree

Instead of a free list, a ***free tree*** can support efficient best-fit



Building an Allocator with **`mmap`**

Our allocator implementation so far depends on a contiguous heap:



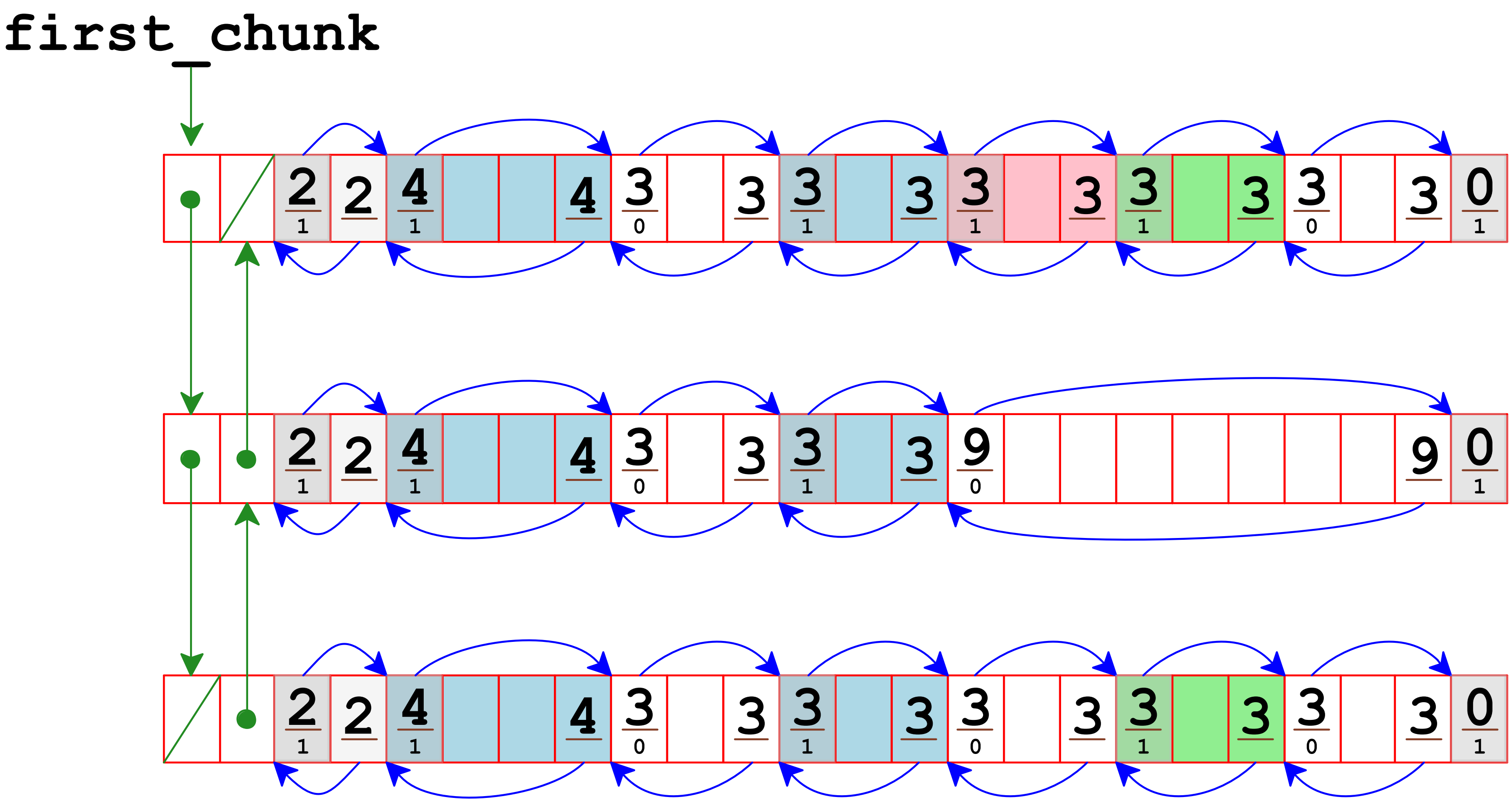
Allocators can use **`mmap`** instead of **`sbrk`**

using **`mmap`** works in more environments

Unlike **`sbrk`**, separate **`mmap`** calls don't always return contiguous addresses

Building an Allocator with **mmap**

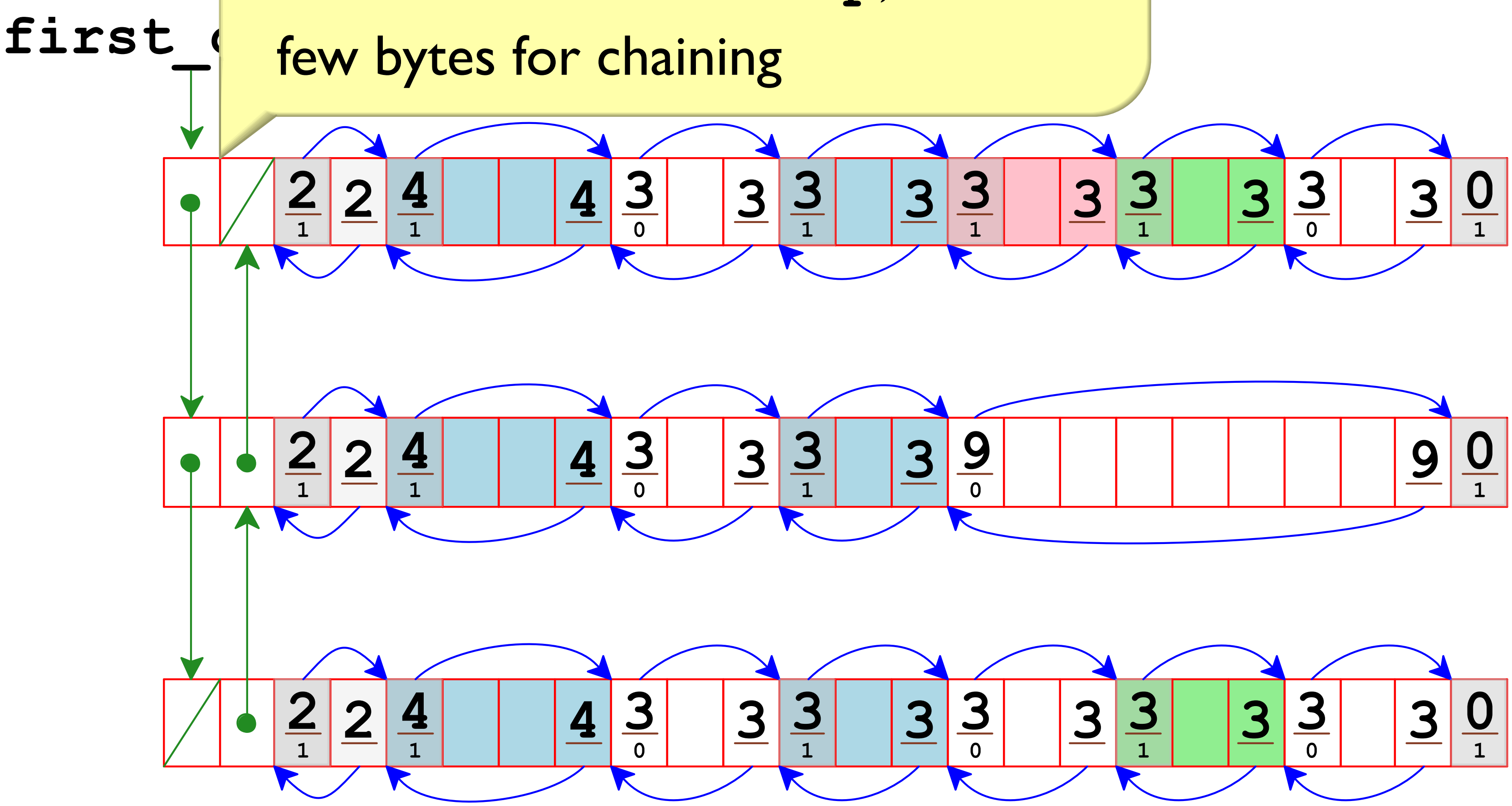
Chain together mapped pages as mini **sbrk**-like allocators



Building an Allocator with **mmap**

Chain together mapped pages as mini **sbrk**-like allocators

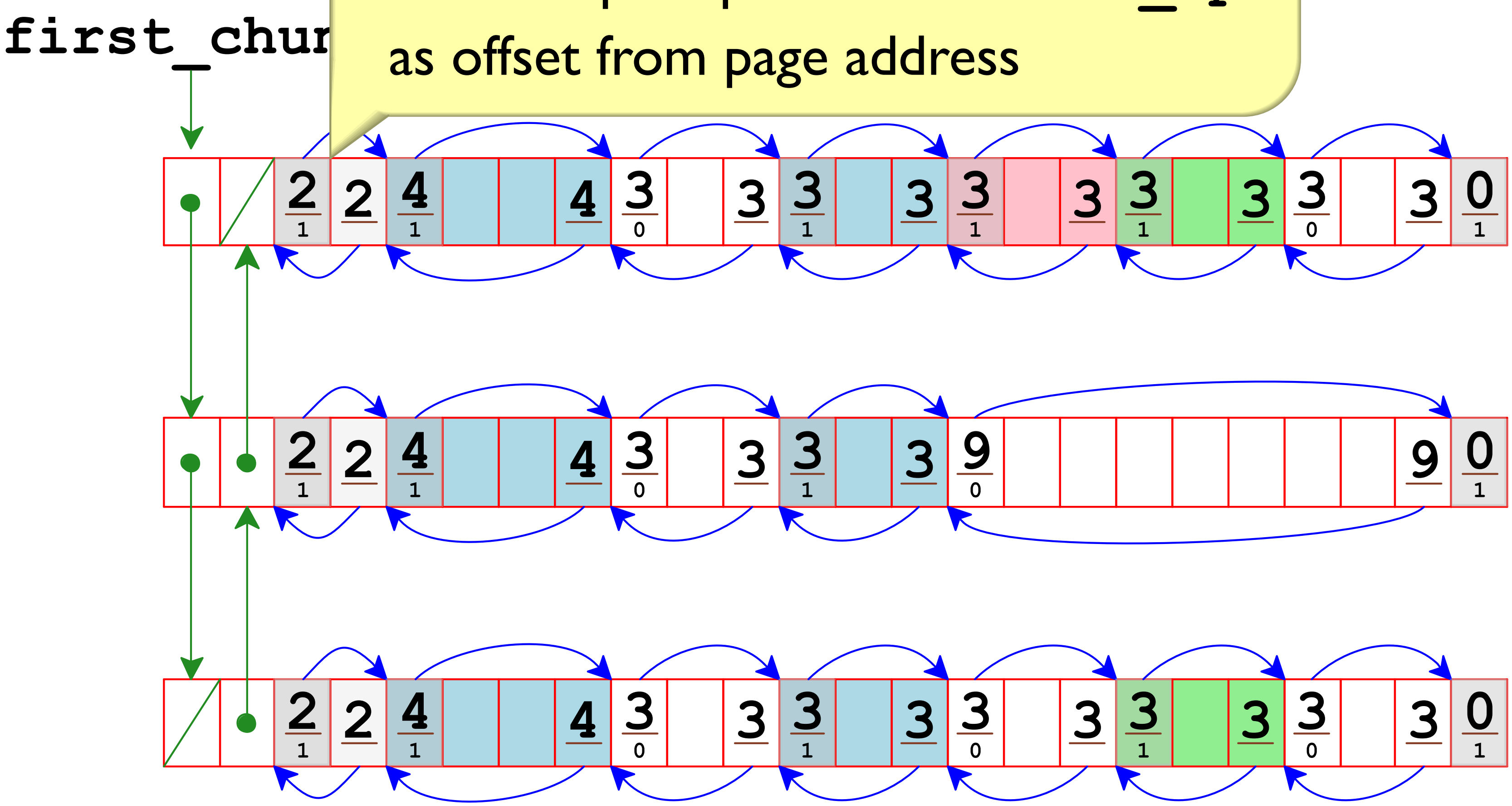
In each chunk from **mmap**, use first few bytes for chaining



Building an Allocator with **`mmap`**

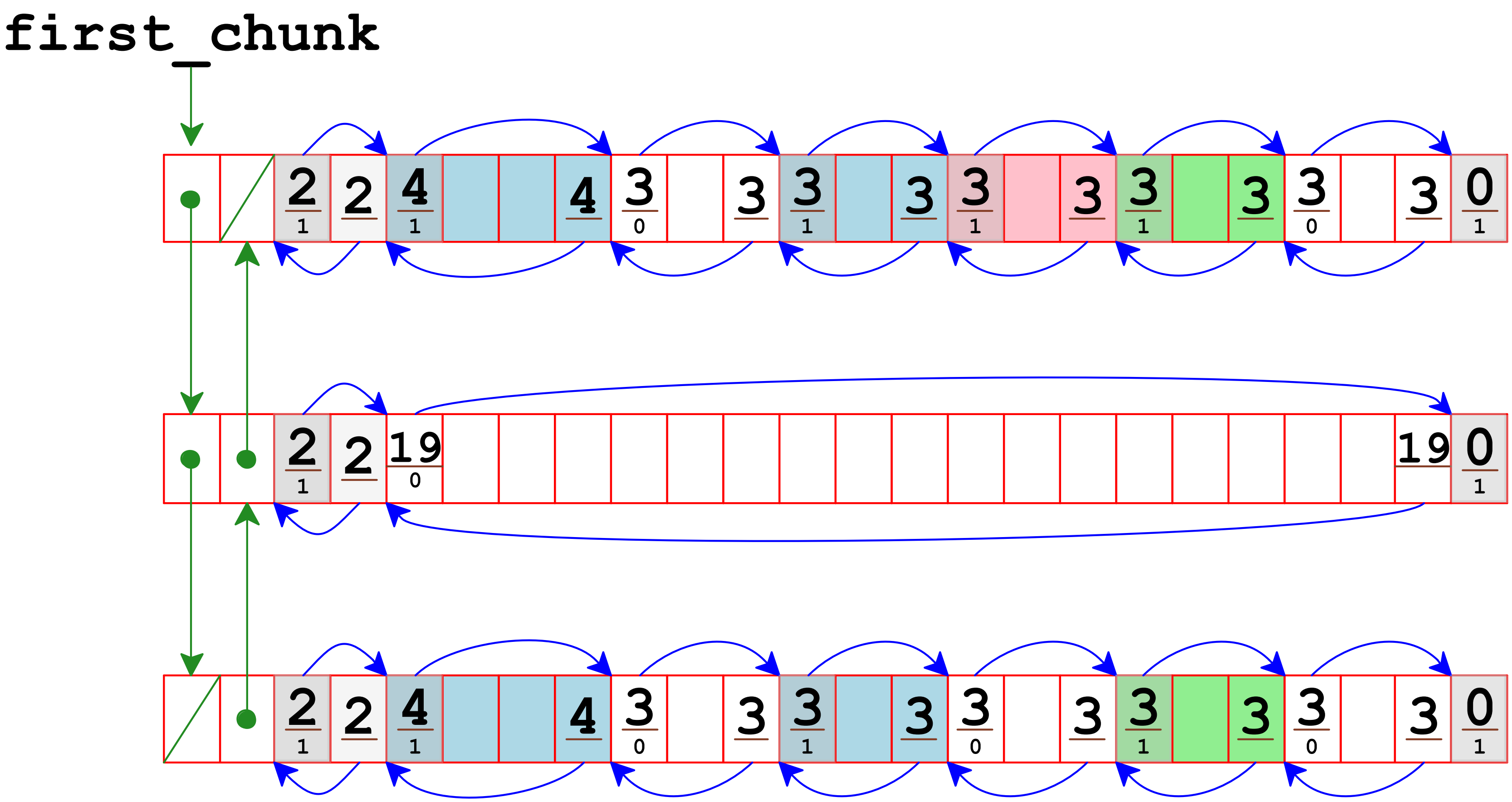
Chain together mapped pages as mini **`sbrk`**-like allocators

Can compute per-chunk **`first_bp`**
as offset from page address



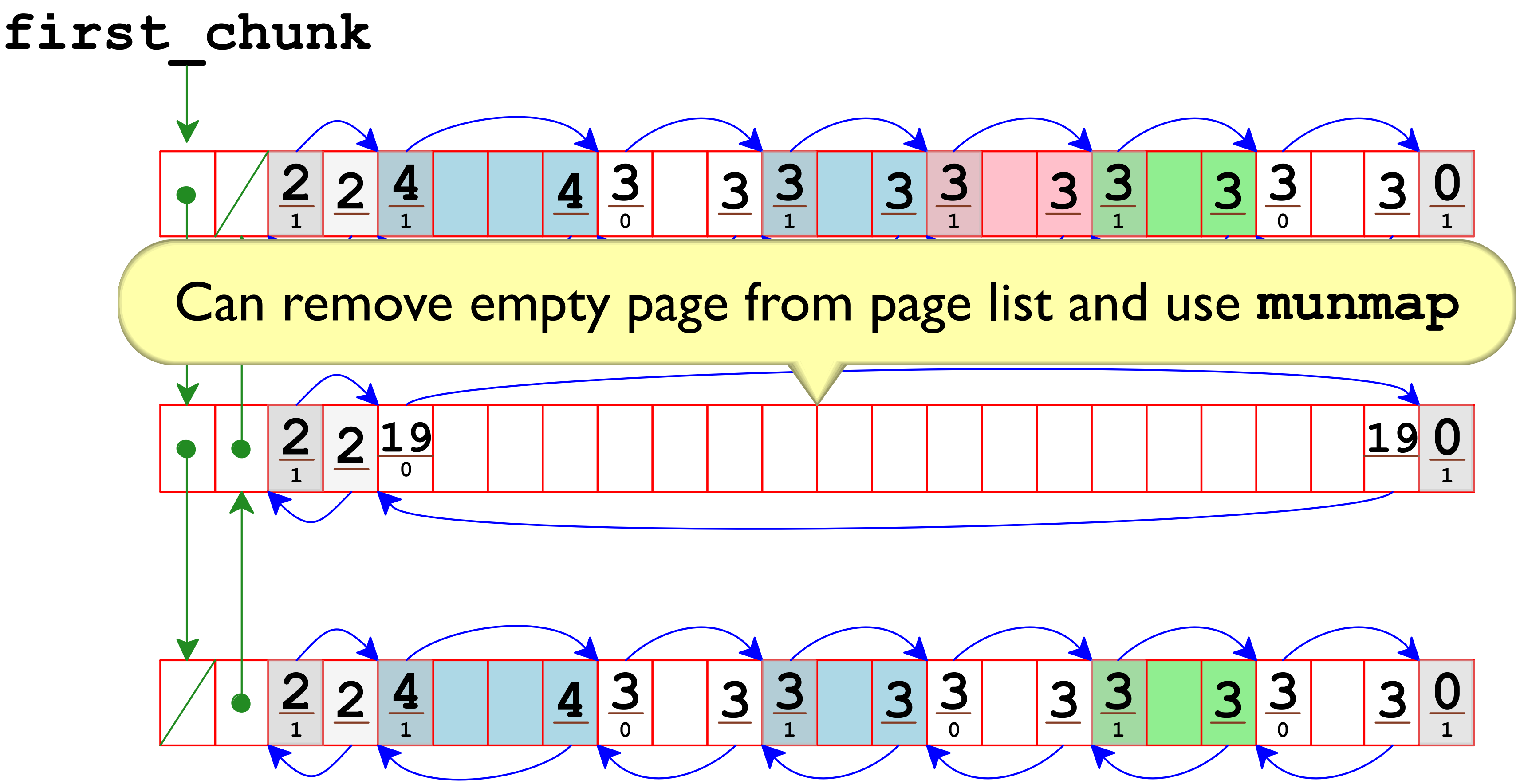
Building an Allocator with **mmap**

Chain together mapped pages as mini **sbrk**-like allocators



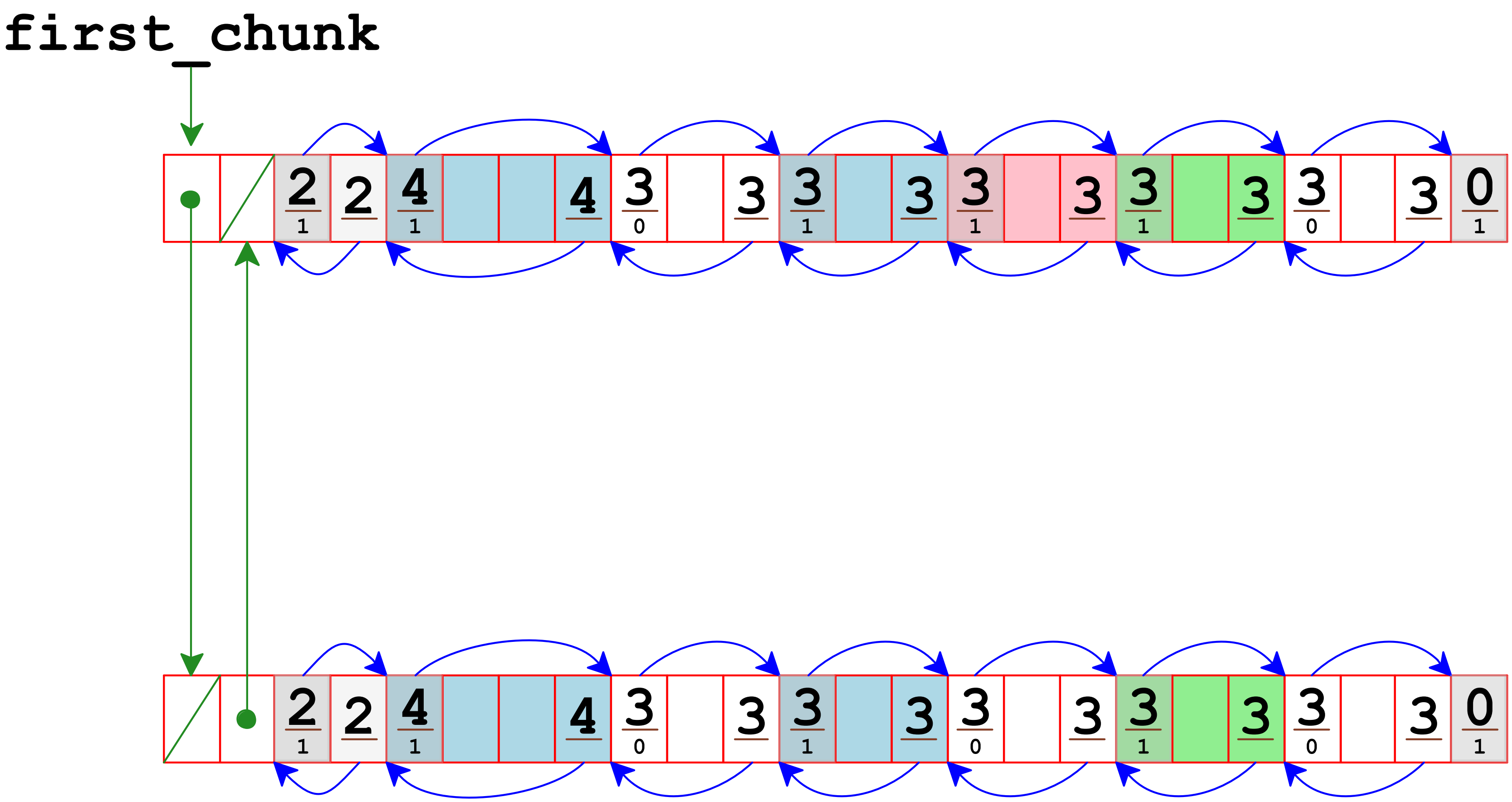
Building an Allocator with **mmap**

Chain together mapped pages as mini **sbrk**-like allocators



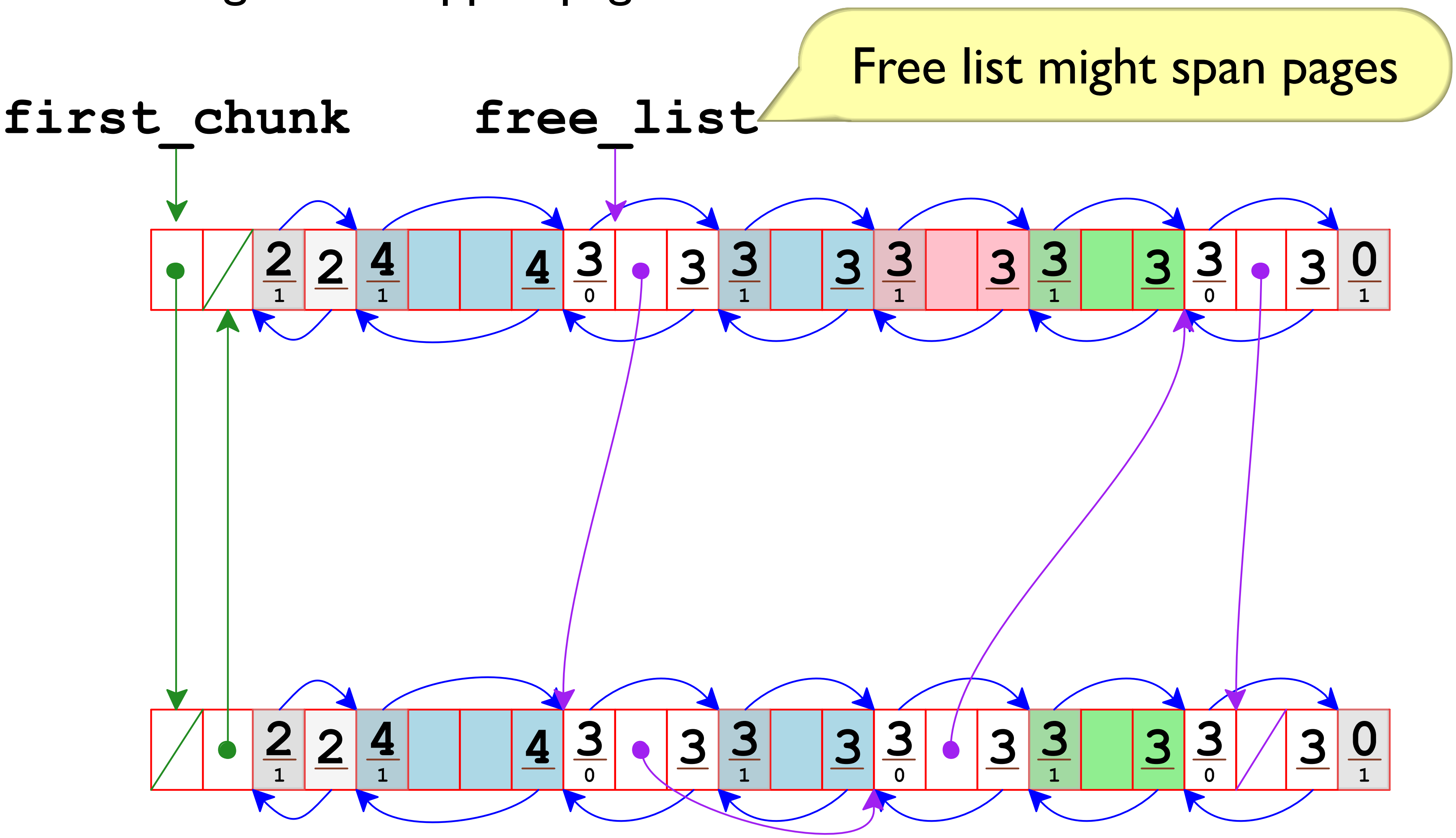
Building an Allocator with **mmap**

Chain together mapped pages as mini **sbrk**-like allocators



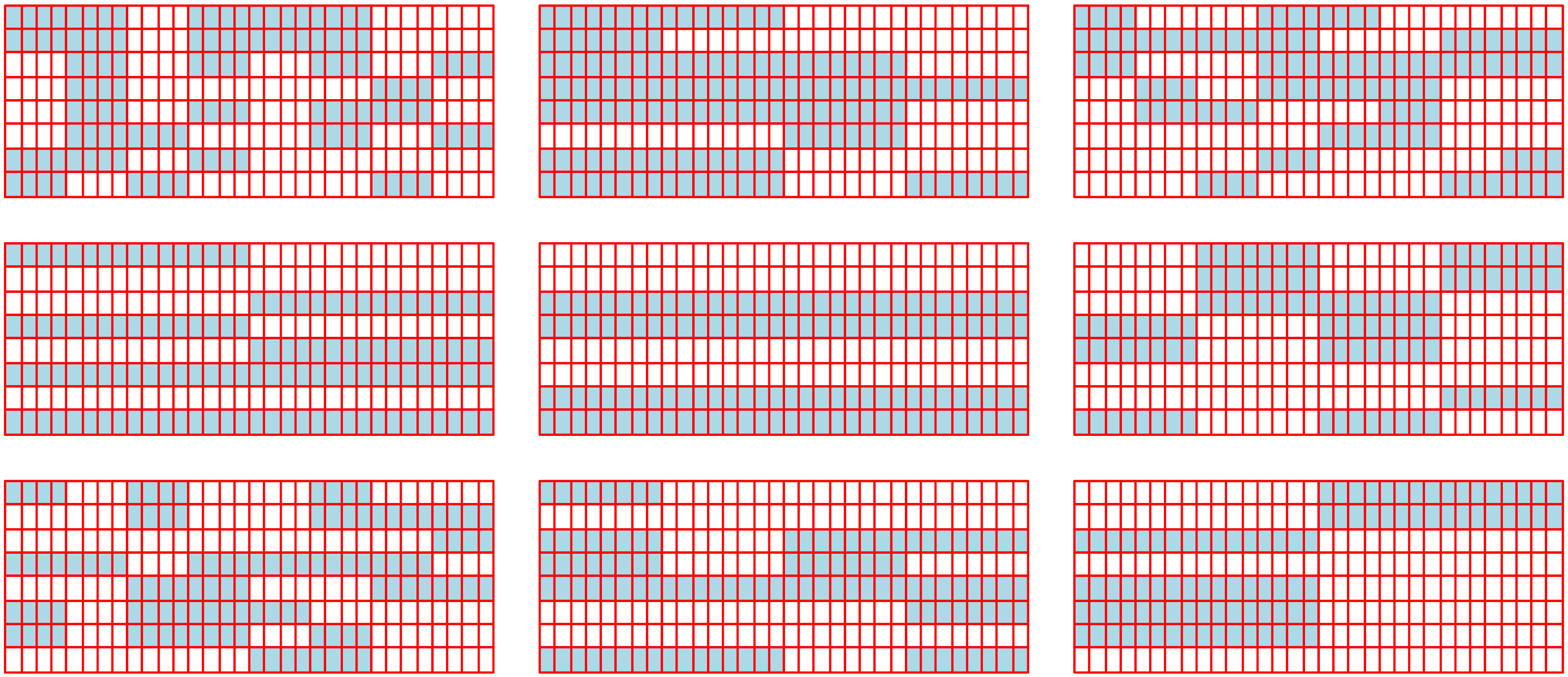
Building an Allocator with **mmap**

Chain together mapped pages as mini **sbrk**-like allocators



Something Completely Different: Segregated Allocation

Each chunk of memory has uniform-sized blocks



1 How does `free` know an allocated block's size?

Based on the address: the allocator keeps a mapping of address ranges to block sizes

Something Completely Different: Segregated Allocation

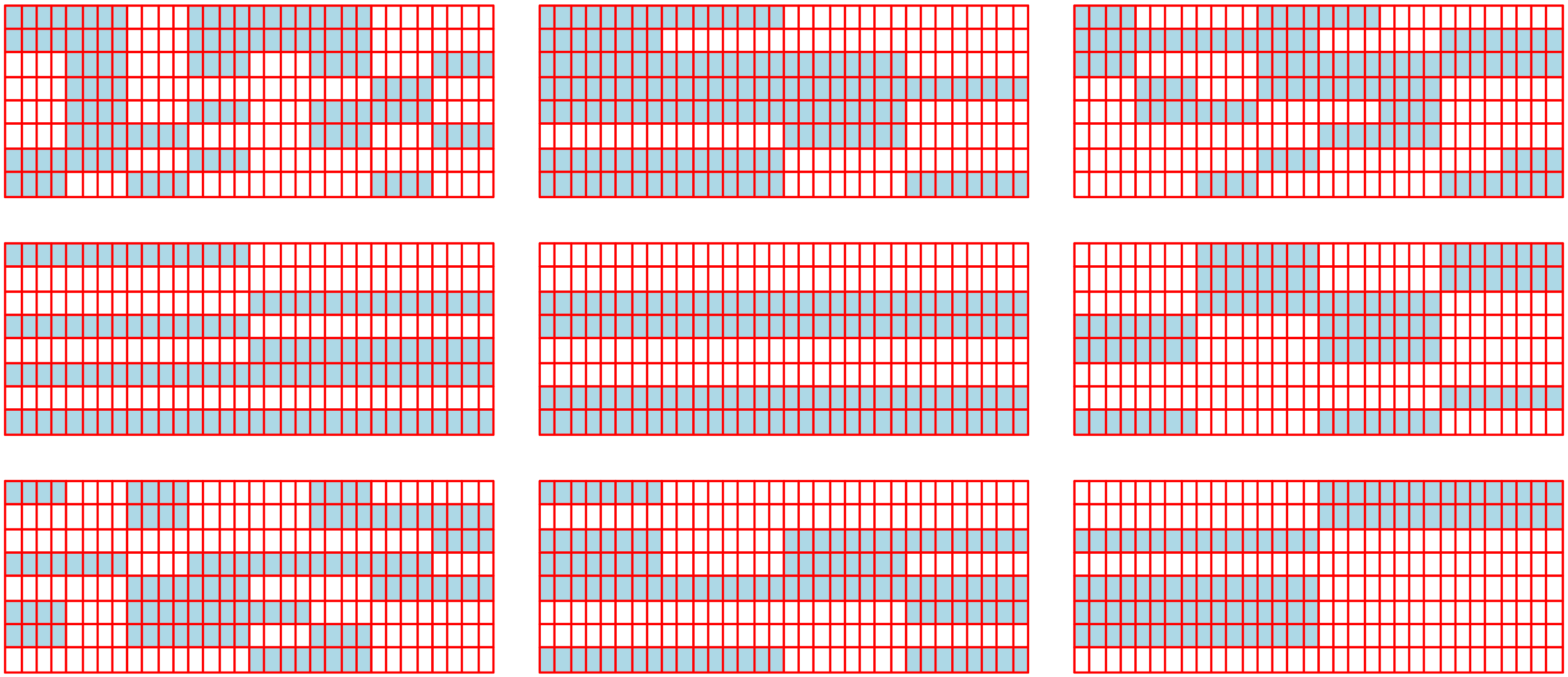
Each chunk of memory has uniform-sized blocks

❷ How is unallocated space represented?

Through a free list or chunk-specific bitmap

Something Completely Different: Segregated Allocation

Each chunk of memory has uniform-sized blocks

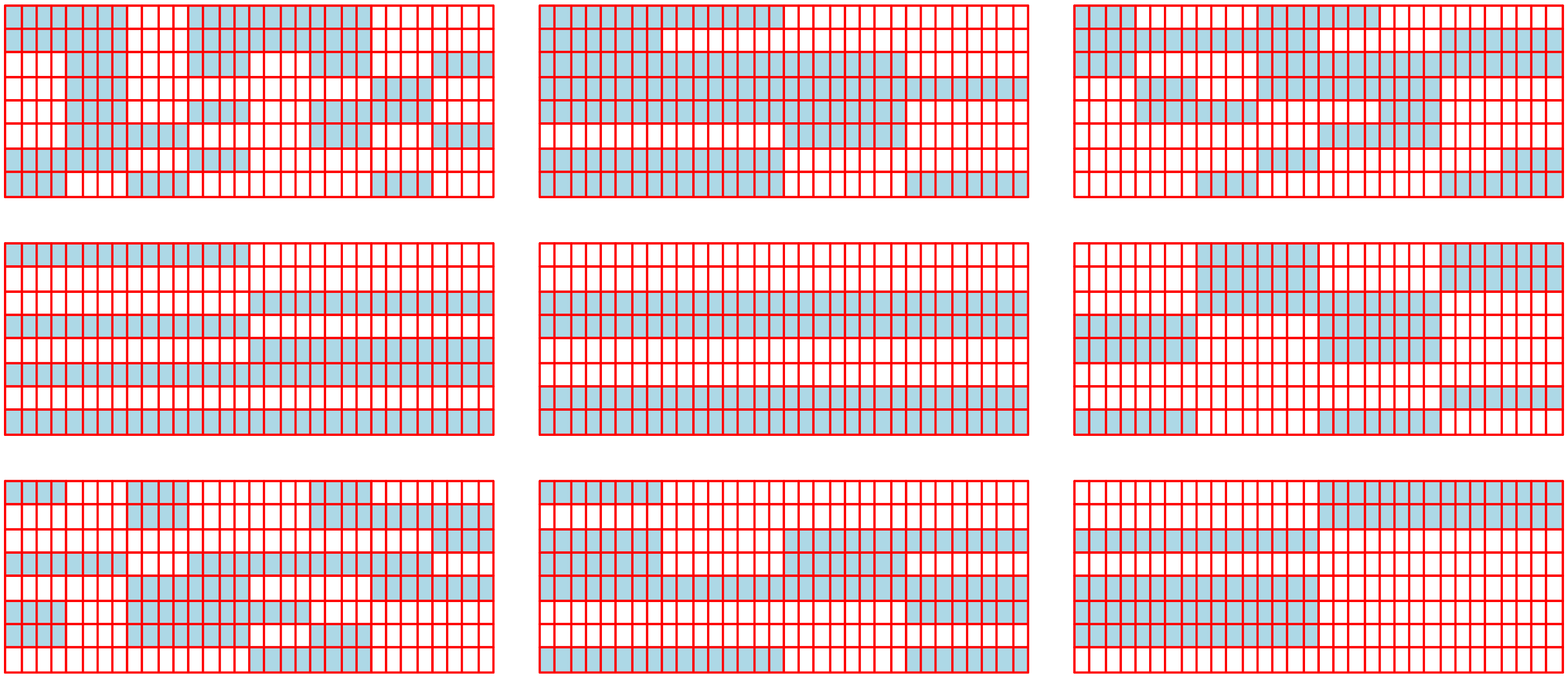


❸ How is unallocated space selected for each allocation?

Any unallocated block will work within a chunk that holds the block size

Something Completely Different: Segregated Allocation

Each chunk of memory has uniform-sized blocks



4 How finely is unallocated space tracked?

Block sizes must be rounded up to match some chunk's content

Something Completely Different: Segregated Allocation

Each chunk of memory has uniform-sized blocks

5 When are more pages needed from the kernel?

When no chunk for the block size has any unallocated blocks