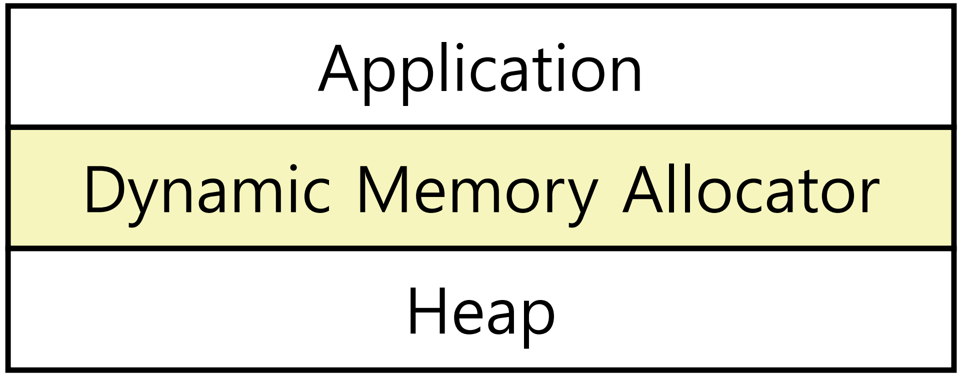
Virtual Memory
(Part 2)

HGU

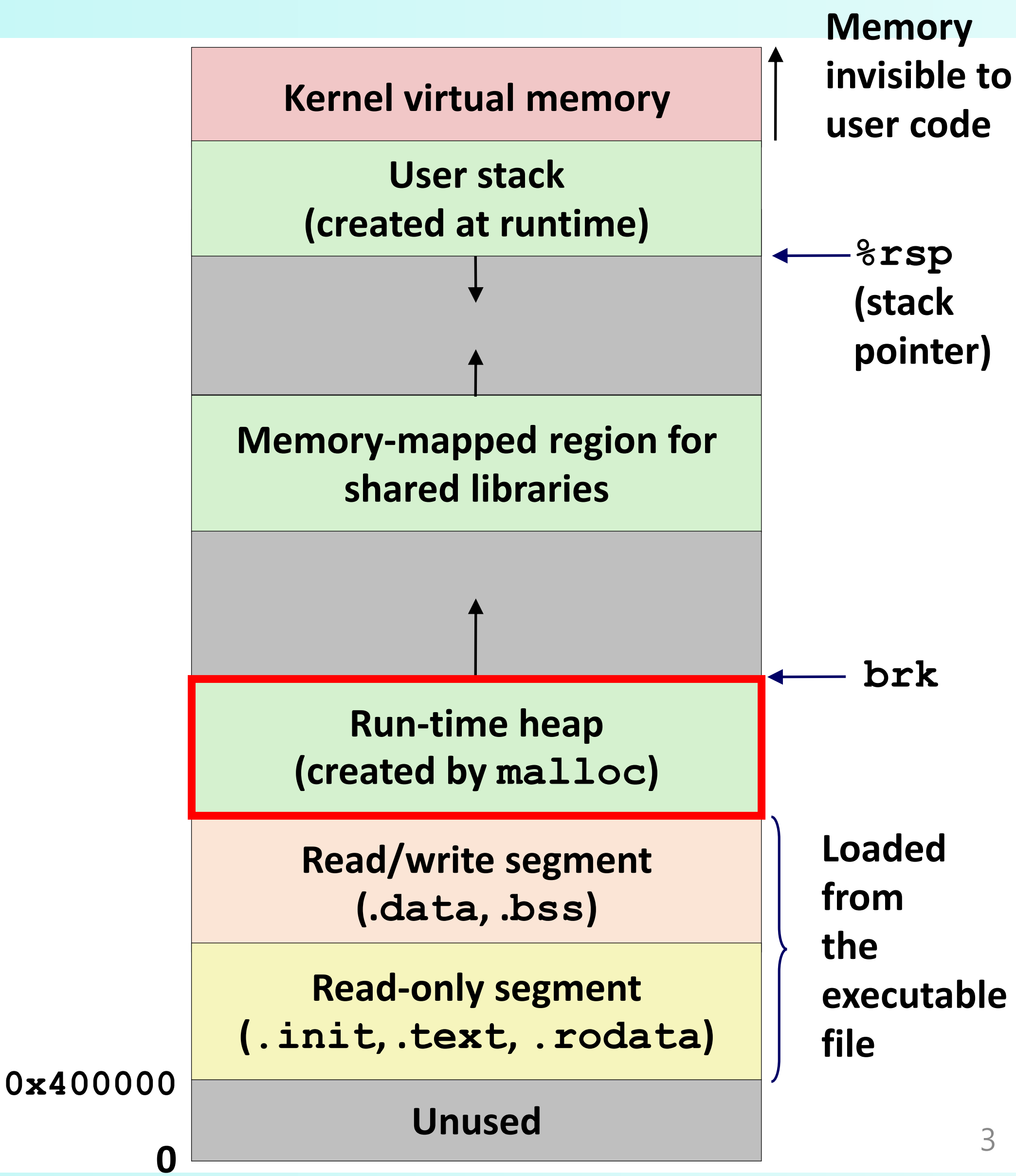
Contents

- Basic Concepts
- Implicit Free Lists
- Explicit Free Lists

Dynamic Memory Allocation



- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire virtual memory (VM) at run time.
    - for data structures whose size is only known at runtime
- Dynamic memory allocators manage an area of process VM known as the heap.



Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized blocks, which are either allocated or free

- Types of allocators
  - Explicit allocator: application allocates and frees space
    - E.g., `malloc` and `free` in C
  - Implicit allocator: application allocates, but does not free space
    - E.g., `new` and garbage collection in Java

The `malloc` Package

- `malloc()`, `calloc()`, `realloc()`: Allocate dynamic memory

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *_Nullable ptr, size_t size);
```

  - `malloc()`: it allocates *`size`* bytes and returns a pointer to the allocated memory. The memory is not initialized.
  - `calloc()`: it allocates memory for an array of *`nmemb`* elements of *`size`* bytes each and returns a pointer to the allocated memory. The memory is set to zero.
- `realloc()`: it changes the size of the memory block pointed to by *`ptr`* to *`size`*
- Return
  - Success: pointer to the allocated memory, which is suitably aligned for any type that fits into the requested size or less
  - Failure: NULL

https://man7.org/linux/man-pages/man3/malloc.3.html

The `malloc` Package

- `free()`: free dynamic memory

```
#include <stdlib.h>
void free(void *_Nullable ptr);
```

  - It frees the memory space pointed to by *`ptr`*, which must have been returned by a previous call to `malloc()` or related functions.
  - Otherwise, or if *`ptr`* has already been freed, undefined behavior occurs. If *`ptr`* is NULL, no operation is performed.
  - The `free()` function returns no value.

https://man7.org/linux/man-pages/man3/malloc.3.html

The `malloc` Package

- `brk()`, `sbrk()`: change data segment size

```
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t increment);
```

- Change the location of the *program break*, which defines the end of the process's data segment
  - the *program 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*
- `sbrk()` increments the program's data space by increment bytes
- Return
  - brk(): zero on success, -1 on error
  - sbrk(): previous program break, -1 on error

https://man7.org/linux/man-pages/man2/brk.2.html

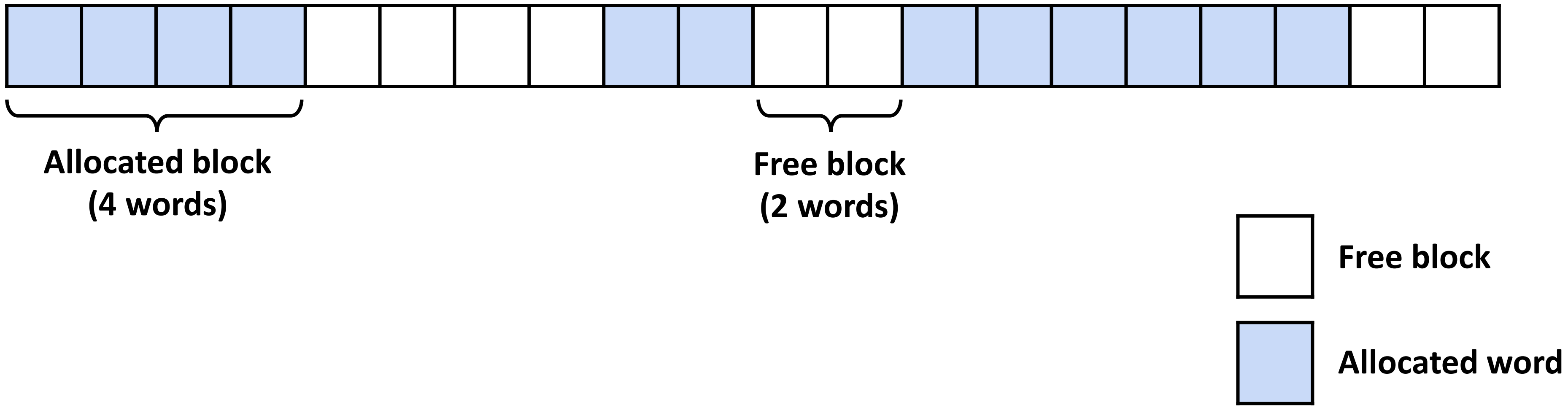
malloc Example

memprog.c

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

void foo(long n) {
    long i, *p;

    /* Allocate a block of n longs */
    p = (long *) malloc(n * sizeof(long));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;
    /* Do something with p */
    . . .
    /* Return allocated block to the heap */
    free(p);
}
```

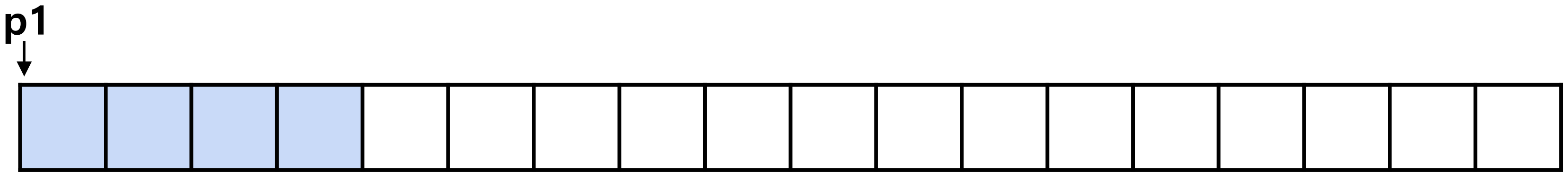
Visualization Conventions

- Show 8-byte words as squares
- Allocations are double-word aligned.

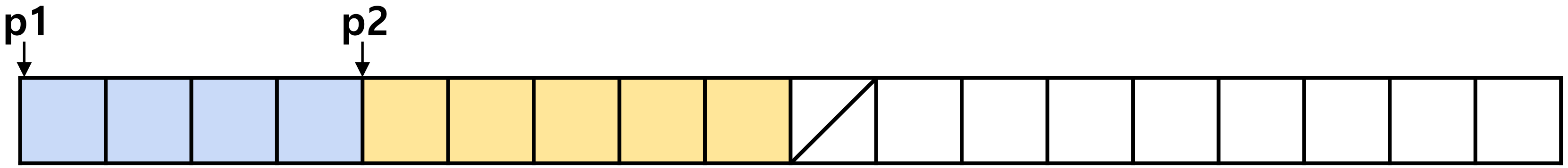


Allocation Example

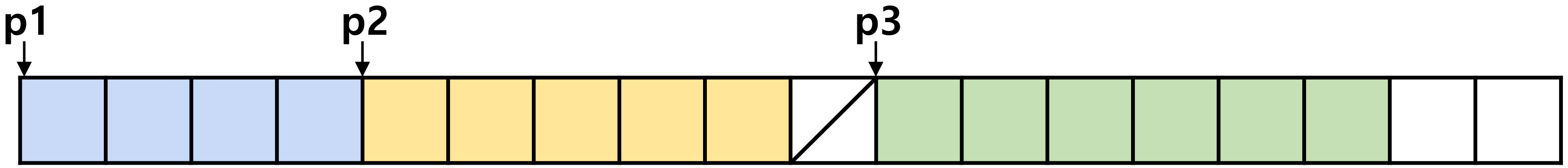
```
p1 = malloc(4*sizeof(int))
```



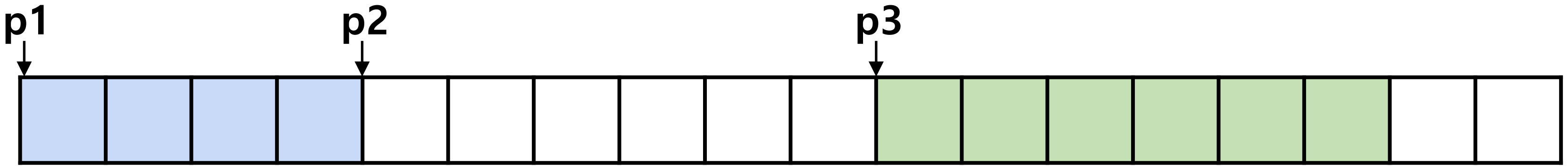

```
p2 = malloc(5*sizeof(int))
```



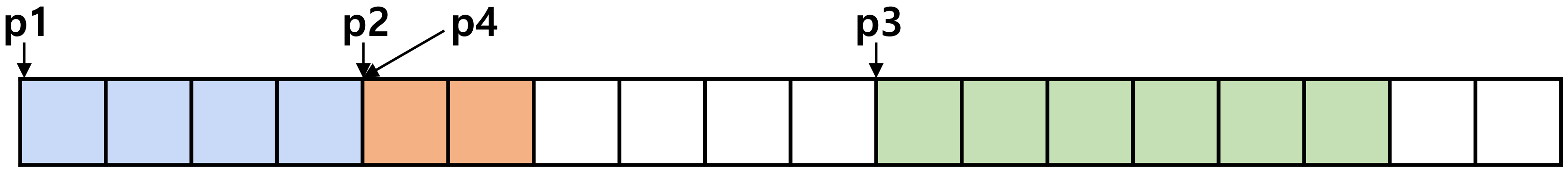

```
p3 = malloc(6*sizeof(int))
```




```
free(p2)
```




```
p4 = malloc(2*sizeof(int))
```



Why Dynamic Memory Allocation?

- The most important reason programs use dynamic memory allocation is that the size of certain data structures is often unknown until the program is running.
- With dynamic allocation, the program can request memory as needed during runtime, allowing for greater flexibility.
- In contrast, if the size of a data structure is fixed at compile-time (static allocation), any changes to its size would require modifying the source code and recompiling the program.
- This approach is less flexible and harder to maintain, especially for programs that handle varying or unpredictable data sizes.

Requirements

- Applications
  - Can issue arbitrary sequence of `malloc` and `free` requests
  - `free` request must be to a `malloc`'d block
- Explicit Allocators
  - Can't control number or size of allocated blocks
  - Must respond immediately to `malloc` requests
    - i.e., can't reorder or buffer requests
  - Must allocate blocks from free memory
    - i.e., can only place allocated blocks in free memory
  - Must align blocks so they satisfy all alignment requirements
    - 16-byte (x86-64) alignment on 64-bit systems
  - Can manipulate and modify only free memory
  - Can't move the allocated blocks once they are `malloc`'d
    - i.e., compaction/defragmention is not allowed. Why not?

Performance Goal: Throughput

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

- Goals: maximize throughput and peak memory utilization
  - These goals are often conflicting

- Throughput:
  - Number of completed requests per unit time
  - Example:
    - 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds
    - Throughput is 1,000 operations/second

Performance Goal: Memory Utilization

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

- Def: Aggregate payload P_k
  - `malloc(p)` results in a block with a payload of p bytes
  - After request R_k has completed, the aggregate payload P_k is the sum of currently allocated payloads
- Def: Current heap size H_k
  - Assume H_k is monotonically nondecreasing
    - i.e., heap only grows when allocator uses `sbrk`

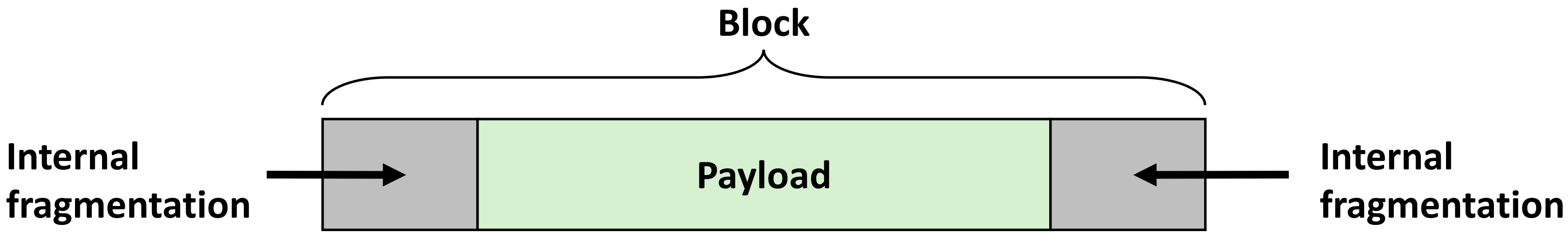
- Memory utilization: U_k
  - Ratio of program data (payload) too the current heap space
  - $U_k = (\max_{i \le k} P_i) / H_k$ ↔ Overhead: $O_k = H_k / (\max_{i \le k} P_i) - 1.0$

Fragmentation

- Poor memory utilization caused by fragmentation
  - Internal fragmentation
  - External fragmentation

Internal Fragmentation

- For a given block, internal fragmentation occurs if payload is smaller than block size

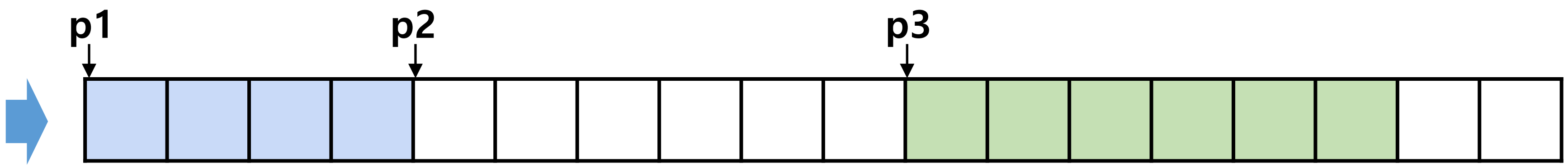


- Caused by
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - Explicit policy decisions
    (e.g., to return a big block to satisfy a small request)
- Depends only on the pattern of previous requests
  - Thus, easy to measure

External Fragmentation

- External fragmentation occurs when there is enough aggregate heap memory, but no single free block is large enough

```
p4 = malloc(7*sizeof(int))
```



What would happen now?

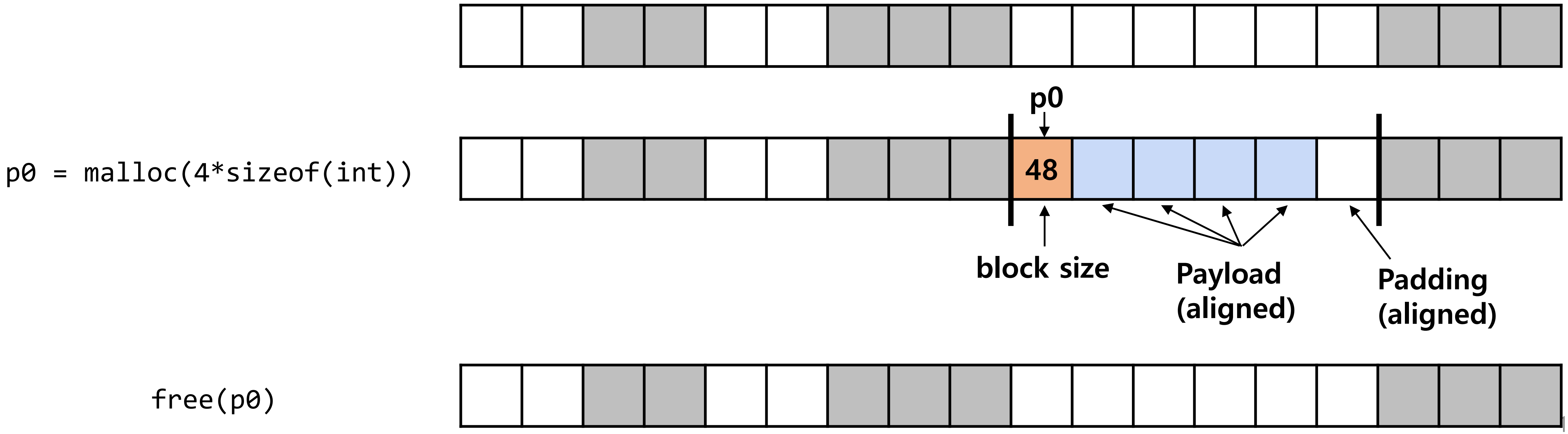
- Amount of external fragmentation depends on the pattern of future requests
  - Thus, difficult to measure

Implementation Issues

- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation -- many might fit?
- How do we reuse a block that has been freed?

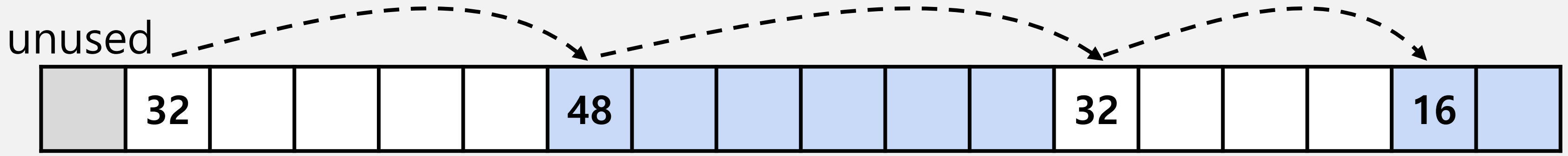
Knowing How Much to Free

- Standard method
  - Keep the length (in bytes) of a block in the word *preceding* the block.
    - Including the header
    - This word is often called the ***header field*** or ***header***
  - Requires an extra word for every allocated block



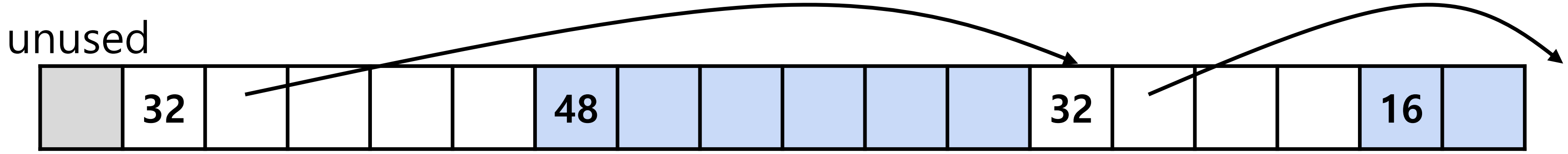
Keeping Track of Free Blocks

- Method 1: ***Implicit list*** using length — links all blocks



Need to tag each block as allocated/free

- Method 2: ***Explicit list*** among the free blocks using pointers



Need space for pointers

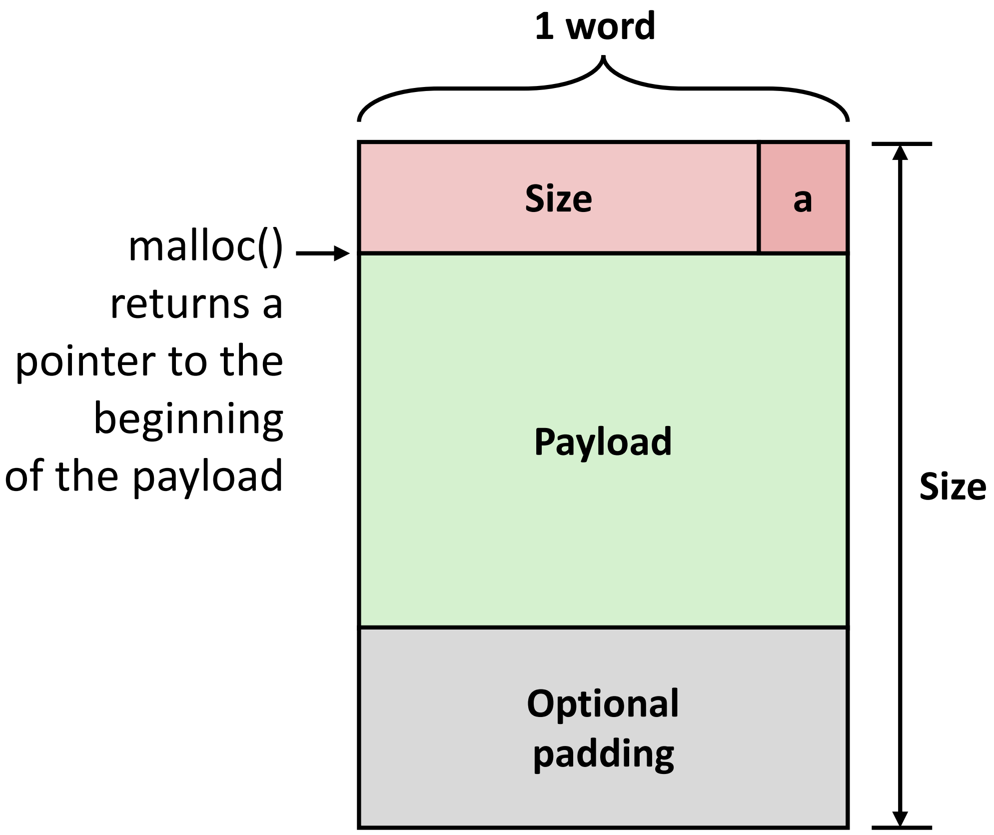
- Method 3: ***Segregated free list***
  - Different free lists for different size classes
- Method 4: ***Blocks sorted by size***
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Contents

- Basic Concepts
- Implicit Free Lists
- Explicit Free Lists

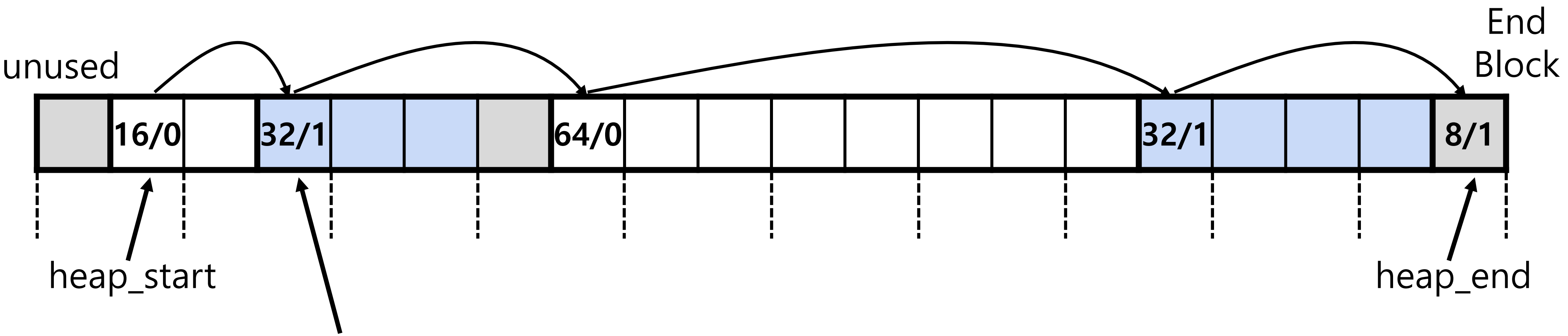
Implicit Free List

- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!
- Standard trick
  - When blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as an allocated/free flag
  - When reading the Size word, must mask out this bit

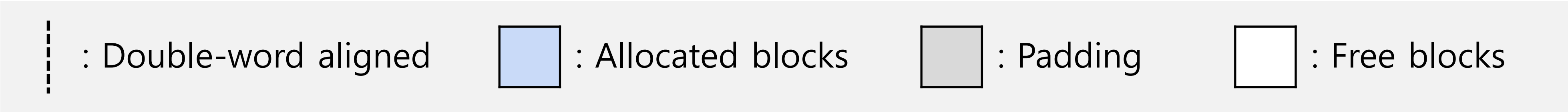


- **Block Size: total block size**
- **a = 1: Allocated block**
  a = 0: Free block
- **Payload: application data (allocated blocks only)**

Detailed Implicit Free List Example



Headers: "size in words / allocated bit"
Headers are at non-aligned positions → Payloads are aligned



Implicit List: Data Structures

- Block declaration

```
typedef uint64_t word_t;
```

```
typedef struct block {
    word_t header;
    unsigned char payload[0];    // Zero length array
} block_t;
```

- Getting payload from block pointer   // block_t *block

```
return (void *) (block->payload);
```

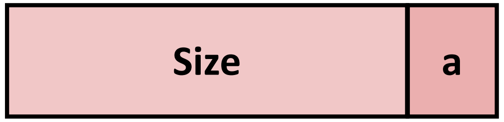
- Getting header from payload   // bp points to a payload

```
return (block_t *) ((unsigned char *) bp - offsetof(block_t, payload));
```

C function `offsetof(struct, member)` returns offset of member within struct

https://man7.org/linux/man-pages/man3/offsetof.3.html

Implicit List: Header access



- Getting allocated bit from header

```
return header & 0x1;
```

- Getting size from header

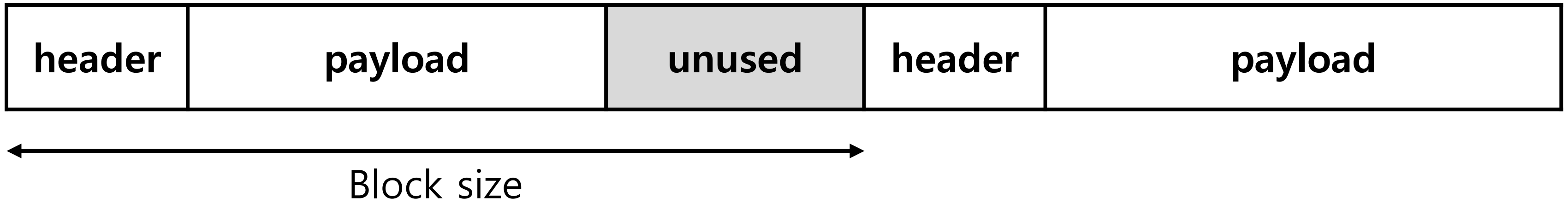
```
return header & ~0xfL;
```

- Initializing header

```
// block_t *block
```

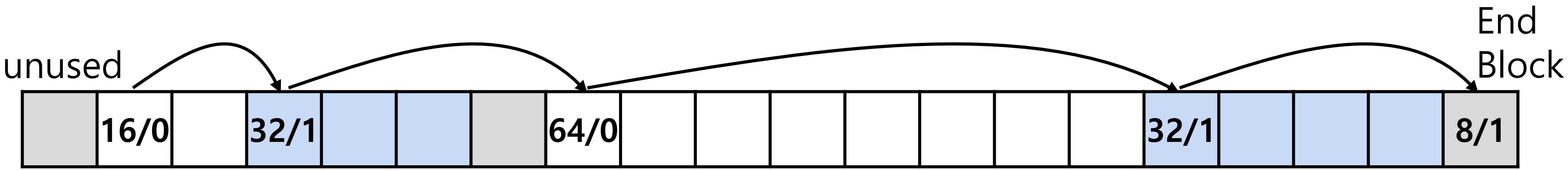
```
block->header = size | alloc;
```

Implicit List: Traversing list



- Find next block

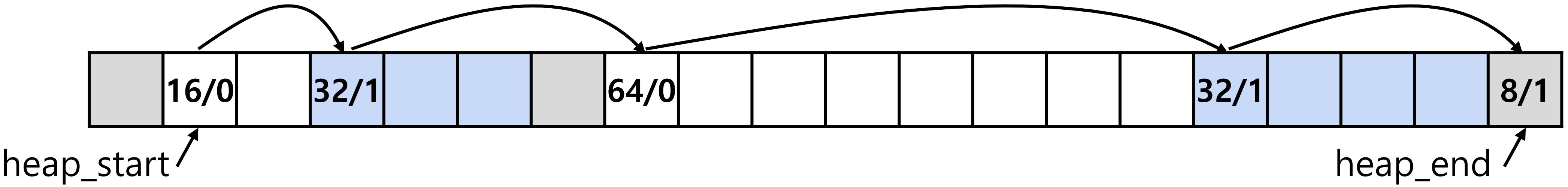
```
static block_t *find_next(block_t *block)
{
    return (block_t *) ((unsigned char *) block + get_size(block));
}
```



Implicit List: Finding a Free Block

- First fit:
  - Search list from beginning, choose first free block that fits:
  - Finding space for `asize` bytes (including header):

```
static block_t *find_fit(size_t asize)
{
    block_t *block;
    for (block = heap_start; block != heap_end; block = find_next(block))
    {
      if (!(get_alloc(block)) && (asize <= get_size(block)))
         return block;
    }
    return NULL; // No fit found
}
```

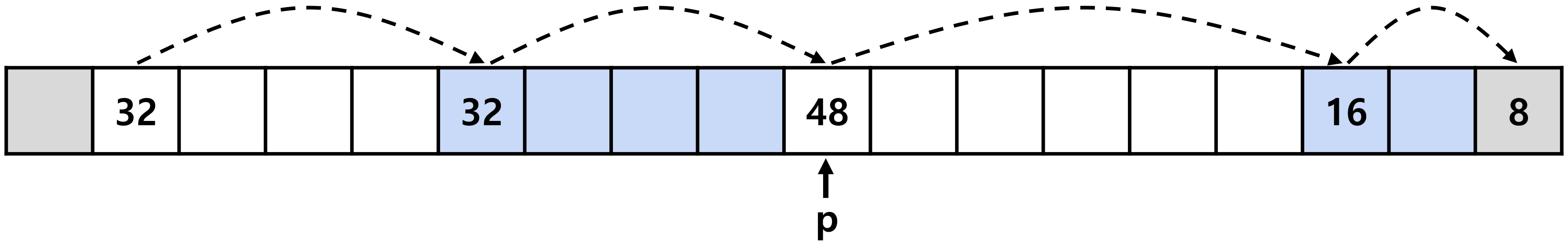


Implicit List: Finding a Free Block

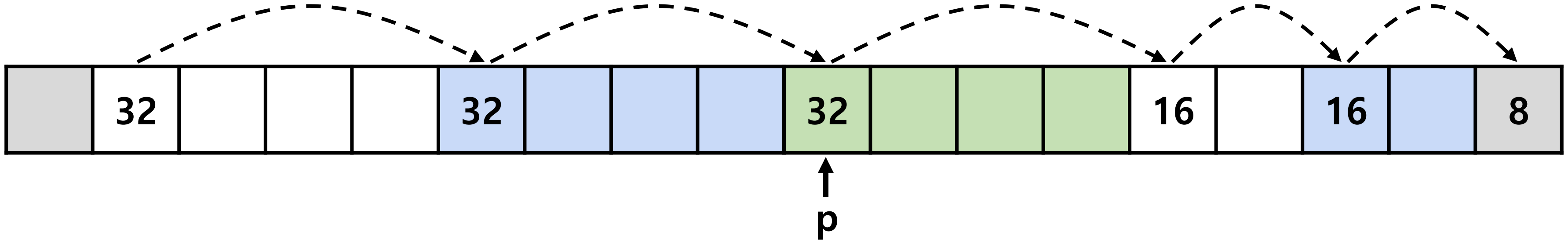
- First fit:
  - Search list from beginning, choose first free block that fits:
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- Next fit:
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- Best fit:
  - Search the list, choose the best free block: fits, with fewest bytes left over
  - Keeps fragments small—usually improves memory utilization
  - Will typically run slower than first fit
  - Still a greedy algorithm. No guarantee of optimality

Implicit List: Allocating in Free Block

- Allocating in a free block: *splitting*
  - Since allocated space might be smaller than free space, we might want to split the block




```
split_block(p, 32)
```



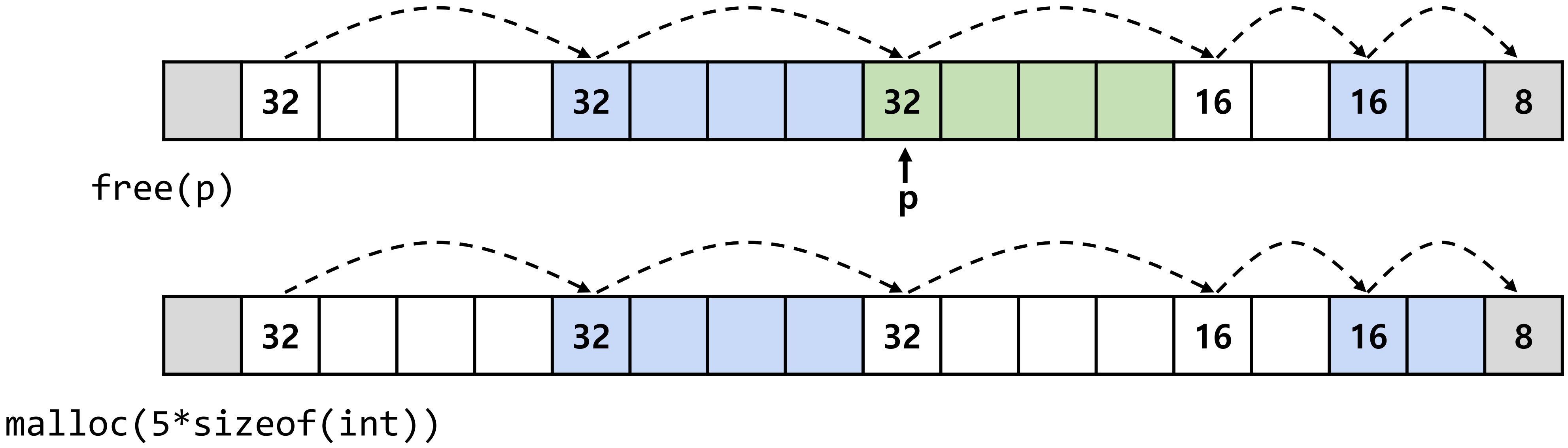
Implicit List: Splitting Free Block

split_block(p, 32)


```
// Warning: This code is incomplete
static void split_block(block_t *block, size_t asize)
{
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next, block_size - asize, false);
}
```

Implicit List: Freeing a Block

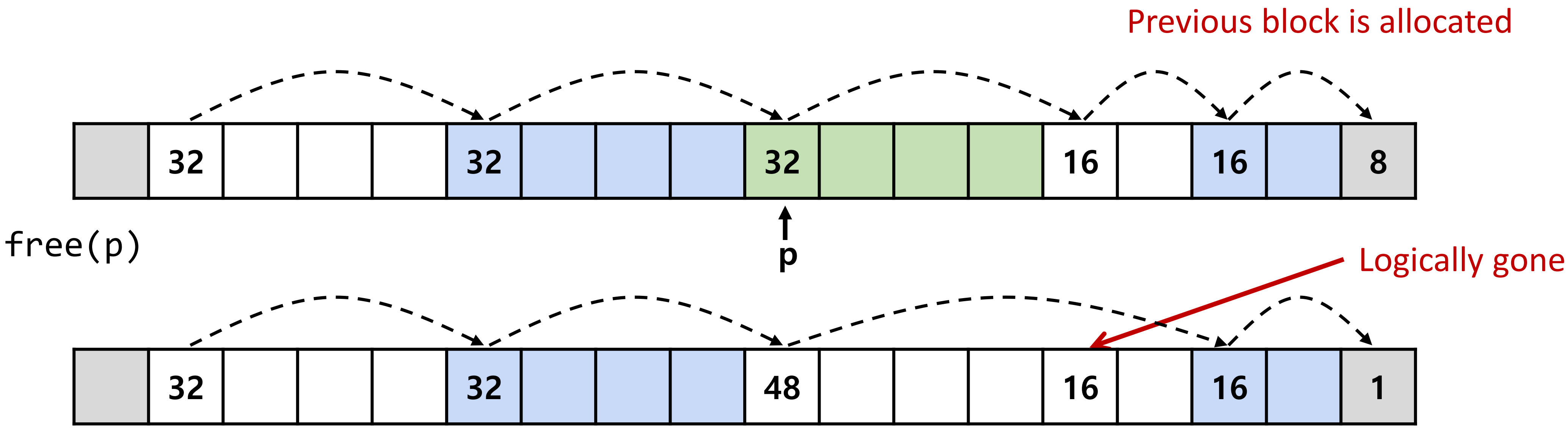
- Simplest implementation:
  - Need only clear the "allocated" flag
  - But can lead to "false fragmentation"



There is enough contiguous free space, but the allocator won't be able to find it!

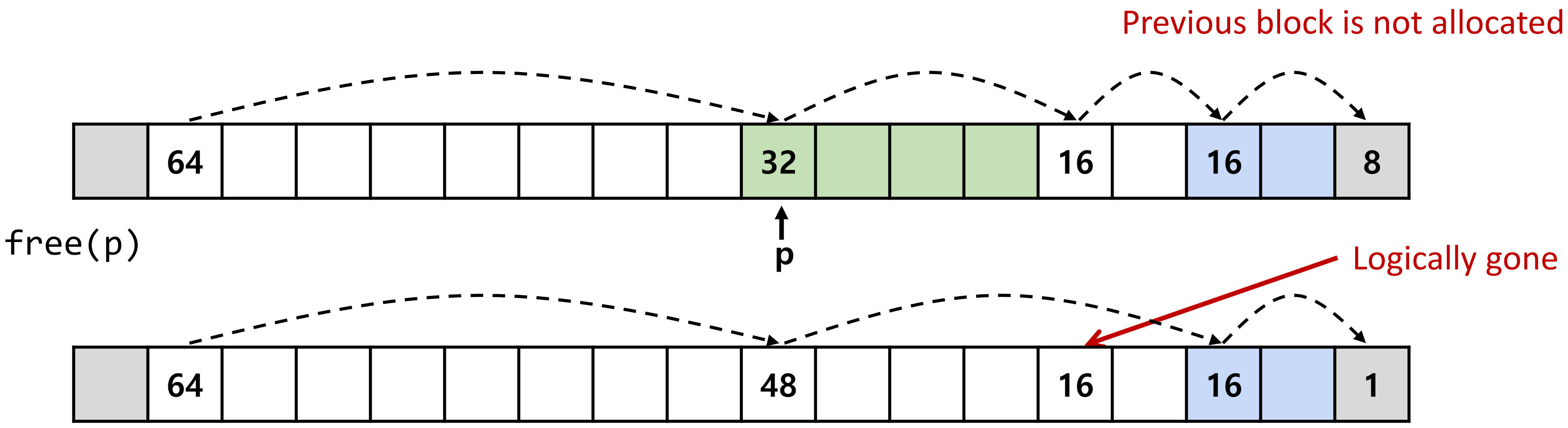
Implicit List: Coalescing

- Join (coalesce) with next/previous blocks, if they are free
  - Coalescing with next block



Implicit List: Finding a Free Block

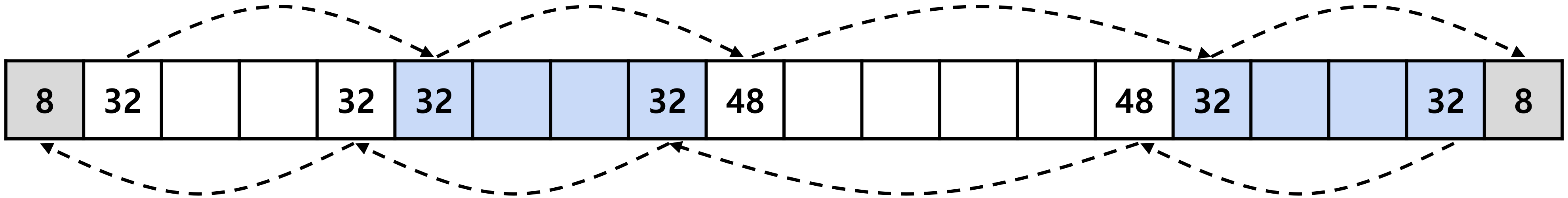
- Join (coalesce) with next/previous blocks, if they are free
  - Coalescing with next block

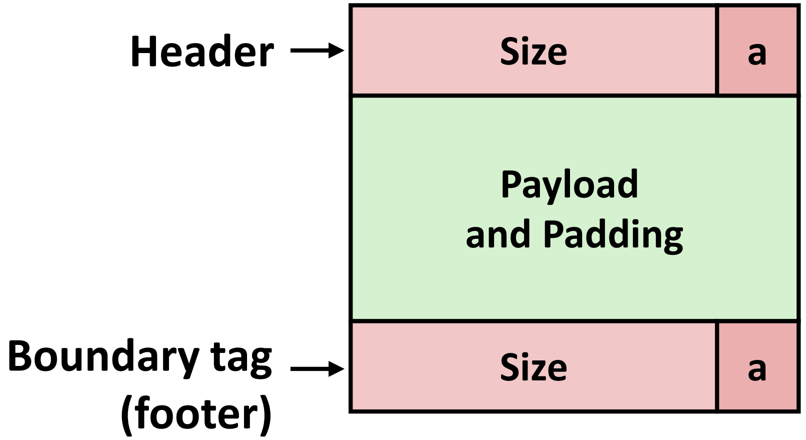


- Need to coalesce with previous block. But how?
  - How do we know where it starts?
  - How can we determine whether its allocated?

Implicit List: Bidirectional Coalescing

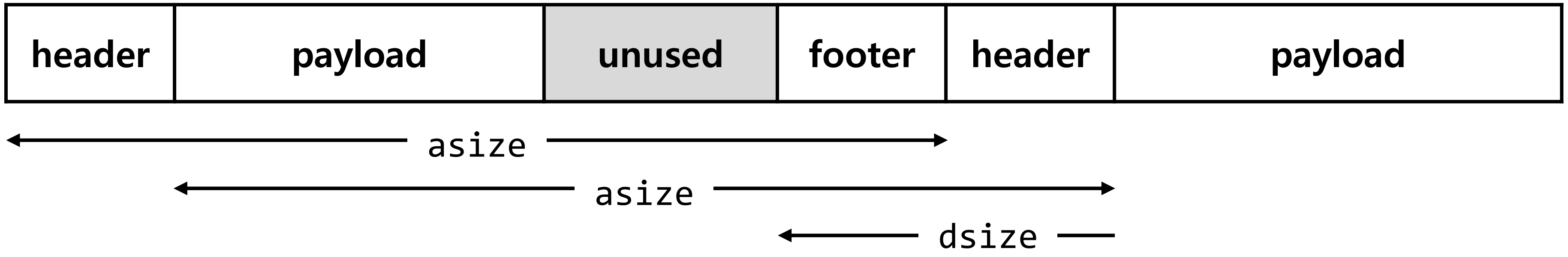
- Boundary tags [Knuth73]
  - Replicate size/allocated word at "bottom" (end) of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique!





- Block Size: total block size
- a = 1: Allocated block
  a = 0: Free block
- Payload: application data
  (allocated blocks only)

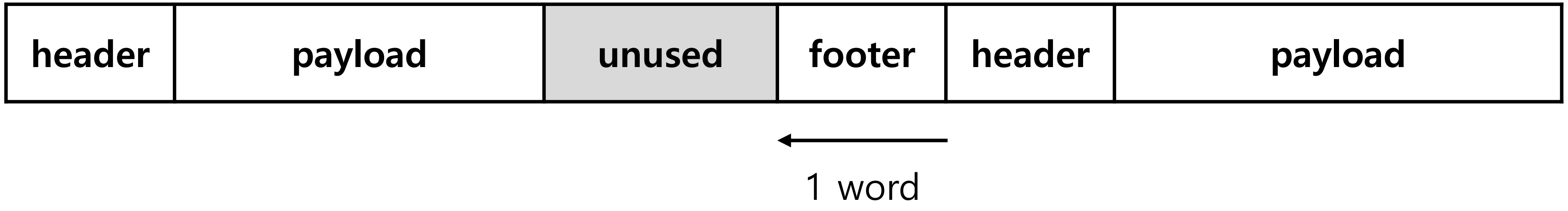
Implementation with Footers



- Locating footer of current block

```
const size_t dsize = 2*sizeof(word_t);

static word_t *header_to_footer(block_t *block)
{
    size_t asize = get_size(block);
    return (word_t *) (block->payload + asize - dsize);
}
```

Implementation with Footers



- Locating footer of previous block

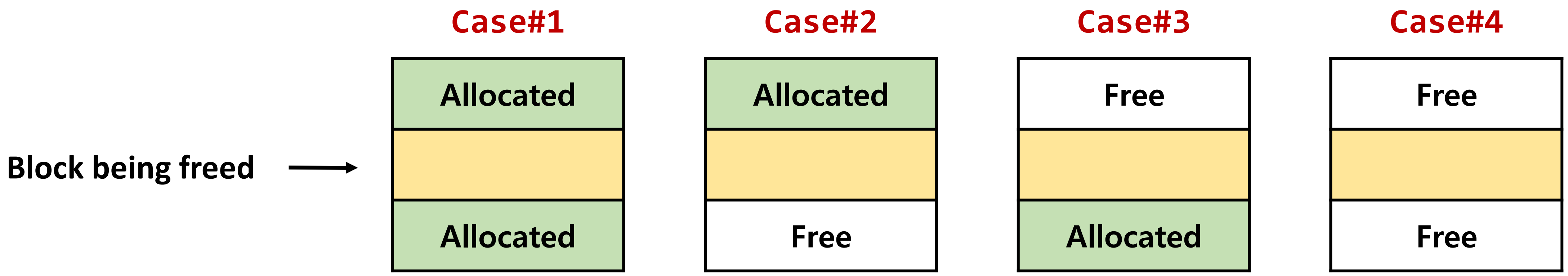
```
static word_t *find_prev_footer(block_t *block)
{
    return &(block->header) - 1;
}
```

Splitting Free Block: Full Version

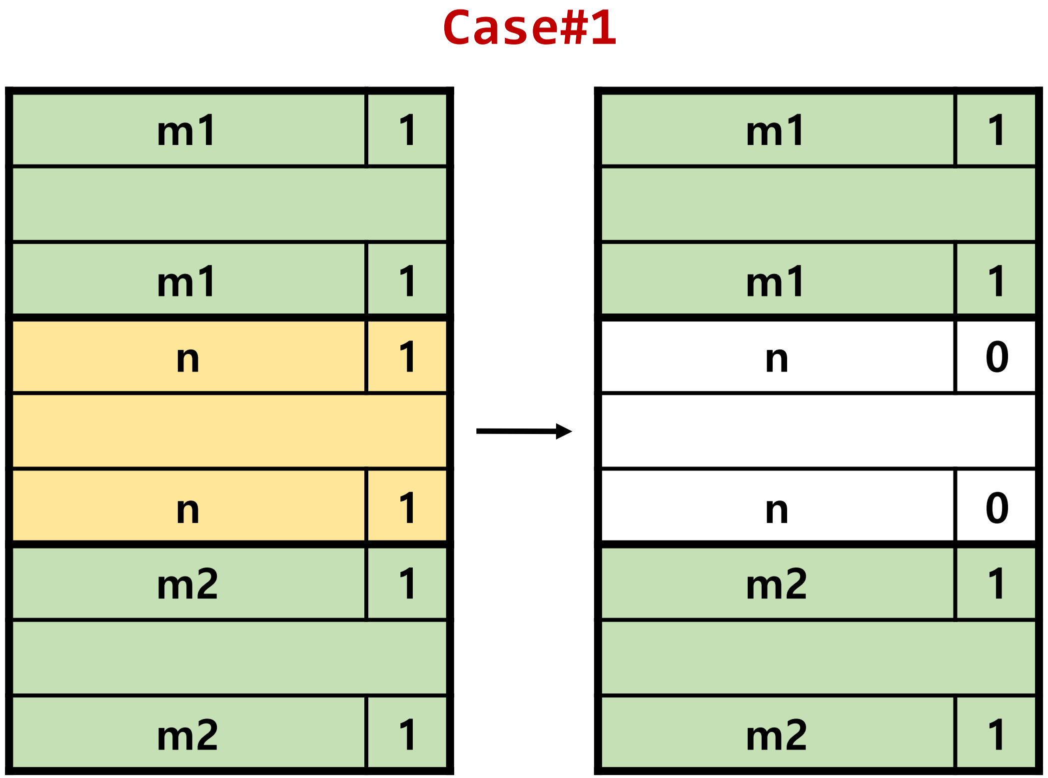
```
split_block(p, 32)
```

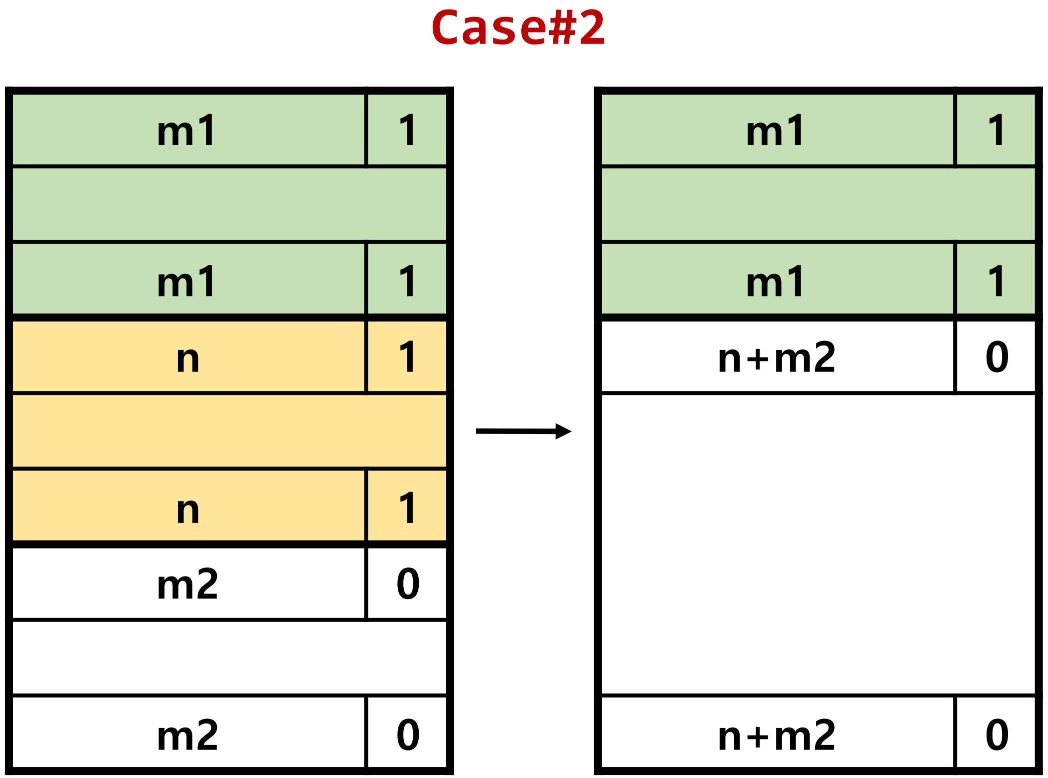

```
static void split_block(block_t *block, size_t asize){
    size_t block_size = get_size(block);

    if ((block_size - asize) >= min_block_size) {
        write_header(block, asize, true);
        write_footer(block, asize, true);
        block_t *block_next = find_next(block);
        write_header(block_next, block_size - asize, false);
        write_footer(block_next, block_size - asize, false);
    }
```

Constant Time Coalescing

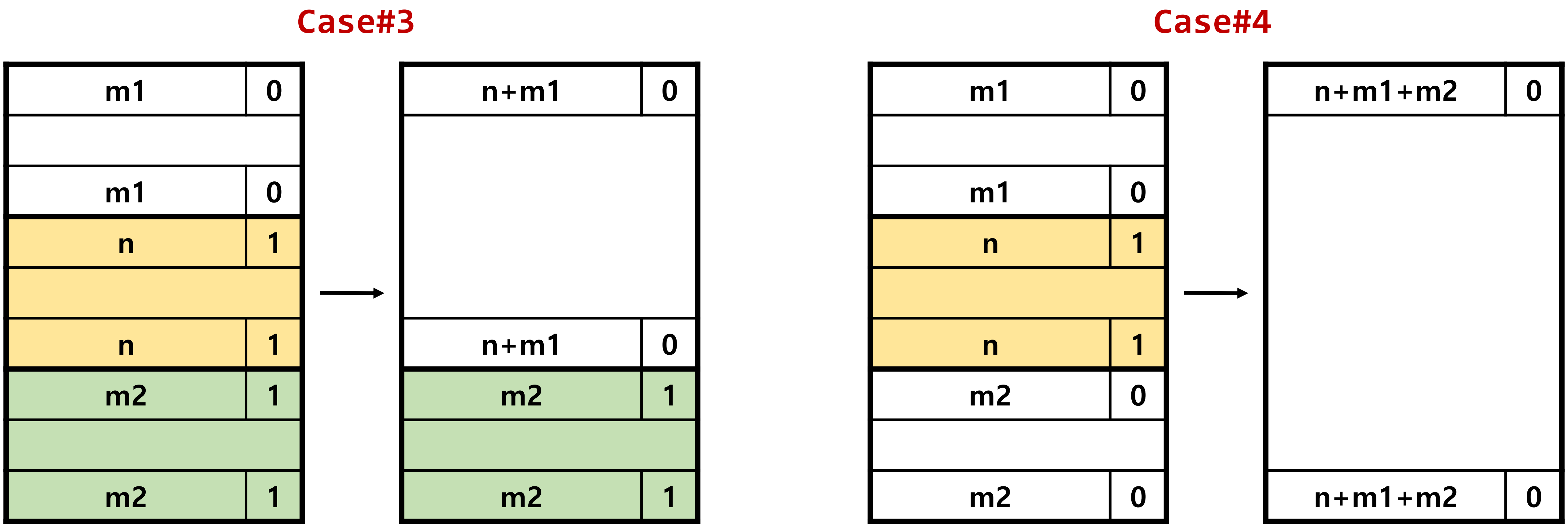


Constant Time Coalescing: Case #1, #2

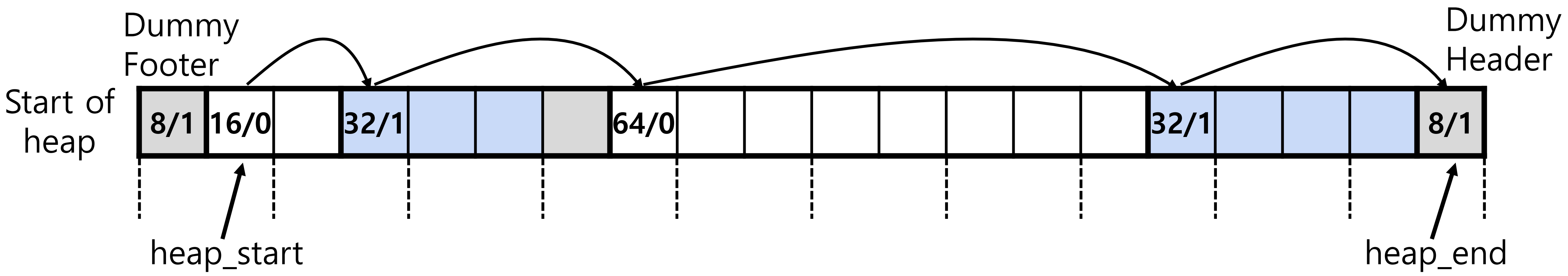




Constant Time Coalescing: Case #3, #4



Heap Structure



- Dummy footer before first header
  - Marked as allocated
  - Prevents accidental coalescing when freeing first block
- Dummy header after last footer
  - Prevents accidental coalescing when freeing final block

Top-Level Malloc Code

```
const size_t dsize = 2*sizeof(word_t);

void *mm_malloc(size_t size)
{
    size_t asize = round_up(size + dsize, dsize);

    block_t *block = find_fit(asize);

    if (block == NULL)
        return NULL;

    size_t block_size = get_size(block);
    write_header(block, block_size, true);
    write_footer(block, block_size, true);

    split_block(block, asize);

    return header_to_payload(block);
}
```

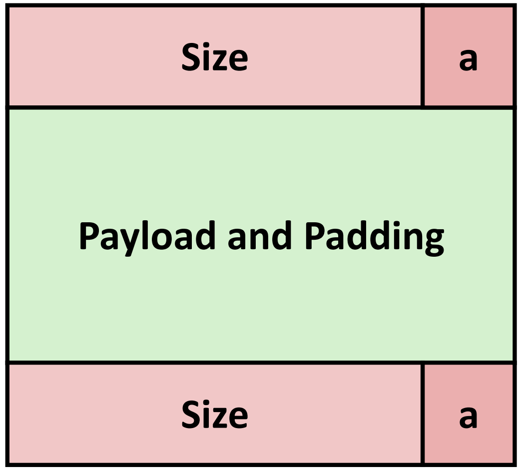
```
void mm_free(void *bp)
{
    block_t *block = payload_to_header(bp);
    size_t size = get_size(block);

    write_header(block, size, false);
    write_footer(block, size, false);

    coalesce_block(block);
}
```

```
round_up(n, m) = m *((n+m-1)/m)
```

(Rounds n up to the nearest multiple of m)

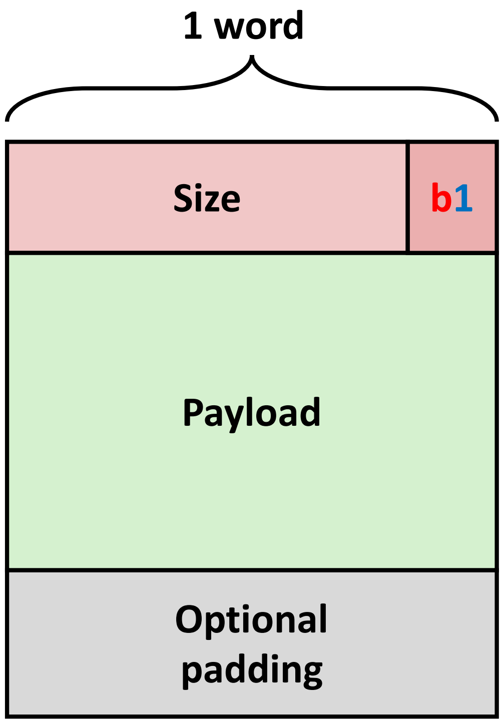
Disadvantages of Boundary Tags

- Internal fragmentation
- Can it be optimized?
  - Which blocks need the footer tag?
  - What does that mean?



No Boundary Tag for Allocated Blocks

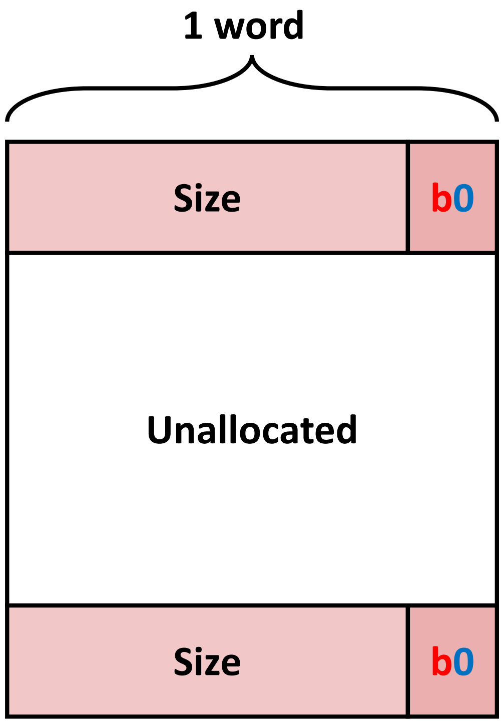
- Boundary tag needed only for free blocks
- When sizes are multiples of 16, have 4 spare bits



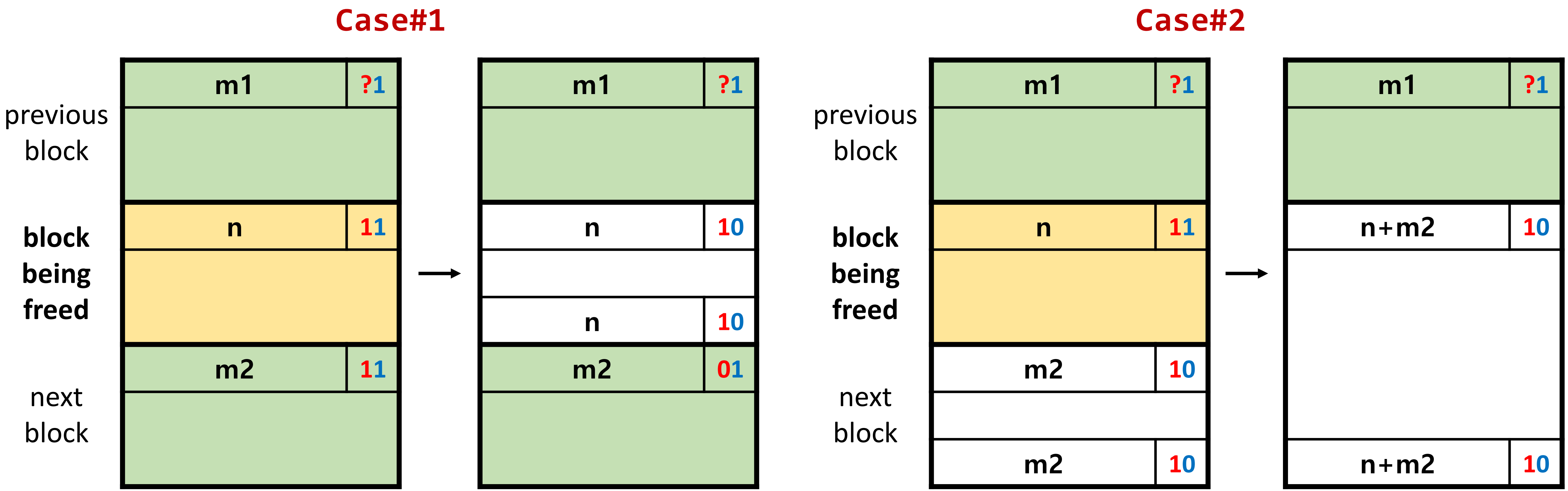
a = 1: Allocated block
a = 0: Free block
b = 1: Previous block is allocated
b = 0: Previous block is free

Size: block size

Payload: application data

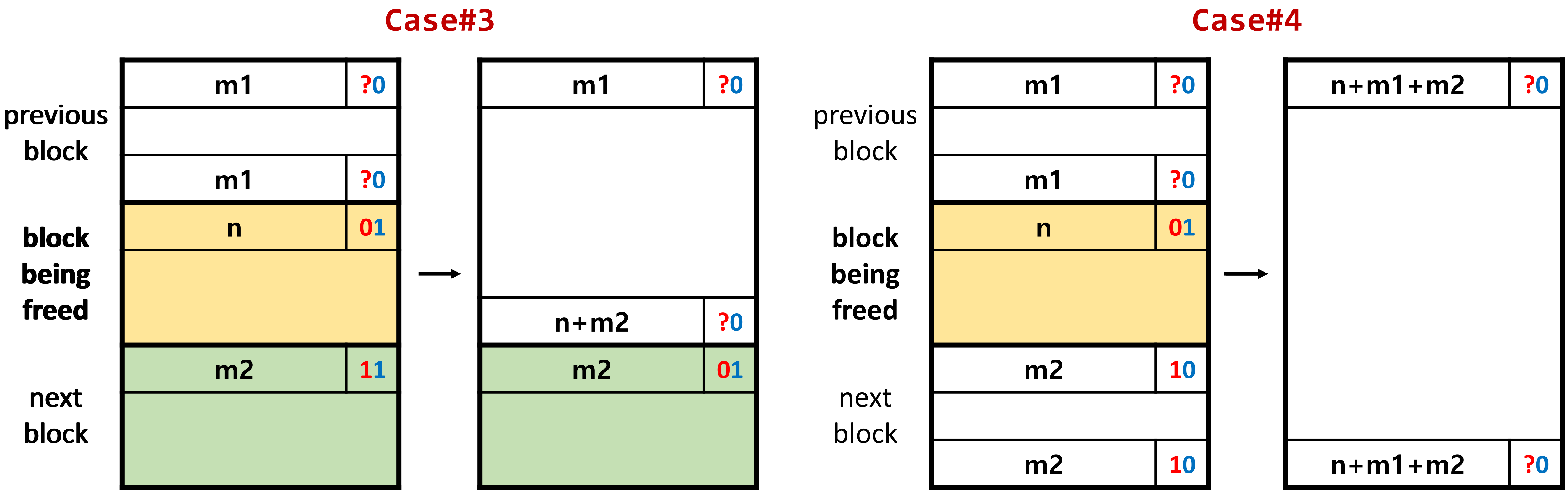


No Boundary Tag for Allocated Blocks: Case #1, #2



Header: Use 2 bits (address bits always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

No Boundary Tag for Allocated Blocks: Case #3



Header: Use 2 bits (address bits always zero due to alignment):
(previous block allocated)<<1 | (current block allocated)

Summary of Key Allocator Policies

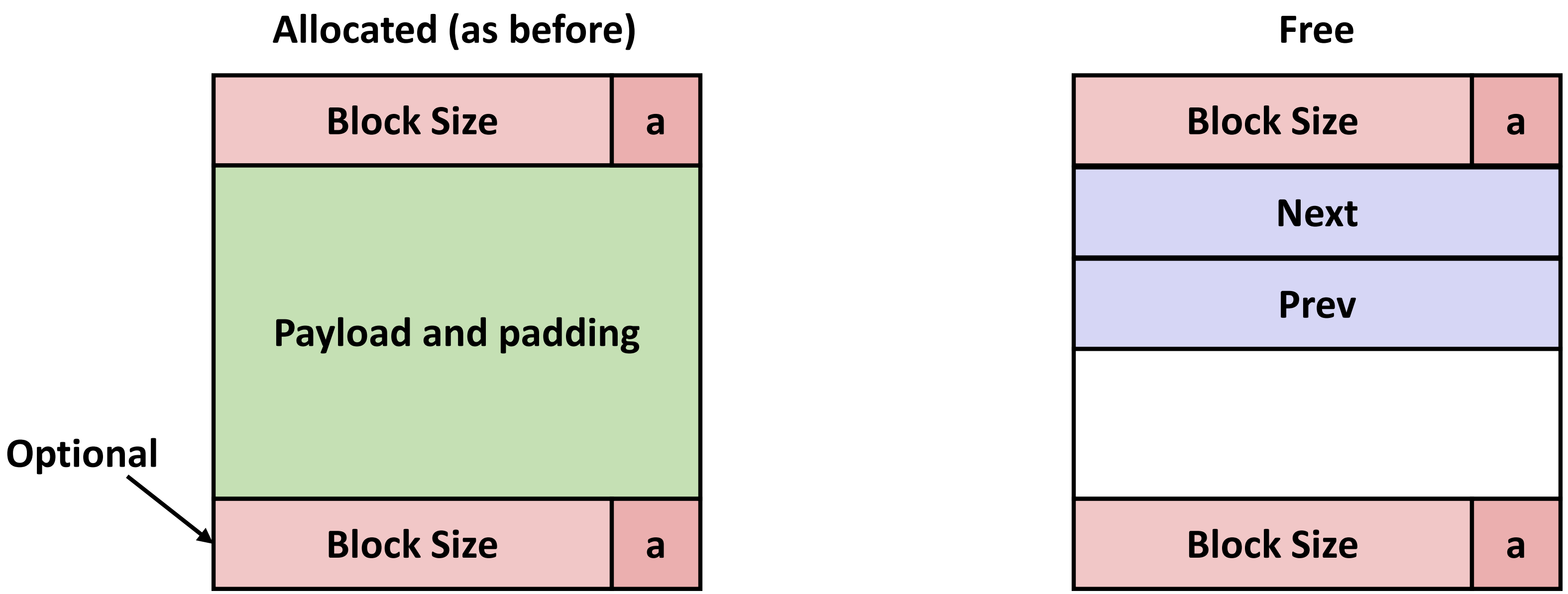
- Placement policy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - ***Interesting observation***: segregated free lists (next lecture) approximate a best fit placement policy without having to search entire free list
- Splitting policy:
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- Coalescing policy:
  - ***Immediate coalescing:*** coalesce each time **`free`** is called
  - ***Deferred coalescing:*** try to improve performance of **`free`** by deferring coalescing until needed.

Implicit Lists: Summary

- Implementation: very simple
- Allocate cost:
  - linear time worst case
- Free cost:
  - constant time worst case
  - even with coalescing
- Memory Overhead
  - will depend on placement policy
  - First-fit, next-fit or best-fit
- Not used in practice for `malloc/free` because of linear-time allocation
  - used in many special purpose applications
- However, the concepts of splitting and boundary tag coalescing are general to all allocators

Contents

Explicit Free Lists

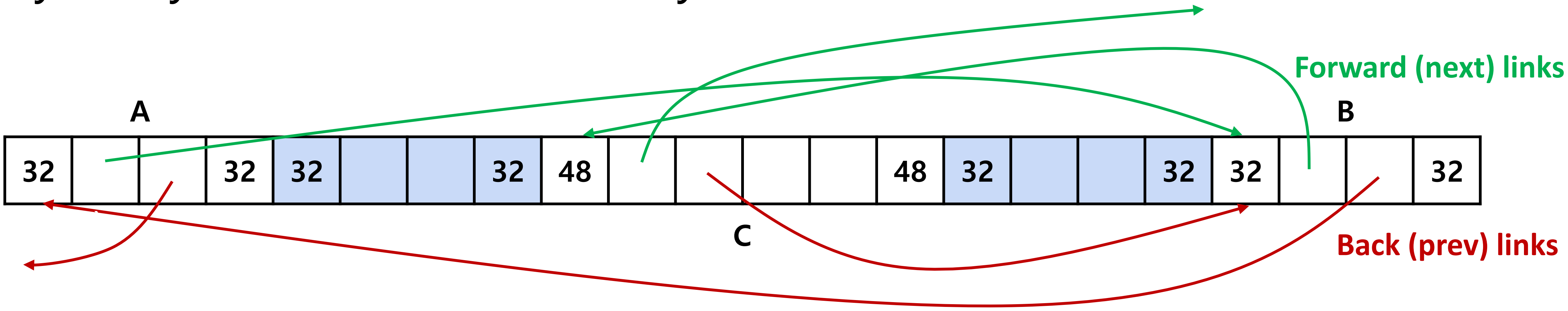


- Maintain list(s) of free blocks, not all blocks
  - Luckily we track only free blocks, so we can use payload area
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
    - To find adjacent blocks according to memory order

Explicit Free Lists

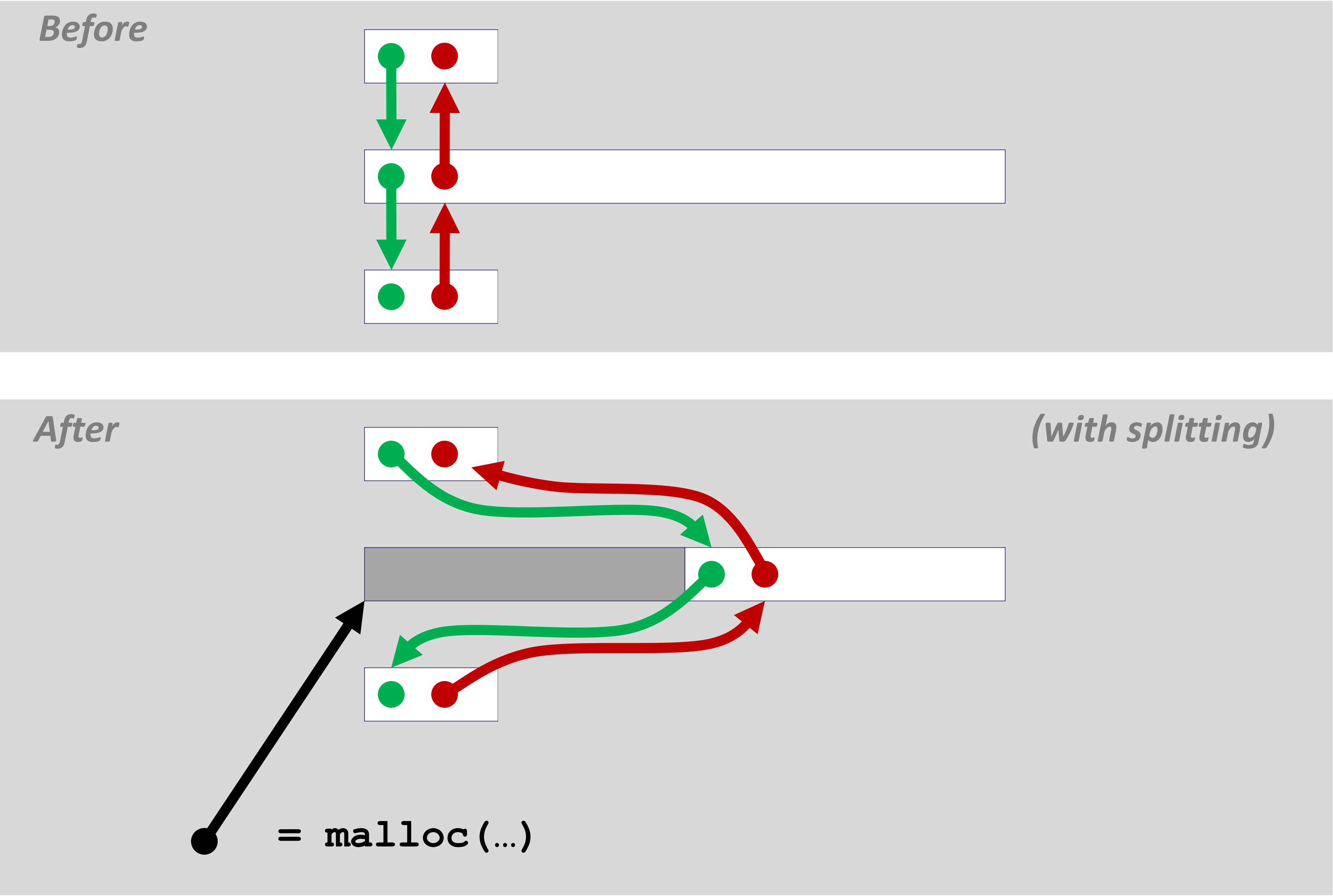
- Logically:

- Physically: blocks can be in any order



Allocating From Explicit Free Lists

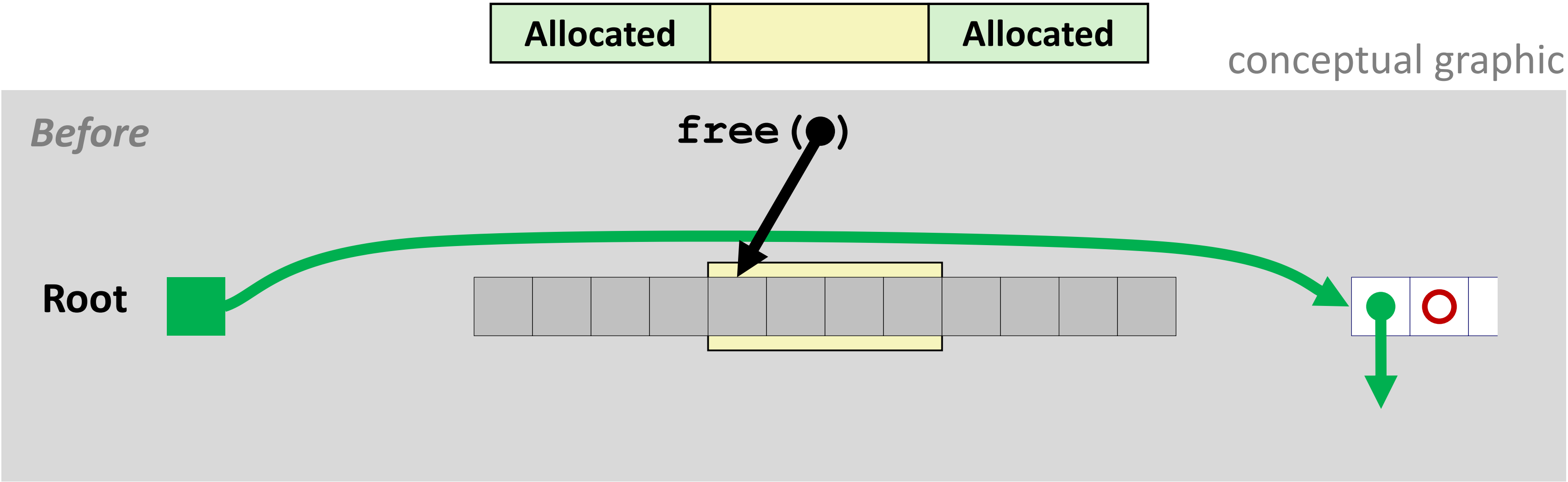
conceptual graphic



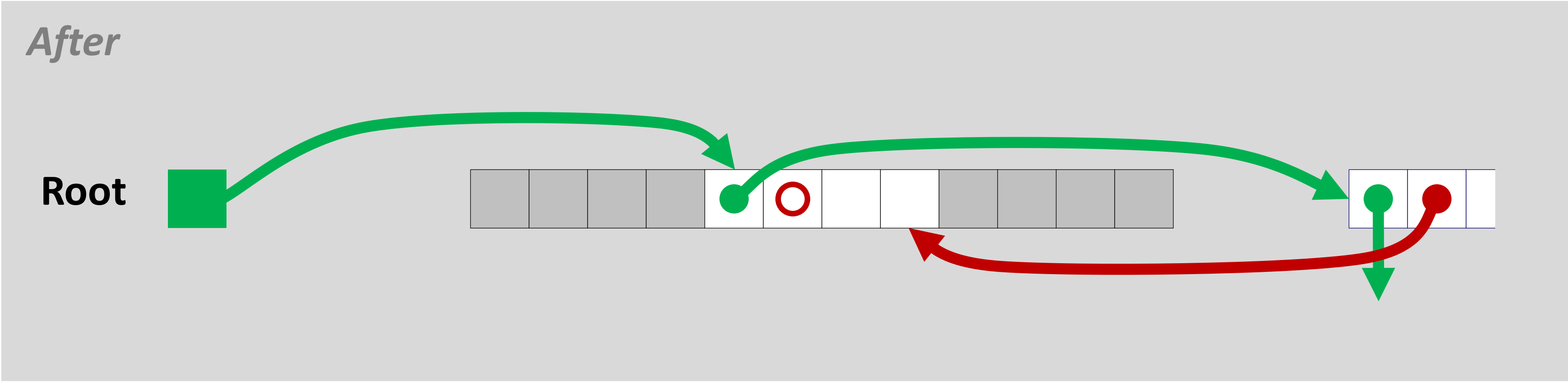
Freeing with Explicit Free Lists

- Insertion policy: Where in the free list do you put a newly freed block?
- Unordered
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
  - FIFO (first-in-first-out) policy
    - Insert freed block at the end of the free list
  - Pro: simple and constant time
  - Con: studies suggest fragmentation is worse than address ordered
- Address-ordered policy
  - Insert freed blocks so that free list blocks are always in address order:
    addr(prev) < addr(curr) < addr(next)
  - Con: requires search
  - Pro: studies suggest fragmentation is lower than LIFO/FIFO

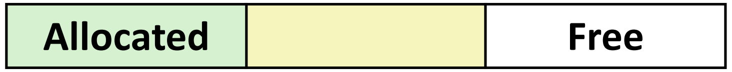
Freeing With a LIFO Policy (Case 1)



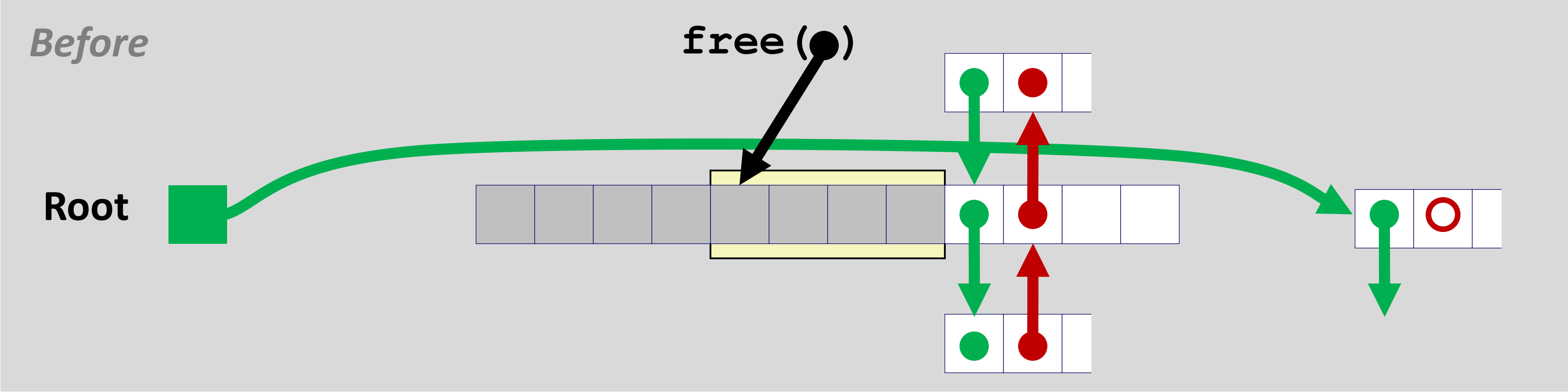
- Insert the freed block at the root of the list



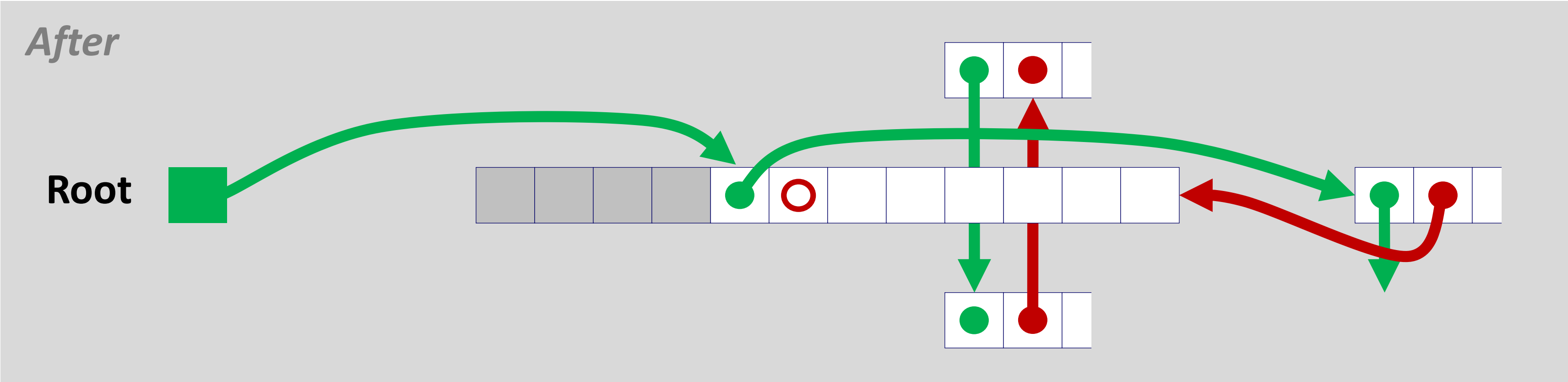
Freeing With a LIFO Policy (Case 2)



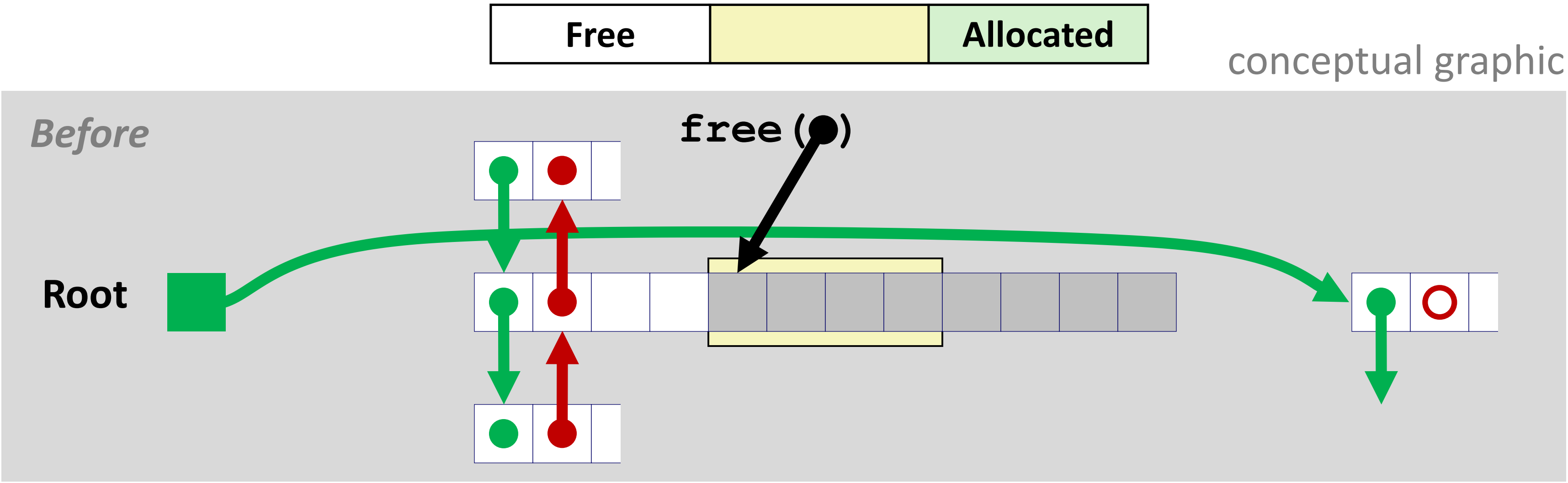
conceptual graphic



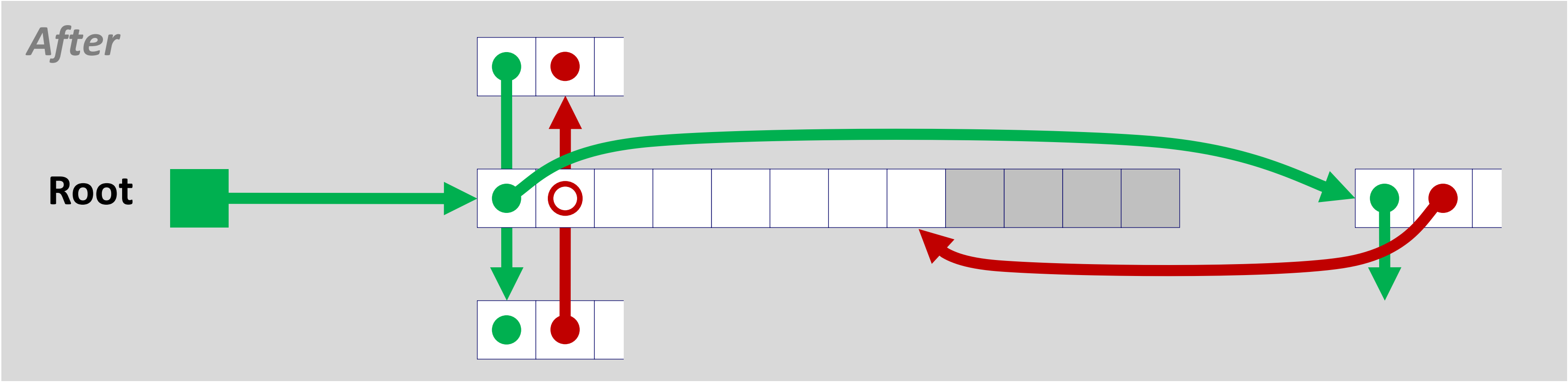
- Splice out adjacent successor block, coalesce both memory blocks, and insert the new block at the root of the list



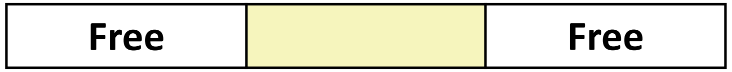
Freeing With a LIFO Policy (Case 3)



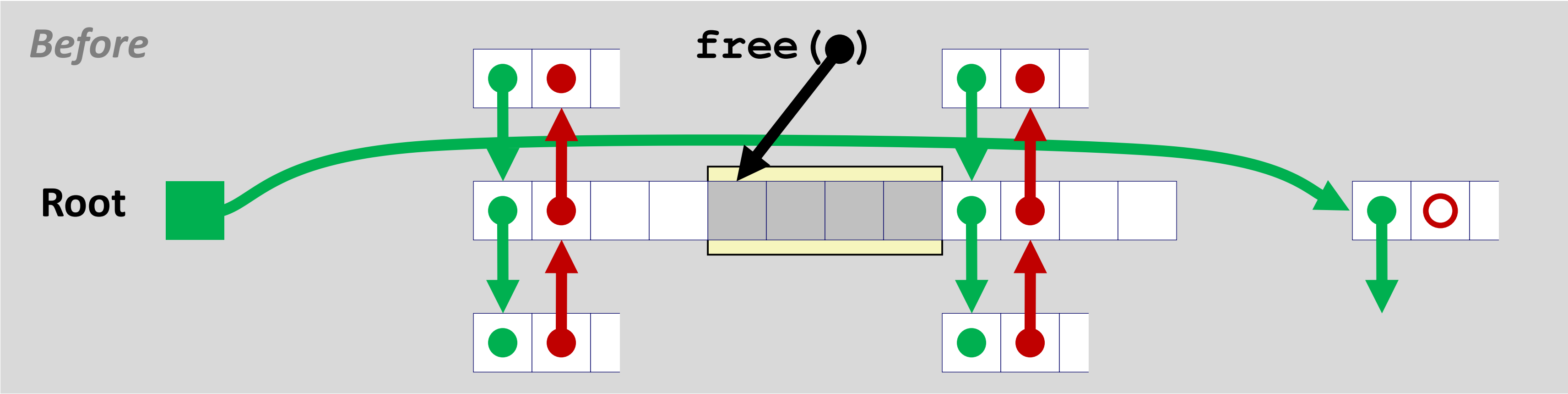
- Splice out adjacent predecessor block, coalesce both memory blocks, and insert the new block at the root of the list



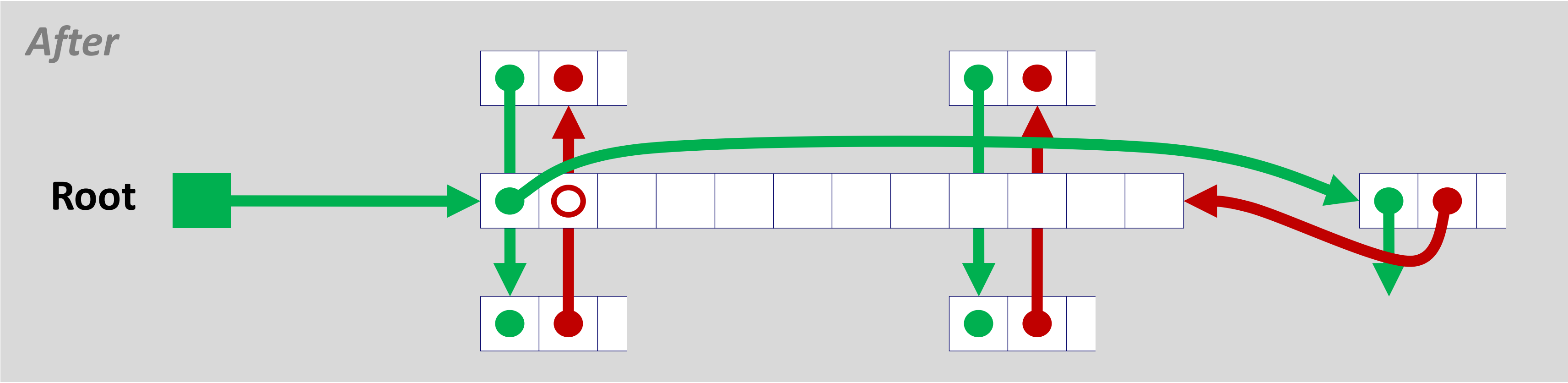
Freeing With a LIFO Policy (Case 4)



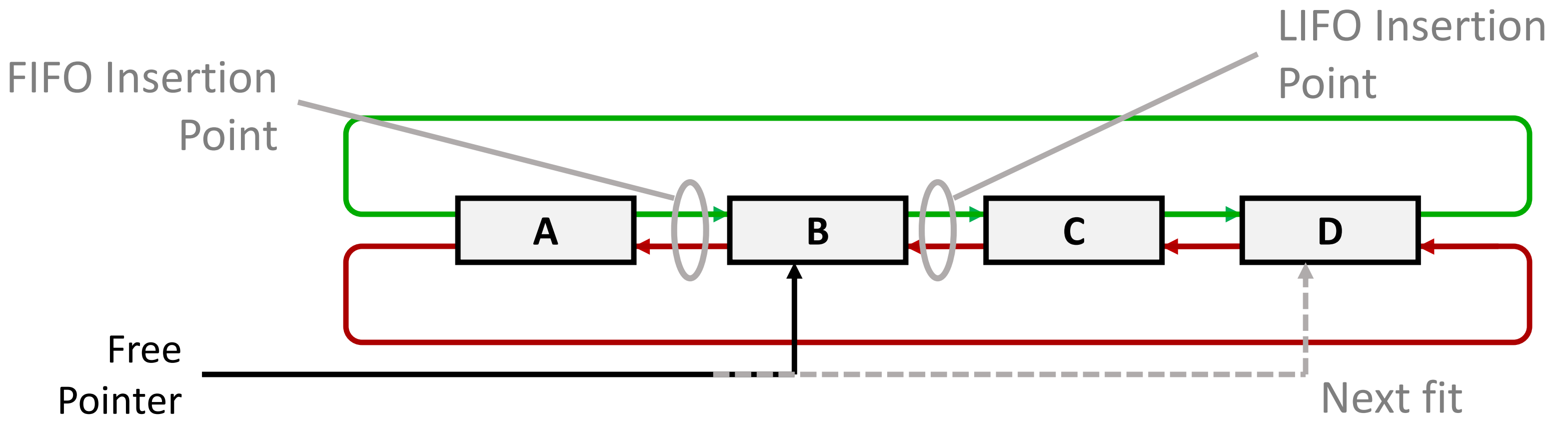
conceptual graphic



- Splice out adjacent predecessor and successor blocks, coalesce all 3 blocks, and insert the new block at the root of the list



Some Advice: An Implementation Trick



- Use circular, doubly-linked list
- Support multiple approaches with single data structure
- First-fit vs. next-fit
  - Either keep free pointer fixed or move as search list
- LIFO vs. FIFO
  - Insert as next block (LIFO), or previous block (FIFO)

Explicit List Summary

- Comparison to implicit list:
  - Allocate is linear time in number of free blocks instead of all blocks
    - Much faster when most of the memory is full
  - Slightly more complicated allocate and free because need to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each block)
    - Does this increase internal fragmentation?