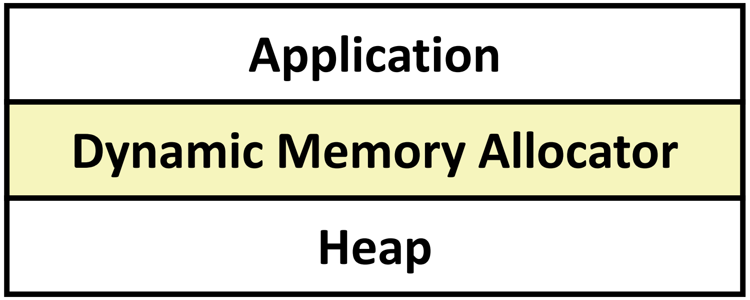
Dynamic Memory Allocation

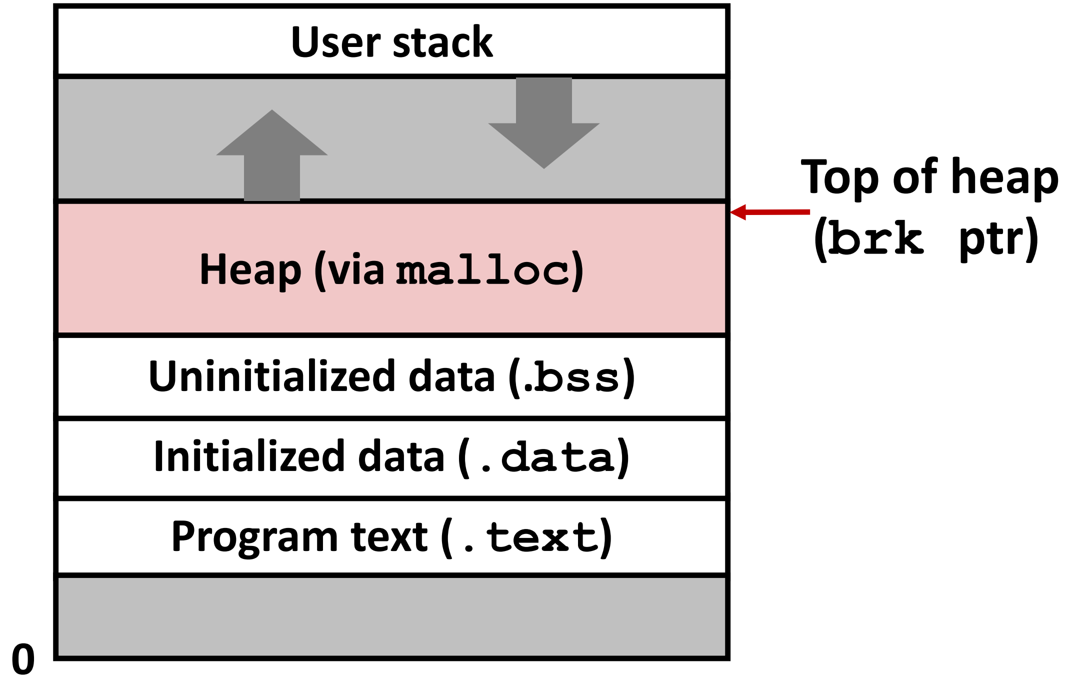
CSE4100: Multicore Programming

Sungyong Park (PhD)
Data-Intensive Computing and Systems Laboratory (DISCOS)
https://discos.sogang.ac.kr
Office: R908A, E-mail: parksy@sogang.ac.kr

Dynamic Memory Allocation

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





Dynamic Memory Allocation (Cont)

- **Allocator maintains heap as collection of variable sized memory *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., garbage collection in Java, ML, and Lisp

- **Will focus on simple explicit memory allocation**

The `malloc` Package

`#include <stdlib.h>`

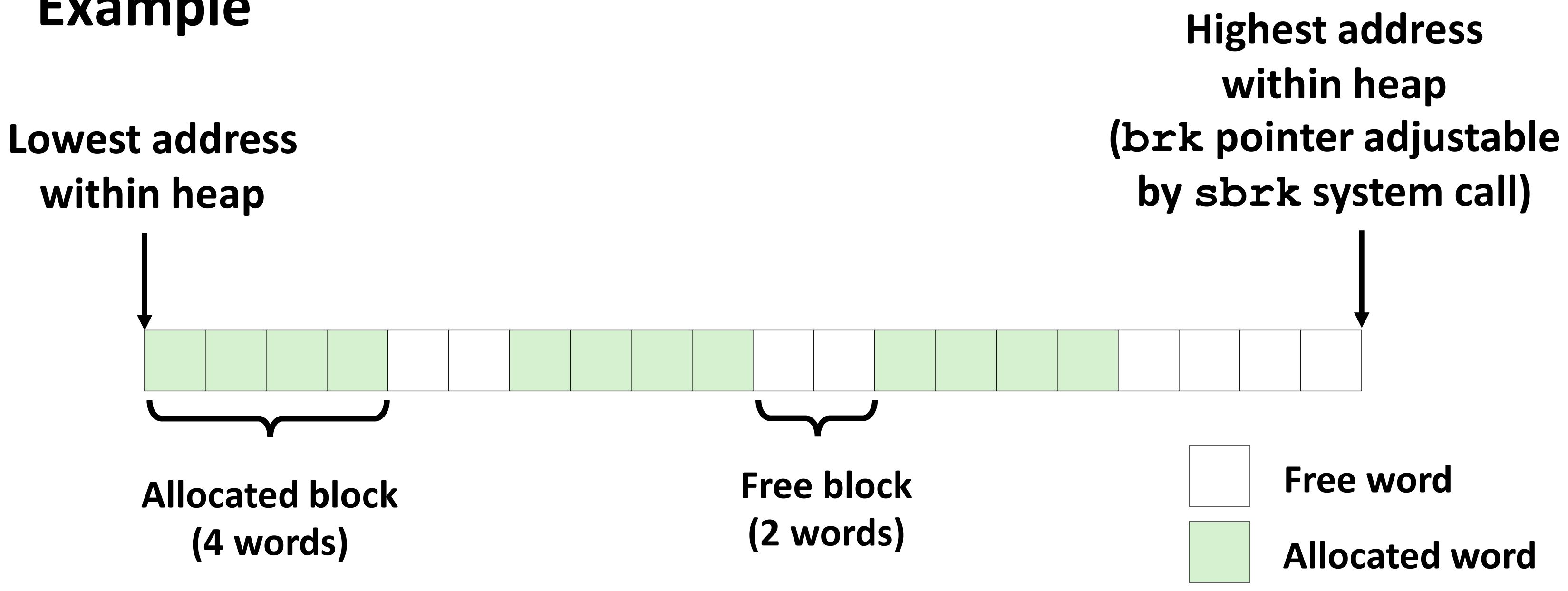
- `void *malloc(size_t size)`
  - Successful:
    - Returns a pointer to a memory block of at least **`size`** bytes aligned to an **8-byte (x86)** or **16-byte (x86-64)** boundary
    - If **`size == 0`**, returns NULL
  - Unsuccessful: returns NULL (0) and sets **`errno`**
- `void free(void *p)`
  - Returns the block pointed at by **`p`** to pool of available memory
  - **`p`** must come from a previous call to **`malloc`**, **`calloc`**, or **`realloc`**
- **Other functions**
  - **`calloc`**: Version of **`malloc`** that initializes allocated block to zero
  - **`realloc`**: Changes the size of a previously allocated block
  - **`sbrk`**: Used internally by allocators to grow or shrink the heap

malloc Example

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

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

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

    /* Initialize allocated block */
    for ( i = 0; i < n; i++ )
         p[i] = i;


    /* Return allocated block to the heap */
    free(p);
}
```

Assumptions Made

- **1 square = 1 "word" = 4 bytes**
  - *Word* is the number of bits processed by CPU at the same time
- **Alignment = 8-byte alignment**
  - Alignment depends on whether the code is compiled to run in 32-bit mode (8-byte alignment) or 64-bit mode (16-byte alignment)
- **Example**



Allocation Example

`SIZE: word size`

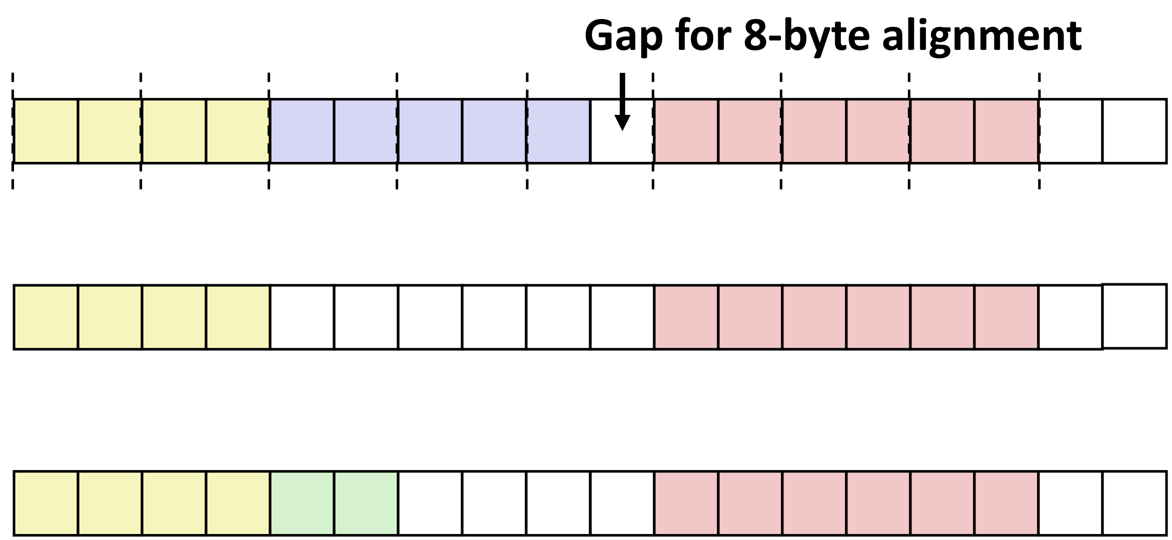
`p1 = malloc(4*SIZE)`

`p2 = malloc(5*SIZE)`

`p3 = malloc(6*SIZE)`

`free(p2)`

`p4 = malloc(2*SIZE)`



Constraints

- **Applications**
  - Can issue arbitrary sequence of **`malloc`** and **`free`** requests
  - **`free`** request must be to a **`malloc`**’d block
- **Explicit allocators**
  - Can’t control the 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 that they satisfy all alignment requirements
    - 8-byte (x86) or 16-byte (x86-64) alignment
  - Can manipulate and modify only free memory
  - Can’t move the allocated blocks once they are **`malloc`**’d
    - *i.e.*, compaction is not allowed

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: Peak Memory Utilization

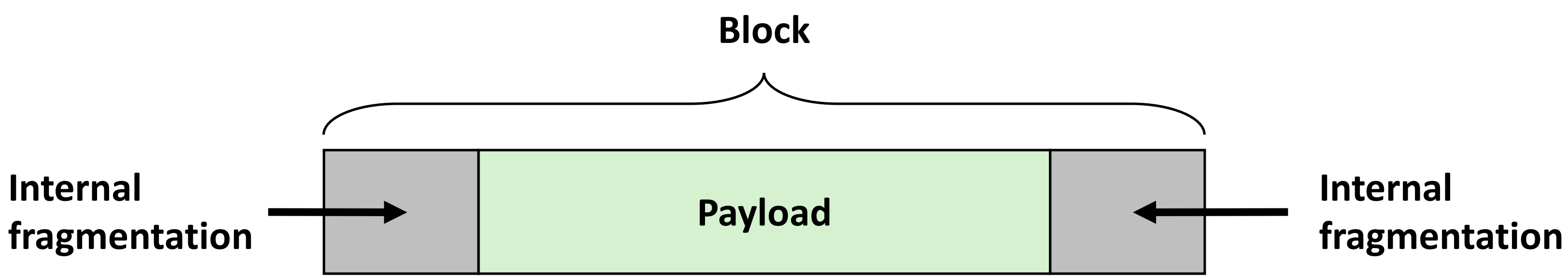
- **Given some sequence of `malloc` and `free` requests**
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- ***Def:* Aggregate payload P_k**
  - `malloc(p)` results in a block with a ***payload*** of `p` bytes
  - After request R_k has completed, the ***aggregate payload*** P_k is the sum of currently allocated payloads
- ***Def:* Current heap size H_k**
  - Assume H_k is monotonically non-decreasing
    - i.e., heap only grows when allocator uses `sbrk`
- ***Def:* Peak memory utilization after k+1 requests**
  - $U_k = (\max_{i \le k} P_i) \ / \ H_k$

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**



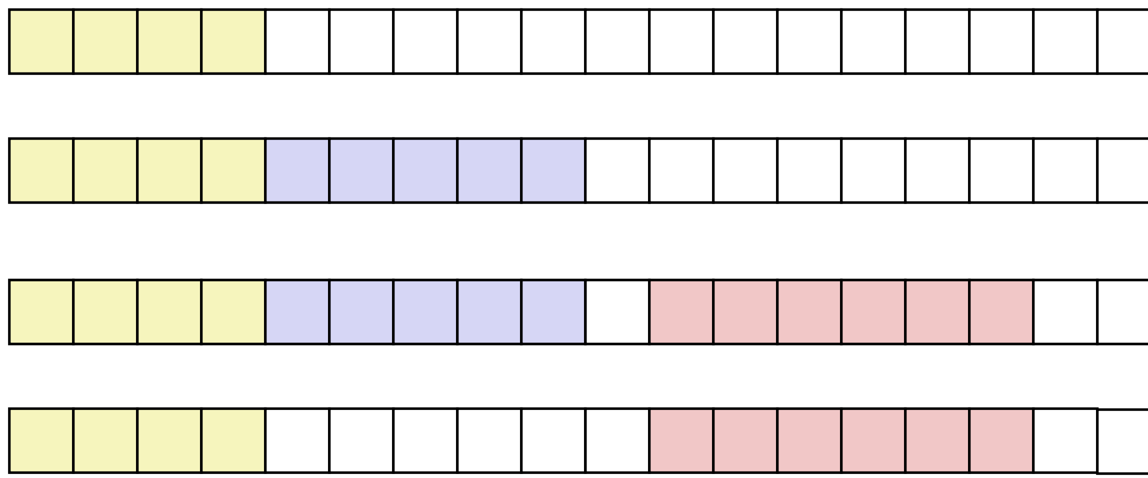
- **Caused by**
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - Explicit policy decisions
    (e.g., to return a big block to satisfy a small request)
- **Depends on the pattern of currently allocated requests**
  - Thus, easy to measure

External Fragmentation

`SIZE: word size`

- **Occurs when there is enough aggregate heap memory, but no single free block is large enough**

```
p1 = malloc(4*SIZE)
p2 = malloc(5*SIZE)
p3 = malloc(6*SIZE)
free(p2)
p4 = malloc(7*SIZE)
```



Oops! (what would happen now?)

- **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 (i.e., `free(p)`)?**

- **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 reinsert freed block?

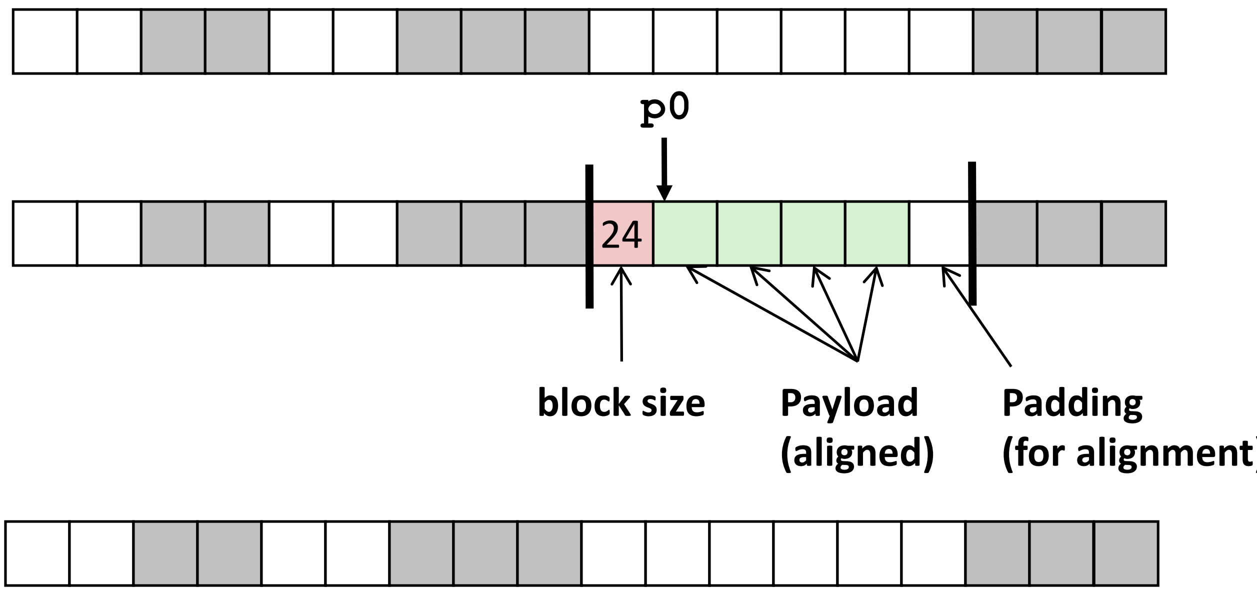
Knowing How Much to Free

- **Standard method**
  - Keep the length (**in bytes**) of a block in the word ***preceding*** the block
    - This word is often called the ***header field*** or ***header***
    - The length field in the header *may* include the size of header and padding
  - Requires an extra word for every allocated block

SIZE: word size

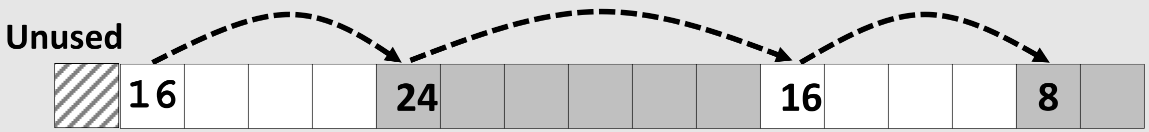
`p0 = malloc(4*SIZE)`

`free(p0)`

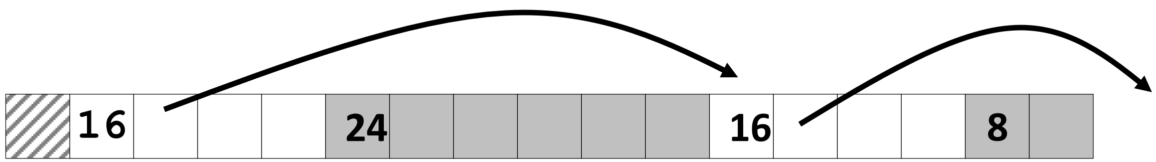


Keeping Track of Free Blocks

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



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



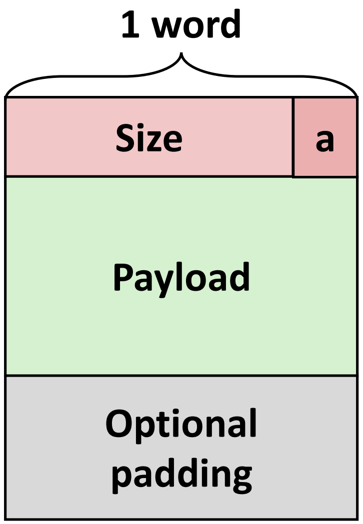
- **Method 3: *Segregated free list***
  - Different free lists for different size classes

- **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

Method 1: Implicit List

- **For each block we need both *size* and *allocation status***
  - Could store this information in two words: wasteful!
- **Standard trick**
  - If blocks are aligned, some low-order address bits are always 0
    - **8-byte alignment**: **low-order 3 bits** are always 0
  - Instead of storing always-0 bits, use 1 bit as an allocated/free flag
  - When reading size word, this bit should be *masked out*

Format of allocated and free blocks

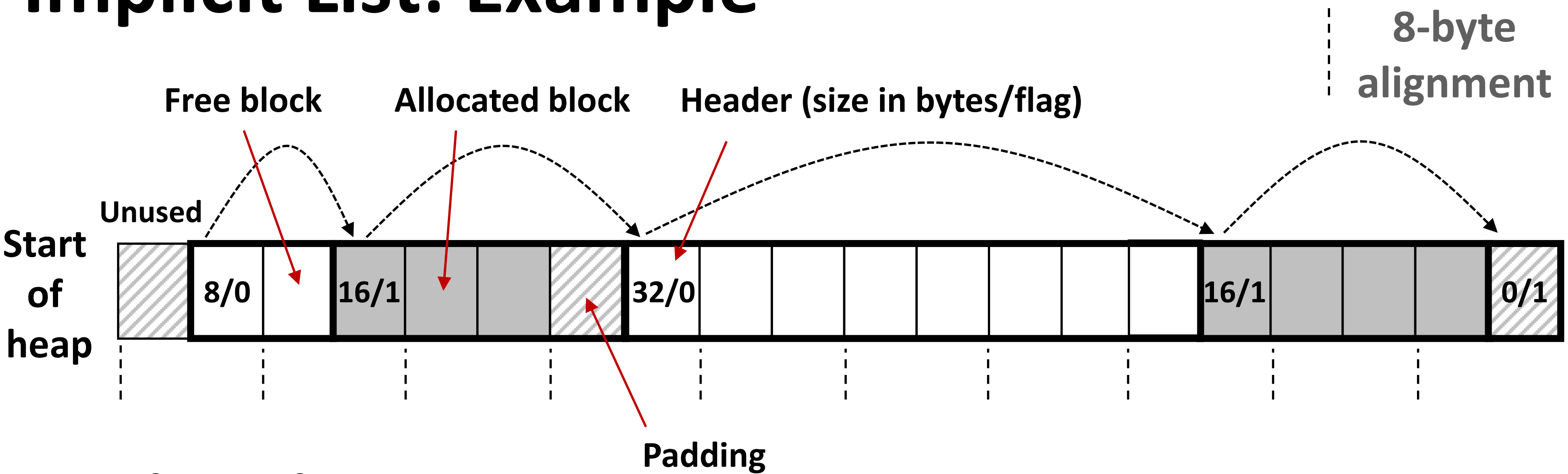


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Implicit List: Example



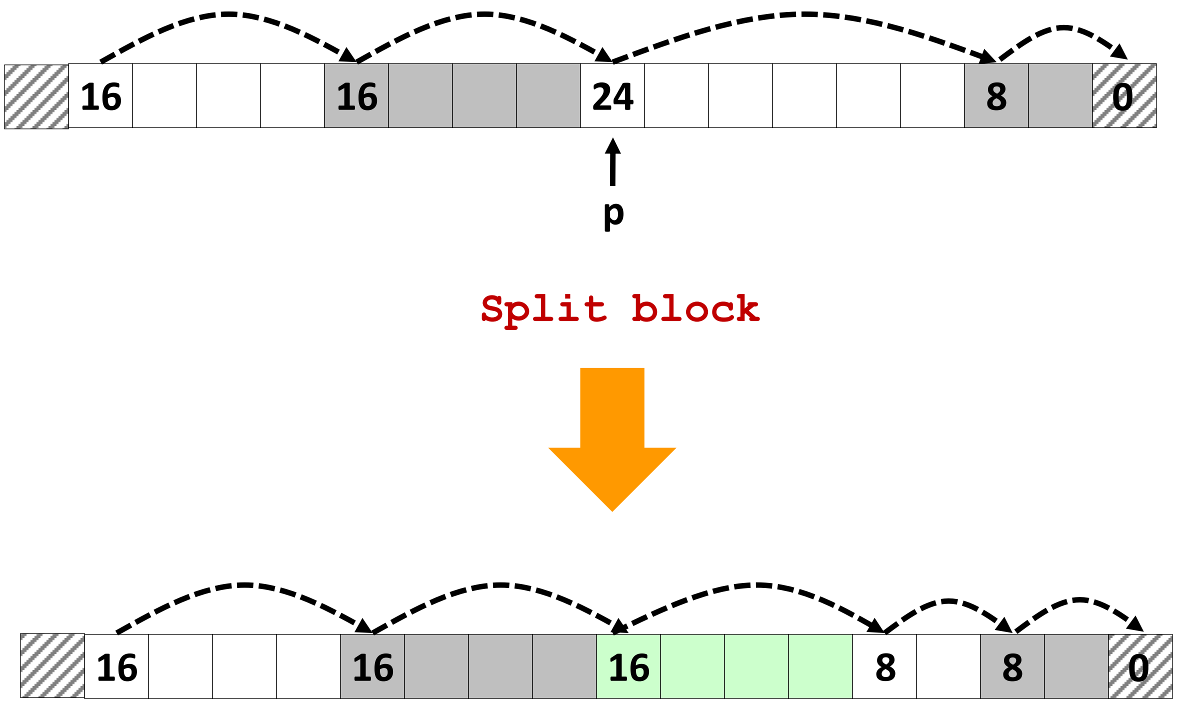
- **8-byte alignment**
  - Requires headers are at *non-aligned* positions and payloads are *aligned*
  - May require initial unused word and cause some internal fragmentation
  - 1 word (0/1) to mark the end of list
- **Advantage**
  - Simplicity
- **Disadvantage**
  - Placing an allocated block requires a search of the list to find a free block (sometimes the whole list)

Implicit List: Finding a Free Block

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

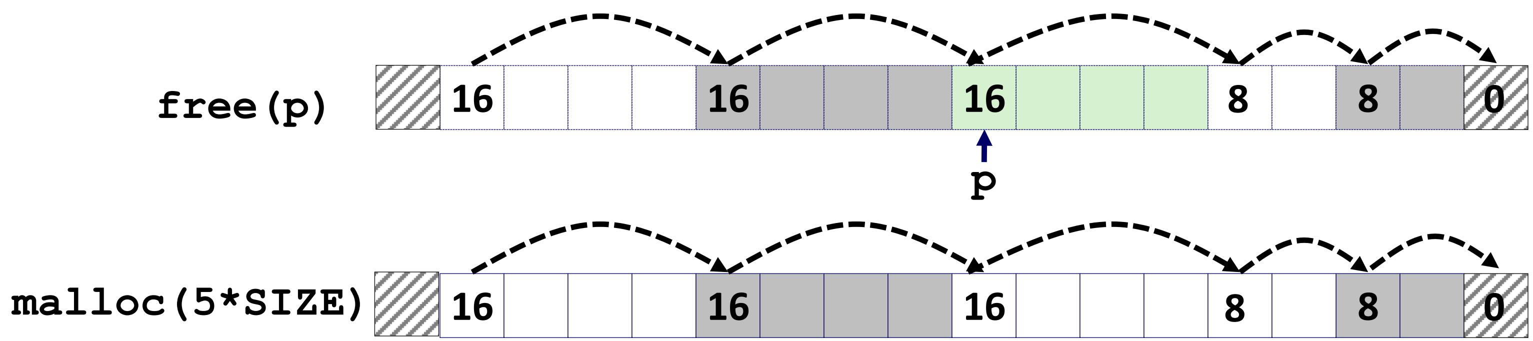
Implicit List: Allocating in Free Block

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



Implicit List: Freeing a Block

- **Simplest implementation**
  - Need only clear the “allocated” flag
  - But can lead to “false fragmentation”



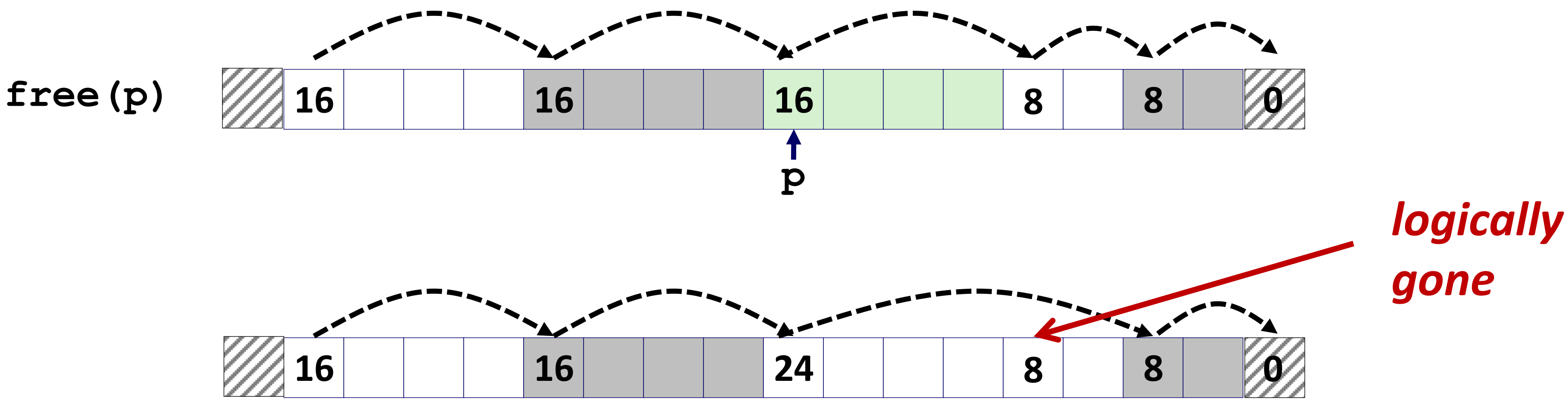
`SIZE: word size`

Oops!

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

Implicit List: Coalescing

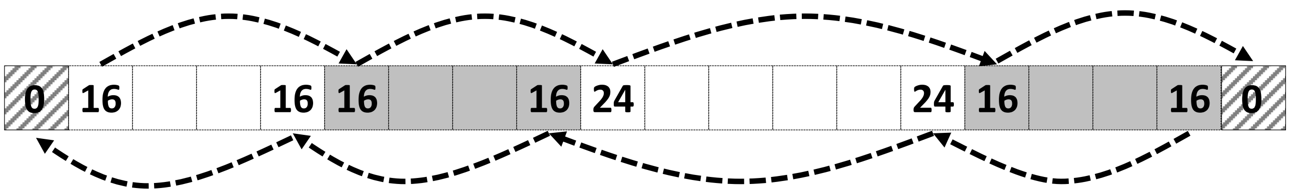
- **Join *(coalesce)* with next/previous blocks, if they are free**
  - Coalescing with next block



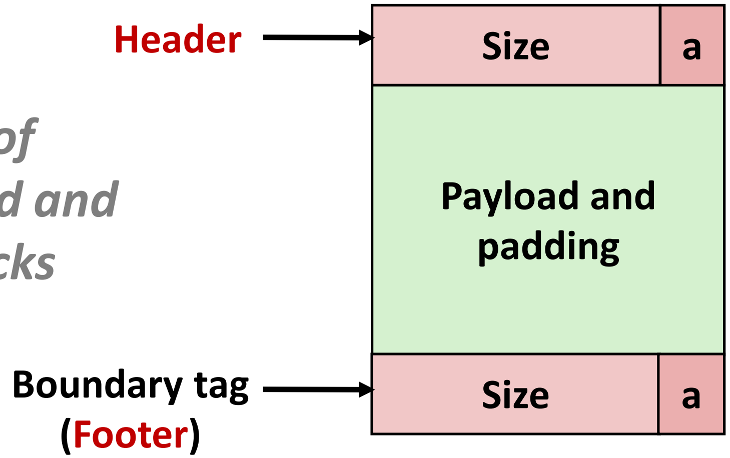
  - How do we coalesce with ***previous*** block?
    - How do we know where it starts?
    - How can we determine whether its allocated?

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!



Format of allocated and free blocks

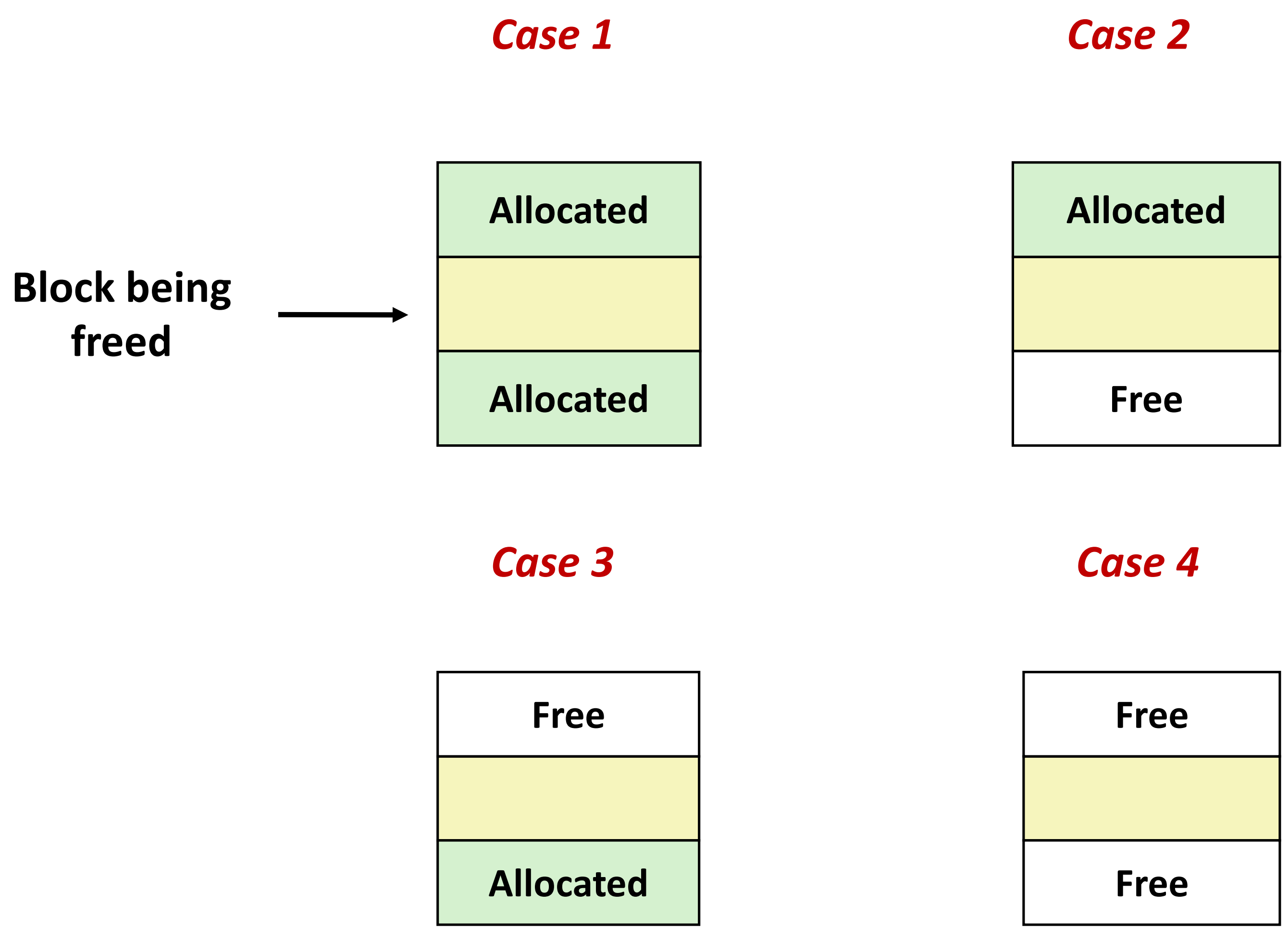


a = 1: Allocated block
a = 0: Free block

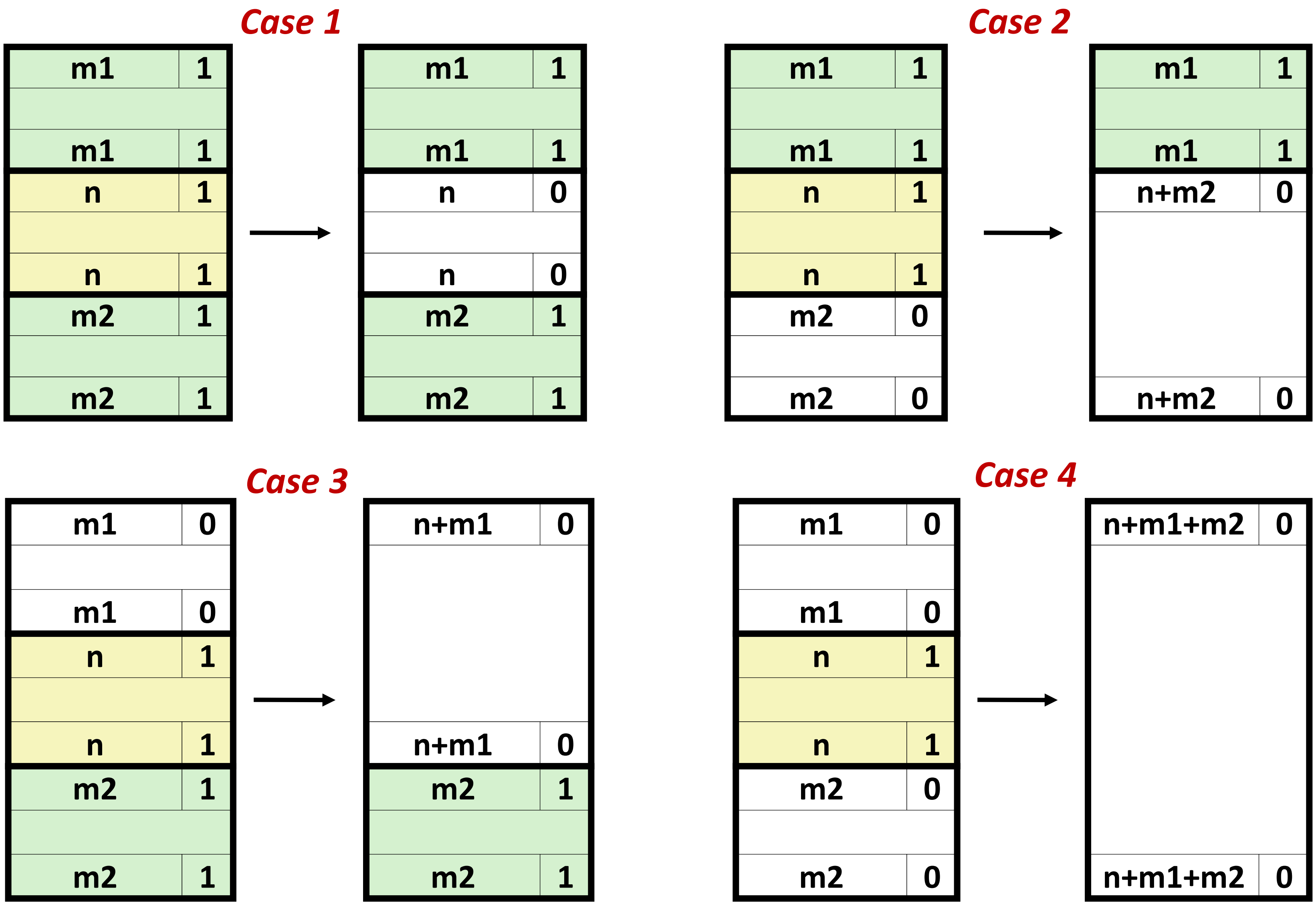
Size: Total block size

Payload: Application data (allocated blocks only)

Constant Time (Immediate) Coalescing



Constant Time (Immediate) Coalescing (Cont)



Disadvantages of Boundary Tags

- **Extra space overhead**

- **Which blocks need the footer tag?**
  - If the previous block is not free (allocated), we don't need the boundary tag of the previous block
  - Otherwise, we need the boundary tag of the previous block, because it should be able to tell the size of the block

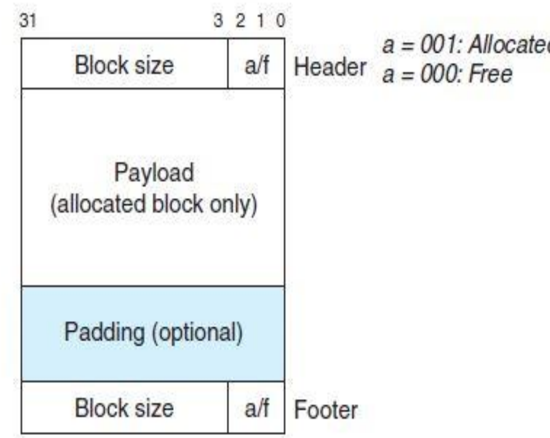
- **Can it be optimized?**
  - Store the allocated/free bit of the previous block in one of the excess low-order bits of the current block
  - Then, allocated blocks would not need footers and the extra space can be used for payload

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 (covered later) 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
    - Coalesce as you scan the free list for `malloc`
    - Coalesce when the amount of external fragmentation reaches some threshold

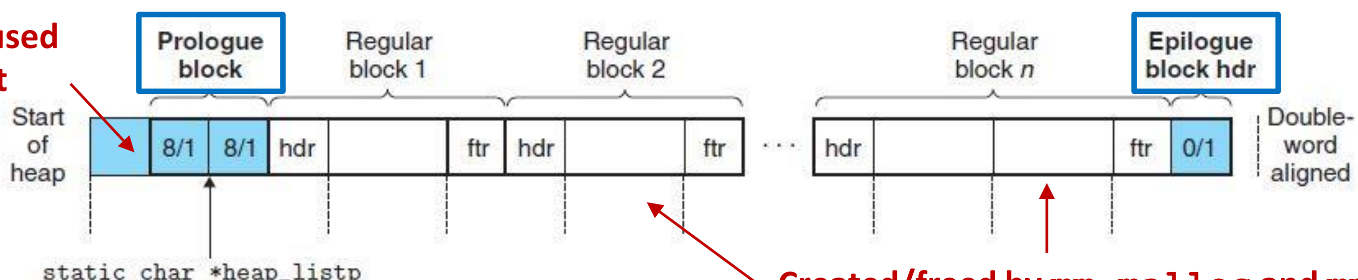
Implementing a Simple Allocator

- **Export three functions to application programs**
  - `int` **`mm_init(void)`**
  - `void` **`*mm_malloc(size_t size)`** -> same as **`malloc()`**
  - `void` **`*mm_free(void *ptr)`** -> same as **`free()`**
- **Assumption**
  - Use *implicit list* with *boundary tag* (right)
  - Use **4-byte** word and **8-byte** alignment
  - Minimum block size is **16 bytes** (4 words) due to alignment and block format
  - Heap consists of 1 prologue block, 0 or more regular blocks and 1 epilogue block



Used for coalescing

Intentionally unused for alignment



Created/freed by `mm_malloc` and `mm_free`

Implementing a Simple Allocator (Cont)

- Useful constants and macros

code/vm/malloc/mm.c

```c
/* Basic constants and macros */
#define WSIZE       4       /* Word and header/footer size (bytes) */
#define DSIZE       8       /* Double word size (bytes) */
#define CHUNKSIZE  (1<<12)  /* Extend heap by this amount (bytes) */

#define MAX(x, y) ((x) > (y)? (x) : (y))

/* Pack a size and allocated bit into a word */
#define PACK(size, alloc)  ((size) | (alloc))

/* Read and write a word at address p */
#define GET(p)       (*(unsigned int *)(p))
#define PUT(p, val)  (*(unsigned int *)(p) = (val))

/* Read the size and allocated fields from address p */
#define GET_SIZE(p)  (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

/* Given block ptr bp, compute address of its header and footer */
#define HDRP(bp)       ((char *)(bp) - WSIZE)
#define FTRP(bp)       ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp)  ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp)  ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```

8-byte alignment

LSB is alloc/free bit

code/vm/malloc/mm.c

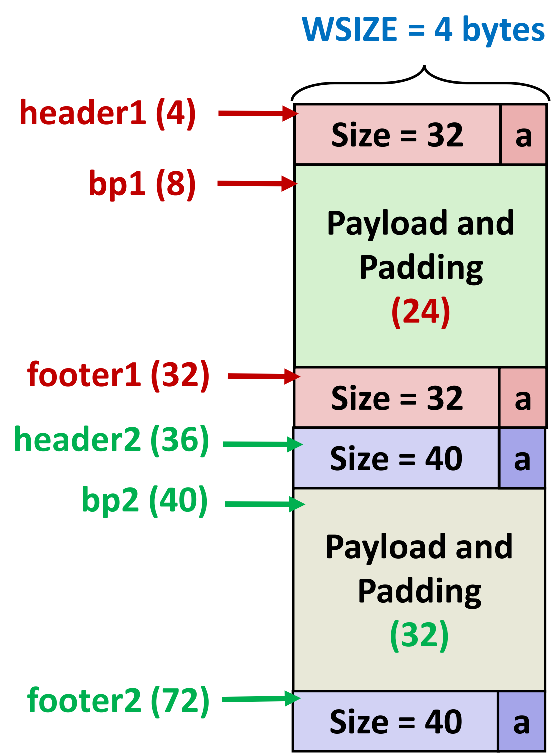


Figure 9.43 Basic constants and macros for manipulating the free list.

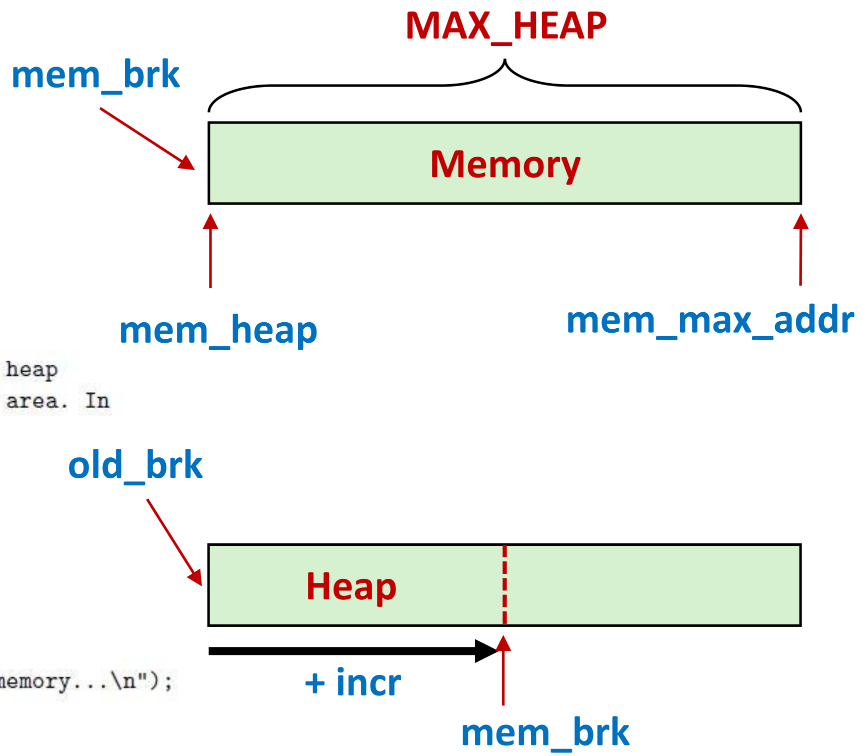
Implementing a Simple Allocator (Cont)

- Memory initialization for heap and heap extension

code/vm/malloc/memlib.c

```
/* Private global variables */
static char *mem_heap;     /* Points to first byte of heap */
static char *mem_brk;      /* Points to last byte of heap plus 1 */
static char *mem_max_addr; /* Max legal heap addr plus 1*/

/*
 * mem_init - Initialize the memory system model
 */
void mem_init(void)
{
    mem_heap = (char *)Malloc(MAX_HEAP);
    mem_brk = (char *)mem_heap;
    mem_max_addr = (char *)(mem_heap + MAX_HEAP);
}

/*
 * mem_sbrk - Simple model of the sbrk function. Extends the heap
 *    by incr bytes and returns the start address of the new area. In
 *    this model, the heap cannot be shrunk.
 */
void *mem_sbrk(int incr)
{
    char *old_brk = mem_brk;

    if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
        errno = ENOMEM;
        fprintf(stderr, "ERROR: mem_sbrk failed. Ran out of memory...\n");
        return (void *)-1;
    }
    mem_brk += incr;
    return (void *)old_brk;
}
```

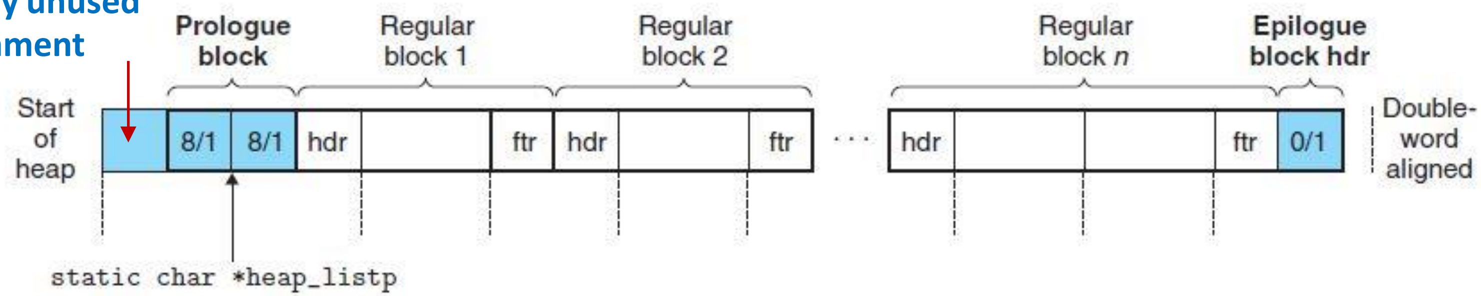
code/vm/malloc/memlib.c



Implementing a Simple Allocator (Cont)

- **Heap initialization**

Intentionally unused for alignment



code/vm/malloc/mm.c

```
int mm_init(void)
{
    /* Create the initial empty heap */
    if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
        return -1;
    PUT(heap_listp, 0);                          /* Alignment padding */
    PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* Prologue header */
    PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
    PUT(heap_listp + (3*WSIZE), PACK(0, 1));     /* Epilogue header */
    heap_listp += (2*WSIZE);

    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}
```

Create an empty heap with only prologue and epilogue blocks (16 bytes – shaded in blue)

Initially, create a 1024 word heap (4096 bytes)

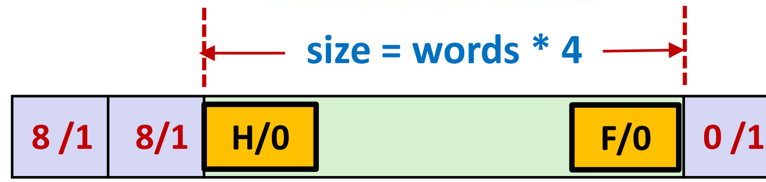
code/vm/malloc/mm.c

Figure 9.44 mm_init creates a heap with an initial free block.

Implementing a Simple Allocator (Cont)

- **Extend a heap**

code/vm/malloc/mm.c




```
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;

    /* Allocate an even number of words to maintain alignment */
    size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;

    /* Initialize free block header/footer and the epilogue header */
    PUT(HDRP(bp), PACK(size, 0));         /* Free block header */
    PUT(FTRP(bp), PACK(size, 0));         /* Free block footer */
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */

    /* Coalesce if the previous block was free */
    return coalesce(bp);
}
```

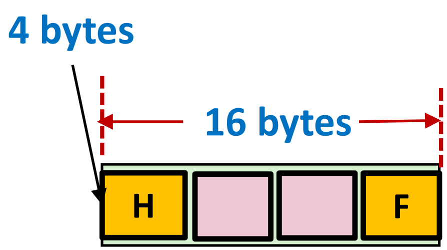
code/vm/malloc/mm.c

Figure 9.45 extend_heap extends the heap with a new free block.

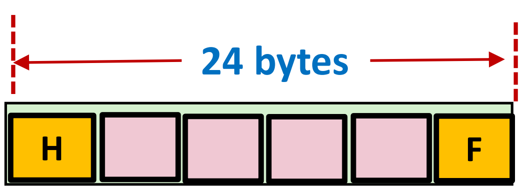
Implementing a Simple Allocator (Cont)

- Heap allocation

If requested size <= 8 bytes, allocate the minimum block size (16 bytes) to include header and footer



mm_malloc(7)



mm_malloc(9)

code/vm/malloc/mm.c

```
void *mm_malloc(size_t size)
{
    size_t asize;      /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    char *bp;

    /* Ignore spurious requests */
    if (size == 0)
        return NULL;

    /* Adjust block size to include overhead and alignment reqs. */
    if (size <= DSIZE)
        asize = 2*DSIZE;
    else
        asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);

    /* Search the free list for a fit */
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }

    /* No fit found. Get more memory and place the block */
    extendsize = MAX(asize,CHUNKSIZE);
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
        return NULL;
    place(bp, asize);
    return bp;
}
```

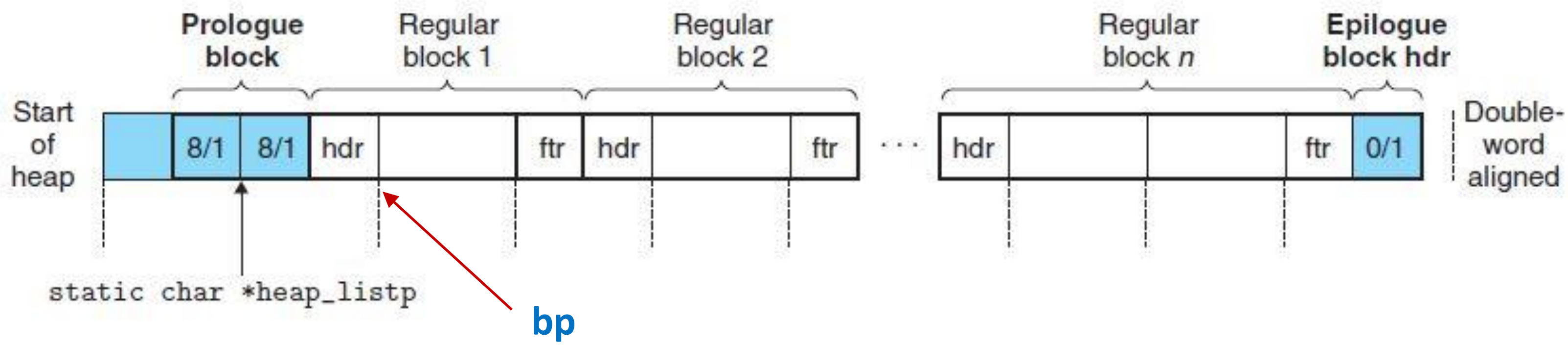
code/vm/malloc/mm.c

$$\frac{\text{size} + 15}{8}$$

Figure 9.47 mm_malloc allocates a block from the free list.

Implementing a Simple Allocator (Cont)

- **Search the free list (*first-fit*)**



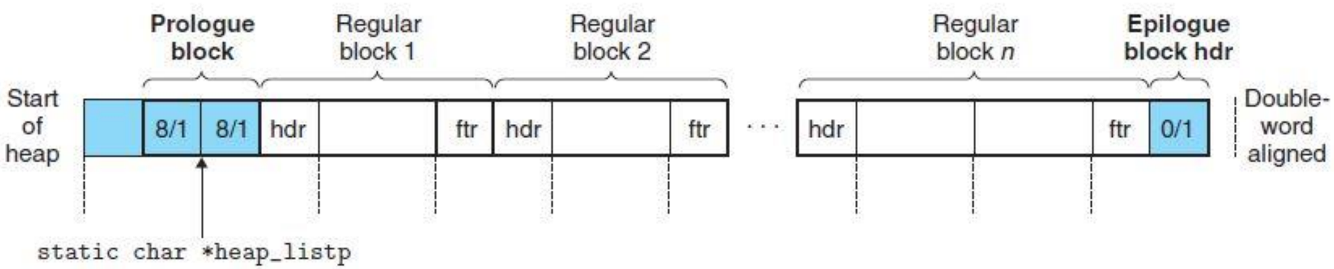
code/vm/malloc/mm.c

```
static void *find_fit(size_t asize)
{
    /* First-fit search */
    void *bp;

    for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
        if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
            return bp;
        }
    }
    return NULL; /* No fit */
#endif
}
```

code/vm/malloc/mm.c

Implementing a Simple Allocator (Cont)

- **Split the free list if needed**



code/vm/malloc/mm.c

```
static void place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));

    if ((csize - asize) >= (2*DSIZE)) {
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp), PACK(csize-asize, 0));
        PUT(FTRP(bp), PACK(csize-asize, 0));
    }
    else {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

code/vm/malloc/mm.c

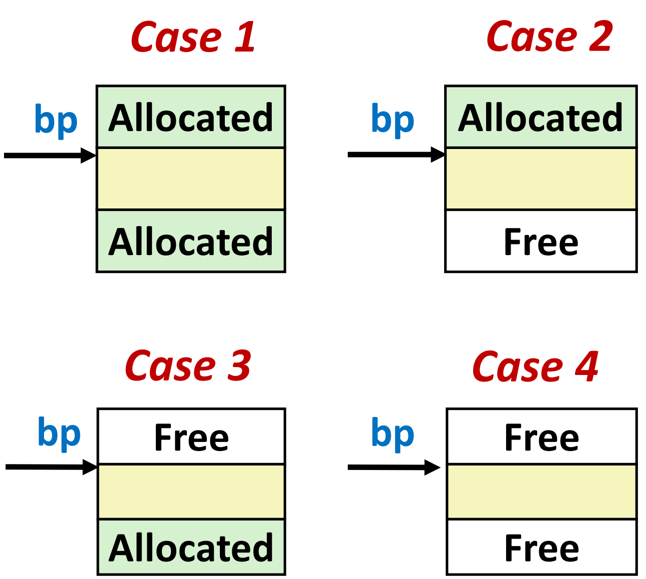
If enough space is left, split the list and update metadata

If left-over size is less than the minimum block size (16 bytes), ignore (internal fragmentation)

Implementing a Simple Allocator (Cont)

- Free allocated heap

```
void mm_free(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));

    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    coalesce(bp);
}

```




```
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

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

    else if (prev_alloc && !next_alloc) {      /* Case 2 */
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size,0));
    }

    else if (!prev_alloc && next_alloc) {      /* Case 3 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }

    else {                                     /* Case 4 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
            GET_SIZE(FTRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    return bp;
}
```

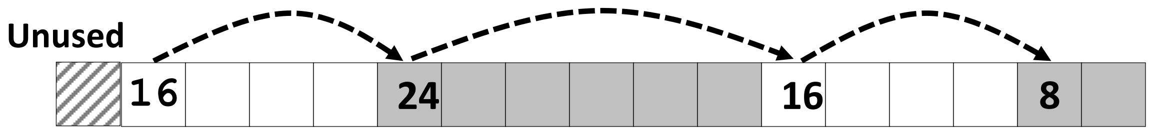
code/vm/malloc/mm.c

Implicit Lists Summary

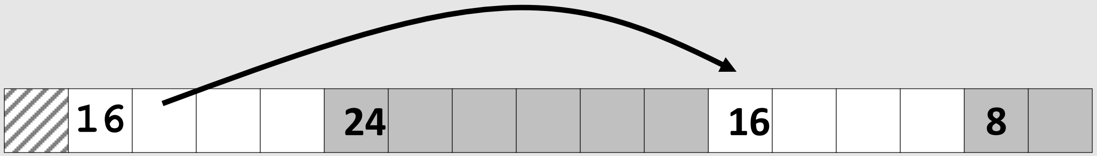
- **Implementation: very simple**
- **Allocate cost**
  - linear time worst case
- **Free cost**
  - constant time worst case
  - even with coalescing
- **Memory usage**
  - will depend on placement policy
  - First-fit, next-fit or best-fit
- **Not used in practice for `malloc/free` because of linear-time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

Keeping Track of Free Blocks

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



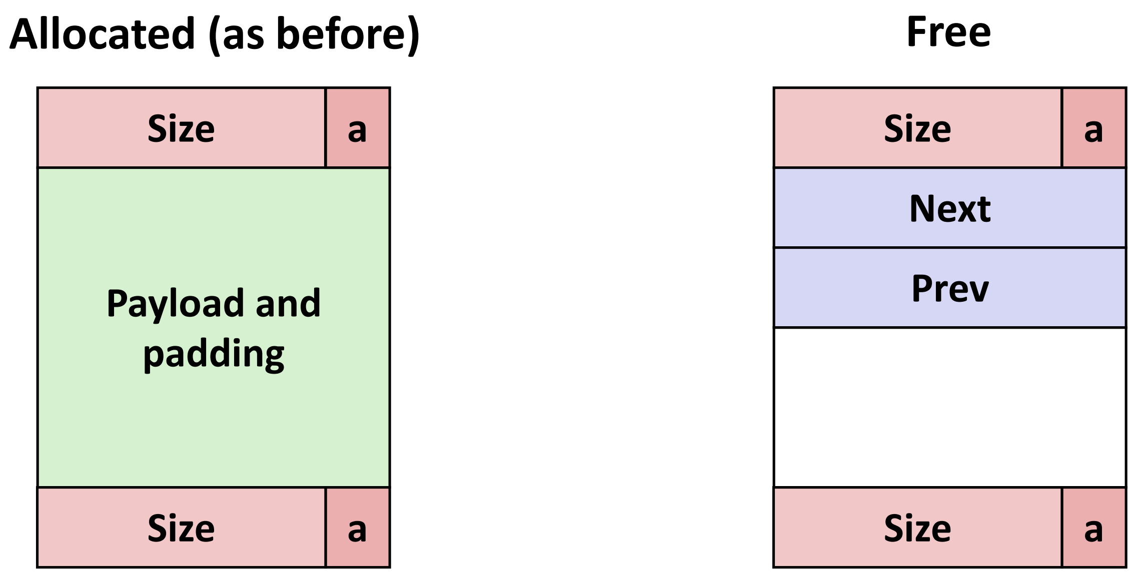
- **Method 2: *Explicit list* among the free blocks using pointers**



- **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

Method 2: Explicit Free Lists

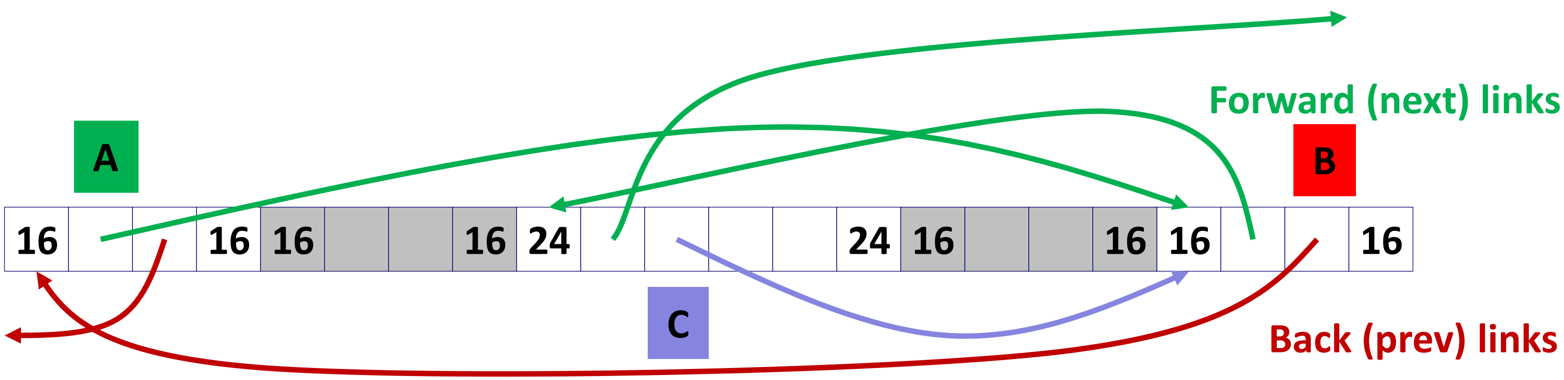


- **Maintain list(s) of *free* blocks, not *all* blocks**
  - The *next* free block could be anywhere
    - So, we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
  - Luckily, we track only free blocks, so we can use payload area

Explicit Free Lists

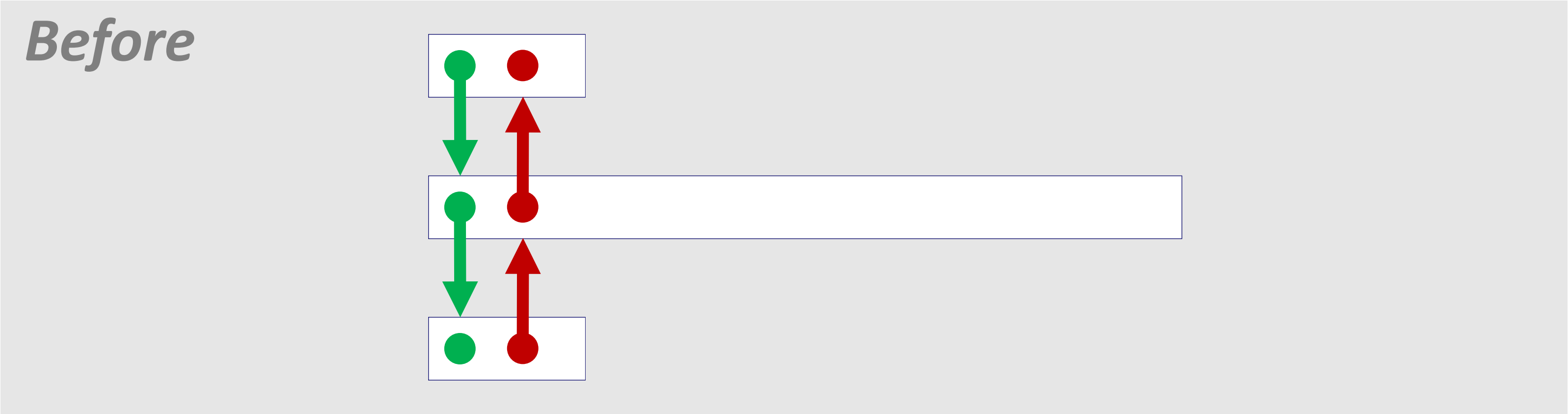
- Logically:

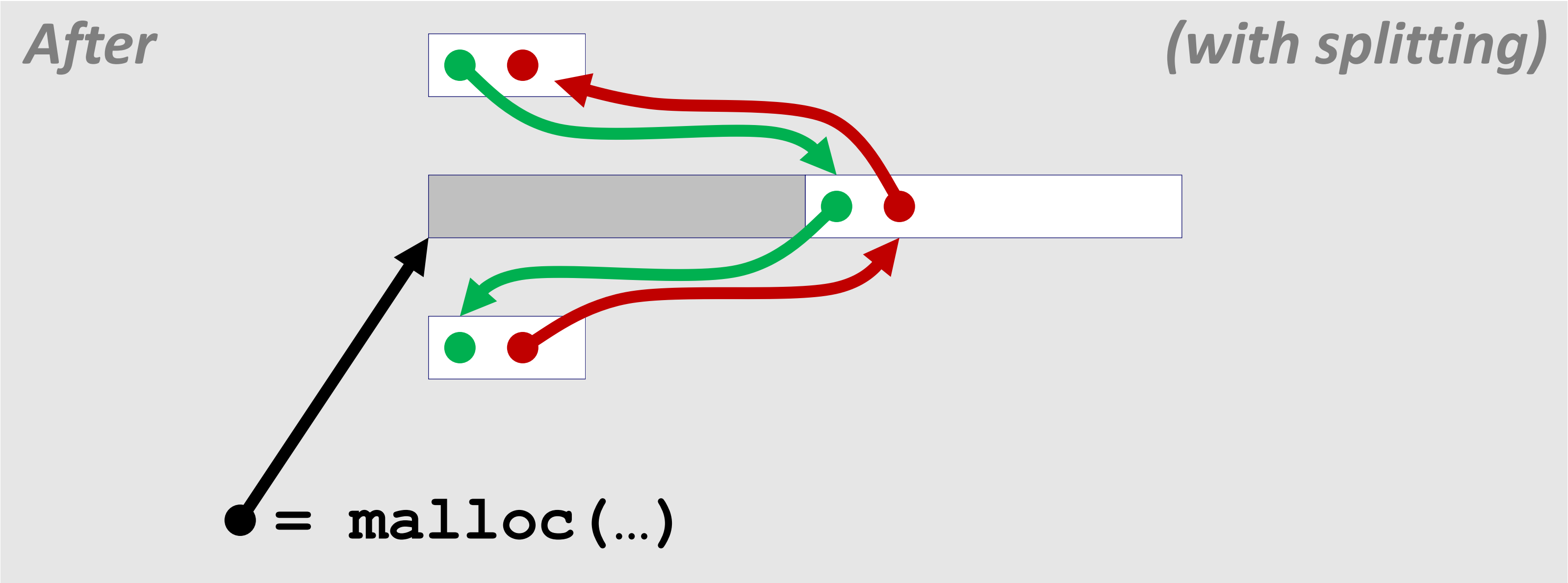
- Physically: blocks can be in any order



Allocating From Explicit Free Lists

conceptual graphic





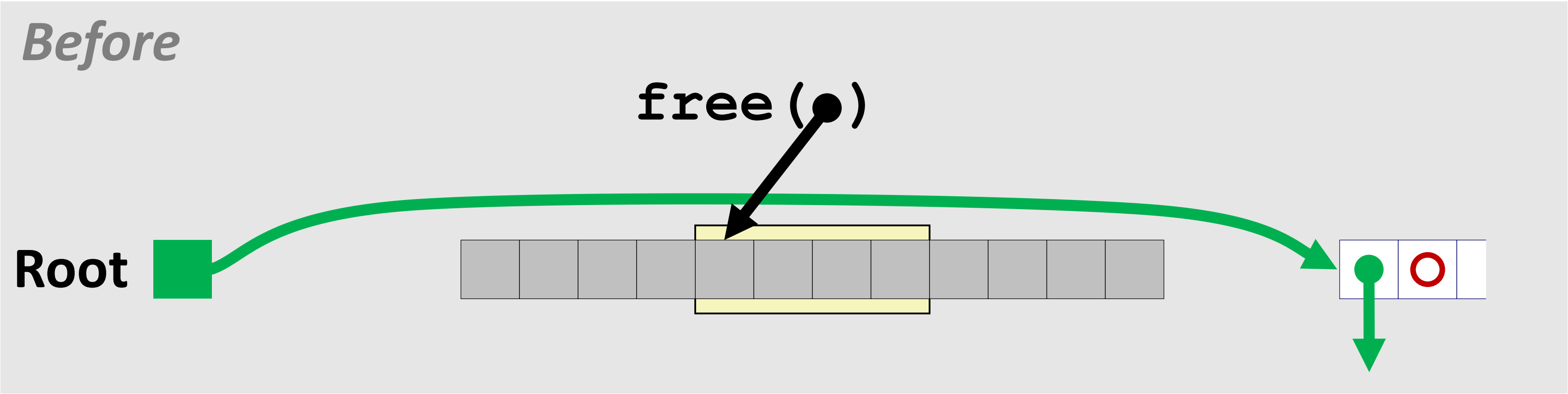
Freeing With Explicit Free Lists

- ***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 that fragmentation is *worse* than address ordered
- **Address-ordered policy**
  - Insert freed blocks so that free list blocks are always in *address order*

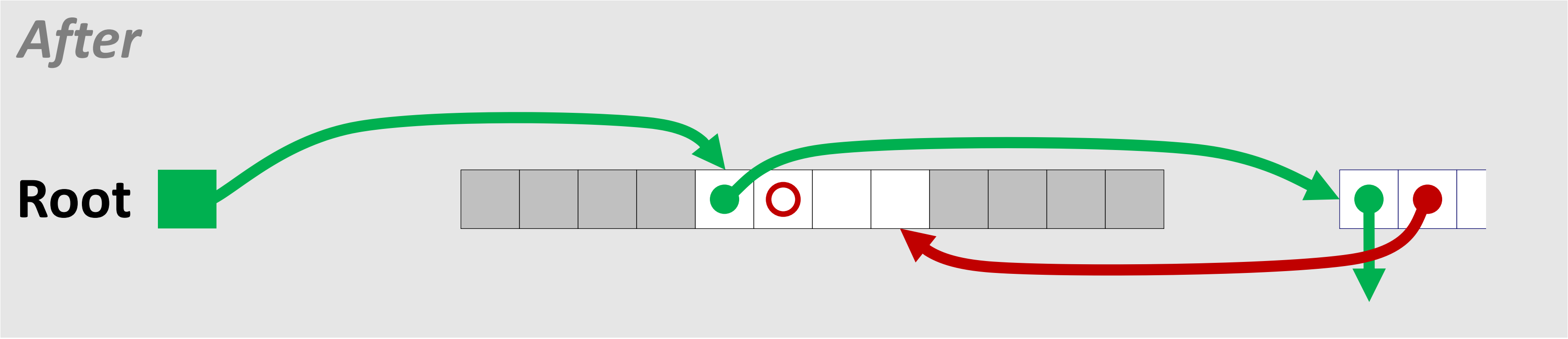
    addr(prev) < addr(curr) < addr(next)
  - ***Con:*** requires search
  - ***Pro:*** studies suggest that fragmentation is *lower* than LIFO/FIFO

Freeing With a LIFO Policy (Case 1)

conceptual graphic

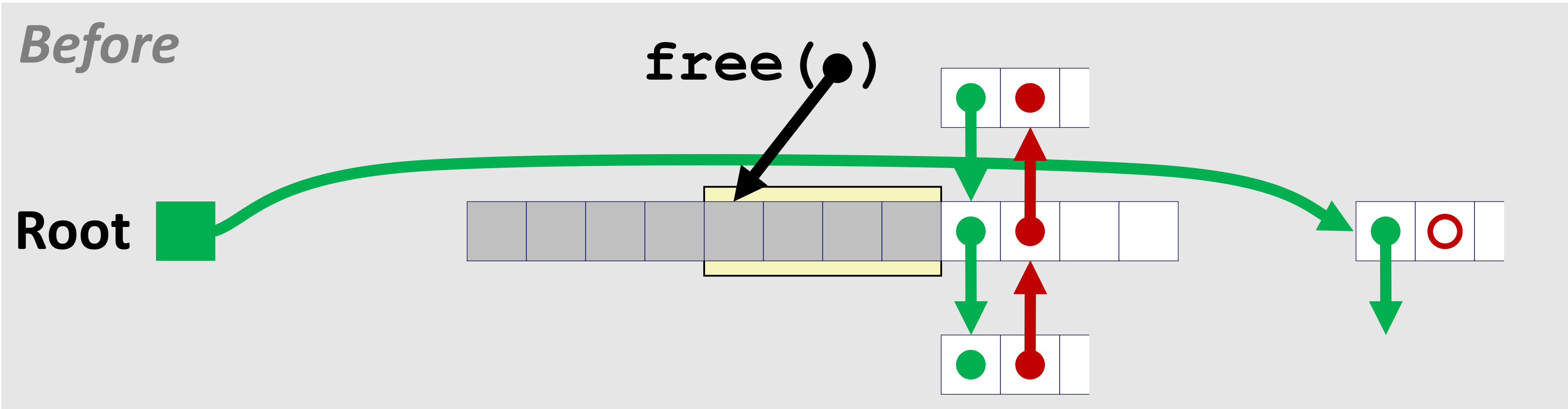


- Insert the freed block at the root of the list

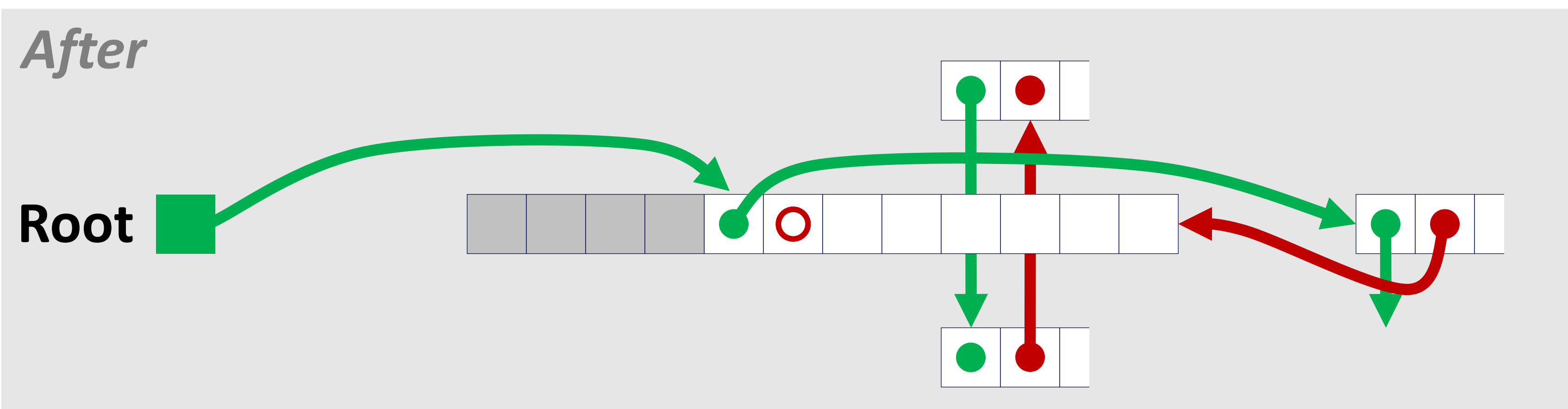


Freeing With a LIFO Policy (Case 2)

conceptual graphic

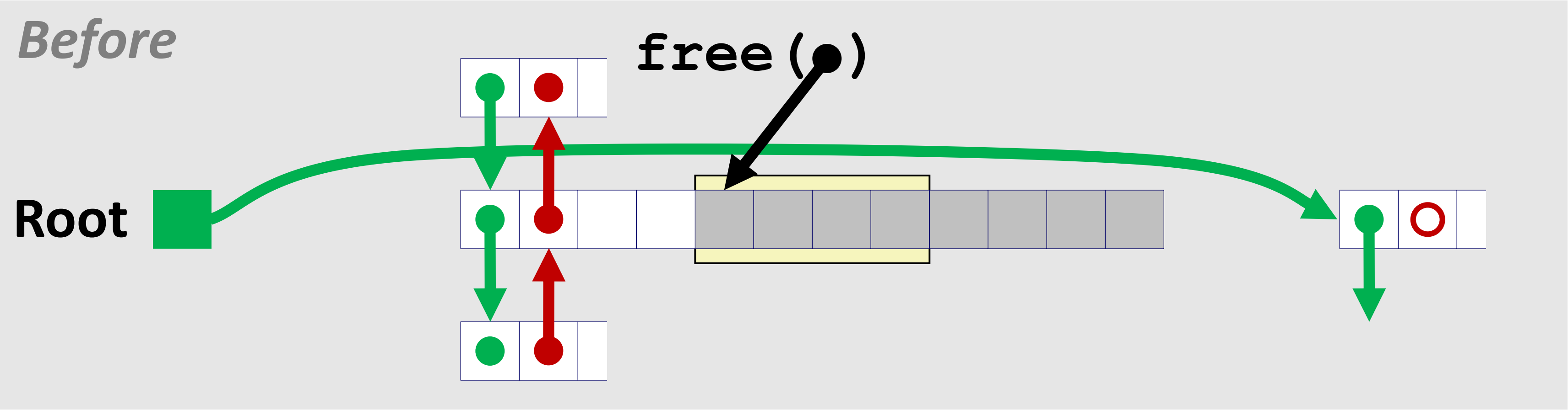


- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

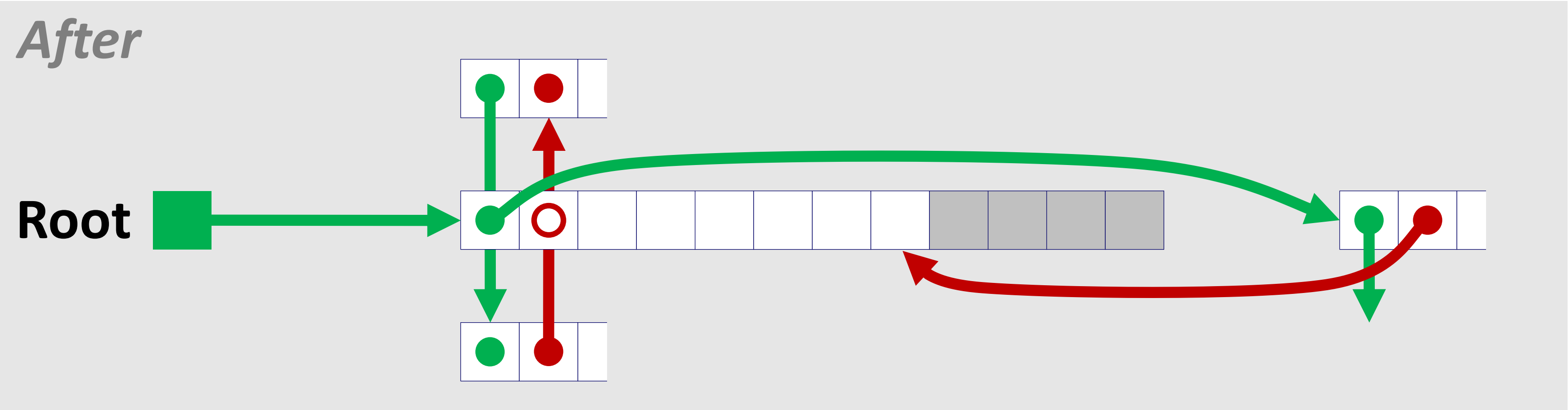


Freeing With a LIFO Policy (Case 3)

conceptual graphic

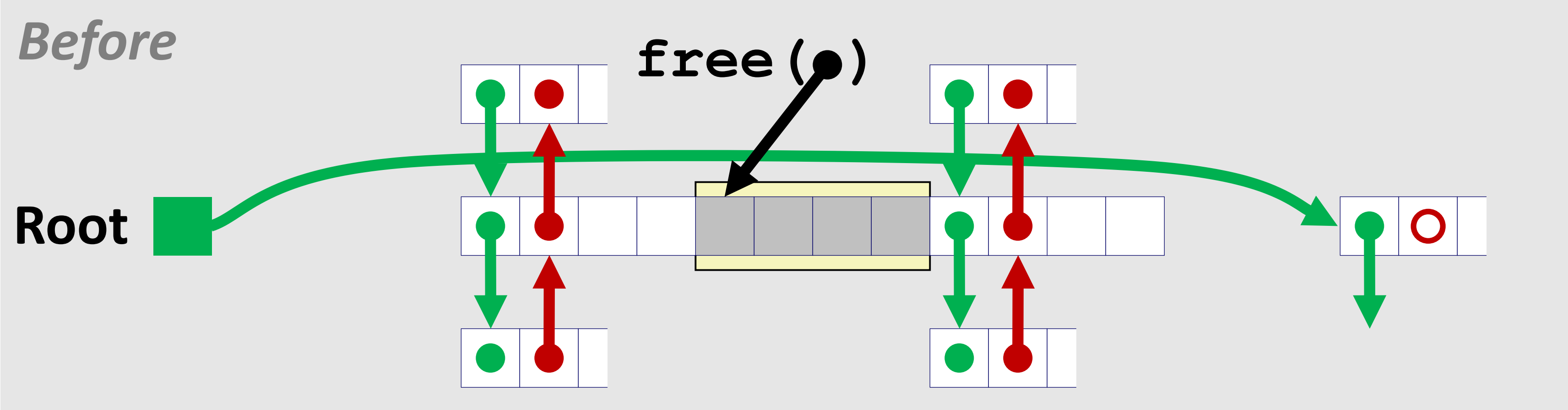


- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list

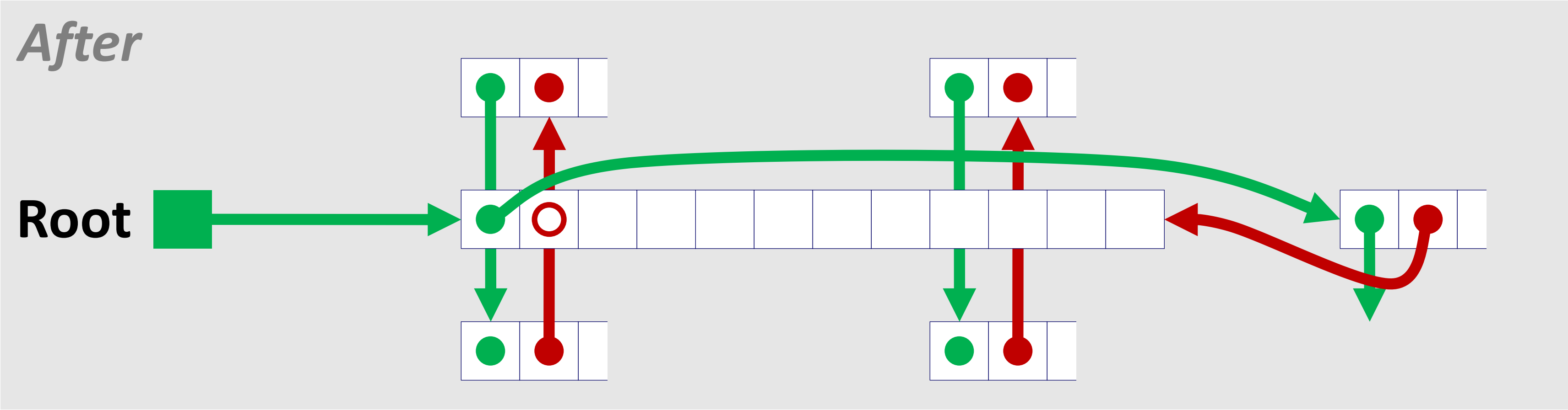


Freeing With a LIFO Policy (Case 4)

conceptual graphic



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



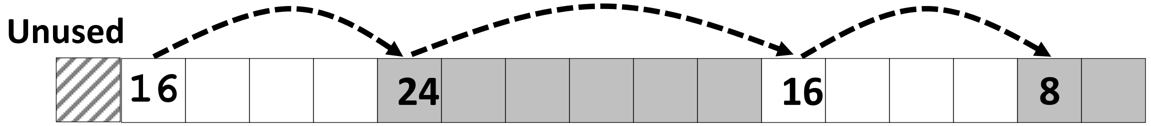
Explicit List Summary

- **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 since splicing blocks in and out of the list is needed
  - Some extra space for the links (2 extra words needed for each block)
    - Does this increase internal fragmentation?

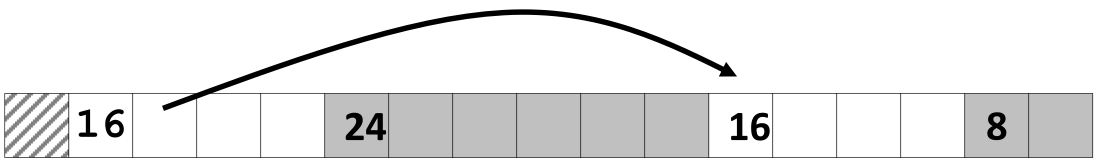
- **Most common use of linked lists is in conjunction with segregated free lists**
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

Keeping Track of Free Blocks

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



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

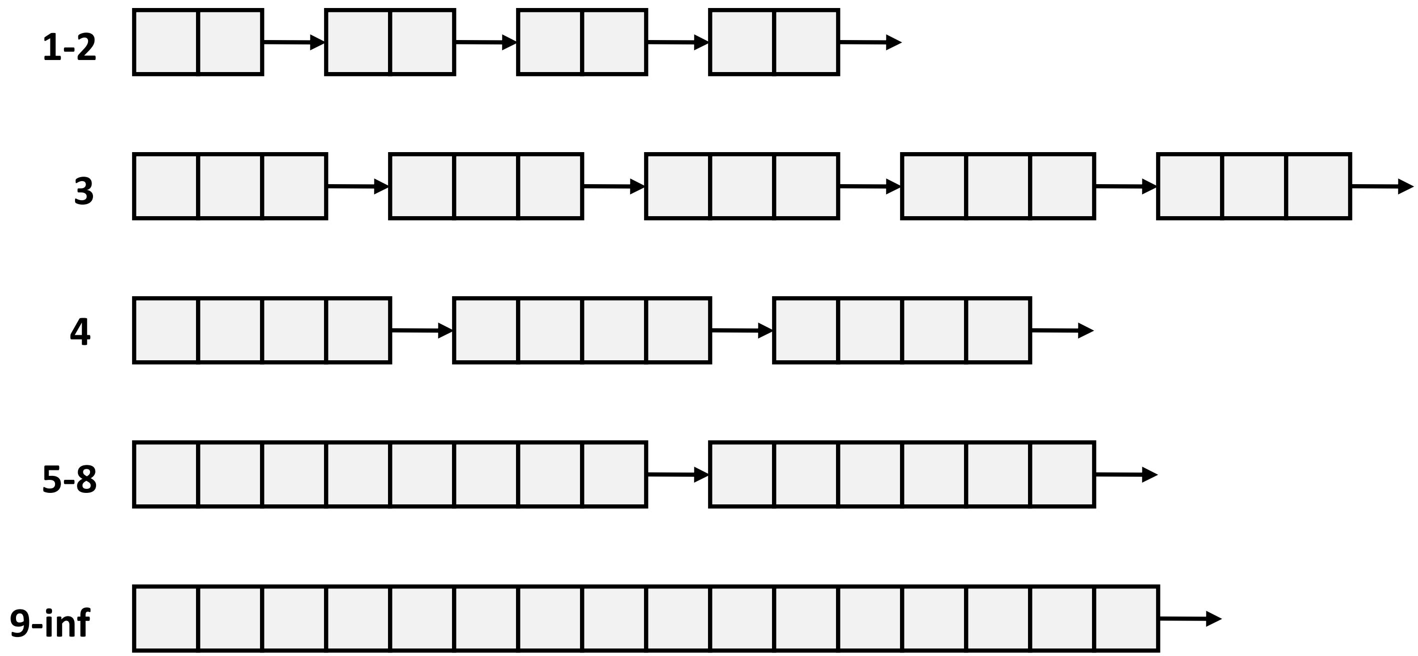


- **Method 3: *Segregated free list***
  - Different free lists for different size classes

- **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

Method 3: Segregated List (Seglist) Allocators

- Each ***size class*** of blocks has its own free list



- Often have separate classes for each small size
- For larger sizes: One class for each power-of-two (2^n) size

Seglist Allocator

- **Given an array of free lists, each one for some size class**

- **To allocate a block of size *n***
  - Search appropriate free list for block of size $m > n$
  - If an appropriate block is found
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found

- **If no block is found**
  - Request additional heap memory from OS (using **`sbrk()`**)
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in the largest size class

Seglist Allocator (Cont)

- **To free a block**
  - Coalesce and place on appropriate list

- **Advantages of seglist allocators**
  - Higher throughput
    - Log time for power-of-two size classes
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of entire heap
    - Extreme case: Giving each block its own size class is equivalent to best-fit

More Info on Allocators

- **D. Knuth, "*The Art of Computer Programming*", 2nd edition, Addison Wesley, 1973**
  - The classic reference on dynamic storage allocation

- **Wilson et al, "*Dynamic Storage Allocation: A Survey and Critical Review*", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept., 1995**
  - Comprehensive survey

Implicit Memory Management

- ***Garbage collection:*** **automatic reclamation of heap-allocated storage—application never has to free**

```
void foo() {
   int *p = malloc(128);
   return; /* p block is now garbage */
}
```

- **Common in many dynamic languages**
  - Python, Ruby, Java, Perl, ML, Lisp, Mathematica
- **Variants ("conservative" garbage collectors) exist for C and C++**
  - However, cannot necessarily collect all garbage

Garbage Collection

- **How does the memory manager know when memory can be freed?**
  - In general, we cannot know when memory is going to be used in the future
  - But, we can tell that certain blocks cannot be used if there are *no pointers* to them

- **Must make certain assumptions about pointers**
  - Memory manager can distinguish pointers from non-pointers
  - All pointers point to the start of a block
  - Cannot hide pointers
    (e.g., by type casting them to an **`int`**, and then back again)

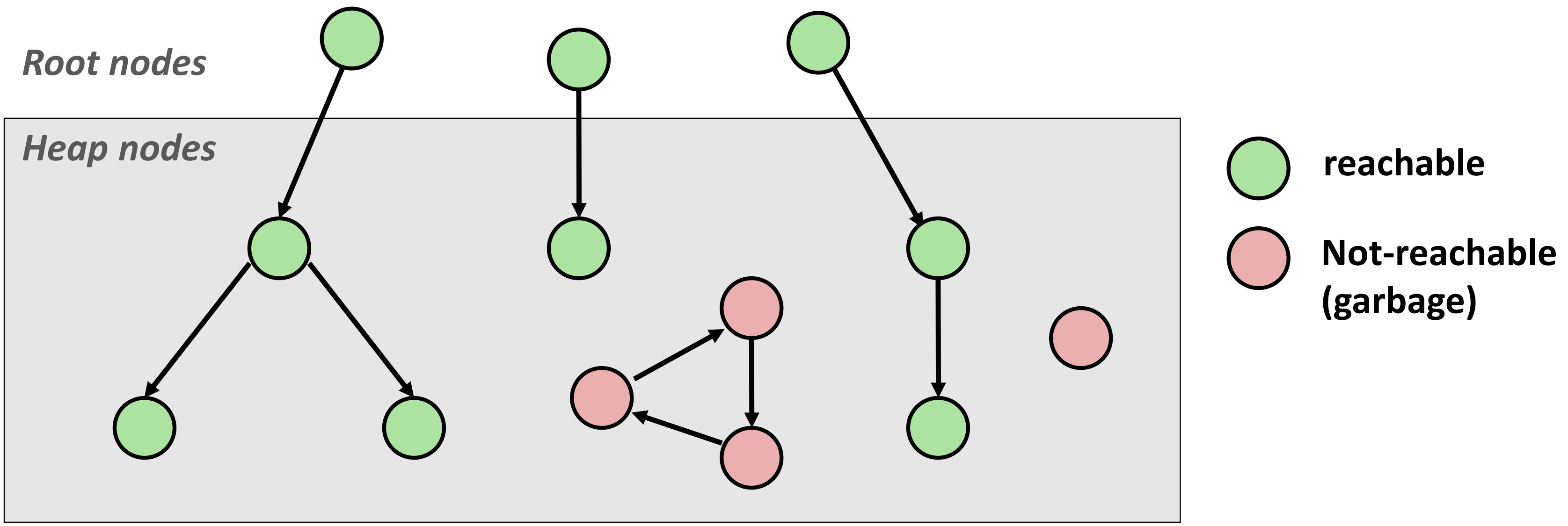
Classical GC Algorithms

- **Mark-and-sweep collection (McCarthy, 1960)**
  - Does not move blocks (unless you also *compact*)
- **Reference counting (Collins, 1960)**
  - Does not move blocks (not discussed)
- **Copying collection (Minsky, 1963)**
  - Moves blocks (not discussed)
- **Generational Collectors (Lieberman and Hewitt, 1983)**
  - Collection based on lifetimes (not discussed)

- **For more information:**
  Jones and Lin, "*Garbage Collection: Algorithms for Automatic Dynamic Memory*", John Wiley & Sons, 1996

Memory as a Graph

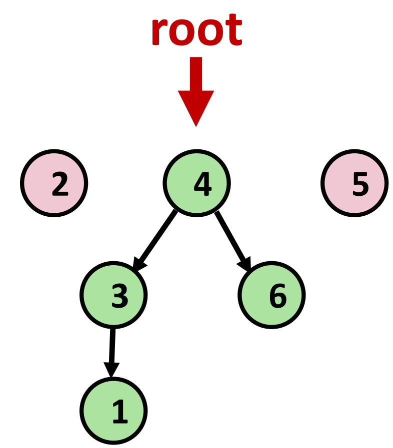
- **View memory as a directed graph**
  - Each block is a *node* in the graph
  - Each pointer is an *edge* in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root nodes*** (e.g., registers, variables on the stack, global variables)



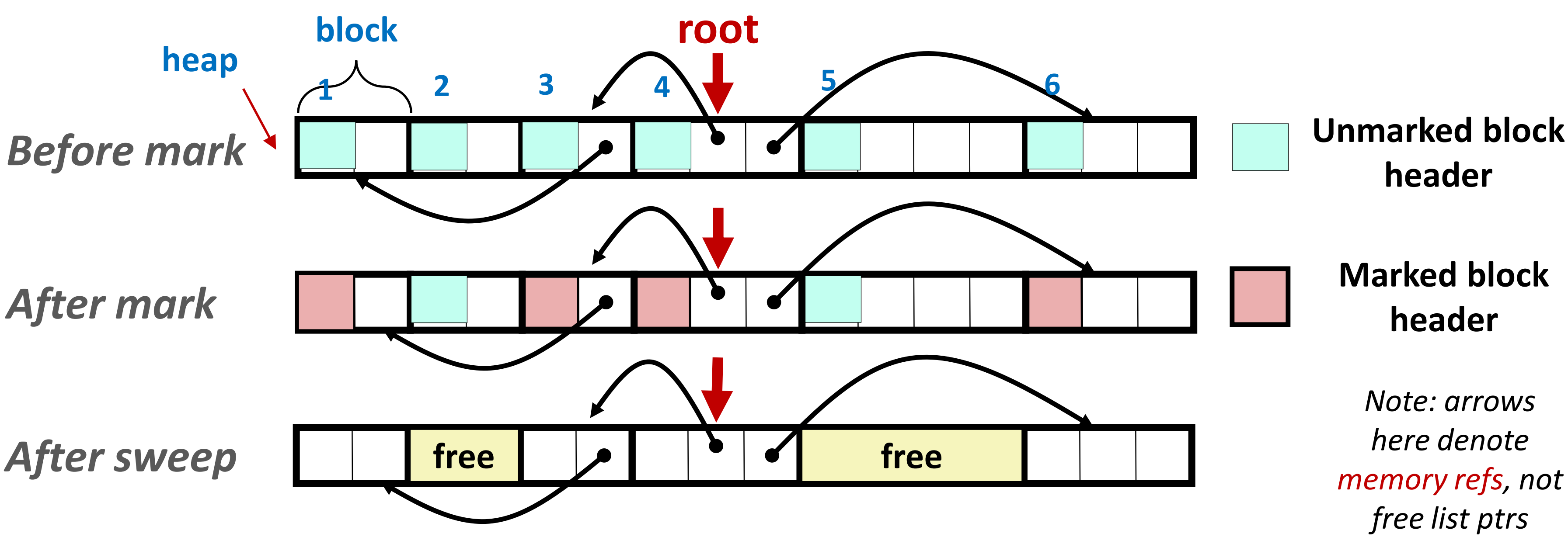
A node (block) is *reachable* if there is a path from any root node to that node

Non-reachable nodes are *garbage* (can never be used again by the application)

Mark and Sweep Collection



- Can build on top of `malloc/free` package
  - Allocate using `malloc` until you run out of space
- When we run out of space
  - Use extra ***mark bit*** in the header of each block
  - ***Mark phase***: Start at roots and set mark bit on each reachable block
  - ***Sweep phase***: Scan all blocks and free blocks that are not marked



Pseudocode for Mark and Sweep

- **Pseudocode**

Call mark function for each root node (mark phase)

(a) mark function

```
void mark(ptr p) {
    if ((b = isPtr(p)) == NULL)
      return;
    if (blockMarked(b))
      return;
    markBlock(b);
    len = length(b);
    for (i=0; i < len; i++)
      mark(b[i]);
    return;
}
```

Called once (sweep phase)

(b) sweep function

```
void sweep(ptr b, ptr end) {
    while (b < end) {
        if (blockMarked(b))
            unmarkBlock(b);
        else if (blockAllocated(b))
            free(b);
        b = nextBlock(b);
    }
    return;
}
```

- **Helper functions**

Our description of Mark&Sweep will assume the following functions, where `ptr` is defined as `typedef void *ptr`:

`ptr isPtr(ptr p)`. If `p` points to some word in an allocated block, it returns a pointer `b` to the beginning of that block. Returns NULL otherwise.

`int blockMarked(ptr b)`. Returns true if block `b` is already marked.

`int blockAllocated(ptr b)`. Returns true if block `b` is allocated.

`void markBlock(ptr b)`. Marks block `b`.

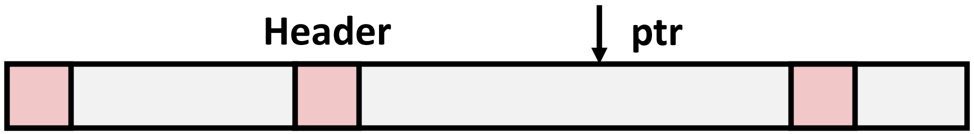
`int length(ptr b)`. Returns the length in words (excluding the header) of block `b`.

`void unmarkBlock(ptr b)`. Changes the status of block `b` from marked to unmarked.

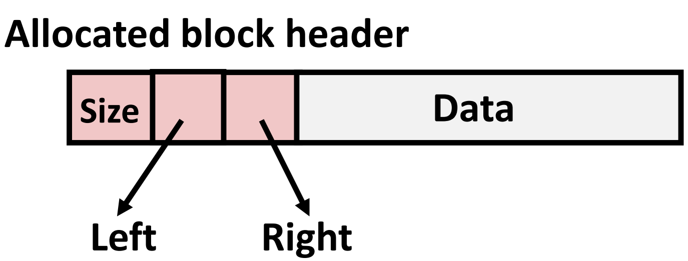
`ptr nextBlock(ptr b)`. Returns the successor of block *b* in the heap.

Conservative Mark & Sweep in C

- A **conservative garbage collector** for C programs
  - `is_ptr(p)` determines whether `p` is a pointer by checking if it points to an allocated block of memory but it is hard to determine in C
  - C pointers can point to the middle of a block



- How to find the beginning of a block?
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)
  - Balanced-tree pointers can be stored in the header of each allocated block -> `is_ptr(p)` uses this tree to see if **p** falls within the block



Left: smaller addresses
Right: larger addresses

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing non-existent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

C Operators and Precedence

Postfix

<table>
<tr><th>Operators</th><th>Associativity</th></tr>
<tr><td><code>() [] -> . ++ --</code></td><td>left to right</td></tr>
<tr><td><code>! ~ ++ -- + - * & (type) sizeof</code></td><td>right to left</td></tr>
<tr><td><code>* / %</code></td><td>left to right</td></tr>
<tr><td><code>+ -</code></td><td>left to right</td></tr>
<tr><td><code><< >></code></td><td>left to right</td></tr>
<tr><td><code>< <= > >=</code></td><td>left to right</td></tr>
<tr><td><code>== !=</code></td><td>left to right</td></tr>
<tr><td><code>&</code></td><td>left to right</td></tr>
<tr><td><code>^</code></td><td>left to right</td></tr>
<tr><td><code>|</code></td><td>left to right</td></tr>
<tr><td><code>&&</code></td><td>left to right</td></tr>
<tr><td><code>||</code></td><td>left to right</td></tr>
<tr><td><code>?:</code></td><td>right to left</td></tr>
<tr><td><code>= += -= *= /= %= &= ^= != <<= >>=</code></td><td>right to left</td></tr>
<tr><td><code>,</code></td><td>left to right</td></tr>
</table>

Unary

Prefix

Binary

Source: K&R page 53, updated

- `->`, `()`, and `[]` have high precedence, with `*` and `&` just below
- Unary `+`, `-`, and `*` have higher precedence than binary forms

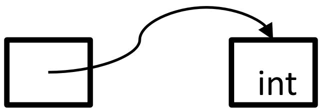
C Pointer Declarations: Test Yourself!

`int *p`	p is a pointer to int
`int **p`	p is a pointer to a pointer to an int
`int *p[4]`	p is an array[4] of pointer to int
`int *(p[4])`	p is an array[4] of pointer to int
`int (*p)[4]`	p is a pointer to an array[4] of int
`int *func()`	func is a function returning a pointer to int
`int (*func)()`	func is a pointer to a function returning int

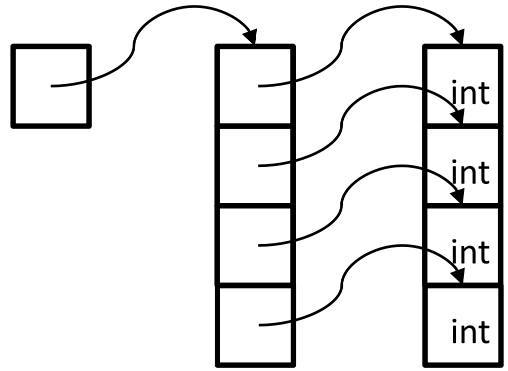
Source: K&R Sec 5.12

Example

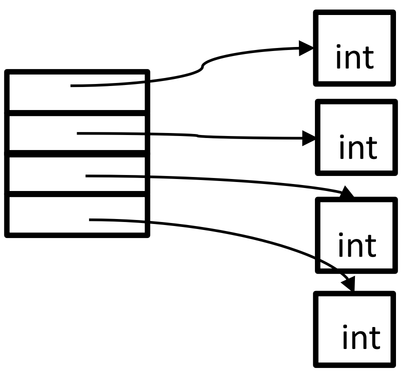
`int *p`



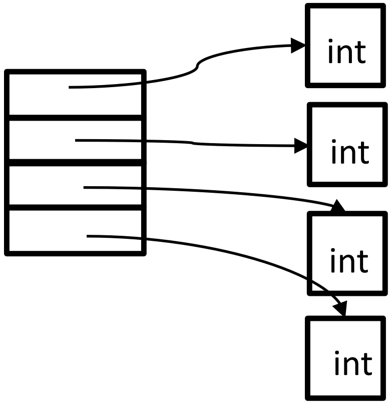
`int **p`



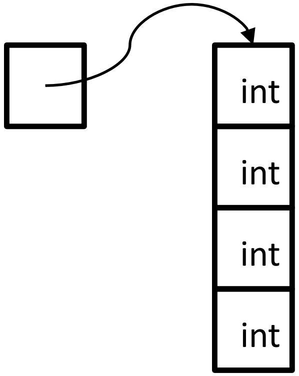
`int *p[4]`



`int *(p[4])`



`int (*p)[4]`



`int *f()`

int *p = f()

`int (*f)()`

f is a pointer to a function returning int

Dereferencing Bad Pointers

- The classic `scanf` bug

```
int val;

...

scanf("%d", val);
```

Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for ( i = 0; i < N; i++ )
      for ( j = 0; j < N; j++ )
         y[i] += A[i][j] * x[j];
   return y;
}
```

Overwriting Memory (Case 1)

- **Allocating the (possibly) wrong sized object**

```
int **p;

p = malloc(N*sizeof(int)); /* sizeof(int *)*/

for ( i = 0; i < N; i++ ) {
   p[i] = malloc(M*sizeof(int));
}
```

Return values for sizeof(int) and sizeof(int *) are different in 64 bit machines!

Overwriting Memory (Case 2)

- Off-by-one errors

```
char **p;

p = malloc(N*sizeof(int *));

for ( i = 0; i <= N; i++ ) {
   p[i] = malloc(M*sizeof(int));
}
```

```
char *p;

p = malloc(strlen(s));
strcpy(p,s);
```

Copy including null character

Overwriting Memory (Case 3)

- Not checking the max string size

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

Basis for classic buffer overflow attacks!

Overwriting Memory (Case 4)

- Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {

   while ( *p && *p != val )
      p += sizeof(int); /* should be p++ */

   return p;
}
```

Overwriting Memory (Case 5)

- Referencing a pointer instead of the object it points to

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--; /* This should be (*size)-- */
   Heapify(binheap, *size, 0);
   return(packet);
}
```

Operators	Associativity
`() [] -> . ++ --`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= != <<= >>=`	right to left
`,`	left to right

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```
int *foo () {
   int val;

   return &val;
}
```

Freeing Blocks Multiple Times

- Nasty!

```
x = malloc(N*sizeof(int));
        <manipulate x>
free(x);

y = malloc(M*sizeof(int));
        <manipulate y>
free(x);
```

Referencing Freed Blocks

- **Evil!**

```
x = malloc(N*sizeof(int));
  <manipulate x>
free(x);
    ...
y = malloc(M*sizeof(int));
for ( i = 0; i < M; i++ )
   y[i] = x[i]++;
```

Failing to Free Blocks (Memory Leaks)

- Slow, long-term killer!

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return; /* x is garbage at this point */
}
```

Dealing With Memory Bugs

- **Debugger: `gdb`** (https://www.sourceware.org/gdb/)
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- **Binary translator: `valgrind`** (https://valgrind.org/)
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Checks each individual reference at runtime
    - Bad pointers, overwrites, references outside of allocated block
- **glibc `malloc` contains checking code**
  - `setenv MALLOC_CHECK_ 3`
  - `int mallopt(int param, int value)`