- Lecture Notes for Lecture 7 of CS 5600 (Computer Systems) for the Fall, 2019 session at the Northeastern University Silicon Valley Campus.
- *Managing Memory*

- Philip Gust,
  Clinical Instructor
  Khoury College of Computer and Information Science

- *Part 2 of a two-part lecture on managing memory*

Lecture 6 Review

- In the previous lecture we turned our attention to how memory is managed in a C program, including the use of the stack and heap segments.
- We saw that the stack segment is where *automatic memory* is managed. This is the storage used when a procedure is called. We looked at the call process, and how the stack is managed.
- We also began looking at how the heap segment is where dynamic memory is managed. This is storage allocated when malloc() is called in C or the *new* operator is used in C++.

Managing Memory

Overview

- In this lecture we continuing to explore memory management by looking in more detail at how heap storage is managed and look at how the original C dynamic memory allocator was designed.
- As we will see, there are several issues that memory allocators must address. How they address these issues depends on assumptions about allocation patterns and memory usage.
- We will also consider ways of optimizing how memory is managed using several advanced techniques for representing the free list and recording additional information.

Managing Memory

Managing the heap segment

- One of the challenges of memory allocators is fragmentation. This occurs when otherwise unused memory is not available to satisfy allocation requests. There are two causes.
  - *Internal fragmentation* occurs when a block is allocated that has more space than is required by an allocation request. This is quantified as sum of the unused space in all allocated blocks.
  - *External fragmentation* occurs when there is enough aggregate free memory to satisfy a request, but no single block free block is large enough for the request. This is much harder to quantify.

Managing Memory

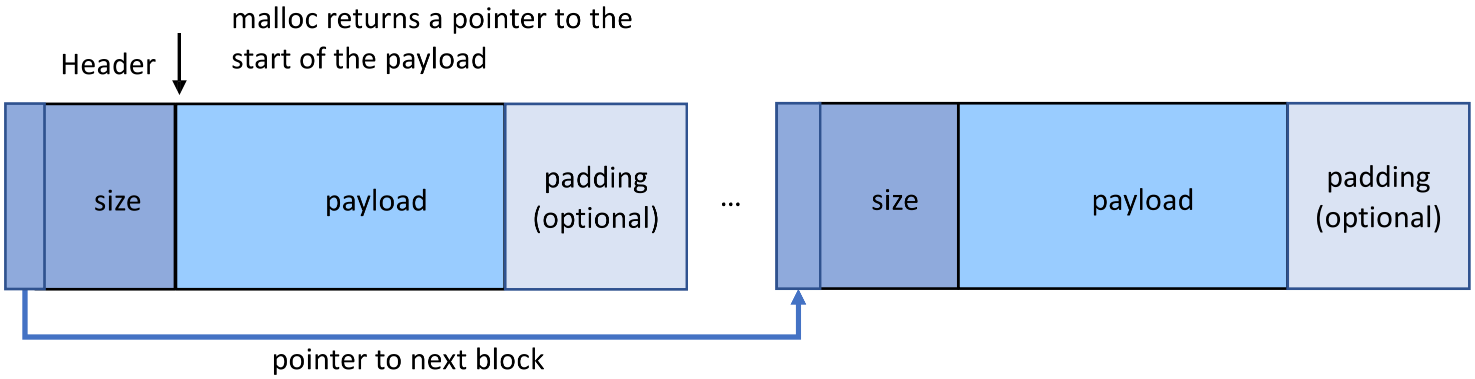
Managing the heap segment

- A the design of a practical allocator must strikes a balance between throughput and utilization by considering the following issues:
  - *Free block organization:* How do we keep track of free blocks?
  - *Placement:* How do we choose an appropriate free block in which to place a newly allocated block?
  - *Splitting:* After we place a newly allocated block in some free block, what do we do with the remainder of the free block?
  - *Coalescing:* What do we do with a block that has just been freed?

Managing Memory

Managing the heap segment

- The K&R allocator manages storage as an *explicit block list* that contains a header with the a pointer to the next block and the block size.
- Only free blocks are in the list. Allocated blocks are returned to the free list when being freed, in memory address order to facilitate coalescing free blocks.



Managing Memory

Managing the heap segment

- Allocators must pay attention to memory alignment because there are hardware restrictions on what memory addresses a given type of data can occupy.
- Integers and floats generally must fall on 4-byte memory boundaries, while longs, doubles, and pointers must generally fall on 8-byte boundaries.
- Since the allocator is not aware of what type of data is being allocated, it must:
  - Make all memory blocks a multiple of the header size
  - Ensure header is aligned with largest data type (e.g., **size_t**)

Managing Memory

Managing the heap segment

- C based allocators use a union to force memory alignment. Here is the definition of a block header struct:

```
// Allocation unit for header of memory blocks
typedef union Header {
   struct {
      union Header *ptr;      // next block if on free list
      size_t size;            // size of this block
                                     //     including the Header itself
                                     //     measured in count of Header chunks
                                     //     not less than NALLOC Header's
   } s;
   max_allign_t _align;              // force alignment of blocks
} Header;
```

Managing Memory

Managing the heap segment

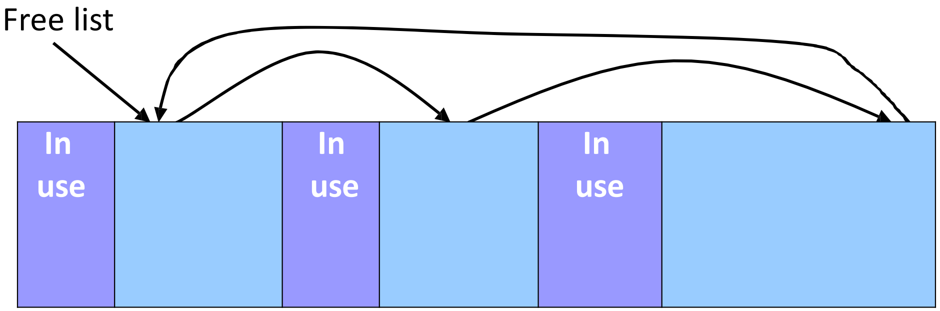
- Keep memory aligned
  - Requested size is rounded up to multiple of header size
- Rounding up when asked for nbytes
  - Header has size **sizeof(Header)**
  - Round: **(nbytes + sizeof(Header) – 1) / sizeof(Header)**
- Allocate space for user data, plus the header itself

```
void *mm_malloc(size_t nbytes) {

    /* smallest count of Header-sized memory chunks */
    /*  (+1 additional chunk for the Header itself) needed to hold nbytes */
    size_t nunits = (nbytes + sizeof(Header) - 1) / sizeof(Header) + 1;
```

Managing Memory

Managing the heap segment

- Free blocks, linked together
  - Example: circular linked list
- Keep list in order of increasing addresses
  - Makes it easier to coalesce adjacent free blocks



Managing Memory

Managing the heap segment

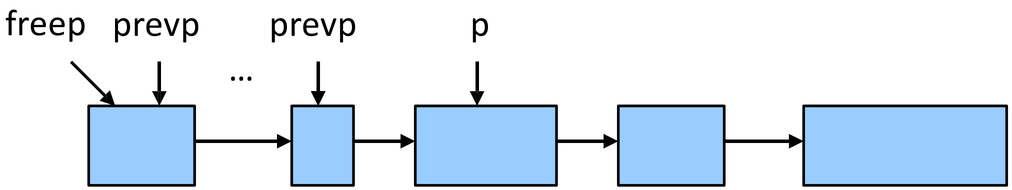
- Handling a request for memory (e.g. malloc)
    - Find a free block that satisfies the request
    - Must have a ”size” that is big enough or bigger
- Which block to return?
    - First-fit algorithm
        - Keep a linked list of free blocks
        - Search for the *first* one that is big enough
    - Best-fit algorithm
        - Keep a linked list of free blocks
        - Search for the *smallest* one that is big enough
        - Helps avoid fragmenting the free memory

Managing Memory

Managing the heap segment

First-fit algorithm

- Start at the beginning of the list (freep)
- Sequence through the list
  - Keep a pointer to the previous element (prevp)
- Stop when reaching first block that is big enough
  - Patch up the list
  - Return a block to the user

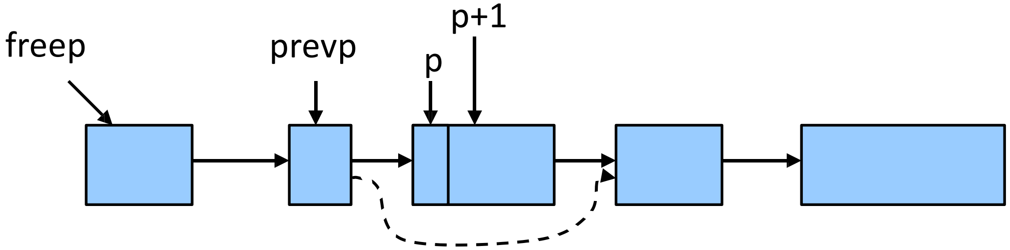


Managing Memory

Managing the heap segment

First case: a perfect fit

- Suppose the first fit is a perfect fit
  - Remove the element from the list
  - Link the previous element with the next element
    prev->s.ptr = p->s.ptr
  - Return current element to the user skipping header
    return (void *) (p+1)



Managing Memory

Managing the heap segment

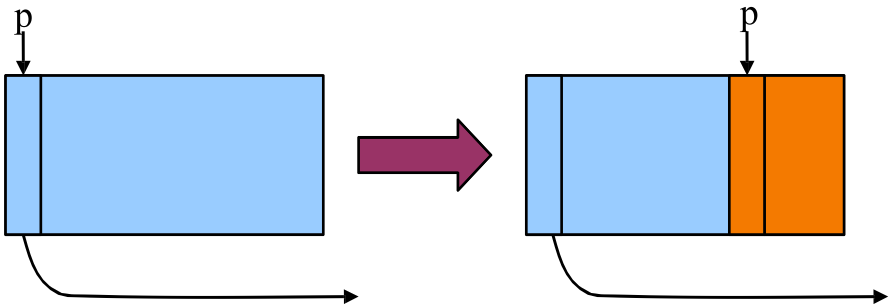
Second case: block is too big

- Suppose the block is bigger than requested
  - Divide the free block into two blocks
  - Keep first (now smaller) block in the free list

    p->s.size -= nunits;
  - Allocate the second block to the user

    p += p->s.size;

    p->s.size = nunits



Managing Memory

Managing the heap segment

Combining the two cases

```
if (p->s.size >= nunits) {         /* big enough */
        if (p->s.size == nunits)        /* exactly */
           prevp->s.ptr = p->s.ptr;
        else {                       /* split and allocate tail end */
           /* adjust the size to split the block */
           p->s.size -= nunits;
           /* find the address to return */
           p += p->s.size;
           p->s.size = nunits;
        }
        freep = prevp;  // move the head
        return (void *)(p+1);
     }
```

Managing Memory

Managing the heap segment

Beginning of the free list

- Benefit of making free list a circular list
  - Any element in the list can be the beginning
  - Do not have to handle the “end” of the list as special
  - Optimization: make head be where last block was found

```
for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
    if (p->s.size >= nunits) {        /* big enough */
        // Do stuff on previous side
        freep = prevp; /* move the head */
        return (void *)(p+1);
    }
}
```

Managing Memory

Managing the heap segment

No block is big enough

- Cycling completely through the list
  - Check if the ”for” loop returns back to the head of the list

```
for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {          /* big enough */
                    // Do stuff on previous side
                    freep = prevp; /* move the head */
                    return (void *)(p+1);
        }
        if (p == freep)                  /* wrapped around free list */
                // Now, do something about it…
}
```

Managing Memory

Managing the heap segment

- The morecore() function requests additional storage from the operating system. Details are platform-specific.

```
/** Request additional memory to be added to this process
 * @param nu the number of Header-chunks to be added
 */
static Header *morecore(size_t nu) {
    /* get at least NALLOC Header-chunks from the OS */
    if (nu < NALLOC) {
        nu = NALLOC;
    }

    char *cp = (char *) sbrk(nu * sizeof(Header));
    if (cp == (char *) -1) {            /* no space at all */
        fprintf(stderr, "... sbrk() no space for %ld chunks\n", nu);
        return NULL;
    }
    fprintf(stderr, "... sbrk() claimed %ld chunks\n", nu);
    totmem += nu;                       /* keep track of allocated memory */
```

Managing Memory

Managing the heap segment

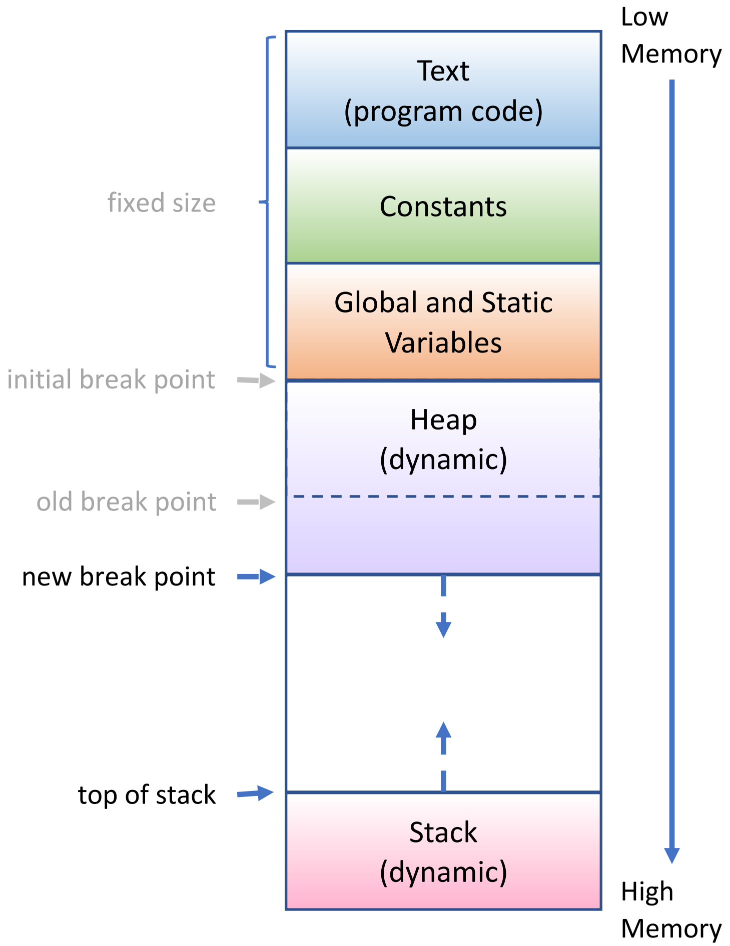
- The morecore() function requests additional storage from the operating system. Details are platform-specific.

```
    /** Request additional memory to be added to this process
    Header *up = (Header *) cp;
    up->s.size = nu;

    /* add the free space to the circular list */
    void *n = (void *)(up+1);
    mm_free(n);

    return freep;
}
```

Managing Memory

Managing the heap segment

- The brk() and sbrk() calls increase the heap by changing the *break point*, which marks the the top of the heap.
- The break point begins at the end of the global and static variables segment and increases as the heap expands.
- The break point cannot go past the top of the stack.



Managing Memory

Managing the heap segment

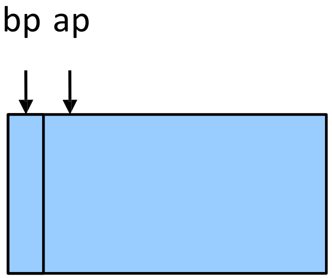
- Upon successful completion, brk() returns the new break point and sbrk returns the old break point. Both return -1 if no more memory available, with *errno* set to the ENOMEM.
- In the original Unix system, brk and sbrk were the only ways in which applications could acquire additional data space.
- Modern POSIX systems use other means to increase the heap space, but provide these function for backward compatibility.

Managing Memory

Managing the heap segment

Free

- User passes a pointer to the memory block
  void free(void *ap);
- Free function inserts block into the list
  - Identify the start of entry: **bp = (Header *) ap – 1;**
  - Find the location in the free list
  - Add to the list, coalescing entries, if needed



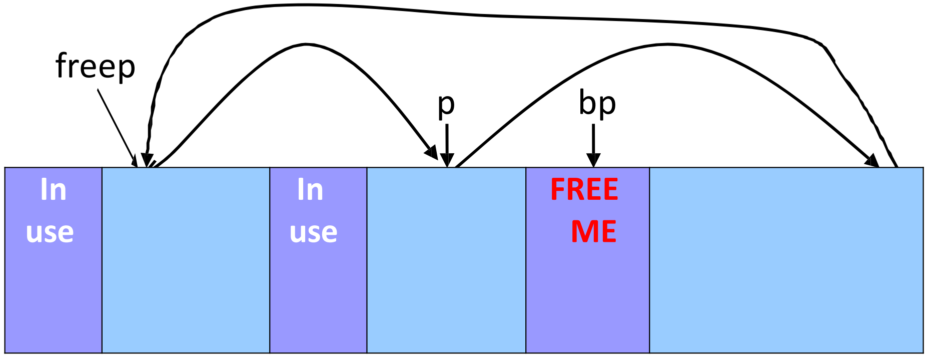
Managing Memory

Managing the heap segment

Scanning free list for the spot

- Start at the beginning: **p = freep;**
- Sequence through the list: **p = p->s.ptr;**
- Stop at last entry before the to-be-freed element

  (bp > p) && (bp < p->s.ptr);



Managing Memory

Managing the heap segment

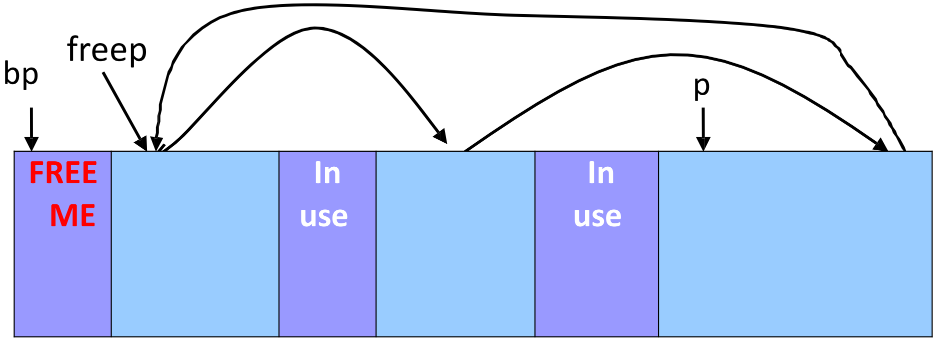
Corner cases: beginning or end

- Check for wrap-around in memory

  p >= p->s.ptr;

- See if to-be-freed element is located there

  (bp > p) || (bp < p->s.ptr)



Managing Memory

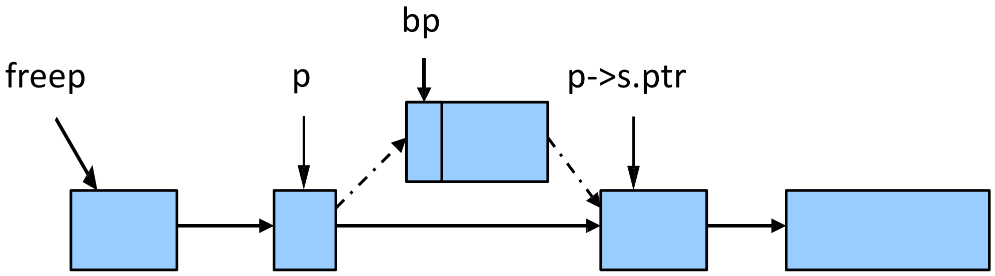
Managing the heap segment

Inserting into free list

- New element to add to free list: **bp**
- Insert in between **p** and **p->s.ptr**

```
bp->s.ptr = p->s.ptr;
p->s.ptr = bp;
```

- But, there may be opportunities to coalesce

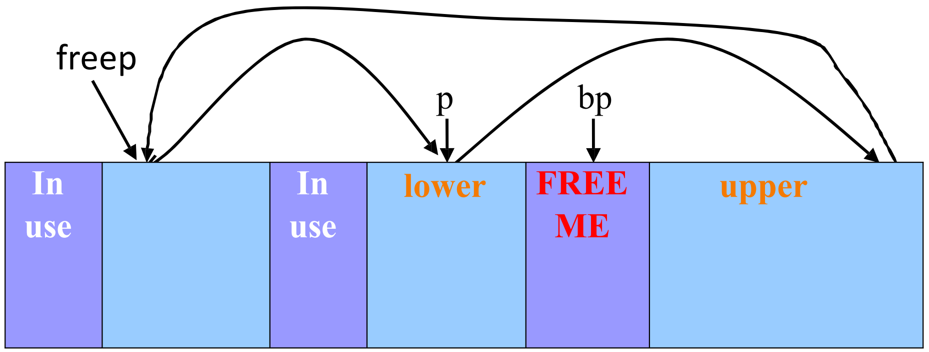


Managing Memory

Managing the heap segment

Coalescing with neighbors

- Scanning the list finds the location for inserting
  - Pointer to to-be-freed element: **bp**
  - Pointer to previous element in free list: **p**
- Coalescing into larger free blocks
  - Check if contiguous to upper and lower neighbors

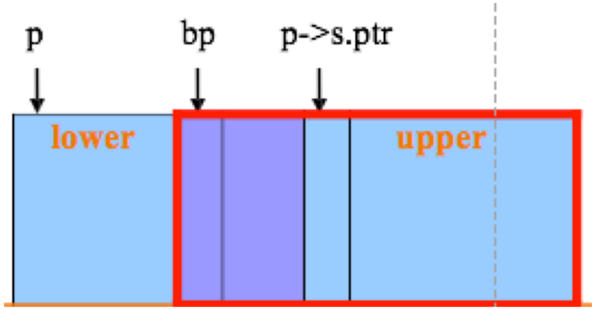


Managing Memory

Managing the heap segment

Coalesce with upper neighbor

- Check if next part of memory is in the free list
  if (bp + bp->s.size == p->s.ptr)
- If so, make into one bigger block
  - Larger size: **bp->s.size += p->s.ptr->s.size;**
  - Copy next pointer: **bp->s.ptr = p->s.ptr->s.ptr;**
- Else, simply point to the next free element
  bp->s.ptr = p->s.ptr;

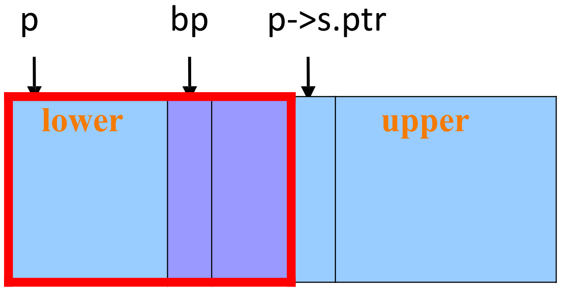


Managing Memory

Managing the heap segment

Coalesce with lower neighbor

- Check if previous part of memory is in the free list
  if (p + p->s.size == bp)
- If so, make into one bigger block
  - Larger size: **p->s.size += bp->s.size;**
  - Copy next pointer: p->s.ptr = bp->s.ptr;



Managing Memory

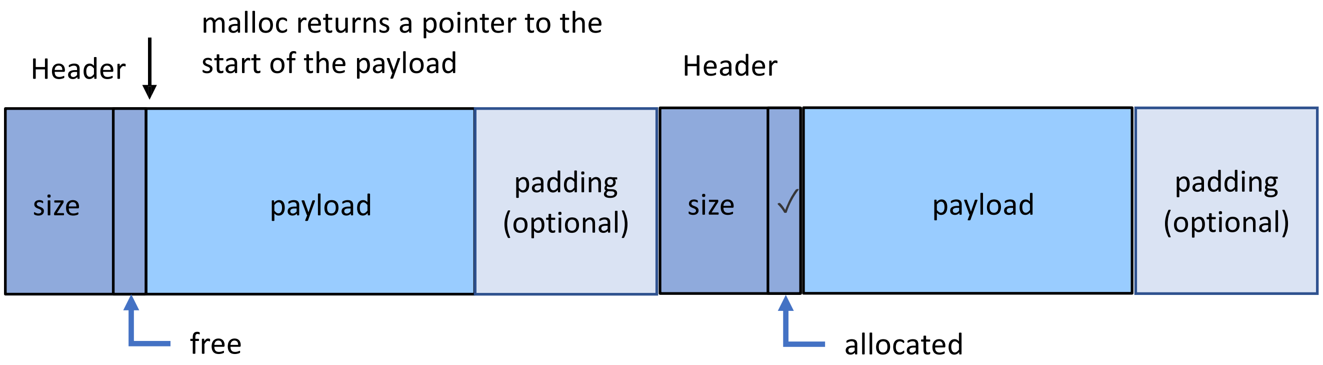
Managing the heap segment

- The benefit of the K&R allocator is its simplicity
    - Simple header with pointer and size in each free block
    - Simple circularly linked list of free blocks
    - Relatively small amount of code (~25 lines each)
- Among the limitations of K&R allocator are
    - The malloc function requires scanning the free list to find first free block that is big enough
    - The free function requires scanning the free list to find where to insert block being freed by ascending address
    - The free function cannot accept a pointer interior to the freed block.
    - Does not determine whether freed block was allocated or has already been freed.

Managing Memory

Managing the heap segment

- Another way to manage memory is using an *implicit block list* with a header containing the block size and a flag indicating whether the block is allocated.
- Adding the block size to the block address gives the next block. Both free and allocated blocks are in the list.



Managing Memory

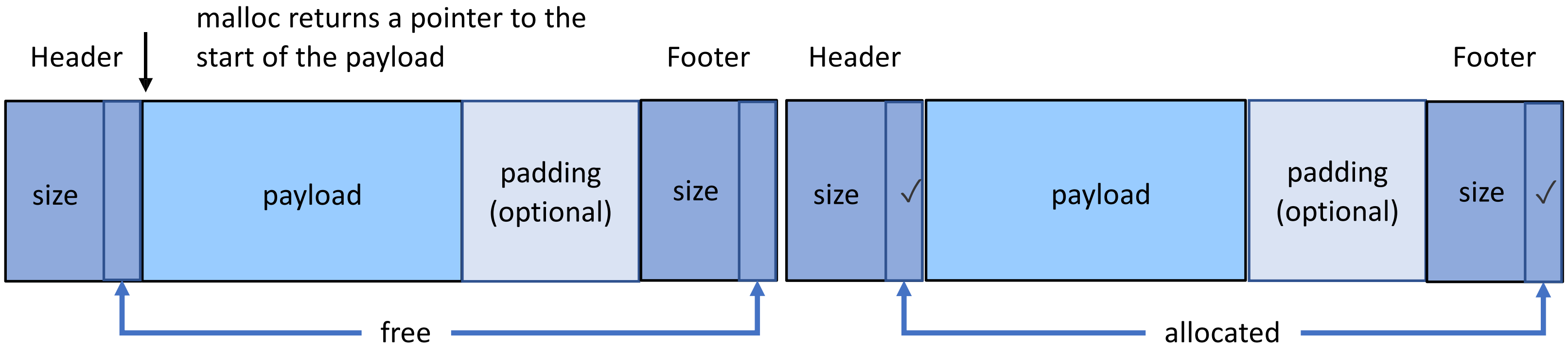
Managing the heap segment

- An implicit block list solves several of the problems with an explicit block list:
  - The free function does not require scanning the free list. Instead, the allocated flag is simply set to false.
  - It is now possible to determine whether the block being freed is already on the free list by examining its allocated flag.
  - It is easy to locate its successor block in order to coalesce the newly freed block with it; just skip forward by the recorded block size.
- However several problems remain:
  - How to locate the preceding block in order to coalesce the newly freed block with it
  - How to free a block given a pointer to its interior.

Managing Memory

Managing the heap segment

- Finding the preceding block can be done by adding a footer to each block that is a duplicate of the header. The implicit list is now doubly-linked.
- The footer of the preceding block is adjacent to the header of the next one, and the head of the preceding block can be located using the size in the footer block.



Managing Memory

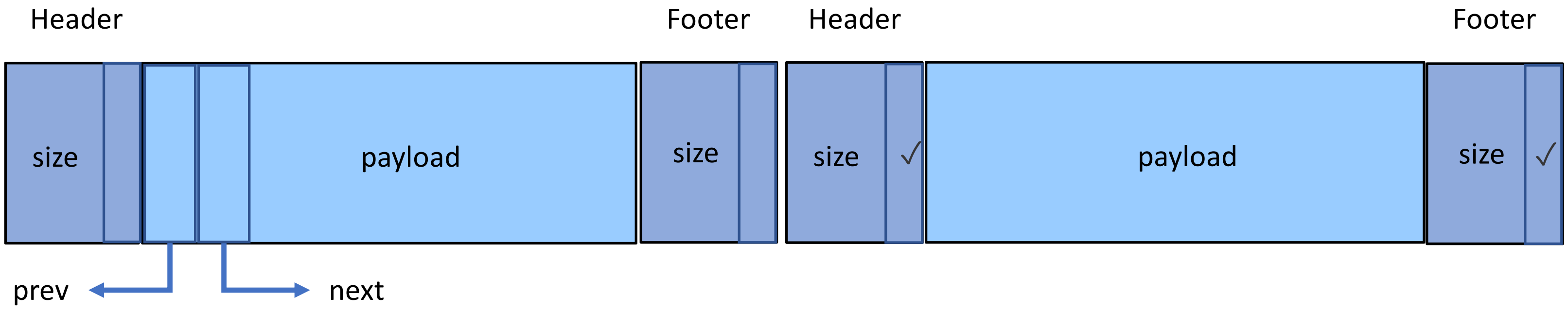
Managing the heap segment

- While an implicit list solves many of the problems mentioned earlier, there is still the issue of placement: which block to allocate. Blocks are still stored in order of increasing memory.
- One way to address this is to use space in the payload of free blocks to store pointers for linked lists used to organize the free blocks in other ways. Examples include
  - Organized by increasing size (e.g. sorted list or heap)
  - Separate free and allocated lists (linked lists)
  - Organized by most recently used (similarly sized blocks cluster)
  - Pools based on block sizes (e.g. small, medium, large block pools)

Managing Memory

Managing the heap segment

- In this example, space in the payload is used to store pointers to the next and previous free blocks in the a free list.
- The payload must be allocated to ensure there is room for these pointers.



Managing Memory

Managing the heap segment

- One approach is to define an allocation unit that can used as the block header, and can also be used as an allocation unit for the next/previous pointers within the free blocks.
- When used as the header, the allocation unit records the number of header-size allocation units in the memory block, and whether the block is allocated or free.
- When used to record the free block link information, the allocation unit contains a pointer to the the next or previous block in the free list

Managing Memory

Managing the heap segment

- Here is one possible implementation using a union of the size/allocation and the next/prev pointer. The size/allocation can use bit fields to save space and simplify addressing.

```
// Allocation unit for header/footer of allocated blocks
// and for free list pointers within free block payload
typedef union Header {
    struct {
        size_t blksize: (8*sizeof(size_t)-1);   // size of this block including header+footer
                                                // measured in multiples of header size;
        size_t isalloc: 1;                      // 1 if block allocated, 0 if free
    } s;
    union Header *blkp;                         // pointer to adjacent block on free list
    max_align_t _align;                         // force alignment to max align boundary
} Header;
```