EECS-343 Operating Systems
Lecture 8:
Free-Space Management

Steve Tarzia

Spring 2019

Northwestern

Announcements

- Project 2 due Monday
- HW2 is out and due on Wednesday.
- Midterm in one week (Thursday)
- Tuesday will be a midterm review
    - A Piazza post will ask for topics to cover.

Last Lecture: Swapping

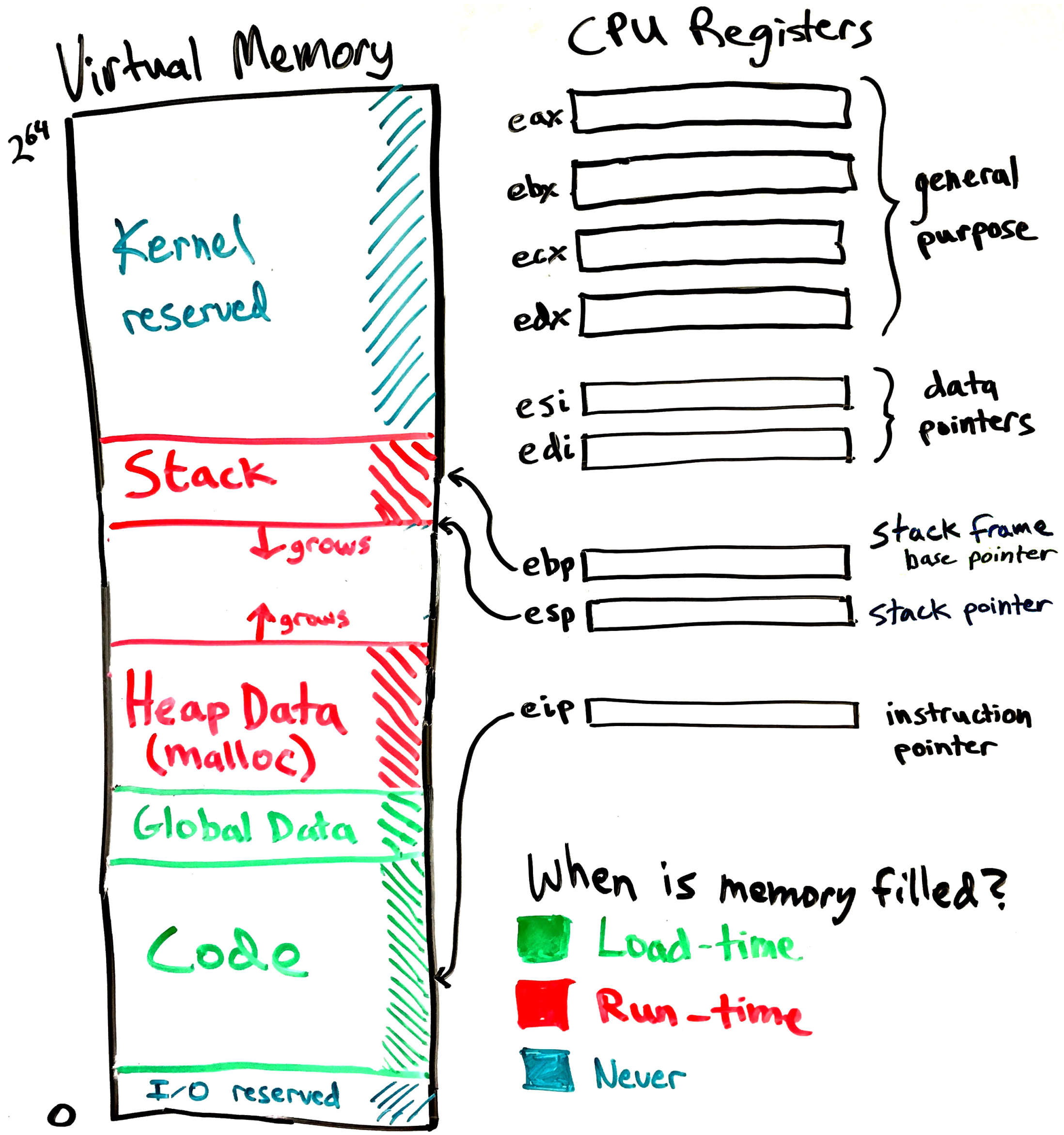
- Disk is slow, but large, and can be used to store RAM's overflow
  - Disks have high ***throughput*** (transfer bitrate) but high ***latency*** (delay)
  - Magnetic disks have even higher latency than SSDs, due to moving parts.
- Paging and swapping work together, using the same CPU mechanisms
  - If a page is marked "not present" it may be either invalid or swapped to disk.
    - Or it might indicate *lazy allocation*, *lazy loading*, or *copy-on-write*, as we saw last time.
  - High bits of page table entry can store disk location of swapped page.
- ***Page replacement policy*** decides which page(s) to ***evict*** to free memory
  - Swapping can be done ***on demand*** or in the ***background***
  - Having some free physical frames will prevent delays for allocations.
  - ***Accessed bit*** and ***Dirty bit*** in PTEs inform the page replacement policy
- ***Thrashing*** is when swapping prevents the system from doing any work.
- ***Unified page cache*** handles both traditional paging and ***file caching***.
  - Makes filesystem access seem just as fast as memory access.

Paging overview

- **Virtual memory** addresses are translated to physical memory addresses by the CPU, and the translation is dynamically configured by the OS in each process' page table.
- **Swapping** is the movement of pages between disk and physical mem.
- Page tables also allow several memory management optimizations:
  - **Copy-on-write fork** – delays memory copies
  - **Shared libraries** – read/execute-only code can be shared by several processes
  - **Lazy allocation/demand zeroing** – wait before allocating user memory.
- **Filesystem caching** allows page-sized portions of files to be stored in physical memory.

Memory management illustrations

The process' view of memory



- Code and global data are filled by ***exec*** syscall to load a program.
- A new ***frame*** is pushed on the stack whenever a function is called. (And popped on return.)
- Heap data is managed by malloc

How does `malloc` work?

Free-space management

<u>Given</u>: A single block of contiguous memory

<u>Goals</u>:

- Handle ***malloc*** and ***free*** requests
  - `void* malloc(int n)`: find a unused block of size n
  - `free(void* ptr)`: reclaim a block that was previously malloc-ed
- Minimize the total extent of memory required (be compact)
- Minimize the time required to malloc and free

Free-space management will also be a topic in filesystems.

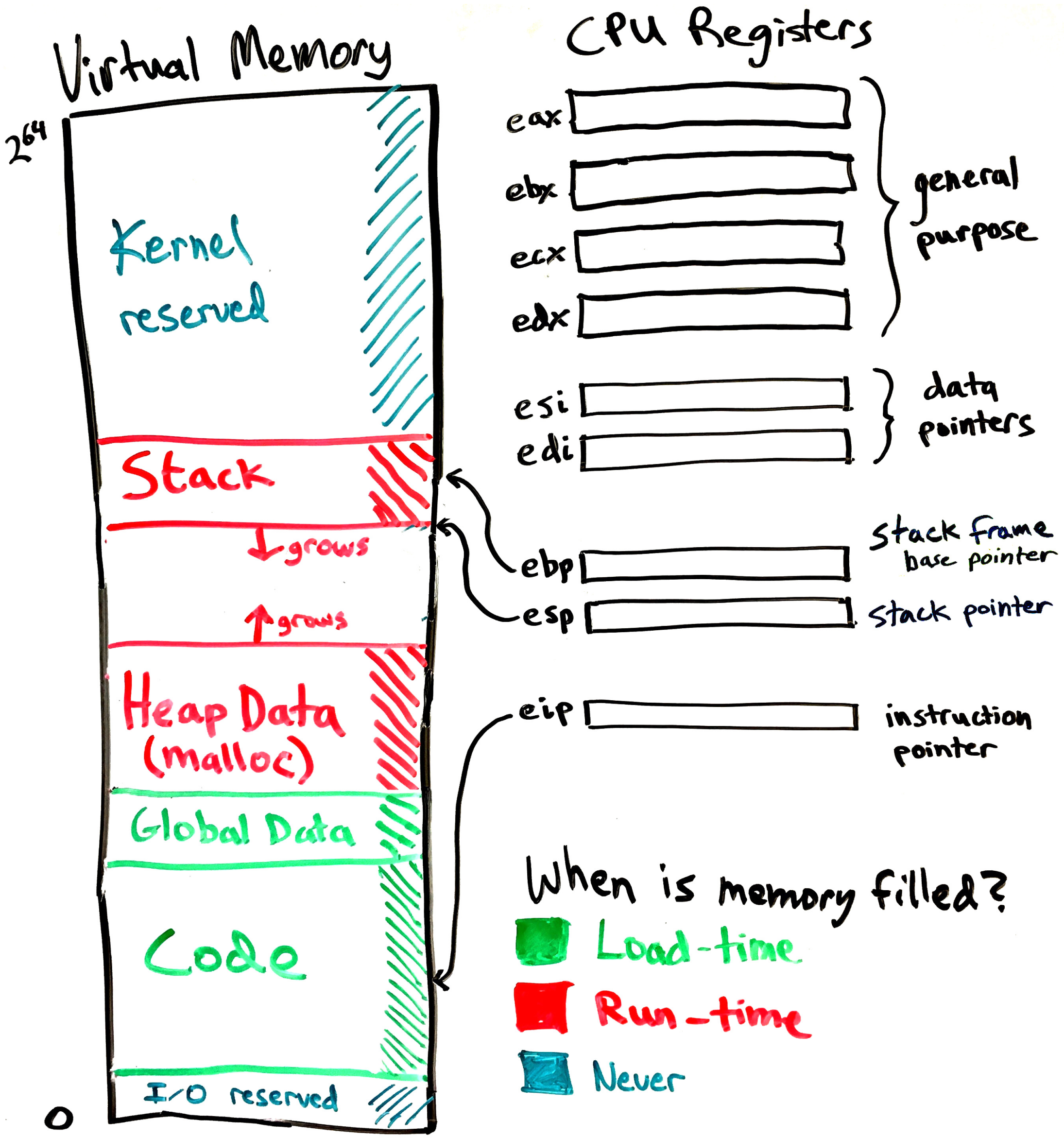
Heap dynamics

- Ideally, malloc would waste no space:

- Actually, Stack memory looks nice and compact like above.
- But malloc is used for ***dynamic*** memory
  - it will be freed later, at some unknown time
- Frees create **vacancies** in the Heap that waste space, but can be re-allocated:

- Remember that user programs expect malloc to return a ***contiguous*** block of (virtual) memory.

Grow the Heap only as a last resort

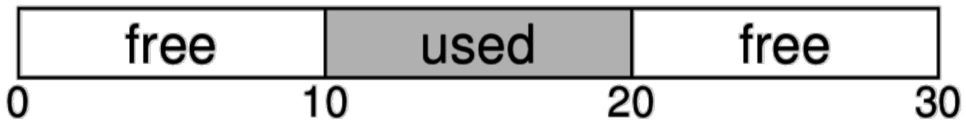


- Goal is to make maximum use of the Heap range already in use.
- Look for a free block >= the current malloc request
- If none is found, then we have to expand the heap.
  - On xv6, malloc uses ***sbrk*** syscall to tell the OS that the process's address range has increased and page table must be expanded.
  - Modern OSes do not use sbrk, but ***mmap*** or perhaps nothing.
- ***Memory leaks*** cause heap to grow indefinitely (if you forget to ***free***)

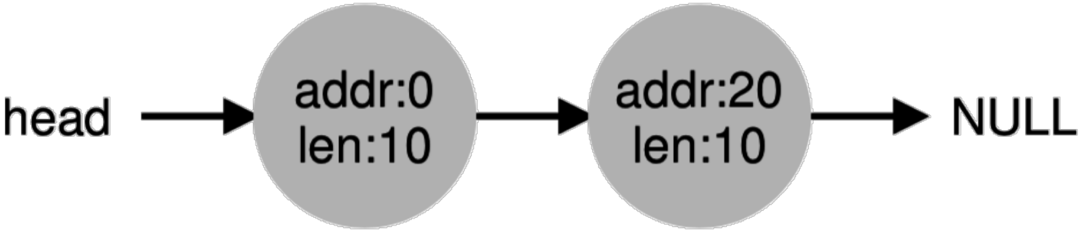
Malloc in user and kernel code

- Both the kernel and user code must dynamically allocate memory
- Free space management algorithms ***can*** be the same in both cases
- But different implementations are used. In xv6:
  - User level: `umalloc.c`: 90 lines of code
    - Basic C memory management: malloc & free
    - Missing: calloc & realloc (it's not ANSI C)
  - Kernel level: `kalloc.c`: 72 lines of code
    - xv6 lacks dynamic memory allocation in the kernel
    - No malloc & free, just allocate and free a **full page** of memory (4kb block)
- Linux has kmalloc, kfree (like user-level malloc and free)

Free list is a linked list tracking free blocks



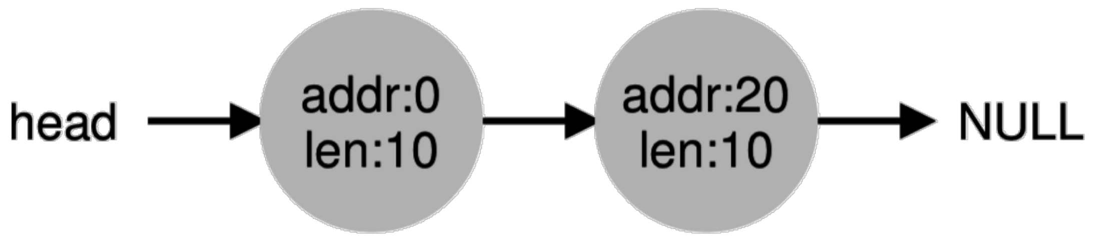
Above is represented as:



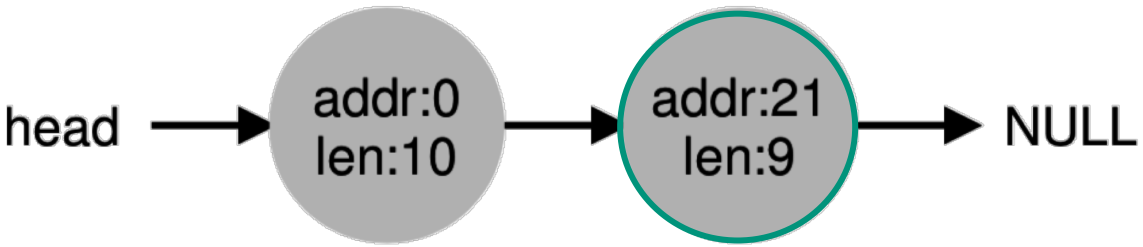
- Ordering is arbitrary
- This example shows two free blocks of size 10.
  - Space between two free blocks can be used by multiple mallocs
  - We'll track *used* blocks another way
- Like any linked list:
  - It's easy to insert and delete nodes
  - Finding the n^{th} node is slow
    - Start at the head node
    - Traverse *n* pointers
  - Lacks any *indexing*
    - Must examine every node when searching

Allocate memory by *splitting* free blocks

Before:



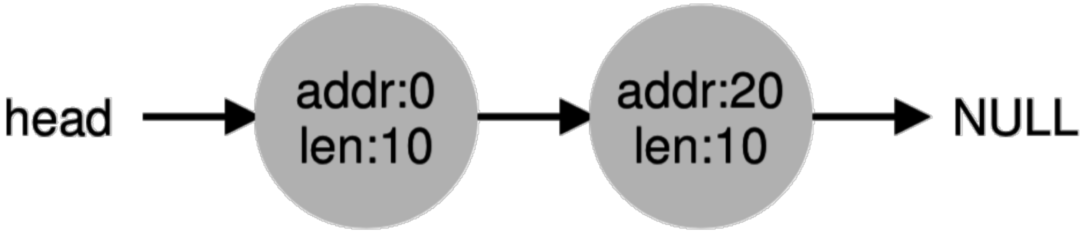
After allocating one byte from the second free block:



We just found a free block large enough for the allocation and claimed a chunk of it. (The ***policy*** decides which block to choose.)

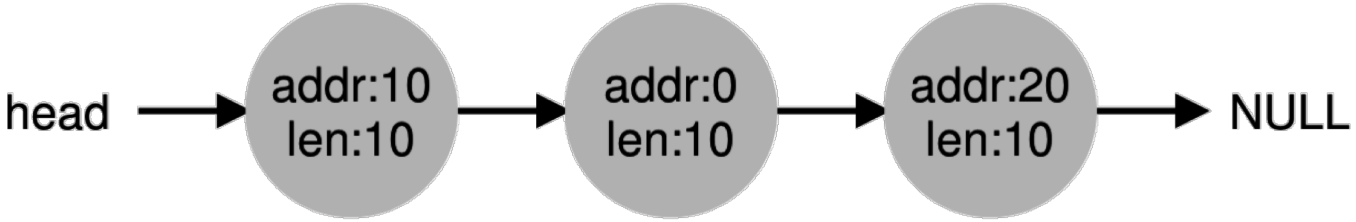
Coalescing eliminates artificial fragmentation

Before:



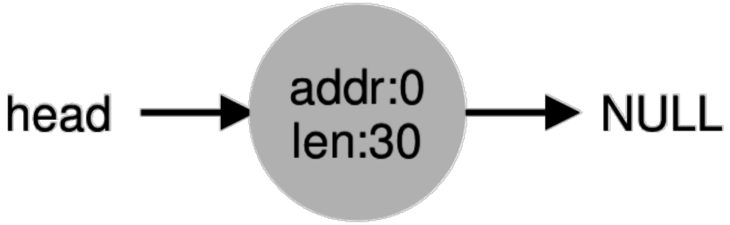
After freeing 10 bytes of data at address 10

(We usually add new nodes to a linked list's head):



Splitting this big block of free space among three small nodes makes it difficult to recognize it as a large contiguous free block

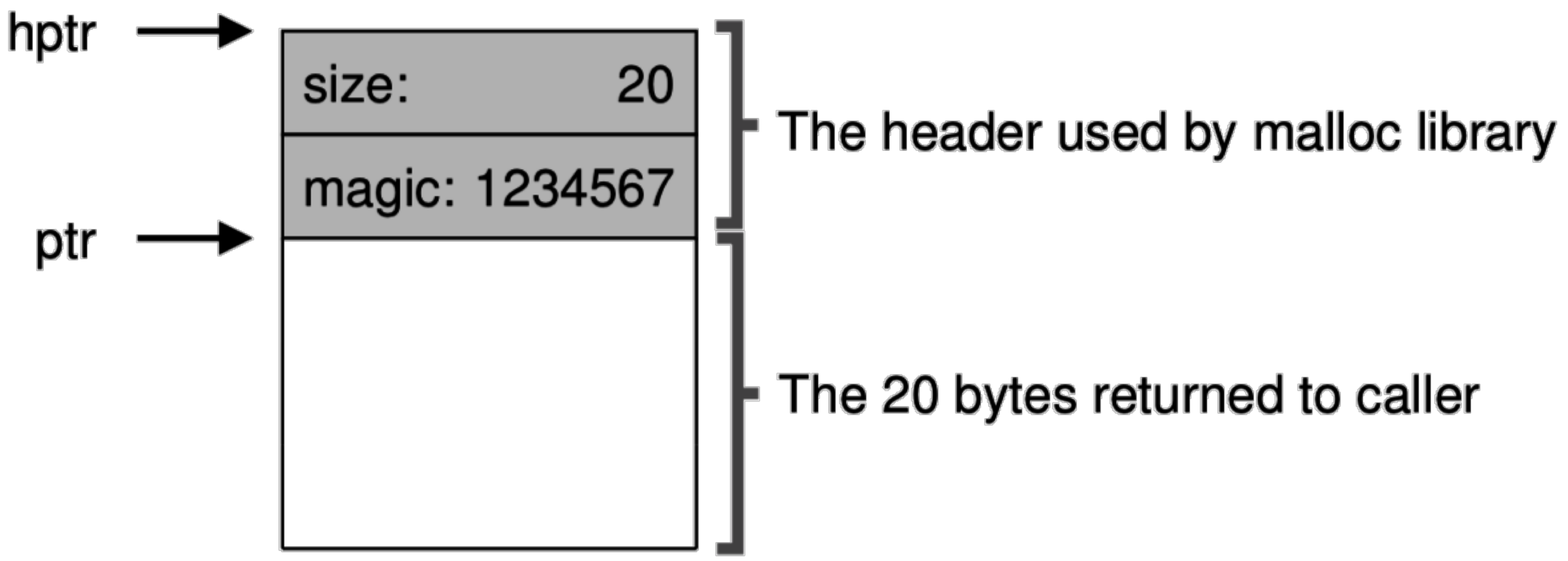
After coalescing:



A trick to help with ***frees***

- `free(ptr)` doesn't tell us how long the block is, just where it starts
  - But we need that information to free the block.
- Solution: cleverly prefix the block with a header:

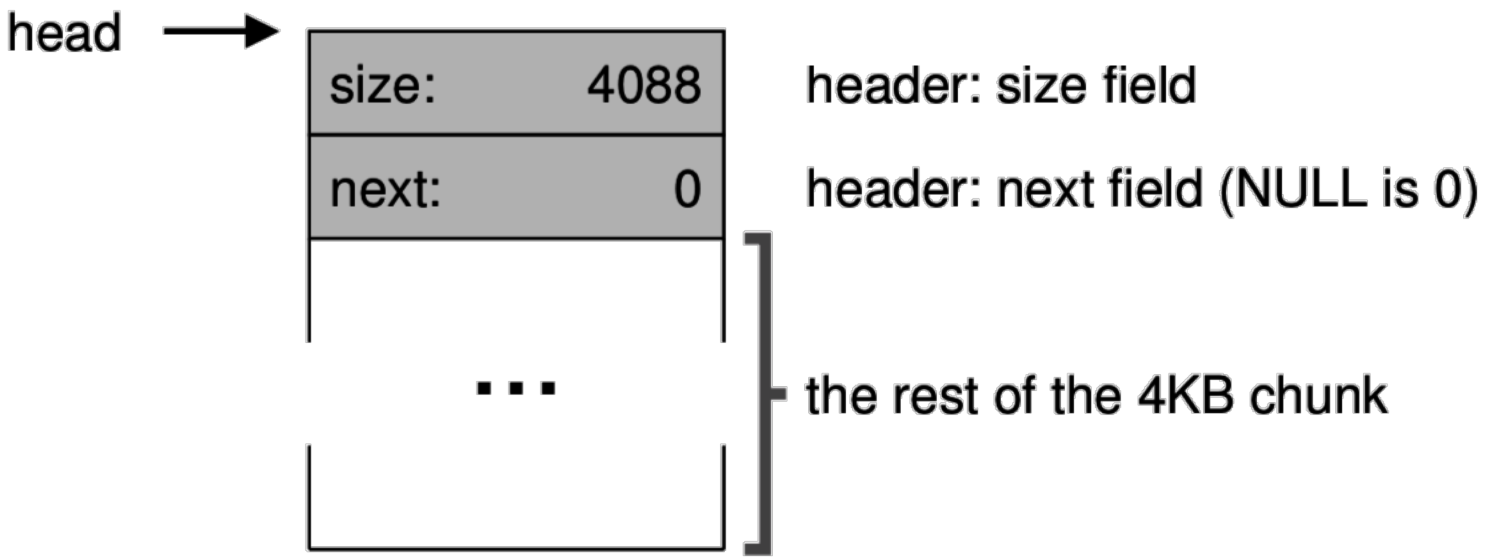
  Eg., malloc(20):



To handle `free(ptr)` just look at `(ptr - (sizeof hptr))` to find the block size.

Heap example: step 1: initialization

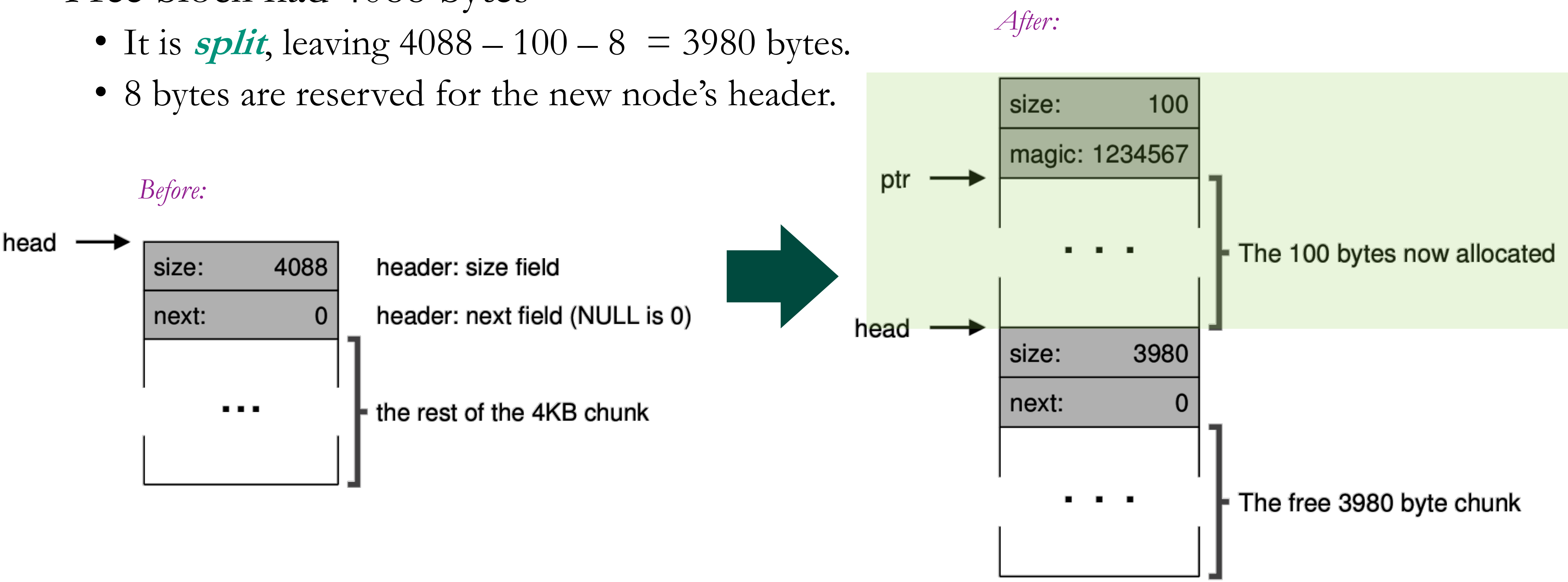
- Start with an empty 4kb block of memory (4096 bytes)
- Part of the Heap is always used to store the ***free list***.
- We just have a free list with one node, representing one big free block:



- Assuming pointers are 32 bits = 4 bytes, the free list node occupies 8 bytes, leaving 4088 bytes free below it.

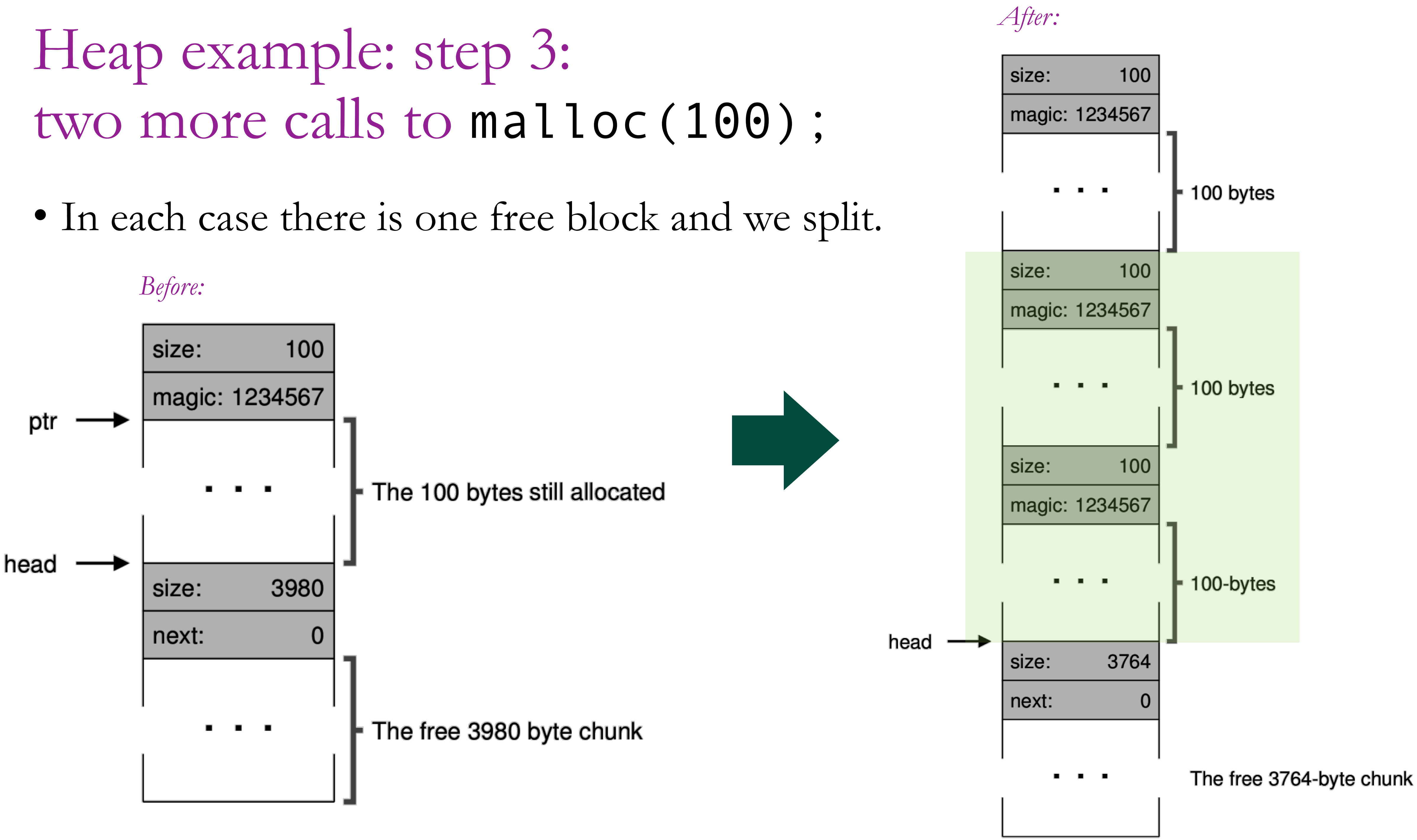
Heap example: step 2: `malloc(100);`

- The malloc implementation will traverse the free list (starting at "head") and find the one node that has >= 100 bytes.
- Free block had 4088 bytes
  - It is ***split***, leaving 4088 – 100 – 8 = 3980 bytes.
  - 8 bytes are reserved for the new node's header.

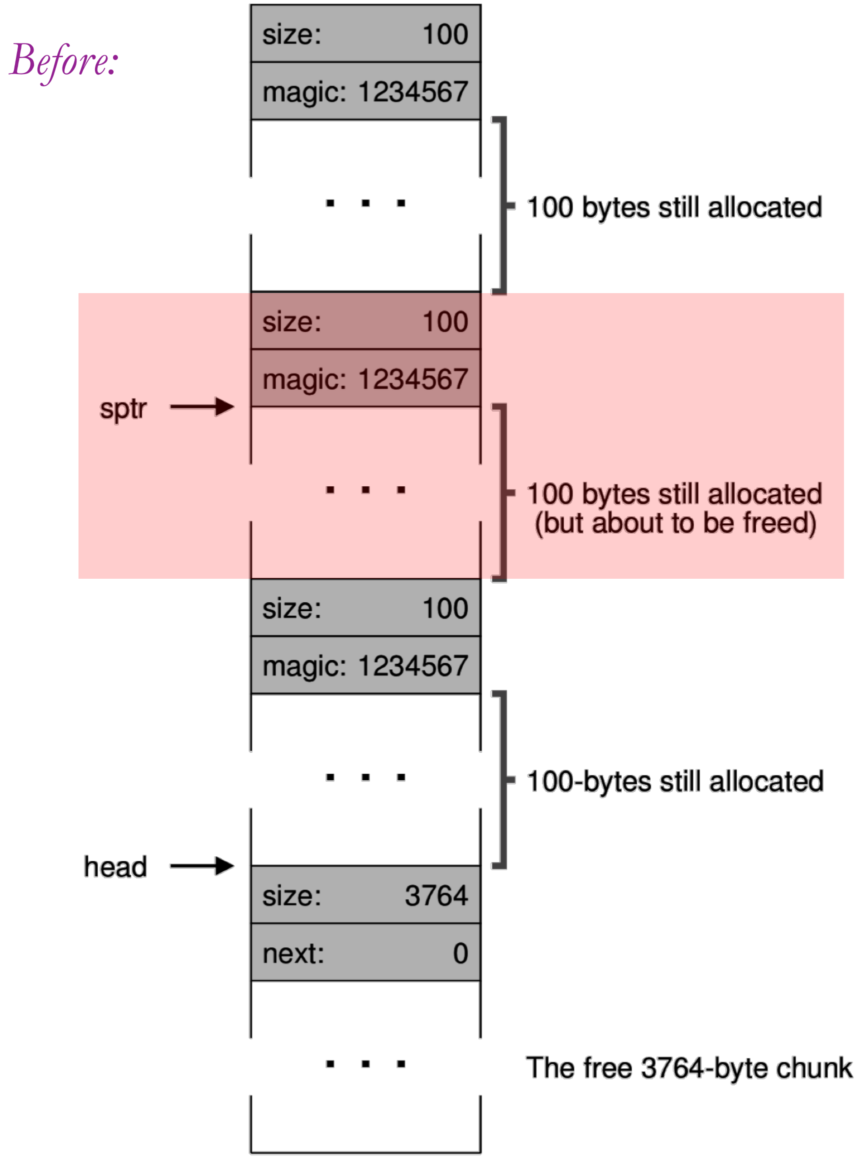


Heap example: step 3: two more calls to `malloc(100);`

- In each case there is one free block and we split.

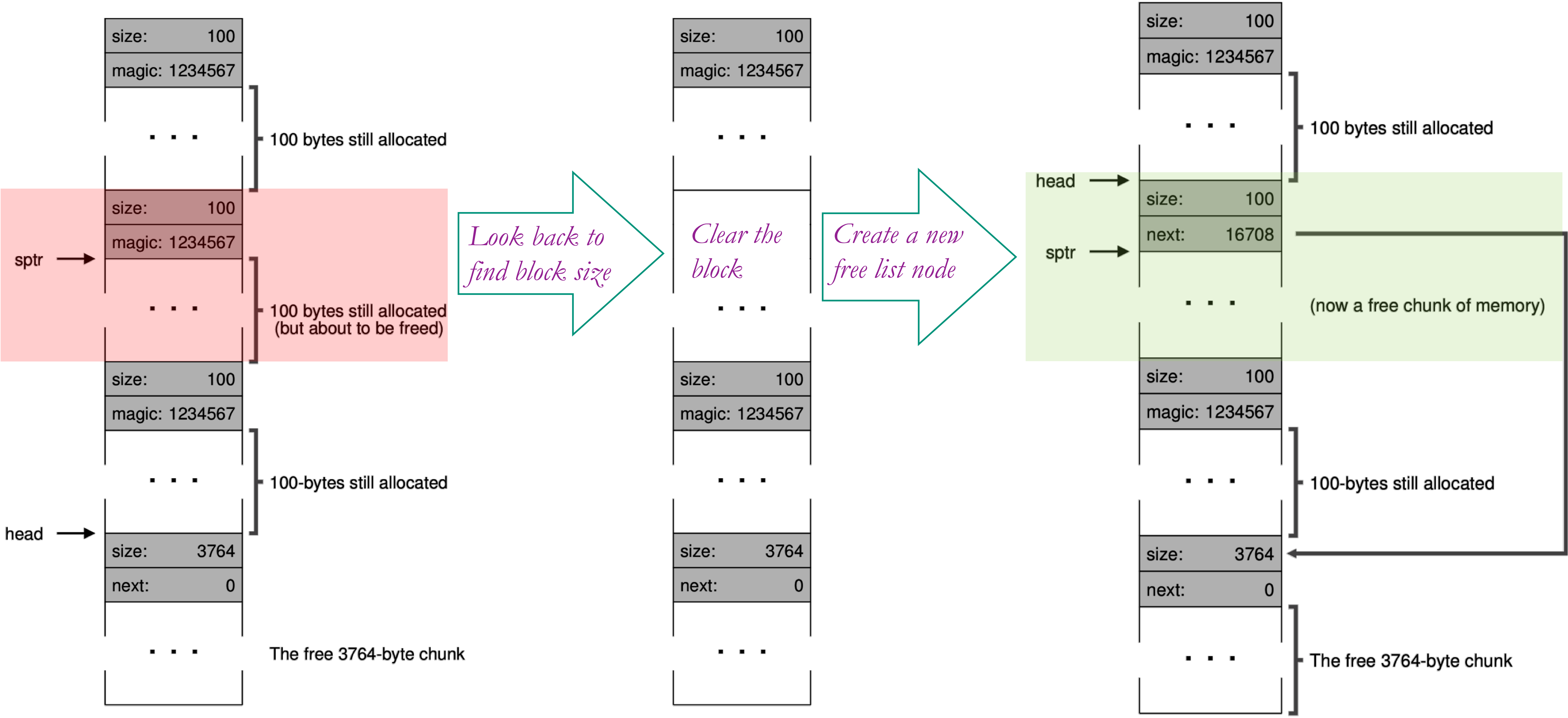


Heap example: step 4: `free(sptr)`



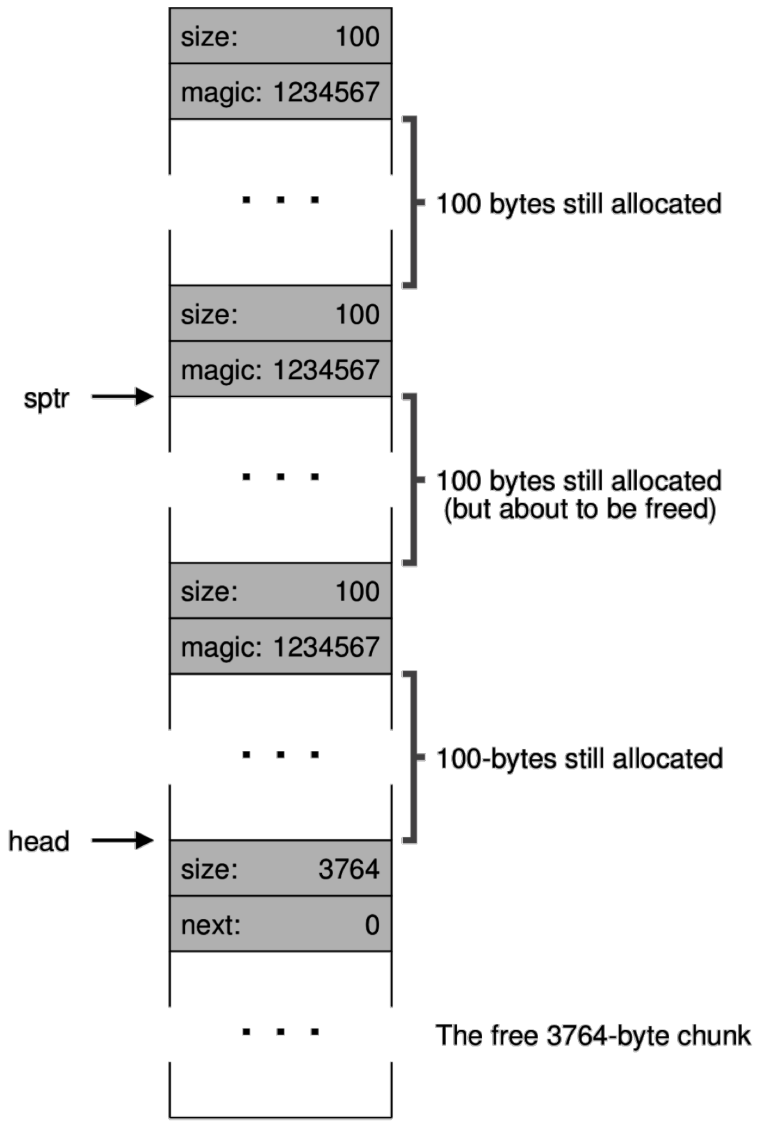
- When handling the "free," all we have is ***sptr*** (the *free* parameter) and ***head*** (a global variable).
- The caller didn't tell us how large a block to free, but we look back from *sptr* to find that size=100.
- To free the block, we just convert the malloc header into a free list node by changing the magic number into a pointer to the former head of the free list.

Heap example: step 4: `free(sptr)` *…in slow motion*



Heap example: step 4: `free(sptr)` …*the net change*

Before:

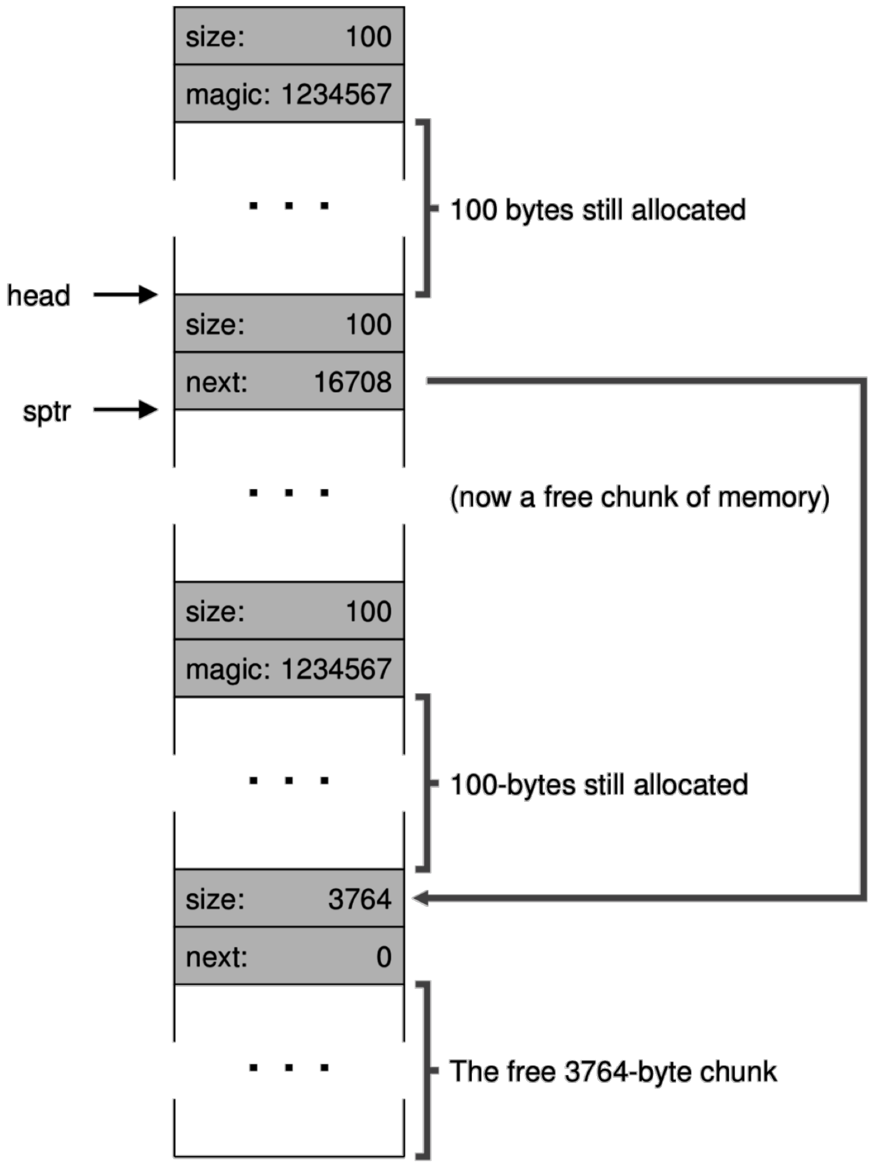


Actually, just two small things changed!

- `sptr - 4 = head`
- `head = sptr - 8`

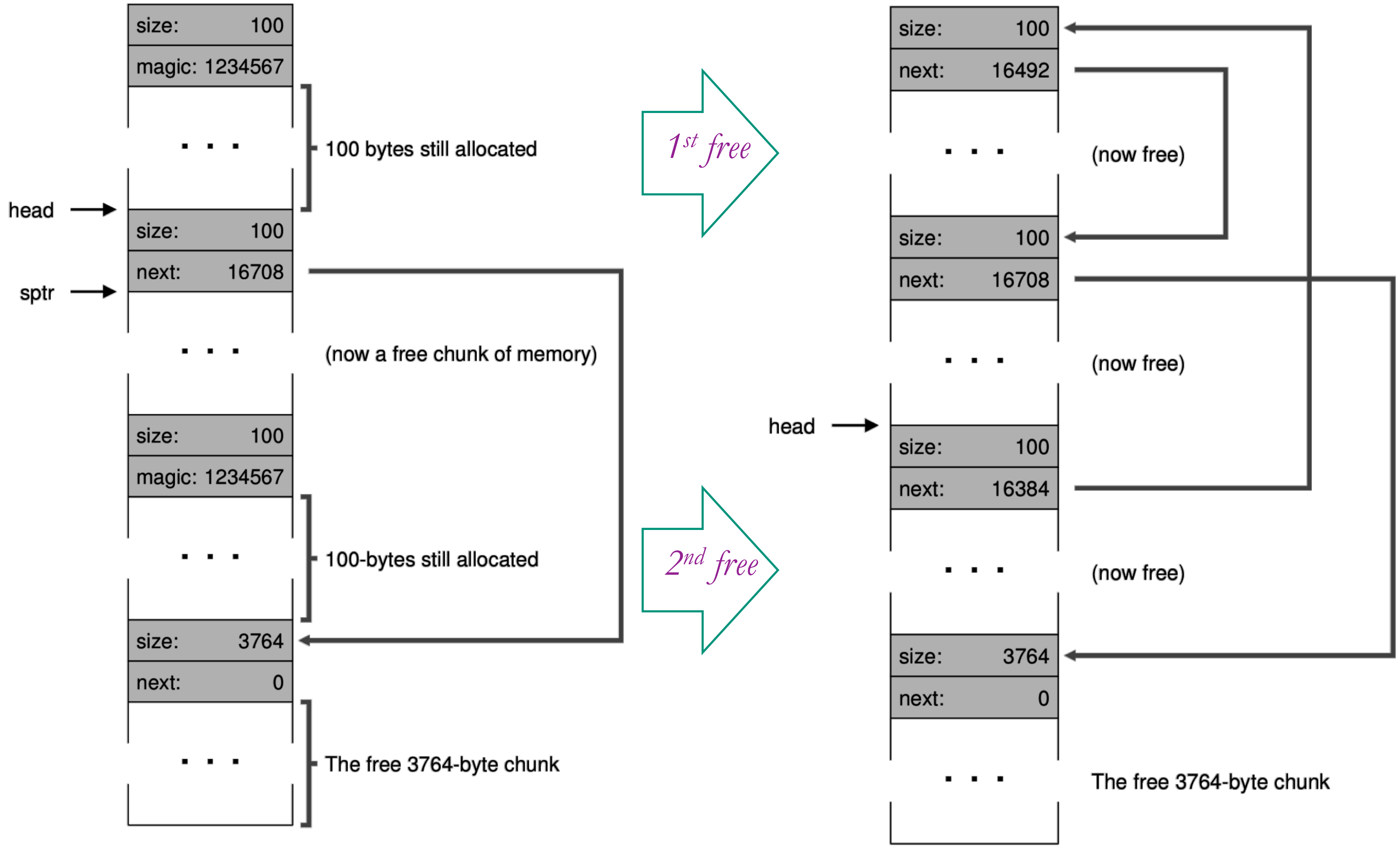
Our work is easy because malloc block headers are very similar to free list nodes

After:



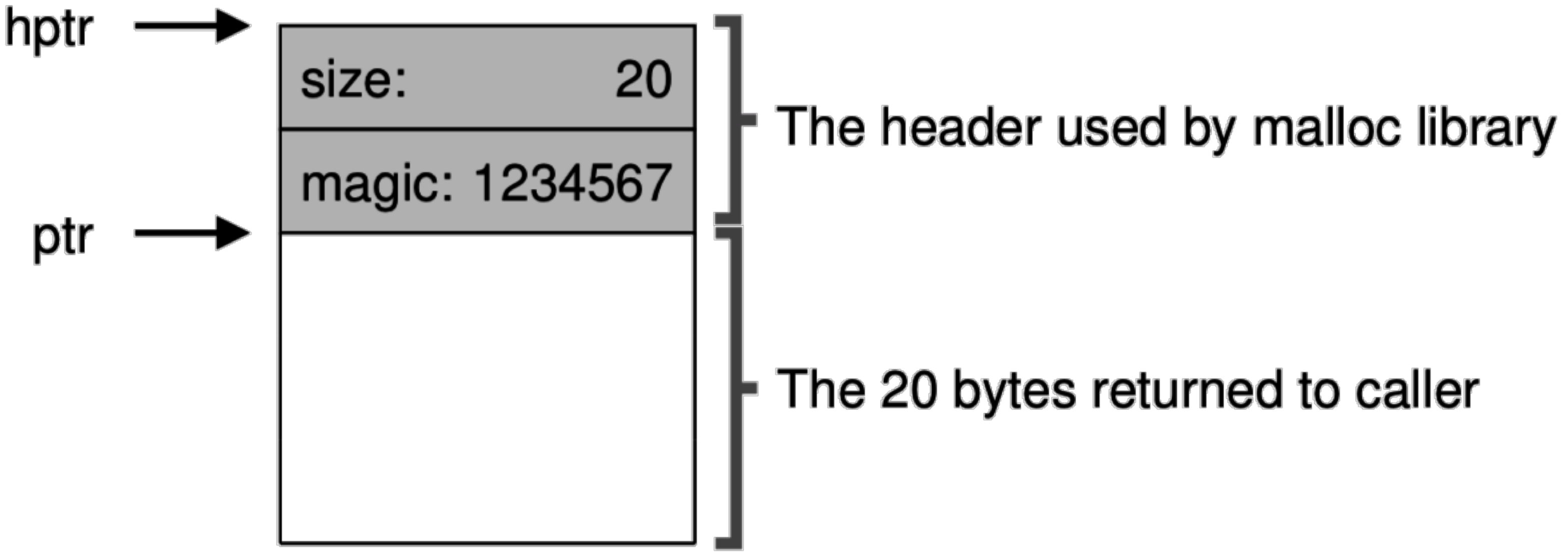
Heap example: step 5: `free` everything else

- This leaves a free list with four chunks.
- Notice that the free list is out of order.
- And it badly needs to be ***coalesced***.



What's up with the magic number?

- It's just an unusual, large numeric constant. *Always the same number.*
- It allows `free` to detect whether the pointer it received is valid.
- If there is no magic number behind the pointer, then
  - Free should abort and warn the user.
  - Maybe the code already called free? In other words, a "double free" error.



Intermission

Question: What could happen if you ignored the magic number and allowed a double free to proceed?

Linked lists are just one way to track free space

- There are many alternatives, especially if you want:
  - Quickly find a block of a given size
  - Quickly find neighboring blocks for coalescing
- One alternative is a ***bitmap***:
  - Divide the memory into fixed-size chunks and use a bit to indicate whether the chunk has been allocated.
  - Memory allocations would be rounded up to a multiple of the chunk size.
  - Eg., "1110000000001000000"

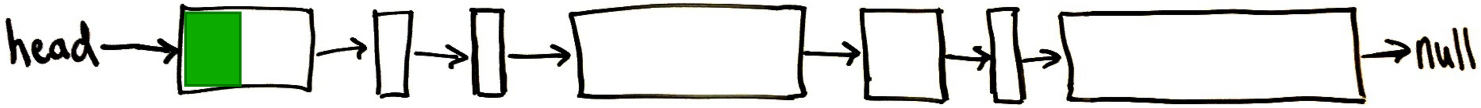
    a used block   a large free block
- But lists are a very common choice

Free space management policies & optimizations

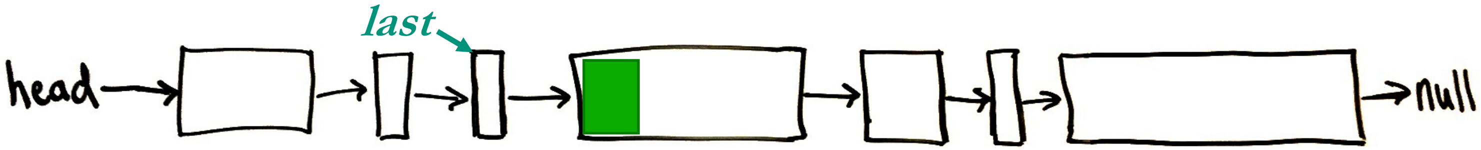
- We have seen a basic malloc/free mechanism
  - Glossed over some details of maintaining the linked list, but that's trivial.
- There are still some policy decisions to make:
  - **Which of the free blocks do we choose for a given allocation?**
  - **When do we coalesce?**
- In other words, how do we avoid memory fragmentation?

Choosing a free block to serve an allocation:

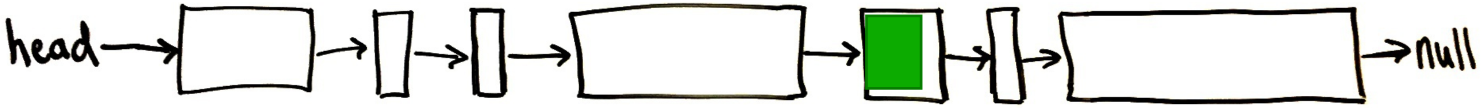
- **First fit** – simple and fast:



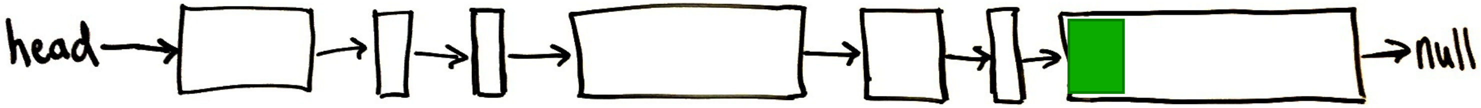
- **Next fit** – start looking where you left off:



- **Best fit** – try to leave the smallest remainder
  - But have to search the whole list and leaves small holes (hard to reuse)

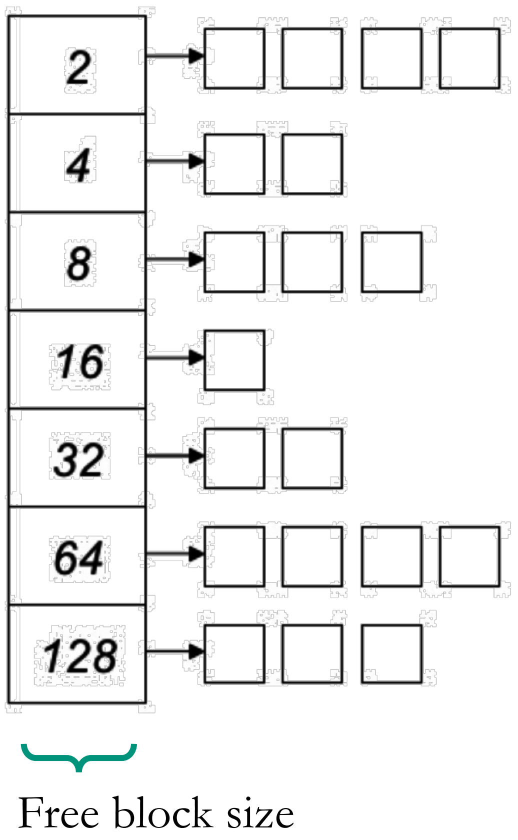


- **Worst fit** – try to leave large remainders that are easy to reuse



Segregated lists (Slab memory allocator)

- Instead of keeping one list of free blocks, we can keep different lists for small, medium, and large blocks.
- This will allow us to find the right sized block more efficiently.
- Just keep an array of free lists (many *heads*)
- On free (or if splitting), add free block to the appropriate list
- Many variations are possible, but this design often uses fixed-length chunks (powers of 2).



Warning: bad style

- xv6 codebase suffers from some code style problems.
  - Lacks comments
  - Variable names are too short: np, bp, hp, s, b, *etc*. Why so short!?
  - if/else should *always* use {}
  - goto
- Most of these are due to obsolete habits. Computer screens used to be small, so shorter code was favored.
- Don’t learn these bad habits!

Malloc in xv6

- Taken from Section 8.7 of Kernighan and Ritchie "C book"
- Free list is circular (tail points back to head instead of to null)
- Uses the "next fit" policy – head pointer changes each time
- Free list is ordered according to memory address
  - This enables easy coalescing
  - When freeing a block, don't just place it at the list head:
    - Scan for the correct location in the list for that address
    - Check whether neighbors are directly adjacent in memory (and **coalesce**)
- When free list cannot serve the request, use ***sbrk*** syscall to get a pointer to a block of new memory from the OS.

xv6/umalloc.c

```c
typedef long Align;  // for alignment to long boundary

union header {  // block header
  struct {
    union header *ptr; // next block, if on free list
    uint size;         // size of this block (in 64-bit units)
  } s;
  Align x;  // force alignment of blocks
};

typedef union header Header;

// global variables:
static Header base;   // the first free list node
static Header *freep; // start of the free list (head)
```

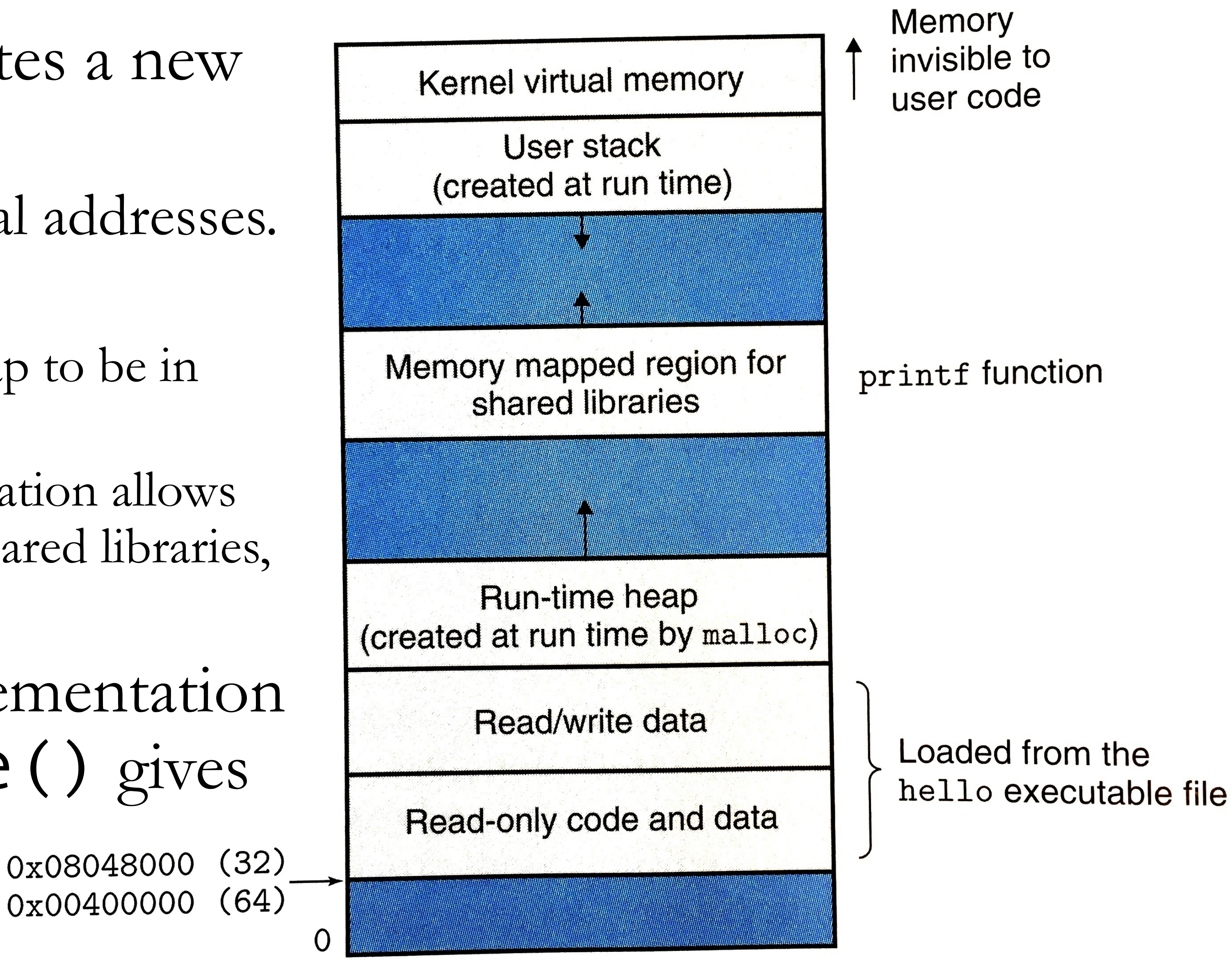
```c
// user program's general purpose storage allocator
void* malloc(uint nbytes) {
  Header *p, *prevp;

  // round up allocation size to fit memory alignment (long)
  uint nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
  // if there is no free list yet, set up a list with one empty block
  if((prevp = freep) == 0){
    base.s.ptr = freep = prevp = &base;
    base.s.size = 0;
  }
  // scan through the free list
  for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
    // if it's big enough
    if(p->s.size >= nunits){
      // if exactly the right size, remove from the list
      if(p->s.size == nunits){
        prevp->s.ptr = p->s.ptr;
      }
      // split the free block by allocating the tail end
      else {
        p->s.size -= nunits;  // make the free block smaller

        // Modify our copy of the free block's header "p"
        // to make it represent the newly allocated block.
        p += p->s.size;
        p->s.size = nunits;
      }
      freep = prevp;  // change the start of the free list
                      // to implement the "next fit" policy
      return (void*)(p + 1);  // allocated chunk, past the header
    }
    // if we looped around to list start again, no blocks are big enough
    if(p == freep) {
      // ask the OS for another chunk of free memory
      if((p = morecore(nunits)) == 0) {
        return 0;  // the memory allocation failed
      }
    }
  }
}
```

```
// minumum number of units to request
#define NALLOC 4096

// ask the OS for more memory
static Header* morecore(uint nu) {
  if(nu < NALLOC){  // never ask for just a tiny bit of memory
    nu = NALLOC;
  }
  // sbrk asks the OS to let us use more memory at the end of
  // the address space and returns a pointer to the beginning
  // of the new chunk
  char* p = sbrk(nu * sizeof(Header));
  // on failure, sbrk will return -1
  if(p == (char*)-1){
    return 0;
  }
  Header *hp = (Header*)p;  // cast the new memory as a Header*
  hp->s.size = nu;  // set up the new header
  free((void*)(hp + 1));  // add the new memory to the free list
  return freep;
}
```

```
// put new block "ap" on the free list because we're done using it
void free(void *ap) {
  Header *bp = (Header*)ap - 1;  // the block header

  // Scan through the free list looking for the right place to insert.
  // Stop when we find a block p that is before the new block,
  // but the new block is before p's "right neighbor"
  Header *p;
  for(p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr) {
    // There is a special case when the new block belongs at the start or end.
    // If the scan got to the block with the highest address,
    // and the new block is > the highest, or < the lowest
    if(p >= p->s.ptr && (bp > p || bp < p->s.ptr)) {
      break;  // block is at the start or end of the range
    }
  }
  // p will become the new block's "left neighbor" so insert after it,
  // but first check whether to coalesce.

  // if the end of the new block touches the right neighbor, coalesce-right
  if(bp + bp->s.size == p->s.ptr){
    bp->s.size += p->s.ptr->s.size;  // add the size of the right neighbor
    bp->s.ptr = p->s.ptr->s.ptr;     // point to the neighbor's neighbor
  }
  // if there is a gap to the right, just point to the right neighbor
  else bp->s.ptr = p->s.ptr;

  // if the end of left neighbor touches the new block's start, coalesce-left
  if(p + p->s.size == bp){
    p->s.size += bp->s.size;  // add the new block's size to the left neighbor
    p->s.ptr = bp->s.ptr;     // make the left neighbor point to the right neighbor
  }
  // if there is a gap to the left, the left neighbor points to the new block
  else p->s.ptr = bp;

  freep = p;  // change the start of the free list, for "next fit" policy
}
```

Malloc with mmap

- Linux's **mmap** syscall activates a new range of virtual addresses.
    - The kernel chooses the virtual addresses.
    - Why let kernel choose?
        - The simpler **sbrk** requires heap to be in contiguous memory.
        - Mmap with kernel-chosen location allows allocation to happen *around* shared libraries, stack, etc.
- Notice that the malloc implementation does not care if `morecore()` gives adjacent memory.



Recap

- Freed memory is put on a **free list** to be reused for later allocations.
- A single header can be cleverly used and re-used for two purposes:
  - As a linked list node when the block is free/available
  - To store the size of the allocated block to help service *free* calls.
- Free space management **policy** determines:
  - which free blocks to choose for an allocation, and
  - When to **coalesce** (join) adjacent free blocks
- Free block choice policies include:
  - **First**, **next**, **best**, and **worst** fit.