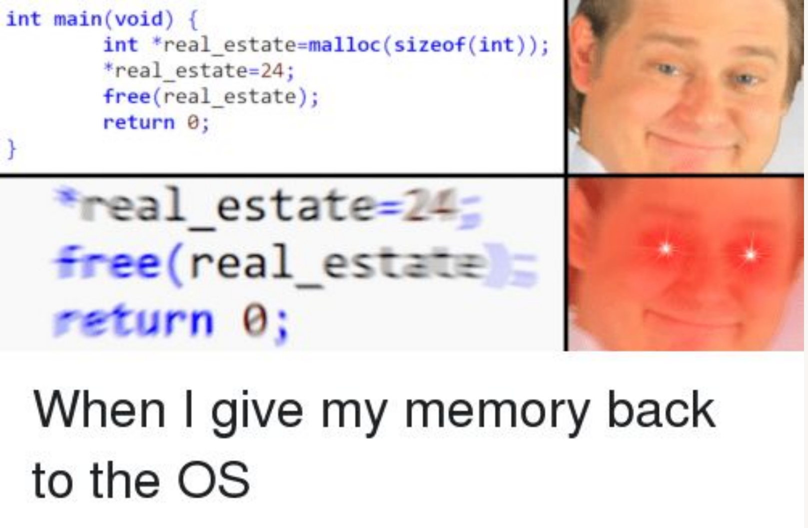


CSE 351 Section 9

Memory Allocation and Lab 5

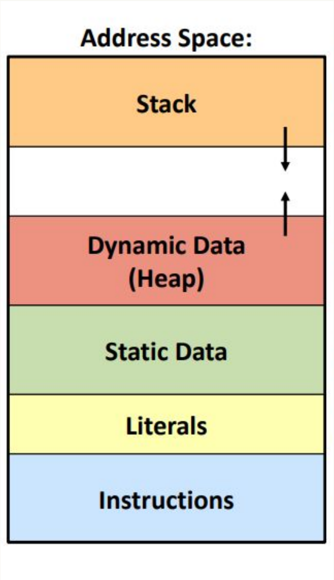
Administrivia

- **Homework 24**
  - Due Friday, 12/2
- **Homework 25**
  - Due Wednesday, 12/7
- **Lab 5**
  - Due Friday, 12/9 (only one late day allowed!)
- **Next week's section will be Final Exam review**

Dynamically Allocated Memory

The Heap

- Dynamic memory is memory that is “requested” at run-time. Dynamic data is stored in the *heap*.
    - Memory is allocated dynamically by the programmer (malloc)
    - Must be explicitly freed (free)
        - Free it as soon as you don't need it!
    - Distinct from normal variables, which are always on the stack



Use cases:

- Variable-length data, like arrays or strings (think: Java's ArrayList)
- Long-lived data passed between functions (think: Linked lists)

Why Dynamic Allocation?

Goal: Dynamically add/remove/sort nodes in a large linked list

Option 1: Without dynamically-allocated memory:

- Use the mmap() or equivalent system call to map a virtual address to a page of physical memory
  - This essentially gives you a page of memory to use
- Use pointer addition/subtraction to segment the page into linked list nodes
- Manage which regions of the page have been used
- Request a new page when that one fills up
- MESSY! NOBODY DOES THIS!

Why Dynamic Allocation?

Goal: Dynamically add/remove/sort nodes in a large linked list

Option 2: With dynamically-allocated memory:

- Use malloc() from the C standard library to request a node-sized chunk of memory for every node in the linked list
- When removing a node, simply carry out the necessary pointer manipulation and use free() to allow that space to be used for something else
- You will come to love malloc() because it does all the heap management for you…
- …But for the next week it may be more annoying because you are in charge of implementing it

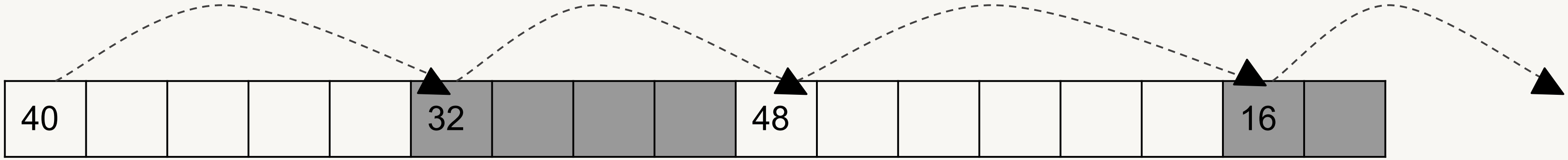
malloc() and free()

- Provided by the C standard library in `<stdlib.h>`
- How to use `malloc()`:
  - Takes a `size_t` representing the number of bytes requested
  - Returns a `void*` pointing to the start of the ***payload*** or `NULL` if there was an error
- How to use `free()`:
  - Takes a pointer to a block received from malloc() and deallocates its space on the heap
    - Be careful - don't free the same block twice!

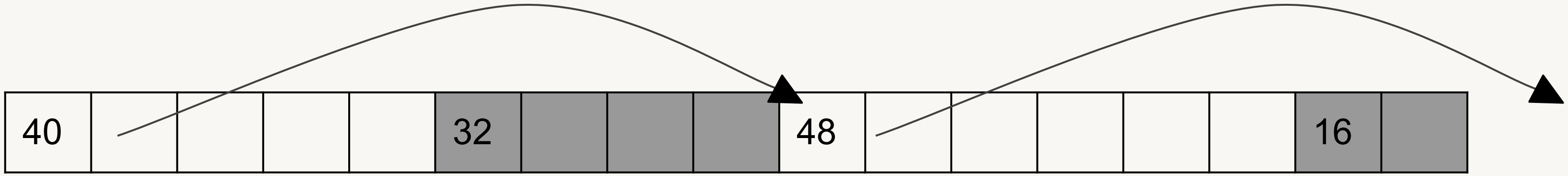
```
int* array = (int*) malloc(10 * sizeof(int))
...
free(array);
```

Implicit vs. Explicit Free List

Implicit: Using sizes to traverse blocks, checking to see if each block is allocated



Explicit: Using pointers to create linked list of free blocks (oft. doubly linked)



Comparison: free-lists

Implicit

- Find the next block via incrementing by the current block's size
- It may or may not be free
  - Potentially lots of extra blocks in the way!
- Requires only knowledge of each block's size

Explicit

- Find the next block by following a pointer
- All blocks in the free-list are guaranteed to be *free*
- Requires space in each free block to store pointers to the blocks before/after it

Reminder: Implicit/explicit free-lists are separate from implicit and explicit allocators.

For the remainder of this section, we'll be looking at explicit free-lists.

Block Header Format

- Every block has a 8-byte (64-bit) header, and needs to indicate its size, if it is used, and if the preceding block is used
- Size must be 8-aligned, so can use lowest 3 bits for tags
  - LSB is set if the block is currently used (not in the free list)
  - Next bit set if the block preceding it (in memory) is used
  - The third bit from the right is not used (for our current purposes)
  - Be careful with masking!
- The upper 61 bits store the size of the block
- Entire 64-bit value is a field called “size_and_tags” in Lab 5

```
+-----------------------------------------------+
| 63 | 62 | 61 | 60 |  . . .  | 3 | 2 | 1 | 0 |
+-----------------------------------------------+
```

Size

X

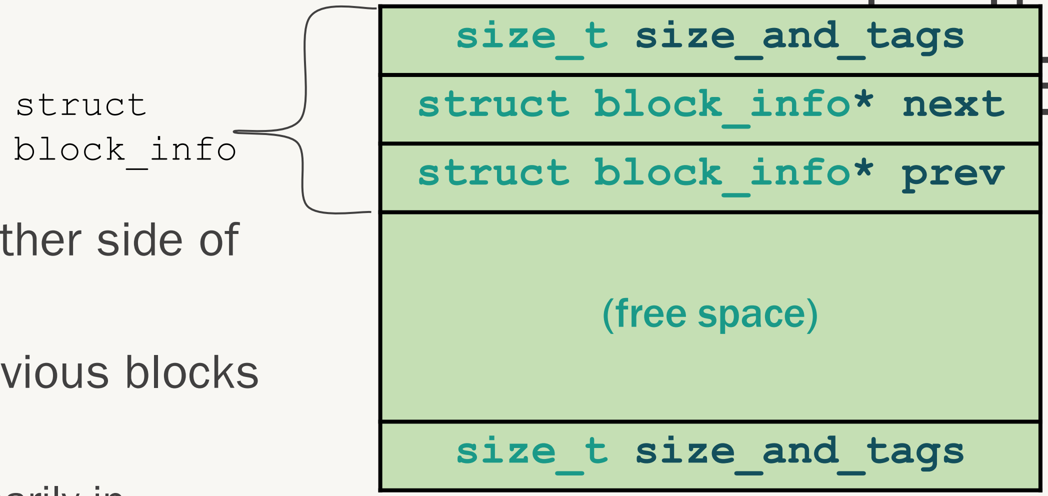
Prec. Used?

Used?

Free Blocks

A free block has:

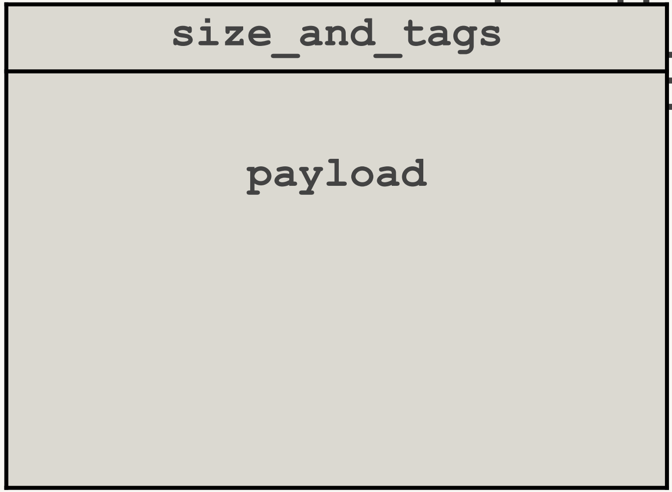
- A size_and_tags value on either side of the free space.
- Pointers to the next and previous blocks in the list.
    - The blocks are not necessarily in address order, so the pointers can point to blocks anywhere in the heap
- Each free block is a block_info struct followed by free space and the boundary tag (footer)




```
struct block_info {
  size_t size_and_tags;
  struct block_info* next;
  struct block_info* prev;
};
```

Used Blocks

- Used blocks only have a size_and_tags, followed by the payload
- In Lab 5, used blocks have no footer!
- The payload is the actual block of memory returned to a user program that invokes malloc()



Key Steps (Important!!)

- Allocation
  - Search for a free block of sufficient size
  - Remove selected block from free-list
  - If sufficient space for another block, split into two and add the smaller free block to free-list
  - Mark the allocated block as allocated
  - Return a pointer to the payload
- Deallocation (freeing)
  - Mark as free
  - Coalesce with adjacent blocks if possible
  - Add block to free-list
    - If using LIFO insertion policy, this free block becomes the new "root"

Walkthrough of Example Heap

Initial Heap

Note FREE_LIST_HEAD always points to the first block in the free list

FREE_LIST_HEAD

Walkthrough of Example Heap

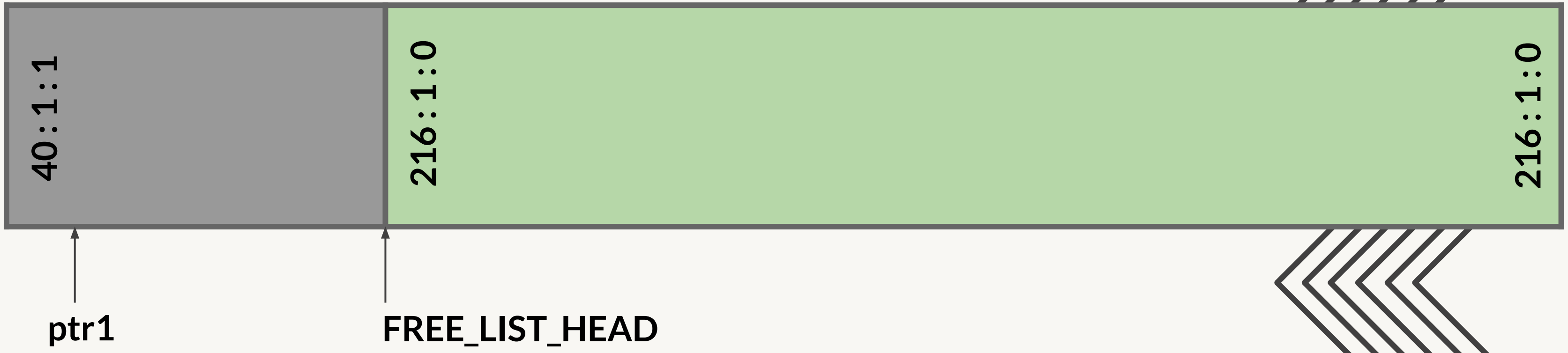
void* ptr1 = malloc(32);

- Need to search free list to find a block big enough for 40 (32 + header) bytes

Walkthrough of Example Heap

void* ptr1 = malloc(32);

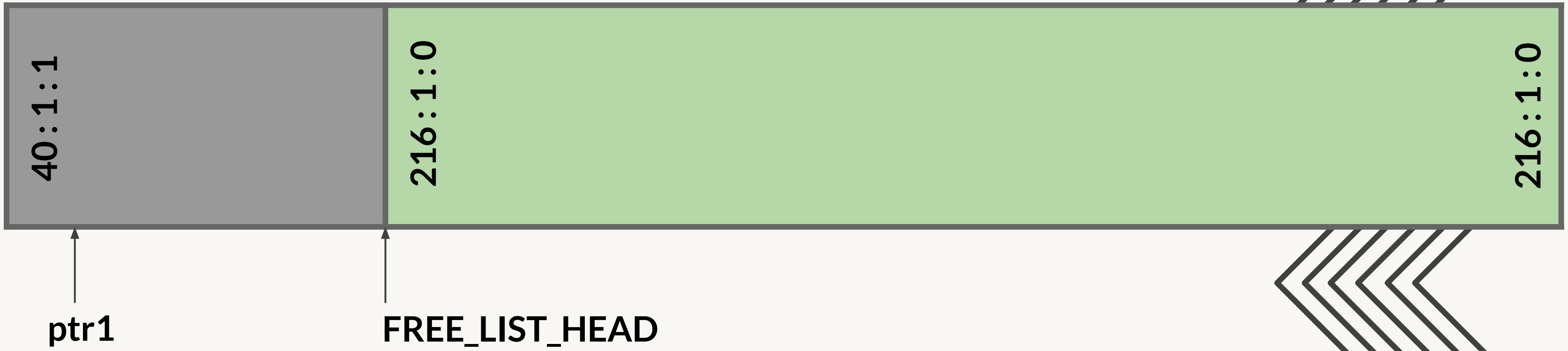
- Note that ptr1 points to the start of the payload, NOT the start of the block
- The initially 256 byte free block is split to maximize memory usage!



Walkthrough of Example Heap

void* ptr2 = malloc(16);

- Only need a block of 24 (16 + header) bytes, but what if we needed to free it later… think about what the minimum block size needs to be



Walkthrough of Example Heap

void* ptr2 = malloc(16);

- Need at least 32 bytes to create a free block, meaning we must allocate at least this much for a used block!

Walkthrough of Example Heap

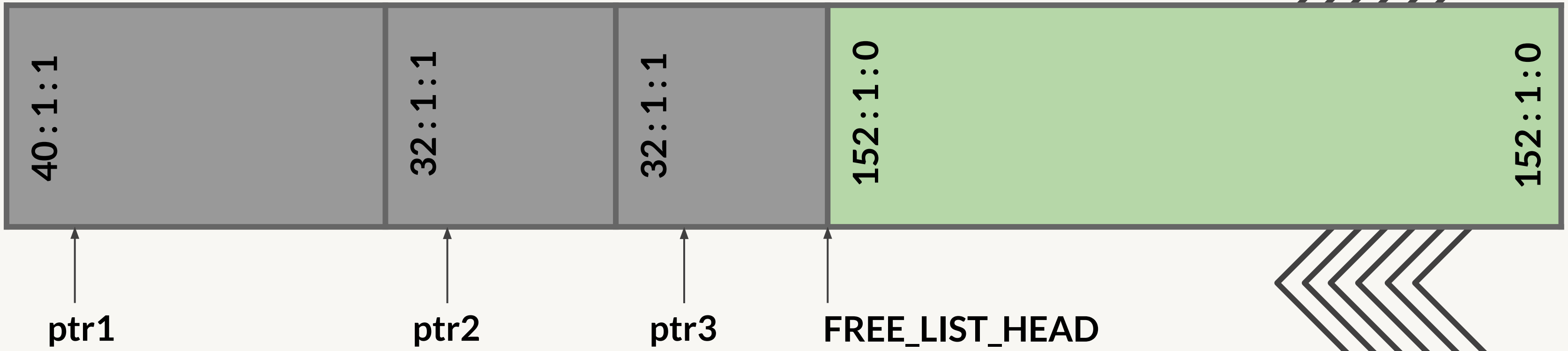
```
void* ptr3 = malloc(24);
```

- Same procedure as before

Walkthrough of Example Heap

void* ptr3 = malloc(24);

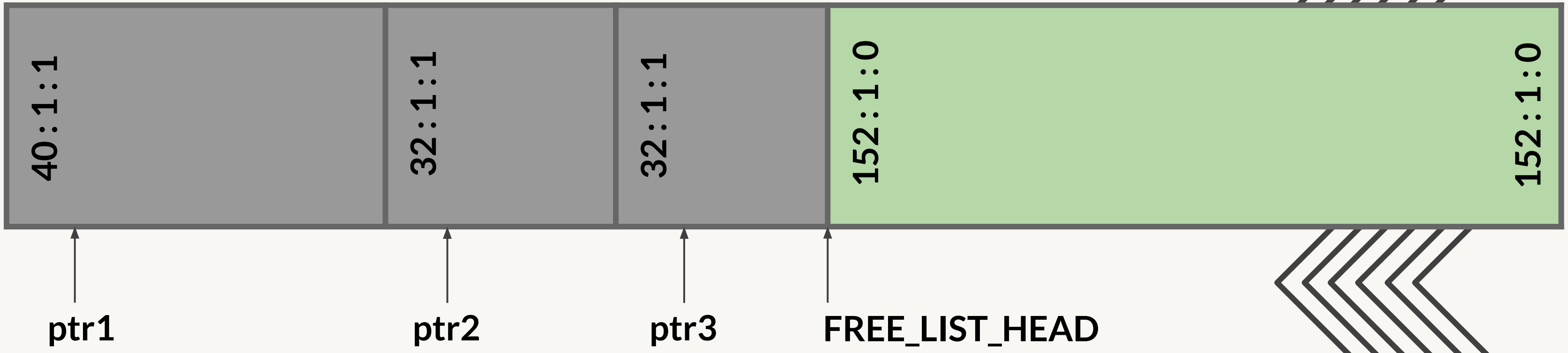
- Same procedure as before



Walkthrough of Example Heap

free(ptr2);

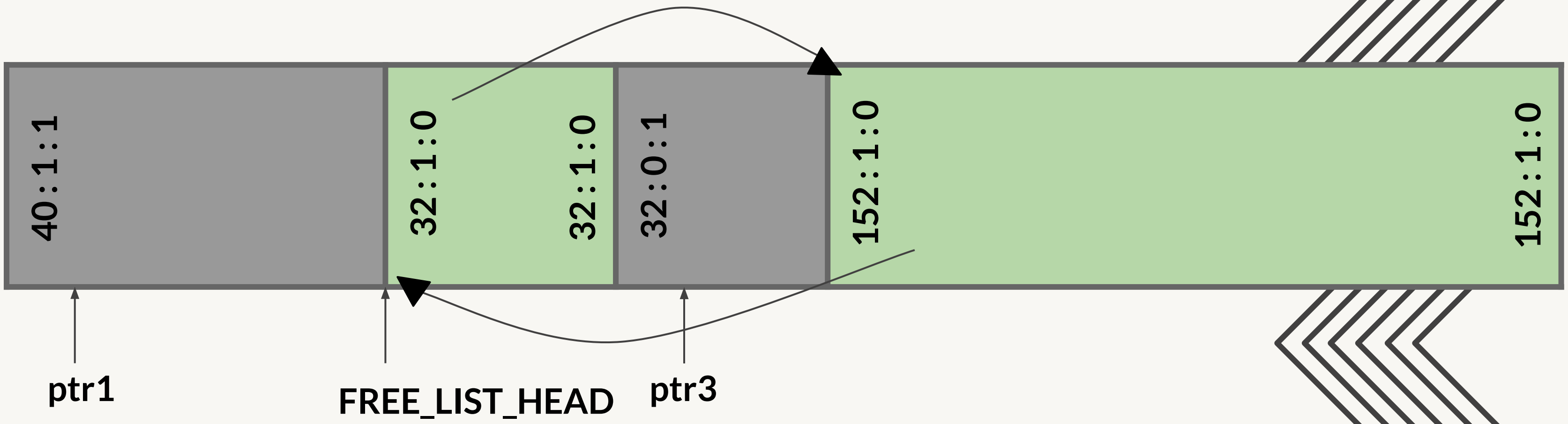
- Now we need to free a block!



Walkthrough of Example Heap

free(ptr2);

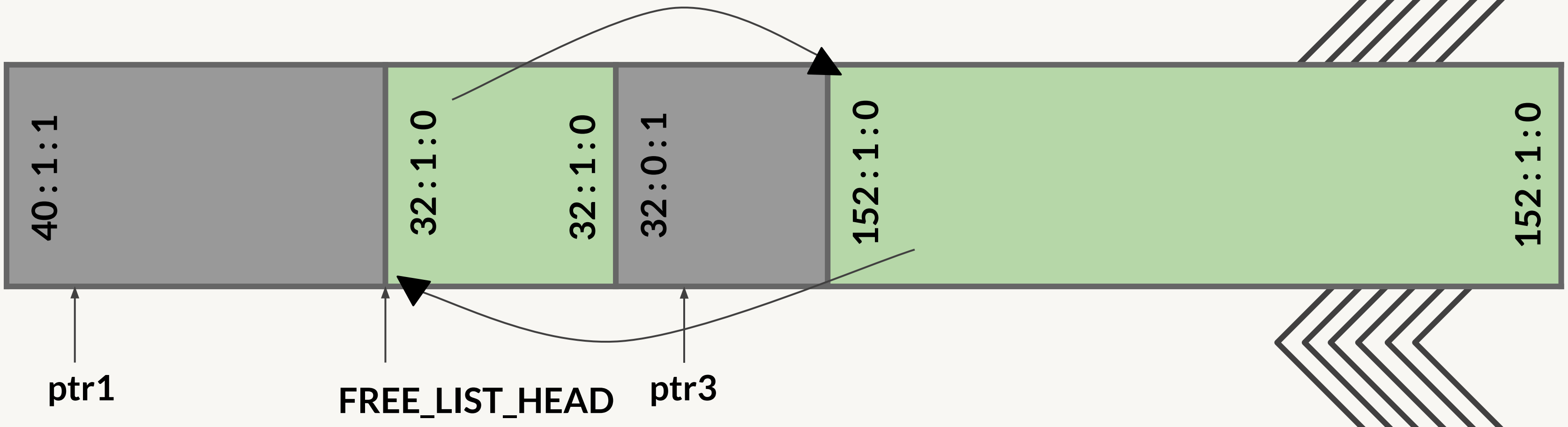
- Need to insert block allocated for ptr2 into the free list (and update tags!)
- Which tags get updated?



Walkthrough of Example Heap

free(ptr3);

- Same thing as before, except now the pointers get really messy...

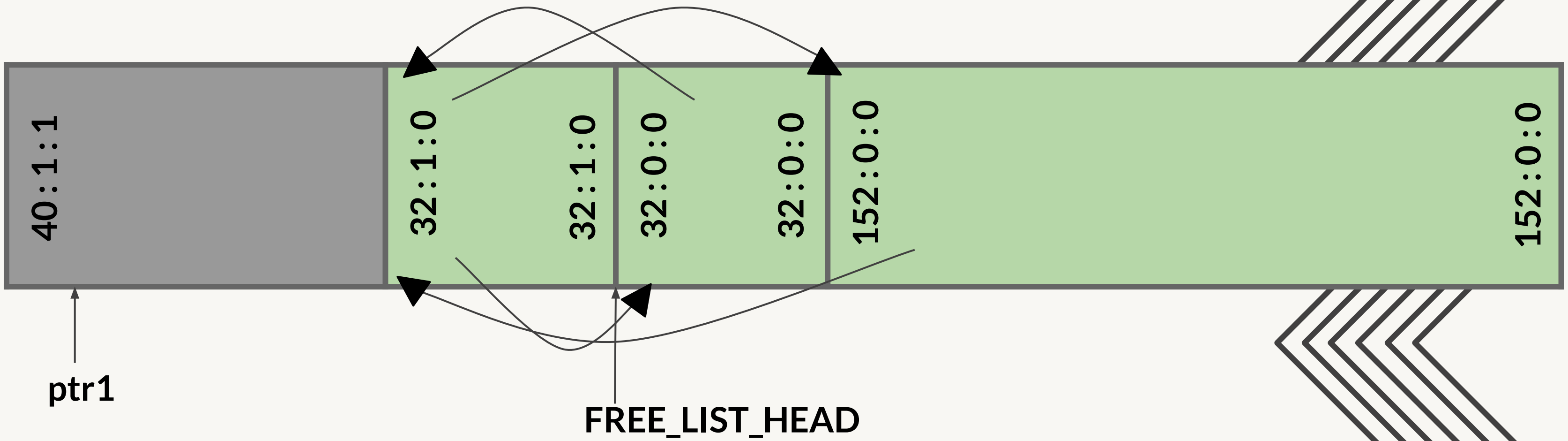


Walkthrough of Example Heap

*THIS IS AN INVALID STATE, JUST FOR DEMO PURPOSES

free(ptr3);

- Same thing as before, except now the pointers get really messy…
    - next pointers are the ones higher up in the diagram, prev lower down...

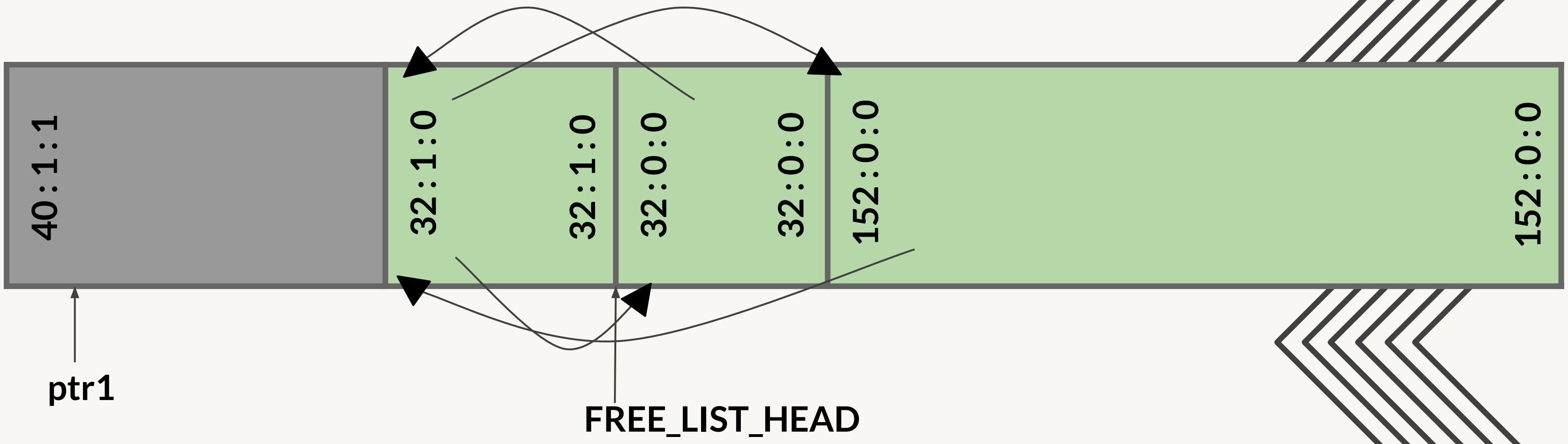


Walkthrough of Example Heap

*THIS IS AN INVALID STATE, JUST FOR DEMO PURPOSES

free(ptr3);

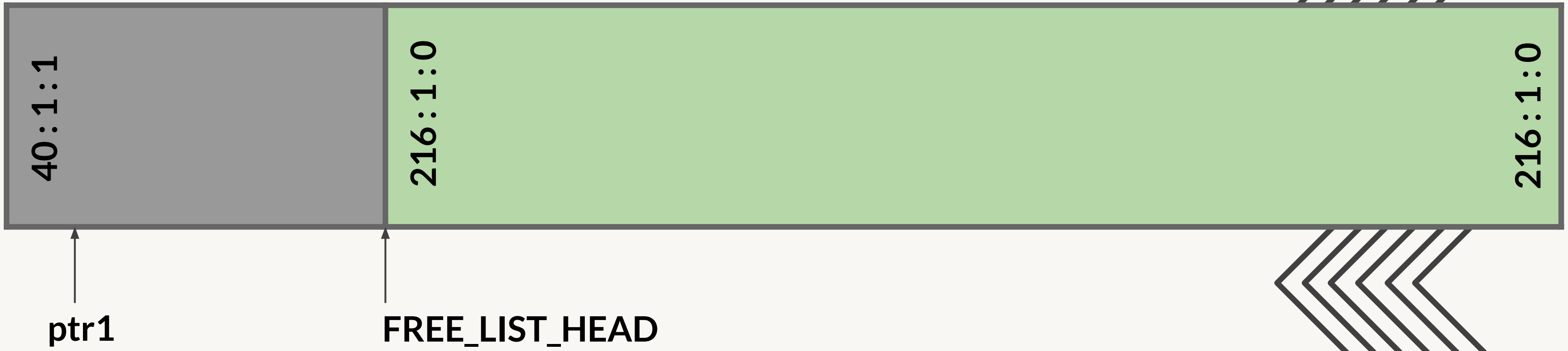
- Good enough? What happens if user calls malloc(200)?



Walkthrough of Example Heap

free(ptr3);

- Coalesce neighboring free blocks into one large free block!
- Allows for larger future mallocs, can still split later for smaller chunks



Heap Simulator

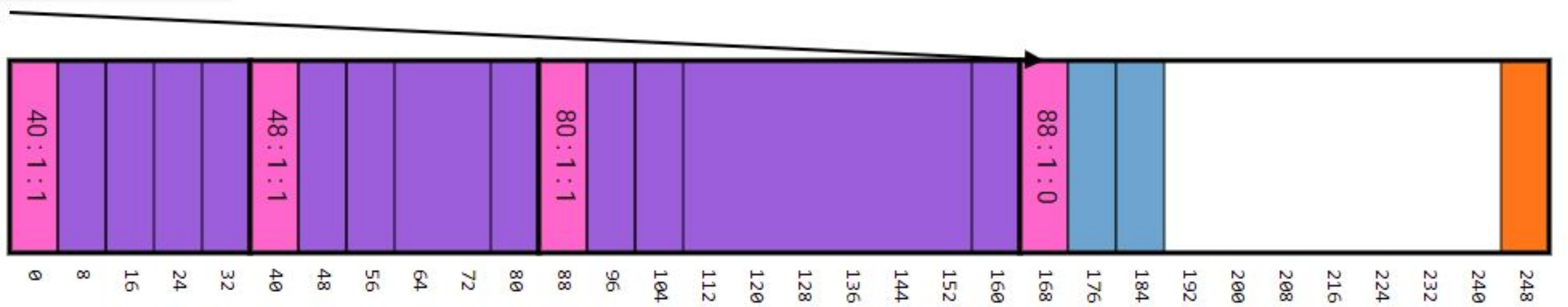
https://courses.cs.washington.edu/courses/cse351/heapsim/

Worksheet

Worksheet Problem 1

Starting with an empty heap (you can empty the heap by refreshing the page), “Execute” the following code:

```
void* ptr1 = malloc(30);
void* ptr2 = malloc(40);
void* ptr3 = malloc(70);
```



a. What pointer is returned if we execute another malloc now?

176, a word past the free list start (next available free block)

b. Which block(s) could you free that would cause fragmentation in the heap?

The block w/ pointer 48, as it is between two allocated blocks (could argue that pointer 8 works too)

c. Which block(s) could you free that would cause coalescing to occur?

The block w/ pointer 96, the only block bordered by a free one

d. Suppose free(ptr2) is run immediately after malloc(70). Draw a diagram of what the free list looks like afterwards.

⇔ [48 : 1 : 0] ⇔ [88 : 1 : 0] ⇔

e. What is the maximum size payload that we could allocate (i.e. the argument to malloc) such that we are returned a pointer to the address 48 (0x30)

40, the space to fill is 48 bytes = 8 bytes header + 40 for payload

Getting Started Lab 5

Lab 5

- You get to implement `malloc()` and `free()`!
- Less overwhelming than it may sound, we give you many functions already including:
  - `search_free_list()`
  - `insert_free_block()`
  - `remove_free_block()`
  - `coalesce_free_block()`
  - `request_more_space()`
  - see spec/starter code for full list!

Getting Started in Lab 5

- If you are struggling to understand where to get started, read through `coalesce_free_block()`
  - Understanding the details of this function will provide clarity on the general structure you are manipulating
- Make sure you use the provided static inline functions and macros!
  - This will help to minimize potential bugs and make your code more readable
- HINT: The variables defined for you at the top of the `mm_malloc()` and `mm_free()` functions are good indicators of the code you will write

Lab 5 Provided Code

- **Static inline functions**
  - `UNSCALED_POINTER_ADD(p,x)` Add without using "pointer arithmetic"
  - `UNSCALED_POINTER_SUB(p,x)` Subtract without using "pointer arithmetic"
  - `SIZE(x)` Extracts the size from the size_and_tags field
- **Macros**
  - `MIN_BLOCK_SIZE` The size of the smallest block that is safe to allocate
  - `TAG_USED` Mask for the used tag (1 = 0b1)
  - `TAG_PRECEDING_USED` Mask for the preceding used tag (2 = 0b10)
  - `WORD_SIZE` Size of a word on this architecture
- **There are lots more, don't forget to use them!**
  - They will absolutely make your life easier
  - Part of good C style (which will be part of this assignment's grade)

The block_info struct

In Lab 5, we will use struct pointers to read and manipulate block metadata in the heap:

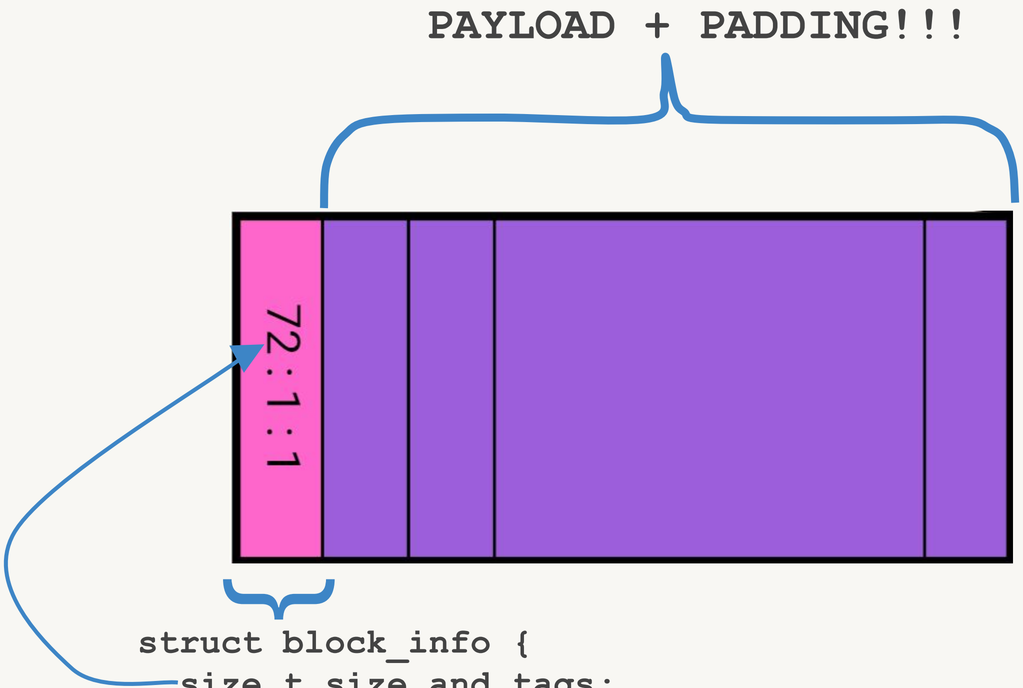
```
    struct block_info {
    // Size of the block and tags (preceding-used?, 
is-allocated?)
      size_t size_and_tags;
      struct block_info* next;
      struct block_info* prev;
    };
    typedef struct block_info block_info;
```

Lab 5 Free Blocks Revisited


```
struct block_info {
  size_t size_and_tags;
  struct block_info* next;
  struct block_info* prev;
};
```

`size_t size_and_tags;`

Lab 5 Allocated Blocks Revisited




```
struct block_info {
  size_t size_and_tags;
  struct block_info* next;
  struct block_info* prev;
};
```

- We can still manipulate an allocated block using a `struct block_info*`. Why?


No one cares about your memory leaks
JAVA
CLANG
GCC
INTEL C++ COMPILER
imgflip.com

Worksheet

Practice!

```
struct block_info {
  size_t size_and_tags;
  struct block_info* next;
  struct block_info* prev;
};
typedef struct block_info block_info
```

Given **void* ptr** is a pointer to the ***beginning*** of a free block.

Give a C expression that sets the previous blocks next pointer to ptr’s next block, as would be done if we were removing ptr from the free list.

```
((block_info*)ptr)->prev->next = ((block_info*)ptr)->next
```

Practice!

```
struct block_info {
  size_t size_and_tags;
  struct block_info* next;
  struct block_info* prev;
};
typedef struct block_info block_info
```

Given **void* ptr** is now a pointer to the ***payload*** of an allocated block, use macros and inline functions provide C expressions that get the following in terms of **ptr** :

NOTE: **UNSCALED_POINTER_ADD/SUB** returns a **void***

Size of allocated block

```
size_t size_curr_blk = SIZE(((block_info*)UNSCALED_POINTER_SUB(ptr,
                       WORD_SIZE))->size_and_tags)
```

Set **TAG_PRECEDING_USED** of following block to **True**

```
block_info* flw_blk = (block_info*)UNSCALED_POINTER_ADD(ptr, size_curr_blk -
                      WORD_SIZE)

flw_blk->size_and_tags = (flw_blk->size_and_tags) | TAG_PRECEDING_USED
```

C Macros

Pre-compile time "find and replace" your code text

Defining constants:

- `#define NUM_ENTRIES 100`
  - OK

Defining simple operations:

- `#define twice(x) 2*x`
  - *Not* OK, `twice(x+1)` becomes `2*x+1` because preprocessor uses naive find and replace
- `#define twice(x) (2*(x))`
  - OK, now `twice(x+1)` becomes `2*(x+1)`
  - Always wrap in parentheses!
  - Usually less dangerous and easier to debug if converted to a static inline function

Why even use Macros?

- Why macros?
  - Create more readable/reusable code for constants
  - “Faster” than function calls
  - In malloc: Quick access to header information (payload size, used tag, etc.)
- Drawbacks
  - Less expressive than functions
  - Arguments are not typechecked, local variables
  - They can easily lead to errors that are more difficult to find (see previous slide)

copy_tags

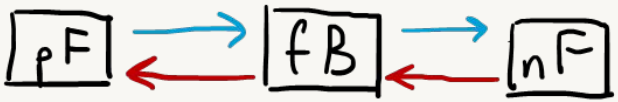
```c
// Bit masks used to retrieve tags from size_and_tags.
#define TAG_USED 1
#define TAG_PRECEDING_USED 2
// SIZE(block_info->size_and_tags) extracts the size of a 'size_and_tags' field.
static inline size_t SIZE(size_t x) {return ((x) & ~(ALIGNMENT - 1));}

// Copies the tags (TAG_PRECEDING_USED and TAG_USED) from block_to_copy
// to original_block. Leaves the size of original_block unchanged.
void copy_tags(block_info* original_block, block_info* block_to_copy) {
  size_t copy_used =
      (block_to_copy->size_and_tags) & TAG_USED;
  size_t copy_preceding_used =
      (block_to_copy->size_and_tags) & TAG_PRECEDING_USED;
  original_block->size_and_tags =
      SIZE(original_block->size_and_tags) | copy_preceding_used | copy_used;
}
```

* Note, this is not the full remove_free_block function

remove_free_block

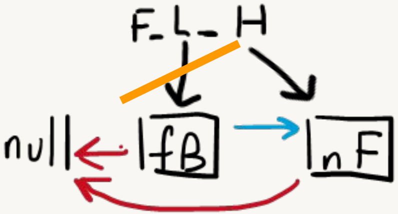
fB = free_block
nF = next_free
pF = prev_free
F_L_H = FREE_LIST_HEAD

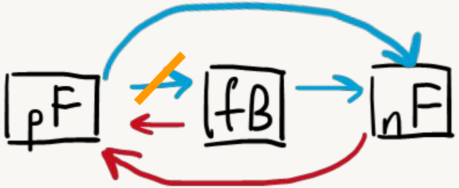



```c
block_info* FREE_LIST_HEAD;
// Removes a block from the free list.
void remove_free_block(block_info* free_block) {
  block_info *next_free, *prev_free;
  next_free = free_block->next;
  prev_free = free_block->prev;

  // If the next block is not NULL, patch its prev pointer
  if (next_free != NULL) next_free->prev = prev_free;

  // If we're removing the head of the free list, patch the head
  // Otherwise, patch the previous block's next pointer
  if (FREE_LIST_HEAD == free_block)
    FREE_LIST_HEAD = next_free;
  else
    prev_free->next = next_free;
}
```





F_L_H == fB iff pF == null!

Implementing `malloc()`

1. Figure out how big a block you need (factor in alignment, minimum block size, etc)
2. Call `search_free_list()` to get a free block that is large enough
   a. NOTE: this will yield a block that is AT LEAST the request size
   b. What happens if we run out of space in the heap?
3. Remove that block from the free list
   a. May need to split this block to prevent excessive internal fragmentation (see NOTE above) – what dictates whether we can split?
   b. May need to reinsert extra block into the free list if we split
4. Update size_and_tags appropriately (do preceding and following blocks need updating?)
5. Return a pointer to the payload of that block

Implementing `free()`

1. Convert the given used block into a free block (what is used to mark whether the block is free or not?)
2. Update size_and_tags appropriately (do preceding and following blocks need updating?)
   a. Don’t forget to update the footer size_and_tags as well!
3. Reinsert free block into the head of the free list
4. Coalesce preceding and following blocks if necessary

Other Hints and Quirks about the Code

- Structs: we can use arrow notation on a struct pointer as a way of accessing struct fields.
  - `ptr1->size_and_tags` is essentially syntactic sugar for taking the struct pointer, dereferencing it, and accessing the `size_and_tags` field from this struct instance

- Heap boundaries: the start of the heap (lowest addresses) and the end of the heap (highest addresses) are marked by a "useless" word
  - Make sure to keep these boundaries in mind when updating tags and that these boundary words have tags, too!

- Use bit masking to extract important information
  - Also recall that 0 is a falsy value and all other values are truthy values

That's All, Folks!

Thanks for attending section! Feel free to stick around for a bit if you have quick questions (otherwise post on Ed or go to office hours).