CSE 351 Section 9 – Memory Allocation

The Heap Simulator

The heap simulator can be accessed via (1) this heap simulator link, (2) Simulators → Heap Simulator from the course website navigation bar, or (3) the Resources section of the Lab 5 specs, and mimics the setup and desired behavior of the Lab 5 memory allocator.

The "Help" tab in the upper-left will provide information on how to interpret the visuals and how to use the simulator. If "Simulation Mode" is enabled (enabled by default), it will walk you through each step of the allocation or deallocation process, depending on your current action. Notice that each boundary tag (*i.e.*, header or footer) is 8 bytes wide and contains three fields in the last few bits, corresponding to:

```
block size : prec-used? : is-used?
```

- For the used? tags, `1` means allocated and `0` means unallocated.
- The block size includes all padding and metadata (*i.e.*, headers and footers).

Exercise 1: Heap Terminology and Behaviors

In the Heap Simulator, start with an empty heap (*i.e.*, refresh the page, if needed) and then "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?

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

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

d) How many boundary tags do we need to update when calling `free(ptr2)`?

e) After calling `free(ptr2)`, which block is at the head of the free list? How many non-null free-list pointers are there?

Lab 5 Coding Introduction

You will implement part of a dynamic memory allocator that uses an explicit free list. Each free block has pointers to the next and previous blocks in the free list. We use the following `struct` to manipulate the heap blocks:

```
struct block_info {
  // Size of the block (in the high bits) and tags for whether the
  // block and its predecessor in memory are in use.
  size_t size_and_tags;
  struct block_info* next;
  struct block_info* prev;
};
typedef struct block_info block_info;
```

We provide you with a lot of resources (*e.g.*, helper functions, macros, static inline functions). Here are some of them (see `mm.c` for all of them):

- `UNSCALED_POINTER_ADD(p,x)` – Add without using "pointer arithmetic", returns void*
- `UNSCALED_POINTER_SUB(p,x)` – Subtract without using "pointer arithmetic", returns void*
- `SIZE(x)` – Extracts the size from the size_and_tags field
- `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

Example: block_info Usage

Given a **`void*`** `ptr` pointer to the *beginning* of a free block (*i.e.*, the header), give C code that sets the previous block's `next` pointer to `ptr`'s `next` block, as would be done if we were removing `ptr` from the free list.

Solution: We can do this by casting `ptr` to a **`block_info*`** and accessing its relevant metadata using the struct's fields. This is done step-by step on separate lines by finding the current, previous, and next blocks:

```
1) block_info* curr_blk = (block_info*)ptr;
2) block_info* prev_blk = curr_blk->prev;
3) block_info* next_blk = curr_blk->next;
4) prev_blk->next = next_blk;
```

Or, we can do this all in one line:

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

Either way works, but the step-by-step formulation is a lot more readable and makes debugging easier.

Exercise 2: Get Block Size

`void*` `ptr` points to the *payload* of an allocated block. Use the above Lab 5 provided code to **get the size of the allocated block**:

```
size_t size_curr_blk = _____;
```

Exercise 3: Set Prec-used?

`void*` `ptr` points to the *payload* of an allocated block. Use the above Lab 5 provided code to **set TAG_PRECEDING_USED of the *following* block to True** (can use `size_curr_blk`):

```
block_info* flw_blk = _____;
```

```
flw_blk->size_and_tags = _____;
```

Exercise 4: Copy Tags

Implement the following function. Try using bitwise operators to access the tags in `size_and_tags`.

```
// 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 = _____;

    size_t copy_preceding_used = _____;

    original_block->size_and_tags = _____;
}
```