Dynamic Memory Management and Data Structures ─ Heap Walker

Your assignment is to develop assembly language components for 64-bit x86 on Linux that will display the current metadata that describes the state of the libc heap. Your code must implement the interface described by the C header file assign6.h (supplied separately), which contains the following declarations.

```
typedef struct _heap_stats {
    uint32_t num_free_blocks;          //offset 0
    uint32_t num_allocated_blocks;     //offset 4
    size_t    largest_allocated;       //offset 8
    size_t    largest_free;            //offset 16
    void      *first_allocated;        //offset 24
    void      *last_allocated;         //offset 32
    void      *tail_chunk;             //offset 40
} heap_stats;

/*****
This function walks the libc heap that begins at heap_base and
prints out information about each block (allocated or unallocated)
that exists in the heap to include start address, size, allocation
status, and next and prev free-list pointers if the block is
unallocated.

In addition to generating the output described above, the
function also populates the heap_stats structure pointed to by
stats with various summary data about the heap.

Please refer to the description of heap_stats above as well as
the assign6.h header file.

*****/
void heap_walk(void *heap_base, heap_stats *stats);
```

The libc heap manager is responsible for obtaining a block of memory from the kernel and then satisfying a program's dynamic memory requests made through the *malloc* and *free* interfaces. Each time a program makes a memory request from malloc, a suitable block of free memory of the appropriate size must be located, and a pointer to the located memory returned to the caller of malloc.

To keep track of the state of the heap, the heap manager maintains data structures that are closely associated with each malloc'ed chunk of memory. These data structures help the heap manager quickly locate block of memory capable of satisfying a caller's request for memory. A functional heap manager should be able to distinguish free blocks from allocated blocks in order to avoid allocating the same block twice. It must also be able to determine the size of any existing block, particularly when calls to free are made, in order to know how much memory is being returned by a caller of the free function. Allocation and freeing of memory should

happen relatively quickly (O(1) would be ideal), without the need for an exhaustive search to find a suitable unallocated chunk to satisfy malloc requests.

Heap Layout

During the lifetime of a program, the heap manager subdivides a larger block of memory (the *heap*) into smaller blocks of contiguous memory known as *chunks*. The heap manager is capable of managing chunks of non-uniform size. The Linux heap manager makes use of three distinct chunk types while managing the heap. Each chunk type has an associated data structure used by the heap manager to keep track of the chunk. A chunk consists of two or more header fields followed by the data region that may (allocated) or may not (unallocated) be in used by the program. The metadata contained in the chunk header fields allow malloc (or you, in this case) to locate any chunk within the heap and determine its size and whether it is currently in use or not.

In this assignment, you will iterate over all chunks in the heap to report the state of the heap.

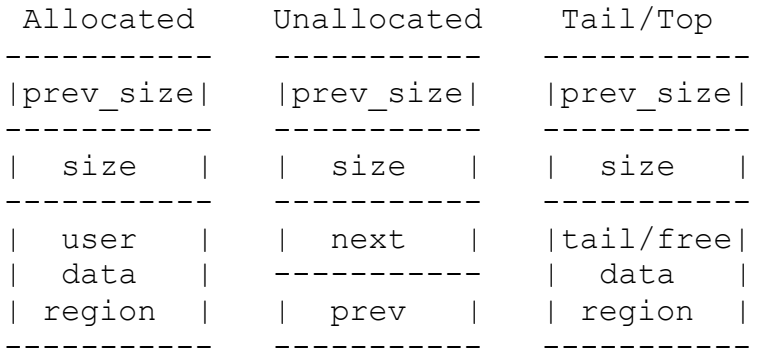
Independent of its status, the first chunk in the heap may always be found at the start of the heap (i.e., *heap_base*). The first two fields in any chunk type are both 8-byte size fields known respectively as `prev_size` and `size`. The size field will always be a multiple of 16 bytes as a result of rounding up done with the size value that the program passes to malloc. Keep in mind for later that any number that is a multiple of 16 ends with a zero when expressed as a hexadecimal number. The size field represents the size of the entire chunk as allocated to include the size of the header field (16 bytes for allocated and tail chunks) and the associated data region of the chunk. The header for an allocated chunk is followed immediately by the data block that malloc has returned to the caller.

A second type of chunk utilized by the heap manager is called the *tail* or *top* chunk. Within the heap there is always exactly one tail chunk, which represents the available (as in free/unallocated) memory that is immediately adjacent to the current program brk address (i.e., the address at which we fall off the end of the heap). Upon initial creation the heap contains a single chunk encompassing all of the available memory in the heap. This chunk, by virtue of being adjacent to the brk is a tail chunk. Like an allocated chunk, the only two fields in the tail chunk's header are a *prev_size* and a *size* field. The tail chunk is used by malloc only as a last resort to satisfy malloc requests that are too large to be satisfied by any other free chunk (including the case in which there are no other free chunks) within the heap.

The third type of chunk that may be present within the heap is known as a *free* or *unallocated* chunk. While such a chunk describes space available to malloc to satisfy future malloc requests, unallocated chunks are treated differently than the tail chunk. When malloc receives a request for memory, it is a waste of time for malloc to spend any time looking at allocated chunk as no allocated chunk can possible satisfy the new malloc request. Malloc is made more efficient if it looks only at free chunks that might satisfy a new request for memory, and more efficient still if it can limit its scan of free chunks to just those chunks that are most likely to be large enough to satisfy the new request. It is a waste of time to observe that a chunk of 80 bytes is available if

the caller is requesting a 200-byte block. In order to speed up this search process, free chunks are linked together using linked lists (the heap manager maintains many of these lists) which serve to link together free chunks of similar sizes. The range of sizes associated with a list is known as a *bin*. As a result of being a member of a linked list, all free chunks contain additional header fields that hold the linked list pointers used to track the chunk in its assigned list. Depending on the size of the chunk, the list into which it gets linked will be either a singly linked list (smaller chunks) or a doubly linked list (larger chunks). With the chunk header, the *next* and potentially *prev* link fields immediately follow the *prev_size* and *size* fields that are common to all chunks.

Each chunk type is shown in high definition below:



In 64-bit Linux, each field in a chunk header occupies 8 bytes.

With a proper understanding of the layout of chunks, it is hopefully clear that given a pointer to a chunk (such as the one found as the beginning of the heap) that we can find the next chunk by reading the chunk's size field and adding it the chunk start address to derive the address of the next chunk, and so, continue to walk the heap from beginning to end.

Occasionally, it is useful to be able to answer the question "Is this chunk allocated" which should result in a simple true or false answer. You may note that there is no information contained in a chunk header that explicitly indicates whether the chunks type, and more specifically whether it is allocated or free. This is where the Linux heap developers got a little obscure. Recall that all sizes will be multiples of 16 and thus all end in a 4 binary zeros. Because we know that the correct value of those bits must be zero, we can overlay additional information onto those 4 bits (such as 4 true/false flags) that we simply ignore if we want to retrieve the actual size. The Linux heap developers make use of these bits as Boolean flags. We will only interest ourselves with bit 0 of the size which the heap developer call the `PREV_IN_USE` bit, which indicates whether the chunk that precedes the current chunk is free or not. In other words, given a chunk address (A), in order to know whether A is free or allocated, we must find the next chunk (B = A + A.size) and then examine bit 0 of B's size field (B.size & 1) to determine whether A is in use or not. This works for all chunks except the tail chunk, which, by definition, has no next chunk that follows it. However, also by definition, the tail chunk is never in use.

What you need to do

Your program must start at a given (as a parameter) address and report the status of every block in the heap, while also generating some summary statistics about the heap.

When you encounter an allocated block, you should print something like the following:

```
0x605ca0: Allocated (1072 bytes)
```

by using the format strings you find in formats6.asm.

When you encounter an unallocated block, you should print something like the following:

```
0x606770: Unallocated (32 bytes). next: 0x603260, prev: (nil)
```

by using the format strings you find in formats6.asm.

Your block summary should be ordered by increasing chunk address.

While you are iterating over the chunks in the heap, you may want to gather some of the statistics that you need to fill in the caller's heap_status structure.

After printing all of the chunk header information, you must print out the number of bytes that are allocated within the heap as well as the number of bytes that are free.

```
A total of 12384 bytes are allocated
A total of 121616 bytes are free
```

Your code must adhere to the following guidelines:

1. You may not use any third party libraries, other than libc installed from a standard Ubuntu package for I/O functions, and formats6.asm provided to you. Otherwise, all code must be your own.
2. Your code must implement the `heap_walk` interface described in assign6.h. `heap_walk` must be declared global within your code.
3. You may write and utilize any additional functions and/or data structures that you find necessary to support your code. Any such functions should not be declared global.
4. The functionality of assign6.h must be implemented in a file named assign6.asm.

Your program must be thoroughly commented. Endline comments must be justified, must not crowd the code, and must not wrap; start a new line instead. Use standalone comments to describe complex logic.

To assist you in generating random heap chunks to test your heap walker, a sample test program (tester.c) is provided. This is not as comprehensive as the test harness program that will be used to test your code. Among other things it does not tell you whether your output is correct or not.

An example run of the heap walker might look like this:

```
$ ./assign6_test
0x603000: Allocated (192 bytes)
0x6030c0: Allocated (192 bytes)
0x603180: Allocated (320 bytes)
0x6032c0: Unallocated (96 bytes). next: 0x603890, prev: (nil)
0x603320: Allocated (80 bytes)
0x603370: Allocated (304 bytes)
0x6034a0: Unallocated (96 bytes). next: (nil), prev: (nil)
0x603500: Allocated (224 bytes)
0x6035e0: Unallocated (112 bytes). next: 0x6042e0, prev: (nil)
0x603650: Allocated (96 bytes)
0x6036b0: Allocated (48 bytes)
0x6036e0: Allocated (112 bytes)
0x603750: Allocated (320 bytes)
0x603890: Unallocated (96 bytes). next: 0x605e10, prev: (nil)
0x6038f0: Allocated (432 bytes)
0x603aa0: Unallocated (160 bytes). next: (nil), prev: (nil)
0x603b40: Allocated (256 bytes)
0x603c40: Allocated (384 bytes)
0x603dc0: Unallocated (64 bytes). next: (nil), prev: (nil)
0x603e00: Allocated (224 bytes)
0x603ee0: Allocated (464 bytes)
0x6040b0: Unallocated (128 bytes). next: 0x606940, prev: (nil)
0x604130: Allocated (64 bytes)
0x604170: Allocated (304 bytes)
0x6042a0: Unallocated (64 bytes). next: 0x604e30, prev: (nil)
0x6042e0: Allocated (112 bytes)
0x604350: Allocated (112 bytes)
0x6043c0: Allocated (112 bytes)
0x604430: Allocated (288 bytes)
0x604550: Allocated (272 bytes)
0x604660: Unallocated (112 bytes). next: (nil), prev: (nil)
0x6046d0: Allocated (96 bytes)
0x604730: Allocated (320 bytes)
0x604870: Unallocated (128 bytes). next: (nil), prev: (nil)
0x6048f0: Allocated (512 bytes)
0x604af0: Unallocated (224 bytes). next: (nil), prev: (nil)
0x604bd0: Allocated (304 bytes)
0x604d00: Allocated (304 bytes)
0x604e30: Unallocated (64 bytes). next: 0x603dc0, prev: (nil)
0x604e70: Allocated (48 bytes)
0x604ea0: Allocated (128 bytes)
0x604f20: Allocated (1120 bytes)
0x605380: Unallocated (80 bytes). next: (nil), prev: (nil)
0x6053d0: Allocated (528 bytes)
0x6055e0: Unallocated (32 bytes). next: (nil), prev: (nil)
0x605600: Allocated (112 bytes)
0x605670: Allocated (32 bytes)
0x605690: Allocated (128 bytes)
0x605710: Allocated (192 bytes)
```

```
0x6057d0: Allocated (320 bytes)
0x605910: Allocated (224 bytes)
0x6059f0: Allocated (288 bytes)
0x605b10: Allocated (160 bytes)
0x605bb0: Allocated (608 bytes)
0x605e10: Unallocated (96 bytes). next: 0x603650, prev: (nil)
0x605e70: Allocated (96 bytes)
0x605ed0: Allocated (80 bytes)
0x605f20: Allocated (96 bytes)
0x605f80: Allocated (208 bytes)
0x606050: Unallocated (32 bytes). next: (nil), prev: (nil)
0x606070: Allocated (112 bytes)
0x6060e0: Allocated (160 bytes)
0x606180: Unallocated (64 bytes). next: (nil), prev: (nil)
0x6061c0: Allocated (80 bytes)
0x606210: Allocated (208 bytes)
0x6062e0: Allocated (80 bytes)
0x606330: Allocated (224 bytes)
0x606410: Allocated (736 bytes)
0x6066f0: Unallocated (160 bytes). next: (nil), prev: (nil)
0x606790: Allocated (128 bytes)
0x606810: Allocated (304 bytes)
0x606940: Unallocated (128 bytes). next: 0x606e70, prev: (nil)
0x6069c0: Allocated (240 bytes)
0x606ab0: Unallocated (144 bytes). next: (nil), prev: (nil)
0x606b40: Allocated (288 bytes)
0x606c60: Unallocated (160 bytes). next: (nil), prev: (nil)
0x606d00: Allocated (192 bytes)
0x606dc0: Allocated (176 bytes)
0x606e70: Allocated (128 bytes)
0x606ef0: Allocated (320 bytes)
0x607030: Allocated (160 bytes)
0x6070d0: Unallocated (96 bytes). next: (nil), prev: (nil)
0x607130: Allocated (160 bytes)
0x6071d0: Allocated (864 bytes)
0x607530: Unallocated (208 bytes). next: (nil), prev: (nil)
0x607600: Unallocated (117248 bytes). next: (nil), prev: (nil)
A total of 14384 bytes are allocated
A total of 119408 bytes are free
$
```

Deliverables:

1. Your assembly language source file, which must be named **exactly** as assign6.asm.
2. Use comments in each file to document the commands required to assemble your code, your name(s), the class number, and the assignment number.
3. Submit your source code to the assignment submission page in Sakai.