CS 361: Computer Systems **Lab Session 7 Worksheet**
Fall 2019

Garbage Collection

1. In this lab, we will be building a garbage collector for malloc. As usual, you should start by going to Gradescope and skimming the questions. Also, this lab will be walking through fundamental parts of the homework, so please (if you haven't already) go to the coursepage, accept the hw4 assignment, and get the skeleton code.

2. You will be using the mark and sweep algorithm for garbage collection in this lab. You will first start with any given set of pointers and mark the memory they point to, then you will crawl through all allocated blocks and free any that are unmarked.

3. To begin, it is helpful to know something about malloc's structure. The following excerpt is taken from malloc's source code, available here:
https://code.woboq.org/userspace/glibc/malloc/malloc.c.html

4. After reading the excerpt below, answer the question about it on Gradescope.

```
An allocated chunk looks like this:


    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if allocated            | |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                       |M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
            .             (malloc_usable_size() bytes)                      .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk                                     |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

             Where "chunk" is the front of the chunk for the purpose of most of
    the malloc code, but "mem" is the pointer that is returned to the
    user.  "Nextchunk" is the beginning of the next contiguous chunk.

    Chunks always begin on even word boundaries, so the mem portion
    (which is returned to the user) is also on an even word boundary, and
    thus at least double-word aligned.
```

```
    Free chunks are stored in circular doubly-linked lists, and look like this:

    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk                            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `head:' |             Size of chunk, in bytes                         |P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Forward pointer to next chunk in list             |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Back pointer to previous chunk in list            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Unused space (may be 0 bytes long)                .
            .                                                               .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `foot:' |             Size of chunk, in bytes                            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

    The P (PREV_INUSE) bit, stored in the unused low-order bit of the
    chunk size (which is always a multiple of two words), is an in-use
    bit for the *previous* chunk.  If that bit is *clear*, then the
    word before the current chunk size contains the previous chunk
    size, and can be used to find the front of the previous chunk.
    The very first chunk allocated always has this bit set,
    preventing access to non-existent (or non-owned) memory. If
    prev_inuse is set for any given chunk, then you CANNOT determine
    the size of the previous chunk, and might even get a memory
    addressing fault when trying to do so.

    Note that the `foot' of the current chunk is actually represented
    as the prev_size of the NEXT chunk. This makes it easier to
    deal with alignments etc but can be very confusing when trying
    to extend or adapt this code.
```

Skeleton Code: Working with Chunks

1. Once you feel comfortable with malloc's structures, you should familiarize yourself with the skeleton code. You will be working with the code in hw4.c - this is where all of your garbage collector functionality will go. Additionally, we provide you with main.c and debugmain.c for testing, and a Makefile. Start by looking in hw4.c.

2. Let's start by looking at the helper function hw4.c provides you for working with the malloc memory chunks. First, note that it defines

```
#define chunk_size(c)  ((*((unsigned int *)c))& ~(unsigned int)7 )
```

   This is getting the size of the chunk by dereferencing a pointer, and then bitwise ANDing it with the inverted constant 7. If you look at code that uses this function, you can see that the pointer it is being called on is the chunk pointer in the diagrams above. Recall that three in binary will be 00 . . . . 00111, and inverting it will give you 11 . . . . 11000. What is this function doing? What is the point of the AND? Answer the question on chunk_size on Gradescope.

3. Next, let's look at the code that gets the next chunk.

```
void* next_chunk(void* c) {
  if(chunk_size(c) == 0) {
    fprintf(stderr,"Panic, chunk is of zero size.\n");
  }
  if((c+chunk_size(c)) < mem_heap_hi())
    return ((void*)c+chunk_size(c));
  else
    return 0;
}
```

   Notice that it's adding the size of the chunk to the pointer to the chunk. What will this point at? Additionally, notice that it is checking that the result is less than mem_heap_hi(), a call which will simply return the address of the last valid byte of the heap. Why is it doing this check? Answer the question on next_chunk on Gradescope.

4. Next, look at the in_use function, below:

```
int in_use(void *c) {
  return *(unsigned int *)(c) & 0x1;
}
```

   Here, we dereference the pointer c passed into the function before bitwise ANDing it with 1. What does the pointer point to? When will this return 0? When will it return 1? Answer the question on in_use on Gradescope.

Skeleton Code: Marking

1. Look at the code for is_marked, mark, and clear_mark. (The code for mark is below.)

```
void mark(unsigned int * chunk) {
  (*chunk)|=0x2;
}
```

Notice that all of these functions set or check the second bit by ORing or ANDing with the constant 2 (010 in binary). We are taking advantage of the fact that the last three bits of the chunk's size will always be zero, and using the second bit to hold our mark. Answer the question on clear_mark on backboard.

Finding .data, heap, and stack

1. For the mark part of the code, you will need to scan the .data section and stack to find pointers, while in the sweep section, you will need to go through all of the chunks on the heap. We set up start and end points of all of these sections for you in two functions: init_gc, and gc. Init_gc will always be called first, before we allocate any memory or call the garbage collector, while gc will be called whenever we want to garbage collect.

   ```
   void init_gc() {
     size_t stack_var;
     init_global_range();
     // since the heap grows down, the end is found first
     stack_mem.end=(size_t *)&stack_var;
   }
   ```

   Both init_gc and gc use a local variable to find where they are on the stack. This works because local variables are always on the stack, and since we know initgc is our first function to be called, we can use it demarcate the beginning of the stack. Likewise, we call malloc in init_gc to find the beginning of the heap. (init_global_range uses some linux features to find the .data section. It is to complicated to discuss here, but feel free to ask Professor Kanich if you have questions about it.) All init_gc does is set up the start of each memory section we are interested in.

2. The gc function both finds the ends of the heap and stack, and calls the functions you will be implementing in order to do garbage collection.

Functions You Will Write

1. `sweep()` This is the function that will sweep through your heap, freeing anything that is not marked.

2. `is_pointer(void* ptr)` This is a helper function you will write to help with marking. Given a number, it should either return 0 if the number is not an address on the heap, or return the pointer to the chunk that holds the address that the pointer passed in as a parameter points to.

3. `walk_region_and_mark(void* start, void* end)` This is the function that will go through the data and stack sections, finding pointers and marking the blocks they point to.

4. That's it! That's all the code you write! However, this assignment can be very tricky and difficult to debug. Start working on it right away to avoid running out of time! I recommend implementing the is_pointer function, then walk_region_and_mark, then sweep.

5. Don't know where to start? Try reading about the mark and sweep algorithm in section 9.9 of the book.