Assignment 2 – Dynamic Memory Management

CMSC257 - Computer Systems

Due date: Oct 25, 2017 (11:59pm)

In this assignment, you will develop a C program to implement the "malloc", "free", "calloc" and "realloc" functionalities.

**Instructions:**

1. Check out the malloc tutorial from Dan Luu on how these functions can be implemented at: http://danluu.com/malloc-tutorial/
   For your convenience, it is also appended in this file. The tutorial implements working versions of malloc, free, calloc and realloc and you can use that code as-is.

2. Write a driver function (i.e., main function) to demonstrate the usage and efficiency of these functions. The main() should have at least 10 calls each to malloc, calloc and realloc and at least 5 calls to free. Set up the function to print out the heap start/end addresses and also print out the memory leaks in your code.

3. Your main task is to implement the exercises mentioned in the tutorial. These are also shown below:

   (a) Convert the single linked list implementation of the tutorial into a doubly linked list version; make changes to all the functions accordingly to work with the doubly linked list.

   (b) malloc is supposed to return a pointer "which is suitably aligned for any built-in type". Does our malloc do that? If so, why? If not, fix the alignment. Note that "any built-in type" is basically up to 8 bytes for C.

   (c) The implemented malloc is really wasteful if we try to re-use an existing block and we don't need all of the space. Implement a function that will split up blocks so that they use the minimum amount of space necessary

   (d) After doing (c), if we call malloc and free lots of times with random sizes, we'll end up with a bunch of small blocks that can only be re-used when we ask for small amounts of space. Implement a mechanism to merge adjacent free blocks together so that any consecutive free blocks will get merged into a single block.

   (e) The current implementation implements a first fit algorithm for finding free blocks. Implement a best fit algorithm instead.

4. Repeat Step (2) with this implementation to demonstrate usage of the functions and memory leakage.

5. Your code must use the set-up mentioned in this tutorial. Other online resources can be consulted but NOT copied. The tutorial mentions some other implementations for splitting/merging blocks that can only be consulted but not copied.

To turn in:

1. You will need 4 files: (a) a header file with all the function prototypes, (b) a .c file with all the function definitions, (c) a .c file with the driver main() function and (d) a Makefile to compile your code. Note that you only

need to submit the improved versions of the functions and not the preliminary implementations from the tutorial. Additionally, include a text file mentioning which portions of your code are working and which portions don't and why.

2. Create a tarball file with all these files. Upload the program on BlackBoard by the assignment deadline (11:59pm of the day of the assignment). The tarball should be named LASTNAME-EID-assign2.tgz, where LASTNAME is your last name in all capital letters and EID is your VCU E-ID.

---

Note: Late assignments will lose 5 points per day upto a maximum of 3 days. Code must be submitted in the prescribed format.

---

**A quick tutorial on implementing and debugging malloc, free, calloc, and realloc**

---

Let's write a malloc and see how it works with existing programs!

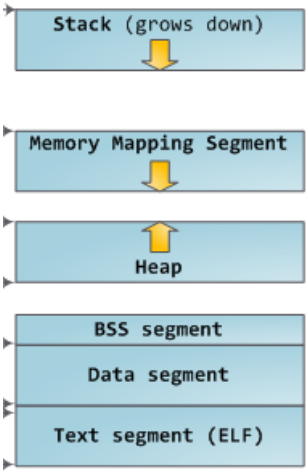
This tutorial is going to assume that you know what pointers are, and that you know enough C to know that `*ptr` dereferences a pointer, `ptr->foo` means `(*ptr).foo`, that malloc is used to dynamically allocate space, and that you're familiar with the concept of a linked list. For a basic intro to C, Pointers on C is one of my favorite books. If you want to look at all of this code at once, it's available here.

Preliminaries aside, malloc's function signature is

```
void *malloc(size_t size);
```

It takes as input a number of bytes and returns a pointer to a block of memory of that size.

There are a number of ways we can implement this. We're going to arbitrarily choose to use the sbrk syscall. The OS reserves stack and heap space for processes and sbrk lets us manipulate the heap. `sbrk(0)` returns a pointer to the current top of the heap. `sbrk(foo)` increments the heap size by `foo` and returns a pointer to the previous top of the heap.




If we want to implement a really simple malloc, we can do something like

```
#include <assert.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>


void *malloc(size_t size) {
  void *p = sbrk(0);
  void *request = sbrk(size);
  if (request == (void*) -1) {
    return NULL; // sbrk failed.
  } else {
    assert(p == request); // Not thread safe.
    return p;
  }
}
```

When a program asks malloc for space, malloc asks sbrk to increment the heap size and returns a pointer to the start of the new region on the heap. This is missing a technicality, that `malloc(0)` should either return `NULL` or another pointer that can be passed to free without causing havoc, but it basically works.

But speaking of free, how does free work? Free's prototype is

```
void free(void *ptr);
```

When free is passed a pointer that was previously returned from malloc, it's supposed to free the space. But given a pointer to something allocated by our malloc, we have no idea what size block is associated with it. Where do we store that? If we had a working malloc, we could malloc some space and store it there, but we're going to run into trouble if we need to call malloc to reserve space each time we call malloc to reserve space.

A common trick to work around this is to store meta-information about a memory region in some space that we squirrel away just below the pointer that we return. Say the top of the heap is currently at `0x1000` and we ask for `0x400` bytes. Our current malloc will request `0x400` bytes from `sbrk` and return a pointer to `0x1000`. If we instead save, say, `0x10` bytes to store information about the block, our malloc would request `0x410` bytes from `sbrk` and return a pointer to `0x1010`, hiding our `0x10` byte block of meta-information from the code that's calling malloc.

That lets us free a block, but then what? The heap region we get from the OS has to be contiguous, so we can't return a block of memory in the middle to

the OS. Even if we were willing to copy everything above the newly freed region down to fill the hole, so we could return space at the end, there's no way to notify all of the code with pointers to the heap that those pointers need to be adjusted.

Instead, we can mark that the block has been freed without returning it to the OS, so that future calls to malloc can use re-use the block. But to do that we'll need be able to access the meta information for each block. There are a lot of possible solutions to that. We'll arbitrarily choose to use a single linked list for simplicity.

So, for each block, we'll want to have something like

```
struct block_meta {
  size_t size;
  struct block_meta *next;
  int free;
  int magic; // For debugging only. TODO: remove this in non-debug mode.
};

#define META_SIZE sizeof(struct block_meta)
```

We need to know the size of the block, whether or not it's free, and what the next block is. There's a magic number here for debugging purposes, but it's not really necessary; we'll set it to arbitrary values, which will let us easily see which code modified the struct last.

We'll also need a head for our linked list:

```
void *global_base = NULL;
```

For our malloc, we'll want to re-use free space if possible, allocating space when we can't re-use existing space. Given that we have this linked list structure, checking if we have a free block and returning it is straightforward. When we get a request of some size, we iterate through our linked list to see if there's a free block that's large enough.

```
struct block_meta *find_free_block(struct block_meta **last, size_t size) {
  struct block_meta *current = global_base;
  while (current && !(current->free && current->size >= size)) {
    *last = current;
    current = current->next;
  }
  return current;
}
```

If we don't find a free block, we'll have to request space from the OS using sbrk and add our new block to the end of the linked list.

```
struct block_meta *request_space(struct block_meta* last, size_t size) {
  struct block_meta *block;
  block = sbrk(0);
  void *request = sbrk(size + META_SIZE);
  assert((void*)block == request); // Not thread safe.
  if (request == (void*) -1) {
    return NULL; // sbrk failed.
  }

  if (last) { // NULL on first request.
    last->next = block;
  }
  block->size = size;
  block->next = NULL;
  block->free = 0;
  block->magic = 0x12345678;
  return block;
}
```

As with our original implementation, we request space using `sbrk`. But we add a bit of extra space to store our struct, and then set the fields of the struct appropriately.

Now that we have helper functions to check if we have existing free space and to request space, our malloc is simple. If our global base pointer is `NULL`, we need to request space and set the base pointer to our new block. If it's not `NULL`, we check to see if we can re-use any existing space. If we can, then we do; if we can't, then we request space and use the new space.

```
void *malloc(size_t size) {
  struct block_meta *block;
  // TODO: align size?

  if (size <= 0) {
    return NULL;
  }

  if (!global_base) { // First call.
    block = request_space(NULL, size);
    if (!block) {
```

```
      return NULL;
    }
    global_base = block;
  } else {
    struct block_meta *last = global_base;
    block = find_free_block(&last, size);
    if (!block) { // Failed to find free block.
      block = request_space(last, size);
      if (!block) {
        return NULL;
      }
    } else {      // Found free block
      // TODO: consider splitting block here.
      block->free = 0;
      block->magic = 0x77777777;
    }
  }

  return(block+1);
}
```

For anyone who isn't familiar with C, we return block+1 because we want to return a pointer to the region after block_meta. Since block is a pointer of type `struct block_meta`, `+1` increments the address by one `sizeof(struct(block_meta))`.

If we just wanted a malloc without a free, we could have used our original, much simpler malloc. So let's write free! The main thing free needs to do is set `->free`.

Because we'll need to get the address of our struct in multiple places in our code, let's define this function.

```
struct block_meta *get_block_ptr(void *ptr) {
  return (struct block_meta*)ptr - 1;
}
```

Now that we have that, here's free:

```
void free(void *ptr) {
  if (!ptr) {
    return;
  }

  // TODO: consider merging blocks once splitting blocks is implemented.
  struct block_meta* block_ptr = get_block_ptr(ptr);
  assert(block_ptr->free == 0);
  assert(block_ptr->magic == 0x77777777 || block_ptr->magic == 0x12345678);
  block_ptr->free = 1;
  block_ptr->magic = 0x55555555;
}
```

In addition to setting `->free`, it's valid to call free with a NULL ptr, so we need to check for NULL. Since free shouldn't be called on arbitrary addresses or on blocks that are already freed, we can assert that those things never happen.

You never really need to assert anything, but it often makes debugging a lot easier. In fact, when I wrote this code, I had a bug that would have resulted in silent data corruption if these asserts weren't there. Instead, the code failed at the assert, which make it trivial to debug.

Now that we've got malloc and free, we can write programs using our custom memory allocator! But before we can drop our allocator into existing code, we'll need to implement a couple more common functions, realloc and calloc. Calloc is just malloc that initializes the memory to 0, so let's look at realloc first. Realloc is supposed to adjust the size of a block of memory that we've gotten from malloc, calloc, or realloc.

Realloc's function prototype is

```
void *realloc(void *ptr, size_t size)
```

If we pass realloc a NULL pointer, it's supposed to act just like malloc. If we pass it a previously malloced pointer, it should free up space if the size is smaller than the previous size, and allocate more space and copy the existing data over if the size is larger than the previous size.

Everything will still work if we don't resize when the size is decreased and we don't free anything, but we absolutely have to allocate more space if the size is increased, so let's start with that.

```
void *realloc(void *ptr, size_t size) {
  if (!ptr) {
    // NULL ptr. realloc should act like malloc.
    return malloc(size);
  }

  struct block_meta* block_ptr = get_block_ptr(ptr);
  if (block_ptr->size >= size) {
    // We have enough space. Could free some once we implement split.
```

```
    return ptr;
  }

  // Need to really realloc. Malloc new space and free old space.
  // Then copy old data to new space.
  void *new_ptr;
  new_ptr = malloc(size);
  if (!new_ptr) {
    return NULL; // TODO: set errno on failure.
  }
  memcpy(new_ptr, ptr, block_ptr->size);
  free(ptr);
  return new_ptr;
}
```

And now for calloc, which just clears the memory before returning a pointer.

```
void *calloc(size_t nelem, size_t elsize) {
  size_t size = nelem * elsize; // TODO: check for overflow.
  void *ptr = malloc(size);
  memset(ptr, 0, size);
  return ptr;
}
```

Note that this doesn't check for overflow in `nelem * elsize`, which is actually required by the spec. All of the code here is just enough to get something that kinda sorta works.

Now that we have something that kinda works, we can use our with existing programs (and we don't even need to recompile the programs)!

First, we need to compile our code. On linux, something like

```
clang -O0 -g -W -Wall -Wextra -shared -fPIC malloc.c -o malloc.so
```

should work.

`-g` adds debug symbols, so we can look at our code with `gdb` or `lldb`. `-O0` will help with debugging, by preventing individual variables from getting optimized out. `-W -Wall -Wextra` adds extra warnings. `-shared -fPIC` will let us dynamically link our code, which is what lets us use our code with existing binaries!

On macs, we'll want something like

```
clang -O0 -g -W -Wall -Wextra -dynamiclib malloc.c -o malloc.dylib
```

Note that `sbrk` is deprecated on recent versions of OS X. Apple uses an unorthodox definition of deprecated – some deprecated syscalls are badly broken. I didn't really test this on a Mac, so it's possible that this will cause weird failures or or just not work on a mac.

Now, to use get a binary to use our malloc on linux, we'll need to set the `LD_PRELOAD` environment variable. If you're using bash, you can do that with

```
export LD_PRELOAD=/absolute/path/here/malloc.so
```

If you've got a mac, you'll want

```
export DYLD_INSERT_LIBRARIES=/absolute/path/here/malloc.so
```

If everything works, you can run some arbitrary binary and it will run as normal (except that it will be a bit slower).

```
$ ls
Makefile  malloc.c  malloc.so  README.md  test  test-0  test-1  test-2  test-3  test-4
```

If there's a bug, you might get something like

```
$ ls
Segmentation fault (core dumped)
```

**Debugging**

Let's talk about debugging! If you're familiar with using a debugger to set breakpoints, inspect memory, and step through code, you can skip this section and go straight to the exercises.

This section assumes you can figure out how to install gdb on your system. If you're on a mac, you may want to just use lldb and translate the commands appropriately. Since I don't know what bugs you might run into, I'm going to introduce a couple of bugs and show how I'd track them down.

First, we need to figure out how to run gdb without having it segfault. If ls segfaults, and we try to run `gdb ls`, gdb is almost certainly going to segfault, too. We could write a wrapper to do this, but gdb also supports this. If we start gdb and then run `set environment LD_PRELOAD=./malloc.so` before running the program, `LD_PRELOAD` will work as normal.

```
$ gdb /bin/ls
(gdb) set environment LD_PRELOAD=./malloc.so
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7bd7dbd in free (ptr=0x0) at malloc.c:113
113       assert(block_ptr->free == 0);
```

As expected, we get a segfault. We can look around with `list` to see the code near the segfault.

```
(gdb) list
108     }
109
110     void free(void *ptr) {
111       // TODO: consider merging blocks once splitting blocks is implemented.
112       struct block_meta* block_ptr = get_block_ptr(ptr);
113       assert(block_ptr->free == 0);
114       assert(block_ptr->magic == 0x77777777 || block_ptr->magic == 0x12345678);
115       block_ptr->free = 1;
116       block_ptr->magic = 0x55555555;
117     }
```

And then we can use `p` (for print) to see what's going on with the variables here:

```
(gdb) p ptr
$6 = (void *) 0x0
(gdb) p block_ptr
$7 = (struct block_meta *) 0xfffffffffffffe8
```

`ptr` is `0`, i.e., `NULL`, which is the cause of the problem: we forgot to check for NULL.

Now that we've figured that out, let's try a slightly harder bug. Let's say that we decided to replace our struct with

```
struct block_meta {
  size_t size;
  struct block_meta *next;
  int free;
  int magic;    // For debugging only. TODO: remove this in non-debug mode.
  char data[1];
};
```

and then return `block->data` instead of `block+1` from malloc, with no other changes. This seems pretty similar to what we're already doing – we just define a member that points to the end of the struct, and return a pointer to that.

But here's what happens if we try to use our new malloc:

```
$ /bin/ls
Segmentation fault (core dumped)
gdb /bin/ls
(gdb) set environment LD_PRELOAD=./malloc.so
(gdb) run

Program received signal SIGSEGV, Segmentation fault.
_IO_vfprintf_internal (s=s@entry=0x7fffff7ff5f0, format=format@entry=0x7ffff7567370 "%s%s%s:%u: %s%sAssertion `%s'
failed.\n%n", ap=ap@entry=0x7fffff7ff718) at vfprintf.c:1332
1332    vfprintf.c: No such file or directory.
1327    in vfprintf.c
```

This isn't as nice as our last error – we can see that one of our asserts failed, but gdb drops us into some print function that's being called when the assert fails. But that print function uses our buggy malloc and blows up!

One thing we could do from here would be to inspect `ap` to see what `assert` was trying to print:

```
(gdb) p *ap
$4 = {gp_offset = 16, fp_offset = 48, overflow_arg_area = 0x7fffff7ff7f0, reg_save_area = 0x7fffff7ff730}
```

That would work fine; we could poke around until we figure out what's supposed to get printed and figure out the fail that way. Some other solutions would be to write our own custom assert or to use the right hooks to prevent `assert` from using our malloc.

But in this case, we know there are only a few asserts in our code. The one in malloc checking that we don't try to use this in a multithreaded program and the two in free checking that we're not freeing something we shouldn't. Let's look at free first, by setting a breakpoint.

```
$ gdb /bin/ls
(gdb) set environment LD_PRELOAD=./malloc.so
(gdb) break free
Breakpoint 1 at 0x400530
(gdb) run /bin/ls

Breakpoint 1, free (ptr=0x61c270) at malloc.c:112
```

```
112       if (!ptr) {
```

`block_ptr` isn't set yet, but if we use `s` a few times to step forward to after it's set, we can see what the value is:

```
(gdb) s
(gdb) s
(gdb) s
free (ptr=0x61c270) at malloc.c:118
118       assert(block_ptr->free == 0);
(gdb) p/x *block_ptr
$11 = {size = 0, next = 0x78, free = 0, magic = 0, data = ""}
```

I'm using `p/x` instead of `p` so we can see it in hex. The `magic` field is 0, which should be impossible for a valid struct that we're trying to free. Maybe `get_block_ptr` is returning a bad offset? We have `ptr` available to us, so we can just inspect different offsets. Since it's a `void *`, we'll have to cast it so that gdb knows how to interpret the results.

```
(gdb) p sizeof(struct block_meta)
$12 = 32
(gdb) p/x *(struct block_meta*)(ptr-32)
$13 = {size = 0x0, next = 0x78, free = 0x0, magic = 0x0, data = {0x0}}
(gdb) p/x *(struct block_meta*)(ptr-28)
$14 = {size = 0x7800000000, next = 0x0, free = 0x0, magic = 0x0, data = {0x78}}
(gdb) p/x *(struct block_meta*)(ptr-24)
$15 = {size = 0x78, next = 0x0, free = 0x0, magic = 0x12345678, data = {0x6e}}
```

If we back off a bit from the address we're using, we can see that the correct offset is 24 and not 32. What's happening here is that structs get padded, so that `sizeof(struct block_meta)` is 32, even though the last valid member is at `24`. If we want to cut out that extra space, we need to fix `get_block_ptr`.

That's it for debugging!

**Exercises**

Personally, this sort of thing never sticks with me unless I work through some exercsies, so I'll leave a couple exercises here for anyone who's interested.

1. malloc is supposed to return a pointer "which is suitably aligned for any built-in type". Does our malloc do that? If so, why? If not, fix the alignment. Note that "any built-in type" is basically up to 8 bytes for C because SSE/AVX types aren't built-in types.
2. Our malloc is really wasteful if we try to re-use an existing block and we don't need all of the space. Implement a function that will split up blocks so that they use the minimum amount of space necessary
3. After doing `2`, if we call malloc and free lots of times with random sizes, we'll end up with a bunch of small blocks that can only be re-used when we ask for small amounts of space. Implement a mechanism to merge adjacent free blocks together so that any consecutive free blocks will get merged into a single block.
4. Find bugs in the existing code! I haven't tested this much, so I'm sure there are bugs, even if this basically kinda sorta works.

**Resources**

I read this tutorial by Marwan Burelle before sitting down and trying to write my own implementation, so it's pretty similar. The main implementaiton differences are that my version is simpler, but more vulnerable to memory fragmentation. In terms of exposition, my style is a lot more casual. If you want something a bit more formal, Dr. Burelle has you covered.

For more on how Linux deals with memory management, see this post by Gustavo Duarte.

For more on how real-world malloc implementations work, dlmalloc and tcmalloc are both great reading. I haven't read the code for jemalloc, and I've heard that it's a bit more more difficult to understand, but it's also the most widely used high-performance malloc implementation around.

For help debugging, Address Sanitizer is amazing. If you want to write a thread-safe version, Thread Sanitizer is also a great tool.

There's a spanish translation of this post here thanks to Matias Garcia Isaia.

**Acknowledgements**

Thanks to Gustavo Duarte for letting me use one of his images to illustrate sbrk, to Ian Whitlock and Danielle Sucher for finding some typos, to Nathan Kurz for suggestions on additional resources, and to "tedu" for catching a bug. Please let me know if you find other bugs in this post (whether they're in the writing or the code).

*← The performance cost of integer overflow checking*

*The effect of markets on discrimination is more nuanced than you think →*