HW 3: Malloc

CS 162

Due: Wednesday, November 4, 2015

1 Setup

Note: This homework is longer than previous assignments so please start early!
Your task in this homework is to implement your own memory allocator in `mm_alloc.c` from scratch. This will expose you to POSIX interfaces, force you to reason about memory, and pose interesting algorithmic challenges. Real systems often must deal with high memory pressure, so learning how to improve your memory allocator's performance is a very useful thing to know. It's also why custom allocators are ubiquitous (e.g in CPython, C++ Boost, and Linux).
This assignment will be due **11:59 pm PST Wednesday, November 4, 2015**

```
cd ~/code/personal
git pull staff master
cd hw3
```

You will find a simple skeleton in `mm_alloc.c`. `mm_alloc` defines three functions `mm_malloc`, `mm_free`, `mm_realloc`. You will need to implement these methods. Do not change the names!

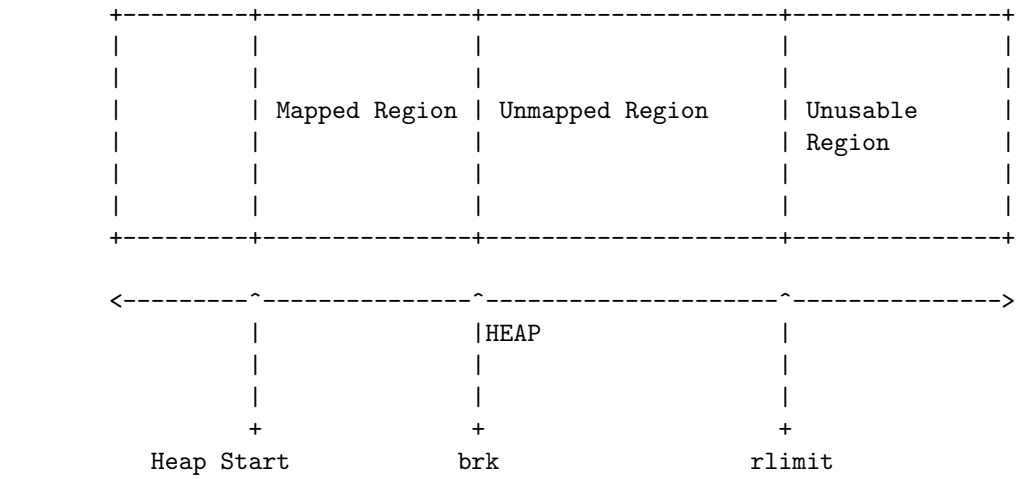
Your version of `malloc()` must use `sbrk()` to request memory from the kernel. The main job of `malloc()` is to allocate memory from the heap. Your `free()` routine is responsible for deallocating memory and releasing it back to the kernel as memory pressure decreases. Please note that you must *allocate your own memory* for any heap data structures you choose to use.

The `man` pages for `malloc` and `sbrk` are an excellent resource.

2 Getting Memory from the OS

2.1 Process Memory

Each process has its own virtual address space dynamically translated into physical memory address space by the MMU (and the kernel.) This space is divided in several parts: space for the code segment, a stack where local and volatile data is stored, some space for constant and global variables, and an unorganized space for the program's data called the heap. The heap is a continuous (in terms of virtual addresses) space of memory with three bounds: a starting point, a maximum limit (managed through `sys/resource.h's` functions `getrlimit(2)` and `setrlimit(2)`) and an end point called the break. The break marks the end of the mapped memory space, i.e. the part of the virtual address space that has correspondence into real memory. The below figure marks the organization.



2.2 brk(2) and sbrk(2)

We can find the description of these syscalls in their manual pages:

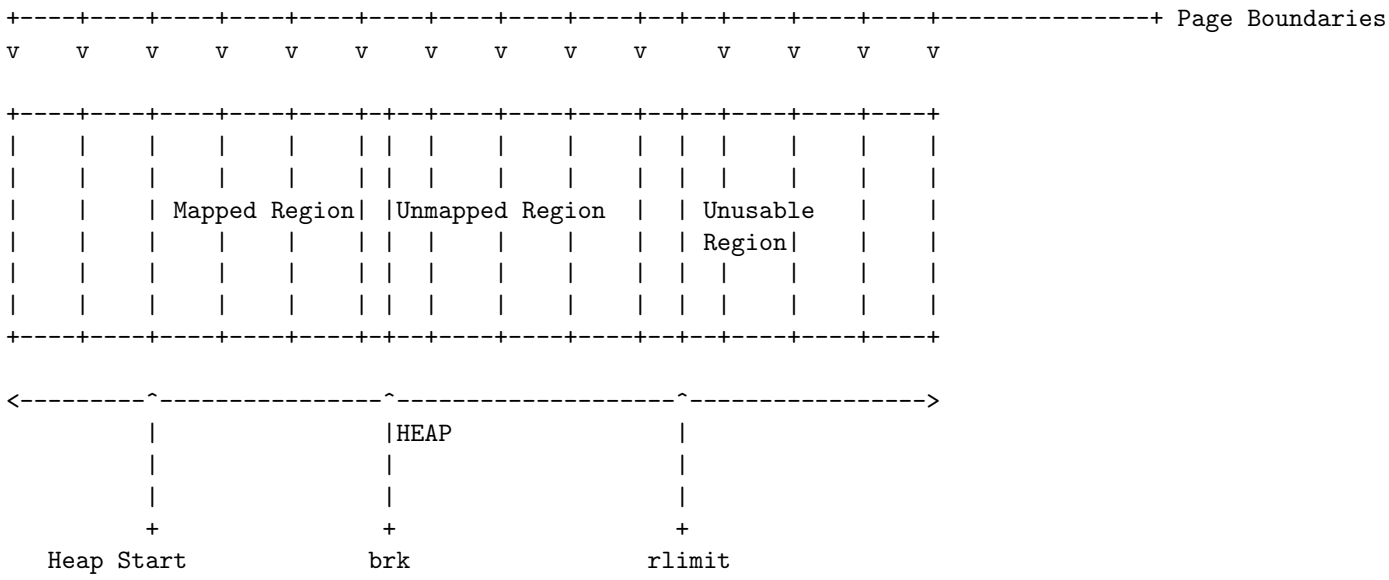
```
int brk(void *addr);
void *sbrk(intptr_t increment);
```

`brk(2)` places the break at the given address `addr` and return 0 if successful, -1 otherwise. The global `errno` symbol indicates the nature of the error. `sbrk(2)` moves the break by the given increment (in bytes.) On success it returns the previous address (on Linux systems). On failure, it returns (void *)-1 and sets `errno`. On Linux `sbrk` accepts negative values (in order to free some mapped memory.)

We will use `sbrk` as our main tool to implement `malloc`. This allows us to acquire more space in the heap if needed.

2.3 Unmapped Region and No Man's Land

We saw earlier that the break marks the end of the mapped virtual address space: accessing addresses above the break should trigger an error ("bus error" or "segmentation fault"). The remaining space between the break and the maximum limit of the heap is not mapped to physical memory by the virtual memory manager of the system (the MMU and the dedicated part of the kernel.)



The above figure shows that the break may not occur directly on a page boundary. What is the status of the memory between the break and the next page boundary? In fact, this space is available! You can access (for reading or writing) bytes in this space. The issue is that you don't have any clue on the position of the next boundary: you can find it – but it is system dependant and not advised. This so-called "no-man's land" is often a root of bugs: some wrong manipulations of pointers to create addresses outside of the heap may succeed with small tests and fail only when manipulating larger amount of data.

3 Managing Memory

3.1 Organizing the Heap

If we consider this organization problem outside of the programming context, we can infer what kind of information we need to solve our issues. Take the following analogy: say you are a farmer, you own a field and want to rent chunks of it to some clients. Clients ask for different length (you divide your field using only one dimension) which they expect to be continuous. After a bountiful harvest season, they return the chunk they rented, and you want to rent that same chunk to a new client.

To tackle this problem, we buy a programmable car which travels along one side of the field and records distance from the beginning. In order to find which portions of the field are free, we need to know both the beginning address of each free chunk as well as the address of the next free chunk.

One solution is to put a sign at the beginning of each chunk that indicates the address of the next one (and the size of the current one to avoid unnecessary computing.) We also add a mark on our sign to indicate whether or not a chunk is free. Now, when a client wants a portion of a certain size we get

in our car and travel from sign to sign until we find a large enough, free chunk. Then, we modify the sign to indicate that the chunk is in use, perhaps adding a new sign at the end of our chunk if there is enough room in the field before the next sign. If we reach the last chunk simply add a new sign and extend our field (using sbrk).

Now, transposing this idea to our programming context: we need extra information at beginning of each chunk to indicate the size, the address of the next chunk, and whether or not it is free.

3.2 Finding a fit

Finding a free and sufficiently wide chunk is quite simple: we begin at the base address of the heap and test the current chunk. If it fits our need and is free, we mark it as not free and just return its address. Otherwise, we continue to the next chunk until we find a fitting one or there is no next chunk. If that's the case, we call `sbrk` to extend the end of the heap.

You may have noticed that we use the first available block regardless of its size, provided that its wide enough. If we do that we will lose a lot of space (think of the scenario where you ask for 4 bytes and find a block of 256 bytes. What happens then?) To solve this, we will implement a method to split blocks: when a chunk is wide enough to hold the asked size plus a new chunk, we insert a new chunk in the list.

3.3 Data Structures

Consider using a data structure to manage memory (these go from simple to more complicated and efficient):

- A free list, a linked list of free chunks of memory (i.e. what is described above)
- A list of memory sizes, each of which contain a linked list of free chunks of memory of that size. (This is essentially a list of memory buckets)
- a free interval tree. These let you represent every memory allocation as an extent `(start, length)`. The leaves of the tree correspond to regions of unused memory. If N bytes are requested, it should be possible to scan the interval tree for $(> N)$-sized pieces of memory in $O(\log n)$ time, so long as it's properly balanced.

All of the above and more are acceptable data structures to manage your memory.

4 Deallocation

Calls to `free(3)` give memory back to the process to hand out to other allocations. It is important to understand that this is not the same as handing memory back to the operating system!

You must figure out which chunk corresponds to the given pointer and mark the appropriate chunk as "free", so future allocations can use that chunk. This means that if we `malloc` a block, `free` it, and then `malloc` the same size again, we should use the same block the second time.

If you look at the `man` page for `malloc`, you'll see that a call to `malloc(0)` should return either `NULL` or a unique pointer suitable to be passed in to `free`. For the purposes of this homework, just return `NULL` for `malloc(0)`. Make sure you also handle the case where the pointer passed in to `free` is `NULL`.

4.1 Fragmentation

Deallocating memory can cause fragmentation, which reduces the amount of available contiguous memory and increases lookup times. Your allocator **must** serve a request for N bytes of memory without a call to `sbrk` if at least N bytes of continuous memory exist in your previously `sbrk`-ed slab! In other words,

your allocator must be intelligent enough to combine two contiguous free chunks to achieve its allocation needs, and avoid calls to `sbrk` whenever necessary. A simple solution is the following: when we free a chunk and its neighbors also happen to be free, we can fuse them into one bigger chunk.

5 Realloc

Realloc is pretty straight forward. It returns a pointer to the newly allocated memory or `NULL` if the request fails. If the new pointer is different from the old pointer, it makes sure to `free` the old pointer. If size was equal to 0, return `NULL`. If `realloc()` fails, the original block is left untouched; do not `free` or move the block.

You can get away with a simple implementation that calls your version of `free` and `malloc`, and `memcpy`. Make sure to think about the case where the new size is smaller than the old size! When we want to `realloc` a smaller size, different implementations of `realloc` have different behaviors. For this homework, and when you encounter `realloc` in real life, you cannot assume that we return the same pointer. In this case, assume that a different pointer is returned and the original space is freed.

For example, if I have code as follows:

```
p = malloc(100);
q = malloc(200);
free(p);
r = realloc(q, 100);
s = malloc(200);
```

When we call `realloc`, we reuse the `p` block. In this case, we would not have fragmentation and would not need to ask for more memory when we call `malloc` again for `s`.

6 Memory Layout

We will give you complete creative freedom in designing your allocator (as long as it works!). But to make autograding simpler, your allocated memory must be **0 filled** - you can use `memset` for this. We know this isn't how `malloc` behaves, but it will help us a lot, so thanks in advance for making your `malloc` behave like `calloc`!

7 ε Bonus

ε The bonus isn't worth extra points.
There many things you can do to improve your allocator.

- One of the most interesting/challenging of which is to make it threadsafe! This doesn't mean putting a lock around all calls to `malloc`, but actually protecting the data structures such that multiple threads can make allocations at the same time. A good way to do this (not the only way) is to lock certain sizes of allocations, so two threads asking for `4kb` would block, but two thread askings for `4kb` and `32kb` would not block.
- You can improve your allocation algorithm. First fit is one of the simplest to implement. Another more advanced strategy is the buddy allocator
- Implement `realloc` properly to extend the current allocation chunk if possible.

8 Suggestions

Create your struct in `mm_alloc.h` that will store information necessary to implement the chunks. As mentioned earlier in Section 3.1, there are some necessary bookkeeping for each chunk. The struct should contain all those variables, and might want to end with a pointer to the data.

For example, if your bookkeeping needs b bytes and we wanted a block of size s, consider the following sample layout if you are using a free list:

```
-------------------------------------------------------------
| *prev | *next | free? | size | <size> zero-filled bytes |
-------------------------------------------------------------
0                               b                           b+s
```

To end with a pointer to the data, add a zero-length array `char data[0]` to the end of your struct. This would allow you to use `sizeof(struct your_struct)` to isolate the size of your bookkeeping, while also giving you access the data directly through the pointer `your_struct->data`.

Feel free to add extra fields to your struct and helper methods to simplify your code. And as always, there are many different ways to complete the assignment, so do not feel obligated to follow the sample layout.

9 Autograder & Submission

The autograder will replace your `Makefile` before it tests your code.

To submit and push to autograder:

```
git add -u .
git commit
git push personal master
```

Within 30 minutes you should receive an email from the autograder. (If you haven't received an email within half an hour, please notify the instructors via a private post on Piazza.)