HW 4: Malloc

CS 162

Due: October 26, 2020

Contents

1 Introduction

Your task in this assignment is to implement your own memory allocator from scratch. This will expose you to POSIX interfaces, give you the chance to reason about memory, and pose interesting algorithmic challenges.

Hint: The `man` pages for `malloc` and `sbrk` are excellent resources for this assignment.

Note: You **MUST** use `sbrk` to allocate the heap region. You are **NOT** allowed to call the standard `malloc/free/realloc` functions (doing so would defeat the purpose of the homework).

2 Setup

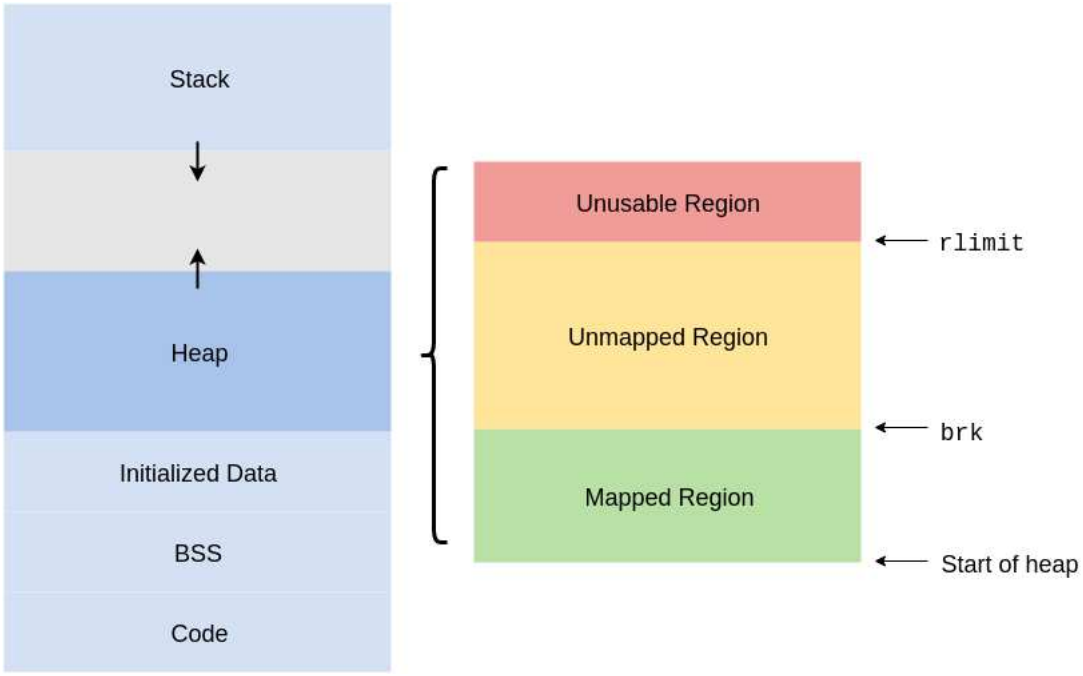
```
cd ~/code/personal
git pull staff master
cd hw4
```

You will find a simple skeleton in `mm_alloc.c`. `mm_alloc` defines an interface with three functions `mm_malloc`, `mm_free`, `mm_realloc`. You will need to implement these functions. Do not change the names! You will also find the file `mm_test.c`, which contains code to load your implementation and a basic sanity check. Feel free to change this file as it won't be graded.

3 Background: Getting Memory from the OS

3.1 Process Memory

Each process has its own virtual address space. Parts of this address space are mapped to physical memory through address translation. In order to build a memory allocator, we need to understand how the heap in particular is structured.



The heap is a continuous (in terms of virtual addresses) space of memory that grows upward in memory with three bounds:

- The start (bottom) of the heap.
- The top of the heap, known as the break. The break can be changed using `brk` and `sbrk`. The break marks the end of the mapped memory space. Above the break lies virtual addresses which have not been mapped to physical addresses by the OS.
- The hard limit of the heap, which the break cannot surpass (managed through `sys/resource.h`'s functions `getrlimit(2)` and `setrlimit(2)`)

In this assignment, you'll be allocating blocks of memory in the mapped region and moving the break appropriately whenever you need to expand the mapped region.

3.2 sbrk

Initially the mapped region of the heap will have a size of 0. To expand the mapped region, we have to manipulate the position of the break. The recommended syscall for doing this is `sbrk`.

```
void *sbrk(int increment);
```

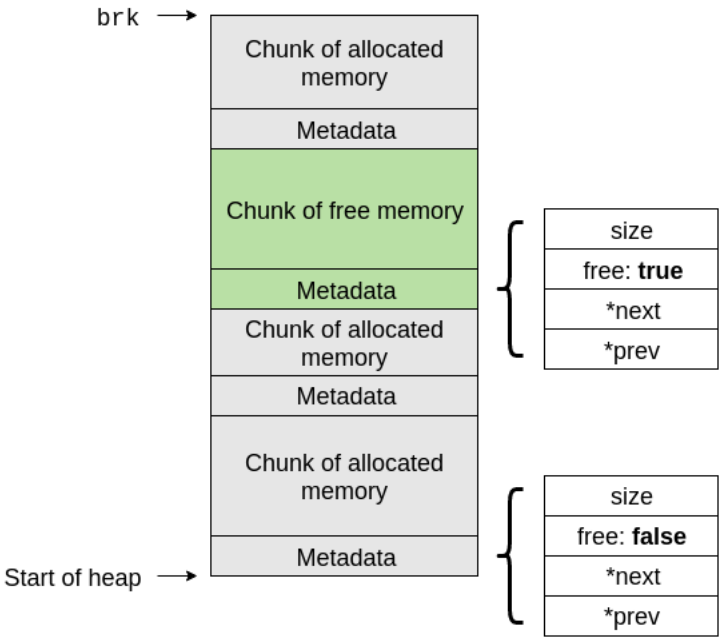
`sbrk` increments the position of the break by `increment` bytes and returns the address of the previous break (i.e. the beginning of newly mapped memory). To get the current position of the break, pass in an `increment` value of 0. For more information, read the man page.

3.3 Heap Data Structure

A simple memory allocator for the heap can be implemented using a linked list data structure. The elements of the linked list will be the allocated blocks of memory on the heap. To structure our data, each allocated block of memory will be preceded by a header containing metadata.

For each block, we include the following metadata:

- `prev, next`: pointers to metadata describing the adjacent blocks
- `free`: a boolean describing whether or not this block is free
- `size`: the allocated size of the block of memory



You might also consider using a zero-length array to serve as a pointer to the memory block.

4 Your Assignment

There are many ways to structure a memory allocator. You will be implementing a memory allocator using a linked list of memory blocks, as described in the previous section. In this section, we'll describe how allocation, deallocation, and reallocation should work in this scheme. To make your implementation succeed, you will need to modify `mm_alloc.c`.

4.1 Allocation

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

The user will pass in the requested allocation `size`. Make sure the returned pointer is pointing to the beginning of the allocated space, not your metadata header.

One simple algorithm for finding available memory is called **first fit**. When your memory allocator is called to allocate some memory, it iterates through its blocks until it finds a sufficiently large free block of memory.

Implementation Details:

- If no sufficiently large free block is found, use `sbrk` to create more space on the heap.
- If the first block of memory you find is so large that it can accommodate both the newly allocated block and another block in addition, then the large block is split in two; one block to hold the newly allocated block, the other to be a residual free block.
- If the first block of memory you find is only a bit larger than what you need, but not large enough for a new block (i.e. it's not big enough to hold the metadata of a new block), be aware that you will have some unused space at the end of the newly allocated block.
- Return `NULL` if you cannot allocate the new requested size.
- Return `NULL` if the requested size is 0.
- For grading purposes, please **zero-fill your allocated memory** before returning a pointer to it.

4.2 Deallocation

```
void mm_free(void *ptr);
```

When a user is done using their memory, they'll call upon your memory allocator to free their memory, passing in the pointer `ptr` that they received from `mm_alloc`.

Note: Deallocating doesn't mean you have to release the memory back to the OS; you just have be able to allocate that block for future use now.

Implementation Details:

- As a side-effect of splitting blocks in your allocation procedure, you might run into issues of **fragmentation**: when your blocks become too small for large allocation requests, even though you have a sufficiently large section of free memory. To solve this, you must **coalesce** consecutive free blocks upon freeing a block that is adjacent to other free block(s).
- Your deallocation function should do nothing if passed a `NULL` pointer.

4.3 Reallocation

```
void* mm_realloc(void* ptr, size_t size);
```

Reallocation should resize the allocated block at `ptr` to `size`. A suggested implementation is to first free the block referenced by `ptr`, then `mm_alloc` a block of the specified size, zero-fill the block, and finally `memcpy` the old data to the new block.

Make sure you handle the following edge cases:

- Return `NULL` if you cannot allocate the new requested size. In this case, do not modify the original block.
- `mm_realloc(ptr, 0)` is equivalent to calling `mm_free(ptr)` and returning `NULL`.
- `mm_realloc(NULL, n)` is equivalent to calling `mm_malloc(n)`.
- `mm_realloc(NULL, 0)` is equivalent to calling `mm_malloc(0)`, which should just return `NULL`.
- Make sure you handle the case where `size` is less than the original size.

5 Submission Details

To submit and push to autograder, first commit your changes, then do:

```
git push personal master
```

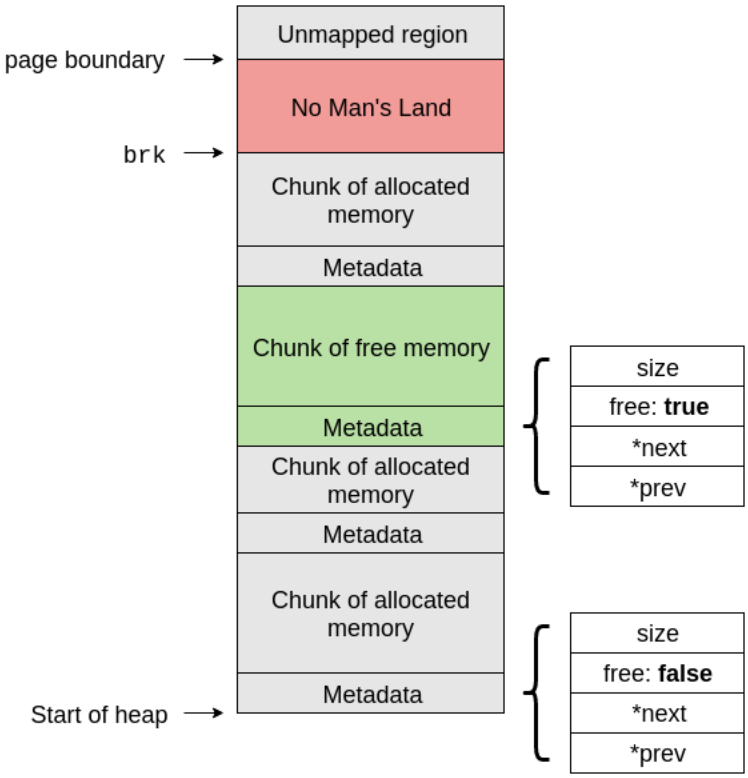
Within 30 minutes you should receive an email from the autograder. (If you haven't received an email within 30 minutes, please notify the instructors via a private post on Piazza.)

A Additional Information

A.1 Unmapped Region and No Man's Land

We saw earlier that the break marks the end of the mapped virtual address space. By this assumption, accessing addresses above the break should trigger an error ("bus error" or "segmentation fault").

The virtual address space is mapped in quanta of pages (usually some multiple of 4096 bytes). When `sbrk` is called, the OS will have to map more memory to the heap. To do that, it maps an entire page from physical memory to the mapped region of the heap. Now, it is possible that the break doesn't end up exactly on a page boundary. In this situation, what is the status of the memory between the break and the page boundary? It turns out that this memory is accessible, even though it is above the break and thus should be unmapped in theory. Bugs related to this issue are particularly insidious, because no error will occur if you read from or write to this "no man's land."



A.2 Improvements

This is **NOT** worth extra points.

There are many things you can do to improve your allocator. (**Warning:** The autograder expects you to use the first-fit algorithm. If you want to do extra features, do them only AFTER you've gotten full credit on the autograder.)

- 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 block if possible.

A.3 Other Implementations

Using other data structures, it is also possible to create a memory allocator.

The following are two such data structures:

- A list of free blocks for each allocation size. The advantage here is that `malloc` can be a constant time operation if a sufficiently large free block exists, as opposed to the linear time operation of iterating over a linked list in search of a large enough free block, without even knowing if there exists such a block.
- A free interval tree. This lets 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.