Homework 5: Malloc

Due: April 25, 2022

Contents

1 Overview

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

1.1 Getting Started

```
> cd ~/code/personal
> git pull staff master
> cd hw-memory
```

Inside, you will find a simple skeleton in `mm_alloc.c`. `mm_alloc.h` defines an interface with three functions: `mm_malloc`, `mm_free`, `mm_realloc`. You will need to implement these functions. **Do not change the headers of any of these!**

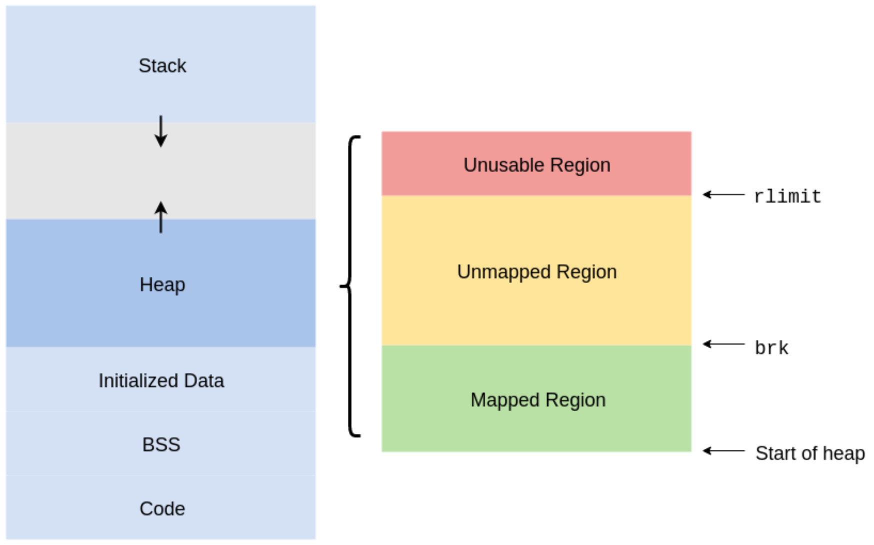
To compile the code, simply run `make`. The given `mm_test.c` performs some sanity checks but is not an exhaustive test. We recommend you write some custom tests when testing locally by changing this file.

1.2 Background

The man pages for `malloc` and `sbrk` are excellent resources for this assignment. **Note that you must use `sbrk` to allocate the heap region. You are NOT allowed to call the standard `malloc/free/realloc` functions.**

1.2.1 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.

1.2.2 `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 `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, check out `man sbrk`.

1.2.3 Heap Data Structure

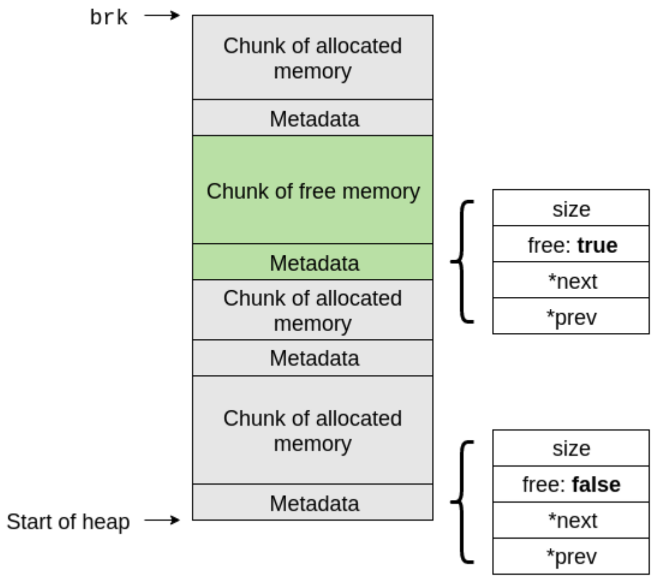
A simple memory allocator for the heap can be implemented using a linked list data structure. The elements of the linked list will be the allocated blocks of memory on the heap. To structure our data, each allocated block of memory will be preceded by a header containing metadata.

For each block, we include the following metadata:

`prev`, `next`
: Pointers to metadata describing the adjacent blocks

`free`
: Boolean describing whether or not this block is free

`size`
: Allocated size of the block of memory



You might also consider using a flexible array member[1] to serve as a pointer to the memory block.

[1] https://en.wikipedia.org/wiki/Flexible_array_member

2 Tasks

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`.

2.1 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.

Here are some implementation details to be aware of.

- 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.

2.2 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 that 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.

Here are some implementation details to be aware of.

- 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.

2.3 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.

3 Submission

To submit and push to the autograder, push your changes to your repo which should trigger the autograder unless you're using slip days in which case you need to manually run it. Within a few minutes you should receive an email from the autograder. If you don't receive an email from the autograder within half an hour, please notify staff via a private post on Piazza. Your code should not include extraneous or debugging print statements as this will interefere with the autograder. If there are written responses, they should be submitted to the Gradescope and will not be graded by the autograder.